J Comb Optim (2014) 27:397–416
DOI 10.1007/s10878-012-9592-6

Improvements to MCS algorithm for the maximum clique problem

**Mikhail Batsyn · Boris Goldengorin ·
Evgeny Maslov · Panos M. Pardalos**

Published online: 9 January 2013
© Springer Science+Business Media New York 2013

Abstract In this paper we present improvements to one of the most recent and fastest branch-and-bound algorithm for the maximum clique problem—MCS algorithm by Tomita et al. (Proceedings of the 4th international conference on Algorithms and Computation, WALCOM'10, pp. 191–203, 2010). The suggested improvements include: incorporating of an efficient heuristic returning a high-quality initial solution, fast detection of clique vertices in a set of candidates, better initial colouring, and avoiding dynamic memory allocation. Our computational study shows some impressive results, mainly we have solved p_hat1000-3 benchmark instance which is intractable for MCS algorithm and got speedups of 7, 3000, and 13000 times for gen400_p0.9_55, gen400_p0.9_65, and gen400_p0.9_75 instances correspondingly.

Keywords Maximum clique problem · Branch-and-bound algorithm · Heuristic solution · Graph colouring

M. Batsyn (✉) · B. Goldengorin · E. Maslov · P. M. Pardalos
Laboratory of Algorithms and Technologies for Network Analysis, National Research University Higher School of Economics, 136 Rodionova, Nizhniy Novgorod, Russian Federation
e-mail: batsyn@yandex.ru

B. Goldengorin
e-mail: bgoldengorin@hse.ru

E. Maslov
e-mail: lyriccoder@gmail.com

P. M. Pardalos
Center of Applied Optimization, University of Florida, 401 Weil Hall,
P.O. Box 116595, Gainesville, FL 32611-6595, USA
e-mail: pardalos@ufl.edu

1 Introduction

The maximum clique problem refers to the problem of finding a clique (a complete subgraph) with the largest number of vertices in a given simple graph. It has a lot of applications, because many practical problems can be formulated in terms of the maximum clique problem (Bomze et al. 1999). Biochemistry and genomic problems represented by a clique-detection model include integration of genome mapping data, nonoverlapping local alignments, matching and comparing molecular structures, and protein docking (Butenko et al. 2006). Another problem is to find a binary code as large as possible which can correct a certain number of errors for a given size of the binary words (vectors) (Brouwer et al. 1990; Sloane 1989). Among these binary words there must be two words which differ in a certain number of positions so that a misspelled word can be detected and corrected (Du and Pardalos 1999). A clique depicts a feasible set of vectors for a code. Error-correcting codes are used in cellular phones, high-speed modems, and CD players (when computing checksums). Finding large cohesive subgroups (cliques) in social networks is used in criminal network analysis. One more application is analysing cliques in a stock market graph (Boginski et al. 2003).

A well-known algorithm for enumerating all cliques in a graph is the algorithm of Bron and Kerbosch (1973). It finds all maximal cliques (cliques which cannot be further enlarged by adding any vertex) in an arbitrary graph. Since this method has only branching and no bound is used to reduce the number of branches, it takes enormous amount of time to find the maximum clique even in small dense graphs. The worst-case running time of the Bron–Kerbosch algorithm is $O(3^{\frac{n}{3}})$ (Jenelius et al. 2006).

One of the first Branch-and-bound algorithms is the algorithm developed by Carraghan and Pardalos (1990). Its main idea is using bound strategy and prune branches in case when future expanding will not lead to a clique with size bigger than the currently best found clique. The bound used in this algorithm is very simple: the maximum clique size in a candidates subgraph is not greater than the number of the candidates. Another idea of this algorithm is to sort vertices in a special order which reduces the size of the search tree. This order suggested by Carraghan and Pardalos (1990) is also used by one the most recent exact algorithms—MCS algorithm developed by Tomita et al. (2010).

The algorithm proposed by Fahle (2002) applies a more efficient bounding strategy. The idea is to use the chromatic number as an upper bound on the size of the maximum clique. To be more precise, a graph of candidate vertices is coloured by means of a heuristic sequential colouring. And the number of colours (which is an upper bound for the chromatic number) is then used as an upper bound for the maximum clique size. This algorithm also applies domain filtering techniques based on the following two observations. First, if there is a vertex in the set of candidates which is connected to all candidates, then this vertex will be included in the clique on many branches. Such vertex should be added to the current clique immediately without any branching. Second, a vertex in the set of candidates must be excluded from consideration if its degree in the subgraph of candidates plus the size of the current clique is less than the size of the best clique found so far.

The MCQ algorithm developed by Tomita and Seki (2003) uses the idea of graph colouring not only as a bounding strategy but also as a branching strategy. Initially vertices are sorted in a non-increasing degree order. At each branching step a greedy standard sequential colouring is computed for the subgraph of candidates. The candidate vertex which is coloured in the biggest colour (colour with the biggest number) is considered first.

The MCR algorithm by Tomita and Kameda (2007) and MCS algorithm by Tomita et al. (2010) are further improvements of MCQ. The only difference between MCQ and MCR is in the ordering of vertices performed at the beginning. In MCR vertices are sorted as it is suggested by Carraghan and Pardalos (1990). In MCS algorithm a new routine is added which tries to recolour a vertex with the biggest colour into a smaller one. Another recent algorithm developed by Li and Quan (2010a) and its improved version (Li and Quan 2010b) apply Max-Sat based upper bound which is tighter than colouring based one. According to the published results for DIMACS graphs MCS algorithm and the Max-Sat based algorithms report the best performance among the existing exact methods.

Since the maximum clique problem is NP-hard (Garey and Johnson 1979), there exists a number of heuristic approaches which can find high-quality solutions. The so-called greedy heuristics either try to create a clique by gradually adding a vertex to a current clique or to find a clique by repeatedly removing a vertex from a current set which is not a clique (Kopf and Ruhe 1987). Genetic algorithms (Marchiori 2002; Singh and Gupta 2006) are based on simplified mechanism of natural system evolution. Such an algorithm starts with some initial randomly generated population and then performs the routines of reproduction, crossover and mutation. The reproduction operator chooses elements which should be taken to the next generation. When such elements are chosen, the crossover operator is applied to produce new descendants. After all, the mutation operator modifies some bits in "genetic codes" of the descendants with some probability.

There are many other heuristics including simulated annealing (Jerrum 1992), neural networks (Bertoni et al. 1997; Funabiki et al. 1992), GRASP (Feo and Resende 1995), tabu search (Glover and Laguna 1997) and others. However in practice, the most successful heuristic algorithms are local search heuristics. At each step a local search heuristic finds a maximal clique, then tries to improve this solution by, for example, a $(j, k) - swap$, that is removing some j vertices from a clique and adding other k vertices to it. Pullan and Hoos (2006) developed a very efficient heuristic called *dynamic local search* which consists of fast neighbourhood search and usage of penalties to promote diversification. Grosso et al. (2008) improved the performance of this dynamic local search algorithm by introducing restart rules. Their GLP algorithm has found new cliques unknown before for very large instances. One of the most efficient heuristic algorithms is iterated local search (ILS) developed by Andrade et al. (2012). The authors of ILS algorithm suggested to use local search instead of an elaborate plateau search applied in GLP algorithm. Their incremental implementation of a local search procedure is faster than the standard one and runs in sublinear time. In this paper we use the ILS algorithm to obtain an initial solution to the maximum clique problem by solving heuristically the maximum independent set problem for a complementary graph.

Though it is a well-known fact that any Branch-and-Bound (BnB) algorithm benefits much when used together with a heuristic applied to obtain an initial solution, none of the existing BnB approaches to the maximum clique problem applies this powerful technique. In this paper we improve one of the recent BnB algorithms, namely MCS algorithm by Tomita et al. (2010). We apply ILS heuristic (Andrade et al. 2012) to obtain an initial high-quality solution which is then used to prune branches in our BnB algorithm. The computational results show that this improvement leads to a considerable reduction of the search tree size especially for large dense graphs.
We also suggest a number of the following improvements:

- Whenever a set of candidates contains a vertex adjacent to all candidates we detect it immediately by means of colouring and thus avoid unnecessary branching.
- Our computational experiments show that for dense graphs with a moderate number of vertices (like the majority of Dimacs graphs) it is more efficient to store vertices of a candidates set and their colours on stack than in dynamic memory on all levels of recursion.
- In contrast to MCS algorithm we use the original Carraghan and Pardalos (1990) sorting of vertices on the first step without any reordering after colouring. Also on the first step when a candidates set contains all the vertices we colour them with a standard greedy sequential colouring (Matula et al. 1972) and then do not reorder any of these vertices as MCS algorithm does.

This paper is organized as follows. In Sect. 2 we formulate the maximum clique problem. In Sect. 3 a detailed description of our algorithm is provided. Computational results showing a comparison with the original MCS algorithm are presented in Sect. 4. Short summary is given in Sect. 5.

2 Maximum clique problem

In this paper we use the following definitions:

- Graph—a simple graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$ is the set of vertices, $E \subseteq V \times V$ is a set of edges.
- Adjacency matrix—a matrix $A = (a_{ij})$ representing graph $G = (V, E)$, where $\forall i, j \in V$ $a_{i,j} = 1$ if $(i, j) \in E$ and $a_{i,j} = 0$ if $(i, j) \notin E$.
- Complementary graph—for a given graph $G(V, E)$ it is the graph $\overline{G}(V, \overline{E})$, where $\overline{E} = \{(i, j) \mid (i, j) \in (V \times V) \setminus E\}$.
- Complete graph—a graph which vertices are all pairwise adjacent, i.e. $\forall i, j \in V, \ (i, j) \in E$.
- Clique—a subset of vertices $C \subset V$ of graph $G(V, E)$ such that all vertices in this subset are pairwise adjacent.
- Maximum clique—a clique which has the maximum size (number of vertices) in a graph.
- Maximal clique—a clique in a graph which cannot be enlarged by adding any vertex of the graph to it.
- Independent set—a subset of vertices $S \subset V$ of graph $G(V, E)$ such that all vertices in this subset are pairwise non-adjacent.

- Candidate—a vertex which can extend the current clique, i.e. a vertex adjacent to all vertices of the clique.
- Colouring—an assignment of colours (natural numbers) to graph vertices such that any two adjacent vertices have different colours.
- Colour—a natural number assigned to a graph vertex in a colouring.
- Chromatic number—the minimum number of colours in which a graph can be coloured.
- Clique number—the maximum clique size of a graph.
- Sequential colouring—a colouring in which vertices are ordered in some sequence and then coloured one by one according to this sequence.
- Reasonable (tight) colouring—a sequential colouring in which a vertex considered on every step is coloured in one of the already used colours if it is possible (otherwise a new colour is introduced).
- Greedy colouring—a sequential colouring in which a vertex considered on every step is coloured in the smallest possible colour.

The maximum clique problem (MCP) refers to the problem of finding the maximum clique in a given graph. The maximum clique size for graph G is denoted by $\omega(G)$. MCP has a number of mathematical programming formulations. One of them is as follows:

$$\max\left\{\sum_{i=1}^{n} x_i \mid x_i + x_j \leq 1, \forall (i, j) \in \overline{E},\ x_i \in \{0, 1\}, i = 1, ..., n\right\}.$$

Here $x_i = 1$ if vertex i is in the maximum clique C and $x_i = 0$ if $i \notin C$.

The maximum independent set problem (MISP) refers to the problem of finding the maximum independent set in a given graph. MCP and MISP are equivalent problems because a clique in graph G is an independent set in the complementary graph of G and vice versa. (See the maximum clique $\{1, 2, 3, 4\}$ in an example graph and the maximum independent set $\{1, 2, 3, 4\}$ in its complementary graph on figure 1).

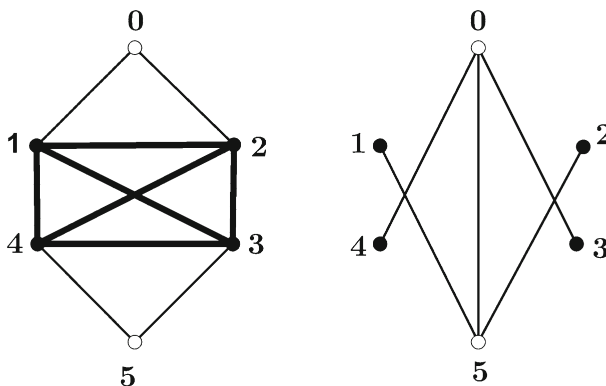

Fig. 1 Maximum clique and maximum independent set problems

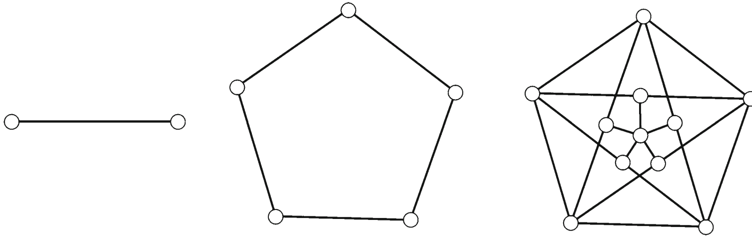

Fig. 2 Mycielski's graphs with chromatic number 2, 3, and 4 and clique number 2

Colouring can be used to obtain an upper bound for the maximum clique problem (Proposition 1).

Proposition 1 *(Balas and Yu 1986) The maximum clique size of an arbitrary graph is not greater than its chromatic number.*

This proposition follows immediately from the fact that a clique of k vertices can be coloured only in k colours because its vertices are all pairwise adjacent. Note that the chromatic number of a graph can be arbitrarily greater than its clique number (maximum clique size). The following theorem of Mycielski demonstrates this fact.

Theorem 1 *(Mycielski 1955) For any natural number n there exists a finite triangle-free graph which cannot be coloured in n colours.*

It is obvious that the maximum clique size of any triangle-free graph cannot be more than two. But the chromatic number of such a graph may be very far from the size of the maximum clique. Figure 2 shows triangle-free graphs with chromatic numbers 2, 3, and 4, and the maximum clique size 2.

It is possible to formulate the notion of a clique in terms of a chromatic number: a clique is a graph which chromatic number is equal to the number of vertices. It can help to identify a clique by means of colouring (see proposition 2).

Proposition 2 *If a graph with k vertices is coloured in k colours by a reasonable colouring, then this graph is complete, i.e. its vertices form a clique.*

Proof By contradiction: let this graph be incomplete. This means that it has at least two non-adjacent vertices. Let these vertices be i and j, and vertex i is coloured before vertex j by our reasonable colouring. Since k vertices are coloured in k colours then every vertex has a unique colour. But then vertex j will be coloured into the same colour as vertex i because no other vertex is coloured in this colour and our colouring is reasonable. This contradicts with the condition that k colours are used. □

Vertices with higher degree are usually contained in larger cliques. For example, a vertex which is adjacent to all other vertices in a graph is contained in all maximal cliques including the maximum one. Such vertices can be found faster after colouring of the given graph (see propositions 3 and 4).

Proposition 3 *If a vertex is adjacent to all vertices of a graph, then it will have a unique colour in any colouring of the graph.*

Proof Another vertex cannot have the same colour in a colouring because it is adjacent to this vertex. □

Proposition 4 *For any greedy colouring if a vertex has a unique colour then it is adjacent to all vertices with greater colour.*

Proof By contradiction: let there be a vertex i with a unique colour k and a vertex j with a greater colour which is not adjacent to i. Vertex j should be coloured in colour k or even smaller colour because no vertices have colour k except i which is not adjacent to j and our greedy colouring always uses a minimal possible colour by definition. This contradicts with the condition that j has a greater colour than k. □

Following the df algorithm (Fahle 2002) we check whether there are candidates which are adjacent to all other candidate vertices and thus should be immediately added to the current clique without any branching. In contrast with the df implementation we can check it much faster using propositions 3 and 4, because we have to check only vertices which have a unique colour after colouring. For such a vertex we should only check if it is adjacent to all vertices with a smaller colour.

Following the MCS algorithm of Tomita et al. (2010) for an upper bound in our algorithm we use the same greedy colouring suggested by Matula et al. (1972).

3 Algorithm description

In all the procedures of our algorithm described below the following variables are used:

- V—the list of vertices in the graph
- E—the list of edges in the graph
- $\bar{E}$—the list of edges in the complementary graph
- A—the adjacency matrix of the graph
- L—a list of candidates
- L^0—a list of candidates sorted according to the initial ordering
- $\tilde{L^0}$—a copy of the list of candidates sorted according to the initial ordering
- Q—the current clique
- Q^*—the largest clique found so far
- $|X|$—the size of set (or array, list, clique) X
- C_k—a list of vertices coloured in colour k
- $C_k[i]$—the i-th vertex in list C_k
- υ—a graph vertex
- c_υ—the colour number assigned to vertex υ
- $N_L(\upsilon)$—the neighbourhood of vertex υ in subgraph of candidates L i.e. the set of vertices adjacent to υ from L
- ρ_υ—the degree of vertex υ
- σ_υ—the sum of vertex υ neighbours degrees (the degree of υ neighbourhood)

The main procedure (algorithm 1) of our algorithm $MCSWithHeuristic()$ first runs ILS heuristic to obtain an initial solution of high quality and writes this solution to Q^*. Then we perform initial ordering of all vertices same as in Carraghan and

Pardalos (1990) and store the ordered vertices in list L^0. Vertices in L^0 are then coloured sequentially in a standard way (using the minimal possible colour on each step). The colouring is stored in an array of lists C_k so that for any colour k we can quickly get all vertices coloured in k. For every vertex υ we also store its colour c_υ so that we can always quickly get this colour. We branch on every vertex of L^0 in inverse order (starting from the last vertex). Such branching rule was first applied in Carraghan and Pardalos (1990) and proved to provide a smaller search tree.

Algorithm 1 MCS with incorporated ILS heuristic and other improvements

```
function MCSWithHeuristic( )
  HeuristicSolution( )
  InitialOrderingAndColouring(L^0)
  for i = |L^0|, 1 do                          ▷ L^0 is not ordered by colour numbers
    υ = L^0_i
    if UpperBound(υ) > |Q*| then
      ProcessBranch(υ, L^0)
    end if
  end for
end function
```

To apply ILS heuristic (algorithm 2) we build the complementary graph and run ILS algorithm for it. An independent set found by ILS is a clique in the original graph. We use this clique as an initial solution. A complete description of ILS algorithm can be found in Andrade et al. (2012).

Algorithm 2 Run ILS heuristic on the complementary graph

```
function HeuristicSolution( )
  Q* = ILS(V, Ē)                 ▷ See ILS description in Andrade et al. (2012)
end function
```

Initial ordering (algorithm 3) is performed in the same way as in Tomita et al. (2010). Find the vertex υ with the minimum degree ρ_υ in the graph and place it to the last position $|V|$ of candidates list L^0. If several vertices have the same minimum degree then take the vertex which has the minimum support σ_υ (sum of the vertex neighbors degrees) among them. For several vertices having the same minimum support ties are broken arbitrarily. Then delete this vertex from the graph and find the next vertex with the minimum degree in the remaining graph in the same way. This vertex is placed to position $|V| - 1$ of list L^0. An so on until all vertices are placed into L^0. Then the standard sequential colouring is applied to vertices in L^0. Following MCS algorithm we also permute the adjacency matrix A.

The colouring algorithm (algorithm 4) together with its "re-number" improvement (algorithm 5) are taken from MCS algorithm.

The permutation of the adjacency matrix (algorithm 6) is made so that the vertices in L^0 have numbers $1, 2, ..., n$ after the permutation. This increases the CPU cache usage because candidates are always taken in L^0 ordering and so the adjacency matrix is also accessed in this order (Tomita et al. 2010).

Algorithm 3 Initial ordering and colouring

```
function InitialOrderingAndColouring(L^0)
    for i = 1, |V| do
        ρ_i = |N_V(i)|
        σ_i = Σ_{j∈N_V(i)} ρ_j
    end for
    for i = |V|, 1 do
        υ ← vertex with min ρ, and min σ if there are several vertices with min ρ
        L^0_i = υ
        ρ_υ = 0                                  ▷ Remove this vertex from further consideration
        for j ∈ N_V(υ), ρ_j ≠ 0 do               ▷ Correct degrees of its neighbours
            ρ_j = ρ_j − 1
        end for
        for i = 1, |V| do                        ▷ Recompute neighborhood degrees for all vertices
            σ_i = Σ_{j∈N_V(i)} ρ_j
        end for
    end for
    SolveRelaxedProblem(L^0)
    PermuteVertices(L^0)
end function
```

Algorithm 4 Colour vertices

```
function SolveRelaxedProblem(L)
    k_0 = |Q*| − |Q|
    k_max = 1
    for i = 1, |L| do
        for k = 1, |L| do
            if C_k ∩ N_L(L_i) = ∅ then
                break
            end if
        end for
        C_k = C_k ∪ {L_i}
        if k > k_max then
            k_max = k
            if k > k_0 then
                ReNumber(L_i, k)
                if C_k = ∅ then
                    k_max = k_max − 1
                end if
            end if
        end if
    end for
end function
```

The colour c_υ of vertex υ actually shows the number of colours in which this vertex together with its neighbours in candidates subgraph can be coloured when we branch on this vertex (algorithm 7). So the current clique size plus this number give the maximum possible clique size for the current branch.

For the branch of vertex υ (algorithm 8) we first remove this vertex from candidates list L^0, then find a new candidates list L which has only neighbours of υ in the

Algorithm 5 Recolour vertices

```
function RENUMBER(p, k)
    k_0 = |Q*| − |Q|
    for k_1 = 1, k_0 − 1 do
        if |C_{k_1} ∩ N_L(p)| = 1 then
            q = (C_{k_1} ∩ N_L(p))[1]
            for k_2 = k_1 + 1, k_0 do
                if C_{k_2} ∩ N_L(q) = ∅ then
                    C_k = C_k \ {p}
                    C_{k_1} = C_{k_1} \ {q} ∪ {p}
                    C_{k_2} = C_{k_2} ∪ {q}
                    return
                end if
            end for
        end if
    end for
end function
```

Algorithm 6 Permute adjacency matrix according to the initial ordering

```
function PERMUTEVERTICES(L^0)
    A' = A
    for i = 1, |L^0| do
        for j = 1, |L^0| do
            A_{ij} = A'_{L_i L_j}
        end for
    end for
    for i = 1, |L^0| do
        L^0_i = i
    end for
end function
```

Algorithm 7 Upper bound using colouring

```
function UPPERBOUND(υ)
    return |Q| + c_υ          ▷ Return maximum possible clique size for this branch
end function
```

candidates subgraph. If there are no neighbours then we have found an inclusion-maximal clique and should return from recursion. If this clique is greater than the currently best clique then it replaces the best clique. If there are new candidates we colour them and check if they form a clique. If it is so we return from recursion. Otherwise we check if there are vertices which should be added to the current clique Q without branching. Then list L is sorted according to the colours of the vertices. We add vertex υ to the current clique Q and recursively process the new candidates. After it the current clique is restored to be used on other branches.

Using proposition 2 function $DetectClique()$ (algorithm 9) checks if the candidates in list L form a clique. If it is so then this clique is added to the current clique Q and if the obtained clique is larger than Q^* then it is stored to Q^*.

Function $DetectCliqueVertices()$ (algorithm 10) uses propositions 3 and 4 to find vertices adjacent to all candidates in L. According to proposition 3 only vertices coloured in a unique colour should be checked. Moreover according to proposition 4

Algorithm 8 Process one branch

```
function PROCESSBRANCH(υ, L^0)
    L^0 = L^0 \ {υ}
    L~^0 = L = N_{L^0}(υ)                          ▷ Find new candidates list
    if L = ∅ then
        if |Q| ≥ |Q*| then                         ▷ Check if this clique is the largest one found so far
            Q* = Q ∪ {υ}
        end if
        return                                     ▷ Return since the current clique is inclusion-maximal
    end if
    SOLVERELAXEDPROBLEM(L)
    Q^0 = Q                                        ▷ Save the current clique to restore it then
    if DETECTCLIQUE(L) then
        return                                     ▷ Return if L forms a clique
    else
        DETECTCLIQUEVERTICES(L~^0, L)
    end if
    Quick sort L by colours C_k so that vertex L_1 has colour 1
    Q = Q ∪ {υ}
    PROCESSBRANCHES(L~^0, L)
    Q = Q^0                                        ▷ Restore the current clique
end function
```

Algorithm 9 Check if candidates from L form a clique

```
function DETECTCLIQUE(L)
    k_max = max{k|C_k ≠ ∅}
    if |L| = k_max then
        if |Q| + |L| > |Q*| then
            Q* = Q ∪ L
        end if
        return true
    end if
    return false
end function
```

such vertices are always adjacent to vertices with a greater colour. So we check adjacency only with vertices having a smaller colour.

Function $ProcessBranches()$ (algorithm 11) considers candidates starting from the vertex with the greatest colour and ending when the upper bound is not greater than the current best clique size.

It is well known that applying a fast heuristic before running an exact BnB algorithm always reduces the search tree size and usually reduces the total running time. A heuristic finds a high-quality solution which is not very far from the optimal solution. This solution is then used to prune branches which have an upper bound not greater than the value of the objective function for this solution.

Along with combining of the BnB algorithm with the heuristic we suggest a number of improvements reducing the computational time. Our first improvement is avoiding of dynamic memory allocation which allows to save the computational time spent on memory allocating/deallocating every time a new candidates set is formed and coloured. Instead of working with dynamic memory we allocate stack memory of

Algorithm 10 Check if some candidates from L are adjacent to all vertices in L

```
function DetectCliqueVertices(L^0, L)
  for k = 1, max{k|C_k ≠ ∅} do
    if |C_k| > 1 then                      ▷ Check only vertices with a unique colour
      continue
    end if
    υ = C_k[1]
    for l = 1, k − 1 do                     ▷ Check adjacency only to vertices with a smaller colour
      for i = 1, |C_l| do
        ω = C_l[i]
        if A_υω == 0 then
          goto next-color
        end if
      end for
    end for
    Q = Q ∪ {υ}                             ▷ Add υ adjacent to all candidates to Q without branching
    L^0 = L^0 \ {υ}
    L = L \ {υ}
    next-color:
  end for
  if |Q| > |Q*| then
    Q* = Q
  end if
end function
```

Algorithm 11 Process branches corresponding to candidates from L

```
function ProcessBranches(L^0, L)
  for i = |L|, 1 do                         ▷ L is ordered by colour numbers
    υ = L_i
    if UpperBound(υ) > |Q*| then
      ProcessBranch(υ, L^0)
    else
      return                                ▷ Next vertices have smaller colour and upper bound
    end if
  end for
end function
```

a predetermined size enough for any number of vertices up to 1,500. This constant can be easily changed (it only requires the program recompilation). However, too big value of this constant quickly increases the size of the stack and causes a lot of memory swapping performed by the operating system. This, of course, slows down the overall performance of the program. Since we use a depth-first search strategy in our BnB algorithm then we have to store vertices of a candidates set together with their colours on all levels of recursion. This needs many allocation/deallocation operations. If the memory is allocated on the stack, its size is predefined and the only operation performed by the processor is the movement of the stack pointer. In contrast with it, in the case when dynamic memory is allocated/deallocated on the heap a lot of processor operations are needed when the context is switched to the operating system mode and back and when the heap manager finds a free chunk of memory or adds a deallocated chunk to the free memory. So we improve the performance of the program by eliminating dynamic memory allocation (see Table 2).

The next our improvement is based on propositions 2, 3, and 4. If a subgraph of candidates has k vertices and is coloured in k colours then it is a clique. In this case we do not need to branch any more and waste time for finding candidates. It costs nothing to check it, because on every step of branching we colour the subgraph of candidates. Applying propositions 3 and 4 we also quickly find the vertices adjacent to all candidates and then add them to the current clique without branching. This approach reduces the search tree size and the running time of our algorithm (see Table 3).

Another our improvement is induced by the order of vertices. In MCS algorithm vertices are sorted in ascending order of their colours after colouring. But at the beginning of the algorithm the vertices are sorted as in Carraghan and Pardalos (1990), because it provides a smaller search tree. In order to keep this ordering after colouring MCS algorithm does not use the standard sequential colouring on the first step. It simply assigns a unique colour to every vertex of the whole graph except a special set $Rmin$ (Tomita et al. 2010) without considering any connections between the vertices. As a result many vertices have a big colour number and thus MCS algorithm has a lot of branching on the first level of the search tree. We suggest to use a standard sequential colouring on the first step, but do not reorder the vertices by their colour numbers. So on the first level we have to keep in the candidates set those vertices for which their colour number plus the current clique size is not greater than the currently best clique size. Our experimental results show that our approach reduces the search tree size and the overall running time (see Table 1).

4 Computation results

We perform our computational study on Intel Core i7 machine with 2.3 GHz CPU and 8 Gb of memory. First we test initial colouring, memory usage, and clique detection improvements separately. Tables 1, 2, and 3 present the results for every improvement of MCS algorithm tested separately. The experiments are performed on random graphs with edge probabilities distributed uniformly in [0,1]. Tables 1, 2, and 3 show the

Table 1 Search tree size and running time with initial colouring improvement

Vertices	Density	Running time (%)	Search tree size (%)
200	0.95	90.5	91.1
200	0.90	95.1	96.0
800	0.50	100.1	100.3
10000	0.16	99.8	100.0

Table 2 Running time with memory usage improvement

Vertices	Density	Running time (%)
200	0.95	91.2
200	0.90	94.6
800	0.50	75.1

Table 3 Search tree size and running time with clique detection improvement

Vertices	Density	Running time (%)	Search tree size (%)
200	0.95	98.4	97.1
200	0.90	99.1	98.0
800	0.50	99.7	99.2
10000	0.16	100.0	100.0

ratio of search tree size and running time of improved algorithm to search tree size and running time of original MCS algorithm. For every considered combination of vertices number and density 50 instances are generated and the average running time and search tree size are computed. The number of vertices and density of the generated instances are chosen so that solving of one instance takes not more than 10 min (or 500 min for 50 instances) and thus all experiments can be performed in a reasonable time. Note that memory usage improvement reduces only the running time of the program. For large sparse graphs with 10,000 vertices or more it is not possible to use the stack memory in our implementation because we use an adjacency matrix and not adjacency lists.

We test our algorithm with all improvements on DIMACS benchmark instances. The computational results for graphs from DIMACS library are presented in Tables 4 and 5. Table 4 shows the search tree size for our algorithm and for MCS algorithm implemented in complete accordance with the paper of Tomita et al. (2010). The last two columns contain the size of the maximum clique and the size of the best clique found by the ILS heuristic applied in our algorithm. It is clear that the better clique is found by the ILS heuristic the greater is the reduction in the search tree size we observe. We run the ILS heuristic with 100,000 scans for all the considered instances except *gen400_p0.9_55* and *p_hat1000-3* for which we use 60 millions scans because these two instances are very hard both for the ILS heuristic and for our BnB algorithm. The greatest reduction of the search tree size compared to MCS algorithm is obtained for *gen*, *san*, *c-fat*, and *p_hat1000-3* instances. It varies from 60 times reduction for *gen400_p0.9_55* up to more than 7,000 times reduction for *gen400_p0.9_65* and several *san* instances. This is because for these graphs colouring usually gives a good upper bound and when the maximum clique is already found a lot of branches are pruned quickly.

The comparison of the total computational time for our algorithm and MCS algorithm is given in Table 5. The second column contains the running time of ILS heuristic. The third and the fourth columns contain the total running time of our and MCS algorithm correspondingly. For hard instances (which need more than 15 min to be solved by MCS algorithm) our approach shows better results and reduces the computational time by 35 % in average. The total time of computing all DIMACS graphs is reduced by 75% (MCS time for *p_hat1000-3* instance is taken equal to 1,000,000 s). The best results are obtained for *gen400_p0.9_75* (our algorithm is 13,000 times faster for it), *gen400_p0.9_65* (3000 times faster), *gen400_p0.9_55* (7 times faster), and *p_hat1000-3* graphs. MCS algorithm is unable to solve *p_hat1000-3* instance (at least in 10 days) while our algorithm solves it in 3 days. However, our approach is usually slower for simple graphs (solved in less than 15 min by MCS) than MCS because it is not efficient to perform 100,000 scans of the ILS heuristic for such graphs.

Table 4 Search tree size

Instance	MCS	Our algorithm	Maximum clique	ILS solution
brock200_1	266180	237473	21	21
brock200_2	3505	2275	12	12
brock200_3	13016	12728	15	15
brock200_4	51526	27135	17	17
brock400_1	88555048	87946118	27	23
brock400_2	34145195	30682956	29	24
brock400_3	66379744	66280298	31	24
brock400_4	29696341	17963868	33	26
brock800_1	1097174023	1095645796	23	19
brock800_2	972110520	970862419	24	20
brock800_3	625234820	625139200	25	19
brock800_4	424176492	424101537	26	19
c–fat200–1	188	3	12	12
c–fat200–2	176	0	24	24
c–fat200–5	142	26	58	58
c–fat500–1	486	0	14	14
c–fat500–10	374	0	126	126
c–fat500–2	474	0	26	26
c–fat500–5	436	0	64	64
gen200_p0.9_44	96070	81052	44	44
gen200_p0.9_55	288654	1739	55	55
gen400_p0.9_55	3425049256	55079436	55	54
gen400_p0.9_65	6500277298	822991	65	64
gen400_p0.9_75	10140428816	41445	75	75
hamming10–2	511	255	512	473
hamming6–2	31	0	32	32
hamming6–4	81	80	4	4
hamming8–2	127	0	128	128
hamming8–4	35347	35336	16	16
johnson16–2–4	293670	256098	8	8
johnson8–2–4	30	22	4	4
johnson8–4–4	125	114	14	14
keller4	8441	8214	11	11
keller5	10339211493	10337321299	27	27
MANN_a27	33345	55	126	126
MANN_a45	221476	219979	345	344
MANN_a9	799002	44	16	16
p_hat1000–1	171929	165611	10	10
p_hat1000–2	25209207	21587044	46	44
p_hat1000–3	–	8773710250	68	67

Table 4 continued

Instance	MCS	Our algorithm	Maximum clique	ILS solution
p_hat1500–1	1086203	1007917	12	11
p_hat1500–2	660539819	607200969	65	61
p_hat300–1	1876	1201	8	8
p_hat300–2	3526	2334	25	25
p_hat300–3	565792	265698	36	35
p_hat500–1	9903	9306	9	9
p_hat500–2	89836	56552	36	34
p_hat500–3	17259920	15398976	50	48
p_hat700–1	29656	15827	11	11
p_hat700–2	670369	416003	44	42
p_hat700–3	98911559	81631372	62	60
san1000	188132	0	15	15
san200_0.7_1	990	0	30	30
san200_0.7_2	1262	0	18	18
san200_0.9_1	83047	0	70	70
san200_0.9_2	11118	0	60	60
san200_0.9_3	15708	0	44	44
san400_0.5_1	3197	0	13	13
san400_0.7_1	64568	0	40	40
san400_0.7_2	23471	0	30	30
san400_0.7_3	253044	0	22	22
san400_0.9_1	20537	4	100	100
sanr200_0.7	115666	98857	18	18
sanr200_0.9	8103466	4723495	42	42
sanr400_0.5	245271	239833	13	12
sanr400_0.7	51507583	51501398	21	20

5 Summary and future research directions

In this paper we report a number of improvements incorporated in one of the best BnB algorithms, namely MCS algorithm by Tomita et al. (2010) for solving the maximum clique problem. We have enriched the MCS algorithm by means of the high-quality ILS heuristic by Andrade et al. (2012) which reduces the search tree size essentially. The most impressive reduction of the search tree size evaluates by more than 7,000 times for *gen400_p0.9_65* benchmark instance. Our algorithm is able to solve *p_hat1000-3* instance which cannot be solved in a reasonable time by MCS algorithm.

It seems that all well-known BnB algorithms for solving the maximum clique problem, as far as we are aware, have not used any efficient heuristic at least described explicitly (see e.g. Carmo and Zuge 2012). Hence it will be worthwhile to experiment with different high-quality heuristics incorporated into different BnB algorithms.

Table 5 Computational time

Instance	ILS heuristic	Our algorithm	MCS
brock200_1	6	7	1
brock200_2	12	12	0
brock200_3	10	10	0
brock200_4	9	9	0
brock400_1	26	483	460
brock400_2	25	212	199
brock400_3	26	346	320
brock400_4	25	135	161
brock800_1	167	6482	6337
brock800_2	164	5886	5737
brock800_3	166	3994	3830
brock800_4	166	3009	2849
c–fat200–1	11	11	0
c–fat200–2	10	10	0
c–fat200–5	7	7	0
c–fat500–1	62	62	0
c–fat500–10	42	42	0
c–fat500–2	63	63	0
c–fat500–5	56	56	0
gen200_p0.9_44	2	3	0
gen200_p0.9_55	2	2	1
gen400_p0.9_55	4320	5133	39015
gen400_p0.9_65	8	25	77620
gen400_p0.9_75	7	8	110579
hamming10–2	5	0	0
hamming6–2	0	0	0
hamming6–4	2	2	0
hamming8–2	1	1	0
hamming8–4	14	14	0
johnson16–2–4	2	2	0
johnson8–2–4	0	0	0
johnson8–4–4	1	1	0
keller4	7	7	0
keller5	96	78957	78875
MANN_a27	2	4	0
MANN_a45	5	130	126
MANN_a9	0	0	0
p_hat1000–1	302	302	0
p_hat1000–2	172	360	272

Table 5 continued

Instance	ILS heuristic	Our algorithm	MCS
p_hat1000–3	54185	214611	>1000000
p_hat1500–1	693	696	3
p_hat1500–2	346	10231	11859
p_hat300–1	27	27	0
p_hat300–2	17	17	0
p_hat300–3	10	11	3
p_hat500–1	75	75	0
p_hat500–2	41	42	1
p_hat500–3	24	171	164
p_hat700–1	147	147	0
p_hat700–2	77	81	6
p_hat700–3	44	1303	1529
san1000	464	464	3
san200_0.7_1	6	6	0
san200_0.7_2	7	7	0
san200_0.9_1	2	2	0
san200_0.9_2	2	2	0
san200_0.9_3	2	2	0
san400_0.5_1	65	65	0
san400_0.7_1	26	26	1
san400_0.7_2	33	33	0
san400_0.7_3	35	35	2
san400_0.9_1	6	6	1
sanr200_0.7	7	7	1
sanr200_0.9	2	28	42
sanr400_0.5	49	50	1
sanr400_0.7	31	180	171

Acknowledgements The authors would like to thank professor Mauricio Resende and his co-authors for the source code of their powerful ILS heuristic. The authors are supported by LATNA Laboratory, National Research University Higher School of Economics (NRU HSE), Russian Federation government grant, ag. 11.G34.31.0057. Boris Goldengorin and Mikhail Batsyn are partially supported by NRU HSE Scientific Fund grant "Teachers-Students" #11-04-0008 "Calculus for tolerances in combinatorial optimization: theory and algorithms".

References

Andrade DV, Resende MGC, Werneck RF (2012) Fast local search for the maximum independent set problem. J Heuristics 18(4):525–547. doi:10.1007/s10732-012-9196-4

Balas E, Yu CS (1986) Finding a maximum clique in an arbitrary graph. SIAM J Comput 15(4):1054–1068

Bertoni A, Campadelli P, Grossi G (1997) A discrete neural algorithm for the maximum clique problem: analysis and circuit implementation. In: Proceedings of workshop on algorithm, engineering, WAE'97, pp 84–91

Boginski V, Butenko S, Pardalos PM (2003) Innovations in financial and economic networks. In: Nagurney A (ed) On structural properties of the market graph. Edward Elgar Publishing, London, pp 29–45

Bomze I, Budinich MPMP, Pelillo M (1999) The maximum clique problem. In: Handbook of combinatorial optimization. Kluwer Academic Publishers, Boston

Bron C, Kerbosch J (1973) Algorithm 457: finding all cliques of an undirected graph. Commun ACM 16(9):575–577. doi:10.1145/362342.362367

Brouwer A, Shearer J, Sloane N, Smith W (1990) A new table of constant weight codes. IEEE Trans Inf Theory 36(6):1334–1380. doi:10.1109/18.59932

Butenko S, Wilhelm WE (2006) Clique-detection models in computational biochemistry and genomics. Eur J Oper Res 173(1):1–17. doi:10.1016/j.ejor.2005.05.026

Carmo R, Zuge A (2012) Branch and bound algorithms for the maximum clique problem under a unified framework. J Braz Comput Soc 18(2):137–151

Carraghan R, Pardalos PM (1990) An exact algorithm for the maximum clique problem. Oper Res Lett 9(6):375–382. doi:10.1016/0167-6377(90)90057-C

Du D, Pardalos PM (1999) Handbook of combinatorial optimization, Supplement, vol A. Springer, New York

Fahle T (2002) Simple and fast: improving a branch-and-bound algorithm for maximum clique. In: Proceedings of the 10th annual European symposium on algorithms, ESA '02. Springer-Verlag, London, pp 485–498

Feo TA, Resende MGC (1995) Greedy randomized adaptive search procedures. J Global Optim 6(2):109–133. doi:10.1007/BF01096763

Funabiki N, Takefuji Y, Lee KC (1992) A neural network model for finding a near-maximum clique. J Parallel Distrib Comput 14(3):340–344. doi:10.1016/0743-7315(92)90072-U

Garey MR, Johnson DS (1979) Computers and intractability: a guide to the theory of NP-completeness, vol 24. W. H Freeman and Co, New York

Glover F, Laguna M (1997) Tabu search. Kluwer Academic Publishers, Norwell

Grosso A, Locatelli M, Pullan W (2008) Simple ingredients leading to very efficient heuristics for the maximum clique problem. J Heuristics 14(6):587–612. doi:10.1007/s10732-007-9055-x

Jenelius E, Petersen T, Mattsson L (2006) Importance and exposure in road network vulnerability analysis. Transport Res A Policy Pract 40(7):537–560. doi:10.1016/j.tra.2005.11.003

Jerrum M (1992) Large cliques elude the metropolis process. Random Struct Algorithms 3(4):347–359. doi:10.1002/rsa.3240030402

Kopf R, Ruhe G (1987) A computational study of the weighted independent set problem for general graphs. Found Control Eng 12(4): 167–180

Li CM, Quan Z (2010a) Combining graph structure exploitation and propositional reasoning for the maximum clique problem. In: Proceedings of the 2010 22nd IEEE international conference on tools with artificial intelligence, Vol 01, ICTAI'10. IEEE, Arras, pp 344–351

Li CM, Quan Z (2010b) An efficient branch-and-bound algorithm based on maxsat for the maximum clique problem. In: Proceedings of the twenty-fourth AAAI conference on artificial intelligence, AAAI-10. AAAI Press, Atlanta, pp 128–133

Marchiori E (2002) Genetic, iterated and multistart local search for the maximum clique problem. In: Applications of evolutionary computing. Springer-Verlag, New York, pp 112–121

Matula DW, Marble G, Isaacson JD (1972) Graph coloring algorithms. In: Graph theory and computing. Academic Press, New York, pp 109–122

Mycielski J (1955) Sur le coloriage des graphes. Colloq Math 3:161–162

Pullan W, Hoos HH (2006) Dynamic local search for the maximum clique problem. J Artif Int Res 25(1):159–185

Singh A, Gupta AK (2006) A hybrid heuristic for the maximum clique problem. J Heuristics 12(1–2):5–22. doi:10.1007/s10732-006-3750-x

Sloane NJA (1989) Unsolved problems in graph theory arising from the study of codes. Graph Theory Notes NY 18(11):11–20

Tomita E, Kameda T (2007) An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. J Global Optim 37(1):95–111

Tomita E, Seki T (2003) An efficient branch-and-bound algorithm for finding a maximum clique. In: Proceedings of the 4th international conference on discrete mathematics and theoretical computer science, DMTCS'03. Springer-Verlag, Berlin, Heidelberg, pp 278–289

Tomita E, Sutani Y, Higashi T, Takahashi S, Wakatsuki M (2010) A simple and faster branch-and-bound algorithm for finding a maximum clique. In: Proceedings of the 4th international conference on algorithms and computation, WALCOM'10. Springer-Verlag, Berlin, Heidelberg, pp 191–203