Observations from Parallelising Three Maximum Common (Connected) Subgraph Algorithms

Ruth Hoffmann[1], Ciaran McCreesh[2](✉), Samba Ndojh Ndiaye[3], Patrick Prosser[2], Craig Reilly[2], Christine Solnon[4], and James Trimble[2]

[1] University of St Andrews, St Andrews, UK
[2] University of Glasgow, Glasgow, Scotland
ciaran.mccreesh@glasgow.ac.uk
[3] Université Lyon 1, LIRIS, UMR5205, 69621 Villeurbanne, France
[4] INSA-Lyon, LIRIS, UMR5205, 69621 Villeurbanne, France

Abstract. We discuss our experiences adapting three recent algorithms for maximum common (connected) subgraph problems to exploit multi-core parallelism. These algorithms do not easily lend themselves to parallel search, as the search trees are extremely irregular, making balanced work distribution hard, and runtimes are very sensitive to value-ordering heuristic behaviour. Nonetheless, our results show that each algorithm can be parallelised successfully, with the threaded algorithms we create being clearly better than the sequential ones. We then look in more detail at the results, and discuss how speedups should be measured for this kind of algorithm. Because of the difficulty in quantifying an average speedup when so-called anomalous speedups (superlinear and sublinear) are common, we propose a new measure called *aggregate speedup*.

1 Introduction

Finding a maximum common subgraph is the key step in measuring the similarity or difference between two graphs [3,12,19]. Because of this, maximum common subgraph problems frequently arise in biology and chemistry [10,14,33] where graphs represent molecules or reactions, and also in computer vision [5,7], computer-aided manufacturing [23], the analysis of programs and malware [13,31], crisis management [8], and social network analysis [11].

A *subgraph isomorphism* is an injective mapping from a *pattern* graph to a *target* graph which *preserves adjacency*—that is, it maps adjacent vertices to adjacent vertices. The isomorphism is *induced* if additionally it maps non-adjacent vertices to non-adjacent vertices, preserving non-adjacency as well.

C. McCreesh, P. Prosser, C. Reilly and J. Trimble—This work was supported by the Engineering and Physical Sciences Research Council [grant numbers EP/K503058/1, EP/M508056, and EP/P026842/1].

S. N. Ndiaye and C. Solnon—This work was supported by the ANR project SoLStiCe (ANR-13-BS02-0002-01).

W.-J. van Hoeve (Ed.): CPAIOR 2018, LNCS 10848, pp. 298–315, 2018.
https://doi.org/10.1007/978-3-319-93031-2_22

When working with labelled graphs, a subgraph isomorphism must preserve labels, and on directed graphs, it must preserve orientation. A *common induced subgraph* of two graphs G and H is a pair of induced subgraph isomorphisms from a pattern graph P, one to G and one to H. A *maximum common induced subgraph* is one with as many vertices as possible. (The *maximum common partial subgraph* problem is non-induced, with as many edges as possible; this paper discusses only induced problems.) A common variant of the problem requires a largest *connected* subgraph [10,23,33,36].

Although both the connected and non-connected variants are NP-hard, recently progress has been made towards solving the problem in practice. This paper looks at three branch and bound algorithms for maximum common (connected) induced subgraph problems, each of which is the state of the art for certain classes of instance. We discuss our experiences in adding parallel tree-search to these three algorithms. In each case, our results show that the parallel version of the algorithm is clearly better than the sequential version, although a closer look at the results shows many nuances. Thus this paper focusses primarily on presenting and interpreting the experimental data, rather than heavy implementation details, in the hopes that the lessons we learned are helpful to other practitioners—in particular, we introduce a new measure called *aggregate speedup* which is suitable for determining speedups for decision problems or optimisation problems where anomalous speedups are common.

2 Sequential Algorithms

There are three competitive approaches for the maximum common subgraph problem, each being the strongest on certain classes of instance. The first involves a reduction to the maximum clique problem, whilst the other two approaches are inspired by constraint programming.

2.1 Reduction to Maximum Clique

A *clique* in a graph is a subgraph where every vertex is adjacent to every other. There is a well-known reduction from the maximum common subgraph problem to the problem of finding a maximum clique in an *association graph* [21,25,33]; this reduction resembles the microstructure encoding [17] of the constraint programming approach described below. When combined with a modern maximum clique solver [35], this is the current best approach for solving the problem on labelled graphs [25]. A modified clique-like algorithm can also be used to solve the maximum common connected subgraph problem, by ensuring connectedness during search [25]; again, this is the best known way of solving the problem on labelled graphs. However, the association graph encoding is extremely memory-intensive, limiting its practical use to pairs of graphs with no more than a few hundred vertices.

2.2 Constraint Programming

The maximum common induced subgraph problem may be reformulated as a constraint optimisation problem, as follows. Observe that an equivalent definition of a common subgraph of graphs G and H is an injective *partial* mapping from G to H which preserves both adjacency and non-adjacency. Hence we pick whichever input graph has fewer vertices, and call it the *pattern*; the other graph is called the *target*. The model then follows from this new definition: for each vertex in the pattern, we create a variable, whose domain ranges over each vertex in the target graph, plus an additional value $\perp$ representing an unmapped vertex. We then have three sets of constraints. The first set says that for each pair of adjacent vertices in the pattern (that is, for each edge in the pattern), if neither of these vertices are mapped to $\perp$ then these vertices must be mapped to an adjacent pair of target vertices. The second set is similar, but looks at non-adjacent pairs (or non-edges). Finally, the third set ensures injectivity, by enforcing that the variables must be all different except when using $\perp$. This final set of constraints may either be implemented using binary constraints between all pairs of variables, or a special global "all different except $\perp$" propagator [32]. The objective is simply to find an assignment of values to variables, maximising the number of variables not set to $\perp$. The state of the art for this technique is a dedicated (non-toolkit) implementation of a forward-checking branch and bound search over this model [25,30].

Two approaches exist for ensuring connectedness: either a conventional global constraint and propagator can be used [25], or a special branching rule can enforce connectedness during search [36]. The two techniques are broadly comparable performance-wise [25], but the branching rule is simpler to implement.

2.3 Domain Splitting (McSplit and McSplit↓)

McCreesh et al. [28] observe that due to the special structure of the maximum common subgraph problem, the following property holds throughout the search process using the constraint programming model: any two variables either have domains with no values in common (with the possible exception of $\perp$), or have identical domains. The McSplit algorithm exploits this property. It explores essentially the same search tree as the basic forward-checking constraint programming approach, but using different supporting algorithms and data structures. Rather than storing a domain for each vertex in the pattern graph, equivalence classes of vertices in both graphs are stored in a special data structure which is modified in-place and restored upon backtracking. This enables fast propagation of the constraints and smaller memory requirements. In addition, this data structure enables stronger branching heuristics to be calculated cheaply. The McSplit algorithm effectively dominates conventional constraint programming approaches, being consistently over an order of magnitude faster.

The McSplit↓ algorithm is a variant designed for instances where we expect nearly all of the smaller graph to be found. It branches first on result size, from largest possible result downwards.

2.4 *k*-Less Subgraph Isomorphism

A different take on the constraint programming approach is presented by Hoffmann et al. [16]. They approach maximum common subgraph via the subgraph isomorphism problem, asking the question "if a pattern graph cannot be found in the target, how much of the pattern graph can be found?". The $k\downarrow$ algorithm tries to solve the subgraph isomorphism problem first for $k = 0$ (asking whether the whole pattern graph can be found in the target). Should that not be satisfiable, it tries to solve the problem for $k = 1$ (one vertex cannot be matched), and should that also not be satisfiable, it iteratively increases k until the result is satisfiable. This approach exploits strong invariants using paths and the degrees of vertices to prune large portions of the search space.

This algorithm is aimed primarily at large instances, where the two graphs are of different orders, and where it is expected that the solution will involve most of the smaller graph (that is, k is expected to be low). The sequential implementation we start with does not support labels or the connected variant.

3 Benchmark Instances

Most of the benchmark instances we will use come from a standard database for maximum common subgraph problems [6,34]. This benchmark set can be used in a number of ways, for different variants of the problem. Following other recent work [16,25,28], we use it to create five families of instances, as follows:

Unlabelled undirected instances, by selecting the first ten members of each parameter class where the graphs have up to 50 vertices each—this gives us a total of 4,110 instances.

Vertex labelled undirected instances, by selecting the first ten members of each parameter class (and so graphs have up to 100 vertices each), using the 33% labelling scheme [34] for vertices only. This gives 8,140 instances.

Both labelled, directed instances, by selecting the first ten members of each parameter class, and applying the 33% labelling scheme [34] to both vertices and edges. Again, this gives 8,140 instances.

Unlabelled, connected instances, as per the *unlabelled* case.

Both labelled, connected instances, starting in the same way as the *both labelled, directed* case. These are then converted to undirected graphs by treating edges as undirected, picking the label of the lower-numbered edge.

Following Hoffmann et al. [16], we also work with the 5,725 **Large** instances originally introduced for studying portfolios of subgraph isomorphism algorithms [18]. These graphs are unlabelled and undirected, and can include up to 6,671 vertices. We do not use the clique encoding on these instances due to its memory requirements.

4 Parallel Search

The clique and $k\downarrow$ algorithms already make use of fine-granularity bit-parallelism. To introduce coarse-grained thread parallelism, we will parallelise search: viewing backtracking search as forming a tree, we can explore different portions of the tree using different threads. We use a shared incumbent, so better solutions found by one thread can be used by others immediately. In this paper we use C++11 native threads, and so only support shared memory systems.

Parallel tree-search has a long history [1]. Of particular interest to us are so-called *anomalies* [2,20,22]: because we are not performing a fixed amount of work, we should have no expectation of a linear speedup, and instead we could see a sublinear speedup (much less than n from n processors, if speculative work turns out to be wasted) or a superlinear speedup (much more than n from n processors, if a strong incumbent is found more quickly). An absolute slowdown (a speedup much less than 1) is also possible when using some parallelisation techniques.

We stress that these anomalies are due to changes in the amount of work done, and are not due to work balance problems (although work balance is *also* unusually difficult for this problem). Anomalies can have a very strong effect on these algorithms, and we will therefore try to mitigate them as far as possible. In the evaluation of their "embarrassingly parallel search" technique, Malapert et al. [24] "consider unsatisfiable, enumeration and optimization [problem] instances", and "ignore the problem of finding a first feasible solution because the parallel speedup can be completely uncorrelated to the number of workers, making the results hard to analyze". They do "consider optimization problems for which the same variability can be observed, but at a lesser extent because the optimality proof is required". Unfortunately, many of the instances we consider behave more like decision problem instances than optimisation instances: due to the combination of a low solution density, good value-ordering heuristics, and a strong bound function in cases where the optimal solution is relatively large, it is often the case that the runtime is determined almost entirely by how long it takes to find an optimal solution, with the proof of optimality being nearly trivial. Indeed, attempts to parallelise the basic constraint programming approach by static decomposition have had limited success [29].

4.1 Parallel Maximum Clique

Thread-parallel versions of state-of-the-art maximum clique algorithms already exist. McCreesh et al. [27] compare several of these approaches, and make an important observation: although work balance is a problem due to the irregularity of the search tree, often the interaction between search order and parallel work decomposition is the dominating factor in determining speedups. They explain why anomalies are in fact common in practice: many clique problem instances benefit immensely from having found a strong incumbent, but have solutions which are either unique or rare, and are hard to find. They propose a work splitting mechanism which offsets anomalies, guaranteeing reproducibility (two runs with the same instance on the same hardware will give similar runtimes), scalability (increasing the number of cores cannot make things worse), and no absolute slowdowns. Additionally, this mechanism explicitly offsets the commitment to early branching choices, where search ordering heuristics are most likely to be inaccurate [4,15], making superlinear speedups common.

We will use this mechanism for our experiments. The clique-based maximum common subgraph algorithm effectively differs only in the preprocessing stage, and the clique-inspired connected algorithm described by McCreesh et al. [25] is sufficiently similar that it may be parallelised in exactly the same way. Based upon preliminary experiments, we set the mechanism's splitting depth limit parameter to be five rather than the original three, since maximum common subgraph instances appear to give even more irregular search trees than normal clique problem instances.

4.2 Parallel Constraint-Based Search

A similar approach may be used for the $k{\downarrow}$ algorithm. Although it is not quite a conventional branch and bound algorithm, each individual k pass is a tree-search, and may be parallelised. For each pass, we use the same work splitting mechanism as in the clique algorithm, starting by splitting only at the top level of search to explicitly introduce diversity, and then iteratively increasing the splitting depth as additional work is needed (up to a limit of five levels deep). Because the $k{\downarrow}$ algorithm uses a conventional constraint programming domain store, there is no need to use recomputation; the state is naturally copied at each branching point.

In principle the McSplit algorithm may be parallelised in exactly the same way. However, this algorithm makes heavy use of an in-place, backtrackable data structure, which is not copied for recursive calls. In order to introduce the *potential* for parallelism, we must make copies of the state data structure. Implemented naïvely, this can give an order of magnitude slowdown to the sequential algorithm, which can be hard to recover using parallelism. To lessen the effects, rather than copying state for each recursive call, we copy once before the main branching loop, and then copy that copy in each "helper" thread, replaying the branching loop without making duplicate recursive calls. (We believe a better approach using partial recomputation may be possible, and intend to investigate this further in the future.)

5 Empirical Evaluation

We perform our experiments on systems with dual Intel Xeon E5-2697A v4 processors and 512 GBytes RAM, running Ubuntu 17.04, with GCC 6.3.0 as the compiler. Each machine has a total of thirty-two cores. We run all our experiments with a one thousand second timeout for each instance. All of our sequential runtimes are from optimised implementations by their original authors which were not designed with parallelism in mind—that is, speedups from parallelism are genuine improvements over the state of the art.

5.1 Parallel Search Is Better Overall

In Fig. 1 we plot empirical cumulative distribution functions showing the number of instances solved over time, for both sequential (solid lines) and parallel (dotted lines) versions of each algorithm. To read these plots, make a choice of timeout along the x-axis (which uses a log scale). The y value at that point shows the number of instances whose runtime (individually) is at most x, for a particular algorithm. In other words, at any given x value, the highest line shows which algorithm is able to solve the largest number of instances using a per-instance timeout of that x value, bearing in mind that the actual sets of instances solved by each algorithm may be completely different.

With one exception, each plot gives the same conclusion: if we are working with a solving time of at least 100 ms, then for any problem family and any sequential algorithm, if given the option of switching to the corresponding parallel algorithm, then we should do so. For the McSplit algorithm on both labelled, connected instances, the parallel algorithm does not quite catch up to the sequential algorithm.

Although good at showing general trends, cumulative plots can hide interesting details. We therefore now take a closer look at each of the three algorithms in turn.

5.2 Clique Results in Depth

In the first column of Fig. 2, we see scatter plots comparing the sequential and parallel runtimes of the clique algorithm on an instance by instance basis, using a log-log plot. Each point represents one instance, with the x-axis being the sequential runtime and the y-axis the parallel runtime. Instances which timed out using one algorithm but not the other are shown as points along the outer borders. Points below the $x-y$ diagonal line represent speedups. The colour of the points indicates the relative size of the solution—darker points represent instances where the solution uses most of the vertices of the input graphs. (We use these conventions for scatter plots throughout this paper.)

Broadly speaking, the results are similar on each of the five families. For runtimes below 100 ms, overheads and the preprocessing step dominate, and we are usually only able to achieve a small speedup. At higher runtimes, most speedups appear to be between ten and thirty, except on the final family of both labelled connected instances, where they are mostly between five and ten. For a few instances, the speedups are lower (but they are still clearly speedups), whilst in the first four families, we also see evidence of superlinear speedups being relatively common.

However, attempting to determine a speedup by staring at a scatter plot is not particularly quantitative. We *could* attempt to find a best fit line through these points, pretending that the superlinear speedups are outliners. We might perhaps get away with this if outliers were rare enough, but in practice we are not expecting linear speedups (and for the other two algorithms, we will see that superlinear speedups are even more common). Alternatively, we could rig

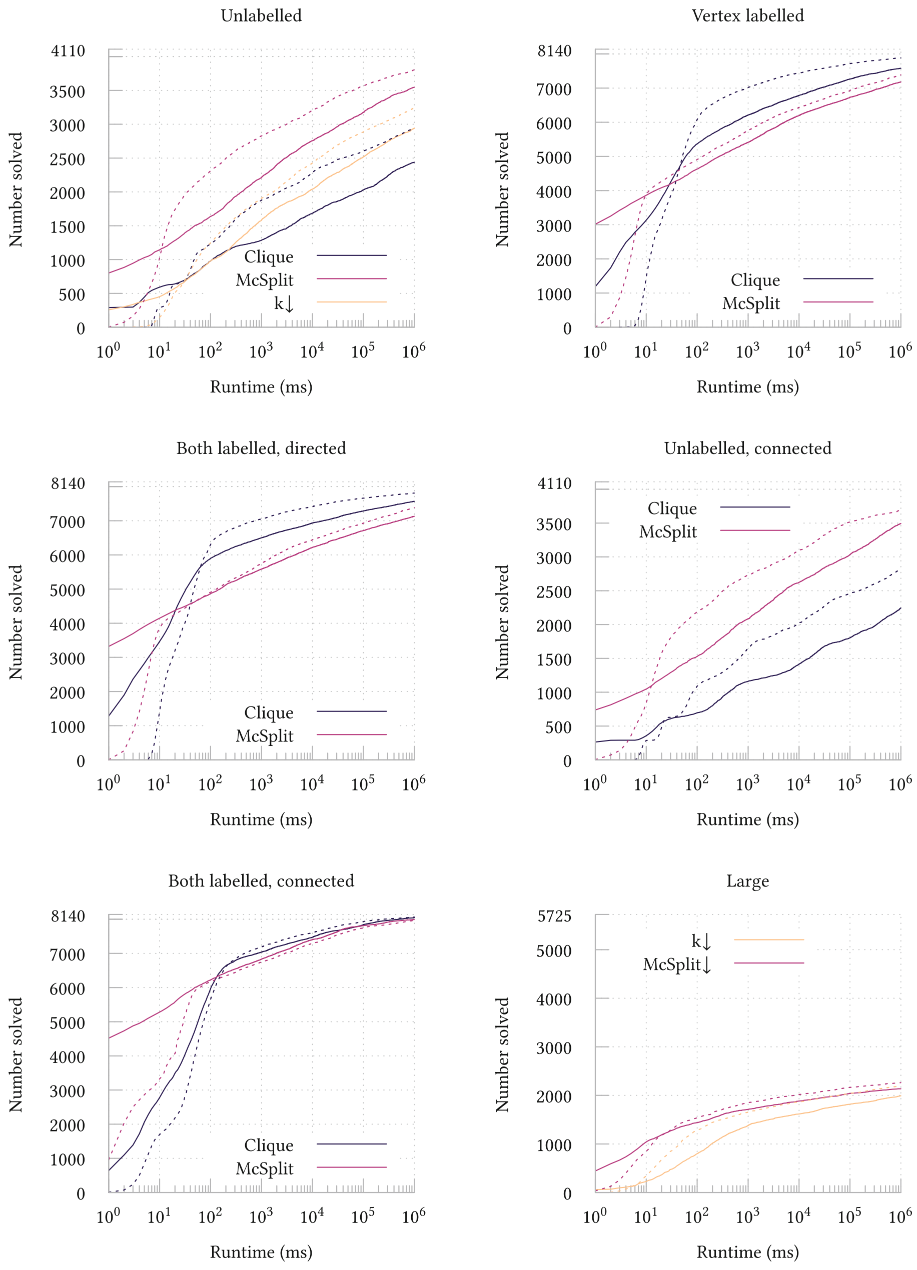

Fig. 1. The cumulative number of instances solved over time. Except in the bottom left plot, the 32 threaded parallel versions (shown using dotted lines) are always better in aggregate than the sequential versions (shown using solid lines).

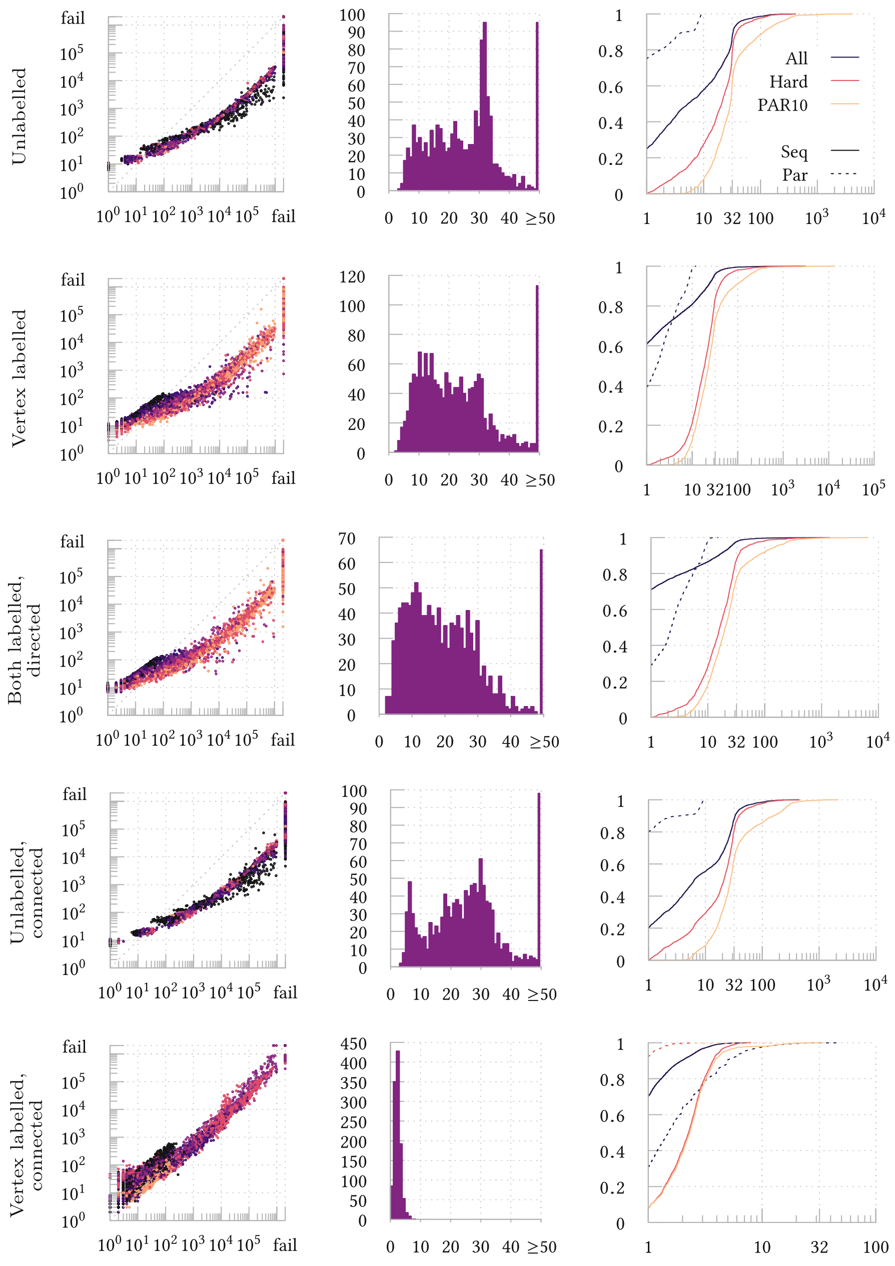

Fig. 2. In the left column, per-instance speedups, using the clique algorithm. The x-axis is sequential performance and the y-axis is 32 threaded performance. In the centre, histograms plotting the distribution of speedups for instances whose sequential runtime was at least 500 ms, and below the timeout. On the right, performance profiles.

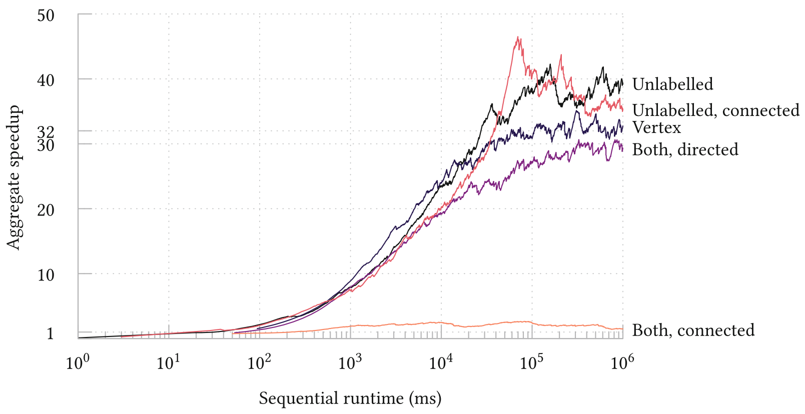

Fig. 3. Aggregate speedups from 32 threads, shown as a function of sequential runtime, for each family supported by the clique algorithm.

our experiments to remove anomalies, by priming search with a known-optimal solution; however, since the time to find an optimal solution (but not prove its optimality) is so important, we do not consider this to be a fair measure of algorithm performance [27].

A more principled approach is given in the second column of Fig. 2. For instances where the sequential run both succeeded and took at least 500 ms, we plot the distribution of speedups obtained. These histograms confirm our informal observations. However, these plots are still not especially satisfactory: in order to calculate a speedup, we can only consider instances where the sequential algorithm succeeded, and so these plots underestimate superlinear speedups. The choice of a 500 ms minimum sequential runtime is also rather arbitrary, and is acceptable only if we expect the parallel algorithms will only be used on relatively hard instances.

In the third column we show performance profiles [9]. A performance profile is a cumulative plot of how many times worse the performance of an algorithm is relative to the virtual best algorithm. Each plot shows three options as different lines. The ‘all’ lines include easy instances whose sequential runtime is below 500 ms, whilst the other two lines exclude them. The ‘hard’ line treats sequential timeouts as having been solved at the time limit, whilst the ‘PAR10’ line treats timeouts as taking ten times longer than the timeout (this convention is common in portfolios [37]). The solid lines show the sequential algorithms, whilst the dotted lines show the parallel algorithms. (There are no dotted lines on the top four plots for the ‘hard’ and ‘PAR10’ cases, since the parallel algorithm always beats the sequential algorithm in these cases.) We have normalised the y-axis to the number of counted instances in a given class.

Unfortunately, these three lines can paint very different pictures. For example, for unlabelled instances on the top row, if we include easy instances, it appears that the parallel algorithm can be up to ten times worse, whereas if we exclude them, it is never worse. If we do not use the PAR10 scheme, the performance profile also suggests that there are around twenty-five percent of the hard instances where the speedup is below 10, whilst using PAR10 correctly shows that such instances are rare. However, PAR10 is only effective in this regard because the "typical" speedup is in the region of 10 (and this is a particular inconvenience because we seek a way of characterising speedups which does not rely upon us already knowing that 10 is a reasonable choice of penalty).

A further problem is that to deal with the large superlinear speedups sometimes observed, a log scale must be used on the x-axis; this makes speedups of 10 and 30 look very similar, whilst in practice the difference is important.

To avoid these weaknesses, we propose a new way of characterising speedups. Refer back to the cumulative plots in Fig. 1. The usual way of comparing two algorithms on these plots is by measuring the vertical difference between lines, which would tell us how many more instances the parallel algorithm can solve than the sequential algorithm can with a particular choice of timeout. However, measuring the *horizontal* distance between lines also conveys information. Suppose the sequential algorithm can solve y instances with a selected timeout of s. By moving to the left on a cumulative plot, we can find the timeout p required for the parallel algorithm to solve the same number of instances, bearing in mind that *the two sets of instances could have completely different members.* We define the *aggregate speedup* to be $^s/_p$; this can be expressed as a function of time (i.e. s) or of the number of instances solved (y).

We plot aggregate speedups as a function of time in Fig. 3. For a sequential timeout of one thousand seconds, we get speedups of thirty to forty in the unlabelled, vertex labelled, and both labelled, directed cases. In the unlabelled cases, our aggregate speedup are over thirty-two, which is superlinear. With some detailed knowledge of the underlying sequential algorithm, this should perhaps not surprise us: for instances with a large solution, once we have found that solution, a proof of optimality is relatively easy. However, finding that solution can be unusually hard, particularly since the branching strategy for the connected constraint necessarily interferes with the tailored search order used by modern clique algorithms. In contrast, for the both labelled connected case, our aggregate speedup is barely larger than one. A closer inspection of the results shows that the search tree is unusually narrow and deep for these instances, making work balance harder and contention higher.

What about scalability and reproducibility? The first plot in Fig. 4 shows the effects of going from sequential to threaded with two cores, and the next four plots show the effects of doubling the number of threads each time. These plots show that most of the superlinear effects occur with fairly small numbers of threads, with nearly all of the benefits of increased diversity in search being obtained once eight threads are used. As expected, in no case does increasing the number of threads make things substantially worse. The final plot in Fig. 4 shows that runtimes are reproducible: running the same instance on the same hardware twice takes almost exactly the same amount of time.

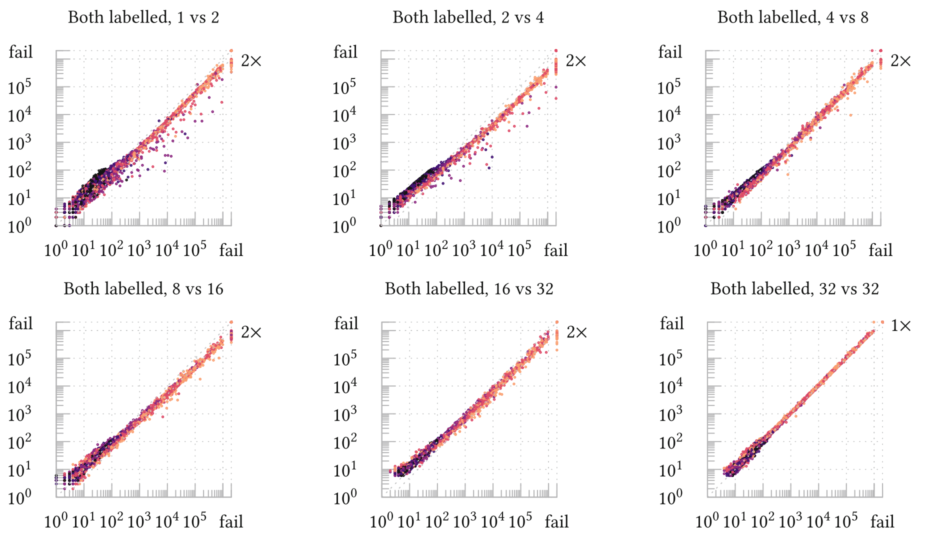

Fig. 4. Per-instance speedups from the clique algorithm on vertex- and edge-labelled, directed instances, when going from sequential to two threads in the first plot, then increasing the number of threads in subsequent plots. The final plot shows 32 threads versus a repeated run also with 32 threads.

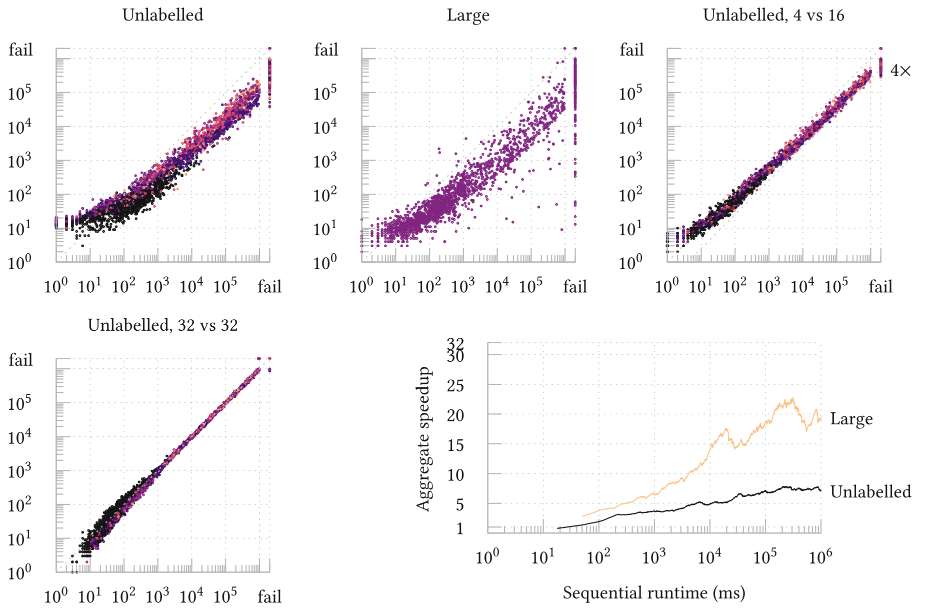

Fig. 5. In the first two plots, per-instance speedups, using the $k\downarrow$ algorithm. The x-axis is sequential performance and the y-axis is 32 threaded performance. Next, scalability and reproducibility, and finally, aggregate speedups for both families.

These results are comforting: they show that anomalies can be controlled, and that switching to a parallel algorithm is not only better, but also safe from a scientific reproducibility perspective.

5.3 $k\downarrow$ Results in Depth

In Fig. 5 we show per-instance and aggregate speedups for the $k\downarrow$ algorithm. On unlabelled instances, we see a range of speedups between 0.9 and ten, with an aggregate speedup of seven. These results are not as good as with the clique algorithm. Profiling suggests memory allocation problems: although the amount of work done would suggest good parallelism, the time taken to perform each domain copy operation increases as the number of threads increases. Unlike the clique algorithm, which has very small, cache-friendly data structures which are modified in-place, the state for the $k\downarrow$ algorithm is large and much of the runtime is spent copying data structures. (Our hardware is a dual multi-core processor configuration, and each core has its own low-level cache, but memory bandwidth is shared. Interestingly, on older Xeon E5 v2 systems, this problem is much more pronounced.)

For the large instances, our aggregate speedup is higher, at around twenty. This has two causes: for larger graphs, the computational effort per recursive call increases by more than the amount the memory copying does, reducing the memory problem slightly, and additionally a much larger number of superlinear speedups occurred with this family of instances. We could perhaps anticipate this latter effect: in many of these instances the maximum common subgraph covers all or nearly all of the smaller of the two graphs, and so once it is found, the proof of optimality is trivial. However, finding a witness can be difficult. We should also expect value-ordering heuristics in these algorithms to be weak at the top of search (they are based upon degree, and many graphs do not have a large degree spread), and so the benefits of high-up diversity can be extremely large [4,15,27]. Indeed, similar results were seen with a parallel version of the subgraph isomorphism algorithm upon which $k\downarrow$ is based [26].

The third and fourth plots in Fig. 5 show that as with the clique algorithm, this parallelism is reproducible, and that runtimes do not get worse when the number of threads is increased. (Although not shown, we also tried to parallelise $k\downarrow$ using randomised work-stealing from Intel Cilk Plus. Doing so gives generally reasonable results on average, as it does for the clique algorithm [27], but now repeat runtimes can differ by more than an order of magnitude.)

5.4 McSplit Results in Depth

Finally, we look at our attempts to parallelise the McSplit algorithm. Recall that doing so required heavy modifications to the implementation, introducing significant amounts of speculative copying of a data structure that is usually backtrackable and modified in-place.

For unlabelled, unlabelled connected, and large instances, Fig. 6 shows a particularly high proportion of strongly superlinear speedups. This is because the

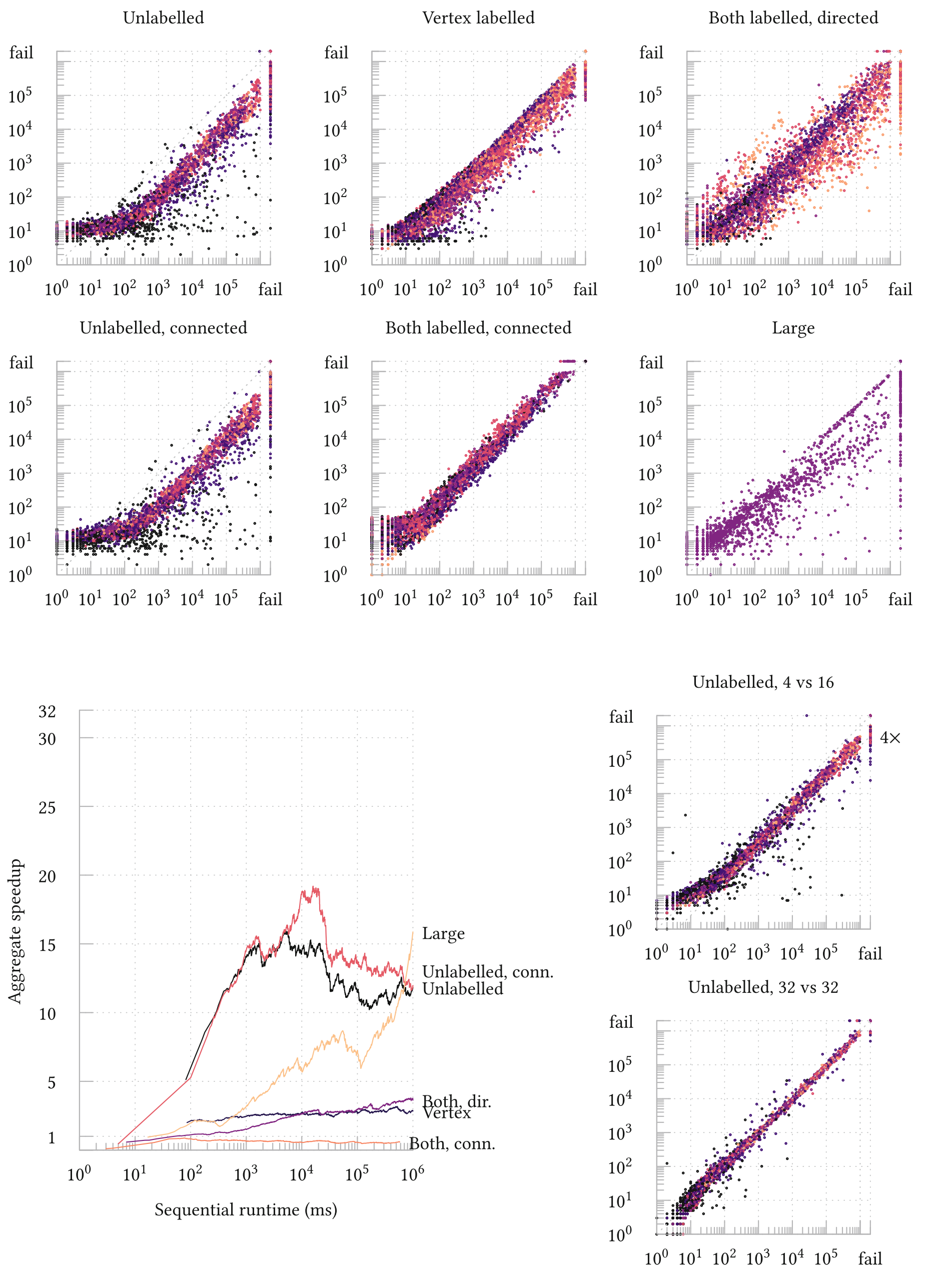

Fig. 6. On the first two rows, per-instance speedups, using McSplit. Below, aggregate speedups on the left, and on the right, scalability and reproducibility.

McSplit algorithm is focussed upon exploring the search space very quickly, and its branching heuristics do not have the advantage of the domain filtering performed by $k\downarrow$, or the rich inter-domain knowledge coming from the combination of the association graph encoding and the colour ordering used by clique algorithms. Thus making a correct value-ordering choice at the top of search is harder for McSplit than for other algorithms, and so increased diversity can be particularly beneficial.

For the large instances, we see evidence of work balance problems. McSplit's use of a "smallest domain first" variable-ordering heuristic, combined with the presence of $\bot$ in domains, tends to produce narrow (nearly binary) and deep search trees. These balance problems are even more evident in the labelled cases (where following a guessed assignment, many domains are left with only two values), and often lead to little to no speedup being obtained. Indeed, for the labelled, connected case, we see a slight aggregate *slowdown*.

The scatter plots also show occasional large absolute slowdowns, sometimes by over an order of magnitude. These are due to the changes which had to be made to the sequential algorithm (and because we are benchmarking against the sequential algorithm, not a parallel algorithm with one thread), rather than search order effects. In cases where parallelism cannot be exploited, the cost of speculatively copying domains at each level of search can dominate the runtimes. Because of this, fixing work balance problems by increasing the splitting depth typically makes matters much worse, not better.

What about scalability and reproducibility? Figure 6 presents a less ideal picture than for the previous two algorithms—again, this is due to speculative overheads that fail to pay off, rather than being anomalies in the classical sense.

6 Conclusion

We have parallelised three state-of-the-art maximum common (connected) subgraph algorithms with a reasonable degree of success by using dynamic work-splitting. Despite having a branch and bound flavour, all three sequential algorithms had their own difficulties and performance characteristics which prevented them from cleanly fitting into common abstraction frameworks. Nonetheless, our results show that the parallel algorithms are not just better in aggregate, but also preserve the desirable reproducibility properties of sequential algorithms. A large part of our success was down to using parallelism to explicitly introduce diversity into the search process, offsetting weak early value-ordering branching choices.

There is room for improvement, particularly with respect to work balance. However, improvements to work balance must not come at the expense of the search order properties, nor at the cost of increased overheads.

More generally, we introduced the idea of *aggregate speedups*, to deal with measuring a speedup in the presence of anomalies. This measure gives sensible answers even when working with instances which behave like decision problems. Aggregate speedups informed part of our analysis, but our results highlight the importance of viewing results in multiple ways, and in using large families of instances with different characteristics when evaluating parallel search algorithms—had we looked only at unlabelled instances, or only at labelled connected instances, our conclusion would be very different.

References

1. Bader, D.A., Hart, W.E., Phillips, C.A.: Parallel algorithm design for branch and bound. In: Greenberg, H.J. (ed.) Tutorials on Emerging Methodologies and Applications in Operations Research. ISOR, vol. 76, pp. 1–44. Springer, New York (2005). https://doi.org/10.1007/0-387-22827-6_5
2. de Bruin, A., Kindervater, G.A.P., Trienekens, H.W.J.M.: Asynchronous parallel branch and bound and anomalies. In: Ferreira, A., Rolim, J. (eds.) IRREGULAR 1995. LNCS, vol. 980, pp. 363–377. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60321-2_29
3. Bunke, H.: On a relation between graph edit distance and maximum common subgraph. Pattern Recogn. Lett. **18**(8), 689–694 (1997)
4. Chu, G., Schulte, C., Stuckey, P.J.: Confidence-based work stealing in parallel constraint programming. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 226–241. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04244-7_20
5. Combier, C., Damiand, G., Solnon, C.: Map edit distance vs. graph edit distance for matching images. In: Kropatsch, W.G., Artner, N.M., Haxhimusa, Y., Jiang, X. (eds.) GbRPR 2013. LNCS, vol. 7877, pp. 152–161. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38221-5_16
6. Conte, D., Foggia, P., Vento, M.: Challenging complexity of maximum common subgraph detection algorithms: a performance analysis of three algorithms on a wide database of graphs. J. Graph Algorithms Appl. **11**(1), 99–143 (2007)
7. Cook, D.J., Holder, L.B.: Substructure discovery using minimum description length and background knowledge. J. Artif. Intell. Res. **1**, 231–255 (1994)
8. Delavallade, T., Fossier, S., Laudy, C., Lortal, G.: On the challenges of using social media for crisis management. In: Rogova, G., Scott, P. (eds.) Fusion Methodologies in Crisis Management, pp. 137–175. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-22527-2_8
9. Dolan, E.D., Moré, J.J.: Benchmarking optimization software with performance profiles. Math. Program. **91**(2), 201–213 (2002)
10. Ehrlich, H.C., Rarey, M.: Maximum common subgraph isomorphism algorithms and their applications in molecular science: a review. Wiley Interdisc. Rev.: Comput. Mol. Sci. **1**(1), 68–79 (2011)
11. Fang, M., Yin, J., Zhu, X., Zhang, C.: Trgraph: cross-network transfer learning via common signature subgraphs. IEEE Trans. Knowl. Data Eng. **27**(9), 2536–2549 (2015)
12. Fernández, M., Valiente, G.: A graph distance metric combining maximum common subgraph and minimum common supergraph. Pattern Recogn. Lett. **22**(6/7), 753–758 (2001)
13. Gao, D., Reiter, M.K., Song, D.: BinHunt: automatically finding semantic differences in binary programs. In: Chen, L., Ryan, M.D., Wang, G. (eds.) ICICS 2008. LNCS, vol. 5308, pp. 238–255. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88625-9_16

14. Gay, S., Fages, F., Martinez, T., Soliman, S., Solnon, C.: On the subgraph epimorphism problem. Discret. Appl. Math. **162**, 214–228 (2014)
15. 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, 20–25 August 1995, vol. 2, pp. 607–615. Morgan Kaufmann (1995)
16. Hoffmann, R., McCreesh, C., Reilly, C.: Between subgraph isomorphism and maximum common subgraph. In: Singh, S.P., Markovitch, S. (eds.) Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, 4–9 February 2017, San Francisco, California, USA, pp. 3907–3914. AAAI Press (2017)
17. Jégou, P.: Decomposition of domains based on the micro-structure of finite constraint-satisfaction problems. In: Fikes, R., Lehnert, W.G. (eds.) Proceedings of the 11th National Conference on Artificial Intelligence, Washington, DC, USA, 11–15 July 1993, pp. 731–736. AAAI Press/The MIT Press (1993)
18. Kotthoff, L., McCreesh, C., Solnon, C.: Portfolios of subgraph isomorphism algorithms. In: Festa, P., Sellmann, M., Vanschoren, J. (eds.) LION 2016. LNCS, vol. 10079, pp. 107–122. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50349-3_8
19. Kriege, N.: Comparing graphs. Ph.D. thesis, Technische Universität Dortmund (2015)
20. Lai, T., Sahni, S.: Anomalies in parallel branch-and-bound algorithms. Commun. ACM **27**(6), 594–602 (1984)
21. Levi, G.: A note on the derivation of maximal common subgraphs of two directed or undirected graphs. CALCOLO **9**(4), 341–352 (1973)
22. Li, G., Wah, B.W.: Coping with anomalies in parallel branch-and-bound algorithms. IEEE Trans. Comput. **35**(6), 568–573 (1986)
23. Luo, C., Wang, X., Su, C., Ni, Z.: A fixture design retrieving method based on constrained maximum common subgraph. IEEE Trans. Autom. Sci. Eng. **PP**(99), 1–13 (2017)
24. Malapert, A., Régin, J., Rezgui, M.: Embarrassingly parallel search in constraint programming. J. Artif. Intell. Res. **57**, 421–464 (2016)
25. McCreesh, C., Ndiaye, S.N., Prosser, P., Solnon, C.: Clique and constraint models for maximum common (connected) subgraph problems. In: Rueher, M. (ed.) CP 2016. LNCS, vol. 9892, pp. 350–368. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44953-1_23
26. McCreesh, C., Prosser, P.: A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 295–312. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23219-5_21
27. 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)
28. 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, 19–25 August 2017 (2017, to appear)
29. Minot, M., Ndiaye, S.N., Solnon, C.: A comparison of decomposition methods for the maximum common subgraph problem. In: 27th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2015, Vietri sul Mare, Italy, 9–11 November 2015, pp. 461–468. IEEE Computer Society (2015)

30. Ndiaye, S.N., Solnon, C.: CP models for maximum common subgraph problems. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 637–644. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23786-7_48
31. Park, Y.H., Reeves, D.S., Stamp, M.: Deriving common malware behavior through graph clustering. Comput. Secur. **39**, 419–430 (2013)
32. Petit, T., Régin, J.-C., Bessière, C.: Specific filtering algorithms for over-constrained problems. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 451–463. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45578-7_31
33. Raymond, J.W., Willett, P.: Maximum common subgraph isomorphism algorithms for the matching of chemical structures. J. Comput. Aided Mol. Des. **16**(7), 521–533 (2002)
34. Santo, M.D., Foggia, P., Sansone, C., Vento, M.: A large database of graphs and its use for benchmarking graph isomorphism algorithms. Pattern Recogn. Lett. **24**(8), 1067–1079 (2003)
35. Segundo, P.S., Matía, F., Rodríguez-Losada, D., Hernando, M.: An improved bit parallel exact maximum clique algorithm. Optim. Lett. **7**(3), 467–479 (2013)
36. Vismara, P., Valery, B.: Finding maximum common connected subgraphs using clique detection or constraint satisfaction algorithms. In: Le Thi, H.A., Bouvry, P., Pham Dinh, T. (eds.) MCO 2008. CCIS, vol. 14, pp. 358–368. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87477-5_39
37. Xu, L., Hoos, H., Leyton-Brown, K.: Hydra: automatically configuring algorithms for portfolio-based selection. In: Fox, M., Poole, D. (eds.) Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, 11–15 July 2010. AAAI Press (2010)