Generalizing `AllDifferent`: The `SomeDifferent` Constraint

Yossi Richter, Ari Freund, and Yehuda Naveh

IBM Haifa Research Lab, Haifa University Campus, Haifa 31905, Israel
{richter, arief, naveh}@il.ibm.com

Abstract. We introduce the `SomeDifferent` constraint as a generalization of `AllDifferent`. `SomeDifferent` requires that values assigned to *some* pairs of variables will be different. It has many practical applications. For example, in workforce management, it may enforce the requirement that the same worker is not assigned to two jobs which are overlapping in time. Propagation of the constraint for hyper-arc consistency is NP hard. We present a propagation algorithm with worst case time complexity $O(n^3\beta^n)$ where n is the number of variables and $\beta \approx 3.5$ (ignoring a trivial dependence on the representation of the domains). We also elaborate on several heuristics which greatly reduce the algorithm's running time in practice. We provide experimental results, obtained on a real-world workforce management problem and on synthetic data, which demonstrate the feasibility of our approach.

1 Introduction

In this paper we consider a generalization of the well known `AllDifferent` constraint. The `AllDifferent` constraint requires that the variables in its scope be assigned different values. It is a fundamental primitive in constraint programming (CP), naturally modeling many classical constraint satisfaction problems (CSP), such as the n-queen problem, air traffic management [2,9], rostering problems [18], and many more. Although a single `AllDifferent` constraint on n variables is semantically equivalent to $n(n-1)/2$ binary `NotEqual` constraints, in the context of maintain-arc-consistency algorithms it is vastly more powerful in pruning the search space and reducing the number of backtracks. It also simplifies and compacts the modeling of complex problems. It has thus attracted a great deal of attention in the CP literature (see [19] for a survey).

Despite its usefulness, the `AllDifferent` constraint is too restrictive in many applications. Often we only desire that certain pairs of variables assume different values, and do not care about other pairs. A simple example of this is the following workforce management problem [21]. We are given a set of jobs, a set of workers, and a list specifying which jobs can be done by which workers. In addition, the jobs may be specified to start and end at different times, or require only partial availability of a worker. Consequently, some pairs of jobs may be assigned to the same worker while others may not. A simple way to model the problem is to let each job correspond to a variable, and let the domain of each

F. Benhamou (Ed.): CP 2006, LNCS 4204, pp. 468–483, 2006.

variable be the set of workers qualified to do the job. Then one can add a binary NotEqual constraint for every two jobs which overlap in their time of execution and require together more than 100% availability. Additional constraints can then be added according to the detailed specification of the problem. While semantically correct, this model suffers from the same disadvantages of modeling an AllDifferent problem by multiple NotEqual constraints. On the other hand, the AllDifferent constraint is inappropriate here because in general, the same worker can be assigned to two jobs.

While workforce management is a CP application of prime, and growing, importance (especially in light of the current trends towards globalization and strategic outsourcing), it is not the only problem in which the above situation arises. Some further examples include: circuit design, in which any two macros on a chip may or may not overlap, depending on their internal structure; university exam scheduling, in which the time-slot of any two exams may or may not be the same, depending on the number of students taking both exams; and computer-farm job scheduling, in which, depending on the types of any two jobs, the same machine may or may not process them simultaneously.

All of this calls for a generalization of AllDifferent in which some, but not all, pairs of variables require distinct values. We call this generalization SomeDifferent. Formally, the SomeDifferent constraint is defined over a set of variables $X = \{x_1, \ldots, x_n\}$ with domains $D = \{D_1, \ldots, D_n\}$, and an underlying graph $G = (X, E)$. The tuples allowed by the constraint are: $\texttt{SomeDifferent}(X, D, G) = \{(a_1, \ldots, a_n) : a_i \in D_i \ \wedge \ a_i \neq a_j \text{ for all } (i, j) \in E(G)\}$. The special case in which G is a clique is the familiar AllDifferent constraint. Another special case that has received some attention is the case of two AllDifferent constraints sharing some of their variables [1].

Our focus in this paper is on hyper-arc consistency propagation for the SomeDifferent constraint. Since most CSP algorithms use arc-consistency propagation as a subroutine (see, e.g., [7]), it is important to develop specialized propagation algorithms for specific constraint types—algorithms that are able to exploit the concrete structure of these constraints. For example, the AllDifferent constraint admits a polynomial-time hyper-arc consistency propagation algorithm based on its bipartite graph structure [14]. In contrast, and despite the similarity between SomeDifferent and AllDifferent, there is little hope for such an algorithm for the SomeDifferent constraint, as the hyper-arc consistency propagation problem for SomeDifferent contains the NP hard problem of graph 3-colorability as a special case.

The NP hardness of the propagation problem accommodates two approaches. One is to aim for relaxed or approximated propagation. This approach has been taken previously in the context of other NP Hard propagation problems [15,16,17]. The second approach is to tackle the problem heuristically. In this paper we combine a theoretically grounded exact algorithm (with exponential worst case running time) with several heuristics that greatly speed it up in practice.

Our results. We introduce an exact propagation algorithm for hyper-arc consistency of the `SomeDifferent` constraint. The algorithm has time complexity of $O(n^3\beta^n)$, with $\beta \approx 3.5$, and depends on the domain sizes only for the unavoidable deletion operations. We have implemented the algorithm (with multiple additional heuristics) and tested it on two kinds of data:

- IBM's workforce management instances.
- Synthetic data generated through a random graph model.

In both cases the implementation performed well, much better than expected from the theoretical bounds. It also compared favorably (though not in all cases) with the approach of modeling the problem by `NotEqual` constraints.

Organization of this paper. The remainder of the paper is organized as follows. In Section 2 we describe a graph theoretical approach to the problem of propagating the `SomeDifferent` constraint for hyper-arc consistency. In Section 3 we present our algorithm and its worst-case analysis. In Section 4 we discuss a few practical heuristic improvements. In Section 5 we report experimental results. In Section 6 we conclude and discuss future work.

2 A Graph-Theoretical Approach

The problem of propagation of `SomeDifferent` for hyper-arc consistency can be formulated as the following graph coloring problem. The input is a graph $G = (V, E)$, where each vertex u is endowed with a finite set of *colors* D_u. The vertices correspond to variables; the sets of colors correspond to the domains of the respective variables; the graph edges correspond to pairs of variables that may not be assigned the same value, as mandated by the constraint. A valid *coloring* of the graph is an assignment $c : V \to \bigcup_{u \in V} D_u$ of colors to vertices such that: (1) each vertex u is assigned a color $c(u) \in D_u$; and, (2) no two adjacent vertices are assigned the same color. If a valid coloring exists, we say the graph is *colorable*.[1] Valid colorings correspond to assignments respecting the `SomeDifferent` constraint. We view a coloring of the graph as a collection of individual *point colorings*, where by *point coloring* we mean a coloring of a single vertex by one of the colors. We denote the point coloring of vertex u by color c by the ordered pair (u, c). We say that (u, c) is *extensible* if it can be *extended* into a valid coloring of the entire graph (i.e., if there exists a valid coloring of the entire graph in which u is colored by c). The problem corresponding to propagation of `SomeDifferent` for hyper-arc consistency is: given the graph and color sets, prune the color sets such that: (1) every extensible point coloring in the original graph remains in the pruned graph; and (2) every point coloring in the pruned graph is extensible. (We view the color sets as part of the graph. Thus we speak of the "pruned graph" despite the fact that it is actually the color sets that are pruned, not the graph itself.) This problem is NP hard, as even the problem

[1] We emphasize that we do not refer here to the usual terminology of graph coloring, but rather to coloring from domains.

of determining whether the graph is colorable contains the NP hard problem of 3-colorability as the special case in which all color sets are identical and contain three colors.

The solution to the above pruning problem is of course unique. It consists of pruning all point colorings that are non-extensible with respect to the original graph. Thus the problem reduces to identifying the non-extensible point colorings and eliminating them. The direct approach to this is to test each point coloring (u, c) for extensibility by applying it (i.e., conceptually coloring u by c, deleting c from the color sets of the neighbors of u, and deleting u from the graph) and testing whether the remaining graph is colorable. Thus the number of colorability testings is $\sum_u |D_u|$, and the time complexity of each is generally exponential in the total number of colors available. This can be prohibitively costly when the color sets are large, even if the number of vertices is small. Fortunately, it is possible to do better in such cases. Let us denote $D(U) = \bigcup_{u \in U} D_u$ for any $U \subseteq V$. Following the terminology of [13], we make the following definition.

Definition 1. *We say that a set of nodes U is a* failure set *if $|D(U)| < |U|$, in which case we also say that the subgraph induced by U is a* failure subgraph. *Note that the empty set is not a failure set.*

The key observation on which our algorithm is based is contained in Lemma 2 below, whose proof requires the next lemma.

Lemma 1. *If the graph contains no failure sets, then it is colorable.*

Proof. This is a straightforward application of Hall's Theorem (see, e.g., Reference [20], Chapter 3.1):

> Let $H = (L, R, F)$ be a bipartite graph, and for every subset $U \subseteq L$, let $N(U) = \{v \in R \,|\, \exists u \in U, uv \in F\}$. If $|N(U)| \geq |U|$ for all $U \subseteq L$, then F admits a matching that *saturates* L, i.e., a matching in which every vertex in L is matched.

We apply Hall's theorem by defining the following bipartite graph $H = (L, R, F)$: $L = V$; $R = D(V)$; and $F = \{uc \,|\, u \in L \wedge c \in D_u\}$. The condition that no subset of V is a failure set translates into the condition of Hall's Theorem for H, and therefore a matching saturating L exists. This matching defines a coloring of G in which each vertex is assigned a color from its corresponding color set, and no color is shared by two vertices. Such a coloring is necessarily valid. □

Lemma 2. *The graph is colorable if and only if each of its failure subgraphs is colorable.*

Proof. Clearly, if the graph is colorable, then so is every failure subgraph. Conversely, if the graph is not colorable, then by Lemma 1 it contains a failure subset. Let $U \subseteq V$ be a maximal failure set, i.e., U is a failure set and is not a proper subset of any other failure set. If $U = V$ we are done. Otherwise, let $D_1 = D(U)$ and $D_2 = D(V) \setminus D_1 = D(V \setminus U) \setminus D_1$. Consider the subgraph induced by $V \setminus U$ with its color sets restricted to the colors in D_2 (i.e., with

each color set D_v ($v \in V \setminus U$) replaced by $D_v \setminus D_1$). Neither this graph, nor any of its induced subgraphs may be failure subgraphs, for had any subset of $V \setminus U$ been a failure set with respect to D_2, then so would its union with U be (with respect to $D_1 \cup D_2$, i.e., when the original color sets for the vertices in $V \setminus U$ are reinstated), contradicting the maximality of U. Thus, by Lemma 1, the subgraph induced by $V \setminus U$ is colorable using only colors from D_2. It follows that the subgraph induced by U is not colorable. (Otherwise the entire graph would be colorable, since the coloring of U would only use colors from D_1 and so could coexist with the coloring of $V \setminus U$ that uses only colors from D_2). Thus, the non-colorability of the graph implies the existence of a failure subgraph that is non-colorable, which completes the proof. □

3 The Algorithm

Testing whether a point coloring (u, c) is extensible can be carried out by removing c from the color sets of u's neighbors and deleting u (thus effectively coloring u by c), and checking whether the resulting graph is colorable. Lemma 2 implies that it is sufficient to check whether some induced subgraph of the resulting graph is colorable. This seems to buy us very little, if anything, since we now replace the checking of a single graph with the checking of many (albeit smaller) subgraphs. (And, of course, we must still do this for each of the $\sum_u |D_u|$ point colorings.) However, we can realize significant savings by turning the tables and enumerating subsets of vertices rather than point assignments. More specifically, we enumerate the non-empty proper subsets of V, and test point colorings only for those that are failure sets. As we shall see shortly, this reduces the number of colorability checkings to at most $n^2 2^n$, where n is the number of vertices.

Postponing to later the discussion of how to check whether a subgraph is colorable, we now present the pruning algorithm.

1. For each $\emptyset \subsetneq U \subsetneq V$:
2. If $|D(U)| \leq |U|$:
3. For each $c \in D(U)$ and $v \in V \setminus U$ such that $c \in D_v$:
4. Check whether the subgraph induced by U is colorable with c removed from the color sets of the neighbors of v. If not, report (v, c) as non-extensible.
5. Prune the point colorings reported as non-extensible.
6. Check whether the pruned graph contains a vertex with an empty color set. If so, announce that the graph is not colorable.

Note that as the algorithm stands, the same point coloring may be reported multiple times. We address this issue in Section 4.

3.1 Correctness of the Algorithm

We now argue that the algorithm correctly identifies the non-extensible point coloring when the graph is colorable, and announces that the graph is not colorable when it is not.

Proposition 1. *If the graph is colorable, the algorithm reports all non-extensible point colorings, and only them.*

Proof. A point coloring is reported by the algorithm as non-extensible only if applying it renders some subgraph (and hence the entire graph) non-colorable. Thus all point colorings reported by the algorithm are indeed non-extensible. Conversely, to see that all non-extensible point colorings are reported, consider any such point coloring (v, c). Coloring v by c effectively removes c from the color sets of v's neighbors, and the resulting graph (in which v is also removed) is non-colorable (since (v, c) is non-extensible). By Lemma 2, this graph contains a subgraph induced by some failure set U that is not colorable. But originally this subgraph was colorable, since the original graph is colorable. This is only possible if $c \in D(U)$. Additionally, because U is a failure set with c removed, it must be the case that $|D(U)| \leq |U|$ before c is removed. Thus, when the algorithm considers U it will report (v, c) as non-extensible. □

Proposition 2. *If the graph is non-colorable, the algorithm detects this.*

Proof. If the graph is non-colorable, then by Lemma 2 it contains a non-colorable subgraph induced by some failure set U. If U is a singleton $U = \{v\}$, then $D_v = \emptyset$, and we are done. Otherwise, U must contain at least one vertex v such that $D_v \subseteq D(U \setminus \{v\})$, for otherwise U would not be a failure set (and furthermore, the subgraph induced by it would be colorable). Thus, $|D(U \setminus \{v\}\,| = |D(U)| \leq |U \setminus \{v\}\,|$, so when the algorithm considers $U \setminus \{v\}$ it will enter the inner loop and report all the point colorings involving v as non-extensible. □

3.2 Checking Whether a Subgraph is Colorable

We are left with the problem of testing whether a given subgraph is colorable. We do this by reducing the colorability problem to a *chromatic number* computation. The *chromatic number* of a given graph G, denoted $\chi(G)$, is the minimum number of colors required for coloring G's vertices such that no two neighbors receive the same color. In the terms of our coloring framework, we are considering here the special case in which all color sets are identical. The chromatic number of the graph is the minimum size of the color set for which a valid coloring still exists. Computing the chromatic number of a graph is known to be NP hard.

We use the following construction. To test the colorability of the subgraph induced by U, extend this subgraph as follows. Let $r = |D(U)|$. Add r new vertices, each corresponding to one of the colors in $D(U)$, and connect every pair of these by an edge (i.e., form a clique). Then add an edge between each vertex $v \in U$ and each new vertex that corresponds to a color that is *not* in D_v. Let G' be the resulting graph.

Claim. The subgraph induced by U is colorable if and only if $\chi(G') = r$.

Proof. If the subgraph induced by U is colorable, then we already have a coloring of the vertices in U by at most r colors. We can extend this coloring to

G' by coloring each of the new vertices by the color (from $D(U)$) to which it corresponds. It is easy to see that in so doing, no two neighbors in G' receive the same color. Thus $\chi(G') \leq r$ and as G' contains a clique of size r, $\chi(G') = r$ Conversely, if $\chi(G') = r$, then there exists a coloring of the vertices in G', using r colors, such that no two neighbors receive the same color. Note that the new vertices all have different colors since they are all adjacent to each other. Without loss of generality, assume that the colors are $D(U)$ and that each of the new vertices is given the color it corresponds to. (Otherwise, simply rename the colors.) Now consider a vertex $v \in U$. For each of the colors $c \in D(U) \setminus D_v$ the new vertex corresponding to c is adjacent to v and is colored by c. Thus v cannot be colored by any of these colors, and is therefore colored by a color in its color set D_v. Thus the restriction of the coloring to the subgraph induced by U is valid. □

Therefore, to test whether the subgraph induced by U is colorable we construct the extended graph G' and compute its chromatic number. Computing the chromatic number of a graph is a well researched problem, and a host of algorithms were developed for it and related problems (e.g., [3,4,5,6,8,10,11]). Some of these algorithms (e.g., DSATUR [5]) are heuristic—searching for a good, though not necessarily optimal, coloring of the vertices. By appropriately modifying them and properly setting their parameters (e.g., backtracking versions of DSATUR) they can be forced to find an optimal coloring, and hence also the chromatic number of the graph. In contrast, there are other algorithms, so called *exact* algorithms, that are designed specifically to find an optimal coloring (and the chromatic number) while attempting to minimize the worst case running time (e.g., [4,6,8,11]). These algorithms are based on enumerating (maximal) independent sets of vertices. Their time complexity is $O(a^k)$, where a is some constant (usually between 2 and 3) and k is the number of vertices in the graph. The algorithms differ from one another by the various divide-and-conquer techniques they employ to reduce the running time or the memory consumption. The worst case running times of the heuristic algorithms are of course exponential too, but with higher exponent bases.

An important observation regarding the complexity of computing the chromatic number of G' is that the number of vertices it contains is $|U| + |D(U)| \leq 2|U|$, since we only perform the computation for sets U such that $|D(U)| \leq |U|$. Thus for an $O(\alpha^k)$ algorithm, the actual time complexity bound is $O\big((\alpha^2)^{|U|}\big)$.

While little can be said analytically with respect to the heuristic algorithms, the exact algorithms are open to improvement due to their reliance on independent sets and the fact that the new vertices in G' form a clique. This can be demonstrated, e.g., on Lawler's algorithm [11], whose running time (in conjunction with Eppstein's maximal independent set enumeration algorithm [8]) is $O(2.443^k)$. The proof of the following proposition is omitted due to lack of space.

Proposition 3. *A suitably modified version of Lawler's algorithm runs on G' in $O(|U| \cdot 2.443^{|U|})$, rather than $O\big((2.443^2)^{|U|}\big)$.*

3.3 Complexity Analysis

The time complexity of the algorithm is determined by the loops. The outer loop iterates $2^n - 2$ times (n is the number of vertices). For each subset U with $|D(U)| \leq |U|$, the inner loop is iterated at most $|D(U)| \cdot |V \setminus U| \leq |U| \cdot |V \setminus U| \leq n^2/4$ times. Thus the total number of chromatic-number computations is at most $\frac{1}{4}n^2 2^n$. Letting $\mathrm{O}(n\alpha^n)$ denote the time complexity of the chromatic-number algorithm used, we get a time bound of $\mathrm{O}(n^3 2^n \alpha^n)$ on the work involved in chromatic-number computations. This bound can be tightened to $\mathrm{O}(n^3(\alpha+1)^n)$ by observing that the chromatic-number algorithm is called at most $n^2\binom{n}{k}$ times with a subgraph of size k, and so the total time is bounded by $\mathrm{O}\big(n^2 \sum_{k=1}^{n} \binom{n}{k} k\alpha^k\big) = \mathrm{O}(n^3(\alpha+1)^n)$.

In addition to the chromatic number computations, there is some work involved in manipulating color sets—specifically, evaluating $|D(U)| \leq |U|$ for each subset U, determining which vertices have colors from $D(U)$ in their color sets, and deleting colors from color sets. The time complexity of these operations will depend on the way the color sets are represented (e.g., by explicit lists, number ranges, bit masks, etc.), but, under reasonable assumptions, will not exceed $\mathrm{O}(n^3)$ time in the worst case (and in practice will be much better) for each iteration of the outer loop. There is an additional, unavoidable, dependence on the domain sizes of vertices not in U, which is incurred when the algorithm considers and deletes point colorings. This dependence is logarithmic at worst, using a straightforward implementation. Such unavoidable dependences notwithstanding, the worst case total running time of the algorithm is $\mathrm{O}(n^3 2^n + n^3(\alpha+1)^n) = \mathrm{O}(n^3(\alpha+1)^n)$.

4 Practical Improvements

There are a number of heuristics which when added to the algorithm greatly enhance its practicality. We list some of them next.

Connected components and superfluous edges. The two sources of exponentiality are the enumeration of subsets and the chromatic number computations (which are exponential in the number of vertices). It is fairly obvious that if the graph is not connected, the algorithm can be run on each connected component separately without affecting its correctness. Also, if an edge has two endpoints with disjoint color sets, then it can be dropped without affecting the semantics of the graph. We can therefore delete any such edges, decompose the resultant graph into connected components and run the algorithm on each separately. Doing so yields a speedup that is (potentially) exponential in the difference between the graph size and the connected component size (e.g., if the graph contains 200 nodes and consists of twenty components of size 10 each, then the potential speedup is $\beta^{200}/(20 \cdot \beta^{10}) = \frac{1}{20} \cdot \beta^{190}$).

A refinement of this heuristic is based on the observation that a failure subgraph is non-colorable if and only if it contains a non-colorable connected failure

subgraph of itself. Thus it is sufficient to enumerate only subsets that induce connected subgraphs (although we do not know how to do so efficiently).

Enumeration order. A further reduction in running time comes from the simple observation that if a subset U satisfies $D(U) \geq n$, then so do all of U's supersets, and so they need not be considered at all. A simple way to exploit this is to enumerate the subsets by constructing them incrementally, using the following recursive procedure. Fix an order on the vertices $v_1, \ldots, v_n$, and call the procedure below with $i = 0$, $U = \emptyset$.

Procedure **Enumerate**(i, U)
1. If $i = n$, process the subgraph induced by U.
2. Else:
3. Enumerate$(i + 1, U)$.
4. If $|D(U \cup \{v_{i+1}\})| \leq n$, Enumerate$(i + 1, U \cup \{v_{i+1}\})$.

This procedure is sensitive to the order of vertices; it makes sense to sort them by decreasing size of color set. A further improvement is to perform a preprocessing step in which all vertices u such that $|D_u| \geq n$ are removed from the list.

Early pruning of point colorings. It is not difficult to verify that the correctness of the algorithm is preserved if instead of just reporting each non-extensible point coloring it finds, it also immediately deletes it (so that the subsequent computation applies to the partially pruned graph). The advantage here is that we avoid having to consider over and over again point assignments that have already been reported as non-extensible. This eliminates the problem of the same point assignment being reported multiple times, and, more importantly, it also eliminates the superfluous chromatic-number computation performed each time the point coloring is unnecessarily considered. In addition, the algorithm can immediately halt (and report that the original graph is not colorable) if the color set of some vertex becomes empty. The downside of this heuristic is that the sets $D(U)$ grow smaller as the algorithm progresses, thus causing more subsets U to pass the $|D(U)| \leq |U|$ test and trigger unnecessary chromatic number computations. Of course, in this case it is an easy matter to eat the cake and have it too. We simply compute $|D(U)|$ with respect to the original graph, and do everything else on the (partially) pruned graph.

We remark that this last heuristic is especially powerful in conjunction with the **Enumerate** procedure above, because that procedure considers (to some extent) subsets before it considers their supersets. This allows the algorithm to detect, and prune, point assignments using small subsets, and so avoid chromatic-number computations on larger subgraphs. This phenomenon could be enhanced by enumerating the subsets in order of increasing size (i.e., first all singletons, then all pairs, then all triplets, etc.) but then it would not be easy to avoid enumerating subsets with $|D(U)| > n$. A hybrid approach might work best.

Speeding the chromatic number computation. In our algorithm we are really only interested in knowing whether the chromatic number of the extended

graph G' is r, and not in finding its actual value. Thus we can preprocess G' as follows: as long as G' contains a vertex whose degree is less than r, delete this vertex. (The process is iterative: deleting a vertex decreases the degree of its neighbors, possibly causing them to be deleted too, etc.) The rationale is that deleting a vertex cannot increase the chromatic number of the graph, and, on the other hand, if after deleting the vertex the graph can be colored with r colors, then this coloring can be extended to the vertex by giving it a color different from the (at most $r-1$) colors given to its neighbors.

Redundant chromatic number computations. It is only necessary to perform a chromatic-number computation on failure subgraphs. Although a (non-failure) set U such that $|D(U)| = |U|$ must pass into the inner loop, it is quite possible that for a given point assignment (v, c), $v \in V \setminus U$, $c \in D(U)$, the color c appears in the color set of some vertex in U that is not adjacent to v. In such a case, applying the point coloring does not decrease $|D(U)|$ and does not turn U into a failure set, thus obviating the chromatic-number computation.

5 Experimental Results

We created a prototype implementation of our algorithm, incorporating most of the improvements mentioned in the previous section. The code was created by adapting and extending pre-existing generic CP code implementing the MAC-3 algorithms, and was not optimized for the needs of our algorithm. For chromatic-number computations, we used Michael Trick's implementation of an exact version of the DSATUR algorithm (available, as of July 5, 2006, at !http:// www . cs . sunysb . edu / algorith / implement / trick / distrib / trick.c!). We evaluated the code's performance on a Linux machine powered by a 3.6 GHz Intel Pentium 4 processor.

We performed two sets of performance testings: one using real workforce management data, and the other using synthetically generated data. Next we describe the data on which we ran the algorithm, and following that, the results we obtained.

Workforce management data. The real data we used originated in an instance of a workforce management (WM) problem in a certain department in IBM, the likes of which are routinely handled by that department. We were given a file containing 377 job descriptions. Each job description consisted of the dates during which the job was to be performed and the list of people qualified to perform it. The total number of people was 1111. In our model of the problem, jobs correspond to vertices, people correspond to colors, and pairs of jobs overlapping in time correspond to graph edges.

By running our algorithm on randomly selected subsets of jobs we discovered that subsets of more than 50 jobs were almost always unsatisfiable because there were quite a few instances of pairs of jobs overlapping in time that could only be performed by a single person—the same person. The algorithm would detect this almost instantly. In order to stress the algorithm and test its limits in a more meaningful manner, we artificially deleted the offending jobs (leaving one of each

pair). We ended up with 312 jobs. We tested the algorithm on random subsets of these jobs, in sizes ranging from 20 to 300 jobs, at increments of ten. For each subset size $n = 20, 30, \ldots, 300$, we selected (uniformly and independently) ten subsets of n jobs and ran the algorithm on each subset. In total, we ran the algorithm on 290 instances.

Synthetic data. In addition to the WM data on which we evaluated our algorithm, we also tested it on randomly generated graphs. The random process by which the graphs were generated was controlled by four parameters: n, the number of vertices; p, the edge creation probability; m, the total number of colors; and k, the maximum color-set size. Each random graph was generated as follows (all random choices were made independently and uniformly). A set of n vertices was created. For each (unordered) pair of vertices, an edge was added between them with probability p. Then, for every vertex v a random number k_v was chosen in the range $1, 2, \ldots, k$, and then k_v colors were randomly chosen from the common pool of m colors. The values we used for the parameters were: n ranged from 20 to 100 at increments of 10; p was either 0.1, 0.3, or 0.6; $m = 300$ throughout; and k was either 10 or 20. For each set of parameters we generated ten graphs and ran the algorithm on them. In total, we ran the algorithm on 540 instances.

Results. We carried out two types of performance measurement, both on the same data sets. The first type is the absolute performance of a single call to the `SomeDifferent` propagator. The second type of measurement was comparative: we compared the running time of a CSP solver on two models, one consisting of a single `SomeDifferent` constraint, and the second consisting of the equivalent set of `NotEqual` constraints. In addition to the running time measurements we also collected various statistics concerning the input instances and the algorithm's run-time behavior. This extra data helped us understand the variation in the algorithm's performance. We used a CSP solver implementing MAC-3 and breaking arc-consistency by instantiating a random variable with a random value [12].

Figure 1 shows the running times of the propagator on the WM data instances. We see that the running time follows a slowly increasing curve from about 4msec (for 20 vertices) to about 100msec (for 300 vertices), but in the range 140–240 vertices, roughly half of the instances have increased running times, up to a factor of about 3 relative to the curve. Examination of these instances revealed that they entailed significantly more chromatic number computations than the other instances, which explains the increased running times.

We remark that up to 110 vertices nearly all instances were satisfiable, starting at 120 the percentage of unsatisfiable instances increased rapidly, and from 150 onward nearly all instances were unsatisfiable. It is interesting to note that the running times do not display a markedly different behavior in these different ranges.

In summary, despite the algorithm's worst case exponential complexity, it seems to perform well on the real-life instances we tested. The main contributing factor to this is the fact that the graphs decomposed into relatively small

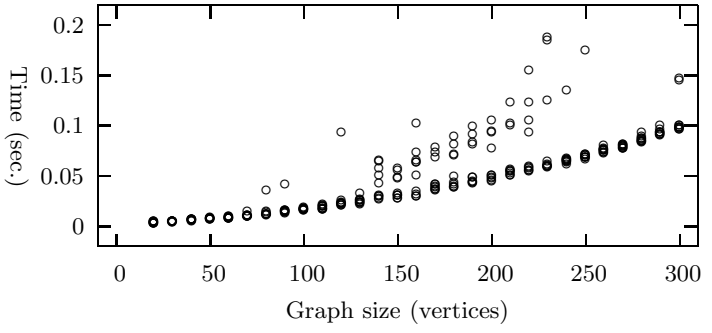

Fig. 1. Running time of the SomeDifferent propagator on WM data instances. Some of the instances shown were satisfiable, while others were unsatisfiable.

connected components. An earlier version that did not implement the *connected components* heuristic performed less well by several orders of magnitude.

Table 1. Running time (msec.) of the propagator on randomly generated graphs

	$n = 20$	$n = 30$	$n = 40$	$n = 50$	$n = 60$	$n = 70$	$n = 80$	$n = 90$	$n = 100$
$k = 10$									
$p = 0.1$	3.26	3.59	4.00	4.08	6.84	4.40	7.63	12.4	18.3
$p = 0.3$	3.27	3.64	4.93	9.39	345	13037	170416	471631	606915
$p = 0.6$	3.26	5.40	42.8	625	7065	121147	574879	607262	608733
$k = 20$									
$p = 0.1$	3.30	4.23	4.14	8.06	28.5	90.7	591	3954	13939
$p = 0.3$	3.31	5.27	12.5	42.1	167	1442	5453	29478	186453
$p = 0.6$	3.92	6.20	14.2	57.8	309	1398	6549	38196	270353

Table 1 shows the running times, in milliseconds, on the randomly generated graphs, all of which were satisfiable. The time shown in each table entry is the average over the ten instances. We see that for small graphs the algorithm's performance is similar to its performance on the WM data instances. However, as the graphs become larger, the running time increases steeply. The reason for this is the well known phenomenon whereby graphs generated by the random process we have described contain with high probability a "giant" connected component (i.e., one that comprises nearly all vertices), rendering the *connected components* heuristic powerless. All the same, the algorithm was able to handle graphs containing 100 vertices in approximately 10 minutes—well below what its worst case complexity would suggest.

We also see a significant difference between the cases $p = 0.1$ and $p \in \{0.3, 0.6\}$. At $p = 0.1$, the *giant component* effect had not set in in full force and the connected components were fairly small, whereas at $p \in \{0.3, 0.6\}$ the phenomenon was manifest and the giant component accounted for more than 90% of the graph.

There is also a noticeable difference between the cases $k = 10$ and $k = 20$. There are two contradictory effects at play here. A small value of k increases the likelihood of edges becoming superfluous due to their endpoints having disjoint color sets, thus mitigating the *giant component* effect. But at the same time, it reduces the color set sizes, thus diminishing the effectiveness of the *enumeration order* heuristic and forcing the algorithm to spend more time enumerating sets. Indeed, we have observed that in large connected components the algorithm spends most of its time enumerating sets that do not pass the $|D(U)| \leq |U|$ test; the chromatic number computations tend to be few and involve small subgraphs, and therefore do not contribute much to the running time. For $p = 0.1$ the graph is sparse enough so that the *giant component* effect is indeed canceled out by the emergence of superfluous edges, whereas for $p \in \{0.3, 0.6\}$, the small decrease in component size due to superfluous edges is outweighed by the effect of the smaller color sets.

We now turn to a comparison of the `SomeDifferent` and `NotEqual` models. We measured the time it took the solver to solve the CSP (or prove it unsatisfiable). When using `NotEqual` constraints, the backtrack limit was set to 10000. (The `SomeDifferent` model is backtrack free.)

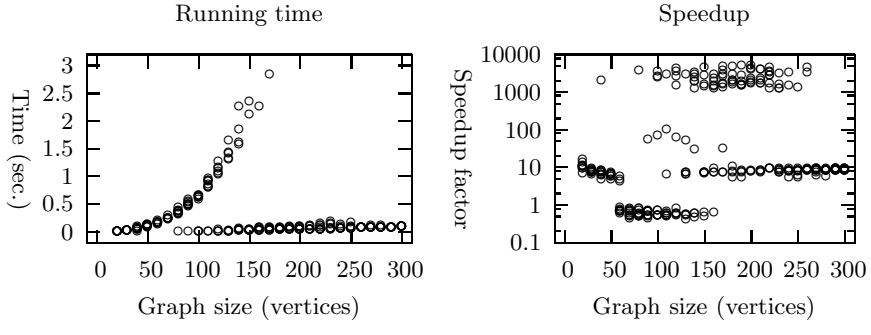

Fig. 2. Running time of the CSP solver on the `SomeDifferent` model, and speedup factor relative to `NotEqual`, on WM data instances

The graph on the left of Figure 2 shows the running times on the WM data instances using the `SomeDifferent` model. The graph on the right shows (in log-scale) the corresponding speedup factors relative to the `NotEqual` model. We see that the running times follow two curves. The rising curve represents the satisfiable instances, where the `SomeDifferent` propagator is called multiple times. The flat curve represents the unsatisfiable instances, where the unsatisfiability is established in a single `SomeDifferent` propagation. The speedup factors are grouped (on the logarithmic scale) roughly into three bands. The top band (consisting of 75 instances with speedup factors around 5000) represents instances that caused the solver to hit the backtrack limit. The middle band represents a mixture of 104 large instances identified as unsatisfiable and 42 small instances which the solver was able to satisfy, as well as 6 satisfiable instances on which

the solver hit the backtrack limit. The lower band represents the remaining 63 instances, which are of intermediate size (60–160 vertices), and which were satisfied by the solver with a very small number of backtracks (nearly always 0). For these instances the NotEqual model was superior to the SomeDifferent model; the slowdown factors were around 2 (the maximum slowdown factor was 2.35).

Upon a closer examination of the data we observed that in nearly all cases (in all three bands) the NotEqual model either hit the backtrack limit or solved the CSP with no backtracks at all. For most cases in which the solver proved unsatisfiability, this was a result of the instances containing several vertices with singleton color sets which immediately forced inconsistency. The satisfiable cases had singleton color sets too, which definitely helped the solver. In contrast, our SomeDifferent propagator does not benefit from singleton color sets in any obvious manner. Adding a preprocessing stage (essentially, a NotEqual propagation stage) to handle them can obviously improve its performance.

In summary, our WM data instances seem biased in favor of the NotEqual model, due to the presence of singleton color sets. Nonetheless, the SomeDifferent approach still outperformed the NotEqual approach in the cases requiring backtracking. In virtually all of these cases, the NotEqual model hit the limit, whereas the SomeDifferent model performed quite well. Adding a preprocessing stage to our algorithm would make it as good as NotEqual propagation in the easy cases too.

On the random graph instances the variations in the speedup factors did not display easily understandable patterns. Generally speaking, the speedup factor started in the range 40–100 for 20 vertices, and declined roughly linearly to 1 at 40 or 50 vertices (depending on the k and p) parameters. For larger instances the speedup became slowdown, which grew to a maximum in the range 4000–6000 for the largest instances. Here too, all instances were satisfied in the NotEqual model with no backtracks. This demonstrates again that the main source of hardness for our algorithm, primarily the graph's connectivity, has little effect on the NotEqual approach.

6 Conclusion and Future Work

We have introduced the SomeDifferent constraint, which generalizes the familiar AllDifferent constraint in that it requires that only *some* pairs of variables take on different values. We have developed a hyper-arc consistency propagation algorithm for the SomeDifferent constraint. The problem is NP hard, and our algorithm's worst-case time complexity is exponential in the number of variables, but is largely independent of the domain sizes. We have also suggested and implemented some heuristics to improve the algorithm's running time in practice. We have offered empirical evidence suggesting that our algorithm is indeed practical.

Future work. The results we have obtained are encouraging, but clearly more experimentation with real data, preferably from a variety of application domains, is required in order to establish the usefulness of our approach. Also, we have

already identified some areas for improvement, namely, adding a preprocessing stage to handle singleton color sets, and improving the set enumeration algorithm. A different, and important, research direction is to devise approximation techniques for unsatisfiable instances. A natural problem with practical importance is to remove as small (or cheap) a subset of vertices as possible in order to make the instance satisfiable.

Acknowledgments. We are grateful to Dan Connors and Donna Gresh for analyzing the raw workforce management data, extracting the parts pertinent to our work, and bringing them into a form we could use in our experiments.

References

1. G. Appa, D. Magos, and I. Mourtos. On the system of two all_different predicates. *Info. Process. Letters*, 94(3):99–105, 2005.
2. N. Barnier and P. Brisset. Graph coloring for air traffic flow management. In *Proc. of 4th CP-AI-OR workshop*, pages 133–147, 2002.
3. R. Beigel and D. Eppstein. 3-coloring in time $O(1.3289^n)$. *J. of Algorithms*, 54(2):168–204, 2005.
4. H. L. Bodlaender and D. Kratsch. An exact algorithm for graph coloring with polynomial memory. Technical Report UU-CS-2006-015. Department of Information and Computing Sciences. Utrecht University, 2006.
5. D. Brelaz. New methods to color the vertices of a graph. *Communication of ACM*, 22(4):251–256, 1979.
6. J. M. Byskov. Exact algorithms for graph colouring and exact satisfiability. PhD thesis, University of Aarhus, Aarhus, Denmark. 2005.
7. R. Dechter. *Constraint Processing.* Morgan Kaufmann Publishers, San Francisco, 2003.
8. D. Eppstein. Small maximal independent sets and faster excat graph coloring. *J. Graph Algorithms and Applications*, 7(2):131–140, 2003.
9. M. Grönkvist. A constraint programming model for tail assignment. In *Proc. of 1st CP-AI-OR Conf.*, pages 142–156, 2004.
10. W. Klotz. Graph coloring algorithms. Technical Report Mathematik-Bericht 2002/5 TU Clausthal, 2002.
11. E. L. Lawler. A note on the complexity of the chromatic number problem. *Info. Process. Letters*, 5(3):66–67, 1976.
12. Y. Naveh and R. Emek. Random stimuli generation for functional hardware verification as a cp application. In Peter van Beek, editor, *CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 882–882. Springer, 2005.
13. C.-G. Quimper, P. van Beek, A. López-Ortiz, A. Golynski, and S. B. Sadjad. An efficient bounds consistency algorithm for global cardinality constraint. In *Proc. of Principles and Practice of Constraint Programming (CP)*, pages 600–614, 2003.
14. J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. Technical Report R.R.LIRMM 93-068, 1993. Conf. version at AAAI-94.
15. M. Sellmann. Approximated consistency for knapsack constraints. In *Proc. of Principles and Practice of Constraint Programming (CP)*, pages 679–693, 2003.
16. M. Sellmann. Cost-based filtering for shorter path constraints. In *Proc. of Principles and Practice of Constraint Programming (CP)*, pages 694–708, 2003.

17. M. Sellmann. Approximated consistency for the automatic recording problem. In *Proc. of Principles and Practice of Constraint Programming (CP)*, pages 822–826, 2005.
18. E. Tsang, J. Ford, P. Mills, R. Bradwell, R. Williams, and P. Scott. ZDC-rostering: A personnel scheduling system based on constraint programming. Technical Report 406, University of Essex, Colchester, UK, 2004.
19. W.-J. van Hoeve. The Alldifferent constraint: a systematic overview. 2006. Under construction. *http://www.cs.cornell.edu/∼vanhoeve*.
20. D. B. West. *Introduction to graph theory*. Prentice Hall, 2000.
21. R. Yang. Solving a workforce management problem with constraint programming. In *The 2nd International Conference on the Practical Application of Constraint Technology*, pages 373–387. The Practical Application Company Ltd, 1996.