Chapter 4
Algorithm Case Studies

In this chapter we present detailed descriptions of six high-performance algorithms for the graph colouring problem. Implementations of each of these can be found in the online suite of graph colouring algorithms described in Section 1.6.1 and Appendix A.1. In Section 4.2 onwards we then compare the performance of these algorithms over a wide range of graphs in order to gauge their relative strengths and weaknesses.

4.1 Algorithm Descriptions

4.1.1 The TABUCOL *Algorithm*

As we mentioned in the previous chapter, since its proposal by Hertz and de Werra in 1987, TABUCOL has been used as a local search subroutine in a number of high-performing hybrid algorithms, including those of Avanthay et al. (2003), Dorne and Hao (1998), Galinier and Hao (1999), and Thompson and Dowsland (2008). The specific version of TABUCOL that we consider here is the so-called "improved" variant, which was originally used by Galinier and Hao (1999). The various features of this algorithm are now reviewed.

TABUCOL operates in the space of complete improper k-colourings using an objective function that simply counts the number of clashes, as defined by f_2 in Equation (3.15). Given a candidate solution $\mathcal{S} = \{S_1, \ldots, S_k\}$, moves in the solution space are performed by selecting a vertex $v \in S_i$ whose assignment to colour class S_i is currently causing a clash, and then assigning it to a new colour class $S_j \neq S_i$. Note that previous incarnations of this algorithm also allowed nonclashing vertices to be moved between colours, though this is generally seen to worsen performance (Galinier and Hertz, 2006).

The tabu list of the algorithm is stored using a matrix $\mathbf{T}_{n \times k}$. If, at iteration l of the algorithm, the neighbourhood operator transfers a vertex v from S_i to S_j, then the element T_{vi} is set to $l+t$, where t is a positive integer that will be defined presently. This signifies that the moving of v back to colour class S_i is *tabu* (i.e., disallowed) for t iterations of the algorithm (or, in other words, that v cannot be moved back to S_i until at least iteration $l+t$). Note that this has the effect of making *all* solutions containing the assignment of vertex v to S_i tabu for t iterations.

As is typical in applications of tabu search, in each iteration of TABUCOL the entire set of neighbouring solutions is considered. That is, the cost of moving each clashing vertex into all other $k-1$ colour classes is evaluated. This process consumes the majority of the algorithm's execution time; however, it can be sped up considerably through the use of appropriate data structures. To explain, let x denote the number of vertices involved in a clash in the current solution $\mathcal{S}$. This leads to $x(k-1)$ members in the set of neighbouring solutions $N(\mathcal{S})$. (Obviously, there is a strong positive correlation between x and the objective function, so better solutions will tend to have smaller neighbourhoods.) A naïve implementation of the TABUCOL would set about separately performing the $x(k-1)$ different neighbourhood moves and evaluating all the resulting solutions. However, this is not necessary, particularly because only two colour classes are effected by each neighbourhood move.

A more efficient approach involves making use of an additional matrix $\mathbf{C}_{n\times k}$ where, given the current solution $\mathcal{S}=\{S_1,\ldots,S_k\}$, element C_{vj} denotes the number of vertices in colour class S_j that are adjacent to vertex v. When an initial solution is generated, all elements in $\mathbf{C}$ will need to be calculated. However, in each subsequent iteration of TABUCOL, the act of moving a vertex v from S_i to S_j will result in a new solution $\mathcal{S}'$ whose cost is simply:

$$f_2(\mathcal{S}') = f_2(\mathcal{S}) + C_{vj} - C_{vi}. \tag{4.1}$$

Since $f_2(\mathcal{S})$ will already be known, this means that the cost of all neighbouring solutions can be determined by simply scanning each row of $\mathbf{C}$ corresponding to clashing vertices in $\mathcal{S}$.

Once a move has been selected and performed (i.e., once v has been moved from S_i to S_j), the matrix $\mathbf{C}$ can be updated using the procedure shown in Figure 4.1. As shown in this pseudocode, neighbours of v are now marked as being adjacent to one fewer vertex in colour class S_i and one additional vertex in colour class S_j.

```
UPDATE-C $(v,i,j)$
(1) forall $u \in \Gamma(v)$ do
(2)     $C_{ui} \leftarrow C_{ui} - 1$
(3)     $C_{uj} \leftarrow C_{uj} + 1$
```

Fig. 4.1 Procedure for updating the matrix **C** once TABUCOL has moved a vertex v from colour i to colour j. As usual, $\Gamma(v)$ denotes the set of all vertices adjacent to vertex v

Having evaluated all neighbouring solutions, TABUCOL selects and performs the non-tabu move that brings about the largest decrease (or failing that, the smallest increase) in cost. Any ties in this criterion are broken randomly. In addition, TABUCOL also employs an *aspiration criterion* which allows tabu moves to be made on occasion. Specifically, they are permitted if they are seen to improve on the best solution found so far during the run. This is particularly helpful if a tabu move is seen to lead to a solution $\mathcal{S}'$ with zero cost, at which point the algorithm can halt. Finally, if *all* moves are seen to be tabu, then a vertex $v \in V$ is selected at random and moved to a new randomly selected colour class. The tabu list is then updated as usual.

In the version of TABUCOL that we use here, an initial candidate solution is constructed by taking a random ordering of the vertices and applying a modified version of the GREEDY algorithm in which only k colours are permitted. Thus, if vertices are encountered that cannot be assigned to any of the k colours without inducing a clash, these are assigned to one of the existing colours randomly. Of course, we could use more sophisticated constructive methods here, but it is stated by both Galinier and Hertz (2006) and Blöchliger and Zufferey (2008) that the method of initial solution generation is not critical in TABUCOL's performance.

Finally, with regard to the *tabu tenure*, Galinier and Hao (1999) have suggested making t a random variable that is proportional to the incumbent solution's cost. The idea here is that when the incumbent solution is poor, its high cost will lead to large values for t, which will hopefully force the algorithm into different regions of the solution space where better solutions can be found. On the other hand when the incumbent solution has a low cost, the algorithm should focus on the current region by using low values for t. Galinier and Hao (1999) suggest using $t = 0.6f_2 + r$, where r is an integer uniformly selected from the range 0 to 9 inclusive. These particular settings have been used in various other applications of TABUCOL (Blöchliger and Zufferey, 2008; Galinier and Hao, 1999; Thompson and Dowsland, 2008) and are generally thought to give good results; however, it should be noted that other schemes for determining t are likely to be more appropriate for certain graphs.

4.1.2 *The* PARTIALCOL *Algorithm*

The PARTIALCOL algorithm of Blöchliger and Zufferey (2008) operates in a similar fashion to TABUCOL in that it uses the tabu search metaheuristic to seek a proper k-colouring. However, in contrast to TABUCOL, PARTIALCOL does not consider improper solutions; instead, vertices that cannot be assigned to any of the k colours without causing a clash are put into a set of uncoloured vertices U. The aim of PARTIALCOL is to thus make alterations to the partial solution $\mathcal{S}$ so that U can be emptied, giving $f_3 = |U| = 0$ and, consequently, a feasible k-coloured solution.

Because of its use of partial proper solutions, the neighbourhood operator of PARTIALCOL is somewhat different from that of TABUCOL. Specifically, a move in the solution space is achieved by selecting an uncoloured vertex $v \in U$ and assigning it to a colour class $S_j \in \mathcal{S}$. The move is then completed by taking all vertices $u \in S_j$ that are adjacent to v and transferring them from S_j into U. Having performed such a move, all corresponding elements T_{uj} in the tabu list are then marked as tabu for the next t iterations of the algorithm.

In each iteration of PARTIALCOL, the complete set of $|U| \times k$ neighbouring solutions is examined. The move to be performed is then chosen using the same criteria as TABUCOL. As with TABUCOL the matrix **C** can again be used to speed up the process of evaluating the neighbourhood set. In this case, the act of moving vertex v from U to colour class S_j leads to a new solution $\mathcal{S}'$ whose cost is simply:

$$f_3(\mathcal{S}') = f_3(\mathcal{S}) + C_{vj} - 1. \tag{4.2}$$

Once a move has been performed (that is, the vertex $v \in U$ has been transferred to S_j and all vertices in the set $\{u \in S_j : u \in \Gamma(v)\}$ have been moved to U), the **C** matrix is updated using the procedure given in Figure 4.2.

```
    UPDATE-C $(v, j)$
(1) forall $u \in \Gamma(v)$ do
(2)     $C_{uj} \leftarrow C_{uj} + 1$
(3)     if $c(u) = j$ then
(4)         forall $w \in \Gamma(u)$ do
(5)             $C_{wj} \leftarrow C_{wj} - 1$
```

Fig. 4.2 Procedure for updating **C** once PARTIALCOL has moved vertex v from the set U to colour j

An initial solution to PARTIALCOL is generated using a greedy process analogous to that of TABUCOL. The only difference is that when vertices are encountered for which there exists no clash-free colours, these are put into the set U. The only other operational difference between the two algorithms relates to the calculation of the tabu tenure t. In their original paper, Blöchliger and Zufferey (2008) use an algorithm variant known as FOO-PARTIALCOL. Here, FOO abbreviates "Fluctuation Of the Objective-function", and indicates their use of a mechanism that alters t based on the algorithm's search progress. In essence, if during a run the objective function has not altered for a lengthy period of time, it is assumed that the search has stagnated in a particular region of the solution space and so t is increased to try to encourage the algorithm to leave this region. Similarly, when the objective function is seen to be fluctuating, t is slowly reduced, counteracting these effects. Note that this scheme requires values to be assigned to a number of parameters, the meanings of which are described by Blöchliger and Zufferey (2008). In our case, we choose to use settings recommended by the authors and these are included in our source code of this algorithm. We are perfectly at liberty to use other simpler schemes for calculating t if required, however.

4.1.3 The Hybrid Evolutionary Algorithm (HEA)

The third algorithm that we shall consider is the hybrid evolutionary algorithm (HEA) of Galinier and Hao (1999). The HEA operates by maintaining a population of candidate solutions that are evolved via a problem-specific recombination operator and a local search method. Like TABUCOL, the HEA operates in the space of complete improper k-colourings using cost function f_2.

The algorithm begins by creating an initial population of candidate solutions. Each member of this population is formed using a modified version of the DSATUR algorithm for which the number of colours k is fixed at the outset. To provide diversity between members, the first vertex is selected at random and assigned to the first colour. The remaining vertices are then taken in sequence according to the maximum saturation degree (with ties being broken randomly) and assigned to the lowest indexed colour class S_i seen to be feasible (where $1 \leq i \leq k$). When vertices are encountered for which no feasible colour class exists, these are kept to one side and are assigned to random colour classes at the end of this process. Upon construction of this initial population, an attempt is then made to improve each member by applying the local search routine.

As is typical for an evolutionary algorithm, for the remainder of the run the algorithm evolves the population using recombination, mutation, and evolutionary pressure. In each iteration two parent solutions $\mathcal{S}_1$ and $\mathcal{S}_2$ are selected from the population at random, and copies of these are used in conjunction with the recombination operator to produce one child solution $\mathcal{S}'$. This child is then improved via the local search operator, and is inserted into the population by replacing the weaker of its two parents. Note that there is no bias towards selecting fitter parents for recombination; rather evolutionary pressure only exists due to the offspring replacing their weaker parent (regardless of whether the parent has a better cost than its child).

	Parent $\mathcal{S}_1$	Parent $\mathcal{S}_2$	Offspring $\mathcal{S}'$	Comments
1)	$\{\{v_1,v_2,v_3\},$ $\{v_4,v_5,v_6,v_7\},$ $\{v_8,v_9,v_{10}\}\}$	$\{\{v_3,v_4,v_5,v_7\},$ $\{v_1,v_6,v_9\},$ $\{v_2,v_8,v_{10}\}\}$	$\{\}$	To start, the offspring solution $\mathcal{S} = \emptyset$.
2)	$\{\{v_1,v_2,v_3\},$ $\{v_8,v_9,v_{10}\}\}$	$\{\{v_3\},$ $\{v_1,v_9\},$ $\{v_2,v_8,v_{10}\}\}$	$\{\{v_4,v_5,v_6,v_7\}\}$	Select the colour class with most vertices and copy it into $\mathcal{S}'$. (Class $\{v_4,v_5,v_6,v_7\}$ from $\mathcal{S}_1$ in this case.) Delete the copied vertices from both $\mathcal{S}_1$ and $\mathcal{S}_2$.
3)	$\{\{v_1,v_3\},$ $\{v_9\}\}$	$\{\{v_3\},$ $\{v_1,v_9\}\}$	$\{\{v_4,v_5,v_6,v_7\},$ $\{v_2,v_8,v_{10}\}\}$	Select the largest colour class in $\mathcal{S}_2$ and copy it into $\mathcal{S}'$. Delete the copied vertices from both $\mathcal{S}_1$ and $\mathcal{S}_2$.
4)	$\{\{v_9\}\}$	$\{\{v_9\}\}$	$\{\{v_4,v_5,v_6,v_7\},$ $\{v_2,v_8,v_{10}\},$ $\{v_1,v_3\}\}$	Select the largest colour class in $\mathcal{S}_1$ and copy it into $\mathcal{S}'$. Delete the copied vertices from both $\mathcal{S}_1$ and $\mathcal{S}_2$.
5)	$\{\{v_9\}\}$	$\{\{v_9\}\}$	$\{\{v_4,v_5,v_6,v_7\},$ $\{v_2,v_8,v_{10},v_9\},$ $\{v_1,v_3\}\}$	Having formed k colour classes, assign any missing vertices to random colours to form a complete but not necessarily proper offspring solution $\mathcal{S}$.

Fig. 4.3 Example application of the Greedy Partition Crossover of Galinier and Hao (1999), using $k = 3$

The recombination operator proposed Galinier and Hao (1999) is the so-called Greedy Partition Crossover (GPX). The idea behind GPX is to construct offspring using large colour classes inherited from both parent solutions. A demonstration of how this is done is given in Figure 4.3. As shown, the largest (not necessarily proper) colour class from the parents is first selected and copied into the offspring (ties are broken randomly). In order to avoid duplicate vertices occurring in the offspring at a later stage, these copied vertices are then removed from both parents. To form the next colour, the other (modified) parent is then considered and, again, the largest colour class is selected and copied into the offspring, before these vertices are removed from both parents. This process is continued by alternating between the parents until the offspring's k colour classes have been formed.

At this point, each colour class in the offspring will be a subset of a colour class existing in one or both of the parents. That is:

$$\forall S_i \in \mathcal{S}' \;\; \exists S_j \in (\mathcal{S}_1 \cup \mathcal{S}_2) \; : \; S_i \subseteq S_j \tag{4.3}$$

where $\mathcal{S}'$, $\mathcal{S}_1$, and $\mathcal{S}_2$ represent the offspring, and the first and second parents respectively. However, some vertices may be missing in the offspring (as is the case with vertex v_9 in Figure 4.3). This issue is resolved by assigning the missing vertices to random colour classes.

Once a complete offspring solution is formed, it is then modified and improved via a local search procedure before being inserted into the population. For this purpose the TABUCOL algorithm is used for a fixed number of iterations I using the same tabu tenure scheme as described in Section 4.1.1. In their original paper, Galinier and Hao (1999) present results for a small sample of problem instances and manually tune I for each case. In our case we choose not to follow this strategy and require a setting for I to be determined automatically by the algorithm. We also need to be wary that if I is set too low, then insufficient local search will be carried out on each newly created solution, while an I that is too high will result in too much effort being placed on local search as opposed to the global search carried out by the evolutionary operators. Ultimately we choose to settle on $I = 16n$, which corresponds roughly to the settings used in the most successful runs reported by Galinier and Hao (1999). In all cases, we also use a population size of 10, as recommended by the authors.

4.1.4 The ANTCOL *Algorithm*

Like the HEA, the ANTCOL algorithm of Thompson and Dowsland (2008) is another metaheuristic-based method that combines global and local search operators, in this case using the ant colony optimisation (ACO) metaheuristic.

ACO is an algorithmic framework that was originally inspired by the way in which real ants determine efficient paths between food sources and their colonies. In their natural habitat, when no food source has been identified, ants tend to wander about randomly. However, when a food source is found, the discovering ants will take some of this back to the colony leaving a pheromone trail in their wake. When other ants discover this pheromone, they are less likely to continue wandering at random, but may instead follow the trail. If they go on to discover the same food source, they will then follow the pheromone trail back to the nest, adding their own pheromone in the process. This encourages further ants to follow the trail. In addition to this, pheromones on a trail also tend to evaporate over time, reducing the chances of an ant following it. The longer it takes for an ant to traverse a path, the more time the pheromones have to evaporate; hence shorter paths tend to see a more rapid build-up of pheromone, making other ants more likely to follow it and deposit their own pheromone. This positive feedback eventually leads to all ants following a single, efficient path between the colony and food source.

As might be expected, initial applications of ACO were aimed towards problems such as the travelling salesman problem and vehicle routing problems, where we seek to identify efficient paths for visiting the vertices of a graph (see for example the work of Dorigo et al. (1996) and Rizzoli et al. (2007)). However, applications to many other problems have also been made.

The idea behind the ANTCOL algorithm is to use ants to produce individual candidate solutions. During a run each ant produces its solution in a nondeterministic manner, using probabilities based on heuristics and also on the quality of solutions produced by previous ants. In particular, if previous ants have identified features that are seen to lead to better-than-average solutions, the current ant is more likely to include these features in its own solution, generally leading to a reduction in the number of colours during the course of a run.

A full description of the ANTCOL algorithm is provided in Figure 4.4. As shown in the pseudocode, in each cycle of the algorithm (lines (3) to (19)), a number of ants each produce a complete, though not necessarily feasible, solution. In line (16) the details of each of these solutions are then added to a trail update matrix δ and, at the end of a cycle, the contents of δ are used together with an evaporation rate ρ to update the global trail matrix t.

At the start of each cycle, each individual ant attempts to construct a solution using the procedure BUILDSOLUTION. This is based on the RLF method (see Section 2.4) which, we recall, operates by building up each colour class in a solution one at a time. Also recall that during the construction of each class $S_i \in \mathcal{S}$, RLF makes use of two sets: X, which contains uncoloured vertices that can currently be added to S_i without causing a clash; and Y, which holds the uncoloured vertices that *cannot* be feasibly added to S_i. The modifications to RLF that BUILDSOLUTION employs are as follows:

- In the procedure a maximum of k colour classes is permitted. Once these have been constructed, any remaining vertices are left uncoloured.
- The first vertex to be assigned to each colour class S_i $(1 \leq i \leq k)$ is chosen randomly from the set X.
- In remaining cases, each vertex v is then assigned to colour S_i with probability

```
ANTCOL $(G=(V,E))$
(1)  $t_{uv} \leftarrow 1\ \forall u,v \in V : u \neq v$
(2)  $k = n$
(3)  while (not stopping condition) do
(4)      $\delta_{uv} \leftarrow 0\ \forall u,v \in V : u \neq v$
(5)      $best \leftarrow k$
(6)      $foundFeasible \leftarrow$ false
(7)      for ($ant \leftarrow 1$ to $nants$) do
(8)          $\mathcal{S} \leftarrow$BUILDSOLUTION$(k)$
(9)          if ($\mathcal{S}$ is a partial solution) then
(10)             Randomly assign uncoloured vertices to colour classes in $\mathcal{S}$
(11)             Run TABUCOL
(12)         if ($\mathcal{S}$ is feasible) then
(13)             $foundFeasible \leftarrow$ true
(14)             if ($|\mathcal{S}| \leq best$) then
(15)                 $best \leftarrow |\mathcal{S}|$
(16)         $\delta_{uv} \leftarrow \delta_{uv} + F(\mathcal{S})\ \forall u,v : c(u) = c(v) \wedge u \neq v$
(17)     $t_{uv} \leftarrow \rho \times t_{uv} + \delta_{uv}\ \forall u,v \in V : u \neq v$
(18)     if ($foundFeasible$=true) then
(19)         $k \leftarrow best - 1$
```

Fig. 4.4 The ANTCOL algorithm. At termination, the best feasible solution found uses $k+1$ colours

$$P_{vi} = \begin{cases} \dfrac{\tau_{vi}^{\alpha} \times \eta_{vi}^{\beta}}{\sum_{u \in X}(\tau_{ui}^{\alpha} \times \eta_{ui}^{\beta})} & \text{if } v \in X \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

where τ_{vi} is calculated

$$\tau_{vi} = \frac{\sum_{u \in S_i} t_{uv}}{|S_i|}. \tag{4.5}$$

Note that the calculation of τ_{vi} makes use of the global trail matrix t, meaning that higher values are associated with combinations of vertices that have been assigned the same colour in previous solutions. The value η_{vi}, meanwhile, is associated with a heuristic rule which, in this case, is the degree of vertex v in the graph induced by the set of currently uncoloured vertices $X \cup Y$. Larger values for τ_{vi} and η_{vi} thus contribute to larger values for P_{vi}, encouraging vertex v to be assigned to colour class S_i. The parameters α and β are used to control the relative strengths of τ and η in the equation.

The ANTCOL algorithm also makes use of a "multi-sets" operator in the BUILD-SOLUTION procedure. Since the process of constructing a colour class is probabilistic, the operator makes ν separate attempts to construct each colour class. It then selects the one that results in the minimum number of edges in the graph induced by the set of remaining uncoloured vertices Y (since such graphs will tend to feature lower chromatic numbers).

On completion of BUILDSOLUTION, the generated solution $\mathcal{S}$ will be proper, but could be partial. If the latter is true, all uncoloured vertices are assigned to random colour classes to form a complete, improper solution, and TABUCOL is run for I iterations. Details on the solution are then written to the trail update matrix δ using the evaluation function:

$$F(\mathcal{S}) = \begin{cases} 1/f_2 & \text{if } f_2 > 0 \\ 3 & \text{otherwise.} \end{cases} \tag{4.6}$$

This means that higher-quality solutions contribute larger values to δ, encouraging their features to be included in solutions produced by future ants.

The parameters used in our application, and recommended by Thompson and Dowsland (2008), are as follows: $\alpha = 2$, $\beta = 3$, $\rho = 0.75$, $nants = 10$, $I = 2n$, and $\nu = 5$. The tabu tenure scheme of TABUCOL is the same as in previous descriptions.

4.1.5 The Hill-Climbing (HC) Algorithm

In contrast to the preceding four algorithms, the Hill-Climbing (HC) algorithm of Lewis (2009) operates in the space of feasible solutions, with the initial solution being formed using the DSATUR heuristic. During a run, the algorithm operates on a single feasible solution $\mathcal{S} = \{S_1, \ldots S_{|\mathcal{S}|}\}$ with the aim of minimising $|\mathcal{S}|$. To begin, a small number of colour classes are removed from $\mathcal{S}$ and are placed into a second set $\mathcal{T}$, giving two partial proper solutions. A specialised local search procedure is then run for I iterations. This attempts to feasibly transfer vertices from colour classes in $\mathcal{T}$ into colour classes in $\mathcal{S}$ such that both $\mathcal{S}$ and $\mathcal{T}$ remain proper. If successful, this has the effect of increasing the cardinality of the colour classes in $\mathcal{S}$ and may also empty some of the colour classes in $\mathcal{T}$, reducing the total number of colours being used. At the end of the local search procedure, all colour classes in $\mathcal{T}$ are copied back into $\mathcal{S}$ to form a feasible solution.

The first iteration of the local search procedure operates by considering each vertex v in $\mathcal{T}$ and checking whether it can be feasibly transferred into any of the colour classes in $\mathcal{S}$. If this is the case, such transfers are performed. The remaining iterations of the procedure then operate as follows. First, an alteration is made to a randomly selected pair of colour classes $S_i, S_j \in \mathcal{S}$ using either a Kempe chain interchange or a pair swap (see Definitions (3.1) and (3.2)). Since this will usually alter the make-up of two colour classes,[1] this then raises the possibility that other vertices in $\mathcal{T}$ can now also be moved to S_i or S_j. Again, these transfers are made

[1] Note that in some cases a Kempe chain will contain all vertices in both colour classes: that is, the graph induced by $S_i \cup S_j$ will form a connected bipartite graph. Kempe chains of this type are known as *total*, and interchanging their colours serves no purpose since this only results in the two colour classes being relabelled. Consequently total Kempe chains are ignored by the algorithm.

if they are seen to retain feasibility. The local search procedure continues in this fashion for I iterations.

On completion of the local search procedure, the independent sets in $\mathcal{T}$ are copied back into $\mathcal{S}$ to form a feasible solution. The independent sets in $\mathcal{S}$ are then ordered according to some (possibly random) heuristic, and a new solution $\mathcal{S}'$ is formed by constructing a permutation of the vertices in the same manner as that of the Iterated Greedy algorithm (see Section 3.2.1) and then applying the GREEDY algorithm. This latter operation is intended to generate large alterations to the incumbent solution, which is then passed back to the local search procedure for further optimisation. Note that none of the stages of this algorithm allow the number of colour classes being used to increase, thus providing its hill-climbing characteristics.

As with the previous algorithms, a number of parameters have to be set with this algorithm, each that can influence its performance. The values used in our experiments here were determined in preliminary tests and according to those reported by Lewis (2009). For the local search procedure, independent sets are moved into $\mathcal{T}$ by considering each $S_i \in \mathcal{S}$ in turn and transferring it with probability $1/|\mathcal{S}|$. The local search procedure is then run for $I = 1,000$ iterations, and in each iteration the Kempe chain and swap neighbourhoods are called with probabilities 0.99 and 0.01 respectively. Finally, when constructing the permutation of the vertices for passing to the GREEDY algorithm, the independent sets are ordered using the same 5:5:3 ratio as detailed in Section 3.2.1.

4.1.6 *The Backtracking* DSATUR *Algorithm*

The sixth and final algorithm considered in this chapter is the backtracking approach of Korman (1979). Essentially, this operates in the same manner as the basic backtracking approach discussed in Section 3.1.1, though with the following modifications:

- The initial order of the vertices is determined by the DSATUR algorithm. Hence vertices with the fewest available colours are coloured first, with ties being broken by the degrees, and further ties being broken randomly.
- After performing a backward step, vertices are dynamically reordered so that the next vertex to be coloured is also the one with the fewest available colours. If the vertex has no feasible colours available, the algorithm takes a further backward step.

An example run-through of this algorithm is shown in Figure 4.5. This should be interpreted in the same manner as Figure 3.1. Note that a number of parameters can be set when applying this algorithm, some of which might alter the performance quite drastically. These include specifying the maximum number of branches that can be considered at each node of the tree and prohibiting branching at certain levels of the tree. In practice, it is not obvious how these settings might be chosen a priori for individual graphs, so in our case we opt for the most natural configuration, which

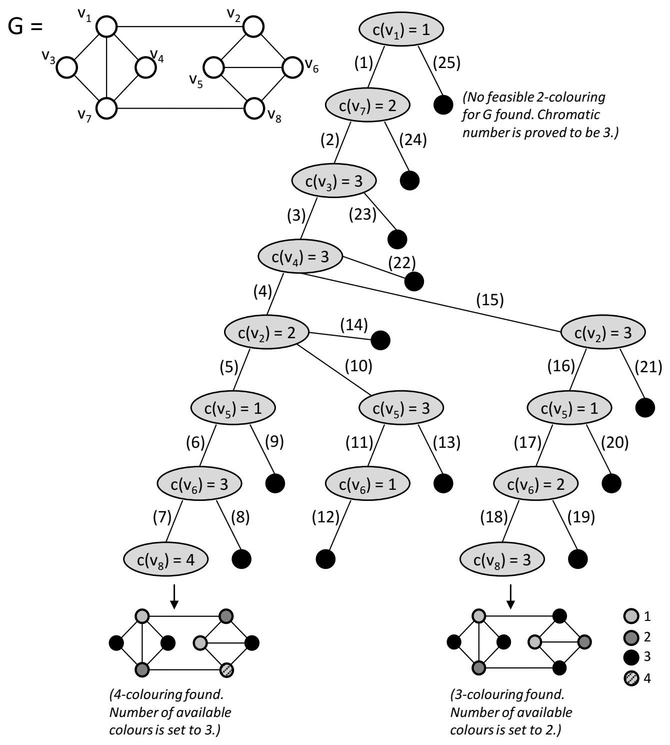

Fig. 4.5 Example run of the backtracking algorithm of Korman (1979)

is to simply attempt a complete exploration of the search tree.[2] This means that the algorithm is exact under excess time, though of course such run-lengths will not be possible in most cases.

4.2 Algorithm Comparison

In this section we now compare the above six algorithms using a selection of different graph types. As with our comparison of constructive algorithms in Chapter 2, we begin by considering random graphs. We then go on to consider a further type of artificially generated graph, the flat graph, before looking more closely at sets of graphs arising in two real-world practical problems, namely university timetabling and social networking.

As with our previous experiments, computational effort for these algorithms is measured by counting the number of constraint checks (see Section 1.6.1). Due to the operational differences of the algorithms, during a run solution quality is mea-

[2] These parameters can be altered in the implementation, however.

sured by simply observing the smallest number of colours used in a feasible solution up until that point. Note that because TABUCOL, PARTIALCOL and the HEA operate using infeasible solutions, settings for k are also required which might then need to be modified during a run. In our case initial values are determined by executing DSATUR on each instance and setting k to the number of colours used in the resultant solution. During runs, k is then decremented by 1 each time a feasible k-colouring is found, with the algorithms being restarted. In all trials a computation limit of 5×10^{11} constraint checks was imposed. This value is chosen to be deliberately high in order to provide some notion of excess time in the trials. Example run times (in seconds) using this computation limit are given in Table 4.6 later.

4.2.1 Artificially Generated Graphs

According to Definition 2.15, random graphs are generated such that each pair of vertices is made adjacent with probability p. For the following experiments we used values of p ranging from 0.05 (sparse) to 0.95 (dense), incrementing in steps of 0.05, with $n \in \{250, 500, 1000\}$. Twenty-five instances were generated in each case.

The second type of artificial graph we consider are flat graphs. These are produced by taking a graph $G = (V, E = \emptyset)$ and then partitioning the n vertices into q almost equi-sized independent sets (i.e., each set contains either $\lfloor n/q \rfloor$ or $\lceil n/q \rceil$ vertices). Edges are then added between pairs of vertices in different independent sets with probability p in such a way that the variance in vertex degrees is kept to a minimum.

It is well known that q-coloured solutions to flat graphs are quite easy to achieve for most values of p. This is because for lower values for p, problems will be under-constrained, perhaps giving $\chi(G) < q$, and making q-coloured solutions easily identifiable. On the other hand, high values for p can result in over-constrained problems with prominent global optima that are easily discovered. Hard-to-solve q-colourable graphs are known to occur for a region of p's at the boundary of these extremes, commonly termed the *phase transition region* (Cheeseman et al., 1991; Turner, 1988). Flat graphs, in particular, are known to have rather pronounced phase transition regions because each colour class and vertex degree is deliberately similar, implying a lack of heuristic information for algorithms to exploit.

For our experiments, flat graphs were generated using publicly available software designed by Joseph Culberson which can be downloaded at web.cs.ualberta.ca/∼joe/coloring. Graphs were produced for $q \in \{10, 50, 100\}$ using various settings of p in and around the phase transition regions. In each case we used $n = 500$, implying approximately 50, ten, and five vertices per colour respectively. Twenty instances were generated in each case.

Note that according to the structure of random graphs, vertex degrees are characterised by the binomial distribution $\mathrm{B}(n-1, p)$. This means that the standard deviation of the vertex degrees, calculated

$$\sigma = \sqrt{(n-1)p(1-p)}, \tag{4.7}$$

does not exceed 15.8 in this test set of random graphs. It also implies that the degree coefficient of variation (CV), which is defined as the ratio of the standard deviation to the mean σ/μ, never exceeds 28% (being maximised at $n = 250$, $p = 0.05$). In a similar fashion, flat graphs are constructed such that variance in degrees is minimised, and for our generated instances this means that the CV never exceeds 28.5%. Compared to many of the more real-world graphs considered later, these values imply a rather high level of vertex homogeneity (i.e., vertices tend to "look the same"), helping to explain some of the following results.

4.2.1.1 Performance on Random Graphs

Table 4.1 shows the number of colours used in solutions produced by the six algorithms for random graphs with edge probability $p = 0.5$ and varying numbers of vertices. The results indicate that for the smaller graphs ($n = 250$), the TABUCOL, PARTIALCOL and HEA algorithms produce solutions with fewer colours than the remaining algorithms.[3] However, no statistical difference between these three algorithms is apparent. For larger graphs however, the HEA produces the best results, allowing us to conclude that, for $n = 500$ and $n = 1,000$, the HEA algorithm is able to produce the best solutions across the set of all graphs and their isomorphisms under this particular computation limit.

Table 4.1 Summary of results produced at the computation limit using random graphs $G_{n,0.5}$

	Algorithm[a]					
n	TABUCOL	PARTIALCOL	HEA	ANTCOL	HC	Bktr
250	**28.04± 0.20**	**28.08 ± 0.28**	**28.04 ± 0.33**	28.56 ± 0.51	29.28 ± 0.46	34.24 ± 0.78
500	49.08 ± 0.28	49.24 ± 0.44	**47.88 ± 0.51**	49.76 ± 0.44	54.52 ± 0.77	62.24 ± 0.72
1000	88.92 ± 0.40	89.08 ± 0.28	**85.48 ± 0.46**	89.44 ± 0.58	101.44 ± 0.82	112.88 ± 0.97

[a] Mean plus/minus standard deviation in number of colours, taken from runs across 25 graphs.

Moving on to other densities, the graphs shown in Figure 4.6 summarise the mean solution quality achieved by the six algorithms on all random graphs generated. In each figure, the bars show the number of colours used in solutions produced by DSATUR and the lines then give the proportion of this number used in the solutions of the six algorithms. Note that all algorithms achieve a reduction in the number of colours realised by DSATUR, though in all but the smallest, sparsest graphs, the backtracking algorithm exhibits the smallest margins of improvement, apparently due to the high levels of vertex homogeneity in these instances, which makes it difficult for favourable regions of the search tree to be identified.

[3] As in Chapter 2, statistical significance is claimed here according to the nonparametric Related Samples Wilcoxon Signed Rank test (for pairwise comparisons), and the Related Samples Friedman's Two-way Analysis of Variance by Ranks (for group comparisons). For the remainder of this chapter statistical significance is claimed at the 1% level.

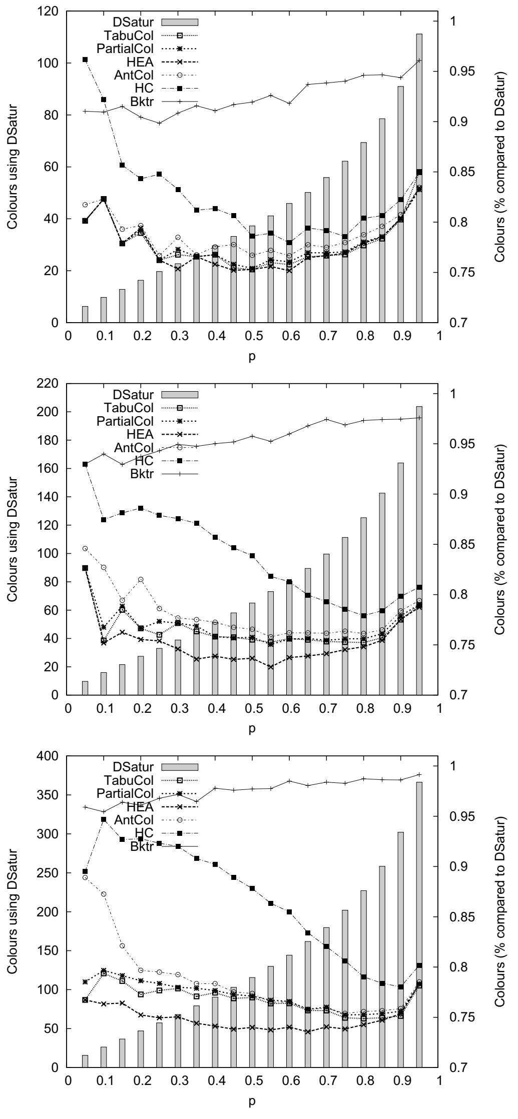

Fig. 4.6 Mean quality of solution achieved on random graphs using $n = 250$, 500, and 1,000 (respectively) for various edge probabilities p. All points are the mean of 25 runs on 25 different instances

It is clear from Figure 4.6 that TABUCOL, PARTIALCOL, and the HEA in particular, produce the best results for the random graphs. For $n = 250$ these algorithms produce mean results that, across the range of values for p, show no significant difference among one another, perhaps indicating that the achieved solutions are consistently close to the optimal solutions. For larger graphs however, the HEA's solutions are seen to be significantly better, though its rates of improvement are slightly slower than those of TABUCOL and PARTIALCOL, as illustrated by Figure 4.7. Similar behaviour during runs was also witnessed with the smaller random instances.

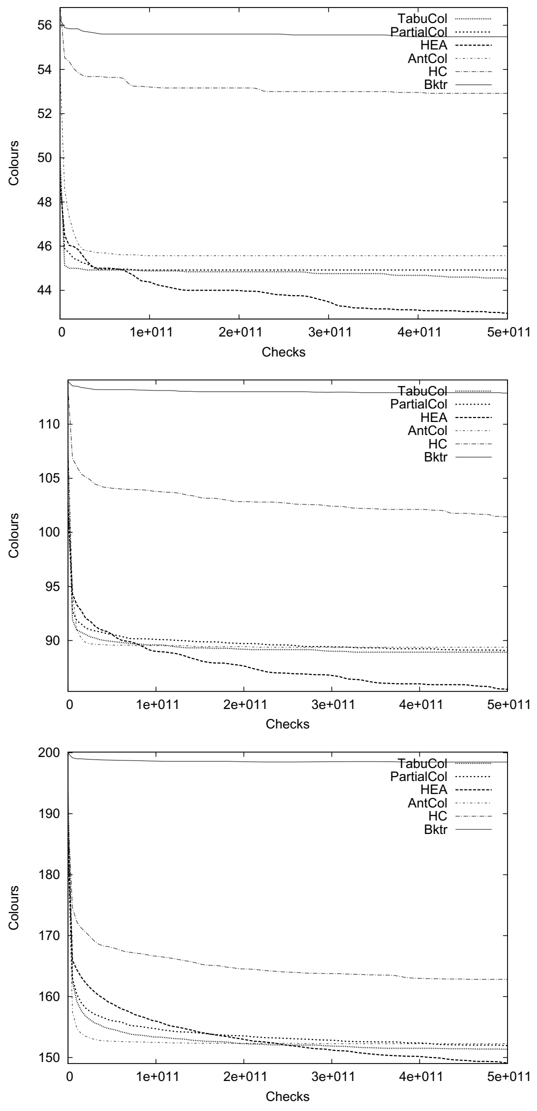

Fig. 4.7 Run profiles on random graphs of $n = 1,000$ with edge probabilities $p = 0.25$, 0.5, and 0.75 respectively. Each line represents a mean of 25 runs on 25 different instances

Overall, the patterns shown in Figure 4.6 indicate that the HEA's strategy of exploring the space of infeasible solutions using both global and local search operators is the most beneficial of those considered here. Indeed, although the HC algorithm also uses both global and local search operators, here its insistence on preserving feasibility implies a lower level of connectivity in its underlying solution space, making navigation more restricted and resulting in noticeably inferior solutions.

Figure 4.6 also reveals that ANTCOL does not perform well with large sparse instances, though it does become more competitive with denser instances. The reasons for this are twofold. First, the degrees of vertices in sparse graphs are naturally lower, reducing the heuristic bias provided by η and perhaps implying an over-dominant role of τ during applications of BUILDSOLUTION (see Equation (4.4)). Secondly, sparse graphs also feature greater numbers of vertices per colour—thus, even if very promising independent sets *are* identified by ANTCOL, their reconstruction by later ants will naturally depend on a longer sequence of random trials, making them less likely. To back these assertions, we also repeated the trials of ANTCOL using the same local search iteration limit as the HEA, $I = 16n$. However, though this brought slight improvements for denser graphs, the results were still observed to be significantly worse than the HEA's, suggesting the difference in performance indeed lies with the global-search element of ANTCOL in these cases.

4.2.1.2 Performance on Flat Graphs

Similar patterns are also revealed when we turn our attention towards the performance of the six algorithms with flat graphs, as shown in Figure 4.8. Again, we see that the HEA, TABUCOL, and PARTIALCOL exhibit the best performance on instances within the phase transition regions, with the HC and backtracking algorithms proving the least favourable. One pattern to note is that for the three values of q, the HEA tends to produce the best-quality results on the left side of the phase transition region, but PARTIALCOL produces better results for a small range of p's on the right side. However, this difference is not due to the "FOO" tabu tenure mechanism of PARTIALCOL, because no significant difference was observed when we repeated our experiments using PARTIALCOL under TABUCOL's tabu tenure scheme. Thus, it seems that PARTIALCOL's strategy of only allowing solutions to be built from independent sets is favourable in these cases, presumably because this restriction facilitates the formation of independent sets of size n/q—structures that will be less abundant in denser graphs, but which also serve as the underlying building blocks in these cases.

Another striking feature of Figure 4.8 is the poor performance of ANTCOL on the right side of the phase transition regions. This again seems to be due to the diminished effect of heuristic value η, which in this case is due to the variance in vertex degrees being deliberately low, making it difficult to distinguish between vertices. Furthermore, in denser graphs fewer combinations comprising n/q vertices will form independent sets, decreasing the chances of an ant constructing one. This reasoning is also backed by the fact that ANTCOL's poor performance lessens with

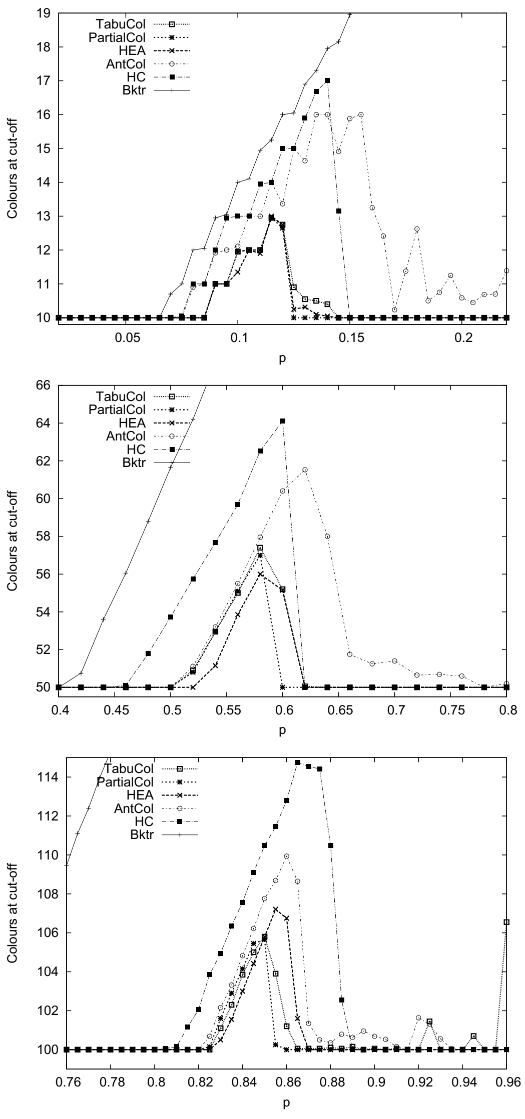

Fig. 4.8 Mean quality of solution achieved with flat graphs of $n = 500$ with $q = 10$, 50, and 100 (respectively) for various edge probabilities p. All points are the mean of 20 runs on 20 different instances

larger values of q where, due to there being fewer vertices per colour, the reproduction of independent sets is dependent on shorter sequences of random trials.

4.2.2 Exam Timetabling Problems

Our first set of "real world" problem instances in this comparison concerns graphs representing university timetabling problems. As we saw in Section 1.1.2, timetabling problems involve assigning a set of "events" (exams, lectures, etc.) to a fixed number of "timeslots", and a pair of events "conflict" when they require the same single resource: e.g., there may be a student or lecturer who needs to attend both events, or the events may require use of the same room. As a result conflicting events need to be assigned to different timeslots. Under this constraint, timetabling problems can be modelled as graph colouring problems by considering each event as a vertex, with edges occurring between pairs of events that conflict. Each colour then represents a timeslot, and a feasible colouring corresponds to a complete timetable with no conflict violations.

In practice, universities will often have a predefined number of timeslots in their timetable and their task will be to determine a feasible solution using fewer or equal timeslots. In many cases however, it might be difficult to ascertain whether a timetable with a given number of timeslots is achievable for a particular problem, or it may be desirable to use as few timeslots as possible, particularly if it provides extra time for marking, or allows for a shorter teaching day. Here we concern ourselves with the latter problem, and use a well-known set of real-world timetabling problems compiled by Carter et al. (1996). This set contains 13 exam timetabling problems encountered in various universities from across the globe during the 1980s and 1990s.

			Degree		
Instance	n	Density	Min;Med;Max	Mean μ	CV (σ/μ)
hec-s-92	81	0.415	9; 33; 62	33.7	36.3%
sta-f-83	139	0.143	7; 16; 61	19.9	67.4%
yor-f-83	181	0.287	7; 51; 117	52	35.2%
ute-s-92	184	0.084	2; 13; 58	15.5	69.1%
ear-f-83	190	0.266	4; 45; 134	50.5	56.1%
tre-s-92	261	0.180	0; 45; 145	47	59.6%
lse-f-91	381	0.062	0; 16; 134	23.8	93.2%
kfu-s-93	461	0.055	0; 18; 247	25.6	120.0%
rye-s-93	486	0.075	0; 24; 274	36.5	111.8%
car-f-92	543	0.138	0; 64; 381	74.8	75.3%
uta-s-92	622	0.125	1; 65; 303	78	73.7%
car-s-91	682	0.128	0; 77; 472	87.4	70.9%
pur-s-93	2419	0.029	0; 47; 857	71.3	129.5%

Table 4.2 Details of the 13 timetabling instances of Carter et al. (1996)

A summary of these problem instances is provided in Table 4.2. The names of the graphs start with a three-letter code denoting the name of the university. This is followed by an "s" or "f" specifying whether the problem occurred in the summer or fall semester, and this is then followed by the year. We see that the set contains problems ranging in size from $n = 81$ to $2,419$ vertices, and densities of 2.9% up to 41.5%.

It is also known that many of these problem instances feature high numbers of rather large cliques. As Ross et al. (2003) have noted:

> Consider the instance kfu-s-93, by no means the hardest or largest in this set. It involves 5,349 students sitting 461 exams, ideally fitted into 20 timeslots. The problem contains two cliques of size 19 and huge numbers of smaller ones. There are 16 exams that clash with over 100 others.

	Colours at Cut-off: Mean (best)					
Instance	TabuCol	PartialCol	HEA	AntCol	HC	Bktr
hec-s-92	17.22 (17)	17.00 (17)	17.00 (17)	17.04 (17)	17.00 (17)	19.00 (19)
sta-f-83	13.35 (13)	13.00 (13)	13.00 (13)	13.13 (13)	13.00 (13)	*13.00 (13) [100%, <0.1%]
yor-f-83	19.74 (19)	19.00 (19)	19.06 (19)	19.87 (19)	19.00 (19)	20.00 (20)
ute-s-92	10.00 (10)	10.00 (10)	10.00 (10)	11.09 (10)	10.00 (10)	10.00 (10)
ear-f-83	26.21 (24)	22.46 (22)	22.02 (22)	22.48 (22)	22.00 (22)	*22.00 (22) [100%, 0.7%]
tre-s-92	20.58 (20)	20.00 (20)	20.00 (20)	20.04 (20)	20.00 (20)	23.00 (23)
lse-f-91	19.42 (18)	17.02 (17)	17.00 (17)	17.00 (17)	17.00 (17)	*17.00 (17) [100%, 1.3%]
kfu-s-93	20.76 (19)	19.00 (19)	19.00 (19)	19.00 (19)	19.00 (19)	19.00 (19)
rye-s-93	22.40 (21)	21.06 (21)	21.04 (21)	21.55 (21)	**21.00** (21)	22.00 (22)
car-f-92	39.92 (36)	32.48 (31)	28.50 (28)	30.04 (29)	27.96 (27)	***27.00** (27) [100%, 8.2%]
uta-s-92	41.65 (39)	35.66 (34)	30.80 (30)	32.89 (32)	30.27 (30)	**29.00 (29)**
car-s-91	39.10 (32)	30.20 (29)	29.04 (28)	29.23 (29)	29.10 (28)	**28.00** (28)
pur-s-93	50.70 (47)	45.48 (42)	33.70 (33)	33.47 (33)	33.87 (33)	**33.00** (33)
Total	341.05 (315)	302.36 (294)	280.16 (277)	286.84 (281)	**279.20 (276)**	282.00 (282)
Rank	(6)	(5)	(2)	(4)	(1)	(3)

Table 4.3 Summary of algorithm performance on the 13 timetabling instances of Carter et al. (1996). All statistics are collected from 50 runs on each instance. Asterisks in the rightmost column indicate where the backtracking algorithm was able to produce a provably optimal solution. In these cases, the square brackets indicate the % of runs where this occurred, and the average % of the computation limit that this took

Table 4.3 summarises the results achieved at the computation limit with the six graph colouring algorithms. Note that in contrast to the artificial instances from the previous section, the worst overall performance now occurs with those methods relying solely on local search: that is, TabuCol and to a lesser extent PartialCol. Indeed, we find that these methods are often incapable of achieving feasible solutions even using the initial setting for k determined by DSatur.[4] The cause of this poor performance seems to lie in the fact that, as shown in Table 4.2, the degree coefficient of variations (CVs) of these 13 problems are considerably higher than that of the artificially generated instances seen in the previous subsection. The effects

[4] Consequently, the reported results for TabuCol and PartialCol in Table 4.3 are produced using an initial k generated by executing the Greedy algorithm with a random permutation of the vertices.

of this are shown in Figure 4.9 where, compared to a random graph of a similar size and density, the differences in cost between neighbouring solutions vary much more widely. This suggests a more "spiky" cost landscape in which the use of local search mechanisms in isolation is insufficient, exhibiting a susceptibility to becoming trapped at local optima.

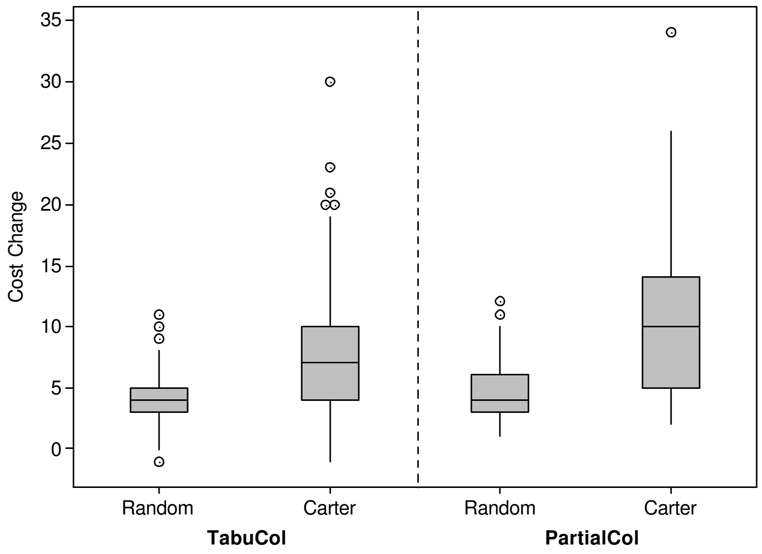

Fig. 4.9 Cost-change distributions for a random graph ($n = 500$, $p = 0.15$, CV= 10.7%, using $k = 16$) and timetable graph car-f-92 ($n = 543$, $p = 0.138$, CV= 75.3%, using $k = 27$). In all cases samples are taken from candidate solutions with costs of 8

Table 4.3 also shows that the most consistent performance with these graphs is achieved by the HC and HEA algorithms (no significant difference between the two methods across the instances is apparent). This demonstrates that the issues of using local search in isolation are alleviated by the addition of a global search-based operator. On individual instances, the relative performances of HC and HEA do seem to vary, however. With the problem instances car-f-92 and car-s-91, for example, the HEA's best observed solutions are determined within approximately 1% of the computation limit, while HC's progress is much slower. On the other hand, with instances such as rye-s-93, HC consistently produces the best observed results in less that 0.3% of the computation limit, suggesting that its operators are somehow suited to this instance. This issue is considered further in Section 4.4.1.

In contrast to the artificially generated graphs seen earlier, we also observe that the backtracking algorithm is quite competitive with these instances. For four of the problem instances the algorithm has managed to find and prove the optimal solutions in all runs using a small fraction of the computation limit; however, these do not correspond to the smallest instances as we might have expected. In addition, the algorithm has produced the best average performance out of all algorithms with the four *largest* problem instances, seeming to contradict the often-held belief that complete algorithms are only suitable for graphs with less that 100 vertices or so (see Section 3.2). It seems in these cases that the large degree CVs characterise an abundance of heuristic information that is being successfully exploited by the algorithm. Indeed, for the four largest instances, all of the solutions reported in Table 4.3 were actually found in less than 2% of the computation limit, implying that the algorithm quickly arrives at the correct regions of the search tree. However, counterexamples in which the backtracking algorithm consistently produces the worst performance can also be seen in Table 4.3, such as with the smallest instance, hec-s-92.

Finally, we also note the sporadic performance of ANTCOL with these instances. For all but the four largest problems, ANTCOL's best solutions equal those of the other algorithms; however, its averages are less favourable, particularly compared to the HEA and HC algorithms. Consider, for example, the results of ute-s-92 in the table. This problem is consistently solved using ten colours by all methods except ANTCOL, which often requires 11 or 12 colours. In fact, we find that for instances such as these, ANTCOL's performance depends very much on the quality of solutions produced in the first cycle of the algorithm. Due to the low vertex degrees (and reduced influence of η that results), Equation (4.4) is predominantly influenced by the pheromone values τ; however, if an 11- or 12-colour solution is produced during the first cycle, features of these suboptimal solutions are still used to update the pheromone matrix t, making their reoccurrence in later cycles more likely. The upshot is that ANTCOL is rarely seen to improve upon solutions found in the initial cycle of the algorithm with these instances.

4.2.3 Social Networks

Our final set of experiments in this chapter involves graphs representing social networks. Social networks consist of "nodes" (usually individual people) that are "tied" by some sort of inter-dependency such as friendship or belief. Here we consider the social networks of school friends, compiled as part of the USA-based National Longitudinal Study of Adolescent Health project (Moody and White, 2003). The colouring of such networks might be required when we wish to partition the students into groups such that individuals are kept separate from their friends, e.g., for group assignments and team-building exercises (see also Section 1.1.1).

To construct these networks, surveys were conducted in various schools, with each student being asked to list all of his or her friends. In some cases, students were only allowed to nominate friends attending the same school, while in others they could include friends attending a "sister school" (e.g., middle-school students could include friends in the local high school), leading to single-cluster and double-cluster networks respectively. In the resultant graphs, each student is represented by a vertex, with edges signifying a claimed friendship between the associated individuals (see Figure 4.10). Note that in the original data, edges signifying friendships are both directed and weighted; however, in our case directions and weights have been removed to form a simple graph.

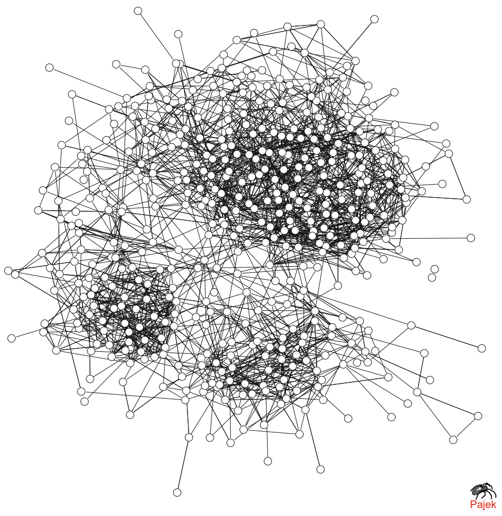

Fig. 4.10 Illustration of a double-cluster social network collected in the National Longitudinal Study of Adolescent Health project (Moody and White, 2003)

Instance	n	Density	Degree		
			Min;Med;Max	Mean μ	CV (σ/μ)
Single-Cluster					
#1	380	0.021	0; 8; 23	8.1	50.5%
#2	542	0.013	0; 7; 35	7.1	61.7%
#3	563	0.013	0; 7; 23	7.3	55.4%
#4	578	0.015	0; 8; 24	8.8	52.7%
#5	626	0.013	0; 7; 30	7.8	58.7%
#6	746	0.010	0; 7; 28	7.3	58.6%
#7	828	0.008	0; 6; 23	6.2	59.3%
#8	877	0.009	0; 7; 29	7.8	58.2%
#9	1229	0.003	0; 4; 17	4.1	54.6%
#10	2250	0.002	0; 4; 25	4.3	78.0%
Double-Cluster					
#11	291	0.027	0; 8; 21	7.8	54.6%
#12	426	0.018	0; 7; 26	7.5	56.2%
#13	457	0.016	0; 7; 23	7.4	58.8%
#14	495	0.017	0; 8; 22	8.5	46.8%
#15	569	0.017	0; 9; 34	9.4	50.9%
#16	586	0.016	0; 9; 30	9.6	48.4%
#17	689	0.010	0; 6; 22	6.8	62.0%
#18	795	0.011	0; 9; 24	8.7	53.7%
#19	1089	0.007	0; 8; 29	8.1	57.9%
#20	1246	0.007	0; 9; 33	8.6	54.4%

Table 4.4 Details of the 20 social networks used

In our tests we took a random sample of ten single-cluster networks and ten double-cluster networks from the Adolescent Health data set. Summary statistics of these graphs are given in Table 4.4. These figures indicate that the vertex degrees are far lower that the timetabling graphs from the previous section, with the highest degree across the whole set being just 29. Consequently, the densities of the graphs are also much lower.

	Colours at Cut-off: Mean (best)					
Instance	TabuCol	PartialCol	HEA	AntCol	HC	Bktr
Single-Cluster						
#1	8 (8)	8 (8)	8 (8)	8.15 (8)	8 (8)	8 (8)
#2	6 (6)	6 (6)	6 (6)	6.76 (6)	6 (6)	*6 (6) [100%, <1%]
#3	7 (7)	7 (7)	7 (7)	7.45 (7)	7 (7)	7.02 (7)
#4	8 (8)	8 (8)	8 (8)	8.75 (8)	8 (8)	8 (8)
#5	8 (8)	8 (8)	8 (8)	8.41 (8)	8 (8)	8 (8)
#6	6 (6)	6 (6)	6 (6)	6 (6)	6 (6)	*6 (6) [90%, <1%]
#7	6 (6)	6 (6)	6 (6)	6.38 (6)	6 (6)	6 (6)
#8	8 (8)	8 (8)	8 (8)	8.23 (8)	8 (8)	8 (8)
#9	6 (6)	6 (6)	6 (6)	6.10 (6)	6 (6)	6 (6
#10	5 (5)	5 (5)	5 (5)	5 (5)	5 (5)	*5.38 (5) [52%, <1%]
Double-Cluster						
#11	6 (6)	6 (6)	6 (6)	6.70 (6)	6 (6)	6.02 (6)
#12	5 (5)	5 (5)	5 (5)	5 (5)	5 (5)	*5 (5) [96%, 4%]
#13	6 (6)	6 (6)	6 (6)	6 (6)	6 (6)	*6.32 (6) [46%, 1%]
#14	7 (7)	7 (7)	7 (7)	7.46 (7)	7 (7)	*7 (7) [42%, <1%]
#15	7 (7)	7 (7)	7 (7)	7 (7)	7 (7)	*7 (7) [100%, <1%]
#16	10 (10)	10 (10)	10 (10)	10.13 (10)	10 (10)	10 (10)
#17	7 (7)	7 (7)	7 (7)	7.28 (7)	7 (7)	7 (7)
#18	6 (6)	6 (6)	6 (6)	6 (6)	6 (6)	*6.14 (6) [86%, 1%]
#19	7 (7)	7 (7)	7 (7)	7.65 (7)	7 (7)	7.13 (7)
#20	7 (7)	7 (7)	7 (7)	7.69 (7)	7 (7)	7.02 (7)
Total	136 (136)	136 (136)	136 (136)	142.14 (136)	136 (136)	137.03 (136)
Rank	(1)	(1)	(1)	(6)	(1)	(5)

Table 4.5 Summary of algorithm performance on the 20 Social Networks. All statistics are collected from 50 runs on each instance. Asterisks in the rightmost column indicate where the backtracking algorithm was able to produce a provably optimal solution. In these cases, the square brackets indicate the % of runs where this occurred, and the average % of the computation limit that this took

As before, each algorithm was executed 50 times on each instance. The relatively straightforward outcomes of these trials are summarised in Table 4.5. Here, we see that the number of colours needed for these problems ranges from five to ten, though no obvious correlations exist to suggest any links with instance size, density, or the presence of clusters. We also see that the HEA, HC, TabuCol, and PartialCol methods have all produced the best observed (or optimal) solutions for all instances in all runs. It seems, therefore, that the underlying structures and relative sparsity of these graphs make their solving relatively "easy" with these algorithms.

In addition, for six of the instances, the backtracking algorithm has managed to find provably optimal solutions, though this does not occur in all runs. Indeed, when this does happen, it seems to occur early in the process (<5% of the computation limit), suggesting that the random elements of the algorithm can have a large effect on the structure of the search tree. We also observe the poor performance of

AntCol, which seems to be due to the negative performance features noted in the previous subsection, with a high-quality solution either being produced very quickly (in the first cycle), or not at all.

4.3 Conclusions

As we have seen, the results of the comparison above reveal a complicated picture, with different algorithms outperforming others on different occasions. This suggests that the underlying structures of graphs are often critical in an algorithm's resultant performance. In terms of overall patterns we offer the following observations:

- Algorithms that rely solely on local search (in this case TabuCol and PartialCol) often struggle with instances whose cost landscapes are "spiky", commonly characterised by high coefficient of variations (CVs) in vertex degrees. On the other hand, these methods do show more promise when the degree CV is quite low, such as with random and flat graphs, suggesting that they have a natural aptitude for navigating spaces in which neighbouring solutions feature costs that are often close or equal to that of the incumbent.
- One obvious advantage of the backtracking algorithm is its ability to produce provably optimal solutions. This has occurred for a number of the real-world problem instances considered in our trials, including some rather large instances; however, predicting when this will happen seems difficult. For graphs that are more "regular" in structure, such as the random and flat instances, the performance of the backtracking algorithm is also significantly worse than that of the other approaches.
- Across the trials, HEA has proved to be by far the most consistent of the six approaches. We suggest this is due to a combination of the following attributes:
  - *The HEA operates in the space of infeasible solutions.* Unlike the HC algorithm, which only permits changes to a solution that maintain feasibility, the strategy of allowing infeasible solutions seems to offer higher levels of connectivity (and thus less restriction of movement) within the solution space, helping the algorithm to navigate its way towards high-quality solutions more effectively.
  - *The HEA makes use of global as well as local search operators.* On many occasions TabuCol performs poorly when used in isolation; however, the HEA's use of global search operators in conjunction with TabuCol seems to alleviate these problems by allowing the algorithm to regularly escape from local optima.
  - *The HEA's global search operators are robust.* Unlike AntCol's global search operator, which sometimes hinders performance, the HEA's use of recombination in conjunction with a small population of candidate solutions seems beneficial across the instances. This is despite the fact that across all of our tests, recombination was never seen to consume more than 2% of the

available run time. Note in particular that the GPX operator does not consider any problem-specific information in its operations (such as the connectivity or degree of vertices), yet it still seems to strike a useful balance between (a) altering the solution sufficiently, while (b) maintaining useful substructures formed in earlier iterations of the algorithm.

One of our intentions in this comparison has been to test the robustness of our six algorithms by executing them blindly on each problem instance. As we have seen, this has involved using the same parameter values (or methods for calculating them) across all trials. However, it should be noted that different settings may lead to better results in some cases. A broader issue concerns how me might go about reliably predicting the performance of a particular graph colouring algorithm on a previously unseen graph. Accurate predictions would obviously be useful here because, given a particular graph, we would then be able to apply the most appropriate method from the available portfolio of algorithms. Work in this area with this chapter's six algorithms has been carried out by Smith-Miles et al. (2014), who use machine learning to classify the types of graph that the different algorithms are seen to perform well with. This information is then used to help predict the best performing algorithm on future unseen problem instances.

Finally, as with Chapter 2, this chapter's comparison has been carried out using a platform independent measure of computational effort. In terms of CPU time, Table 4.6 shows the relative run times of the algorithms for a small sample of graphs. Perhaps the most striking feature is that the HEA is among one of the quickest to execute, a fact that further endorses the method. On the other hand, the ANTCOL and the HC algorithms seem to require significantly more time, apparently due to the computational overheads associated with their BUILDSOLUTION and Kempe chain operators respectively.

	$n = 250$	500	1000
TABUCOL	1346	1622	1250
PARTIALCOL	1435	1372	1356
HEA	1469	1400	1337
ANTCOL	4152	3840	4349
HC	5829	5473	5320
Bktr	6328	4794	3930

Table 4.6 Time (in seconds) to complete runs of 5×10^{11} constraint checks with random graphs $G_{n,0.5}$ using a 3.0 GHz Windows 7 PC with 3.87 GB RAM

4.4 Further Improvements to the HEA

We now conclude this chapter by looking at some of the individual elements of the HEA and proposing some ideas as to how its performance might be further improved in some cases.

4.4.1 Maintaining Diversity

In general, an important factor behind the behaviour of an evolutionary algorithm (EA) is the level of diversity that is maintained within its population during a run. Typically, in early iterations of an EA the diversity of a population will be high, allowing the algorithm to consider many different parts of the solution space. This is often known as the "exploration" phase of the algorithm. As the population evolves over time this diversity then generally falls as the algorithm "homes in" on promising regions of the solution space and seeks to search within these areas more thoroughly. This is often called the "exploitation" phase.

It is clear that when applying an EA to a computational problem, a suitable balance will need to be established between exploration and exploitation. A fall in diversity that is too slow is undesirable because the algorithm will devote too much energy into broadly scanning the whole solution space, as opposed to intensively searching specific regions within it. On the other hand, a fall in diversity that is too rapid can also be problematic because the EA will spend too much time focussing on limited regions of the solution space. The latter issue is often called *premature convergence*.

To examine the issue of diversity with the HEA for graph colouring, let us first define a metric for measuring the distance between two candidate solutions.

Definition 4.1 *Given a solution $\mathcal{S}$, let $\mathcal{P}_{\mathcal{S}} = \{\{u,v\} : u,v \in V \wedge u \neq v \wedge c(u) = c(v)\}$. The* distance *between two solutions $\mathcal{S}_1$ and $\mathcal{S}_2$ can then be evaluated using the Jaccard distance measure on the sets $\mathcal{S}_1$ and $\mathcal{S}_2$. That is:*

$$D(\mathcal{S}_1,\mathcal{S}_2) = \frac{|\mathcal{P}_{\mathcal{S}_1} \cup \mathcal{P}_{\mathcal{S}_2}| - |\mathcal{P}_{\mathcal{S}_1} \cap \mathcal{P}_{\mathcal{S}_2}|}{|\mathcal{P}_{\mathcal{S}_1} \cup \mathcal{P}_{\mathcal{S}_2}|}. \tag{4.8}$$

This distance measure gives the proportion of vertex pairs (assigned to the same colour) that exist in just one of the two solutions. Consequently, if the solutions $\mathcal{S}_1$ and $\mathcal{S}_2$ are identical, then $\mathcal{P}_{\mathcal{S}_1} \cup \mathcal{P}_{\mathcal{S}_2} = \mathcal{P}_{\mathcal{S}_1} \cap \mathcal{P}_{\mathcal{S}_2}$, giving $D(\mathcal{S}_1,\mathcal{S}_2) = 0$. Conversely, if no vertex pair is assigned the same colour, $\mathcal{P}_{\mathcal{S}_1} \cap \mathcal{P}_{\mathcal{S}_2} = \emptyset$, implying $D(\mathcal{S}_1,\mathcal{S}_2) = 1$.

Given this distance measure, we are also able to define a population diversity metric. This is defined as the mean distance between each pair of solutions in the population.

Definition 4.2 *Given a population of solutions defined as a multiset $\mathbf{S} = \{\mathcal{S}_1,\mathcal{S}_2,\ldots,$ $\mathcal{S}_{|\mathbf{S}|}\}$, the* diversity *of $\mathbf{S}$ is calculated:*

$$\text{Diversity}(\mathbf{S}) = \frac{1}{\binom{|\mathbf{S}|}{2}} \sum_{\forall \mathcal{S}_i, \mathcal{S}_j \in \mathbf{S}: i<j} D(\mathcal{S}_i, \mathcal{S}_j). \tag{4.9}$$

When applying the HEA to the graphs considered in this chapter, we found that satisfactory levels of diversity were maintained in most cases. However, for some of the timetabling problem instances we also observed that large colour classes of low-degree vertices are often formed in early stages of the algorithm, and that these quickly come to dominate the population, causing premature convergence. Indeed, as can be seen in Table 4.3, the HEA often produces inferior results with these problems.

One method by which population diversity might be prolonged in EAs is to make larger changes (mutations) to an offspring in order to increase its distance from its parents. However, this must be used with care, particularly because changes that are too large might significantly worsen a solution, undoing much of the work carried out in previous iterations of the algorithm. For the HEA, one obvious way of making more changes to a solution is to increase the iteration limit of the local search procedure. However, although this might allow further improvements to be made to a solution, it might also slow the algorithm unnecessarily.

An alternative method for maintaining diversity might be to alter the HEA's recombination operator so that it works exclusively with proper colourings. As noted in Section 4.1.3, the GPX operator considers candidate solutions in which clashes are permitted; however, in practice this could allow large colour classes containing clashes to be unduly promoted in the population, when perhaps the real emphasis should be on the promotion of large *independent sets*. Consequently, we might refine the GPX operator by first removing all clashing vertices from each parent before performing recombination. This implies that, before assigning missing vertices to random colours, an offspring will always be proper. A further effect is that a greater number of vertices will usually need to be recoloured because the vertices originally removed from the parents may also be missing in the resultant offspring. Hence the resultant offspring will tend to be less similar to its parents.

If the above option is chosen, then before randomly reassigning missing vertices to colours, we also have the opportunity to alter the partial proper solution using Kempe chain interchanges. Recall from Theorem 3.1 that this operator, when applied to a proper solution, does not introduce any clashes. Thus we are provided with a mechanism by which we can make changes to a solution without compromising its quality in any way.

To illustrate the potential effects of this latter scheme, Figure 4.11 shows the levels of diversity that exist in the HEA's population for the first 3,000 iterations of a run using the timetabling graph car-s-91, which has a chromatic number of 28. When using the original HEA, the population has converged at around 500 iterations and, as we saw in Table 4.3, the algorithm produces solutions using more than 29 colours on average. On the other hand, by applying a series of random Kempe chain moves ($2k$ moves per iteration in this case), population diversity is maintained at a much higher level. In our tests this modification enabled the algorithm to determine optimal 28-colourings in all runs.

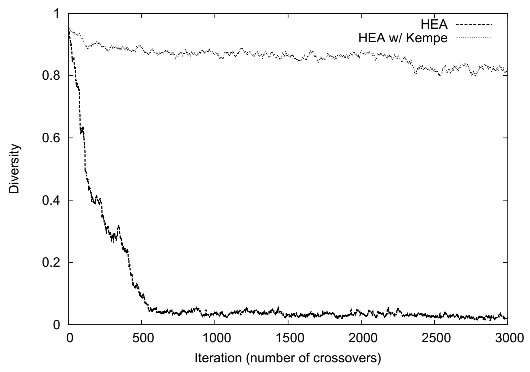

Fig. 4.11 Population diversity using a population of size 10 with the timetabling problem instance car-s-91, using $k = 28$

We should note that using Kempe chain interchanges in this way is not always beneficial, however. For instance, similar tests to the above were also carried using random and flat graphs. When using a suitably low value for k in these cases, we found that the Kempe chain interchange operator was usually unable to alter the underlying structures of solutions because its application nearly always resulted in colour relabellings (or in other words, the bipartite graphs induced by each pair of colour classes in these solutions were nearly always connected, giving total Kempe chains). As an aside, it would be interesting to investigate whether this property in a solution gives any clues as to how close it is to the optimal solution.

Note that within this book's suite of graph colouring algorithms, the HEA contains run-time options for outputting the population diversity and for applying Kempe chain interchanges in the manner described above. (Refer to the algorithm user guide in Appendix A.1 for further information.)

4.4.2 Recombination

Since the proposal of the GPX by Galinier and Hao (1999), further recombination operators based on their scheme have also been suggested, differing primarily on the criteria used for deciding which colour classes to copy to the offspring. Porumbel et al. (2010), for example, suggest that instead of choosing the largest available colour class at each stage of the recombination process, classes with the *least number of clashes* should be prioritised, with class size and information regarding the degrees of the vertices then being used to break ties. Lü and Hao (2010a), on the other hand, have proposed extending the GPX operator to allow more than two parents to play a part in producing a single offspring. In this multi-parent operator, offspring are constructed in the same manner as the GPX, except that at each stage the largest colour class from *multiple parents* is chosen to be copied into the offspring. The intention behind this increased choice is that larger colour classes will be identified, resulting in fewer uncoloured vertices once the k colour classes have been constructed. In order to prohibit too many colours being inherited from one particular parent, the authors make use of a parameter q, specifying that if the ith colour class in an offspring is copied from a particular parent, then this parent should not be considered for a further q colours. Note that GPX is simply an application of this operator using two parents with $q = 1$.

Another method of recombination with the graph colouring problem involves considering the individual assignments of vertices to colours as opposed to their partitions. Here, a natural way of representing a solution is to use a vector $(c(v_1), c(v_2), \ldots, c(v_n))$, where $c(v_i)$ gives the colour of vertex v_i. However, it has long been argued that this sort of approach has disadvantages, not least because it leads to a solution space that is far larger than it needs to be, since any solution using k colours can be represented in $k!$ ways (refer to Section 1.4.1). Furthermore, authors such as Falkenauer (1998) and Coll et al. (1995) have also argued that "traditional" recombination schemes such as 1-, 2-, and n-point crossover with this representation have a tendency to recklessly break up building blocks that we might want promoted in a population.

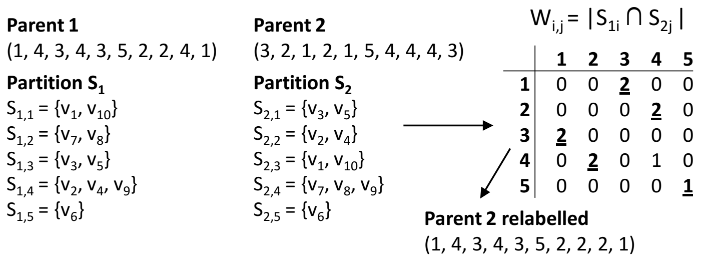

Fig. 4.12 Example of the relabelling procedure proposed by Coll et al. (1995). Here, parent 2 is relabelled $1 \rightarrow 3$, $2 \rightarrow 4$, $3 \rightarrow 1$, $4 \rightarrow 2$, and $5 \rightarrow 5$

In recognition of the perceived disadvantages of the assignment-based representation, Coll et al. (1995) have proposed a procedure for relabelling the colours of one of the parents before applying one of these "traditional" crossover operators. Consider two (not necessarily feasible) parent solutions represented as partitions: $\mathcal{S}_1 = \{S_{1,1}, \ldots, S_{1,k}\}$ and $\mathcal{S}_2 = \{S_{2,1}, \ldots, S_{2,k}\}$. Now, using $\mathcal{S}_1$ and $\mathcal{S}_2$, a complete bipartite graph $K_{k,k}$ is formed. This bipartite graph has k vertices in each partition, and the weights between two vertices from different partitions is defined as $W_{i,j} = |S_{1,i} \cap S_{2,j}|$. Given $K_{k,k}$, a maximum weighted matching can then be determined using any suitable algorithm (such as the Hungarian algorithm (Munkres, 1957) or Auction algorithm (Bertsekas, 1992)), and this matching can be used to relabel the colours in one of the parents. Figure 4.12 gives an example of this procedure and shows how the second parent can be altered so that its colour labellings maximally match those of the first parent. In this example we see that the colour classes $\{v_1, v_{10}\}$, $\{v_3, v_5\}$, and $\{v_6\}$ occur in both parents and will be preserved in any offspring produced via a traditional operator such as uniform crossover. However, this will not always be the case and will depend very much on the best matching available in each case.

An interesting point regarding the structure of solutions and the resultant effects of recombination has also been raised by Porumbel et al. (2010). Specifically, they propose that when solutions involve a small number of large colour classes (such as solutions to sparse graphs), good quality solutions will tend to occur through the identification of large independent sets, perhaps suggesting that the GPX and its multi-parent variant are naturally suited in these cases. On the other hand, if a solution involves many small colour classes, quality seems to be determined more through the identification of good *combinations* of independent sets.

Parent S_1	**Parent S_2**		**Offspring S'**
$S_{1,1} = \{v_1, v_{10}\}$	$S_{2,1} = \{\mathbf{v_1}, \mathbf{v_9}\}$		$S'_1 = \{v_1, v_{10}\}$
$S_{1,2} = \{v_7, v_8\}$	$S_{2,2} = \{\mathbf{v_7}\}$	$\longrightarrow$	$S'_2 = \{v_7, \cancel{v_8}\}$
$S_{1,3} = \{v_3, v_5\}$	➧$S_{2,3} = \{\mathbf{v_3}, \mathbf{v_5}\}$		$S'_3 = \{\mathbf{v_3}, \mathbf{v_5}\}$
$S_{1,4} = \{v_2, v_4, v_9\}$	➧$S_{2,4} = \{\mathbf{v_2}, \mathbf{v_4}, \mathbf{v_8}\}$		$S'_4 = \{\mathbf{v_2}, \mathbf{v_4}, \mathbf{v_8}\}$
$S_{1,5} = \{v_6\}$	$S_{2,5} = \{\mathbf{v_6}, \mathbf{v_{10}}\}$		$S'_5 = \{v_6\}$ **Uncoloured** $= \{v_9\}$

Fig. 4.13 Demonstration of the GGA recombination operator. Here, the colour classes in parent 2 have been labelled to maximally match those of parent 1

To these ends a further recombination operator for graph colouring is also proposed by Lewis (2015) which, unlike GPX, shows no bias towards offspring inheriting larger colour classes, or towards offspring inheriting half of its colour classes from each parent. An example of this operator is given in Figure 4.13. Given two parents, the colour classes in the second parent are first relabelled using Coll et al.'s procedure from above. Using the partition-based representations of these solutions, a subset of colour classes from the second parent is then selected randomly, and these replace the corresponding colours in a copy of the first parent. Duplicate vertices are then removed from colour classes originating from the first parent, and any uncoloured vertices are assigned to random colour classes. Tests by Lewis (2015) indicate that this recombination operator can produce marginally better solutions than the GPX operator when colour classes are small (approximately five vertices per colour), though worse results are also seen to occur in other cases.

Note that the recombination operators listed in this subsection are also included as run-time options within this book's suite of graph colouring algorithms (see Appendix A.1).

4.4.3 Local Search

Finally, from the analyses in this chapter it is apparent that graph colouring algorithms such as the HEA usually benefit when used in conjunction with an appropriate local search procedure. For algorithms operating in the space of complete improper solutions, this has often been provided by the TABUCOL algorithm. The tabu search metaheuristic seems to be very suitable for this purpose because, by extending the steepest descent algorithm, it allows rapid improvements to be made to a solution.

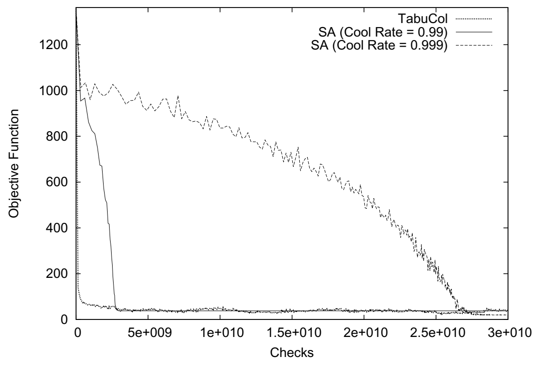

Fig. 4.14 Example run profiles of TABUCOL and an analogous SA algorithm using a random graph $G_{1000,0.5}$ with $k = 86$. Here the SA algorithm uses an initial value for $t = 0.7$, with $z = 500,000$

To contrast this, consider the rates of improvement achieved by an analogous simulated annealing algorithm that uses the same neighbourhood operator as TABUCOL but which follows the pseudocode given in Figure 3.5. For this algorithm, values need to be determined for the initial temperature t, the cooling rate α, and the frequency of temperature updates z. Figure 4.14 compares the run profile of TABUCOL to this simulated annealing algorithm on an example random graph. It can be seen that TABUCOL quickly reduces the objective function (Equation (4.1)), while the SA approach takes much longer. In addition, the SA algorithm seems quite sensitive to adjustments in its parameters, with inappropriate values potentially hindering performance. On the other hand, it is well known that when the temperature is reduced more slowly, runs of SA tend to produce better quality solutions (van Laarhoven and Aarts, 1987). Hence, with extended run times SA may have the potential to produce superior solutions to TABUCOL in some cases.