A Parallel, Backjumping Subgraph Isomorphism Algorithm using Supplemental Graphs

Ciaran McCreesh* and Patrick Prosser

University of Glasgow, Glasgow, Scotland
c.mccreesh.1@research.gla.ac.uk,
patrick.prosser@glasgow.ac.uk

Abstract. The subgraph isomorphism problem involves finding a pattern graph inside a target graph. We present a new bit- and thread-parallel constraint-based search algorithm for the problem, and experiment on a wide range of standard benchmark instances to demonstrate its effectiveness. We introduce supplemental graphs, to create implied constraints. We use a new low-overhead, lazy variation of conflict directed backjumping which interacts safely with parallel search, and a counting-based all-different propagator which is better suited for large domains.

1 Introduction

The subgraph isomorphism family of problems involve "finding a copy of" a pattern graph inside a larger target graph; applications include bioinformatics [3], chemistry [31], computer vision [12,37], law enforcement [7], model checking [33], and pattern recognition [9]. These problems have natural constraint programming models: we have a variable for each vertex in the pattern graph, with the vertices of the target graph being the domains. The exact constraints vary depending upon which variation of the problem we are studying (which we discuss in the following section), but generally there are rules about preserving adjacency, and an all-different constraint across all the variables.

This constraint-based search approach dates back to works by Ullmann [39] and McGregor [25], and was improved upon in the LV [20] and VF2 [11] algorithms. More recently, ILF [41], LAD [36] and SND [1] are algorithms which take a "deep thinking" approach, using strong inference at each stage of the search. This is powerful, but we observe LAD or SND sometimes make less than one recursive call per second with larger target graphs, and cannot always explore enough of the search space to find a solution in time. This motivates an alternative approach: on the same hardware, we will be making 10^4 to 10^6 recursive calls per core per second. The main features of our algorithm are:

1. We introduce supplemental graphs, which generalise some of the ideas in SND. The key idea is that a subgraph isomorphism $i : P \rightarrowtail T$ induces a

* This work was supported by the Engineering and Physical Sciences Research Council [grant number EP/K503058/1]

subgraph isomorphism $F(i) : F(P) \rightarrowtail F(T)$, for certain functors F. This is used to generate implied constraints: we may now look for a mapping which is simultaneously a subgraph isomorphism between several carefully selected pairs of graphs.
2. We use weaker inference than LAD and SND: we do not achieve or maintain arc consistency. We introduce a cheaper, counting-based all-different propagator which has better scalability for large target graphs, and which has very good constant factors on modern hardware thanks to bitset encodings.
3. We describe a clone-comparing variation of conflict-directed backjumping, which does not require conflict sets. We show that an all-different propagator can produce reduced conflict explanations, which can improve backjumping.
4. We use thread-parallel preprocessing and search, to make better use of modern multi-core hardware. We explain how parallel search may interact safely with backjumping. We use explicit, non-randomised work stealing to offset poor early heuristic choices during search.

Although weaker propagation and backjumping have fallen out of fashion in general for constraint programming, here this approach usually pays off. In section 4 we show that over a large collection of instances commonly used to compare subgraph isomorphism algorithms, our solver is the single best.

2 Definitions, Notation, and a Proposition

Throughout, our graphs are finite, undirected, and do not have multiple edges between pairs of vertices, but may have loops (an edge from a vertex to itself). We write $\mathrm{V}(G)$ for the vertex set of a graph G, and $\mathrm{N}(G, v)$ for the neighbours of a vertex v in G (that is, the vertices adjacent to v). The *degree* of v is the cardinality of its set of neighbours. The *neighbourhood degree sequence* of v, denoted $\mathrm{S}(G, v)$, is the sequence consisting of the degrees of every neighbour of v, from largest to smallest. A vertex is *isolated* if it has no neighbours. By $v \sim_G w$ we mean vertex v is adjacent to vertex w in graph G. We write $G[V]$ for the subgraph of G induced by a set of vertices V.

A *non-induced subgraph isomorphism* is an injective mapping $i : P \rightarrowtail T$ from a graph P to a graph T which preserves adjacency—that is, if $v \sim_P w$ then we require $i(v) \sim_T i(w)$ (and thus if v has a loop, then $i(v)$ must have a loop). The *non-induced subgraph isomorphism problem* is to find such a mapping from a given pattern graph P to a given target graph T. (The *induced subgraph isomorphism problem* additionally requires that if $v \not\sim_P w$ then $i(v) \not\sim_T i(w)$, and variants also exist for directed and labelled graphs; we discuss only the non-induced version in this paper. All these variants are NP-complete.)

If R and S are sequences of integers, we write $R \preceq S$ if there exists a subsequence of S with length equal to that of R, such that each element in R is less than or equal to the corresponding element in S. For a set U and element v, we write $U - v$ to mean $U \setminus \{v\}$, and $U + v$ to mean $U \cup \{v\}$.

A *path* in a graph is a sequence of distinct vertices, such that each successive pair of vertices are adjacent; we also allow a path from a vertex to itself, in which case the first and last vertices in the sequence are the same (and there is a *cycle*). The *distance* between two vertices is the length of a shortest path between them. We write G^d for the graph with vertex set $\mathrm{V}(G)$, and edges between v and w if the distance between v and w in G is at most d. We introduce the notation $G^{[c,l]}$ for the graph with vertex set $\mathrm{V}(G)$, and edges between vertices v and w (not necessarily distinct) precisely if there are at least c paths of length exactly l between v and w in G. The following proposition may easily be verified by observing that injectivity means paths are preserved:

Proposition 1. *Let $i : P \rightarrowtail T$ be a subgraph isomorphism. Then i is also*

1. *a subgraph isomorphism $i^d : P^d \rightarrowtail T^d$ for any $d \geq 1$, and*
2. *a subgraph isomorphism $i^{[c,l]} : P^{[c,l]} \rightarrowtail T^{[c,l]}$ for any $c, l \geq 1$.*

The (contrapositive of the) first of these facts is used by SND, which dynamically performs distance-based filtering during search. We will instead use the second fact, at the top of search, to generate implied constraints.

3 A New Algorithm

Algorithm 1 describes our approach. We begin (line 3) with a simple check that there are enough vertices in the pattern graph for an injective mapping to exist. We then (line 4) discard isolated vertices in the pattern graph—such vertices may be greedily assigned to any remaining target vertices after a solution is found. This reduces the number of variables which must be copied when branching. Next we construct the supplemental graphs (line 5) and initialise domains (line 6). We then (line 7) use a counting-based all-different propagator to reduce these domains further. Finally, we perform a backtracking search (line 8). Each of these steps is elaborated upon below.

3.1 Preprocessing and Initialisation

Following Proposition 1, in Algorithm 2 we construct a sequence of supplemental graph pairs from our given pattern and target graph. We will then search for a mapping which is simultaneously a mapping from each pattern graph in the

Algorithm 1: A non-induced subgraph isomorphism algorithm

```
1 nonInducedSubgraphIsomorphism (Graph $\mathcal{P}$, Graph $\mathcal{T}$) $\rightarrow$ Bool
2 begin
3     if $|\mathrm{V}(\mathcal{P})| > |\mathrm{V}(\mathcal{T})|$ then return false
4     Discard isolated vertices in $\mathcal{P}$
5     $L \leftarrow$ createSupplementalGraphList($\mathcal{P}, \mathcal{T}$)
6     $D \leftarrow$ init($\mathrm{V}(\mathcal{P}), \mathrm{V}(\mathcal{T}), L$)
7     if countingAllDifferent($D$) $\neq$ Success then return false
8     return search($L, D$) = Success
```

Algorithm 2: Supplemental graphs for Algorithm 1

```
1 createSupplementalGraphList (Graph $\mathcal{P}$, Graph $\mathcal{T}$) $\to$ GraphPairs
2 begin
3     return $\Big[(\mathcal{P},\ \mathcal{T}),\ (\mathcal{P}^{[1,2]},\ \mathcal{T}^{[1,2]}),\ (\mathcal{P}^{[2,2]},\ \mathcal{T}^{[2,2]}),\ (\mathcal{P}^{[3,2]},\ \mathcal{T}^{[3,2]}),$
          $(\mathcal{P}^{[1,3]},\ \mathcal{T}^{[1,3]}),\ (\mathcal{P}^{[2,3]},\ \mathcal{T}^{[2,3]}),\ (\mathcal{P}^{[3,3]},\ \mathcal{T}^{[3,3]})\Big]$
```

sequence to its paired target graph—this gives us implied constraints, leading to additional filtering during search.

Our choice of supplemental graphs is somewhat arbitrary. We observe that distances of greater than 3 rarely give additional filtering power, and constructing $G^{[c,4]}$ is computationally very expensive (for unbounded l, the construction is NP-hard). Checking $c > 3$ is also rarely beneficial. Our choices work reasonably well in general on the wide range of benchmark instances we consider, but can be expensive for trivial instances—thus there is potential room to improve the algorithm by better selection on an instance by instance basis [23].

Algorithm 3 is responsible for initialising domains. We have a variable for each vertex in the (original) pattern graph, with each domain being the vertices in the (original) target graph. It is easy to see that a vertex of degree d in the pattern graph P may only be mapped to a vertex in the target graph T of degree d or higher: this allows us to perform some initial filtering. By extension, we may use compatibility of neighbourhood degree sequences: v may only be mapped to w if $\mathrm{S}(P, v) \preceq \mathrm{S}(T, w)$ [41]. Because any subgraph isomorphism $P \rightarrowtail T$ is also a subgraph isomorphism $F(P) \rightarrowtail F(T)$ for any of our supplemental graph constructions F, we may further restrict initial domains by considering only the intersection of filtered domains using each supplemental graph pair individually (line 5). At this stage, we also enforce the "loops must be mapped to loops" constraint.

Following this filtering, some target vertices may no longer appear in any domain, in which case R will be reduced on line 6. If this happens, we iteratively repeat the domain construction, but do not consider any target vertex no longer

Algorithm 3: Variable initialisation for Algorithm 1

```
1 init (Vertices $V$, Vertices $R$, GraphPairs $L$) $\to$ Domains
2 begin
3     repeat
4         foreach $v \in V$ do
5             $D_v \leftarrow \bigcap_{(P,\,T) \in L} \Big\{ w \in R :\ v \underset{P}{\sim} v \Rightarrow w \underset{T}{\sim} w \wedge \mathrm{S}(P, v) \preceq \mathrm{S}(T[R], w) \Big\}$
6         $R \leftarrow \bigcup_{v \in V} D_v$
7     until $R$ is unchanged
8     return $D$
```

Algorithm 4: Recursive search for Algorithm 1

```
search (GraphPairs $L$, Domains $D$) $\rightarrow$ Fail $F$ or Success
begin
    if $D = \emptyset$ then return Success
    $D_v \leftarrow$ a domain in $D$ with minimum size, tiebreaking on static degree in $\mathcal{P}$
    $F \leftarrow \{v\}$
    foreach $v' \in D_v$ ordered by static degree in $\mathcal{T}$ do
        $D' \leftarrow$ clone($D$)
        case assign($L, D', v, v'$) of
            Fail $F'$ then $F \leftarrow F \cup F'$
            Success then
                case search($L, D' - D_v$) of
                    Success then return Success
                    Fail $F'$ then
                        if $\nexists w \in F'$ such that $D_w \neq D'_w$ then return Fail $F'$
                        $F \leftarrow F \cup F'$
    return Fail $F$
```

in R when calculating degree sequences. (Note that for performance reasons, we do not recompute supplemental graphs when this occurs.)

3.2 Search and Inference

Algorithm 4 describes our recursive search procedure. If every variable has already been assigned, we succeed (line 3). Otherwise, we pick a variable (line 4) to branch on by selecting the variable with smallest domain, tiebreaking on descending static degree only in the original pattern graph (we tried other variations, including using supplemental graphs for calculating degree, and domain over degree, but none gave an overall improvement). For each value in its domain in turn, ordered by descending static degree in the target graph [13], we try assigning that value to the variable (line 8). If we do not detect a failure, we recurse (line 11).

The assignment and recursive search both either indicate success, or return a nogood set of variables F which cannot all be instantiated whilst respecting assignments which have already been made. This information is used to prune the search space: if a subproblem search has failed (line 13), but the current assignment did not remove any value from any of the domains involved in the discovered nogood (line 14), then we may ignore the current assignment and backtrack immediately. In fact this is simply conflict-directed backjumping [27] in disguise: rather than explicitly maintaining conflict sets to determine culprits (which can be costly when backjumping does nothing [2,14]), we lazily create the conflict sets for the variables in F' as necessary by comparing D before the current assignment with the D' created after it. Finally, as in backjumping, if none of the assignments are possible, we return with a nogood of the current

Algorithm 5: Variable assignment for Algorithm 5

```
1 assign (GraphPairs $L$, Domains $D$, Vertex $v$, Vertex $v'$) $\rightarrow$ Fail $F$ or Success
2 begin
3     $D_v \leftarrow \{v'\}$
4     foreach $D_w \in D - D_v$ do
5         $D_w \leftarrow D_w - v'$
6         foreach $(P, T) \in L$ do
7             if $v \sim_P w$ then $D_w \leftarrow D_w \cap \mathrm{N}(T, v')$
8         if $D_w = \emptyset$ then return Fail $\{w\}$
9     return countingAllDifferent($D$)
```

variable (line 5) combined with the union of the nogoods of each failed assignment (line 9) or subsearch (line 15).

For assignment and inference, Algorithm 5 gives the value v' to the domain D_v (line 3), and then infers which values may be eliminated from the remaining domains. Firstly, no other domain may now be given the value v' (line 5). Secondly, for each supplemental graph pair, any domain for a vertex adjacent to v may only be mapped to a vertex adjacent to v' (line 7). If any domain gives a wipeout, then we fail with that variable as the nogood (line 8).

To enforce the all-different constraint, it suffices to remove the assigned value from every other domain, as we did in line 5. However, it is often possible to do better. We can sometimes detect that an assignment is impossible even if values remain in each variable's domain (if we can find a set of n variables whose domains include strictly less than n values between them, which we call a *failed Hall set*), and we can remove certain variable-value assignments that we can prove will never occur (if we can find a set of n variables whose domains include only n values between them, which we call a *Hall set*, then those values may be removed from the domains of any other variable). The canonical way of doing this is to use Régin's matching-based propagator [30].

However, matching-based filtering is expensive and may do relatively little, particularly when domains are large, and the payoff may not always be worth the cost. Various approaches to offsetting this cost while maintaining the filtering power have been considered [15]. Since we are not maintaining arc consistency in general, we instead use an intermediate level of inference which is not guaranteed to identify every Hall set: this can be thought of as a heuristic towards the matching approach. This is described in Algorithm 6.

The algorithm works by performing a linear pass over each domain in turn, from smallest cardinality to largest (line 4). The H variable contains the union of every Hall set detected so far; initially it is empty. The A set accumulates the union of domains seen so far, and n contains the number of domains contributing to A. For each new domain we encounter, we eliminate any values present in previous Hall sets (line 6). We then add that domain's values to A and increment

Algorithm 6: Counting-based all-different propagation

```
countingAllDifferent (Domains $D$) $\rightarrow$ Fail $F$ **or** Success
begin
    $(F, H, A, n) \leftarrow (\emptyset, \emptyset, \emptyset, 0)$
    foreach $D_v \in D$ from smallest cardinality to largest do
        $F \leftarrow F + v$
        $D_v \leftarrow D_v \setminus H$
        $(A, n) \leftarrow (A \cup D_v, n + 1)$
        if $D_v = \emptyset$ or $|A| < n$ then return Fail $F$
        if $|A| = n$ then $(H, A, n) \leftarrow (H \cup A, \emptyset, 0)$
    return Success
```

n (line 7). If we detect a failed Hall set, we fail (line 8). If we detect a Hall set, we add those values to H, and reset A and n, and keep going (line 9).

It is important to note that this approach may fail to identify some Hall sets, if the initial ordering of domains is imperfect. However, the algorithm runs very quickly in practice: the sorting step is $\mathcal{O}(v \log v)$ (where v is the number of remaining variables), and the loop has complexity $\mathcal{O}(vd)$ (where d is the cost of a bitset operation over a target domain, which we discuss below). We validate this trade-off experimentally in the following section.

In case a failure is detected, the F set of nogoods we return need only include the variables processed so far, not every variable involved in the constraint. This is because an all-different constraint implies an all-different constraint on any subset of its variables. A smaller set of nogoods can increase the potential for backjumping (and experiments verified that this is beneficial in practice).

We have been unable to find this algorithm described elsewhere in the literature, although a sort- and counting-based approach has been used to achieve bounds consistency [28] (but our domains are not naturally ordered) and as a preprocessing step [29]. Bitsets (which we discuss below) have also been used to implement the matching algorithm [19].

3.3 Bit- and Thread-Parallelism

The use of bitset encodings for graph algorithms to exploit hardware parallelism dates back to at least Ullmann's algorithm [39], and remains an active area of research [32,40]. We use bitsets here: our graphs are stored as arrays of bit vectors, our domains are bit vectors, the neighbourhood intersection in Algorithm 5 is a bitwise-and operation, the unions in Algorithm 4 and Algorithm 6 are bitwise-or operations, and the cardinality check in Algorithm 6 is a population count (this is a single instruction in modern CPUs).

In addition to the SIMD-like parallelism from bitset encodings, we observed two opportunities for multi-core thread parallelism in the algorithm:

Graph and domain construction We may parallelise the outer `for` loops involved in calculating neighbourhood degree sequences and in initialising the domains of variables in Algorithm 3. Similarly, constructing each supplemental graph in Algorithm 2 involves an outer **for** loop, iterating over each vertex in the input graph. These loops may also be parallelised, with one caveat: we must be able to add edges to (but not remove edges from) the output graph safely, in parallel. This may be done using an atomic "or" operation.

Search Viewing the recursive calls made by the `search` function in Algorithm 4 as forming a tree, we may explore different subtrees in parallel. The key points are:

1. We do not know in advance whether the **foreach** loop (Algorithm 4 line 6) will exit early (either due to a solution being found, or backjumping). Thus our parallelism is speculative: we make a single thread always preserve the sequential search order, and use any additional threads to precompute subsequent entries in the loop which *might* be used. This may mean we get no speedup at all, if our speculation performs work which will not be used.
2. The **search** function, parallelised without changes, could attempt to exit early due to backjumping. We rule out this possibility by refusing to pass knowledge to the left: that is, we do not allow speculatively-found backjumping conditions to change the return value of **search**. This is for safety [38] and reproducibility: value-ordering heuristics can alter the performance of unsatisfiable instances when backjumping, and allowing parallelism to select a different backjump set could lead to an absolute slowdown [26]. To avoid this possibility, when a backjump condition is found, we *must* cancel any speculative work being done to the right of its position, and *cannot* cancel any ongoing work to the left. This means that unlike in conventional backtracking search without learning, we should *not* expect a linear speedup for unsatisfiable instances.

   (In effect we are treating the **foreach** loop as a parallel lazy fold, so that a subtree does not depend upon items to its left. Backjumping conditions are left-zero elements [21], although we do not have a unique zero.)
3. If any thread finds a solution, we *do* succeed immediately, even if this involves passing knowledge to the left. If there are multiple solutions, this can lead to a parallel search finding a different solution to the one which would be found sequentially—since the solution we find is arbitrary, this is not genuinely unsafe. However, this means we could witness a superlinear (greater than n from n threads) speedup for satisfiable instances [4].
4. For work stealing, we explicitly prioritise subproblems highest up and then furthest left in the search tree. This is because we expect our value-ordering heuristics to be weakest early on in search [18], and we use parallelism to offset poor choices early on in the search [6,24].

4 Experimental Evaluation

Our algorithm was implemented[1]in C++ using C++11 native threads, and was compiled using GCC 4.9.0. We performed our experiments on a machine with dual Intel Xeon E5-2640 v2 processors (for a total of 16 cores, and 32 hardware threads via hyper-threading), running Scientific Linux 6.6. For the comparison with SND in the following section, we used Java HotSpot 1.8.0_11. Runtimes include preprocessing and thread launch costs, but not the time taken to read in the graph files from disk (except in the case of SND, which we were unable to instrument).

For evaluation, we used the same families of benchmark instances that were used to evaluate LAD [36] and SND [1]. The "LV" family [20] contains graphs with various interesting properties from the Stanford Graph Database, and the 793 pattern/target pairs give a mix of satisfiable and unsatisfiable queries. The "SF" family contains 100 scale-free graph pairs, again mixing satisfiable and unsatisfiable queries. The remainder of these graphs come from the Vflib database [11]: the "BVG" and "BVGm" families are bounded degree graphs (540 pairs all are satisfiable), "M4D" and "M4Dr" are four-dimensional meshes (360 pairs, all satisfiable), and the "r" family is randomly generated (270 pairs, all satisfiable). We expanded this suite with 24 pairs of graphs representing image pattern queries [12] (which we label "football"), and 200 randomly selected pairs from each of a series of 2D image ("images") and 3D mesh ("meshes") graph queries [37]. The largest number of vertices is 900 for a pattern and 5,944 for a target, and the largest number of edges is 12,410 for a pattern and 34,210 for a target; some of these graphs do contain loops. All 2,487 instances are publicly available in a simple text format[2].

4.1 Comparison with Other Solvers

We compare our implementation against the Abscon 609 implementation of SND (which is written in Java) [1], Solnon's C implementation of LAD [36], and the VFLib C implementation of VF2 [11]. (The versions of each of these solvers we used could support loops in graphs correctly.)

Note that SND is not inherently multi-threaded, but the Java 8 virtual machine we used for testing makes use of multiple threads for garbage collection even for sequential code. On the one hand, this could be seen as giving SND an unfair advantage. However, nearly all modern CPUs are multi-core anyway, so one could say that it is everyone else's fault for not taking advantage of these extra resources. We therefore present both sequential (from a dedicated implementation, *not* a threaded implementation running with only a single thread) and threaded results for our algorithm.

In Fig. 1 we show the cumulative performance of each algorithm. The value of the line at a given time for an algorithm shows the total number of instances

[1] source code, data, experimental scripts and raw results available at `https://github.com/ciaranm/cp2015-subgraph-isomorphism`

[2] `http://liris.cnrs.fr/csolnon/SIP.html`

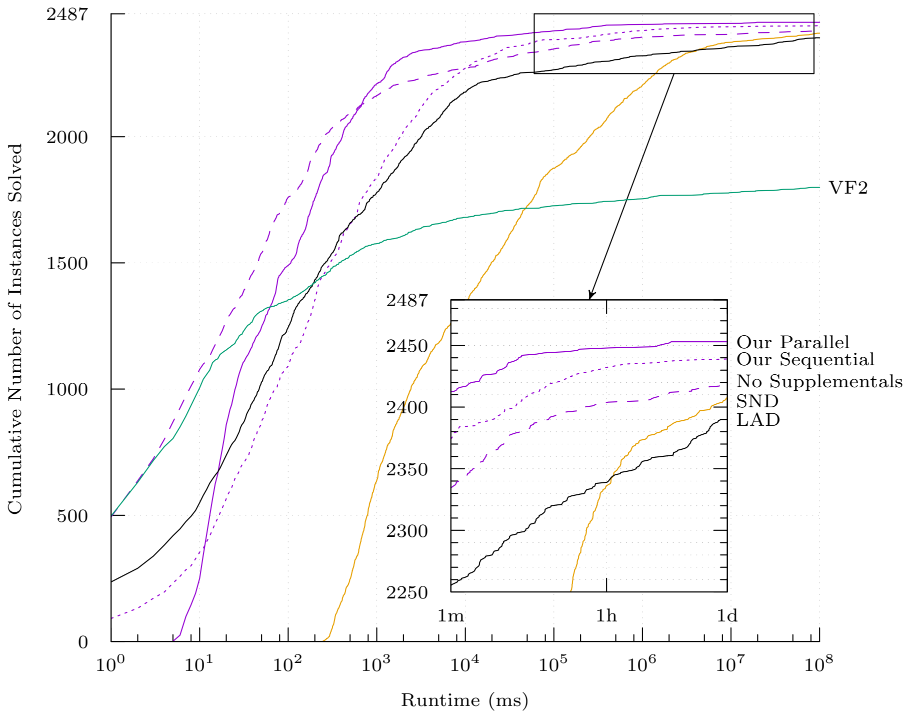

Fig. 1. Cumulative number of benchmark instances solved within a given time, for different algorithms: at time t, the value is the size of the set of instances whose runtime is at most t for that algorithm. Parallel results are using 32 threads on a 16 core hyper-threaded system.

which, individually, were solved in at most that amount of time. Our sequential implementation beats VF2 for times over 0.2s, LAD for times over 0.6s, and always beats SND. Our parallel implementation beats VF2 for times over 0.06s, LAD for times over 0.02s, and always beats SND; parallelism gives us an overall benefit from 12ms onwards. Finally, removing the supplemental graphs from our sequential algorithm gives an improvement below 10s (due to the cost of preprocessing), but is beneficial for longer runtimes.

Fig. 2 presents an alternative perspective of these results. Each point represents an instance, and the shape of a point shows its family. For the y position for an instance, we use our sequential (top graph) or parallel (bottom graph) runtime. For the x position, we use the runtime from the virtual best other solver; the colour of a point indicates which solver this is. For any point below the $x = y$ diagonal line, we are the best solver. A limit of 10^8 ms was used—points along the outer axes represent timeouts.

Although overall ours is the single best solver, VF2 is stronger on trivial instances. This is not surprising: we must spend time constructing supplemental

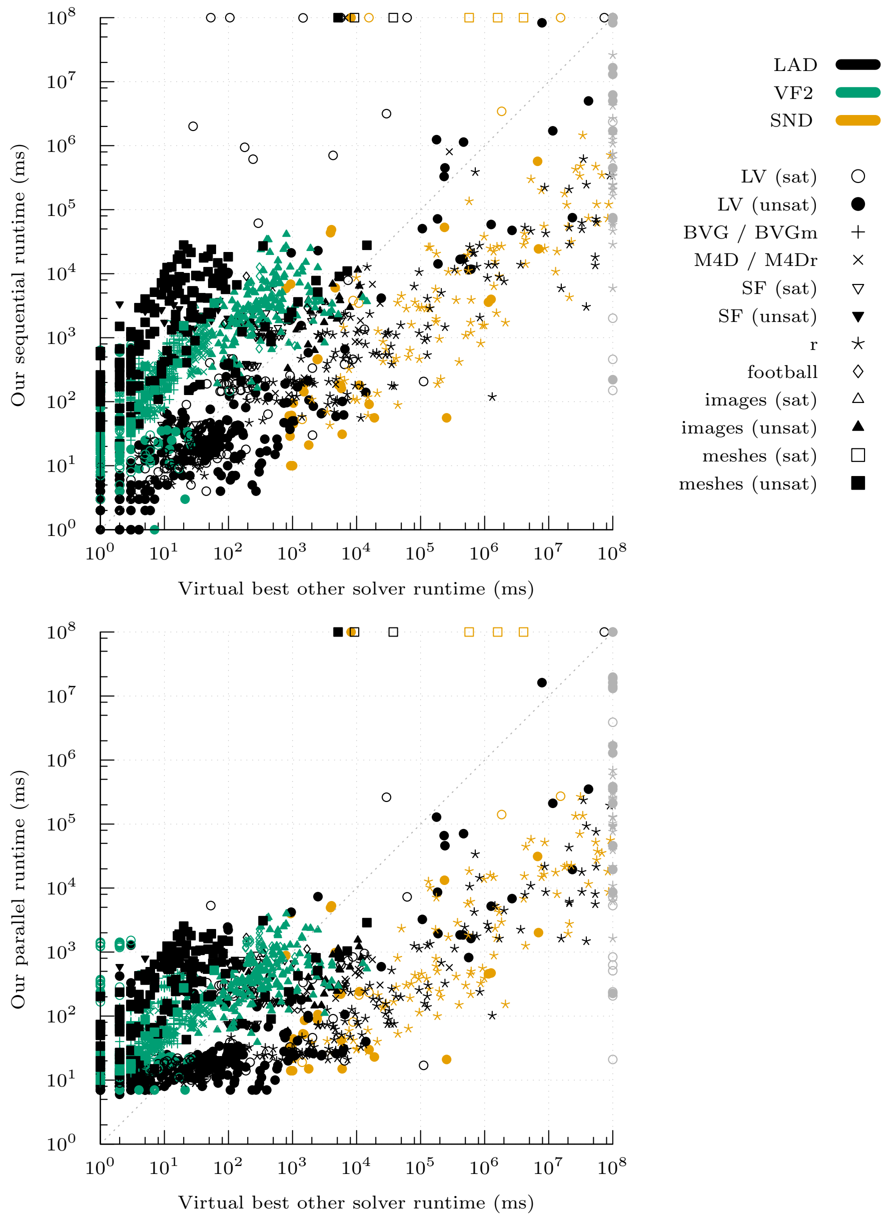

Fig. 2. Above, our sequential runtime compared to the virtual best other sequential solver, for each benchmark instance; below, the same, with our parallel runtimes and including parallel solvers. For points below the diagonal line, ours is the best solver for this instance; for points above the diagonal, the point colour indicates the best solver.

graphs. Thus it may be worth using either VF2 or our own algorithm without supplemental graphs as a presolver, if short runtimes for trivial instances is desirable—this may be the case in graph database systems where many trivial queries must be run [16] (although these systems could cache the supplemental graphs for targets). These results also suggest potential scope for algorithm portfolios, or instance-specific configuration: for example, we could omit or use different supplemental graphs in some cases.

4.2 Parallelism

Fig. 3 shows, for each instance, the speedup obtained from parallelism. Except at very low sequential runtimes, we see a reasonable general improvement. For some satisfiable instances, we see strongly superlinear speedups. These instances are exceptionally hard problems [35]: we would have found a solution quickly, except for a small number of wrong turns at the top of search. Our work stealing strategy was able to avoid strong commitment to early value-ordering heuristic choices, providing an alternative to using more complicated sequential search strategies to offset this issue. (Some of these results were also visible in Fig. 2, where we timed out on satisfiable instances which another solver found trivial.)

Some non-trivial satisfiable instances exhibited a visible slowdown. This is because we were using 32 software threads, to match the advertised number of hardware hyper-threads, but our CPUs only have 16 “real” cores between them. For these instances parallelism did not reduce the amount of work required to find a solution, but did result in a lower rate of recursive calls per second on the sequential search path—this is similar to the risk of introducing a slower processing element to a cluster [38]. Even when experimenting with 16 threads, we sometimes observed a small slowdown due to worse cache and memory bus performance, and due to the overhead of modifying the code to allow for work stealing (recall that we are benchmarking against a dedicated sequential implementation).

In a small number of cases, we observe low speedups for non-trivial unsatisfiable instances. These are from cases where backjumping has a substantial effect on search, making much of our speculative parallelism wasted effort. (Additionally, if cancellation were not to be used, some of these instances would exhibit large absolute slowdowns.)

4.3 Effects of Backjumping

In Fig. 4 we show the benefits of backjumping: points below the diagonal line indicate an improvement to runtimes from backjumping. Close inspection shows that backjumping usually at least pays for itself, or gives a slight improvement. (This was not the case when we maintained conflict sets explicitly: there, the overheads lead to a small average slowdown.)

For a few instances, backjumping makes an improvement of several orders of magnitude. The effects are most visible for some of the LV benchmarks, which consist of highly structured graphs. This mirrors the conclusions of Chen and Van Beek [5], who saw that “adding CBJ to a backtracking algorithm ... can

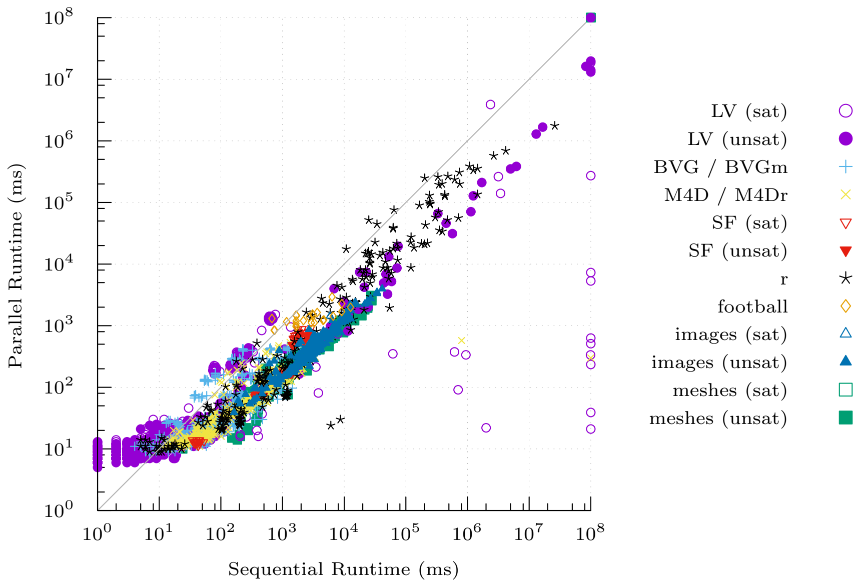

Fig. 3. The effects of parallelism, using 32 threads on a 16 core hyper-threaded system. Each point is a problem instance; points below the diagonal line indicate a speedup.

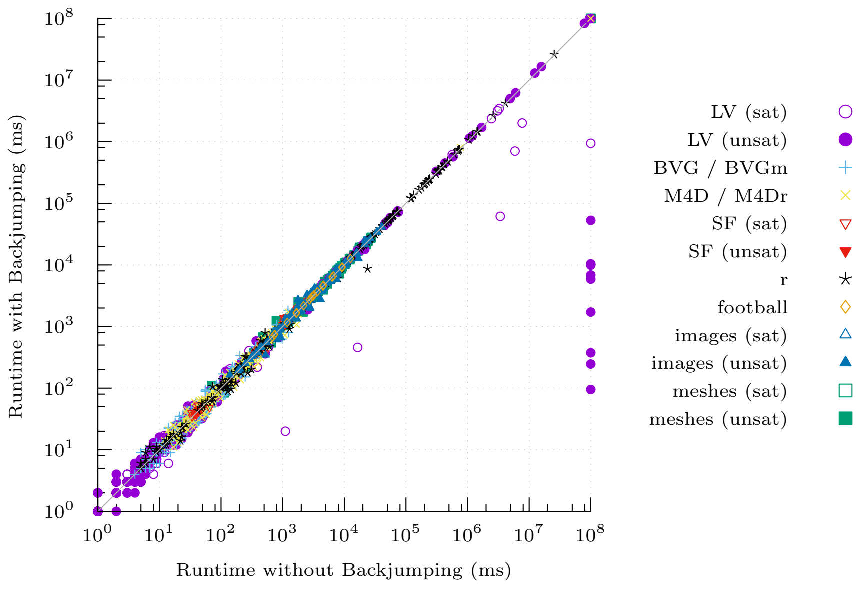

Fig. 4. The effects of backjumping. Each point is one benchmark instance; points below the diagonal line indicate a speedup.

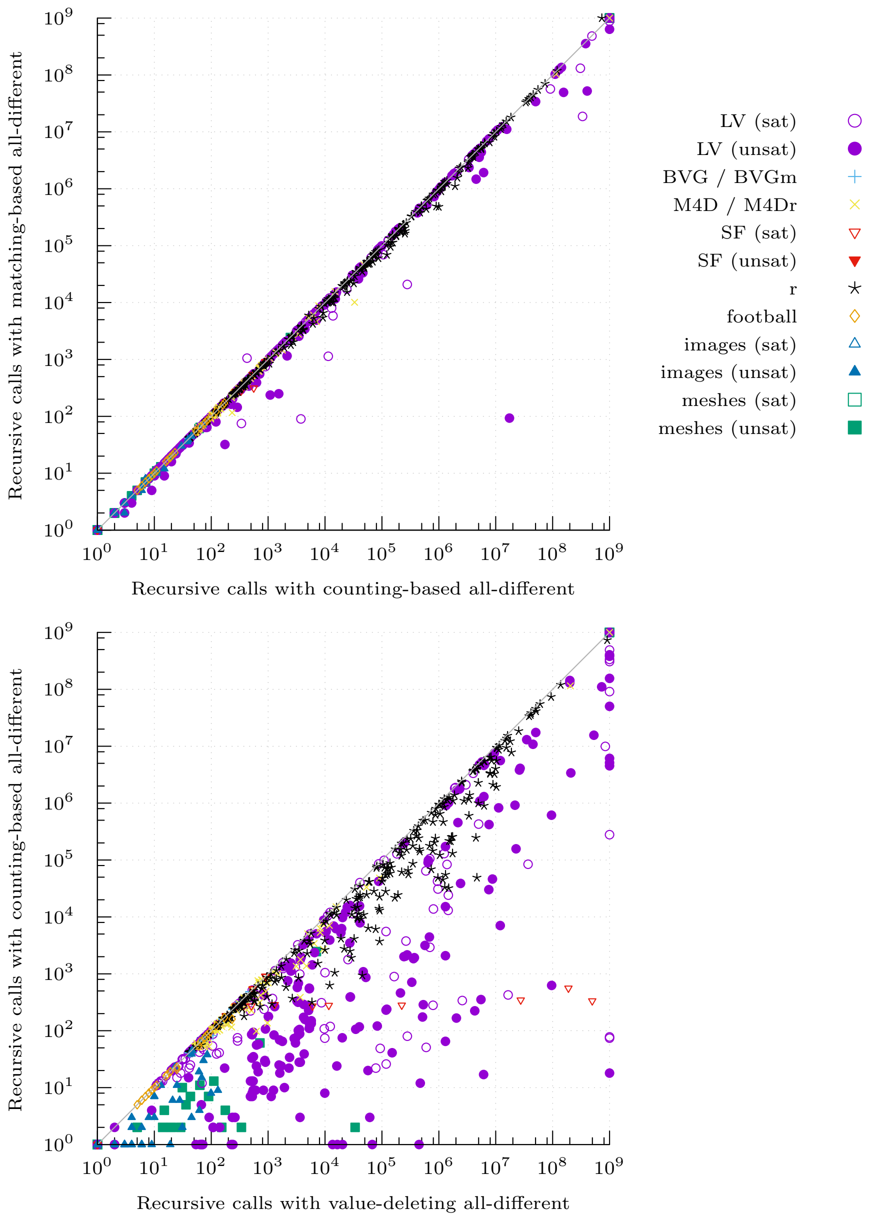

Fig. 5. Above, the improvement to the search space size which would be given by Régin's matching-based all-different propagator. Below, the improvement given by using counting all-different rather than simple deletion. Each point is one benchmark instance; the point style shows the benchmark family. Points below the diagonal line indicate a reduction in the search space size.

(still) speed up the algorithm by several orders of magnitude on hard, structured problems". Real-world graphs often have unexpected structural properties which are not present in random instances [22,34], so we consider backjumping to be worthwhile.

4.4 Comparing All-Different Propagators

We now justify our use of the counting all-different propagator. In the top half of Fig. 5 we show the benefits to the size of the search space that would be gained if we used Régin's algorithm at every step instead of our counting propagator (cutting search off after 10^9 search nodes). We see that for most graph families, there would be little to no benefit even if there was no additional performance cost. Only in a small portion of the LV graphs do we see a gain (and in one case, due to dynamic variable ordering, there is a penalty).

Thus, either our counting propagator is nearly always as good as matching, or neither propagator does very much in this domain. In the bottom half of Fig. 5 we show the benefits to the size of the search space that are gained from using counting, rather than simply deleting a value from every other domain on assignment. The large number of points below the diagonal line confirm that going beyond simple value deletion for all-different propagation is worthwhile.

5 Conclusion

Going against conventional wisdom, we saw that replacing strong inference with cheaper surrogates could pay off, and that backjumping could be implemented cheaply enough to be beneficial. We also saw parallelism give a substantial benefit. This was true even for relatively low runtimes, due to us exploiting parallelism for pre-processing as well as for search. Parallel backjumping has only been given limited attention [8,17,10]. However, a simple approach has worked reasonably well here (in contrast to stronger clause-learning systems, where successes in parallelism appear to be rare).

There is also plenty of scope for extensions of and improvement to our algorithm. We have yet to deeply investigate the possibility of constructing domain- or instance-specific supplemental graphs. Nor did we discuss directed graphs or induced isomorphisms: supplemental graphs can be taken further for these variations of the problem. In particular, composing transformations for induced isomorphisms would allow us to reason about "paths of non-edges", which may be very helpful. Finally, we have yet to consider exploiting the symmetries and dominance relations which we know are present in many graph instances.

Acknowledgements

The authors wish to thank Christine Solnon for discussions, providing the graph instances and the LAD implementation, Christophe Lecoutre for discussion and the SND implementation, and Lars Kotthoff and Alice Miller for their comments.

References

1. Audemard, G., Lecoutre, C., Modeliar, M.S., Goncalves, G., Porumbel, D.: Scoring-based neighborhood dominance for the subgraph isomorphism problem. In: Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings. pp. 125–141 (2014), http://dx.doi.org/10.1007/978-3-319-10428-7_12
2. Bessière, C., Régin, J.: MAC and combined heuristics: Two reasons to forsake FC (and cbj?) on hard problems. In: Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, Cambridge, Massachusetts, USA, August 19-22, 1996. pp. 61–75 (1996), http://dx.doi.org/10.1007/3-540-61551-2_66
3. Bonnici, V., Giugno, R., Pulvirenti, A., Shasha, D., Ferro, A.: A subgraph isomorphism algorithm and its application to biochemical data. BMC Bioinformatics 14(Suppl 7), S13 (2013), http://www.biomedcentral.com/1471-2105/14/S7/S13
4. de Bruin, A., Kindervater, G., Trienekens, H.: Asynchronous parallel branch and bound and anomalies. In: Ferreira, A., Rolim, J. (eds.) Parallel Algorithms for Irregularly Structured Problems, Lecture Notes in Computer Science, vol. 980, pp. 363–377. Springer Berlin Heidelberg, Berlin, Heidelberg (1995), http://dx.doi.org/10.1007/3-540-60321-2_29
5. Chen, X., van Beek, P.: Conflict-directed backjumping revisited. J. Artif. Intell. Res. (JAIR) 14, 53–81 (2001), http://dx.doi.org/10.1613/jair.788
6. Chu, G., Schulte, C., Stuckey, P.J.: Confidence-based work stealing in parallel constraint programming. In: Gent, I.P. (ed.) Principles and Practice of Constraint Programming - CP 2009, Lecture Notes in Computer Science, vol. 5732, pp. 226–241. Springer Berlin Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-04244-7_20
7. Coffman, T., Greenblatt, S., Marcus, S.: Graph-based technologies for intelligence analysis. Commun. ACM 47(3), 45–47 (Mar 2004), http://doi.acm.org/10.1145/971617.971643
8. Conrad, J., Mathew, J.: A backjumping search algorithm for a distributed memory multicomputer. In: Parallel Processing, 1994. ICPP 1994 Volume 3. International Conference on. vol. 3, pp. 243–246 (Aug 1994)
9. Conte, D., Foggia, P., Sansone, C., Vento, M.: Thirty years of graph matching in pattern recognition. International Journal of Pattern Recognition and Artificial Intelligence 18(03), 265–298 (2004), http://www.worldscientific.com/doi/abs/10.1142/S0218001404003228
10. Cope, M., Gent, I.P., Hammond, K.: Parallel heuristic search in Haskell. In: Selected papers from the 2nd Scottish Functional Programming Workshop (SFP00), University of St Andrews, Scotland, July 26th to 28th, 2000. pp. 65–76 (2000)
11. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. IEEE Trans. Pattern Anal. Mach. Intell. 26(10), 1367–1372 (2004), http://doi.ieeecomputersociety.org/10.1109/TPAMI.2004.75
12. Damiand, G., Solnon, C., de la Higuera, C., Janodet, J.C., Émilie Samuel: Polynomial algorithms for subisomorphism of nD open combinatorial maps. Computer Vision and Image Understanding 115(7), 996 – 1010 (2011), http://www.sciencedirect.com/science/article/pii/S1077314211000816, special issue on Graph-Based Representations in Computer Vision

13. Geelen, P.A.: Dual viewpoint heuristics for binary constraint satisfaction problems. In: ECAI. pp. 31–35 (1992)
14. Gent, I.P., Miguel, I., Moore, N.C.: Lazy explanations for constraint propagators. In: Carro, M., Pea, R. (eds.) Practical Aspects of Declarative Languages, Lecture Notes in Computer Science, vol. 5937, pp. 217–233. Springer Berlin Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-11503-5_19
15. Gent, I.P., Miguel, I., Nightingale, P.: Generalised arc consistency for the alldifferent constraint: An empirical survey. Artificial Intelligence 172(18), 1973 – 2000 (2008), http://www.sciencedirect.com/science/article/pii/S0004370208001410, special Review Issue
16. Giugno, R., Bonnici, V., Bombieri, N., Pulvirenti, A., Ferro, A., Shasha, D.: Grapes: A software for parallel searching on biological graphs targeting multi-core architectures. PLoS ONE 8(10), e76911 (10 2013), http://dx.doi.org/10.1371%2Fjournal.pone.0076911
17. Habbas, Z., Herrmann, F., Merel, P.P., Singer, D.: Load balancing strategies for parallel forward search algorithm with conflict based backjumping. In: Parallel and Distributed Systems, 1997. Proceedings., 1997 International Conference on. pp. 376–381 (Dec 1997)
18. Harvey, W.D., Ginsberg, M.L.: Limited discrepancy search. In: IJCAI (1). pp. 607–615. Morgan Kaufmann, San Francisco, CA, USA (1995)
19. Kessel, P.V., Quimper, C.: Filtering algorithms based on the word-RAM model. In: Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada. (2012), http://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/view/5135
20. Larrosa, J., Valiente, G.: Constraint satisfaction algorithms for graph pattern matching. Mathematical Structures in Computer Science 12(4), 403–422 (2002), http://dx.doi.org/10.1017/S0960129501003577
21. Lobachev, O.: Parallel computation skeletons with premature termination property. In: Schrijvers, T., Thiemann, P. (eds.) Functional and Logic Programming, Lecture Notes in Computer Science, vol. 7294, pp. 197–212. Springer Berlin Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-29822-6_17
22. MacIntyre, E., Prosser, P., Smith, B.M., Walsh, T.: Random constraint satisfaction: Theory meets practice. In: Maher, M.J., Puget, J. (eds.) Principles and Practice of Constraint Programming - CP98, 4th International Conference, Pisa, Italy, October 26-30, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1520, pp. 325–339. Springer (1998), http://dx.doi.org/10.1007/3-540-49481-2_24
23. Malitsky, Y.: Instance-Specific Algorithm Configuration. Springer (2014), http://dx.doi.org/10.1007/978-3-319-11230-5
24. McCreesh, C., Prosser, P.: The shape of the search tree for the maximum clique problem and the implications for parallel branch and bound. ACM Trans. Parallel Comput. 2(1), 8:1–8:27 (Apr 2015), http://doi.acm.org/10.1145/2742359
25. McGregor, J.J.: Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. Inf. Sci. 19(3), 229–250 (1979), http://dx.doi.org/10.1016/0020-0255(79)90023-9
26. Prosser, P.: Domain filtering can degrade intelligent backtracking search. In: Proceedings of the 13th International Joint Conference on Artifical Intelligence - Volume 1. pp. 262–267. IJCAI'93, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1993), http://dl.acm.org/citation.cfm?id=1624025.1624062
27. Prosser, P.: Hybrid algorithms for the constraint satisfaction problem. Computational Intelligence 9, 268–299 (1993), http://dx.doi.org/10.1111/j.1467-8640.1993.tb00310.x

28. Puget, J.: A fast algorithm for the bound consistency of alldiff constraints. In: Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA. pp. 359–366 (1998), `http://www.aaai.org/Library/AAAI/1998/aaai98-051.php`
29. Quimper, C., Walsh, T.: The all different and global cardinality constraints on set, multiset and tuple variables. In: Recent Advances in Constraints, Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2005, Uppsala, Sweden, June 20-22, 2005, Revised Selected and Invited Papers. pp. 1–13 (2005), `http://dx.doi.org/10.1007/11754602_1`
30. Régin, J.: A filtering algorithm for constraints of difference in CSPs. In: Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1. pp. 362–367 (1994), `http://www.aaai.org/Library/AAAI/1994/aaai94-055.php`
31. Régin, J.C.: Développement d'outils algorithmiques pour l'Intelligence Artificielle. Application à la chimie organique. Ph.D. thesis, Université Montpellier 2 (1995)
32. San Segundo, P., Rodriguez-Losada, D., Galan, R., Matia, F., Jimenez, A.: Exploiting CPU bit parallel operations to improve efficiency in search. In: Tools with Artificial Intelligence, 2007. ICTAI 2007. 19th IEEE International Conference on. vol. 1, pp. 53–59 (Oct 2007)
33. Sevegnani, M., Calder, M.: Bigraphs with sharing. Theoretical Computer Science 577(0), 43 – 73 (2015), `http://www.sciencedirect.com/science/article/pii/S0304397515001085`
34. Slater, N., Itzchack, R., Louzoun, Y.: Mid size cliques are more common in real world networks than triangles. Network Science 2, 387–402 (12 2014), `http://journals.cambridge.org/article_S2050124214000228`
35. Smith, B.M., Grant, S.A.: Modelling exceptionally hard constraint satisfaction problems. In: Smolka, G. (ed.) Principles and Practice of Constraint Programming-CP97, Lecture Notes in Computer Science, vol. 1330, pp. 182–195. Springer Berlin Heidelberg (1997), `http://dx.doi.org/10.1007/BFb0017439`
36. Solnon, C.: Alldifferent-based filtering for subgraph isomorphism. Artif. Intell. 174(12-13), 850–864 (2010), `http://dx.doi.org/10.1016/j.artint.2010.05.002`
37. Solnon, C., Damiand, G., de la Higuera, C., Janodet, J.C.: On the complexity of submap isomorphism and maximum common submap problems. Pattern Recognition 48(2), 302 – 316 (2015), `http://www.sciencedirect.com/science/article/pii/S0031320314002192`
38. Trienekens, H.W.: Parallel Branch and Bound Algorithms. Ph.D. thesis, Erasmus University Rotterdam (1990)
39. Ullmann, J.R.: An algorithm for subgraph isomorphism. Journal of the ACM (JACM) 23(1), 31–42 (1976)
40. Ullmann, J.R.: Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. J. Exp. Algorithmics 15, 1.6:1.1–1.6:1.64 (Feb 2011), `http://doi.acm.org/10.1145/1671970.1921702`
41. Zampelli, S., Deville, Y., Solnon, C.: Solving subgraph isomorphism problems with constraint programming. Constraints 15(3), 327–353 (2010), `http://dx.doi.org/10.1007/s10601-009-9074-3`