Reducing the Branching in a Branch and Bound Algorithm for the Maximum Clique Problem

Ciaran McCreesh and Patrick Prosser

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

Abstract. Finding the largest clique in a given graph is one of the fundamental NP-hard problems. We take a widely used branch and bound algorithm for the maximum clique problem, and discuss an alternative way of understanding the algorithm which closely resembles a constraint model. By using this view, and by taking measurements inside search, we provide a new explanation for the success of the algorithm: one of the intermediate steps, by coincidence, often approximates a "smallest domain first" heuristic. We show that replacing this step with a genuine "smallest domain first" heuristic leads to a reduced branching factor and a smaller search space, but longer runtimes. We then introduce a "domains of size two first" heuristic, which integrates cleanly into the algorithm, and which both reduces the size of the search space and gives a reduction in runtimes.

1 Introduction

A clique in a graph is a subset of vertices, each of which is adjacent to every other vertex in this subset—we illustrate this in Fig. 1. Finding the size of a maximum clique in a given graph is one of Garey and Johnson's fundamental NP-hard problems [1]. The maximum clique problem has been studied in a

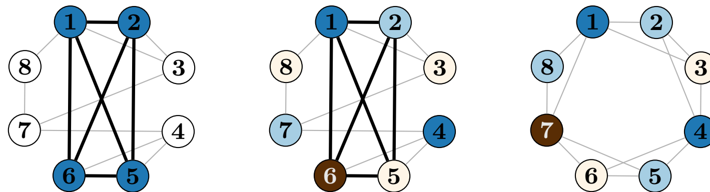

Fig. 1. On the left, a graph with a maximum clique of size four. Next, a greedy four-colouring of this graph: vertices $\{1, 4\}$ have been coloured dark blue, vertices $\{2, 7\}$ are light blue, vertices $\{3, 5, 8\}$ are pale cream and vertex 6 is dark chocolate. On the right, a graph which requires four colours but does not contain a clique of size four.

constraint programming setting by Régin [2], and using MaxSAT and MaxSAT-inspired algorithms by Li et al. [3,4,5,6]. However, we will be looking at a family of dedicated branch and bound algorithms due to Tomita et al. [7,8,9]. These algorithms are widely used on "real" problems in practice [10,11,12,13,14,15,16], and many variations have been proposed [17,18,19,20], notably including bit- and thread-parallel versions [21,22,23,15]; the techniques we will investigate have also been reused to solve other problems, including maximum common subgraph [15] and maximum balanced induced biclique [24]. But despite this wide use, it is not entirely clear *why* these algorithms work so well. In particular, in one part of the algorithm we iterate over a certain array backwards. It is easy to check that this leads to a much smaller search space than iterating over this array in the forwards direction instead, but recent experimental work contradicts the conventional explanation for why this should be the case.

By rephrasing the algorithm using language from constraint programming, we will see that these "forward" and "backward" iterations are effectively two different variable selection heuristics (although the variables only exist implicitly). We take measurements inside search to demonstrate that "backwards", by coincidence, approximates a "smallest domain first" (SDF) heuristic, and that "forwards" is roughly "largest domain first". It is then easy to modify the algorithm to use a genuine SDF heuristic; doing so leads to a smaller search space due to a reduced branching factor, but longer runtimes due to the cost of performing a sort for each recursive call made by the algorithm. Finally, we show how to get both a smaller search space *and* improved runtimes by using a cheaper alternative to SDF, which is effectively "domains of size two first".

2 Algorithms for the Maximum Clique Problem

Throughout, let $G = (V, E)$ be a graph with vertex set V and edge set E. We write $\mathrm{V}(G)$ for V, and $\mathrm{N}(G, v)$ for the neighbourhood of a vertex v (that is, the vertices adjacent to it). The size of a maximum clique is denoted ω. When discussing random graphs, we use $G(n, p)$ to denote an Erdős-Rényi random graph with n vertices, and with edges between each pair of distinct vertices with independent probability p.

We start by describing Algorithm 1, a generic, exact branch and bound algorithm for the maximum clique problem. This algorithm is essentially Tomita et al.'s "MCS" [9] with a simpler initial vertex ordering and the colour repair step omitted (Prosser's computational study [18] calls this combination "MCSa1"), but we describe it in a new, more flexible way that allows us to investigate and explain its behaviour.

The key to this algorithm is a relationship between cliques and colourings. A *colouring* of a graph is an assignment of vertices to colours, where adjacent vertices are given different colours. A set of like-coloured vertices in a given colouring is called a *colour class*. A clique in a coloured graph can contain at most one vertex from each colour class (see Fig. 1), so if we can colour a graph using k colours, we have shown that $\omega \leq k$. This is not generally an equality: the

Algorithm 1: A generic exact algorithm to deliver a maximum clique.

```
1  maxClique :: (Graph G) → Set
2  begin
3      global C_max ← ∅
4      expand(G, ∅, V(G))
5      return C_max

6  expand :: (Graph G, Set C, Set P)
7  begin
8      colourClasses ← colour(G, P)
9      while colourClasses ≠ ∅ do
10         colourClass ← select(colourClasses)
11         for v ∈ colourClass do
12             if |C| + |colourClasses| ≤ |C_max| then return
13             P′ ← P ∩ N(G, v)
14             C ← C ∪ {v}
15             if |C| > |C_max| then C_max ← C
16             if P′ ≠ ∅ then expand(G, C, P′)
17             C ← C \ {v}
18         colourClasses ← remove(colourClasses, colourClass)
```

third graph in Fig. 1 has $\omega = 3$, but cannot be coloured using three colours. Like finding a maximum clique, finding a minimum colouring is NP-hard. However, we may construct a greedy colouring in polynomial time.

These facts are used to grow a maximum clique, as follows. We have a variable C, which contains the current growing clique, and a variable P (the candidate set) which contains vertices which may potentially be added to C. Initially C is empty, and P contains every vertex in the graph (line 4). Now we produce a greedy colouring of the subgraph induced by P (line 8); we describe how this is performed in Algorithm 2. From this colouring, we select a colour class (line 10), and from that colour class we select a vertex v (line 11). We then consider every clique which contains the vertices in C plus v, by filtering from P any vertices not adjacent to v (line 13), and recursing (line 16). Next we consider every clique which contains the vertices in C but not v: we remove v from consideration (line 17), and select a new vertex from the current colour class (line 11). When the current colour class has been explored we remove it (line 18) to consider the possibility of not selecting any vertex at all from this colour class, and then select a new colour class (line 9).

We keep track of the largest clique we have found so far (line 15), which we call the *incumbent*, and store it in C_{max}. At any point during the search, if the number of colour classes remaining plus the number of vertices in C is not strictly greater than $|C_{max}|$, we cannot unseat the incumbent and so we backtrack (line 12).

Algorithm 2: Greedily colour vertices, delivering a list of colour classes.

```
1  colour :: (Graph G, Set P) → List of Set
2  begin
3      colourClasses ← ∅
4      uncoloured ← copy(P)
5      while uncoloured ≠ ∅ do
6          current ← ∅
7          for v ∈ uncoloured do
8              if current ∩ N(G, v) = ∅ then current ← current ∪ {v}
9          uncoloured ← uncoloured \ current
10         colourClasses ← append(colourClasses, current)
11     return colourClasses
```

Note that we produce a new colouring each time we recurse—we generally get tighter colourings as we consider smaller subproblems. Thus being able to produce a colouring very quickly is important. San Segundo et al. [21,22] observed that using a bitset encoding for the entire algorithm would lead to a performance improvement of between two and twenty times, without changing the steps taken. We will be using (but not explicitly describing) a bitset encoding throughout, and refer the reader to these papers for implementation details.

In constraint programming terms, we can think of colour classes as variables, and vertices within a colour class as values. We are forming a clique by picking a vertex from each colour class. There is also a "nothing from this colour class" value, which we wish to take as infrequently as possible. On line 10 we are selecting a variable, and on line 11 we are trying a value for that variable; the filtering on line 13 is propagation. There is the slight conceptual complication in that we are producing a new set of variables at each recursive call—perhaps this is why this explanation has not been considered previously.

There are three choices to be made when implementing this algorithm. The first is how the colouring is produced (line 8). For this paper, we will use a simple greedy sequential colouring, where colour classes are filled by selecting vertices in order. We show this in Algorithm 2. The order in which vertices are considered has a large effect upon the colourings produced; here we select vertices in a static non-increasing degree order (this is implemented by permuting the graph at the top of search). Other initial vertex orders and more sophisticated (but also more computationally expensive) colouring algorithms sometimes give better results, and sometimes give worse results. A computational study by Prosser [18] examines this issue in depth; our approach is compatible with other colourings and initial vertex orderings, and we are describing the simplest options for clarity.

Note that the choice of vertex ordering here is made to improve the quality of the colouring, in the hope that the greedy colouring will be close to optimal. For computationally challenging graphs, the degree of a vertex is often not an indication of whether it is present in a solution—either by design [25,26], because

the degree spread is very narrow (as in Erdős-Rényi random graphs), or because the graph contains many maximum cliques but the difficulty lies in proving optimality [27,28,29].

The second choice to be made is the order in which colour classes are selected (the `select` function used in line 10) and the third is the order in which vertices are selected from within a colour class (iteration over the colour class in line 11). These choices have not been investigated deeply (nor have they been presented explicitly as choices to be made). To emulate Tomita's algorithms we would select (line 10) and then remove (line 18) the last colour class from the list of colour classes constructed by Algorithm 2. Vertices within colour classes would also be selected in reverse order, with the last vertex in that colour class chosen first.

But why select vertices in colour class order, and why select from right to left? This may be implemented very efficiently by using a pair of arrays, as in Fig. 2. The first array (drawn as coloured vertices) controls the iteration order: it contains vertex numbers, in the reverse order that they are to be considered. The second array holds the bound (drawn as a list of numbers): in the ith entry we store the number of colours that were used to colour the induced subgraph which contains only the first i vertices from the first array. Since vertices with the same colour are adjacent in the iteration order, the bound is decreasing when iterating from right to left, which allows Algorithm 1 to be implemented using a single loop—in fact, this algorithm has not previously been described in any other way.

However, recursing from left to right (and thus selecting from the first colour class first, rather than the last colour class first) may be implemented equally efficiently, so why use a reverse order? Tomita claims that vertices in the rightmost colour class are "generally expected [to have a] high probability of belonging to a maximum clique" [8]. This claim was not tested experimentally, beyond verifying that the reverse ordering gives much worse performance, and recent experiments by the authors [29] and by Batsyn et al. [19] suggest that for several families of graphs, these algorithms are not particularly good at finding a maximum clique quickly.

We argue that there is another factor contributing to the success of the reverse selection order. Intuitively, one might think that early colour classes are likely to be larger: colour classes are filled greedily, with vertices being placed in the first available colour class. Selecting from small colour classes first is beneficial: consider Fig. 2, and suppose $C_{max} = 3$. If we select v from the rightmost colour class (which contains only one vertex) first, we make only a single recursive call which cannot be eliminated by the bound. But if we were to select from any other colour class, we would have to make either two or three recursive calls before our bound would decrease. (This also shows why we commit entirely to a selected colour class: we want to eliminate colour classes as quickly as possible.)

In constraint programming terms, selecting from small colour classes first is a "smallest domain first" variable selection heuristic. Such a heuristic tends to give a low branching factor locally (that is, it reduces the number of recursive

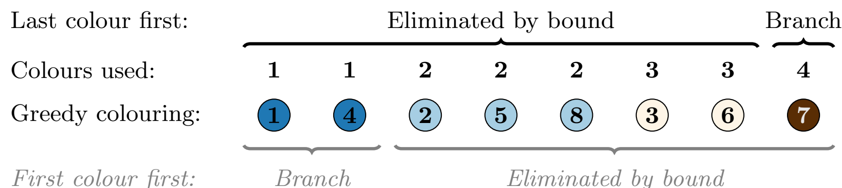

Fig. 2. The colour classes from the third graph of Fig. 1, in colour order. Above the vertices is an array of bounds: the ith entry shows the number of colours used to colour the subgraph containing only the first i vertices from the colouring. Now suppose an incumbent of size three had already been found. If we select from the rightmost colour class first, and discover that there is no clique of size four containing vertex 7, then we may abandon search. But if we select from the leftmost colour class first (as drawn below), we must recurse twice: once to show that there is no clique of size four containing vertex 1, and then again for vertex 4.

calls made) [30]. This does not necessarily produce the best possible search tree globally, but we will demonstrate that it is generally beneficial in this context.

2.1 Are Colour Classes Roughly Sorted by Size?

We will now test our intuition, by augmenting Algorithm 2 to take measurements inside the search. The hypothesis we are testing is as follows: is there a correlation between the position a colour classes is in, and the position it would be in if colour classes were sorted by size (largest first)? To measure this, we use Spearman's rank correlation coefficient (with rank ties) [31]; this will give us a value of 1 if there is a perfect monotonically increasing relationship, -1 if it is perfectly monotonically decreasing, and a value in-between otherwise.

We performed this test for each colouring produced, over 100 samples of random graphs $G(150, 0.9)$. The results are plotted in the top left graph of Fig. 3. For the x-axis, we use the number of colour classes used. For the y-axis, rather than show the average, we show the distribution of the results of the statistical test (so the colours in each column sum to 1). For comparison purposes, the bottom left graph shows what we would see if the colour classes were in no particular order (we shuffle the colour classes before running the test), and the bottom right graph shows the color classes fully sorted (i.e. SDF). These results confirm our suspicions that colour classes are "roughly" sorted by size, as a side effect of the greedy colouring process: the top left graph is much more heavily weighted towards 1 (sorted) than the shuffled graph. In other words, the greedy colouring process and backwards iteration is approximating an SDF heuristic.

The top right graph shows the effects of our "domains of size two first" heuristic, which we describe below. As its name suggests, a partial sort increases the degree to which colour classes are sorted by size, but does not sort them fully—it is a cheap surrogate for SDF.

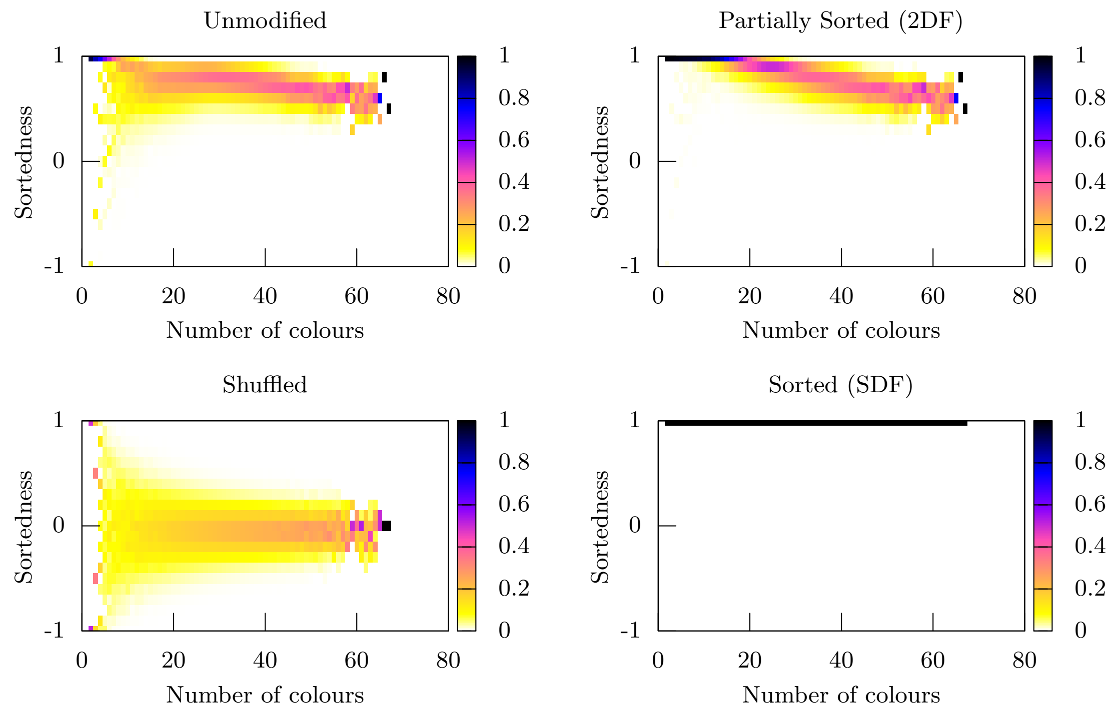

Fig. 3. Are colour classes roughly ordered by size? A value of 1 means "yes, largest first", 0 means "no", and -1 means "yes, smallest first". In the top left graph, the original algorithm, and in the top right, the effects of a partial sort. For comparison, the bottom left graph shows shuffled colour classes, and the bottom right graph shows fully sorted colour classes. Results are from 100 samples of $G(150, 0.9)$.

2.2 Reordering Colour Classes to Reduce the Branching Factor

We have established empirically that smaller colour classes tend to be picked earlier by Tomita's algorithms, and explained theoretically why this is beneficial. Now ask what would happen if we increased this effect. We consider two approaches.

The "sorted", or "smallest domains first" variation. We could explicitly select from the smallest colour class (i.e. the smallest domain) first. This could be implemented directly, via a different `select` function used with Algorithm 1, or we could use Tomita's "two arrays" approach and add a (stable) sort to the end of Algorithm 2.

The "partially sorted", or "domains of size two first" (2DF) variation. We also consider a potentially cheaper alternative: instead of fully sorting colour classes by size, we propose a partial sort that moves colour classes containing only one vertex (which we call *singleton* colour classes) to the end of the list of colour classes, so that they are selected first. In other words, we are picking from domains with two values (a single vertex, plus the "nothing" option) first. We show how to do this in a way which is compatible with a bitset encoding in Algorithm 3: when we produce a colour class containing only a single vertex, we append that colour class onto the list *singletons* (line 12) and when every vertex has been processed we return the concatenated list of colour classes with the singletons appearing at the end (line 15). We then replace the call to `colour` in line 8 of Algorithm 1 with a call to `colourSort2DF`, and implement the `select` step and the bound by using two arrays and selecting the rightmost entry first, as in Tomita's algorithms.

2.3 Tie-breaking

But why are we preserving the relative order of the partially sorted colour classes—that is, why do we specify a stable sort, or why is it important to put the last singleton colour class at the end of the list of colour classes? Suppose Algorithm 2 produced the colour classes shown in Fig. 4. Due to the greediness of the colouring, vertices 5, 6, 7, 8 and 9 must all be adjacent to vertex 4 (the only member of the dark chocolate colour class), for if one were not, it would also have been coloured dark chocolate. Thus if Algorithm 1 selects colour class $\{4\}$ in preference to the other singleton colour class $\{7\}$, the new candidate set P' will contain *some* of the vertices from the set $\{1, 2, 3\}$, and *all* of the vertices from the sets $\{5, 6\}$, $\{7\}$ and $\{8, 9\}$. However, by the same kind of reasoning, if the colour class $\{7\}$ is selected before $\{4\}$, P' will contain *some* of the vertices from the sets $\{1, 2, 3\}$ and $\{5, 6\}$ and *all* of the vertices in the sets $\{4\}$ and $\{8, 9\}$, so the new candidate set will potentially be smaller. This is why we preserve the order: selecting from the latest-coloured singleton colour class first can increase the amount of filtering done on P, giving a smaller P' in the recursive call (line 16), further reducing the branching in the search process.

Algorithm 3: Greedily colour vertices, delivering a partially sorted list of colour classes, with singleton colour classes deferred to the end.

```
1  colourSort2DF :: (Graph G, Set P) → List of Set
2  begin
3      colourClasses ← ∅
4      singletons ← ∅
5      uncoloured ← copy(P)
6      while uncoloured ≠ ∅ do
7          current ← ∅
8          for v ∈ uncoloured do
9              if current ∩ N(G, v) = ∅ then current ← current ∪ {v}
10         uncoloured ← uncoloured \ current
11         if |current| = 1 then
12             singletons ← append(singletons, current)
13         else
14             colourClasses ← append(colourClasses, current)
15     return concatenate(colourClasses, singletons)
```

2.4 Compatibility with Other Improvements

Both the "sorted" and "partially sorted" changes are compatible with other recent improvements that have been proposed for this family of algorithms. They do not interfere with priming the search with a heuristic solution [19], they are not sensitive to alternative vertex orderings, and critically, they are compatible with multi-core parallelism [15,23,29].

One improvement with which they are *not* compatible is a relaxed colouring proposed by San Segundo and Tapia [20]. Relaxed colourings also do not interact cleanly with parallel branch and bound or with priming (with relaxed colourings, finding a larger incumbent earlier can be a penalty rather than a benefit), so we do not consider this to be a substantial weakness.

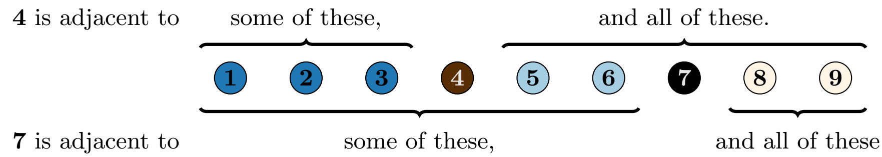

Fig. 4. Due to the greedy colouring, singleton colour classes are not equally powerful from a filtering perspective. For any singleton colour class, its vertex is adjacent to every vertex with a later colour, but only some vertices with an earlier colour. Here, branching on vertex 7 rather than vertex 4 is likely to lead to more filtering, giving a smaller subproblem at the next recursive call.

3 Experimental Results

We now evaluate our "SDF" and "2DF" changes experimentally on a range of standard and random problems. Where runtimes are given, experimental results are produced on a machine with an Intel Xeon E5645 CPU. The time taken to read in a graph from a file is not measured, but preprocessing time is included. The algorithms were implemented in C++. Sometimes we report the number of "nodes" explored by an algorithm, by which we mean the number of recursive calls made to the `expand` function in Algorithm 1.

For our baseline, we use a bitset encoded version of the variant Prosser calls "MCSa1" [18]; our implementation has been shown to perform very competitively with other implementations of the same algorithm [23]. We also have a parallel implementation of these algorithms, although we are reporting sequential results for ease of understanding (parallel branch and bound is speculative, and we often see anomalous speedups [15,23] due to additional diversity from differing search orders [29]). For the "SDF" implementation, we add a stable sort (using the C++ standard library function) at the end of Algorithm 2; for the "2DF" implementation, we replace Algorithm 2 with Algorithm 3. In both cases, we use Tomita's "two arrays" approach, and have `select` pick from right to left.

3.1 Random Graphs

In the top graph in Fig. 5 we show the number of search nodes required for random graphs $G(200, x)$ for values of x between 0.70 and 0.99 (averaged over 100 graph instances for each x). We see that there is a clear benefit to reordering colour classes, either by sorting or by partially sorting. However, sorting and partial sorting give lines which are too close to be easily distinguishable—at least for these graphs, simply deferring singleton colour classes is as effective as a full sort.

But do reductions in the size of the search space help with performance? In the graph below, we present the same data, but measuring runtimes. We see that the improvements to runtimes from a partial sort reflects the improvements to search nodes. On the other hand, it is clear that a full sort is extremely expensive: we get a factor of five slowdown despite the smaller search space.

3.2 Standard Benchmark Problems

Next we consider a range of standard benchmark problems from the Second DIMACS Implementation Challenge[1] and from BHOSLIB ("Benchmarks with Hidden Optimum Solutions for Graph Problems")[2]. From DIMACS, we have omitted graphs where the number of search nodes is below 10^4. Some extremely hard instances, where an optimal solution is either unknown or takes more than

[1] `http://dimacs.rutgers.edu/Challenges/`
[2] `http://www.nlsde.buaa.edu.cn/~kexu/benchmarks/graph-benchmarks.htm`

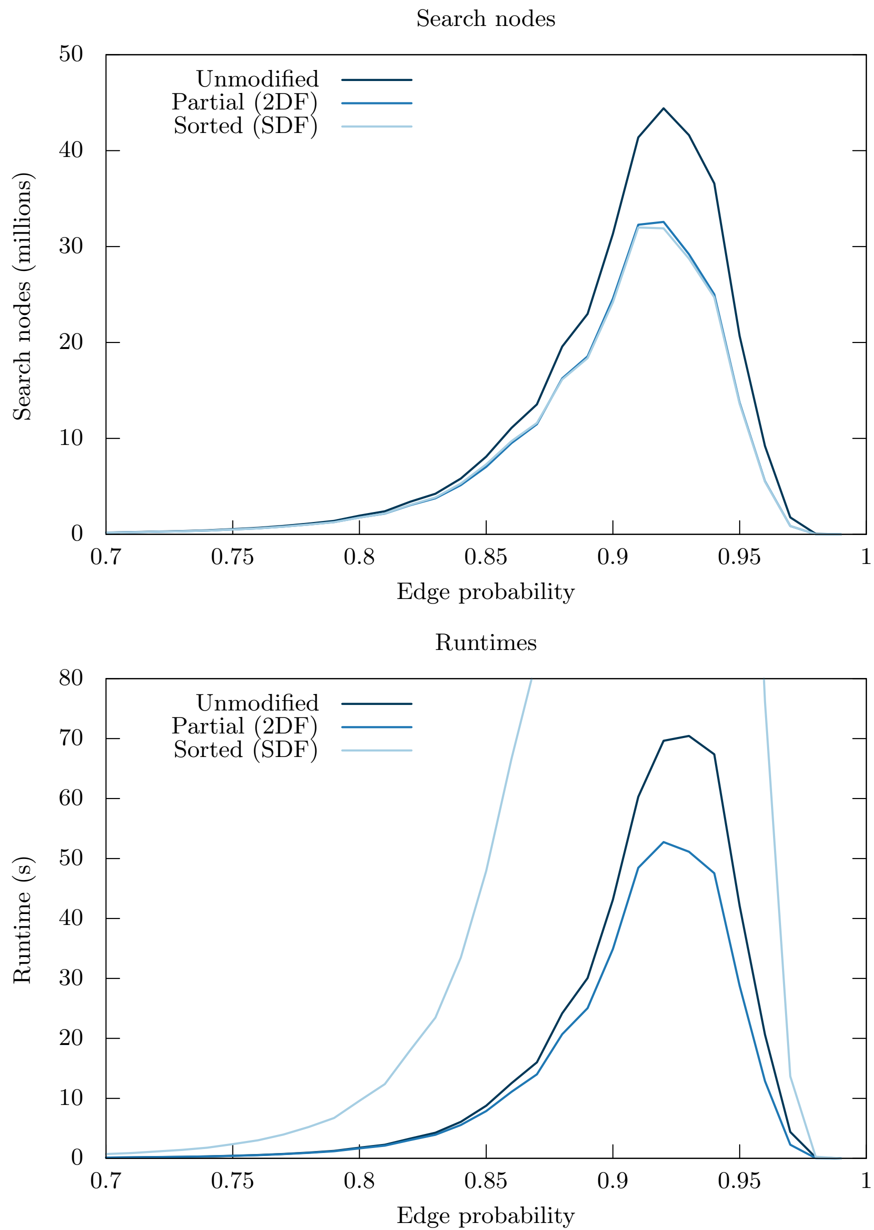

Fig. 5. The number of search nodes (top), and runtimes (bottom) for random graphs $G(200, x)$ with varying edge probabilities (steps of 0.01, 100 samples per probability). In the top graph, the partially sorted and sorted results are nearly indistinguishable, but on the bottom graph we see the high cost to runtime of doing a full sort.

two weeks to produce, are also omitted. However, our results include problems where the baseline runtimes are between less than a second to more than 9 days. For BHOSLIB, we have selected the smaller families, i.e. instances that we can solve within a few days.

Table 1 gives, for each problem instance, the size of the largest clique (ω) and then the performance of the baseline implementation, in number of search nodes and in runtimes. We then give the performance using partial sorting, and then full sorting. Performance is given as a percentage of the baseline.

We see that a full sort (SDF) gives a strict reduction in nodes (in all but 15 cases) but with a substantial increase in runtimes (in all but one instance, "san1000"). Partial sorting (2DF) gives a strict reduction in both search nodes and runtimes in all but 11 instances (those not shown in bold) and generally a reduction in nodes corresponds to a similar reduction in runtimes. The BHOSLIB ("frb" family) problems are worthy of note: these problems take between 5 minutes and 59 hours, and a partial sort gives improvements to both the search space size and runtimes of between ten and twenty five percent. A full sort here would give even more benefit, if it could be done economically, as it roughly halves the size of the search space in four of the instances. However, for some other graphs, a full sort actually gives a larger search space than a partial sort—this should not be a surprise, since a local improvement to the branching factor is not guaranteed to produce the best search tree globally.

4 Conclusion

Previously, in Tomita et al.'s maximum clique algorithms, the vertex selection rules and colourings were tightly coupled and not fully understood. Inspired by constraint programming techniques, we have provided a different way of looking at these algorithms. This allows us to treat the vertex selection and colouring processes separately, and to discuss this process in terms of variable and value ordering heuristics.

By looking inside the search process, we showed that the greedy colouring process produced vertices in an approximation of a "smallest domain first" order, which would reduce the amount of branching done locally at each step. We saw that using that order exactly would reduce the size of the search space in many cases, but that doing a full sort to obtain it had a large impact on runtimes. We introduced a partial sorting technique, and showed that this reduced both the size of the search space and runtimes. (Although not reported, we also investigated exploring colour classes in decreasing size and in the reverse order they were constructed. Both of these resulted in an increase in both nodes and runtimes.)

This is the first investigation into the effects of different vertex selection rules within Tomita's algorithms, and we have shown that genuine improvements can be made. A further benefit is that we now understand more about *why* these algorithms work so well in practice. Our decoupling also gives us more scope for improving the colouring process: previously, a different initial vertex ordering

Table 1. Experimental results for medium-sized DIMACS and smaller BHOSLIB graphs. Shown for each instance is the size of a maximum clique, then the number of search nodes (recursive calls) and runtime for the baseline algorithm. We then give the search space size and runtime for partial sorting, as a proportion of the baseline, and then the same for a full sort. Instances in bold are those where a partial sort gives a strict improvement in both nodes and runtimes. The omitted result takes more than two weeks to complete.

Instance	ω	**Unmodified**		**2DF** (%)		**SDF** (%)		**Instance**	ω	**Unmodified**		**2DF** (%)		**SDF** (%)	
		Nodes	Time	Nodes	Time	Nodes	Time			Nodes	Time	Nodes	Time	Nodes	Time
brock200_1	21	5.25×10^{5}	411 ms	93.8	96.4	87.5	526.5	**p_hat700-2**	44	7.51×10^{5}	3.2 s	95.7	95.0	205.8	683.1
brock200_3	15	1.46×10^{4}	11 ms	97.9	100.0	96.6	509.1	**p_hat700-3**	62	2.82×10^{8}	1677.9 s	95.3	94.3	324.1	1144.9
brock200_4	17	5.87×10^{4}	42 ms	96.4	97.6	83.2	476.2	**p_hat1000-1**	10	1.77×10^{5}	251 ms	99.6	97.6	100.7	278.1
brock400_1	27	1.98×10^{8}	287.8 s	97.0	99.1	85.3	388.2	**p_hat1000-2**	46	3.45×10^{7}	164.8 s	94.8	94.2	131.1	444.8
brock400_2	29	1.46×10^{8}	209.8 s	93.6	94.0	123.0	540.9	**p_hat1000-3**	68	1.30×10^{11}	225.8 h	93.8	98.8		
brock400_3	31	1.20×10^{8}	166.1 s	94.3	94.6	92.8	423.7	p_hat1500-1	12	1.18×10^{6}	3.6 s	99.7	102.5	87.4	177.9
brock400_4	33	5.44×10^{7}	80.6 s	92.2	93.4	193.1	832.8	**p_hat1500-2**	65	2.01×10^{9}	7.7 h	94.0	96.9	145.7	309.0
brock800_1	23	2.23×10^{9}	5325.8 s	97.3	95.1	104.6	326.0	san200_0.7_1	30	1.34×10^{4}	16 ms	100.1	106.2	52.4	243.8
brock800_2	24	2.24×10^{9}	5313.7 s	97.3	95.0	97.0	305.1	**san200_0.9_1**	70	8.73×10^{4}	100 ms	76.1	91.0	75.4	516.0
brock800_3	25	2.15×10^{9}	4924.1 s	97.3	95.5	80.2	255.6	san200_0.9_2	60	2.30×10^{5}	373 ms	340.9	303.2	316.8	1631.9
brock800_4	26	6.40×10^{8}	1851.7 s	97.9	95.5	93.8	289.2	**san200_0.9_3**	44	6.82×10^{6}	9.3 s	72.8	75.1	74.1	453.4
C125.9	34	5.02×10^{4}	47 ms	71.5	78.7	71.7	510.6	**san400_0.7_1**	40	1.19×10^{5}	234 ms	94.2	93.2	79.4	435.9
C250.9	44	1.08×10^{9}	1732.6 s	83.1	83.8	82.8	526.8	**san400_0.7_2**	30	8.89×10^{5}	2.1 s	89.9	91.1	52.8	186.8
C2000.5	16	1.82×10^{10}	21.4 h	98.9	100.2	95.6	170.4	**san400_0.7_3**	22	5.21×10^{5}	1.3 s	90.7	91.4	38.6	107.4
DSJC500_5	13	1.15×10^{6}	1.1 s	98.6	98.4	93.6	372.6	**san400_0.9_1**	100	4.54×10^{6}	24.2 s	78.8	79.5	73.9	335.5
DSJC1000_5	15	7.70×10^{7}	145.5 s	98.8	96.3	96.1	274.4	**san1000**	15	1.51×10^{5}	2.0 s	98.3	98.6	10.9	6.0
gen200_p0.9_44	44	1.77×10^{6}	2.7 s	80.2	83.2	87.3	536.0	**sanr200_0.7**	18	1.53×10^{5}	112 ms	95.6	97.3	93.8	546.4
gen200_p0.9_55	55	1.70×10^{5}	229 ms	86.2	89.1	85.9	533.2	**sanr200_0.9**	42	1.49×10^{7}	21.4 s	91.5	91.8	87.2	545.3
gen400_p0.9_65	65	1.76×10^{11}	126.5 h	59.5	61.5	58.1	279.5	**sanr400_0.5**	13	3.20×10^{5}	263 ms	98.5	98.5	93.3	397.7
gen400_p0.9_75	75	1.05×10^{11}	72.2 h	35.7	36.0	34.1	165.8	**sanr400_0.7**	21	6.44×10^{7}	75.5 s	95.4	95.5	94.2	426.5
hamming8-4	16	3.65×10^{4}	44 ms	100.9	100.0	60.0	270.5	**frb30-15-1**	30	2.92×10^{8}	677.8 s	88.6	89.0	86.8	369.4
johnson16-2-4	8	2.56×10^{5}	49 ms	100.0	106.1	88.8	551.0	**frb30-15-2**	30	5.57×10^{8}	1228.8 s	81.8	82.8	42.2	183.7
keller4	11	1.37×10^{4}	8 ms	98.7	125.0	84.1	425.0	**frb30-15-3**	30	1.67×10^{8}	375.2 s	80.7	82.2	78.5	333.7
keller5	27	5.07×10^{10}	44.1 h	108.8	113.2	69.7	239.5	**frb30-15-4**	30	9.91×10^{8}	2074.0 s	85.2	86.3	54.0	231.7
MANN_a27	126	3.80×10^{4}	274 ms	100.0	99.6	100.0	518.6	**frb30-15-5**	30	2.83×10^{8}	608.9 s	79.8	81.8	80.8	361.5
MANN_a45	345	2.85×10^{6}	250.0 s	100.0	100.8	100.0	206.9	**frb35-17-1**	35	1.33×10^{10}	14.8 h	82.7	88.4	44.9	154.1
p_hat300-3	36	6.25×10^{5}	1.1 s	92.9	94.7	109.6	562.3	**frb35-17-2**	35	2.34×10^{10}	26.1 h	82.3	88.8	48.8	171.6
p_hat500-2	36	1.14×10^{5}	266 ms	95.0	95.5	97.9	423.3	**frb35-17-3**	35	8.25×10^{9}	9.7 h	80.8	86.5	101.4	336.6
p_hat500-3	50	3.93×10^{7}	114.1 s	93.6	93.7	211.6	997.8	**frb35-17-4**	35	8.85×10^{9}	10.7 h	86.6	91.7	95.7	318.9
p_hat700-1	11	2.66×10^{4}	42 ms	99.4	97.6	106.3	285.7	**frb35-17-5**	35	5.80×10^{10}	58.9 h	76.5	82.2	71.0	249.4

or a more sophisticated colouring algorithm could have undesirable knock-on effects upon which vertex is selected first; we may now ignore these effects, or study them separately. This could make it easier to integrate Li et al.'s MaxSAT-inspired inference into these algorithms.

We stress that more advanced vertex selection rules must not come at the cost of greater runtimes: we saw this issue when a full sort gave a significant slowdown, despite the smaller search space. However, we saw that a very cheap partial sort was an excellent surrogate. We hope that with further investigation into these kinds of techniques, even greater gains can be had.

References

1. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1990)
2. Régin, J.C.: Using constraint programming to solve the maximum clique problem. In Rossi, F., ed.: Principles and Practice of Constraint Programming - CP 2003. Volume 2833 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg (2003) 634–648
3. Li, C.M., Quan, Z.: An efficient branch-and-bound algorithm based on MaxSAT for the maximum clique problem (2010)
4. Li, C.M., Quan, Z.: Combining graph structure exploitation and propositional reasoning for the maximum clique problem. In: Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on. Volume 1. (Oct 2010) 344–351
5. Li, C.M., Zhu, Z., Manyà, F., Simon, L.: Minimum satisfiability and its applications. In: Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume One. IJCAI'11, Palo Alto, CA, USA, AAAI Press (2011) 605–610
6. Li, C.M., Fang, Z., Xu, K.: Combining MaxSAT reasoning and incremental upper bound for the maximum clique problem. In: Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on. (Nov 2013) 939–946
7. Tomita, E., Seki, T.: An efficient branch-and-bound algorithm for finding a maximum clique. In: Proceedings of the 4th international conference on Discrete mathematics and theoretical computer science. DMTCS'03, Berlin, Heidelberg, Springer-Verlag (2003) 278–289
8. Tomita, E., Kameda, T.: An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. Journal of Global Optimization **37**(1) (2007) 95–111
9. Tomita, E., Sutani, Y., Higashi, T., Takahashi, S., Wakatsuki, M.: A simple and faster branch-and-bound algorithm for finding a maximum clique. In Rahman, M., Fujita, S., eds.: WALCOM: Algorithms and Computation. Volume 5942 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg (2010) 191–203
10. Okubo, Y., Haraguchi, M.: Finding conceptual document clusters with improved top-n formal concept search. In: Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence. WI '06, Washington, DC, USA, IEEE Computer Society (2006) 347–351

11. Konc, J., Janežič, D.: A branch and bound algorithm for matching protein structures. In Beliczynski, B., Dzielinski, A., Iwanowski, M., Ribeiro, B., eds.: Adaptive and Natural Computing Algorithms. Volume 4432 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2007) 399–406
12. Yan, B., Gregory, S.: Detecting communities in networks by merging cliques. In: Intelligent Computing and Intelligent Systems, 2009. ICIS 2009. IEEE International Conference on. Volume 1. (Nov 2009) 832–836
13. San Segundo, P., Rodríguez-Losada, D., Matía, F., Galán, R.: Fast exact feature based data correspondence search with an efficient bit-parallel MCP solver. Applied Intelligence **32**(3) (2010) 311–329
14. Fukagawa, D., Tamura, T., Takasu, A., Tomita, E., Akutsu, T.: A clique-based method for the edit distance between unordered trees and its application to analysis of glycan structures. BMC Bioinformatics **12**(Suppl 1) (2011) S13
15. Depolli, M., Konc, J., Rozman, K., Trobec, R., Janežič, D.: Exact parallel maximum clique algorithm for general and protein graphs. Journal of Chemical Information and Modeling **53**(9) (2013) 2217–2228
16. Regula, G., Lantos, B.: Formation control of quadrotor helicopters with guaranteed collision avoidance via safe path. Electrical Engineering and Computer Science **56**(4) (2013) 113–124
17. Konc, J., Janezic, D.: An improved branch and bound algorithm for the maximum clique problem. MATCH Communications in Mathematical and in Computer Chemistry (June 2007)
18. Prosser, P.: Exact algorithms for maximum clique: a computational study. Algorithms **5**(4) (2012) 545–587
19. Batsyn, M., Goldengorin, B., Maslov, E., Pardalos, P.: Improvements to mcs algorithm for the maximum clique problem. Journal of Combinatorial Optimization **27**(2) (2014) 397–416
20. San Segundo, P., Tapia, C.: Relaxed approximate coloring in exact maximum clique search. Computers & Operations Research **44**(0) (2014) 185 – 192
21. San Segundo, P., Rodríguez-Losada, D., Jiménez, A.: An exact bit-parallel algorithm for the maximum clique problem. Comput. Oper. Res. **38**(2) (February 2011) 571–581
22. San Segundo, P., Matia, F., Rodriguez-Losada, D., Hernando, M.: An improved bit parallel exact maximum clique algorithm. Optimization Letters **7**(3) (2013) 467–479
23. McCreesh, C., Prosser, P.: Multi-threading a state-of-the-art maximum clique algorithm. Algorithms **6**(4) (2013) 618–635
24. McCreesh, C., Prosser, P.: An exact branch and bound algorithm with symmetry breaking for the maximum balanced induced biclique problem. In Simonis, H., ed.: Integration of AI and OR Techniques in Constraint Programming. Volume 8451 of Lecture Notes in Computer Science. Springer International Publishing (2014) 226–234
25. Brockington, M., Culberson, J.C.: Camouflaging independent sets in quasi-random graphs. In: DIMACS series in discrete mathematics and theoretical computer science. Volume 26. (1996) 75–88
26. Soriano, P., Gendreau, M.: Tabu search algorithms for the maximum clique problem. In: DIMACS series in discrete mathematics and theoretical computer science. Volume 26. (1996) 221–242
27. Mannino, C., Sassano, A.: Solving hard set covering problems. Operations Research Letters **18**(1) (1995) 1 – 5

28. Marconi, J., Foster, J.: A hard problem for genetic algorithms: finding cliques in Keller graphs. In: Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on. (May 1998) 650–655
29. McCreesh, C., Prosser, P.: The shape of the search tree for the maximum clique problem, and the implications for parallel branch and bound. ArXiv e-prints (January 2014)
30. Haralick, R.M., Elliott, G.L.: Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence **14**(3) (1980) 263 – 313
31. Spearman, C.: The proof and measurement of association between two things. American Journal of Psychology **15** (1904) 88–103