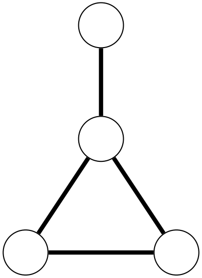
Finding Little Graphs Inside Big Graphs

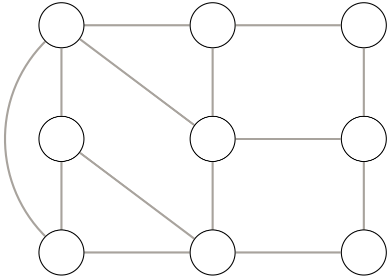
Ciaran McCreesh

University of Glasgow

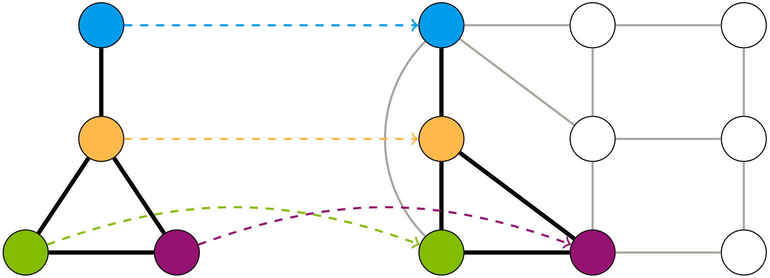
Royal Academy of Engineering

Subgraph Isomorphism

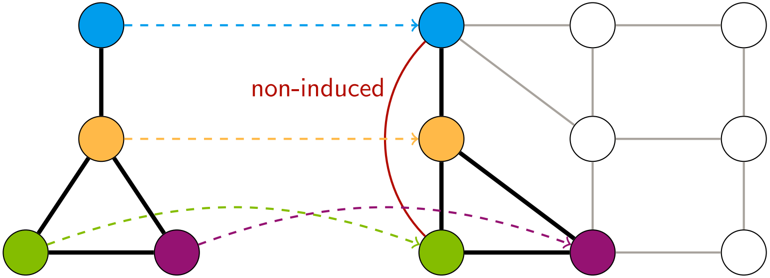




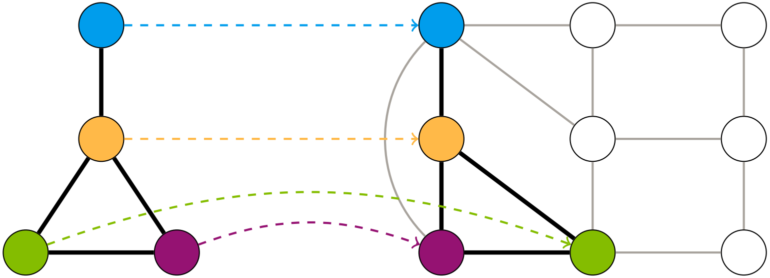
Subgraph Isomorphism



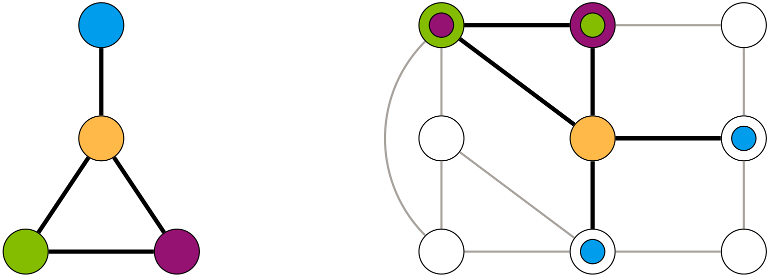
Subgraph Isomorphism



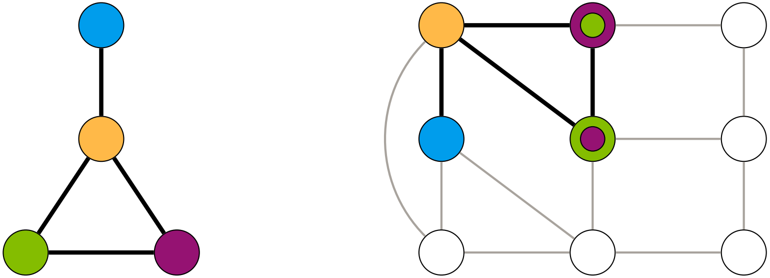
Subgraph Isomorphism



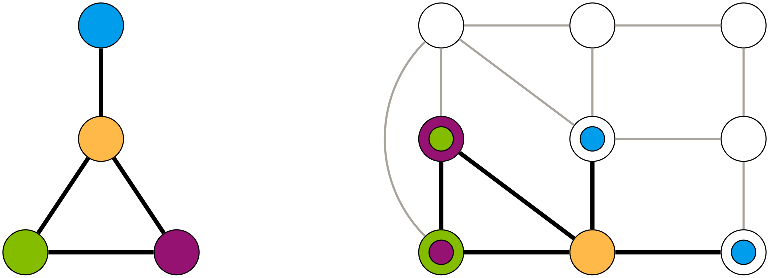
Subgraph Isomorphism



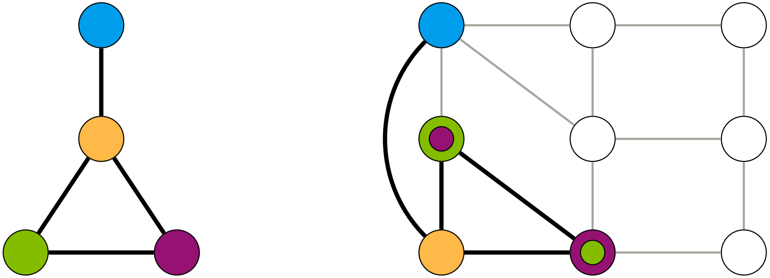
Subgraph Isomorphism



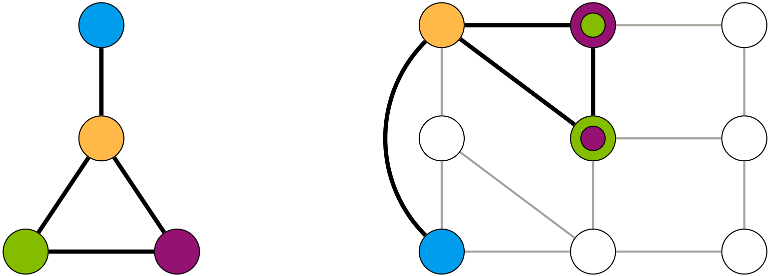
Subgraph Isomorphism



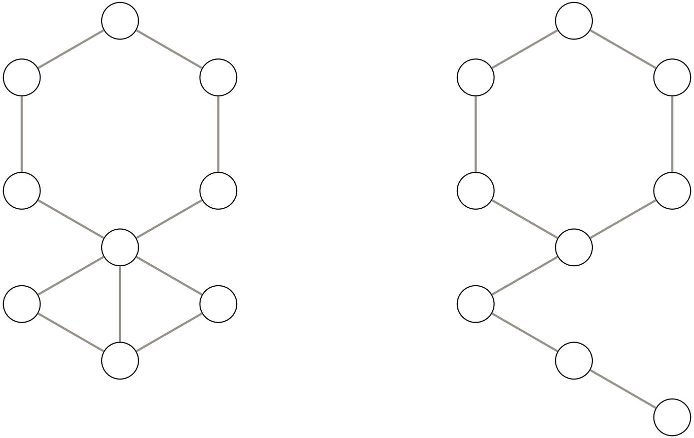
Subgraph Isomorphism



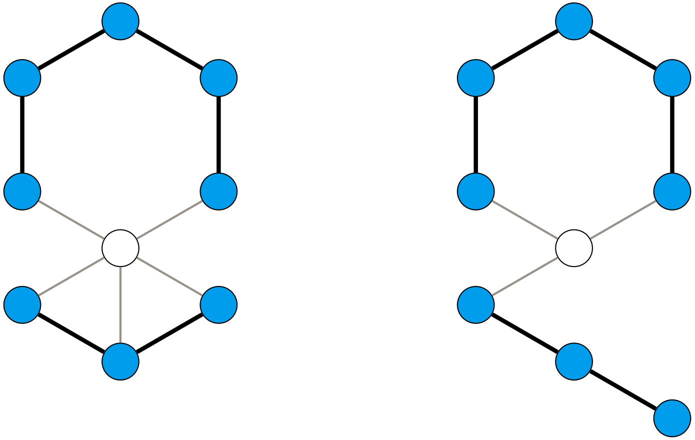
Subgraph Isomorphism



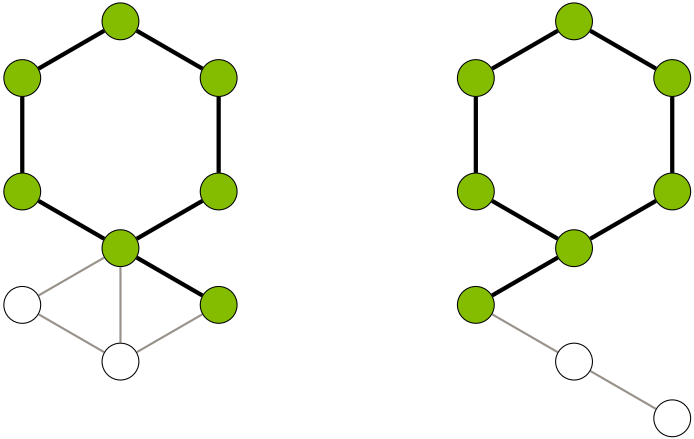
Maximum Common Induced Subgraph



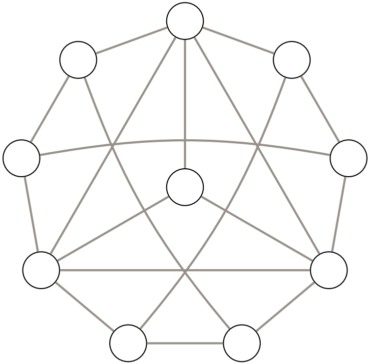
Maximum Common Induced Subgraph



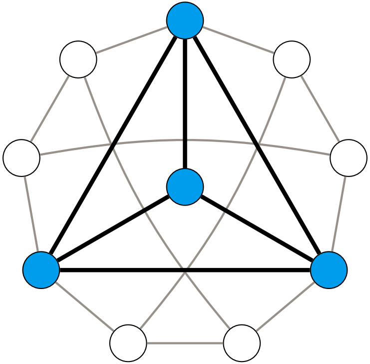
Maximum Common Induced Connected Subgraph



Maximum Clique



Maximum Clique



Who Cares?

- Chemistry, biochemistry, and drug design (graphs are molecule fragments or proteins).
- Computer vision.
- Compilers (instruction generation, code rewriting).
- Plagiarism and malware detection.
- Livestock epidemiology (contact and trade graphs).
- Designing mechanical lock systems.

In Theory. . .

- Subgraph finding is hard.
- Subgraph counting is hard.
- Approximate subgraph finding is hard.

In Practice. . .

- We have good *solvers* for subgraph problems.
- Some applications involve solving thousands of subgraph isomorphism queries per second.
- We can solve clique on larger graphs than we can solve all-pairs shortest path.[1]

[1] Terms and conditions apply.

In Practice. . .

- We have good *solvers* for subgraph problems.
- Some applications involve solving thousands of subgraph isomorphism queries per second.
- We can solve clique on larger graphs than we can solve all-pairs shortest path.[1]
- Maximum common subgraph is still a nightmare. . .

[1] Terms and conditions apply.

In Practice. . .

- We have good *solvers* for subgraph problems.
- Some applications involve solving thousands of subgraph isomorphism queries per second.
- We can solve clique on larger graphs than we can solve all-pairs shortest path.[1]
- Maximum common subgraph is still a nightmare. . .
- People often don't actually want to solve simple subgraph isomorphism.

[1] Terms and conditions apply.

Graphs Aren't Just Graphs

- Vertex and / or edge labels, or broader compatibility functions.
- Directed edges.
- Multi-edges, more than one edge between vertices.
- Hyper-edges, between more than two vertices.
- Partially defined graphs?
- No need for injectivity (homomorphism), or only local injectivity.

Graphs Aren't Just Graphs

- Vertex and / or edge labels, or broader compatibility functions.
- Directed edges.
- Multi-edges, more than one edge between vertices.
- Hyper-edges, between more than two vertices.
- Partially defined graphs?
- No need for injectivity (homomorphism), or only local injectivity.
- Don't forget about loops!

Graphs Aren't Just Graphs

- Vertex and / or edge labels, or broader compatibility functions.
- Directed edges.
- Multi-edges, more than one edge between vertices.
- Hyper-edges, between more than two vertices.
- Partially defined graphs?
- No need for injectivity (homomorphism), or only local injectivity.
- Don't forget about loops!
- Might want all solutions, or a count.

Two Solver Design Philosophies

1. Pick a vertex, guess where it goes, and start trying to grow a connected component.
   - Popular solvers: VF2, VF3, RI, TurboISO, . . .
   - Very fast to start up.
   - Often good on easy instances.
   - Spectacularly bad on hard instances, and on some easy instances.
2. Use constraint programming, build a mapping from the pattern graph to the target graph.
   - LAD, Glasgow Subgraph Solver.
   - Consistent performance on easy instances.
   - Much better on hard instances.

The Glasgow Subgraph Solver

`https://github.com/ciaranm/glasgow-subgraph-solver`

- A CP style solver specifically for subgraph algorithms.
- Subgraph isomorphism, and all its variants (induced / non-induced, homomorphism, locally injective, labels, side constraints, directed, . . .).
- Also special algorithms for clique.
- Guaranteed no bugs!

The Glasgow Subgraph Solver

https://github.com/ciaranm/glasgow-subgraph-solver

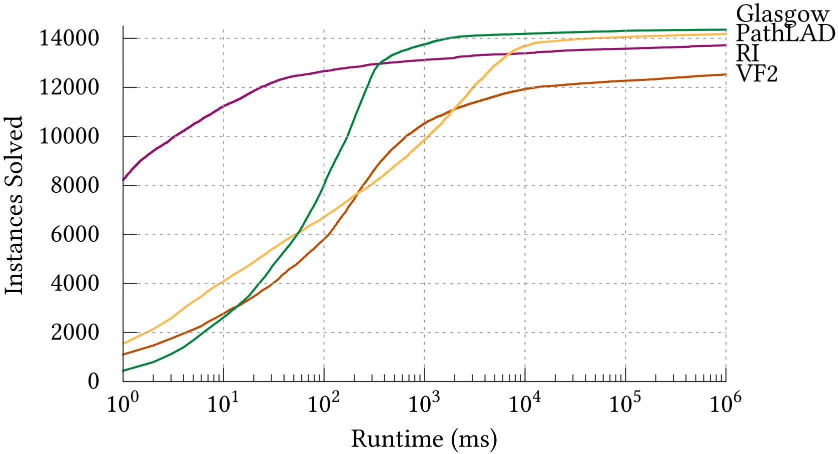
- A CP style solver specifically for subgraph algorithms.
- Subgraph isomorphism, and all its variants (induced / non-induced, homomorphism, locally injective, labels, side constraints, directed, . . .).
- Also special algorithms for clique.
- Guaranteed no bugs!
  - Or at least, any buggy output will always be detected, if you enable proof logging.

Benchmark Instances

http://perso.citi-lab.fr/csolnon/SIP.html

- 14,621 instances from Christine Solnon's collection:
  - Randomly generated with different models (MIVIA suite).
  - Real-world graphs.
  - Computer vision problems.
  - Biochemistry problems.
  - Phase transition instances.
- At least...
  - $\geq$ 2,110 satisfiable.
  - $\geq$ 12,322 unsatisfiable.
- A lot of them are very easy for good algorithms.

Is It Any Good?



Easy Conclusion!

- CP is best!

An Observation about Certain Datasets

- All of the randomly generated instances from the MIVIA suites are satisfiable.
- The target graphs are randomly generated, and patterns are made by selecting random connected subgraphs and permuting them.
- These instances are usually rather easy. . .
- Many papers use *only* these instances for benchmarking.

A Different Easy Conclusion!

- CP is slow! RI is best!

Constraint Programming

- We have some variables, each of which has a domain of possible values.
- Give each variable a value from its domain, whilst respecting all constraints.

Building a Mapping

- One variable per pattern vertex.
- Domains and values are target vertices.
- We think of these variables as defining a function.

Injectivity

- Can't map to the same target vertex twice.
- Could say that each pair of pattern vertices are not equal?

Injectivity

- Can't map to the same target vertex twice.
- Could say that each pair of pattern vertices are not equal?
- We prefer high-level constraints, so we just say "all different".

Adjacency

- If A and B are adjacent in the pattern, $f(A)$ must be adjacent to $f(B)$ in the target.
- Various ways of encoding this. In SAT we'd need n^4 clauses, or n^3 if we're sneaky.
- In practice: we write a special constraint propagator to do this efficiently.

Backtracking Search, Maintaining Consistency

- Pick a variable V that has more than one value remaining.
- For each of its values v in turn:
  - Try $V = v$, and do some inference.
    - No other variable can take the value v.
    - Variables adjacent to V must be given values adjacent to v.
  - If we get an empty domain, we made a bad guess.
  - If every variable has one value left, we have a solution.
  - Otherwise, recurse.

Data Structures

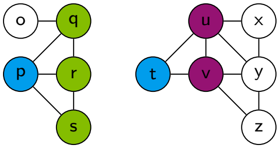
- We store a set of values for every variable.
- Need to be able to test whether a specific value is present, remove values, count how many values remain.
- Must either be copyable, or have some way of doing backtracking.

Data Structures

- We store a set of values for every variable.
- Need to be able to test whether a specific value is present, remove values, count how many values remain.
- Must either be copyable, or have some way of doing backtracking.
- Objectively correct answer: bitsets.

Degree Filtering

- Can't map a vertex of degree d to a vertex of degree less than d.



Neighbourhood Degree Sequences

- Can't map a vertex whose neighbours have degree 4, 3, 2 to a vertex whose neighbours have degree 4, 2, 2, 2.

Dynamic Degrees?

- If a target vertex disappears from every domain, can pretend it's not there at all.
- This reduces the degree of all of its neighbours.
- Maybe this leads to more filtering?

Dynamic Degrees?

- If a target vertex disappears from every domain, can pretend it's not there at all.
- This reduces the degree of all of its neighbours.
- Maybe this leads to more filtering?
- Problem: detecting this can be moderately expensive, so possibly not worth doing?

Adjacency Filtering

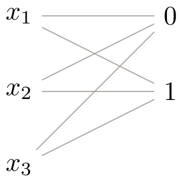
- When P gets mapped to t, neighbours of P can only be mapped to neighbours of t.
- Store domains and neighbourhoods as bitsets.

Injectivity Filtering

$$A \in \{1, 2\}$$
$$B \in \{2, 3\}$$
$$C \in \{1, 3\}$$
$$D \in \{1, 4, 5, 6\}$$
$$E \in \{2, 5\}$$
$$F \in \{3, 5\}$$

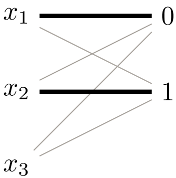
Matchings and All-Different

- Draw a vertex on the left for each variable, and a vertex on the right for each value.
- Draw edges from each variable to each of its values.
- A *maximum cardinality matching* is where you pick as many edges as possible, but each vertex can only be used at most once.
- We can find this in polynomial time.
- There is a matching which covers each variable if and only if the constraint can be satisfied.
- In fact, there is a one to one correspondence between perfect matchings and solutions to the constraint.



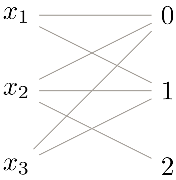
Matchings and All-Different

- Draw a vertex on the left for each variable, and a vertex on the right for each value.
- Draw edges from each variable to each of its values.
- A *maximum cardinality matching* is where you pick as many edges as possible, but each vertex can only be used at most once.
- We can find this in polynomial time.
- There is a matching which covers each variable if and only if the constraint can be satisfied.
- In fact, there is a one to one correspondence between perfect matchings and solutions to the constraint.



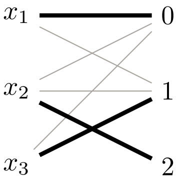
Matchings and All-Different

- Draw a vertex on the left for each variable, and a vertex on the right for each value.
- Draw edges from each variable to each of its values.
- A *maximum cardinality matching* is where you pick as many edges as possible, but each vertex can only be used at most once.
- We can find this in polynomial time.
- There is a matching which covers each variable if and only if the constraint can be satisfied.
- In fact, there is a one to one correspondence between perfect matchings and solutions to the constraint.



Matchings and All-Different

- Draw a vertex on the left for each variable, and a vertex on the right for each value.
- Draw edges from each variable to each of its values.
- A *maximum cardinality matching* is where you pick as many edges as possible, but each vertex can only be used at most once.
- We can find this in polynomial time.
- There is a matching which covers each variable if and only if the constraint can be satisfied.
- In fact, there is a one to one correspondence between perfect matchings and solutions to the constraint.



Sudoku

From Wikipedia, the free encyclopedia

Not to be confused with Sodoku or Sudeki.

Sudoku (数独 *sūdoku*?, digit-single) /suːˈdoʊkuː/, /-ˈdɒ-/, /sə-/; originally called **Number Place**,[1] is a logic-based,[2][3] combinatorial[4] number-placement puzzle. The objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 sub-grids that compose the grid (also called "boxes", "blocks", "regions", or "sub-squares") contains all of the digits from 1 to 9. The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a unique solution.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

A typical Sudoku puzzle

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

The same puzzle with solution numbers marked in red

How do Humans Solve Sudoku?

18	23	23	245	456	456	279	378	23589

How do Humans Solve Sudoku?

18	23	23	245	456	456	279	378	23589

How do Humans Solve Sudoku?

1	23	23	245	456	456	279	378	23589

How do Humans Solve Sudoku?

1	23	23	245	456	456	279	378	23589

How do Humans Solve Sudoku?

1	23	23	245	456	456	279	378	23589

How do Humans Solve Sudoku?

1	23	23	245	456	456	279	378	23589

How do Humans Solve Sudoku?

1	23	23	45	456	456	79	78	589

How do Humans Solve Sudoku?

1	23	23	45	456	456	79	78	589

How do Humans Solve Sudoku?

1	23	23	45	456	456	79	78	589

How do Humans Solve Sudoku?

1	23	23	45	456	456	79	78	89

Generalised Arc Consistency

- Arc Consistency (AC): for a binary constraint, each value is supported by at least one value in the other variable.
- Generalised Arc Consistency (GAC): for a global constraint, we can pick any value from any variable, and find a supporting set of values from each other variable in the constraint simultaneously.
  - Each remaining value appears in at least one solution to the constraint.

Hall Sets

- A *Hall set* of size n is a set of n variables from an "all different" constraint, whose domains have n values between them.
- If we can find a Hall set, we can safely remove these values from the domains of every other variable involved in the constraint.
- Hall's Marriage Theorem: doing this is equivalent to deleting every edge from the matching graph which cannot appear in any perfect matching.
- So, if we delete every Hall set, we delete every value that cannot appear in at least one way of satisfying the constraint. In other words, we obtain GAC.

GAC for All-Different

- There are 2^n potential Hall sets, so considering them all is probably a bad idea...
- Similarly, enumerating every perfect matching is #P-hard.
- However, there is a polynomial algorithm!

GAC for All-Different

18	23	23	245	456	456	279	378	23589

GAC for All-Different

18	23	23	245	456	456	279	378	23589

x_0

x_1

x_2

x_3

x_4

x_5

x_6

x_7

x_8

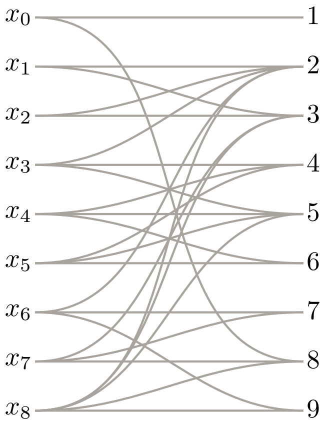
GAC for All-Different

18	23	23	245	456	456	279	378	23589

x_0	1
x_1	2
x_2	3
x_3	4
x_4	5
x_5	6
x_6	7
x_7	8
x_8	9

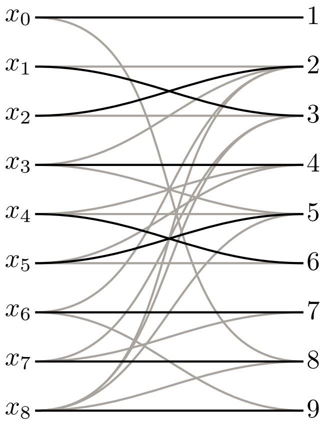
GAC for All-Different

18	23	23	245	456	456	279	378	23589



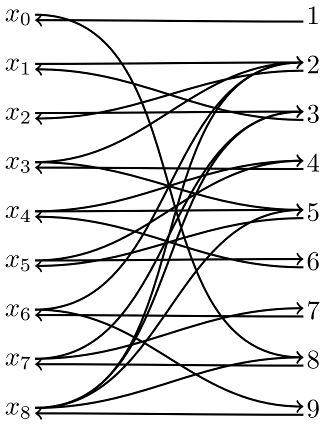
GAC for All-Different

18	23	23	245	456	456	279	378	23589



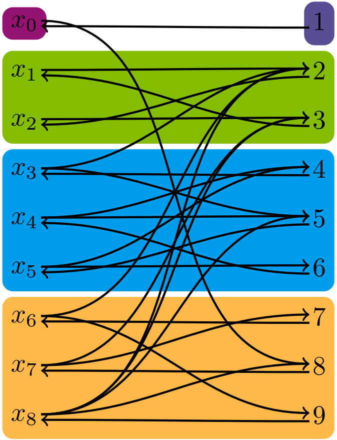
GAC for All-Different

18	23	23	245	456	456	279	378	23589



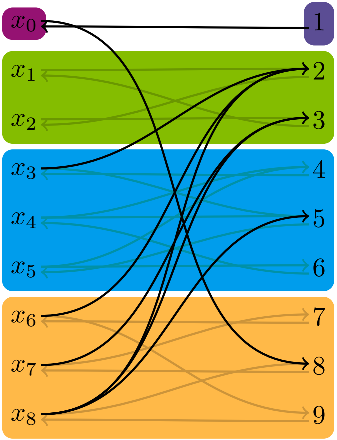
GAC for All-Different

18	23	23	245	456	456	279	378	23589



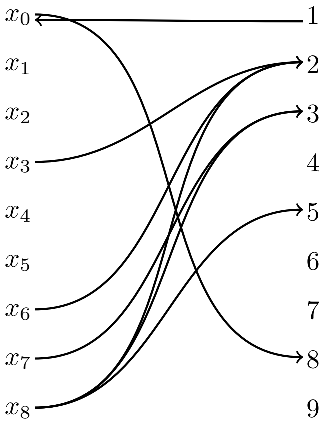
GAC for All-Different

18	23	23	245	456	456	279	378	23589



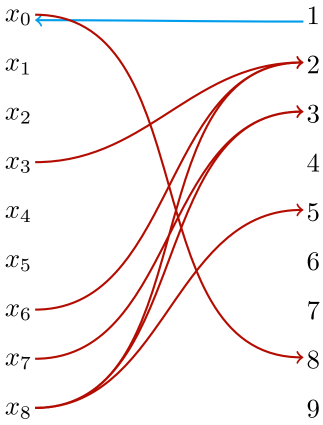
GAC for All-Different

18	23	23	245	456	456	279	378	23589



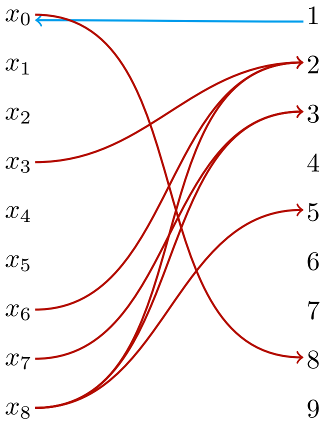
GAC for All-Different

18	23	23	245	456	456	279	378	23589



GAC for All-Different

18	23	23	245	456	456	279	378	23589



Is This a Good Idea?

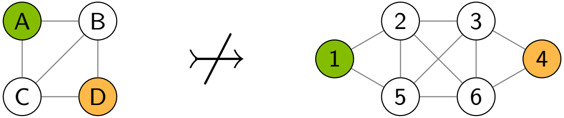
- Various techniques to avoid running all-different all of the time.
  - Faster bit-parallel propagator that can miss some deletions.
- Can also do all-different on edges. . .

Distance Filtering

- Adjacent vertices must be mapped to adjacent vertices.

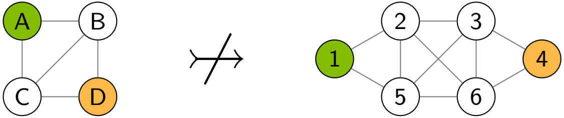
Distance Filtering

- Adjacent vertices must be mapped to adjacent vertices.
- Vertices that are distance 2 apart must be mapped to vertices that are within distance 2.



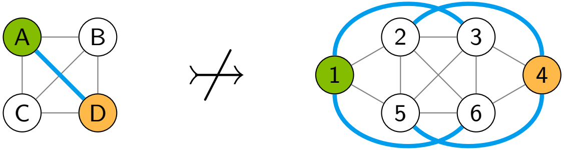
Distance Filtering

- Adjacent vertices must be mapped to adjacent vertices.
- Vertices that are distance 2 apart must be mapped to vertices that are within distance 2.
- Vertices that are distance k apart must be mapped to vertices that are within distance k.



Distance Filtering

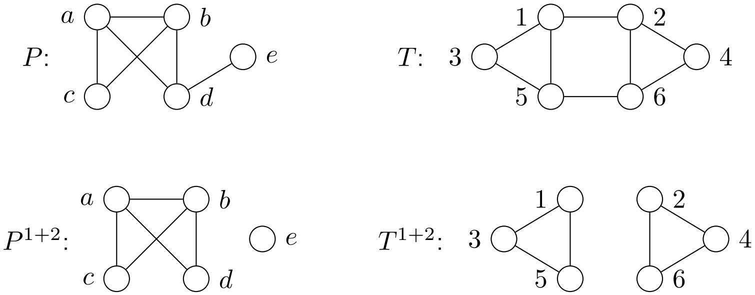
- G^d is the graph with the same vertex set as G, and an edge between v and w if the distance between v and w in G is at most d.
- For any d, a subgraph isomorphism $i : P \rightarrowtail T$ is also a subgraph isomorphism $i^d : P^d \rightarrowtail T^d$.



Distance Filtering

- We can do something stronger: rather than looking at distances, we can look at (simple) paths, and we can count how many there are.
- This is NP-hard in general, but only lengths 2 and 3 and counts of 2 and 3 are useful in practice.
- We construct these graph pairs once, at the top of search, and use them for degree-based filtering at the top of search, and “adjacency” filtering during search.

Supplemental Constraints



Induced Subisomorphisms

- Find something that is a non-induced subisomorphism

$$P \rightarrowtail T$$

and simultaneously a non-induced subisomorphism

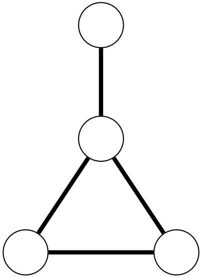
$$\overline{P} \rightarrowtail \overline{T}$$

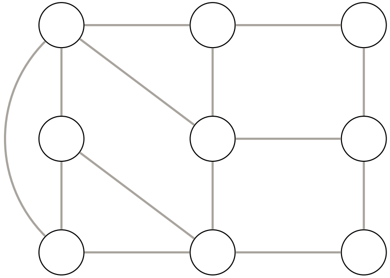
Partially Defined Graphs

- Challenge for you!

Clique Neighbourhood Filtering

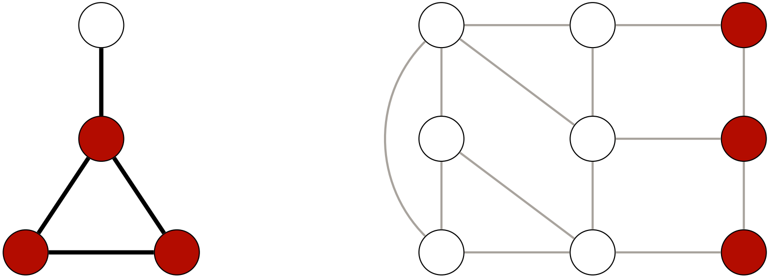
- If a pattern vertex is contained in a k-vertex clique, it must be mapped to a target vertex contained in at least a k-vertex clique.
- Valid without injectivity (with a caveat for loops).





Clique Neighbourhood Filtering

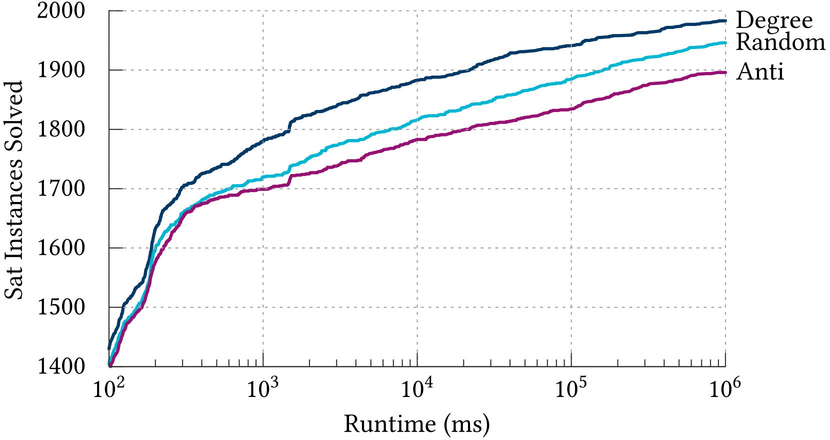
- If a pattern vertex is contained in a k-vertex clique, it must be mapped to a target vertex contained in at least a k-vertex clique.
- Valid without injectivity (with a caveat for loops).



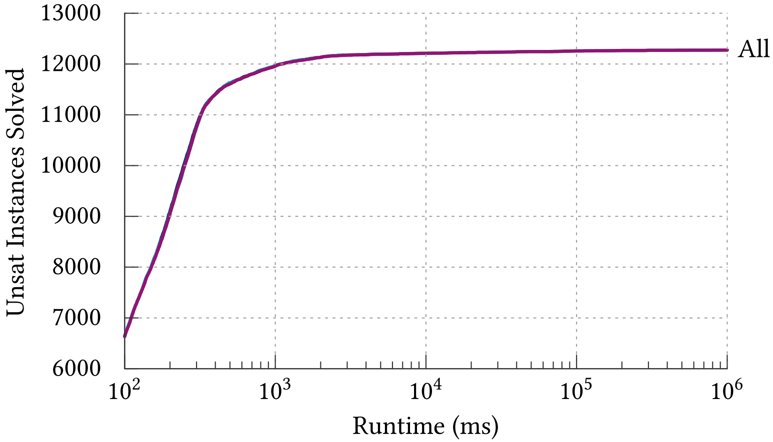
Variable and Value Ordering Heuristics

- Variable ordering (i.e. pattern vertices): smallest domain first, tie-breaking on highest degree.
  - Tends to pick vertices adjacent to things we've already picked.
- Value ordering (i.e. target vertices): highest degree to lowest.

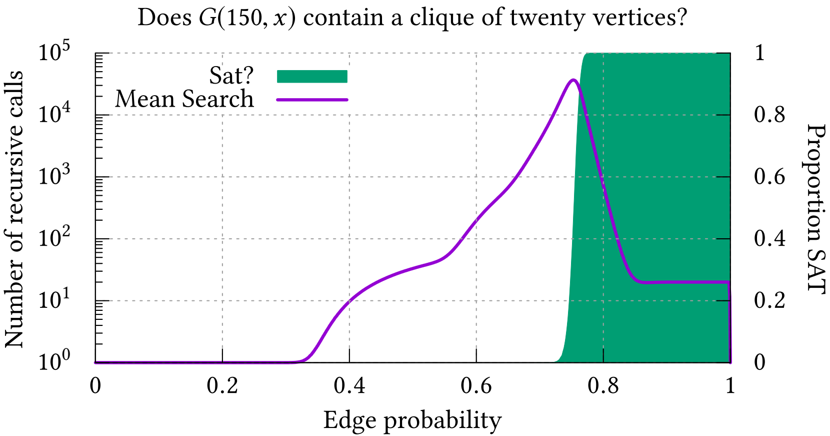
Sanity Check



Sanity Check



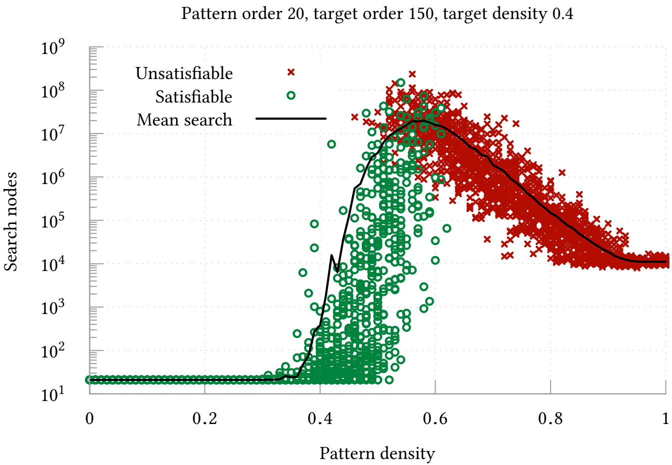
Clique in Random Graphs



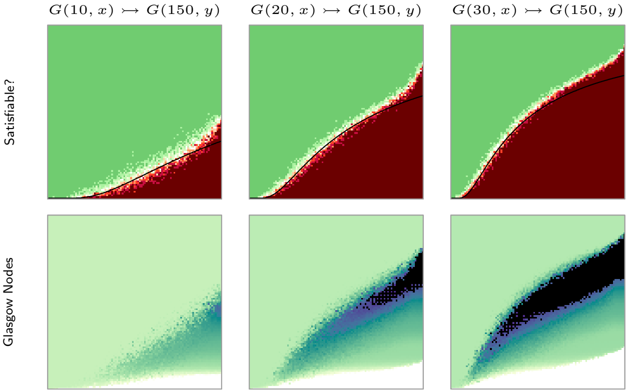
Let's Generate Random Instances a Different Way

- Decide upon a pattern graph order (number of vertices) and density.
- Decide upon a target graph order and density.
- Generate instances at random, independently.

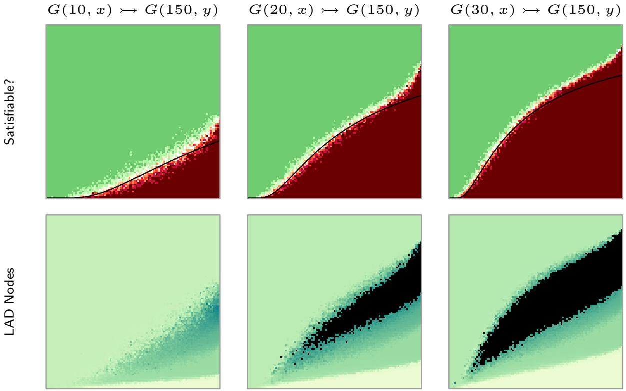
When is Non-Induced Subgraph Isomorphism Hard?



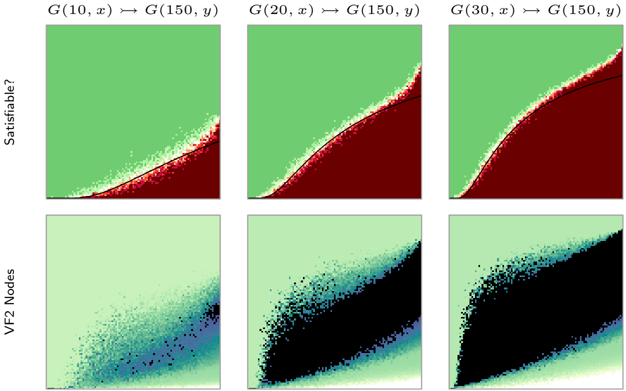
When is Non-Induced Subgraph Isomorphism Hard?



When is Non-Induced Subgraph Isomorphism Hard?



When is Non-Induced Subgraph Isomorphism Hard?



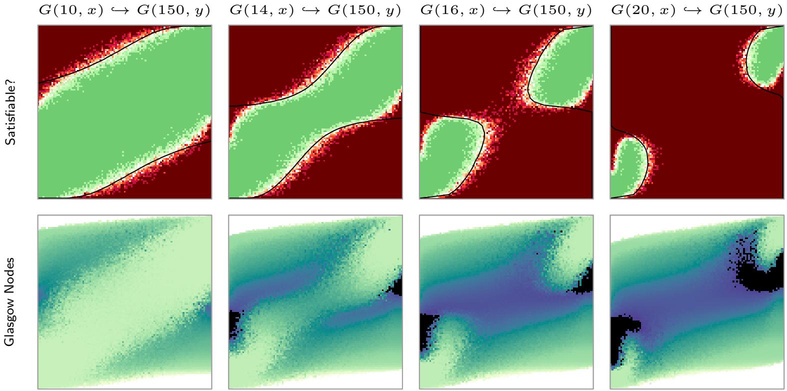
Hand-Wavy Theoretical Justification

- Maximise the expected number of solutions during search?
- If $P = G(p, q)$ and $T = G(t, u)$,

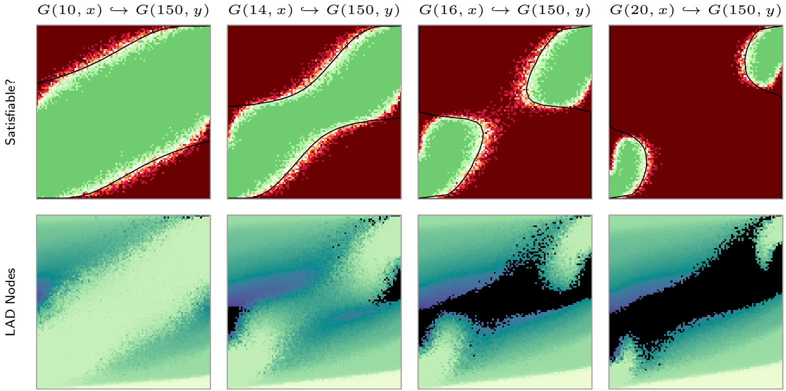
$$\langle Sol \rangle = \underbrace{t \cdot (t-1) \cdot \ldots \cdot (t-p+1)}_{\text{injective mapping}} \cdot \underbrace{u^{q \cdot \binom{p}{2}}}_{\text{adjacency}}$$

- Smallest domain first keeps remaining domain sizes large.
- High pattern degree makes the remaining pattern subgraph sparser, reducing q.
- High target degree leaves as many vertices as possible available for future use, making u larger.

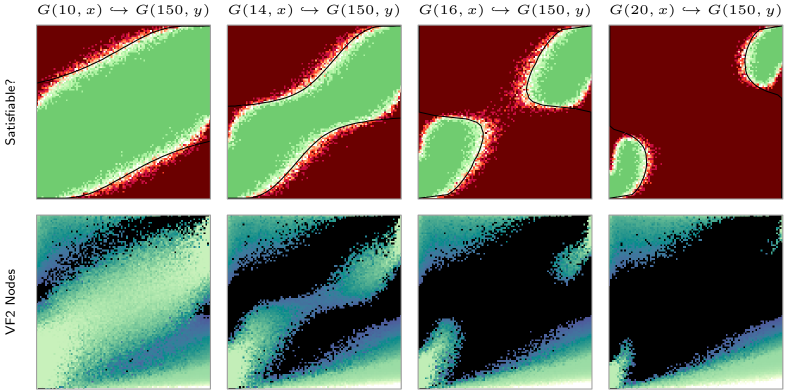
Induced is Much More Complicated



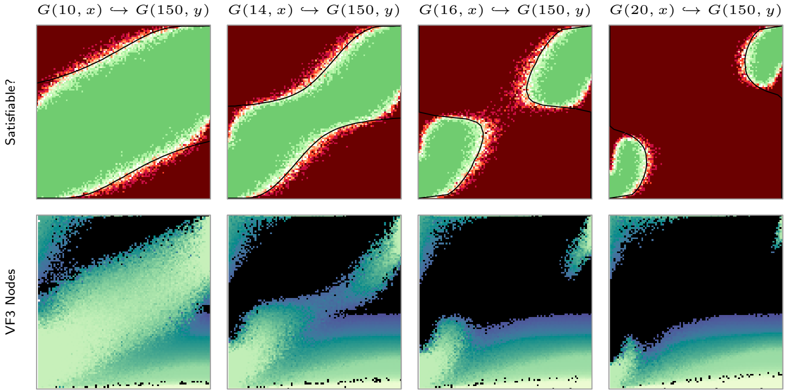
Induced is Much More Complicated



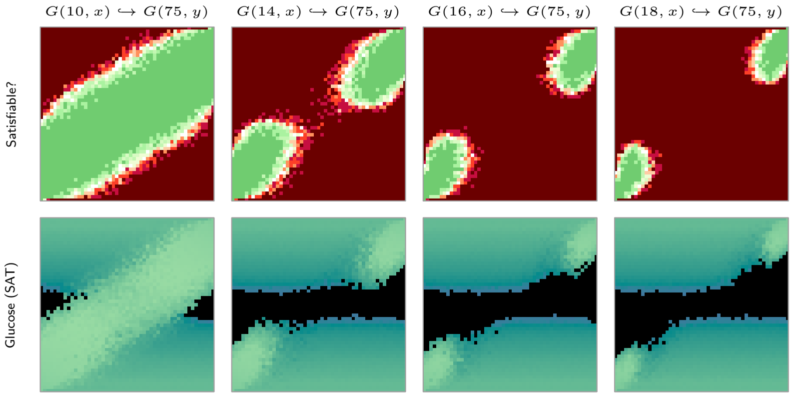
Induced is Much More Complicated



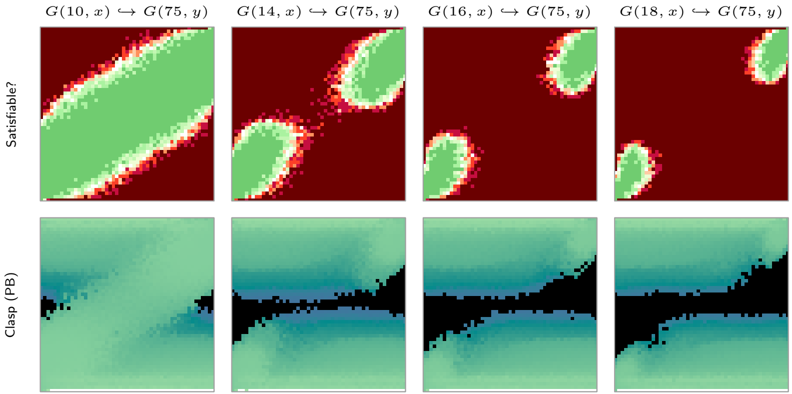
Induced is Much More Complicated



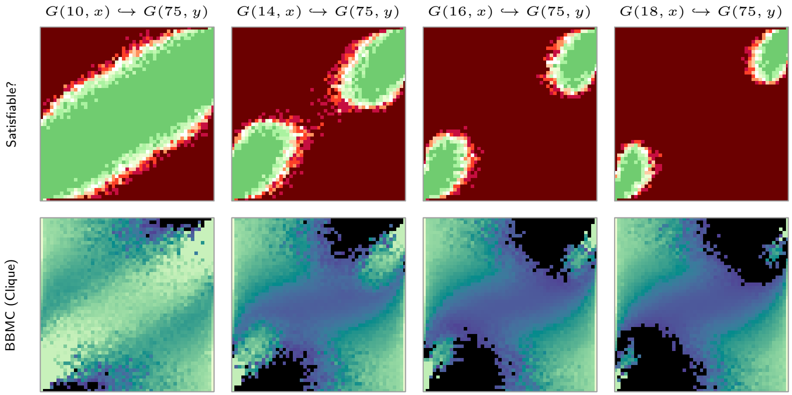
Is This Algorithm-Independent?



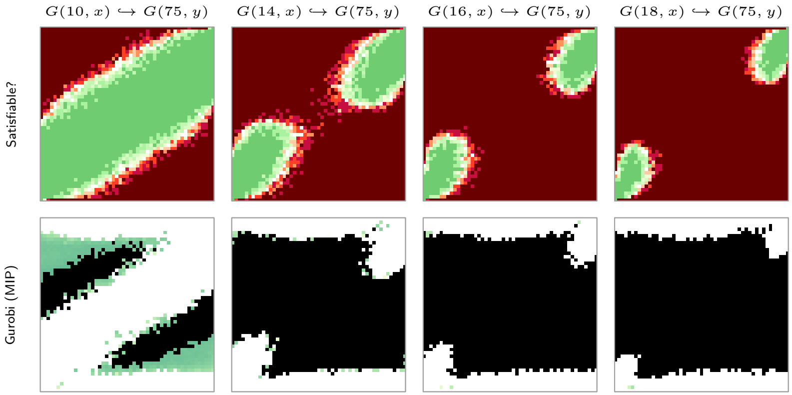
Is This Algorithm-Independent?



Is This Algorithm-Independent?



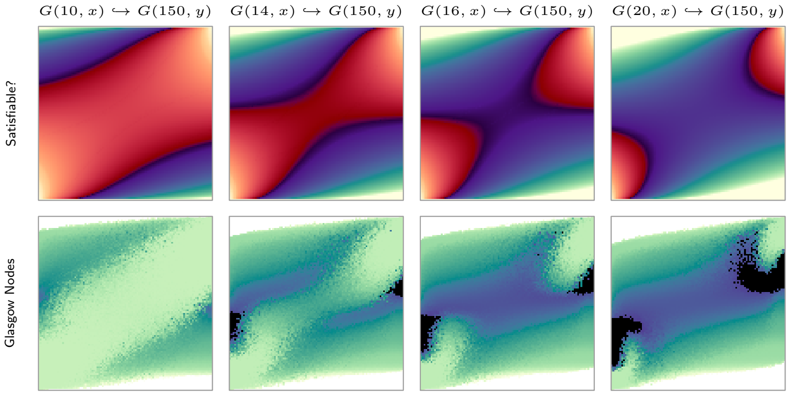
Is This Algorithm-Independent?



Constrainedness

$$\kappa = 1 - \frac{\log\left(t^{\underline{p}} \cdot u^{q \cdot \binom{p}{2}} \cdot (1-u)^{(1-q) \cdot \binom{p}{2}}\right)}{\log t^{\underline{p}}}$$

Constrainedness

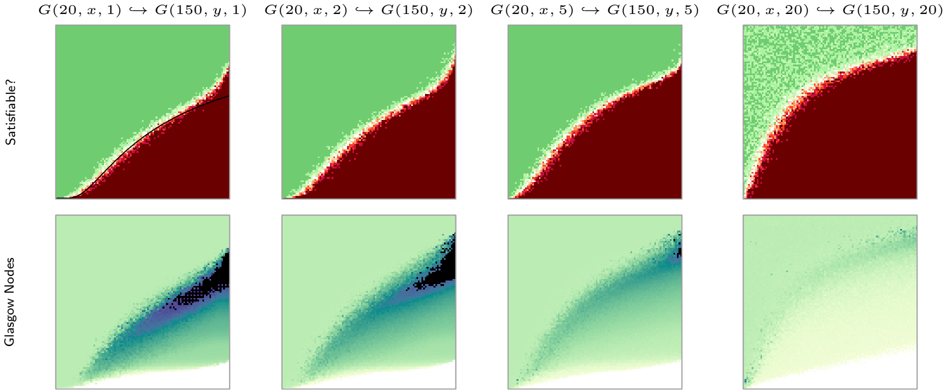


Labelled Subgraph Isomorphism

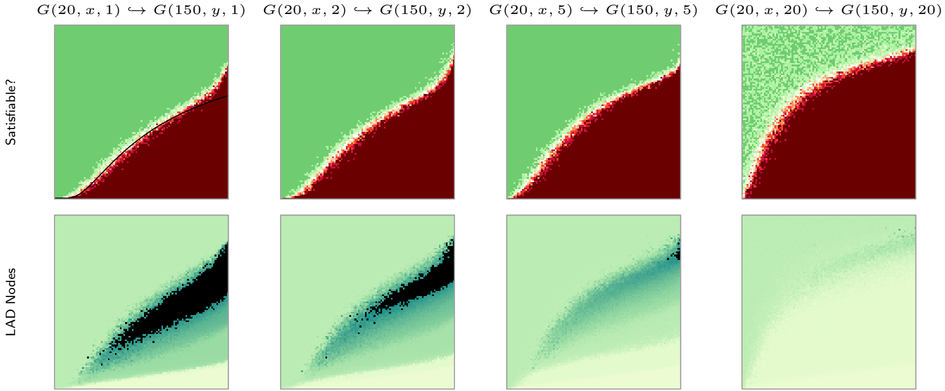
- Vertices have labels, and the isomorphism must preserve labels.
- Carbon must map to carbon, hydrogen to hydrogen, . . .

$$\langle Sol \rangle = \left(\frac{\Gamma\left({}^{t}\!/\!{}_{k} + 1\right)}{\Gamma\left({}^{t}\!/\!{}_{k} - {}^{p}\!/\!{}_{k} + 1\right)} \right)^{k} \cdot u^{q \cdot \binom{p}{2}}$$

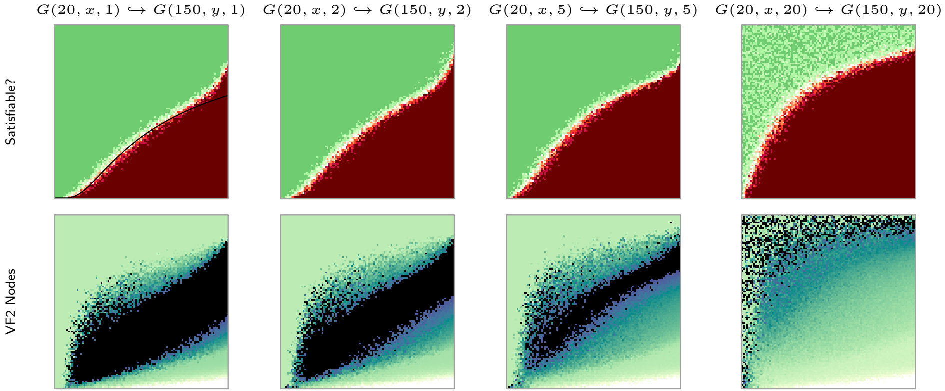
Labels and Phase Transitions



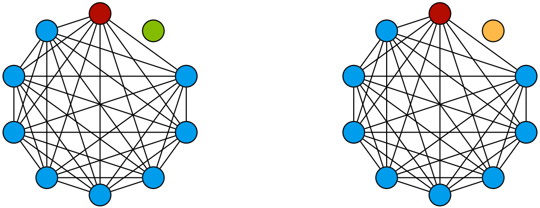
Labels and Phase Transitions



Labels and Phase Transitions



Connectivity Algorithms are Really Stupid



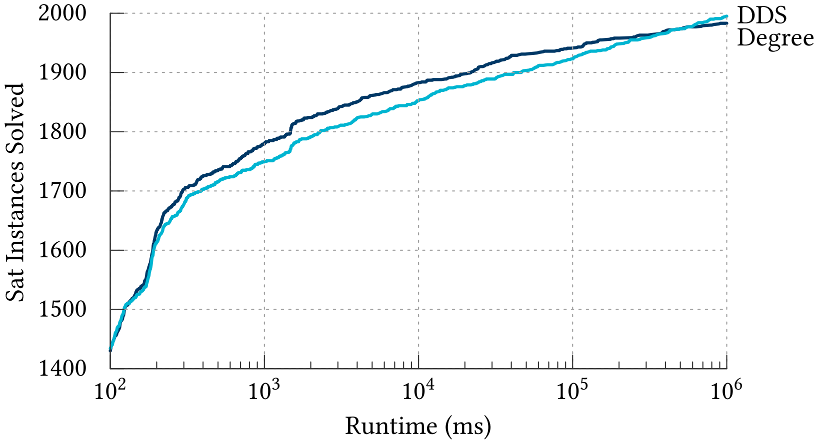
Back to Value-Ordering Heuristics

- Largest target degree first.

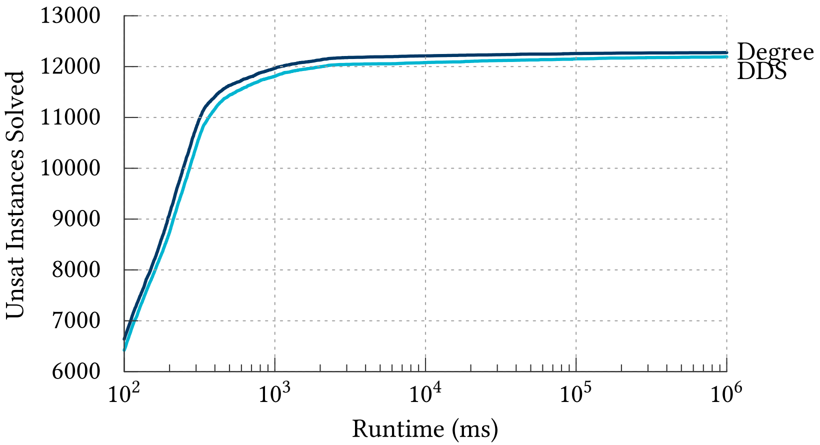
However. . .

- What if several vertices have the same degree?
- Is a vertex of degree 10 really that much better than a vertex of degree 9?

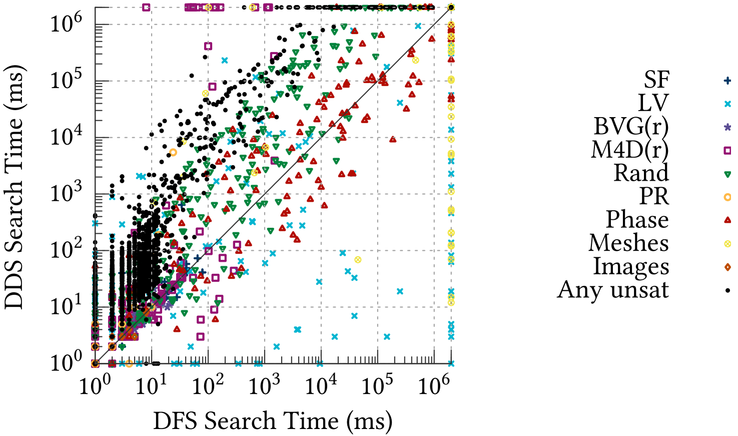
Discrepancy Search?



Discrepancy Search?



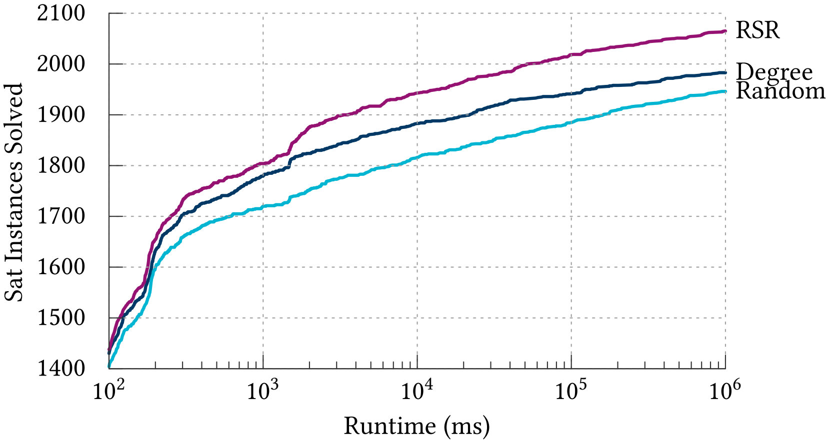
Discrepancy Search?



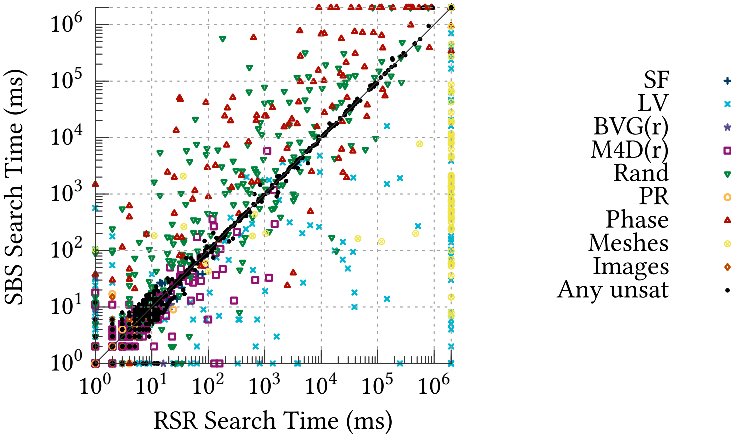
Random Search with Restarts and Nogood Recording

- Back to the random value-ordering heuristic.
- Aggressive restarts: every 100ms.
- Nogood recording and 2WL to avoid repeating work.

Random Search with Restarts and Nogood Recording



Random Search with Restarts and Nogood Recording



Value-Ordering Heuristics as Distributions

- Traditional view: value-ordering defines a search order.
- New view: value-ordering defines what proportion of the search effort should be spent on different subproblems.
- According to people who know more statistics than me, if solutions are uniformly distributed, then random search with restarts should be better than DFS.

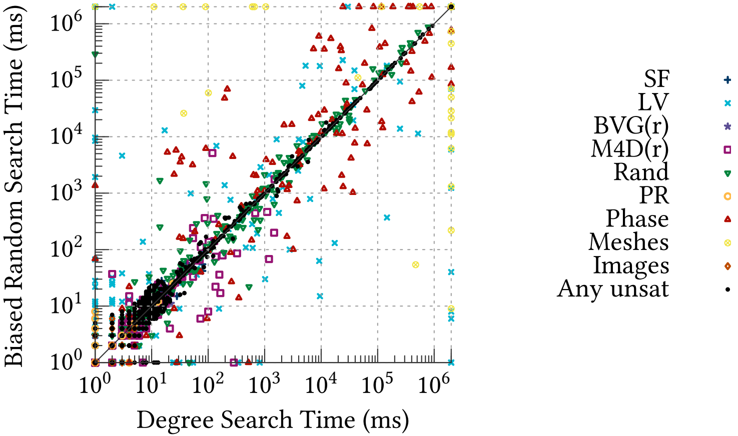
A Slightly Random Value-Ordering Heuristic

- For a fixed domain D_v, pick a vertex v' from a domain D_v with probability

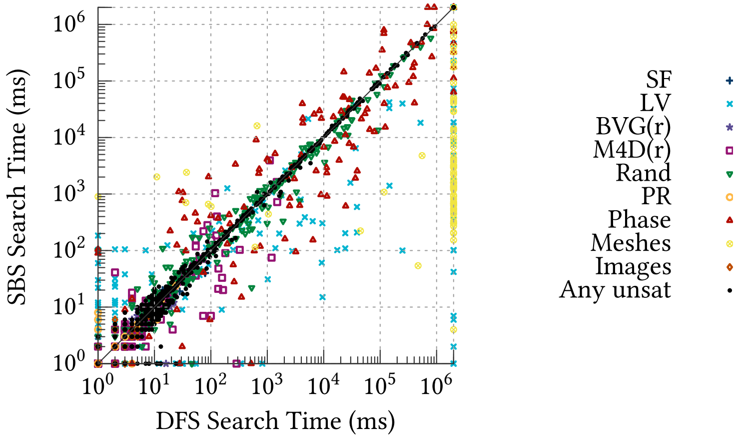
$$p(v') = \frac{2^{\deg(v')}}{\sum_{w \in D_v} 2^{\deg(w)}}$$

- Equally likely to pick between two vertices of degree d.
- Twice as likely to select a vertex of degree d than a vertex of degree $d - 1$.
- Justification: solution density and expected distribution of solutions.

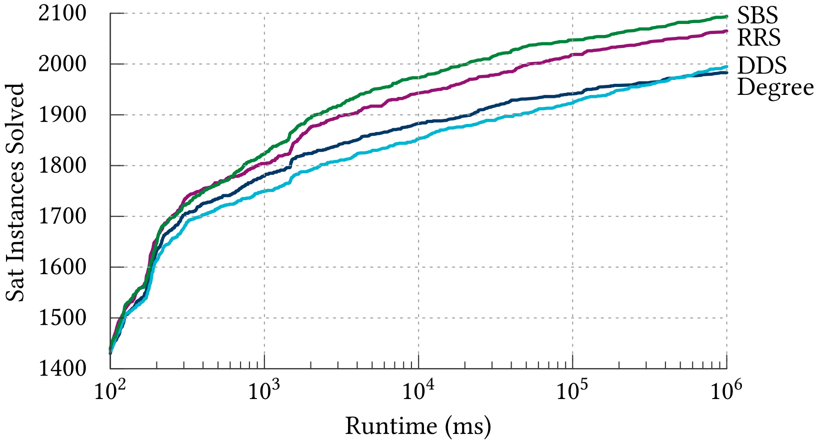
A Slightly Random Value-Ordering Heuristic



Is It Better?


SBS Search Time (ms)
DFS Search Time (ms)
10^0 10^1 10^2 10^3 10^4 10^5 10^6
SF
LV
BVG(r)
M4D(r)
Rand
PR
Phase
Meshes
Images
Any unsat

Is It Better?

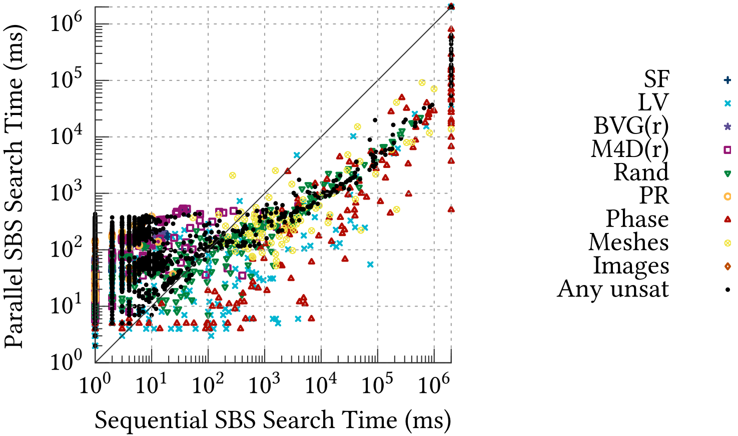


2100
2000
1900
1800
1700
1600
1500
1400
Sat Instances Solved
10^2
10^3
10^4
10^5
10^6
Runtime (ms)
SBS
RRS
DDS
Degree

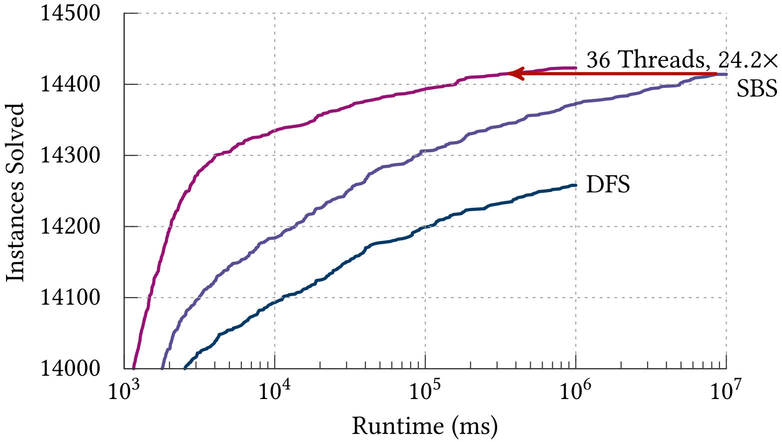
Parallel Search

- Each thread gets its own random seed.
- Barrier synchronise on restarts.
- Share nogoods.

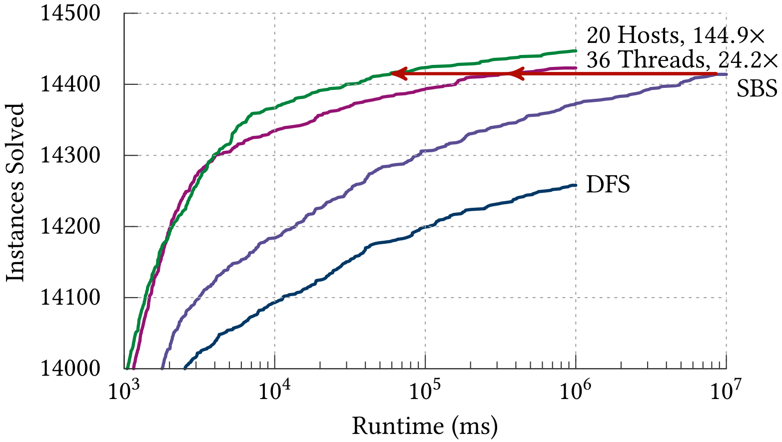
Is It Even Betterer?



Is It Even Betterer?



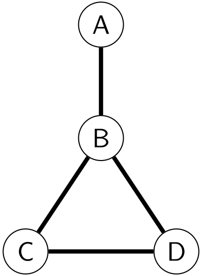
Is It Even Betterer?



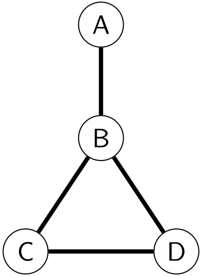
Lessons Learned

- Got to get a lot of things right:
  - Design.
  - Engineering.
  - Evaluation.
  - Understanding the hardware.
- Being clever only pays off if you can do it quickly.
  - Except sometimes it pays off even if it's really expensive.
- Not always clear what problem people really want to solve.

Symmetries

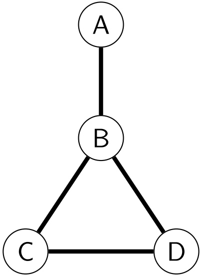


Symmetries



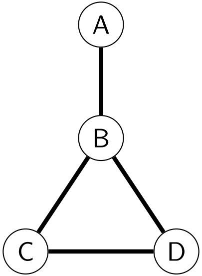
- Only find solutions where $C < D$.

Symmetries



- Only find solutions where $C < D$.
- What about for arbitrary symmetries, in both pattern and target graphs?

Symmetries



- Only find solutions where $C < D$.
- What about for arbitrary symmetries, in both pattern and target graphs?
- Dynamic symmetries?

Counting and Sampling

- We can easily enumerate all solutions.
- If we only need a count, can we speed things up?

Counting and Sampling

- We can easily enumerate all solutions.
- If we only need a count, can we speed things up?
- What if an approximate count is OK?

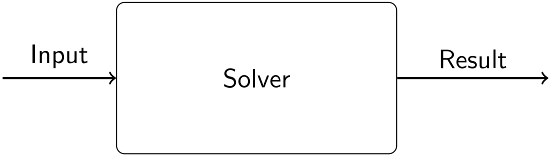
Counting and Sampling

- We can easily enumerate all solutions.
- If we only need a count, can we speed things up?
- What if an approximate count is OK?
- What if we want a few solutions, but sampled uniformly?
  - Common in term-rewriting systems.

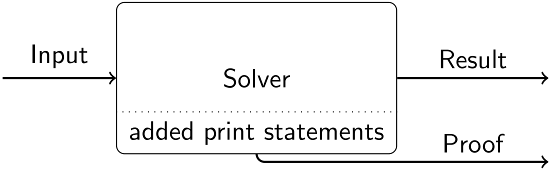
Counting and Sampling

- We can easily enumerate all solutions.
- If we only need a count, can we speed things up?
- What if an approximate count is OK?
- What if we want a few solutions, but sampled uniformly?
  - Common in term-rewriting systems.
- How does this interact with symmetries and decomposition?

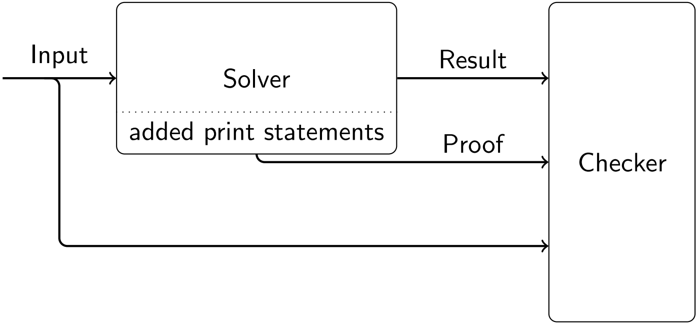
Proof Logging



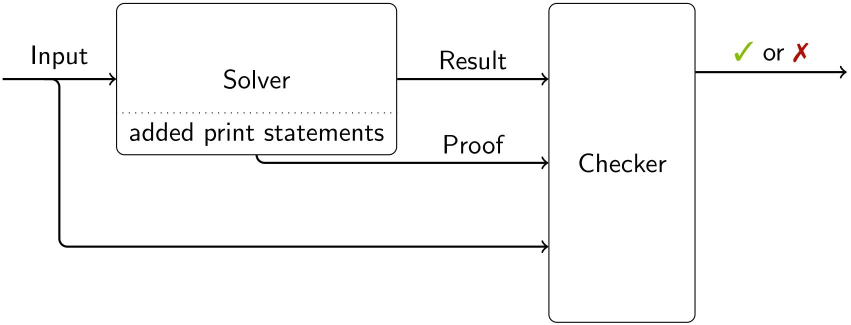
Proof Logging



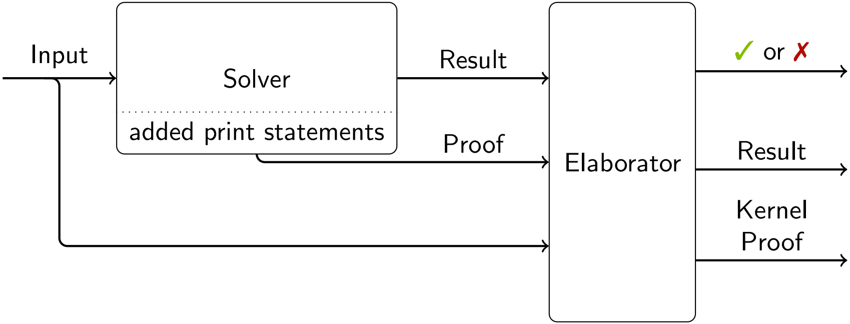
Proof Logging



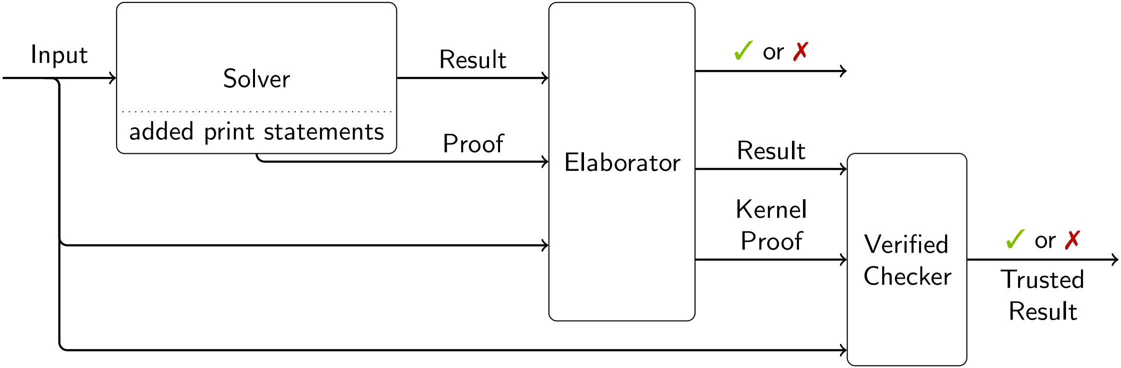
Proof Logging



Proof Logging



Proof Logging



Components

- What if the target graph has two components?

Components

- What if the target graph has two components?
- What if the pattern graph has two components?

Components

- What if the target graph has two components?
- What if the pattern graph has two components?
- What if the graphs are "nearly" two components?

Learning

- Backtracking is bad. We should do CDCL!
- Except it doesn't seem to work very well...

Inference on Fancy Graphs

- What's the equivalent of neighbourhood degree sequence for directed graphs?
- What about if we have labels?
- Can these be computed efficiently?

Automatic Configuration

- What if the pattern graph is a triangle? A claw? One edge and one non-edge? A large clique?

Automatic Configuration

- What if the pattern graph is a triangle? A claw? One edge and one non-edge? A large clique?
- Which supplemental graphs should we use?
- Which inference rules are helpful?

Presolving

- Constraint programming solvers take too long to start up for “really easy” instances.
- Run a “fast” solver for 0.1s and then switch?

Presolving

- Constraint programming solvers take too long to start up for "really easy" instances.
- Run a "fast" solver for 0.1s and then switch?
- Doesn't help us for very solution-dense enumeration problems though.

Performance Portability

- Will algorithms designed on this year's hardware work well next year?

Performance Portability

- Will algorithms designed on this year's hardware work well next year?
- Or on Mac ARM hardware rather than Intel / AMD x64?

Performance Portability

- Will algorithms designed on this year's hardware work well next year?
- Or on Mac ARM hardware rather than Intel / AMD x64?
- On heterogeneous multi-core?

File Formats

- Design a graph file format that isn't terrible.

https://ciaranm.github.io/

ciaran.mccreesh@glasgow.ac.uk

University of Glasgow

Royal Academy of Engineering