Improvements to Satisfy and ChromaticNumber

Curtis Bright, University of Waterloo

Satisfy

The command Satisfy accepts a logical formula and returns a satisfying assignment of the formula if possible and *NULL* if no satisfying assignment exists.

Example

```
> with(Logic):
  F := &and(x, y, z);
  G :=&and(&not(x),&not(y), z);
  H :=&or(x, y);
  J :=&or(&not(x),&not(y));
```

$$F := x \wedge y \wedge z$$
$$G := (\neg x) \wedge (\neg y) \wedge z$$
$$H := x \vee y$$
$$J := (\neg x) \vee (\neg y) \tag{1.1.1}$$

```
> Satisfy(F);
```

$$\{x = true, y = true, z = true\} \tag{1.1.2}$$

```
> Satisfy(G);
```

$$\{x = false, y = false, z = true\} \tag{1.1.3}$$

```
> Satisfy(&and(H, J));
```

$$\{x = false, y = true\} \tag{1.1.4}$$

```
> Satisfy(&and(F, G));
```

- The Boolean satisfiability problem (SAT) is the archetypical NP-complete problem.
- Despite this, a lot of work has gone into making SAT solvers more efficient and in practice they can efficiently solve many problems of interest.

Pigeonhole principle (PHP)

- As an example of a non-trival formula, consider an encoding of the *pigeonhole principle*, the fact that n pigeons cannot fit into $n-1$ holes if each hole can contain at most one pigeon.
- To enocode this proposition we use the variable $x[i, j]$ (where $i = 1,..., n$ and $j = 1,..., n-1$) to represent if pigeon i is in hole j.

```
> n := 3:
> # Every pigeon is in a hole
```

> $PositiveClauses := seq(\&\mathbf{or}(seq(x[i,j], j = 1..n-1)), i = 1..n);$

$$PositiveClauses := x_{1,1} \vee x_{1,2}, x_{2,1} \vee x_{2,2}, x_{3,1} \vee x_{3,2} \quad \textbf{(1.1.5)}$$

> *# No hole contains two pigeons*
> $NegativeClauses := seq(seq(seq(\&\mathbf{or}(\&\mathbf{not}(x[i,j]), \&\mathbf{not}(x[k,j])), i = k+1 .. n), k = 1..n), j = 1..n-1);$

$$NegativeClauses := (\neg x_{2,1}) \vee (\neg x_{1,1}), (\neg x_{3,1}) \vee (\neg x_{1,1}), (\neg x_{3,1}) \vee (\neg x_{2,1}), (\neg x_{2,2}) \vee (\neg x_{1,2}), (\neg x_{3,2}) \vee (\neg x_{1,2}), (\neg x_{3,2}) \vee (\neg x_{2,2}) \quad \textbf{(1.1.6)}$$

> $PHP := \&\mathbf{and}(PositiveClauses, NegativeClauses);$

$$PHP := (x_{1,1} \vee x_{1,2}) \wedge (x_{2,1} \vee x_{2,2}) \wedge (x_{3,1} \vee x_{3,2}) \wedge ((\neg x_{2,1}) \vee (\neg x_{1,1})) \wedge ((\neg x_{3,1}) \vee (\neg x_{1,1})) \wedge ((\neg x_{3,1}) \vee (\neg x_{2,1})) \wedge ((\neg x_{2,2}) \vee (\neg x_{1,2})) \wedge ((\neg x_{3,2}) \vee (\neg x_{1,2})) \wedge ((\neg x_{3,2}) \vee (\neg x_{2,2})) \quad \textbf{(1.1.7)}$$

> $time(Satisfy(PHP));$

0.001 **(1.1.8)**

- Using a larger value of *n*...

> $n := 5:$
> $PositiveClauses := seq(\&\mathbf{or}(seq(x[i,j], j = 1..n-1)), i = 1..n):$
> $NegativeClauses := seq(seq(seq(\&\mathbf{or}(\&\mathbf{not}(x[i,j]), \&\mathbf{not}(x[k,j])), i = k+1 .. n), k = 1..n), j = 1..n-1):$
> $PHP := \&\mathbf{and}(PositiveClauses, NegativeClauses):$
> $nops(PHP);$

45 **(1.1.9)**

> $time(Satisfy(PHP));$

0.004 **(1.1.10)**

- In fact, it is known that the solving method that modern SAT solvers use will take exponential time to determine that *PHP* is unsatisfiable.

Solving methods

- Until Maple 2016, the solving method used was not designed to handle large problems.
- New in Maple 2018: A method option which specifies which solving method to use.
- Currently, the only methods which are supported are "maplesat" and "legacy".

> $time(Satisfy(PHP, method = \text{"maplesat"}));$

0.003 **(1.2.1)**

> $time(Satisfy(PHP, method = \text{"legacy"}));$

1.011 **(1.2.2)**

Default solver

- In Maple 2018 the default SAT solver is MapleSAT.
- In Maple 2016 and 2017 the default SAT solver was Minisat (the solver MapleSAT is based on).
- It is possible to fine-tune the behaviour of MapleSAT by using Satisfy's *solveroptions* parameter.

> $F := \text{:-}Import(\text{"/home/cbright/uf20-01.cnf"}) :$

> $Satisfy(F, method = \text{"maplesat"}, solveroptions = [rnd\_init\_act = true, random\_seed = 1]);$
> $Satisfy(F, method = \text{"maplesat"}, solveroptions = [rnd\_init\_act = true, random\_seed = 2]);$

$$\{B = true, B0 = false, B1 = false, B10 = false, B11 = true, B12 = true, B13 = true, B14 = false, B15 = true, B16 = false, B17 = false, B18 = true, B2 = true, B3 = false, B4 = false, B5 = false, B6 = false, B7 = false, B8 = true, B9 = false\}$$

$$\{B = false, B0 = true, B1 = true, B10 = false, B11 = false, B12 = true, B13 = true, B14 = false, B15 = true, B16 = true, B17 = true, B18 = true, B2 = true, B3 = false, B4 = false, B5 = false, B6 = true, B7 = true, B8 = true, B9 = true\} \quad (1.3.1)$$

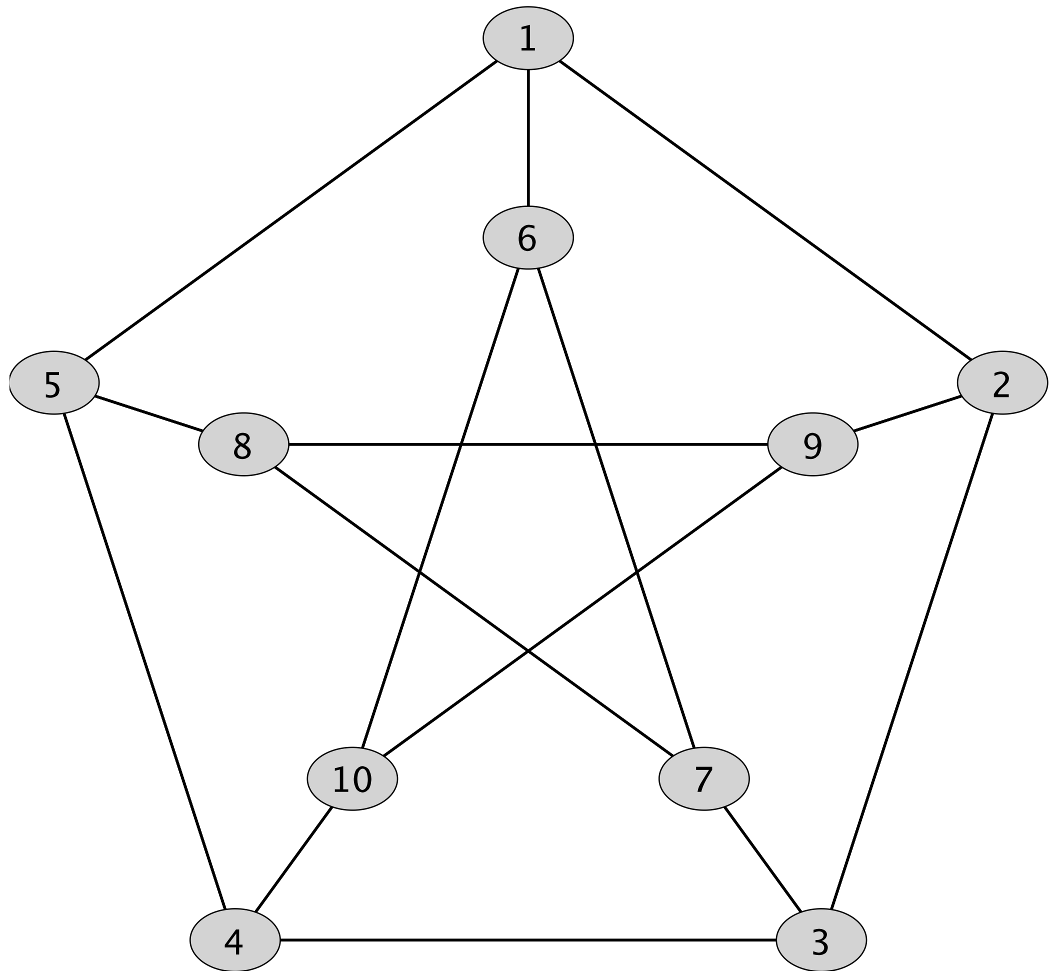
ChromaticNumber

The command ChromaticNumber accepts a graph and returns the minimum number of colours necessary to colour the vertices of the graph so that no adjacent vertices are coloured the same.

Examples

> $with(GraphTheory) :$
> $with(SpecialGraphs) :$

> $P := PetersenGraph(\,) :$
> $DrawGraph(P);$



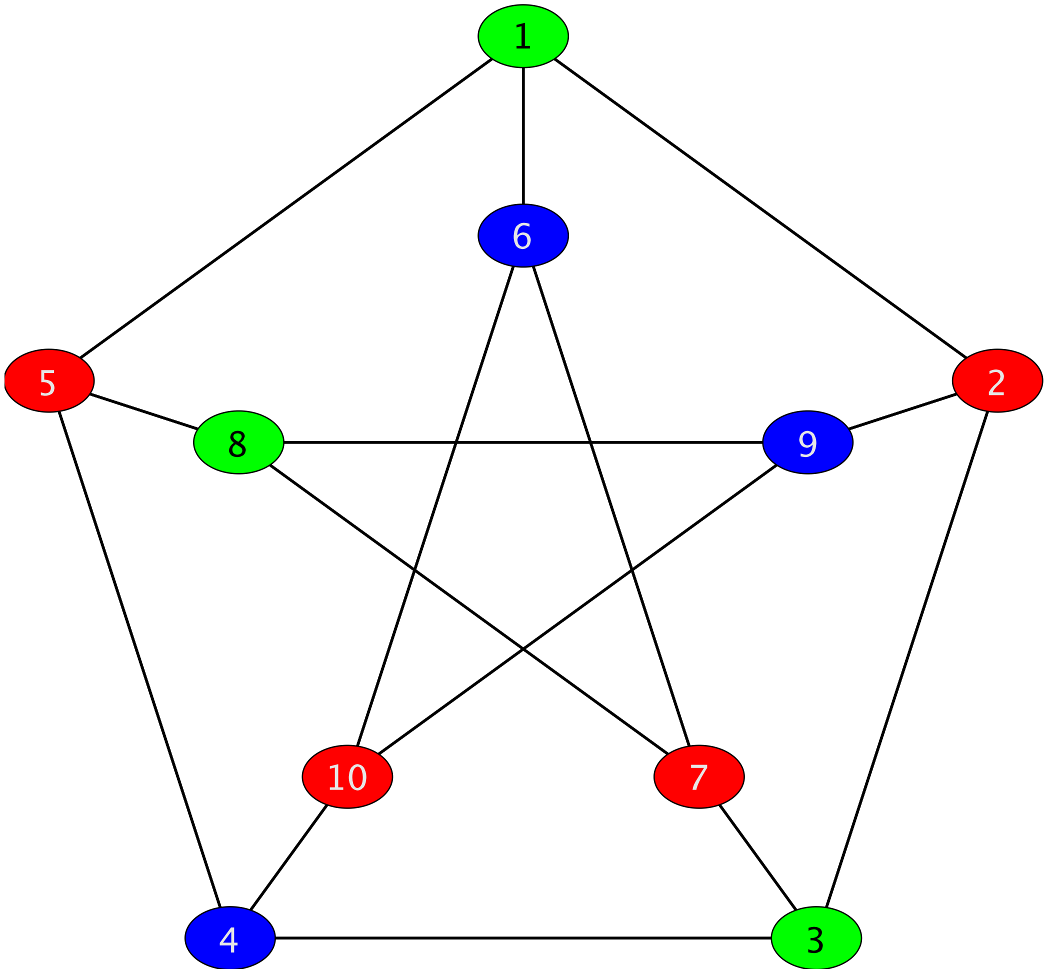
> *ChromaticNumber*(*P*);

3 **(2.1.1)**

> *ChromaticNumber*(*P*,'*col*') :
col;

[[2, 5, 7, 10], [4, 6, 9], [1, 3, 8]] **(2.1.2)**

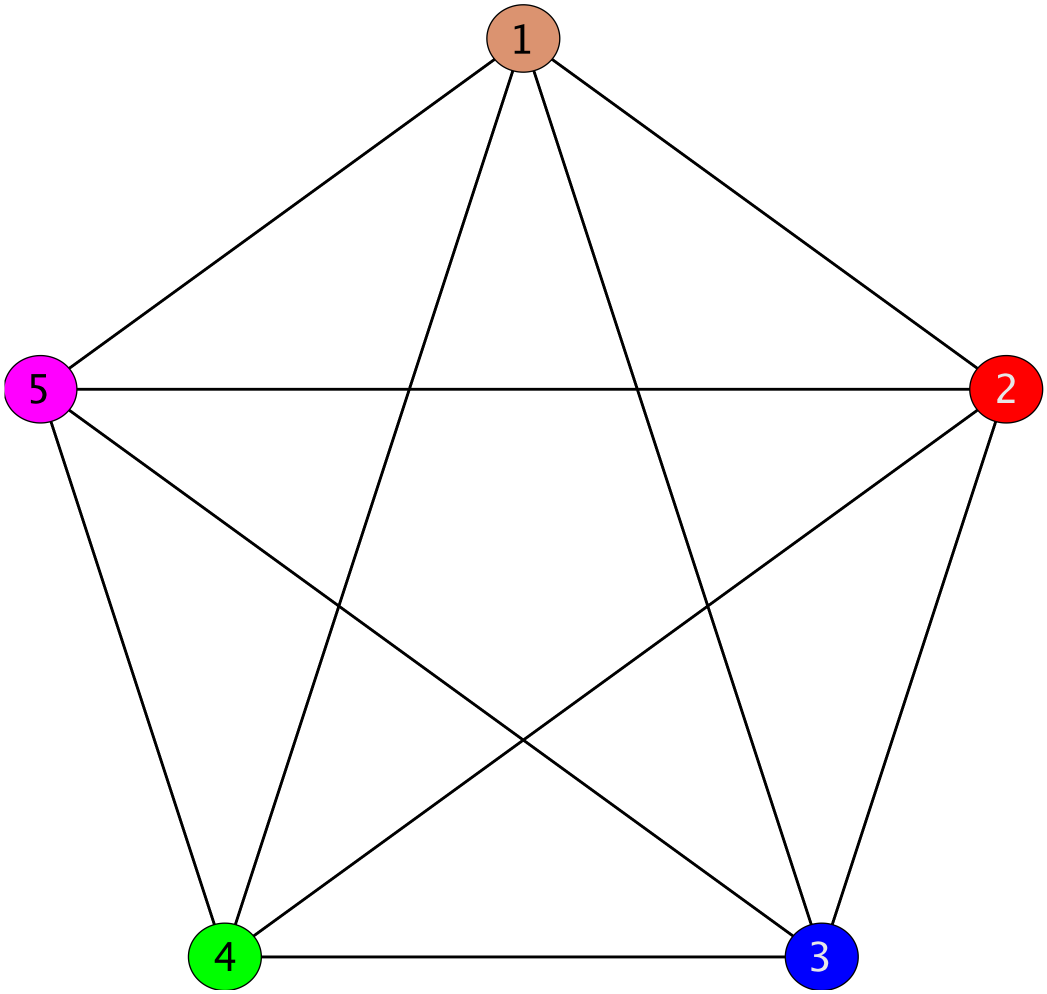
> *HighlightVertex*(*P*, *col*[1],'*red*');
HighlightVertex(*P*, *col*[2],'*blue*');
HighlightVertex(*P*, *col*[3],'*green*');
DrawGraph(*P*);




```
> K := CompleteGraph(5) :
  ChromaticNumber(K,'col');
```

$$5 \tag{2.1.3}$$

```
> HighlightVertex(K, col[1],'red');
  HighlightVertex(K, col[2],'blue');
  HighlightVertex(K, col[3],'green');
  HighlightVertex(K, col[4],'magenta');
  HighlightVertex(K, col[5],'tan');
  DrawGraph(K);
```



- Maple's previous method of computing the chromatic number of a graph computed a max clique inside the graph as a first step.
- The size of a max clique in a graph gives lower bound on the chromatic number.
- When the max clique is as large as possible (i.e., in the case of a complete graph) the method performs very well.
- The method does not perform particularly well in general (even just finding a max clique is NP hard).

Opportunity to use SAT solver

- The problem is naturally translated into a Boolean satisfiability setting.
- Say we want to determine if a graph with n vertices V is k-colourable.
- For each vertex $v \in V$ we use the Boolean variables $v[1], \ldots, v[k]$ with $v[i]$ denoting that v can be coloured with colour i.

Positive clauses

Each vertex has to be coloured some colour: $v[1] \vee v[2] \vee \cdots \vee v[k]$ for each $v \in V$

Negative clauses
Each vertex cannot be coloured two colours: $\neg v[c] \vee \neg v[d]$ for each pair of distinct colours (c, d) and $v \in V$
Adjacent vertices cannot be coloured the same color: $\neg u[c] \vee \neg v[c]$ for each pair of adjacent vertices (u, v) and each colour c.

- For each $k = 1, 2, 3, ..., n$ we construct the above Boolean formulas S_k and check whether the set of all such formulas is satisfiable.
- We know that S_k is satisfiable for $k = n$ and unsatisfible for $k = 1$ (assuming there is at least one edge).
- The value of k for which S_k is satisfiable but S_{k-1} is unsatisfiable is the chromatic number of the graph.

In practice: hard cases

- The hardest set of formulas to determine the satisfiability of is S_{k-1} where k is the chromatic number of the graph.
- For example, when G is the complete graph on n vertices the formulas S_{n-1} say that the complete graph can be coloured with $n-1$ colours which is false (this is equivalent to the pigeonhole principle).
- When $n = 11$ it starts taking the SAT solver minutes to determine that S_{n-1} is not satisfiable, even though the complete graphs should be easy to compute the chromatic number for (and Maple's previous implementation instantly solves this case).

In practice: continued...

- However, the SAT method typically outperforms the previous Maple implementation.
- In fact, running both methods on a set of competition benchmarks the SAT method was always faster and solved a number of benchmarks that the previous method could not (in a reasonable amount of time).
- In short, the SAT method was only slower on complete graphs.

What to do?

- The SAT method is generally better but performs poorly on complete (or almost-complete) graphs.
- Because complete graphs should be some of the easiest graphs to colour, using this method by itself is not adequate.
- We arrived at a **hybrid** stategy: we run both methods in parallel and return the result of whichever method finishes first.
- Each method was run on a separate node using the Grid package.

Code snippet:

```
> Grid:-Setup(numnodes = 2);
  Grid:-Run(0, GraphTheory:-ColorOptimal, [args]);
```

```
Grid:-Run(1, GraphTheory:-ColorSAT, [args]);
 firstnode := Grid:-WaitForFirst( );
Grid:-Interrupt( );
Grid:-Wait( );
result := Grid:-GetLastResult(firstnode);
```

Example of performance

- In the following example Maple 2017 is unable to determine the chromatic number after an hour of CPU time while Maple 2018 does so in seconds.
- The example is a random graph which appears in the paper "Optimization by Simulated Annealing: An Experimental Evaluation; Part II, Graph Coloring and Number Partitioning".

```
> G :=:-Import("example/DSJC125.1.s6", base = datadir);
  time(ChromaticNumber(G));
```

G :=

Graph 1: an undirected unweighted graph with 125 vertices and 736 edge(s)

0.281 **(2.6.1)**