A TECHNIQUE FOR EXACT COMPUTATION OF PRECOLORING EXTENSION ON INTERVAL GRAPHS*

Martin R. Ehmsen

Department of Mathematics and Computer Science, University of Southern Denmark, Campusvej 55, DK-5230 Odense M, Denmark
ehmsen@imada.sdu.dk

Kim S. Larsen†

Department of Mathematics and Computer Science, University of Southern Denmark, Campusvej 55, DK-5230 Odense M, Denmark
kslarsen@imada.sdu.dk
http://www.imada.sdu.dk/∼kslarsen/

Received (Day Month Year)
Accepted (Day Month Year)
Communicated by (xxxxxxxxxx)

Inspired by a real-life application, we investigate the computationally hard problem of extending a precoloring of an interval graph to a proper coloring under some bound on the number of available colors. We are interested in quickly determining whether or not such an extension exists on instances occurring in practice in connection with campsite bookings on a campground. A naive exhaustive search does not terminate in reasonable time. We have formulated a new approach which moves the computation time within the usable range on all the data samples available to us.

Keywords: Graph algorithms; interval graphs; graph coloring.

1. Introduction

A campground typically has different categories of campsites, such as tent sites and cabins of various sizes. For the campgrounds we have considered, the number of sites in each category typically ranges from 10 to 150.

Customers book a campsite on a campground for a period of time. This booking can be made at any time before arrival, of course depending on availability. A customer has the choice of booking a specific or an unspecified campsite. The desire to book a specific site often comes from knowing the campground and wishing to book the same site as a previous year or a site near some facility on the campground. We consider the problem of booking a campsite for the different categories separately.

*This work was supported in part by the Danish Natural Science Research Council.
†Corresponding author.

Customers who book an unspecified campsite are told on the day of arrival which concrete campsite they have been assigned. Clearly, given a booking, we must determine if the booking will overlap in time and campsite with already accepted bookings or if there is a way of assigning campsites to all unspecified bookings such that no bookings overlap in time and campsites assigned.

The above could be viewed as an online problem [2] since bookings arrive over time and an irrevocable decision of whether or not to accept a booking must be made immediately, typically via a web site or over the phone. Accepting or rejecting a booking clearly influences which future bookings can be accepted. However, campgrounds typically do not want to turn down any booking requests if it is at all possible to accept the bookings. Hence, the problem is turned into a sequence of offline decision problems of deciding whether or not the addition of one concrete booking renders the assignment of unspecified bookings to concrete campsites impossible.

This offline problem can be modeled as the precoloring extension problem (PrExt) [1] on interval graphs [6]. As indicated by the name, interval graphs can be defined via intervals. Given a number of open intervals on the real line, we let each interval represent a vertex and define that two vertices are connected by an edge in the graph if their intervals overlap. We refer to the graph defined from a collection of intervals in this manner as the *implied* graph. Each booking is an interval in time from a given date to some later date and it represents a vertex in the graph.

If we associate a color with each campsite at the campground, then a vertex coloring of the implied graph corresponds to an assignment of bookings to campsites, provided that no more than k colors are used where k is the number of campsites. Recall that for an assignment of colors to vertices to be a vertex coloring, each vertex must be given one color, and any two adjacent vertices must have different colors (for emphasize sometimes referred to as a *proper* coloring). This expresses that two bookings that overlap in time must be assigned to different campsites.

The set of bookings of specific campsites now corresponds to a precoloring, i.e., an interval is precolored with the color of the specific campsite. The overall problem can now be formulated as deciding if the precoloring of the implied interval graph can be extended to a proper k-coloring, where k is the total number of colors (campsites).

Let n denote the number of bookings. As a function of both k and n, the PrExt problem is NP-complete [1]. Even when making the restriction that each color is used at most twice in the precoloring, the problem remains NP-complete. If the number of colors is fixed and the problem is viewed as a function of n alone, then the problem becomes polynomial-time solvable [8], with a running time of $O(kn^{k+2})$. Though this is indeed polynomial, for $k = 150$ and some thousand bookings, this approach will not be feasible in practice. An alternative algorithm for graphs of bounded treewidth in [7] indicates an algorithm running in $O(k^k n)$. This is linear in n, but again not realistic in practice because of the particular dependency of k.

Hence, a new approach is needed, which takes advantage of the specific structure

of the real-life instances. In general such instances have a structure where many bookings start and end on the same day (bookings that span a weekend, bookings that span a week, etc.). Without being able to identify precisely which properties data must have for the problem to become manageable in practice, the approach we discuss below is designed with the aim of taking advantage of the real-life experience that many bookings start and end on the same days.

It is clear that even though this problem is a decision problem, there is still the problem of deciding which campsite a customer gets when the customer turns up on the first day of the booking. However, as we show below, given our solution to the decision problem, it is also possible to extract an arrangement of the given bookings in an efficient manner.

Due to the complexity theoretical nature of the problem, one would not, in general, expect to terminate with a negative answer in reasonable time, since a complete exploration of the search space is required in order to conclude that a proper coloring does not exist. In the section on experimental results, we test a large collection of known positive instances and demonstrate that our algorithm quickly determines that a proper coloring exists.

2. Definitions

In the algorithm we develop, we process the intervals by processing their end points from left to right. We will use start point and finish point to refer to the left and the right end point of an interval, respectively.

An interval which is not precolored is denoted a *movable* interval. An interval for which we have processed its start point, but not processed its finish point, is called an *active* interval. A color c is said to be *used* if an interval precolored with color c is active; otherwise the color is said to be *free*.

A *color group* is a 2-tuple $(\mathbb{C}, \mathbb{I})$, where $\mathbb{C}$ is a set of colors and $\mathbb{I}$ is a set of intervals (both precolored and movable). Our intention in the algorithm to follow is that $\mathbb{I}$ is a set of intervals that are active at a given point in time. Thus, they form a (not necessarily maximal) clique. This means that they must be given distinct colors and all the colors must come from $\mathbb{C}$.

Inserting an interval I into a color group, i.e., extending $\mathbb{I}$ with I, should be interpreted as that interval being colored with one of the colors from $\mathbb{C}$ in the final solution. The *capacity* of a color group is the number of elements in $\mathbb{C}$, and inserting an interval into a color group is said to be *exceeding* the capacity of the color group if the number of elements in $\mathbb{I}$ would then be strictly larger than the capacity of the color group. If an interval I can be inserted into a color group C without exceeding the capacity of C, then we say that I *fits* in C. We refer to a color group as being *empty* if $\mathbb{I} = \emptyset$.

A coloring of the intervals is said to be *valid* if all intervals are assigned colors such that if two intervals intersect they are not assigned the same color and all precolored intervals are assigned their precolored color.

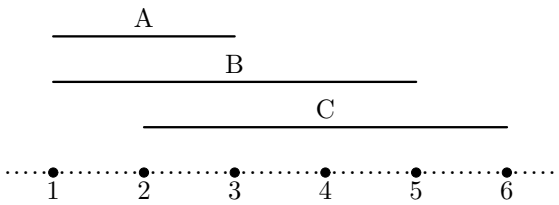

Fig. 1. An example of the progression of color groups.

3. The Algorithm

In the following, we process interval end points from left to right, i.e., in non-decreasing order of time. If some start and finish points coincide, we first process the finish points. The algorithm for processing one end point can be seen in Algorithms 1 and 2.

Instead of being an exhaustive search over all possible colorings of intervals, the algorithm now realizes an exhaustive search over all possible assignments of intervals to color groups. Especially because the merging of color groups when possible, the number of color groups, and therefore the number of steps in the search, is in practice kept much smaller than the number of colors. One could view this as merging equivalent branches in the search tree representing the search space.

In the main algorithm, the set S is used to keep track of this exhaustive search. An element in S is one concrete assignment of active intervals to color groups, and the set S represents all such assignments that are possible, i.e., which could lead to valid solutions if the input ended after the current point (as we show below). This ensures the complete traversal of the search space.

The main property of a color group which guides the design of the algorithm and underlies its correctness is that when a movable interval is placed in a color group, it can receive any of the free colors in the group.

Color groups change in particular when intervals finish. Consider the example in Figure 1 where only three colors are available and all intervals are movable.

We enumerate the intervals from top to bottom. Up until point 2, everything is in the same color group. At point 3, the first interval finishes. This means that future intervals (starting at point 4, for instance) could be given the color of interval A, but not necessarily the color of other intervals in the color group (such as interval B or C). Thus, we split the color group, by letting the color of interval A form its own singleton color group.

Next, we discover that interval B finishes. For the same reason, the color of interval B now gets its own color group. However, as the last step in the treatment of an end point, we merge color groups with no movable intervals. Thus, the colors of intervals A and B now form a color group.

The action taken when an interval starts is to examine all possibilities for placing it in a color group. In addition to the naive density test as to whether we are

Algorithm 1 The main algorithm for treating one end point.

Require: p is the next end point and S is the current set of divisions of the active intervals into color groups. Initially, S contains one element containing one color group of all the available colors and no intervals.

```
1: if p is a start point for interval I then
2:     if I is precolored with color c then
3:         for element s in S do
4:             Let C be the color group of c in s
5:             if I fits in C then
6:                 Insert I into C in s
7:             else
8:                 Remove s from S
9:     else
10:        for element s in S do
11:            Let M = {C ∈ s : I fits in C}
12:            Prune M using Algorithm 2
13:            if |M| = 0 then
14:                Remove s from S
15:            else
16:                for each color group C in M do
17:                    Create a new element s' from s
18:                    In s', insert I into C
19:                    Insert s' into S
20:                Remove s from S
21: else {p is a finish point for interval I}
22:    if I is precolored with color c then
23:        for element s in S do
24:            Let C be the color group of c in s
25:            Remove I from C
26:            Remove color c from C
27:            Create an empty color group C' of color c
28:            Insert C' into s
29:    else
30:        for element s in S do
31:            Let C be the color group containing I in s
32:            Remove I from C
33: if S is empty then
34:    Output that there exists no solution
35: for element s in S do
36:    Merge all color groups with no movable intervals
```

Algorithm 2 An algorithm for pruning search directions.

Require: I is a movable interval which is considered for assignment to the color group C.

Ensure: Return false if assigning I to C would exceed the capacity of C in the future, given knowledge of future precolored intervals.

```
1: Let cap be the number of free colors in C
2: Let den be the number of movable intervals assigned to C
3: Let P_M be the set of finish points for movable intervals assigned to C
4: Let P_P be the set of end points of the first future precolored interval for each
   of the free colors in C
5: Sort P = P_M ∪ P_P
6: for p in P do
7:     if p is a start point for interval I then
8:         Increase den by one
9:     else {p is a finish point for interval I}
10:        Decrease den by one
11:        if I is precolored then
12:            Decrease cap by one
13:    if den ≥ cap then
14:        return False
15: return True
```

considering placing an interval in a color group which already contains as many intervals as it has colors, one can add any number of additional local tests to try to determine early that the placement of an interval in a color group will not lead to overall success. A fairly efficient and also effective test of this type can be seen in Algorithm 2.

Assume that we are currently considering assigning an interval I to a color group. At that point, we know which colors are in the color group and we know the currently assigned intervals to the color group. In addition, we know which precolored intervals come in the future. Given that knowledge, it is possible to simulate the algorithm by only considering the future precolored intervals with a color belonging to that color group, and check to see if the capacity of the color group will be exceeded in the future, in which case we know for certain that assigning I to the color group will not result in valid solutions. Naturally, we only have to run the simulation until we have passed the finish point for I. This could be viewed as a further pruning of the search tree representing the search space.

Specifically, in the algorithm in Algorithm 2, we keep track of the number of active movable intervals, *den* (density), and the number of free colors, *cap* (capacity).If the density becomes at least as large as the capacity, it will not be possible to assign an *additional* interval to the color group. At the finish of a precolored interval, the main algorithm would split off a singleton color group for that one color.

This would reduce the capacity of the current color group by one. We simulate the main algorithm by looking one precolored interval ahead per color.

For other applications, if one faces searches that are too time consuming, it would be obvious to experiment further with variations of this approach, and find a good trade-off point between searching and pruning.

Overall, if the main algorithm does not output that no solution exists, then there exists at least one solution to the given problem, and all solutions (as shown below) can be extracted from the sets S created by the algorithm.

4. Correctness

Let S_m denote the set S after the algorithm has processed m end points, with S_0 denoting the initial set S, i.e., S_0 contains one element, which contains one color group of all the available colors and no intervals. Observe that each element s in S_m is derived from an element in S_{m-1} by the algorithm. Hence, the sets $S_0, S_1, \ldots, S_m$ give rise to a tree-like structure, where each element s in S_m has exactly one parent element in S_{m-1}, etc. We denote the path from the root of the tree (the one element in S_0) to a leaf element s in S_m as the *history* of s. After having processed m end points, some intervals have been processed completely (both their start and finish point have been processed), some intervals are active, and some have not yet been considered. Assume that the input ends just after the m'th end point, in the sense that all currently active intervals end in the same point. Given an element s in S_m and its history, we can color the intervals seen so far by considering each color group in s. In each color group, the precolored intervals are assigned their precolored color, and the movable intervals are assigned the remaining colors in the color group in any order. Now consider the parent s' to s in the history tree discussed above. We now process s' in the same manner as we did for s. However, some of the intervals have already been colored in the previous step, and the color of such intervals cannot be changed. Continue in this way all the way to the root in the history tree. We show below that this process creates a valid coloring of the intervals.

In the following, we repeatedly use the observation that for all m and for all s in S_m, all colors exist in exactly one color group in s. This is clearly true since all colors are in the one initial color group, and color groups are split in Line 26 and Line 27, and merged in Line 36.

Lemma 1. *For all m and for all s in S_m and all color groups C in s, the free colors in C can be assigned in any way to the active movable intervals in C.*

Proof. We first ignore the extra pruning in Line 12 and include this part again at the end of the proof.

We show this by induction in m. It is clear that the statement is initially true, since there exists only one color group which is empty. Now assume that the statement is true for $S_0, S_1, \ldots, S_{m-1}$, and consider each of the possible types for the m'th point.

First, assume that the m'th point is a start point for a precolored interval I with color c. Consider element s in S_{m-1} and let C be the color group of c in s. If I does not fit in C, the partial solution s is removed from S and hence the statement is trivially true for all elements in S_m that have s as parent (there are none). Now, assume that I fits in C. Obviously the number of free colors in C is decreased by one, but since the possible ways of assigning the free colors to the active movable intervals after I is placed in C is a subset of the possible ways that existed before I was placed in C, it follows from the induction hypothesis that the statement is still true after placing I in C, i.e., the statement is true for all children of s.

Next, assume that the m'th point is a start point for a movable interval. Consider any s in S_{m-1} and consider any child s' of s created by the algorithm where I is inserted into some color group C (without exceeding the capacity of C). By the same argument as in the previous case, if we assign (in s') any of the free colors to I, by the induction hypothesis, the remaining free colors can be assigned in any way to the other active movable intervals in C. Hence, the statement is true for s'.

Now, assume that the m'th point is a finish point for a precolored interval I with color c. Consider any s in S_{m-1}, and let C be the color group of c in s. When processing the finish point, the algorithm splits C into two color groups: One empty color group consisting of just color c, and one of all colors (except c) and all intervals (except I) from C. It is clear that the statement is true for the first color group (since it is empty). For the second color group, the active movable intervals and the free colors are exactly the same as for C. Hence, by the induction hypothesis, the statement is also true for this color group.

Finally, assume that the m'th point is a finish point for a movable interval I. Consider any s in S_{m-1} and let C be the color group containing I in s. After processing the point, the free colors in C are the same as before processing the point. The number of active movable intervals have decreased by one. After processing the point, consider any assignment of the free colors to the active movable intervals. There must be at least one free color which is not assigned to any active interval since I just ended. Pretending that we assign this color to I, we get an assignment of the free colors to the movable intervals which were active before we processed the m'th point. By the induction hypothesis, the result follows.

Finally, observe that in Line 36, only color groups (for each s in S_m) which do not contain any movable intervals are merged. Hence, the lemma still holds after this merging.

We now return to the extra pruning carried out in Line 12. It is clear that the algorithm shown in Algorithm 2 simulates the development of the color group under the assumption that no future movable intervals exist. If the capacity of the color group is exceeded under that assumption, then it is clear that assigning I to cc would not result in any valid solutions. Thus, we only prune parts of the search space that will definitely not lead to a valid solution in the main algorithm. □

It follows directly from the above lemma that the process of creating colorings

from the sets $S_0, S_1, \ldots, S_m$ described above does indeed create valid colorings, and hence, the colorings created in this way form a subset of all the possible solutions to the problem (under the assumption that the input ends just after the first m points). We now show that the algorithm actually creates *all* the solutions, establishing the correctness of our algorithm: if there is a solution, the final set S_m after processing all m end points will be non-empty, and otherwise the algorithm will have output that there exists no solution.

Lemma 2. *All possible solutions to the problem can be found as a coloring that can be created from S_m.*

Proof. Let P_m denote the set of all possible solutions to the problem after the first m points. Consider any solution p in P_m, i.e., p is a coloring of all the intervals under the assumption that the input ends just after the first m points (as described above). We show by induction that the algorithm creates the same coloring. The base case is trivial. Assume that the statement is true for the first $m-1$ points. If the m'th point is the finish point of an interval, then the statement follows directly from the induction hypothesis, since the intervals remain the same going from the first $m-1$ points to the first m points. Now, assume that the m'th point is a start point for some interval I (precolored or not). Assume that in p, the interval I is assigned some color c. Hence, there exists a solution p' in P_{m-1} where all intervals (except I) are assigned the same colors as in p and the color c is not used for the intervals that are active after the first $m-1$ points. By the induction hypothesis, there exists an s' in S_{m-1} where the coloring process can result in the same coloring as p' (if, when there is a choice in a color group, we color the interval according to the color assigned to it in p'). Since color c is not used in the coloring, inserting another interval into the color group in s' containing color c would not exceed the capacity of the color group. It now follows directly from the algorithm that it creates an element s in S_m which can be colored identically to p. □

Theorem 3. *Algorithm Correctness: the set of all colorings that can be created from S_m by the above process is exactly all the possible solutions to the problem.*

Proof. Follows from Lemmas 1 and 2. □

5. Implementation

The algorithm is implemented in the programming language PYTHON. Due to the large recursion depth, we have implemented the recursive depth-first search in the form of a backtracking algorithm. (The reason was that PYTHON simply would not accept the recursion depth.)

The conversion of a recursive algorithm, such as the one presented here, into a backtracking algorithm is trivial. The only difference is that the call stack has to

be maintained explicitly in the program rather than implicitly by the programming language and compiler/interpreter.

In addition, we have implemented the algorithm for extracting an assignment of the bookings to campsites in the case where a valid solution is found. When a valid solution is found, we assign all movable intervals a color from the color group in which they were placed for this particular path in the history tree. All precolored intervals are of course given their precolored color. By Lemma 1, this will result in a valid solution.

We have focused on the algorithmic improvements rather than fine-tuning of the code. Thus, optimizing the code or porting to C could likely improve the running times with roughly an order of magnitude.

Our implementation simply outputs either "True", indicating that the given instance has a solution, or "False", indicating that the given instance does not have a solution.

For the source code, as well as data and test runs, see [4].

6. Complexity

In the worst case, the algorithm can have a very long running time. If there are k colors, then it is clear that at any point in time during the execution of the algorithm at most k color groups can exist. Since each of the n start points can make the algorithm branch each of the current solutions into k new possible solutions (Line 17), we get a bound on the total running time of $O(k^n)$, which is much worse than both $O(kn^{k+2})$ and $O(k^k n)$ from [8] and [7], respectively, since n is in general much larger than k. However, with a slight modification of our algorithm, we can get the running time down to $O(k^k n)$.

Observe that at any point in time during the execution of the algorithm at most k intervals are active. Hence, these at most k intervals can be placed into the at most k color groups in $O(k^k)$ different ways. It is clear that the placement of the set of active intervals into the current color groups uniquely determines if the intervals yet to be processed can be colored by the algorithm and a solution found, i.e., the actions prior to the set of active intervals do not add any additional constraints. Hence, if we have concluded at some point that a particular node in the history tree cannot lead to a valid solution and we arrive at the same division of the set of active intervals into color groups during the backtracking, then we can immediately conclude that the current state cannot lead to a valid solution and the algorithm can start to backtrack without exploring the current subtree of the history tree further.

If we extended the algorithm with this functionality, we would arrive at an algorithm with a running time of $O(k^k n)$. We would of course need to implement some sort of look-up table where the algorithm could check if the current division of the set of active intervals into color groups has already been tried.

The space consumption of the algorithm as presented in this paper, i.e., without

the additional functionality discussed above, is the optimal $O(n+k)$. The algorithm only considers one path down the history tree at a time, the path length of which is bounded by $O(n)$, and at any point in time the number of active intervals and color groups is $O(k)$.

7. Experimental Results

We performed several experiments on real-life data from two different campgrounds, each having several categories of campsites.

The experiments were performed on an Intel Core2Duo 2.33 GHz CPU running Ubuntu Linux 2.6.24-27-generic. Only one of the cores were used.

The results from the full algorithm with and without the extra pruning and the list-coloring algorithm in [10] are displayed in Table 1. Since it does not matter what type of booking it is, we have just enumerated them. There is no connection between bookings of different types. Thus, in everything that follows, we assume that we are discussing one selected type at one selected campground.

In Table 1, "Sites" is the number of campsites (number of colors) available. "Size" is the number of bookings. This is over approximately three years.

We have made two significantly different type of test runs.

For the first test run, we have considered the instance with these three years of bookings. "Total" shows the running time of this. However, this was just in order to try our algorithm on large instances. Since bookings come in over time, this instance has never occurred in practice at the campground.

Secondly, we have run the algorithm once for every booking. The data we have available lists (for all types) the date of booking, the date of arrival, the date of departure, which campsite (color) any given booking was historically given, and if the booking was for a specific site (precolored) or not.

When processing a given booking, we consider the situation at the time of booking. Thus, the instance consists of all bookings made before that time for a later stay. All bookings for stays before the time of the booking under consideration are ignored, except for all current bookings, i.e., bookings with arrival time before and departure time after the time of the booking under consideration. These bookings are precolored to be given the site that they were historically given. Because of this, these instances are not necessarily subinstances of the large instance consisting of all the bookings, and though they are smaller, some of them could in principle be harder to solve than the large instance. Under "Max", we list the maximal time it took to process such an instance. This is the maximum time a customer would have experienced when booking via a home page, via a phone call to the campground, etc.

Due to the nature of how the data was collected at the campground, all instances are positive instances. The campground solely collected data concerning who was staying at the campground, date of arrival, date of departure, and at which specific campsites they stayed. No data was collected concerning potential customers they

				Pruning		*No pruning*		[10]	
	Type	*Sites*	*Size*	*Total*	*Max*	*Total*	*Max*	*Total*	*Max*
Campground 1	1.1	8	604	0.02	0.00	0.01	0.00	17.41	0.01
	1.2	15	827	0.05	0.01	0.04	0.01	–	–
	1.3	10	811	0.04	0.01	0.02	0.01	266.91	0.20
	1.4	10	1016	0.06	0.01	0.03	0.01	9.75	0.09
	1.5	10	1586	0.09	0.01	0.03	0.01	5.18	5.88
	1.6	50	1416	0.44	0.01	0.08	0.01	0.23	0.02
	1.7	70	2659	0.95	0.11	0.23	–	6.85	0.21
	1.8	145	7474	8.08	0.27	–	0.28	88.94	0.66
	1.9	26	711	0.07	0.01	–	–	–	0.03
Campground 2	2.1	2	72	0.00	0.00	0.00	0.00	0.00	0.00
	2.2	12	1140	0.05	0.01	0.03	0.01	–	0.39
	2.3	5	455	0.01	0.00	0.01	0.00	0.01	0.00
	2.4	4	193	0.00	0.00	0.00	0.00	0.00	0.00
	2.5	17	3119	0.26	0.02	0.09	–	–	0.07
	2.6	3	62	0.00	0.00	0.00	0.00	0.00	0.00
	2.7	191	7912	5.58	0.21	1.12	0.26	11.47	0.34

Table 1. Results in seconds on the described data.

had to turn down. Hence, we have no real-life negative instances.

All times are in seconds. "–" indicates timeout which means that some call to the main algorithm ran for 10 minutes without reporting an answer. During the testing, we ran our algorithm and the algorithm from [10] for hours on a number of instances, but we found that it did not make any difference: either we found a positive answer fairly quickly or no answer would be returned, even if we let it run for hours. Thus, the 10 minute deadline was set just to be way above the time required to get a positive answer for the instances that were at all solvable for one of the two algorithms.

Obviously, the extra work in applying the pruning increases the total running time for some instances, but for others, this additional work is essential to achieve a result within reasonable time.

In addition, we have observed that the exact algorithm from [10] is not capable of handling all instances without timeout. However, it should be noted that their algorithm is solving the more general, and hence harder, list-coloring problem, and

with a deeper understanding of their algorithm, it might be possible to improve it when the input is restricted to instances of precoloring extension.

8. Concluding Remarks

Due to the nature of the problem and the exhaustive search techniques which are employed, it is, in general, impossible to conclude that an interval can *not* be accepted, since, in order to do so, one would have to traverse the entire search space without finding a solution. Reporting a negative answer is of course possible in very simple situations, for instance if at some point the number of active intervals exceeds the total number of colors. However, in general, to give a negative answer with certainty requires an exhaustive search which cannot terminate within a reasonable time frame. So, neither our algorithm nor the algorithm in [10] is capable of "saying no". Instead, one has to decide on a time limit, and if no positive answer is obtained within that time limit, a negative answer is returned which should be interpreted as the algorithm "cannot say yes". Based on the numbers in Table 1 and the discussions of that table, this approach works on these real-life datasets for our algorithm.

Our focus has been on instances where the answer is positive, and on all such instances, we have been able to report that answer quickly. As explained in the paper, instances where the answer is negative are less interesting, since we cannot expect to be able to compute such results within a reasonable time frame.

Due to the complexity theoretical nature of the problem, it is clear that our algorithm exploits some naturally occurring properties of the data. We leave as an interesting open problem to determine if this can be theoretically substantiated, using, for instance, techniques from parameterized complexity [3, 5, 9]. Initial considerations indicate that the most obvious parameters do not capture the essence of the algorithm, but intuitively some parameter related to the clique sizes must come in to play.

Acknowledgments

We would like to thank Dancamps for providing real-life experimental data and their representative, Chief Developer Nicolai Dvinge, for valuable initial discussions.

References

[1] M. Biró, M. Hujter, and Zs. Tuza. Precoloring extension. I. interval graphs. *Discrete Mathematics*, 100(1-3):267–279, 1992.

[2] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[3] Rodney G. Downey and M. R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999.

[4] Martin R. Ehmsen and Kim S. Larsen. Testing a heuristic for exact computation of precoloring extension on interval graphs on real-life campsite reservation data. `http://www.imada.sdu.dk/~kslarsen/Archive/`, 2010.

[5] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. Springer, 2006.
[6] Tommy R. Jensen and Bjarne Toft. *Graph Coloring Problems*. John Wiley & Sons, 1995.
[7] J. Kratochvíl. Precoloring extension with fixed color bound. *Acta Mathematica Universitatis Comenianae*, 62(2):139–153, 1993.
[8] Dániel Marx. Parameterized coloring problems on chordal graphs. *Theoretical Computer Science*, 351(3):407–424, 2006.
[9] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*, volume 31 of *Oxford Lecture Series in Mathematics and Its Applications*. Oxford University Press, 2006.
[10] Thomas Zeitlhofer and Bernhard Wess. List-coloring of interval graphs with application to register assignment for heterogeneous register-set architectures. *Signal Processing*, 83:1411–1425, 2003.