Computers & Operations Research 66 (2016) 81–94

Contents lists available at ScienceDirect

Computers & Operations Research

journal homepage: www.elsevier.com/locate/caor

A new exact maximum clique algorithm for large and massive sparse graphs

Pablo San Segundo [a,*], Alvaro Lopez [a], Panos M. Pardalos [b,c]

[a] Center for Automation and Robotics (UPM-CSIC), Jose Gutiérrez Abascal 2, Madrid 28006, Spain
[b] Center for Applied Optimization, University of Florida, 401 Weil Hall, P.O. Box 116595, Gainesville, FL 32611-6595, USA
[c] Laboratory of Algorithms and Technologies for Network Analysis, National Research University Higher School of Economics, 136 Rodionova, Nizhny Novgorod, 603093, Russian Federation

ARTICLE INFO

Available online 10 August 2015

Keywords:
Branch and bound
Bitstring
Massive
Sparse
Maximum clique

ABSTRACT

This paper describes a new very efficient branch-and-bound exact maximum clique algorithm BBMCSP, designed for large and massive sparse graphs which appear frequently in real life problems from different fields.

State-of-the-art exact maximum clique algorithms encode the adjacency matrix in full but when dealing with sparse graphs some form of compression is required. The new algorithm is based on a leading bit-parallel non-sparse solver but employs a novel sparse encoding for the adjacency matrix. Moreover, it also improves on recent optimizations proposed in literature for the sparse case such as core-based bounds.

Reported results show that it is several orders of magnitude better than state-of-the-art. Moreover, a number of real networks with many millions of nodes are solved in a few seconds.

1. Introduction

A simple undirected graph $G=(V,E)$ consists of a finite set of vertices $V=\{v_1,v_2,\ldots,v_n\}$ and edges E made up of pairs of distinct vertices ($E\subseteq V\times V$). Two vertices are said to be adjacent (or neighbors) if they are connected by an edge. For any vertex $v\in V$, $N(v)$(or $N_G(v)$ when the graph is explicitly specified) is the neighbor set of v in G. Any subset of vertices $U\subseteq V$ *induces* a new subgraph $G[U]$ with vertex set $U$and edge-set $G[E]\subseteq E$ such that all its edges have both endpoints in U.

In a complete subgraph, or clique, all vertices are pairwise adjacent. For a given graph, finding a clique of a fixed size k is a well known and deeply studied NP-complete problem referred to as *k-clique* [1]. The corresponding optimization problem is the *maximum clique problem* (MCP), which has the goal of finding the largest possible clique. The size of the maximum clique $\omega(G)$ is known as the *clique number* of the graph.

The *vertex coloring problem* (VCP) is another well known NP-complete problem very much related to the maximum clique problem (MCP); the VCP is well known to be an upper bound of the MCP. The goal of VCP is to find a *proper* color number assignment $c(v):V\to\mathbb{N}$ with the minimum number of colors. A *proper* assignment is such that all pairwise adjacent vertices have different color numbers, i.e. $v_2\in N(v_1)\Rightarrow c(v_2)\neq c(v_1)$. The notation $C(G)=\{C_1,C_2,\ldots,C_k\}$ in the paper refers to a proper coloring of *size* $|C(G)|=k$ (also known as a *k-coloring*), and color class C_i is the set of vertices with color number i. The size of a solution to the vertex coloring problem is known as the *chromatic number* of the graph $\chi(G)$.

Besides its theoretical value as an NP-hard problem, the maximum clique problem is known to have direct applications in a wide spectrum of fields such as data association problems appearing in bioinformatics and computational biology [2,3], computer vision [4] and robotics [5]. Typically the association problem is reduced to a maximum clique search in a *correspondence graph* which subsumes the matching criteria between two entities.

With the upsurge of web technologies there has also been renewed interest in cliques to capture structure over massive networks. For example, in social networks a clique can identify a group of cooperating agents (e.g. a terrorist cell); in the World Wide Web, cliques or quasi-cliques can help to detect frequently visited pages concerning a certain topic, clique kernels help to identify communities and so on.

In many of the above mentioned applications, networks tend to be *large* and *sparse* because relational models in life frequently involve many elements and show some form locality in their

* Corresponding author.
E-mail address: pablo.sansegundo@upm.es (P. San Segundo).

http://dx.doi.org/10.1016/j.cor.2015.07.013
0305-0548/© 2015 Elsevier Ltd. All rights reserved.

structure. The term *large* is employed rather loosely in literature; for example, Stanford's *large* network data set SNAP[1] contains real graphs ranging from 4000 to 96 million vertices. Graphs with millions of vertices are also referred to as *massive* or *monster* in the same informal way. Of interest to this work is the problem of finding a maximum clique in large and massive sparse graphs.

Exact algorithms for maximum cliques are profuse. State-of-the-art are currently branch-and-bound solvers which use approximate coloring to compute bounds for each subproblem, a very active line of the research in the last decade. The theoretical foundation for this is that the inequality $\omega(G) \le \chi(G) \le \Delta G + 1$ – ΔG denotes maximum graph degree – holds for any graph and follows trivially from [6]. Notable examples of this family of solvers in the past are [7,8] and in recent years [9-16]. [17] is a comparative study which claims that bit-parallel BBMC [13,14] performs best for a number of small and middle size graphs from public benchmarks; MCS [12] was the other most successful algorithm reported in that work.

The BBMC kernel uses bitstrings to encode the problem domain in such a way that critical operations, such as subproblem generation and coloring, are reduced to efficient bit masks. BBMC's careful design reduces bit-twiddling (e.g. vertex enumeration in a bit-encoded vertex set) as much as possible, something which is critical for overall efficiency.

Only very recently a number of improvements have appeared connected with more sophisticated approximate coloring techniques: BBMCL [15] improves BBMC using a selective coloring scheme and IncMaxCLQ [16] encodes each colored subproblem to MaxSAT and uses logical inferences to find subsets of vertices assigned k colors which cannot possibly make part of a *k-clique* (referred to as *inconsistent subsets*). In a similar vein is a variant of BBMC denoted as *infra-chromatic* which searches for a simpler form of inconsistent subsets without resorting to logical reasoning [18]. We note, however, that these more sophisticated techniques *are not useful for massive sparse graphs because it is known that the extra effort spent in pruning is not worthwhile in structures with low density* [14].

Many approximate methods for finding maximum cliques in massive graphs have been proposed in literature, as in [19-21]. Exact methods, however, are much less common as the problem quickly becomes intractable as the size of graphs increases. Here, the main line of research has focused on exploiting multi-core parallelism using a state-of-the-art single-core algorithm, such as BBMC in [22], for individual subproblems. Each subproblem is the workload of a different task.

Additional to these attempts is research on specific optimization techniques for massive sparse graphs. Particularly relevant is the fact that the majority of existing exact algorithms encode the adjacency matrix in full, which is the wrong approach for these types of networks. Two publicly available algorithms stand out to the best of our knowledge: PMC [23] and FMC [24]. Both use an adjacency list representation of the network and *unroll the first level of the search tree* to enforce early pruning. However, while FMC uses degree-based bounds PMC employs the much stronger notion of *core* as well as a more sophisticated recursive search procedure similar to the one used by BBMC. Moreover, the PMC web-site reports an excellent performance over a large data set of real graphs[2], so it is considered as reference in this paper.

Motivated by the above considerations this paper describes a new *very efficient* bit-parallel algorithm which uses a novel compressed representation of the bit-encoded adjacency matrix used by BBMC. Moreover, it also improves on the ideas described by PMC and adapts them to the new representation. Reported results in the paper show improvements of up to several orders of magnitude over a set of 276 publicly available real graphs.

The remaining part of the paper is structured as follows: Section 2 covers notation and a detailed outline of the reference bit-parallel exact maximum clique algorithm starting point of this research. Section 3 describes the new algorithm. Section 4 reports empirical validation and finally Section 5 presents some conclusions and summarizes contribution.

2. Previous work on exact bit-parallel maximum clique algorithms

This section gives a detailed outline of a state-of-the-art bit-parallel exact maximum clique solver. The description is somewhat extensive but presents the necessary background to understand the contribution of this paper.

2.1. Bit-parallelism

In this paper we take the view of a bitstring B_n as an encoding of a subset S of the set $[n]=\{0, 1, 2, 3, \ldots, n-1\}$ such that its elements map to 1-bits in B_n, i.e. for every $i \in S$, $B_n[i]=1$. For example, the bitstring encoding of subset $S=\{1,3,5,7\}$ of $\{0,1,2,\ldots,7\}$is 01010101. The *natural* data structure to encode B_n is not an array of bits, but an array of *blocks of bits* having the size of the register word w (typically 32 or 64 in today's commercial computers; unless otherwise stated $w=64$ is assumed).

By bit-parallelism we refer to the ability of a CPU to process simultaneously bitwise operations over data in two registers. If the data has bitstring semantics this can be viewed as a parallel computation with a speedup of O(w) with respect to a *classical* encoding.

Ground operations for bitstrings are LSB(B_n), the index of the least significant bit in B_n, MSB(B_n), the index of the most significant bit in B_n and POPCNT(B_n), the total number of elements (1-bits) in B_n. In the case of a set $S \subseteq [64]$, efficient implementations for these operators are available in many processors and accessible to programmers via assembly code or specific higher level libraries. The extension to B_n, $n>64$, requires ad-hoc software solutions with heavy optimization. One such solution, very much related to this paper, is the C++ BITSCAN library used by BBMC [25].

2.2. The bit-parallel framework

An efficient bit-parallel encoding for the exact maximum clique problem was first described in BBMC. The main data structures are the following:

- Input graph $G=(V,E)$, $|V|=n$: an array of n bitstrings B_n, each corresponding to a row of the adjacency matrix A. If B_n^i is the encoding of A_i then each 1-bit mapped to $[n]$ in B_n^i corresponds to a vertex adjacent to i.
- A set of vertices $U \subseteq V$: a bitstring B_n in which each 1-bit is the index of the corresponding vertex in $[n]$. Vertices not in U will always have their corresponding bit set to 0 in B_n.
- An induced graph $G[U]$: a bitstring B_n representing the set of vertices in U, together with A. Vertices not in U will always have their corresponding bit to 0 in B_n.

One important bottleneck operation for bit-parallel maximum clique is vertex-set enumeration. Unfortunately, maximum clique requires enumeration both during branching (the vertices of each subproblem) and coloring (the candidate vertices to enlarge the current color class). Implementation details related

[1] http://snap.stanford.edu/data/.
[2] https://www.cs.purdue.edu/homes/dgleich/codes/maxcliques/mctables.html

to efficient enumeration of 1-bits in B_n may be found in the BITSCAN library.

The remaining part of the section describes the reference algorithm of this work. Specifically, we highlight those aspects which relate to the proposed new algorithm.

2.3. The reference algorithm

Listing 1 describes SEARCH, an efficient exact bit-parallel maximum clique algorithm which roughly corresponds with BBMCI [14], an improved version of BBMC [13]. An important contribution of this paper is to tailor SEARCH for massive sparse graphs by compressing its bit-encoding. In this and the following subsection we present the necessary background to understand the new procedure; for a detailed description of BBMC and BBMCI we refer the reader to the original papers.

Relevant definitions and notations used in the listings are:

- U: list of vertices of the current subproblem. Vertices are always sorted according to the initial ordering.
- U_v: list of vertices of the child subproblem resulting from branching on vertex v.
- S: the current growing clique.
- S_{max}: the incumbent (best) clique found at any point during search.
- $L \subseteq U$: list of candidate vertices for branching in the current subproblem. Vertices are sorted by non-decreasing color number.
- $L_v \subseteq U_v$: analogous to L but for the child subproblem resulting from branching on vertex v.
- $c(v)$: the color number assigned to vertex v.
- k_{min}: a pruning threshold; vertices assigned a lower color number in any child subproblem U_v will be pruned.
- *greedy sequential coloring procedure* (SEQ): an approximate coloring heuristic which assigns to each vertex in a predefined order the lowest possible color number consistent with the current partial coloring.
- *class coloring*: an independent set implementation of SEQ which produces *color sets sequentially* in increasing order (also known as *greedy independent set coloring*).
- $C(U)$: class coloring of the subgraph induced by U. U may be removed from the notation when it is clear from the context.
- C^v: class coloring of the subgraph induced by the neighborhood of v, i.e. $C(U_v)$.
- *COL*: current color set being constructed in a class coloring procedure.
- *UNCOL*: set of remaining uncolored vertices in a class coloring procedure.

Other definitions concerning sorting of vertices used in the paper are:

- *width of vertex* v_i: $w(v_i) = |N(v_i) \backslash \{v_{i+1}, v_{i+2}, \ldots, v_n\}|$, i.e. the number of preceding vertices adjacent to v_i;
- *width of a vertex ordering*: the maximum width of any of its vertices;
- *minimum-degree-last ordering*: a *minimum width* ordering of vertices obtained by iteratively *removing* vertices with minimum degree and *placing them in reverse order* in the new list V'. More formally, let $V = \{v_1, v_2, \ldots, v_n\}$ be the set of vertices sorted according to *minimum-degree-last*. Then v_n is a vertex of minimum degree in G, v_{n-1} has minimum degree in $G \backslash \{v_n\}$, v_{n-2} in $G \backslash \{v_n, v_{n-1}\}$ and so on.

SEARCH is a branch-and-bound recursive procedure which combines enumeration of maximal cliques (each branch of the search tree) with pruning based on approximate vertex coloring. At each step (i.e. a new call to SEARCH) it selects a vertex v from the branching set L (step 2) and adds it to the current clique S (step 5). The child subgraph $G(U_v)$ induced by the neighbors of v in the current subproblem U is computed efficiently as a bitset-intersection in step 6:

$$U_v \leftarrow U \cap N_U(v) \tag{1}$$

Iterative application of (1) ensures that for every node of the search tree vertices in U are pairwise adjacent to *all* vertices in S (and not just to the last vertex added). The process continues until a leaf node is reached ($U_v = \phi$), when S is a maximal clique in G. For every branch the algorithm checks if S can become the incumbent clique and updates S_{max} if appropriate (step 8). On backtracking, it removes the last vertex added to S (step 15) before branching on a new vertex from L.

Pruning of the search space occurs if the current subproblem is found unable to improve the incumbent solution (condition $|S| + c(v) \le |S_{max}|$ in step 3. The actual coloring for the child subproblem $C^v$is computed in COLOR, and is described in Listing 2. COLOR receives as input the child subgraph $G_v = G[U_v]$ to be colored and a pruning color threshold $k_{min} = \max(|S_{max}| - |S|, \ 1)$. Besides the coloring, it also outputs the new branching list L_v for the child subproblem.

k_{min} was first described explicitly in [11] and represents the minimum color number of any candidate vertex that should be considered for branching in the child subproblem U_v. The explanation is that vertices with $c(v) < k_{min}$ will be cut in the pruning step of the main procedure (step 3, Listing 1) so, before this happens, *it is more efficient to prune them implicitly inside* COLOR (steps 7–10 of Listing 2). In SEARCH pruned vertices will not make part of the branching list L of the new subproblem.

COLOR implements sequential coloring as class coloring using two nested loops and two auxiliary bitsets *COL* and *UNCOL*. The former set contains the current color class assignment and the latter the remaining uncolored vertices. The outer loop keeps track of the color number k of the current color class assignment and initializes *COL* with the remaining vertices to be colored in *UNCOL* (step 2). The inner loop computes each color class k (steps 3–11) by iteratively adding a new vertex each time (step 4). Note that the main coloring computation is conceived as an efficient bit-set-difference operation between *COL* and the neighbor set of v (step 5).

We end this description with a brief note on initialization of data structures. Initial values for $C(V)$are typically not tight: i.e. in the listing, SEARCH simply assigns vertices the minimum number between their index and maximum graph degree incremented by one. In contrast, initial sorting *is* critical. A well known general strategy at the root node is to branch on vertices with smallest degree first. In practice, vertices are sorted according to minimum-degree-last heuristic and taken *in reverse order* (step 2, Listing 1).

Initial sorting becomes *even more* critical for massive graphs of interest to this paper. In the proposed new algorithm, the recursive search procedure is called only after a *preprocessing* step which determines, amongst other things, the initial ordering and color numbers of each particular subproblem fed to SEARCH. Preprocessing builds on *core* analysis. We relate the minimum-degree-last heuristic with cores in Section 3.3.

Listing 1. Reference maximum clique algorithm. The box marks a critical operation which benefits from bit-parallelism.

```
Input: A simple graph $G=(V,E)$ sorted by minimum-degree-last
Output: A maximum clique in $S_{max}$
Initial values: $U \leftarrow V,\ S \leftarrow \phi,\ S_{max} \leftarrow \phi,\ c(v_i) \leftarrow \min\{i,\ \Delta G+1\},\ L \leftarrow V$
SEARCH ($U, S, S_{max}, C, L$)
1.  repeat until $L=\phi$
2.     select a vertex $v$ from $L$ in reverse order        //maximum color branching
3.     if ($|S|+c(v) \le |S_{max}|$) then return             //pruning step
4.     $U \leftarrow U \backslash \{v\}$
5.     $S \leftarrow S \cup \{v\}$
6.     $U_v \leftarrow U \cap N_U(v)$                          //$U_v$ always preserves initial order of vertices
7.     if $U_v=\phi$ then                                   //maximal clique found (leaf node)
8.        if $|S|>|S_{max}|$ then $S_{max} \leftarrow S$
9.        $S \leftarrow S \backslash \{v\}$
10.       continue
11.    endif
12.    $k_{min} \leftarrow \max(|S_{max}|-|S|,\ 1)$
13     COLOR ($U_v, k_{min}, C^v, L_v$)                    //returns coloring $C^v$ and list $L_v$ for branching
14.    SEARCH ($U_v, S, S_{max}, C^v, L_v$)
15.    $S \leftarrow S \backslash \{v\}$
```

Listing 2. COLOR procedure. Boxes mark critical operations which benefit from bit-parallelism.

```
Input: Induced graph $G_v=G[U_v]$, color threshold $k_{min}$
Output:   1) A coloring $C$ of $G_v$;
          2) A list $L$ of vertices sorted according to increasing color number
COLOUR ($U_v, k_{min}, C, L$)
Initial step: $UNCOL \leftarrow U_v,\ COL \leftarrow \phi,\ L \leftarrow \phi,\ k \leftarrow 1,\ C \leftarrow \phi$
1.  repeat until $UNCOL=\phi$
2.     $COL \leftarrow UNCOL$
3.     repeat until $COL=\phi$
4.        $v \leftarrow$ take the next vertex from $COL$
5.        $COL \leftarrow COL \backslash N(v)$                   //main coloring computation
6.        $UNCOL \leftarrow UNCOL \backslash \{v\}$
7.        if ($k \ge k_{min}$) then
8.           $c(v) \leftarrow k$
9.           $L \leftarrow L \cup \{v\}$                        //L is sorted by increasing color number
10.       endif
11.    endrepeat
12.    $k \leftarrow k+1$                                     //index of a new color set
13. endrepeat
```

2.4. Implementation details

In this section we discuss implementation details of the bit-encoding employed by SEARCH relevant to this paper. Before this we mention two important decision rules which are necessary background:

I. *Branching-on-highest-color*: Vertices with higher color numbers are selected first. This was first described in MCQ algorithm [10].
II. *The order of vertices remains as determined initially in every subproblem*: Sequential greedy coloring is known to produce reasonably tight colorings when vertices are ordered by non-increasing degree (in fact, colorings not greater than $\Delta G+1$, a property known as the *Welsh–Powell rule* [26]). As explained, initial sorting uses minimum width which tends to place vertices with high degree first. Thus, this rule improves pruning without a high computational effort. It was first described independently in BBMC and MCS [12].

With the previous background it is now possible to discuss implementation details of SEARCH related to bit-parallelism. The main points may be summarized as follows:

- The branching list L is *not* bit encoded because it requires enumeration. Neither are S and S_{max} vertex sets.
- Computation of each new subproblem U_v (step 6, Listing 1) benefits from bit-parallelism implemented as set-intersection.
- The main coloring computation (step 5, Listing 2) benefits from bit-parallelism implemented as set-difference.
- Enumeration of vertices to determine a new color assignment in COLOR is unavoidable (step 4, Listing 2). This bit-scanning operation is critical for overall efficiency. We refer the reader to the source code of BITSCAN [25] for concrete implementation details.
- Auxiliary sets *COL* and *UNCOL* in COLOR are bit-encoded.
- *Branching-on-highest-color* (Rule I) is implemented by selecting vertices in reverse order from L. Previously COLOR fills *L in the same order color sets are produced*, so vertices placed last in L always have the highest colors.

– *Fixed order of vertices in all subproblems* (Rule II) is implemented implicitly by the bitstring encoding, since bit-parallel masking operations are order-invariant. Once the initial sorting is computed, *a new bit-encoded graph* G′ *is constructed with the new ordering.* G′ is the actual input to SEARCH.

2.5. Motivation

SEARCH described previously is an efficient algorithm for small and middle size graphs. It does not scale well for large and massive graphs because the adjacency matrix is stored in full, a common occurrence in the vast majority of efficient exact branch-and-bound maximum clique algorithms in literature. Addressing real graphs requires some form of compression to reduce memory requirements as well as spurious operations. Noteworthy existing algorithms mentioned in Section 1 are PMC and FMC. Both use row compression in the form of an edge list partitioned in $|V|$ subsets, so that each subset j contains $N(v_j)$.

The new algorithm BBMCSP (acronym for BBMC SParse) presented in this paper takes a different view: *it compresses the bitstring encoding of the adjacency matrix* so that each new row of the new adjacency matrix is a sparse representation of the old one.

Table 1
Bit (B_n) and bit-sparse (BS_n) encoding of graph $G=(V,E)$ with vertex and edge sets $V=\{1,2,\ldots,128\}$ and $E=\{(1,2),(1,3),(2,3),(1,128)\}$respectively. Each row in the table corresponds with a row of the adjacency matrix.

Row	Type	Elem 1 (1–64)	Elem 2 (65–128)	Type	Elem 1 (1–64)	Elem 2 (65–128)
1	$B_{\|V\|}$	0110…0	0…01	$BS_{\|V\|}$	[1], 0110…0	[2], 0…01
2	$B_{\|V\|}$	1010…0	0…00	$BS_{\|V\|}$	[1], 1010…0	x
3	$B_{\|V\|}$	1100…0	0…00	$BS_{\|V\|}$	[1], 1100…0	x
4–127	$B_{\|V\|}$	0000…0	0…00	$BS_{\|V\|}$	x	x
128	$B_{\|V\|}$	1000…0	0…00	$BS_{\|V\|}$	[1], 1000…0	x

Initial tests to encode elements as $<$index, block$>$ pairs of an associative container did not yield the desired efficiency results. BBMCSP encodes BS_n bitsets as an *ordered-by-index array* of B_{64} blocks. Listing 3 shows a possible declaration of BS_n in C++ (the source code is also available as part of the BITSCAN library).

Unfortunately, this compressed representation introduces additional complexity as individual bit-blocks cannot be accessed in constant time.

Listing 3. An example of a sparse bitstring declaration in C++

```
typedef unsigned long long BITBLOCK;
struct element{
   unsigned int block_index;
   BITBLOCK b64;
};
vector < element > BS;              //a sparse bitstring; elements are always be sorted by block_index
```

A similar sparsification is also applied to those (bit encoded) vertex sets in SEARCH (such as U_v, *COL*, *UNCOL*) involved in operations marked as critical in Listing 1 and Listing 2.

Moreover BBMCSP also improves the initial preprocessing step described in PMC taking into consideration the new encoding. The result is a boost in performance of up to several orders of magnitude for a wide set of publicly available real graphs. The following section describes BBMCSP in detail.

3. Maximum clique for large sparse graphs

This section describes the new algorithm BBMCSP. It is structured in five parts: Sections 3.1 and 3.2 introduce the new encoding and the relevant implementation details concerning the recursive search procedure. Section 3.3 introduces the notion of *core* and its relevance as a pruning strategy for massive sparse graphs. Section 3.4 provides pseudocode listings of the algorithm. Finally the last subsection presents an illustrative example.

3.1. The sparse bit-encoding

We now extend bitstring notation B_n to BS_n, and denote it as a *sparse bitstring*, in a natural way. BS_n refers to an encoding of a subset S of $[n]=\{0, 1, 2, 3, \ldots, n-1\}$ such that its elements map to 1-bits in BS_n *and only blocks of bits with at least one element from S make part of the bitstring.* As a consequence, every element (bit block) of BS_n is now made up of the two numbers:

– A non-zero B_{64} block.
– The index of the block in B_n.

Related to its application in SEARCH, critical operations over vertex-sets are:

– *Insertion of individual vertices* (e.g. step 5 of Listing 1): If the corresponding B_{64} block is present, this is a logarithmic operation. If, on the other hand, the block has to be inserted as well the BS_n array will need to shift blocks with higher index in memory.
– *Deletion of individual vertices*: As in the previous case, finding the corresponding block is a logarithmic operation; but unlike insertion no block-shifting is required if the block is not present. We note that both deletion and insertion take constant time in B_n.
– *Set-intersection and set-difference*: These operations have additional overhead in BS_n because only blocks with the same index can be masked. On the other hand, massive sparse bitsets will contain much fewer blocks than its B_n counterpart.
– *Enumeration of elements in the set*: This is similar in complexity to B_n as vertices are typically enumerated in order.

The efficiency of a *sparse bitstring-based* maximum clique algorithm will depend on how it deals with the above considerations. The next subsection provides the relevant implementation details of the proposed new algorithm. This discussion is postponed until then.

Concerning the graph encoding, it is similar to the non-sparse case except that now neighbor vertex sets (the rows of the adjacency matrix) are sparse bitsets. Table 1 shows an example of the new sparse encoding of a graph. As can be seen, the empty bit blocks in B_n have disappeared in BS_n (in the cells marked with a cross), but the latter encoding now stores block indexes (numbers in brackets).

Table 2
Complexity analysis and related implementation details of critical steps in the new algorithm. SEARCH is the maximum clique procedure described in Listing 1. COLOR corresponds to Listing 2.

Procedure	Step	Operation	Computation	O(B)	O(BS)	Comments
SEARCH	2	Select a new vertex from L	Enumeration	$O(1)$	$O(1)$	L is *not* bit-encoded
SEARCH	4	$U \leftarrow U \backslash \{v\}$	Single deletion	$O(1)$	$O(\log n)$	The corresponding block has to be found. *Resulting empty bit-blocks are not removed*
SEARCH	5	$S \leftarrow S \cup \{v\}$	Single insertion	$O(1)$	$O(1)$	S is *not* bit-encoded
SEARCH	6	$U_v \leftarrow U \cap N_U(v)$	Set intersection	$O(n)$	$O(n)$	Ordered enumeration of blocks
SEARCH	14	$S \leftarrow S \backslash \{v\}$	Single deletion	$O(1)$	$O(1)$	S is *not* bit-encoded
COLOR	2	$COL \leftarrow UNCOL$	Set insertion	$O(n)$	$O(n)$	Ordered enumeration (and copy) of blocks
COLOR	4	Select a new vertex to color from COL	Enumeration	$O(n)$	$O(n)$	Ordered enumeration of vertices as part of a bit-scanning loop
COLOR	5	$COL \leftarrow COL \backslash N(v)$	Set difference	$O(n)$	$O(n)$	Ordered enumeration of blocks.
COLOR	6	$UNCOL \leftarrow UNCOL \backslash \{v\}$	Single deletion	$O(1)$	$O(\log n)$	The corresponding block has to be found. *Resulting empty bit-blocks are not removed*

3.2. *The recursive search procedure*

We discuss in this section the main recursive procedure of the new algorithm, referred to as SPARSE_SEARCH. The outline is similar to SEARCH described in Listing 1 and Listing 2 but solutions must be provided to reduce the overhead of critical set-operations in the new representation. Table 2 summarizes complexity analysis and gives additional implementation details when required. Each row is a step of the algorithm.

The table shows an important increment in complexity for the case of individual deletion of vertices, because the particular BS block needs to be found in the sparse set-encoding. This is countered by the fact that *in large sparse graphs much fewer blocks are to be expected*. Noteworthy is the fact that *individual insertion of vertices* – more expensive than deletion – *is not required*. Also worth noting is that parameter n in the complexity analysis is the constant $\lceil |V|/w \rceil$ for all sets in the non-sparse case, whereas in a sparse set it is the actual number of B_{64} blocks present. The best case for the sparse representation is the empty set, mapped to an empty array, while in B_n it still requires $\lceil |V|/w \rceil$ blocks.

Another important implementation detail is to enforce the non-empty-block condition for BS_n *only at the moment of creation*. For example, a new generated subproblem U_v (output of step 6 of SEARCH) will never have empty blocks. However, after a number of branching operations empty blocks may appear. Removing them is inefficient because all blocks with higher indexes would have to be shifted as well. They *will* be eliminated, however, if the set is again initialized, and this happens actually quite often: i.e. in step 6 of SEARCH, as well as in the initial step and step 2 of COLOR for sets U_v, $UNCOL$ and COL respectively.

In the remaining steps reported in Table 2 blocks are processed in order so complexity in both encodings is comparable. As already mentioned, in a set-intersection or set-difference operation between bitsets A and B, blocks must be *aligned by index* to perform the masking operation. The easiest solution is to alternatively run through both input sets depending on the relative value of indexes. If the index of a new block in A is greater than its counterpart then B takes control and vice versa. Our procedure runs in linear time in the total number of blocks of both sets. Finally, we note that enumeration cannot be avoided in step 4 of COLOR. Again we refer the reader to the source code in BITSCAN [25] for concrete implementation details concerning fast bit-scanning in BS_n.

3.3. *Cores*

The notion of *core* was introduced by Seidman in [27] and is *very important* for large scale graph analysis and specifically maximum clique analysis. We will explain why this is so after introducing the necessary background.

Notation and definitions related to core analysis of interest to this paper are:

- *k-core* (alias *core of order k*): a subgraph H such that all its vertices have degree at least k and H is maximal with this property ($\max_H\{H \subseteq V \mid \forall v \in H : \deg_H(v) \geq k\}$).
- *core number of a vertex* ($K(v)$): the core of the highest order to which that vertex belongs.
- *core number of a graph G* ($K(G)$): the highest order core of any of its vertices. Also known as *degeneracy* of the graph. In the notation, G may be substituted by a set of vertices in $K(G)$ to refer to the core number of the subgraph induced by the set.
- *degeneracy ordering of G*: the ordering that results by sorting vertices according to non-increasing core number [28]. *It is also a minimum width ordering.*

The outline of the algorithm which achieves core decomposition and, at the same time, degeneracy ordering is the similar to the one described for minimum-degree-last sorting in Section 2.3, but with an important difference. In the case of minimum-degree-last, when a vertex with minimum degree is removed and placed preceding the last sorted vertex in the new list, the degrees of the remaining vertices adjacent to the removed vertex are decreased by one. In the case of degeneracy ordering, *the updated number for each remaining vertex is bounded below by the number of the vertex currently picked* (its core number). Listing 4 outlines how degeneracy is computed.

State-of-the-art exact algorithms for the maximum clique problem report several detailed descriptions of initial sorting as minimum width (e.g. [12,29]), but almost never refer to core-based orderings. The reason is that cores tend to be of high order in dense small or middle size graphs and become useless as clique bounds. One example is the *brock200_1* graph from the *Second DIMACS Challenge* [30] in which all vertices belong to a single core of order 134 whereas the clique number of the graph is only 21. The situation is reversed in massive sparse graphs: core numbers of vertices can be very tight bounds for the clique number and the core number of the graph is in general much smaller than its maximum degree. Moreover, *an algorithm which computes k-cores in $O(|E|)$ is known* [31]. We note that a straightforward implementation of degeneracy sorting in listing 4 will run in $O(|V|^2)$ which is impractical for massive graphs. We use the algorithm described in [31] to compute cores in BBMCSP. C++ source code for the actual implementation has been released by one of the authors as part of GRAPH [32], a small library which manages bit-encoded graphs, both general and sparse.

Of interest to this work is that the core number of a graph may be used as a maximum clique bound. On general grounds, all bound expressions which involve degrees may be substituted

Listing 4. Outline of degeneracy vertex ordering of a graph.

```
Input: a simple undirected graph $G(V,E)$
Output: the input graph $G$ with vertices sorted according to degeneracy
DEGENERACY ($G$)
Initial steps:
i1. $D=\{d(v), v\in V \mid d(v)\leftarrow \deg(v)\}$          //initialized with vertex degree
i2. $U\leftarrow V,\ V'\leftarrow \phi$
i3. Order $U$ by non-decreasing vertex degree
1.  repeat until $U=\phi$
2.    select (and remove) the next vertex $v$ from $U$
3.    place $v$ in reverse order in $V'$
4.    for every vertex $w\in N_U(v),\ d(w)>d(v)$
5.      $d(w)\leftarrow d(w)-1$
6.    endfor
7.    sort $N_U(v)$ in $U$ by non-decreasing $d(v)$          //critical step for efficiency
8.  endrepeat
9.  sort $G$ according to $V'$
10. return $G$
```

freely by core numbers. Showing that $K(G)\le \Delta G$ is trivial since the core number of any vertex is always bounded by its degree. It is also easy to see that if a vertex v belongs to a core of order $k-1$ it cannot be part of a clique larger than k. If this were so all vertices in that clique would have at least k neighbors, and therefore a core of order at least k containing v exists which contradicts the initial hypothesis. Therefore $\omega(G)\le K(G)+1\le \Delta G+1$.□

Concerning sequential greedy coloring SEQ, it is known from [26] that if vertices are sorted by non-increasing degree, i.e. $\deg(v_1)\ge \deg(v_2)\ge \ldots \ge \deg(v_n)$, then the size of the coloring $|C(G)|$ cannot be greater than $\Delta G+1$. This result may be extended to cores: the size of a SEQ coloring *when vertices follow degeneracy ordering*$K(v_1)\ge K(v_2)\ge \ldots \ge K(v_n)$ cannot be greater than $K(G)+1$. It also immediately follows that $\omega(G)\le |C(G)| \le K(G)+1\le \Delta G+1$.□

3.4. The new algorithm

This section provides the pseudocode for BBMCSP, the new maximum clique algorithm optimized for large and massive sparse graphs proposed in this paper. The algorithm has two stages: an initial preprocessing stage which efficiently reduces the scale of the problem in polynomial time, and the exponential time recursive search procedure. BBMCSP implements the latter using as reference SPARSE_SEARCH described in Section 3.2. Listing 5 and Listing 6 in this section are concerned with the initial stage.

Listing 5 describes the outline of the new algorithm. BBMCSP first analyses cores (step 1) and a reasonably large initial clique in S_{max} (step 2). Vertices with a core number lower than S_{max} cannot make part of a better solution (see previous section) so they are explicitly removed from the graph in step 3. The new reduced subgraph $G'\subseteq G$ is then sorted according to degeneracy in step 4. The main search loop (steps 5–9) branches on vertices in G' in reverse order (i.e. smallest k-core number first) and calls INITIAL_SEARCH to generate the child subproblem, attempt further pruning and ultimately call the recursive search procedure SPARSE_SEARCH.

INITIAL_SEARCH is described in Listing 6. It *unrolls the first level of the search tree* before calling the recursive search procedure, an important idea used both by FMC and PMC. This has the following advantages:

– The derived subproblem resulting from branching on vertex v, $N_{G'}(v)$, may be efficiently pruned using color and core bounds (with luck, the recursive search procedure might not even be called). Here the structure of the problem helps: massive sparse graphs have low density and neighborhoods are expected to be much smaller than the order of the graph.

Listing 5. The new maximum clique algorithm for massive sparse graphs.

```
Input: A simple graph $G(V, E)$
Output: A maximum clique in $S_{max}$
BBMCSP ($G$)
1.  Compute cores $K(V)$
2.  $S_{max}\leftarrow$ INITIAL_CLIQUE(G)          //an initial solution
3.  $G'(V',E')\leftarrow G-\{v\in V \mid K(v)<|S_{max}|\}$          //explicit graph reduction
4.  DEGENERACY($G'$)
5.  repeat until $|V'|=\phi$
6.      select vertex $w$ from $V'$ in reverse order
7.      INITIAL_SEARCH ($G'[V']$, $w$)
8.      remove $w$ from $V'$
9.  endrepeat
10. return $S_{max}$
```

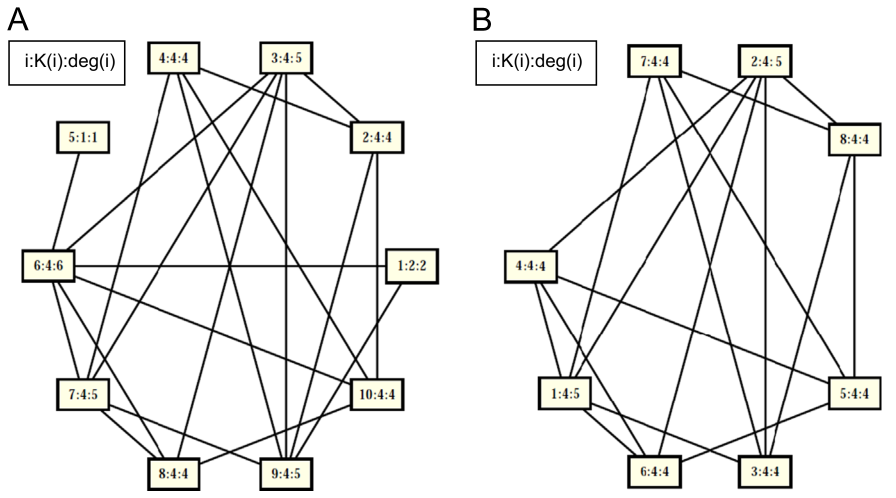

Fig. 1. A case study for the new algorithm: (A) a simple undirected graph G. and (B) G' after step 4 of BBMCSP.

Table 3
The main steps of the new algorithm related to INITIAL_SEARCH. Each row is a different call to the procedure. The input graph G' is depicted in Fig. 1B. After each call vertex v is removed implicitly from G'.

INITIAL_SEARCH (v, G') v	U_v	$c(v)$	$K(U_v+\{v\})$	$\lvert S_{max}\rvert$	**Comments**
8	2 3 5 7	1 2 1 3	2 2 2 2, $K(8)=2$	3	Vertex 8 has neighbors 2, 3, 5, 7. $\lvert C\rvert=3$ not pruned $\boldsymbol{K(G)=2<\lvert S_{max}\rvert}$ **pruned**
7	1 3 5	1 2 1		3	Vertex 7 has neighbors 1, 3, 5. $\boldsymbol{\lvert C\rvert=2<\lvert S_{max}\rvert}$ **pruned**
6	1 2 4 5	1 2 1 3	3 3 3 2, $K(6)=3$	4	Vertex 6 has neighbors 1, 2, 4, 5. $K(5)=2<S_{max}=3$ so **vertex 5 is removed** SPARSE_SEARCH finds clique {1, 2, 4}**New $\boldsymbol{S_{max}}$={1, 2, 4, 6}**
5	4			4	Vertex 5 has neighbor 4 $\boldsymbol{\lvert U_v\rvert=1<\lvert S_{max}\rvert}$ **pruned**
4 (or 3)	1 2			4	Vertex 4 (or 3) have neighbors 1, 2 $\boldsymbol{\lvert U_v\rvert=2<\lvert S_{max}\rvert}$ **pruned**
Remaining subproblems are trivially pruned. $\omega(G)=S_{max}=4$					

– The new subproblem may be again sorted (this time implicitly) according to degeneracy ordering. This further helps to reduce the size of the search tree which derives from it.
– As a side effect, unrolling allows an easy parallelization of the algorithm by assigning a task to each call to INITIAL_SEARCH (in PMC this is achieved almost trivially using the OpenMP[3] framework). Note we *do not consider parallel execution in this paper*, amongst other things to avoid noise in the evaluation of the proposed optimizations.

In INITIAL_SEARCH, pruning occurs in steps 2, 4, 7 and 8: step 2 is concerned with the trivial maximum graph degree bound $\Delta G+1$, step 4 with the color bound$\lvert C(G)\rvert$, step 7 with the graph core bound $K(G)+1$ and finally step 8 considers the core order of individual vertices. In the first three cases the entire subproblem is pruned, since it cannot improve the current initial solution in S_{max}. In the latter case individual vertices are removed from U_v; in our experiments *explicit* graph reduction was shown not worthwhile here. Note we use the neighbor set of v (U_v) as a subproblem to

Listing 6. INITIAL_SEARCH procedure.

```
INITIAL_SEARCH ($G$, $v$)
1.  $U_v \leftarrow N_G(v)$                                   //root child subproblem
2.  **if** $\lvert U_v\rvert < \lvert S_{max}\rvert$ **return**
3.  $col \leftarrow$ size of greedy sequential coloring SEQ($U_v$)
4.  **if** $col < \lvert S_{max}\rvert$ **return**
5.  $W \leftarrow U_v + \{v\}$
6.  Compute cores $K(W)$
7.  **if** $K(W) < \lvert S_{max}\rvert$ **return**
8.  $U'_v \leftarrow W - \{w \in W \mid K(w) < \lvert S_{max}\rvert\} - \{v\}$
9.  $L \leftarrow U'_v$ in degeneracy ordering
10. $C(U'_v) \leftarrow \{c(w \in U'_v \subseteq W) \mid c(w) \leftarrow K(w)\}$    //uses core numbers as colors
11. SPARSE_SEARCH ($U'_v$, $\{v\}$, $S_{max}$, $C(U'_v)$, $L$)    //main search procedure; updates $S_{max}$
```

Table 4
Comparison between the new algorithm BBMCSP and PMC over large sparse networks between 100,000 and 3,000,000 nodes. Times are measured in seconds with precision up to a millisecond. In bold, ratios above one order of magnitude in favor of the new algorithm.

Type	**Name**	**$\|V\|$**	**$\|E\|$**	**d_{max}**	**d_{avg}**	**$K(G)+1$**	ω	**BBMCSP ω_o**	BBMCSP heur	BBMCSP search	**PMC[23] ω_o**	**PMC heur**	**PMC search**	**PMC/ BBMCSP**
Affiliation	*aff-digg*	872622	22501700	75587	51.6	646	32	25	17.8	832	67	37.0	3640	4.37
Affiliation	*aff-flickr-user-groups*	395979	8537703	34990	43.1	187	14	10	4.98	11.7	11	6.41	42.6	3.65
Collaboration	ca-*IMDB*	896308	3782447	1590	8.4	24	3	3	0.708	0.921	3	0.933	2.77	3.01
Collaboration	*misc-IMDB-bi*	896308	3782447	1590	8.4	24	3	3	0.724	0.914	3	1.40	3.44	3.76
Interaction	*ia-wiki-Talk-dir_add_one*	2394385	4659565	100029	3.9	132	26	16	5.10	0.576	22	1.77	5.78	**10.0**
Recommendation	*rec-dating*	168792	9942260	144540	117.8	142	16	15	14.2	13.9	15	172	408	**29.3**
Recommendation	*rec-epinions*	755761	7953517	552206	21.0	76	10	8	19.3	6.38	9	269	531	**83.3**
Recommendation	*rec-libimseti-dir_noweights*	220970	5581942	153343	50.5	80	14	13	14.4	4.20	14	139	316	**75.3**
Scientific comp.	*delaunay_n20*	1048576	3145686	23	6.0	5	4	4	0.504	0.370	4	0.685	1.58	4.27
Scientific comp.	*delaunay_n21*	2097152	6291408	23	6.0	5	4	4	1.04	0.767	4	1.52	3.49	4.55
Scientific comp.	*packing-500x100x100-b050*	2145852	17488243	18	16.3	10	4	4	1.97	3.00	4	3.98	9.66	3.22
Scientific comp.	*sc-ldoor*	952203	20770807	76	43.6	35	21	21	1.29	1.05	21	5.08	10.8	**10.3**
Scientific comp.	*sc-msdoor*	415863	9378650	76	45.1	35	21	21	0.606	0.461	21	2.23	4.89	**10.6**
Social	*soc-buzznet*	101163	2763066	64289	54.6	154	31	23	1.56	1.40	24	2.85	16.4	**11.7**
Social	*soc-catster*	149700	5448197	80634	72.8	420	81	58	2.88	0.801	77	4.41	13001	**16228**
Social	*soc-digg*	770799	5907132	17643	15.3	237	50	42	3.28	1.31	46	1.52	11.1	8.46
Social	*soc-dogster*	426820	8543549	46503	40.0	249	44	33	5.26	2.37	36	7.48	22.2	9.36
Social	*soc-flickr*	513969	3190452	4369	12.4	310	58	40	1.05	3.30	43	0.976	21.5	6.49
Social	*soc-flickr-und*	1715255	15555041	27236	18.1	569	98	68	7.31	99.5	77	7.01	717	7.21
Social	*soc-flixster*	2523386	7918801	1474	6.3	69	31	29	1.51	0.495	30	0.767	2.01	4.06
Social	*soc-FourSquare*	639014	3214986	106218	10.1	64	30	27	4.36	0.299	29	25.5	49.7	**166**
Social	*soc-lastfm*	1191805	4519330	5150	7.6	71	14	14	1.03	0.562	14	0.834	2.77	4.93
Social	*soc-LiveMocha*	104103	2193083	2980	42.1	93	15	10	0.683	1.07	13	0.586	3.62	3.38
Social	*soc-orkut*	2997166	106349209	27466	71.0	231	47	46	35.2	51.9	43	40.8	199	3.83
Social	*soc-pokec*	1632803	22301964	14854	27.3	48	29	29	3.74	3.74	29	4.81	10.2	2.72
Social	*soc-wiki-Talk-dir*	2394385	4659565	100029	3.9	132	26	17	3.41	0.558	21	1.21	4.87	8.73
Social (facebook)	*socfb-B-anon*	2937612	20959854	4356	14.3	64	24	24	6.66	7.98	24	5.94	19.5	2.44
Technological	*tech-ip*	2250498	21643497	1833161	19.2	254	4	4	57.1	14.3	3	32.0	843	**59.0**
Web links	*web-baidu-baike*	2141300	17014946	97848	15.9	79	31	31	15.7	4.47	30	19.8	47.9	**10.7**
Web links	*web-wiki-ch-internal*	1930275	8956902	29005	9.3	121	33	33	3.94	1.15	32	6.84	15.3	**13.3**

compute degree and color bounds in steps 2 and 4, but the full subproblem $W = U_v + \{v\}$ for core analysis. This slightly simplifies sequential greedy coloring in step 3 because the size of $C(W)$ is always the size of $C(U_v)$ incremented by one (i.e. the extra color used by v which is adjacent to every vertex in U_v).

Once simplifications are completed, steps 9–10 initialize the data structures required by the tailored recursive algorithm SPARSE_SEARCH. Specifically, branching list L is ordered following degeneracy and *vertex core numbers are used as initial color number assignments*. Note this is not a proper coloring in the general case, but search remains complete because vertex core numbers (incremented by one) are always a bound for the size of any clique containing that vertex. As a result, all subproblems with a vertex w at the root node such that the incumbent solution is greater than its core number (i.e. $K(w) < |S_{max}|$) will be pruned during search. To see this it suffices to examine the pruning condition of the reference search algorithm $|S| + c(v) \le |S_{max}|$ (step 3 of Listing 1). In the initial call to SPARSE_SEARCH, $|S| = 1$ and $c(v) = K(v)$ so that $1 + K(v) \le |S_{max}| \Rightarrow K(v) < |S_{max}|$.□

In our implementation, greedy sequential coloring SEQ in step 3 is a simplified version of reference COLOR (sparsified) described in Listing 2 which returns just the color size of the coloring. Specifically, steps 7–10 have been removed from that listing and color index k is now the output.

The preprocessing stage of BBMCSP is inspired by the PMC algorithm. The main differences are the following:

– *Vertices are never explicitly removed from the graph once* INITIAL_SEARCH *is called*. During search, PMC uses an auxiliary list to store vertices v_i which cannot be part of the solution because $K(v_i) < |S_{max}|$ and subproblems containing one of the vertices in the list are automatically pruned; moreover, after a predefined time interval they are explicitly removed from the graph. BBMCSP takes a different view: *it uses as initial coloring the core number of vertices* and filtering related to cores is restricted to branching at the root node, which is where it is most effective. In the remaining subproblems normal color bounds are used. BBMCSP *does not* employ an auxiliary list of pruned vertices.
– In INITIAL_SEARCH, BBMCSP *computes color pruning before core pruning*. The reason is that $|C(G)|$ is tighter than $K(G)$ and coloring is not more expensive than core analysis on average because the size of U_v is expected to be small in real massive graphs.
– Coloring in INITIAL_SEARCH is restricted to the neighbor set $N_G(v)$ and not the full subproblem $N_G(v) + \{v\}$.
– The new algorithm *makes efficient use of bit-parallelism* in the recursive search procedure SPARSE_SEARCH (as described in Sections 3.1 and 3.2).

INITIAL_CLIQUE in step 2 of Listing 5 computes an initial good solution in the same way as PMC. We have chosen this heuristic, and not a state-of-the-art approximate algorithm, for the following reasons:

– The compromise between size and computation effort is *good* for massive sparse graphs: INITIAL_CLIQUE is able to find the correct solution in 96 out of the set of 276 real graphs

Table 5
Comparison between the new algorithm BBMCSP and PMC over massive networks with more than 3,000,000 nodes. Times are measured in seconds with precision up to a millisecond. In bold, ratios above one order of magnitude in favor of the new algorithm.

Type	Name	$\|V\|$	$\|E\|$	d_{max}	d_{avg}	$K(G)+1$	ω	BBMCSP ω_o	BBMCSP heur	BBMCSP search	PMC[23] ω_o	PMC heur	PMC search	PMC/ BBMCSP
Affiliation	*aff-orkut-user2groups*	8730857	327037420	318268	74.92	472	6	5	852	1377	6	350	29042	**21.1**
Infrastructure	*inf-germany_osm*	11548845	12369181	13	2.14	4	3	3	<0.001	<0.001	3	1.02	2.88	**2879**
Infrastructure	*inf-great-britain_osm*	7733822	8156517	8	2.11	4	3	3	<0.001	<0.001	3	0.680	1.94	**1940**
Infrastructure	*inf-italy_osm*	6686493	7013978	9	2.10	4	3	3	<0.001	<0.001	3	0.471	1.54	**1540**
Scientific comp.	*adaptive*	6815744	13624320	4	4.00	3	2	2	1.35	1.15	2	2.77	6.94	6.03
Scientific comp.	*channel-500x100x100-b050*	4802000	42681372	18	17.78	10	4	4	4.47	5.22	4	9.37	21.2	4.06
Scientific comp.	*delaunay_n22*	4194304	12582869	23	6.00	5	4	4	2.14	1.58	4	3.22	7.40	4.70
Scientific comp.	*delaunay_n23*	8388608	25165784	28	6.00	5	4	4	4.46	3.29	4	6.53	14.9	4.52
Scientific comp.	*delaunay_n24*	16777216	50331601	26	6.00	5	4	4	8.96	6.84	4	13.2	29.4	4.29
Scientific comp.	*hugebubbles-00020*	21198119	31790179	3	3.00	3	2	2	6.42	3.62	2	11.8	34.0	9.40
Scientific comp.	*hugetrace-00000*	4588484	6879133	3	3.00	3	2	2	1.14	0.689	2	2.10	5.20	7.54
Scientific comp.	*hugetrace-00010*	12057441	18082179	3	3.00	3	2	2	3.06	1.83	2	5.65	14.8	8.09
Scientific comp.	*hugetric-00000*	5824554	8733523	3	3.00	3	2	2	1.34	0.880	2	2.52	6.32	7.18
Scientific comp.	*hugetric-00010*	6592765	9885854	3	3.00	3	2	2	1.80	1.07	2	3.25	8.10	7.56
Scientific comp.	*hugetric-00020*	7122792	10680777	3	3.00	3	2	2	2.01	1.19	2	3.64	9.03	7.60
Scientific comp.	*venturiLevel3*	4026819	8054237	6	4.00	4	3	3	0.788	0.574	3	1.53	3.58	6.24
Social	*soc-friendster*	65608366	1806067135	5214	55.06	305	129	119	692	1027	129	861	7681	7.48
Social	*soc-livejournal07*	5204176	48709773	15017	18.72	375	358	356	0.052	0.007	358	5.39	6.34	**862**
Social	*soc-livejournal-user-groups*	7489073	112307315	1053749	29.99	117	9	8	771	718	9	1333	36246	**50.5**
Social	*soc-ljournal-2008*	5363260	49514271	19432	18.46	426	400	400	0.033	0.001	400	6.10	7.22	**7051**
Social	*soc-orkut-dir*	3072441	117185083	33313	76.28	254	51	50	38.7	55.8	47	50.7	229	4.10
Social	*soc-sinaweibo*	58655849	261321033	278489	8.91	194	44	41	662	94.8	37	1090	2605	**27.5**
Social (facebook)	*socfb-A-anon*	3097165	23667394	4915	15.28	75	25	24	7.18	9.01	23	6.79	21.8	2.41
Social (facebook)	*socfb-konect. edges*	46009640	72040814	4960	3.13	16	6	6	4.77	0.638	6	11.9	21.9	**34.4**
Social (facebook)	*socfb-uci-un*	58790782	92208195	4960	3.14	17	6	6	8.95	0.916	6	19.5	33.1	**36.1**
Web links	*web-ClueWeb09-50 m*	428136612	446534058	308477	2.09	193	56	53	267	4.77	56	238	636	**133**
Web links	*web-indochina-2004-all*	7414866	150984819	256425	40.72	6870	6848	6848	14.7	0.282	6848	1110	1137	**4039**
Web links	*web-it-2004-all*	41291318	1027474947	1326744	49.77	3225	3222	3222	3.59	0.195	3222	141	162	**833**
Web links	*web-wikipedia_link_de*	3930109	68714064	437732	34.97	461	339	339	14.4	0.007	339	7.86	10.3	**1587**
Web links	*web-wikipedia_link_fr*	5115915	104591689	1274642	40.89	818	332	332	253	0.115	332	15.0	23.8	**206**

considered for this report, and in most cases is close to the optimal solution.
– The comparison with PMC is cleaner and simpler.
– It integrates well in the algorithm since core analysis has to be done anyway.

Essentially the heuristic greedily constructs cliques starting from an initial vertex seed and enlarging it by picking the first possible candidate vertex according to degeneracy ordering. Assume the set of vertices V follows degeneracy. INITIAL_CLIQUE starts with seed v_1 and constructs a first maximal clique Q_1. It then proceeds with seed v_2 to produce maximal clique Q_2 and so on. The final output $Q_{max}$is the largest of all Q_i. Moreover, core number information is used at each choice to prune vertices that cannot possibly make part of a clique larger than the incumbent Q_{max}, i.e. $K(v) < Q_{max}$. A full description may be found in the original PMC paper.

3.5. An example

We now show the main steps of the new algorithm with the help of graph $G=(V,E)$ depicted in Fig. 1A. In the figure, each vertex label contains index, core number and degree separated by colons.

INITIAL_SEARCH receives as input V' and the first vertex of each new subproblem. Table 3 shows the steps taken by the procedure: each row refers to a new call to INITIAL_SEARCH with the vertex in column v. Note that vertices are picked in reverse order from V'so the calls are made from top to bottom. After each call vertex v is implicitly removed from V' and so is not considered in the neighborhood U_v of table rows below it.

G has an order of 10, a core number of 4 and a maximum degree of 6; this information is obtained during k-core analysis in the first step of the algorithm. The second step finds a suboptimal

Table 6
Comparison between the new algorithm BBMCSP and FMC and BBMC algorithms over networks with less than 100,000 nodes. Times are measured in seconds with precision up to a millisecond. Time limit is fixed at 3 h in all cases. In bold, ratios above one order of magnitude in favor of the new algorithm.

Type	Name	$\lvert V\rvert$	$\lvert E\rvert$	d_{max}	d_{avg}	$K(G)+1$	ω	ω_o	BBMCSP search	FMC[24] search	FMC/ BBMCSP	BBMC[13] search	BBMC/ BBMCSP
Interaction	*ia-enron-large*	33696	180811	1383	10.73	44	20	16	0.010	0.539	**56.6**	0.222	**23.3**
Interaction	*ia-wiki-Talk*	92117	360767	1220	7.83	59	15	11	0.041	3.26	**79.9**	1.55	**38.1**
Scientific comp.	*fe_rotor*	99617	662431	125	13.30	9	5	5	0.086	0.093	1.08	2.81	**32.7**
Scientific comp.	*fe-tooth*	78136	452591	39	11.58	8	5	5	0.049	0.069	1.42	1.57	**32.2**
Scientific comp.	*sc-nasasrb*	54870	1311227	275	47.79	36	24	24	0.051	2.90	**57.3**	1.42	**28.0**
Scientific comp.	*sc-pkustk11*	87804	2565054	131	58.43	48	36	36	0.029	188	**6519**	2.12	**73.6**
Scientific comp.	*sc-pkustk13*	94893	3260967	299	68.73	42	36	36	0.155	2460	**15888**	4.70	**30.4**
Social	*soc-BlogCatalog*	88784	2093195	9444	47.15	222	45	37	2.68	> 3 h	∞	15.8	5.89
Social	*soc-brightkite*	56739	212945	1134	7.51	53	37	36	< 0.001	2.03	**2030**	0.519	**1292**
Social	*soc-epinions*	26588	100120	443	7.53	33	16	16	0.002	0.075	**30.5**	0.121	**49.0**
Social	*soc-slashdot*	70068	358647	2507	10.24	54	26	25	0.012	6.55	**562**	0.921	**78.9**
Social (facebook)	*socfb-Berkeley13*	22900	852419	3434	74.45	65	42	42	0.163	55.2	**338**	0.266	1.63
Social (facebook)	*socfb-CMU*	6621	249959	840	75.50	70	45	44	0.021	10.3	**492**	0.033	1.56
Social (facebook)	*socfb-Duke14*	9885	506437	1887	102.47	86	34	31	0.097	201	**2065**	0.088	0.907
Social (facebook)	*socfb-Indiana*	29732	1305757	1358	87.84	77	48	44	0.323	161	**498**	0.550	1.70
Social (facebook)	*socfb-MIT*	6402	251230	708	78.48	73	33	32	0.031	89.1	**2877**	0.037	1.18
Social (facebook)	*socfb-OR*	63392	816886	1098	25.77	53	30	28	0.065	3.32	**51.0**	1.00	**15.4**
Social (facebook)	*socfb-Penn94*	41536	1362220	4410	65.59	63	44	43	0.233	22.6	**97.0**	0.721	3.09
Social (facebook)	*socfb-Stanford3*	11586	568309	1172	98.10	92	51	51	0.090	508	**5621**	0.096	1.07
Social (facebook)	*socfb-Texas84*	36364	1590651	6312	87.48	82	51	50	0.342	3469	**10148**	0.745	2.18
Social (facebook)	*socfb-UCLA*	20453	747604	1180	73.10	66	51	50	0.104	14.2	**137**	0.194	1.87
Social (facebook)	*socfb-UConn*	17206	604867	1709	70.31	66	50	49	0.063	113	**1796**	0.109	1.72
Social (facebook)	*socfb-UCSB37*	14917	482215	810	64.65	66	53	51	0.010	72.4	**7151**	0.060	5.97
Social (facebook)	*socfb-UF*	35111	1465654	8246	83.49	84	55	51	0.254	> 3 h	∞	0.639	2.52
Social (facebook)	*socfb-UIllinois*	30795	1264421	4632	82.12	86	57	55	0.172	> 3 h	∞	0.428	2.49
Social (facebook)	*socfb-Wisconsin87*	23831	835946	3484	70.16	61	37	35	0.175	60.6	**347**	0.302	1.73
Technological	*tech-WHOIS*	7476	5s6943	1079	15.23	89	58	56	0.001	7381	**9146283**	0.012	**14.7**
Temporal	*scc_retweet*	18469	65990	551	7.15	175	166	164	0.001	> 3 h	∞	0.023	**23.3**
Temporal	*scc_twitter-copen*	8580	473614	1516	110.40	583	581	581	< 0.001	> 3 h	∞	0.004	4.00
Web links	*web-spam*	4767	37375	477	15.68	36	20	20	0.001	0.284	**519**	0.005	8.57

initial clique S_{max} of size 3 so vertices 1 and 5 will be explicitly removed from the graph in step 3 because their core number is less than $S_{max} - K(1) = 2$, $K(5) = 1$. In the current implementation of BBMCSP, we first order G by degeneracy and only then explicitly remove vertices 1 and 5, i.e. steps 3 and 4 are actually reversed.

The degeneracy ordering algorithm (outlined in listing 4) starts with assigning vertex degree to number set D: 5:1 ∣ 1:2 ∣ 2:4 4:4 8:4 10:4 ∣ 3:5 7:5 9:5 ∣ 6:6. Vertex 5, with minimum number, is selected first and the new updated D becomes 1:2∣2:4 4:4 8:4 10:4∣3:5 7:5 9:5 6:5∣, since vertex $6 \in N(5)$. Vertex 1 is the next choice and the resulting number list 2:4 4:4 8:4 10:4 6:4 9:4 ∣ 3:5 7:5, since vertices $\{6, 9\} \subset N(1)$. At this point, vertices 2, 4, 8, 10, 6 and 9 are selected in order because they belong to the same 4-core and their numbers remain fixed. Finally, the number assigned to vertex 3 is updated to $K(3) = 4$ when vertex 2 is selected and, similarly, vertex 7 is updated to $K(7) = 4$ on selection of vertex 4. The core number of the sorted graph is therefore $K(G') = 4$. The resulting order of the new graph G' after step 4 (expressed in the form *old_index→new_index*) is: 2→8, 3→2, 4→7, 6→4, 7→1, 8→6, 9→3, 10→5. Note that vertices are sorted in the reverse order they are selected and that vertex 1 and 5 have been removed. Fig. 1B depicts G'.

The first call to INITIAL_SEARCH takes vertex 8 and analyses subproblem {2, 3, 5, 7, 8}. The vertex coloring of the neighborhood of vertex 8 {2, 3, 5, 7} has size 3 so the subproblem cannot be pruned (note that $\lvert C(\{2, 3, 5, 7, 8\})\rvert = 4$, since 8 is adjacent to all other vertices, and so it can still improve S_{max}). The subproblem *is* pruned, however, in step 6 by core analysis, which gives a graph core number $K(G') = 2$ below the threshold *Smax*. In a similar fashion the subproblem starting with vertex 7 is eliminated, this time because of the color bound. The critical step corresponds to the subproblem starting with vertex 6, which actually contains the maximum clique. Now the entire subproblem obviously cannot be pruned, but core analysis removes vertex 5 in step 7 ($K(5)=2$, so no clique containing 5 can improve the incumbent clique in S_{max}). The subproblem finally passed to the recursive search procedure {1, 2, 4, 6} – {3, 6, 7, 8} in G – is the maximum clique solution to the problem. The remaining subproblems are pruned trivially given the new clique in S_{max}.

4. Experiments

This section reports tests carried out to evaluate the performance of the new algorithm. Section 4.1 describes test configuration and the different algorithms chosen for the comparison. Section 4.2 is concerned with the data sets. The remaining two subsections show performance results.

4.1. Reported algorithms

To evaluate the new algorithm the following other algorithms have been considered:

a) BBMC: The state-of-the-art bit-parallel algorithm for small and middle-size graphs described in Section 2.3. It encodes the adjacency matrix in full as do most of the relevant exact maximum clique algorithms.
b) Fast Max-Cliquer (FMC): A reference solver for massive sparse graphs. It uses an edge list to compress the adjacency matrix. Specific optimization techniques include a preprocessing stage which unrolls the first level of the search tree and prunes subproblems based on their size (measures as degree of neighborhoods) both during preprocessing and search.
c) Parallel Maximum Clique (PMC): In our view *the* leading reference solver for massive sparse graphs. It employs similar techniques as FMC but uses the stronger notion of core (instead of degree) for pruning. Moreover it also employs approximate color bounds during search (as BBMC), whereas FMC does not.

For the tests we have used the source code publicly available for PMC[4] and FMC[5] – version 1.1. Our experiments were performed using a Linux workstation running a 64-bit Ubuntu release at 3.00 GHz with an Intel(R) Xeon(R) CPU E5-2690 v2 multi-core processor and 128 GB of main memory. All algorithms considered are implemented in C++ and compiled using *gcc 4.8.1* (parameter *–o2*). In none of the algorithms any form of multi-core parallelism has been allowed, i.e. just one core is used at run time out of a possible 20 available in the machine. In the case of PMC this is achieved with parameter *–t 1*.

As mentioned in previous sections, when considering massive graphs a good initial solution can have a very strong impact on performance since it may prune an exponential number of subproblems during the preprocessing stage. BBMCSP uses the same initial heuristic proposed in PMC but there are marginal differences due to the tie-breaking strategy employed for vertices with the same core number. FMC is not distributed with an initial heuristic in its exact version and currently BBMC uses a simple greedy heuristic not k-core based. Therefore to establish a fair comparison *we have fed both FMC and BBMC with the same initial solution given by BBMCSP*.

In all networks there was a fixed time limit of 3 h and only user time for searching the graph has been considered. All the data obtained from the tests (including raw output from the different algorithms) is available in our server [33]. The site contains a Linux binary of BBMCSP as well. Source code for BBMCSP has recently been released in [34].

4.2. Data sets

For this report we have considered a total data set of 276 real graphs taken from the *Network Data Repository* [35], a recent network repository which has gathered a big collection from different sources. All networks have been preprocessed when required to discard weights and self-loops and all directed edges are considered bidirected. In the repository most of the names of the graphs have been added a prefix indicating a category, as in *fe-* for finite elements, *soc-* for social and *aff-* for affinity. We use the same notation in the paper. Finally we note that our desire would be cite the original sources of the 276 different networks considered; but this is impractical for obvious reasons.

Of the 276 real graphs, 96 were solved trivially – there was no call to SEARCH-SP – by the new algorithm during the preprocessing stage (which validates the strategy of unrolling (and pruning) the first level of the search tree). Some of these graphs have millions of vertices – as in Infrastructure networks *inf-europe_osm* (more than 50 million) or *inf-asia_osm* (more than 11 million) – and their k-core number tends to be low although this is not always so. For example Collaboration network ca-*hollywood-2009* has a high k-core number of 2209, but still the heuristic finds an initial solution just as large so no search is required. It is also worth noting that in all cases leading to triviality the initial heuristic INITIAL_CLIQUE does not take more than a few milliseconds to compute.

From the remaining 180 non-trivially solved networks, only a selection of 90 has been included in the report for space reasons. These are: 30 graphs below 100,000 nodes, 30 graphs between 100,000 and 3,000,000 nodes and 30 massive networks above 3,000,000. The concrete choice in each category is based on the longest search time taken by BBMCSP. As mentioned previously, results for the full 276 networks are available in our server.

The 90 reported networks fall into the following categories (in parenthesis the number of graphs selected):

– **Affiliation networks** (3): Nodes are typically individuals an links represent membership to a collective or group. These networks originate from other networks and may be considered as metadata from these (e.g. *aff-digg*, *aff-flickr*).
– **Collaboration networks** (2): These are networks in which nodes represent individuals and edges represent scientific or movie production collaborations. In the 90 networks reported only movie collaborations appear (e.g. ca-*IMDB*).
– **Interaction networks** (3): Nodes are also individuals, but edges now refer to communication interaction via message posts (e.g. *ia-enron*, *ia-wiki-Talk*)
– **Infrastructure networks** (3): The nodes here are typically points of interest and edges represent land communication between them —roads, streets. Once source of information is OpenStreetMap, as in *inf-italy_osm* or *inf-germany_osm*.
– **Recommendation networks** (3): Here nodes are people and an edge between them refers to a communication in terms of recommendation or opinion. An example is in *rec-epinions* in which weighted edges represent degree of trust.
– **Scientific computing networks** (22): These are meshes that appear from mathematical analysis in different fields of science: finite elements (*fe-*), sparse matrixes (*sc-ldoor*), numerical simulation (*packing-*), dynamic grids (*huge-*), Delaunay triangulations (*delaunay_n*) etc.

[4] https://www.cs.purdue.edu/homes/dgleich/codes/maxcliques.
[5] http://cucis.ece.northwestern.edu/projects/MAXCLIQUE.

– **Social networks** (42): Here nodes are individuals and links refer to relationships of different types such as friendship (e.g. Facebook) or follower (flickr, orkut, friendster, FourSquare, etc.).
– **Technological networks** (2): In this case nodes are routers and edges refer to communications between them (*tech-ip*, *tech-WHOIS*).
– **Temporal networks** (2): In temporal graphs nodes are entities and weighted edges refer to relations between entities in time. In the reported networks nodes are Twitter users and edges refer to a tweet or retweet (e.g. *scc-retweet)*.
– **Web link networks** (8): In web graphs, nodes are web-pages and edges refer to hyperlinks between them. Since we only consider bidirected links, a clique of web-pages represents full traversal in the set (e.g. *web-indochina*, *web-baidu-baike*).

4.3. *Comparison with Parallel Maximum Clique algorithm*

In this section we report comparison results between BBMCSP and state-of-the-art PMC. Table 4 shows performances for both algorithms over 30 real networks with orders ranging from 100,000 up to 3,000,000 nodes. Table 5 covers 30 massive graphs with more than 3,000,000 vertices. Both tables include maximum and average graph degree (headers d_{max} and d_{avg}), the core bound, i.e. the core number of the graph incremented by one, the initial solution found by the heuristic (ω_0) and performance times measured in seconds. The time taken to find the initial solution has been included for completeness.

Before analyzing the results it is worth noting that although the heuristic used to compute the initial solution is essentially the same in both algorithms, it differs on how vertices with the same core number are selected to enlarge the growing initial solution. These differences concern the sparse encoding and, as a result, the heuristic in BBMCSP tends to be faster in many networks (up to 14 times faster in the best case for *rec-epinions*) but produces slightly worse solutions. Specifically, out of the 60 instances considered in both tables BBMCSP averages an initial clique order of 208.8, whereas PMC averages a superior 210.5. We note that in spite of this disadvantage, BBMCSP clearly outperforms its counterpart in search time. We also note that in a few cases the initial solution found by BBMCSP is better. In particular, out of the 96 networks solved trivially by BBMCSP, 7 required recursive search by PMC because the initial solution had one vertex less than the optimal solution.

The last column in the tables shows the performance ratio in both algorithms related to search (time spent on initialization of data structures, reading the graph or finding an initial solution is not counted). In the case of large – but not massive – graphs of Table 4, the worst comparative performance for BBMCSP occurs in Facebook network *socfb-B-anon* when it is more than twice as fast as PMC. The best performance for BBMCSP occurs in Social *soc-catster* in which it outperforms PMC by more than 4 orders of magnitude. Average ratio speedup for all networks in Table 4 is 560 times in favor of the new algorithm.

In the case of massive graphs in Table 5, the average performance ratio is even better: BBMCSP turns out to be 728 times faster than PMC. At the lower end – not considering the three Infrastructure networks solved by both algorithms in less than a millisecond – is Facebook *socfb-A-anon* network with a 2.4 ratio of improvement and at the higher end lies Social *soc-ljournal-2008* network in which BBMCSP runs 7051 times faster.

Possible explanations for the excellent computational results of the new algorithm lie in the new sparse encoding, in the way k-core bounds are passed to the recursive search function as initial color numbers, and in the fact that, in contrast to PMC, it does not require to upkeep a list of pruned vertices.

4.4. *Comparison with other algorithms*

We have also compared the new algorithm with sparse FMC and non-sparse BBMC algorithms. Specifically, Table 6 reports performance results over 30 networks with less than 100,000 nodes. The information provided for each graph is analogous to that shown for PMC in the previous section.

BBMC is representative of the family of efficient maximum clique algorithms for small and middle-size graphs. This is a reasonable assumption because, as mentioned in the introductory section, the more recent algorithms, such as improved variants of BBMC [15] or IncMaxCLQ [16], are not expected to work well over the networks of interest to this paper. FMC has been designed for massive sparse graphs but, as will be shown, performs very poorly in the reported set, so it is enough to establish PMC as the better algorithm and the reason why we have not considered necessary to add more results of FMC in this report.

BBMC performs comparably to BBMCSP in Facebook networks *socfb-Duke14*, *socfb-Stanford3* and *socfb-MIT*, even outperforming BBMCSP by a 10% in the first case. The reason is that these graphs have less than 12,000 nodes and high average degree so farthest from the type of graphs concerned in the paper. In the remaining cases BBMCSP outperforms its counterpart, the more so in graphs with low average degree and high order. Average ratio of improvement in favor of BBMCSP is 59.3 in the reported 30 instances, and the peak performance ratio is a 3 orders of magnitude improvement in Social network *soc-brightkite*.

FMC performs very poorly against this data set. A possible explanation is that it uses the weaker notion of degree instead of core to prune the subproblems. Also it builds on the Carraghan and Pardalos algorithm [8] from the 90 s in the search stage, which does not use approximate color bounds. While this approach might be reasonable for non structured instances like *Erdös-Rényi* graphs or for very sparse graphs, it *does not capture the structure of many real graphs*. The result is that in only two of the networks reported (specifically *fe-rotor* and *fe-tooth*, both derived from numerical analysis) does FMC perform comparably to the new algorithm. In the remaining graphs, FMC fails after 3hours of computation on 5 networks, is outperformed by 3 orders of magnitude on 7 networks, by 4 orders of magnitude on 2 more networks and by almost 8 orders of magnitude in the Technological network *tech-WHOIS*.

It is only fair to say that in the full tests available online [33], FMC performs comparably to BBMCSP in 10 networks – in 6 cases it actually outperforms the latter – most of them originated from Delaunay triangulations. A possible explanation is that all these graphs have a maximum degree "reasonably" close to their core number. For example the graph *delaunay_n22*, in which FMC achieves best performance, has maximum graph degree 23 and core number 4.

5. Conclusions

This paper presents BBMCSP, a new very efficient exact maximum clique algorithm for large and massive sparse graphs, which are a common occurrence in real life.

BBMCSP is based in a prior bit-parallel algorithm BBMC and achieves its efficiency by using specific optimization techniques for the domain. These include a new sparse adjacency matrix and vertex set encoding, core-based pruning, a preprocessing stage which unrolls the first level of the search tree and a new way to pass *k*-core bound information obtained during preprocessing to the recursive search procedure. As a result, maximum clique problems of millions of vertices may be solved in a few seconds. Moreover, reported results show that BBMCSP outperforms by several orders of magnitude leading state-of-the-art algorithms over real large and massive networks.

Acknowledgments

This work is funded by the Spanish Ministry of Economy and Competitiveness: Grants ARABOT-DPI 2010-21247-C02-01 and NAVEGASE-DPI 2014-53525-C3-1-R. Panos M. Pardalos is supported by RSF grant 14-41-00039. We would also like to thank researcher Alvaro Pereda for his collaboration in the initial stages of this research. Special mention deserves researcher Jorge Artieda for his help with the administration of servers and portability of the source code.

References

[1] Karp RM. In: Miller RE, Thatcher JW, editors. Reducibility among combinatorial problems. New York; 1972, p. 85–103.
[2] Bahadur DKC, 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 2006;13:143–52.
[3] Clustering challenges in biological networks. In: Butenko S, Chaovalitwongse W, Pardalos P, editors. Singapore: World Scientific; 2009.
[4] San Segundo P, Artieda J. A novel clique formulation for the visual feature matching problem. Appl Intell 2015;43(2):325–42. http://dx.doi.org/10.1007/s10489-015-0646-1.
[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 Intell 2010;32(3):311–29.
[6] Balas E, Yu CS. Finding a maximum clique in an arbitrary graph. SIAM J Comput 1986;15(4):1054–68.
[7] Bron C, Kerbosch J. Algorithm 457: finding all cliques of an undirected graph. Commun. ACM 1973;16(9):575–7.
[8] Carraghan R, Pardalos PM. An exact algorithm for the maximum clique problem. Oper Res Lett 1990;9:375–82.
[9] ÖOstergård PRJ. A fast algorithm for the maximum clique problem. Discrete Appl Math 2002;120(1):97–207.
[10] Tomita E, Seki, T. An efficient branch and bound algorithm for finding a maximum clique. In: Proceedings of the discrete mathematics and theoretical computer science. LNCS, vol. 2731, 2003, p. 278–89.
[11] Konc J, Janežič D. An improved branch and bound algorithm for the maximum clique problem. MATCH Commun Math Comput Chem 2007;58:569–90.
[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, editors. Lecture notes in computer science, vol. 5942. 2010. p. 191–203.
[13] San Segundo P, Rodriguez-Losada D, Jimenez A. An exact bit-parallel algorithm for the maximum clique problem. Comput Oper Res 2011;38(2):571–81.
[14] San Segundo P, Matia F, Rodriguez-Losada D, Hernando M. An improved bit parallel exact maximum clique algorithm. Optim Lett 2013;7(3):467–79.
[15] San Segundo P, Tapia C. Relaxed approximate coloring in exact maximum clique search. Comput Oper Res 2014;44:185–92.
[16] Chu-Min L, Zhiwen F, Ke X. Combining MaxSAT reasoning and incremental upper bound for the maximum clique problem. In: Proceedings of the tools with artificial intelligence (ICTAI). 2013. p. 939–46.
[17] Prosser P. Exact algorithms for maximum clique: a computational study. Algorithms 2012;5(4):545–87.
[18] San Segundo P, Nikolaev A, Batsyn M. Infra-chromatic bound for exact maximum clique search. Comput Oper Res 2015;64:293–303 [in press].
[19] Andrade DV, Resende MGC, Werneck RF. Fast local search for the maximum independent set problem. J Heuristics 2012;18(4):525–47.
[20] Pullan W, Hoos HH. Dynamic local search for the maximum clique problem. J Artif Int Res 2006;25(1):159–85.
[21] Wu Q, Hao JK. An adaptive multistart tabu search approach to solve the maximum clique problem. J Comb Optim 2013;26(1):86–108.
[22] McCreesh C, Prosser P. Multi-threading a state-of-the-art maximum clique algorithm. Algorithms 2013;6(4):618–35.
[23] Rossi R, Gleich D, AssefawG, Mostofa,Md. Fast maximum clique algorithms for large graphs. In: Proceedings of the World wide web companion conference (WW Companion'14), 2014, p. 365–6.
[24] Pattabiraman B, Ali Patwary M, Gebremedhin A, Liao W, Choudhary A. Fast algorithms for the maximum clique problem on massive sparse graphs. LNCS 2013;8305:156–69.
[25] BITSCAN C++ library. ⟨https://www.biicode.com/pablodev/bitscan⟩.
[26] Welsh D, Powell M. An upper bound for the chromatic number of a graph and its application to timetabling problem. Comput J 1976;10(1):85–6.
[27] Seidman SB. Network structure and minimum degree. Soc Netw 1983;5:269–87.
[28] Erdös P, Hajnal A. On chromatic number of graphs and set-systems. Acta Math Acad Scientiarum Hung 1966;17:61–99.
[29] San Segundo P, Lopez A, Batsyn M. Initial sorting of vertices in the maximum clique problem reviewed. Learn Intell Optim LNCS 2014:111–20.
[30] Johnson D, Trick M, editors. Cliques, coloring, and satisfiability: second DIMACS implementation challenge. American Mathematical Society; 1996.
[31] Batagelj V, Zaversnik M. An $O(m)$ algorithm for cores decomposition of networks. Comput Res Repos 2003;cs.DS/0310.
[32] GRAPH C++ library. URL: ⟨https://www.biicode.com/pablodev/graph⟩.
[33] Server with results of full tests. URL: ⟨http://venus.elai.upm.es/logs/results_sparse/⟩.
[34] BBMCSP source code. URL: ⟨https://www.biicode.com/pablodev/examples_clique⟩.
[35] The *Network Data Repository*. URL:⟨http://networkrepository.com/⟩.