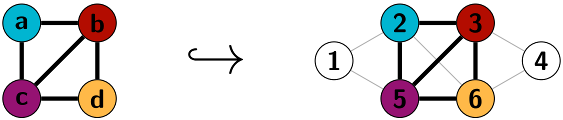
University of Glasgow

The Subgraph Isomorphism Problem: Three New Ideas

Ciaran McCreesh and Patrick Prosser

The Subgraph Isomorphism Problem

- Given a little *pattern* graph and a large *target* graph, find "a copy of" the pattern inside the target.
- We'll look at the *non-induced* or *monomorphism* variation: find an injective mapping that preserves adjacency, but not necessarily non-adjacency.



Existing Algorithms

- VF2: widely used, and extremely fast on small, sparse, low degree graphs. But if it doesn't find a result within ten milliseconds, it is unlikely to find a result within a day.
- LAD and SND: very clever CP-like algorithms with deep reasoning. But for some larger target graphs, a single propagation takes over a second.
    - We'll do much less reasoning, but can manage $> 100,000$ propagations per second.

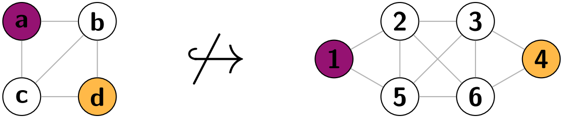
A CP-Like Model

- One variable per vertex in the pattern graph. The domain is the vertex in the target graph that it gets mapped to.
- For each adjacent pair of vertices in the pattern graph, their values must be adjacent in the target graph.
- All variables have different values.
- We can filter initial domains using degree, neighbourhood degree sequence, loops, ...

Supplemental Graphs

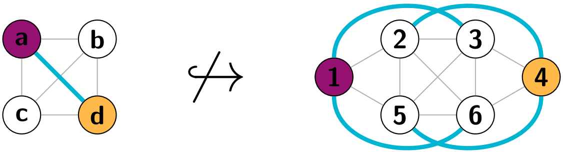
Distance-Based Filtering

- If two vertices are distance *d* apart in the pattern graph, they can only be mapped to a pair of vertices which are within distance *d* (or less) in the target graph.



Distance-Based 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 $i : P \hookrightarrow T$ is also a subgraph isomorphism $i^d : P^d \hookrightarrow T^d$.



Implied Constraints

- We're now trying to find a mapping i which is simultaneously a subgraph isomorphism

$$\begin{array}{rlcl} & i & : & P \hookrightarrow T \\ \text{and} & i^2 & : & P^2 \hookrightarrow T^2 \\ \text{and} & i^3 & : & P^3 \hookrightarrow T^3 \end{array}$$

  and so on.
- So we can filter on adjacency, degree, neighbourhood degree sequences, etc, in these graph pairs too.
- Open question: we can take the intersection, but is there a stronger operation which we can compute with reasonable complexity?

Path-Based Filtering

- In practice, this only seems to be useful for $d \leq 3$.
- Stronger: if two vertices in the pattern graph are connected by k paths of length exactly d, then they can only be mapped to a pair of vertices which have at least k paths of length exactly d between them.
    - We can also look at cycles: a vertex in a cycle of length k must be mapped to a vertex in a cycle of length k.
- We can do this as using graph transformation too. Let $G^{[d,k]}$ be the (loopy) graph with the same vertex set as G, and an edge between v and w if there are at least k paths or cycles of length exactly d between v and w in G.
- This is NP-hard to produce in general, but for $d \leq 3$ and small k we can calculate it quickly in practice.

Supplemental Graphs

- We just build these graphs once, at the top of search.
  - We could recreate them whenever a vertex disappears from every target domain, but this is costly.
  - We can cache these if we have a database of target graphs.
- Other transformations are sometimes helpful. We can either pick a good, general set, or use domain knowledge.
- Different transformations are helpful for other variations of the problem.
  - For the induced variant, we can also look at $\overline{G}$.
  - And we can compose transformations.

Is This Actually New?

- SND uses distances (not paths) for filtering.
- Inference using G^d is stolen from k-clique algorithms.

Counting All-Different

Injectivity / Enforcing All-Different

- When assigning $D_v \leftarrow w$, remove w from every other domain. If a domain ends up being empty, fail and backtrack.
- This enforces the constraint, but does not provide much additional inference.

Hall Sets

- If we have a subset of n variables, whose domains include exactly n values between them, then those values can *only* be used by those variables.
- If we have a subset of n variables, whose domains include less than n values between them, then we cannot give every variable a different value.

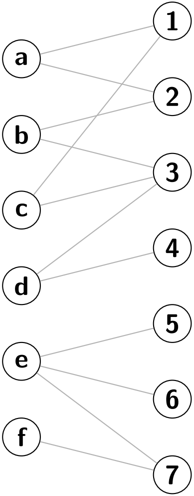
Régin's Matching-Based All-Different Filtering

- Build a bipartite graph, with variables on the left, values on the right, and edges for allowed assignments.
- Find a matching that covers every variable, or fail and backtrack if there isn't one.
- Remove every edge (variable-value assignment pair) which cannot occur in any maximum cardinality matching.

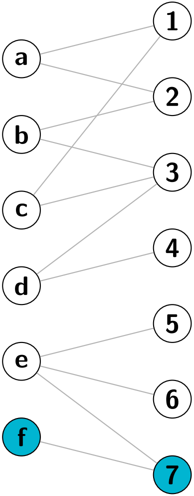
All-Different Filtering via Counting

- Go through each variable, from smallest domain to largest, and take the union of the domains as we go along.
- If we reach a failed Hall set, fail.
- If we reach a Hall set, remove all these values from every remaining domain, reset the counters, and keep going.
- This is much faster, especially when domains are already bitsets, but may miss some deletions that matching would find.

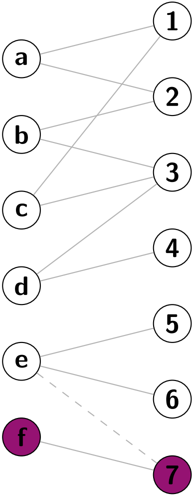
All-Different Filtering via Counting



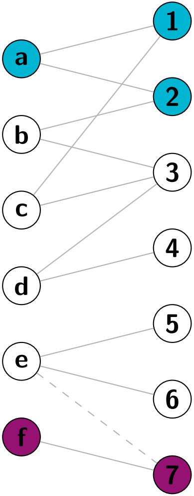
All-Different Filtering via Counting



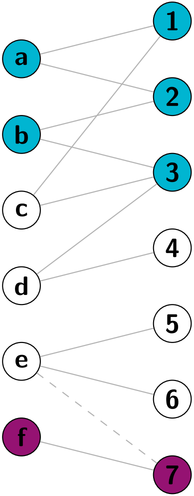
All-Different Filtering via Counting



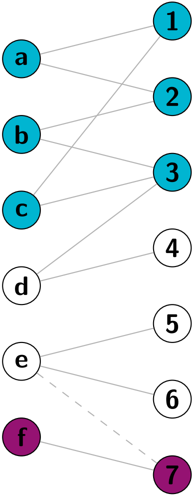
All-Different Filtering via Counting



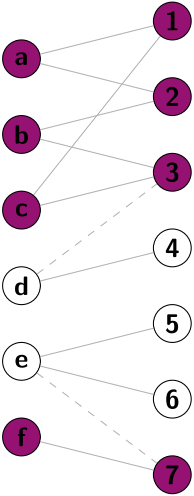
All-Different Filtering via Counting



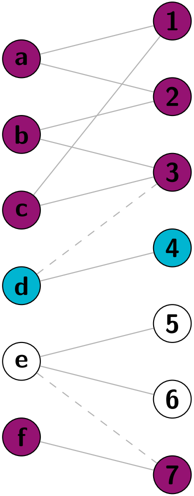
All-Different Filtering via Counting



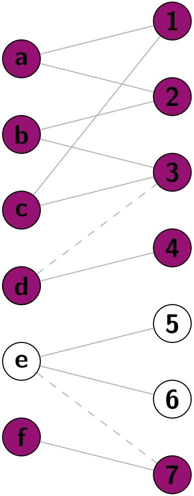
All-Different Filtering via Counting



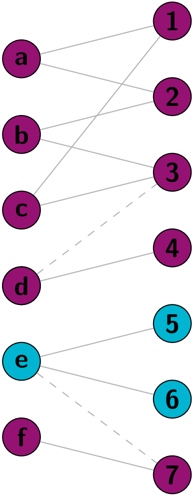
All-Different Filtering via Counting



All-Different Filtering via Counting



All-Different Filtering via Counting



All-Different Filtering via Counting

- In this case we found both variable-value assignments which could never occur.
- Had we done tie-breaking in a different order, we could have missed one of these.

Is This Actually New?

- Claude-Guy Quimper and Toby Walsh used counting as preprocessing in the context of set variables, but they use it to determine whether it's worth trying a matching.
- Javier Larrosa and Gabriel Valiente counted neighbours for SIP.
- There are other propagators for bounds consistency.
- I can't find this variation in the literature, possibly because it doesn't enforce any particular kind of consistency.

Backjumping

Backtracking is Dumb

- When we hit a failure, we could backtrack.
- Maybe the previous assignment didn't contribute to the failure, though.

Conflict-Directed Backjumping

- Conflict-directed backjumping keeps a conflict set for each variable. We track which assignments removed a value from a variable. When we backtrack, if we did not cause the failure, we can keep going backwards.
- But copying conflict sets gives a performance hit inside a “fast and dumb” algorithm.

Variable-Directed Backjumping

- When we assign and fail, return which variables were involved in the failing constraint.
- When we cannot find any value to assign to a variable, return the union of the variables in failed sub-searches, plus ourself. (Intuition: we might be able to succeed, if either we had another value, or if another problematic variable had another value.)
- When a search subproblem fails, determine whether the assignment we just made removed any values from any of the failing variables. If not, jump back another step straight away.
  - We don't need to track any additional information to do this, because we have both the domains we were given, and the clone which has had propagation applied to it.

Backjumping plus All-Different

- All-different(D) implies all-different(D') for any subset D' of D.
- If we can produce a small failed Hall set, we might be able to jump back further.
- We can just return the variables that we've seen so far.
  - This sometimes helps a lot in practice.
  - Maybe we could do more work to find an even better (not necessarily smaller) set?

Is This Actually New?

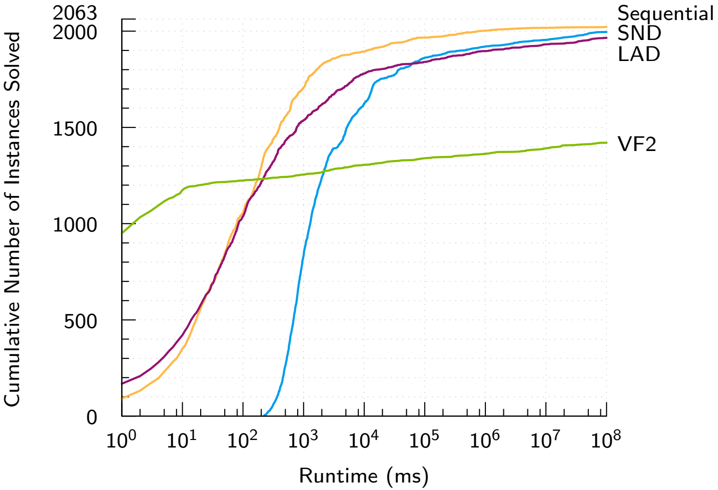
- Current subgraph isomorphism algorithms just backtrack.
- Neil Moore implemented lazy explanation generation for CP, but in a different way.
- Guillaume Rochart, Narendra Jussien and Franois Laburthe worked out better explanations for all-different via flows, in the context of interactive CP.

Preliminary Results

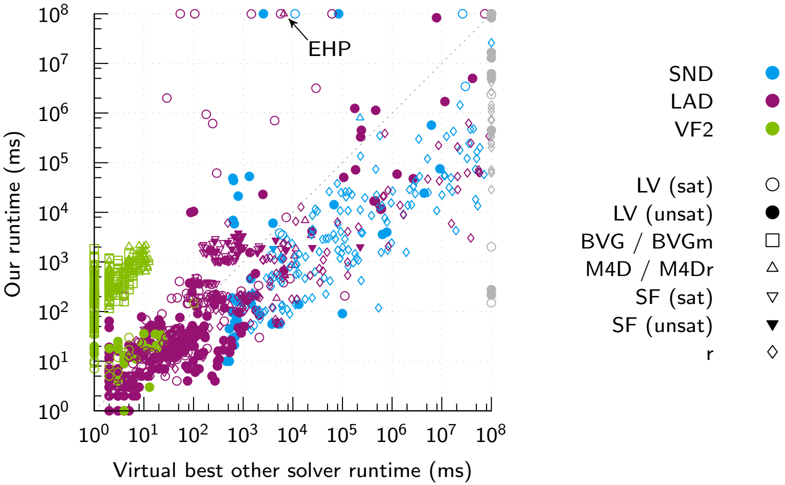
Is This Any Good?

- Fast and dumb isn't really fashionable for CP.
- Backjumping isn't fashionable anywhere...
- We'll look at the 2063 benchmark instances used to evaluate LAD and SND.
  - A mix of random, randomly structured, heavily structured, and real-world graphs.

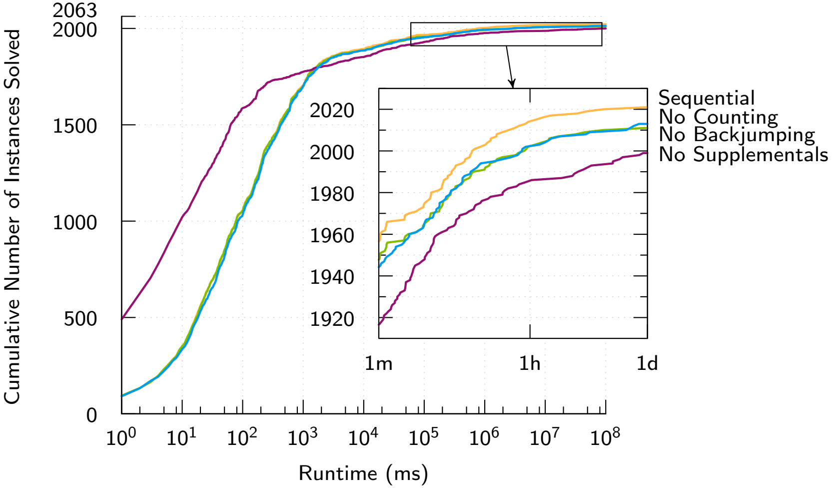
Cumulative Performance



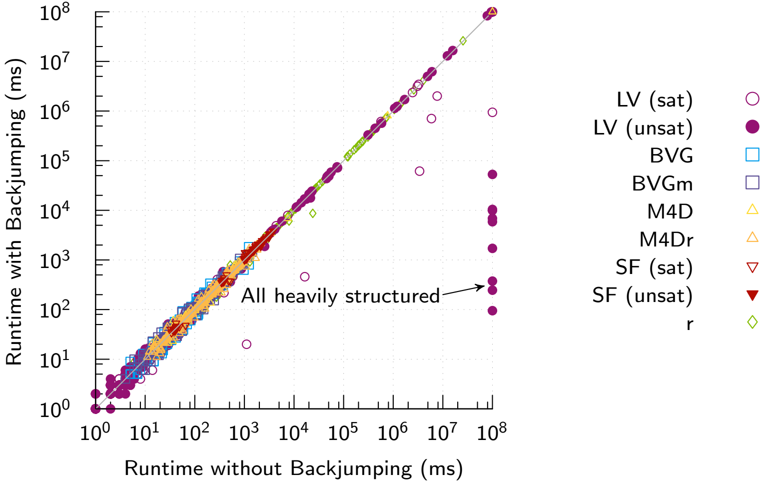
Per-Instance Comparison



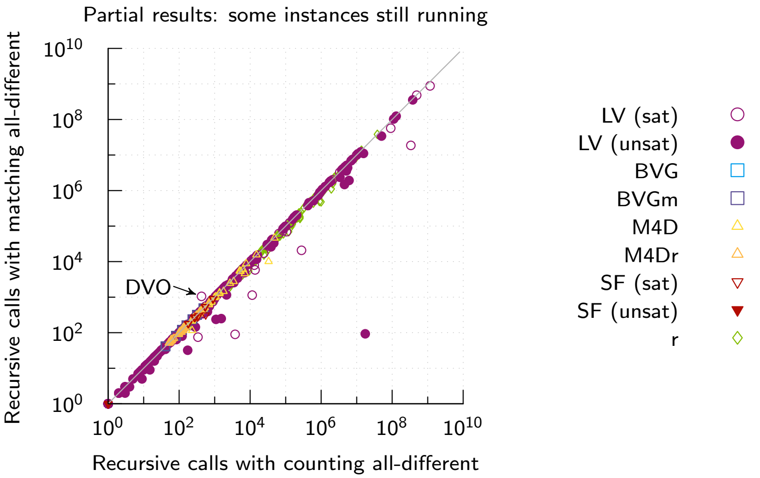
Is Each Feature Helpful?



Is Backjumping Any Good?



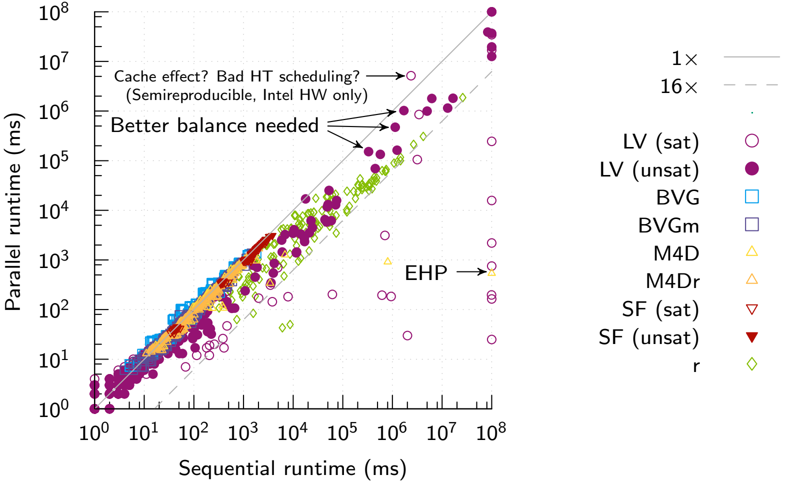
How Much Worse is Counting All-Different?



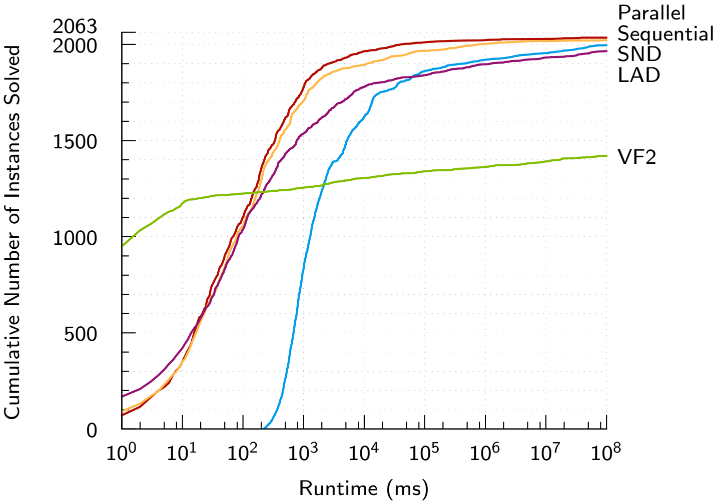
Very Quick Attempt at Threaded Tree-Search

- Half an hour's coding to check that the idea is sane.
- Distance 1 splitting to a queue, no extra load balancing yet.
- No parallelisation of supplemental graph construction yet.
- Speculative parallelism, so linear speedup should not be expected.
  - Not even for unsat instances, due to backjumping!
- 16 threads.

Very Quick Attempt at Threaded Tree-Search



Very Quick Attempt at Threaded Tree-Search



Proper Thread Parallelism?

- Work order matters: parallel diversity is a good alternative to discrepancy search.
- Proper load balancing is necessary.
- Lack of parallel supplemental graph construction means we can't ignore Amdahl's law.
- Backjumping makes all this quite fiddly.
    - It's easy to implement using Cilk, but we lose control of the work stealing strategy.
    - We could theoretically get an absolute slowdown due to backjumping. This is preventable by not "sharing backwards", but might not be worth it.

What's Next?

- All the variants (labels, directed edges, induced, ...)
- Other supplemental graphs
  - I can concoct additional transformations which can close half of the remaining open instances, but they're rather specialised.
- Portfolios and instance-specific algorithm configuration?
  - Does this legitimise special transformations?
- Symmetries and dominance?
- Better typesetting for $\overline{P} \not\rightarrowtail \overline{T}$?

University of Glasgow

http://dcs.gla.ac.uk/~ciaran
c.mccreesh.1@research.gla.ac.uk