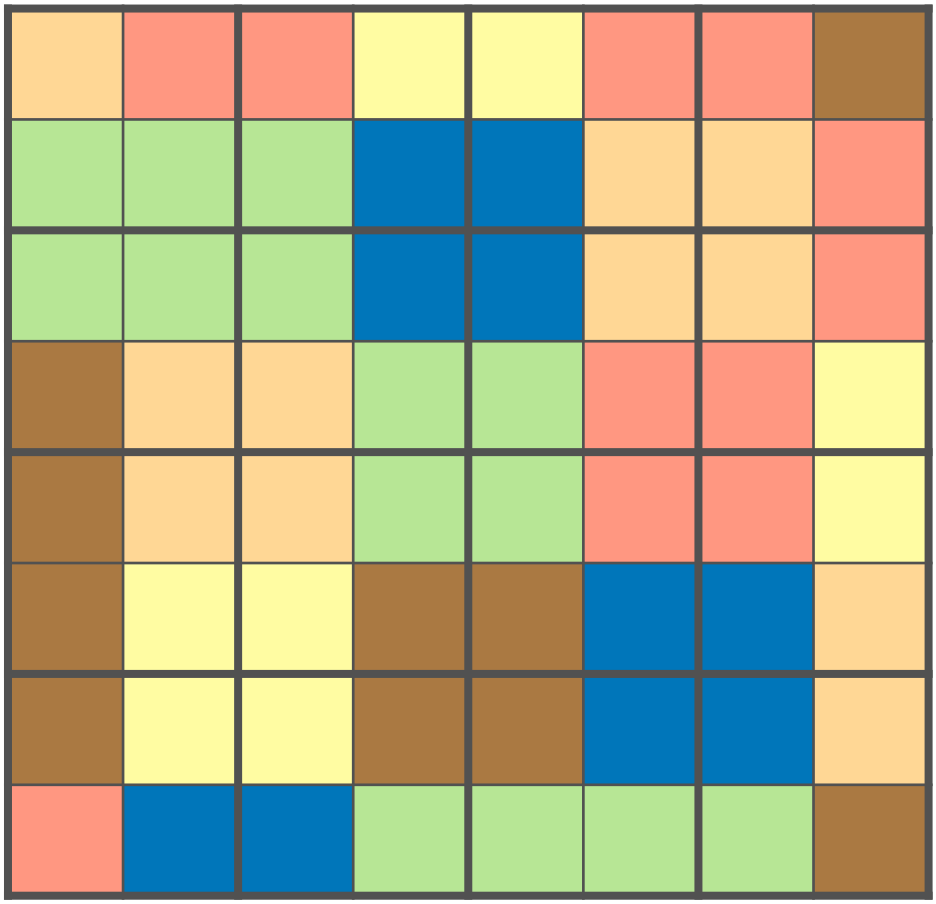
Colors Revisited

Ein Lösungsansatz

Patrick Wenger - 27. März 2022



Einführung

Die Aufgabenstellung ist in den Beilagen 1 (Colors) und 2 (Colors Revisited) im Detail beschrieben. Kurz gesagt geht es darum, 16 Puzzleteile, die aus je 2x2 Farbfeldern bestehen so nebeneinander zu positionieren, dass sämtliche benachbarten Farbfelder übereinstimmen und die Form ein Quadrat beschreibt. Es sollen alle möglichen Lösungsvarianten gefunden werden.

Bei diesem Problem Handel es sich um eine sog. NP-vollständige Aufgabe. D.h. es ist ein Instanz des P = NP?-Problems. Stark vereinfacht bedeutet dies, dass man vorhandene Lösungen sehr schnell darauf überprüfen kann, ob sie den Regeln entsprechen, es aber umgekehrt sehr schwierig ist, solche Lösungen zu finden. Konkret muss man einen Zeitaufwand von der Grössenordnung NP (nicht deterministisch polynominelle Zeit) aufwenden, um alle Lösungen zu finden. Nun ist leider bis heute nicht bekannt, ob es einen Algorithmus gibt, der solche Probleme schneller als in NP lösen kann. Um es genau zu formulieren: man weiss nicht, ob es einen solchen geben kann, denn man kann weder beweisen, dass es einen solchen Algorithmus gibt noch dass es ihn nicht gibt.

Das P=NP?-Problem ist eine der grossen Fragestellungen der Informatik und der Mathematik. Fast jeder namhafte Mathematiker hat sich mit diesem Problem mehr oder weniger intensiv auseinandergesetzt und trotzdem wurde bis heute keine Lösung gefunden. Es ist eines der Millenium-Probleme für deren Lösung ein Preisgeld von 1 Million US$ ausgeschrieben wurde (http://www.claymath.org/millennium-problems). Aufgrund der massiven Bemühungen, einen effizienten Algorithmus zu finden, dies aber bis heute niemandem gelungen ist, ist wohl davon auszugehen, dass es eine solchen schlicht nicht gibt.

Ohne hier näher auf die Problemklasse NP einzugehen, sei einfach darauf hingewiesen, dass selbst bei einer enormen Rechenleistung das Finden von Lösungen schon bei relativ kleinen Probleminstanzen fast unmöglich ist (bzw. wir die Antwort nicht mehr erleben würden). Das macht das vorliegende Rätsel auch so interessant: obwohl es sich nur um ein Pronleminstanz von 16 Teilen handelt, kommen handelsübliche Rechner rasch an ihre Leistungsgrenzen - wenn man das Problem nicht sorgfältig analysiert.

Problemkategorie

Dass es sich bei „Colors Revisited“ um ein algorithmisches Problem handelt, dürfte klar sein. Es stellt sich also primär die Frage, mit welchem Algorithmus am Effizientesten eine Lösung gefunden werden kann.

Meine Problemlösung basiert auf einer Problemtransformation. Zunächst gehe ich von einem sog. CSP-Ansatz (Constrained Satisfaction Problem) aus. Dieses lasse ich dann in ein SAT-Problem (Satisfiability Problem) transformieren und dieses wird schlussendlich durch einen SAT-Solver gelöst. SAT-Solver sind enorm effiziente Tools, deren einzige Aufgabe darin bestehe, ein Instanz eines SAT-Problems als „erfüllbar“ oder „unerfüllbar“ zu kategorisieren. Ein SAT-Instanz ist eine Sammlung von Klauseln und Variablen und sieht beispielsweise wie folgt aus:

(a /\ b) \/ (a /\ c) \/ b

Sofern es mindestens eine Lösung gibt, die den gesamten Ausdruck wahr werden lässt, handelt es sich um eine „SAT“-Instanz. Sonst ist die Instanz „UNSAT“

Das obige Beispiel lässt sich leicht mit einer Wahrheitstabelle lösen:

a	b	c	(a /\ b)	(a /\ c)	b	(a /\ b) \/ (a /\ c) \/ b
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	1	1
0	1	1	0	0	1	1
1	0	0	0	0	0	0
1	0	1	0	1	0	1
1	1	0	1	0	1	1
1	1	1	1	1	1	1

Wir diesem Beispiel entnommen werden kann, hat diese Probleminstanz 5 Belegungen, die den gesamten Ausdruck wahr werden lassen (a=0,b=1,c=0 a=0,b=1,c=1 a=1,b=0,c=1, …) d.h. der Ausdruck ist „SAT“.

Das aufgeführte Beispiel ist natürlich trivial. Die Schwierigkeit bei real-life Problemen besteht natürlich darin, eine geeignete Codierung für des Problems als SAT-Instanz zu finden. Und hier kommen CSPs zum Einsatz. Ein CSP beschreibt ein Problem mit sogenannten Constraints (also Einschränkungen). Anstatt, dass man - wie bei einem klassischen Computer-Programm - dem Computer genau vorgibt, was er zu machen hat, sagt man ihm nur, unter welchen Umständen eine gültige Lösung gefunden werden kann. Man definiert also ein Set an Einschränkungen, die sämtliche Regeln des Problems abbilden, und lässt den Computer dann „sein Ding“ tun. Insofern sind CSP-basierte Ansätze eher mit SQL als mit klassischer Programmierung zu vergleichen.

Der einzig noch fehlende Schritt liegt somit noch in der Transformation des CSP-Modells in ein SAT-Modell. Glücklicherweise gibt es dazu fertige Programme, die dies für einen übernehmen. Eines dieser Programme ist MiniZinc (https://www.minizinc.org). Und dieses habe ich für meinen Lösungsansatz benutzt.

Die Lösung

Mein Lösungsvorschlag sieht wie folgt aus:

```
int: nfp = 4;              % number of fields in each piece
int: nfsp = 2;             % number of fields per side of piece
int: nc = 6;               % number of colors
int: ns = 4;               % number of pieces per side
int: nfs = ns * nfsp;      % number of fields per side
int: npt = ns * ns;        % number of pieces total
int: nft = nfp * npt;      % number of fields total

set of int: noColors = 1..nc;
set of int: noPieces = 1..npt;
set of int: noFields = 1..nft;

enum COLORS = { RED, ORANGE, YELLOW, GREEN, BLUE, BROWN };
array[1..nfs,1..nfs] of var COLORS: SOLMATRIX;


predicate even(var int:x) =
             let { var int: y = x div 2; } in x = 2 * y;

predicate use_piece(COLORS: c1, COLORS: c2, COLORS: c3, COLORS: c4) =
         exists(i,j in 1..nfs where (not even(i) /\ not even(j)))
                ( SOLMATRIX[i,j] == c1 /\ SOLMATRIX[i,j+1] = c2 /\
                  SOLMATRIX[i+1,j] = c3 /\ SOLMATRIX[i+1,j+1] = c4 \/
                  SOLMATRIX[i,j] == c3 /\ SOLMATRIX[i,j+1] = c1 /\
                  SOLMATRIX[i+1,j] = c4 /\ SOLMATRIX[i+1,j+1] = c2 \/
                  SOLMATRIX[i,j] == c4 /\ SOLMATRIX[i,j+1] = c3 /\
                  SOLMATRIX[i+1,j] = c2 /\ SOLMATRIX[i+1,j+1] = c1 \/
                  SOLMATRIX[i,j] == c2 /\ SOLMATRIX[i,j+1] = c4 /\
                  SOLMATRIX[i+1,j] = c1 /\ SOLMATRIX[i+1,j+1] = c3
                );

constraint use_piece(ORANGE,RED,GREEN,GREEN);
constraint use_piece(YELLOW,BLUE,RED,GREEN);
constraint use_piece(YELLOW,RED,BLUE,ORANGE);
constraint use_piece(RED,BROWN,ORANGE,RED);
constraint use_piece(BROWN,GREEN,ORANGE,GREEN);
constraint use_piece(GREEN,BLUE,ORANGE,GREEN);
constraint use_piece(RED,GREEN,ORANGE,BLUE);
constraint use_piece(ORANGE,RED,RED,YELLOW);
constraint use_piece(BROWN,BROWN,YELLOW,ORANGE);
constraint use_piece(ORANGE,GREEN,YELLOW,BROWN);
constraint use_piece(GREEN,RED,BROWN,BLUE);
constraint use_piece(RED,YELLOW,BLUE,ORANGE);
constraint use_piece(BROWN,YELLOW,RED,BLUE);
constraint use_piece(YELLOW,BROWN,BLUE,GREEN);
constraint use_piece(BLUE,GREEN,BROWN,GREEN);
constraint use_piece(BLUE,ORANGE,GREEN,BROWN);

constraint forall(i,j in 1.. nfs-1 where even(j))
         (SOLMATRIX[i,j]==SOLMATRIX[i,j+1] /\ SOLMATRIX[i+1,j]==SOLMATRIX[i+1,j+1]);
constraint forall(i,j in 1.. nfs-1 where even(i))
         (SOLMATRIX[i,j]==SOLMATRIX[i+1,j] /\ SOLMATRIX[i,j+1]==SOLMATRIX[i+1,j+1]);

solve satisfy;
```

Das Programm besteht aus folgenden Blöcken: Variablendeklarationen, predicate-Definitionen (Funktionen), constraint-Definitionen und einer solve-Anweisung:

Variablen-Deklarationen

Die verschiedenen n*-Variablen dienen der dynamischen Verwendung des Programmes. So kann beispielsweise die Seitenlänge einer Probleminstanz von 4x4 auf eine andere (z.B. 3x3) geändert werden.

Die enum COLORS enthält sämtliche möglichen Farben des Puzzles. Dies wäre nicht notwendig, da intern sowieso nur mit dem numerischen Wert (1-6 in unserem Fall) gearbeitet wird. Es vereinfacht aber die Interpretation des Ergebnisses.

Das Array SOLMATRIX ist eine Lösungsvariable (definiert durch das keyword „var“). D.h. diese Variable korrekt zu setzen, ist die Aufgabe des Programmes. Es handelt sich in unserem Fall um eine 8x8 Matrix (so viele Farbfelder gibt es pro Seite, die matchen müssen).

predicate-Definition (Funktionen)

Das Predicate „even“ gibt true zurück, wenn der übergebene Wert gerade ist, sonst false.

Das Predicate „use_piece“ prüft die Verwendung eines bestimmten Puzzelteiles an einer bestimmten Position. Damit ein Puzzle-Teil an einer bestimmten Position stehen kann, müssen die Farbfelder auf allen vier Positionen des Teiles der Vorgabe entsprechen. Bsp: constraint use_piece(ORANGE,RED,GREEN,GREEN); Das Puzzle-Teil, das in der oberen Reihe links die Farbe Orange und oben rechts Rot aufweist sowie in der unteren Reihe, sowohl links wie auch rechts Grün aufweist, kann nur dann auf der Position [i,j] liegen, wenn sämtliche übergebenen Farben in exakt dieser Reihenfolge vorkommen. Sonst muss es sich um ein anderes Teil handeln. Durch die Iteration der beiden Zähler-Variablen i und j ausschliesslich durch die ungeraden Zahlen (not even()), werden keine „versetzten“ Teile akzeptiert und die „exists“-Anweisung bedeutet, dass es mindestens einen Treffer geben muss (für jedes Teil ist eine Postion zu finden).

constraint-Definition

Die Constraints sind die Einschränkungen, die das Programm bei der Suche nach Lösungen berücksichtigen muss. Zunächst gibt es für jedes Puzzle-Teil eine Constraint, die gem. obiger Beschreibung sicherstellt, dass nur gültige Teile zum Einsatz kommen. Ohne diese Constraints würde das Programm einfach Teile erfinden, die den weiteren Constraints genügen.

Danach folgen noch zwei Constraints, die Sicherstellen, dass die Farben aller Nachbarfelder identisch sind mit denjenigen an einer bestimmten Position. Zuerst werden

die horizontalen Einschränkungen gebildet und danach die vertikalen. Beispiel Horizontal:

SOLMATRIX[i,j]==SOLMATRIX[i,j+1] /\ SOLMATRIX[i+1,j]==SOLMATRIX[i+1,j+1]);

das heisst, die Farbe eines Puzzleteils auf der Position [i,j] der Lösungsmatrix muss gleich sein, wie diejenige auf der Position [i,j+1] UDN diejenige der Position [i+1,j] muss gleich sein wie diejenige auf der Position [i+1,j+1] , wobei nur gerade Zeilen geprüft werden.

Solve-Anweisung

solve satisfy startet die Suche nach einer Belegung aller Lösungsvariablen (in unserem Fall nur die SOLMATRIX), wobei sämtliche definierten Constraints einzuhalten sind. Diese CSP-Instanz wird anschliessend in eine SAT-Instanz transformiert.

Ergebnis

Das Programm liefert folgenden Output:

```
minizinc --solver org.chuffed.chuffed --output-time -a /Users/wepa/Documents/
Rätsel/colors/minizinc/colors_solver_v8.mzn
SOLMATRIX =
[| ORANGE,    RED,    RED, YELLOW, YELLOW,    RED,    RED,  BROWN
 |  GREEN,  GREEN,  GREEN,   BLUE,   BLUE, ORANGE, ORANGE,    RED
 |  GREEN,  GREEN,  GREEN,   BLUE,   BLUE, ORANGE, ORANGE,    RED
 |  BROWN, ORANGE, ORANGE,  GREEN,  GREEN,    RED,    RED, YELLOW
 |  BROWN, ORANGE, ORANGE,  GREEN,  GREEN,    RED,    RED, YELLOW
 |  BROWN, YELLOW, YELLOW,  BROWN,  BROWN,   BLUE,   BLUE, ORANGE
 |  BROWN, YELLOW, YELLOW,  BROWN,  BROWN,   BLUE,   BLUE, ORANGE
 |    RED,   BLUE,   BLUE,  GREEN,  GREEN,  GREEN,  GREEN,  BROWN
 |];
% time elapsed: 28.26 s
----------
SOLMATRIX =
[|   RED,  BROWN,  BROWN,  BROWN,  BROWN,  GREEN,  GREEN, ORANGE
 |  BLUE, YELLOW, YELLOW, ORANGE, ORANGE,  GREEN,  GREEN,    RED
 |  BLUE, YELLOW, YELLOW, ORANGE, ORANGE,  GREEN,  GREEN,    RED
 | GREEN,  BROWN,  BROWN,  GREEN,  GREEN,   BLUE,   BLUE, YELLOW
 | GREEN,  BROWN,  BROWN,  GREEN,  GREEN,   BLUE,   BLUE, YELLOW
 | GREEN,   BLUE,   BLUE,    RED,    RED, ORANGE, ORANGE,    RED
 | GREEN,   BLUE,   BLUE,    RED,    RED, ORANGE, ORANGE,    RED
 | BROWN, ORANGE, ORANGE, YELLOW, YELLOW,    RED,    RED,  BROWN
 |];
% time elapsed: 39.85 s
----------
```

```
SOLMATRIX =
[|  BROWN,  GREEN,  GREEN,  GREEN,  GREEN,   BLUE,   BLUE,    RED
 | ORANGE,   BLUE,   BLUE,  BROWN,  BROWN, YELLOW, YELLOW,  BROWN
 | ORANGE,   BLUE,   BLUE,  BROWN,  BROWN, YELLOW, YELLOW,  BROWN
 | YELLOW,    RED,    RED,  GREEN,  GREEN, ORANGE, ORANGE,  BROWN
 | YELLOW,    RED,    RED,  GREEN,  GREEN, ORANGE, ORANGE,  BROWN
 |    RED, ORANGE, ORANGE,   BLUE,   BLUE,  GREEN,  GREEN,  GREEN
 |    RED, ORANGE, ORANGE,   BLUE,   BLUE,  GREEN,  GREEN,  GREEN
 |  BROWN,    RED,    RED, YELLOW, YELLOW,    RED,    RED, ORANGE
 |];
% time elapsed: 57.49 s
----------
SOLMATRIX =
[|  BROWN,    RED,    RED, YELLOW, YELLOW, ORANGE, ORANGE, BROWN
 |    RED, ORANGE, ORANGE,    RED,    RED,   BLUE,   BLUE, GREEN
 |    RED, ORANGE, ORANGE,    RED,    RED,   BLUE,   BLUE, GREEN
 | YELLOW,   BLUE,   BLUE,  GREEN,  GREEN,  BROWN,  BROWN, GREEN
 | YELLOW,   BLUE,   BLUE,  GREEN,  GREEN,  BROWN,  BROWN, GREEN
 |    RED,  GREEN,  GREEN, ORANGE, ORANGE, YELLOW, YELLOW,  BLUE
 |    RED,  GREEN,  GREEN, ORANGE, ORANGE, YELLOW, YELLOW,  BLUE
 | ORANGE,  GREEN,  GREEN,  BROWN,  BROWN,  BROWN,  BROWN,   RED
 |];
% time elapsed: 60.45 s
----------
==========
% time elapsed: 70.10 s
```

Das heisst, es gibt genau 4 Lösungen, wobei es sich dabei eigentlich nur um eine Lösung handelt, die drei mal gedreht wird. Das ist aufgrund der Regeln nicht nur möglich, sondern muss sogar so sein. Demzufolge ist die Anzahl der Lösungen zwingend immer ein Mehrfaches von vier.

Gemäss Kommandoaufruf „minizinc --solver org.chuffed.chuffed --output-time -a /Users/wepa/Documents/Rätsel/colors/minizinc/colors_solver_v8.mzn“ soll zur effektiven Lösung des transformierten Problems der SAT-Solver Chuffed eingesetzt werden (--solver org.chuffed.chuffed) und es sollen sämtliche Lösungen ausgegeben werden (-a). Im vorliegenden. Fall wurde die erste Lösung nach rund 28 Sekunden gefunden, die letzte nach rund 60 Sekunden. Danach dauerte es nochmals rund 10 Sekunden, bis ausgeschlossen werden konnte, dass es noch weitere Lösungen gibt. In den Worten von SAT heisst dies: es gibt exakt 4 Belegungen für das Problem und handelt sich somit um ein „SAT“-Problem.

Wenn ich zu Testzwecken das erste Puzzleteil von (ORANGE,RED,GREEN,GREEN) auf ORANGE,RED,GREEN,BLUE) ändere, dann gibt der SAT-Solver nach rund 96 Sekunden „UNSAT“ aus, was soviel bedeutet, dass es keine Belegung gibt, die die SAT-Instaz wahr werden lässt (also keine Lösung. hat)

minizinc --solver org.chuffed.chuffed --output-time -a /Users/wepa/Documents/Rätsel/colors/minizinc/colors_solver_v8.mzn

```
=====UNSATISFIABLE=====
% time elapsed: 96.46 s
```