An improved bit parallel exact maximum clique algorithm

Pablo San Segundo · Fernando Matia · Diego Rodriguez-Losada · Miguel Hernando

Abstract This paper describes new improvements for BB-MaxClique (San Segundo et al. in Comput Oper Resour 38(2):571–581, 2011), a leading maximum clique algorithm which uses bit strings to efficiently compute basic operations during search by bit masking. Improvements include a recently described recoloring strategy in Tomita et al. (Proceedings of the 4th International Workshop on Algorithms and Computation. Lecture Notes in Computer Science, vol 5942. Springer, Berlin, pp 191–203, 2010), which is now integrated in the bit string framework, as well as different optimization strategies for fast bit scanning. Reported results over DIMACS and random graphs show that the new variants improve over previous BB-MaxClique for a vast majority of cases. It is also established that recoloring is mainly useful for graphs with high densities.

Keywords Maximum clique · Branch and bound · Exact search · Graph

1 Introduction

A complete graph, or *clique*, is a graph such that all its vertices are pairwise adjacent. Finding a clique of a fixed size k is a well known and deeply studied NP-complete problem known as *k-clique* [1]. The corresponding optimization problem is known as the *maximum clique problem* (MCP), i.e. to find largest possible complete subgraph. MCP finds applications in many fields: bioinformatics and computational biology [2,3], computer vision [4], robotics [5] etc (cf. also chapter 7 of [6]).

Since MCP is NP-hard no efficient exact polynomial time algorithms are expected to be found. However, many efforts have been made at implementing fast algorithms in

P. San Segundo (✉) · F. Matia · D. Rodriguez-Losada · M. Hernando
Center of Automatic Control and Robotics-CAR (UPM-CSIC), Madrid, Spain
e-mail: pablo.sansegundo@upm.es

practice. One of the most successful paradigms for fast MCP algorithms is *branch-and-bound*, where a systematic enumeration of maximal cliques is pruned by bounding the size of the largest possible clique in the remaining subproblem. A good compromise between computational overhead and tight bounding is obtained through approximate sequential *vertex-coloring* (usually referred to as SEQ), as in [7–11], and two recent leading algorithms MCS [12] and BB-MaxClique [13].

BB-MaxClique is our previous effective branch and bound algorithm, reported to be the fastest, at the time of writing, for a large number of graphs. Its distinguishing feature is that it combines efficient bit string operations with the algorithmic improvement of keeping an initial non increasing degree ordering constant throughout the search. BB-MaxClique is a bit-parallel algorithm, which uses the ability of modern CPUs to compute bit masking operations in parallel the size of the ALU registers (typically 64). It performs well even for large sparse instances where the overhead introduced by bit scanning and bit masking operations reaches its peak.

Recently (and independently) a very fast new algorithm MCS has been described in [12], which combines the algorithmic improvements proposed in [13] (using conventional data structures) together with a new *recoloring* strategy (or *renumbering* as denoted by the authors). Reported tests clearly validate the combined improvements w.r.t. a previous algorithm MCQ [10] (and other state of the art algorithms) but the concrete contribution of recoloring is unfortunately not made explicit.

The remaining part of the paper is structured as follows: Sects. 2 and 3 deal with definitions and related work in the field. Sections 4 and 5 present the new improvements. Section 6 reports experiments and finally Sect. 7 summarizes the contribution.

2 Definitions and notation

A simple undirected graph $G = (V, E)$ consists of a finite set of vertices $V = \{v_1, v_2, \ldots, v_n\}$ and a finite set of edges E made up of pairs of distinct vertices ($E \subseteq VxV$). Two vertices are said to be adjacent (alias neighbors) if they are connected by an edge. For any vertex $v \in V$, $N(v)$(or $N_G(v)$ when the graph needs to be made explicit) denotes the neighbor set of v in G, i.e. the set of all vertices which are adjacent to v. $N_{\overline{G}}(v)$ refers to the set of non adjacent vertices to v, the neighbors in the complement graph. For any $U \subseteq V$, $G(U) = (U, E(U))$ is the subgraph *induced* over G by vertices in U.

A clique in G is any induced subgraph U with all its vertices pairwise adjacent. The largest clique in a graph is called the maximum clique. $\omega(G)$ and $\omega(U)$ refer to the sizes of the maximum clique in G or in an induced graph U respectively. UB_U (or simply UB when graph U is clear from the context) refers to any upper bound on the maximum clique size in U ($UB_U \geq \omega(U)$).

$\deg(v)$ is the degree of vertex v, i.e. the number of neighbor vertices. $\Delta(G)$ denotes the degree of the graph, the maximum degree of any of its vertices. $C(G) = \{C_1, C_2, \ldots, C_m\}$ is a feasible vertex *m-coloring* in G, made up of m independent color sets C_i; $C(U \subset V)$ refers to a coloring of the graph induced by vertex set U, a partial coloring in G. $c(v)$ denotes the color label of vertex v, so that for a given coloring $c(v) = k \Leftrightarrow v \in C_k$.

3 Related work

This section briefly summarizes related state of the art algorithms for exact maximum clique search.

3.1 Enumeration of maximal cliques

The general outline of branch and bound maximum clique exact search is a constructive enumeration of all maximal cliques, starting from a single vertex at the root node and gradually building larger and larger cliques until a leaf node is reached. Procedure RMCP illustrates the overarching idea and is based on [8] and [10].

Input: $U := G$, $C(G) = \{c(v_1), \ldots, c(v_n)\}$ $/c(v_i) = \min\{i, \Delta G\}$, $S = \phi$, $S_{\max} = \phi$
Procedure *RMCP* (U, $C(U)$, S, $S_{\max}$)
Output: A maximum clique of G in $S_{\max}$

Step 1: Select a vertex v with maximum color in U
Step 2: $U \leftarrow U/\{v\}$
Step 3: **if** $(|S| + c(v) \leq |S_{\max}|)$ **return** to previous level of recursion
Step 4: $S \leftarrow SU\{v\}$, $U_v \leftarrow U \cap N_U(v)$
Step 5: **if** $(U_v = \phi)$ **then** /*maximal clique*/
if$(|S| > |S_{\max}|)$**then** $|S_{max}| \leftarrow |S|$ and **goto** step 8
endif
Step 6: $C(U_v) \leftarrow COLOR(G(U_v))$
Step 7: *RMCP* $(U_v, C(U_v), S, S_{\max})$
Step 8: $S \leftarrow S/\{v\}$
Step 9: Repeat all steps from 1 to 9 until $U = \phi$

Search takes place in a graph space. RMCP uses two global variables S and $S_{\max}$, where S is the currently growing clique and S_{max} the largest clique found so far. It starts with empty set S and recursively adds vertices until it verifies that it is no longer possible to improve the current best solution in S_{max}.

U is the set of candidate vertices in a given step, initially the set of input vertices to the algorithm. At each step, a vertex $v \in U$ is selected (step 1) producing a new bigger clique $S_v = S \cup \{v\}$, with new candidate set $U_v = U \cap N_U(v)$, i.e. the remaining vertices adjacent to every vertex in S (step 4). The process is repeated until a leaf node is reached ($U_v = \phi$), when S is a maximal clique in G. For every maximal clique, RMCP computes $|S| > |S_{\max}|$ and the current best solution is updated accordingly if the premise holds. On backtracking, the algorithm iteratively picks remaining vertices in U until all possible maximal cliques have been enumerated.

3.2 Branch and bound approximate coloring

A good compromise between a tight upper bound for $\omega(U)$ and extra overhead is achieved by applying SEQ to U (COLOR in step 6). The coloring obtained prunes the child subproblem if $c(v) + |S| \leq |S_{max}|$ holds (step 3), since the current champion cannot be unseated by any maximal clique containing vertices in $SU\{v\}$.

We note that initial color labels use simple graph degree information rather than SEQ (i.e. $C(G) = \{c(v_1), \ldots, c(v_n)\}\, / c(v_i) = \min\{i, \Delta G\}$), so $C(G)$ might actually be infeasible for vertices v_j such that $\Delta G \geq j$. The algorithm remains complete anyhow because $c(v_j)$ is always an upper bound for any clique with vertices in $\{v_1, v_2, \ldots, v_j\}$.

Moreover, COLOR arranges the list of vertices by non increasing color on output, so that step 1 in the child subproblem can be computed in constant time. In practice, highest colored vertices are placed last in the list (and selected from last to first) so that the new candidate set in the child subproblem is $U_v = \{C_1 \cup C_2 \cup \cdots \cup C_k\}$.

3.3 Vertex ordering of approximate coloring

The bound obtained by COLOR strongly depends on the way vertices are ordered on input; in particular the bound improves on average if vertices are fed to COLOR by non increasing degree. Unfortunately, this is incompatible with the sorting by color required to select vertices in step 1 of RMCP.

In [11] the effect of arranging vertices by degree at each step prior to COLOR was analysed. It was concluded that the reduction of the search space was only effective when restricted to the shallowest levels of the search tree.

Ref. [13] reports BB-MaxClique, which fixes the initial non increasing degree order of vertices throughout the whole search. Underlying this key idea was the use of bit strings to encode G (its adjacency matrix) as well as the list of vertices in U at every step. With the help of bit-parallelism, the switch from color ordering (output of COLOR) to degree order (input to COLOR in the child subproblem) was achieved in constant time. Reported results showed that BB-MaxClique outperformed other leading algorithms for a large number of graphs.

Independently, a similar idea was described in algorithm MCS [12] and implemented with conventional data structures. The paper also proposed a new approximate coloring strategy (Re-NUMBER), which will be referred to as *recoloring*. Reported results for MCS include times for the combined two strategies, so the effect of recoloring cannot clearly be established.

3.4 Recoloring

Recoloring in [12] is based on the following property. For a given admissible *m-coloring* of a given graph $C(G) = \{C_1, C_2, \ldots, C_m\}$, it is possible to reassign a vertex $v \in C_k$ to a different independent set C_j, $j < k$, iff the following properties hold:

(1) There exists a vertex $w \in C_j$ such that $N(v) \cap C_j = \{w\}$, i.e. w is the only member of the neighbor set of v in C_j.
(2) There exists a color class C_l such that $|N(w) \cap C_l| = \phi$, i.e. does not contain any neighbor of w.

If this is the case, it is possible to produce a new admissible coloring by relabeling v and w so that $c(v) = j$ and $c(w) = l$.

When embedded in a maximum clique algorithm, recoloring is reported to be useful only if $j < l < k$ [12]. Moreover it introduces linear overhead on the size of U

and should only be applied selectively. A reasonable threshold for this is $k = k_{\min} = |S_{\max}| - |S|$, where $k_{\min}$ is the color label below which all vertices will be pruned in the derived child subproblem [11]. In steps where $j < l < k_{\min}$, both swapped vertices will end up with a lower label than k_{min} and therefore the subproblem hanging from v will be pruned. MCS [12] reported best performance when recoloring was applied dynamically during COLOR and restricted only to local maximum color vertices with a label higher or equal to k_{min}.

4 Recoloring in the bit parallel framework

In the paper, algorithms named starting by "BB_" refer to the use of bit string data structures which benefit from bit-parallelism. The concrete bit strings will be denoted by suffix *BB* when required in the context. Notation is taken from [13].

We have integrated (and enriched) recoloring as in [12] into BB-MaxClique by means of a new COLOR procedure BB-ColorR, based on previous BB_Color in [13]. BB_Color procedure in BB-MaxClique deviates from typical SEQ by obtaining a full color class at each iteration instead of labelling each vertex in turn. This presents a number of advantages related to the bit string framework which were described in [13] and still outputs the same coloring. A detailed description can be found there.

The computation of each color independent set in turn influences the outline of new BB_ColorR (described in detail at the end of the section), which runs as follows:

Step 1: For every uncolored vertex and threshold $k_{\min}$, determine color classes using previous BB_Color.
Step 2: Determine each new color class $C_{new}(k_{\min} \leq new)$ iteratively by the following operations:
Step 2.1: For each uncolored vertex v attempt to recolor (using new BB_ReCol).
Step 2.2: If successful, swap vertices and go back to step 2.1; Else label v with *new*.
Step 2.3: Remove all adjacent vertices to v from the remaining uncolored vertices.
Step 2.4: Go back to step 2.1

By first computing an initial set of color classes below $k_{\min}$, every additional candidate vertex, unless swapped, will be a local maximum color vertex in the new partial coloring, so all of them have potential to prune the search space if recolored successfully. Figure 1 describes the recoloring algorithm *BB_ReCol* in pseudocode.

BB_Recol receives as input an initial set $C_1, C_2, \ldots, C_{k_{\min}-1}$ of independent sets. These color classes can change throughout the search but are not subject to recoloring between themselves. BB_Recol also receives the remaining set of uncolored vertices which can still make part of the current color class (referred to as C_{new}) and the candidate vertex submitted to recoloring ($v \in C_{new}$). Step 2 evaluates the *only-one-neighbor* condition required in a color class $k_1$below $k_{\min} - 1$, while step 4 determines the swap if w can be upgraded to a neighborless color class k_2 such that $k_1 < k_2 < k_{\min}$.

Step 10 captures a further twist not mentioned in [12]. In SEQ admissible colorings, any vertex in a color class k must have at least one adjacent vertex in every lower color class (else it would have been labeled with that color in the first place);

Bit-paralell procedure $BB\_ReCol\ (v, k_{min}, C_1, C_2, \ldots, C_{k_{min}-1}, C_{new})$
Output: Recolors $v \in C_{new}$ if possible

1. **for** k_1:=1 to k_1:= k_{min}-2
2. **if** $|C_{k_1} \cap N(v)| = 1$ **then** w:= vertex in $C_{k_1} \cap N(v)$
3. **for** k_2:= k_1+1 to k_2:= k_{min}-1
4. **if** $C_{k_1} \cap N(w) = \phi$ **then** *//double swap*
5. $C_{new} := C_{new} \setminus \{v\}$
6. $C_{k_1} := C_{k_1} \cup \{v\}$
7. $C_{k_2} := C_{k_2} \cup \{w\}$
8. **endif**
9. **endfor**
10. **elseif** $C_{k_1} \cap N(v) = \phi$ **then** *//single swap*
11. $C_{new} := C_{new} \setminus \{v\}$
12. $C_{k_1} := C_{k_1} \cup \{v\}$
13. **endif**
14. **endfor**

Fig. 1 New *BB_ReCol* algorithm based on [12]. Inside boxes are computations which benefit from bit parallelism

as a consequence, condition $C_{k_1} \cap N(v) = \phi$ in step 10 should, in theory, never hold. However, this is not so if a previous color swap has occurred, because the absence of the upgraded vertex from C_{k_1} can now make the condition hold. This new possibility determines the *single-swap* condition of vertex v.

We note that this idea can also be extended to the swap movement of vertex w. Apparently w can only go *forward* to a higher color set ($k_2 > k_1$), but after the first swap it can also find a destination *backwards* ($k_2 < k_1$); the explanation is similar to *single-swap*. At present however, tests carried out extending iterations in step 3 below k_1 have not improved overall performance so they have not been reported.

The new COLOR algorithm BB_ColorR is shown in Fig. 2. The algorithm receives the bit string of vertices to be colored (S_{BB}) and returns them in a conventional list (S_L) ordered by increasing color labels ($C(S_{BB})$). Auxiliary bit string data structures are U_{BB}, the remaining uncolored vertices, and Q_{BB}, the candidate set of vertices which can still make part of the current color set C_k.

The rough version described at the beginning of the section attempts to recolor every vertex which is not in the initial color classes $C_1, C_2, \ldots, C_{k_{\min}-1}$. Since the overhead introduced by BB_ReCol is high w.r.t. its pruning ability, we tried a number of variants looking for the best compromise. The strategy which performed best on average selectively applies recoloring to vertices which can still improve pruning globally, i.e. produce an empty color class C_k and therefore lower the labels of all remaining uncolored vertices. This can only be achieved at steps where the current Q_{BB} is a singleton (evaluated in step 5).

Procedure *BB_ColorR* (S_{BB}, k_{min}, S_L, $C(S_{BB})$)
Input: S_{BB} (a bit string of vertices preserving initial degree ordering), k_{min}
Outputs: S_L a list of vertices ordered by SEQ coloring $C(S_{BB})$

0. $Q_{BB} := S_{BB}$; $U_{BB} := S_{BB}$; $k = 1$; *//initialization*
1. **while** $U_{BB} \neq \phi$
2. $C_k := \phi$;
3. **while** $Q_{BB} \neq \phi$
4. select the first possible vertex $v \in Q_{BB}$
5. **if** $k \geq k_{min}$ and $Q_{BB} = \{v\}$ **then**
6. *BB-ReCol* $(v, C_1, C_2, \cdots, C_{k_{min}-1}, Q_{BB})$
7. **else** $C_k := C_k \cup \{v\}$
8. **endif**
9. **if** v has not been recolored **then**
11. $Q_{BB} := Q_{BB} \cap N_{Q_{BB}}(v)$;
12. **endif**
13. **endwhile**
14. $U_{BB} := U_{BB} \setminus C_k$; *//k color set computed*
15. $Q_{BB} := U_{BB}$;
16. **if** $k \geq k_{min}$ **then**
17. store C_k in S_L in the same order
18. **endif**
19. **if** $C_k \neq \phi$
20. $k := k + 1$; *//next color*
21. **endif**
22. **endwhile**

Fig. 2 New *BB_ColorR* algorithm. Inside boxes are operations which benefit from bit parallelism

5 Bit string framework enhancements

The bit string framework allows for a number of optimizations. We mention two important changes that have been implemented w.r.t. BB-MaxClique. The first one is improved bit scanning during approximate coloring, unfortunately incompatible with current recoloring. The second one is the use of compiler intrinsics to compute basic bit string operations, also included in BB_ColorR.

5.1 Improved bit scanning during approximate coloring

Previous BB_Color [13] selected the next candidate vertex by bit scanning Q_{BB}, the bit string which encodes remaining uncolored vertices that could still make part of the current color set C_k. In practice, the algorithm did not explicitly use storage C_k, but dynamically used bit string Q_{BB} instead; i.e. at each iteration Q_{BB} stores vertices belonging to C_k and uncolored vertices non adjacent to all of them.

Fig. 3 First steps of the new formulation of BB_Color

1. **while** $U_{BB} \neq \phi$
2. **while** $Q_{BB} \neq \phi$
3. select *the first* vertex $v \in Q_{BB}$
4. $Q_{BB} := Q_{BB} \cap N_{\overline{Q_{BB}}}(v)$;
5. $U_{BB} := U_{BB} \setminus \{v\}$
6. **endwhile**
7. $Q_{BB} := U_{BB}$;
8. //...

The first steps of the actual implementation were:

1. **while** $U_{BB} \neq \phi$
2. **while** $Q_{BB} \neq \phi$
3. select *next* vertex v in Q_{BB}
4. $Q_{BB} := Q_{BB} \setminus N_{Q_{BB}}(v)$;
5. **endwhile**
6. $U_{BB} := U_{BB} \setminus Q_{BB}$;
7. $Q_{BB} := U_{BB}$;
8. //...

However, this approach has the disadvantage that bit scanning for a new vertex (step 4 in Fig. 2) requires extra overhead since it is not the first (least significant) bit in the bit string.

We have improved BB_Color by making bit scanning Q_{BB} a least significant bit procedure, as in the original description of BB_Color, but without requiring extra overhead and space for C_k(Fig. 3). This can be achieved by deleting from U_{BB} each selected vertex (step 5, Fig. 3) *after* computing the remaining possible candidates in step 4 (note that after step 4, $v \notin Q_{BB}$).

5.2 Intrinsics

Two basic instrinsic functions for 64 bit string operations are available for the Microsoft C++ compiler: population count (*__popcnt64*) and bit scanning (*_BitScanForward64*). Specifically, the latter has substituted previous DeBruijn magic number hashing implementation used in BB-MaxClique (cf. section 2.2 in [13]).

6 Experiments

This section reports a number of computational experiments in order to evaluate the bit string framework optimizations and the new recoloring strategy. We will refer to the former algorithm as BBMCI and BBMCR the latter.

Both algorithms have been implemented in C++. The computer employed for the report is an Intel(R) Core(TM) i7 CPU 950 @ 3.07GHz with a 64-bit Windows O.S. Tests were carried out over random graphs of varying sizes and densities as well as a subset of DIMACS benchmark graphs [14].

We also report calibrated comparison times with the two clearly leading algorithms at present: our previous BB-MaxClique [12] (which, for convenience, will be referred to as BBMC) and MCS [13]. Calibration times are established based on DIMACS benchmark program dfmax [14], see appendix.

Reported user times in tables are measured in seconds with a precision up to milliseconds. In all cases time was fixed to 3,600 s. Boldface times in both tables show the best time in each row.

6.1 Initialization

New BBMCI and BBMCR use the same initialization as BB-MaxClique. Vertices in V are initially ordered by non increasing degree and ties are broken randomly; the initial labelling described for RMCP ensures that they will be picked from last to first at the root node (for specific details, cf. [13]). MCS uses a more sophisticated tie-break strategy inherited from previous versions, but we have not found it significantly effective neither here nor in [13], so it has not been applied.

Initial values of U, C, S, and S_{max} are the same as for the reference procedure RMCP.

6.2 Results for random instances

Each row in Table 1 compares running times for uniform random graphs with different sizes (column n) and densities (column p). In these graphs there exists an edge with probability p for every pair of vertices. Each row reports performances of reference BB-MaxClique [13] (BBMC) and MCS [12], as well as new BBMCI, and BBMCR. Times are averaged over 10 instances for each pair (n, p).

In general, BBMCI outperforms previous BBMC for the majority of instances, the ratio of improvement increasing for the more difficult graphs (large and/or dense), where the number of bit masking and bit scanning operations is greater. For the easier graphs the effect is much less acute. Best ratio is achieved for the larger sparse graphs ($n \geq 5{,}000$), where BBMCI is fastest by a factor of more than 2.

Recoloring in BBMCR outperforms BBMCI only for high densities ($n \geq 0.8$) where phase transition does not make the instance trivial (as in (100, 0.95) or (200, 0.95–0.98)), but the improvement ratio is never more than double. For small to medium size graphs and lower densities there is no advantage w.r.t. BBMCR. Moreover, performance degrades with large graphs ($n \geq 5{,}000$). This might be explained partly because of the useless overhead during recoloring of bit masking and bit scanning operations over almost long empty bit strings. At the moment we are currently working on a new improvement for bit scanning and empty detection of sparse bit strings based on the *two-watched-literals* strategy used in SAT solvers [15].

With respect to MCS, either BBMCI or BBMCR are fastest, if only by a small margin, except in 4 cases. BBMCR is the fastest in graphs up to medium size and high densities, which make the most of recoloring in the bit string framework. In particular it is nearly twice as fast in (150, 0.95) and (200, 0.95). However, it is gradually outperformed by MCR as density decreases in the largest instances for

Table 1 User times over a set of random graphs

n	p	ω	BBMC [13]	BBMCI	BBMCR	MCS [12]
100	0.60	11–12	0.005	**<0.001**	**<0.001**	
100	0.70	14–15	0.002	**0.002**	0.0032	
100	0.80	19–21	0.008	0.005	**0.002**	0.004
100	0.90	29–32	0.013	0.003	0.003	**0.001**
100	0.95	39–48	0.013	0.002	**<0.001**	
100	0.98	55–67		**<0.001**	**<0.001**	
150	0.60	12–13	0.005	0.008	**<0.001**	
150	0.70	16–17	0.023	**0.015**	0.016	
150	0.80	23	0.144	0.091	**0.081**	0.126
150	0.90	35–38	0.798	0.510	**0.429**	0.548
150	0.95	50–54	0.172	0.129	**0.091**	0.192
150	0.98	74–86	**<0.001**	0.005	**<0.001**	0.003
200	0.60	13–14	0.038	**0.022**	0.025	
200	0.70	17–18	0.203	**0.155**	**0.155**	
200	0.80	25–26	2.914	1.66	**1.58**	2.47
200	0.90	40–42	45.9	33.40	**27.10**	40.55
200	0.95	60–64	48.331	29.00	**18.70**	32.33
200	0.98	90–100		0.292	0.232	**0.110**
300	0.60	15–16	0.398	**0.32**	0.351	0.548
300	0.70	20–21	6.20	**4.27**	4.66	6.58
300	0.80	28–29	206.5	205	**196**	215.91
500	0.50	13	1.545	**1.24**	1.29	1.534
500	0.60	17	22.43	**17.9**	19.9	21.92
500	0.70	22–23		956	**919**	
1,000	0.10	5–6	0.013	**0.006**	**0.006**	
1,000	0.20	7–8	0.074	**0.062**	0.070	
1,000	0.30	9–10	0.755	**0.627**	0.735	
1,000	0.40	12	9.181	7.33	8.4	**7.234**
1,000	0.50	15	216.7	174.1	200.0	**158.9**
5,000	0.10	7	3.574	**1.580**	4.58	1.808
5,000	0.20	9–10	202.7	**70.35**	260	75.62
10,000	0.10	7–8	75.43	**27.50**	87.9	32.89
15,000	0.10	8	394.6	**148.0**	507	179.2

Times are measured in seconds and averaged for 10 instances in each row. In boldface the best times in each row. Reported times for MCS [12] are based on a different set of random instances than the other three algorithms

the reasons previously explained. In general, recoloring is useless in the large sparse graphs altogether.

MCS is fastest for very dense graphs which have not reached phase transition to triviality, (i.e. (200, 0.98)) because recoloring in BBMCR is more selective than the one proposed in MCS. Also MCS is preferable, if only by a slight margin, in middle

sized graphs (i.e. $n = 1{,}000$) with relatively low density (not low enough for BBMCI, nor high enough for BBMCR).

6.3 Results for a subset of DIMACS graphs

Table 2 reports CPU user times required for the same 4 algorithms to solve a subset of the DIMACS benchmark graphs.

Table 2 User times over a set of DIMACS benchmark graphs

Name	n	p	ω	BBMC [13]	BBMCI	BBMCR	MCS [12]
brock200_1	200	0.745	21	0.405	0.327	**0.312**	
brock200_2	200	0.496	12	**<0.001**	**<0.001**	**<0.001**	
brock200_3	200	0.605	15	0.012	**<0.001**	**<0.001**	
brock200_4	200	0.658	17	0.063	**0.062**	0.063	
brock400_1	400	0.75	27	407.6	**341.0**	348.0	379.5
brock400_2	400	0.75	29	171.2	144.0	**140.0**	162.6
brock400_3	400	0.75	31	272.5	**229.0**	240.0	256.3
brock400_4	400	0.75	33	162.3	**133.4**	143.0	135.8
phat300-2	300	0.489	25	0.025	0.015	**<0.001**	
phat300-3	300	0.744	36	1.558	**1.25**	1.31	1.369
phat500-1	500	0.253	9	0.026	**<0.001**	**<0.001**	
phat500-2	500	0.505	36	0.494	0.39	0.39	**0.383**
phat500-3	500	0.752	50	103.3	**73.90**	76.10	82.10
phat700-1	700	0.249	11	0.063	**0.047**	**0.047**	
phat700-2	700	0.498	44	4.433	3.51	3.79	**3.07**
phat700-3	700	0.748	62	2,204	1,720	1,640	**1,309**
phat1000-1	1,000	0.245	10	0.367	**0.328**	0.421	
phat1000-2	1,000	0.49	46	221.0	187.0	193.0	**121.0**
phat1500-1	1,500	0.253	12	3.584	**3.23**	3.92	
hamming8-4	256	0.639	16	0.025	0.032	**0.015**	
hamming10-2	1,024	0.99	512	0.152	**0.031**	0.063	
johnson16-2-4	120	0.765	8	0.089	0.078	**0.062**	
keller4	171	0.649	11	0.013	0.016	**<0.001**	
Mann_a27	378	0.990	126	0.443	0.312	**0.187**	0.438
Mann_a45	1,035	0.996	345	196.7	144.0	**42.40**	153.9
san200_0.9_1	200	0.900	70	0.240	0.156	**0.094**	0.120
san200_0.9_2	200	0.900	60	0.152	0.109	**0.062**	0.219
san200_0.9_3	200	0.900	44	0.038	**0.015**	**0.015**	
san400_0.5_1	400	0.500	13	0.013	**0.016**	**0.016**	
san400_0.7_1	400	0.700	40	0.291	0.234	**0.125**	0.296
san400_0.7_2	400	0.700	30	0.114	0.078	**0.063**	0.071
san400_0.7_3	400	0.700	22	0.621	0.514	**0.437**	0.767

Table 2 Continued

Name	n	p	ω	BBMC [13]	BBMCI	BBMCR	MCS [12]
san400_0.9_1	400	0.900	100	0.190	0.141	**0.031**	0.055
san1000	1,000	0.502	15	0.709	0.686	**0.375**	1.150
sanr200_0.7	200	0.702	18	0.152	**0.125**	**0.125**	0.186
sanr200_0.9	200	0.898	42	22.53	18.20	**13.90**	22.45
sanr400_0.5	400	0.501	13	0.354	**0.297**	0.327	
sanr400_0.7	400	0.700	21	108.3	**91.00**	102.0	99.12
gen200_p0.9_44	200	0.900	44		0.328	**0.187**	0.257
gen200_p0.9_55	200	0.900	55		0.546	**0.437**	0.657
C125.9	125	0.900	34		0.031	**0.016**	
C250.9	250	0.899	44		1,650	**1,290**	1,784

Times are measured in seconds with precision up to milliseconds. In boldface the best times in each row

Compared to random graphs, BBMCR is now fastest in more instances because many of the DIMACS graphs have high density (i.e *san*, *sanr*, *gen*, *C* etc.). Best times are obtained in the denser instances such as *Mann_a45*, which is the fastest by a factor of more than 3.

MCS is fastest (although not by more than 25% in the worst case) in the *phat* family as density and size rises. This is consistent with the behaviour reported for random graphs, since the *phat* family is built using a generalisation of the classical uniform random graph generator. Finally, here BBMCI outperforms BBMC in all cases, except for the easy *brock200_2* instance. This validates the bit string framework improvements and encourages research for further optimizations along this line.

7 Conclusions

This paper improves our previous leading bit string maximum clique algorithm BB-MaxClique [13]. New BBMCI uses intrinsic functions for basic 64 bit population count and bit scanning operations, as well as reformulates approximate coloring for more efficient bit scanning of the list of vertices. BBMCR implements bit string recoloring based on [12] and introduces a new variant to improve the compromise between pruning and the extra overhead in the bit parallel framework.

Reported experiments show that recoloring improves performance only for high density graphs, an analysis that was lacking in [12], and also that BBMCI clearly improves over its predecessor on average. We also note that either BBMCR or BBMCI are fastest w.r.t. leading reference algorithm MCS in a large number of instances.

Besides the obvious practical importance of fast NP-hard algorithms, we believe this research is interesting also from a theoretical point of view, since we are not aware of *fully* encoded bit parallel algorithms for NP-hard problems that scale well with size, and this seems to be the case here.

Appendix

Calibration is established based on DIMACS benchmark program *dfmax* [14] and averaged over running times for random instances r100.5, r200.5, r300.5, r400.5 and r500.5. These were, for the computer used in the experiments, <0.001, 0.003, 0.203, 1.186 and 4.587 s respectively, 1.66 times faster than the computer used in [13], and 1.83 times faster than the one used in [12].

References

1. Karp, R.M: In: Miller, R.E., Thatcher, J.W. (eds.) Reducibility among Combinatorial Problems, pp. 85–103. Plenum, New York (1972)
2. Bahadur, D.K.C., Akutsu, T., Tomita, E., Seki, T., Fujijama, A.: Point matching under non-uniform distortions and protein side chain packing based on efficient maximum clique algorithms. Genome Inform. **13**, 143–152 (2006)
3. Butenko, S., Wilhelm, W.E.: Clique-detection models in computational biochemistry and genomics. Eur. J. Operat. Res. **173**, 1–17 (2006)
4. Hotta, K., Tomita, E., Takahashi, H.: A view invariant human FACE detection method based on maximum cliques. Trans. IPSJ **44**(SIG14(TOM9)), 57–70 (2003)
5. San Segundo, P., Rodríguez-Losada, D., Matía, F., Galán, R.: Fast exact feature based data correspondence search with an efficient bit-parallel MCP solver. Appl. Intel. **32**(3), 311–329 (2010)
6. Bomze, I.M., Budinich, M., Pardalos, P.M., Pelillo, M.: HandBook of Combinatorial Optimization. Supplement A. pp. 1–74. Kluwer Academic Publishers, Dordrecht (1999)
7. Wood, D.R.: An algorithm for finding a maximum clique in a graph. Operat. Res. Lett. **21**, 211–217 (1977)
8. Carraghan, R., Pardalos, P.M.: An exact algorithm for the maximum clique problem. Operat. Res. Lett. **9**, 375–382 (1990)
9. Östergård, P.R.J.: A fast algorithm for the maximum clique problem. Discrete Applied Mathematics **120**(1), 97–207 (2002)
10. Tomita, E., Seki, T.: An efficient branch and bound algorithm for finding a maximum clique. In: Calude, C., Dinneen, M., Vajnovszki, V. (eds) Discrete Mathematics and Theoretical Computer Science. LNCS, vol. 2731, pp. 278–289, Springer, Berlin (2003)
11. Konc, J., Janečič, D.: An improved branch and bound algorithm for the maximum clique problem. MATCH Commun. Math. Comput. Chem. **58**, 569–590 (2007)
12. 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 MS, Fujita S. (eds.) Proceedings of the 4th International Workshop on Algorithms and Computation. Lecture Notes in Computer Science, vol. 5942, pp. 191–203. Springer, Berlin (2010)
13. San Segundo, P., Rodriguez-Losada, D., Jimenez, A.: An exact bit-parallel algorithm for the maximum clique problem. Comput. Oper. Resour. **38**(2), 571–581 (2011)
14. Johnson, D.S., Trick, M.A. (eds.): Cliques, coloring and Satisfiability. DIMACS Series in Discrete Mathematics and Theoretical Computer Science 26. American Mathematical Society, Providence (1996)
15. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Chaff, S.M.: Engineering an efficient SAT solver. In: XXXVIII Proceedings of Design Automation Conference (DAC '01), pp. 530–535. ACM, New York (2001)