Online Algorithms: A Study of Graph-Theoretic Concepts

Susanne Albers

Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany.
albers@mpi-sb.mpg.de, http://www.mpi-sb.mpg.de/~albers/

Abstract. In this paper we survey results on the design and analysis of online algorithms, focusing on problems where graphs and graph-theoretic concepts have proven particularly useful in the formulation or in the solution of the problem. For each of the problems addressed, we also present important open questions.

1 Introduction

The traditional design and analysis of algorithms assumes that an algorithm, which generates output, has complete knowledge of the entire input. However, this assumption is often unrealistic in practical applications. Many of the algorithmic problems that arise in practice are *online*. In these problems the input is only partially available because some relevant input arrives only in the future and is not accessible at present. An online algorithm must generate output without knowledge of the entire input. Online problems arise in areas such as resource allocation in operating systems, data structuring, robotics, distributed computing and scheduling. We give some illustrative examples.

Caching: In a two-level memory system, consisting of a small fast memory and a large slow memory, a caching algorithm has to keep actively referenced pages in fast memory not knowing which pages will be requested next.

Data structures: In a given data structure, we wish to access elements at low cost. The goal is to keep the data structure in a good state, not knowing which elements will have to be accessed in the future.

Robot exploration: A robot is placed in an unknown environment and has to construct a complete map of the environment using a path that is as short as possible. The environment is explored only as the robot travels around.

Distributed data management: In a network of processors we wish to dynamically re-allocate files so that accesses can be served with low communication cost. Future accesses are unknown.

We will address these problems in more detail in the following sections.

During the last ten years *competitive analysis* has proven to be a powerful tool for analyzing the performance of online algorithms. In a competitive analysis, introduced by Sleator and Tarjan [50], an online algorithm A is compared to an *optimal offline algorithm* OPT. An optimal offline algorithm knows the entire input in advance and can process it optimally. Given an input I, let $C_A(I)$ and

$C_{OPT}(I)$ denote the costs incurred by A and OPT in processing I. Algorithm A is called c-competitive if there exists a constant a such that

$$C_A(I) \leq c \cdot C_{OPT}(I) + a,$$

for all inputs I. Note that a competitive algorithm must perform well on *all* input sequences.

2 Caching

In this section, we first investigate uniform caching, also known as paging. For this problem, access graphs are very useful in the modeling of realistic request sequences. In the second part of this section we discuss caching problems that arise in large networks such a the world-wide web.

2.1 Paging

We first define the paging problem formally. In the two-level memory system, all memory pages have the same size. We assume that the fast memory can store k pages at any time. A paging algorithm is presented with a *request sequence* $\sigma = \sigma(1), \ldots, \sigma(m)$, where each request $\sigma(t)$, $1 \leq t \leq m$, specifies a page in the memory system. A request can be served immediately, if the requested page is in fast memory. If a requested page is not in fast memory, a *page fault* occurs. Then the missing page has to be copied from slow memory to fast memory so that the request can be served. At the same time the paging algorithm has to evict a page from fast memory in order to make room for the incoming page. Typically, a paging algorithm has to work online, i.e. the decision which page to evict on a fault must be made without knowledge of any future requests. The goal is to minimize the number of page faults.

We note that an online paging algorithm A is called c-competitive if, for all request sequences σ, the number of faults incurred by A is at most c times the number of faults incurred by an optimal offline algorithm OPT.

There are two very well-known online algorithms for the paging problem.

Least-Recently-Used (LRU): On a fault evict the page that has been requested least recently.

First-In First-Out (FIFO): On a fault evict the page that has been in fast memory longest.

An optimal offline algorithm was presented by Belady [16]. The algorithm is called MIN and has a polynomial running time.

MIN: On a fault evict the page whose next request is farthest in the future.

Sleator and Tarjan [50] analyzed the performance of LRU and FIFO and showed that on any request sequence the number of page faults incurred by the algorithms is bounded by k times the number of faults made by OPT. They also showed that this is optimal.

Theorem 1. [50] *LRU and FIFO are k-competitive.*

Theorem 2. [50] *No deterministic online algorithm for the paging problem can achieve a competitive ratio smaller than k.*

While Theorem 1 gives a strong worst-case analysis of LRU and FIFO, the competitive ratio of k is not very meaningful from a practical point of view. A fast memory can often hold several hundreds or thousands of pages. In fact, the competitive ratio of k is much higher than observed in practice. In an experimental study presented by Young [53], LRU achieves a competitive ratio between 1 and 2. Also, in practice, the performance of LRU is much better than that of FIFO. This does not show in the competitive analysis.

The high competitive ratio is due to the fact that a competitive analysis allows arbitrary request sequences whereas in practice only restricted classes of request sequences occur. Request sequences generated by real programs typically exhibit locality of reference: Whenever a page is requested, the next request is usually to a page that comes from a very small set of associated pages. Locality of reference can be modeled by *access graphs*, introduced by Borodin *et al.* [27]. In an access graph, the nodes represent the memory pages. Whenever a page p is requested, the next request can only be to a page that is adjacent to p in the access graph.

Formally, let $G = (V, E)$ be an unweighted graph. As mentioned above, V represents the set of memory pages. E is a set of edges. A request sequence $\sigma = \sigma(1), \ldots, \sigma(m)$, is *consistent with* G if $(\sigma(t), \sigma(t+1)) \in E$ for all $t = 1, \ldots, m-1$. We say that an online algorithm A is c-competitive on G if there exists a constant a such that $C_A(\sigma) \leq c \cdot C_{OPT}(\sigma) + a$ for all σ consistent with G. The competitive ratio of A on G, denoted by $R(A, G)$, is the infimum of all c such that A is c-competitive on G. Let

$$R(G) = \min_A R(A, G)$$

be the best competitive ratio achievable on G. Borodin *et al.* [27] showed that LRU achieves the best possible competitive ratio on access graphs that are trees. Trees represent the access graphs for many data structures. For any tree T, let $l(T)$ denote the number of leaves of T. Let

$$\mathcal{T}_n(G) = \{T \mid T \text{ is an } n\text{-node subtree of } G\}.$$

Theorem 3. [27] *For any graph G consisting of at least $k+1$ nodes,*

$$R(G) \geq \max_{T \in \mathcal{T}_{k+1}} \{l(T) - 1\}.$$

Theorem 4. [27] *Let G be a tree. Then $R(LRU, G) = \max_{T \in \mathcal{T}_{k+1}} \{l(T) - 1\}$.*

Borodin *et al.* [27] also analyzed $R(LRU, G)$ on arbitrary graphs. They showed that the ratio depends on the number of *articulation nodes* of G. An articulation node is a node whose removal disconnects the graph. Another important result,

due to Chrobak and Noga [26], is that LRU is never worse than FIFO on access graphs. Moreover, Borodin *et al.* [27] showed that there exist graphs for which the competitive ratio of FIFO is much higher than that of LRU.

Theorem 5. [26] *For any graph G, $R(LRU, G) \leq R(FIFO, G)$.*

Theorem 6. [27] *For any connected graph consisting of at least $k+1$ nodes, $R(FIFO, G) \geq (k+1)/2$.*

Borodin *et al.* [27] also presented an optimal online algorithm for any access graph.

FAR: The algorithm is a marking strategy working in phases. Whenever a page is requested, it is marked. If there is a fault at a request to a page p, then FAR evicts an unmarked page from fast memory that has the largest distance to a marked page in the access graph. A phase ends when all pages in fast memory are marked and a fault occurs. At this point all marks are erased and a new phase is started.

Irani *et al.* [39] showed that this algorithm achieves the best possible competitive ratio, up to a constant factor, for all access graphs.

Theorem 7. [39] *For any graph G, $R(FAR, G) = O(R(G))$.*

Fiat and Karlin [30] presented randomized online paging algorithms for access graphs that achieve an optimal competitive ratio.

A disadvantage of the algorithm FAR is that the access graph has to be known in advance. Fiat and Rosen [31] proposed a scheme that grows a dynamic weighted access graph over time. Whenever two pages p and q are requested successively for the first time, an edge (p, q) of weight 1 is inserted into the graph. Every time p and q are requested successively again, the weight w of the edge is decreased to $\min\{\alpha w, 1\}$ for some $\alpha < 1$. After each round of γk requests, all weights are increased by β, where $\beta, \gamma > 1$ are some fixed chosen constants. Fiat and Rosen [31] proposed the following variant of the algorithm FAR, called FARL: If there is a fault, the algorithm evicts the page that has the largest distance from the page requested just before the fault. Fiat and Rosen presented an experimental study in which FARL incurs fewer page faults than even LRU.

So far we have addressed undirected access graphs. An initial investigation of directed access graph was presented by Irani *et al.* [39], who considered structured program graphs.

Open Problem: Develop online paging algorithms for general directed access graphs.

2.2 Generalized Caching

Caching problems that arise in large networks such as the world-wide web differ from ordinary caching in two aspects. Pages may have different sizes and may

incur different costs when loaded into fast memory. The pages or *documents* to be cached may be text files, pictures or web pages; the cost of loading a missing page into fast memory may depend on the size of the page and on the distance to the nearest node in the network holding the page. In generalized caching we have again a two-level memory system consisting of a fast and a slow memory. (In the network setting, the fast memory is the memory of a given node. The slow memory is the memory of the remaining network.) We assume that the fast memory has a capacity of K. For any page p, let $\text{SIZE}(p)$ be the size and $\text{COST}(p)$ be the cost of p. The total size of the pages in fast memory may never exceed K. The goal is to serve a sequence of requests so that the total loading cost is as small as possible. Various cost models have been proposed in the literature.

1. **The Bit Model [38]:** For each page p, we have $\text{COST}(p) = \text{SIZE}(p)$. (The delay in bringing the page into fast memory depends only upon its size.)
2. **The Fault Model [38]:** For each page p, we have $\text{COST}(p) = 1$ while the sizes can be arbitrary.
3. **The Cost Model:** For each page p, we have $\text{SIZE}(p) = 1$ while the costs can be arbitrary.
4. **The General Model:** For each page p, both the cost and size can be arbitrary.

In the Bit Model, and hence in the General Model, computing an optimal offline service schedule for a given request sequence is NP-hard. The problem is polynomially solvable in the Cost Model [25]. In fact, caching in the Cost Model is also known as *weighted caching*, a special instance of the k-server problem. In the Fault Model the complexity is unknown.

Open Problem: Determine the complexity of caching in the Fault Model.

Young [54] gave a k-competitive online algorithm for the General Model.

Landlord: For each p in fast memory, the algorithm maintains a variable $\text{CREDIT}(p)$ that takes values between 0 and $\text{COST}(p)$. If a requested page p is already in fast memory, then $\text{CREDIT}(p)$ is reset to any value between its current value and $\text{COST}(p)$. If the requested page is not in fast memory, then the following two steps are executed until there is enough room to load p. (1) For each page q in fast memory, decrease $\text{CREDIT}(q)$ by $\Delta \cdot \text{SIZE}(q)$, where $\Delta = \min_{q \in F} \text{CREDIT}(q)/\text{SIZE}(q)$ and F is the set of pages in fast memory. (2) Evict any page q from fast memory with $\text{CREDIT}(q) = 0$. When there is enough room, load p and set $\text{CREDIT}(p)$ to $\text{COST}(p)$.

Theorem 8. [54] *Landlord is k-competitive in the General Model.*

For the Bit and the Fault Model, Irani presented $O(\log^2 k)$-competitive online algorithms and $O(\log k)$-approximation offline algorithms. Here k is the ratio of K to the size of the smallest page requested. For the offline problem, Albers *et al.* [2] gave constant factor approximation algorithms using only a small amount of additional space in fast memory, say $O(1)$ times the largest page size. Note that the largest page size is typically a very small fraction of the total size of

the fast memory, say 1%. The approach is to formulate the caching problems as integer linear programs and then solve a relaxation to obtain a fractional optimal solution. The *integrality gap* of the linear programs is unbounded, but nevertheless one can show the following.

Theorem 9. [2] *There is a polynomial-time algorithm that, given any request sequence, finds a solution of cost* $c_1 \cdot \mathrm{OPT}_{\mathrm{LP}}$, *where* $\mathrm{OPT}_{\mathrm{LP}}$ *is the cost of the fractional solution (with fast memory* K*). The solution uses* $K + c_2 \cdot S$ *memory, where* S *is the size of the largest page in the sequence. The values of* c_1 *and* c_2 *are as follows for the various models. Let* ϵ *and* δ *be real numbers with* $\epsilon > 0$ *and* $0 < \delta \leq 1$.

1. $c_1 = 1/\delta$ *and* $c_2 = \delta$ *for the Bit Model,*
2. $c_1 = (1+\epsilon)/\delta$ *and* $c_2 = \delta(1 + 1/(\sqrt{1+\epsilon} - 1))$ *for the Fault Model,*
3. $c_1 = 4 + \epsilon$ *and* $c_2 = 6/\epsilon$ *for the General Model.*

The c_1, c_2 values in the above theorem express trade-offs between the approximation ratio and the additional memory needed. For example, in the Bit Model, we can get a solution with cost $\mathrm{OPT}_{\mathrm{LP}}$ using at most S additional memory. In the Fault model, we can get a solution with cost $4\mathrm{OPT}_{\mathrm{LP}}$ using at most $2S$ additional memory. The approximation ratio can be made arbitrarily close to 1 by using $c_2 S$ additional memory for a large enough c_2. In the General Model we obtain a solution of $5\mathrm{OPT}_{\mathrm{LP}}$ using $6S$ additional memory, but we can achieve approximation ratios arbitrarily close to 4.

Open Problem: Are there constant factor approximation algorithms that do not require extra space in fast memory?

3 Data structures

Many online problems arise in the area of data structures. We consider the list update problem which is among the most extensively studied online problems.

The list update problem is to maintain a dictionary as an unsorted linear list. Consider a set of items that is represented as a linear linked list. We receive a request sequence σ, where each request is one of the following operations. (1) It can be an *access* to an item in the list, (2) it can be an *insertion* of a new item into the list, or (3) it can be a *deletion* of an item. To access an item, a list update algorithm starts at the front of the list and searches linearly through the items until the desired item is found. To insert a new item, the algorithm first scans the entire list to verify that the item is not already present and then inserts the item at the end of the list. To delete an item, the algorithm scans the list to search for the item and then deletes it.

In serving requests a list update algorithm incurs cost. If a request is an access or a delete operation, then the incurred cost is i, where i is the position of the requested item in the list. If the request is an insertion, then the cost is $n+1$, where n is the number of items in the list before the insertion. While processing a request sequence, a list update algorithm may rearrange the list. Immediately

after an access or insertion, the requested item may be moved at no extra cost to any position closer to the front of the list. These exchanges are called *free exchanges*. Using free exchanges, the algorithm can lower the cost on subsequent requests. At any time two adjacent items in the list may be exchanged at a cost of 1. These exchanges are called *paid exchanges*.

With respect to the list update problem, we require that a c-competitive online algorithm has a performance ratio of c *for all size lists*. More precisely, a deterministic online algorithm for list update is called c-competitive if there is a constant a such that for all size lists and all request sequences σ, $C_A(\sigma) \leq c \cdot C_{OPT}(\sigma) + a$.

Linear lists are one possibility to represent a dictionary. Certainly, there are other data structures such as balanced search trees or hash tables that, depending on the given application, can maintain a dictionary in a more efficient way. In general, linear lists are useful when the dictionary is small and consists of only a few dozen items [19]. Furthermore, list update algorithms have been used as subroutines in algorithms for computing point maxima and convex hulls [18,32]. Recently, list update techniques have been very successfully applied in the development of data compression algorithms [6,20,24].

There are three well-known deterministic online algorithms for the list update problem.

Move-To-Front: Move the requested item to the front of the list.

Transpose: Exchange the requested item with the immediately preceding item in the list.

Frequency-Count: Maintain a frequency count for each item in the list. Whenever an item is requested, increase its count by 1. Maintain the list so that the items always occur in nonincreasing order of frequency count.

The formulations of list update algorithms generally assume that a request sequence consists of accesses only. It is obvious how to extend the algorithms so that they can also handle insertions and deletions. On an insertion, the algorithm first appends the new item at the end of the list and then executes the same steps as if the item was requested for the first time. On a deletion, the algorithm first searches for the item and then just removes it.

In the following, we discuss the algorithms Move-To-Front, Transpose and Frequency-Count. We note that Move-To-Front and Transpose are *memoryless* strategies, i.e., they do not need any extra memory to decide where a requested item should be moved. Thus, from a practical point of view, they are more attractive than Frequency-Count. Sleator and Tarjan [50] analyzed the competitive ratios of the three algorithms.

Theorem 10. [50] *The Move-To-Front algorithm is 2-competitive.*

Proposition 11. *The algorithms Transpose and Frequency-Count are not c-competitive, for any constant c.*

Albers [1] presented another deterministic online algorithm for the list update problem. The algorithm belongs to the Timestamp(p) family of algorithms that

were introduced in the context of randomized online algorithms and are defined for any real number $p \in [0,1]$, see [1]. For $p = 0$, the algorithm is deterministic and can be formulated as follows.

Timestamp(0): Move the requested item, say x, in front of the first item in the list that precedes x and that has been requested at most once since the last request to x. If there is no such item or if x has not been requested so far, then leave the position of x unchanged.

Theorem 12. [1] *The Timestamp(0) algorithm is 2-competitive.*

Note that Timestamp(0) is not memoryless. We need information on past requests in order to determine where a requested item should be moved. Timestamp(0) is interesting because it has a better overall performance than Move-To-Front. The algorithm achieves a competitive ratio of 2, as does Move-To-Front. However, Timestamp(0) is considerably better than Move-To-Front on request sequences that are generated by probability distributions [6]. For any probability distribution, the asymptotic expected cost incurred by TS(0) is at most 1.5 times the asymptotic expected cost incurred by an optimal offline algorithm. The corresponding bound for Move-To-Front is not better than $\pi/2$.

Karp and Raghavan [42] developed a lower bound on the competitiveness that can be achieved by deterministic online algorithms. This lower bound implies that Move-To-Front and Timestamp(0) have an optimal competitive ratio.

Theorem 13. [42] *Let A be a deterministic online algorithm for the list update algorithm. If A is c-competitive, then $c \geq 2$.*

An important question is whether the competitive ratio of 2 can be improved using randomization. We analyze randomized online algorithms problem against oblivious adversaries [17]. An oblivious adversary has to construct the entire request sequence in advance and is not allowed to see the random choices made by an online algorithm.

Many randomized online algorithms for list update have been proposed [1,7,35,36,49]. We present the two most important algorithms. Reingold *et al.* [49] gave a very simple algorithm, called Bit.

Bit: Each item in the list maintains a bit that is complemented whenever the item is accessed. If an access causes a bit to change to 1, then the requested item is moved to the front of the list. Otherwise the list remains unchanged. The bits of the items are initialized independently and uniformly at random.

Theorem 14. [49] *The Bit algorithm is 1.75-competitive against any oblivious adversary.*

Interestingly, it is possible to combine the algorithms Bit and Timestamp(0), see Albers *et al.* [7]. This combined algorithm achieves the best competitive ratio that is currently known for the list update problem.

Combination: With probability 4/5 the algorithm serves a request sequence using Bit, and with probability 1/5 it serves a request sequence using Timestamp(0).

Theorem 15. [7] *The algorithm Combination is 1.6-competitive against any oblivious adversary.*

Teia [51] presented a lower bound for randomized list update algorithms.

Theorem 16. [51] *Let A be a randomized online algorithm for the list update problem. If A is c-competitive against any oblivious adversary, then $c \geq 1.5$.*

A slightly better lower bound of 1.50084 was presented recently by Ambühl *et al.* [8]. However, the lower bound only holds in the partial cost model where the cost of serving a request to the i-th item in the list incurs a cost of $i-1$ rather then i.

Open Problem: Give tight bounds on the competitive ratio achieved by randomized online algorithms against any oblivious adversary.

4 Robot exploration

In robot exploration problems, a robot has to construct a complete map of an *unknown environment* using a path that is as short as possible. Many geometric and graph-theoretic problems have been studied in the past [3,13,22,28,29,33,34,46]. A general problem setting was introduced by Deng *et al.* [28]. The robot is placed in a room with obstacles. The exterior wall of the room as well as the obstacles are modeled by simple polygons. Figure 1 shows an example in which the room is a rectangle and all obstacles are rectilinear. The robot has 360° vision. Its task is to move through the scene so that it sees all parts of the room. More precisely, every point in the room must be visible from some point on the path traversed.

Given a scene S, let $L_A(S)$ be the length of the path traversed by algorithm A to explore S. Since A does not know S in advance it is also referred to as an *online algorithm*. Let $L_{OPT}(S)$ be the length of the path of an optimum algorithm that *knows the scene in advance*. We call an online exploration algorithm A *c-competitive* if for all scenes S, $L_A(S) \leq c \cdot L_{OPT}(S)$.

Exploration algorithms achieving a constant competitive ratio were given for rooms without obstacles [28,33,34,46]. Note that the exploration problem is non-trivial even in rooms without obstacles because the room might be a general polygon. Deng *et al.* [28] gave an $O(n)$-competitive algorithm for exploring rectilinear rooms with n rectilinear obstacles. Albers and Kursawe [5] showed that no exploration algorithm in rooms with n obstacles can be better than $\Omega(\sqrt{n})$-competitive. This lower bound holds even if the obstacles are rectangles.

4.1 Exploration of grid graphs

In the scenario described above it is assumed that the robot can see an infinite range as long as no obstacle or exterior wall blocks the view. However, in practice, a robot's sensors can often scan only a distance of a few meters. This situation can be modeled by adding a grid to the scene, as shown in Figure 2, and requiring that the robot moves on the nodes and edges of the grid. A node in the grid

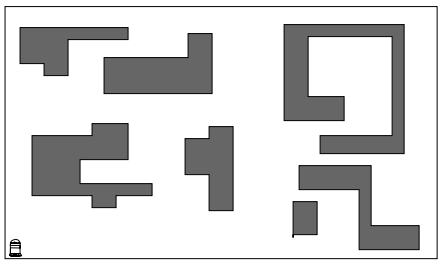

Fig. 1. A sample scene

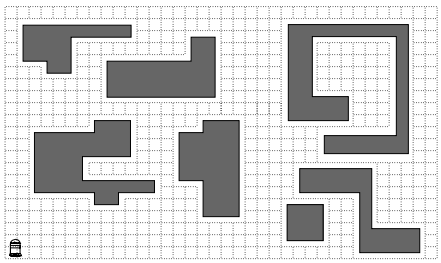

Fig. 2. A sample scene with a grid

models the vicinity that the robot can see at a given point. Now the robot has to explore all nodes and edges of the grid using as few edge traversals as possible. A node is explored when it is *visited* for the first time and an edge is explored when it is *traversed* for the first time. At any node the robot knows its global position and the directions of the incident edges. Note that using a depth-first strategy, the graph can be explored using $O(m)$ edge traversals, which is optimal. Here m denotes the total number of edges of the graph.

Betke *et al.* [22] introduced an interesting, more complicated variant of this problem where an additional *piecemeal constraint* has to be satisfied, i.e, the robot has to return to a start node s every so often. These returns might be necessary because the robot has to refuel or drop samples collected on a trip. Betke *et al.* developed two algorithms for piecemeal exploration of grids with rectangular obstacles. The algorithms, called *Wavefront* and *Ray*, need $O(m)$ edge traversals. The *Wavefront* algorithm implements a breadth-first strategy while the *Ray* algorithm implements a simple and elegant depth-first strategy.

Theorem 17. [22] *A grid with rectangular obstacles can be explored in a piecemeal fashion using $O(m)$ edge traversals.*

Albers and Kursawe [5] present an algorithm that explores a grid with arbitrary (rectilinear) obstacles using $O(m)$ edge traversals, which is optimal. The algorithm is a generalization of the *Ray* algorithm by Betke, Rivest and Singh. In the original *Ray* algorithm it is required that the robot always knows a path back to the start node whose length is most the radius of the graph. When exploring grids with arbitrary obstacles, this constraint cannot be satisfied. Albers and Kursawe [5] solve this problem by presenting an efficient strategy for exploring the boundary of arbitrary obstacles.

Theorem 18. [5] *A grid with arbitrary rectilinear obstacles can be explored in a piecemeal fashion using $O(m)$ edge traversals.*

4.2 Exploration of general graphs

The graph-theoretic abstraction of a scene can be taken even further. Suppose that the environment is modeled by a strongly connected graph $G = (V, E)$. G

can be directed or undirected. Such a general, graph-theoretic modeling of a scene allows us to neglect geometric features of the environment and to concentrate on combinatorial aspects of the exploration problem. Let n denote the number of nodes and m denote the number of edges of G.

Awerbuch *et al.* [13] consider piecemeal exploration of arbitrary undirected graphs and give a nearly optimal algorithm. The algorithm explores the graphs in strips, where each strip is explored using a breadth-first strategy.

Theorem 19. *An undirected graph can be explored in a piecemeal fashion using* $O(m + n^{1+o(1)})$ *edge traversals.*

Open Problem: Is there an algorithm that achieves an optimal bound of $O(m+n)$ on the number of traversals?

The most general graph-theoretic exploration problem was formulated by Deng and Papadimitriou [29]. The environment is now modeled by a strongly connected directed graph. At any point during the exploration process the robot knows (1) all visited nodes and edges and can recognize them when encountered again; and (2) the number of unvisited edges leaving any visited node. The robot does not know the head of unvisited edges leaving a visited node or the unvisited edges leading into a visited node. At each point in time, the robot visits a *current node* and has the choice of leaving the current node by traversing a specific known or an arbitrary (i.e. given by an adversary) unvisited outgoing edge. An edge can only be traversed from tail to head, not vice versa. As usual, the goal is to minimize the total number T of edge traversals. A piecemeal constraint does not have to be satisfied here.

If the graph is Eulerian, $2m$ edge traversals suffice [29]. For a non-Eulerian graph, let the *deficiency* d be the minimum number of edges that have to be added to make the graph Eulerian. Deng and Papadimitriou [29] suggested to study the dependence of T on m and d and showed the first upper and lower bounds. They gave a graph such that any algorithm needs $\Omega(d^2 m/\log d)$ edge traversals. This lower bound was improved by Koutsoupias [46].

Theorem 20. [46] *There exist graphs for which every exploration algorithm needs* $\Omega(d^2 m)$ *edge traversals.*

Deng and Papadimitriou gave an exponential upper bound.

Theorem 21. [29] *There is an algorithm that explores a graph with deficiency* d *using* $d^{O(d)}m$ *edge traversals.*

Deng and Papadimitriou asked the question whether the exponential gap between the upper and lower bound can be closed. The paper by Albers and Henzinger [3] is a first step in this direction: They give an algorithm that is sub-exponential in d, namely it achieves an upper bound of $d^{O(\log d)}m$. Albers and Henzinger also show that several exploration strategies based on greedy, depth-first and breadth-first approaches do not work well. There are graphs for which these strategies need $2^{\Omega(d)}m$ traversals.

We sketch the basic idea of the sub-exponential algorithm. At any time, the algorithm tries to explore new edges that have not been visited so far. That is, starting at some visited node x with unvisited outgoing edges, the robot explores new edges until it gets *stuck* at a node y, i.e., it reaches y on an unvisited incoming edge and y has no unvisited outgoing edge. Since the robot is not allowed to traverse edges in the reverse direction, an adversary can always force the robot to visit unvisited nodes until it finally gets stuck at a visited node.

The robot then relocates, using visited edges, to some visited node z with unexplored outgoing edges and continues the exploration. The *relocation* to z is the only step where the robot traverses visited edges. To minimize T one has to minimize the total number of edges traversed during all relocations. It turns out that a locally greedy algorithm that tries to minimize the number of traversed edges during each relocation is not optimal. Instead, the algorithm uses a divide-and-conquer approach. The robot explores a graph with deficiency d by exploring d^2 subgraphs with deficiencies $d/2$ each and uses the same approach recursively on each of the subgraphs. To create subgraphs with small deficiencies, the robot keeps track of visited nodes that have more visited outgoing than visited incoming edges. Intuitively, these nodes are *expensive* because the robot, when exploring new edges, can get stuck there. The relocation strategy tries to keep portions of the explored subgraphs "balanced" with respect to their expensive nodes. If the robot gets stuck at some node, then it relocates to a node z such that "its" portion of the explored subgraph contains the minimum number of expensive nodes.

Theorem 22. [3] *There is an algorithm that explores a graph with deficiency d using $d^{O(\log d)}m$ edge traversals.*

Open Problem: Is there an exploration algorithm for directed graphs that achieves an upper bound on the number of edge traversals that is polynomial in d?

5 Online problems in networks

Many online problems also arise in the area of distributed computing. We describe only a few problems here. Consider a network of processors each of which has its own local memory. Such a network can be modeled by a weighted undirected graph. The nodes of the graph represent the processor in the network and the edges represent the communication links. Let n be the number of nodes and m be the number of edges of the graph.

5.1 Migration and replication problems

First we address a problem in distributed data management, known as the *file allocation problem*. The goal is to dynamically re-allocate files in the network so that a sequence read and write requests to files an be served at low communication costs. The configuration of the system can be changed by *migrating*

and *replicating* files, i.e., a file is moved resp. copied from one local memory to another.

In the investigation of the problem, we generally concentrate on one particular file in the system. We say that *a node v has the file* if the file is contained in v's local memory. *A request at a node v* occurs if v wants to read or write the file. Immediately after a request, the file may be migrated or replicated from a node holding the file to another node in the network. We use the cost model introduced by Bartal *et al.* [15] and Awerbuch *et al.* [12]. (1) If there is a read request at v and v does not have the file, then the incurred cost is $dist(u, v)$, where u is the closest node with the file. (2) The cost of a write request at node v is equal to the cost of communicating from v to all other nodes with a file replica. (3) Migrating or replicating a file from node u to node v incurs a cost of $d \cdot dist(u, v)$, where d is the file size factor. (4) A file replica may be erased at 0 cost.

Theorem 23. [12,15] *There exist deterministic and randomized online algorithms for the file allocation problem that achieve competitive ratios of $O(\log n)$.*

The randomized solution, due to Bartal *et al.* [15], is very simple and elegant.

Coinflip: If there is a read request at node v and v does not have the file, then with probability $1/d$, replicate the file to v. If there is a write request at node v, then with probability $1/\sqrt{3}d$, migrate the file to v and erase all other file replicas.

The *file migration* problem is a restricted version of the file allocation problem where we keep only one copy of each file in the entire system. If a file is writable, this avoids the problem of keeping multiple copies of a file consistent. For this problem, constant competitive algorithms are known, see [12,14,55]. In the *file replication* problem, files are assumed to be read-only and we have to determine which local memories should contain copies of the read-only files. Constant competitive algorithms are known for specific network topologies such as uniform networks, trees and rings [4,23]. A uniform network is a complete graph in which all edges have the same length.

All of the solutions mentioned above assume that the local memories of the processors have infinite capacity. Bartal *et al.* [15] showed that if the local memories have finite capacity, then no online algorithm for file allocation can be better than $\Omega(N)$-competitive, where N is the total number of files that can be accommodated in the system. They also presented an $O(N)$-competitive algorithm for uniform networks.

Open Problem: Is there an $O(N)$-competitive algorithm for arbitrary network topologies when the nodes have limited memory capacity?

5.2 Routing problems

Many different online routing problems have been studied in the literature, see [47] for a survey. In the *virtual circuit routing problem* each communication

link e in the network has a given maximum capacity c_e. The input consists of a sequence σ of communication requests, where each request $\sigma(t)$ can be describe by a 5-tupel $(u_t, v_t, r_t, d_t, b_t)$. Here u_t and v_t are the nodes to be connected, r_t is the bandwidth requirement of the request, d_t is its duration and b_t is a certain benefit. In response to each request we wish to establish a virtual circuit on a path connecting u_t and v_t with the given bandwidth. The benefit parameter is only specified in problems where calls may also be rejected. A benefit is obtained if the call is indeed routed.

Aspnes *et al.* [9] considered the problem variant when connection requests have unlimited duration and every call has to be routed. The goal is to minimize the maximum load on any of the links. The idea of their algorithm is to assign with every edge in the network a cost that is exponential in the fraction of the capacity of the edge assigned to on-going circuits. Let $\mathcal{P} = \{P_1, \ldots, P_t\}$ be the routes assigned by the online algorithm to the first t requests. Similarly, let $\mathcal{P}^{OPT} = \{P_1^{OPT}, \ldots, P_t^{OPT}\}$ be the routes assigned by the optimal offline algorithm. For every edge e in the network, we define a *relative load* after t requests,

$$l_e(t) = \sum_{\substack{s : e \in P_s \\ s \leq t}} r_s / c_e.$$

The online algorithm given below assumes knowledge of a value Λ which is an estimate on the maximum load obtained by an optimal offline algorithm when all the requests are routed. Such a value can be obtained using a doubling strategy. Whenever the current guess turns out to bee too small, it is doubled.

Assign-Route: Let a be a constant and let (u, v, r) be the current request to be routed. Set $\overline{r} = r/\Lambda$ and $\overline{l}_e = l_e/\Lambda$ for all $e \in E$. Let

$$cost_e = a^{\overline{l}_e + \overline{r}/c_e} - a^{\overline{l}_e}$$

for all $e \in E$. Let P be a shortest path from u to v in the graph with respect to costs $cost_e$. Route the request along P and set $l_e = l_e + r/c_e$ for all edges on P.

Aspnes *et al.* [9] show that for any sequence of requests that can be routed using the given edge capacities, the maximum load achieved by Assign-Route is at most $O(\log n)$ times as large as the maximum load of an optimal solution.

Theorem 24. [9] *Assign-Route is an $O(\log n)$-competitive algorithm for the problem of minimizing the maximum load on the links.*

The virtual circuit routing problem has also been studied in its throughput version. In this variant, called the *call control problem*, a benefit is associated with every request. Requests can be accepted or rejected while link capacities may not be exceeded. Awerbuch *et al.* [11] examined the case that each call has a limited duration and showed the following result based on an algorithm similar to Assign-Route.

Theorem 25. [11] *There is an $O(\log nT)$-competitive algorithm for the problem of maximizing throughput. T denotes the maximum duration of a call.*

The bound given in Theorem 25 is tight.

6 Concluding remarks

There are many online problems related to graphs that we have not addressed in this survey. A classical problem is *online coloring*: The nodes of a graph arrive online and we wish to color them using as few colors as possible. There is a significant body of work on this problem, see e.g. [37,45,48,52]. In *online matching*, a newly arriving node can be matched to a node already present. Unweighted and weighted versions of this problem have been considered [43,41,44]. The *generalized Steiner tree problem* is an extensively studied problem where we have to construct a minimum weight tree in a graph such that certain connectivity requirements are satisfied. In the online variant nodes as well as connectivity requirements arrive online [10,21,40,56].

References

1. S. Albers. Improved randomized on-line algorithms for the list update problem. *SIAM Journal on Computing*, 27:670–681, 1998.
2. S. Albers, S. Arora and S. Khanna. Page replacement for general caching problems. *Proc. 10th Annual ACM-SIAM Symp. on Discrete Algorithms*, 31–40, 1999.
3. S. Albers and M. Henzinger. Exploring unknown environments. *Proc. 29th Annual ACM Symposium on Theory of Computing*, 416–425, 1997.
4. S. Albers and H. Koga. New on-line algorithms for the page replication problem. *Journal of Algorithms*, 27:75–96, 1998.
5. S. Albers and K. Kursawe. Exploring unknown environments with obstacles. *Proc. 10th Annual ACM-SIAM Symp. on Discrete Algorithms*, 842–843, 1999.
6. S. Albers and M. Mitzenmacher. Average case analyses of list update algorithms, with applications to data compression. *Algorithmica*, 21:312–329, 1998.
7. S. Albers, B. von Stengel and R. Werchner. A combined BIT and TIMESTAMP algorithm for the list update problem. *Information Processing Letters*, 56:135–139, 1995.
8. C. Ambühl, B. Gärtner and B. von Stengel. Towards new lower bounds for the list update problem. To appear in *Theoretical Computer Science*.
9. J. Aspnes, Y. Azar A. Fiat, S. Plotkin and O. Waarts. On-line load balancing with applications to machine scheduling and virtual circuit routing. *Proc. 25th ACM Annual ACM Symp. on the Theory of Computing*, 623–631, 1993.
10. B. Awerbuch, Y. Azar and. Y. Bartal. On-line generalized Steiner tree problem. *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms*, 68–74, 1996.
11. B. Awerbuch, Y. Azar and S. Plotkin. Throughput-competitive online routing. *34th IEEE Symp. on Foundations of Computer Science*, 32–40, 1993.
12. B. Awerbuch, Y. Bartal and A. Fiat. Competitive distributed file allocation. *Proc. 25th Annual ACM Symp. on Theory of Computing*, 164–173, 1993.
13. B. Awerbuch, M. Betke, R. Rivest and M. Singh. Piecemeal graph learning by a mobile robot. *Proc. 8th Conference on Computational Learning Theory*, 321–328, 1995.
14. Y. Bartal, M. Charikar and P. Indyk. On page migration and other relaxed task systems. *Proc. of the 8th Annual ACM-SIAM Symp. on Discrete Algorithms*, 43–52, 1997.

15. Y. Bartal, A. Fiat and Y. Rabani. Competitive algorithms for distributed data management. *Proc. 24th Annual ACM Symp. on Theory of Computing*, 39–50, 1992.
16. L.A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:78–101, 1966.
17. S. Ben-David, A. Borodin, R.M. Karp, G. Tardos and A. Wigderson. On the power of randomization in on-line algorithms. *Algorithmica*, 11:2-14,1994.
18. J.L. Bentley, K.L. Clarkson and D.B. Levine. Fast linear expected-time algorithms for computing maxima and convex hulls. *Proc. of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, 179–187, 1990.
19. J.L. Bentley and C.C. McGeoch. Amortized analyses of self-organizing sequential search heuristics. *Communication of the ACM*, 28:404–411, 1985.
20. J.L. Bentley, D.S. Sleator, R.E. Tarjan and V.K. Wei. A locally adaptive data compression scheme. *Communication of the ACM*, 29:320–330, 1986.
21. P. Berman and C. Coulston. Online algorithms for Steiner tree problems. *Proc. 29th Annual ACM Symposium on Theory of Computing*, 344–353, 1997.
22. M. Betke, R. Rivest and M. Singh. Piecemeal learning of an unknown environment. *Proc. 5th Conference on Computational Learning Theory*, 277–286, 1993.
23. D.L. Black and D.D. Sleator. Competitive algorithms for replication and migration problems. Technical Report Carnegie Mellon University, CMU-CS-89-201, 1989.
24. M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. DEC SRC Research Report 124, 1994.
25. M. Chrobak, H. Karloff, T. Paye and S. Vishwanathan. New results on the server problem. *SIAM Journal on Discrete Mathematics*, 4:172–181, 1991.
26. M. Chrobak and J. Noga. LRU is better than FIFO. *Proc. 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, 78–81, 1998.
27. A. Borodin, S. Irani, P. Raghavan and B. Schieber. Competitive paging with locality of reference. *Journal on Computer and System Sciences*, 50:244–258, 1995.
28. X. Deng, T. Kameda and C. H. Papadimitriou. How to learn an unknown environment. *Journal of the ACM*, 45:215–245, 1998.
29. X. Deng and C. H. Papadimitriou. Exploring an unknown graph. *Proc. 31st Symposium on Foundations of Computer Science*, 356–361, 1990.
30. A. Fiat and A. Karlin. Randomized and multipointer paging with locality of reference. *Proc. 27th Annual ACM Symposium on Theory of Compting*, 626–634, 1995.
31. A. Fiat and Z. Rosen. Experimental studies of access graph based heuristics: Beating the LRU standard. *Proc. 8th Annual ACM-SIAM Symp. on Discrete Algorithms*, 63–72, 1997.
32. M.J. Golin. PhD thesis, Department of Computer Science, Princeton University, 1990. Technical Report CS-TR-266-90.
33. F. Hoffmann, C. Icking, R. Klein and K. Kriegel. A competitive strategy for learning a polygon. *Proc. 8th ACM-SIAM Symposium on Discrete Algorithms*, 166–174, 1997.
34. F. Hoffmann, C. Icking, R. Klein and K. Kriegel. The polygon exploration problem:A new strategy and a new analysis technique. *Proc. 3rd International Workshop on Algorithmic Foundations of Robotics*, 1998.
35. S. Irani. Two results on the list update problem. *Information Processing Letters*, 38:301–306, 1991.
36. S. Irani. Corrected version of the SPLIT algorithm. Manuscript, January 1996.
37. S. Irani. Coloring inductive graphs on-line. *Algorithmica*, **11**(1):53–62, 1994.

38. S. Irani. Page replacement with multi-size pages and applications to Web caching. *Proc. 29th Annual ACM Symp. on Theory of Computing*, 701–710, 1997.
39. S. Irani, A.R. Karlin and S. Phillips. Strongly competitive algorithms for paging with locality of reference. *Proc. 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, 228–236, 1992.
40. M. Imase and B.M. Waxman. Dynamic Steiner tree problems. *SIAM Journal on Discrete Mathematics*, 4:349–384, 1991.
41. B. Kalyanasundaram and K. Pruhs. Online wieghted matching. *Journal of Algorithms*, 14:139–155, 1993.
42. R. Karp and P. Raghavan. From a personal communication cited in [49].
43. S. Khuller, S.G. Mitchell and V.V. Vazirani. On-line weighted bipartite matching. *Proc. 18th International Colloquium on Automata, Languages and Programming (ICALP)*, Springer LNCS, Vol. 510, 728–738, 1991.
44. R. Karp, U. Vazirani and V. Vazirani. An optimal algorithm for online bipartite matching. *Proc. 22nd ACM Symp. on Theory of Computing*, 352–358, 1990.
45. H.A. Kierstead. The linearity of First-Fit coloring of interval graphs. *SIAM Journal on Discrete Mathematics*, 1:526–530, 1988.
46. J. Kleinberg. On-line search in a simple polygon. *Proc. 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, 8–15, 1994.
47. S. Leonardi. On-line network routing. *Online Algorithms: The State of the Art*, Edited by A. Fiat and G. Woeginger, Springer LNCS, Volume 1442, 242-267, 1998.
48. L. Lovasz, M. Saks and M. Trotter. An online graph coloring algorithm with sublinear performance ratio. *Discrete Mathematics*, 75:319–325, 1989.
49. N. Reingold, J. Westbrook and D.D. Sleator. Randomized competitive algorithms for the list update problem. *Algorithmica*, 11:15–32, 1994.
50. D.D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.
51. B. Teia. A lower bound for randomized list update algorithms. *Information Processing Letters*, 47:5–9, 1993.
52. S. Vishwanathan. Randomized online graph coloring. *Journal of Algorithms*, 13:657–669, 1992.
53. N. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11:525–541, 1994.
54. N.E. Young. Online file caching. *Proc. 9th Annual ACM-SIAM Symp. on Discrete Algorithms*, 82–86, 1998.
55. J. Westbrook. Randomized algorithms for the multiprocessor page migration. *SIAM Journal on Computing*, 23:951–965, 1994.
56. J. Westbrook and D.C. Yan. Lazy and greedy: On-line algorithms for Steiner problems. *Proc. Workshop on Algorithms and Data Structures*, Springer LNCS Volume 709, 622–633, 1993.