Tech. Rpt. # UMIACS-TR-99-31, Institute for Advanced Computer Studies,
University of Maryland, College Park, MD 20742, June 3, 1999.

Approximation Algorithms and Heuristics for the Dynamic Storage Allocation Problem

Praveen K. Murthy
Angeles Design Systems
pmurthy@angeles.com

Shuvra S. Bhattacharyya
ECE Dept., and Institute for Advanced Computer Studies, University of Maryland, College Park
ssb@eng.umd.edu

Abstract

In this report, we look at the problem of packing a number of arrays in memory efficiently. This is known as the dynamic storage allocation problem (DSA) and it is known to be NP-complete. We develop some simple, polynomial-time approximation algorithms with the best of them achieving a bound of 4 for a subclass of DSA instances. We report on an extensive experimental study on the FirstFit heuristic and show that the average-case performance on random instances is within 7% of the optimal value.

1 Introduction

Formally, the DSA problem is the following:

Definition 1: Let B be the set of intervals (corresponding to the arrays). Let $N = |B|$, the number of elements in B. For each $b \in B$, $s(b)$ is the time at which it becomes live, $e(b)$ is the time at which it dies, and $w(b)$ is the **width** of interval b; the width represents, for example, the size of an array that needs allocation. Note that the **duration** of an interval is $e(b) - s(b)$. Given the s, e, w values for each $b \in B$, and an integer K, is there an allocation of these intervals that requires total storage of K units or less? By an allocation, we mean a function $A: B \to \{0, \ldots, K-1\}$ such that $0 \le A(b) \le K - w(b)$ for each $b \in B$, and if two intervals b_1 and b_2 intersect; i.e, if $s(b_1) \le s(b_2) \le e(b_1)$ or $s(b_2) \le s(b_1) \le e(b_2)$, then $A(b_1) + w(b_1) \le A(b_2)$ or $A(b_2) + w(b_2) \le A(b_1)$. The allocation function is also called a **coloring** sometimes.

The "dynamic" in DSA refers to the fact that many times, the problem is online in nature: the allocation has to be performed as the intervals come and go. Our interest in this problem stems from scheduling dataflow graphs [7]. The subset of dataflow that we are interested in [7][1], called synchronous dataflow (SDF) [5], can be scheduled statically at compile time. Hence, the problem is not really "dynamic" since the lifetimes and size of all the arrays that need to be allocated are known at compile time; thus, the problem should perhaps be called static storage allocation. But we will use the term DSA since this is consistent with the literature.

Theorem 1: [2] DSA is NP-complete, even if all the widths are 1 and 2.

2 Some notation

An **instance** is a set of lifetimes. An **enumerated instance** is an instance with some ordering of the intervals. For an instance, we have associated with it a **weighted interval graph** (**WIG**) $G_B = (V, E)$ where V is the set of intervals, and E is the set of edges. There is an edge between two intervals iff they overlap in time. The graph is node-weighted by the widths of the intervals. For any subset of nodes $U \subset V$, we define the weight of U, $w(U)$ to be the sum of the widths $w(v)$ for all $u \in U$. The **maximum clique *weight*** (**MCW**) in the WIG is the clique with the largest weight, and is denoted $\tilde{\omega}(G_B)$. We also denote the **clique of the largest *size* (MCS)** (i.e, number of nodes) by $\omega(G_B)$. The MCW corresponds to the maximum number of values that are live at any point. The MCS corresponds to the maximum number of arrays that are live at any point. The **chromatic number (CN)**, denoted $\chi(G_B)$, for G is the minimum K such that there is a feasible allocation in definition 1.

Theorem 2: [4] $\frac{5}{4} \le sup_B\left(\frac{\chi(G_B)}{\tilde{\omega}(G_B)}\right) \le 6$,

where the supremum is taken over all instances B of the DSA problem. What this means is that there exist instances where the chromatic number is at least 5/4 times the MCW, but for any instance, the chromatic number is less than 6 times the MCW. The upper bound in theorem 2 is not tight.

Theorem 3: If all of the widths are the same, then

$$\tilde{\omega}(G_B) = \chi(G_B).$$

First fit (FF) is the algorithm that performs allocation for an enumerated instance by assigning the smallest feasible location to each interval in order. It does not reallocate intervals, and it does not consider intervals not yet assigned. The pseudocode for this algorithm is shown in figure 1. As can be seen, building the interval graph takes $O(N^2)$ time in the worst case (all intervals overlap with each other), and the first-Fit procedure takes time $O(N^2 \cdot \log(N))$ in the worst case if every interval overlaps with every other interval.

Theorem 4: FF on an enumerated instance where all the widths are the same, ordered by start times, is optimal.

Proof: Theorem 3 and 4 are classic [3] and follow easily from the observation that FF will assign the ith interval I location M if and only if there is some interval live at every location in $\{0, \ldots, M-1\}$. But this means that the MCW must be at least $M + w(I)$.

When all the intervals have the same size, FF by start times can be made to run in time $O(N \cdot \log(N))$, where N is the number of intervals. The time is dominated by the sorting step; without the sorting, the running time is actually $O(N)$. The pseudocode is shown in figure 2; notice that we do not

```
Procedure FirstFit(enumerated instance I)

G = buildIntervalGraph(I)
Vector allocate //allocate is an array to contain the allocations

foreach interval i in I do
      allocate(i) ← 0  //initial allocation at 0
      neighborsAllocations ← { }
      foreach neighbor j of i from G
            if (j appears before i in I)
                  neighborsAllocations ← neighborsAllocations ∪ {allocate(j)}
            fi
      end for
      sort(neighborsAllocations)
      foreach allocation a ∈ neighborsAllocations
            if allocate(i) conflicts with a
                  //width(a) = width of interval with allocation a
                  allocate(i) ← a + width(a)
            fi
      end for
end for
Procedure buildIntervalGraph(enumerated instance I)

sort I by start times
N ← number of intervals in I
// G is an adjacency list representation containing N rows
// and list pointer at each G(i)
Graph G

foreach i in {1,...,N}
      j ← i + 1
      while (start time of I(j) < end time of I(i))
            G(i) ← G(i) ∪ {j}
            G(j) ← G(j) ∪ {i}
            j ← j + 1
      end while
end for
```

Fig 1. Pseudocode definition of the FirstFit heuristic.

need to create the interval graph, and it does not matter whether we assign the smallest possible space to an interval (so the algorithm is not technically "first" fit, but the idea is the same).

3 Approximation algorithms

Since the general problem is NP-complete, we have to resort to heuristics. The best known approximation guarantee for DSA is 6 [8][4]. In this section, we will develop a number of very simple approximation results that do not necessarily improve on the bound of 6 for all cases, but do so for certain, data-dependent cases.

```
Procedure FirstFitStart(instance I)

L ← array of start and stop times from I
// in the sorting below, if two elements of L have the same value, and
// one is a stop time and one is a start time, then the stop time comes
// first in the sorted order
L ← sort(L)

availableLocations ← { }
maxLoc ← 1

// for an element e of L, interval(e) is the interval in I with
// start time or stop time given by e
foreach element e of L
      if (e is a start time)
            if (availableLocations ≠ ∅)
                  loc ← pop an element from availableLocations
            else
                  loc ← maxLoc
                  maxLoc ← maxLoc + 1
            fi
            assign loc to interval(e)
      fi
      if (e is a stop time)
            add location assigned to interval(e) to availableLocations
      fi
end for
```

Fig 2. First fit by start times on instances where all widths are the same.

3.1 Separate color (SC)

The simplest approximation algorithm is to sort the intervals by width into M sets, where M is the number of distinct widths in the instance. We then apply FF by start times on each of these M instances. Denote this algorithm as separate color (SC).

Theorem 5: SC has a performance guarantee of M.

Proof: Let the M instances be denoted $B_1, \ldots, B_M$. Clearly, we have $\tilde{\omega}(G_{B_i}) \le \tilde{\omega}(G_B)$ for each i. The total amount of memory used by SC is $\tilde{\omega}(G_{B_1}) + \ldots + \tilde{\omega}(G_{B_M})$. Hence, we get the guarantee of M.

If there are fewer than 6 distinct widths, then this bound beats the general bound!

3.2 Separate color with rounding (SC-R)

Since SC would have a bound of 2 if there are only 2 distinct widths, we can ask what happens if we “round” an instance with M distinct widths to 2 widths. Clearly, the rounded instance with 2 widths will have an MCW bigger than the original instance. Denoting the MCW of the original, and the MCW of the rounded instance as c_1, c_2 respectively, the rounding will pay off if $2c_2 < Mc_1$.

Suppose the instance is sorted by widths $S_N \ge S_{N-1} \ge \ldots \ge S_1$. Define

$$R_2 = MIN_{S_i}\left(MAX\left(\frac{S_i}{S_1}, \frac{S_N}{S_{i+1}}\right)\right), i = 1, \ldots, N \qquad \textbf{(EQ 1)}$$

Theorem 6: SC-R has a performance guarantee given by $2R_2$.

Proof: Let the distinct widths S_i be in sorted order: $S_N > S_{N-1} > \ldots > S_1$. Let S_a be the S_i that minimizes the maximum in the expression for R_2 above. The algorithm SC-R then rounds the widths to S_a and S_N; these are then allocated using separate coloring (SC). Let the MCW for the instance be $OPT = C_{max} = p_1 S_1 + \ldots + p_N S_N$, for some $0 \le p_i \le n_i$, where n_i is the number of intervals having width S_i. After the rounding, we have $OPT' \ge C'_{max} = (p_1 + \ldots + p_a)S_a + (p_{a+1} + \ldots + p_m)S_m$. This may not be the maximum clique weight anymore in the rounded instance; there could exist some other clique of greater weight. Let this clique have value given as

$$\left(\sum_{i=1}^{a} r_i\right)S_a + \left(\sum_{i=a+1}^{N} r_i\right)S_N > \left(\sum_{i=1}^{a} p_i\right)S_a + \left(\sum_{i=a+1}^{N} p_i\right)S_N .$$

Notice that we have represented the weights that multiply S_a and S_N has partitions into a and $N-a$ non-negative integers respectively. However, these integers r_i must satisfy

$$\sum_{i=1}^{N} r_i S_i \leq \sum_{i=1}^{N} p_i S_i$$

since the sum on the right is the maximum clique weight, and the sum on the left represents a clique weight. (Note that the rounding does not change the topology of the interval graph; a clique in the rounded graph is also a clique in the original graph and vice versa. Hence the partition r_i arise naturally when translated back to the corresponding clique in the original graph.) So the question is, how much bigger can these r_i be? Specifically, we want to solve

$$\begin{gathered} max\left\{\left(\sum_{i=1}^{a} r_i\right)S_a + \left(\sum_{i=a+1}^{N} r_i\right)S_N\right\} \\ \text{subject to} \\ \sum_{i=1}^{N} r_i S_i \leq \sum_{i=1}^{N} p_i S_i \end{gathered} \tag{EQ 2}$$

First we prove a lemma; this will be used to prove the theorem.

Lemma 1: There is a solution r'_i to the above with $r_j' = 0 \qquad \forall j \notin \{1, a+1\}$.

Proof: Suppose that r_i is any solution. Let $r_j > 0$ for some $j \neq 1$, $j \neq a+1$. Suppose $j \in \{2, \ldots, a\}$. Consider the new solution

$$r'_1 = r_1 + \left\lfloor \frac{r_j S_j}{S_1} \right\rfloor, r'_j = 0, r'_k = r_k \qquad \forall k \neq 1, k \neq j$$

It satisfies the constraint:

$$\begin{aligned} \sum_{i=1}^{N} r'_i S_i &= r_1 S_1 + \left\lfloor \frac{r_j S_j}{S_1} \right\rfloor S_1 + \sum_{k \neq 1, k \neq j} r_k S_k \\ &\leq r_1 S_1 + r_j S_j + \sum_{k \neq 1, k \neq j} r_k S_k \\ &= \sum_{i=1}^{N} r_i S_i \leq \sum_{i=1}^{N} p_i S_i \end{aligned}$$

and satisfies

$$\sum_{i=1}^{a} r'_i = r_1 + \left\lfloor \frac{r_j S_j}{S_1} \right\rfloor + \sum_{k \neq 1, k \neq j}^{a} r_k$$
$$\geq \sum_{i=1}^{a} r_i$$

since $S_j / S_1 > 1$. Since the solution r_i maximized the objective function, the new solution r'_i cannot give a bigger objective function; hence, there is equality in the equation above:

$$\sum_{i=1}^{a} r'_i = \sum_{i=1}^{a} r_i .$$

However, as shown, $r'_j = 0$ in the new solution.

We can continue in this manner, each time picking a non-zero component and constructing a new solution that zeros that component out. The process ends when $r'_j = 0$ for all $1 < j < a + 1$. Similarly, to zero out the rest of the r_i, we can pick an $r_j > 0$ for some $j \in \{a + 2, \ldots, N\}$, and construct the new solution:

$$r'_{a+1} = r_{a+1} + \left\lfloor \frac{r_j S_j}{S_{a+1}} \right\rfloor, r'_j = 0, r'_k = r_k \qquad \forall k \neq a+1, k \neq j.$$

Identical arguments show that this new solution satisfies the constraint and also maximizes the objective function. **QED**.

Hence, we can assume that the maximal solution has $r_j = 0$ for all $j \neq 1, j \neq a + 1$. Equation 2 becomes:

$$max\{r_1 S_a + r_{a+1} S_m\}$$

subject to **(EQ 3)**

$$r_1 S_1 + r_{a+1} S_{a+1} \leq \sum_{i=1}^{N} p_i S_i$$

Let us consider the linear relaxation of the above integer program. It's clear that the maximal solution will meet the constraint through equality since we could increase both the objective function and the function in the constraint were it not the case. Let

$$K = \sum_{i=1}^{N} p_i S_i .$$

The solution to the linear relaxation of equation 3 is the pair

$$\left(r_1, \frac{K - r_1 S_1}{S_{a+1}}\right).$$

The objective function becomes

$$f(r_1) = r_1\left(S_a - \frac{S_1 S_m}{S_{a+1}}\right) + K\frac{S_m}{S_{a+1}}.$$

Letting

$$P = S_a - \frac{S_1 S_m}{S_{a+1}},$$

we see that if $P > 0$, then the function $f(r_1)$ will be maximized when r_1 is maximized. r_1 is maximized if $r_{a+1} = 0$, making $r_1 = K/S_1$. Hence the maximum value of $f(r_1)$ is

$$K\frac{S_a}{S_1}.$$

If $P = 0$, then $f(r_1) = K\frac{S_m}{S_{a+1}}$ is the maximum value.

Finally, if $P < 0$, then the maximum value of $f(r_1)$ occurs at $r_1 = 0$, and the value is $f(r_1) = K\frac{S_m}{S_{a+1}}$. In summary, we can state that

$$f(r_1) \le K \cdot max\left\{\frac{S_a}{S_1}, \frac{S_m}{S_{a+1}}\right\} \quad \textbf{(EQ 4)}$$

It is clear that introducing the integer constraint will only make the integer solution pair less than or equal to

$$\left(r_1, \frac{K - r_1 S_1}{S_{a+1}}\right)$$

since all feasible points have to lie inside the polytope formed by the line $r_1 S_1 + r_{a+1} S_{a+1} \le K$, and the axes, in the first quadrant of $\Re^2$. Hence, the maximum value of the objective function will be less than or equal to $f(r_1)$.

The theorem is proven now by observing that

$$\frac{OPT'}{OPT} = \frac{r_1 S_a + r_{a+1} S_m}{K} \le \frac{f(r_1)}{K} \le max\left\{\frac{S_a}{S_1}, \frac{S_m}{S_{a+1}}\right\}.$$

QED.

In fact, we could even round to one width, S_N, and allocate that optimally. Define

$$R_1 = \frac{S_N}{S_1}. \quad \textbf{(EQ 5)}$$

Theorem 7: Rounding to one width, and allocating that instance optimally gives a performance guarantee of R_1.

Proof: The integer program that we need to solve becomes:

$$max\left\{\left(\sum_{i=1}^{N} r_i\right)S_N\right\}$$

subject to **(EQ 6)**

$$\sum_{i=1}^{N} r_i S_i \leq \sum_{i=1}^{N} p_i S_i$$

By lemma 1, we can again assume that there is a maximal solution to equation 6 with $r_j = 0$ for all $j > 1$. Hence, the solution is simply

$$r_1 = \left\lfloor \frac{\sum_{i=1}^{N} p_i S_i}{S_1} \right\rfloor, r_j = 0, j > 1.$$

So

$$\frac{OPT'}{OPT} = \left(\left\lfloor \frac{\sum_{i=1}^{N} p_i S_i}{S_1} \right\rfloor S_N\right) \Big/ \left(\sum_{i=1}^{N} p_i S_i\right) \leq \frac{S_N}{S_1}.$$

QED.

If we redefine SC-R to be the algorithm that tries both types of rounding, and also the separate color, and takes the best, our approximation guarantee becomes

$$MIN(M, R_1, 2R_2).$$

Example 1: Suppose we have an instance with 4 distinct widths: 2,3,6,9. This example comes from a non-uniform filterbank implementation in SDF [6]. For this example, clearly $R_2 = 3/2$, and the guarantee becomes

$$MIN(4, 4.5, 3) = 3.$$

Note that this guarantee holds for all instances that have these widths, regardless of starting and ending times.

3.3 Sorted width coloring (SWC)

In this algorithm, following [4], we first round each width to the nearest power of two, and then sort in order of decreasing width. This is done to make the sorted width sequence divisible; i.e, each width will be divisible by widths smaller than it. If the width sequence is divisible to begin with, the rounding will not be needed. Note that the rounding procedure will increase the size of the MCW by two times at most. Denote the distinct widths by $S_N \geq S_{N-1} \geq \ldots \geq S_1$.

The algorithm is to simply apply FF on this enumerated instance. The advantage of the rounding is the following observation:

Observation 1: If an interval of width S_i, with start and end times b, e, is allocated by FF at location L, it must be the case that for every location $\{0, \ldots, L-1\}$, there is some interval I', with start and end times b', e', occupying that location with $[b, e] \cap [b', e'] \neq \varnothing$. Basically, this means that there cannot be any location interval $[l_1, l_2] \subseteq [0, L-1]$ that is free in the time interval $[b, e]$.

Proof: (sketch) For contradiction, suppose there is such a location interval $[l_1, l_2] \subseteq [0, L-1]$. When widths of type S are allocated, any fragmentation that results must have location widths that are multiples of S. When an interval of width S_i is allocated, intervals that have already been allocated are either of width S_i, or of width $S_j, j > i$. Since $S_i | S_j, j > i$, any fragmentation that had occurred would have a location width that would be a multiple of S_i, meaning S_i could have been allocated there. Since FF chooses the smallest feasible location, this contradicts the fact that location L was chosen instead.

For each width S_i, let $I(S_i)$ be the set of intervals having width S_i. Also, for an interval I, let $s(I), e(I)$ denote the start and end times respectively. Now suppose that this enumerated sequence also satisfies the following condition: for any interval $I \in I(S_i)$, and any interval $J \in I(S_j), j \geq i$, either $s(I) \in I \cap J$ or $e(I) \in I \cap J$. In other words, J cannot be a proper subset of I. An enumerated instance that satisfies this condition is called **subset-free**.

Theorem 8: For subset-free enumerated instances, FF has a performance guarantee of 4 (2 if the sorted width sequence was divisible to begin with—i.e, did not require rounding to powers of 2).

Proof: Consider an interval I that was allocated at location $A(I)$. By observation 1, we have that there is some interval that is alive in $[b(I), e(I)]$ at every location in $[0, A(I)-1]$. Since the enumerated instance is subset-free, these intervals in $[0, A(I)-1]$ overlap with I either at $b(I)$, or overlap at $e(I)$. The sum of the widths of the intervals that overlap with I at $b(I)$ or $e(I)$ has to be less than the MCW. Hence, we must have

$$A(I) + w(I) \le 2\tilde{\omega}(G_B).$$

Since this is true for every interval I, the overall guarantee is 2. If the rounding to powers of 2 was done to make the sequence divisible, then the guarantee becomes 4.

3.4 The case where widths are 1 and X.

Finally, we consider instances where the widths have value 1 or X, ordered by starting times. For these, FF has the following guarantee:

Theorem 9: For instances with widths of 1 or X, ordered by starting times, FF has a guarantee of

$$2 - \frac{1}{X}$$

Proof: (Sketch) Consider the instance shown in figure 3. After a little thought, it is evident that this is the sort of instance that will cause FF to behave in the worst possible way. The long instances of 1 originate from the "OPT" region; OPT is not one big single interval; it is simply the point at which the maximum clique weight is. The idea is that in the worst possible case, we could pick a point where the allocation is fragmented, and keep going up until we reach a point where there is no fragmentation. In the worst case, the point of no fragmentation could be as wide as the maximum clique weight. Now suppose that there are n blocks of width X that are placed as shown by FF. They are placed that way presumably because a) the memory under the OPT region gets fragmented, and the worst possible way for this to occur would be for there to be $(OPT)/X$ blocks of width 1, spaced $X-1$ apart, forcing any block of X that arrives later to be placed outside the OPT region, and b) because the region after OPT till the point of placement is also filled. At the point where the last block of X begins, there must be blocks filling up the nX region completely as otherwise the last block would not have been placed where it is. The space outside of OPT can-

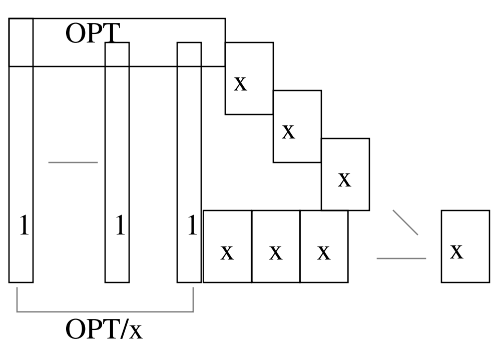

Fig 3. A worst case instance for FF with only two sizes, 1 and X.

not contain any blocks of width 1 since these can be placed under the OPT region; it can only contain blocks of width X. Hence, we have the following inequality at the point where the last block of width X begins:

$$\frac{OPT}{X} + nX \leq OPT$$

since OPT is the maximum clique weight, and the clique weight anywhere else cannot be greater. Since $OPT + nX$ is the allocation achieved by FF, it follows that the approximation bound is

$$\frac{OPT + nX}{OPT} \leq 2 - \frac{1}{X}.$$

QED.

So if $X = 2$ (recall that even the case where the widths are 1 or 2 is NP-complete), then the guarantee is 1.5, and this bound is tight.

It is interesting to note that the two-size case comes up frequently in many DSP implementations of image processing systems where there might be a basic vector or matrix data type, and a basic number data type.

3.5 Worst case behavior

The interesting question of a guarantee, if any, of FF on enumerated instances ordered by starting times is not known. We can give the following example to show that the guarantee cannot be better than 2.5; i.e, a lower bound on the guarantee. It would also be interesting if this lower bound could be improved

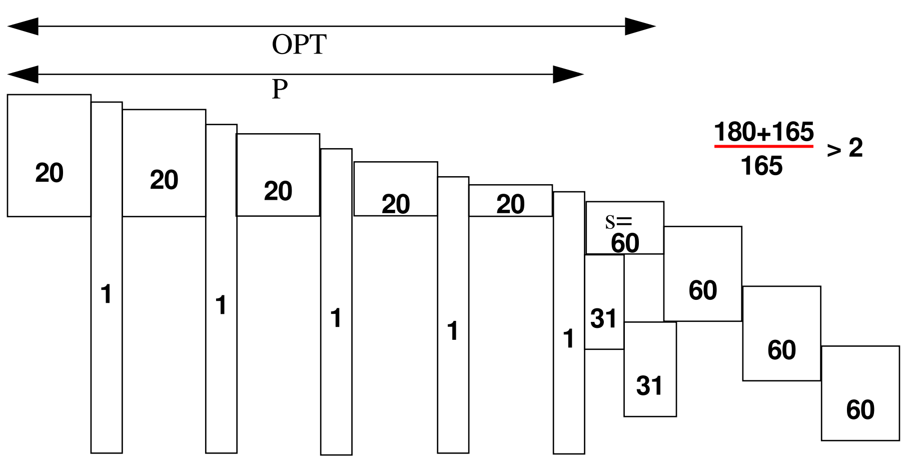

Fig 4. Example to illustrate that FF can have performance that is more than twice as bad as the optimum value.

(presumably by constructing an example that causes FF to achieve an allocation of worse ratio, say 3, than the one in this example). Note that the example below can be scaled to achieve a ratio of at most 2.5. By scaling, we mean replacing the width of size 20 by $20x$, or giving a different value for s (in this example, it has a value of 60) and so on. This is because the inequality $P + s \geq 2s$ has to hold, since we are assuming that the width indicated as OPT is the point of maximum overlap. Since $OPT = P + s$, we have the inequality. So $P \geq s$, and the performance ratio is given by $\frac{P + s + 3s}{P + s} \leq \frac{5}{2}$.

3.6 Related work

As already mentioned, the best known approximation results for DSA in general have guarantees of 6 [8][4]. There appears to be a close relationship between the DSA problem, and the problem of devising online coloring algorithms for enumerated interval graphs where all intervals have the same width. The latter problem can be solved optimally, as shown in section 2, if the intervals are ordered (enumerated) by start times. However, an interesting question is how well an online algorithm like FF can do if the intervals are ordered arbitrarily. Defining $A(B)$ to be the number of colors used by an algorithm A to color the (non-weighted) instance B, it has been shown by Kierstead and Slusarek that $4.45 \leq R_{FF} \leq 32$, where $R_{FF} = sup_B\{FF(B)/\omega(G_B)\}$. In [4], Kierstead gives another online algorithm called OIC that has a performance guarantee of $3\omega(G_B) - 2$. Slusarek also develops another online algorithm called SCC in [8] that gives the same performance guarantee of $3\omega(G_B) - 2$. Again, these bounds are for coloring interval graphs in an online manner, where the intervals all have the same width, and are ordered arbitrarily.

The approximation algorithm of [4] for DSA is based on a simultaneous recursive formulation that actually colors an exponentially bigger graph F than the weighted interval graph (enumerated by decreasing width); the interval graph F is obtained simply by considering an interval v of width $w(v)$ as $w(v)$ intervals of width 1. Kierstead uses OIC to color F and obtains an approximation guarantee of 6. While this formulation does not give a polynomial time algorithm, Kierstead states that it can be made to run in polynomial time with a careful implementation. Slusarek uses a similar approach, basing his algorithm on SCC.

The question of how bad FF is for DSA if particular orderings are considered, like ordering by widths, or arrival times seems to be open. Another interesting open issue is whether non-online algorithms can do better than online algorithms like FF, OIC, and SCC. As our experimental results below show, FF performs very well on average, so it would be difficult to beat it in practice.

4 Experimental results

We tested FF on random instances, ordered by arrival times, durations, and widths. Random instances were created using three parameters: number of intervals (N), maximum stop time of any interval (K), and maximum width of any interval (W). Given these parameters, for each interval I, the stop time $e(I)$ is drawn uniformly from $[1, K]$, the integer start time $b(I)$ is drawn uniformly from $[1, e(I) - 1]$, and the width $w(I)$ is drawn uniformly from $[1, W]$. Three types of orderings were tested: sorting by starting times (FFA), sorting by durations (FFD), and sorting by widths (FFW). The amount of memory required was compared to the MCW in each case. We denote a particular combination of parameters by the tuple (N, K, W). For each combination, the statistics were collected for 500 random instances with those parameters. The table below shows average and maximum ratios to the MCW, and also with each other, for several combinations of parameters. As can be seen, FF performs very well in practice, and it is never off by more than 70% from the optimum. Sorting by durations gives the best results, compared

Table 1. FirstFit behavior on random interval instances.

	(20,20, 20)	(40,40, 40)	(60,60, 60)	(20,40, 60)	(40,80, 120)	(20,100 ,500)	(80,500 ,1000)	(50,25, 100)
FFA/MCW: MAX	1.49	1.37	1.32	1.43	1.37	1.49	1.34	1.34
FFA/MCW: AVG	1.10	1.11	1.12	1.11	1.12	1.12	1.12	1.11
FFD/MCW: MAX	1.31	1.30	1.21	1.31	1.37	1.39	1.22	1.25
FFD/MCW: AVG	1.05	1.06	1.07	1.06	1.07	1.06	1.07	1.06
FFW/MCW: MAX	1.50	1.45		1.68		1.79		
FFW/MCW: AVG	1.15	1.18		1.15		1.15		
FFA/FFD: MAX	1.34	1.29	1.23	1.32	1.31	1.34	1.17	1.23
FFA/FFD: MIN	0.85	0.90	0.91	0.84	0.89	0.80	0.91	0.91
FFA/FFD: AVG	1.05	1.05	1.05	1.05	1.05	1.05	1.05	1.05

to sorting by starting times, and widths. Sorting by widths gives the worst results. From these results, it is clear that first fit on instances sorted by duration times gives allocations that are within 7% of the optimum value. In fact, this is a pessimistic estimate since the CN can be greater than the MCW, and the (unknown) CN is the optimum value.

5 Conclusion

We have presented a number of simple approximation algorithms for the NP-complete problem of dynamic storage allocation. The best of these results achieves a bound of 4 for a subclass of DSA instances called subset-free instances. We have shown that in practice, the firstfit heuristic gives allocations that are within 7% of the optimum on average.

6 References

[1] S. S. Bhattacharyya, P. K. Murthy, E. A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, 1996.

[2] M. R. Garey, D. S. Johnson, *Computers and Intractability-A guide to the theory of NP-completeness*, Freeman, 1979.

[3] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, 1980.

[4] H. A. Kierstead, "Polynomial Time Approximation Algorithm for Dynamic Storage Allocation," *Discrete Mathematics*, Vol. 88, pp231-237, 1991.

[5] E. A. Lee, D. G. Messerschmitt, "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing," *IEEE Trans. on Computers*, Feb., 1987.

[6] P. K. Murthy, S. S. Bhattacharyya, E. A. Lee, "Joint Code and Data Minimization for Synchronous Dataflow Graphs," *Journal on Formal methods in System Design*, Vol. 11, No. 1, pp. 41-70, July 1997.

[7] P. K. Murthy, and S. S. Bhattacharyya, "Shared Memory Implementations of Synchronous Dataflow Specifications using Lifetime Analysis Techniques," Tech. Rpt. UMIACS-TR-99-32, University of Maryland Institute for Advanced Computer Studies, College Park, MD 20742, http://www.cs.umd.edu/TRs/TRumiacs.html, 1999.

[8] M. Slusarek, "A Coloring Algorithm for Interval Graphs," Proceedings of the 14th International Symposium on Mathematical Foundations of Computer Science, LNCS 379, pp471-480, 1989.