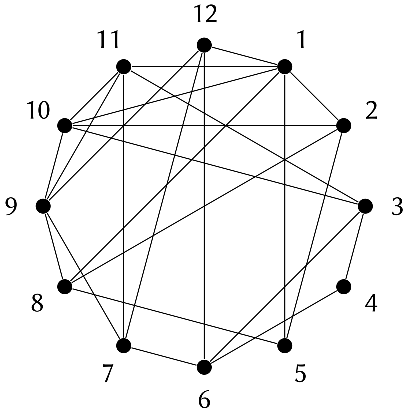
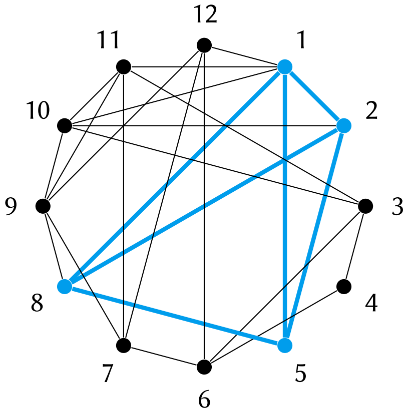
What Maximum Clique Algorithms Can Teach Us, and Vice-Versa

Ciaran McCreesh

The Maximum Clique Problem



The Maximum Clique Problem



Ancient History

- Second DIMACS Implementation Challenge, 1992-93.
  - Maximum clique.
  - Graph colouring.
  - Satisfiability.
- "Recent results in complexity theory have shown that, in a worst-case sense, the problems that are the subject of this Challenge are not only hard to solve optimally, they are hard to approximate. Our goal in this Challenge is to provide an impetus for a coordinated attack on the question of how hard they are in practice." [1]

[1] http://archive.dimacs.rutgers.edu/pub/challenge/call.txt

File Format

```
p edge 12 25
e 1 2
e 1 5
e 1 8
e 1 10
e 1 11
e 1 12
e 2 5
e 2 8
e 2 10
e 3 4
e 3 6
e 3 10
e 3 11
e 4 6
e 5 8
e 6 7
e 6 12
e 7 9
e 7 11
e 7 12
e 8 9
e 9 10
e 9 11
e 9 12
e 10 11
```

Set of Instances

- Random graphs with various sizes, densities, and models.
- Graphs with a large known clique hidden in a quasi-random graph.
- Fault diagnosis applications.
- Coding theory (Hamming, Johnson).
- Mathematical conjectures (Keller, Steiner triples).

Experimental Protocol

"BENCHMARKING: Whether you are studying exact or approximate algorithms, this code can serve as a simple benchmark for calibrating your machine's speed relative to those of the other participants. If possible, please compile and run it on the supplied p=0.5 random graphs r100.5.b, r200.5.b, r300.5.b, r400.5.b, and r500.5.b and report your times. For comparison purposes, the SGI Challenge "user" times (not counting input time) for one run on each graph were 0.08, 1.00, 7.87, 47.68, and 179.98 seconds, respectively. The code can also be used as a common point of reference for papers implementing exact clique algorithms."[2]

[2] http://archive.dimacs.rutgers.edu/pub/challenge/graph/solvers/README

Experimental Protocol

```
$ gcc -o dfmax -O2 dfmax.c
dfmax.c:70:1: warning: return type defaults to 'int' [-Wimplicit-int]
dfmax.c:81:11: warning: implicit declaration of function 'time'; did y
dfmax.c:90:3: warning: implicit declaration of function 'exit' [-Wimpl
dfmax.c:90:3: warning: incompatible implicit declaration of built-in f
dfmax.c:15:1: note: include '<stdlib.h>' or provide a declaration of '
dfmax.c:92:2: warning: implicit declaration of function 'strcpy' [-Wim
dfmax.c:92:2: warning: incompatible implicit declaration of built-in f
dfmax.c:15:1: note: include '<string.h>' or provide a declaration of '
dfmax.c:106:25: warning: implicit declaration of function 'atoi' [-Wim
dfmax.c:148:13: warning: implicit declaration of function 'maxind'; di
dfmax.c:256:7: warning: implicit declaration of function 'get_params'
dfmax.c:287:16: warning: implicit declaration of function 'calloc' [-W
dfmax.c:287:16: warning: incompatible implicit declaration of built-in
dfmax.c:308:2: warning: incompatible implicit declaration of built-in
dfmax.c:253:2: warning: ignoring return value of 'fread', declared wit
$ time dfmax r500.5
real    0m4.100s
$ time glasgow_clique_solver r500.5
real    0m0.695s
```

Experimental Protocol

"4. The CPU mark. In order to compensate for different processor speeds, Controller will standardize (i.e., scale) times according to the CPU marks provided by PassMark Single Thread Performance 3 . Currently, the top CPU mark is 3,255, while mid-range desktop processors have marks around 2,000. So, we choose the mark 2,000 to define our standardized times. This means that if a run is performed in a processor Intel Core i9-9900T @ 2.10GHz that has mark 2,400, all local elapsed times will be multiplied by 1.2 to obtain the corresponding standardized times. This also means that a standardized time limit of 1,800 seconds will actually correspond to 1,500 seconds in that particular machine."[2]

[2] http://dimacs.rutgers.edu/programs/challenge/vrp/cvrp/

A Very Incomplete Guide to Maximum Clique Solvers

- Since 1992, several papers per year.
- Many omissions, particularly of ideas that only showed up in one paper.
- Prosser, Algorithms 5(4) 2012 and my PhD thesis have more history.

Branch...

- Grow a maximum clique A, from the vertex set P:
- Pick a vertex v. Either v is in a maximum clique:
  - Let P' be the set of vertices in P that are adjacent to v.
  - Recurse, finding a maximum clique in $A' = A \cup \{v\}$ and P'.
- Or it isn't. Throw out v from P and pick another v.

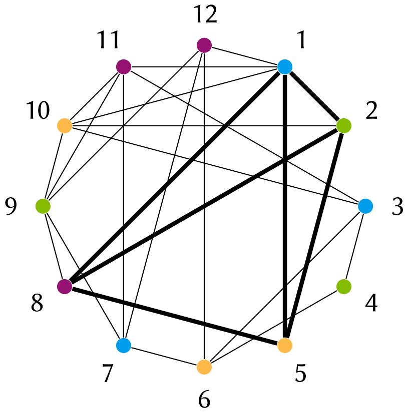
…and Bound

- Keep track of the largest clique found so far, the *incumbent*, ω.
- If $|C| + |P| \leq \omega$, backtrack.

Degree Filtering

- Every vertex in a clique of size ω has degree at least $\omega - 1$.
- Can calculate degree either in the original graph, or in $A \cup P$.

The Colour Bound



The Colour Bound

- But how do we produce a colouring quickly?
    - Greedily!
    - But what order do we use for colouring the vertices?
- More details: papers by Etsuji Tomita and co-authors.
- Interesting fact: the colour bound is *strictly* better than degree.

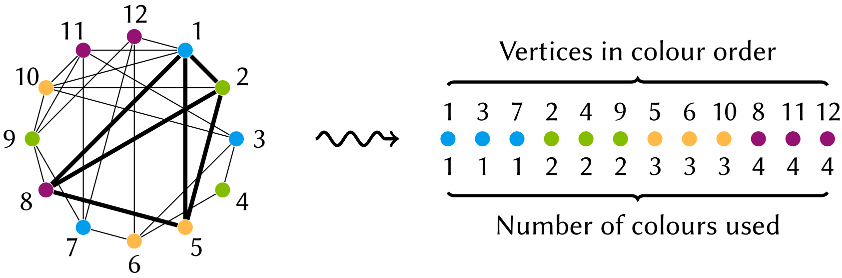
Bit-Parallelism

- Propagation is a few clock cycles.
- Over a million full colourings and recursive calls per second on a 512 vertex graph.
- More details: papers by Pablo San Segundo and co-authors.

Thread-Parallelism

- Average speedups of around 28× on a 32 core processor, but high variance.
- Large superlinear speedups extremely common.
- More details: McCreesh and Prosser, ACM TOPC 2(1) 2015.

Colour Ordering



Colour Ordering

- Vertices in the rightmost colour class are "generally expected [to have a] high probability of belonging to a maximum clique" according to Tomita and Kameda, J. Global Optimization, 37(1) 2007.

Colour Ordering

- Vertices in the rightmost colour class are “generally expected [to have a] high probability of belonging to a maximum clique” according to Tomita and Kameda, J. Global Optimization, 37(1) 2007.
- It’s not true.

Colour Ordering

- Vertices in the rightmost colour class are “generally expected [to have a] high probability of belonging to a maximum clique” according to Tomita and Kameda, J. Global Optimization, 37(1) 2007.
- It’s not true.
- Right to left is still better even if the algorithm is only proving optimality.
- Better clique algorithms have *worse* anytime behaviour and take *longer* to find a strong incumbent.
- More details: McCreesh and Prosser, CP 2014, and part two of this talk.

Priming

- Run local search before starting the main algorithm, to get a good initial incumbent.
- “We run the ILS heuristic with 100000 scans for all the considered instances except gen400_p0.9_55 and p_hat1000-3 for which we use 60 millions scans because these two instances are computationally difficult.”, Maslov et al, J. Global Optimization, 59(1) 2014.
- More details: papers by Etsuji Tomita and Pablo San Segundo.

Stronger Bounds

- Produce a better colouring than a greedy colouring.
- Produce a bound that's better than colouring.
  - Fractional colourings?
  - MaxSAT-inspired bounds?

MaxSAT Bounds

- Encode as MaxSAT:
  - Hard clauses for each non-edge: $\overline{x_2} \vee \overline{x_3}$.
  - Naïve encoding: soft clause for each x_i.
  - Better encoding: soft clause for each colour class, $x_1 \vee x_3 \vee x_7$.
- MaxSAT not competitive:
  - Unit propagation way too slow.
  - Learned clauses not re-used?
  - Need to use dynamic colourings.
- But we can steal MaxSAT conflict analysis algorithms.
- This can, for example, identify three differently coloured vertices that do not form a triangle.
- More details: papers by Pablo San Segundo and Chu-Min Li.

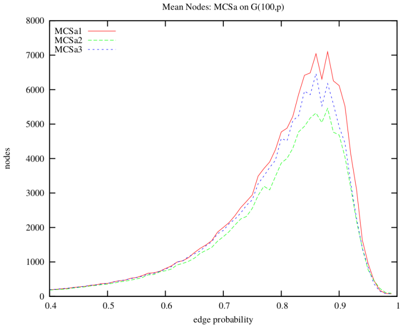
Proof Logging

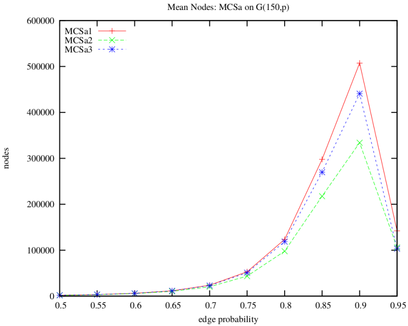
- A solver widely claimed to be state of the art is buggy, but nearly always produces the right answer.
- Proof logging could catch this, but...
  - Clique solvers are much faster than SAT solvers.
  - Bounds involve strong reasoning.
- More details: Gocht et al, CP 2020 and part two of this talk.

The DIMACS Instances

- Heavy bias towards "large hidden solution" instances and random instances.
- The application instances are extremely easy.
- Still a few open instances, and a few more that can only be solved with years of CPU time.
- Sufficiently few hard instances that, e.g. random permutations can easily change which algorithm is "state of the art".
  - Particularly on the "large hidden clique" instances.
- Still widely used.

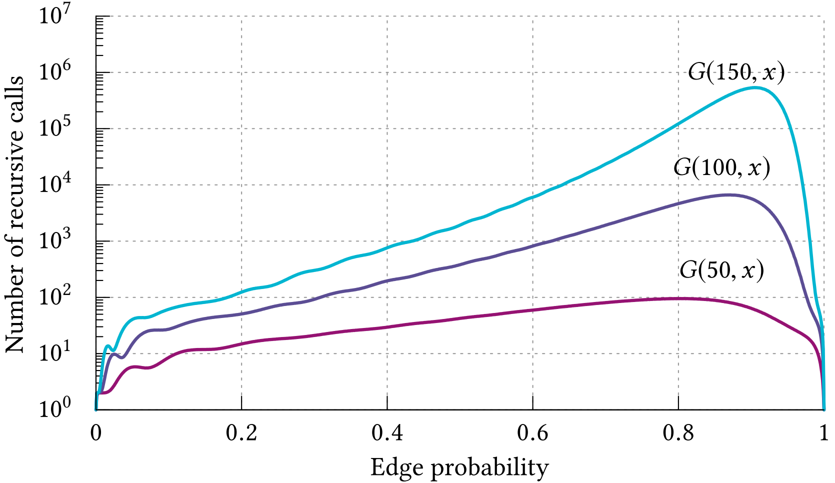
Random Instances



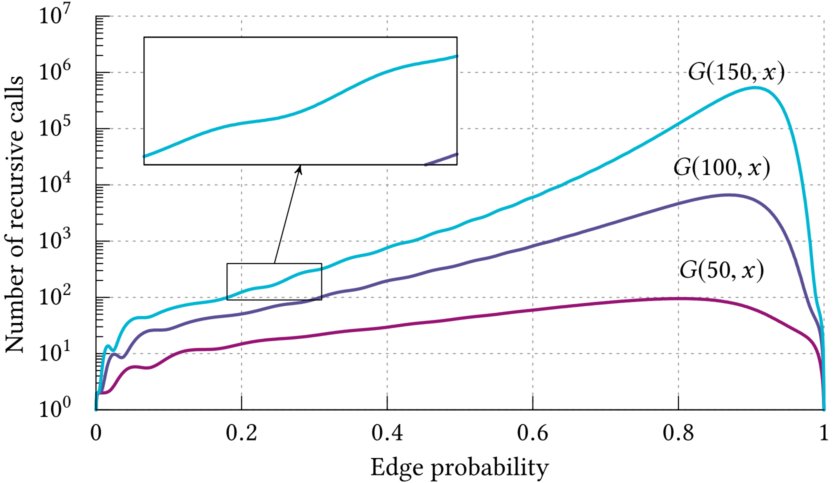


Prosser, Algorithms 5(4) 2012. Both plots have 100 samples per density step. The left-hand plot seems to go up in density steps of 0.01, and the right-hand plot, 0.05.

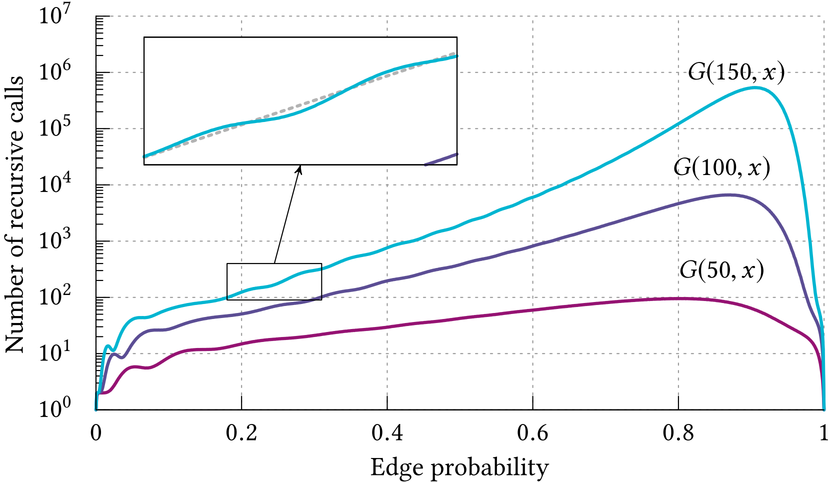
Random Instances



Random Instances



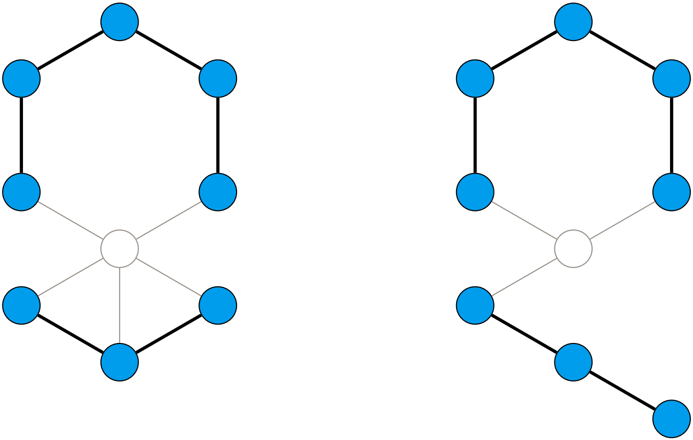
Random Instances



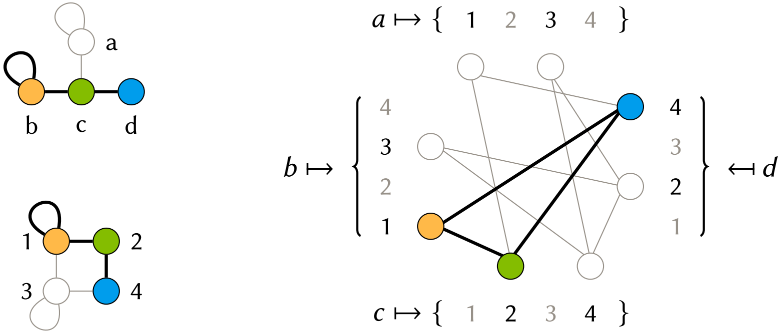
Large Sparse Graphs

- Biggest DIMACS graph: 4,000 vertices, density 0.5.
  - This will fit in an adjacency matrix.
- Often claimed that large sparse graphs are "more realistic".
  - Not immediately clear why anyone would want to find a clique in a road network...
- Not in this talk: algorithms that run on large sparse graphs.
  - Sneaky data structures.
  - Replace the colouring with something faster.

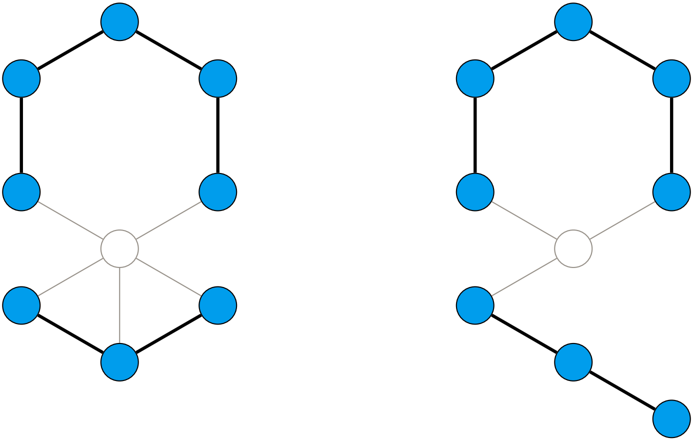
Maximum Common Subgraph



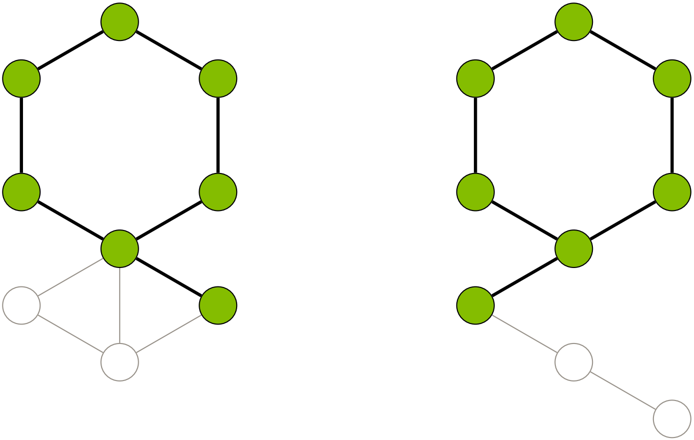
Maximum Common Subgraph



Maximum Common Connected Subgraph?



Maximum Common Connected Subgraph?



Enumeration

- Output all maximal cliques.
- Several published tables of results are wrong…

Weighted

- Vertices have weights.
  - Weighted colour bounds?
- Lack of good benchmark instances, so authors reuse DIMACS graphs and assign weight $v \bmod 200 + 1$ to vertex v.
- Why this is bad, and better instances: McCreesh et al, CP 2017.
- Proof logging: Gocht et al, CP 2020.
  - Cutting planes proofs are awesome.

Relaxations

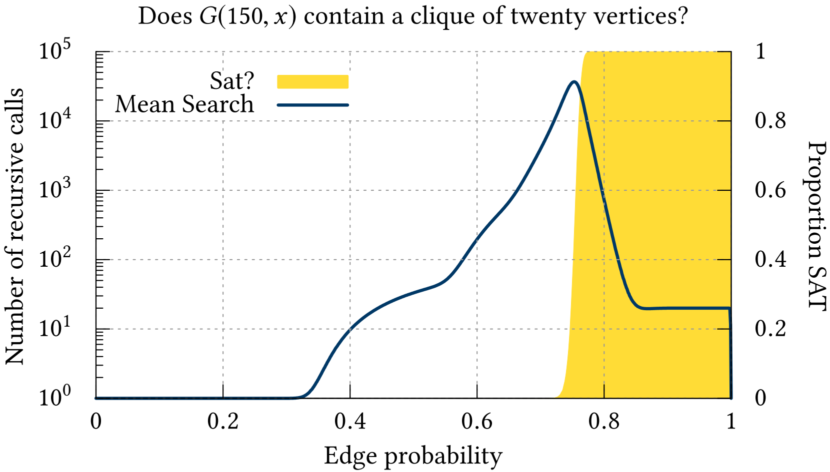
- Distance: k-clique, k-club.
- Degree relaxations: k-plex.
- Density relaxations.
- More: Pattillo et al, Eur. J. of Operational Research, 226(1) 2013.

https://ciaranm.github.io/

ciaran.mccreesh@glasgow.ac.uk

University of Glasgow

Phase Transitions



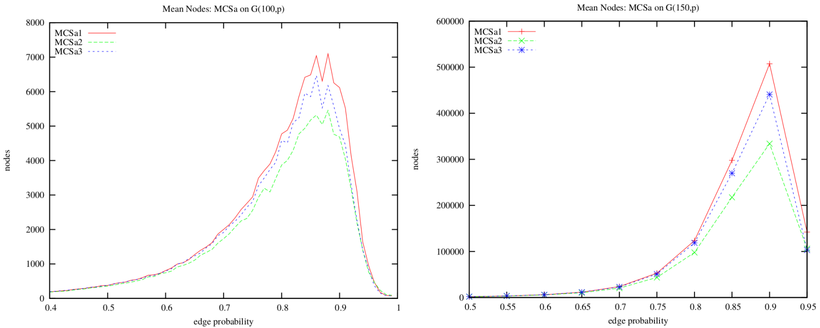
Intuition

- Low density means no occurrences, and we can quickly show we run out of edges after doing a bit of branching.
- High density means lots of occurrences, so wherever we look, it's easy to find one of them.
- If we expect there to be just one solution, it's really hard to find it if it exists, and really hard to rule it out if it doesn't exist.

Intuition

- Low density means no occurrences, and we can quickly show we run out of edges after doing a bit of branching.
- High density means lots of occurrences, so wherever we look, it's easy to find one of them.
- If we expect there to be just one solution, it's really hard to find it if it exists, and really hard to rule it out if it doesn't exist.

Optimisation, an Incomplete Picture



Prosser, Algorithms 5(4) 2012. Both plots have 100 samples per density step. The left-hand plot seems to go up in density steps of 0.01, and the right-hand plot, 0.05.

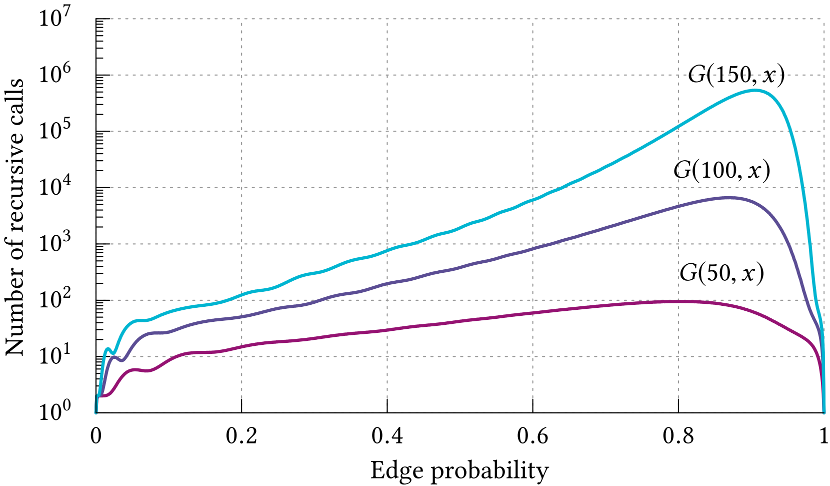
Which is the Hardest Density?

- Which density is hardest, for the optimisation problem?
- Does this change depending upon the number of vertices? The algorithm used? The random graph model selected?
- Is this the same as the hardest density for the decision problem, if we can also pick the decision number? And if so, which decision number do we pick?

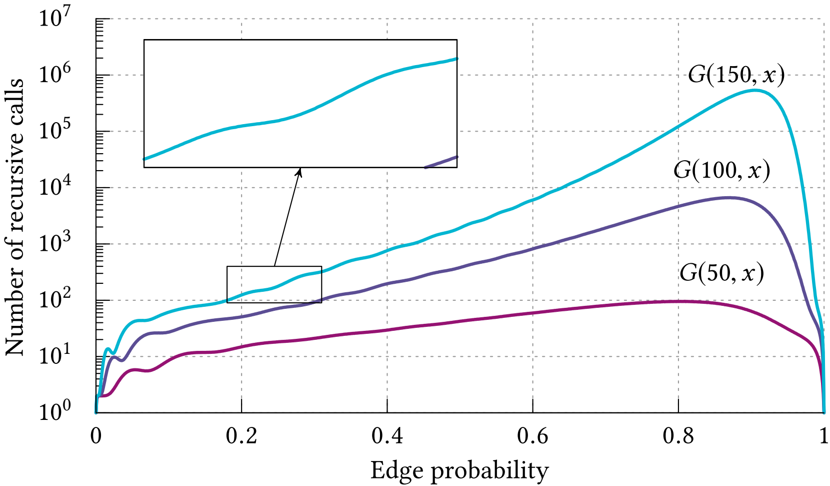
Really Big Experiments

- Increase density from 0 to 1 in steps of 0.001? This is around one pixel per step.
- Mean runtimes seem to settle down at around 10,000 samples. We probably want 100,000 samples to be safe.
- Back of the envelope feasibility estimates: 18 years.
- Conveniently, this is around 150,000 core hours.
  - ...And the rest of this work was a bit below 1,000,000 core hours.

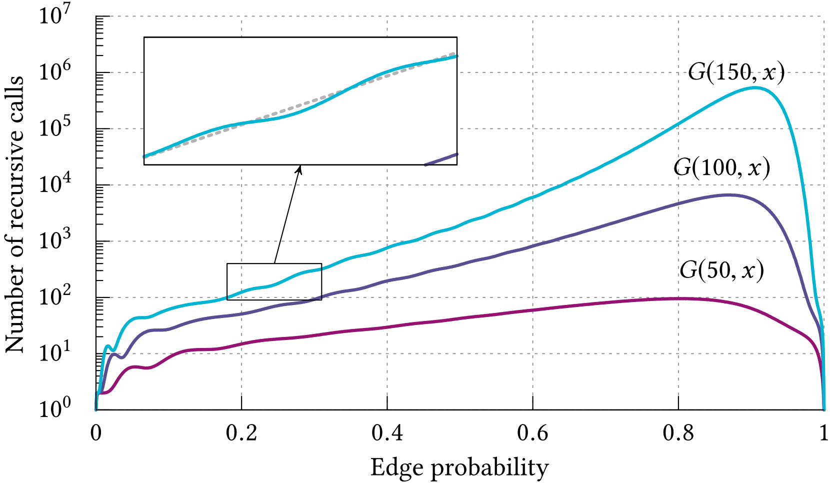
Optimisation, Refined



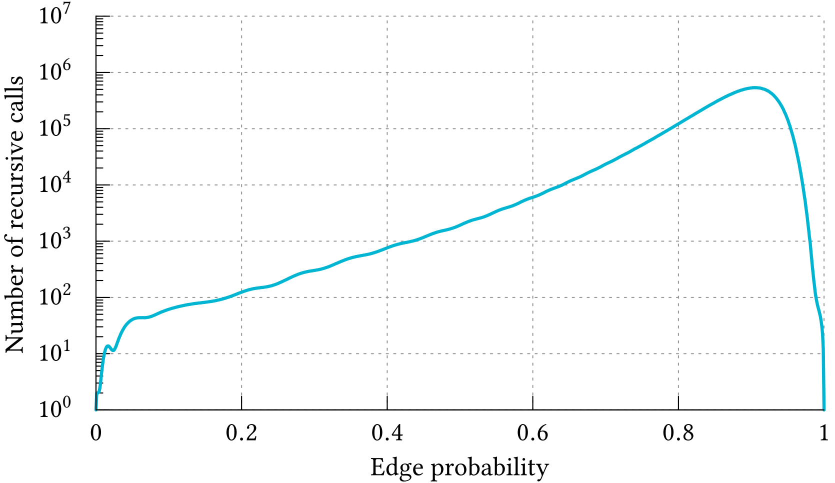
Optimisation, Refined



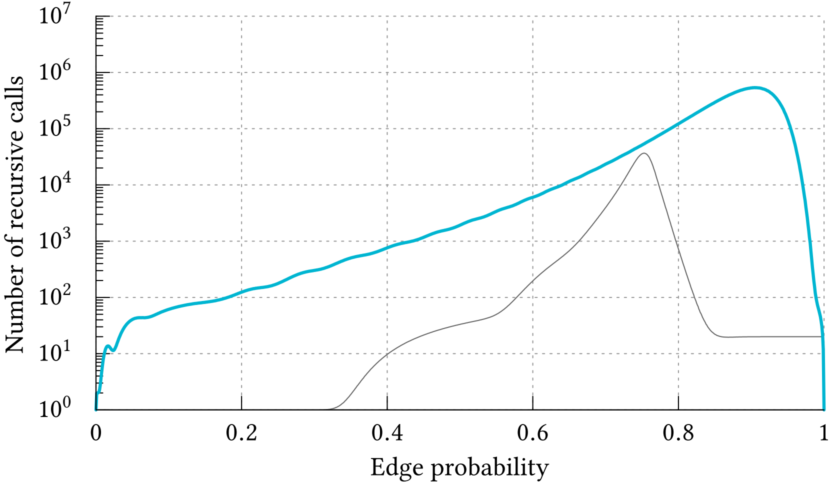
Optimisation, Refined



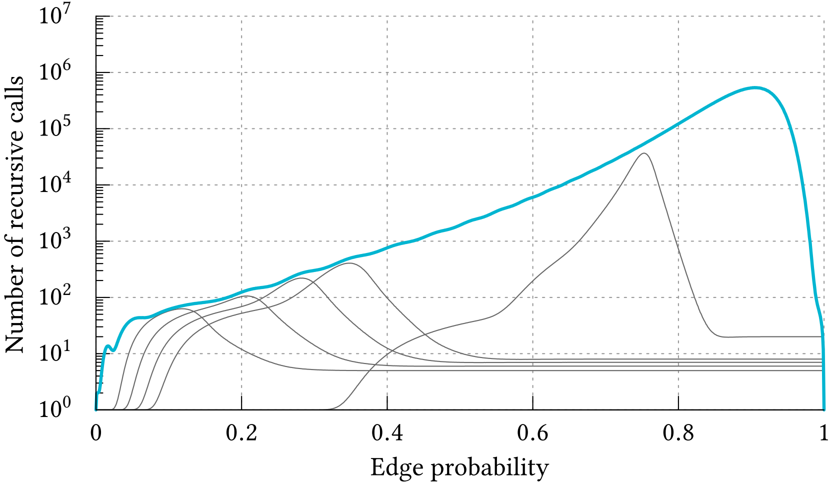
Optimisation versus Decision



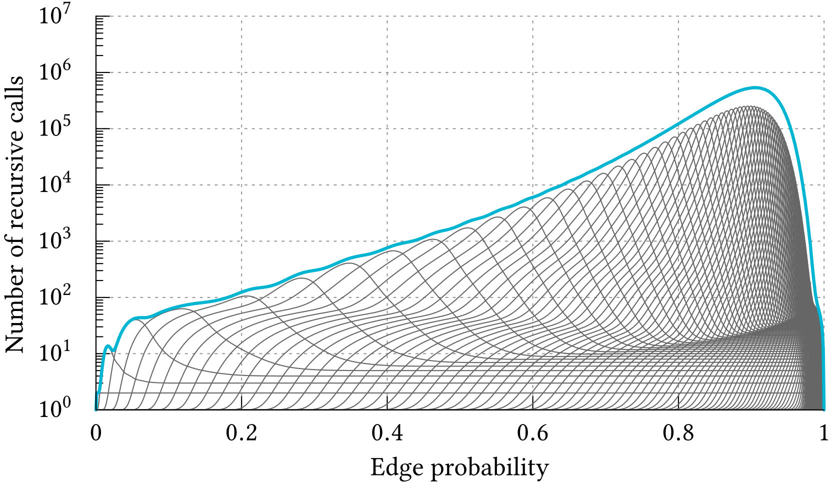
Optimisation versus Decision



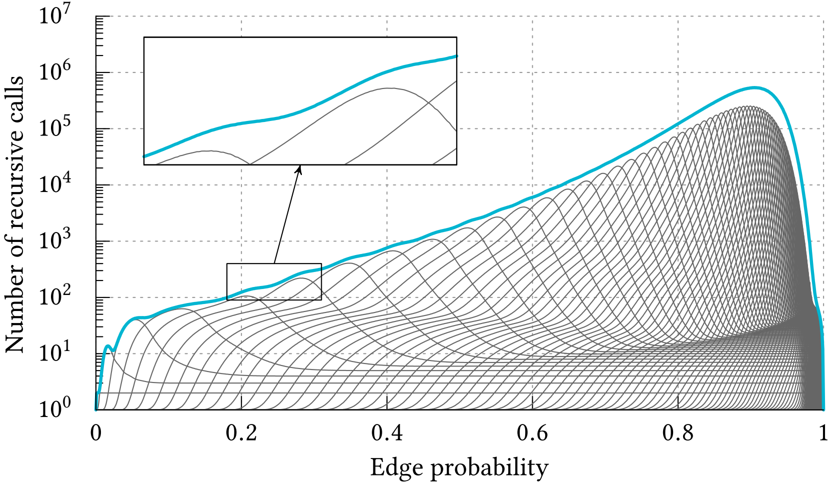
Optimisation versus Decision



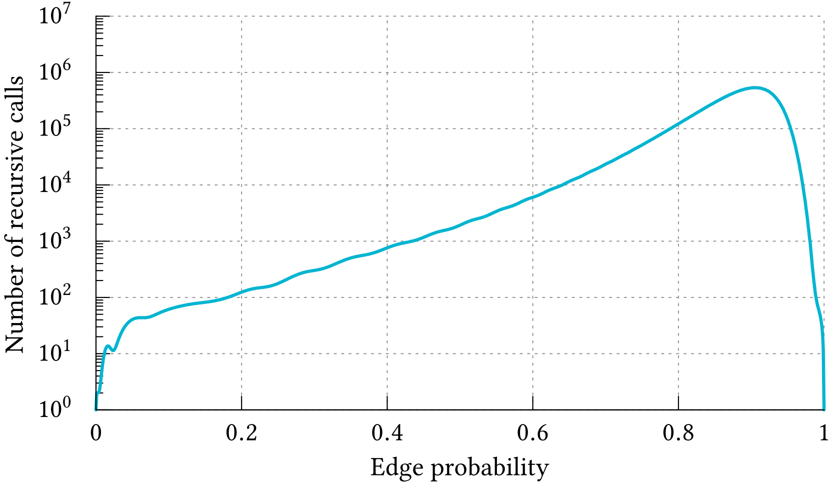
Optimisation versus Decision



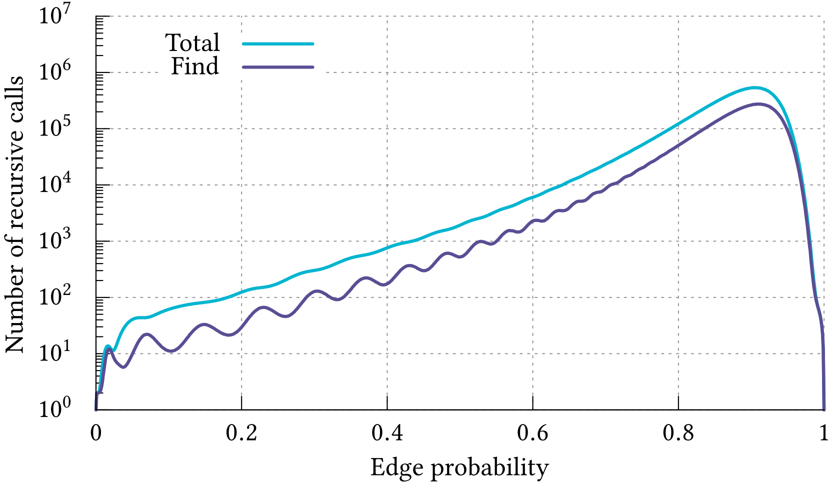
Optimisation versus Decision



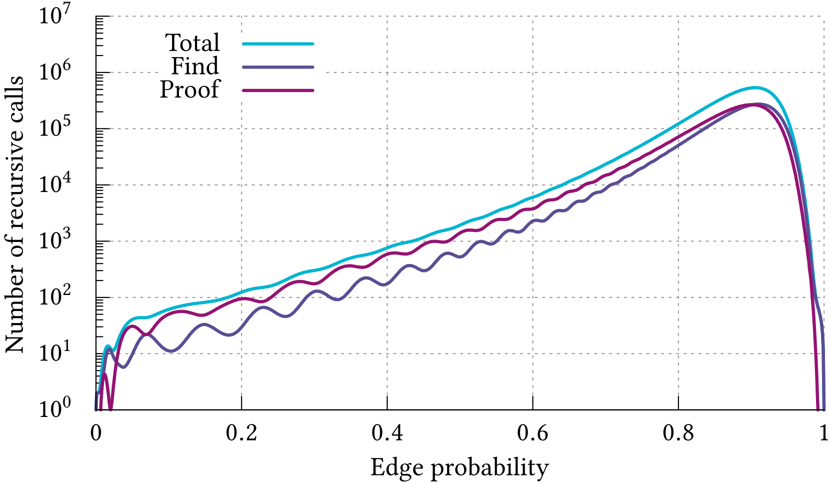
Finding versus Proving Optimality



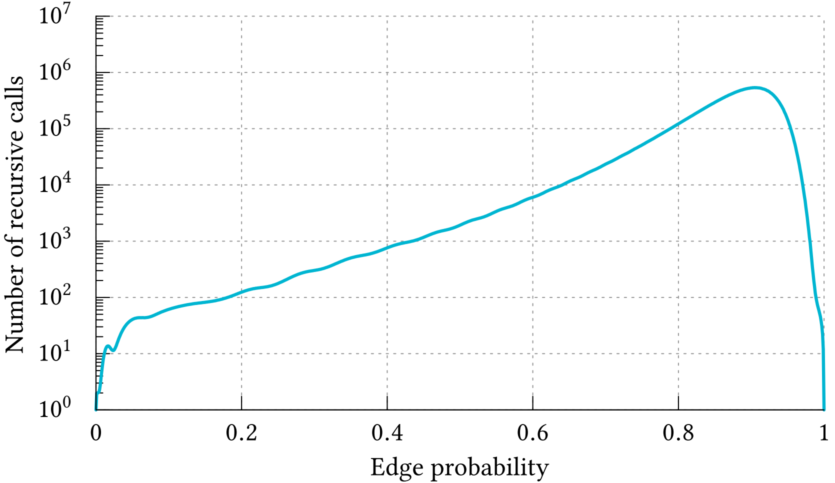
Finding versus Proving Optimality



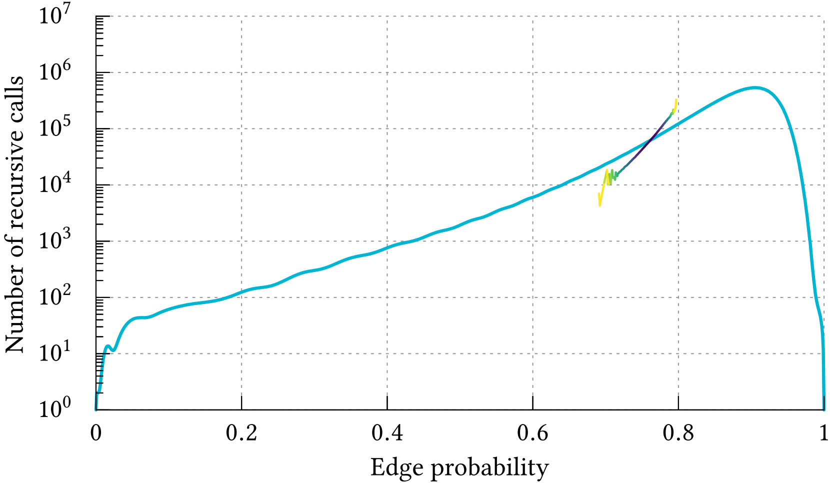
Finding versus Proving Optimality



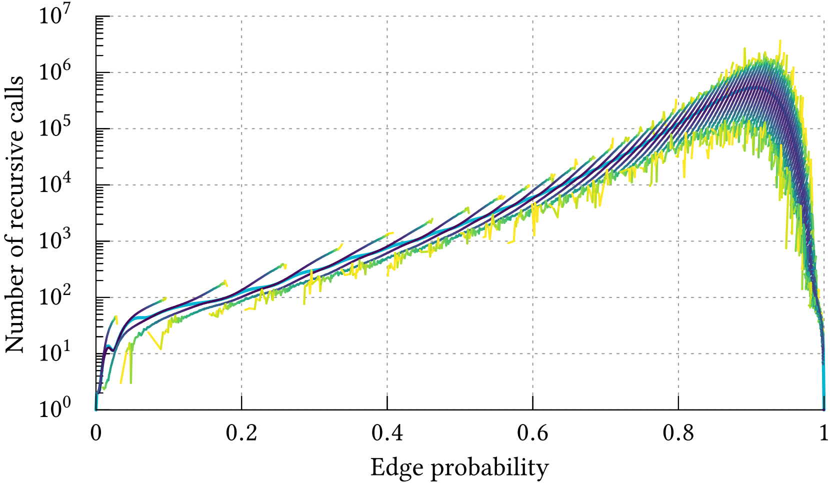
Difficulty by Solution Size



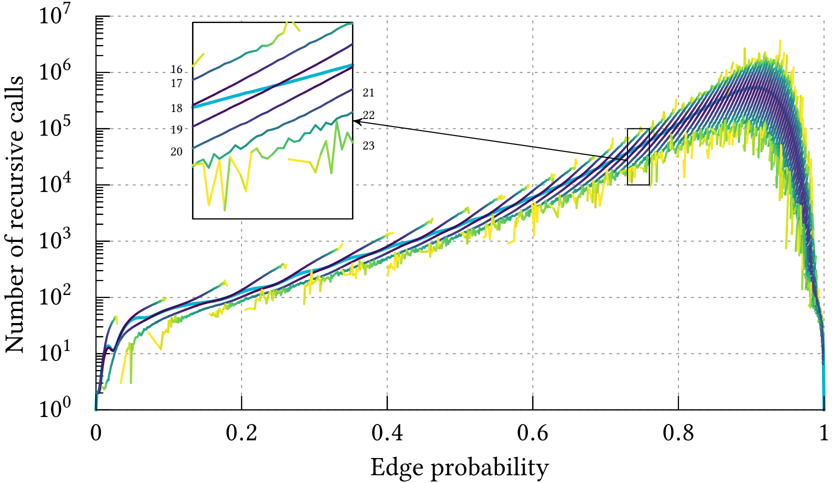
Difficulty by Solution Size



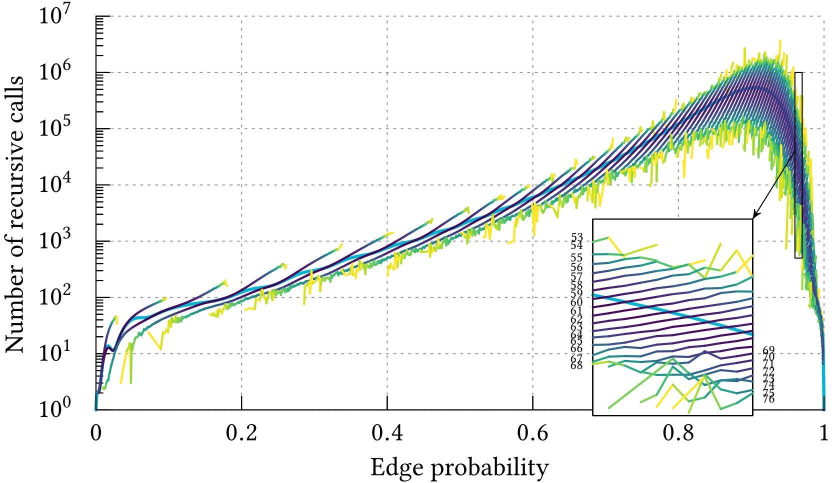
Difficulty by Solution Size



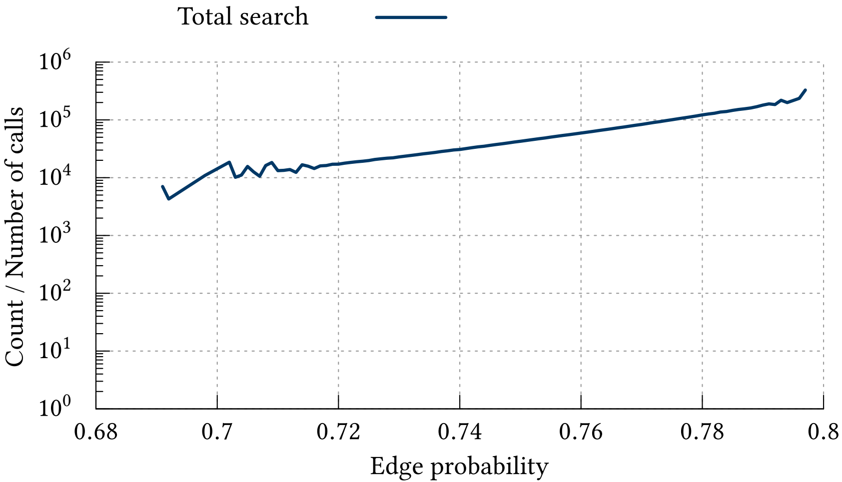
Difficulty by Solution Size



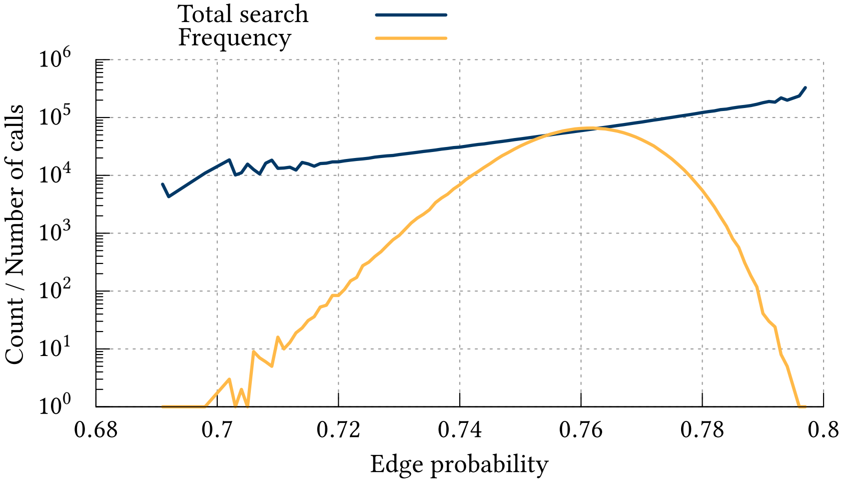
Difficulty by Solution Size



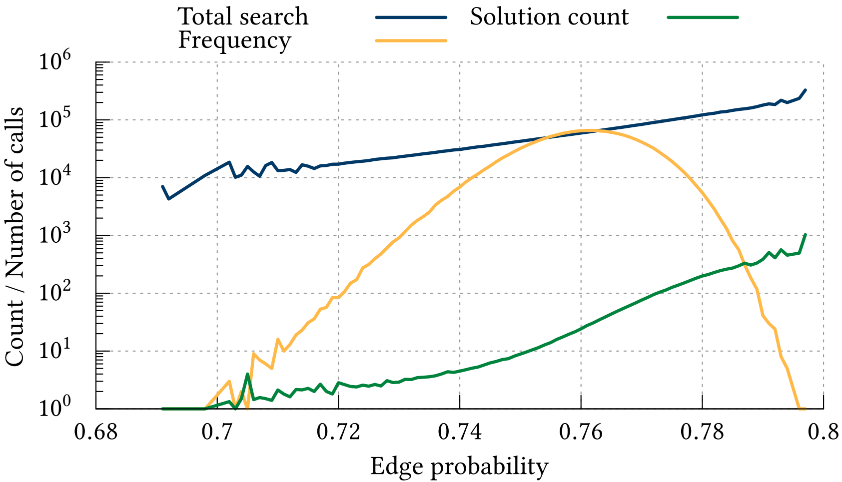
Difficulty by Optimal Solution Frequency



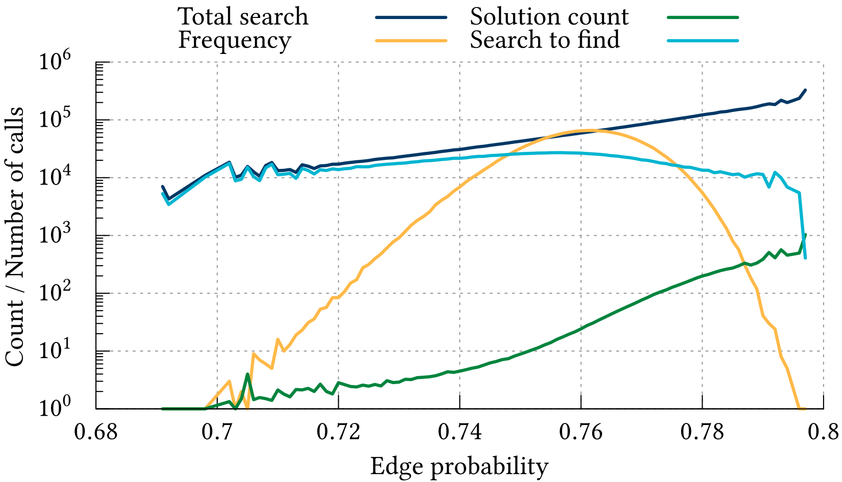
Difficulty by Optimal Solution Frequency



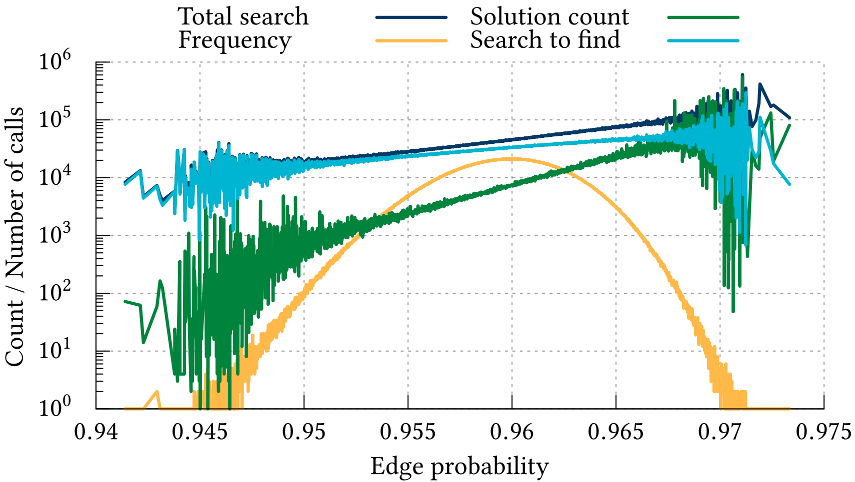
Difficulty by Optimal Solution Frequency



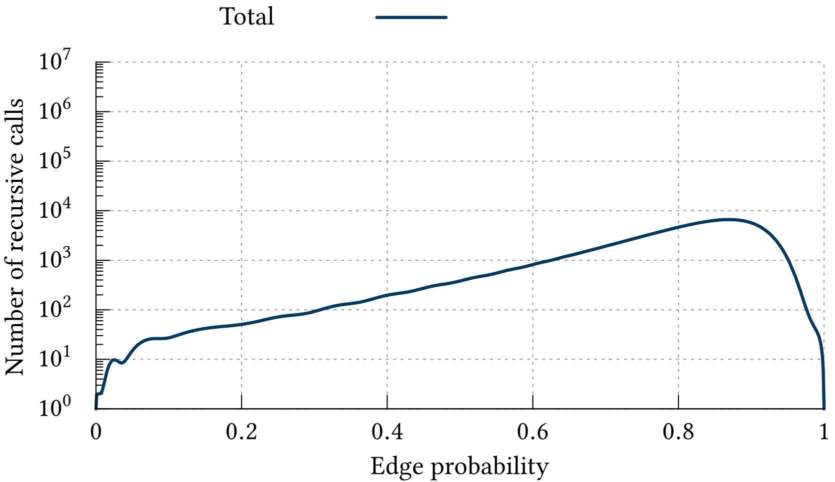
Difficulty by Optimal Solution Frequency



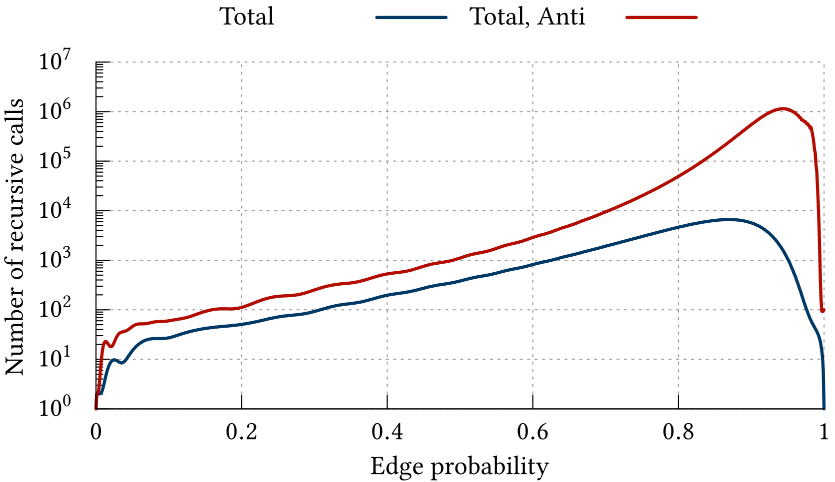
Difficulty by Optimal Solution Frequency



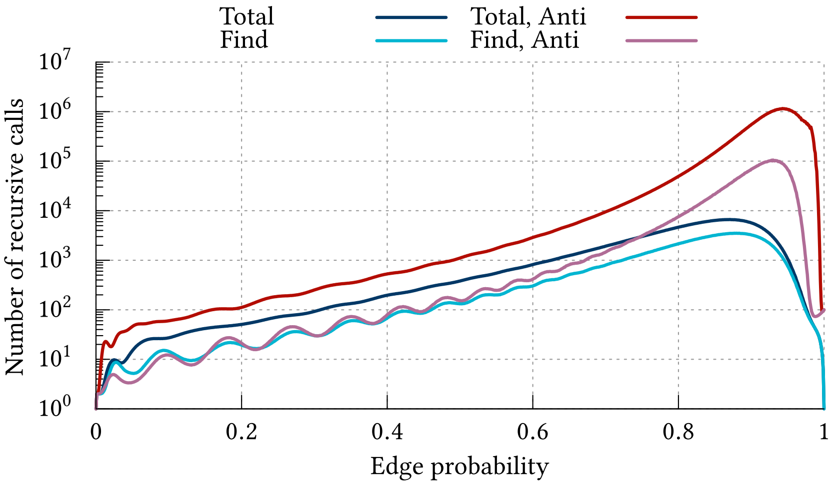
Search Order



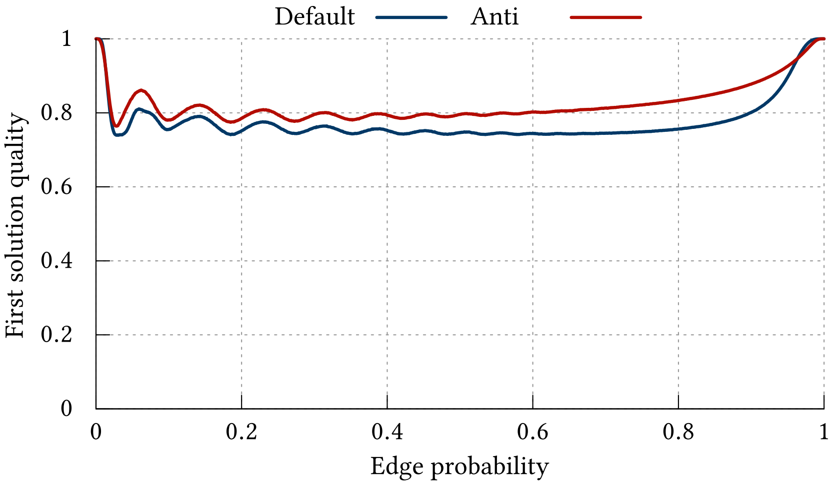
Search Order



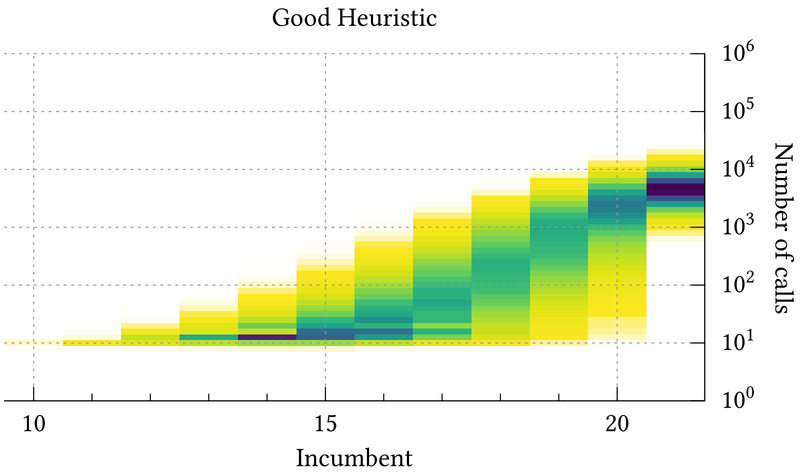
Search Order



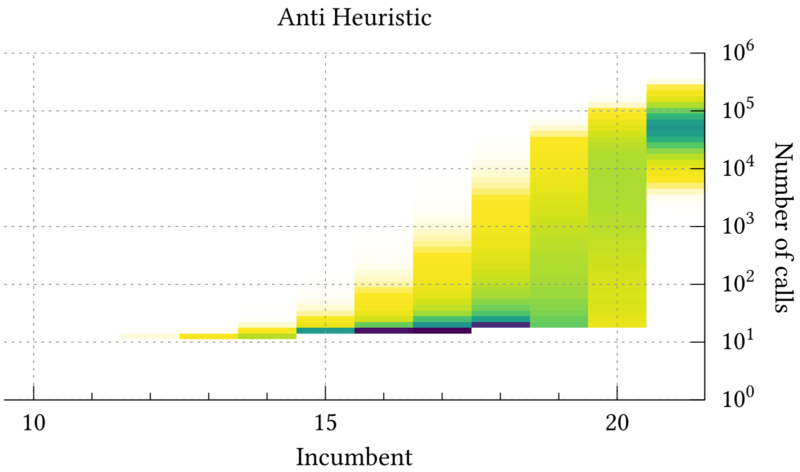
First Solution Quality



Solution Quality over Time



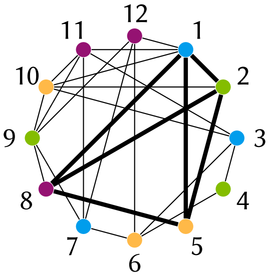
Solution Quality over Time



Open Questions

- Other solvers?
  - Trying to use other people's solvers for this work is a large part of why I now believe proof logging needs to become standard for algorithm implementation papers...
- What do MaxSAT, pseudo-Boolean, and MIP solvers do on these instances?
- Other models of randomness?

Colour Class Ordering



Vertices in colour order

1	3	7	2	4	9	5	6	10	8	11	12
1	1	1	2	2	2	3	3	3	4	4	4

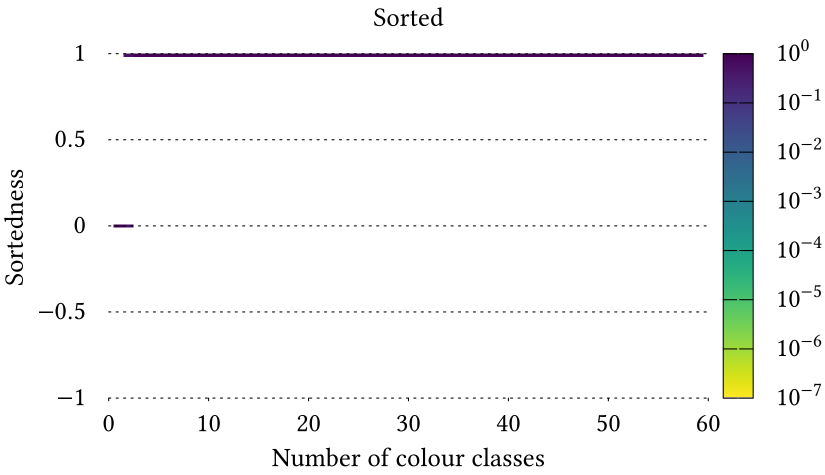
Number of colours used

Smallest Domain First?

- Branching on a colour class is like branching on a domain in CP, where the values are the vertices in the colour class plus a null value.
- Smallest domain first is a good heuristic.
- Greedy colourings tend to produce larger colour classes first.
- Right to left is smallest domain first.

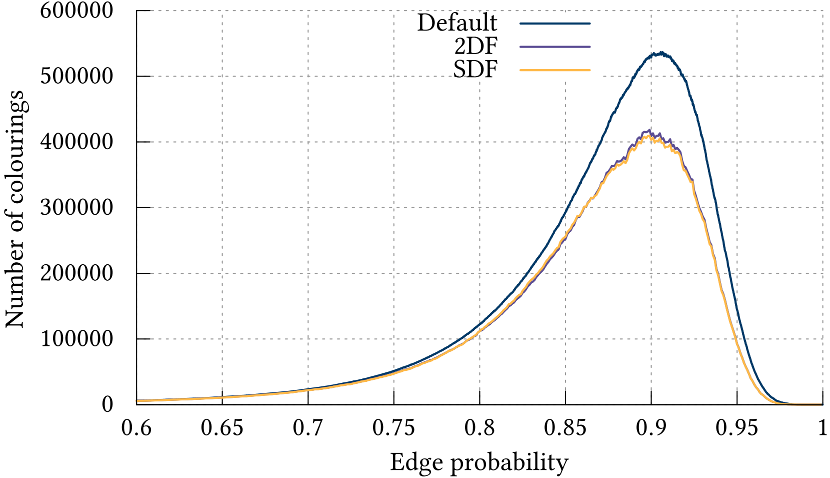
We Can Measure This!

We Can Measure This!



We Can Measure This!

Increasing Sortedness Decreases Search Space Size



However...

- Small impact on runtime.
- Hard to sell "understanding why this algorithm works" compared to "this new algorithm is better".
- John N. Hooker: "Testing heuristics: We have it all wrong". J. Heuristics 1(1) 1995.

Certifying Solvers

- Alongside an answer, a certifying solver produces an easily verifiable *certificate*.
- Provides a way of guaranteeing an *output* is correct.
- Certificate checking can be done independently, using a simple solver-independent tool.
- Doesn't guarantee that a solver is correct, just that if it ever produces an incorrect answer then it will be detected.
    - Even if due to hardware or compiler errors.

Pseudo-Boolean Proof Logging

- In the SAT community: CNF formulae, proofs in RUP, DRAT, LRAT, GRIT, ...
    - But not if the SAT solver does cardinality reasoning.
- Here: pseudo-Boolean formulae, proofs using reverse unit propagation and cutting planes derivations.

In Action...

```
$ ./glasgow_clique_solver p_hat500-2.clq
nodes = 108217
clique = 37 59 63 68 71 102 124 133 137 150 160 186 206 222 231 238
runtime = 175ms
```

In Action...

```
$ ./glasgow_clique_solver p_hat500-2.clq
nodes = 108217
clique = 37 59 63 68 71 102 124 133 137 150 160 186 206 222 231 238
runtime = 175ms

$ ./glasgow_clique_solver p_hat500-2.clq --prove proof
runtime = 16,347ms
```

In Action...

```
$ ./glasgow_clique_solver p_hat500-2.clq
nodes = 108217
clique = 37 59 63 68 71 102 124 133 137 150 160 186 206 222 231 238
runtime = 175ms

$ ./glasgow_clique_solver p_hat500-2.clq --prove proof
runtime = 16,347ms

$ ls -lh proof.log proof.opb
-rw-rw-r-- 1 ciaranm ciaranm 558M Aug 23 21:43 proof.log
-rw-rw-r-- 1 ciaranm ciaranm 1.4M Aug 23 21:42 proof.opb
```

In Action...

```
$ ./glasgow_clique_solver p_hat500-2.clq
nodes = 108217
clique = 37 59 63 68 71 102 124 133 137 150 160 186 206 222 231 238
runtime = 175ms

$ ./glasgow_clique_solver p_hat500-2.clq --prove proof
runtime = 16,347ms

$ ls -lh proof.log proof.opb
-rw-rw-r-- 1 ciaranm ciaranm 558M Aug 23 21:43 proof.log
-rw-rw-r-- 1 ciaranm ciaranm 1.4M Aug 23 21:42 proof.opb

$ veripb proof.opb proof.log
INFO:root:total time: 428.89s
maximal used database memory: 0.003 GB
Verification succeeded.
```

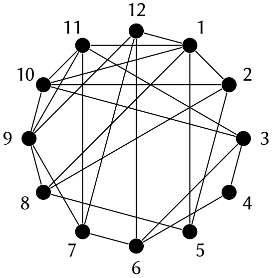
Observations

- The techniques we give are general, and not limited to one specific solver.
- Implementation effort is small, and can even speed up development.
- With the right proof logging format:
  - It's natural to express combinatorial and graph arguments (even if the proof format doesn't know what a graph is).
  - Proofs are "the same length as" the amount of work done by the solver.
  - Reformulations can be proof logged.
- It's time for competition organisers to start requiring proof logging support.

The Certifying Process

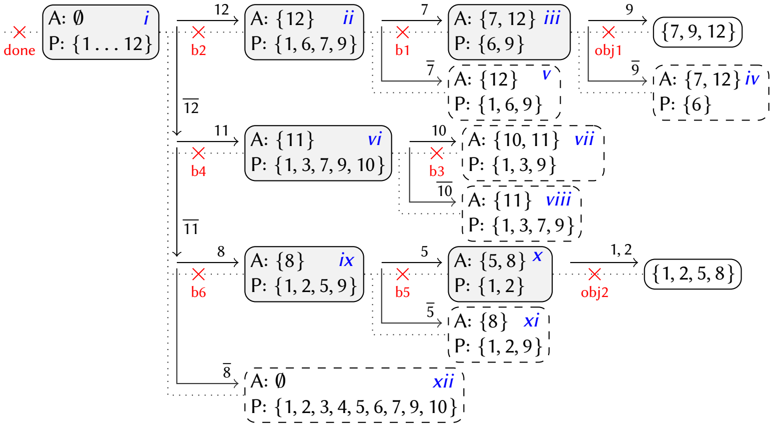
- Express the problem in pseudo-Boolean form (0/1 integer linear program; a superset of CNF):
  - A set of $\{0, 1\}$-valued variables x_i.
  - We define $\overline{x}_i = 1 - x_i$.
  - Integer linear inequalities $\sum_i c_i x_i \geq C$.
  - Optionally, an objective min $\sum_i c_i x_i$.
- Write this out as an OPB file.
- Provide a proof log for this OPB file.
  - For unsat decision instances, prove $0 \geq 1$.
  - Can also log sat decision instances, enumeration, and optimisation.
- Feed the OPB file and the proof log to VeriPB.

A Pseudo-Boolean Encoding




```
* #variable= 12 #constraint= 41
min: -1 x1 -1 x2 -1 x3 -1 x4 ...and so on... -1 x11 -1 x12 ;
1 ~x3 1 ~x1 >= 1 ;
1 ~x3 1 ~x2 >= 1 ;
1 ~x4 1 ~x1 >= 1 ;
* ...and a further 38 similar lines for the remaining non-edges
```

A Search Tree



A Proof Describing This Search Tree

```
pseudo-Boolean proof version 1.0
f 41 0
o x7 x9 x12
u 1 ~x12 1 ~x7 >= 1 ;
u 1 ~x12 >= 1 ;
u 1 ~x11 1 ~x10 >= 1 ;
u 1 ~x11 >= 1 ;
o x1 x2 x5 x8
u 1 ~x8 1 ~x5 >= 1 ;
u 1 ~x8 >= 1 ;
u >= 1 ;
c done 0
```

⇝ done

A Proof Describing This Search Tree

```
pseudo-Boolean proof version 1.0
f 41 0
o x7 x9 x12
u 1 ~x12 1 ~x7 >= 1 ;
u 1 ~x12 >= 1 ;
u 1 ~x11 1 ~x10 >= 1 ;
u 1 ~x11 >= 1 ;
o x1 x2 x5 x8
u 1 ~x8 1 ~x5 >= 1 ;
u 1 ~x8 >= 1 ;
u >= 1 ;
c done 0
```

⇝ done

A Proof Describing This Search Tree

```
pseudo-Boolean proof version 1.0
f 41 0
o x7 x9 x12
u 1 ~x12 1 ~x7 >= 1 ;
u 1 ~x12 >= 1 ;
u 1 ~x11 1 ~x10 >= 1 ;
u 1 ~x11 >= 1 ;
o x1 x2 x5 x8
u 1 ~x8 1 ~x5 >= 1 ;
u 1 ~x8 >= 1 ;
u >= 1 ;
c done 0
```

⇝ done

A Proof Describing This Search Tree

```
pseudo-Boolean proof version 1.0
f 41 0
o x7 x9 x12
u 1 ~x12 1 ~x7 >= 1 ;
u 1 ~x12 >= 1 ;
u 1 ~x11 1 ~x10 >= 1 ;
u 1 ~x11 >= 1 ;
o x1 x2 x5 x8
u 1 ~x8 1 ~x5 >= 1 ;
u 1 ~x8 >= 1 ;
u >= 1 ;
c done 0
```

⇝ done

A Proof Describing This Search Tree

```
pseudo-Boolean proof version 1.0
f 41 0
o x7 x9 x12
u 1 ~x12 1 ~x7 >= 1 ;
u 1 ~x12 >= 1 ;
u 1 ~x11 1 ~x10 >= 1 ;
u 1 ~x11 >= 1 ;
o x1 x2 x5 x8
u 1 ~x8 1 ~x5 >= 1 ;
u 1 ~x8 >= 1 ;
u >= 1 ;
c done 0
```

⇝ done

A Proof Describing This Search Tree

```
pseudo-Boolean proof version 1.0
f 41 0
o x7 x9 x12
u 1 ~x12 1 ~x7 >= 1 ;
u 1 ~x12 >= 1 ;
u 1 ~x11 1 ~x10 >= 1 ;
u 1 ~x11 >= 1 ;
o x1 x2 x5 x8
u 1 ~x8 1 ~x5 >= 1 ;
u 1 ~x8 >= 1 ;
u >= 1 ;
c done 0
```

⇝ done

A Proof Describing This Search Tree

```
pseudo-Boolean proof version 1.0
f 41 0
o x7 x9 x12
u 1 ~x12 1 ~x7 >= 1 ;
u 1 ~x12 >= 1 ;
u 1 ~x11 1 ~x10 >= 1 ;
u 1 ~x11 >= 1 ;
o x1 x2 x5 x8
u 1 ~x8 1 ~x5 >= 1 ;
u 1 ~x8 >= 1 ;
u >= 1 ;
c done 0
```

⇝ done

A Proof Describing This Search Tree

```
pseudo-Boolean proof version 1.0
f 41 0
o x7 x9 x12
u 1 ~x12 1 ~x7 >= 1 ;
u 1 ~x12 >= 1 ;
u 1 ~x11 1 ~x10 >= 1 ;
u 1 ~x11 >= 1 ;
o x1 x2 x5 x8
u 1 ~x8 1 ~x5 >= 1 ;
u 1 ~x8 >= 1 ;
u >= 1 ;
c done 0
```

⇝ done

A Proof Describing This Search Tree

```
pseudo-Boolean proof version 1.0
f 41 0
o x7 x9 x12
u 1 ~x12 1 ~x7 >= 1 ;
u 1 ~x12 >= 1 ;
u 1 ~x11 1 ~x10 >= 1 ;
u 1 ~x11 >= 1 ;
o x1 x2 x5 x8
u 1 ~x8 1 ~x5 >= 1 ;
u 1 ~x8 >= 1 ;
u >= 1 ;
c done 0
```

⇝ done

A Proof Describing This Search Tree

```
pseudo-Boolean proof version 1.0
f 41 0
o x7 x9 x12
u 1 ~x12 1 ~x7 >= 1 ;
u 1 ~x12 >= 1 ;
u 1 ~x11 1 ~x10 >= 1 ;
u 1 ~x11 >= 1 ;
o x1 x2 x5 x8
u 1 ~x8 1 ~x5 >= 1 ;
u 1 ~x8 >= 1 ;
u >= 1 ;
c done 0
```

$\rightsquigarrow$ done

A Proof Describing This Search Tree

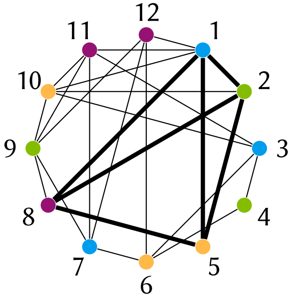
```
pseudo-Boolean proof version 1.0
f 41 0
o x7 x9 x12
u 1 ~x12 1 ~x7 >= 1 ;
u 1 ~x12 >= 1 ;
u 1 ~x11 1 ~x10 >= 1 ;
u 1 ~x11 >= 1 ;
o x1 x2 x5 x8
u 1 ~x8 1 ~x5 >= 1 ;
u 1 ~x8 >= 1 ;
u >= 1 ;
c done 0
```

⇝ done

Reverse Unit Propagation?

- Unit propagation is integer bounds consistency (the same as for SAT on clauses, but stronger on linear inequalities).
- Given the constraints we know so far C and a new constraint c, check that C combined with the negation of c leads to contradiction just through unit propagation.
- If so, we may add c as a new constraint.
- This is great for solver authors, because we don't have to explicitly justify adjacency reasoning.

Bound Functions



- Given a k-colouring of a subgraph, that subgraph cannot have a clique of more than k vertices.
  - Each colour class describes an at-most-one constraint.
- This does *not* follow from reverse unit propagation.

Cutting Planes Proofs

- We can add together two constraints to make a new constraint.
- We can multiply a constraint by a non-negative integer.
- We can divide a constraint by a positive integer, with rounding up.
- Using these steps, manually deriving at-most-one constraints for colour classes is easy to implement, and efficient.
- RUP can be written as cutting planes steps, but it's more work for solver authors.

What This Looks Like

```
pseudo-Boolean proof version 1.0
f 41 0
o x7 x9 x12
u 1 ~x12 1 ~x7 >= 1 ;
u 1 ~x12 >= 1 ;
* at most one [ x1 x3 x9 ]
p nonadj1_3 2 * nonadj1_9 + nonadj3_9 + 3 d          ⇝ tmp1
p obj1 tmp1 +
u 1 ~x11 1 ~x10 >= 1 ;                                ⇝ b3
* at-most-one [ x1 x3 x7 ]
p nonadj1_3 2 * nonadj1_7 + nonadj3_7 + 3 d          ⇝ tmp2
p obj1 tmp2 +
u 1 ~x11 >= 1 ;                                       ⇝ b4
o x1 x2 x5 x8                                         ⇝ obj2
u 1 ~x8 1 ~x5 >= 1 ;                                  ⇝ b5
p obj2 nonadj1_9 +
u 1 ~x8 >= 1 ;                                        ⇝ b6
* at-most-one [ x1 x3 x7 ] [ x2 x4 x9 ] [ x5 x6 x10 ]
p nonadj1_3 2 * nonadj1_7 + nonadj3_7 + 3 d          ⇝ tmp3
p obj2 tmp3 +
p nonadj2_4 2 * nonadj2_9 + nonadj4_9 + 3 d          ⇝ tmp4
p obj2 tmp3 + tmp4 +
p nonadj5_6 2 * nonadj5_10 + nonadj6_10 + 3 d        ⇝ tmp5
p obj2 tmp3 + tmp4 + tmp5 +
u >= 1 ;                                              ⇝ done
c done 0
```

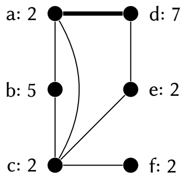
Results

- Implemented in the Glasgow Subgraph Solver.
  - Bit-parallel, can perform a colouring and recursive call in under a microsecond.
- 59 of the 80 DIMACS instances take under 1,000 seconds to solve without logging.
- Produced and verified proofs for 57 of these 59 instances (the other two reached 1TByte disk space).
- Mean slowdown from proof logging is 80.1 (due to disk I/O).
- Mean verification slowdown a further 10.1.
- Approximate implementation effort: one Masters student.
- Once you've done one solver, the rest are easy.

Maximal Clique Enumeration

- There are contradictory results for several graphs in the literature...
- For proof logging:
  - Maximality property is easily expressed in PB ("either take v, or at least one of v's neighbours").
  - Proof log every backtrack and every solution.
  - No need to proof log the "not set".
- This works for *all* maximal clique algorithms.
- Implementation effort: roughly one day for someone who had never implemented any kind of proof logging before.
- Works for standard benchmark graphs of up to 10,000 vertices.

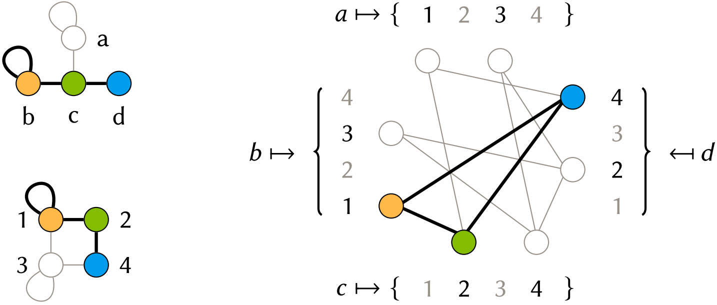
Maximum Weight Clique




```
pseudo-Boolean proof version 1.0
f 8 0
o xa xd                                                      ⇝ obj
p nonadja_e 2 * nonadja_f + nonadje_f + 3 d 2 *              ⇝ cc1
p nonadjb_d 5 *                                              ⇝ cc2
p nonadjc_d 2 *                                              ⇝ cc3
p obj cc1 + cc2 + cc3 +                                      ⇝ done
c done 0
```

- Colour classes have weights.
  - Just multiply a colour class by its weight.
- Vertices can split their weights between colour classes.
  - That's fine, no changes needed.
- Implementation effort: an afternoon, having seen roughly how it's done for unweighted cliques.

Maximum Common Subgraph via Clique



- We can encode this reduction using cutting planes rules. No need for a different OPB file.
- The clique solver does not need to be modified.
- This even works for connectivity.

Results

- Implemented alongside the algorithm in under a day.
- 11,400 instances verified, proof logging slowdown of 28.6 and 39.7.
- Verification slowdown of 11.3 and 73.1.
- Caught a bug in the implementation that testing had missed.

Implementation Effort

- Cheap to implement.
- Can potentially speed up development.
- With the right proof logging format, proofs are easy to write, but still simple.
  - No need to be aware of every single bound function or propagator.
  - Proofs can still be “efficient”.

Proof Logging as Standard?

- Lots of buggy solvers.
  - Culture of "my solver is faster on these benchmark instances!", and testing only on benchmark instances.
  - Particularly annoying because solvers are re-used for other problems.
- Proof-logging is too slow to require it to be on all the time.
  - Graph solvers can have much faster and stronger propagation than SAT solvers.
- Usable in practice for medium-sized instances.
- Allows for reformulation.

https://ciaranm.github.io/

ciaran.mccreesh@glasgow.ac.uk

University of Glasgow