Chapter 2
Bounds and Constructive Algorithms

Towards the end of Chap. 1 we saw a variety of different types of graphs that are relatively straightforward to colour optimally, including complete graphs, bipartite graphs, cycle and wheel graphs, and grid graphs. With regard to the chromatic number, we also saw that it is easy to determine when $\chi(G) = 1$ (G is an empty graph), and when $\chi(G) = 2$ (G is bipartite). But can we go further than this? In this chapter we review and analyse a number of fast constructive algorithms for the graph colouring problem. We also make statements on how we are able to bound the chromatic number.

The fact that graph colouring is an intractable problem implies that there is a limited amount that we can say about the chromatic number of an arbitrary graph in general. One simple rule is that, given a graph G with n vertices and m edges, if $m > \lfloor n^2/4 \rfloor$ then $\chi(G) \geq 3$, since any graph fitting this criteria must contain a triangle and therefore cannot be bipartite (Bollobás, 1998); however, even the problem of deciding whether $\chi(G) = 3$ is NP-complete for arbitrary graphs.

In spite of this rather bleak situation, a variety of heuristic-based approximation algorithms are available for graph colouring that are often able to produce very pleasing results. In this chapter we will consider three fast constructive methods which operate by assigning each vertex to a colour one at a time using rules that are intended to keep the overall number of colours as small as possible. As we will see, for certain graph topologies some of these algorithms turn out to be exact, though in most cases they only produce approximate solutions. The first of these algorithms, the so-called GREEDY algorithm, is perhaps the most fundamental method in the field of graph colouring and is also useful for establishing bounds on the chromatic number. Towards the end of the chapter we also present an empirical comparison of the three constructive algorithms in order to provide information on their relative strengths and weaknesses.

At this point it is useful to introduce some further graph terminology. Recall that a graph $G = (V, E)$ is defined by a vertex set V of n vertices and an edge set E of m edges.

Definition 2.1 *If $\{u, v\} \in E$, vertices u and v are said to be* adjacent. *Vertices v and u are also said to be* incident *to the edge $\{u, v\} \in E$. If $\{u, v\} \notin E$, then vertices u and v are* nonadjacent.

R.M.R. Lewis, *A Guide to Graph Colouring*,
DOI 10.1007/978-3-319-25730-3_2

Definition 2.2 *The* neighbourhood *of a vertex v, written* $\Gamma_G(v)$*, is the set of vertices adjacent to v in the graph G. That is,* $\Gamma_G(v) = \{u \in V : \{v,u\} \in E\}$*. The* degree *of a vertex v is the cardinality of its neighbourhood set,* $|\Gamma_G(v)|$*, usually written* $\deg_G(v)$*. When the graph being referred to is made clear by the text, these can be written in their shorter forms,* $\Gamma(v)$ *and* $\deg(v)$*, respectively.*

Definition 2.3 *The* density *of a graph* $G = (V,E)$ *is the ratio of the number of edges to the number of pairs of vertices. For a simple graph with no loops this is calculated* $m/((n(n-1))/2)$*. Graphs with low densities are often referred to as* sparse *graphs; those with high densities are known as* dense *graphs.*

Definition 2.4 *A graph* $G' = (V',E')$ *is a* subgraph *of G, denoted by* $G' \subseteq G$*, if* $V' \subseteq V$ *and* $E' \subseteq E$*. If* G' *contains all edges of G that join two vertices in* V' *then* G' *is said to be the graph* induced *by* V'*.*

Definition 2.5 *Let* $W \subseteq V$*, then* $G - W$ *is the subgraph obtained by deleting the vertices in W from G, together with the edges incident to them.*

Definition 2.6 *A* path *is a sequence of edges that connect a sequence of distinct vertices. A path between two vertices u and v is called a uv-*path*. If a uv-path exists between all pairs of vertices* $u, v \in V$*, then G is said to be* connected*; otherwise it is* disconnected*.*

Definition 2.7 *The* length *of a uv-path* $P = (u = v_1, v_2, \ldots, v_l = v)$*, is the number of edges it contains, equal to* $l - 1$*. The* distance *between two vertices u and v is the minimal path length between u and v.*

Definition 2.8 *A* cycle *is a uv-path for which* $u = v$*. All other vertices in the cycle must be distinct. A graph containing no cycles is said to be* acyclic*.*

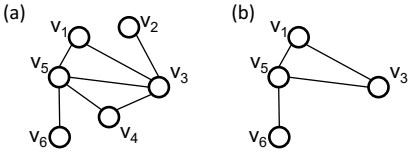

Fig. 2.1 (a) A graph G, and (b) a subgraph G' of G

To illustrate these definitions, Figure 2.1(a) shows a graph G where, for example, vertices v_1 and v_3 are adjacent, but v_1 and v_2 are nonadjacent. The neighbourhood of v_1 is $\Gamma(v_1) = \{v_3, v_5\}$, giving $\deg(v_1) = 2$. The density of G is $7/(1/2 \times 6 \times 5) = 0.467$. The subgraph G' in Figure 2.1(b) has been created via the operation $G - \{v_2, v_4\}$, and in this case both G and G' are connected. Paths in G from, for example, v_1 to v_6 include $(v_1, v_3, v_4, v_5, v_6)$ (of length 4) and (v_1, v_5, v_6) (of length 2). Since the latter path is also the shortest path between v_1 to v_6, the distance between these vertices is also 2. Cycles also exist in both G and G', such as (v_1, v_3, v_5, v_1).

2.1 The Greedy Algorithm

Recall the example from Section 1.1.1 where we sought to partition some students into a minimal number of groups for a team building exercise. The process we used to try and achieve this is known as the GREEDY algorithm, which is one of the simplest but most fundamental heuristic algorithms for graph colouring. The algorithm operates by taking vertices one by one according to some (possibly arbitrary) ordering and assigns each vertex its first available colour. Because this is a heuristic algorithm, the solutions it produces may very well be suboptimal; however, it can also be shown that GREEDY can produce an optimal solution for any graph given the correct sequence of vertices (see Theorem 2.2 below). As a result, various algorithms for graph colouring have been proposed that seek to find such orderings of the vertices (see Chapter 3).

```
    GREEDY (S ← ∅, π)
(1) for i ← 1 to |π| do
(2)     for j ← 1 to |S|
(3)         if (S_j ∪ {π_i}) is an independent set then
(4)             S_j ← S_j ∪ {π_i}
(5)             break
(6)         else j ← j + 1
(7)     if j > |S| then
(8)         S_j ← {π_i}
(9)         S ← S ∪ S_j
```

Fig. 2.2 The GREEDY algorithm for graph colouring

Pseudocode for the GREEDY algorithm is given in Figure 2.2. To start, the algorithm takes an empty solution $\mathcal{S} = \emptyset$ and an arbitrary permutation of the vertices π. In each outer loop the algorithm takes the ith vertex in the permutation, π_i, and attempts to find a colour class $S_j \in \mathcal{S}$ into which it can be inserted. If such a colour class currently exists in $\mathcal{S}$, then the vertex is added to it and the process moves on to consider the next vertex π_{i+1}. If not, lines (8–9) of the algorithm are used to create a new colour class for the vertex. An example run of the algorithm on a small graph is shown in Figure 2.3.

Let us now estimate the computational complexity of the GREEDY algorithm with regard to the number of constraint checks that are performed. We see that one vertex is coloured at each iteration, meaning $n = |\pi|$ iterations of the algorithm are required in total. At the ith iteration $(1 \leq i \leq n)$, we are concerned with finding a feasible colour for the vertex π_i. In the worst case this vertex will clash with all vertices that have preceded it in π, meaning that $(i-1)$ constraint checks will be performed before a suitable colour is determined. Indeed, if the graph we are colouring is the complete graph K_n, the worst case will occur for all vertices; hence

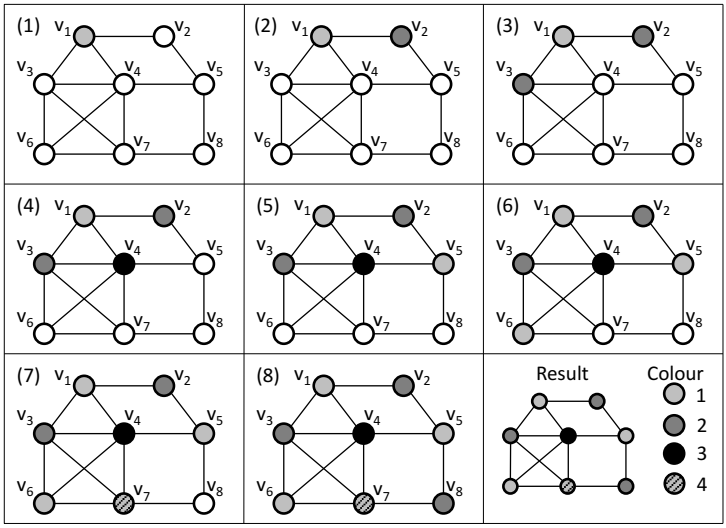

Fig. 2.3 Example application of GREEDY using the permutation $\pi_i = v_i$ $(1 \leq i \leq n)$. Here, uncoloured vertices are shown in white

a total of $0+1+2+\ldots+(n-1)$ constraint checks will be performed. This gives GREEDY an overall worst-case complexity $\mathcal{O}(n^2)$.

In practice, the GREEDY algorithm produces feasible solutions quite quickly; however, these solutions can often be quite poor in terms of the number of colours that the algorithm requires compared to the chromatic number. Consider, for example, the bipartite graph $G = (V_1, V_2, E)$ in which n is even and where the vertex sets and edge set are defined $V_1 = \{v_1, v_3, \ldots, v_{n-1}\}$, $V_2 = \{v_2, v_4, \ldots, v_n\}$, and $E = \{\{v_i, v_j\} : v_i \in V_1 \wedge v_j \in V_2 \wedge i+1 \neq j\}$. Figure 2.4 shows examples of such a graph using $n = 10$. Clearly for $n \geq 4$ such graphs will have a chromatic number $\chi(G) = 2$ because V_1 and V_2 constitute independent sets. However, colouring this graph using GREEDY with the permutation $\pi = (v_1, v_2, v_3, \ldots, v_n)$ will actually lead to a solution using $n/2$ colours, as Figure 2.4(a) illustrates. On the other hand, a permutation of the form $\pi = (v_1, v_3, \ldots, v_{n-1}, v_2, v_4, \ldots, v_n)$ will give the optimal solution shown in Figure 2.4(b). Clearly then, the order that the vertices are fed into the GREEDY algorithm can be very important.

One very useful feature of the GREEDY algorithm involves using existing feasible colourings of a graph to help generate new permutations of the vertices which can then be fed back into the algorithm. Consider the situation where we have a feasible colouring $\mathcal{S}$ of a graph G. Consider further a permutation π of G's vertices that has been generated such that the vertices occurring in each colour class of $\mathcal{S}$ are placed into adjacent locations in π. If we now use this permutation with GREEDY, the result will be a new solution $\mathcal{S}'$ that uses no more colours than $\mathcal{S}$, but possibly fewer. This is stated more concisely as follows:

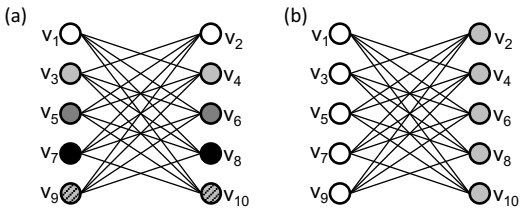

Fig. 2.4 Two different colourings of a bipartite graph achieved by the GREEDY algorithm

Theorem 2.1 *Let $\mathcal{S}$ be a feasible colouring of a graph G. If each colour class $S_i \in \mathcal{S}$ (for $1 \leq i \leq |\mathcal{S}|$) is considered in turn, and all vertices are fed one by one into the greedy algorithm, the resultant solution $\mathcal{S}'$ will also be feasible, with $|\mathcal{S}'| \leq |\mathcal{S}|$.*

Proof. Because $\mathcal{S} = \{S_1, \ldots S_{|\mathcal{S}|}\}$ is a feasible solution, each set $S_i \in \mathcal{S}$ is an independent set. Obviously any subset $T \subseteq S_i$ is also an independent set. Now consider an application of GREEDY using $\mathcal{S}$ to build a new candidate solution $\mathcal{S}'$. In applying this algorithm, each set $S_1, \ldots, S_{|\mathcal{S}|}$ is considered in turn, and all vertices $v \in S_i$ are assigned one by one to some set $S'_j \in \mathcal{S}'$ according to the rules of GREEDY (that is, v is first considered for inclusion in S'_1, then S'_2, and so on). Considering each vertex $v \in S_i$, two situations and resultant actions will occur in the following order of priority:

Case 1: An independent set $S'_{j<i} \in \mathcal{S}'$ exists such that $S'_j \cup \{v\}$ is also an independent set. In this case v will be assigned to the jth colour class in $\mathcal{S}'$.

Case 2: An independent set $S'_{j=i} \in \mathcal{S}'$ exists such that $S'_j \cup \{v\}$ is also an independent set.

In both cases it is clear that v will always be assigned to a set in $\mathcal{S}'$ with an index that is less than or equal to that of its original set in $\mathcal{S}$. Of course, if a situation arises by which *all* items in a particular set S_i are assigned according to Case 1, then at termination of GREEDY, $\mathcal{S}'$ will contain fewer colours than $\mathcal{S}$.

Now assume that it is necessary to assign a vertex $v \in S_i$ to a set $S'_{j>i}$. For this to occur, it is first necessary that the proposed actions of Cases 1 and 2 (i.e., adding v to a set $S'_{j\leq i}$) cause a clash. However, $S'_i \subset S_i$ and is therefore an independent set. By definition, $S'_i \cup \{v\} \subseteq S_i$ is also an independent set, contradicting the assumption. □

To show these concepts in action, the colouring shown in Figure 2.5(a) has been generated by the GREEDY algorithm using the permutation $\pi = (v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8)$, giving the 4-colouring $\mathcal{S} = \{\{v_1, v_4, v_8\}, \{v_2, v_7\}, \{v_3, v_5\}, \{v_6\}\}$. This solution might then be used to form a new permutation $\pi = (v_1, v_4, v_8, v_2, v_7, v_3, v_5, v_6)$ which could then be fed back into the algorithm. However, our use of sets in defining a solution $\mathcal{S}$ means that we are free to use any ordering of the colour classes in $\mathcal{S}$ to form π, and indeed any ordering of the vertices within each colour class. One alternative permutation of the vertices formed from solution $\mathcal{S}$ in this way is therefore $\pi = (v_2, v_7, v_5, v_3, v_6, v_4, v_8, v_1)$. This permutation has been used with GREEDY

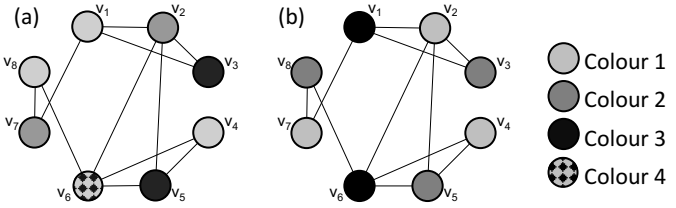

Fig. 2.5 Feasible 4- and 3-colourings of a graph

to give the solution shown in Figure 2.5(b), which we see is using fewer colours than the solution from which it was formed.

These concepts give rise to the following theorem:

Theorem 2.2 *Let G be graph with an optimal graph colouring solution $\mathcal{S} = \{S_1, \ldots, S_k\}$, where $k = \chi(G)$. Then there are at least*

$$\chi(G)! \prod_{i=1}^{\chi(G)} |S_i|! \tag{2.1}$$

permutations of the vertices which, when fed into GREEDY*, will result in an optimal solution.*

Proof. This arises immediately from Theorem 2.1: Since $\mathcal{S}$ is optimal, an appropriate permutation can be generated from $\mathcal{S}$ in the manner just described. Moreover, because the colour classes and vertices within each colour class can themselves be permuted, the above formula holds. □

Note that if $\chi(G) = 1$ or $\chi(G) = n$ then, trivially, the number of permutations decoding into an optimal solution will be $n!$. That is, every permutation of the vertices will decode to an optimal colouring using GREEDY.

2.2 Bounds on the Chromatic Number

In this section we now review some of the upper and lower bounds that can be stated about the chromatic number of a graph. Some of the bounds that we cover make use of the GREEDY algorithm in their proofs, helping us to further understand the behaviour of the algorithm. While these upper and lower bounds can be quite useful, or even exact for some topologies, we will see that in many cases they are either too difficult to calculate, or give us bounds that are too inaccurate to be of any practical use. This latter point will be demonstrated empirically later in Section 2.5.

2.2.1 Lower Bounds

To start, we make the observation that if a graph G contains as a subgraph the complete graph K_k, then a feasible colouring of G will obviously require at least k colours. Stating this in another way, let $\omega(G)$ denote the number of vertices contained in the largest clique in G (this is sometimes known as G's *clique number*). Since $\omega(G)$ different colours will be needed to colour this clique, we deduce that $\chi(G) \geq \omega(G)$.

From another perspective, we can also consider the independent sets of a graph. Let $\alpha(G)$ denote the *independence number* of a graph G, defined as the number of vertices contained in the largest independent set in G . In this case, $\chi(G)$ must be at least $\lceil n/\alpha(G) \rceil$ since to be less than this value would imply the existence of an independent set larger than $\alpha(G)$.

These two bounds can be combined into the following:

$$\chi(G) \geq \max\{\omega(G), \lceil n/\alpha(G) \rceil\} \tag{2.2}$$

The accuracy of the bounds given in Inequality (2.2) will vary on a case to case basis. Their major drawback is the fact that the tasks of calculating $\omega(G)$ and $\alpha(G)$ are themselves NP-hard problems, namely the maximum clique problem and the maximum independent set problem. However, this does not mean that the bounds are useless: in some practical applications the sizes of the largest cliques and/or independent sets might be quite obvious from the graph's topology, or even specified as part of the problem itself (see, for example, the sport scheduling models used in Chapter 7). In other cases, we might also be able to approximate $\omega(G)$ and/or $\alpha(G)$ using heuristics or by applying probabilistic arguments.

To illustrate how we might estimate the size of a maximum clique in probabilistic terms, consider a graph G with n vertices that has been generated such that each pair of vertices is joined by an edge with probability p. Assuming independence, the probability that a subset of $x \leq n$ vertices forms a clique K_x is calculated to be $p^{\binom{x}{2}}$, since there are $\binom{x}{2}$ edges that are required to be present among the x vertices. The probability that the x vertices do not form a clique is therefore simply $1 - p^{\binom{x}{2}}$. Since there are $\binom{n}{x}$ different subsets of x vertices in G, the probability that none of these are cliques is calculated to be $(1 - p^{\binom{x}{2}})^{\binom{n}{x}}$. Hence the probability that there exists at least one clique of size x in G is defined as

$$\mathrm{P}(\exists K_x \subseteq G) = 1 - (1 - p^{\binom{x}{2}})^{\binom{n}{x}} \tag{2.3}$$

for $2 \leq x \leq n$. In practice we might use this formula to estimate a lower bound with a certain confidence. For example, we might say "with greater than 99% confidence we can say that G contains a clique of size y", where y represents the largest x value for which Equation 2.3 is greater than 0.99. We might also collect similar information on the size of the largest maximum independent set in G by simply replacing p with $p = (1 - p)$ in the above formula. We must be careful in calculating the latter,

however, because dividing n by an underestimation of $\alpha(G)$ could lead to an invalid bound that exceeds $\chi(G)$. We should also be mindful that, for larger graphs, the numbers involved in calculating Equation (2.3) might be very large indeed, perhaps requiring rounding and introducing inaccuracies.

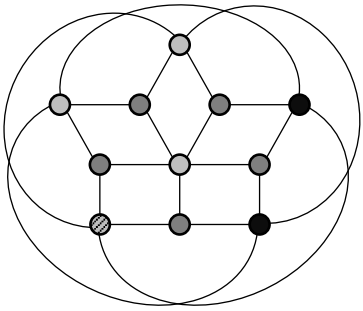

Fig. 2.6 Optimal 4-colouring of the Grötzch graph

Even if we are able to estimate or determine values such as $\omega(G)$, we must still bear in mind that they may still constitute a very weak lower bound in many cases. Consider, for example, the graph shown in Figure 2.6, known as the Grötzch graph. This graph is considered "triangle free" in that it contains no cliques of size 3 or above; hence $\omega(G) = 2$. However, as illustrated in the figure, the chromatic number of the Grötzch graph is four: double the lower bound determined by $\omega(G)$. In fact, the Grötzch graph is the smallest graph in a set graphs known as the Mycielskians, named after their discoverer Jan Mycielski (1955). Mycielskian graphs demonstrate the potential inaccuracies involved in using $\omega(G)$ as a lower bound by showing that for any $q \geq 1$ there exists a graph G with $\omega(G) = 2$ but for which $\chi(G) > q$. Hence we can encounter graphs for which $\omega(G)$ gives us a lower bound of 2, but for which the chromatic number can actually be arbitrarily large.

2.2.1.1 Bounds on Interval Graphs

While topologies such as the Mycielskian graphs demonstrate the potential for $\omega(G)$ to produce very poor lower bounds, in other cases this bound turns out to be both exact and easy to calculate. One practical application where this occurs is with *interval graphs*. Given a set of intervals defined on the real line, an interval graph is defined as a graph in which adjacent vertices correspond to overlapping intervals. More formally:

Definition 2.9 *Let $\mathcal{I} = \{I_1, \ldots, I_n\}$ be a set of intervals defined on the real line such that each interval $I_i = \{x \in \mathbb{R} : a_i \leq x < b_i\}$, where a_i and b_i define the start and end values of interval I_i. The* interval graph *of $\mathcal{I}$ is the graph $G = (V, E)$ for which $V = \{v_1, \ldots, v_n\}$ and where $E = \{\{v_i, v_j\} : I_i \cap I_j \neq \emptyset\}$.*

An example interval graph has already been provided in Section 1.1.3 where we sought to assign taxi journeys with known start and end times to a minimal number of vehicles. Figure 1.5(a) in this section shows ten taxi journeys corresponding to ten intervals over the real line (representing time in this case). These intervals are then used to construct the interval graph shown in Figure 1.5(b).

One feature of interval graphs are that they are known to contain a "perfect elimination ordering". This is defined as an ordering of the vertices such that, for every vertex, all of its neighbours to the left of it in the ordering form a clique.

Theorem 2.3 *Every interval graph G has a perfect elimination ordering.*

Proof. To start, arrange the intervals of $\mathcal{I}$ in ascending order of start values, such that $a_1 \leq a_2 \leq \ldots \leq a_n$. Now label the vertices $v_1, v_2, \ldots, v_n$ to correspond to this ordering. This implies that for any vertex v_i, the corresponding intervals of all neighbours to its left in the ordering must contain the value a_i; hence all pairs of v_i's neighbours must also share an edge, thereby forming a clique. □

The presence of a perfect elimination ordering is demonstrated in the example interval graph in Figure 1.5. Here we see, for example, that v_3 forms a clique of size 3 with its neighbours v_2 and v_1, and that v_{10} forms a clique of size 2 with its neighbour v_9. The fact that all interval graphs contain a perfect elimination ordering allows us to produce optimal colourings to such graphs according to the following theorem.

Theorem 2.4 *Let G be a graph with a perfect elimination ordering. An optimal colouring for G is obtained by labelling the vertices $v_1, \ldots, v_n$ such that $a_1 \leq a_2 \leq \ldots \leq a_n$, and then applying the* GREEDY *algorithm with the permutation $\pi_i = v_i$, for $i \leq i \leq n$. Moreover, $\chi(G) = \omega(G)$.*

Proof. During execution of GREEDY, each vertex $v_i = \pi_i$ is assigned to the lowest indexed colour not used by any of its neighbours preceding it in π. Clearly, each vertex has less than $\omega(G)$ neighbours. Hence at least one of the colours labelled 1 to $\omega(G)$ must be feasible for v_i. This implies $\chi(G) \leq \omega(G)$. Since $\omega(G) \leq \chi(G)$, this gives $\chi(G) = \omega(G)$. □

The optimal 3-colouring provided in Figure 1.5(c) shows the result of this colouring process using the permutation $\pi = (v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10})$.

More generally, graphs featuring perfect elimination orderings are usually known as *chordal* graphs. All interval graphs are therefore a type of chordal graph. The problem of determining whether a graph is chordal or not can be achieved in polynomial time by algorithms such as lexicographic breadth-first search (Rose et al., 1976). Hence any chordal graph can be recognised and coloured optimally in polynomial time.

2.2.2 Upper Bounds

Upper bounds on the chromatic number are often derived by considering the degrees of vertices in a graph. For instance, when a graph has a high density (that is, a high proportion of vertex pairs that are neighbouring), often a larger number of colours will be needed because a greater proportion of the vertex pairs will need to be separated into different colour classes. This admittedly rather weak-sounding proposal gives rise to the following theorem.

Theorem 2.5 *Let G be a connected graph with maximal degree $\Delta(G)$ (that is, $\Delta(G) = \max\{\deg(v) : v \in V\}$). Then $\chi(G) \leq \Delta(G) + 1$.*

Proof. Consider the behaviour of the GREEDY algorithm. Here the ith vertex in the permutation π, namely π_i, will be assigned to the lowest indexed colour class that contains none of its neighbouring vertices. Since each vertex has at most $\Delta(G)$ neighbours, no more than $\Delta(G) + 1$ colours will be needed to feasibly colour all vertices of G. □

Another bound concerning vertex degrees can be calculated by examining all of a graph's subgraphs and identifying the minimal degree in each case, and then taking the maximum of these. For practical purposes this might be less useful than Theorem 2.5 for computing bounds quickly since the total number of subgraphs to consider might be prohibitively large. However, the following result still has its uses, particularly when it comes to colouring planar graphs and graphs representing circuit boards (see Chapter 5).

Theorem 2.6 *Given a graph G, suppose that in every subgraph G' of G there is a vertex with degree less than or equal to δ. Then $\chi(G) \leq \delta + 1$*

Proof. We know there is a vertex with a degree of at most δ in G. Call this vertex v_n. We also know that there is a vertex of at most δ in the subgraph $G - \{v_n\}$, which we can label v_{n-1}. Next, we can label as v_{n-2} a vertex of degree of at most δ to form the graph $G - \{v_n, v_{n-1}\}$. Continue this process until all of the n vertices have been assigned labels. Now assign these vertices to the permutation π using $\pi_i = v_i$, and apply the GREEDY algorithm. At each step of the algorithm, v_i will be adjacent to at most δ of the vertices $v_1, \ldots, v_{i-1}$ that have already been coloured; hence no more than $\delta + 1$ colours will be required. □

Let us now examine some implications of these two theorems. It can be seen that Theorem 2.5 provides tight bounds for both complete graphs, where $\chi(K_n) = \Delta(K_n) + 1 = n$, and for odd cycles, where $\chi(C_n) = \Delta(C_n) + 1 = 3$. However, such accurate bounds will not always be so forthcoming. Consider, for example, the wheel graph comprising 100 vertices, W_{100}. This features a "central" vertex of degree $\Delta(W_{100}) = 99$, meaning that Theorem 2.5 merely informs us that the chromatic number of W_{100} is less than 100, despite the fact that it is actually just four! On the other hand, for any wheel graph it is relatively easy to show that all of its subgraphs will contain a vertex with a degree of no more than 3 (i.e., $\delta = 3$). For

wheel graphs where n is even, this allows Theorem 2.6 to return a tight bound since $\delta + 1 = \chi(W_n) = 4$.

Beyond complete graphs and odd cycles, Theorem 2.5 can also be marginally strengthened due to the result of Brooks (1941). This proof is slightly more involved than that of Theorem 2.5 and requires some further definitions.

Definition 2.10 *A* component *of a graph G is a subgraph G' in which all pairs of vertices are connected by paths. A graph that is itself connected has exactly one component, comprising the whole graph.*

Definition 2.11 *A* cut vertex *v is a vertex whose removal from a graph G (together with all incident edges) increases the number of components. Thus a cut vertex of a connected graph is a vertex whose removal disconnects the graph. More generally, a* separating set *of a graph G is a set of vertices whose removal increases the number of components.*

Definition 2.12 *A graph G is said to be k-*connected *if it remains connected whenever fewer than k vertices are removed. In other words, G will only become disconnected if a* separating set *comprising k or more vertices is deleted.*

Definition 2.13 *A component of a graph is considered a* block *if it is 2-connected.*

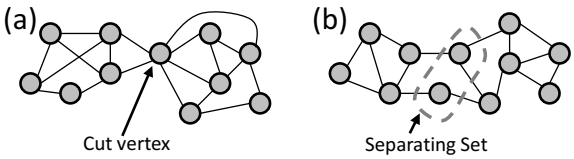

Fig. 2.7 Illustrations of a cut vertex and separating set

To illustrate these definitions, Figure 2.7(a) shows a graph G comprising one component. Removal of the indicated cut vertex would split G into two components. Figure 2.7(b) can be considered a block in that it does not contain a cut vertex (i.e., it is 2-connected). However, it is not 3-connected, because removal of the two vertices in the indicated separating set increases the number of components from one to two.

Having gone over the necessary terminology, we are now in a position to state and prove Brooks' theorem.

Theorem 2.7 (Brooks (1941)) *Let G be a connected graph with maximal degree $\Delta(G)$. Suppose further that G is not complete and not an odd cycle. Then $\chi(G) \leq \Delta(G)$.*

Proof. The theorem is obviously correct for $\Delta(G) \leq 2$. For $\Delta(G) = 0$ and $\Delta(G) = 1$, the corresponding graphs will be K_1 and K_2 respectively, and are therefore not included in the theorem. For $\Delta(G) = 2$ on the other hand, G will be a path or even

cycle (giving $\chi(G) = 2$) or will be an odd cycle, meaning it is not included in the theorem.

Assuming $\Delta(G) \geq 3$, let G be a counterexample with the smallest possible number of vertices for which the theorem does not hold: i.e., $\chi(G) > \Delta(G)$. We therefore assume that all graphs with fewer vertices than G can be feasibly coloured using $\Delta(G)$ colours.

Claim 1: G is connected. If G were not connected, then G's components would be a smaller counterexample, or all of G's components would be $\Delta(G)$-colourable.

Claim 2: G is 2-connected. If G were not 2-connected, then G would have at least one cut vertex v, and each block of G would be $\Delta(G)$-colourable. The colourings of each block could then be combined to form a feasible $\Delta(G)$-colouring.

Claim 3: G must contain three vertices v, u_1 and u_2 such that (a) u_1 and u_2 are non-adjacent; (b) both u_1 and u_2 are adjacent to v; and (c) $G - \{u_1, u_2\}$ is connected. Two cases can now be considered:

Case 1: G is 3-connected. Because G is not complete, there must be two vertices x and y that are nonadjacent. Let the shortest path between x and y in G be $x = v_0, \ldots, v_l = y$, where $l \geq 2$. Since this is the shortest path, v_0 is not adjacent to v_2, so we can choose $u_1 = v_0$, $v = v_1$ and $u_2 = v_2$. This satisfies Claim 3.

Case 2: G is 2-connected but not 3-connected. In this case, there must exist two vertices u and v such that $G - \{u, v\}$ is disconnected. This means that the graph $G - \{v\}$ contains a cut vertex (i.e., u), but there is no cut vertex in G itself. In this case, v must be adjacent to at least one vertex in every block of the graph $G - \{v\}$. Let u_1 and u_2 be two vertices in two different end blocks of $G - \{v\}$ that are adjacent to v. The vertices u_1, u_2 and v now satisfy Claim 3.

Having proved Claims 1, 2, and 3, we now construct a permutation π of the n vertices of G such that $\pi_1 = u_1$, $\pi_2 = u_2$, and $\pi_n = v$. The remaining parts of the permutation $\pi_3, \ldots \pi_{n-1}$ are then formed such that, for $3 \leq i < j \leq n-1$, the distance from π_n to π_i is greater than or equal to the distance from π_n to π_j. If we now apply GREEDY to this permutation, the vertices $\pi_1 = u_1$ and $\pi_2 = u_2$ will first both be assigned to colour S_1, because they are nonadjacent. Moreover, when we colour the vertices π_i ($3 \leq i < n$), there will always be at least one colour $S_{j \leq \Delta(G)}$ available for π_i. Finally, when we come to colour vertex $\pi_n = v$, at most $\Delta(G) - 1$ colours will have been used to colour the neighbours of v (since its neighbours u_1 and u_2 have been assigned to the same colour) and so at least one of the $\Delta(G)$ colours will be feasible for v. This shows that $\chi(G) \leq \Delta(G)$ as required. □

Having analysed the behaviour of the GREEDY algorithm and reviewed a number of bounds on the chromatic number, the following two sections will now consider two further heuristic-based constructive algorithms for the graph colouring problem. As we will see, these algorithms are guaranteed to produce optimal solutions for some simple graph topologies and also often construct solutions that improve on the upper bounds mentioned above. Later, in Chapters 3 and 4, we will also see that

these algorithms, along with GREEDY, are often used as building blocks in many of the more sophisticated algorithms available for graph colouring.

2.3 The DSATUR Algorithm

The DSATUR algorithm (abbreviated from "degree of saturation") was originally proposed by Brélaz (1979). In essence it is very similar in behaviour to the GREEDY algorithm in that it takes each vertex in turn according to some ordering and then assigns it to the first suitable colour class, creating new colour classes when necessary. The difference between the two algorithms lies in the way that these vertex orderings are generated. With GREEDY the ordering is decided before any colouring takes place; on the other hand, for the DSATUR algorithm the choice of which vertex to colour next is decided heuristically based on the characteristics of the current partial colouring of the graph. This choice is based primarily on the *saturation degree* of the vertices, defined as follows.

Definition 2.14 *Let* $c(v) = \text{NULL}$ *for any vertex* $v \in V$ *not currently assigned to a colour class. Given such a vertex* v*, the* saturation degree *of* v*, denoted by* $\text{sat}(v)$*, is the number of different colours assigned to adjacent vertices. That is,* $\text{sat}(v) = |\{c(u) : u \in \Gamma(v) \wedge c(u) \neq \text{NULL}\}|$

```
    DSATUR (S ← ∅, X ← V)
(1) while X ≠ ∅ do
(2)     Choose v ∈ X
(3)     for j ← 1 to |S|
(4)         if (S_j ∪ {v}) is an independent set then
(5)             S_j ← S_j ∪ {v}
(6)             break
(7)         else j ← j + 1
(8)     if j > |S| then
(9)         S_j ← {v}
(10)        S ← S ∪ S_j
(11)    X ← X − {v}
```

Fig. 2.8 The DSATUR algorithm for graph colouring

Pseudocode for the DSATUR algorithm is shown in Figure 2.8. It can be seen that the majority of the algorithm is the same as the GREEDY algorithm in that once a vertex has been selected, a colour is found by simply going through each colour class in turn and stopping when a suitable one has been found. Consequently, the worst-case complexity of DSATUR is the same as GREEDY at $\mathcal{O}(n^2)$, although in practice some extra bookkeeping is required to keep track of the saturation degrees of the uncoloured vertices.

The major difference between GREEDY and DSATUR lies in lines (1), (2) and (11) of the pseudocode. Here, a set X is used to define the set of vertices currently not assigned to a colour. At the beginning of execution $X = V$. In each iteration of the algorithm the next vertex to be coloured is selected from X according to line (2), and once coloured, it is removed from X in line (11). The algorithm terminates when $X = \emptyset$.

Line (2) of Figure 2.8 provides the main power behind the DSATUR algorithm. Here, the next vertex to be coloured is chosen as the vertex in X that has maximal saturation degree. If there is more than one vertex with maximal saturation degree, then ties are broken by choosing the vertex among these with the largest degree. Any further ties can then be broken randomly. The idea behind the maximum saturation degree heuristic is that it prioritises vertices that are seen to be the most "constrained"—that is, vertices that currently have the fewest colour options available to them. Consequently, these "more constrained" vertices are dealt with by the algorithm first, allowing the less constrained vertices to be coloured later.

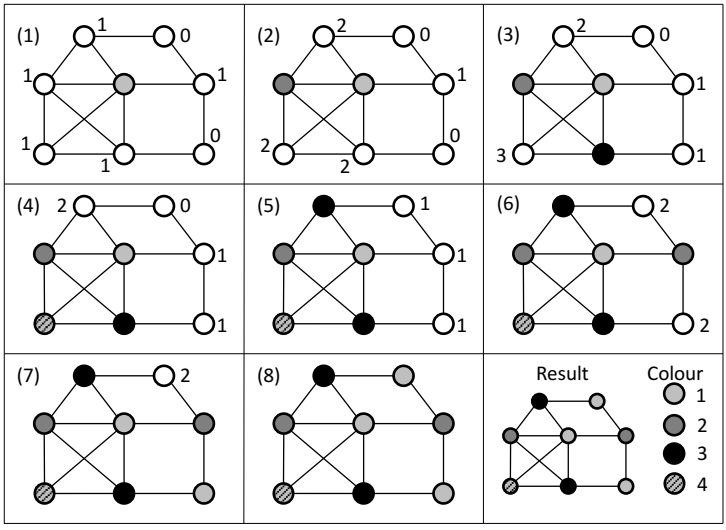

Fig. 2.9 Example application of DSATUR. Here, uncoloured vertices (members of X) are shown in white, and have their saturation degrees written alongside

Figure 2.9 shows an example run-through of the algorithm on a small graph. To begin, all vertices have a saturation degree equal to 0, so the first vertex to be coloured is the one with the highest degree. As shown in Step (1), this is assigned to colour 1. This also leads to five vertices having a saturation degree of 1, so the next vertex to be chosen is the one among these that has the highest degree. This is then assigned to colour 2 as shown in Step (2). Next, three vertices have saturation degrees of 2, so we again choose the vertex among these with the highest degree. Since colours 1 and 2 are not feasible for this vertex, it is assigned to colour 3.

This process continues as shown in the figure until a feasible colouring has been achieved.

Earlier we saw that the number of colours used in solutions produced by the GREEDY algorithm depends on the order that the vertices are fed into the procedure, with results (in terms of the number of colours used in the solution produced) potentially varying a great deal. On the other hand, the DSATUR algorithm reduces this variance by generating the vertex ordering *during* a run according to its heuristics. As a result, DSATUR's performance is more predictable. Indeed DSATUR turns out to be exact for a number of elementary graph topologies. The first of these is the bipartite graph, and to prove this claim it is first necessary to show a classical result on the structure of these graphs.

Theorem 2.8 *A graph is bipartite if and only if it contains no odd cycles.*

Proof. Let G be a connected bipartite graph with vertex sets V_1 and V_2. (It is enough to consider G as being connected, as otherwise we could simply treat each component of G separately.) Let $v_1, v_2, \ldots, v_l, v_1$ be a cycle in G. We can also assume that $v_1 \in V_1$, $v_2 \in V_2$, $v_3 \in V_1$, and so on. Hence, a vertex $v_i \in V_1$ if and only if i is odd. Since $v_l \in V_2$, this implies l is even. Consequently G has no odd cycles.

Now suppose that G is known to feature no odd cycles. Choose any vertex v in the graph and let the set V_1 be the set of vertices such that the shortest path from each member of V_1 to v is of odd length, and let V_2 be the set of vertices where the shortest path from each member of V_2 to v is even. Observe now that there is no edge joining vertices of the same set V_i since otherwise G would contain an odd cycle. Hence G is bipartite. □

This result allows us to prove the following theorem.

Theorem 2.9 (Brélaz (1979)) *The* DSATUR *algorithm is exact for bipartite graphs.*

Proof. Let G be a connected bipartite graph with $n \geq 3$. If G is not connected, it is enough to consider each component of G separately. For purposes of contradiction assume that one vertex v has a saturation degree of 2, meaning that v has two neighbours, u_1 and u_2, assigned to different colours. From these two neighbours we can build two paths which, because G is connected, will have a common vertex u. Hence we have formed a cycle containing vertices v, u_1, u_2, u and perhaps others. Since G is bipartite, the length of this cycle must be even, meaning that the u_1 and u_2 must have the same colour, contradicting our initial assumption. □

To illustrate the usefulness of this result, consider the bipartite graphs shown in Figure 2.4 earlier. Here, many permutations of the vertices used in conjunction with the GREEDY algorithm will lead to colourings using more than two colours. Indeed, in the worst case they may even lead to $(n/2)$-colourings as demonstrated in the figure. In contrast DSATUR is guaranteed to return the optimal solution bipartite graphs, as it is for some further topologies:

Theorem 2.10 *The* DSATUR *algorithm is exact for cycle and wheel graphs.*

Proof. Note that even cycles are 2-colourable and are therefore bipartite. Hence they are dealt with by Theorem 2.9. However, it is useful to consider both even and odd cycles in the following.

Let C_n be a cycle graph. Since the degree of all vertices in C_n is 2, the first vertex to be coloured, v, will be chosen arbitrarily by DSATUR. In the next $(n-2)$ steps, according to the behaviour of DSATUR a path of vertices of alternating colours will be constructed that extends from v in both clockwise and anticlockwise directions. At the end of this process, a path comprising $n-1$ vertices will have been formed, and a single vertex u will remain that is adjacent to both terminal vertices of this path. If C_n is an even cycle, $n-1$ will be odd, meaning that the terminal vertices have the same colour. Hence u can be coloured with the alternative colour. If C_n is an odd cycle, $n-1$ will be even, meaning that the terminal vertices will have different colours. Hence a u will be assigned to a third colour.

For wheel graphs W_n a similar argument applies. Assuming $n \geq 5$, DSATUR will initially colour the central vertex v_n because it features the highest degree. Since v_n is adjacent to all other vertices in W_n, all remaining vertices $v_1, \ldots, v_{n-1}$ will now have a saturation degree of 1. The same colouring process as the cycle graphs C_{n-1} then follows. □

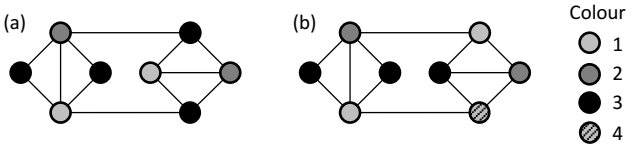

Fig. 2.10 An optimal 3-colouring (a) and a suboptimal 4-colouring produced by DSATUR (b)

Although, as these theorems show, DSATUR is exact for certain types of graph, the NP-hardness of the graph colouring problem obviously implies that it will be unable to produce optimal solutions for *all* graphs. Figure 2.10, for example, shows a small graph that, while actually being 3-colourable, will always be coloured using four colours by DSATUR, regardless of the way any random ties in the algorithm's heuristics are broken. In fact, Janczewski et al. (2001) have proved that this is the smallest such graph where this suboptimality occurs, but there are countless larger graphs where DSATUR will also not return the optimal. In other work, Spinrad and Vijayan (1984) have also identified a graph topology of $\mathcal{O}(n)$ vertices that, despite being 3-colourable, will actually be coloured using n different colours using DSATUR.

2.4 The Recursive Largest First (RLF) Algorithm

While the DSATUR algorithm for graph colouring is similar in behaviour and complexity to the classical GREEDY approach, the next constructive method we exam-

ine, the Recursive Largest First (RLF) algorithm, follows a slightly different strategy. The RLF algorithm was originally designed by Leighton (1979), in part for use in constructing solutions to large timetabling problems. The method works by colouring a graph *one colour at a time*, as opposed to one vertex at a time. In each step the algorithm uses heuristics to identify an independent set of vertices in the graph, which are then associated with the same colour. This independent set is then removed from the graph, and the process is repeated on the resultant, smaller subgraph. This process continues until the subgraph is empty, at which point all vertices have been coloured leaving us with a feasible solution. Leighton (1979) has proven the worst-case complexity of RLF to be $\mathcal{O}(n^3)$, giving it a higher computational cost than the $\mathcal{O}(n^2)$ GREEDY and DSATUR algorithms; however, this algorithm is still of course polynomially bounded.

	RLF $(\mathcal{S} \leftarrow \emptyset, X \leftarrow V, Y \leftarrow \emptyset, i \leftarrow 0)$
(1)	**while** $X \neq \emptyset$ **do**
(2)	$i \leftarrow i+1$
(3)	$S_i \leftarrow \emptyset$
(4)	**while** $X \neq \emptyset$ **do**
(5)	Choose $v \in X$
(6)	$S_i \leftarrow S_i \cup \{v\}$
(7)	$Y \leftarrow Y \cup \Gamma_X(v)$
(8)	$X \leftarrow X - (Y \cup \{v\})$
(9)	$\mathcal{S} \leftarrow \mathcal{S} \cup \{S_i\}$
(10)	$X \leftarrow Y$
(11)	$Y \leftarrow \emptyset$

Fig. 2.11 The RLF algorithm for graph colouring. Here, $\Gamma_X(v)$ denotes the subset of vertices in the set X that are adjacent to vertex v

Pseudocode for the RLF algorithm is given in Figure 2.11. In each outer loop of the process, the ith colour class S_i is build. The algorithm also 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. At the start of execution $X = V$ and $Y = \emptyset$.

Lines (4) to (8) of Figure 2.11 give the steps responsible for constructing the ith colour class S_i. To start, a vertex v from X is selected and added to S_i (i.e., v is coloured with colour i). Next, all vertices neighbouring v in the subgraph induced by X are transferred to Y, to signify that they cannot now be feasibly assigned to S_i. Finally, v and its neighbours are also removed from X, since they are not now considered candidates for inclusion in colour class S_i.

Once $X = \emptyset$, no further vertices can be added to the current colour class S_i. In lines (9) to (11) of the algorithm S_i is therefore added to the solution $\mathcal{S}$ and, if necessary, the algorithm moves on to constructing colour class S_{i+1}. To do this, all vertices in the set of uncoloured vertices Y are moved into X, and Y is emptied. Obviously, once both X and Y are empty, all vertices have been coloured.

The heuristics suggested by Leighton (1979) for selecting the next vertex $v \in X$ to colour in line (5) follow a similar rationale to those of the DSATUR algorithm in that the most "constrained" vertices are prioritised. Consequently the first vertex chosen for insertion into each colour class S_i is the member of X that has the highest degree in the subgraph induced by X. The remaining vertices v for S_i are then selected as the member of X that has the largest degree in the subgraph induced by $Y \cup \{v\}$ (that is, the vertex in X that is adjacent to the largest number of vertices in Y). As with DSATUR, any ties in these heuristics can be broken randomly.

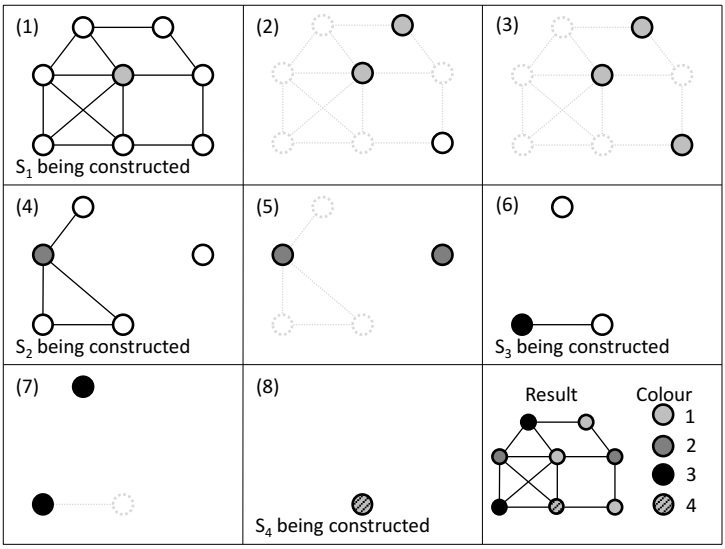

Fig. 2.12 Example application of the RLF algorithm. Here, dotted vertices denote those currently assigned to the set Y. Vertices with solid lines and no colour denote members of X

Figure 2.12 gives an example step-by-step run-through of the RLF algorithm. Steps that involve the creation of a new colour class S_i are indicated. In Step (1) the vertex v with the highest degree in the graph is added to colour class S_1. In Step (2), all vertices adjacent to v have now been moved to Y, leaving a subgraph induced by the set X which contains just two vertices, both of which are subsequently added to colour class S_1 in Steps (2) and (3). In Step (4) a new colour class is created and the process is repeated on the subgraph induced by the remaining uncoloured vertices. This continues until all vertices have been coloured, as shown.

Like DSATUR, the RLF algorithm is also exact for a number of fundamental graph topologies.

Theorem 2.11 *The* RLF *algorithm is exact for bipartite graphs.*

Proof. Let G be a connected bipartite graph with $n \geq 3$ and vertex sets V_1 and V_2. If G is disconnected, it is enough to consider each component separately.

Assume without loss of generality that vertex $v \in V_1$ has the highest degree and is therefore assigned to the first colour. Consequently all neighbours of v (that is, $\Gamma(v) \subseteq V_2$) will be added to Y. It is now sufficient to show that the next $|V_1| - 1$ vertices assigned to the first colour will all be members of V_1. This is indeed the case because, in the next $|V_1| - 1$ steps, uncoloured vertices will be selected that have the largest number of adjacent vertices in the set Y. Since uncoloured vertices from the set $V_2 - Y$ will be nonadjacent to those in Y, only vertices from V_1 will be selected. This applies until all vertices from V_1 have been assigned to the first colour. At this point the subgraph induced by V_2 will have no edges, allowing RLF to colour all remaining vertices with the second colour. □

Theorem 2.12 *The* RLF *algorithm is exact for cycle and wheel graphs.*

Proof. Even cycles are 2-colourable and are thus dealt with by Theorem 2.11. However, for convenience we shall consider both even and odd cycles in the following. Let C_n be a cycle graph with vertices $V = \{v_1, \ldots, v_n\}$ and edges $E = \{\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{n-1}, v_n\}, \{v_n, v_1\}\}$. For bookkeeping purposes, also assume that ties in the RLF selection heuristic (line (4) of Figure 2.11) are broken by taking the vertex with the lowest index, as opposed to choosing arbitrarily. It is easy to see that this theorem holds without this restriction, however.

The degree of all vertices in C_n is 2, so the first vertex to be coloured will be v_1. Consequently, neighbouring vertices v_2 and v_{n-1} are added to Y. According to the heuristics of RLF the next vertex to be coloured will be v_3, leading to v_4 being added to Y; then v_5, leading to v_6 being added to Y; and so on. At the end of this process, we will have colour class $S_1 = \{v_1, v_3, \ldots, v_{n-1}\}$ when n is even, and the colour class $S_1 = \{v_1, v_3, \ldots, v_{n-2}\}$ when n is odd. In the even case, this leaves an uncoloured subgraph with vertices $\{v_2, v_4, \ldots, v_n\}$ and no edges. Consequently RLF will assign all of these vertices to the second colour. In the odd case, we will be left with uncoloured vertices $\{v_2, v_4, \ldots, v_{n-1}, v_n\}$ together with a single edge $\{v_{n-1}, v_n\}$. Following the heuristic rules of RLF, all even-indexed vertices will then be assigned to the second colour, with v_n being assigned to the third.

For wheel graphs W_n similar reasoning applies. Assuming $n \geq 5$, the central vertex v_n will coloured first because it has the highest degree. Since v_n is adjacent to all other vertices, no further vertices can be added to this colour, so the algorithm will move on to the second colour. The remaining uncoloured vertices now form the cycle graph C_{n-1}, and the same colouring process as above follows. □

2.5 Empirical Comparison

In this section we now present a comparison of the GREEDY, DSATUR, and RLF algorithms looking particularly at their run time requirements and the quality of solutions that they tend to produce. The algorithm implementations used in these experiments can be found in the online suite of graph colouring algorithms mentioned in Section 1.6.1 and Appendix A.1.

2.5.1 Experimental Considerations

When new algorithms are proposed for the graph colouring problem, the quality of the solutions it produces will usually be compared to those achieved on the same problem instances by other preexisting methods. A development in this regard occurred in 1992 with the organisation of the Second DIMACS Implementation Challenge (http://mat.gsia.cmu.edu/COLOR/instances.html), which resulted in a suite of differently structured graph colouring problems being placed into the public domain. Since this time, authors of graph colouring papers have often used this set (or a subset of it) and have concentrated on tuning their algorithms (perhaps by altering the underlying heuristics or run time parameters) in order to achieve the best possible results.

More generally, when testing and comparing the accuracy of two graph colouring algorithms (or, indeed, any approximation/heuristic algorithms), an important question is "Are we attempting to show that Algorithm A produces better results than Algorithm B on (a) a particular problem instance, or (b) across a whole set of problem instances?" In some cases we might, for example, be given a difficult practical problem that we only need to solve once, and whose efficient solution might save lots of money or other resources. Here, it would seem sensible to concentrate on answering question (a) and spend our time choosing the correct heuristics and parameters in order to achieve the best solution possible under the given time constraints. If our chosen algorithm involves stochastic behaviour (i.e., making random choices) multiple runs of the algorithm might then be performed on the problem instance to gain an understanding of its average performance for this case. In most situations however, it is more likely that when a new algorithm is proposed, the scientific community will be more interested in question (b) being answered—that is, we will want to understand and appreciate the performance of the algorithm across a whole set of problem instances, allowing more general conclusions to be drawn.

If we choose to follow (b) above, it is first necessary to decide what *types* of graphs (i.e., what population of problems instances) we wish to make statements about. For instance, this might be the set of all 2-colourable graphs, or it could be the set of all graphs containing fewer than 1,000 vertices. Typically, populations like these will be very large, or perhaps infinite in size, and so it will be necessary to test our algorithms on randomly selected samples of these populations. Under appropriate experimental conditions, we might then be able to use the outcomes of these trials to make general statistical statements about the population itself, such as: "With $\geq 95\%$ confidence Algorithm A produces solutions with fewer colours than Algorithm B on this particular graph type".

In this section, in order to compare the performance of the GREEDY, DSATUR, and RLF algorithms, we make use of the following facts to define our population. Given a graph with n vertices, let l denote the number of vertex pairs in G. That is, $l = \binom{n}{2}$. Any graph with n vertices can therefore be represented by an l-dimensional binary vector $\mathbf{b}^{(n)}$ for which element $b_i^{(n)} = 1$ if and only if the corresponding pair of vertices are adjacent, and $b_i^{(n)} = 0$ otherwise.

Now let $\mathcal{B}^{(n)}$ define the set of all l-dimensional binary vectors. Obviously this means that $|\mathcal{B}^{(n)}| = 2^l$. The set $\mathcal{B}^{(n)}$ can therefore be considered as the set of all possible ways of connecting the vertices in an n-vertex graph. However, we must be careful in this interpretation as this is not quite the same as saying that $\mathcal{B}^{(n)}$ represents the set of all *graphs* with n vertices (which it does not), because it fails to take into account the principle of graph isomorphisms.

Consider the example in Figure 2.13, where we show two different six-dimensional binary vectors and illustrate the graphs that they represent, called G_1 and G_2 here. Note that when we come to colour G_1 and G_2 the vertex labels are of little importance and, indeed, without the labels the two graphs might be considered identical. In these circumstances G_1 and G_2 are considered *isomorphic* as there exists a way of converting one graph into the other by simply relabelling the vertices (in this example we can convert G_1 to G_2 by relabelling v_1 as v_2, v_2 as v_4, v_3 as v_3 and v_4 as v_1). Because the set $\mathcal{B}^{(n)}$ fails to take these isomorphisms into account, it must therefore be interpreted as the "set of all n-vertex graphs *and their isomorphisms*", as opposed to the set of all n-vertex graphs itself.

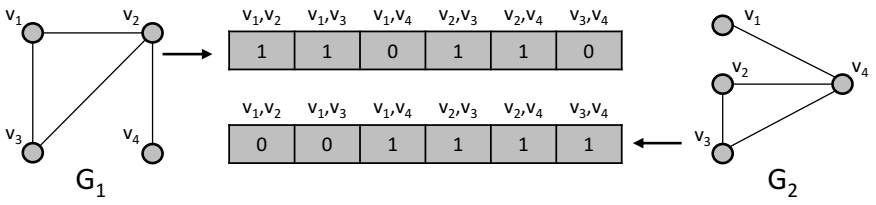

Fig. 2.13 Illustration of how different binary vectors can represent graphs that are isomorphic

To generate a single member of the set $\mathcal{B}^{(n)}$ at random (i.e., to choose an element of $\mathcal{B}^{(n)}$ such that each element is equally likely to be selected), it is simply necessary to generate an l-dimensional vector $\mathbf{b}^{(n)}$ in which each element $b_i^{(n)} = 1$ with probability 0.5, and 0 otherwise. This is the same process as producing a *random graph* with $p = 0.5$:

Definition 2.15 *A* random graph*, denoted by $G_{n,p}$, is a graph comprising n vertices in which each pair of vertices is adjacent with probability p. The degrees of the vertices in a random graph are consequently binomially distributed:* $\deg(v) \sim \mathrm{B}(n-1, p)$.

Random graphs will be the focus of our algorithm comparison in this chapter, though we will also look at other types of graphs in later chapters.

Table 2.1 Summary of results produced by the GREEDY, DSATUR and RLF algorithms on random graphs $G_{n,0.5}$

		Algorithm[a]			
n	LB[b]	GREEDY	DSATUR	RLF	UB[c]
100	9	21.14 ± 0.95	18.48 ± 0.81	17.44 ± 0.61	62.22
500	10	72.54 ± 1.33	65.18 ± 1.06	61.04 ± 0.78	284.06
1000	10	126.64 ± 1.21	115.44 ± 1.23	108.74 ± 0.90	550.76
1500	10	176.20 ± 1.58	162.46 ± 1.42	153.44 ± 0.86	841.92
2000	10	224.18 ± 1.90	208.18 ± 1.02	196.88 ± 1.10	1076.26

[a] Mean plus/minus standard deviation in number of colours, taken from runs across 50 graphs.

[b] Largest value x for which Equation (2.3) is greater than or equal to 0.99.

[c] Generated according to Theorem 2.7. Mean taken across 50 graphs.

2.5.2 Results and Analysis

Table 2.1 summarises the number of colours used in solutions produced by the GREEDY, DSATUR, and RLF algorithms for random graphs with edge probability $p = 0.5$ and varying numbers of vertices. For each value of n, 50 random graphs were generated and each algorithm was executed on it once. In applications of the GREEDY algorithm, the vertex permutation π was generated randomly. Table 2.1 also shows lower and upper bounds that were generated for these problem instances.

The results in Table 2.1 indicate that for all tested values of n, the DSATUR algorithm tends to produce solutions using fewer colours than the GREEDY algorithm. Indeed, in all five cases, these differences were seen to be significantly different.[1] In turn, significant differences were also observed between the results of DSATUR and RLF, indicating that, for all of the tested values of n, the RLF algorithm produces the best solutions across the set of all graphs and their isomorphisms.

The data in Table 2.1 also reveals that the generated lower and upper bounds seem to be some distance from the number of colours ultimately used by the algorithms. This indicates that Brooks' Theorem (2.7) tends to provide a rather inaccurate upper bound for random graphs. It also suggests two factors with regard to the lower bound: (a) that the probabilistic bound determined by Equation (2.3) is also quite inaccurate and/or (b) that the GREEDY, DSATUR, and RLF algorithms are producing solutions whose numbers of colours are some distance from the chromatic number.

The graphs shown in Figure 2.3 expand upon the results of Table 2.1 by considering a range of different values for p. Bounds are also indicated by the shaded areas. We see that the unshaded areas of these graphs are generally quite wide, with the algorithms' results falling in a fairly narrow band within these. This again indicates the inadequacy of the upper bound, particularly for larger values of n.

The differences between the three algorithms themselves across these values of p are presented more clearly in Figure 2.15. Here, the bars in the graphs show the

[1] The samples collected for each algorithm and value of n were not generally found to be derived from an underlying normal distribution according to a Shapiro-Wilk test. Consequently, statistical significance is claimed here according to the results of a nonparametric related samples Wilcoxon Signed Rank test at the 0.1% significance level.

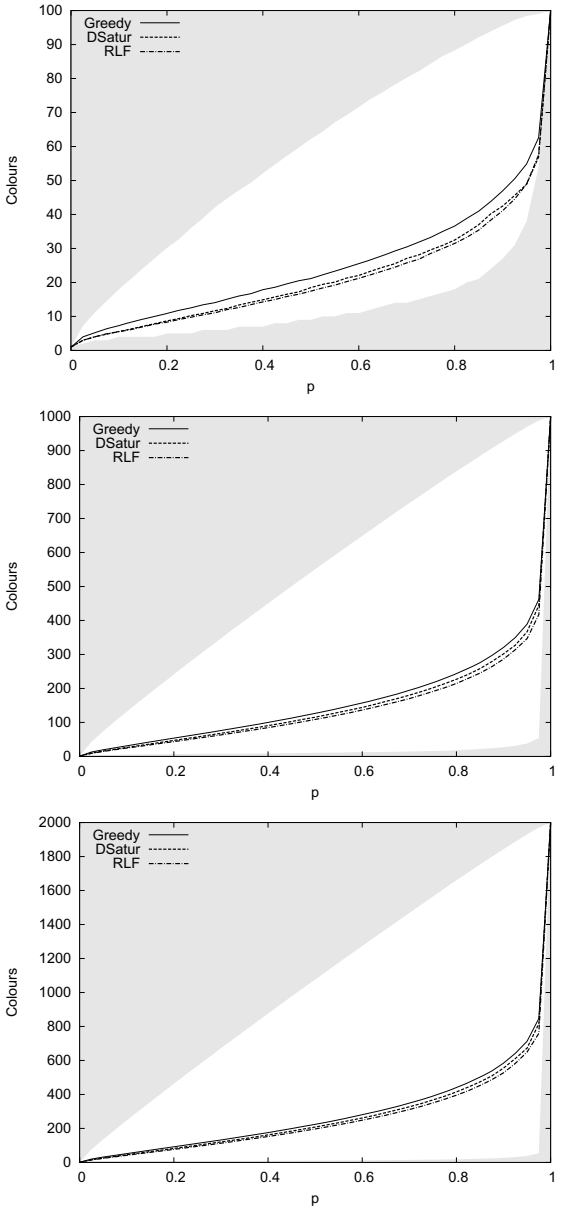

Fig. 2.14 Average number of colours in solutions produced by the GREEDY, DSATUR and RLF algorithms on random graphs $G_{n,p}$ with various values of p, using $n = 100$, 1,000 and 2,000. All points are the mean across 50 graphs

number of colours used in solutions produced by the GREEDY algorithm, and the lines indicate the percentage of this figure used by DSATUR and RLF. We see that the latter two algorithms achieve percentages of less than 100% across all of the tested values for p, indicating their superior performance across the range of random graphs, from sparse to dense. We also see that RLF consistently produces the lowest percentages, once again indicating its general superiority to DSATUR on these graphs.

We now turn to the implications of the computational complexity of the GREEDY, DSATUR, and RLF algorithms. Earlier in this chapter we noted that GREEDY and DSATUR both have worst-case complexities of $\mathcal{O}(n^2)$, while for the RLF this is $\mathcal{O}(n^3)$. What effects does this have when running the algorithms on different graph colouring problems? Figure 2.16 shows the number of constraint checks required by the algorithms for random graphs with $p = 0.5$ using values of n up to 2,000. For larger graphs the RLF algorithm clearly requires more computational effort to complete a run than both GREEDY and DSATUR. Indeed, as n increases, this gap seems to widen quite significantly. In contrast, the GREEDY algorithm requires by far the fewest constraint checks, with its line being barely distinguishable with the horizontal axis in the figure.

The next graph, Figure 2.17, shows the computational requirements of the three algorithms for different values of p. It can be seen that the number of constraint checks required by GREEDY and DSATUR remains fairly stable over the range, suggesting that it is the number of vertices n, and not the edge connectivity p, that is the driving force in determining the two algorithms' computational requirements. In contrast, RLF's requirements once again increase quite rapidly over this range.

Finally, Figure 2.18 demonstrates the strong correlation that exists between the number of constraint checks the algorithms require and the subsequent CPU time that is used (coefficient of determination $R^2 = 0.939$).[2] In fact, the majority of this figure is once again dominated by data generated from runs of the RLF algorithm with GREEDY and DSATUR's results being tightly clustered in the bottom left corner (the GREEDY algorithm never required more than 16 ms on any of the graph colouring problem instances considered in this section; similarly the DSATUR algorithm never required more than 47 ms). This figure demonstrates that the use of constraint checks as a measure of computational effort is suitable for estimating CPU time, but also has the obvious advantage of being independent of any issues to do with computer hardware, programming languages and operating systems.

2.6 Chapter Summary and Further Reading

In this chapter we have reviewed a number of bounds for the graph colouring problem and have also compared and contrasted three constructive algorithms. For random graphs of different sizes and densities (including sets of graphs and their iso-

[2] The CPU times relate to a 3.0 GHz Windows 7 PC with 3.87 GB RAM.

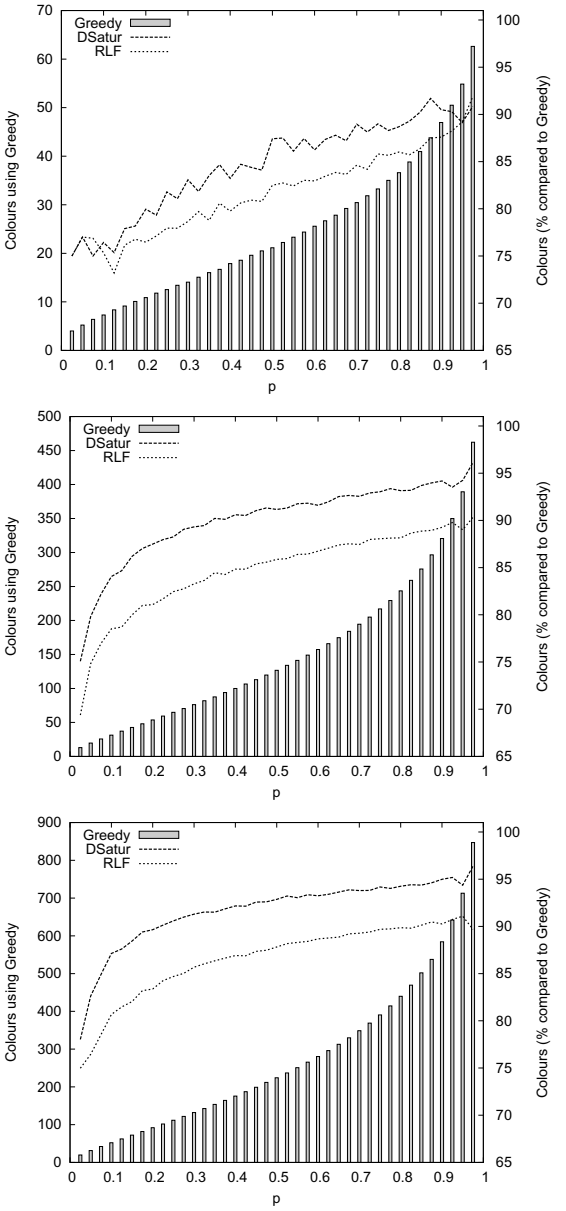

Fig. 2.15 Mean quality of solutions achieved on random graphs $G_{n,p}$ by the RLF and DSATUR algorithms compared to GREEDY. All points are the mean across 50 graphs using $n = 100$, 1,000, and 2,000 respectively

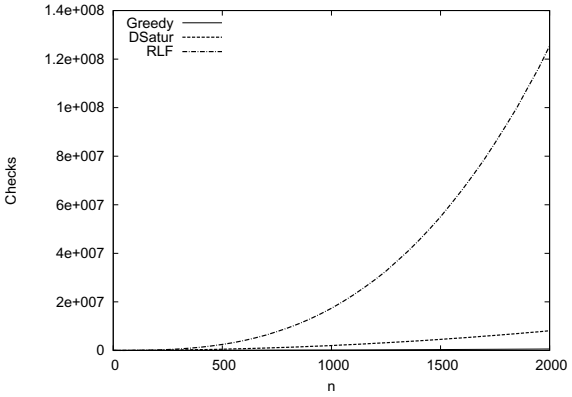

Fig. 2.16 Number of constraint checks required by the GREEDY, DSATUR and RLF algorithms on random graphs $G_{n,p}$ with $p = 0.5$. All points are the mean of 50 trials

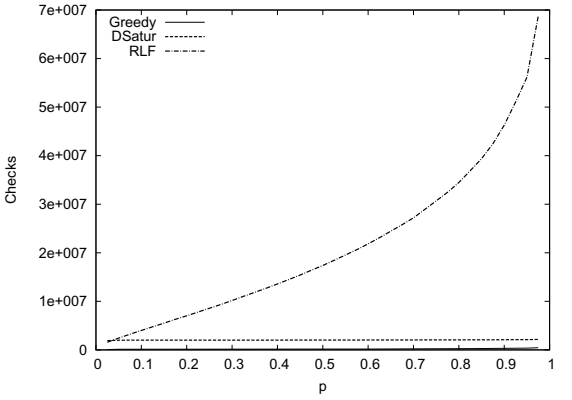

Fig. 2.17 Number of constraint checks required by the GREEDY, DSATUR and RLF algorithms on random graphs $G_{n,p}$ with $n = 1,000$. All points are the mean of 50 trials

morphisms), we have seen that the RLF algorithm generally produces solutions with the fewest colours, though this also comes at the expense of added computation time, particularly for graphs with larger numbers of vertices.

In the next two chapters we will analyse a number of techniques that seek to improve upon the solutions produced by these algorithms. We now end this chapter by providing points of reference for further work on bounds for the chromatic number.

Reed (1999) has shown that Brooks' Theorem (2.7), can be improved by one colour when a graph G has a sufficiently large value for $\Delta(G)$ and also has no cliques of size $\Delta(G)$. Specifically:

Theorem 2.13 (Reed (1999)) *There exists some value δ such that if $\Delta(G) \geq \delta$ and $\omega(G) \leq \Delta(G) - 1$ then $\chi(G) \leq \Delta(G) - 1$.*

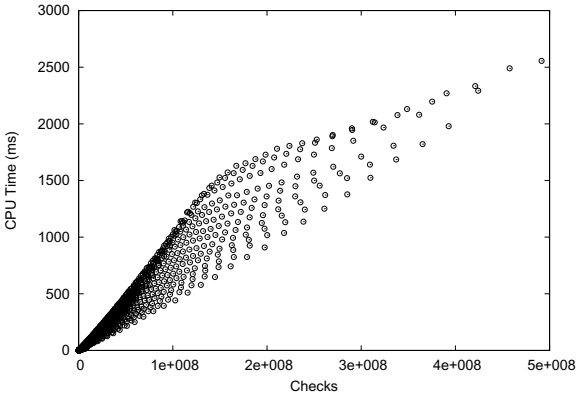

Fig. 2.18 Scatter diagram showing the relationship between number of constraint checks and CPU time. Each point is the mean of the 50 runs performed by each algorithm on each p and n value

In the work a sufficient value for δ is shown to be 10^{14}. Reed's Conjecture, also stated in this work, suggests that for any graph G,

$$\chi(G) \leq \left\lceil \frac{1+\Delta(G)+\omega(G)}{2} \right\rceil. \tag{2.4}$$

A good survey on these issues can be found in the work of Cranston and Rabern (2014).

In Section 2.2.1.1 of this chapter we also saw that interval graphs (and more generally chordal graphs) feature chromatic numbers $\chi(G)$ equal to the their clique numbers $\omega(G)$. Chordal graphs form part of a larger family of graphs known as *perfect* graphs which, in addition to satisfying this criterion, are also known to maintain this property when any of its vertices are removed.

Definition 2.16 *A graph* $G = (V,E)$ *is* perfect *if, for every subgraph* $G' \subseteq G$, $\chi(G') = \omega(G')$.

Defining the structures needed for a graph to be perfect has been the subject of much research in the field of graph theory and was eventually settled by Chudnovsky et al. (2006), who proved the earlier conjecture of Berge (1960), which states that a graph is perfect if and only if it contains no odd hole and no odd antihole. (A hole is an induced subgraph which is a cycle of length at least 4; an antihole is the complement). See MacKenzie (2002) for further details.

Looking at other topologies, bounds on the chromatic number of random graphs have also been determined by Bollobás (1988), who states that with very high probability, a random graph $G_{n,p}$ will have a chromatic number $\chi(G_{n,p})$:

$$\frac{n}{s} \leq \chi(G_{n,p}) \leq \frac{n}{s} + \left(1 + \frac{3\log\log n}{\log n}\right) \tag{2.5}$$

where

$$s = \lceil 2\log_d n - \log_d \log_d n + 2\log_d(e/2) + 1 \rceil \tag{2.6}$$

with $q = 1 - p$ and $d = 1/q$. The finding is based on calculating the expected number of disjoint cliques within a random graph, and provides much tighter bounds than Equation (2.3) and Brooks' Theorem (though recall that the latter applies to *all* graphs, not just random graphs). Further bounds on general graphs have also been given by Berge (1970), who finds

$$\frac{n^2}{n^2 - 2m} \leq \chi(G), \tag{2.7}$$

and Hoffman (1970), who has shown

$$1 - \frac{\lambda_1(G)}{\lambda_n(G)} \leq \chi(G), \tag{2.8}$$

where $\lambda_1(G)$ and $\lambda_n(G)$ are the biggest and smallest eigenvalues of the adjacency matrix of G. Both of these often give very loose lower bounds in practice, however.

Note that, strictly speaking, the three constructive algorithms reviewed in this section should be classed as heuristic algorithms as opposed to approximation algorithms. Unlike heuristics, approximation algorithms are usually associated with provable bounds on the quality of solutions they produce compared to the optimal. So for the graph colouring problem, using $A(G)$ to denote the number of colours used in a feasible solution produced by algorithm A with graph G, a good approximation algorithm should feature an approximation ratio $A(G)/\chi(G)$ as close to 1 as possible. Those seeking an algorithm with a low approximation ratio for the graph colouring problem, however, should take note of the following theorem:

Theorem 2.14 (Garey and Johnson (1976)) *If, for some constant $r < 2$ and constant d, there exists a polynomial-time graph colouring algorithm A which guarantees to produce $A(G) \leq r \times \chi(G) + d$, then there also exists an algorithm A' which guarantees $A'(G) = \chi(G)$.*

In other words, this states that we cannot hope to find an approximation algorithm A for the graph colouring problem that, for all graphs, produces $A(G) < 2 \times \chi(G)$ unless P = NP.

http://www.springer.com/978-3-319-25728-0

A Guide to Graph Colouring
Algorithms and Applications
Lewis, R.M.R.
2016, XIV, 253 p., Hardcover
ISBN: 978-3-319-25728-0