A General-Purpose Hill-Climbing Method for Order Independent Minimum Grouping Problems: A Case Study in Graph Colouring and Bin Packing

Rhyd Lewis[*]
Quantitative Methods Research Group, Cardiff Business School,
Prifysgol Caerdydd/Cardiff University, Cardiff CF10 3EU,
WALES.
tel: 0044 (0)2920 875559; fax: 0044 (0)2920 874419
email: `lewisR9@cf.ac.uk`

September 19, 2008

Abstract

A class of problems referred to as *Order Independent Minimum Grouping Problems* is examined and an intuitive hill-climbing method for solving such problems is proposed. Example applications of this generic method are made to two well-known problems belonging to this class: graph colouring and bin packing. Using a wide variety of different problem instance-types, these algorithms are compared to two other generic methods for this problem type: the iterated greedy algorithm and the grouping genetic algorithm. The results of these comparisons indicate that the presented applications of the hill-climbing approach are able to significantly outperform these algorithms in many cases. A number of reasons for these characteristics are given in the presented analyses.

Keywords: Metaheuristics; Packing; Graph Colouring; Grouping Problems.

1 Introduction

Problems that require the grouping or partitioning of a set of entities arise regularly in computer science and operations research in areas as diverse as timetabling and scheduling [4, 36, 52], packing [39, 50], load balancing [20], frequency assignment [55], in addition to various theoretical and mathematical problems such as graph colouring [30] and partitioning [56]. In

[*]The author would like to give thanks to Bruce Curry, Peter Morgan, and Christine Mumford of Cardiff University, and also two anonymous referees for their helpful comments on earlier versions of this work.

simple terms, grouping problems involve a collection of "items" that are to be divided into a number of "groups", subject to certain constraints. Formally, given a set $\mathcal{I}$ of n distinct items, the task is to construct a set of subsets $U = \{U_1, \ldots, U_G\}$ such that the following three criteria are met:

$$\cup U_i = \mathcal{I} \text{ (for } i = 1, \ldots, G) \tag{1}$$

$$U_i \cap U_j = \emptyset \text{ (for } 1 \leq i \neq j \leq G) \tag{2}$$

$$U_i \neq \emptyset \text{ (for } i = 1, \ldots, G) \tag{3}$$

Though simple in definition, grouping problems are often quite challenging in practice because the imposed constraints might be quite difficult to satisfy. Indeed, many grouping problems, including all of the examples mentioned in this paper, belong to the set of NP-complete problems, implying that we cannot hope to find a polynomially bounded algorithm for solving them in general [26]. Given this latter characteristic it is therefore appropriate to consider heuristic- and metaheuristic-based approximation algorithms for tackling these problems, particularly for those instances where exact algorithms are simply too inefficient for practical purposes.

A useful way of categorising grouping problems is to consider the number G of groups that are used to contain the items. In some cases, such as the bin balancing (equal piles) problem [20] or the k-way graph partitioning problem [56], G will be stated explicitly as part of the problem, and any candidate solution seen to be using more or less than this will be considered illegal. In other cases however, G will not be given, and the aim will be to produce a solution that *minimises* G, or a least gets G below some given bound, while still obeying the constraints of the problem. This latter class of problem can be referred to as *Minimum Grouping Problems (MGPs).*

Within the class of MGPs, two further types of problem can be identified. The first is the *Order Dependent Minimum Grouping Problem.* Here, as well as seeking to minimise the number of groups, the quality and/or validity of a candidate solution is influenced by the way in which the groups are ordered. An example of this is the exam timetabling problem encountered at universities, where the task is to group a set of exams into a certain number of "timeslots" (below some imposed bound) whilst not forcing any student to be in two places at once, and also paying heed to the fact that students do not want to sit exams in successive timeslots [4, 52]. The frequency assignment problem is another example of this sort of problem [55]. Naturally, the other problem type occurring in the class of MGPs is the Order *Independent* Minimum Grouping Problem (OIMGP). In these problems the ordering of the groups is *not* relevant, and it is these problems that form the basis of this study.

Examples of OIMGPs include the following:

The 1D Bin Packing Problem Given n items of various "sizes" and an infinite number of "bins" (each with a fixed finite capacity C), group the items into a minimal number of bins so that no bin is over-filled. [22, 26]

The Graph Colouring Problem Given an undirected graph with n nodes, group the nodes into a minimum number of "colour classes" so that no pair of adjacent nodes is assigned to the same colour. (Here the "items" are the nodes.) [30, 41]

Various Timetabling Problems E.g: Given n events that are attended by various subsets of students within an educational institution, and given a set of rooms, each with a specific seating capacity; group the events into "timeslots" so that (a) no student is required to attend more than one event in the same timeslot, (b) the events in each timeslot can each be allocated a room with sufficient seating capacity, and (c) the number of timeslots being used is less than or equal to the maximum number allowed by the institution. (Here the "items" are the events.) [36, 51]

The Edge Colouring Problem Given an undirected graph with n edges, group all of the edges into a minimum number of "colours" so that no pair of edges sharing a common node are assigned the same colour. (Here the "items" are the edges.) [26, 32]

The Cutting Stock Problem Given an unlimited number of "stocks" of a fixed finite length, and given n requests for items of different lengths to be cut from these stocks, minimise the number of stocks that will be needed in order to produce all of the requested items. [34].

In this paper, the term "feasible" is used exclusively to describe any candidate solution (or group) in which no problem-specific constraints are being violated. The symbol χ is also used to denote the minimum number of groups needed to feasibly contain the n items of a given problem. (For graph colouring, χ is known as the *chromatic number*). An *optimal* solution for an OIMGP is thus any solution that uses χ feasible groups to contain the n items.

Although the above listed problems are usually considered to belong to other problem-classes (such as the class of packing problems, the class of graph-theoretic problems, etc.), it is useful to consider the OIMGP classification in order to highlight the common features that they exhibit. Note also that in all of the above cases the task of calculating χ is considered to be NP-hard [26]. In addition, all of the corresponding decision problems are NP-complete.[1] In view of such similarities, it is thus appropriate to consider whether the shared features of these problems can be exploited via common methods. It is also useful to ask whether techniques proposed for one particular problem might also turn out to be suitable for other problems belonging to this domain.

Considering existing methods for OIMGPs, a large number of algorithms have previously been proposed in the literature, including various constructive algorithms [3, 33, 44] and exact algorithms [4, 39, 40, 45, 49]. For larger and/or more difficult problem instances where such methods are not considered efficient, a number of heuristic and metaheuristic methods have also been proposed, including various applications of evolutionary algorithms [15, 17, 18, 19, 22, 25, 32, 53], iterated local search [7, 43], simulated annealing [5, 41], tabu-search [29], variable neighbourhood search [1], hyper-heuristics [46] and ant colony optimisation [9, 16, 34]. Considering heuristic and metaheuristic methods, we have observed

[1] I.e. For a given positive integer k, is there a feasible solution that uses $\leq k$ groups?

that the majority of these actually follow one of two basic schemes, depending to some extent on whether the decision or optimisation variants of the problem are being considered:

Decision Variant First a target number of groups k is proposed and each of the items is assigned to one of these groups using some heuristics, or possibly randomly. Items that cannot be feasibly assigned to any of the k groups are then left to one side, and/or are assigned to one of the k groups anyway. The aim of the algorithm is to then try and reduce the number of unplaced items (and/or constraint violations) down to zero. Typically, the algorithm will alter k during execution if it is deemed necessary.

Optimisation Variant The number of groups to be used in a candidate solution is not specified in advance. Instead, additional groups are created whenever an item is encountered that cannot be feasibly assigned to an existing group. The number of groups is thus variable, and the aim of the algorithm is to try and reduce this number as much as possible, ensuring that no constraint violations occur in the process.

In purely practical terms, the scheme that turns out to be the most suitable will usually be influenced by the requirements of the end-user: for example, when attempting to timetable courses at a university, an upper bound on the number of available groups (timeslots) might be specified in advance, and it might be necessary to allow some of the items (courses) to remain unplaced or to allow some of the imposed constraints to be broken (see, for example, the work of Corne *et al.* [8] and Paechter *et al.* [42]). Meanwhile, in other situations it might not be appropriate to leave items unplaced (or break constraints) or there may simply be more flexibility as to the number of groups that can be used, suggesting that the optimisation variant might be more suitable. Another issue with the decision variant is that a suitable value for k needs to be defined. If χ is known in advance then we could simply set $k = \chi$. However, if χ is unknown, and a value $k > \chi$ is chosen, then an optimal grouping cannot be achieved unless k is subsequently reduced. Also, if $k < \chi$ then it is obvious that the algorithm will not be able to feasibly place all of the items (or eliminate all of the constraint violations). On the other hand, algorithms of the second type, by their very nature, will always guarantee a feasible solution. In addition, some algorithms of this type, such as those by Erben [18], Culberson and Luo [13], Falkenauer [19], and Levine [34] do not require any choice of k or knowledge of χ in order to be applied.

In this paper, we outline a new method for tackling OIMGPs that also displays the characteristics of these latter algorithms. The aim is to explore the capabilities of this generic method by making applications to two well-known problems of this type: graph colouring and one-dimensional bin packing (herein referred to as simply "bin packing"). To aid this investigation we compare the performance of these applications with two other generic methods for this problem type: the Iterated Greedy (IG) algorithm of Culberson and Luo [13] and the Grouping Genetic Algorithm (GGA), originally proposed by Falkenauer [22]. We have chosen these methods in particular, as they are both well-documented and have both shown to produce good results in the past [13, 18, 21]. In addition, these algorithms are also similar in style to the hill-climbing approach presented here, as both follow the second algorithmic scheme outlined above, both are applicable to a wide variety of grouping problem, both make use of the so-called "greedy algorithm" (Section 2.1), and neither requires any knowledge of χ (or setting for k) to be applied to a problem.

```
GREEDY($\pi$, $G$)
(1)  for ($i \leftarrow 1$ to $|\pi|$)
(2)      $j \leftarrow 1$
(3)      found ← false
(4)      while (not found and $j \leq G$)
(5)          if (Group $j$ is feasible for Item $\pi[i]$)
(6)              Insert Item $\pi[i]$ into Group $j$
(7)              found ← true
(8)          else
(9)              $j \leftarrow j + 1$
(10)     if (not found)
(11)         $G \leftarrow G + 1$ /*A new group is opened*/
(12)         Insert $\pi[i]$ into group $G$
```

Figure 1: The Greedy (First-Fit) Algorithm. If π contains all n items then G is initially set to 1; else G is set to the number of groups in the current partial solution.

In the next section we will describe the hill-climbing (HC) method for OIMGPs. In general, the terms "groups" and "items" will be used in these descriptions, therefore implying universal applicability to OIMGPs (though, for clarity, when problem-specific details are considered later on, we will also use problem-specific terms, such as "colour", "node", and "bins"). In Section 3 we then give details of our application and resultant comparison using the graph colouring problem. Section 4 then contains a similar comparison for the bin packing problem. Section 5 concludes the paper and makes some general points regarding future research.

2 The Hill-Climbing (HC) Algorithm for OIMGPs

2.1 The Greedy Algorithm and Related Features

An integral part of the hill-climbing (HC) approach is the *Greedy* (or *First fit*) algorithm. This procedure – described in the pseudo-code in fig. 1 – operates by taking any arbitrary permutation π of unplaced items and then inserts these one-by-one from left to right into the lowest indexed group where such an insertion preserves feasibility (lines 2-9).[2] If there is no feasible group, then a new one is opened and the item is inserted here (lines 10-12). The greedy algorithm therefore always produces a feasible solution, but depending on the ordering of the items in π, the resultant number of groups G can vary (with a lower bound of χ).

An important characteristic of the greedy algorithm, first used in the graph colouring

[2]For example, in graph colouring, a node $\pi[i]$ will only be assigned to a particular colour class if this class does not contain a node that is adjacent to it. In bin packing, the item $\pi[i]$ will only be inserted into a particular bin if the resultant total size of all the items in the bin does not exceed the bin's capacity.

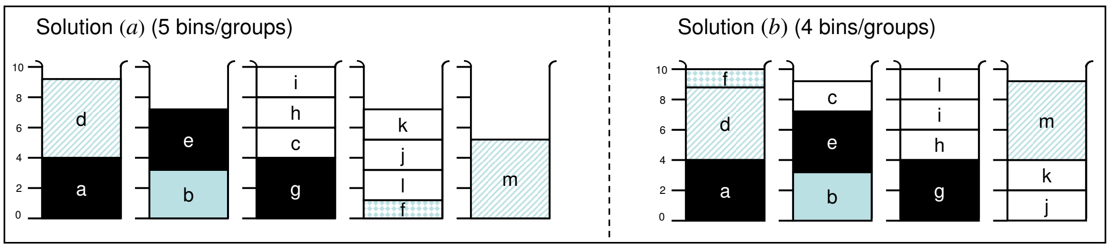

Figure 2: Two Candidate Solutions for a Bin Packing Problem. In this instance, bin capacity $C = 10$ and the sizes of items a through to m (respectively) are: 4, 3, 2, 5, 4, 1, 4, 2, 2, 2, 2, 2, and 5. Here, the right solution is using the optimal (minimum) number of groups which, in this case, is four.

algorithm of Culberson and Luo [13], is that if we take a feasible solution and use it to generate a permutation π of the items that respects the solution's groupings (by positioning all items occurring in each group into adjacent locations in the permutation), then feeding π back into the greedy algorithm will result in new solution that uses no more groups than the previous solution, *but possibly fewer*.

More generally, let us consider a candidate solution to an OIMGP represented by a set of sets $U = \{U_1, \ldots, U_G\}$, such that equations (1), (2), and (3) hold. Additionally, let each $U_i \in U$ represent a set of items constituting a feasible group according to the problem-specific constraints imposed.

Theorem 1 *If each set of items $U_i \in U$ (for $i = 1, \ldots, G$) is considered in turn, and all items $u \in U_i$ are fed one-by-one into the greedy algorithm, the resultant solution U' will be feasible, with $|U'| \leq |U|$.*

A formal proof of Theorem 1 can be found in Appendix 1 which, for the purposes of this paper, is shown to apply for both graph colouring and bin-packing. Note that the labelling of the sets $U_1, \ldots, U_G$ is strictly arbitrary for the purposes of this proof; however, the labelling can have an effect on whether $|U'| = |U|$ or $|U'| < |U|$.

A corollary immediately arising from Theorem 1 is that an optimal solution can always be represented by a permutation of the items since we can always generate such a permutation (in the manner described) *from* an optimal solution. Additionally, note that the groups within such a permutation, and the items within each group can also be permuted with no effect on this optimal property. There are thus a minimum of:

$$\chi! \prod_{i=1}^{\chi} |U_i|! \tag{4}$$

different permutations that decode into an optimal solution. (Trivially, if $\chi = 1$ or $\chi = n$ the number of permutations decoding into an optimal solution will be $n!$: that is, every permutation).

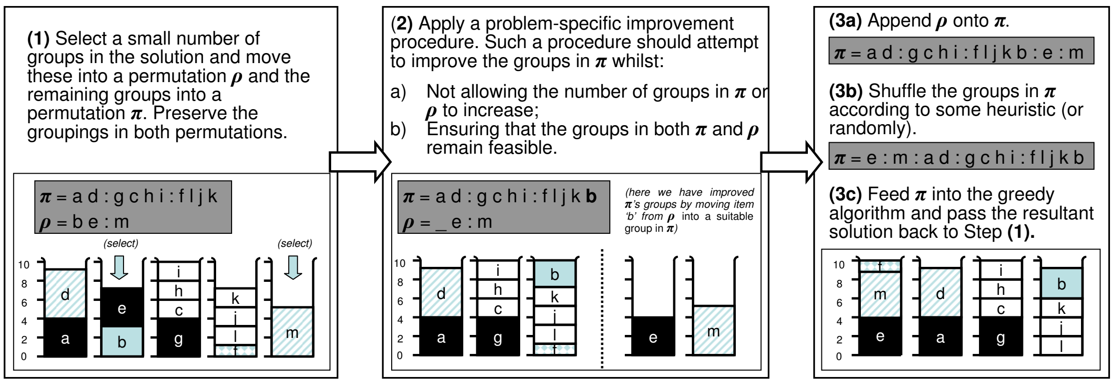
(1) Select a small number of groups in the solution and move these into a permutation ρ and the remaining groups into a permutation π. Preserve the groupings in both permutations.
π = a d : g c h i : f l j k
ρ = b e : m
(select)
(select)
10 8 6 4 2 0
d a e b i h c g k j l f m
(2) Apply a problem-specific improvement procedure. Such a procedure should attempt to improve the groups in π whilst:
a) Not allowing the number of groups in π or ρ to increase;
b) Ensuring that the groups in both π and ρ remain feasible.
π = a d : g c h i : f l j k b
ρ = _ e : m
(here we have improved π's groups by moving item 'b' from ρ into a suitable group in π)
d a i h c g b k j l f e m
(3a) Append ρ onto π.
π = a d : g c h i : f l j k b : e : m
(3b) Shuffle the groups in π according to some heuristic (or randomly).
π = e : m : a d : g c h i : f l j k b
(3c) Feed π into the greedy algorithm and pass the resultant solution back to Step (1).
f m e d a i h c g b k j l

Figure 3: High-Level Description of the Hill-Climbing Method. For clarity, an example is provided in the figure that uses the Solution 1 from fig. 2 as a basis for the starting solution.

To illustrate these concepts further, consider fig. 2(a), where the presented (feasible) bin packing solution is using five bins. One permutation that could be generated from this solution in the manner described is:

$$\pi = \mathsf{a\ d\ :\ b\ e\ :\ g\ c\ h\ i\ :\ f\ l\ j\ k\ :\ m}$$

(where the boundaries between the groups are marked by colons). An application of the greedy algorithm using this permutation now operates as follows: first, items a and d are inserted into group 1 as before. Items b and e, which cannot feasibly be inserted into group 1, are then put into group 2. Similarly, a third group is then opened for item g, which is too big for both groups 1 and 2. At this point, note that the greedy algorithm now inserts item c into group 2, rather than group 3 where it was previously assigned. Group 2 thus contains an additional item, and extra space has now been created in group 3 which could be used to accommodate items contained in the remainder of the permutation. Indeed, continuing the greedy process, we see that items l and m are also inserted into earlier groups, eventually resulting in the solution given in fig. 2(b), which uses one group fewer than the solution from which it was generated.

2.2 High-level Description of the Hill-Climbing Method

Having reviewed the relevant characteristics of the greedy algorithm, we are now in a position to describe the overall HC approach for OIMGPs. The main steps of this method are outlined in fig. 3 where, for demonstrative purposes, the bin packing problem from fig. 2 is also included.

As is shown, given a feasible solution, a small number of groups are first selected.[3] The items within the selected groups are then moved into a placeholder array ρ, with the remaining items being put into a second array π. Note that the orderings implied by the groups are maintained in both ρ and π. In Step 2 an improvement scheme is then applied.

[3]Methods for producing an *initial* feasible solution will be discussed in Sections 3 and 4.

Note that the description of this step is quite vague in the figure due to the fact that problem-specific operators will usually be needed here. However, the aim of the improvement scheme is universal – to make a series of local moves in an attempt to improve the quality of the groups in π, whilst (a) ensuring that the number of groups in π and ρ does not increase, and (b) ensuring that the groups in both permutations remain feasible at all times. For instance, in the bin packing example, we can see that the third group in π has been "improved" via the insertion of item b into the third bin. Finally in Step 3, a full permutation of the items is formed by appending ρ onto π. If desired, the groups within this full permutation can then be reordered, and the resultant permutation is then fed into the greedy algorithm to produce a new solution. This is then passed back to Step 1, whereupon the algorithm can continue as before.

It should be apparent from this description that the HC approach has the opportunity of reducing the number of groups in two different ways. The first occurs in Step 2 when, through a series of local moves, a group in ρ is emptied by transferring all of its items into existing groups in π. The second opportunity occurs in Step 3 when a permutation is formed that, when fed into the greedy algorithm, results in a new solution that uses fewer groups than its predecessor. Note that our exploitation of the underlying OIMGP structure in these operators means that the number of groups cannot increase, thus providing the method's hill-climbing characteristics. Of course, this latter feature raises the issue – one that is often directed at hill-climbing algorithms – that the search will be susceptible to getting caught in local minima. It is for this reason that we advocate the use of Step 3, which will often make large "global" changes to a candidate solution, helping the search to escape from the attraction areas of local minima. The benefits of such "macro-perturbation" operators have been touted by many authors (see [14] and [38]) and can be seen as acting as a compliment to local-based search operators. Note however that in contrast to many other macro-perturbation operators [14, 21], due to the structure of OIMGPs we have the potential to make large changes to a candidate solution without having to make compromises on solution quality in the process.

3 Application to the Graph Colouring Problem

In this section we propose an application of the HC method to the graph colouring problem and analyse its performance over a large number of different problem instances (in all cases, these problems were produced using the problem generator of Joe Culberson [12]). We begin in the next subsection by describing the various problem-specific details that were used in the hill-climbing algorithm; Section 3.2 then contains information on the algorithms used in our analysis for comparative purposes. Section 3.3 then discusses some relevant issues regarding how a fair comparison of the algorithms was achieved, and finally, Sections 3.4 and 3.5 contain the results for "Random" graphs and "k-colourable" graphs respectively.

3.1 A Hill-Climbing Algorithm for Graph Colouring

To produce an initial solution, our implementation of the HC method uses the *Dsatur* algorithm of Brelaz [3]. This algorithm is similar in fashion to the greedy algorithm (fig. 1)

```
GCol-Improvement-Procedure($\pi$, $\rho$, $I$)
(1)  $i \leftarrow 0$
(2)  while ($\rho \neq \emptyset$ and $i \leq I$)
(3)     for ($j \leftarrow 1$ to $|\rho|$)
(4)        for ($g \leftarrow 1$ to $G(\pi)$)
(5)           if (Group $g$ is feasible for item $\rho[j]$)
(6)              Move item $\rho[j]$ into group $g$ in $\pi$
(7)     if ($\rho \neq \emptyset$)
(8)        repeat
(9)           $doneChange$ = false
(10)          Alter $\pi$ using $N_1$ or $N_2$
(11)          if (the groups in $\pi$ remain feasible)
(12)             $doneChange \leftarrow$ true
(13)          else
(14)             Reset the change made in Step (10)
(15)          $i \leftarrow i + 1$
(16)       until ($i \geq I$ or $doneChange = true$)
```

Figure 4: The improvement procedure used in the graph colouring application. Here, $G(\pi)$ represents the number of groups that are contained in the permutation π. The variable I defines the iteration limit of the procedure.

though at each iteration, the next node chosen to be coloured is determined *dynamically* by choosing the uncoloured node that currently has the largest number of distinct colours assigned to adjacent nodes (ties are then broken by choosing the node with the highest degree). We choose this particular algorithm because it is generally able to produce high-quality solutions in short amounts of time, thus allowing the algorithm to start at a relatively good part of the search space.

For Step 1 of the HC method, we also need to decide on how groups are to be chosen for transfer into ρ (Step (1a), fig. 3). In this case, we choose to consider each group in the solution in turn, and then transfer this group into ρ with a probability of $\frac{1}{G}$, where G represents the number of groups in the current solution. Note that various other problem-specific methods could also be used here, such as biasing selection towards groups that fulfil certain characteristics (such as containing a small number of items etc.), though such possibilities are beyond the scope of this paper.

The improvement procedure used in this application is defined in fig. 4. In the first part of the procedure (lines 3-6) attempts are made, in a deterministic manner, to transfer items from ρ into any group in π that can feasibly contain them. Thus, at this point the procedure is trying to (a) improve the groups in π by increasing the number of items contained within them, and (b) eliminate existing groups in ρ. The remainder of the procedure (i.e. lines 8-16) complements these actions by feasibly moving items *between* the various groups in π. Obviously such actions, if performed, will alter the makeups of the groups in π and raise the possibility of further items being transferred from ρ into π when we loop back to line 3. This procedure then continues until an iteration limit I is met. Note that two neighbourhood operators, N_1 and N_2, are used for moving the items between groups in π, which operate as follows:

N_1: **Swap Operator** Select two groups g_1 and g_2 in π such that $g_1 \neq g_2$. Next, randomly select one item i_1 in g_1, and one item i_2 in g_2, and swap their positions (that is, move i_1 into g_2 and move i_2 into g_1).

N_2: **Move Operator** Select two groups g_1 and g_2 in π such that $g_1 \neq g_2$. Next, randomly select one item in g_1, and move it into group g_2.

Operator N_2 changes the number of items per group, and thus has the potential to eliminate groups. N_1 is unable to do this, though it is still able to alter the make-up of groups. In our experiments in each iteration a choice was made between N_1 and N_2 with equal probability. Note that neither N_1 nor N_2 ensures that feasibility of the groups is maintained, meaning that some applications will need to be reset. For our purposes, this feature was compensated for by setting the iteration limit of the procedure I to be quite high. However, as these operators are relatively cheap to perform, this did not seem to be restrictive in practice.

Finally for Step 3b of the hill climbing method, groups were rearranged in the permutation π using the following three heuristics: (a) *Largest First* (where groups are arranged in order of decreasing size); (b) *Reverse Ordering* (where groups are arranged in the reverse order of their labelling in the previous solution); and (c) *Random Ordering.* A ratio of 5:5:3 (respectively) was used here in accordance to the recommendations of Culberson and Luo [13].

3.2 Iterated Greedy and GGA Setup

The first algorithm used for comparison is the Iterated Greedy (IG) algorithm, which was first introduced by Culberson and Luo for graph colouring in [13]. This algorithm operates by first producing an initial feasible solution. At each iteration, a permutation is constructed, respecting group boundaries as before, which is then fed into the greedy algorithm to produce a new solution. This step is then repeated until some sort of stopping criteria is met. For the purposes of comparison, in our implementation we again used the Dsatur algorithm to produce the initial solution, and used the same ratio of group ordering heuristics as our hill climbing application.

The second algorithm considered in our comparison is the *Grouping Genetic Algorithm (GGA).* Originally proposed by Falkenauer in the 1990s [19, 22], the GGA can be seen as a type of evolutionary algorithm that uses specialised genetic operators for working with the *groups* of a candidate solution. This follows the rationale, proposed by Falkenauer, that it is the groups of items that constitute the underlying building-blocks of grouping problems, thus genetic operators that allow the propagation of such building-blocks are appropriate when applying evolution-based search to this problem domain.

Previously, two applications of GGAs have been made to the graph colouring problem: the first due to Eiben *et al.* [17], and the second due to Erben [18]. For this comparison, we chose the latter, which claims superior performance due to the fine-grained, heuristic-based nature of it's fitness function f:

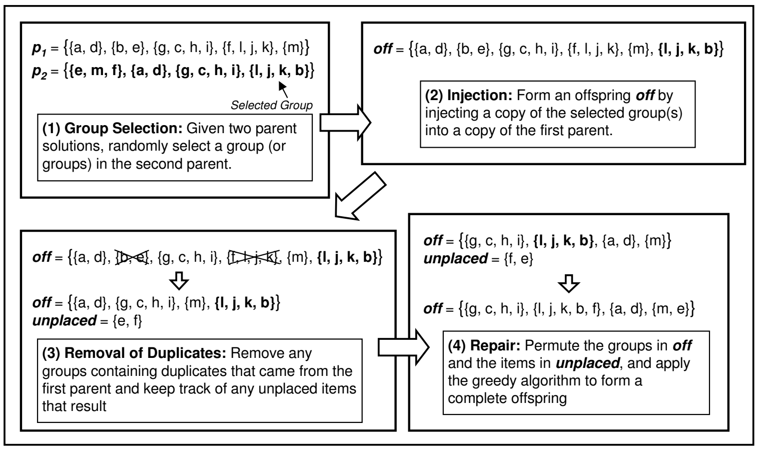
p1 = {{a, d}, {b, e}, {g, c, h, i}, {f, l, j, k}, {m}}
p2 = {{e, m, f}, {a, d}, {g, c, h, i}, {l, j, k, b}}
Selected Group
(1) Group Selection: Given two parent solutions, randomly select a group (or groups) in the second parent.
off = {{a, d}, {b, e}, {g, c, h, i}, {f, l, j, k}, {m}, {l, j, k, b}}
(2) Injection: Form an offspring off by injecting a copy of the selected group(s) into a copy of the first parent.
off = {{a, d}, ~~{b, e}~~, {g, c, h, i}, ~~{f, l, j, k}~~, {m}, {l, j, k, b}}
off = {{a, d}, {g, c, h, i}, {m}, {l, j, k, b}}
unplaced = {e, f}
(3) Removal of Duplicates: Remove any groups containing duplicates that came from the first parent and keep track of any unplaced items that result
off = {{g, c, h, i}, {l, j, k, b}, {a, d}, {m}}
unplaced = {f, e}
off = {{g, c, h, i}, {l, j, k, b, f}, {a, d}, {m, e}}
(4) Repair: Permute the groups in off and the items in unplaced, and apply the greedy algorithm to form a complete offspring

Figure 5: The GGA recombination operator. Using fig. 2 as an example, this figure shows how the first offspring is produced using two "parent" candidate solutions p_1 and p_2. To form a second offspring, the roles of the parents are reversed.

$$f = \frac{\sum_{g=1}^{G} d_g^2}{G} \tag{5}$$

(where d_g represents the total degree of all the nodes residing in group g, and G represents the number of groups being used.) Of course, in evolutionary algorithms a fine-grained fitness function such as this is important as it allows for the differentiation between different candidate solutions for a longer proportion of a run, therefore allowing selection pressure to be maintained.

Erben's GGA operates by maintaining a population of feasible individuals (i.e. candidate solutions) and then "evolves" these in order to try and reduce the number of groups (colours) being used. To start, each individual in the initial population is created by feeding a randomly generated permutation of the items into the greedy algorithm. During a run the population is then evolved using selection pressure (provided by the fitness function), recombination, and mutation. The recombination operator, described in fig. 5, uses two parent solutions to construct two new offspring solutions which are then inserted into the population. The mutation operator operates in a similar fashion by selecting some groups in a solution at random, removing them, and then repairing the solution in the same way as the recombination operator. In [18], Erben makes use of three parameters in order to evolve the population: a population size, a mutation rate, and a recombination rate. For our experiments these were set to 20, 0.01 and 0.2 respectively, which are in line with those suggested by the author. Precise details on the meaning of these parameters can be found in the original publication [18].

3.3 Comparing the Algorithms

In general, the most common measure for analysing the performance of stochastic optimisation algorithms is to use the *Success Rate*, which is the proportion of performed runs where optimality is achieved within some cut-off point [10, 17, 18, 35]. To use this measure, however, it necessary to know what an optimal solution actually is, and/or whether such an optimal solution is achievable. For OIMGPs, this information is gained by knowledge of χ which, as noted, will not always be available in reasonable time. In our case, we therefore choose to use the *number of groups* as a quality measure instead. Such a measure, as well as being appropriate when χ is unknown, is also useful when χ *is* known, as it still gives an indication of the *accuracy* of the algorithm when optimal solutions are not found.

With regards to monitoring the computational effort required by such algorithms, a common approach is to look at the *number of evaluations* performed (used, for example, in [10, 17, 18, 57]). However a problem with this measure is that it can hide the various amounts of work conducted *in-between* evaluations.[4] In addition, this measure is also inappropriate for the IG and HC algorithms, because neither needs to "evaluate" a candidate solution as such, since their operational characteristics guarantee never to produce a candidate solution that is worse than a predecessor. A better alternative for measuring computational effort in this case is to count the number of *constraint checks* that are made during execution. In graph colouring, a constraint check occurs when an algorithm requests some information about a problem instance – i.e. whether two nodes are adjacent in the graph – and is achieved using a simple look-up operation in a standard $n \times n$ Boolean conflicts matrix. According to Juhos and van Hemert [31] and also Craenen [11], constraint checks consume the largest amount of computation time in problems such as these, and therefore play "a key role in the overall performance" of an algorithm [31].

Note, however, that while using the number of constraint checks is certainly an improvement over the number of evaluations, such a measure can still hide the effects of other operations performed during an algorithm's execution. For instance, in our experiments it was observed that the GGA would sometimes take up to 12 times the CPU time required by the IG algorithm to complete a typical run of 1000,000,000 constraint checks. Similarly, the HC algorithm could sometimes take twice the CPU time of the IG algorithm. In the case of the IG algorithm, we saw that this method used conflict checks very intensively, and performed relatively few operations besides; on the other hand the GGA had to perform a significant number of other operations, including those for maintaining the population, copying candidate solutions, and so forth. Given these issues, it is also appropriate to consider CPU time in our comparison. Obviously, the benefit of this measure is that no computational effort is "hidden", thus giving a true picture of how the algorithms perform according to real world time. Note, however, that CPU time is dependent of the hardware that is being used and can also be distorted by various issues concerning the choice of programming language, platform, and compiler as well as the efficiency of the algorithms' coding.[5]

[4]For instance, in the work of Lewis and Paechter [36] it has been shown that GGAs typically require more computation to produce and evaluate candidate solutions at the beginning of a GGA run (when the population is diverse) and much less in the latter stages, when the population has started to converge.

[5]In our case we attempted to avoid potential discrepancies (as far as possible) by implementing all algorithms with the same language and compiler (C++ under Linux using g++ 4.1.1), and by using common

3.4 Comparison using Random Graphs of Unknown χ

Random graphs have n nodes, and are generated so that each of the $\frac{n(n-1)}{2}$ pairs of nodes has an edge joining them with a probability p. Generally, the chromatic number χ for such graphs will not be known – though, of course, this detail is not important here, as this information is not needed by the GGA, HC, or IG algorithms. For our comparison graphs were generated using values of p from 0.05 through to 0.95, incrementing in steps of 0.05. Because we are also interested in the scalability of these algorithms, this was done for graphs of sizes $n = 250$, 500, and 1000. To account for the stochastic nature of algorithms and the problem generator, in each case ten instances were generated for each value of p and n, and ten separate runs were then performed on each of these instances with each algorithm. The iteration limit I used with the HC algorithm was set to $50000n$. Finally, in order to allow all algorithms to start at identical parts of the search space, a second version of the GGA was also used whereby one member of the initial population was generated using the Dsatur algorithm. This GGA variant is denoted "GGA w/ Dsatur" below.

Figures 6, 7, and 8, compare the performance of the algorithms for graphs of $n = 250$, 500, and 1000 respectively. The bars of the first graph in each figure display the number of colours used by the initial solutions produced by the Dsatur algorithm, for the various values of p. The lines of the graphs (that relate to the right vertical axis) then show the percentage of this figure that were used by the best solutions achieved by the algorithms (e.g. in fig. 6(a), for $p = 0.5$ the Dsatur algorithm produces solutions using 37.1 colours on average, and the HC algorithm produces solutions using 81.6% of this figure – i.e. 30.3 colours). The cut-off point used in all of these experiments was 1000,000,000 constraint checks which was deemed sufficiently large to allow each of the algorithms' searches to stagnate (this equated to approximately 5 minutes of CPU-time with the HC algorithm). Finally, in order to give an indication as to how these algorithms perform *during* a run, figs. 6, 7, and 8 also show best-so-far plots for each algorithm (with respect to both constraint checks and CPU time) for instances of $p = 0.5$.

The first notable feature in figs. 6, 7, and 8 is that, compared to the HC algorithm, the IG algorithm tends to produce relatively poor solutions with instances where p is low. However, as p is increased, this quality then improves up to a particular point (that we will call p_{IG}), and beyond p_{IG}, it then begins to fall again. It can also be observed that p_{IG} seems to shift right as n is increased.

Another strong feature of the figures is that the HC algorithm produces the best quality solutions within the cut-off point for the vast majority of p and n-values. One exception to this is when $n = 250$ and $p < 0.4$, as the GGA variants seem to produce very similar results – perhaps because optimal solutions are being produced by all three algorithms in these cases. The results of these experiments also indicate that the HC algorithm is significantly[6] outperformed by the IG algorithm in the small number of values around p_{IG} in each case,

data-structures and code-sharing whenever it was appropriate to do so. Naturally, all timed experiments were also conducted on the same hardware using a PC under Linux with a 1.8GHtz processor and 256MB RAM.

[6]In this case, the term "significant" is used to indicate that a two-sided signed rank test rejected the null hypothesis of the algorithms' results coming from the same underlying population with a significance level of 1% or better.

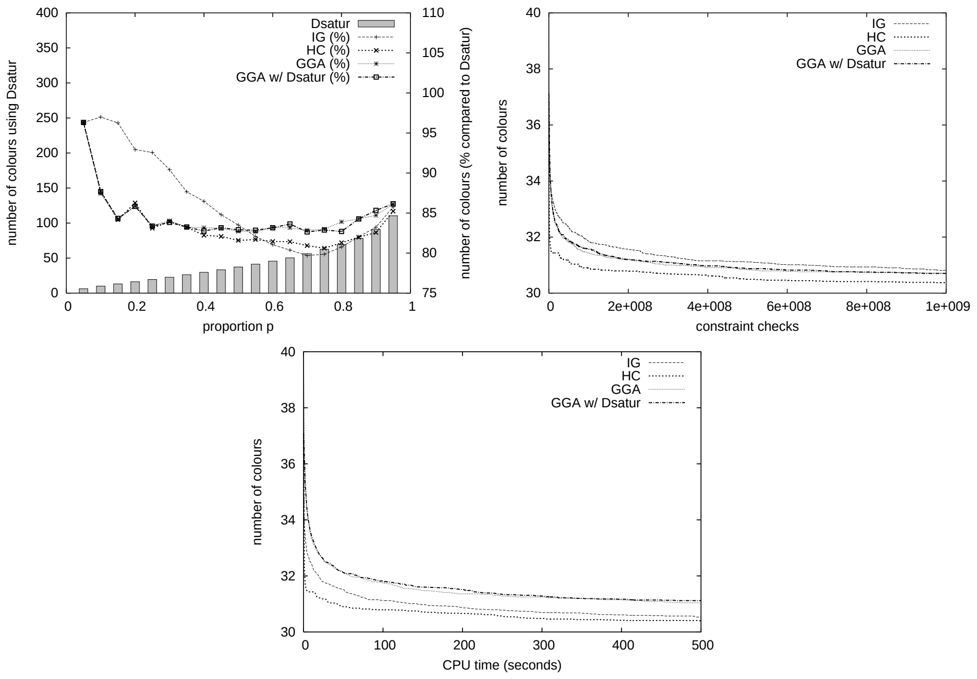

Figure 6: (Top-left): Comparison of the number of colours being used in the best solutions generated by the various algorithms within the cut-off point for graphs of $n = 250$ and varying values of p. All points in the graph are the average of ten runs on each of the ten instances generated (i.e. 100 runs). (Top-right) displays the progress of the algorithms with instances of $p = 0.5$ according to the number of constraint checks performed (each line is an average of ten runs on ten separate instances (i.e. 100 runs)); (Bottom) shows the progress of the same 100 runs with regards to CPU time.

though for all remaining values the reverse is true. It can also be seen that for high values of p (approximately $p > 0.9$) the quality of the solutions produced by the various algorithms tend to become more alike, though the results still show that the improvements offered by the IG and HC algorithms are statistically significant compared to the GGAs' for $n = 500$ and 1000.

Considering the GGA, it can be seen that for larger values of n, the difference in performance between these algorithms and the HC algorithm tends to become more stark. Indeed, the unsatisfactory performance of the GGA is particularly noticable in the best-so-far curves for $n = 1000$, where neither version makes satisfactory progress through the search space. It can also be seen in the figures that there is generally little difference in the performance of the two GGA variants, with the one exception being for $p < 0.55$ and $n = 1000$, where the solutions generated by the original GGA are actually worse than those initially produced by the Dsatur algorithm. In addition, the best-so-far curves of figs. 6 and 7, indicate that although the GGA with Dsatur starts at a better point in the search space, once the original GGA has "caught up" with this enhanced algorithm, the behaviour of the two is very similar, suggesting that it is the GGA's search operators that are responsible for lack of performance with these particular problems, and not the characteristics of the initial populations used

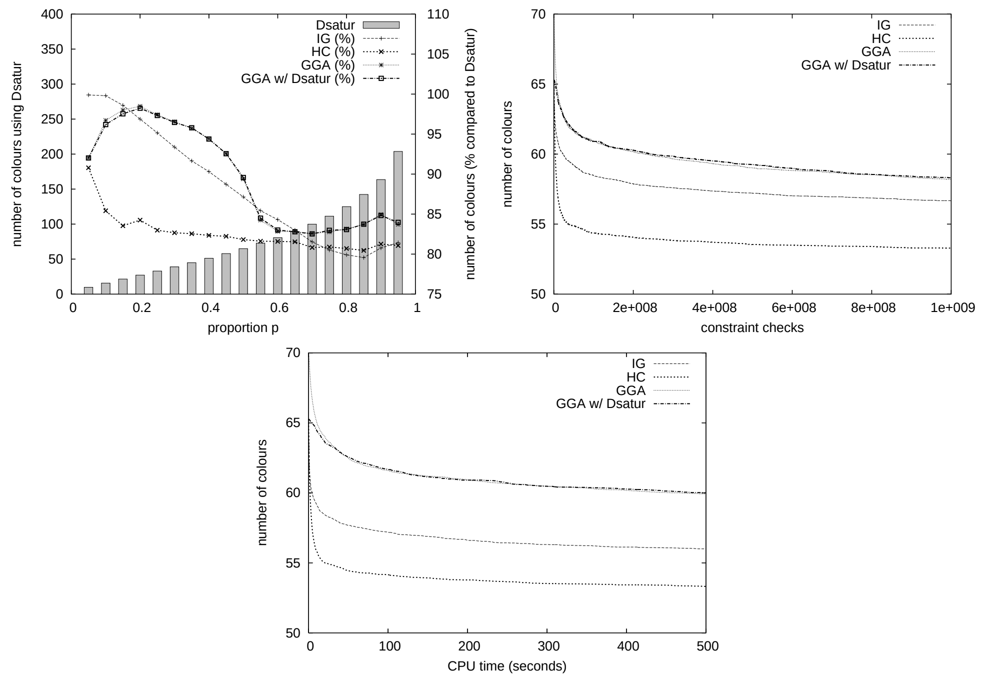

Figure 7: Comparison of results for graphs of $n = 500$. All other details are the same as those given in the caption of fig. 6.

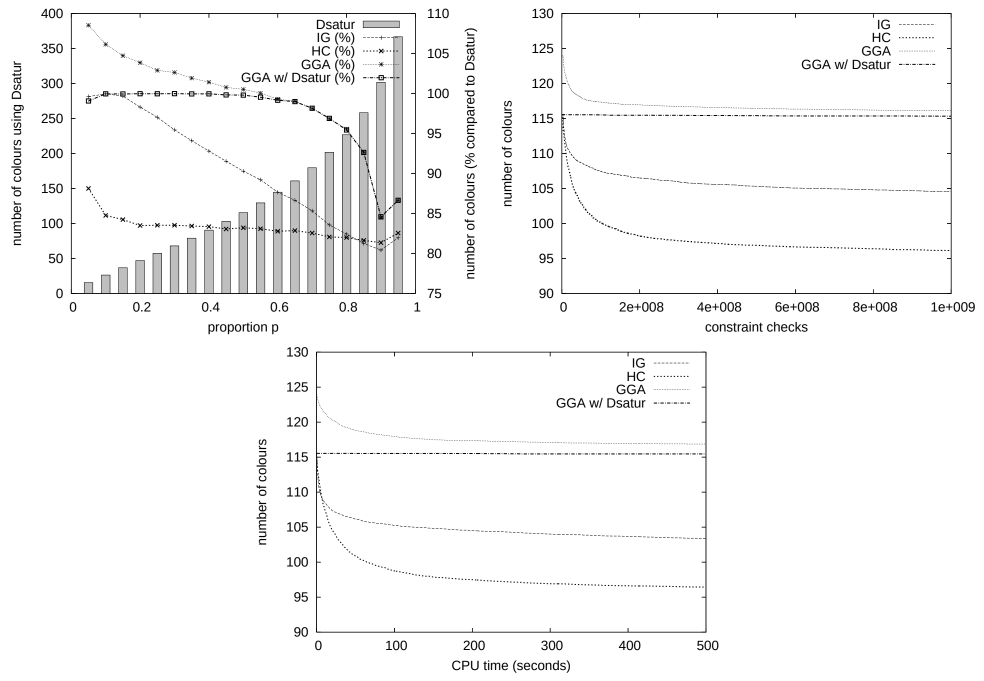

Figure 8: Comparison of results for graphs of $n = 1000$. All other details are the same as those given in the caption of fig. 6.

Table 1: Results reported in the literature for random instances with $p = 0.5$. Figures indicate the number of colours used in the solutions.

	$n = 500$			$n = 1000$		
Method	Worst	Median	Best	Worst	Median	Best
HC	54	53	52	97	96	94
IG	58	57	55	106	104	103
GGA w/ Dsatur	60	59	54	117	115	113
MAXIS + tabu search [13]	-	-	-	89	89	89
Dynamic backtracker [44]	-	-	-	107	107	107
GA + local search [15]	-	-	48[a]	-	-	83[b]
Variable Neighbourhood Search [1]	-	-	49	-	-	90
ACO + tabu search [16]	49	49	49[c]	-	-	-

[a]Approx. 5 hours of run time used
[b]Approx. 3 days of run time used
[c]Approx. 30 minutes of run time used

here.

In summary, from these experiments there is strong evidence to suggest that this HC algorithm outperforms the other two approaches on random graph colouring problems, particularly with larger instances. Another point to note is that although this algorithm generally requires more CPU time than the IG algorithm for performing a fixed number of constraint checks, according to the best-so-far curves in figs. 6, 7, and 8, the HC algorithm still shows quicker movements through the search space with regards to both measures of computational effort.

Finally, because random graphs with $p = 0.5$ for $n = 500$ and 1000 have also been considered in other works, Table 1 summarises our results with these instances and compares them to those achieved by some other well-known approaches appearing in the literature. Note that better solutions for such graphs have been produced by many of these previous methods, though it is best to apply a level of caution when viewing these figures as in most cases different experimental conditions have been used, particularly concerning run-times, the instances used, and the number of trials performed. In addition, some of the listed algorithms might be considered as being specifically designed for random graphs and/or have undergone extensive parameter tuning for each particular problem instance. The same cannot be said for the algorithms tested here, however.

3.5 Comparison using k-colourable graphs

Our second set of graph colouring experiments involves comparing the algorithms using a large number of *k-colourable* problem instances. Such instances are specially constructed so that they can always be coloured using k colours (where k is a parameter to be specified to the problem generator), and are generated by first partitioning the nodes into k distinct sets, and then adding edges according to the edge probability p between pairs of nodes in different sets.

Three of the most well-known classes of k-colourable graphs are *Arbitrary k-colourable* graphs, *Equipartite* graphs and *Flat* graphs, which differ in the following way:

Arbitrary k-colourable graphs. Here, each node is randomly assigned to one of the k sets before the edges are assigned according to p;

Equipartite graphs. Here, the nodes are partitioned into k almost equi-sized sets (the smallest set having at most one node fewer than the largest set), before the edges are assigned;

Flat graphs. Here, like equipartite graphs, the nodes are partitioned into k almost equal-sized sets, but then when assigning edges, the variance in the degrees of each node is kept to a minimum.

Culberson and Luo [13] report that equipartitie graphs are generally more difficult to colour than arbitrary k-colourable graphs, because the number of nodes in each colour in an optimal solution will be nearly equivalent, therefore giving an algorithm less heuristic information with which to work with. Following these arguments, flat graphs tend to be even more difficult, because even less heuristic information is available. Another feature of k-colourable graphs is that they are only difficult to colour optimally within a specific range of values for p, commonly referred to as the *phase transition* region [6, 54].[7] For this comparison, we generated instances for three different k-values: 5, 50, and 100, using p-values in and around the phase transition regions. In all cases $n = 500$ was used, meaning that for the equipartite and flat graphs there are exactly 100, 10, and 5 nodes per colour (respectively) in optimal solutions. In each case five instances were generated for each p- and k-value, and five separate runs were then performed on each instance with each of the algorithms. The cut-off points used were the same as previous experiments.

In choosing a setting for the iteration limit I used in the improvement procedure of the HC algorithm (fig. 4), in initial trials with these instances, we originally chose to use the same setting of $I = 50000n$ used in the previous section. However, under these conditions, we noticed that the HC algorithm would often perform quite poorly, particularly with problem instances occurring in the right-hand side of the phase transition regions. A reason for this is that k-colourable graphs within phase transition regions tend to feature an abundance of local minima [6], which can be difficult to escape when using local-based search operators. Also, graphs occurring in the right-hand side of these regions feature a greater number of edges and are therefore more constrained – consequently, a larger proportion of applications of neighbourhood operators N_1 and N_2 tended to be reset in practice (because they caused infeasibilities), causing the HC algorithm to move through the search space more slowly according to our measures of computational effort. In comparison, the IG algorithm did not seem to suffer from these problems as its search operators tend to make larger changes to a candidate solution, generally allowing it to escape from local minima more easily. For the results reported here, we therefore chose to use a lower setting of $I = 1000n$ for the HC algorithm, which seemed to give a more appropriate balance between the two types of search operator used in this method.

[7]Note: that our use of the term "optimal" is not strictly accurate here, as some instances – particularly those with low values for p – may have a chromatic number χ that is less than k.

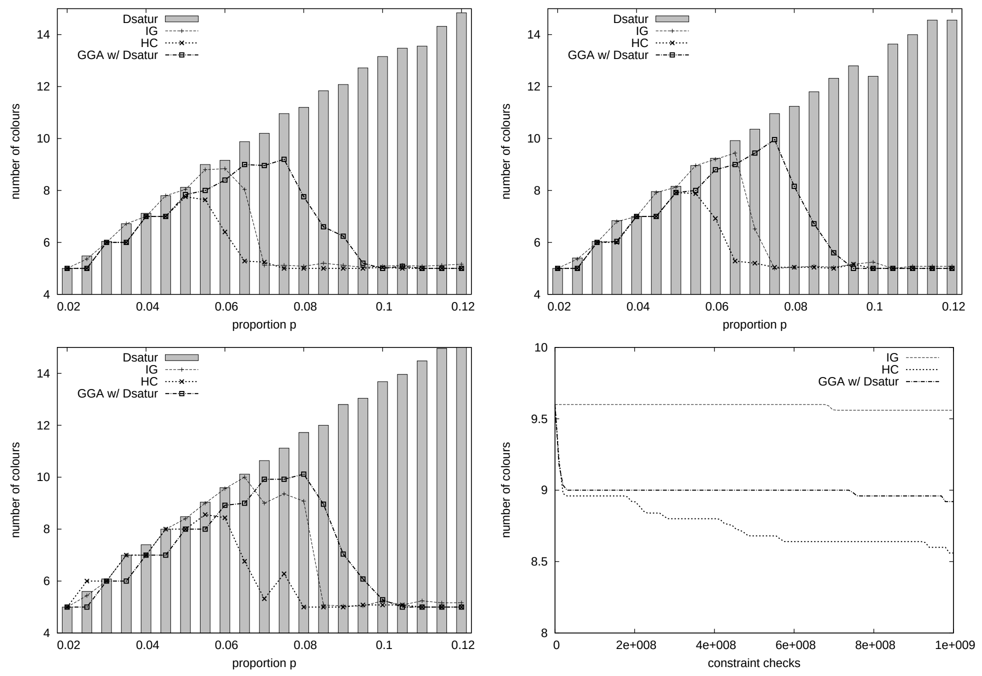

Figure 9: Comparison of the number of colours being used in the best solutions generated within the cut-off point for arbitrary (top-left), equipartite (top-right), and flat (bottom-left) 5-colourable graphs for varying values of p. (Bottom-right) shows best-so-far curves for flat graphs with the indicated values. All points are the average of five runs on each of the five instances generated at the various values of p.

.

The results of our experiments are summarised in figs. 9, 10, and 11 for k-values of 5, 50, and 100 respectively. In these figures details are again included regarding the average number of colours used in the initial solutions produced by Dsatur. Best-so-far plots are also included for a sample of 25 hard-to-colour, flat graphs (the parameters of which are indicated in the figures). Finally, note that only the GGA-version using Dsatur is included here as the original GGA never gave a better performance than this modified version.

Considering fig. 9 first, we see the phase transition curves of the HC algorithm are both narrower and flatter than the remaining algorithms', indicating that it is able to produce consistently better solutions across this region. The best-so-far graph also indicates that for "hard" flat instances, at any point in the run the HC algorithm has found a solution that is equal or superior in quality to those found by the remaining algorithms. Note that the IG algorithm shows a disappointing performance here, perhaps because the number of groups is fairly small, limiting the number of possible permutations that can be formed with each iteration of the algorithm. Note also that the phase transition curves for all algorithms tend to become wider and more pronounced in each successive graph in the figure, indicating the increased difficulty of the equipartitie and flat graphs respectively.

Moving to figs. 10 and 11, we see that for these larger values of k the IG and HC algorithms show similar performance across the phase transition regions in general. In fig. 10

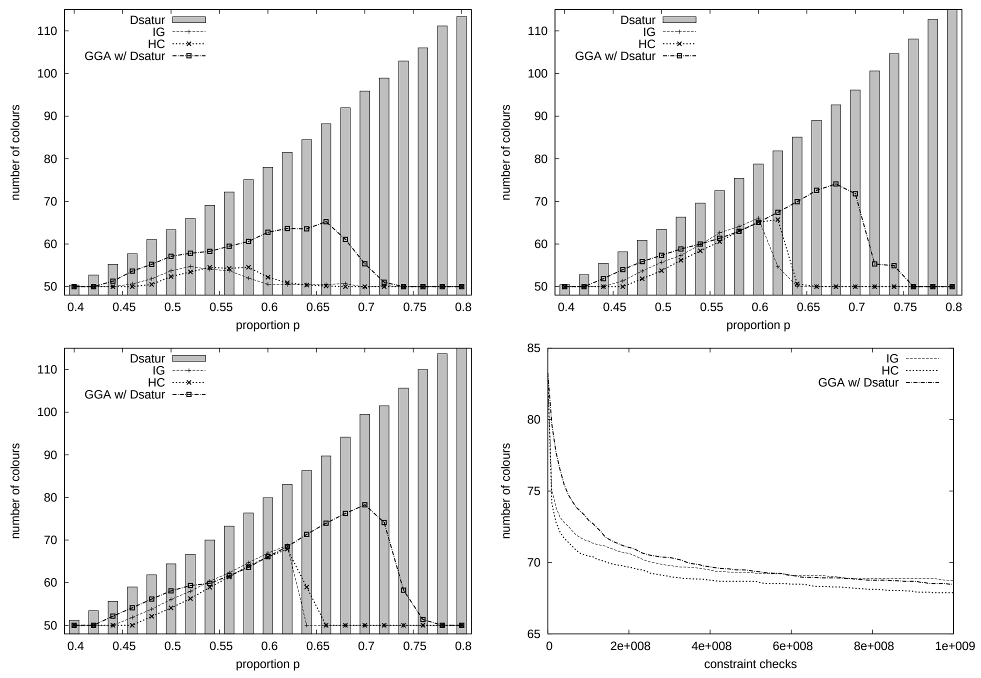

Figure 10: Comparison with 50-colourable graphs for varying values of p. All other details are the same as those given in the caption of fig. 9

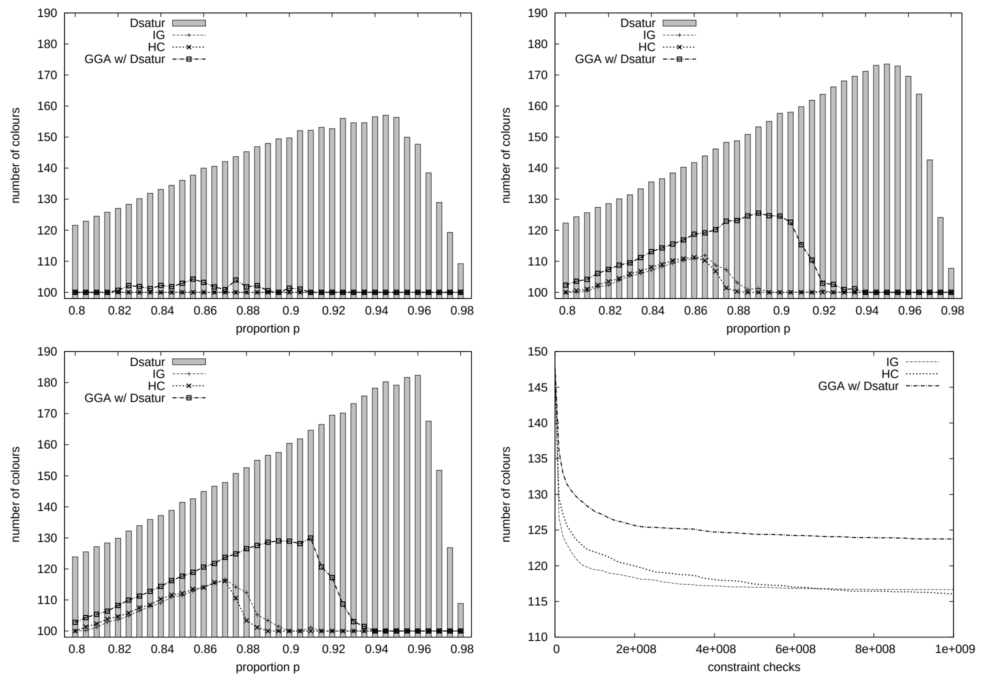

Figure 11: Comparison with 100-colourable graphs for varying values of p. All other details are the same as those given in the caption of fig. 9

the HC algorithm seems to produce slightly better solutions in the left-hand side, with the IG producing better solutions on the right; in fig. 11 this pattern seems to be reversed, though in both figures these differences are only slight. Compared to these results, the GGA once again seems to show disappointing performance overall, consistently featuring a large rightward extension of the phase transition region.

4 Application to the Bin Packing Problem

In this section we now move our attention to the second OIMGP considered in this study – the bin packing problem (see Section 1 for a definition). In the next section we describe our application of the HC method to this problem; Section 4.2 then contains details on the IG and GGA-version used for the comparison. Finally, Section 4.3 gives details on the experiments performed and results gained.

For this problem, recall that n represents the number of items in a problem instance, and χ the smallest number of groups (i.e. bins) that these items can be feasibly packed into. In addition, let $s[i]$ represent the size of an item i, and let C represent the maximum capacity of the bins. Finally, let $F(g)$ represent the "fullness" of a particular bin g in a candidate solution (that is, the total size of all of the items that have been assigned to bin g). Note that none of the algorithms considered in this comparison ever allow a bin to be over-filled in a candidate solution, thus any group g will always have $F(g) \leq C$.

4.1 A Hill-Climbing Algorithm for Bin Packing.

Our application of the hill-climbing method to the bin packing problem follows a similar pattern to the graph colouring application given in Section 3. First, an initial solution is constructed, this time using the *First Fit Descending (FFD)* heuristic, which involves sorting the n items according to their size (largest first) before applying the greedy algorithm in the usual way.

The improvement scheme used for this application is described in fig. 12 and is based heavily on the "dominance"-based search methods of Martello and Toth [39]. Given two permutations, π and ρ, the aim is to move items between these in such a way that the "fullness" of each group (bin) in π increases, while the number of items within each group in π does not. As a side effect, this operator also moves smaller items into ρ; though this could also be useful in many cases, as smaller items are often more easy to pack than larger items. The procedure thus operates by attempting to swap a pair of items from a bin in π with a pair from a bin in ρ (lines 2-7); then a pair of items from a bin in π with one item from ρ (lines 8-13); and finally one item from π with one item from ρ (lines 14-19). For our application, when this procedure is invoked, it is applied repeatedly until a complete parse is performed with no change occurring to either permutation. For simplicity's sake, all remaining operational details of this hill-climbing application were kept the same as in the graph colouring experiments.

```
BPP-Improvement-Procedure($\pi$, $\rho$, $C$)
(1)   for ($g \leftarrow 1$ to $G(\pi)$)
(2)     foreach (pair of items $i, j$ in group $g$ in $\pi$)
(3)       for ($h \leftarrow 1$ to $G(\rho)$)
(4)         foreach (pair of items $k, l$ in group $h$ in $\rho$)
(5)           $\delta = s[k] + s[l] - s[i] + s[j]$
(6)           if ($\delta > 0$ and $F(g) + \delta \leq C$)
(7)             Move items $i$ and $j$ into group $h$ in $\rho$ and move items $k$ and $l$ into group
                $g$ in $\pi$
(8)     foreach (pair of items $i, j$ in group $g$ in $\pi$)
(9)       for ($h \leftarrow 1$ to $G(\rho)$)
(10)        foreach (item $k$ in group $h$ in $\rho$)
(11)          $\delta = s[k] - s[i] + s[j]$
(12)          if ($\delta > 0$ and $F(g) + \delta \leq C$)
(13)            Move items $i$ and $j$ into group $h$ in $\rho$ and move item $k$ into group $g$ in $\pi$
(14)    foreach (item $i$ in group $g$ in $\pi$)
(15)      for ($h \leftarrow 1$ to $G(\rho)$)
(16)        foreach (item $k$ in group $h$ in $\rho$)
(17)          $\delta = s[k] - s[i]$
(18)          if ($\delta > 0$ and $F(g) + \delta \leq C$)
(19)            Move items $i$ into group $h$ in $\rho$ and move item $k$ into group $g$ in $\pi$
```

Figure 12: The improvement procedure used in the bin packing application. Here, $G(\pi)$ represents the number of groups that are contained in the permutation π (similarly for ρ).

4.2 Iterated Greedy and GGA Setup

In this case the IG algorithm was applied using the same experimental conditions as the previous graph colouring experiments, with an initial solution being produced using the FFD heuristic. For the GGA, meanwhile, we used a hybrid version of the algorithm (HGGA), originally proposed by Falkenauer, which was shown to produce excellent results in other works [21, 22]. For the most part, the operators and evolutionary scheme used for HGGA are the same as those used in Erben's GGA for graph colouring (Section 3.2), though in this case, when performing repair during recombination (Step 4, fig. 5), Falkenauer chooses to use FFD heuristic in order to reinsert any unplaced items. In addition to this, the HGGA also uses a local improvement procedure similar in style to our improvement procedure (figure 12), which is intended for locally improving new offspring candidate solutions when they are first constructed. (Falkenauer reports that this twinning of global and local search techniques produces significantly better results than either method individually – similar findings are also reported by Levine and Ducatelle [34] with their hybrid ant colony algorithm.)

One final point about the HGGA is that the fitness function f used to provide evolutionary selection pressure is:

$$f = \frac{\sum_{g=1}^{G} (F(g)/C)^2}{G} \tag{6}$$

where G represents the number of bins being used in the solution. This function is more fine-grained than, say, simply using G as a fitness function, and therefore allows selection pressure to be sustained for longer during a run. One feature of this function is that it favours solutions with "extremist" bins, as fuller bins are more accentuated by the squaring operation. The upshot of this is that a solution that has, say, two bins that are both 90% full will have a lower fitness than a solution that has one bin that is 100% full and one that is 80% full. (Fitness values of 0.81 and 0.82 respectively).

4.3 Experimental Analysis

To compare the performance of the three algorithms, two benchmark instance-sets were used. The first of these is available at the OR Library of Beasley [2] and contains the 160 instances that were created by Falkenauer, and which have been used in various other works since [34, 47]. Within this instance-set there are two types of problem. The first type are the "uniform" instances, using items of integer size (uniformly distributed between 20 and 100) with $C = 150$. The second type, meanwhile, are instances that have been specially generated so that optimal solutions have all bins fully filled to capacity with exactly three items (therefore $\chi = n/3$, with n being a multiple of 3). The rationale for these "triplet" instances, proposed by Falkenauer, is that bin packing problems become easier to solve when the number of items-per-bin is increased, whilst, on the other hand, problems that have two items or less per bin are trivial to solve. Thus triplet instances are proposed to be "the most difficult bin packing instances" [21]. Note that for both the uniform and triplet problems there are a number of instances for various n-values. Further details are given in Table 2.

It is important to note that since the creation of this instance set, there has been some discussion as to how hard the instances actually are, most notably by Gent [27] who was able to find optimal solutions to the uniform instance set using a very simple heuristic algorithm that exploits the instance-specific knowledge that optimal solutions to these problems have nearly all of their bins filled to capacity. The question of whether this feature makes these instances "easy" or not is open to debate and certainly beyond the scope of this paper (though, for more information, refer to the heated discussions of Gent and Falkenauer [23, 28]). However, to ensure that our comparisons were fair and wide-ranging, a second set of instances, available from the bin-packing resource of Scholl and Klein [48] was also used. In this case we used the ten so-called "hard" instances from this resource which are generated using items of sizes between 20,000 and 35,000 with $C = 100,000$. In all ten of these instances the number of items $n = 200$ and the number of items-per-bin lies between 3 and 5. Finally, note that for all problem instances considered here the optimum number of bins χ is already known, having been established in other works [21, 50]. In the case of the uniform and triplet instances the theoretical minimum number of bins $t = \lceil (\sum_{i=1}^{n} s[i])/C \rceil$ is equal to χ in 159 of the 160 instances (the remaining one instance is the fourteenth instance of the uniform class for $n = 250$, in which $\chi = t + 1$.)[8] For the "hard" instances, meanwhile, only three of

[8]The reader should be aware that when these instances were first posted on the web, some of the stated

Table 2: Number of extra bins (beyond χ) used in the best solutions generated by the various algorithms, averaged across the available instances. The "Inst" column indicates the number of different instances that exist in each set; "Cut" indicates the cut-off points used.

Type	Inst.	n	$\chi(\mu \pm \sigma)$	Cut	FFD	IG	HC	HGGA	MT[a]	HACO[b]	BISON[c]
Unif.	20	120	49.1 ± 1.6	10^{10}	0.7	0.7	0.1	0	0.05	0	-
Unif.	20	200	101.6 ± 2.2	10^{11}	1.5	1.45	0.25	0	0.55	0.1	-
Unif.	20	500	201.2 ± 3.3	20^{11}	2.7	2.7	0.15	0	2.2	0	-
Unif.	20	1000	400.6 ± 4.7	30^{11}	4.85	4.85	0.2	0	3.85	0	-
Trip.	20	60	20 ± 0.0	10^{10}	3.2	2.45	0.85	0.6	1.45	-	-
Trip.	20	120	40 ± 0.0	10^{10}	5.8	5.3	1.0	0.85	4.1	-	-
Trip.	20	249	83 ± 0.0	10^{11}	12.1	11.25	1.0	0	7.45	-	-
Trip.	20	501	167 ± 0.0	20^{11}	23.05	22.4	1.0	0	14.85	-	-
Hard	10	200	56.2 ± 0.7	10^{11}	3.4	2.3	0	0.1	1.5	-	0.7

[a]MT: Martello and Toth's branch-and-bound procedure [39]. Results for the hard instances are taken from [49], the remainder from [21]

[b]HACO: Hybrid Ant Colony Optimisation approach of Levine and Ducatelle [34]

[c]The BISON method of Scholl, Klein, and Jürgens [49]

the ten instances have $\chi = t$, and the remaining seven have $\chi = t + 1$.

Table 2 summarises the quality of solutions found by the three algorithms when granted excess computation time. As a benchmark, the table also shows number of extra bins (beyond χ) required in solutions formed with the FFD heuristic. As before, a platform-independent measure of computational effort was used for these trials – the number of "weight checks", which is similar in spirit to the constraint checks measure used in Section 3. Information regarding the cut-off points used is also included in the table. The various parameters used for the HGGA were the same as those used by Falkenauer [21], though in this case one member of the initial population was also constructed using the FFD heuristic. As with [21], the algorithms were run once with each instance.

The first feature to note from Table 2 is that the IG algorithm only makes very marginal improvements to the initial solutions provided by the FFD heuristic. Thus it would seem that the IG's search mechanisms are not sufficient for penetrating the search space in an effective way with these problem instances. In contrast, the HGGA and HC algorithms produce superior solutions, with the former achieving optimality with all of the uniform instances and also the two largest triplet instance-sets. Compared to the HC algorithm, Table 2 indicates that, on average, the HGGA produces solutions using fewer bins with the uniform and triplet instance sets (though when optimality was not achieved by the HC algorithm, only one extra bin beyond χ was ever required). According to a two-sided signed rank test, however, the lower averages offered by the HGGA are only statistically significant with the triplet instances for $n = 249$ and 501; thus we do not have the evidence to show that the HGGA produces better solutions than the HC algorithm with the other instances. Interestingly, for the "hard" instances, this feature is reversed, and the HC algorithm produces better results

optima were incorrect and, unfortunately, still remain so today. The correct χ-values for these instances have, however, been supplied in the work of Gent [27], and it is these values that were used in this analysis.

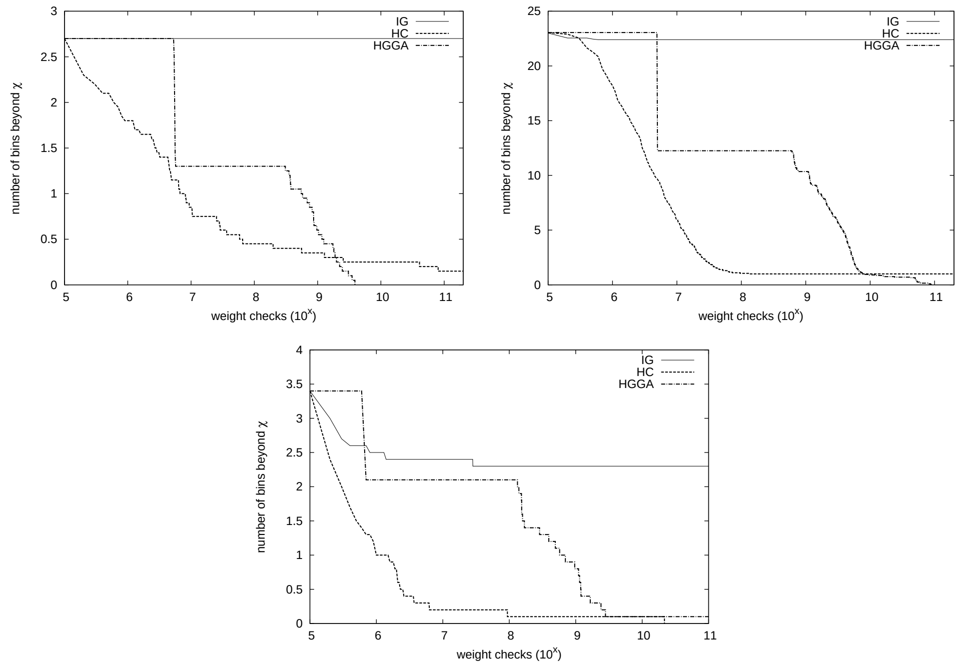

Figure 13: Best-so-far curves (over weight checks) for the various algorithms with (a) the uniform instance set for $n = 500$, (b) the triplet set for $n = 501$, and (c) the "hard" instances ($n = 200$). All points are averaged across one run on all instances in each set.

than the HGGA (although, again, this difference was not seen to be statistically significant). One possible reason for this reversal is that, unlike the uniform and triplet instances, optimal solutions to the "hard" instances do not always have their bins full to capacity – a feature favoured by the HGGA's fitness function. For these particular instances the HGGA might therefore be being deceived, as its fitness function (eq. 6) will tend to favour solution-features that are not necessarily occurring in global optima.

The three rightmost columns of Table 2 also contain results reported in the literature for three other well-known bin packing methods [34, 39, 49]. Although we must again show caution when comparing such figures, with the hard instances we see that the HC algorithm has produced significantly better results than the methods reported in both [39] and [49]. In addition, no significant difference is observed between the results of the HC algorithm and those reported in [34].

Moving beyond Table 2, however, perhaps the most striking characteristic of the HC algorithm is the way that it behaves *during* a run. This is demonstrated in fig. 13 where best-so-far graphs, according to the number of weight checks, are shown for three different instance-sets. (Note that log-scales are used here, emphasising the behaviour of the algorithms in the left of the graphs.) To begin with, we see that the HGGA only makes very small improvements due to the fact that ≈6 million checks (600,000 for the smaller "hard" instances) are used for creating the initial populations. Next, there then occurs a rapid drop, mostly due to the improvements achieved by the HGGA's local improvement operators, before steady, if somewhat slow, progress is then made through the search space. However,

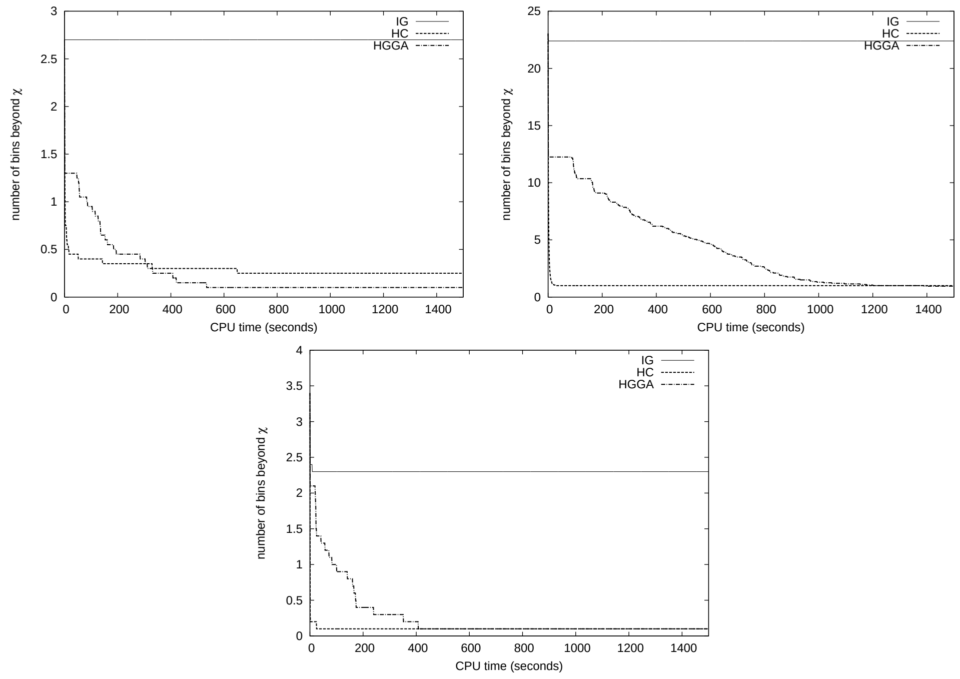

Figure 14: Best-so-far curves (over CPU time) for the various algorithms with (a) the uniform instance set for $n = 500$, (b) the triplet set for $n = 501$, and (c) the "hard" instances ($n = 200$). All points are averaged across one run on all instances in each set.

in stark contrast to this, the HC algorithm's progression through the search space is much more rapid – indeed, we can see that the HGGA only catches up with the HC algorithm at around 10^9 or 10^{10} checks. Obviously, these observations tell us that as well as being able to find solutions of comparable quality in excess time, perhaps more importantly, the HC algorithm is able to identify good solutions using considerably less computational effort than the HGGA. Viewing these runs from the perspective of CPU time (fig 14) further illustrates these observations.

5 Conclusions and Discussion

In this paper we have presented an algorithmic framework for a class of problems referred to as Order Independent Minimum Grouping Problems and have provided example applications to two popular problems of this type. In the author's experience these algorithms were simple to implement and robust to alterations in the small number of parameters that they required. This latter feature contrasts the GGA in particular, which we found to be quite sensitive to parameter alterations.

Two prominent features of this hill-climbing approach are that it only works with feasible solutions, and that it does not permit changes to a solution that worsen its quality. For some practical situations these features may be useful because, in effect, the algorithm can be stopped at any time during a run and it will always be at a point in the search space that is

(a) feasible, and (b) at least as good as anything that has been found thus far. In addition, if an optimal solution *is* found, then the algorithm can still be left to run indefinitely if desired, whereupon it has the potential for producing other feasible optimal solutions from which the user can then choose.

In our analysis of the graph colouring problem, we have seen that the HC method compares well to the well-known IG and GGA methods and is often able to produce equivalent or better solutions using smaller amounts of computational effort. For these experiments we specifically chose to use an automated generator for producing problem graphs, preferring to look at how this general-purpose HC method performs on a wide range of different instances, rather than a small number of "standard" benchmark instances such as the DIMACs problems. Indeed, at this point we believe it is more instructive to examine the general performance characteristics of this method and to identify its relative strengths and weaknesses, before targeting it at specific, special-case instances. For similar reasons we have also refrained from using *instance*-specific heuristics in our applications of the hill-climbing method.

Regarding this latter point, there is, however, plenty of scope for introducing such instance-specific heuristics in the future. For example, when considering equipartite and flat graph colouring instances, there is currently nothing "driving" the algorithm towards producing solutions with equi-sized groups – a feature that optimal solutions to these problems must have. In such cases, the search space could be reduced in size (and altered in shape) by simply stipulating that no group can contain more than $\lceil n/k \rceil$ nodes. A similar strategy could also be used with the "triplet" bin-packing instances where it is known that optimal solutions will have $\chi = n/3$ with all bins filled to capacity with exactly three items. Note, however, that in order to make use of instance-specific rules such as these, it is first necessary to know something about what an optimal solution looks like, though such information will not always be available in practice.

It is possible that performance improvements for the hill climbing method could also be gained in the future by using different, more problem-specific group shuffling operators. For example, in bin packing we may see improvements by using an operator that sorts the groups (bins) according to their spare capacity before applying the greedy algorithm; in graph colouring it might be useful to sort the groups according to the degrees of the nodes contained in each. For the graph colouring problem, performance might also be improved by using different neighbourhood operators in the local improvement procedure, such as the Kempe- and S-chain neighbourhoods operators used by Morgenstern [41] and Thompson and Dowsland [52].

Of course, one final research question to consider is how the HC algorithm might be adapted to cope with other OIMGPs. Of particular interest to us are the more complex two- and three-dimensional bin packing problems, which can have many real-world and industrial applications and can also come in various different forms.[9] One interesting feature of these problems is that, unlike graph colouring and one-dimensional bin packing, the order in which items occur within each group in a permutation, and the *way* in which items are then inserted

[9]For example, in two-dimensional bin packing, we might encounter problems where only rectangles need to be packed; in other cases, however, circles or more complex polygons might also be considered – refer to the review of Lodi *et al.* [37] and the online resource of Erich [24] for a good overview of these types of problem).

into bins will almost certainly influence how well the items are ultimately packed together. In particular, this means that the minimum number of permutations that decode into an optimal solution will not now be defined by eq. (4). New local improvement procedures will also need to be designed in these cases to maximise the potential of the hill-climbing algorithm. This question, as with the others that we have raised are, however, all pending future research.

References

[1] C. Avanthay, A. Hertz, and N. Zufferey. A variable neighborhood search for graph coloring. *European Journal of Operations Research*, 151:379–388, 2003.

[2] J. Beasley. http://people.brunel.ac.uk/∼mastjjb/jeb/orlib/binpackinfo.html.

[3] D Brelaz. New methods to color the vertices of a graph. *Commun. ACM*, 22(4):251–256, 1979.

[4] M. Carter, G. Laporte, and S. Y. Lee. Examination timetabling: Algorithmic strategies and applications. *Journal of the Operational Research Society*, 47:373–383, 1996.

[5] M. Chams, A. Hertz, and O. Dubuis. Some experiments with simulated annealing for coloring graphs. *European Journal of Operations Research*, 32:260–266, 1987.

[6] P. Cheeseman, B. Kanefsky, and W. Taylor. Where the really hard problems are. In *Proceedings of IJCAI-91*, pages 331–337, 1991.

[7] M. Chiarandini and T. Stützle. An application of iterated local search to graph coloring. In In D. Johnson, A. Mehrotra, and M. Trick, editors, *Proceedings of the Computational Symposium on Graph Coloring and its Generalizations*, pages 112–125, New York, USA, 2002.

[8] D. Corne, P. Ross, and H. Fang. Evolving timetables. In Lance C. Chambers, editor, *The Practical Handbook of Genetic Algorithms*, volume 1, pages 219–276. CRC Press, 1995.

[9] D. Costa and A. Hertz. Ants can colour graphs. *Journal of the Operational Research Society*, 48:295–305, 1997.

[10] B. G. W. Craenen. *Solving Constraint Satisfaction Problems with Evolutionary Algorithms*. PhD thesis, Vrije Universiteit Amsterdam, 2005.

[11] B. G. W. Craenen, A. E. Eiben, and J. I. van Hemert. Comparing evolutionary algorithms on binary constrant satisfaction problems. *IEEE Transactions on Evolutionary Computation*, 7(5):424–444, October 2003.

[12] J Culberson. http://web.cs.ualberta.ca/∼joe/coloring/.

[13] J. Culberson and F. Luo. Exploring the k-colorable landscape with iterated greedy. In D. S. Johnson and M. A. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, volume 26, pages 245–284. American Mathematical Society, 1996.

[14] L. Di Gaspero and A. Schaerf. Neighborhood portfolio approach for local search applied to timetabling problems. *Journal of Mathematical Modelling and Algorithms*, 5(1):65–89, 2006.

[15] R. Dorne and J-K. Hao. A new genetic local search algorithm for graph coloring. In A. Eiben, T. Back, M. Schoenauer, and H. Schwefel, editors, *Parallel Problem Solving from Nature (PPSN) V*, volume 1498, pages 745–754. Springer-Verlag, Berlin, 1998.

[16] K. Dowsland and J. Thompson. An improved ant colony optimisation heuristic for graph colouring. *Discrete Applied Mathematics*, 156:313–324, 2008.

[17] A. E. Eiben, J. K. van der Hauw, and J. I. van Hemert. Graph coloring with adaptive evolutionary algorithms. *Journal of Heuristics*, 4(1):25–46, 1998.

[18] E. Erben. A grouping genetic algorithm for graph colouring and exam timetabling. In E. Burke and W. Erben, editors, *Practice and Theory of Automated Timetabling (PATAT) III*, volume 2079, pages 132–158. Springer-Verlag, Berlin, 2001.

[19] E. Falkenauer. A new representation and operators for genetic algorithms applied to grouping problems. *Evolutionary Computation*, 2(2):123–144, 1994.

[20] E. Falkenauer. Solving equal piles with the grouping genetic algorithm. In L. J. Eshelman, editor, *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 492–497. Morgan Kaufmann Inc, 1995.

[21] E. Falkenauer. A hybrid grouping genetic algorithm for bin packing. *Journal of heuristics*, 2(1):5–30, 1996.

[22] E. Falkenauer. *Genetic Algorithms and Grouping Problems*. John Wiley and Sons, 1998.

[23] E. Falkenauer. On method overfitting. *Journal of Heuristics*, 4(3):281–287, 1998.

[24] E Friedman. http://www.stetson.edu/∼efriedma/packing.html.

[25] P. Galinier and Hao J-K. Hybrid evolutionary algorithms for graph coloring. *Journal of Combinatorial Optimization*, 3:379–397, 1999.

[26] M. R. Garey and D. S. Johnson. *Computers and Intractability - A guide to NP-completeness*. W. H. Freeman and Company, San Francisco, first edition, 1979.

[27] I. Gent. Heuristic solution of open bin packing problems. *Journal of Heuristics*, 3(4):299–304, 1998.

[28] I. Gent. A response to “on method overfitting”. *Journal of Heuristics*, 5(1):109–111, 1999.

[29] A. Hertz and D. de Werra. Using tabu search techniques for graph coloring. *Computing*, 39(4):345–351, 1987.

[30] T. R. Jensen and B. Toft. *Graph Coloring Problems.* Wiley-Interscience, 1 edition, 1994.

[31] I. Juhos and J. I. van Hemert. Increasing the efficiency of graph colouring algorithms with a representation based on vector operations. *Journal of Software*, 1(2):24–33, August 2006.

[32] S. Khuri, T. Walters, and Y. Sugono. A grouping genetic algorithm for coloring the edges of graphs. In *SAC '00: Proceedings of the 2000 ACM symposium on Applied computing*, pages 422–427. ACM Press, New York, 2000.

[33] F.T. Leighton. A graph coloring algorithm for large scheduling problems. *Journal of Research of the National Bureau of Standards*, 84(6):489–506, 1979.

[34] J. Levine and F. Ducatelle. Ant colony optimisation and local search for bin packing and cutting stock problems. *Journal of the Operational Research Society*, 55(12)(7):705–716, 2003.

[35] R. Lewis. Metaheuristics can solve sudoku puzzles. *Journal of heuristics*, 13(4):387–401, 2007.

[36] R. Lewis and B. Paechter. Finding feasible timetables using group based operators. *IEEE Transactions on Evolutionary Computation*, 11(3):397–413, 2007.

[37] A. Lodi, S. Martello, and G. Vigo. Recent advances on two-demensional bin packing pproblems. *Discrete Applied Mathematics*, 123:379–396, 2002.

[38] H. Lourenco, O. Martin, and T. Stutzle. *Handbook of Metaheuristics*, chapter Applying Iterated Local Search to the Permutation Flow Shop Problem. Boston, MA: Kluwer Academic Publishers, 2003.

[39] S. Martello and P. Toth. Lower bounds and reduction procedures for the bin packing problem. *Discrete Applied Mathematics*, 28:59–70, 1990.

[40] A. Mehrotra and M. Trick. A column generation approach for graph coloring. *INFORMS Journal on Computing*, 8:344–354, 1996.

[41] C. Morgenstern. *Algorithms for General Graph Coloring.* PhD thesis, University of New Mexico, 1989.

[42] B. Paechter, R. Rankin, A. Cumming, and T. Fogarty. Timetabling the classes of an entire university with an evolutionary algorithm. In T. Baeck, A. Eiben, M. Schoenauer, and H. Schwefel, editors, *Parallel Problem Solving from Nature (PPSN) V*, volume 1498, pages 865–874. Springer-Verlag, Berlin, 1998.

[43] L. Paquete and T. Stützle. An experimental investigation of iterated local search for coloring graphs. In Stefano Cagnoni, Jens Gottlieb, Emma Hart, Martin Middendorf, and Gunther Raidl, editors, *Applications of Evolutionary Computing, Proceedings of EvoWorkshops2002: EvoCOP, EvoIASP, EvoSTim*, volume 2279, pages 121–130, Kinsale, Ireland, 3-4 2002. Springer-Verlag.

[44] S. Prestwich. Using an incomplete version of dynamic backtracking for graph colouring. *Electronic Notes in Discrete Mathematics*, 1:61–73, 1999.

[45] J. Randall-Brown. Chromatic scheduling and the chromatic number problem. *Management Science,*, 19(4):456–463, 1972.

[46] P. Ross, E. Hart, and D. Corne. Genetic algorithms and timetabling. In A. Ghosh and K. Tsutsui, editors, *Advances in Evolutionary Computing: Theory and Applications*, pages 755–771. Springer-Verlag, New York., 2003.

[47] P. Ross, J. G. Marín-Blázquez, S. Schulenburg, and E. Hart. Learning a procedure that can solve hard bin-packing problems: A new ga-based approach to hyper-heuristics. In Erick Cantú-Paz, James A. Foster, Kalyanmoy Deb, Lawrence Davis, Rajkumar Roy, Una-May O'Reilly, Hans-Georg Beyer, Russell K. Standish, Graham Kendall, Stewart W. Wilson, Mark Harman, Joachim Wegener, Dipankar Dasgupta, Mitchell A. Potter, Alan C. Schultz, Kathryn A. Dowsland, Natasa Jonoska, and Julian F. Miller, editors, *GECCO*, volume 2724 of *Lecture Notes in Computer Science*, pages 1295–1306. Springer, 2003.

[48] A. Scholl and R. Klein. http://www.wiwi.uni-jena.de/entscheidung/binpp/index.htm.

[49] A. Scholl, R. Klein, and C. Jurgens. Bison: A fast hybrid procedure for exactly solving the one-dimensional bin packing problem. *Computers and Operations Research*, 25(7):627–645, 1997.

[50] P. Schwerin and G. Wascher. The bin-packing problem: A problem generator and some numerical experiments with ffd packing and mtp. *International Transactions in Operational Research*, 4:377–389, 1997.

[51] J. Thompson and K. Dowsland. Variants of simulated annealing for the examination timetabling problem. *Annals of Operational Research*, 63(1):105–128, 1996.

[52] J. Thompson and K. Dowsland. A robust simulated annealing based examination timetabling system. *Computers and Operations Research*, 25(7/8):637–648, 1998.

[53] A. Tucker, J. Crampton, and S. Swift. Rgfga: An efficient representation and crossover for grouping genetic algorithms. *Evolutionary Computation*, 13(4):477–499, 2005.

[54] J. S. Turner. Almost all k-colorable graphs are easy to color. *Journal of Algorithms*, 9:63–82, 1988.

[55] C. M. Valenzuela. A study of permutation operators for minimum span frequency assignment using an order based representation. *Journal of Heuristics*, 7:5–21, 2001.

[56] G. V. von Lazewski. Intelligent structural operators for the k-way graph partitioning problem. In R. K. Belew and L. B. Booker, editors, *Forth International Conference on Genetic Algorithms*, pages 45–52. Morgan Kaufmann, San Mateo, CA, USA, 1991.

[57] D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Trans. Evolutionary Computation*, 1(1):67–82, 1997.

Appendix 1

For the following proof, recall that $U = \{U_1, \ldots, U_G\}$, such that equations (1), (2), and (3) hold, with each $U_i \in U$ representing a set of items that constitute a feasible group. Additionally, for the bin packing problem let $F(U_i)$ represent the total size of all items in U_i, and recall that C is a constant representing the maximum capacity of each bin.

Proof of Theorem 1 is now based on the following lemma:

Lemma 1 *If U_i is a set of items that constitutes a feasible group, then any subset $V_i \subseteq U_i$ also constitutes a feasible group.*

It is trivial to show that Lemma 1 is true for the bin packing and graph colouring problems. For bin packing, if U_i is feasible, this implies that $F(U_i) \leq C$; thus, $F(V_i) \leq F(U_i) \leq C$. With the graph colouring problem, meanwhile, if U_i is feasible then this implies that no pair of items (nodes) in U_i are adjacent in the corresponding graph. Trivially, this property is also retained if any items are removed from U_i.

Now consider an application of the greedy algorithm using U in order to build a new candidate solution U'. To begin, let $U' = \{U'_1, \ldots, U'_G\}$ with $U'_i = \emptyset$ (for $i = 1, \ldots, G$). Once the greedy algorithm has completed we will then be able to remove any empty sets from U' if present. In applying the greedy algorithm, each set U_i (for $i = 1, \ldots, G$) is now considered in turn, and all items $u \in U_i$ are assigned one-by-one to some set $U'_j \in U'$ according to the rules of the algorithm. (That is, u is first considered for inclusion in U'_1, then U'_2, and so on).

Considering each item $u \in U_i$, two situations and resultant actions will now occur in the following order of priority:

Case 1: $\exists U'_j$ $(1 \leq j < i)$ such that $U'_j \cup \{u\}$ is feasible.

Action: $U'_j \leftarrow U'_j \cup \{u\}$. That is, item u is assigned into a set in U' that has a lower index than its set in U.

Case 2: $U'_j \cup \{u\}$ is feasible, and $i = j$.

Action: $U'_j \leftarrow U'_j \cup \{u\}$. That is, item u is assigned to the set in U' with the *same* index as its set in U.

It is clear that in both cases above, an item u is always assigned to a set in U' with a lower or equal index to its original set in U. Of course, if a situation is encountered where *all* items in a particular set U_i are assigned according to Case 1, then at termination of the greedy algorithm U' will contain an empty set which, once removed, will give $|U'| < |U|$.

For the purposes of contradiction, now assume that we can encounter a situation where it is necessary to assign an item $u \in U_i$ to a set U'_k such that $k > i$. Obviously, for this to occur, it is first necessary that the proposed actions of Case 2 (i.e. adding u to $U'_{j=i}$) would form an infeasible group. However, in cases where u has not been feasibly assigned to a group $U'_{j<i}$ (according to Case 1), then following the rules of the greedy algorithm, the next group to be considered will be U'_i. Thus, $U'_i \subset U_i$ and will be feasible according to Lemma 1. By definition, $U'_i \cup \{u\}$ will also be feasible since $U'_i \cup \{u\} \subseteq U_i$, thus contradicting the assumption. □