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

```haskell
> 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 \land y \land z
  G := (\neg x) \land (\neg y) \land z
  H := x \lor y
  J := (\neg x) \lor (\neg y)

> Satisfy(F);
  {x = true, y = true, z = true}

> Satisfy(G);
  {x = false, y = false, z = true}

> Satisfy(&and(H, J));
  {x = false, y = true}

> 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-trivial 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 encode this proposition we use the variable \( x[i, j] \) (where \( i = 1, \ldots, n \) and \( j = 1, \ldots, n - 1 \)) to represent if pigeon \( i \) is in hole \( j \).

```haskell
> n := 3 :
> # Every pigeon is in a hole
```
Using a larger value of $n$...

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".

```maple
> time(Satisfy(PHP, method = "maplesat"));
0.003
```

```maple
> time(Satisfy(PHP, method = "legacy"));
1.011
```

### 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 \texttt{solveroptions} parameter.

```maple
> F := Import("/home/cbright/uf20-01.cnf") :
> Satisfy(F, method = "maplesat", solveroptions = [ rnd_init_act = true, random_seed = 1 ]);
Satisfy(F, method = "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}
```

\textbf{ChromaticNumber}

The command \texttt{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.

\textbf{Examples}

```maple
> 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);
\[ K := \text{CompleteGraph}(5) : \]
\[
\text{ChromaticNumber}(K, 'col');
\]
\[
\text{HighlightVertex}(K, \text{col}[1], 'red');
\text{HighlightVertex}(K, \text{col}[2], 'blue');
\text{HighlightVertex}(K, \text{col}[3], 'green');
\text{HighlightVertex}(K, \text{col}[4], 'magenta');
\text{HighlightVertex}(K, \text{col}[5], 'tan');
\text{DrawGraph}(K);\]

\[(2.1.3)\]
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 $v[1], ..., v[k]$ with $v[i]$ denoting that $v$ can be coloured with colour $i$.

**Positive clauses**

Each vertex has to be coloured some colour: $v[1] \lor v[2] \lor \cdots \lor v[k]$ for each $v \in V$.
**Negative clauses**
Each vertex cannot be coloured two colours: \( \neg v[c] \lor \neg v[d] \) for each pair of distinct colours \((c, d)\) and \(v \in V\).  
Adjacent vertices cannot be coloured the same color: \( \neg u[c] \lor \neg v[c] \) for each pair of adjacent vertices \((u, v)\) and each colour \(c\).

- For each \(k = 1, 2, 3, \ldots, n\) 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 unsatisfiable 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** strategy: 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]);
```
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".

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

\[
\text{Graph 1: an undirected unweighted graph with 125 vertices and 736 edge(s)}
\]

0.281