Dynamic Storage Allocation with Known Durations*

Joseph (Seffi) Naor† Ariel Orda‡ Yael Petruschka§

Abstract

This paper is concerned with a new version of on-line storage allocation in which the durations of all processes are *known* at their arrival time. This version of the problem is motivated by applications in communication networks and has not been studied previously. We provide an on-line algorithm for the problem with a competitive ratio of $O(\min\{\log \Delta, \log \tau\})$, where Δ is the ratio between the longest and shortest duration of a process, and τ is the maximum number of concurrent active processes that have different durations. For the special case where all durations are powers of two, the competitive ratio achieved is $O(\log \log \Delta)$.

1 Introduction

This paper is concerned with the classic problem of dynamic storage allocation (DSA). In a typical on-line setting of the problem, a sequence of processes, each requesting a certain amount of memory space, arrive at a storage device. Upon arrival of a process, a contiguous storage area is allocated for it. This area is liberated for reuse only when the process terminates at some later point of time, which, in general, is *unknown* at the time of allocation. As a result, "holes" of wasted space are generated over time in the storage device. The objective is to find an allocation algorithm that minimizes the wasted space. The decisions of this algorithm must be based at each point of time on the known current state of the storage device without knowledge of future events.

A common way for measuring the quality of an on-line algorithm is competitive analysis. Here, the costs associated with an on-line algorithm are compared with the costs expended by an optimal off-line algorithm that knows the sequence of events in advance. The maximum ratio between their respective costs, taken over all sequences, is called the *competitive ratio*. This study focuses on competitive analysis. An alternative approach to measure the quality of on-line algorithms is through the concept of *distributional* (or *average case*) complexity, which relies on some hypothesis on the distribution of events. Each of the two approaches has clear advantages as well as limitations, and the reader is referred to [BEY98] for a related discussion. Here we confine ourselves to mention that competitive analysis has been employed in decision theory, and has been found to be of practical

*A preliminary version of this paper appeared in the Proceedings of the 5th Annual European Symposium on Algorithms, pp. 378–387, Graz, Austria, 1997. This work was supported by the Broadband Telecommunications R&D Consortium administered by the chief scientist of the Israeli Ministry of Industry & Trade.

†Computer Science Dept., Technion, Haifa 32000, Israel (naor@cs.technion.ac.il).

‡Electrical Engineering Dept., Technion, Haifa 32000, Israel (ariel@ee.technion.ac.il).

§Computer Science Dept., Technion, Haifa 32000, Israel (petrus@cs.technion.ac.il).

relevance in various fields, such as telecommunication networks (in particular networks that provide Quality of Service guarantees) and financial planning. Indeed, in these and other areas, one often necessitates to provide absolute guarantees on performance, which calls for the worst-case approach of competitive analysis.

Broadband communication networks are an important area of potential application of dynamic storage allocation algorithms. In such networks, bandwidth is allocated to sessions, according to their quality of service requirements. Typically, this allocation is contiguous, e.g., in a time-division multiplexed (TDM) environment a session is allocated successive time slots. Indeed, an uncontiguous allocation may incur higher overhead in terms of bandwidth management (see [SS94] for a related discussion). Replacing the standard terms of "storage" or "memory" with "bandwidth", and "process" with "session", dynamic storage allocation concepts can be applied to the networking framework. Typically, the duration time of broadband applications is either known *a priori* or can be predicted fairly well, e.g., video on demand, reserved video conferences. Moreover, it has been recently recognized [Par94] that quality of service guarantees should often be specified on a *time interval* basis, which is facilitated when the session duration is known. Such knowledge on session duration has been utilized in the design of on-line network algorithms, e.g. [BNCK+95, AAP93, AKP+93]. However, in the context of dynamic storage allocation, this has not been done, and, in effect, is the objective of the present study.

1.1 Previous work

The dynamic storage allocation problem has been extensively investigated since the Fifties. Basic methods are discussed in Knuth [Knu73, Section 2.5]. A survey of results in this area until the early Eighties appears in Coffman [Cof83]. Probabilistic analysis of dynamic storage allocation heuristics was studied in several papers, e.g., [CFL90] and the references therein.

Two intuitive heuristics that were developed early on are *First Fit* and *Best Fit*. First Fit finds the first free area that can fit the requirement of the current process; Best Fit finds the free area that best fits the requirement of the current process, i.e., the one that causes minimal fragmentation. In the early Seventies, Robson [Rob71, Rob74] gave an algorithm which had a competitive ratio of $O(\log(W_{\max}))$, where $W_{\max}$ is the maximum memory space requested by a process. He also showed that this is the best possible bound up to constant factor. Subsequently, he showed [Rob77] that the competitive ratio of First Fit is also $\Theta(\log(W_{\max}))$. Surprisingly, Robson [Rob77] also showed that the competitive ratio of Best Fit can be as bad as $\Omega(W_{\max})$.

Segregated storage methods [Sta95] constitute a different category for DSA. These methods include many variations. The basic idea here is to partition the memory into blocks, such that each block only contains processes that have the same (or similar) requirement.

Recently, Luby, Naor and Orda [LNO96] established a competitive ratio of $\Theta\left(\min\{\log k, \log W_{\max}\}\right)$ for both First Fit and some segregated storage methods, where k is the maximum number of concurrent processes. Furthermore, by proving a lower bound on the competitive ratio in the randomized case which is very close to the deterministic bound, they showed that randomization may be of little help here.

The off-line version of the storage allocation problem is NP-complete in the strong sense [GJ79, page 226]. Yet, polynomial time approximation algorithms for this problem were devised. Kierstead [Kie88] showed that if the processes are sorted by their storage requirements, then the competitive ratio of First Fit is 80. This approximation factor was later improved by Kierstead [Kie91] to 6. Recently, Gergov [Ger96] achieved an approximation factor of 5.

The *strip packing* problem is defined as follows: a set of boxes with given heights and widths has to be packed (orthogonally) in a strip of width W so as to minimize the total height of the strip needed. We note that this problem is related to dynamic storage allocation with known time durations, since each process can be viewed as defining a box, where one dimension corresponds to time and the other dimension corresponds to memory. On-line dynamic storage allocation corresponds to a version of on-line strip packing where the boxes appear from left to right, their horizontal placement is fixed, and only their vertical placement can be changed. Strip packing has been studied mainly in the off-line setting (see, e.g., [BBK81, BCR80]). The only on-line algorithms analyzed from a worst case point of view are the shelf algorithms of [BS83].

1.2 Our contribution

We investigate the dynamic storage allocation problem in the case where the duration of a process is known at the time of its arrival. That is, upon arrival of a process, a contiguous storage area is allocated for it for a prespecified period of time. Our main result is an on-line algorithm for dynamic storage allocation with known durations. Denote by τ the maximum number of *concurrent* active processes that have different durations, and denote by Δ the ratio between the longest and shortest duration of a process. The competitive ratio of our algorithm is $O(\min\{\log \Delta, \log \tau\})$. For the special case that all durations are powers of two, the competitive ratio is actually $O(\log \log \Delta)$. The main feature of our on-line algorithm is that it distinguishes between the allocation for "heavy" and "light" processes, where this distinction is based on *both* storage area requirements and durations of the processes.

An intriguing open problem is to establish a (non-trivial) lower bound on the competitive ratio. We note that the proofs of the lower bounds in the case of unknown durations (e.g., [Rob77, LNO96]) do not seem to carry over to the case of known durations. These proofs heavily use the fact that the adversary can dynamically determine the duration of a process, based on the storage configuration chosen by the on-line algorithm.

2 Preliminaries

The storage allocation problem is equivalent to the problem of finding a weighted coloring of a given weighted interval graph.

Definition 1: An undirected graph $G = (V, E)$ is called an *interval graph* if its set of vertices V can be put into one-to-one correspondence with a set of intervals I, such that two vertices are connected by an edge if and only if their corresponding intervals overlap. An interval graph is called *weighted* if each interval is associated with a positive weight.

Definition 2: A *weighted coloring* of an interval graph G is a coloring of each interval i with a range $[a_i, b_i]$ (where a_i,b_i are integers), such that $b_i - a_i + 1$ is equal to the weight of interval i, and there is no intersection between the ranges of two overlapping intervals.

The processes form an interval graph denoted by $G = (V, E)$, where a process corresponds to the interval defined (on a time axis) between its arrival and departure time. Each interval is associated with a weight which equals the storage area required by the corresponding process. The objective is, thus, to minimize $\max_i(b_i)$ in a weighted coloring of G.

We denote the maximum weight, taken over all intervals, by $W_{\max}$ and the weight of the heaviest clique in G by $\omega^*(G)$ (or simply by ω^* when G is clear from the context). Let τ be the maximum number of concurrent different length intervals and let Δ be the ratio between the longest and shortest interval in G.

In the following, the storage requested by a process is referred to as the weight of the process. For convenience, we refer interchangeably to the duration of a process and to the length of its corresponding interval. The storage device consists of memory cells that are addressed consecutively from zero. The space that is used by an algorithm is equal to the value of the highest address of a memory cell that was used by the algorithm.

The notion of competitive ratio was introduced by Sleator and Tarjan [ST85] (see also [BEY98]). Let I be a finite input sequence for an on-line algorithm A. Let $\mathrm{OPT}(I)$ denote the cost associated with an optimal (off-line) algorithm that receives the input sequence I in advance. Let $A(I)$ denote the cost associated with algorithm A on input sequence I. Then, A is said to be c-competitive if there is a constant α such that for all finite sequences I,

$$A(I) \leq c \cdot \mathrm{OPT}(I) + \alpha$$

In the case of dynamic storage allocation, the cost associated with an algorithm on an input sequence I is the space used by the algorithm for allocating the requests of the processes in I.

3 The Algorithms

Herein, we present two on-line algorithms for DSA with known durations. The algorithm presented in subsection 3.2 achieves a competitive ratio of $O(\log \Delta)$. The algorithm presented in subsection 3.3, achieves a competitive ratio of $O(\log \tau)$. Since Δ and τ are incomparable in general, no algorithm is "better" than the other. In Section 3.4 we show that both algorithms may be used "concurrently" to achieve a competitive ratio of $O(\min\{\log \Delta, \log \tau\})$. The algorithms are presented under the assumption that ω^* is known in advance. In addition, the algorithm in subsection 3.3 is presented under the assumption that τ is known in advance. In Section 3.4 we show how to avoid these assumptions.

We start in subsection 3.1 by presenting algorithms for DSA with known durations in the simple case where the time intervals of the processes are not properly contained in each other. The techniques of these algorithms are used later on, in subsections 3.2 and 3.3, for handling the general case.

3.1 Simple cases

Consider the following two special cases of interval graphs.

Definition 3: A *proper interval graph* is an interval graph induced by a set of intervals such that no interval properly contains another interval.

Definition 4: A *unit interval graph* is an interval graph induced by a set of intervals which are all of the same length.

We note that in general, the classes of unit interval graphs and proper interval graphs coincide [Rob69].

First, we present a simple on-line algorithm for the DSA problem with known durations in the case that the processes form a proper interval graph. This algorithm achieves a competitive ratio of $O(1)$ using at most $2\omega^*$ memory space.

Let w_i denote the weight of process i. Let a_i denote the highest address of a memory cell that is occupied by process i, and let $a_0 = -1$.

The Cyclic Algorithm
Upon arrival of process i:

- If $a_{i-1} + w_i < 2\omega^*$, then allocate w_i consecutive memory cells for process i, starting from memory address $a_{i-1} + 1$.
- Else, allocate w_i consecutive memory cells for process i, starting from memory address 0.

Lemma 1: The Cyclic Algorithm always succeeds in allocating memory space for a process in a storage device of size $2\omega^*$.

Proof: For ease of exposition, we consider a cyclic storage device where address 0 follows address $2\omega^* - 1$. Since the processes form a proper interval graph, it is guaranteed that for all $i \geq 2$, process i terminates after process $i-1$. Hence, by the algorithm, at any point of time, all occupied memory cells must be contained in a single block with at most one hole of size less than $W_{\max}$. The size of this block (with the hole) is at most $\omega^* + W_{\max}$, and therefore a storage device of size $2\omega^*$ is large enough to serve all memory requests. □

We now present an alternative on-line algorithm for the case that the processes form a unit interval graph. This algorithm also achieves a competitive ratio of $O(1)$ using at most $2\omega^*$ memory space. We use it in subsection 3.3 as a procedure for the general case.

Denote by T the length of the intervals in G. Let w_i be the weight of process i and suppose it arrives at time $t = T \cdot j + k$ where $k < T$.

The Alternating Algorithm
Upon arrival of process i:

- If j is even, allocate w_i consecutive memory cells for process i, above the highest occupied address, within the first area of ω^* memory cells.
- If j is odd, allocate w_i consecutive memory cells for process i, above the highest occupied address, within the second area of ω^* memory cells (starting from memory address ω^*).

Lemma 2: The Alternating Algorithm always succeeds in allocating memory space for a process in a storage device of size $2\omega^*$.

Proof: Let us consider time phases of length T, where phase j is defined to be between time $T \cdot j$ and time $T \cdot (j+1) - 1$, $(j \geq 0)$. Phase j is referred to as *even* if j is even, otherwise it is *odd*. Since the processes form a unit interval graph where the length of each interval is T, it is guaranteed that all the processes that arrive within a certain phase must terminate within the next phase and have overlapping intervals. Hence, by the algorithm, at any time, all occupied memory cells in the first (second) area of ω^* memory cells must be contained in a single block whose size is at most ω^*, and this block is vacant and ready for reuse within the next even (odd) phase. Therefore, a storage device of size $2\omega^*$ is large enough to serve all memory requests. □

3.2 An $O(\log \Delta)$ algorithm

Based on the Alternating Algorithm, we present an on-line algorithm whose competitive ratio is $O(\log \Delta)$. We first prove the following technical lemma.

Lemma 3: Let $G = (V, E)$ be a weighted interval graph with intervals of length T, $2^{k-1} \leq T < 2^k$, $(k \geq 1)$. Let the weight of the heaviest clique in G be $\omega^*(G) = \omega^*$. Let G' be the weighted (unit) interval graph that is generated from G by extending to the right all intervals to be of length 2^k. Then, any clique of G' weighs at most $2\omega^*$.

Proof: Assume that G' has a clique that weighs more than $2\omega^*$, and let t_0 be a time point which is contained in all the intervals of that clique. Denote by $\omega_G(t)$ the total weight of intervals in G that contain a time point t. Then, for any time point t, $\omega_G(t) \leq \omega^*$, and especially $\omega_G(t_0) \leq \omega^*$. Correspondingly, denote by $\omega_{G'}(t)$ the total weight of intervals in G' that contain a time point t. By assumption, $\omega_{G'}(t_0) > 2\omega^*$. Now, consider the time point $t_1 = t_0 - 2^{k-1}$. Since the original intervals of G are at least of length 2^{k-1}, and they were extended by at most 2^{k-1}, it follows that $\omega_G(t_1) \geq \omega_{G'}(t_0) - \omega_G(t_0) > \omega^*$. Thus, we get a contradiction. □

In the following algorithm, the length of all intervals is rounded up to the closest power of 2. We assume that the storage area is divided into slots of size $4\omega^*$ memory cells each. The main idea of the algorithm is that, at any time, a slot which is not empty contains only processes having the same (rounded) duration. A slot is referred to as a slot of *type* i if it contains processes whose (rounded) duration is 2^i.

The Slots Algorithm:

Consider a process p with duration $2^{i-1} \leq T < 2^i$ that arrives at time $t = 2^i j + k$, where $k < 2^i$. Let w be the weight of process p.

1. Look for a slot of type i. (If there is no such slot, pick the first empty slot.)
2. Use the Alternating Algorithm to allocate w consecutive memory cells for process p within that slot:
   (a) If j is even, allocate above the highest occupied address, within the first half of the slot (i.e., the first $2\omega^*$ memory cells of the slot).
   (b) If j is odd, allocate above the highest occupied address, within the second half of the slot (i.e., the last $2\omega^*$ memory cells of the slot).

Lemma 4: The Slots Algorithm always succeeds in allocating memory space for a process in the storage device.

Proof: It suffices to show that, for each $i \geq 0$, the Alternating Algorithm in stage 2 succeeds in allocating memory space for all the processes with duration 2^i within the corresponding slot of size $4\omega^*$. For each $i \geq 0$, Lemma 3 yields that the total weight of the processes with duration 2^i is at most $2\omega^*$. Since the processes with duration 2^i form a unit interval graph, Lemma 2 yields that a slot of size $4\omega^*$ is large enough to serve all memory requests by the Alternating Algorithm. □

Performance of the algorithm: At any time there is at most one slot of each possible type, and the number of possible types is at most $\log \Delta$. Since the algorithm always chooses the first empty slot, only the first $\log \Delta$ slots of the storage area may contain processes. Thus the algorithm achieves a competitive ratio of $O(\log \Delta)$ using at most $4\omega^* \log \Delta$ memory space. □

3.3 An $O(\log \tau)$ algorithm

Recall that τ is the maximum number of concurrent different durations. We now present an algorithm whose competitive ratio is $O(\log \tau)$. For the special case that all time intervals are powers of two, the competitive ratio is $O(\log \log \Delta)$, since $\tau \leq \log \Delta$.

The main idea of the algorithm is to separate between the allocation of "heavy" processes, i.e., processes requiring large amount of storage, and "light" processes, i.e., processes requiring small amount of storage. Both storage area requirements and durations of the processes are used to distinguish between "heavy" and "light" processes. This yields that, on one hand, the number of concurrent heavy processes is limited, while on the other hand, a relatively small size slot is required for allocating memory space for light processes with the same duration.

Let the first $6\omega^* + 2\omega^*/\tau$ memory cells of the storage area be divided into $3\tau + 1$ slots of size $2\omega^*/\tau$ memory cells each. All processes contained concurrently in a slot must have the same duration, as is the case with the Slots Algorithm.

To clarify the exposition, we adopt the following notation. A slot is called *empty* if it contains no processes. Otherwise, the slot is called *active*. An active slot is referred to as a slot of *type* T if the processes it contains have duration T. An active slot of any type is called *full* if more than ω^*/τ memory cells of the slot are occupied, otherwise, it is called *absorbing* if memory address ω^*/τ of the slot is vacant, and *releasing* if the first memory address of the slot is vacant.

The Separation Algorithm

Consider a process p with duration T and weight w.

1. If $w \geq \omega^*/\tau$, allocate memory space for process p in the storage area above memory address $6\omega^* + 2\omega^*/\tau$, using either First-Fit or the Coloring Algorithm [LNO96].
2. Else ($w < \omega^*/\tau$):
   (a) If there is no active slot of type T, look for an empty slot and allocate w consecutive memory cells for process p, starting from the first memory address of the slot.
   (b) Else, look for an active slot of type T which contains the process with the latest arrival time:
      i. If it is an absorbing slot, allocate w consecutive memory cells for process p, above the highest occupied address within the slot.
      ii. Else, look for an empty slot and allocate w consecutive memory cells for process p, starting from the first memory address of the slot.

Lemma 5: The Separation Algorithm always succeeds in allocating memory space for a process in the storage device.

Proof: It suffices to show that an empty slot can always be found in stages 2a and 2(b)ii of the algorithm. We show that there are at most 3τ active slots at any point of time. Since the total number of slots is $3\tau + 1$, there is always at least one empty slot.

Consider processes with duration T; these processes form a unit interval graph. Since all occupied memory cells in a slot must be contained in a single block, we claim that an active slot of type T which is not full, must be either an absorbing slot, or a releasing slot (or both). Furthermore, at each point of time there can be at most one absorbing slot of type T, and at most one releasing slot of type T. Indeed, this is the case since any two slots of type T can be ordered such that the arrival time of all processes belonging to the first slot precedes the arrival time of the processes belonging to the second slot. Hence, at any point of time, there can be at most τ absorbing slots of all types, and τ releasing slots of all types. To complete the proof, we note that there can be at most τ concurrent full slots of all types, since a full slot contains at least ω^*/τ occupied memory cells. $\square$

Performance of the algorithm: The first stage of the Separation Algorithm is First Fit with respect to the processes with weights greater than ω^*/τ. This stage requires at most $O(\omega^* \log \tau)$ memory space [LNO96], since the maximum number of concurrent processes is less than or equal to τ. The second stage of the Separation Algorithm requires $O(\omega^*)$ memory space, since it uses $3\tau + 1$ slots of size $2\omega^*/\tau$. Hence, the Separation Algorithm achieves a competitive ratio of $O(\log \tau)$. $\square$

3.4 Implementation issues

In the algorithms presented thus far, we assumed that the value of ω^* is known in advance. Also, the Separation Algorithm is presented under the assumption that τ is known in advance. We now show how to avoid these assumptions. We also outline how to combine the Slots Algorithm and the Separation Algorithm in order to obtain a competitive ratio of $O(\min\{\log\Delta, \log\tau\})$.

At any given point of time, the value of ω^* that should be used by the algorithm is the maximum concurrent storage requirement thus far, rounded up to the closest power of 2. This value is updated during the execution of the algorithm. Upon each such update, the algorithm "resets" the bottom of the storage area to be the highest memory address used thus far. The resets can, at most, double the storage area used in total.

Similarly, at any given point of time, the value of τ which should be used by the algorithm is the maximum number thus far of concurrent processes having different durations, rounded up to the closest power of 2. This value is updated during the execution of the algorithm. Upon each such update, we reset the area allocated for the "light" processes. As there can be at most $O(\log\tau)$ such resets, the total area allocated for the "light" processes is at most $O(\omega^*\log\tau)$. As for the "heavy" processes, it is easy to verify that if they are handled by the Coloring Algorithm of [LNO96], no resets of the area allocated for these processes is needed.

It is also easy to verify that the Slots Algorithm and the Separation Algorithm can be executed in parallel, since essentially, they both use segregated storage methods. Upon arrival of a process, memory space is allocated by the algorithm with the (current) better competitive ratio. Referring to this parallel execution of the two algorithms as to the *Combined Algorithm*, we obtain the following result.

Theorem 6: The competitive ratio of the Combined Algorithm is $O(\min\{\log\Delta, \log\tau\})$.

References

[AAP93] B. Awerbuch, Y. Azar, and S. Plotkin. Throughput-competitive of on-line routing. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, pages 32–40, October 1993.

[AKP+97] Y. Azar, B. Kalyanasundaram, S. Plotkin, K. R. Pruhs, and O. Waarts. On-line load balancing of temporary tasks. *Journal of Algorithms*, 22(1):93–110, 1997.

[BBK81] B. S. Baker, D. J. Brown, and H. P. Katseff. A 5/4 algorithm for two-dimensional packing. *Journal of Algorithms*, 6:348–368, 1981.

[BCR80] B. S. Baker, E. G. Coffman, and R. L. Rivest. Orthogonal packings in two dimensions. *SIAM Journal on Computing*, 9:846–855, 1980.

[BEY98] A. Borodin and R. El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 1998.

[BNCK+99] A. Bar-Noy, R. Canetti, S. Kutten, Y. Mansour, and B. Schieber. Bandwidth allocation with preemption. *SIAM Journal on Computing*, 28:1806–1828, 1999.

[BS83] B. S. Baker and J. S. Schwartz. Shelf algorithms for two-dimensional packing problems. *SIAM Journal on Computing*, 12:508–525, 1983.

[CFL90] E. G. Coffman, Leopold Flatto, and F. T. Leighton. First-fit storage of linear lists: tight probabilistic bounds on wasted space. In *Proc. 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 272–279. SIAM, 1990.

[Cof83] E. G. Coffman. An introduction to combinatorial models of dynamic storage allocation. *SIAM Review*, 25:311–325, 1983.

[Ger96] J. Gergov. Approximation algorithms for dynamic storage allocation. In *Proc. of the 4th Annual European Symposium on Algorithms*, pages 52–61, 1996.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and intractability - a guide to the theory of NP-completeness*. W. H. Freeman, San Francisco, 1979.

[Kie88] H. A. Kierstead. The linearity of first-fit coloring of interval graphs. *SIAM Journal on Discrete Math*, 1:526–530, 1988.

[Kie91] H. A. Kierstead. A polynomial time approximation algorithm for dynamic storage allocation. *Discrete Mathematics*, 88:231–237, 1991.

[Knu73] D. E. Knuth. *Fundamental algorithms*, volume 1. Addison-Wesley, Reading, MA, second edition, 1973.

[LNO96] M. G. Luby, J. Naor, and A. Orda. Tight bounds for dynamic storage allocation. *SIAM Journal on Discrete Math*, 9(1):155–166, 1996.

[Par94] C. Partridge. *Gigabit Networking*. Addison-Wesley, 1994.

[Rob69] F. S. Roberts. Indifference graphs. In F. Harary, editor, *Proof Techniques in Graph Theory*, pages 139–146. Academic Press, N.Y., 1969.

[Rob71] J. M. Robson. An estimate of the store size necessary for dynamic storage allocation. *Journal of the ACM*, 18:416–423, 1971.

[Rob74] J. M. Robson. Bounds for some functions concerning dynamic storage allocation. *Journal of the ACM*, 12:491–499, 1974.

[Rob77] J. M. Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *Computer Journal*, 20:242–244, 1977.

[SS94] O. Sharon and A. Segal. Schemes for slot reuse in CRMA. *IEEE/ACM Transactions on Networking*, 2(3):269–278, 1994.

[ST85] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.

[Sta95] T. A. Standish. *Data structures, algorithms and software principles in C*. Addison-Wesley, Reading, MA, second edition, 1995.