Frequency reassignment in cellular phone networks

Problem presented by

Simon Brusch

Motorola

Problem statement

In cellular communications networks, cells use beacon frequencies to ensure the smooth operation of the network, for example in handling call handovers from one cell to another. These frequencies are assigned according to a frequency plan, which is updated from time to time, in response to evolving network requirements. The migration from one frequency plan to a new one proceeds in stages, governed by the network's base station controllers. Existing methods result in periods of reduced network availability or performance during the reassignment process. The Study Group was asked to develop an algorithm for implementing a new frequency plan that maintains service quality during the transition.

Study Group contributors

John Billingham (University of Nottingham)
Tim Gould (Lancaster University)
Sam Halliday (Heriot-Watt University)
Robert Leese (Smith Institute)
Colin Please (University of Southampton)
Ida Pu (Goldsmiths College London)
Hannu Rajaniemi (University of Edinburgh)
Eddie Wilson (University of Bristol)
Daniel Winterstein (University of Edinburgh)

Report prepared by

John Billingham (University of Nottingham)
Robert Leese (Smith Institute)
Hannu Rajaniemi (University of Edinburgh)

1 Introduction

Finding the optimal frequency assignment of a network of cellular phone base stations is known to be a hard – in fact, NP-hard – problem. The purpose of this report is to address a related question. During the operation of a cellular network service it often becomes necessary to implement a new frequency assignment plan in response to changes in traffic or perhaps after new hardware has been installed. Usually the switch to the new plan involves a global reset of the network which results in a break in the provided service. Hence, it is desirable to investigate the possibility of *dynamic reassignment – i.e.*, a method which can be used to move over to the new plan while the network remains in operation. It turns out that this problem can be naturally formulated in terms of *graph recolouring*. We present a simple but effective algorithm for dynamic reassignment based on a straightforward approach of search and random colouring.

2 Statement of the problem

Background: In a cellular mobile network, **cells** correspond to small areas of coverage, which together provide a telecommunications service to a much larger area. A **frequency plan** for the network consists of an assignment of frequencies to cells subject to various constraints. When modelling the network, its cells are regarded as the vertices of a graph, and neighbouring cells are joined by an edge. For the purposes of this problem, we take the cellular network to be modelled by a simple graph $G = (V, E)$, where V is the set of vertices and E the edges between them.

The particular frequency f assigned to a vertex actually consists of a pair: a **channel** and a **colour code**[1], so that $f = (\mathrm{CH}, \mathrm{CC})$. Each vertex (cell) is associated with a particular **base station**. In general, several cells may be associated with a single base station, for example when they correspond to different sectors of coverage around around an antenna site. Finally, the base stations are configured by a collection $\mathcal{C}$ of **base station controllers** for the network. For reasons that will become obvious below, it makes sense to label a vertex v with the base station controller $c_v \in \mathcal{C}$ that controls its base station. Overall, we associate a tuple

$$A_v = (f_v, c_v) \tag{1}$$

to each vertex v. The frequency plan is then the set $\mathcal{A} = \{A_v | v \in V\}$.

The problem as presented here relates to the beacon frequencies that are used by cells to carry identifying information. In practice, each cell will also have one or more traffic-carrying frequencies. Interference between the traffic frequencies can be an issue, but it was not included in the Study Group problem. Including traffic frequencies would amount to a natural extension of the method we present here.

A frequency plan $\mathcal{A}$ must respect a selection of possible constraints:

[1]The colour codes are part of the network implementation and are distinct from the modelling of the problem in terms of graph colouring.

(1) The 'Neighbour-of-neighbour' constraint: if v_1 and v_2 are each neighbours of v, then v_1 and v_2 must use different frequencies from each other, *i.e.* $f_{v_1} \neq f_{v_2}$. Two frequencies are different if they differ either in the channel or the colour code or both.

(2) The 'Neighbour' constraint: if v_1 and v_2 are neighbours then they must use different frequencies from each other, *i.e.* $f_{v_1} \neq f_{v_2}$.

(3) 'Interference constraints': if v_1 and v_2 are geographically close, even if not neighbours in the sense of (1) and (2), then $|f_{v_1} - f_{v_2}|$ must not be too small.

(4) Some neighbouring cells, for example those corresponding to different sectors on the same base station, must receive assignments that are at least 2 channels apart.

(5) There can be other constraints depending on the type of hardware installed at a site.

Constraint (1) always applies. In a GSM network, (3) and (4) also apply, and in a iDEN network, (2) and (4) also apply. The constraints essentially arise from the fact that a mobile handset moving through the coverage area of the network must always be able to identify the neighbouring cells uniquely: this allows for the smooth handover of services from cell to cell. It is important to note that 'neighbouring' cells in this context are not necessarily geographically adjacent, but are ones between which a direct handover is possible.

The problem: Now suppose that we have a frequency plan $\mathcal{A}$ and wish to change to a new plan $\mathcal{A}'$. We assume that both plans satisfy the relevant constraints.

One way to make the change is to create and download all the new databases containing the new frequency plan into the system and then reset all the cells at the same time. However, this causes an outage of about 20 minutes to the service. To avoid an outage we would prefer to change the channel or colour code in the cells one-by-one in a carefully predetermined way, waiting to change each cell until a time when it is not involved in any calls. The changeover is then invisible to the users.

At each step we are allowed to change *either* CH *or* CC for k vertices, where $k = |\mathcal{C}|$ is the number of base station controllers – in other words, each base station controller can make one change independently, provided that we do not violate the constraints in the process. In an ideal situation, it will take two steps to change from $f = (\mathrm{CH}, \mathrm{CC})$ to $f' = (\mathrm{CH}', \mathrm{CC}')$: either via $(\mathrm{CH}, \mathrm{CC}')$ or $(\mathrm{CH}', \mathrm{CC})$. Moreover, *the network must operate correctly during the whole changeover process.* In other words, the predetermined sequence of steps must be such that the constraints are satisfied in *all* the intermediate states, where some cells are using the original frequency, others have changed either channel or colour code, and others have already changed both. As a result, the direct transition may not be possible, and some cells may have to change from f_i to f'_i in *more* than two steps. In general it is desirable to minimize the total number of steps in the changeover process.

The neighbour-of-neighbour constraint (1) is a hard constraint in the reassignment process, meaning that it cannot be violated, and maintaining this constraint is regarded

as the heart of the problem. If cells have d neighbours, there are effectively $d(d-1)/2$ constraints per cell on the frequency assignment. In contrast, the interference constraints (3) are so not vital, since they are 'soft' constraints. If they are violated, then they will not be violated for very long, although the duration of bad interference conditions should also ideally be minimized.

A typical network might have 2,000–3,000 cells, each cell having 15–30 neighbours, with the frequencies being chosen from a set of 320, made up of say 10 channels and 32 colour codes. Figure 1 shows a typical graph with 500 vertices.

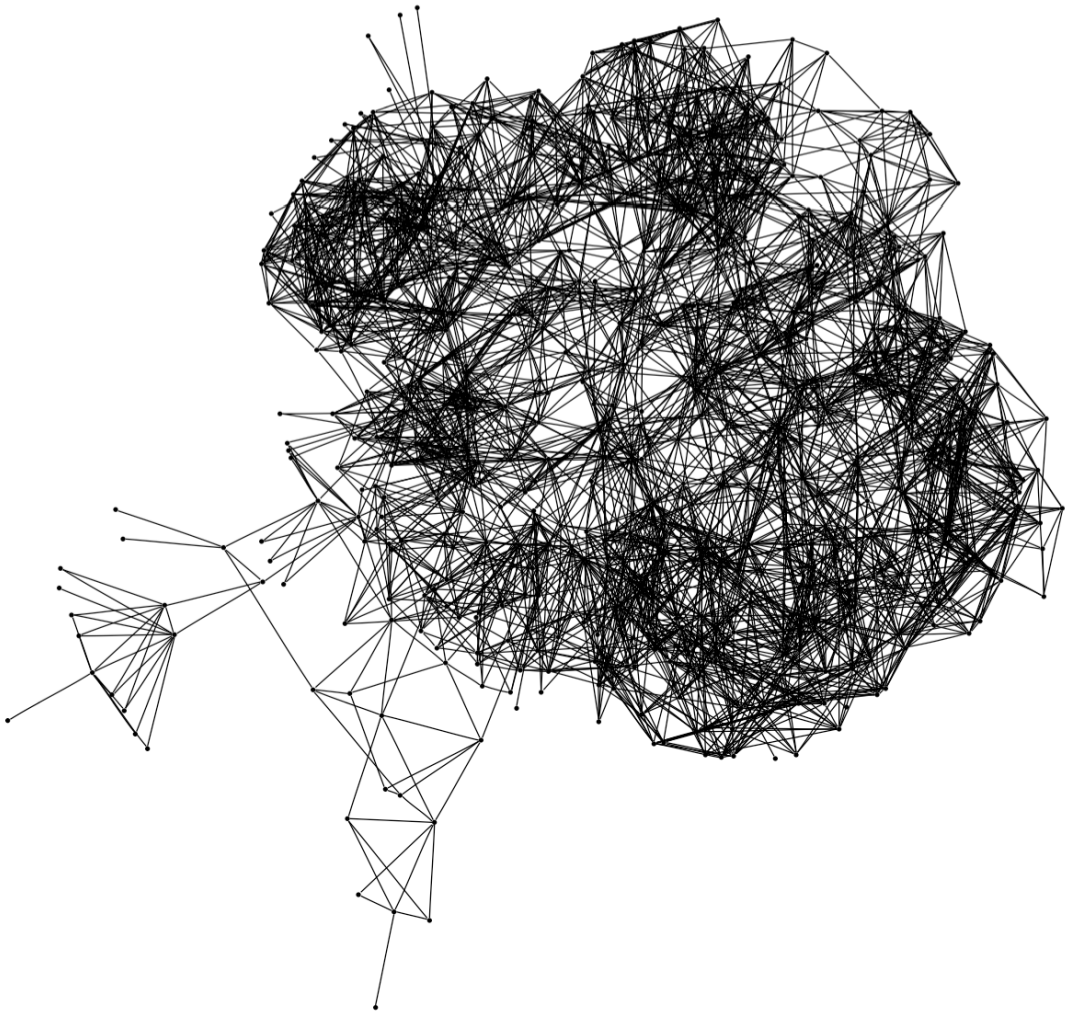

Figure 1: A cellular network represented as a simple graph.

3 Connections with graph theory

3.1 Frequency assignment problems

Frequency assignment problems (FAPs) draw heavily on ideas from graph colouring. We shall be concerned with vertex-colouring, where a colour is assigned to each vertex of

a graph G. The **chromatic number** of the graph, $\chi(G)$, is the smallest number of colours that are needed to colour every vertex so that every pair of adjacent vertices receive different colours. Such a colouring is called **proper**. More generally, we shall be interested in proper k-colourings, using k colours for some $k \geq \chi(G)$.

Frequency assignment problems (FAPs) are closely related, with the colours now being replaced by frequencies, and edges indicating where interference might occur if the same frequency were used at both ends. In FAPs, numerical labels are used to represent the frequencies, so that more general conditions can be handled than simply the requirement that pairs of frequencies at adjacent vertices be different. We can now specify the extent of the separation between two frequencies and require that pairs of geographically close vertices have a higher frequency separation than those that are well separated geographically. Over the last few years there has been extensive analytical and computational progress in solving such problems. The counterpart of the chromatic number is the smallest possible difference between the highest and lowest frequencies, known as the **span** of the assignment. The graphs that arise in FAPs reflect the way in which radio signal strength falls off with distance: the highest potential for interference is at short range and so the graph has a 'localized' structure, with most edges joining vertices that are geographically close. Successful computational solutions to FAPs seem implicitly able to exploit this structure to great effect; see, for example, [5] and references therein. An idealized version of the localized structure of FAPs is provided by **unit disk graphs** [4], in which each vertex is associated with a disk of fixed size in the plane, and edges correspond to a pair of overlapping disks.

3.2 Vertex recolouring

Ideas that are related to frequency reassignment have recently been discussed in the context of vertex colourings by Cereceda, van den Heuvel and Johnson in [3]. This paper looks at the graph $\mathcal{C}_k(G)$ formed by the proper k-colourings of a graph G, under moves which allow the colour at a single vertex to be updated (still with the requirement that the new colouring be a proper one). To be precise, each vertex of $\mathcal{C}_k(G)$ corresponds to a proper k-colouring of G, and two vertices in $\mathcal{C}_k(G)$ are joined by an edge if their corresponding k-colourings can be obtained from each other simply by changing the colour assigned to a single vertex of G. Of particular interest is whether $\mathcal{C}_k(G)$ is *connected*, *i.e.*, whether any two proper k-colourings of G can be connected by a sequence of updates to the colour at individual vertices, with the colouring remaining proper at every intermediate step. If so, then G is said to be k-**mixing**. An initial observation is that any given graph G is k-mixing whenever $k \geq \Delta(G)+2$, where $\Delta(G)$ is the maximum degree of G.[2]

The problem presented by Motorola involves assigning a pair $f = (\mathrm{CH}, \mathrm{CC})$ to each vertex, rather than a single colour, but nevertheless the example datasets brought to the Study Group suggest that we are certainly working in the regime where the graph of assignments (defined analogously to $\mathcal{C}_k(G)$) is connected. In other words, there are many more combinations of channel and colour code at our disposal than the minimum needed to provide a valid assignment. As a result we might expect that there is considerable

[2]The **degree** of a vertex in G is its number of immediate neighbours.

freedom in constructing a path between two given frequency plans, and, as we shall see, this appears to be the case. The evidence is that, in practice, sequences of updates are possible that transform any initial frequency plan into a new plan, without violating any of the assignment constraints along the way.

Returning to the classical vertex-colouring of [3], it is interesting to investigate the boundary, in terms of k, between values where $\mathcal{C}_k(G)$ is connected and disconnected, subject of course to taking $k \geq \chi(G)$. The following examples use the standard terminology of graph theory, as explained in [1] for example.[3] In all cases, we take G itself to be connected. The remainder of this section is not crucial to what follows subsequently.

- The complete graph on n vertices, K_n, has $\chi(K_n) = n$. The graph $\mathcal{C}_k(K_n)$ is connected for $k > n$ and empty (*i.e.* with no edges at all) for $k = n$.
- The cycle C_n has n vertices, each of degree 2, and is therefore k-mixing for any $k \geq 4$. If n is odd then $\chi(C_n) = 3$ and $\mathcal{C}_k(C_n)$ is disconnected for $k = 3$ (see [3] for the proof of this result in the wider context of any graph with chromatic number 3).
- If, instead, n is even in the previous example then C_n has chromatic number 2 and $\mathcal{C}_2(C_n)$ is disconnected. For $k = 3$, we have that $\mathcal{C}_3(C_4)$ is connected, but that $\mathcal{C}_3(C_n)$ is disconnected for higher even values of n. See Figure 3.2 for a 3-colouring of C_6 from which no move can be made to another proper 3-colouring. In general, colourings that admit no updates are called 'frozen' or 'locked'.
- There are also examples where the boundary between connectedness and disconnectedness occurs for values of k much higher than $\chi(G)$. Suppose that the graph L_m is formed by taking a balanced complete bipartite graph $K_{m,m}$ and removing the edges of a perfect matching. Then L_m has $2m$ vertices and chromatic number 2. Since each vertex has degree $m-1$, L_m is certainly k-mixing for $k > m$, and in fact the same is true for $3 \leq k < m$. However, it is not k-mixing for $k = m$ (the proofs of these last two statements are given in [3]).

The last of the four examples above shows that k-mixing is not a monotone property in k, *i.e.*, it may be that G is k_1-mixing but not k_2-mixing for some $k_2 > k_1$. It also shows that there are graphs G that are not k-mixing even when k is arbitrarily larger than $\chi(G)$. However, L_m does not have the structure of graphs that typically arise in frequency assignment. For example, if $m \geq 7$ then L_m contains an induced copy of the bipartite graph $K_{1,6}$ (a star with a central vertex of degree 6), which is not a unit disk graph.[4]

In summary, while we have no guarantee that a path always exists between pairs of Motorola's frequency plans, the evidence strongly points in this direction. It seems that the number of available combinations of channel and color code is in practice comfortably above the levels that the study of vertex recolouring suggests might be needed. It should also be remembered that we have available the additional flexibility of being able

[3] A free and complete electronic version of this book is available at
`http://www.math.uni-hamburg.de/home/diestel/books/graph.theory/GraphTheoryIII.pdf`.

[4] Similarly, for $m \geq 5$, L_m has an induced $K_{3,2}$, which is also not a unit disk graph.

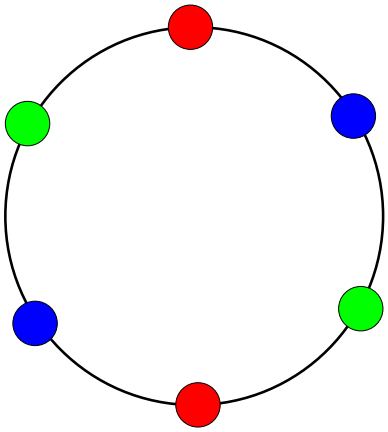

Figure 2: A 'locked' or 'frozen' 3-colouring of a 6-cycle. No recolouring is possible because each vertex has neighbours that are already coloured with each of the possible alternatives.

to update several channels or colour codes simultaneously, one for each base station controller.

4 Voronoi tilings and cellular networks

The Study Group had available a small sample of graphs based on real cellular phone networks. To conduct more extensive tests on our algorithm, we found it desirable to develop a method of randomly generating graphs with similar characteristics to the examples provided by Motorola. These graphs should have a similar average degree as real cellular networks, in addition to other topological properties.

It is immediately clear that unstructured random graphs are not suitable. For an unstructured random graph with n nodes and average degree k, edges are filled in independently of each other with probability $\sim k/2n$. The local connectivity of these graphs differs from that of graphs based on real cellular network data – this difference could be quantified, for example by computing clustering coefficients, which measure the tendency of vertices that share a neighbour to themselves be neighbours of each other. The clustering of vertices in real-life networks is a further reflection of the localized nature of radio signal propagation. We expect that the clustering of vertices influences the rate at which our recolouring algorithm converges.

Therefore, we need a way of generating random graphs using a geometric construction that mimics the way in which network cells provide coverage over a wide area. The method we used is based on the well-known *Voronoi tiling* [2] of a plane. The cells of a Voronoi tiling are irregular polygons which 'house off' a collection of points from each other. Each line segment of a Voronoi cell is a segment of the perpendicular bisector between the two points which it separates. Constructing a Voronoi tiling is a well-known problem in computational geometry and efficient algorithms can be easily found in the literature or in standard software libraries (in MATLAB, for example).

Our algorithm for generating artificial cellular network graphs consists of the following steps:

(1) Generate points at uniformly distributed random positions in the plane.

(2) If the initial distribution of points is too strongly clustered in places, evolve them forward for a short time under a repulsive potential field.

(3) If an inhomogeneous coverage is desired, for example to model the distinction between urban and rural areas, apply a relevant mapping of the plane into itself.

(4) Replace each of the points with a triad of points very close together, to model the different antennae on a single transmitter site — we did this by always placing one antenna due north and the others at angular intervals of $2\pi/3$.

(5) Construct a Voronoi tiling of the resulting set of points.

(6) Construct a graph, whose vertices correspond to the Voronoi cells and where edges corresponds to geometric adjacency of the respective cells.

We did not attempt to implement steps (2) or (3), but the graphs generated by the remaining steps seemed to approximate the behavior of real-life graphs well enough.

Note that it would also be possible to modify this algorithm to take different signal strengths into account, by associating weights to the initial set of points. The Voronoi cell boundaries would then be modified according to the weights of the points involved, instead of simply being bisectors.

Figure 3 shows an example of a Voronoi tiling. The average degree of the graph generated from it agrees roughly with the real-life data provided by Motorola.

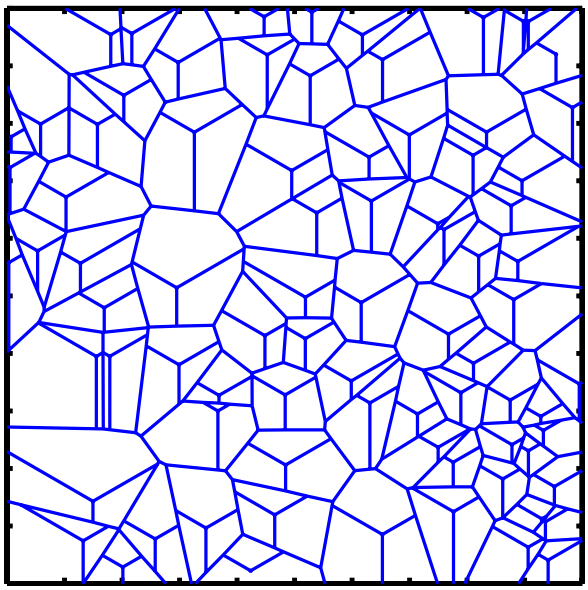

Figure 3: A Voronoi tiling of the plane.

5 The frequency reassignment algorithm

5.1 General approach

In the frequency reassignment problem, the task is to change an initial frequency plan into a (perhaps very different) target plan, in a way that does not violate any constraints along the way. Broadly speaking, our approach is to select individual vertices and to update their frequencies to match the target plan where possible. If this is not possible, owing to constraint violations, then a new frequency is assigned at random and the process continues. The detailed implementation takes account of the fact that several vertices can be reassigned simultaneously, provided they belong to different base station controllers, and that the channels and colour codes must be updated separately.

A similar process is seen in the so-called **Glauber dynamics** on the vertex-colourings of a graph.[5] The Glauber dynamics begin with an initial proper k-colouring of the vertices. At each step, a vertex is chosen at random, and is recoloured with a colour selected uniformly at random from those that do not currently appear at any of its neighbours. In this way, the colouring remains proper at all steps. The Glauber dynamics corresponds to a Markov chain on the set of proper k-colourings. In terms of Subsection 3.2, $\mathcal{C}_k(G)$ being connected corresponds to this Markov chain being irreducible. There has been great interest over recent years in investigating the circumstances for which the chain is **rapidly mixing**, meaning that it approaches its equilibrium distribution in a time that is polynomial in the number of vertices. Rapidly mixing chains give a way of efficiently sampling uniformly at random from the set of all proper k-colourings. A good introduction to these topics is to be found in [6], with a more recent treatment in [7].

The ability of Glauber dynamics to explore the space of proper colourings suggests that a similarly randomized approach to the frequency reassignment problem might be an effective one. Of course, a key difference is that here we are attempting to connect two specific frequency plans, rather than to cover the space of all plans. Our approach combines direct updates to the target plan where possible, alongside a randomized exploration of assignments in the remaining parts of the network. Where the target assignment cannot be reached directly, the hope is that the random moves quickly take us to configurations from which the target assignment may be easily reached.

5.2 Implementation

The time available at the Study Group was sufficient to allow first implementations of the algorithm in both MATLAB and Python, and the source for both codes is given in appendices. The remainder of this section describes the operation of the MATLAB code, which carries out the following steps.

(1) Determine which base station controllers control cells not yet in their final states and choose at least `Ms` of them (see Subsection 5.3 for the definition of variables

[5] In the language of statistical physics, this would more precisely be the 'Glauber dynamics of an antiferromagnetic Potts model at zero temperature'.

used in the code). If all of the remaining cells not in their final states can be changed simultaneously, we are done. Otherwise:

(2) Loop over the chosen base station controllers, executing steps (3)–(6).

(3) Determine which cells are not at their target channels but can be reassigned directly to them, keeping a permissible overall state. For each of these, calculate the change in the soft constraint penalty, and choose the one that affects this most favourably.

(4) If no reassignments to the target channel can be made, attempt the same with the colour code, but, since colour code does not affect the soft constraint penalty, choose a cell at random.

(5) If no reassignments can be made directly to target channels or colour codes, make a permissible change of a random cell not in its final state to another, random permissible state. Note that if a cell already has the target channel or colour code, but not both, this step may take it away from the final state. Doing so allows a little leeway for motion away from the final state, which can be helpful in constructing the last few steps of a permissible path.

(6) If no changes can be made, move on to the next base station controller.

(7) Terminate the algorithm if the path becomes too long or if too many iterations pass without any improvement in the number of cells still to reach their target assignments.

5.3 Input data

We assume that the input data is available in a suitably convenient form, as follows. We assume that there are N cells, corresponding to the graph vertices and numbered from 1 to N. There are n_{f} channels numbered from 1 to n_{f} and n_{col} colour codes numbered from 1 to n_{col}. To keep notation for the channels and colour codes distinct, we will use f to denote channels, and refer to the previous frequencies as 'states'. A state is then a pair (f_i, c_j), with $1 \leq f_i \leq n_{\mathrm{f}}$ and $1 \leq c_j \leq n_{\mathrm{col}}$. In the code, the state (f_i, c_j) is labelled with $k = f_i + (c_j - 1) * n_{\mathrm{f}}$, so that there are $S = n_{\mathrm{f}} n_{\mathrm{col}}$ states in all.

The other inputs required are:

- `A` is the adjacency matrix of the network. Adjacent cells are those that can hand over directly to each other and are indicated by an entry of 1 in the corresponding element of the adjacency matrix. No geographical relation is necessarily implied by the adjacency matrix. We assume that `A` is symmetric. In practice, `A` is almost symmetric, and adding new edges to make it symmetric is a slight and unimportant modification.

- `bsc` contains M vectors, where M is the number of base station controllers. These vectors specify the cells controlled by each base station controller.

- `site` contains n_{site} vectors (where n_{site} is the number of antenna sites), containing the numbers of the cells at each antenna site. Cells within a site, typically three cells or fewer, must have different frequencies.[6]

- `Pa` and `Pc` are sparse matrices whose (i, j)th elements are the penalties for having cells i and j at equal (co-channel) or adjacent channels respectively. We assume that `Pa` and `Pc` have been made symmetric, as this does not affect the result.

- `Ms` is the number of base station controllers that can be accessed simultaneously. In all the data sets provided to the Study Group, `Ms` $= M$.

5.4 Initial and final states

In practice, the initial frequency plan would be known, the desired target plan would already have been calculated, and these could be supplied as inputs. The Study Group had one pair of real initial and target plans available, but the number of available states was far in excess of the number of cells, meaning that it was trivial to change from one permissible assignment to another.

To develop the algorithm, we have therefore used synthetic initial and target plans instead. We find an initial plan by taking a random state, and then checking whether each cell in turn satisfies the neighbour-of-neighbour and site constraints; if not, we change its state to a random permissible assignment. Although this procedure is not guaranteed to converge, in practice the minimum total number of states (channel-code pairs) below which we could not set up a permissible initial assignment was smaller than the minimum total number of states below which we could not find a path from the initial to the target plan, and so it did not hamper development of the algorithm.

We set up the target plan in the same way, except that, at each cell in turn, we determined which states of the cell are permissible, calculated the soft constraint penalty associated with each, and chose the state with the smallest penalty. In this way, the initial plan is random, but the target plan is biassed towards a low (but undoubtedly not globally minimum) soft constraint penalty. Typically, the soft constraint penalty of the initial plan was 5 to 10 times higher than that of the target plan.

5.5 Results

We had available two data sets corresponding to real network topologies, on which to test the algorithm. Data set 1 contained 585 cells, with an average of 30 neighbours-of-neighbours per cell and 6 base station controllers. Data set 2 contained 918 cells, with an average of 21 neighbours-of-neighbours per cell and 9 base station controllers. The algorithm was tested with different numbers of available channels and colour codes, and we present here results for four different instances:

[6] In the MATLAB code, `bsc` and `site` are implemented as cell arrays.

	Network	Cells	Average NoN	BSCs	Channels	Codes
Instance 1	Data set 1	585	30	6	18	16
Instance 2	Data set 1	585	30	6	10	9
Instance 3	Data set 2	918	21	9	18	16
Instance 4	Data set 2	918	21	9	12	12

Figure 4 shows the progress towards the target plan for Instance 1, when there are 18 channels and 16 colour codes available. The algorithm easily generates a permissible path in one pass. That we chose to change channels first and then colour code can clearly be seen. Figure 5 shows the change in the soft constraint penalty for four different runs. In the first three, rather than choosing changes in channel in order to minimize the soft constraint penalty, we chose at random. The fourth run was with the full algorithm. We can see that this makes a large improvement in the total soft constraint penalty over the path.

Figures 6 and 7 show the results for Instance 2, when there are 10 channels and 9 colour codes. This was the smallest set of states for which we were able to find a permissible path, and the algorithm took 4 full attempts with different seeds in the random number generator to find a permissible path. We can see that the progress towards the target plan is more fitful in this case, as the algorithm attempts to make the final few changes. In addition, the smaller number of available channels means that the algorithm cannot change as many channels in the first part of the path, and so the soft constraint penalty decreases more slowly.

Note that there are (slightly) fewer steps in the path than when there were more states available simply because, with fewer states available, the random initial plan has more channels and colours in common with the target plan, and hence there are fewer changes to be made.

We now move on to the larger network, captured in data set 2. Figure 8 shows the progress towards the target plan in Instance 3, when there are 18 channels and 16 colour codes available. The algorithm again easily generates a permissible path in one pass. Figure 9 shows the change in the soft constraint penalty.

Figures 10 and 11 show the results for Instance 4, when there are 12 channels and 12 colour codes. This was the smallest set of states for which we were able to find a permissible path. It is not completely clear why this data set should be more difficult to deal with than data set 1, but it may be because the cells are not very evenly distributed between the base station controllers.

One final test investigates the ability of the algorithm to deal with smaller numbers of available states than in the previous problem instances. It is based on a regular triangular lattice, corresponding to a regular geographical coverage using hexagonal cells [8]. It was shown in [8] that 7 is the minimum number of channels that are needed to ensure that all cells are assigned a different channel from all their nearest neighbours and neighbours-of-neighbours. We took such an optimal assignment and then independently applied Glauber dynamics twice, with 10 available channels, in order to produce initial and target plans. The same colour code was given to each cell. Hence $n_{\mathrm{f}} = 10$ and $n_{\mathrm{col}} = 1$, meaning that there are just 10 states available. We used a lattice of 400 cells, and our algorithm was able to move smoothly from the initial plan to the target plan, as shown in Figure 12.

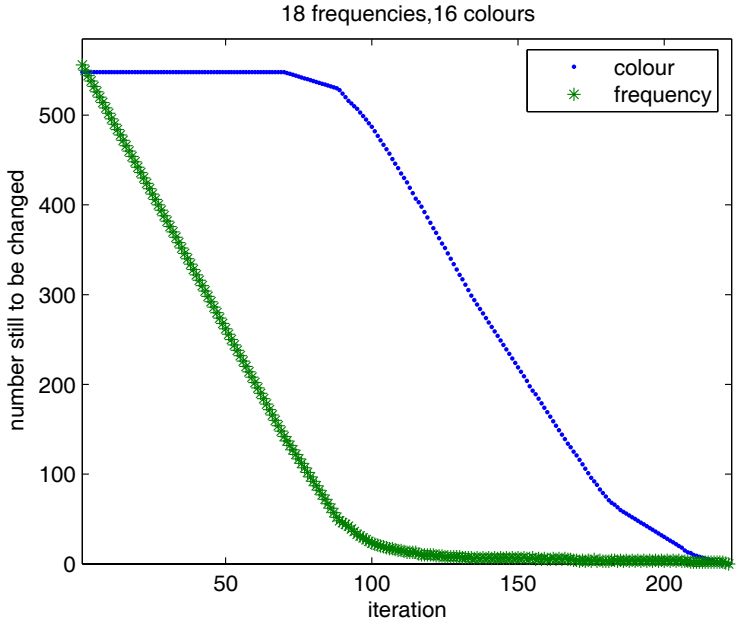

Figure 4: Progress towards the target plan for Instance 1, *i.e.* data set 1, with 18 channels (frequencies) and 16 colours.

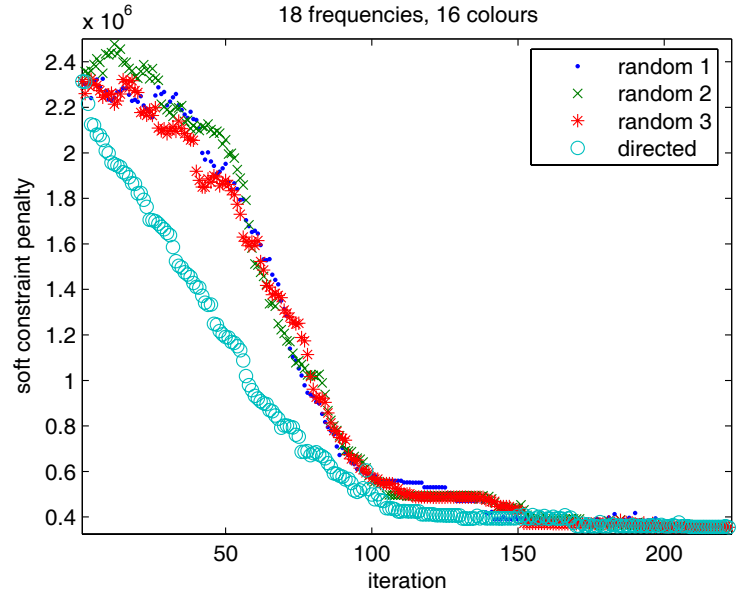

Figure 5: The change in the soft constraint penalty for Instance 1, *i.e.* data set 1, with 18 channels and 16 colours. The runs labelled as 'random' change channel in random cells, without reference to the soft constraint, whilst the run labelled 'directed' chooses the change of channel that minimizes the soft constraint penalty.

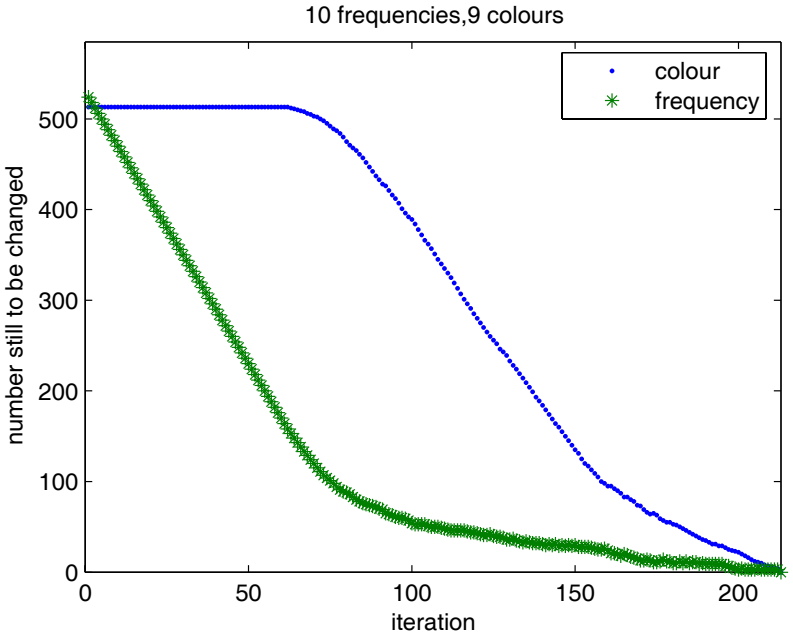

Figure 6: Progress towards the target plan for Instance 2, *i.e.* data set 1, with 10 channels (frequencies) and 9 colours.

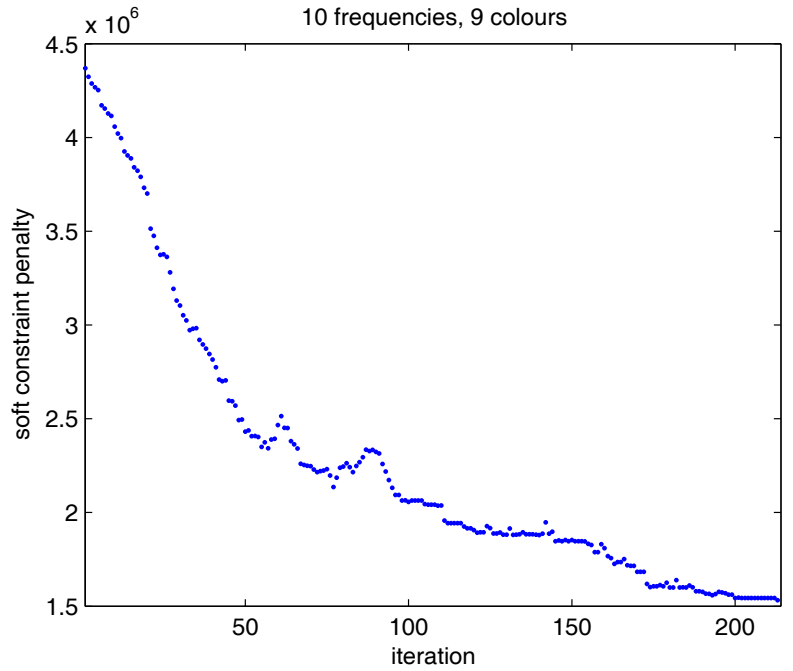

Figure 7: The change in the soft constraint penalty for Instance 2, *i.e.* data set 1, with 10 channels and 9 colours.

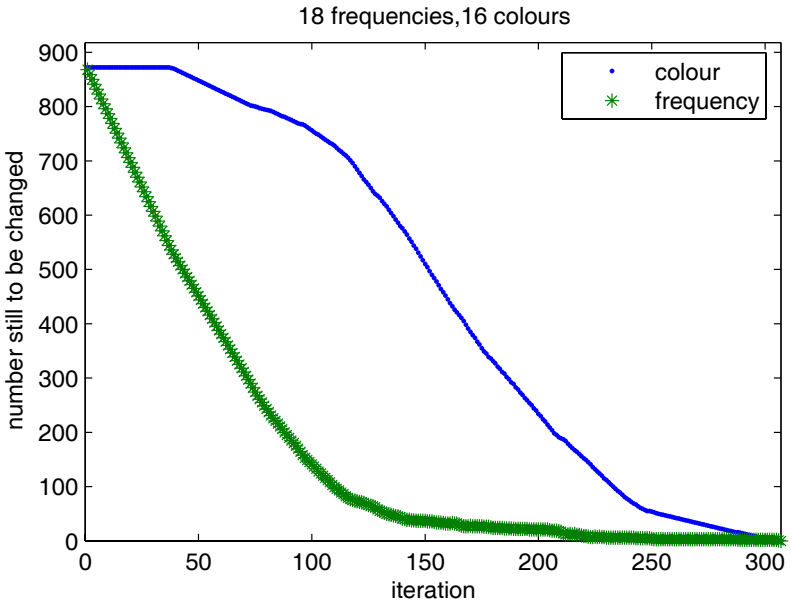

Figure 8: Progress towards the target plan for Instance 3, *i.e.* data set 2, with 18 channels (frequencies) and 16 colours.

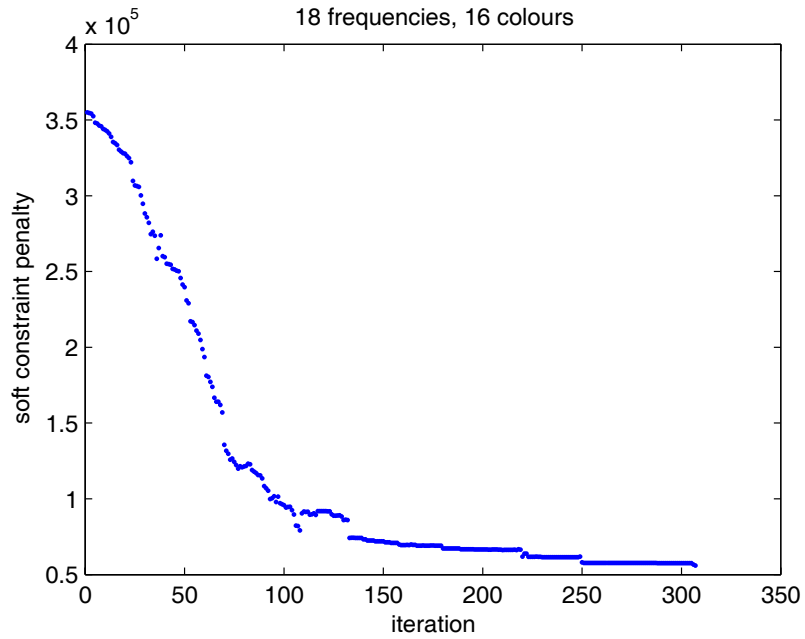

Figure 9: The change in the soft constraint penalty for Instance 3, *i.e.* data set 2, with 18 channels and 16 colours.

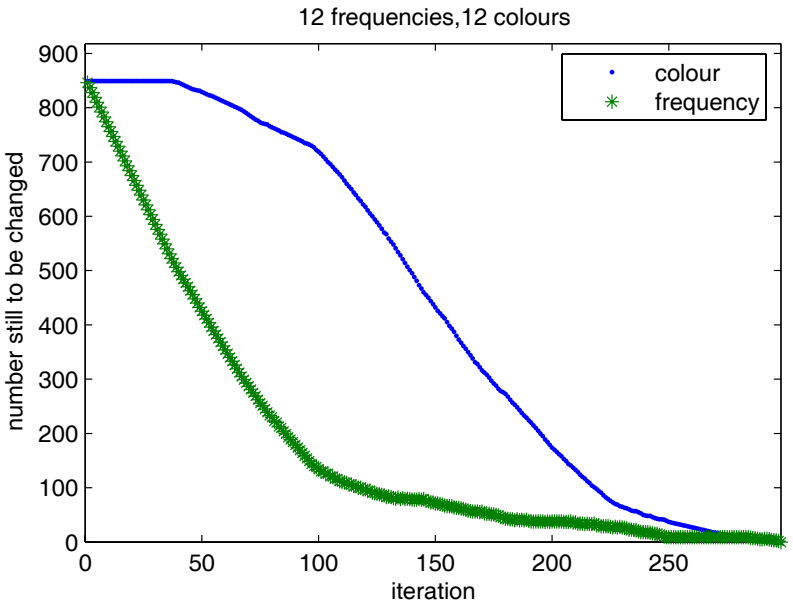

Figure 10: Progress towards the target plan for Instance 4, *i.e.* data set 2, with 12 channels (frequencies) and 12 colours.

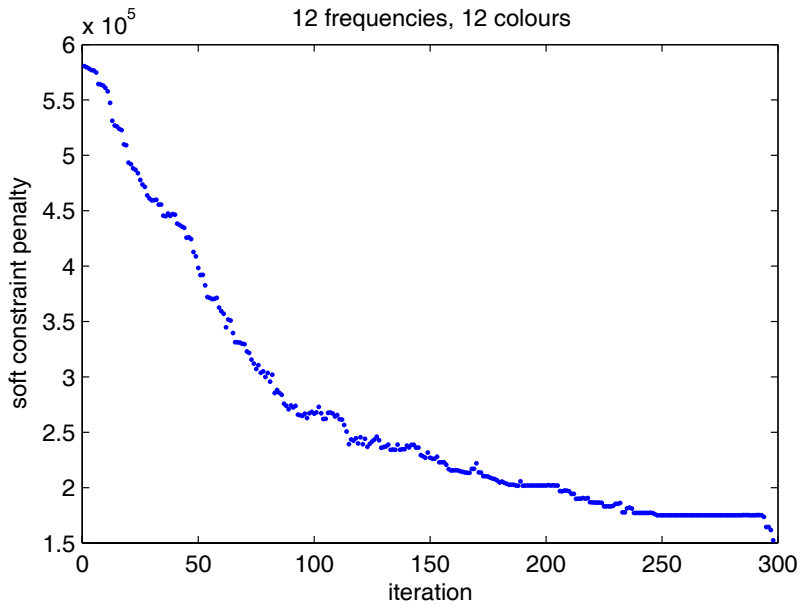

Figure 11: The change in the soft constraint penalty for Instance 4, *i.e.* data set 2, with 12 channels and 12 colours.

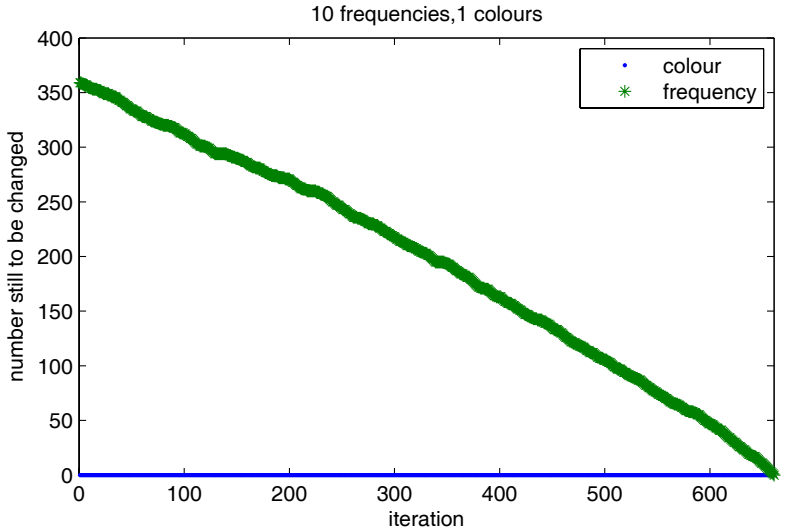

Figure 12: The reassignment algorithm applied to a regular lattice of 400 cells with 10 available states.

6 Conclusions and further work

Although our proposed algorithm is simple, even in the initial implementation it appears to be remarkably effective. Each iteration of the algorithm corresponds to one updating operation of base stations controllers, which in practice takes a few tens of seconds. For the problem instances that we have looked at in detail, the frequency reassignment could therefore be completed within two hours.

We implemented two (slightly different) versions of our algorithm using MATLAB and Python: for detailed code listings, we refer the reader to the appendices.

Although we could pronounce the problem solved, several questions and directions for further work remain:

- The current algorithm makes Glauber-type random moves whenever a direct move towards the target plan is not possible. This requires knowledge only of the current assignments and the target plan. An alternative would be to construct a connected subset of possible plans, containing both the initial and target plans, and the finding a shortest path between them. The current code on both platforms seems to run sufficiently fast to allow this approach, and it may be effective in cases where the randomized approach struggles to reach the target plan.

- It would be interesting to find a graph-theoretic criterion that determines when an allowed path between two frequency plans exists. The more cells that can be reassigned simultaneously, the more scope one has for successfully moving between different plans without violating constraints along the way. This suggests a definition of a 'base station number' of a graph, *i.e.*, the minimum number of simultaneous changes one needs to make to be able to move from any given frequency plan to any other.

- Our reassignment algorithm exhibits behaviour resembling a 'phase transition', where steady progress towards the target plan is replaced with much slower progress as the final few cells are reassigned. It corresponds to the 'kinks' in the figures 4–11. It would be good to understand the cause of this behaviour, both for possible acceleration of the algorithm and for the related theory.

- There are various optimisation techniques that might lead to improvements in our algorithm. For example, one could modify the random regime of the algorithm so that the local connectivity structure of the graph is exploited properly – in other words, the algorithm could proceed by making all possible changes in the neighbourhoods of a set of vertices, and then matching the neighbourhoods.

In a more speculative vein, given the success of our algorithm it seems likely that one could make the whole frequency plan assignment process dynamic: the cellular phone network could simply adapt to added nodes and increased traffic by recolouring itself on demand. It is clear that a great deal of interesting further work could be done on this front.

References

[1] *Graph Theory*, (3rd edition), R. Diestel, Springer-Verlag (2005).

[2] *Computational Geometry: An Introduction*, F. P. Preparata and M. I. Shamos, Springer-Verlag (1985).

[3] *Connectedness of the graph of vertex colourings*, L. Cereceda, J. van den Heuvel and M. Johnson, CDAM Research Report LSE-CDAM-2005-11, to appear in Discrete Mathematics (2005).

[4] *On coloring unit disk graphs*, A. Graf, M. Stumpf and G. Weissenfels, Algorithmica, **20** 277–293 (1998).

[5] *Frequency assignment in mobile radio systems using branch-and-cut techniques*, M. Fischetti, C. Lepschy, G. Minerva, G. Romanin-Jacur and E. Toto, European Journal of Operational Research, **123** 214–255 (2000).

[6] *A very simple algorithm for estimating the number of k-colourings of a low-degree graph*, M. Jerrum, Random Structures and Algorithms, **7** 157–165 (1995).

[7] *The Glauber dynamics on colorings of a graph with high girth and maximum degree*, M. Molloy, SIAM Journal on Computing, **33** 721–737 (2004).

[8] *Graph labelling and radio channel assignment*, J. van den Heuvel, R. A. Leese and M. A. Shepherd, Journal of Graph Theory, **29** 263–283 (1998).

A MATLAB code

```
function [s, nfout, ncout, pen] = ...
softpen(A, bsc, site, Pa, Pc, Ms, nfin, ncolin, rn)

global N M Msim S B NofN nf ncol site_list
global SITE bsc_list geolist adpen copen freq col

% INPUTS
% A is the adjacency matrix of the network. This indicates
% which cells can hand over to which other cells.

% Pa and Pc give the adjacent and co- frequency
% penalties (symmetrized for convenience)

bsc_list = bsc;
% bsc_list(i) is the base station controller for the ith cell

site_list = site;
% site_list(i) is the site number of the ith cell

Msim = Ms;
% Msim is the number of base station controllers that
% can be changed simultaneously

nf = nfin;
% nf is the number of different frequencies available

ncol = ncolin;
% ncol is the number of different colours available

S = nf*ncol;
% S is the total number of different states available

% rn is the seed for the random number generator

N = size(A); N = N(1);

NofN = setupNofN(A);
% NofN contains lists of neighbour of neighbour cells

[SITE, nsite] = setuparrays(site_list);
% SITE contains lists of cells at each site

[B, M] = setuparrays(bsc_list);
% B contains lists of cells controlled by each base station controller

setuppenalty(Pa,Pc)
% set up global variables associated with the soft constraint

% geo_list{k} is a list of cells that interfere with the kth cell

% adpen{k} and copen{k} give the penalties associated with the list
% geo_list{k} for adjacent and co- frequencies respectively.

[freq, col] = state2fcol(1:S);
% these vectors give the frequency and colour corresponding to the N states

disp('Setting up initial and final states')

sinit = setupstate(S,1,0); sfinal = setupstate(S,1e6,1);
% Set up the initial and final states at random,
% biassing the final state to a low soft
% constraint penalty. In practice, these would be inputs.

s = [];

if isempty(sinit)||isempty(sfinal)
    disp('Couldnt find initial states')
    return
end
```

```
rnmax = rn+100;
% Have 100 attempts at finding a permissible path

while (isempty(s))&&(rn<=rnmax)

    rand('state',rn)

    [s, nfout, ncout] = findpath(sinit,sfinal);
    % Find a permissible path from state sinit to state sfinal.

    % nfout and ncout are the number of frequencies and
    % colours still to be changed after each iteration.

    if ~isempty(s)

        % Calculate how the soft constraint penalty changes at each
        % step of the iteration, and plot it.

        disp('Calculating soft constraint penalties')
        pen = [];
        for k = 1:length(s)
            pen = [pen totalpenalty(s{k})];
        end
        totpen = sum(pen);
        figure(2), plot(pen,'.')
        xlabel('iteration'), ylabel('soft constraint penalty')
        title(strcat('total soft constraint penalty = ',num2str(totpen)))
        drawnow

    end

    rn = rn+1;
    % Reseed the random number generator and try again if necessary.

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function s = setupstate(S,rn,flag)
global N NofN SITE site_list freq

% Set up (rather inefficiently!) a random permissible state

s = []; rnmax = rn+10;
% Try 10 times before giving up.

while ~(ispermissible(s))
    rand('state',rn)
    s = mod(1:N,S)+1;
    s = s(randperm(N)); % Initial random ordering of the states

    for n = 1:N
        sn = s(NofN{n});
        % vector of states in adjacent cells - not allowed

        for k = 1:S
            if max(freq(s(SITE{site_list(n)}))==freq(k))
                sn = [sn k];
                % add states with frequencies already
                % used in the same site
            end
        end

        if length(sn(sn==s(n)))>0
            % if the state of the nth cell is not allowed, change it

            newstates = 1:S; newstates(sn) = 0;
            newstates = newstates(newstates~=0);
            if length(newstates)>0
            newstates = newstates(randperm(length(newstates)));
                if flag >0
```

```
                    penmax = 0; penmin = 1e12; kmax = 1; kmin = 1;
                    % Now choose a new state to either
                    % maximise (if flag =2) or minimise
                    % (if flag = 1) the soft constraint penalty.
                    for k = 1:length(newstates)
                        snew = s; snew(n) = newstates(k);
                        newpen = penalty(snew,n);
                        if newpen>penmax
                            penmax = newpen; kmax = k;
                        elseif newpen < penmin
                            penmin = newpen; kmin = k;
                        end
                    end
                    if flag==1
                        s(n) = newstates(kmin);
                    else
                        s(n) = newstates(kmax);
                    end
                else
                    % if flag is neither 1 nor 2,
                    % pick a new state at random
                    s(n) = newstates(1);
                end
            else
                break
            end
        end
    end
    rn = rn+1;
    % if this hasn't worked, reseed the random
    % number generator and try again.
    if rn>rnmax
        s = []; break
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function NofN = setupNofN(A)
global N
% set up list of neighbours of neighbours

for k = 1:N, A(k,k) = 0; end
A1 = ((A+A')>0);
% make adjacency matrix symmetric for convenience
Asq = A1^2;
% square to give neighbour of neighbour adjacency matrix
for k = 1:N, Asq(k,k) = 0; end
%cells are not neighbours of neighbours of themselves
for n = 1:N
    a = 1:N;
    a = a(Asq(n,:)==true);
    NofN{n} = a;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [X, n] = setuparrays(list)
% For a list of some property of the cells (base station
% controller or site), make a list of cells (X) with the
% same property.

X = []; n = 1;
f = find(list==n); X{n} = f;
while f>0
    f = find(list==n);
    X{n} = f;
    n = n+1;
end
X = X(1:end-1); n = n-2;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function ispermissible = ispermissible(s)
global N
% Check whether a state vector s is permissible

if isempty(s)
    ispermissible = false;
else
    j = 1; ispermissible = true;
    while (ispermissible)&&(j<=N)
        ispermissible = (ispermissiblerow(s,j))&&(ispermissiblesite(s,j));
        % must have different states from its neighbours of neighbours
        % and a different frequency from states in the same site
        j = j+1;
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function ispermissiblerow = ispermissiblerow(s,j)
global NofN
% is the jth cell in a different state
% from neighbours of neighbours?

if isempty(s)
    ispermissiblerow = false;
else
    state = s(NofN{j});
    if length(state(state==s(j)))>0
        ispermissiblerow = false;
    else
        ispermissiblerow = true;
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function ispermissiblesite = ispermissiblesite(s,j)
global site_list SITE freq
% is the jth cell at a different frequency
% from those in the same site?

f = freq(s(SITE{site_list(j)}));
if length(f(f==freq(s(j))))>1
    ispermissiblesite = false;
else
    ispermissiblesite = true;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [s, nfout, ncout] = findpath(sinit,sfinal)
global N M Msim B S  nf ncol bsc_list freq col
% find a permissible path from sinit to sfinal

s{1} = sinit; current = 1; stuck = 0; stuckmax = 100;
snew = sinit; s0 = snew;
fnew = freq(snew); cnew = col(snew);
ffinal = freq(sfinal); cfinal = col(sfinal);
nftogo_old = length(fnew(fnew~=ffinal));
nctogo_old = length(cnew(cnew~=cfinal));
nfout = nftogo_old; ncout = nctogo_old;
% nfout and ncout record progress towards the final state

currentmax = 4*N/M;
while (max(s{current}~= sfinal))&&(stuck<stuckmax)&&(current<currentmax)
    current = current+1;

    % determine which base station controllers have all correct
    % final states and exclude them
    B1 = [];
    for m = 1:M
```

```
    if max(snew(B{m}) ~= sfinal(B{m}))
        B1 = [B1 m];
    end
end

Mcurrent = length(B1);
if Mcurrent > Msim
    r = randperm(Mcurrent);
    B1 = B1(r(1:Msim));
else
    B1 = B1(randperm(length(B1)));
end
% B1 contains the numbers of the base
% station controllers to be used

% Now determine whether the remaining changes
% can be made simultaneously
changeflag = false;
tochange = 1:N; tochange = tochange(snew~=sfinal);
if length(tochange)<=Msim
    bsctochange = sort(bsc_list(tochange));
    changeflag = true;
    for k = 1:length(tochange)-1
        if bsctochange(k) == bsctochange(k+1)
            changeflag = false; break
        end
    end
end

if (Mcurrent <= Msim)&&...
(length(snew(snew~=sfinal))==Mcurrent)&&(changeflag)
    % all of the remaining cells can be changed simultaneously
    snew = sfinal;
else
    for Bc = B1
        % loop over base station controllers
        BBc = B{Bc};
        sc = snew(BBc); sfinalc = sfinal(BBc);
        r = randperm(length(sc));

        kbest = 0; penbest = 1e12;
        for k = 1:length(sc)
            % Try to change a cell to the correct final frequency
            if freq(sc(k))~=freq(sfinalc(k))
                sold = sc(k);
                snew(BBc(k)) = fcol2state(freq(sfinalc(k)),col(sc(k)));
                if (ispermissiblerow(snew,BBc(k)))...
                &&(ispermissiblesite(snew,BBc(k)))
                    pen = penalty(snew,BBc(k));
                    if pen < penbest
                        kbest = k; penbest = pen;
                    end
                end
                snew(BBc(k)) = sold;
            end
        end
        if kbest>0
            % choose a state with a frequency that makes those most
            % favourable change to the soft constraint penalty
            snew(BBc(kbest)) = ...
            fcol2state(freq(sfinalc(kbest)),col(sc(kbest)));
            k = length(sc)+1000;
        end

        if k < length(sc)+10
            r = randperm(length(sc)); k = 1;

            while k<=length(sc)
                % Try to change a cell to the correct final colour
                q = r(k);
                if col(sc(q))~=col(sfinalc(q))
                    sold = sc(q);
```

```
                        snew(BBc(q)) = ...
                        fcol2state(freq(sc(q)),col(sfinalc(q)));
                        if ~((ispermissiblerow(snew,BBc(q)))...
                        &&(ispermissiblesite(snew,BBc(q))))
                            snew(BBc(q)) = sold;
                            k = k+1;
                        else
                            % change the current cell to the final
                            % col state as it is permissible
                            k = length(sc)+1000;
                        end
                    else
                        k = k+1;
                    end
                end
            end

            if k < length(sc)+10
                % no changes to the final state are permissible, so
                % find a permissible random change, if possible
                r2 = randperm(length(sc));
                k = 1; r = randperm(S);
                while k <= length(sc)
                    p = 1; q = r2(k);
                    if (sc(q)~=sfinalc(q))...
                    &&(permissiblechange(sc(q),sfinalc(q)))
                        while p <=S
                            if r(p)~=sc(q)
                                sold = sc(q);
                                snew(BBc(q)) = r(p);
                                if ~((ispermissiblerow(snew,BBc(q)))...
                                &&(ispermissiblesite(snew,BBc(q))))
                                    snew(BBc(q)) = sold;
                                    p = p+1;
                                else
                                    % change the current cell to a
                                    % new random, but not final,
                                    % state, as it is permissible
                                    p = S+1000; k = length(sc)+1000;
                                end
                            else
                                p = p+1;
                            end
                        end
                    end
                    k = k+1;
                end
            end
        end
    end

    fnew = freq(snew); cnew = col(snew);
    nftogo = length(fnew(fnew~=ffinal));
    nctogo = length(cnew(cnew~=cfinal));

    if (nftogo+nctogo>nftogo_old+nctogo_old)||(min(snew==s0))
        % no improvement on this iteration
        stuck = stuck+1;
        current = current-1;
    else
        s{current} = snew; s0 = snew; stuck = 0;
        nftogo_old = nftogo; nctogo_old = nctogo;
        nfout = [nfout nftogo]; ncout = [ncout nctogo];
        disp([current nftogo nctogo])
        % display current progress
    end

end

if (stuck >= stuckmax)||(current>=currentmax)
    % algorithm got stuck
    s = []; nfout = []; ncout = [];
```

```
end

if (stuck<stuckmax)&&(current<currentmax)
    % plot final result
    figure(1), plot(1:current,ncout,'.',1:current,nfout,'*')
    legend('colour','frequency')
    ylim([0 N]), xlim([0 current])
    xlabel('iteration'),ylabel('number still to be changed')
    title(strcat(num2str(nf),' frequencies, ',num2str(ncol),' colours'))
    drawnow
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function s = fcol2state(f,col)
global nf

% convert frequency and colour into state number

s = (col-1)*nf+f;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [f, col] = state2fcol(s)
global nf

%convert state number into frequency and colour

f = s - nf*floor(s/nf); f(f==0) = nf;
col = (s-f)/nf+1;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function permissiblechange = permissiblechange(s1,s2)

global freq col

% check whether a change from state s1 to state s2 can be
% made by changing just one of frequency or colour, not both

if (freq(s1)~=freq(s2))&&(col(s1)~=col(s2))
    permissiblechange = false;
else
    permissiblechange = true;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function penalty = penalty(s,k)
global geolist adpen copen freq

% calculate the soft constraint penalty associated with the kth cell

ico = (freq(s(geolist{k}))==freq(s(k)));
iad = (abs(freq(s(geolist{k}))-freq(s(k)))==1);
penalty = sum([copen{k}(ico) adpen{k}(iad)]);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function pen = totalpenalty(s)
global N

% calculate the total soft constraint penalty for a state vector s

pen = 0;
for k = 1:N
    pen = pen + penalty(s,k);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function setuppenalty(Pa,Pc)
```

```
global geolist adpen copen N

% set up global variables related to the soft constraint penalty

geolist = []; adpen = []; copen = [];
n = 1:N;
for k = 1:N
    geolist{k} = n(Pa(k,:)>0);
    adpen{k} = Pa(k,geolist{k}); copen{k} = Pc(k,geolist{k});
end
```

B Python code

B.1 The Graph object and methods

The following code defines a simple graph object and associated methods in Python.

```
   import sys

class Controller:
    def __init__(self,graph):
        graph.controllers.append(self)
        self.name = 0
        self.bases = []
        self.V = []

class BaseStation:
    def __init__(self,controller,graph):
        self.controller = controller
        controller.bases.append(self)
        graph.bases.append(self)
        self.V = []
        self.name=""

class Vertex:
    def __init__(self,name,base,controller,graph):
        graph.V.append(self)
        self.id = graph.deg
        graph.deg += 1
        base.V.append(self)
        controller.V.append(self)
        self.base = base
        self.controller = controller
        self.E = []
        self.colour = -1
        self.targetColour = -1
        self.freq = -1
        self.targetFreq = -1
        name = name.strip()
        self.name=name
        if graph.findV.has_key(name):
            print "ERROR: %s already exists" %name
            sys.exit()
        graph.findV[name] = self

class Graph:
    def __init__(self):
        self.controllers = []
        self.bases = []
        self.V = []
        self.E = []
        self.deg = 0
        self.good = []
        self.bad = []
        self.findV = {}
```

```
        self.adj=[]
        self.adj2=[]
        self.colourList=[]
        self.freqList=[]
    def recolour(self, vertex, colour):
        if vertex.colour == colour:
            print "   Warning: redundant colour setting"
            return
        vertex.colour = colour
        if colour == vertex.targetColour and vertex.freq == vertex.targetFreq:
            if vertex in self.bad:
                self.bad.remove(vertex)
            else: print "   Warning: vertex missing from the bad list"
            self.good.append(vertex)
        else:
            if vertex in self.good:
                self.good.remove(vertex)
            if vertex not in self.bad:
                self.bad.append(vertex)
    def refreq(self, vertex, freq):
        if vertex.freq == freq:
            print "   Warning: redundant freq setting"
            return
        vertex.freq = freq
        if freq == vertex.targetFreq and vertex.colour == vertex.targetColour:
            if vertex in self.bad:
                self.bad.remove(vertex)
            else: print "   Warning: vertex missing from the bad list"
            self.good.append(vertex)
        else:
            if vertex in self.good:
                self.good.remove(vertex)
            if vertex not in self.bad:
                self.bad.append(vertex)
    def addEdge(self,v1,v2):
        "Return: True if the edge was added, False if it already existed, or v1==v2"
        a = self.V.index(v1)
        b = self.V.index(v2)
        edge = (a,b)
        e2 = (b,a)
        if a==b or edge in self.E or e2 in self.E:
            return False
        self.E.append(edge)
        v1.E.append(edge)
        v2.E.append(e2)
        return True

def allowed(graph,vertex,newcolour=False,newfreq=False,checkend=False):
    "Check if graph allows vertex to have a given newcolour and/or newfreq"

    # NOTE: i think this is still a little buggy for non-symmetric
    #       adjacency matrices.

    if checkend:
        if not newcolour: newcolour=vertex.targetColour
        if not newfreq: newfreq=vertex.targetFreq
    else:
        if not newcolour: newcolour=vertex.colour
        if not newfreq: newfreq=vertex.freq

    # the 2nd hard restraint
    for fred in samebasestation(vertex):
        if checkend:
            if fred.targetFreq == newfreq:
                return False
        else:
            if fred.freq == newfreq:
                return False

    # the main hard restraint
    for fred in adjacency2row(graph, vertex):
        if checkend:
```

```
            if (fred.targetColour == newcolour) \
                    and (fred.targetFreq == newfreq):
                return False
        else:
            if (fred.colour == newcolour) \
                    and (fred.freq == newfreq):
                return False
    return True

def adjacency(graph):
    "Return the adjacency matrix"
    adj=[0]*graph.deg
    for i in range(graph.deg):
        adj[i] = [0]*graph.deg
        for j in range(graph.deg):
            if i==j:
                adj[i][j] = 1
                continue
            if (i,j) in graph.V[i].E:
                adj[i][j] = 1
            else:
                adj[i][j] = 0
    return adj

def adjacency2(graph):
    "Return the adjaceny^2 matrix"

    # NOTE: very inefficient, should probably employ some matrix optimisations

    adj2=[0]*graph.deg
    for i in range(graph.deg):
        adj2[i] = [0]*graph.deg
        for j in range(graph.deg):
            for k in range(graph.deg):
                adj2[i][j] += graph.adj[i][k]*graph.adj[k][j]
            if (adj2[i][j] > 0):
                adj2[i][j] = 1
    return adj2

def adjacency2row(graph, vertex):
    "Return the vertices which a change in vertex will effect"

    adj2row=[]
    for i in range(graph.deg):
        if (graph.adj2[i][vertex.id] != 0) and (vertex.id != i):
            adj2row.append(graph.V[i])
    return adj2row

def obstructingverts(graph, vertex, changefreq=False):
    "Return the vertices most likely to be obstructing a valid change"

    obs=[]

    if changefreq:
        for v in adjacency2row(graph, vertex):
            if vertex.colour == v.colour \
                    and vertex.targetFreq == v.freq:
                obs.append(v)
        for v in samebasestation(vertex):
            if vertex.targetFreq == v.freq:
                obs.append(v)
    else:
        for v in adjacency2row(graph, vertex):
            if vertex.targetColour == v.colour \
                    and vertex.freq == v.freq:
                obs.append(v)

    return obs

def samebasestation(vertex):
    "Return the other vertices which have the same base station as vertex"
```

```
    # NOTE: i'm sure this could be optimised

    samebases=[]
    for v in vertex.controller.V:
        if (v.base.name == vertex.base.name) and (v != vertex):
            samebases.append(v)
    return samebases
```

B.2 The Python version of the recolouring algorithm

Figures 13 and 14 show the Python version of the algorithm, tested on a graph of 600 nodes, first with 25 colours and 50 frequencies and then 12 colours and 18 frequencies, respectively. The main difference between this algorithm and the MATLAB one is that now we only change obstructing vertices randomly.

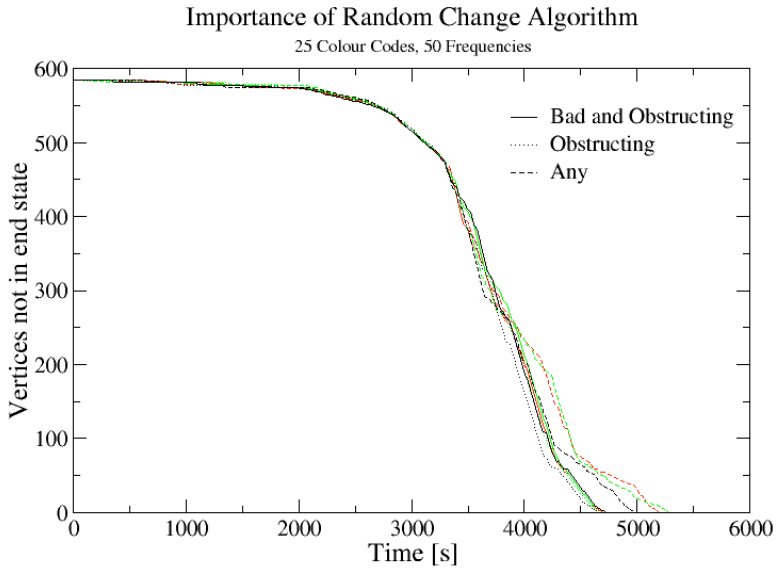

Figure 13: The Python version of the algorithm, 25 colours, 50 frequencies.

```
#!/usr/bin/python

import random
import cPickle
from dangraph import *

# set a maximum time
maximum = 40000

# set the available number of colours/freqs
#colours=75
#frequencies=130
#graph = cPickle.load(open("data-fixed.pickle"))

graph = cPickle.load(open("data-random.pickle"))
colours=len(graph.colourList)
frequencies=len(graph.freqList)

# these can optimise the pool of colours to randomly change to
```

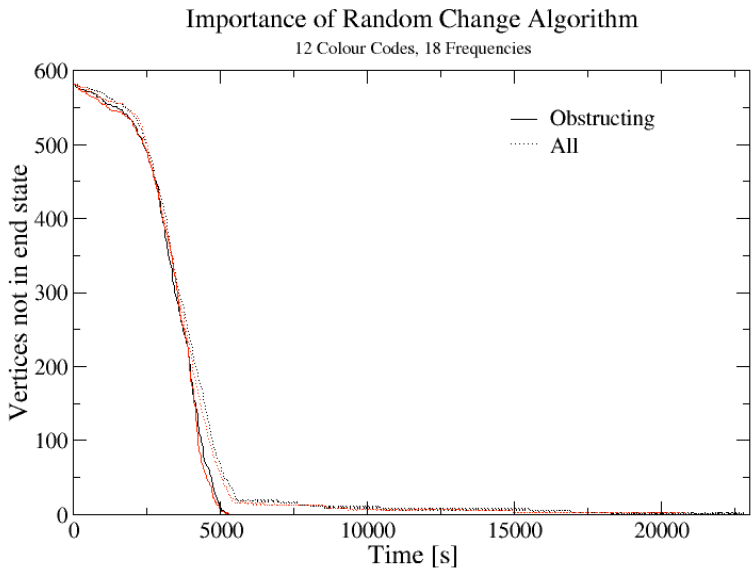

Figure 14: Another Python run with 12 colours and 18 frequencies.

```
maxusedcolours=0
maxusedfreq=0

########################################################################
# step counts the number of parallel steps made
step = 0
# timer is real time seconds
timer = 0

print "#Colours =", colours, "Frequencies =", frequencies
print "#timer bad parallel random"
print timer, len(graph.bad), "0 0"
sys.stdout.flush() # for some reason redirection needs stdout to be flushed

nochanges=0
while (timer < maximum) and (len(graph.bad) != 0) and (nochanges < 100):
    parallel=0
    didcolourchange=0
    didrandom=0
    for bsc in graph.controllers:
        # work on a randomly sorted list of known "bad" vertices
        badvert = filter(lambda v: v in graph.bad, bsc.V)
        if not badvert: continue
        random.shuffle(badvert)
        didstep = 0

        # first change vertices to the final freq if we can
        for v in badvert:
            if v.freq != v.targetFreq:
                if (allowed(graph, v, newfreq=v.targetFreq)):
                    graph.refreq(v, v.targetFreq)
                    didstep=1
                    break
        if didstep == 1:
            parallel += 1
            continue


        # if we couldn't change any freqs, then change colours to the end state
        for v in badvert:
            if v.colour != v.targetColour:
```

```
            if (allowed(graph, v, v.targetColour)):
                graph.recolour(v, v.targetColour)
                didstep=1
                didcolourchange += 1
                break
    if didstep == 1:
        parallel += 1
        continue


    # if we couldn't change any colours or freqs, then do a random change
    # to an obstructing vertex

    # note there is a minor bug that the possibility exists of changing
    # a vertex from another base station controller's domain

    for badv in badvert:
        # find out what needs to be changed in badv
        # set randomcode = 0 for colour, 1 for freq
        if (badv.colour != badv.targetColour) and \
                (badv.freq != badv.targetFreq):
            randomcode=random.randint(0,1)
        elif (badv.colour != badv.targetColour):
            randomcode=0
        else:
            randomcode=1
        # now find out all the obstructing vertices (even good ones)
        if randomcode == 0:
            obstructs=obstructingverts(graph, badv)
        else:
            obstructs=obstructingverts(graph, badv, changefreq=True)
        if not obstructs:
            print "#ERROR:",v.name,"had no obstructing vertices"
            continue
        random.shuffle(obstructs)
        for v in obstructs:
            if randomcode == 0:
                # do a random colour change
                for i in range(2*colours):
                    randomcolour=random.randint(maxusedcolours,colours-1)
                    if (randomcolour != v.colour) \
                            and allowed(graph,v,randomcolour):
                        graph.recolour(v, randomcolour)
                        didstep = 1
                        didrandom += 1
                        didcolourchange += 1
                        break
                if (didstep == 1): break
            elif randomcode == 1:
                # do a random frequency change
                for i in range(2*frequencies):
                    randomfreq=random.randint(maxusedfreq,frequencies-1)
                    if (randomfreq != v.freq) \
                            and allowed(graph,v,newfreq=randomfreq):
                        graph.refreq(v, randomfreq)
                        didstep = 1
                        didrandom += 1
                        break
                if (didstep == 1): break
        if (didstep == 1): break

    if (didstep == 1):
        parallel += 1

# colour changes are 10 secs, freqs 30
if (parallel == didcolourchange) and (parallel != 0):
    timer += 10
elif parallel > 0:
    timer += 30
else:
    nochanges += 1
```

```
    print timer, len(graph.bad), parallel, didrandom
    sys.stdout.flush() # for some reason redirection needs stdout to be flushed

    if (parallel > 0):
        step += 1

print "#DEBUGGING:"
for v in graph.bad:
    print "#BAD:", v.id, v.colour, v.targetColour, v.freq, v.targetFreq
    obstructs=obstructingverts(graph, v)
    random.shuffle(obstructs)
    for vv in obstructs:
        print "#OBS:", vv.id, vv.colour, vv.targetColour, vv.freq, vv.targetFreq
```