DE GRUYTER OPEN

Acta Univ. Sapientiae, Informatica **8,** 1 (2016) 5–15

DOI: 10.1515/ausi-2016-0001

Distance-constrained grid colouring

László ASZALÓS
Faculty of Informatics
University of Debrecen
email: aszalos.laszlo@inf.unideb.hu

Mária BAKÓ
Faculty of Economics
University of Debrecen
email: bakom@unideb.hu

Abstract. Distance-constrained colouring is a mathematical model of the frequency assignment problem. This colouring can be treated as an optimization problem so we can use the toolbar of the optimization to solve concrete problems. In this paper, we show performance of distance-constrained grid colouring for two methods which are good in map colouring.

1 Introduction

The problem of graph colouring is a very active research field nowadays, with very long history. There are different research directions. Some are interested in the theory, namely what the chromatic number of a particular type graph is, i.e. finding the minimum number of colours to colour the nodes of the graph, such that no edge connects nodes of the same colour. Others are interested in practical things, hence invent algorithms that generate colouring with minimal numbers of colours for any, given type of graphs. As the three-colouring is NP-hard problem, we cannot require any algorithm to give a quick solution for every graph.

Computing Classification System 1998: G.1.6
Mathematics Subject Classification 2010: 05C15
Key words and phrases: min-conflicts method, constraint logic programming

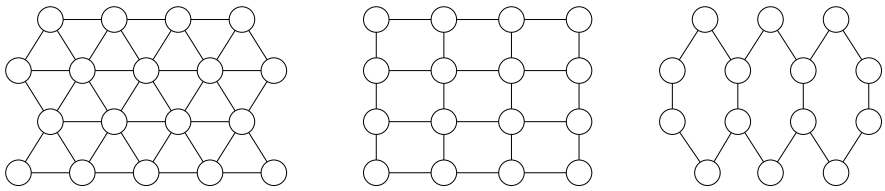

Figure 1: Triangular, square and hexagonal grids of size 4×4

If the graph is fixed, then its colouring with minimal number of colours can be treated as an optimization problem. It is not surprising that almost all of the optimization method have a variant to solve colouring problems [2].

The most elementary method is the exhaustive search for a colouring. This does not mean, that we need to try all the assignment of colours to nodes. If a partial colouring violates any constraint, there is no reason to continue, we can backtrack and start another colouring. Fifth chapter of [6] contains several heuristics to speed up this search by eliminating cases which cannot lead to solutions. This chapter contains an example where the 4-colouring of the map of USA uses around 60 steps instead of 4^{50}, by applying most of the heuristics.

The method of minimal conflicts was introduced in [4]. To solve the 4-colouring of USA map, this method needs about 60 steps too.

Both methods work well for colouring real world maps, but what about colouring other type of graphs? Previously we examined random graphs, so for us their colouring is interesting, but for the sake of reproduction and scalability we will examine very simple graphs: grids (Fig. 1). Although the degrees of the nodes are much smaller than in an ordinary random graph, we hope to get interesting results.

If the number of colours is big, then we use numbers instead of colours. Hence a k-colouring of a graph $G = \langle V, E \rangle$ mathematically is a function $c : V \to \{1, \ldots, k\}$.

The original graph colouring problem has many variants, which are actively researched. One of them is the distance-constrained colouring problem raised from the frequency assignment [3]. In this case given an n-tuple of integers, let say $\langle x_1, x_2, \ldots, x_n \rangle$ and if $d(u, v) = i$ then $|c(u) - c(v)| \geq x_i$, where $1 \leq i \leq n$, $x_j > x_{j+1}$ for $1 \leq j < n$, u and $v \in V$, and d is the graph-distance of the nodes. This distance constrain is denoted by $L_{x_1 \ldots x_n}$ in the literature. Namely at L_{21} colouring the nodes at distance 2 need to be different; and at adjacent nodes the difference of the colour codes is at least 2. Figure 2 shows L_1, L_{21} and

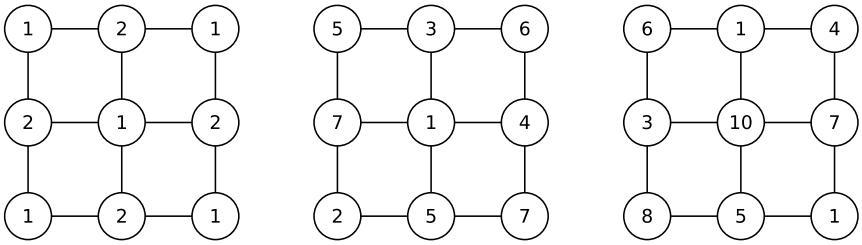

Figure 2: L_1, L_{21} and L_{321} colourings

L_{321} colourings of the 3×3 square grid. On the left, two colours were enough, because only the adjacency counts. At the centre we used seven colours, such that the L_{21} constraints would be satisfied. One could think that less colours might be enough, but no. As we shall see in the following, it is a reasonable step to colour the central node to 1. The central node is adjacent to four nodes, so all of them is 3, or more. These four nodes are second neighbours to each other, so they need to be different. Therefore we need to use at least 6 colours. The nodes in the corner are second neighbours of the central node, so none of them could be 1. As some of them are second neighbour of each others, they cannot be equal. We left to the reader to check the cases and prove, that 6 colour is not enough in this case. Finally on the right there is a 10-colouring, which satisfies L_{321} constraints.

The theoretical results about distant-constrained colouring are surveyed in [5, 1]. In this article we use practical view-point. In the next two section we present the methods we use, and next we discuss our experimental results. Finally we propose new directions.

2 Constraint satisfaction problems

Many problems can be expressed as a constraint satisfaction problem (CSP). Here given an n-tuple of variables: $\langle X_1, \ldots, X_n \rangle$, the tuple of domains of the variables $\langle D_1, \ldots, D_n \rangle$, and a set of constraints which are predicates—boolean valued function of these variables.

Our task is to select values of the variables from the corresponding domains in a such way that all the predicates become true. As the problem SAT could be transformed into CSP, the solution of CSP is NP-hard.

The simplest solution may be gained by using the backtrack method. It

does not use the information given in the problem, so it is usually slow. The 4-colouring of USA map takes more than 10^6 steps. By using variable and value ordering heuristics (minimum remaining values, degree heuristic, least constraining value, forward checking) we can dramatically reduce the number of steps [6].

Many software were built for solving CSP, some of them are stand-alone, while others are modules or libraries and enable us to include the solver in our programs. For its simplicity we have chosen the SWI-Prolog system, and its CLPFD library. Figure 3 contains the source of L_{21} constraints for the coloring of the 3×3 square grid.

Here in the first line we signal the system that we want to use the CLPFD library. Next we define a new predicate (`grid`) which is the counterpart of the function in Prolog. This predicate takes a number as an input and returns a colouring `L`, which is a list of natural numbers. The third line denotes, that the colouring is a 9 long list. The members of this list are our variables (from `X0` to `X8`). The forth line states, that these variables can have `N` different values, the natural numbers from `1` to `N`. The last line of the source starts the systematic search, which determines the values in the list `L`. The word inside the brackets in this line (`ffc`) is a search strategy. The CLPFD library has five different strategies, and this one fits our needs and aims the best. The other lines not mentioned yet contains the constraints. As we need an L_{21} colouring, in the case of adjacent variables (e.g. X0 and X1) the difference of the values is at least 2, so we need to use the absolute value function. Moreover, as the variables have no value at the beginning, instead of relation $\geq$ we need to use the corresponding constraint `#>=`. In case of second neighbours we could use constraint `#\=` (not equal), but for the sake of easier generation of problems, and for the more general L_{pq} constraints we have used `#>=`, again.

Now we are ready to run this code. After loading this file we need to ask the system: `grid(6,L).` The answer arrives immediately: `false`, denoting that there is no solution using six colours. If we ask a 7-colouring with `grid(7,L).` then the answer is one solution. If the solution is too long, the system shows only its beginning. We can ask for the whole solution with a print statement: `grid(7,L), write(L).`

We believe this example shows that in case of constraint logic programming we only need to precisely state the problem, and the solver produces the solution. Moreover, it searches for solutions in a clever way—using the heuristics. If there exists at least one solution, then it determines it; and if not, then after an exhaustive search the system indicates this.

```
:- use_module(library(clpfd)).
grid(N,L) :-
  L = [X0, X1, X2, X3, X4, X5, X6, X7, X8],
  L ins 1..N,
  abs(X0-X1) #>= 2,  abs(X0-X2) #>= 1,  abs(X0-X3) #>= 2,
  abs(X0-X4) #>= 1,  abs(X0-X6) #>= 1,  abs(X1-X2) #>= 2,
  abs(X1-X3) #>= 1,  abs(X1-X4) #>= 2,  abs(X1-X5) #>= 1,
  abs(X1-X7) #>= 1,  abs(X2-X4) #>= 1,  abs(X2-X5) #>= 2,
  abs(X2-X8) #>= 1,  abs(X3-X4) #>= 2,  abs(X3-X5) #>= 1,
  abs(X3-X6) #>= 2,  abs(X3-X7) #>= 1,  abs(X4-X5) #>= 2,
  abs(X4-X6) #>= 1,  abs(X4-X7) #>= 2,  abs(X4-X8) #>= 1,
  abs(X5-X7) #>= 1,  abs(X5-X8) #>= 2,  abs(X6-X7) #>= 2,
  abs(X6-X8) #>= 1,  abs(X7-X8) #>= 2,
  labeling([ffc],L).
```

Figure 3: Prolog program to solve the L_{21}-colouring of 3×3 square grid.

3 Min-conflicts

The n-queens problem is well-known: we need to arrange n queens on a $n \times n$ board, that no one queen can attack any of the others. This problem was a benchmark for a long time, because it was hard to solve, since it contained many constraints: the figures cannot be in the same row, nor in the same column, and neither in the same diagonal. Suddenly, this problem became easy, a 10^6-queens problem could be solved within seconds with the method of minimal conflicts [4].

This method counts the violations of the constraints (conflict), and selects randomly chosen variables with such values for which the number of conflict is minimal. If the solutions are distributed uniformly within the search space, then this method quickly finds a solution, where the number of conflicts is zero.

We wrote in the introduction, that at map colouring problems this method works well. At the usual implementation the starting state is random, i.e. all nodes are assigned a colour randomly and independently from the colours of the other nodes. Next the program randomly select nodes and tries to reduce the number of conflict by recolouring the selected node.

The following question arises: does it works for distance-constrained colouring problems? The first column—with head *random/node*—of Table 1 shows,

that this method works poorly except for a few cases. In most of the cases it cannot solve the problem during hundreds of tests. The numbers in the rubrics denote the rate of the cases when the starting-state and the final-state—after the optimization—satisfied all the constraints.

Naturally the probability of randomly generating a solution is negligible. But the recolouring of the nodes according to the conflicts does not give solution, by our experiments. So it is worth to change the method of recolouring. Originally we recoloured only one node. At the variant we could recolour the whole neighbourhood. For this, at first we uncolour the selected node, and next we check all of its neighbours—if we have $L_{x_1 \ldots x_k}$ colourings, then all nodes whose distance is k or less from the selected node—, and if the colour of this neighbour can be decreased (without introducing conflicts), then we decrease. Finally we choose a colour for the selected node, for which the number of conflicts is minimal. If this whole step increases the number of conflicts, we restore the previous state.

[4] uses the 3-colouring of graphs as an example, and presents a method (Brelaz algorithm) which gives a good starting state. This method uses several considerations, which could be known from solving CSP.

The method begins with an uncoloured graph, and chooses a central node (with the most neighbour), and assigns the minimal colour to it. Next, the method repeatedly chooses an uncoloured node which has the minimum number of conflict-free (possible) colours. In case of a tie the maximal number of uncoloured neighbours determines the winner, or if this gives a tie again, then we can choose randomly from the best nodes. The selected node gets its minimal conflict-free colour.

The last columns in Table 1 denotes the minimal number of colours needed with this method without any constraint on number of colours and hence without conflicts. We typeset with bold the cases when it gives the chromatic number. As in this method a tie is very common, randomness has an important effect. The last columns show the rate of achieving this colouring.

4 Discussion

The second column (with head *random/neighbour*) of Table 1 contains bigger numbers than the preceding column. Hence the neighbour recolouring is advanced according to node recolouring. Unfortunately these numbers are small, even at bigger grids.

The Brelaz algorithm provides a compact colouring, i.e. the colours are very

size	dist.	random node	random neighbour	Brelaz neighbour	c.	r.
		square grid				
3×3	L_{21}	0.00→0.51	0.00→0.64	1.00→1.00	**7**	1.00
	L_{321}	0.00→0.19	0.00→0.64	0.70→0.80	11	0.31
4×4	L_{21}	0.00→0.04	0.00→0.07	0.81→0.81	**7**	0.67
	L_{321}	0.00→0.01	0.00→0.06	0.18→0.18	13	0.15
5×5	L_{21}	0.00→0.00	0.00→0.002	0.00→0.00	8	0.21
	L_{321}	0.00→0.00	0.00→0.002	0.00→0.00	14	0.32
		triangular grid				
3×3	L_{21}	0.00→0.00	0.00→0.011	0.00→0.00	9	0.47
	L_{321}	0.00→0.00	0.00→0.002	0.00→0.18	17	0.05
4×4	L_{21}	0.00→0.00	0.00→0.017	0.00→0.00	11	1.00
	L_{321}	0.00→0.00	0.00→0.00	0.00→0.00	21	1.00
		hexagon grid				
3×3	L_{21}	0.00→0.06	0.00→0.13	0.00→0.00	6	1.00
	L_{321}	0.00→0.25	0.00→0.41	0.33→0.33	**9**	0.32
4×4	L_{21}	0.00→0.03	0.00→0.11	0.05→0.14	**6**	0.10
	L_{321}	0.00→0.02	0.00→0.11	0.39→0.43	**10**	0.35
5×5	L_{21}	0.00→0.00	0.00→0.01	0.32→0.32	**6**	0.26
	L_{321}	0.00→0.00	0.00→0.02	0.00→0.00	**10**	0.23

Table 1: Success rate of the different min-conflicts variants. The column-heads denote the construction of the starting state (random/Brelaz) and the type of recolouring (node/neighbour). The last columns show the number of colours with Brelaz algorithm (c), and the rate of this colouring (r).

close to each other. There is some chance that the colouring will only use a chromatic number of colours, but it makes it almost impossible to recolour the nodes. Hence we can only see a few improvements in the third column of Table 1.

The distance-constrained colouring is a typical combinatorial optimization problem, at least in such sense that it has incredibly numerous local minima, hence the local optimization methods do not perform well, as the table shows. In case of optimization methods using crossover and mutation there is very little chance that by combining two independent individuals we get better individual with less conflicts. By our experiments the methods based on crowd (particle swarm optimization, harmony search, etc.) gave acceptable solutions

only for small grids, for bigger problem the crowd did not containe enough individuals to cope with the huge number of different colourings.

The big drawback of the optimization methods is that at the best case scenario it can only satisfy the existence of a k-colouring of the graph. Otherwise if the optimization method does not give a k-colouring in a fixed time, we *cannot* state, that for this k there does not exist a k-colouring.

The backtrack method gives more in this sense. It looks through the whole state space of the problem systematically. If this search ends without a solution, we can be sure, that the problem has no solution, i.e. the graph has no suitable k-colouring for a given k. The backtrack search in a strict sense is not an exhaustive search, because it does not check all the possible states. But it does not omit any state that can be a solution. Of course the heuristics are very important—because they can help to increase the number of states omitted, even in several orders of magnitude—as they help to discover, that a state is hopeless, i.e. cannot be a solution.

At finding the L_{21}-colouring of the 10×10 hexagonal grid the SWI-Prolog with the default setup found a solution in 311.2 seconds, and with the heuristics `ffc` even 0.07 seconds were enough. Of course as the grid is bigger, the rate of solving times becomes bigger, too.

The modern CPS systems contain many heuristics, so we can use them. If we only have a few colours, then the minimum remaining values' heuristic forces the backtrack many times, because the actual state violates some constraint. If we turn back from the blind alley early, then we perform faster, than if we go up to the walls.

As the number of colours usable at colouring increases this heuristic becomes weaker, it takes more steps and more hypotheses to determine unequivocally the colour of a node, or realize the blind alley at searching. In case of a 5×5 square grid we need to colour 25 nodes. To prove that 25 colours are not enough for the L_{4321}-colouring, the state space has 25^{25} nodes. By using heuristics we can omit most of these states, but a huge number of them remain to be checked, and this takes a lot of time.

We experimented with symmetry-breaking. As we are interested in the existence of the solution, and we need maximum one solution to present it, we can omit the rotated and mirrored solutions. So we can add hypothesis that the central node's colour is a low number (less or equal to the half of the maximal colour), among its neighbours the north one has the minimum value, etc. It is surprising, that finding a solution usually takes more time with this type of acceleration, than without it; the new constraints altered the direction of the search. But the exhaustive search became faster as we reduced the state

size	L_{21}		L_{321}		L_{4321}	
3×3	6:0.023	7:0.002	9: 0.040	10: 0.016	17: 1.574	18:0.544
4×4	6:0.033	7:0.012	11: 0.443	12: 0.017	22:179.563	23:6.235
5×5	6:0.051	7:0.023	11: 0.622	12: 0.036	25: ?	26:0.162
10×10	6:0.081	7:0.048	11: 1.407	12: 0.090	31: ?	32:2.418
20×20	6:0.183	7:0.187	11: 2.022	12: 0.354	35: ?	36:1.679
30×30	6:0.357	7:0.468	11: 2.971	12: 0.885	38: ?	39:5.392
100×100	6:3.398	7:8.009	11:21.673	12:11.819	-: -	-: -

Table 2: Solving time: constraint logic programming for square grids

size	L_{21}		L_{321}		L_{4321}	
3×3	7:0.044	8:0.048	14: 1.147	15:0.461	21:9.544	22:3.179
4×4	8:0.213	9:0.016	16:12.770	17:0.091	26: ?	27:0.167
5×5	8:0.280	9:0.027	17:39.704	18:0.062	29: ?	30:2.511
10×10	8:0.440	9:0.071	23: ?	24:0.169	41: ?	41:0.421
20×20	8:0.727	9:0.282	23: ?	24:1.131	44: ?	45:2.763
30×30	8:0.143	9:0.619	25: ?	26:4.685	46: ?	47:9.072

Table 3: Solving time: constraint logic programming for triangle grids

space to its one eighth in the case of square grids. For example 946 seconds were enough to show that there is no L_{4321}-colouring of 5×5 square grid with 25 colours. This can extend our barriers, but the cases with many colours are hopeless with this method.

The Tables 2–4 shows the time needed to *prove* that for number k there is no k-colouring; and what the search time for a n-colouring is. One cell contains k with proof-time, and n with search-time. If $k+1 = n$, then number n is the chromatic number, because we *proved* that less colours are not enough.

At some cases—denoted with question mark—the time needed for the proof is not known. If k is much smaller than the chromatic number of the graph, then it is easier to violate the constraints, or with other words it is harder to satisfy them. So we got into contradiction much earlier, and even the search space is smaller, and we prove the uncolourability much faster. If n is slightly bigger than the chromatic number, the search time increases as n increases. As we have more and more opportunities to assign a colour to a node, it takes more time to check more cases. If n is much larger than the chromatic number,

size	L_{21}		L_{321}		L_{4321}	
3×3	4:0.009	5:0.009	8:0.022	9:0.009	13: 0.135	14:0.111
4×4	5:0.013	6:0.011	9:0.126	10:0.012	17: 5.710	18:6.724
5×5	5:0.015	6:0.010	9:0.114	10:0.014	19:71.029	20:1.251
10×10	5:0.034	6:0.046	9:1.185	10:0.226	24: ?	25:4.094
20×20	5:0.152	6:0.080	9:2.349	10:1.335	29: ?	30:1.212
30×30	5:0.162	6:0.335	9:2.603	10:4.440	31: ?	32:4.431

Table 4: Solving time: constraint logic programming for hexagon grids

we need less backtracking because more numbers allow satisfying constraints easily, but in the case of backtracking we need to check more cases. By the experiments these two effects compensate each other, and the search times are similar.

For big grids the source with constrains can even take megabytes. Interesting, that for grids with small chromatic number the solving of the problem takes less time than loading into memory and compiling. In the case of L_{4321}-colouring of the 100×100 we did not have enough memory to run the search.

5 Conclusion and future work

In this article we have examined two methods for solving distance-constrained colouring. As colouring in general is a hard task, it was expected that there would be no royal road. The performance of constraint (logic) programming is good for graphs with a small chromatic number. If the chromatic number is high, we could get a solution within a short period of time for a nearby number, but we cannot be sure whether this number is equal with the chromatic number, or not. If there are too many constraints, then the solver cannot solve the problem, due to the shortage of the memory.

Although the method of minimal conflicts has many nice results for solving optimization problems, we were not able to apply it for the distance-constrained colouring. The method needs more fine-tuning, to make bigger realignment of colours in one optimization step.

We are planning to test other optimization methods on this type of problems, in hope of finding a more effective method.

The results of our experiments raise some questions. As Table 2 and 4 shows in case of square and hexagon grids the chromatic number for colouring L_{21} and

L_{321} is constant over a minimal size. Table 3 shows constant chromatic numbers for L_{21}-colourings. Does there exist a constant value in this case for L_{321} or not? Moreover are there constant values for L_{4321}-colourings for these kind of grids? The tables in this column contain many question marks, denoting that the exhaustive search needs a long time (and we had no patience, to wait for the exact time). Our experiments are not enough to answer this question now.

It is obvious, that if we have a colouring of an $n \times m$ grid then we can truncate from it a colouring of a $k \times l$ grid, where $k \leq n$ and $l \leq m$. Can we repeat the first row and column of a 608 different L_{21}-colouring of the 3×3 square grid to get the L_{21}-colouring of the 4×4 square grid? Or in more general: can we rotate, mirror or translate one colouring of a small grid, or combine several colourings of the same small grid to produce colouring of a bigger grid?

Does there exist a circular colouring of the square grids (by treating them as a torus), which can be extended into a colouring of an infinite grid? In this case are the chromatic numbers different, or not?

Acknowledgements

Many thanks for Gábor Halász for valuable questions and remarks.

References

[1] T. Calamoneri, The L(h, k)-labelling problem: A survey and annotated bibliography, *The Computer Journal* **49**, 5 (2006) 585–608. ⇒7

[2] P. Galinier, J. K. Hao, Hybrid evolutionary algorithms for graph coloring, *J. Comb. Optim.*, **3**, 4 (1998) 379–397. ⇒6

[3] W. K. Hale, Frequency assignment: Theory and applications, *Proc. of the IEEE* **68**, 12 (1980) 1497–1514. ⇒6

[4] S. Minton, et al. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems, *Artificial Intelligence* **58,** 1, (1992) 161–205. ⇒6, 9, 10

[5] P. Panigrahi, A survey on radio k-colorings of graphs, *AKCE Int. J. Graphs Comb.* **6**, 1 (2009) 161. ⇒7

[6] S. J. Russel, P. Norvig, *Artificial intelligence: a modern approach*, Pearson 2002. ⇒6, 8

Received: January 31, 2016 • Revised: March 8, 2016