On finding k-cliques in k-partite graphs

M. Mirghorbani and P. Krokhmal*

Department of Mechanical and Industrial Engineering
University of Iowa, Iowa City, IA 52242
e-mail: {smirghor,krokhmal}@engineering.uiowa.edu

Abstract

In this paper, a branch-and-bound algorithm for finding all cliques of size k in a k-partite graph is proposed that improves upon the method of Grunert et al (2002). The new algorithm uses bit-vectors, or bitsets, as the main data structure in bit-parallel operations. Bitsets enable a new form of data representation that improves branching and backtracking of the branch-and-bound procedure. Numerical studies on randomly generated instances of k-partite graphs demonstrate competitiveness of the developed method.

Keywords: maximum clique enumeration problem, k-partite graph, k-clique, bit parallelism

1 Introduction

Given an (undirected) graph $G = (V, E)$, where V is set of nodes and E is the set of arcs, a *clique* in G is defined as a complete subset of G, i.e., a set of nodes in V that are pairwise adjacent. A clique of size k is called *k-clique*;[1] the largest clique in a graph is called the *maximum clique* and its size is denoted by $\omega(G)$. Note that G may contain several cliques of size $\omega(G)$. Closely related to the concept of a clique is that of an *independent set* of G, defined as an induced subgraph of V whose nodes are pairwise disjoint.

The Maximum Clique Problem (MCP) consists in finding the largest clique in a graph, and is of fundamental importance in discrete mathematics, computer science, operations

*Corresponding author.

[1]It is worth noting that the term *k-clique* is used in several different contexts in the literature; for instance, one of its alternative interpretations is that of a subgraph where any two nodes are connected by a path of length at least k [10]. In this work, we use the definition of k-clique as given above.

research, and related fields [1]. In many applications it is of interest to identify all maximum cliques in a graph. This problem is known as the Maximum Clique Enumeration Problem (MCEP). In the present work, we consider a special case of the MCEP, concerned with finding all k-cliques in a k-partite graph. A graph $G = (V, E)$ is called k-partite if the set of nodes V can be partitioned into k independent sets, or *partites* V_r, $r = 1, \ldots, k$:

$$V = \bigcup_{r=1}^{k} V_r, \quad V_r \cap V_s = \emptyset, \quad r \neq s, \text{ such that for all } i, j \in V_r : (i, j) \notin E. \tag{1}$$

Clearly, one has that $\omega(G) \leq k$ in a k-partite graph G, since the maximum clique cannot contain more than one node from each independent set V_r. Note also that the problem of finding all k-cliques in a k-partite graph is not equivalent to MCEP since it does not account for maximum cliques with $\omega(G) < k$.

The problem of finding k-cliques in k-partite graphs has applications in many areas of science and engineering, including textile industry [3], where the braiding problem can be reduced to the problem of finding k-cliques in the path compatibility graph that represents a k-partite graph; data mining, particularly for clustering of categorical attributes over k-domains [12]; identification of protein structures [9], where protein interaction network is represented by a k-partite graph that is mined for k-cliques. Recently, it has been shown that the problem of finding k-cliques in k-partite graphs can be used to find high-quality solutions of large-scale randomized instances of multidimensional assignment problem (MAP) [6, 7, 11].

Grunert et al [3] proposed branch-and-bound algorithm FINDCLIQUE for the problem of finding all k-cliques in k-partite graphs, which takes as an input a graph $G = (V, E)$, where V satisfies (1), and produces the set Q of k-cliques contained in G as an output. FINDCLIQUE is a recursive method, such that level t of recursion corresponds to the level t of branch-and-bound tree, which in turn, is associated with the t-th partite that is branched on in V. Starting at the root ($t = 0$) of the branch-and-bound tree with a partial solution $S = \emptyset$, at each step of branch-and-bound procedure a node is added to or removed from S until S amounts to a k-clique in G, i.e., $|S| = k$, or it is verified that G contains no k-cliques, $\omega(G) < k$.

Let $B = \{1, \ldots, k\}$ be the index set of partites in G, $V = \bigcup_{b \in B} V_b$, and B_S denote the set of partites that have a node in S:

$$B_S = \{b \in B \mid V_b \cap S \neq \emptyset\}.$$

Given a partial solution S, a node is called *compatible* if it is adjacent to all the nodes in S; the set of compatible nodes w.r.t. S is denoted by C_S:

$$C_S = \{i \in V \mid (i, j) \in E \ \ \forall j \in S\}.$$

The set C_S is further partitioned into subsets containing nodes from the same partite:

$$C_S = \bigcup_{b \in \overline{B}_S} C_{S,b},$$

where $\overline{B}_S = B \setminus B_S$, and $C_{S,b} \subseteq V_b$ is given by

$$C_{S,b} = \bigcup_{s \in S} (V_b \cap N(s)),$$

with $N(s)$ being the set of nodes adjacent to node s.

At the root node of the branch-and-bound tree ($t = 0$), one has $S = \emptyset$, $B = \overline{B}_S = \{1, \ldots, k\}$, $B_S = \emptyset$, and $C_{S,b} = V_b$ for all $b \in B$. At a level t of the branch-and-bound tree, $b_t \in \overline{B}_S$ is selected as the partition to branch on. In order to achieve the greatest reduction in the size of the branch-and-bound tree when pruning, b_t is selected as the partition with the smallest number of nodes:

$$b_t \in \arg\min_b \{|C_{S,b}| \mid b \in \overline{B}_S\}. \tag{2}$$

As long as there is a node $n_t \in C_{S,b_t}$ that is not traversed, the search process is restarted from this point with $S := S \cup \{n_t\}$ as the new partial solution. To this end, the set C_S of compatible nodes is updated with respect to $S \cup \{n_t\}$:

$$C_{S,b} := C_{S,b} \cap N(n_t) \text{ for all } b \in \overline{B}_S. \tag{3}$$

Maintaining the sets $C_{S,b}$ of nodes compatible with the current partial solution S is a key aspect of the algorithm, thus for backtracking purposes the nodes that are removed from $C_{S,b}$ during (3) are added to the set $\overline{C} = \bigcup_{t=1}^{k} \overline{C}_t$, which is similarly partitioned into k levels $\overline{C}_t$, each level corresponding to level t of the branch-and-bound tree. In other words, $\overline{C}_t$ contains the nodes in $C_{S,b}$ that are not adjacent to node n_t:

$$\overline{C}_t = \{i \in C_{S,b} \mid (i, n_t) \notin E,\ b \in \overline{B}_S\}.$$

Obviously, after this step, $C_{S,b_t} = \emptyset$. A subproblem with a partial solution S is *promising* if all of the partitions in C_S that do not share a node in the partial solution are nonempty:

$$|C_{S,b}| > 0 \text{ for all } b \in \overline{B}_S,\ \ b \neq b_t. \tag{4}$$

Let P be the number of partitions $C_{S,b} \subseteq C_S$ that contain at least one node; then, an upper bound on the size of the largest clique containing S is given by $|S| + P$. If $|S| + P = k$, the current subproblem is feasible, meaning S may be part of a k-clique. For a feasible subproblem, the algorithm traverses deeper into the branch-and-bound tree, $t := t + 1$, and a new subproblem is created.

Accordingly, a subproblem with partial solution S is pruned if

$$|S| + P < k, \tag{5}$$

i.e., there exists no clique of size k that contains S. For a nonpromising subproblem, set C_{S,b_t} is restored by moving the nodes in $\overline{C}_t$ back to C_S, $C_S := C_S \cup \overline{C}_t$. The last operation implicitly requires that the nodes from $\overline{C}_t$ are put back into the partitions of C_S that they were removed from:

$$C_{S,\pi(v)} := C_{S,\pi(v)} \cup v \;\text{ for all }\; v \in \overline{C}_t, \tag{6}$$

where $\pi(i)$ is the index of the partite that node i belongs to: $i \in V_{\pi(i)}$; moreover, the relative orders of nodes in the partites V_b should be preserved in $C_{S,b}$, given that the nodes in G are assumed to be ordered/numbered.

The search process is then restarted, provided that there exists a node in partition C_{S,b_t} that is not traversed. If there is no such node, FINDCLIQUE returns to the previous level $t-1$ of the branch-and-bound tree.

2 A bitwise algorithm for finding k-cliques in a k-partite graph

In this section, we present an algorithm, referred to as BitCLQ, for the k-clique enumeration problem in a k-partite graph, which improves upon the FINDCLIQUE algorithm of Grunert et al [3] by introducing bitset data structures and utilizing bit parallelism for updating the set of compatible nodes and improving backtracking.

2.1 Bitsets

Bitsets are essentially binary vectors, or sequences of bits, and as such can be utilized efficiently in computer codes. Particularly, bitsets are useful for storing adjacency matrices of graphs, or specific subsets of ordered sets. For example, in a graph on six nodes $\{v_1, \ldots, v_6\} = V$, a clique with nodes v_1, v_2, v_3, v_5 can be represented by a bitset $\{111010\}$, where each bit corresponds uniquely to a node in the graph, with the *significant* bits (i.e., bits equal to 1) indicating the nodes in the clique. *Bit parallelism* is a form of parallel computing that achieves computational improvements by representing the problem data in bitsets of size R, where R is the machine word size (e.g., 32 or 64), such that they can be processed together within a single processor instruction. Bit parallelism has been successfully used in many computational algorithms, particularly for string matching [2, 4, 5]. Recently, bit parallelism has been employed for solving hard combinatorial problems, such as SAT [14] and the Maximum Clique Problem [13].

In the present work, bit parallelism is used to improve the computational procedure for updating the set of compatible nodes in (3), and, moreover, to achieve faster backtracking by eliminating the need for set $\overline{C}$. In addition, use of bitsets allows for improvements in memory storage efficiency for problem data structures, such as the set of compatible nodes and the adjacency matrix of the graph.

Of particular significance in the context of the present work is the operation of indexing the first significant bit in a bitset, also known as the forward bit scanning. One of the techniques for this purpose relies on use of the De Bruijn sequence with a perfect hash table [8]. The value to be looked up in the hash table is given by H_R below:

$$H_R := (x \wedge -x)\, D \gg (R - \log_2 R), \tag{7}$$

where x is the bitset for which the first significant bit has to be indexed, D is an instance of De Bruijn sequence, R is the machine word size, and $\gg$ stands for the binary *shift right* operator. H_R is effective for bitsets of maximum size equal to R. For larger bitsets, special containers need to be devised. The hash table required to look up the value of H_R is created based on the particular De Bruijn sequence used in (7).

Note that in (7) multiplication is performed modulo R and only the last $\log_2 R$ bits of the result will be retained. More details on forward bit scanning and the specification of the De Bruijn sequence used in (7) can be found in [8].

2.2 BitCLQ

Below we present a modification of FINDCLIQUE, which we refer to as BitCLQ, that uses bitset data structures and bit parallelism for keeping track of the nodes in G that are compatible to the current partial solution S, while simultaneously reducing the computational cost of backtracking.

To this end, we introduce a set Z consisting of k levels, $Z_1, \ldots, Z_k$. Each of these k levels will be used to represent the compatible nodes to the partial solution S at the t-th level of the branch-and-bound tree, where $1 \leq t \leq k$. Every level in Z is further partitioned into k sets, each corresponding to a partite V_b in G:

$$Z_t = \bigcup_{b \in B} Z_{t,b}, \quad t = 1, \ldots, k.$$

The sets $Z_{t,b}$ are represented by bitsets of size $|V_b|$. Let $Z_{t,b,i}$ be the i-th bit in $Z_{t,b}$ corresponding to the i-th node in V_b, such that $Z_{t,b,i} = 1$ if the i-th node in V_b is compatible with all the nodes in the partial solution S at the t-th level of the branch-and-bound tree in BitCLQ:

$$Z_{t,b,i} = \begin{cases} 1, & \text{if } (i,j) \in E \text{ for all } j \in S_t; \\ 0, & \text{otherwise.} \end{cases}$$

Clearly, each level Z_t of Z is an ordered set of combination of bitsets with the total size $|V|$. Further, the adjacency matrix M of graph G is stored in the bitset form, with the convention that the i-th row (column) corresponds to the i-th bit in Z_t, $t = 1, \ldots, k$.

BitCLQ is initialized by setting $t := 0$, $S := \emptyset$, $B = \overline{B}_S := \{1, \ldots, k\}$, and $Q := \emptyset$, where Q is the set of all k-cliques in G. Note that since at the beginning all the nodes in G can be added to S to extend its size, all the bits in Z_1 are significant:

$$Z_{1,b,i} = 1 \text{ for all } b \in \overline{B}(S_t),\ i \in V_b.$$

At level t of the branch-and-bound tree, the partition b_t to branch on is selected as

$$b_t \in \arg\min_b \{|Z_{t,b}| \mid b \in \overline{B}_S\}, \tag{8}$$

where $|Z_{t,b}|$ is defined as the number of significant bits in the bitset $Z_{t,b}$. The forward bit scanning method discussed in Section 2.1 is used to identify node $n_t \in V_{b_t}$ that has not been traversed and thus can be added to the partial solution. As long as such a node exists in V_{b_t}, the search process is restarted with $S := S \cup \{n_t\}$ as the partial solution, and the corresponding bit in Z_{t,b_t} is set to 0.

Utilizing bitsets also facilitates the process of updating the compatible nodes: when n_t is added to partial solution, Z_{t+1} is created by performing a logical AND operation with Z_t and the row $M(n_t)$ of the adjacency matrix corresponding to the node n_t as operands:

$$Z_{t+1} = Z_t \wedge M(n_t). \tag{9}$$

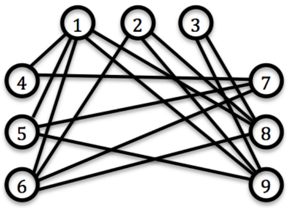
Similarly to FINDCLIQUE, let P denote the number of partitions $Z_{t,b}$ with $|Z_{t,b}| > 0$ at level the t of the branch-and-bound tree. If $|S| + P = k$, the current partial solution is promising, so that a new subproblem is created, and BitCLQ proceeds one level deeper into the branch-and-bound tree, $t := t + 1$. If the partial solution is not promising, the method presented in Section 2.1 is used to select nodes in V_{b_t} that have not been traversed. If such a node is found, the search process is restarted, otherwise backtracking is performed by simply updating $t := t - 1$. Note that due to the special structure of Z, BitCLQ does not need to restore the set of compatible nodes during backtracking, in contrast to the update procedure (6) for the set C_S that is performed in FINDCLIQUE.

2.3 Example

As an illustration, consider the 3-partite graph that is shown along with its adjacency matrix M in Figure 1, where the partite 1 consists of nodes $\{1, 2, 3\}$, partite 2 contains nodes $\{4, 5, 6\}$, and partite 3 contains nodes $\{7, 8, 9\}$. BitCLQ is initialized by setting $S := \emptyset$, $\overline{B}_S := \{1, 2, 3\}$ and $Z_1 := \{111|111|111\}$. Since all the partites are of the same size, i.e. $|Z_{1,b}| = 3$ for all $b \in \overline{B}_S$, the one to branch on is chosen arbitrarily; assume that the first partite $Z_{1,1}$ is chosen for branching. The search process from this point restarts 3 times, each time adding one of the three nodes in $Z_{1,1}$. The first node to add to S is node 1, $Z_{1,1,1}$ is then set to 0, and Z_2 is subsequently created by performing logical AND operation with Z_1 and the corresponding row of the adjacency matrix M as operands:

Algorithm 1 BitCLQ(t)

```
1:  $b_t \in \arg\min_b \{|Z_{t,b}| \mid b \in \overline{B}_S\}$
2:  $i :=$ the first significant bit in $Z_{t,b_t}$
3:  **repeat**
4:      $n_t :=$ the $i$-th node in $b_t$
5:      $Z_{t,b,i} := 0$
6:      $S := S \cup \{n_t\}$
7:      **if** $|S| = k$ **then**
8:          $Q := Q \cup S$
9:          $S := S \setminus \{n_t\}$
10:     **else**
11:         $Z_{t+1,b} := Z_{t,b} \wedge M(n_t)$ for all $b \in \overline{B}_S$
12:         $B_S := B_S \cup \{b_t\}$;  $\overline{B}_S := \overline{B}_S \setminus \{b_t\}$
13:         $P :=$ number of partitions $Z_{t,b}$ with $|Z_{t,b}| > 0$, $b \in \overline{B}_S$
14:         **if** $|S| + P = k$ **then**
15:             BitCLQ($t+1$)
16:             $S := S \setminus \{n_t\}$
17:             $B_S := B_S \setminus \{b_t\}$;  $\overline{B}_S := \overline{B}_S \cup \{b_t\}$
18:         **else**
19:             $S := S \setminus \{n_t\}$
20:             $B_S := B_S \setminus \{b_t\}$;  $\overline{B}_S := \overline{B}_S \cup \{b_t\}$
21:         **end if**
22:     **end if**
23:     $i :=$ the first significant bit in $Z_{t,b_t}$
24: **until** $i \leq |V_{b^t}|$
```



$$M = \begin{array}{c|ccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \hline 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 4 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 5 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 6 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 7 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 8 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 9 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \end{array}$$

Figure 1: A 3-partite graph and its adjacency matrix.

$t := 1$,
$S := \{1\}$,
$Z_2 := Z_1 \wedge M(1) = \{011|111|111\} \wedge \{000|111|011\} = \{000|111|011\}$,
$\overline{B}_S := \{2, 3\}$.

As a result, the set Z_2 of nodes compatible with the partial solution $S = \{1\}$ contains nodes $\{4, 5, 6, 8, 9\}$. Since none of the partites in $\overline{B}_S$ is empty, the partial solution S is promising and a new subproblem is created. The objective in the new subproblem is to find a $|\overline{B}_S|$-clique in Z_2. A node from $Z_{2,3}$ will be added to S (since $|Z_{2,3}| < |Z_{2,2}|$). The first node in $Z_{2,3}$ to add to the partial solution is node 8. The bit corresponding to node 8 is set $Z_{2,3,2} := 0$, and we have

$t := 2$,
$S := \{1, 8\}$,
$Z_3 := Z_2 \wedge M(8) = \{000|111|001\} \wedge \{111|001|000\} = \{000|001|000\}$,
$\overline{B}_S := \{2\}$.

Again, the partites in $\overline{B}_S$ contain at least 1 node (node 6) in Z_3. So the partial solution is promising, and a new subproblem is created. In the next step, node 5 is added to S:

$t := 3$,
$S := \{1, 8, 6\}$.

At this point, since $|S| = k = 3$, i.e., a k-clique is found. To continue the search for other k-cliques, the last node in S is removed. BitCLQ searches $Z_{3,2}$ for another node that can be added to S. Since such a node does not exist, the algorithm backtracks: $t := 2$, node 8 is removed from S, and BitCLQ restarts with $S = \{1, 9\}$ as the partial solution.

Table 1: Average computational time (in seconds) to find all the k-cliques (#CLQ) contained in randomly generated k-partite graphs.

k	m	$\|V\|$	p	#CLQ	FINDCLIQUE	BitCLQ
3	100	300	0.1	1004	0.005	0.002
4	100	400	0.15	1124	0.008	0.002
5	100	500	0.2	1047	0.015	0.003
6	100	600	0.25	939	0.031	0.006
7	50	350	0.35	192	0.009	0.004
8	50	400	0.4	299	0.021	0.007
9	50	450	0.45	683	0.055	0.021
10	50	500	0.5	2672	0.176	0.071

3 Numerical Results

In order to illustrate the performance of the proposed method, the k-clique enumeration problem for k-partite graphs has been solved by BitCLQ and FINDCLIQUE for randomly generated graph instances of several types. Both algorithms were implemented in C++ and ran on a 64-bit Windows machine with 3GHz dual-core processor and 4GB of RAM. It is worth noting that the original implementation of FINDCLIQUE algorithm by Grunert et al [3] relies on the use of `vectors` and `links` data types from the C++ standard template library (STL). In our experiments, we observed that by replacing the original data structure of vectors of lists with arrays, up to 300% improvement in FINDCLIQUE running time is achieved on the data sets used in our case study. The numerical results reported for the FINDCLIQUE algorithm are obtained using this "improved" implementation.

Our numerical experiments involve randomly generated instances of k-partite graphs of two types. The first set of instances consists of two groups: small-size instances and large-size instances. In the small-size instances, k-partite graphs are randomly generated with the number of partites in the range $k \in [3, 10]$. For each value of k, the reported running times and the number of k-cliques in the graph are averaged over 10 instances. Table 1 shows the summary of the experimental results for this first group. The columns of the table show the number k of partites in the k-partite graph, the number m of nodes in each partite of the graph, the total number $|V|$ of nodes in the graph, the graph's density p, and the total number of k-cliques in the graph (#CLQ). The density parameter p is used for generation of the graphs, and is equal to the probability of an edge connecting two nodes from different partites: $\Pr\{(v_i, v_j) \in E\} = p$.

The second group include instances of larger size with the values of $k \in \{25, 50, 75, 100\}$. For each value of k in this group, 10 random instances of the k-partite graph have been generated and solved by FINDCLIQUE and BitCLQ. Table 2 summarizes the results of the experiments for this group. Since the graphs used in this set of experiments are rather large and the list of all k-cliques contained in them may not be found in a reasonable time,

Table 2: Average number of k-cliques found in randomly generated instances of k-partite graphs after 200 seconds.

k	m	$\|V\|$	p	time	FINDCLIQUE	BitCLQ
25	40	1000	0.8	200	13,556,733	23,516,581
50	30	1500	0.9	200	800,369	1,032,111
75	30	2250	0.95	200	557,042,389	735,722,241
100	30	3000	0.95	200	348,416	365,799

Table 3: Average computational time (in seconds) needed to find the first n-clique in an n-partite graph corresponding to a randomized instance of the Multidimensional Assignment Problem with d dimensions and n elements per dimension.

n	d	m	$\|V\|$	p	BitCLQ	FINDCLIQUE
10	3	10	100	0.74	0.00	0.00
20	3	12	240	0.86	0.00	0.00
30	3	13	390	0.91	0.02	0.00
40	3	13	520	0.93	0.76	1.38
50	3	14	700	0.94	0.42	0.42
60	3	14	840	0.95	55.28	86.87
70	3	14	980	0.96	251.78	395.34
10	4	22	220	0.65	0.00	0.00
20	4	28	480	0.82	0.08	0.20
30	4	31	930	0.87	8.18	22.41
10	5	48	480	0.59	0.00	0.01
20	5	68	1360	0.77	13.29	28.23

the solution process has been terminated after 200 seconds and the number of k-cliques found by each method was recorded. BitCLQ outperformed FINDCLIQUE in all cases.

The third set of experiments was conducted to compare the performance of BitCLQ with FINDCLIQUE on randomly generated instances of Multidimensional Assignment Problem (MAP). As was mentioned before, high-quality solutions for randomized MAPs can be obtained as n-cliques in an n-partite subgraph of the underlying graph representing the MAP instance. graphs that are constructed in a special way from the problem's data (in this case, m denotes the number of elements per dimension in a d-dimensional MAP). For MAPs with random iid costs, the resulting n-partite graph can be viewed as randomly generated with a certain density. The corresponding results are reported in Table 3, where n denotes the number of partitions in the graphs, and d is the number of dimensions in the MAP. For each value of n, 10 instances are solved, and the computational time to find the first n-clique is recorded. Algorithms are terminated after finding the first n-clique. The average computational time over 10 runs is reported for each n for each algorithm. In all cases but one, BitCLQ performs better or equally well compared to FINDCLIQUE.

4 Conclusions

In this paper, bitset-based data structures are proposed for the algorithm presented by Grunert et al [3] for the problem of enumerating all k-cliques in a k-partite graph. Utilization of bitsets and the associated bit parallelism enables one to reduce the computational cost of branching and backtracking in the branch-and-bound procedure. Numerical experiments on small- and large-scale randomly generated k-partite graphs show that the proposed approach allows for achieving substantial computational improvements over the original method of [3].

Acknowledgements

The authors would like to acknowledge partial support of AFOSR grant FA9550-12-1-0142 and NSF grant EPS1101284.

References

[1] Immanuel M. Bomze, Marco Budinich, Panos M. Pardalos, and Marcello Pelillo. The maximum clique problem. In *Handbook of Combinatorial Optimization*, pages 1–74. Kluwer Academic Publishers, 1999.

[2] S. Grabowski and K. Fredriksson. Bit-parallel string matching under hamming distance in $O(n\lceil m/w \rceil)O(n\lceil m/w \rceil)$ worst case time. *Information Processing Letters*, 105(5):182–187, 2008.

[3] T. Grunert, S. Irnich, H.J. Zimmermann, M. Schneider, and B. Wulfhorst. Finding all k-cliques in k-partite graphs, an application in textile engineering. *Computers & Operations Research*, 29(1):13–31, 2002.

[4] H. Hyyrö. Bit-parallel approximate string matching algorithms with transposition. *Journal of Discrete Algorithms*, 3(2–4):215–229, 2005.

[5] H. Hyyrö and G. Navarro. Bit-parallel witnesses and their applications to approximate string matching. *Algorithmica*, 41(3):203–231, 2004.

[6] P. Krokhmal, D. Grundel, and P. Pardalos. Asymptotic behavior of the expected optimal value of the multidimensional assignment problem. *Mathematical Programming*, 109(2–3):525–551, 2007.

[7] P. A. Krokhmal and P. M. Pardalos. Limiting optimal values and convergence rates in some combinatorial optimization problems on hypergraph matchings. *Submitted for publication*, 2011.

[8] Charles E. Leiserson, Harald Prokop, and Keith H. Randall. Using de Bruijn sequences to index a 1 in a computer word. *Working paper*, 1998, http://supertech.csail.mit.edu/papers/debruijn.ps.

[9] Q Liu and YPP Chen. High functional coherence in k-partite protein cliques of protein interaction networks. *Bioinformatics and Biomedicine*, 2009.

[10] R. D. Luce. Connectivity and generalized cliques in sociometric group structure. *Psychometrika*, 15(2):169–190, 1950.

[11] M. Mirghorbani, P. Krokhmal, and E. L. Pasiliao. Computational studies of randomized multidimensional assignment problems. In Alexey Sorokin, My T. Thai, and Panos M. Pardalos, editors, *Dynamics of Information Systems*, page in press. Springer.

[12] M Peters. CLICK: Clustering categorical data using k-partite maximal cliques. *IEEE International Conference on Data Engineering*, 2005.

[13] Pablo San Segundo, Diego Rodríguez-Losada, and Agustín Jiminez. An exact bit-parallel algorithm for the maximum clique problem. *Computers & Operations Research*, 38(2):571–581, February 2011.

[14] Pablo San Segundo, Cristbal Tapia, Julio Puente, and Diego Rodrguez-Losada. A new exact bit-parallel algorithm for sat. In *ICTAI (2)'08*, pages 59–65, 2008.