Sequential and Parallel Solution-Biased Search for Subgraph Algorithms*

Blair Archibald[1][0000−0003−3699−6658], Fraser Dunlop[2][0000−0002−4485−4871], Ruth Hoffmann[2][0000−0002−1011−5894], Ciaran McCreesh[1][0000−0002−6106−4871], Patrick Prosser[1][0000−0003−4460−6912], and James Trimble[1][0000−0001−7282−8745]

[1] University of Glasgow, Glasgow, Scotland
[2] University of St Andrews, St Andrews, Scotland
`ciaran.mccreesh@glasgow.ac.uk`

Abstract. The current state of the art in subgraph isomorphism solving involves using degree as a value-ordering heuristic to direct backtracking search. Such a search makes a heavy commitment to the first branching choice, which is often incorrect. To mitigate this, we introduce and evaluate a new approach, which we call "solution-biased search". By combining a slightly-random value-ordering heuristic, rapid restarts, and nogood recording, we design an algorithm which instead uses degree to direct the proportion of search effort spent in different subproblems. This increases performance by two orders of magnitude on satisfiable instances, whilst not affecting performance on unsatisfiable instances. This algorithm can also be parallelised in a very simple but effective way: across both satisfiable and unsatisfiable instances, we get a further speedup of over thirty from thirty-six cores, and over one hundred from ten distributed-memory hosts. Finally, we show that solution-biased search is also suitable for optimisation problems, by using it to improve two maximum common induced subgraph algorithms.

1 Introduction

The subgraph isomorphism problem is to decide whether a copy of a small "pattern" graph occurs inside a larger "target" graph. The problem is broadly applicable, arising in areas including bioinformatics [2], chemistry [46], computer vision [11, 49], law enforcement [8], model checking [47], malware detection [4], compilers [43, 1], pattern recognition [12], program similarity comparison [10], the design of mechanical locks [50], and graph databases [37].

Although the problem is NP-complete, by combining design techniques from artificial intelligence with careful algorithm engineering, modern subgraph isomorphism solvers can often produce exact solutions quickly even on graphs with thousands of vertices. The current single strongest subgraph isomorphism solver

* This work was supported by the Engineering and Physical Sciences Research Council (grant numbers EP/P026842/1, EP/M508056/1, and EP/N007565). This work used the Cirrus UK National Tier-2 HPC Service at EPCC (http://www.cirrus.ac.uk) funded by the University of Edinburgh and EPSRC (EP/P020267/1).

uses “highest degree first” as a value-ordering heuristic to direct a constraint programming style search [35, 25, 37]. This heuristic is much better than branching randomly, but is still far from perfect. To offset mistakes made by this heuristic, this paper proposes a new perspective on value-ordering: rather than defining a search order, we use degree to direct what *proportion* of the search effort should be spent in each subproblem. By combining rapid restarts and nogood recording, and introducing a small amount of randomness into the value-ordering heuristic, we make a state-of-the-art subgraph algorithm perform two orders of magnitude better on a large number of satisfiable instances, whilst performing worse only rarely on satisfiable instances, and never on unsatisfiable instances. This strategy is also effective in an optimisation setting, producing benefits in two maximum common induced subgraph algorithms.

This new form of search can also be parallelised, with a *much* simpler implementation than conventional work-stealing. By running many threads with different random seeds but the same restart schedule, and sharing nogoods only following restarts, we can achieve aggregate speedups [20] of thirty-one from a thirty-six core machine, or over one hundred by using ten such machines.

1.1 Background

The *non-induced subgraph isomorphism problem* is to find an injective mapping from the vertices of a *pattern* graph $\mathcal{P}$ to the vertices of a *target* graph $\mathcal{T}$, such that adjacent vertices in $\mathcal{P}$ are mapped to adjacent vertices in $\mathcal{T}$ (including that vertices with loops in $\mathcal{P}$ may only be mapped to vertices with loops in $\mathcal{T}$). The *induced* problem additionally requires that non-adjacent vertices are mapped to non-adjacent vertices. The *degree* of a vertex is the number of other vertices to which it is adjacent.

This paper looks at improving the Glasgow Subgraph Solver[3], which can solve both the non-induced and the induced subgraph isomorphism problems. The solver is very closely based upon the $k\downarrow$ algorithm of Hoffmann et al. [21] with $k = 0$, and we refer the reader to that paper for full technical details; that algorithm, in turn, is a simplification and re-engineering of an older Glasgow algorithm [35, 25]. Essentially, the solver is a dedicated forward-checking constraint programming implementation specifically for subgraph problems. It works with a model having a variable per pattern graph vertex, with domains ranging over the target graph vertices, and performs a backtracking search to map each pattern vertex to a target vertex whilst propagating adjacency and injectivity constraints (together with further implied constraints based upon degrees and paths). However, it uses specialised bit-parallel data structures and algorithms, and propagates constraints in a fixed order rather than using a queue.

1.2 Experimental Setup

Our experiments are performed on the EPCC Cirrus HPC facility, on systems with dual Intel Xeon E5-2695 v4 CPUs and 256GBytes RAM, running Centos

[3] https://github.com/ciaranm/glasgow-subgraph-solver/

7.3.1611. We use GCC 7.2.0 as the compiler. For parallelism, we use C++ native threads, and for distributed parallelism we also use the SGI MPT implementation of MPI. All timing measurements are steady-clock, and we use a deterministic pseudo-random number generator for reproducibility.

We use the dataset introduced by Kotthoff et al. [25] for evaluation. This dataset brings together a range of randomly-generated and application instance families from earlier papers:

BVG(r), M4D(r), and Rand are families of randomly generated graphs using different models (bounded degree, regular mesh, and uniform), where each pattern is a permuted random connected subgraph of the target (and so each instance is satisfiable) [9]. These benchmark instances are widely used, but have unusual properties and so broad conclusions should not be drawn based solely upon behaviour of these instances [37].

SF contains randomly generated scale-free graphs using a similar method [52].

LV consists of various kinds of graph gathered by Larrosa and Valiente [27] from the Stanford Graph Database. We include both the 50 small graphs, and the 50 large graphs.

Phase contains hand crafted instances that lie near the satisfiable / unsatisfiable phase transition [37].

PR contains graphs generated from segmented images, corresponding to a computer vision problem [49].

Images and Meshes contain graphs representing 2D segmented images and 3D object models, again representing a computer vision problem [11].

Other studies use a random selection of 200 of each of the instances from the "meshes" and "images" families because some earlier solvers find many of these instances extremely hard. We would like to have a larger number of satisfiable instances in our test set, and so we include all pattern / target pairs. This gives a total of 14,621 instances (rather than the original 5,725). At least 2,150 of these instances are known to be satisfiable for the non-induced problem, and at least 12,348 are unsatisfiable.

2 Improving Sequential Search

We begin with a set of baseline performance measurements. In the top two plots of Figure 1 we show the cumulative number of instances solved over time for the non-induced and induced problems respectively. We compare the Glasgow Subgraph Solver using depth-first search (DFS) and with the modifications described in the remainder of this paper (solution-biased search, SBS), the PathLAD variation of the LAD algorithm [48, 25], VF2 [9], RI [2], and VF3 [5] (which only supports the induced problem), in each case using the original implementation provided by the algorithm's authors. The plots show that our starting point comfortably beats PathLAD, VF2, VF3 and RI, except for very low choices of timeout. For each algorithm, the y value gives the cumulative number of instances which (individually) can be solved in no more than x milliseconds. The

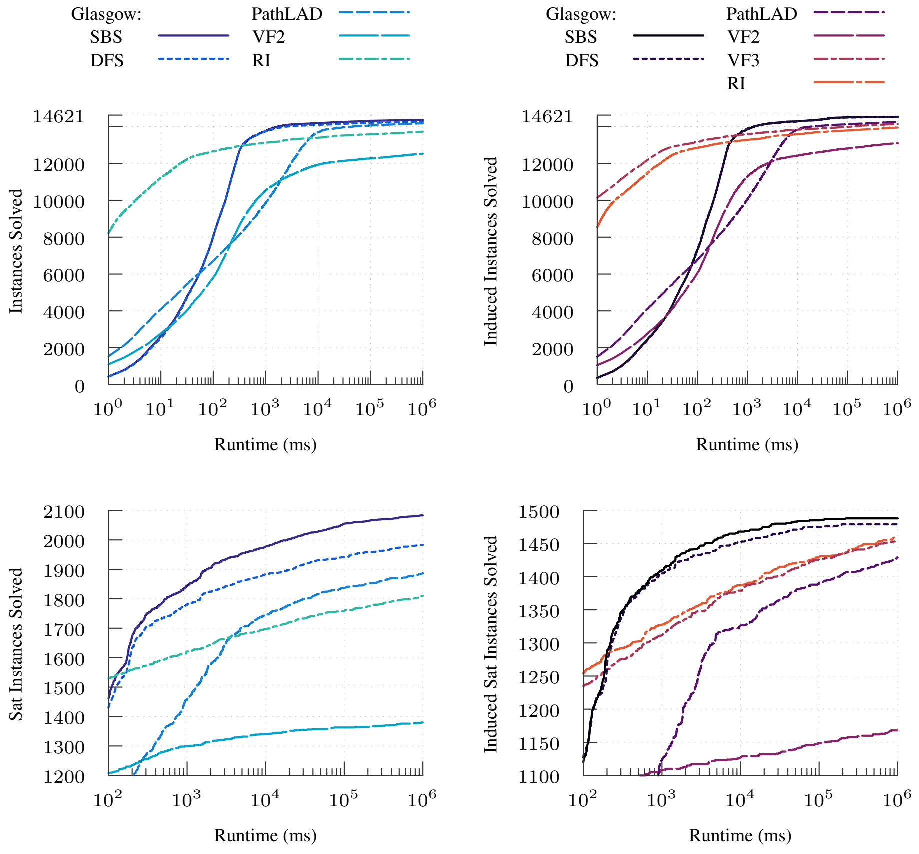

Fig. 1. On the top row, the cumulative number of instances solved over time, comparing the Glasgow Subgraph Solver (both in its basic form, and with the improvements introduced in the remainder of the paper) to other solvers, for the non-induced and induced problems. On the bottom row, the same, considering only satisfiable instances.

vertical distance between two lines therefore shows how many more instances can be solved by one solver than another, if every instance is run separately with the chosen x timeout. The horizontal distance shows how many times longer the per-instance timeout would need to be to allow the rightmost algorithm to succeed on y out of the 14,621 instances (bearing in mind that the two sets of y instances could be different), and gives a measure called *aggregate speedup* [20].

The dataset includes many instances which are extremely easy for a good solver, and so it can be hard to see the differences between the stronger solvers at higher runtimes. This paper focusses upon improving the performance on the remaining hard satisfiable instances, and so in the bottom two plots in Figure 1 (and in subsequent cumulative plots for sequential algorithms) we show only satisfiable instances, and use a reduced range on both axes. For the remainder

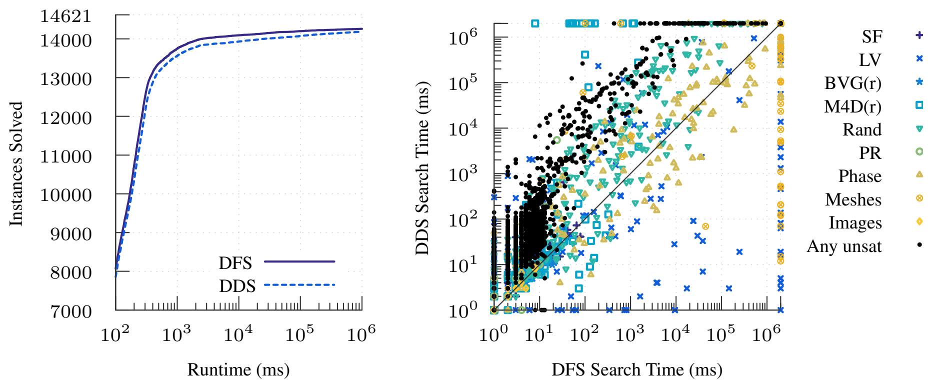

Fig. 2. Comparing depth-bounded discrepancy search (DDS) to depth-first backtracking search (DFS), both using degree as the value-ordering heuristic.

of this paper, we show only the non-induced problem, which tends to be harder; results with the induced variant are similar.

2.1 Discrepancy Searches

A *discrepancy* is where search goes against the advice of a value-ordering heuristic. Discrepancy searches [19, 24, 51, 23] are alternatives to backtracking search that initially search disallowing all discrepancies, and then retry search allowing an increasing number of discrepancies at each iteration until either a solution is found or unsatisfiability is proven. These schemes assume that value-ordering heuristics are usually reliable, and that most solutions can be found with only a small number of discrepancies. In such cases, the heavy commitment to early branching choices made by backtracking search can be extremely costly.

Figure 2 shows the effects of adding Walsh's [51] depth-bounded discrepancy search (DDS) to the solver (results with other discrepancy search variants are similar). On the scatter plot, each point represents the solving time for one instance—to avoid noise for easier instances, we measure only time spent during search, and exclude time spent in preprocessing and initialisation. Points below the $x - y$ diagonal are speedups, whilst points on the top and right axes represent instances which timed out after one thousand seconds with one algorithm, but not the other. For satisfiable instances, the different point styles show the different families, whilst all unsatisfiable instances are shown as dark dots. The points well below the diagonal line and along the right-hand axis on the scatter plot show that DDS can sometimes be extremely beneficial on satisfiable instances. However, on both unsatisfiable and most satisfiable instances, the overheads can be extremely large, and DDS is much worse in aggregate and is not a viable approach (even when only considering satisfiable instances). These large overheads are to be expected: discrepancy searches are aimed primarily at getting better feasible solutions in optimisation problems which are too large for

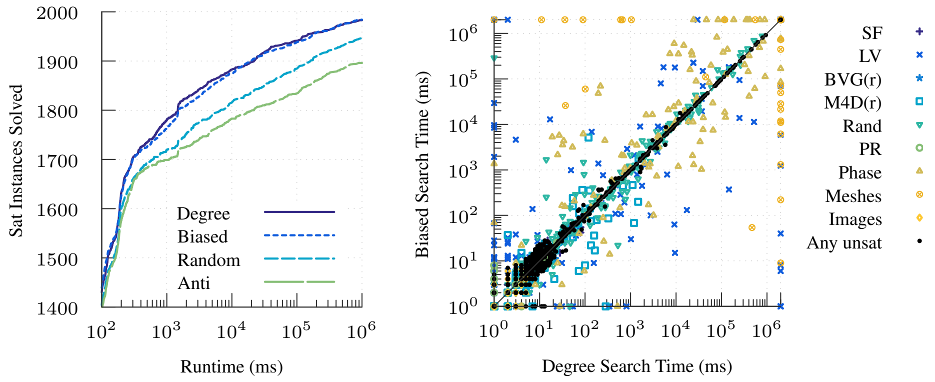

Fig. 3. On the left, the cumulative number of satisfiable instances solved over time, using four different value-ordering heuristics. On the right, an instance by instance comparison of the degree and biased heuristics on all instances. Points on the outer axes are timeouts, and point style shows instance family.

a proof of optimality to be a realistic prospect, and they are not well-suited for unsatisfiable decision problems. Despite this, the extremely large gains on some satisfiable instances confirm our suspicions that we should find an alternative to heavy-commitment backtracking search.

2.2 Value-Ordering Heuristics

Traditionally, value-ordering heuristics are designed to drive search towards the most promising region of the search space [14], or the most constrained [15], or the region with the highest solution density [44]. In subgraph isomorphism, this is done by selecting vertices from highest degree to lowest [37]. The left-hand plot of Figure 3 demonstrates that this is indeed a good choice: the **Degree** heuristic's line shows much better performance on satisfiable instances than the **Random** (branch randomly) or **Anti** (branch from lowest degree to highest) heuristic lines. Meanwhile, on unsatisfiable instances, the value-ordering heuristic has no effect on performance, because a complete search must be performed.

But what happens if our value-ordering heuristic has to choose between mapping a pattern vertex to one of, for example, three target vertices of degree ten, two vertices of degree nine, or five of degree two? When driving conventional backtracking search, the degree heuristic would pick one of the vertices of degree ten, and we would commit all of our search effort to the exponentially large search tree underneath it, not considering any other choice until this tree has been fully explored and eliminated. We will show that this is not a wise choice, and that instead, we should *commit equal search effort* to each of the three subproblems found by mapping to vertices of degree ten. And similarly, should we be certain that a vertex of degree ten is so much better than a vertex of degree nine that we should commit no effort to degree nine vertices until all the degree ten subproblems have been explored? Or might it be better to commit, say, twice as much effort to each degree ten subproblems as to each degree nine subproblems, and only a very small amount of effort to the degree two subproblems? To test this hypothesis, we will now introduce a new alternative to backtracking search, which we call *solution-biased search*. This search is made up of three components: a new slightly-random value-ordering heuristic, rapid restarts, and nogood recording. The aim is to perform a complete search, but spending proportionally more time in parts of the search tree that are preferred by the value-ordering heuristic.

2.3 Biased Value-Ordering

We first define a new **Biased** value-ordering heuristic, as follows. When branching, we select a vertex v' from the chosen domain D_v with probability

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

This heuristic is now equally likely to pick between vertices of equal degree, is twice as likely to pick a vertex of degree d as one of degree $d-1$, and is over a thousand times more likely to pick a vertex of degree d than degree $d-10$.

Figure 3 confirms that this heuristic, when used with backtracking search, will solve close to the same number of instances as the degree heuristic would for any given choice of timeout. In other words, we can introduce an element of randomness into the degree value-ordering heuristic without adversely affecting its performance *in aggregate*. The right-hand plot gives a detailed comparison. It shows that despite the aggregate performance being similar, on a case by case basis, the two heuristics can make a large difference to the performance for individual satisfiable instances. This justifies our belief that although degree is a good heuristic, we should perhaps not commit heavily to a single vertex of highest degree, but also consider vertices of the same or similar degree.

2.4 Restarting Search and Nogood Recording

Having introduced a new value-ordering heuristic, we must now also move away from depth-first backtracking search. We do this by using *restarts* and *nogood recording*. The general idea is to perform a certain amount of search, and then if no solution has been found (and unsatisfiability has not been proven), to abandon search and restart from the beginning. Such an approach can only be beneficial if something changes after restarting—in a constraint programming setting, this is usually the variable-ordering heuristic [28, 13, 29, 16]. In this paper, we instead rely upon randomness in our new *value*-ordering heuristic, and continue to use smallest domain first with static tiebreaking for variable-ordering.[4] Using

[4] It may be possible to further improve the solver by also introducing randomness or some form of learning into its variable-ordering heuristic. However, simultaneously

restarts on value-ordering heuristics is uncommon (although Razgon et al. [45] look at learning value-ordering heuristics from restarts, Chu et al. [6] use a similar scheme in the context of parallel search, and an early approach by Gomes et al. [17] does so in an optimisation context).

Preliminary experiments directed us to use the Luby scheme [33] to determine when to restart. Following convention, we multiply each item in the Luby sequence by a constant—we used the SMAC automatic parameter tuner [22] to select the value 660.

To avoid exploring portions of the search space that we have already visited, every time we restart, we add new constraints to the problem which eliminate already-explored subtrees—such a constraint is called a *nogood*. We generate simple decision nogoods. That is, upon backtracking due to a decision to restart, we post a nogood of the form $(v \mapsto v') \wedge (w \mapsto w') \wedge (x \mapsto x') \Rightarrow \bot$ for every branch to the left of the current (incomplete) branch at every level of the search tree, and when we first make a decision to restart before backtracking, we post a similar nogood eliminating the entire subtree explored. We use the two watched literals technique [42] to propagate stored nogoods. This has two benefits: the propagation complexity does not particularly depend upon the number of stored nogoods, and it does not require any work upon backtracking. Other more sophisticated nogood generation and propagation schemes exist [29, 16], but these are not helpful in this setting (our solver does not maintain arc consistency or use a propagation queue).

2.5 Solution-Biased Search in Practice

In Figure 4 we show the effects of adding restarts and nogood recording to the algorithm. With restarts and nogood recording (random search with restarts, RSR), the random value-ordering heuristic comfortably beats the degree strategy with depth-first search. In other words, although having a good value-ordering heuristic is beneficial, introducing randomness into the search is better, if it is done alongside a mechanism to avoid heavy commitment to any particular random choice. However, the biased heuristic together with restarts (solution-biased search, SBS) is better still—that is, if we are introducing restarts, then it is better to add a small amount of randomness to a tailored heuristic than it is to throw away the heuristic altogether. Indeed, the original algorithm can solve 1983 satisfiable instances by 909 seconds, whilst the biased and random restarting algorithms require only 12 seconds and 35 seconds respectively to solve the same number.

In the more detailed view in the right-hand plot of Figure 4, comparing the original algorithm to solution-biased search, all of the unsatisfiable instances are very close to the $x - y$ diagonal, showing that their performance is nearly unchanged. On the other hand, there are large numbers of satisfiable instances

introducing a second change would considerably complicate the empirical analysis. Additionally, the solver's current hand-crafted variable-ordering heuristics already beat adaptive heuristics like impact or activity-based search.

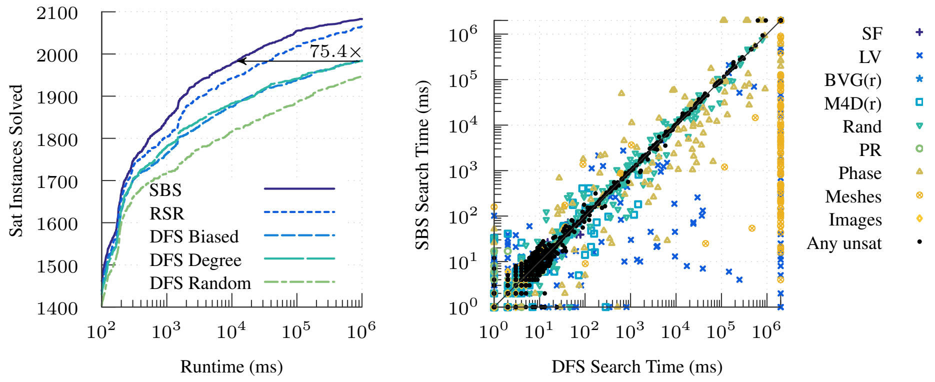

Fig. 4. On the left, the number of satisfiable instances solved over time, comparing solution-biased search (SBS), random search with restarts (RSR), and the three value-ordering heuristics with conventional depth-first search. To the right, a comparison between the original algorithm and solution-biased search.

well below the diagonal line, indicating large speedups. Better yet, there are only a handful of satisfiable instances that are more than a factor of ten times worse. In other words, as well as improving performance, we have made up most of the consistency we lost by introducing randomness.

As we might expect, these properties do not hold if any of the combination of changes are disabled. In the left-hand plot of Figure 5, we see large slowdowns on unsatisfiable instances when disabling nogood recording, and on the right-hand plot we see many more satisfiable instances above the $x - y$ diagonal when using the random value-ordering heuristic as opposed to the degree-biased heuristic.

2.6 Solution-Biased Search in Theory

Although we have shown that it provides good results, we have yet to justify where the biased formula comes from, or indeed why we call this approach "solution-biased". Our goal is to use biased randomness in a value-ordering heuristic to spend time in subproblems proportional to an estimate of their solution density [44]. Such an approach is better than committing entirely to the area of maximum solution density because estimators only give a probability—although we may estimate that one subtree has twice the solution density of another, in reality the "better" subtree may contain no solutions at all.

To estimate solution density, we need an estimate of how big different subproblems are likely to be, and of how many solutions each subproblem is likely to contain. Of course, obtaining exact (or even approximate) values for these figures is at least as hard as solving the problem in its entirety, but we may obtain usable approximations. For pairs of Erdős-Rényi random graphs with large solution counts (i.e. chosen from within the "easy satisfiable" region [37]), we can observe a linear relationship between subproblem size and number of solutions. Thus,

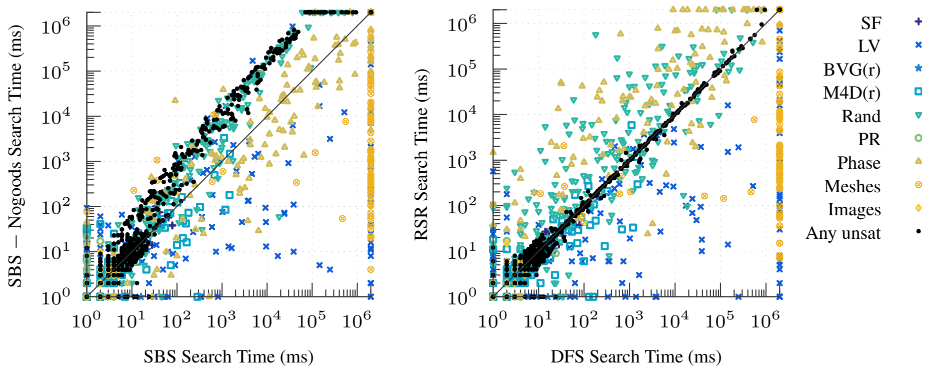

Fig. 5. On the left, not using nogood recording introduces slowdowns, particularly on unsatisfiable instances. On the right, using a random value-ordering gives much worse performance on many satisfiable instances.

for graphs from this distribution, we need only an estimator of subproblem size. Measurements also suggest that, for pairs of Erdős-Rényi graphs, subproblems under a target vertex of degree d tend to contain a small constant times more search nodes than those under a target vertex of degree $d-1$. This empirical analysis suggests that an estimator that is exponential in d will give our method the desired behaviour, at least for Erdős-Rényi graphs. We expect it may be possible to derive better estimators for particular input classes, although over the full range of problem instances, we have verified that exponential estimators substantially outperform polynomial and factorial weightings.

3 Parallel Search

Exploiting multiple cores to speed up constraint programming solvers remains an active area of research, with no universally perfect solution being available. Four of the more common approaches are based upon decompositions [26, 34], work-stealing [39, 6, 35, 20], parallel discrepancy searches [40, 41], and algorithm portfolios [32]. Decomposition approaches are unsuitable for decision problems, or problems where we have good value-ordering heuristics, because the decomposition interferes strongly with the shape of the search tree [34]. Work-stealing, traditionally, also interferes with value-ordering [36], although specially designed exceptions exist [6, 35, 20]. However, these have very complicated implementations. Parallel discrepancy searches are aware of value-ordering heuristics, but have other limitations: they struggles on search trees with heavy filtering, and rely upon inner search tree nodes being orders of magnitude less expensive to process than leaf nodes. Portfolios, meanwhile, typically rely upon running multiple models or heuristics simultaneously, and selecting whichever finishes first, whereas here we have a known good model and set of heuristics.

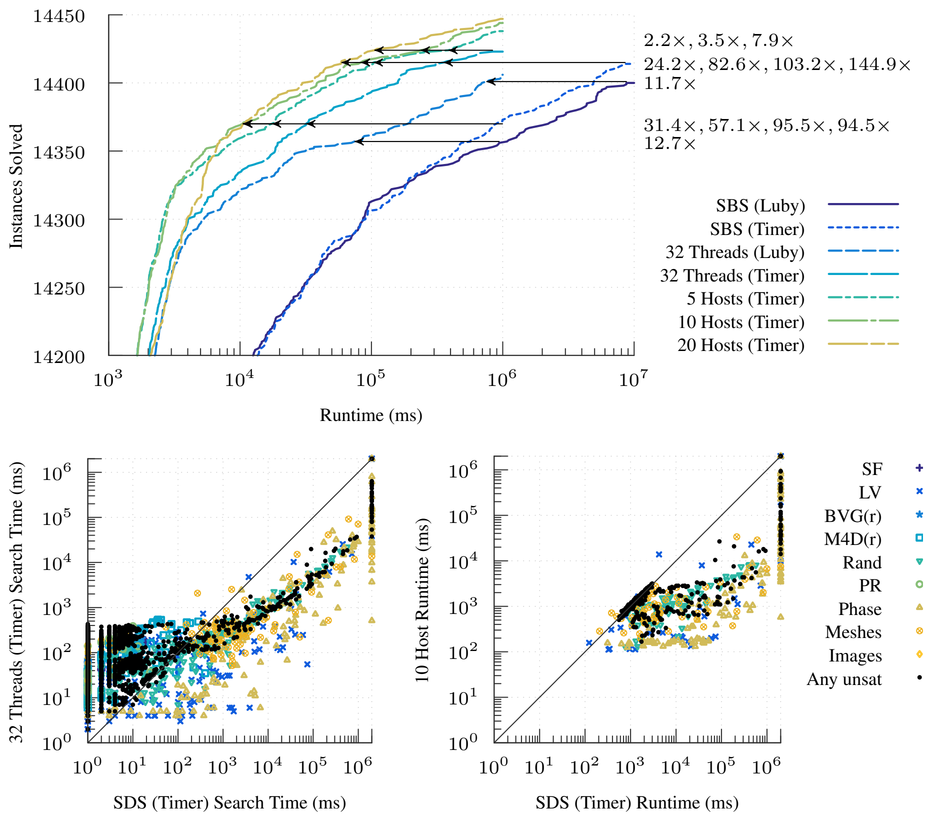

Fig. 6. Above, the cumulative number of instances solved over time, comparing the sequential algorithm to results using 32 threads on a single machine, and using five, ten or twenty distributed memory hosts. Below, instance by instance comparisons.

3.1 Shared Memory Parallelism

Solution-biased search allows for a much simpler parallel implementation. We create a number of threads, and give each thread its own random seed; otherwise each thread performs the same sequential search. Threads synchronise on restarts, a simple barrier causing each thread to wait for every other thread to also restart. Nogoods from all threads are then gathered and combined before search resumes, now with a larger set of nogoods than in a sequential run. Finally, whenever any single thread terminates, either due to having found a solution or proved unsatisfiability, then every other thread may immediately terminate.

This technique requires only limited changes to the top level search driver, and none whatsoever to the main recursive search algorithm. Notably, it does not require any locking or communication during the recursive search, aside from a single atomic boolean flag to assist early termination. A number of factors combine to make this approach feasible:

– Each thread will be run with the same restart schedule, and so will spend approximately the same amount of time between restarts. Because the only synchronisation between threads is at a restart, we expect threads to be busy doing search. (This is in contrast to an alternative method for paralellising restarting search [7], which packs together successive sequence values to produce a balanced workload.) This approach therefore avoids the irregular task size issues which usually arise in parallel combinatorial search.
– Sequentially, on non-trivial instances the algorithm will restart often (many tens of thousands, for instances that reach the thousand second timeout).
– Because the search trees we explore are exponentially large, the randomness in the value-ordering heuristic is sufficient to ensure that most of the time, threads are exploring different parts of the search tree.
– The gathering of nogoods to describe the work done so far provides an alternative to requiring either a specific mechanism to allocate work, or expensive synchronisation between threads. Notably, this completely bypasses the typical difficulties of sharing all learned nogoods in learning solvers [18].
– If sometimes threads do happen to explore part of the same subproblems, this is not a problem: if the instance is satisfiable, either thread might find a solution first, and if the instance is unsatisfiable, we merely introduce some redundancy into the proof.[5] The combination of rapid restarts and nogood recording is enough to ensure that this is only a small overhead.

Figure 6 shows how this scheme performs in practice. Sequentially, we can solve 14,357 instances within the thousand second timeout, with the last instance being solved at 939.0 seconds. Using thirty-six threads on machines with two eighteen core processors, we can solve 14,357 instances with a timeout of only 74.2 seconds, giving an aggregate speedup [20] of 12.7.

Closer inspection of the results reveals that with this many threads, a considerable proportion of the overall search time is spent with threads waiting at the barrier for synchronisation. This is because the time taken to carry out search until 660 backtracks are encountered is only *roughly* a constant (in practice it usually varies by around a factor of two). Furthermore, the Luby sequence includes occasional large multipliers, and if unsatisfiability is proved during towards the end of one of these runs, each thread will end up duplicating a large amount of work.

Because we are using nogood recording, an alternative approach is possible. Rather than using the Luby sequence for restarts, we could restart after either a constant number of backtracks, or simply after a certain time interval has passed—this bounds the maximum possible idle time that threads could spend at a barrier. Figure 6 also shows the effects of restarting every 100ms. Sequentially, this approach is slightly better than using the Luby sequence, being able to solve 14,370 instances, with the last at 996.4 seconds. With thirty-six threads, solving this many instances takes 31.8 seconds, giving an aggregate speedup of 31.4.

[5] For solving a counting or enumeration problem, matters become slightly more complicated, but not devastatingly so.

It is important to emphasise that there is no expectation of a linear speedup from this approach [30, 31, 3], particularly for satisfiable instances. Due to the biased random nature of the value-ordering heuristic, one of our extra threads may "get lucky" and find a solution much quicker than we would sequentially, leading to a superlinear speedup. Conversely, our extra threads may contribute nothing to finding a solution—or worse, due to new nogoods altering the choice made by the random branching heuristic, we could even get an absolute slowdown. We can see both of these effects in the bottom left plot of Figure 6: although we achieve roughly a linear speedup on unsatisfiable instances, satisfiable instances show much greater variability. However, this approach does at least mirror our intuition of allocating search effort in proportion to where the value-ordering heuristic believes it will be most fruitful, and so we should not be too surprised that we see roughly a linear speedup on average on harder instances.

Additionally, for easy instances, most of the algorithm's execution time is spent in a preprocessing phase. We have not parallelised this, which is why our results are poor below the one second mark.

3.2 Distributed Memory Parallelism

To further test the scalability of this technique, we also used MPI to implement a distributed-memory parallelism layer on top of the threaded layer. In contrast to the huge difficulties of implementing work-stealing in a distributed memory setting, this required only the addition of two MPI calls: an "all gather" operation to communicate nogoods, and a "gather" to collect and combine the results of each host. Termination, meanwhile, was handled by posting an empty nogood (and so termination can only occur on a restart).

Figure 6 also shows the results of these experiments, using five, ten, and twenty hosts. Because each host has two CPU sockets, the five host results use ten MPI ranks, each with eighteen threads, and the ten and twenty host results use twenty and forty MPI ranks respectively. The supercomputing service we use is not designed for huge numbers of very short problems, and so we ran only instances whose sequential runtime was at least one second; for the sake of plotting results, we treat skipped instances as taking one second. Due to the job launcher used, it is also not possible to accurately measure "total" runtime including startup costs, and so instead we report the runtime of the rank zero process—this figure is somewhat optimistic for very easy instances. With these caveats in mind, when seeing how long a timeout is needed to solve any 14,370 instances, we get aggregate speedups of 57.1 from five hosts and 95.5 from ten hosts over a sequential baseline; using twenty hosts is slightly slower, due to increased overheads. Finally, if we look at harder instances, by allow a longer sequential timeout, we can solve 14,415 instances sequentially with the last at 8,549 seconds; at this difficulty level, we achieve aggregate speedups of 82.6, 103.2, and 144.9 from five, ten and twenty hosts.

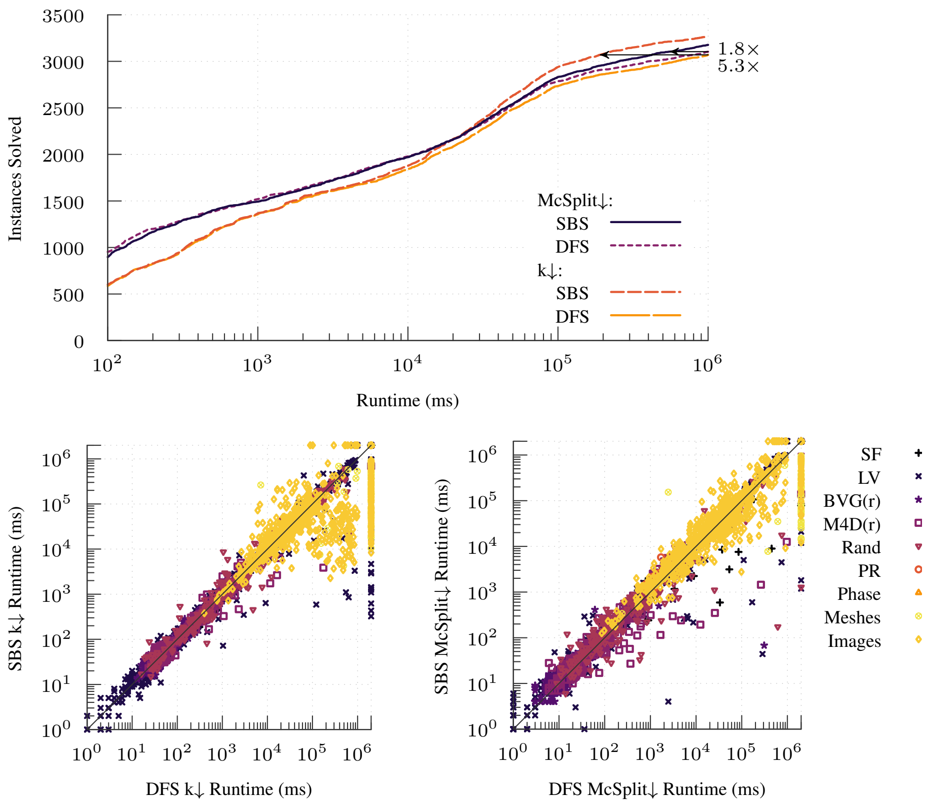

Fig. 7. Above, the cumulative number of maximum common induced subgraph instances solved by k↓ and McSplit↓ over time, with the two forms of search. Below, comparing k↓ (left) and McSplit↓ (right) on an instance by instance basis.

4 Maximum Common Subgraph Algorithms

Having looked at subgraph isomorphism in detail, we now briefly discuss the maximum common induced subgraph problem, to see whether our new approach to search has more general applicability. Two recent algorithms for this problem also make use of backtracking search with degree as a value-ordering heuristic. The k↓ algorithm [21] attempts to solve the problem by first trying to solve the induced subgraph isomorphism problem, and then if that fails, retries allowing a single unmatched vertex (and thus using weaker invariants), and so on. Due to its similarity to the Glasgow Subgraph Solver, we can introduce the same bias and restart strategy.

Meanwhile, the McSplit↓ algorithm [38] uses a constraint programming style search, but with special propagators and backtrackable data structures that exploit special properties of the problem. The unconventional domain store used by McSplit↓ precludes the use of arbitrary unit propagation, and so when introducing restarts, we cannot propagate using nogoods. Instead, we can only detect when we are inside an already-visited branch. We must therefore use the one watched literal scheme instead, and we also introduce a basic subsumption scheme to prune redundant clauses.

Performance results from these two modified algorithms, using the same families of instances as in the previous section, are shown in Figure 7. Although we have moved from a decision problem to an optimisation problem, the same changes remain clearly beneficial. For the k↓ algorithm, the change has a minimal effect on many instances (typically, where the $k = 0$ subproblem is unsatisfiable and hard, and the $k = 1$ subproblem is satisfiable and easy), but gives large benefits on many more instances than it penalises: it is over an order of magnitude better on over three hundred instances, whilst being an order of magnitude worse on only seven.

With McSplit↓, the inability to use two watched literals means that in many cases we introduce a small slowdown. However, the overall pattern is the same: when introducing restarts and a biased value ordering heuristic, it is much more common to see a large speedup than a large slowdown.

5 Conclusion and Future Work

The conventional view of value-ordering heuristics is that they define a search order. We have proposed an alternative perspective, where value-orderings define a weighting specifying how much search effort should be put into different subproblems, based upon a rough estimate of solution densities. We have also shown how to turn this perspective into an algorithm, by combining a biased random value-ordering heuristic with rapid restarts and nogood recording. This combination of techniques gives us, for the first time, a practical alternative to backtracking search where we have a strong *value*-ordering heuristic, and where we care both about satisfiable *and* unsatisfiable instances.

A further benefit is the ease with which such a search can be parallelised. By having each thread carry out the same search with a different random seed, and sharing nogoods only on restarts, we remove the need for intrusive changes to the core search algorithm, eliminate the irregularity problem, and still respect the advice of the value-ordering heuristic.

We believe that these technique are broadly applicable, beyond subgraph algorithms, and we intend to implement them in a full constraint programming solver. We are also interested in making better use of statistical knowledge (either *a priori* or learned during search) to further refine the biased randomisation process. And finally, we are trying hard to work out whether our new perspective also has some relevance to *variable* ordering heuristics.

Acknowledgements

The authors would like to thank Özgür Akgün, Chris Jefferson, Sonja Kraiczy, Christophe Lecoutre, Christine Solnon, and Craig Reilly for their comments.

References

1. Blindell, G.H., Lozano, R.C., Carlsson, M., Schulte, C.: Modeling universal instruction selection. In: Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings. pp. 609–626 (2015). https://doi.org/10.1007/978-3-319-23219-5_42
2. Bonnici, V., Giugno, R., Pulvirenti, A., Shasha, D.E., Ferro, A.: A subgraph isomorphism algorithm and its application to biochemical data. BMC Bioinformatics **14**(S-7), S13 (2013)
3. de Bruin, A., Kindervater, G.A.P., Trienekens, H.W.J.M.: Asynchronous parallel branch and bound and anomalies. In: Parallel Algorithms for Irregularly Structured Problems, Second International Workshop, IRREGULAR '95, Lyon, France, September 4-6, 1995, Proceedings. pp. 363–377 (1995). https://doi.org/10.1007/3-540-60321-2_29
4. Bruschi, D., Martignoni, L., Monga, M.: Detecting self-mutating malware using control-flow graph matching. In: Detection of Intrusions and Malware & Vulnerability Assessment, Third International Conference, DIMVA 2006, Berlin, Germany, July 13-14, 2006, Proceedings. pp. 129–143 (2006). https://doi.org/10.1007/11790754_8
5. Carletti, V., Foggia, P., Saggese, A., Vento, M.: Introducing VF3: A new algorithm for subgraph isomorphism. In: Graph-Based Representations in Pattern Recognition - 11th IAPR-TC-15 International Workshop, GbRPR 2017, Anacapri, Italy, May 16-18, 2017, Proceedings. pp. 128–139 (2017)
6. Chu, G., Schulte, C., Stuckey, P.J.: Confidence-based work stealing in parallel constraint programming. In: Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings. pp. 226–241 (2009)
7. Ciré, A.A., Kadioglu, S., Sellmann, M.: Parallel restarted search. In: Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada. pp. 842–848 (2014)
8. Coffman, T., Greenblatt, S., Marcus, S.: Graph-based technologies for intelligence analysis. Commun. ACM **47**(3), 45–47 (2004). https://doi.org/10.1145/971617.971643
9. 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)
10. Dalla Preda, M., Vidali, V.: Abstract similarity analysis. Electronic Notes in Theoretical Computer Science **331**, 87 – 99 (2017). https://doi.org/10.1016/j.entcs.2017.02.006, proceedings of the Sixth Workshop on Numerical and Symbolic Abstract Domains (NSAD 2016)
11. Damiand, G., Solnon, C., de la Higuera, C., Janodet, J., Samuel, É.: Polynomial algorithms for subisomorphism of nd open combinatorial maps. Computer Vision and Image Understanding **115**(7), 996–1010 (2011)
12. Foggia, P., Percannella, G., Vento, M.: Graph matching and learning in pattern recognition in the last 10 years. IJPRAI **28**(1) (2014). https://doi.org/10.1142/S0218001414500013
13. Gay, S., Hartert, R., Lecoutre, C., Schaus, P.: Conflict ordering search for scheduling problems. In: Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings. pp. 140–148 (2015)

14. Geelen, P.A.: Dual viewpoint heuristics for binary constraint satisfaction problems. In: ECAI. pp. 31–35 (1992)
15. Gent, I.P., MacIntyre, E., Prosser, P., Walsh, T.: The constrainedness of search. In: Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, USA, August 4-8, 1996, Volume 1. pp. 246–252 (1996)
16. Glorian, G., Boussemart, F., Lagniez, J., Lecoutre, C., Mazure, B.: Combining nogoods in restart-based search. In: Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings. pp. 129–138 (2017)
17. Gomes, C.P., Selman, B., Kautz, H.A.: Boosting combinatorial search through randomization. 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. 431–437 (1998)
18. Hamadi, Y., Jabbour, S., Sais, L.: Control-based clause sharing in parallel SAT solving. In: Hamadi, Y., Monfroy, E., Saubion, F. (eds.) Autonomous Search, pp. 245–267. Springer (2012). https://doi.org/10.1007/978-3-642-21434-9_10, https://doi.org/10.1007/978-3-642-21434-9_10
19. Harvey, W.D., Ginsberg, M.L.: Limited discrepancy search. In: Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes. pp. 607–615 (1995)
20. Hoffmann, R., McCreesh, C., Ndiaye, S.N., Prosser, P., Reilly, C., Solnon, C., Trimble, J.: Observations from parallelising three maximum common (connected) subgraph algorithms. In: Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 15th International Conference, CPAIOR 2018, Delft, The Netherlands, June 26-29, 2018, Proceedings. pp. 298–315 (2018). https://doi.org/10.1007/978-3-319-93031-2_22
21. Hoffmann, R., McCreesh, C., Reilly, C.: Between subgraph isomorphism and maximum common subgraph. In: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA. pp. 3907–3914 (2017)
22. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Learning and Intelligent Optimization - 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers. pp. 507–523 (2011)
23. Karoui, W., Huguet, M., Lopez, P., Naanaa, W.: YIELDS: A yet improved limited discrepancy search for csps. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 4th International Conference, CPAIOR 2007, Brussels, Belgium, May 23-26, 2007, Proceedings. pp. 99–111 (2007)
24. Korf, R.E.: Improved limited discrepancy search. In: Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, August 4-8, 1996, Volume 1. pp. 286–291 (1996)
25. Kotthoff, L., McCreesh, C., Solnon, C.: Portfolios of subgraph isomorphism algorithms. In: Learning and Intelligent Optimization - 10th International Conference, LION 10, Ischia, Italy, May 29 - June 1, 2016, Revised Selected Papers. pp. 107–122 (2016)
26. Kotthoff, L., Moore, N.C.A.: Distributed solving through model splitting. CoRR **abs/1008.4328** (2010)

27. Larrosa, J., Valiente, G.: Constraint satisfaction algorithms for graph pattern matching. Mathematical Structures in Computer Science **12**(4), 403–422 (2002)
28. Lecoutre, C., Sais, L., Tabary, S., Vidal, V.: Recording and minimizing nogoods from restarts. JSAT **1**(3-4), 147–167 (2007)
29. Lee, J.H.M., Schulte, C., Zhu, Z.: Increasing nogoods in restart-based search. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA. pp. 3426–3433 (2016)
30. Li, G., Wah, B.W.: How to cope with anomalies in parallel approximate branch-and-bound algorithms. In: Proceedings of the National Conference on Artificial Intelligence. Austin, TX, August 6-10, 1984. pp. 212–215 (1984)
31. Li, G., Wah, B.W.: Coping with anomalies in parallel branch-and-bound algorithms. IEEE Trans. Computers **35**(6), 568–573 (1986). https://doi.org/10.1109/TC.1986.5009434
32. Lindauer, M.T., Hoos, H.H., Hutter, F.: From sequential algorithm selection to parallel portfolio selection. In: Learning and Intelligent Optimization - 9th International Conference, LION 9, Lille, France, January 12-15, 2015. Revised Selected Papers. pp. 1–16 (2015). https://doi.org/10.1007/978-3-319-19084-6_1
33. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. Inf. Process. Lett. **47**(4), 173–180 (1993)
34. Malapert, A., Régin, J., Rezgui, M.: Embarrassingly parallel search in constraint programming. J. Artif. Intell. Res. **57**, 421–464 (2016). https://doi.org/10.1613/jair.5247
35. McCreesh, C., Prosser, P.: A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs. In: Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings. pp. 295–312 (2015)
36. McCreesh, C., Prosser, P.: The shape of the search tree for the maximum clique problem and the implications for parallel branch and bound. TOPC **2**(1), 8:1–8:27 (2015). https://doi.org/10.1145/2742359
37. McCreesh, C., Prosser, P., Solnon, C., Trimble, J.: When subgraph isomorphism is really hard, and why this matters for graph databases. J. Artif. Intell. Res. **61**, 723–759 (2018)
38. McCreesh, C., Prosser, P., Trimble, J.: A partitioning algorithm for maximum common subgraph problems. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017. pp. 712–719 (2017)
39. Michel, L., See, A., Van Hentenryck, P.: Parallelizing constraint programs transparently. In: Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings. pp. 514–528 (2007). https://doi.org/10.1007/978-3-540-74970-7_37
40. Moisan, T., Gaudreault, J., Quimper, C.: Parallel discrepancy-based search. In: Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings. pp. 30–46 (2013). https://doi.org/10.1007/978-3-642-40627-0_6
41. Moisan, T., Quimper, C., Gaudreault, J.: Parallel depth-bounded discrepancy search. In: Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings. pp. 377–393 (2014). https://doi.org/10.1007/978-3-319-07046-9_27
42. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001. pp. 530–535 (2001)

43. Murray, A.C., Franke, B.: Compiling for automatically generated instruction set extensions. In: 10th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2012, San Jose, CA, USA, March 31 - April 04, 2012. pp. 13–22 (2012). https://doi.org/10.1145/2259016.2259019
44. Pesant, G., Quimper, C., Zanarini, A.: Counting-based search: Branching heuristics for constraint satisfaction problems. J. Artif. Intell. Res. **43**, 173–210 (2012). https://doi.org/10.1613/jair.3463
45. Razgon, M., O'Sullivan, B., Provan, G.M.: Search ordering heuristics for restarts-based constraint solving. In: Proceedings of the Twentieth International Florida Artificial Intelligence Research Society Conference, May 7-9, 2007, Key West, Florida, USA. pp. 182–183 (2007)
46. Régin, J.: Développement d'outils algorithmiques pour l'Intelligence Artificielle. Application à la chimie organique. Ph.D. thesis, Université Montpellier 2 (1995)
47. Sevegnani, M., Calder, M.: Bigraphs with sharing. Theor. Comput. Sci. **577**, 43–73 (2015). https://doi.org/10.1016/j.tcs.2015.02.011
48. Solnon, C.: Alldifferent-based filtering for subgraph isomorphism. Artif. Intell. **174**(12-13), 850–864 (2010)
49. Solnon, C., Damiand, G., de la Higuera, C., Janodet, J.: On the complexity of submap isomorphism and maximum common submap problems. Pattern Recognition **48**(2), 302–316 (2015)
50. Vömel, C., de Lorenzi, F., Beer, S., Fuchs, E.: The secret life of keys: On the calculation of mechanical lock systems. SIAM Review **59**(2), 393–422 (2017). https://doi.org/10.1137/15M1030054
51. Walsh, T.: Depth-bounded discrepancy search. In: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes. pp. 1388–1395 (1997)
52. Zampelli, S., Deville, Y., Solnon, C.: Solving subgraph isomorphism problems with constraint programming. Constraints **15**(3), 327–353 (2010)