ELSEVIER

Discrete Applied Mathematics 59 (1995) 225–235

DISCRETE APPLIED MATHEMATICS

On the k-coloring of intervals

Martin C. Carlisle[a], Errol L. Lloyd[b,*,1]

[a] *Computer Science Department, Princeton University, Princeton, NJ 08544, USA*
[b] *Computer and Information Sciences, University of Delaware, 103 Smith Hall, Newark, DE 19716-2586, USA*

Received 20 December 1991; revised 4 October 1993

Abstract

The problem of coloring a set of n intervals (from the real line) with a set of k colors is studied. In such a coloring, two intersecting intervals must receive distinct colors. Our main result is an $O(k + n)$ algorithm for k-coloring a maximum cardinality subset of the intervals, assuming that the endpoints of the intervals are presorted. Previous methods are linear only in n, and assume that k is a fixed constant. In addition to the main result, we provide an $O(kS(n))$ algorithm for k-coloring a set of *weighted* intervals of maximum total weight. Here, $S(n)$ is the running time of any algorithm for finding shortest paths in graphs with $O(n)$ edges. The best previous algorithm for this problem required time $O(nS(n))$. Since in most applications, k is substantially smaller than n, the saving is significant.

1. Introduction

Over the past decade or so, a great deal of research attention has been paid to the tractability of various algorithmic problems concerning graphs. A particularly popular topic has been graph coloring, where two related problems have been considered: (1) Given a graph G, what is the chromatic number of G? (or, stated as a decision problem: Given k colors, is G k-colorable?); and (2) Given a graph G and k colors, find a maximum k-colorable subgraph of G. Unfortunately, these problems are NP-complete [7] for arbitrary graphs. Thus, attention has turned to identifying classes of graphs for which these problems are solvable in polynomial time. *Interval graphs* are one such class [8]. These are graphs for which there exists a set of n finite open intervals on the real line, such that the following two conditions hold:

(1) there is a 1 : 1 correspondence between the intervals and the vertices of the graph, and
(2) two intervals intersect if and only if there is an edge between the vertices corresponding to those two intervals.

*Corresponding author.

[1] Partially supported by the National Science Foundation under grant CCR-9120731. E-mail: elloyd@cis.udel.edu.

0166-218X/95/$09.50
SSDI 0166-218X(93)E0174-W

In this context, there are $O(n + e)$ time algorithms (n and e are the number of vertices and edges, respectively, of G) both for determining the chromatic number of an interval graph, and for the maximum k-coloring of an interval graph [12]. These algorithms depend on the fact that interval graphs are *perfect*, and that, as a consequence, the size of a largest clique is equal to the number of colors in a minimum coloring. Thus, a subgraph of an interval graph is k-colorable if and only if it contains no clique of size greater than k.

While the existing algorithms (mentioned above) are asymptotically optimal for interval graphs as such, they may be quite unsatisfactory for applications involving "interval graphs". In most of these applications (some are discussed in Section 4), the problem is not presented in the form of an interval graph, but simply as the underlying set of n intervals. In this context, the actual problem is to determine the chromatic number of, or a maximum k-coloring of, the set of intervals. In such colorings, intersecting intervals must receive distinct colors, and a set of intervals is k-colorable if and only if not more than k of those intervals intersect at any single point. In this context, there is often no need to explicitly construct the *interval graph* associated with the set of intervals. This is important since the number of edges in the interval graph may be quadratic in the number of intervals (consider n mutually intersecting intervals). It is well known, for example, that the chromatic number of a set of sorted intervals (i.e. the interval endpoints are sorted) can be determined directly in time $O(n)$, without making an explicit conversion to the interval graph representation.

In the next section, we give an $O(k + n)$ algorithm for the maximum k-coloring of a set of n sorted intervals (the $2n$ endpoints of the intervals are sorted). Previous algorithms for finding maximum k-colorings considered only the case where the problem is presented using an interval graph. In that context, and assuming that k is *constant*, an $O(n + e)$ algorithm was given in [12]. A natural extension of the ideas used there to our situation yields an algorithm running in time $O(n \log k)$. Thus, our algorithm (recall, its running time is $O(k + n)$) is an improvement over that extension of [12] by a factor of $\log k$. In Section 3, we consider a weighted version of the problem. Here, a positive *weight* is associated with each interval, and the goal is to find a k-colorable set of intervals of maximum total weight. We give an $O(kS(n))$ algorithm for finding such a coloring. Here, $S(n)$ is the running time of any algorithm for finding shortest paths in graphs having $O(n)$ edges. Previous results include an integer programming based solution [12], and an $O(nS(n))$ time algorithm [1]. In Section 4 we discuss several applications involving k-colorings of interval graphs. Finally, in Section 5, we present some conclusions and discussion.

2. A linear time algorithm for maximum (cardinality) k-coloring

In this section we present an $O(k + n)$ algorithm for the maximum k-coloring of a set of n intervals. As noted above, we assume that the $2n$ endpoints of the intervals have been sorted. Thus, we denote the intervals by $I_1, I_2, \ldots, I_n$, and assume that the

intervals are ordered by increasing value of their right endpoint. For interval I_i, we refer to i as the *index* of the interval. Since the ordering of indices corresponds to the ordering by right endpoints, we will often find it convenient to refer to indices in discussing the left to right ordering of intervals by right endpoint.

As noted in the introduction, the algorithm that we give here, and in the next section, will not explicitly construct the associated interval graph. Nonetheless, in explaining the motivation for, and the correctness of, these methods, we will sometimes find it convenient to refer to that associated interval graph. This should present no problem, since the relationship between the intervals and the interval graph is clear.

2.1. The algorithm

The intuitive idea behind our algorithm for maximum k-coloring is to use a greedy approach. Intervals are considered for coloring (*processed*) in order of increasing right endpoint. As the algorithm proceeds, it maintains a k-colorable subgraph of G. This subgraph is maximal relative to the intervals processed to that point. Each interval I_i when processed, is either:

(1) *discarded*, if there is no available color (that is, the inclusion of that interval in the colored subgraph would produce a $(k+1)$-clique), or
(2) is assigned a color using a "best fit" principle.

To decide which color constitutes the "best fit" (if any), we consider for each color, the interval of largest index that has already been assigned that color. Such an interval is the *leader* for that color. The right endpoint of this interval is, in a certain sense, the right endpoint of that color. The interval being processed, I_i, is assigned the color of the *best fit leader*. That is, the leader with greatest right endpoint no greater than the left endpoint of I_i. Note however that there is some difficulty in finding this best fit leader, since examining all of the colors would take time $O(\log k)$ per interval, assuming the use of a heap or similar structure to store the leaders for each color, whereas, in an amortized sense, our algorithm requires the use of time $O(1)$ per interval. Thus, we use the following method to determine the best fit leader without explicitly considering all k colors.

We begin by defining adjacent(I_i) to be the interval of greatest index that lies strictly to the left of I_i. That is, the right endpoint of adjacent(I_i) is no greater than the left endpoint of I_i. Clearly, as the algorithm executes, the best fit leader for I_i will be adjacent(I_i), if adjacent(I_i) is in fact a leader. In such a case, I_i will be assigned the same color as adjacent(I_i). But what if adjacent(I_i) is either not colored, or colored, but not a leader? In this case, how can we locate the best fit leader? We find this interval, or determine that there is no such interval, using the well-known *union/find* operations on disjoint sets [10].

Here, each interval is initially in a singleton set. Always, the *name of the set* is the interval in the set of least index (more to the point, this is also the interval in this set having the smallest right endpoint, and further, as we will show, is a leader). As the algorithm proceeds, these sets will of course grow, and will represent equivalence

classes of intervals with respect to "best fits" (note that these are NOT sets of intervals of a given color). In particular, if an interval, I_j, is discarded, or is colored, but not a leader (an interval with right endpoint larger than that of I_j is assigned the same color as I_j), then the set containing I_j is unioned with the set containing the interval, I_{j-1}. This is done to indicate that no interval I_i having adjacent$(I_i) = I_j$ can be assigned the same color as I_j, and that the name of the set containing I_j is the new "best fit" for I_i. Thus, at any point in the algorithm, a set, except for the name, consists of discarded intervals, and intervals no longer at the right endpoint of the color to which they are assigned. The name of the set is either an interval that is a leader, or an interval that has not yet been processed (the latter is true only if the set is a singleton).

The complete algorithm for the maximum k-coloring of a set of intervals is described in Fig. 1.

Algorithm 1. Maximum cardinality k-coloring of a set of intervals

Input: An integer k, and a set of n intervals $I_1, I_2, \ldots, I_n$, sorted by right and left endpoints. The intervals are indexed in order of increasing right endpoint, and it is assumed that all endpoints are positive integers.

Output: A k-coloring of the intervals that maximizes the number of colored intervals. Each interval is assigned a value from 0 to k, where $1, 2, \ldots, k$ represent colors, and 0 represents no coloring.

Definitions:

left(I_i) and right(I_i): the left and right endpoints, respectively, of I_i.

color(I_i): the color assigned to I_i. If I_i is not colored, then this is 0.

$I_{-k}, I_{-k+1}, \ldots, I_0$: these are dummy intervals that serve as the initial right endpoints of the colors (including the 0 "no color" color).

adjacent(I_i): the interval whose right endpoint is the largest less than left(I_i).

find(I_i): returns the name of the set containing I_i.

union(I_i, I_j): merges the sets with names I_i and I_j. The name of the combined set will be I_j. The original sets no longer exist after this operation.

Method:

```
for i ← 0 to k do                          (* setup & color the dummy intervals *)
  right(I_-i) ← -i;
  left(I_-i) ← -i - 1;
  color(I_-i) ← k - i;
  create a singleton set containing I_-i;
for i ← 1 to n do                          (* setup adjacent, and the sets *)
  adjacent(I_i) ← max{j: right(I_j) ≤ left(I_i)};
  create a singleton set containing I_i;   (* we let I_i also be the name of this set *)
for i ← 1 to n do                          (* the main loop *)
  I_j ← find(adjacent(I_i));               (* this is I_i's "best fit" *)
  if j = -k
    then
      color(I_i) ← 0;                      (* do not color I_i *)
      union(I_i, find(I_i-1));             (* I_i denotes the singleton set containing I_i *)
    else
      color(I_i) ← color(I_j);
      union(I_j, find(I_j-1));
```

Fig. 1. An algorithm for maximum k-coloring.

2.2. The proofs of correctness and running time

In this section we prove the correctness of the algorithm and establish that it can be implemented in linear time.

We begin by defining, for each processed interval I_i, the *r-closest leader* to I_i to be I_j, the leader of greatest index, with $j \leqslant i$ (equivalently, $\text{right}(I_j) \leqslant \text{right}(I_i)$). Since the r-closest leader to I_i may change as the algorithm proceeds, we have the following claim.

Lemma 1. *As the algorithm proceeds, for each processed interval I_i, the name of the set containing I_i is the r-closest leader to I_i (at that point in the algorithm).*

Proof. Since the lemma is trivially true prior to the execution of the main loop, we proceed inductively to the end of the ith iteration of the main loop. We consider how the sets may have changed since the end of the $(i-1)$st iteration. There are two possibilities:

Case 1: *I_i was discarded (assigned "color" 0).* In this case, the only change to the sets in the ith iteration is that I_i is included in the set containing I_{i-1}. By the induction hypothesis, after the $(i-1)$st iteration, the name of that set is some I_h, the r-closest leader to I_{i-1}. After the ith iteration, I_i has been added to that set, and I_h is still the name of that set. Further, I_h is still a leader, and I_i is not a leader, since it did not get a real color. Thus, it follows from I_h being the r-closest leader to I_{i-1} that I_h is also the r-closest leader to I_i.

Case 2: $\text{color}(I_i)$ *was assigned* $\text{color}(I_j)$. In this case, the only change to the sets is that I_j is included in the set with I_{j-1}. Let I_h be the name of the set containing I_{j-1} after the $(i-1)$st iteration. By the induction hypothesis, I_h is the r-closest leader to I_{j-1}. After the ith iteration, I_j is in that set, and I_h is still the name of that set. Further, I_j is not a leader since I_i is colored the same as I_j and I_j lies strictly to the left of I_i. Also, since $h \neq j$, and the only coloring change was to let $\text{color}(I_i) = \text{color}(I_j)$, it follows that I_h is still a leader. Thus, it follows from I_h being the r-closest leader to I_{j-1}, that I_h is the r-closest leader to I_j. □

Lemma 2. *Interval I_j as found in the ith iteration of the algorithm is the best fit leader for I_i.*

Proof. By definition, $\text{adjacent}(I_i)$ is the interval of largest index strictly to the left of I_i. By Lemma 1, I_j as found in the ith iteration of the algorithm, is the r-closest leader to $\text{adjacent}(I_i)$. That is, I_j is the leader with greatest right endpoint no greater than $\text{right}(\text{adjacent}(I_i))$. It follows that I_j is also the best fit leader for I_i. □

Lemma 3. *As the algorithm proceeds, if interval I_i intersects with g leaders, then the coloring of I_i produces a $(g+1)$-clique of colored intervals.*

Proof. Of the g leaders that intersect with I_i, let I_L be the leader of least index. We claim that $g-1$ other colored intervals (one for each of the other $g-1$ leaders), intersect at p, the right endpoint of I_L. Since I_i also intersects with I_L at p, the lemma will follow once we establish this claim. Thus, assume the claim is false and consider one of the other $g-1$ leaders, say of color 1, such that no interval of color 1 intersects with I_L at p. Let I_b and I_a be the intervals of color 1 such that $\text{right}(I_b) < p$ and $\text{left}(I_a) > p$, and there is no interval of color 1 that lies between I_b and I_a. Now, note that when I_a was processed, I_b was the leader of color 1, and by Lemma 2, was the best fit leader for I_a. But this is a contradiction, since I_L was also a leader when I_a was processed, and $\text{right}(I_L) < \text{left}(I_a)$, and $\text{right}(I_L) > \text{right}(I_b)$. Thus, I_L should have been the best fit leader for I_a. Thus, the claim and the lemma are established. □

Theorem 1. *Algorithm* 1 *creates a maximum k-coloring.*

Proof. We begin with some notation and a definition. First, for $i \leqslant j$, let $I_{i\to j}$ represent intervals $I_i, \ldots, I_j$, and let M be the set of all optimal colorings of $I_{1\to n}$. Also, for a coloring C in M, a *subcoloring of* C *on* $I_{1\to i}$ is a coloring of $I_{1\to i}$ such that every interval has the same color as in C, or has no color if it is not colored in C. We claim that for each i, the coloring C_i created after the ith iteration of the main loop in Algorithm 1 is a subcoloring of some element of M on $I_{1\to i}$. Once this is established, the theorem follows when $i = n$.

Thus, we inductively assume that C_{i-1} is a subcoloring of some coloring in M for $I_{1\to i-1}$, and consider C_i. There are three possibilities.

Case 1: I_i *is not colored in* C_i. From Lemma 2 it follows that the best fit leader for I_i is I_{-k}, and, since this interval has the least right endpoint among all of the leaders, it must be that all of the other leaders intersect with I_i. From Lemma 3, the coloring of I_i would form a $(k+1)$-clique of colored intervals. Thus, since C_{i-1} was a subcoloring of some C^* in M on $I_{1\to i-1}$ then C_i must also be a subcoloring of C^* on $I_{1\to i}$ since C^* cannot have colored I_i without also having created a $(k+1)$-clique of colored intervals.

Case 2: *There is a* C^* *in* M *such that* C_{i-1} *is a subcoloring of* C^* *on* $I_{1\to i-1}$ *and* I_i *is assigned different colors in* C^* *and* C_i. We will construct a C' in M such that C_i is a subcoloring of C' on $I_{1\to i}$. We begin by assuming that $\text{color}(I_i) = 2$ in C^* and $\text{color}(I_i) = 1$ in C_i. Let I_m be the leftmost interval (i.e. interval of least index) of color 1 in C^* that is not colored in C_{i-1} (note that $m > i$). If no such I_m exists, then C' is identical to C^* except that $\text{color}(I_i) = 1$. Otherwise, C' is identical to C^* except that: I_m, and all intervals of color 1 in C^* that follow I_m, are assigned color 2; and all intervals of color 2 in C^* having a right endpoint of $\text{right}(I_i)$ or greater are assigned color 1. This is possible as by the "best fit" rule, the leader of color 2 in C_{i-1} must have an index no greater than that of the leader of color 1, otherwise I_i would have been given color 2 by Algorithm 1. Since $\text{left}(I_m)$ is larger than the right endpoint of color 1 (at the start of the ith iteration of the main loop) I_m can be assigned color 2, which had a smaller or equal right endpoint. Thus, C_i is a subcoloring of C' on $I_{1\to i}$. Further

C' is an optimal coloring since it colors the same number of intervals as C^* which is in M.

Case 3: *There is a* C^* *in* M *such that* C_{i-1} *is a subcoloring of* C^* *on* $I_{1\to i-1}$, *and* I_i *is not colored in* C^*. Again we construct a C' in M such that C_i is a subcoloring of C' on $I_{1\to i}$. Without loss of generality, let color$(I_i) = 1$ in C_i. Now, let I_m be the leftmost interval colored 1 in C^* that is not in C_{i-1} (such an I_m must exist, otherwise we can add I_i to C^*, an optimal coloring – a contradiction). Note that as Algorithm 1 colors intervals in order of increasing right endpoint, I_m has a larger right endpoint than I_i. Thus C' is identical to C^*, except that I_m is not colored, and color$(I_i) = 1$, Thus C' is an optimal legal coloring, andC_i is a subcoloring of C' on $I_{1\to i}$.

Thus, Algorithm 1 produces an optimal coloring. □

Next we consider the running time of Algorithm 1.

Theorem 2. *The running time of algorithm* 1 *is* $O(k + n)$.

Proof. The algorithm consists of three loops, the first of which is trivially seen to require time $O(k)$. The second computes adjacent(I_i) for $1 \leqslant i \leqslant n$. Since the endpoints are sorted, this takes constant time per I_i, hence $O(n)$ time for the entire loop.

For the third loop (the "main" loop), the operations of the loop, excluding the union/find operations, clearly require but constant time per iteration. Since that loop is executed n times, the running time, except for union/find operations, is $O(n)$.

To implement the union/find operations in a total time of $O(n)$, we utilize a method [6] for performing union/find operations on sets where the possible unions are known in advance. Here, the possible unions form a union graph that has the individual (initial) sets as vertices, and directed edges between those sets that *may* be merged. It is shown in [6] that if this graph is in fact a *union tree*, then $O(n)$ union/find operations can be performed in $O(n)$ time in the worst case.

Note that in Algorithm 1, each set is merged only with the set containing the interval with the next smallest right endpoint. Therefore, the union graph is simply a path from the interval with the largest right endpoint to that with the smallest. Since this is clearly a tree, the result of [6] is applicable, and Algorithm 1 can be implemented in time $O(k + n)$. □

3. An improved algorithm for weighted k-coloring

We now turn to the problem of finding a k-coloring of maximum total weight. Recall that each interval has a positive weight associated with it, and that the goal is to find a k-coloring of the intervals of maximum total weight (that is, to maximize the sum of the weights of all of the colored intervals). Note that since interval graphs are perfect, such a k-coloring may be found by locating a maximum weight induced subgraph among all subgraphs having no clique of size greater than k. Using this

approach, Yannakakis and Gavril [12] gave a linear programming based solution to the problem we consider. Later, an $O(nS(n))$ algorithm that uses shortest paths to create a maximum weight k-colorable subgraph was given in [1]. In this section, we describe a solution that is based on a reduction to a certain network flow problem. As it turns out, a solution to that network flow problem depends on shortest paths. Our method has a worst-case running time of $O(kS(n))$, thereby improving on prior results by a factor of n/k.

Our approach is to construct a network for which a minimum cost flow of size k will provide a solution to weighted k-coloring of intervals. We begin with a brief description of that flow problem (see [11] for details).

In the *minimum cost flow of size k* problem, we are given an integer k and a directed graph (a *network*) $G = (V, E, \text{cost}, \text{cap})$, where V is a set of vertices, E is a set of directed edges, and associated with each edge e is a nonnegative *cost* $\text{cost}(e)$ and a positive *capacity* $\text{cap}(e)$. Also, there are designated vertices s and t. A *flow from s to t* is a mapping $f: E \to R$ such that the following conditions hold:

(1) for each vertex v, except s and t, the flow into v is equal to the flow out of v, and

(2) for each edge e, $0 \leqslant f(e) \leqslant \text{cap}(e)$.

The *size* of flow f is the net flow into t. The *cost* of flow f is $\sum_{e \in E} f(e) * \text{cost}(e)$. The objective in the minimum cost flow of size k problem is to find a flow of size k from s to t that is of minimum cost among all such flows. A polynomial time algorithm for finding such a flow is given in [3, 11]. It is also shown in [11] that:

> If G is acyclic, and has only integer capacities, then a minimum cost flow of size k can be found in time $O(e + kS(n))$, even if negative costs are permitted.

Note that if $e = O(n)$, then the running time of that algorithm is $O(kS(n))$. Note also that in the presence of integer capacities, the flow across each edge of G will also be integral.

So, consider a set of weighted intervals, $I_1, \ldots, I_n$. As before, we assume that the endpoints of the intervals are sorted. Let $x_1 < x_2 < \cdots < x_r$ be the unique set of values of right and left endpoints of these intervals. We construct a network G with nodes $s = v_0, \ldots, v_{r+1} = t$, and edges:

For $1 \leqslant i \leqslant r + 1$, a *clique-edge* (v_{i-1}, v_i) of cost zero, and capacity k.

For $1 \leqslant i \leqslant n$, an *interval-edge* (v_j, v_h) where $v_j = \text{left}(I_i)$ and $v_h = \text{right}(I_i)$. This edge has a cost equal to the negative of the weight of I_i, and has a capacity of 1.

Lemma 4. *The cost of a minimum cost flow of size k in G is equal to the negative of the weight of a maximum weight k-coloring of the n intervals.*

Proof. Consider a flow of size k and cost c in G. Since the flow across each edge is integral, we can assume that each unit of flow follows a single path. Thus, consider any single unit of flow and let p be the path followed by that unit. Now, since all of the edges in G are directed from vertices of lower index to vertices of higher index, it follows that the intervals corresponding to the interval-edges in p do not overlap. Thus, all of those

intervals may be assigned the same color in a k-coloring. Hence, corresponding to a flow of size k and cost c in G, there is a k-coloring of the n intervals that has weight $-c$. In a similar fashion, it can be shown that if there is a k-coloring of the n intervals of weight c, then there is a flow of size k in G of cost $-c$. The lemma follows from these two observations. □

Corollary 1. *Given a minimum cost flow of size k in G, the intervals corresponding to the interval-edges of flow* 1 *are exactly the intervals in some maximum weight k-coloring of the n intervals.*

Thus, our algorithm for finding a maximum weight k-coloring of a set of weighted intervals consists of first using the above method to locate the intervals in some maximum weight k-coloring (i.e. the intervals corresponding to the interval-edges of flow 1), and then actually coloring those intervals by using Algorithm 1. The correctness of the method follows from the earlier discussion as does the following theorem.

Theorem 3. *A maximum weight k-coloring of a set of n weighted intervals can be found in time* $O(kS(n))$.

4. Improved results for applications

In this section we briefly describe three applications involving sets of intervals. In each application, the problem translates directly to a problem of finding a maximum (weighted or unweighted) k-coloring of a set of intervals. Thus, the results given here apply directly, and provide solutions for these applications and that are more efficient than those previously known.

Job scheduling [1]: We are given a set of n *tasks* to be executed on a set of k identical *processors*. Associated with each task is a *start time* and an *end time*, indicating that *if* the task is executed, then it must start precisely at the start time, and complete precisely at the end time. Further, each task has an associated *value*. Since, due to their fixed start and end times, it may not be possible to schedule all n of the tasks, the goal is to schedule a set of tasks of maximum total value. It is easy to see that this is equivalent to finding a maximum weight k-coloring of a set of intervals with weights. This is the problem considered in [1], where an $O(nS(n))$ algorithm was given. Our results improves the time to $O(kS(n))$.

Routing of two point nets [9]: Here we are given a channel lying on a VLSI chip (say between two components on that chip). That channel consists of k horizontal tracks. We are also given a set of n two-point nets, where the terminals of the nets lie at fixed positions along the top or bottom sides of the channel. These nets are to be routed in the channel using two vertical wire segments and one horizontal wire segment, such that similarly oriented wires do not intersect. Since it is assumed that but one wire

connects to each terminal, this intersection condition is not a constraint on the vertical wires. Thus, we are only concerned with the routing of the horizontal segments, and the goal is to rout a maximum number of nets (given the fixed number of horizontal tracks). Since this is precisely the problem of maximum k-coloring a set of intervals, we provide an $O(k + n)$ solution.

Register allocation: We consider a problem of allocating a set of k registers to straight-line code within a single basic block (*local register allocation*). Here, each operand must reside in a register when the relevant operator is applied, and when values are assigned they are always stored through to main memory. Thus, the goal of local register allocation is to avoid the loading of values from main memory into registers (i.e. to minimize the number of loads). By carefully modeling variable uses as intervals, this problem becomes that of finding a maximum k-coloring of a set of intervals. Intuitively, each uncolored interval represents a load. Thus, we provide an $O(k + n)$ algorithm for load minimization in local register allocation.

We conclude this section by noting that in all three applications, the notion of an interval arises naturally, and there is no need to ever explicitly construct an interval graph.

5. Conclusions

In this paper we have provided improved algorithms for the k-coloring of a set of intervals in both the weighted and unweighted cases. For the weighted version, our algorithm improves over the best previous result [1] by a factor of n/k. Since n is likely to be much larger than k in practice, this is a considerable saving. For this algorithm, the running time becomes $O(kS(n) + e)$ if an interval *graph* is given, instead of the actual set of intervals. The extra $O(e)$ term arises from the need to convert the interval graph into such a set of intervals [2]. Note that even for interval graphs, our algorithm is an improvement over the $O(nS(n))$ algorithm, provided that $S(n)$ is not $O(n)$. Also, recall that the current best practical algorithm for finding shortest paths is an implementation of Dijkstra's algorithm using Fibonacci heaps [4] that takes time $O(n \log n + e)$ for general graphs, and $O(n \log n)$ if the number of edges is $O(n)$ (as in our application). Thus the explicit running time for our algorithm becomes $O(kn \log n)$. The current best theoretical algorithm for finding shortest paths is the $O(n \log n / \log \log n)$ algorithm of [5].

For the unweighted version, our algorithm improves over the best previous algorithm [12] by a factor of $\log k$. Further, the algorithm that we give is linear in both n and k, in contrast to existing methods, which are linear in n, but assume that k is a constant. Finally, we note that if an interval *graph* is given, then there is no asymptotic improvement over previous methods, since simply the time to input that graph (namely, $O(n + e)$) dominates the running time of the entire algorithm.

Acknowledgement

We thank the referees for their comments on an earlier version of this paper. Those comments provided for a considerable simplification of the results of Section 3, and an overall improved presentation.

References

[1] E.M. Arkin and E.B. Silverberg, Scheduling jobs with fixed start and end times, Discrete Appl. Math. 18 (1987) 1–8.
[2] K.S. Booth and G.S. Leuker, Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms, J. Comput. System Sci. 13 (1976) 335–379.
[3] J. Edmonds and R.M. Karp, Theoretical improvements in algorithmic efficiency for network flow problems, J. ACM 19 (1972) 248–264.
[4] M.L. Fredman and R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, J. ACM 34 (1987) 596–615.
[5] M.L. Fredman and D.E. Willard, Trans-dichotomous algorithms for minimum spanning trees, in: Proceedings of the 31st IEEE FOCS (1990) 719–725.
[6] H.N. Gabow and R.E. Tarjan, A linear-time algorithm for a special case of disjoint set union, J. Comput. System Sci. 30 (1985) 209–211.
[7] M. Garey and D. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness (Freeman, New York, 1979).
[8] M.C. Golumbic, Algorithmic Graph Theory and Perfect Graphs (Academic Press, New York, 1980).
[9] A. Hashimoto and J. Stevens, Wire routing by optimizing channel assignment within large apertures, in: Proceedings of the 8th IEEE Design Automation Workshop (1971) 155–169.
[10] R.E. Tarjan, Efficiency of a good but not linear set union algorithm, J. ACM 22 (1975) 215–225.
[11] R.E. Tarjan, Data Structures and Network Algorithms (SIAM, Philadelphia, PA, 1983).
[12] M. Yannakakis and F. Gavril, The maximum k-colorable subgraph problem for chordal graphs, Inform. Process. Lett. 24 (1987) 133–137.