Internet Mathematics, 11:421–448, 2015

ISSN: 1542-7951 print/1944-9488 online
DOI: 10.1080/15427951.2014.986778

Taylor & Francis
Taylor & Francis Group

FAST ALGORITHMS FOR THE MAXIMUM CLIQUE PROBLEM ON MASSIVE GRAPHS WITH APPLICATIONS TO OVERLAPPING COMMUNITY DETECTION

Bharath Pattabiraman[1], Md. Mostofa Ali Patwary[1], Assefaw H. Gebremedhin[2], Wei-keng Liao[1], and Alok Choudhary

[1]*Department of Electrical Engineering and Computer Science Northwestern University, Evanston, Illinois, USA*

[2]*School of Electrical Engineering and Computer Science Washington State University, Pullman, Washington, USA*

Abstract *The maximum clique problem is a well-known NP-hard problem with applications in data mining, network analysis, information retrieval, and many other areas related to the World Wide Web. There exist several algorithms for the problem, with acceptable runtimes for certain classes of graphs, but many of them are infeasible for massive graphs. We present a new exact algorithm that employs novel pruning techniques and is able to find maximum cliques in very large, sparse graphs quickly. Extensive experiments on different kinds of synthetic and real-world graphs show that our new algorithm can be orders of magnitude faster than existing algorithms. We also present a heuristic that runs orders of magnitude faster than the exact algorithm while providing optimal or near-optimal solutions. We illustrate a simple application of the algorithms in developing methods for detection of overlapping communities in networks.*

1. INTRODUCTION

A clique in an undirected graph is a subset of vertices in which every two vertices are adjacent to each other. The *maximum* clique problem seeks to find a clique of the largest possible size in a given graph.

The maximum clique problem, and the related *maximal* clique and clique *enumeration* problems, find applications in a wide variety of domains, many intimately related to the World Wide Web. A few examples include: information retrieval [1], community detection in networks [17, 41, 47], spatial data mining [52], data mining in bioinformatics [36], disease classification based on symptom correlation [6], pattern recognition [44], analysis of financial networks [4], computer vision [22], and coding theory [7]. More examples of application areas can be found in [20, 43].

Address correspondence to Bharath Pattabiraman, Northwestern University, Evanston, IL 60208, USA. E-mail: bpa342@eecs.northwestern.edu; or Assefaw H. Gebremedhin, Washington State University, Pullman, WA 99163, USA. E-mail: assefaw@eecs.wsu.edu.

Color versions of one or more of the figures in the article can be found online at www.tandfonline.com/uinm.

To get a sense for how clique computation arises in the aforementioned contexts, consider a generic data mining or information retrieval problem. A typical objective here is to retrieve data that are considered similar based on some metric. Constructing a graph in which vertices correspond to data items and edges connect similar items, a clique in the graph would then give a cluster of similar data.

The maximum clique problem is NP-hard [18]. Most exact algorithms for solving it employ some form of *branch-and-bound* approach. While branching systematically searches for all candidate solutions, bounding (also known as *pruning*) discards fruitless candidates based on a previously computed bound. The algorithm of Carraghan and Pardalos [8] is an early example of a simple and effective branch-and-bound algorithm for the maximum clique problem. More recently, Östergård [40] introduced an improved algorithm and demonstrated its relative advantages via computational experiments. Upper bounds computed using vertex coloring to enhance the branch-and-bound approach have been used by Tomita and Seki [49], and later, Konc and Janežič [28]. Other examples of branch-and-bound algorithms for the clique problem include the works of Bomze et al., Segundo et al., and Babel and Tinhofer [6, 54, and 3]. A recent work [45] compares various exact algorithms for the maximum clique problem.

In this article, we present a new exact branch-and-bound algorithm for the maximum clique problem that employs several new pruning strategies in addition to those used in [8], [40], [49] and [28], making it suitable for massive graphs. We run our algorithms on a large variety of test graphs and compare its performance with the algorithms by [8, 40, 49, 50, 28, 48]. We find our new exact algorithm to be up to orders of magnitude faster on large, sparse graphs and of comparable runtime on denser graphs. We also present a new heuristic, which runs several orders of magnitude faster than the exact algorithm while providing solutions that are optimal or near-optimal for most cases. The algorithms are presented in detail in Section 3 and the experimental evaluations and comparisons are presented in Section 4.

Both the exact algorithm and the heuristic are well suited for parallelization. We discuss a simple shared-memory parallelization and present performance results showing its promise in Section 5. We also include (in Section 6) an illustration of how the algorithms can be used as parts of a method for detecting overlapping communities in networks. We have made our implementations publicly available.[1]

2. RELATED PREVIOUS ALGORITHMS

Given a simple undirected graph G, the maximum clique can clearly be obtained by enumerating *all* of the cliques present in it and picking the largest of them. A simple-to-implement algorithm that avoids enumerating all cliques and instead works with a significantly reduced partial enumeration was presented in [8]. The reduction in enumeration is achieved via a *pruning* strategy, which reduces the search space tremendously. The algorithm works by performing at each step i, a *depth first search* from vertex v_i, where

[1] `http://cucis.ece.northwestern.edu/projects/MAXCLIQUE/`

the goal is to find the largest clique containing the vertex v_i. At each *depth* of the search, the algorithm compares the number of remaining vertices that could potentially constitute a clique containing vertex v_i against the size of the largest clique encountered thus far. If that number is found to be smaller, the algorithm backtracks (search is pruned).

An algorithm was devised in [40] that incorporated an additional pruning strategy to the one by [8]. The opportunity for the new pruning strategy is created by *reversing* the order in which the search is done by the that algorithm [8]. This allows for an additional pruning with the help of some auxiliary bookkeeping. Experimental results in [40] showed that the algorithm by [40] is faster than the algorithm by [8] on random and DIMACS benchmark graphs [25]. However, the new pruning strategy used in this algorithm is intimately tied to the order in which vertices are processed, introducing an inherent sequentiality into the algorithm.

A number of existing branch-and-bound algorithms for maximum clique also use a vertex-coloring of the graph to obtain an upper bound on the maximum clique. A popular and recent method based on this idea is the MCQ algorithm of [49]. More recently, an improved version of MCQ, known as MaxCliqueDyn [28] (with the variants MCQD and MCQD + CS), that involves the use of tighter, computationally more expensive upper bounds applied on a fraction of the search space was presented [28]. Another improved version of MCQ is BBMC [48], which makes use of bit strings to sort vertices in constant time as well as to compute graph transitions and bounds efficiently.

3. THE NEW ALGORITHMS

We describe in this section new algorithms that overcome the shortcomings mentioned earlier; the new algorithms use additional pruning strategies, maintain simplicity, and avoid a sequential computational order. We begin by first introducing the following notations. We identify the n vertices of the input graph $G = (V, E)$ as $\{v_1, v_2, \ldots, v_n\}$. The set of vertices adjacent to a vertex v_i, the set of its neighbors, is denoted by $N(v_i)$. And the degree of the vertex v_i, the cardinality of $N(v_i)$, is denoted by $d(v_i)$. In our algorithm, the degree is computed once for each vertex at the beginning.

3.1. The Exact Algorithm

Recall that the maximum clique in a graph can be found by computing the largest clique containing each vertex and picking the largest among these. A key element of our exact algorithm is that during the search for the largest clique containing a given vertex, vertices that cannot form cliques larger than the current maximum clique are *pruned*, in a hierarchical fashion. The method is outlined in detail in Algorithm 1. Throughout, the variable max stores the size of the maximum clique found thus far. Initially, it is set to be equal to the lower bound lb provided as an input parameter. It gives the maximum clique size when the algorithm terminates.

To obtain the largest clique containing a vertex v_i, it is sufficient to consider only the neighbors of v_i. The main routine MaxClique thus generates for each vertex $v_i \in V$ a set $U \subseteq N(v_i)$ (neighbors of v_i that survive pruning) and calls the subroutine Clique on

Algorithm 1 Algorithm for finding the maximum clique of a given graph. **Input**: Graph $G = (V, E)$, lower bound on clique lb (default, 0). **Output**: Size of maximum clique.

```
1: procedure MaxClique(G = (V, E), lb)
2:     max ← lb
3:     for i : 1 to n do
4:         if d(v_i) ≥ max then ( Pruning 1)
5:             U ← ∅
6:             for each v_j ∈ N(v_i) do
7:                 if j > i then ( Pruning 2)
8:                     if d(v_j) ≥ max then
   ( Pruning 3)
9:                         U ← U ∪ {v_j}
10:            Clique(G, U, 1)
```

Subroutine

```
1: procedure Clique(G = (V, E), U, size)
2:     if U = ∅ then
3:         if size > max then
4:             max ← size
5:         return
6:     while |U| > 0 do
7:         if size + |U| ≤ max then (
   Pruning 4
8:             return
9:         Select any vertex u from U
10:        U ← U \ {u}
11:        N'(u) := {w|w ∈ N(u) ∧ d(w) ≥
   max}                                ( Pruning 5
12:        Clique(G, U ∩ N'(u), size + 1)
```

U. The subroutine Clique goes through every relevant clique containing v_i in a recursive fashion and returns the largest. We use $size$ to maintain the size of the clique found at any point through the recursion. Because we start with a clique of just one vertex, the value of $size$ is set to one initially, when Clique is called (Line 10, MaxClique).

Our algorithm consists of several pruning steps. Pruning 1 (Line 4, MaxClique) filters vertices having strictly fewer neighbors than the size of the maximum clique already computed. These vertices can be ignored, since even if a clique were to be found, its size would not be larger than max. While forming the neighbor list U for a vertex v_i, we include only those of v_i's neighbors for which the largest clique containing them has not been found (Pruning 2; Line 7, MaxClique), to avoid recomputing previously found cliques. Pruning 3 (Line 8, MaxClique) excludes vertices $v_j \in N(v_i)$ that have degree less than the current value of max, since any such vertex could not form a clique of size larger than max. Pruning 4 (Line 7, Clique) checks for the case in which, even if all vertices of U were added to get a clique, its size would not exceed that of the largest clique encountered so far in the search, max. Pruning 5 (Line 11, Clique) reduces the number of comparisons needed to generate the intersection set in Line 12. Note that the routine Clique is similar to the algorithm in [8]; Pruning 5 accounts for the main difference. Also, Pruning 4 is used in most existing algorithms, whereas Prunings 1, 2, 3, and 5 are not.

3.2. The Heuristic

The exact algorithm examines all relevant cliques containing every vertex. Our heuristic, shown in Algorithm 2, considers only *one* neighbor with *maximum degree* at each step instead of recursively considering *all* neighbors from the set U, and, thus, is much faster. The vertex with maximum degree is chosen for this intuitive reason: in a relatively fairly

Algorithm 2 Heuristic for finding the maximum clique in a graph. **Input**: Graph $G = (V, E)$. **Output**: Approximate size of maximum clique.

```
1: procedure MaxCliqueHeu (G = (V, E))
2:     for i : 1 to n do
3:         if d(v_i) ≥ max then
4:             U ← ∅
5:             for each v_j ∈ N(v_i) do
6:                 if d(v_j) ≥ max then
7:                     U ← U ∪ {v_j}
8:             CliqueHeu(G, U, 1)
```

Subroutine

```
1: procedure CliqueHeu(G = (V, E), U, size)
2:     if U = ∅ then
3:         if size > max then
4:             max ← size
5:         return
6:     Select a vertex u ∈ U of maximum degree in G
7:     U ← U \ {u}
8:     N'(u) := {w|w ∈ N(u) ∧ d(w) ≥ max}
9:     CliqueHeu(G, U ∩ N'(u), size + 1)
```

connected subgraph, a vertex with the maximum degree is more likely to be a member of the largest clique containing that vertex than to any other.

3.3. Complexity

The exact algorithm, Algorithm 1, examines for every vertex v_i all candidate cliques containing the vertex v_i in its search for the largest clique. Its time complexity is exponential in the worst case. The heuristic, Algorithm 2, loops over the n vertices, each time possibly calling the subroutine CliqueHeu, which effectively is a loop that runs until the set U is empty. Clearly, $|U|$ is bounded by the max degree Δ in the graph. The subroutine also includes the computation of a neighbor list, whose runtime is bounded by $O(\Delta)$. Thus, the time complexity of the heuristic is bounded by $O(n \cdot \Delta^2)$.

3.4. Graph Data Structure

Our implementation uses a simple adjacency list representation to store the graph. This is done by maintaining two arrays. Given a graph $G = (V, E)$, with its vertices numbered from 0 to $|V| - 1$, the edge array maintains the concatenated list of sorted neighbors of each vertex. The size of this array is $2|E|$. The vertex array is $|V|$ elements long, one for each vertex in sequential order, and each element points (stores the array index) to the starting point of its neighbor list in the edge array. Figure 1 shows our representation of the data structure used for a sample graph.

4. EXPERIMENTAL EVALUATION

We present in this section results comparing the performance of our algorithm with other existing algorithms. Our experiments were performed on a Linux workstation running

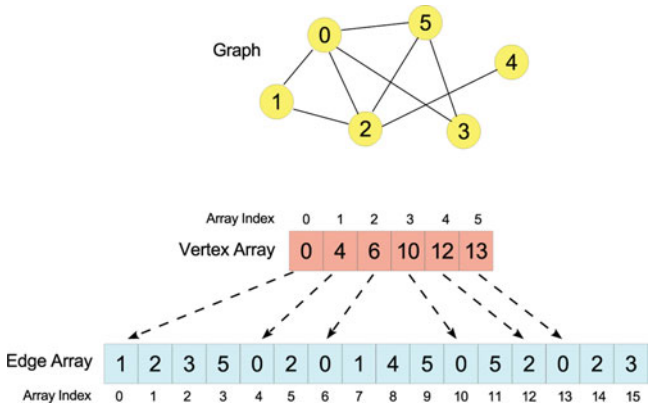

Figure 1 The adjacency-list data structure used in our implementation shown for a sample graph. Each element in the vertex array stores the index to the starting element of its neighbor list in the edge array.

64-bit Red Hat Enterprise Linux Server release 6.2 with a 2GHz Intel Xeon E7540 processor with 32GB of main memory. Our implementation is in C++, compiled using gcc version 4.4.6 with −O3 optimization.

4.1. Test Graphs

Our testbed consists of 91 graphs, grouped into three categories:

1. **Real-world graphs.** Under this category, we consider 10 graphs downloaded from the University of Florida (UF) Sparse Matrix Collection [12], 5 graphs from Pajek data-sets [3], and 10 graphs from the Stanford Large Network Dataset Collection [31]. The graphs originate from various real-world applications. Table I gives a quick overview of the graphs and their origins.
2. **Synthetic Graphs.** In this category we consider 15 graphs generated using the R-MAT algorithm [9]. The graphs are subdivided into three categories, depending on the structures they represent:
   a. **Random graphs** (5 graphs) – Erdős-Rényi random graphs generated using R-MAT with the parameters (0.25, 0.25, 0.25, 0.25). We denoted these with prefix *rmat_er*.
   b. **Skewed Degree, Type 1 graphs** (5 graphs) – graphs generated using R-MAT with the parameters (0.45, 0.15, 0.15, 0.25). These are denoted with prefix *rmat_sd1*.
   c. **Skewed Degree, Type 2 graphs** (5 graphs) – graphs generated using R-MAT with the parameters (0.55, 0.15, 0.15, 0.15). These are denoted with prefix *rmat_sd2*.
3. **DIMACS graphs.** This last category consists of 51 graphs selected from the Second DIMACS Implementation Challenge [25]. Among these, 5 graphs are

considered for discussion in the next few sections, and the results for the rest are reported in Table X in the Appendix.

The DIMACS graphs are an established benchmark for the maximum clique problem, but they are of rather limited size and variation. In contrast, the real-work networks included in category 1 of the testset and the synthetic (RMAT) graphs in category 2 represent a wide spectrum of large graphs posing varying degrees of difficulty for the algorithms. The *rmat_er* graphs have *normal* degree distribution, whereas the *rmat_sd1* and *rmat_sd2* graphs have skewed degree distributions and contain many dense local subgraphs. The *rmat_sd1* and *rmat_sd2* graphs differ primarily in the magnitude of maximum vertex degree they contain; the *rmat_sd2* graphs have much higher maximum degree. Table II lists basic structural information (the number of vertices, number of edges, and the maximum degree) about 45 of the test graphs (25 real-world, 15 synthetic, and 5 DIMACS).

4.2. Algorithms for Comparison

The algorithms we consider for comparison are those by

- Carraghan Pardalos (CP), [8]; we used our own implementation of this algorithm.
- Östergård algorithm [40]; we used the publicly available *cliquer* source code [39].
- Konc and Janežič [28]; we used the code *MaxCliqueDyn* or MCQD.[3] Among the variants available in MCQD, we report results on MCQD+CS (which uses improved coloring and dynamic sorting), because it is the best-performing variant. The *MaxCliqueDyn* code was not capable of handling large input graphs and had to be aborted for many instances. For those that ran successfully, we had to first modify the graph reader to make it able to handle graphs with multiple connected components.
- MCQ [49].
- MCS [50].
- BBMC [48].

For MCQ, MCS, and BBMC, we used the publicly available Java implementation, MCQ1, MCSa1, and BBMC1, respectively.[4] These implementations failed to run because of memory limitations in spite of making available 20 GB of memory for almost all (except two) of the larger data-sets. Hence, timings are reported only for DIMACS graphs.

In addition to comparing with the aforementioned algorithms, for the Pajek and Stanford data-sets we also provide comparison of the timing results of our algorithm with the maximal clique enumeration algorithm [14]. For this, we directly quote numbers published in [14], and the results are listed in Table VIII in the Appendix. It should be kept in mind that maximal clique enumeration in general is a more difficult problem than

[3] Available at `http://www.sicmm.org/~konc/maxclique/`.
[4] By [45]; available at `http://www.dcs.gla.ac.uk/~pat/maxClique/`.

Graph	Description
cond-mat-2003 [38]	A collaboration network of scientists posting preprints on the condensed matter archive[2] in the period between January 1, 1995 and June 30, 2003.
email-Enron [34]	A communication network representing email exchanges.
dictionary28 [3]	Pajek network of words.
Fault_639 [16]	A structural problem discretizing a faulted gas reservoir with tetrahedral Finite Elements and triangular Interface Elements.
audikw_1 [12]	An automotive crankshaft model of TETRA elements.
bone010 [51]	A detailed microfinite element model of bones representing the porous bone microarchitecture.
af_shell [12]	A sheet metal forming simulation network.
as-Skitter [34]	An Internet topology graph from trace routes run daily in 2005.
roadNet-CA [34]	A road network of California. Nodes represent intersections and endpoints and edges represent the roads connecting them.
kkt_power [12]	An Optimal Power Flow (nonlinear optimization) network.
foldoc [23]	A searchable dictionary of terms related to computing.
eatRS [27]	The Edinburgh Associative Thesaurus, a set of word association norms showing the counts of word association as collected from subjects.
hep-th [26]	Citation data from KDD Cup 2003, a knowledge discovery and data mining competition held in conjunction with the Ninth ACM SIGKDD Conference.
patents [21]	Data-set containing information on almost 3 million U.S. patents granted between January 1963 and December 1999, and all citations made to these patents between 1975 and 1999.
days-all [11]	Reuters terror news network obtained from the CRA networks produced by Steve Corman and Kevin Dooley at Arizona State University.
roadNet-TX [34]	Road network of Texas.
amazon0601 [32]	Amazon product copurchasing network from June 1 2003.
email-EuAll [35]	E-mail network from a EU research institution.
web-Google [19]	Web graph released by Google in 2002 as a part of Google Programming Contest.
soc-wiki-Vote [33]	Wikipedia who-votes-on-whom network.
soc-slashdot0902 [34]	Slashdot social network from November 2008.
cit-Patents [21]	Citation network among US Patents.
soc-Epinions1 [46]	Who-trusts-whom network of Epinions.com.
soc-wiki-Talk [33]	Wikipedia talk (communication) network.
web-berkstan [34]	Web graph of Berkeley and Stanford.

Table I Overview of real-world graphs in the testbed (UF, Pajek, and Stanford datasets) and their origins.

[2]www.arixiv.org

G	$\|V\|$	$\|E\|$	Δ	G	$\|V\|$	$\|E\|$	Δ
cond-mat-2003	31,163	120,029	202	*rmat_sd1_1*	131,072	1,046,384	407
email-Enron	36,692	183,831	1,383	*rmat_sd1_2*	262,144	2,093,552	558
dictionary28	52,652	89,038	38	*rmat_sd1_3*	524,288	4,190,376	618
Fault_639	638,802	13,987,881	317	*rmat_sd1_4*	1,048,576	8,382,821	802
audikw_1	943,695	38,354,076	344	*rmat_sd1_5*	2,097,152	16,767,728	1,069
bone010	986,703	35,339,811	80	*rmat_sd2_1*	131,072	1,032,634	2,980
af_shell10	1,508,065	25,582,130	34	*rmat_sd2_2*	262,144	2,067,860	4,493
as-Skitter	1,696,415	11,095,298	35,455	*rmat_sd2_3*	524,288	4,153,043	6,342
roadNet-CA	1,971,281	2,766,607	12	*rmat_sd2_4*	1,048,576	8,318,004	9,453
kkt_power	2,063,494	6,482,320	95	*rmat_sd2_5*	2,097,152	16,645,183	14,066
foldoc	13,356	91,470	728	*rmat_er_1*	131,072	1,048,515	82
eatRS	23,219	304,938	1090	*rmat_er_2*	262,144	2,097,104	98
hep-th	27,240	341,923	2411	*rmat_er_3*	524,288	4,194,254	94
patents	240,547	560,943	212	*rmat_er_4*	1,048,576	8,388,540	97
days-all	13,308	148,035	2265	*rmat_er_5*	2,097,152	16,777,139	102
				hamming6-4	64	704	22
roadNet-TX	1,393,383	1,921,660	12	*johnson8-4-4*	70	1,855	53
amazon0601	403,394	2,247,318	2752	*keller4*	171	9,435	124
email-EuAll	265,214	364,481	7636	*c-fat200-5*	200	8,473	86
web-Google	916,428	4,322,051	6332	*brock200_2*	200	9,876	114
soc-wiki-Vote	8,297	100,762	1065				
soc-slashdot0902	82,168	504,230	2252				
cit-Patents	3,774,768	16,518,947	793				
soc-Epinions1	75,888	405,740	3044				
soc-wiki-Talk	2,394,385	4,659,565	100029				
web-berkstan	685,230	6,649,470	84230				

Table II Structural properties—the number of vertices $|V|$; the umber of edges $|E|$; and the maximum degree Δ—of the graphs G in the testbed. The graphs on the left side are graphs from the UF collection, Pajek data-sets, and Stanford Large Dataset collection. On the right are the RMAT graphs; and the DIMACS Challenge graphs.

maximum clique finding, and that these experiments have been performed on different test environments. Therefore, the runtime results should be understood in a qualitative sense.

4.3. Results

Table III shows the size of the maximum clique (ω) and the runtimes of our exact algorithm, Algorithm 1, and the algorithms of CP, *cliquer*, and MCQD+CS for all the graphs in the testbed except for the Pajek and Stanford test graphs; results on the Pajek and Stanford graphs are presented in Table IV. The last two columns in Table III show the results of our heuristic (Algorithm 2)—the size of the maximum clique returned (ω_{A_2}) and its runtime (τ_{A_2}). The columns labeled $P1$, $P2$, $P3$, and $P5$ list the number of vertices/branches pruned in the respective pruning steps of Algorithm 1. Data on Pruning 4 is omitted because that pruning is used by all of the algorithms compared in the table. The displayed pruning numbers have been rounded (K stands for 10^3, M for 10^6, and B for 10^9); the exact numbers can be found in Table IX in the Appendix.

In Table III, the fastest runtime for each instance is indicated with boldface. An asterisk (*) indicates that an algorithm did not terminate within 25,000 seconds for a particular instance. A hyphen (-) indicates that the publicly available implementation (the

Graph	ω	τ_{CP}	$\tau_{cliquer}$	τ_{MCQD_+CS}	τ_{A_1}	$P1$	$P2$	$P3$	$P5$	ω_{A_2}	τ_{A_2}
cond-mat-2003	25	4.87	11.17	2.41	**0.011**	29K	48K	6,527	17K	25	<0.01
email-Enron	20	7.01	15.08	3.70	**0.998**	32K	155K	4,060	8M	18	0.26
dictionary28	26	7.70	32.74	7.69	<**0.01**	52K	4,353	2,114	107	26	<0.01
Fault_639	18	14571.20	4437.14	—	**20.03**	36	13M	126	1,116	18	5.80
audikw_1	36	*	9282.49	—	**190.17**	4,101	38M	59K	721K	36	58.38
bone010	24	*	10002.67	—	**393.11**	37K	34M	361K	44M	24	24.39
af_shell10	15	*	21669.96	—	**50.99**	19	25M	75	2,105	15	10.67
as-Skitter	67	24385.73	*	—	**3838.36**	1M	6M	981K	737M	66	27.08
roadNet-CA	4	*	*	—	**0.44**	1M	1M	370K	4,302	4	0.08
kkt_power	11	*	*	—	**2.26**	1M	4M	401K	2M	11	1.83
rmat_er_1	3	256.37	215.18	49.79	**0.38**	780	1M	915	8,722	3	0.12
rmat_er_2	3	1016.70	865.18	—	**0.78**	2,019	2M	2,351	23K	3	0.24
rmat_er_3	3	4117.35	3456.39	—	**1.87**	4,349	4M	4,960	50K	3	0.49
rmat_er_4	3	16419.80	13894.52	—	**4.16**	9,032	8M	10K	106K	3	1.44
rmat_er_5	3	*	*	—	**9.87**	18K	16M	20K	212K	3	2.57
rmat_sd1_1	6	225.93	214.99	50.08	**1.39**	39K	1M	23K	542K	6	0.45
rmat_sd1_2	6	912.44	858.80	—	**3.79**	90K	2M	56K	1M	6	0.98
rmat_sd1_3	6	3676.14	3446.02	—	**8.17**	176K	4M	106K	2M	6	1.78
rmat_sd1_4	6	14650.40	13923.93	—	**25.61**	369K	8M	214K	5M	6	4.05
rmat_sd1_5	6	*	*	—	**46.89**	777K	16M	455K	12M	6	9.39
rmat_sd2_1	26	427.41	213.23	**48.17**	242.20	110K	853K	88K	614M	26	32.83
rmat_sd2_2	35	4663.62	**851.84**	—	3936.55	232K	1M	195K	1B	35	95.89
rmat_sd2_3	39	13626.23	**3411.14**	—	10647.84	470K	3M	405K	1B	37	245.51
rmat_sd2_4	43	*	**13709.52**	—	*	*	*	*	*	42	700.05
rmat_sd2_5	N	*	*	—	*	*	*	*	*	51	1983.21
hamming6-4	4	<**0.01**	<**0.01**	<**0.01**	<**0.01**	0	704	0	0	4	<0.01
johnson8-4-4	14	0.19	<**0.01**	<**0.01**	0.23	0	1,855	0	0	14	<0.01
keller4	11	22.19	0.15	**0.02**	23.35	0	9,435	0	0	11	<0.01
c-fat200-5	58	0.60	0.33	**0.01**	0.93	0	8,473	0	0	58	0.04
brock200_2	12	0.98	0.02	<**0.01**	1.10	0	9,876	0	0	10	<0.01

Table III Comparison of runtimes (in seconds) of algorithms *CP* [8], *cliquer* [40], MCQD+CS [28] and and our new exact algorithm (A_1) for the graphs in the testbed. An asterisk (*) indicates that the algorithm did not terminate within 25,000 seconds for a particular instance. A hyphen (-) indicates that the publicly available implementation we used could not handle this instance due to its large size. Columns $P1$, $P2$, $P3$ and $P5$ list the number of vertices/branches pruned in steps Pruning 1, 2, 3, and 5 of our exact algorithm (K stands for the quantity 10^3, M for 10^6, and B for 10^9). The column ω (second column) lists the maximum clique size in each graph, the column ω_{A_2} lists the clique size returned by our heuristic, and the column τ_{A_2} lists the heuristic's runtime.

MaxCliqueDyn code) had to be aborted because the input graph was too large for the implementation to handle. For the graph *rmat_sd2_5*, none of the algorithms computed the maximum clique size in a reasonable time; the entry there is marked with N, standing for "Not Known."

We discuss in what follows our observations from this table for the exact algorithm and the heuristic.

4.3.1. Exact algorithms. As expected, our exact algorithm gave the same size of maximum clique as the other three algorithms for all test cases. In terms of runtime, its relative performance, compared to the other three, varied in accordance with the advantages afforded by the various pruning steps.

G	ω	$\tau_{cliquer}$	$\tau_{MCQD_{+}CS}$	τ_{A1}	ω_{A2}	τ_{A2}
foldoc	9	2.23	0.58	0.05	9	<0.01
eatRS	9	7.53	1.86	1.81	9	1.80
hep-th	23	9.43	2.43	4.48	23	0.06
patents	6	2829.48	239.66	0.13	6	0.03
days-all	28	2.47	0.59	75.37	21	0.05
roadNet-TX	4	*	*	0.26	4	0.04
amazon0601	11	3190.11	717.99	0.20	11	0.30
email-EuAll	16	943.74	270.67	0.92	14	0.06
web-Google	44	10958.68	*	0.35	44	0.6
soc-wiki-Vote	17	0.89	0.24	4.31	14	0.02
soc-slashdot0902	27	88.38	23.63	25.54	22	0.06
cit-Patents	11	2.47	*	19.99	10	4.15
soc-Epinions1	23	73.85	19.95	15.01	19	0.06
soc-wiki-Talk	26	*	*	6885.13	18	0.45
web-berkstan	201	6416.61	*	44.70	201	32.03

Table IV Comparison of runtimes of algorithms: [40] ($\tau_{cliquer}$), [28] ($\tau_{MCQD+CS}$), with that of our new exact algorithm (τ_{A1}) for selected medium and large Pajek and Stanford data sets. An asterisk (*) indicates that the algorithm did not terminate within 15,000 seconds for that instance. The column ω lists the maximum clique size in each graph, the column ω_{A_2} lists the clique size returned by our heuristic and the column τ_{A_2} lists the heuristic's runtime.

Vertices that are discarded by Pruning 1 are skipped in the main loop of the algorithm, and the largest cliques containing them are not computed. Pruning 2 avoids recomputing previously computed cliques in the neighborhood of a vertex. In the absence of Pruning 1, the number of vertices pruned by Pruning 2 would be bounded by the number of edges in the graph (note that this is more than the total number of vertices in the graph). Whereas Pruning 3 reduces the size of the input set on which the maximum clique is to be computed, Pruning 5 brings down the time taken to generate the intersection set in Line 12 of the subroutine. Pruning 4 corresponds to backtracking. Unlike Pruning steps 1, 2, 3, and 5, Pruning 4 is used by all three of the other algorithms in our comparison. The primary strength of our algorithm is its ability to take advantage of pruning in multiple steps in a hierarchical fashion, allowing for opportunities for one or more of the steps to be triggered and positively impact the performance.

In Figure 2 we show the number of vertices discarded by all the pruning steps of the exact algorithm normalized by the total number of edges in a graph for the real-world graphs in Table III. We cut few bars reaching 140% because their correspnding values are much higher. In the Appendix, we provide a complete tabulation of the raw numbers for the pruned vertices in all the steps for all the graphs in the testbed. It can be seen for these graphs, pruning steps 2 and 5 in particular discard a large percentage of vertices, potentially resulting in large runtime savings. The general behavior of the pruning steps Pruning 1, 2, 3, and 5 for the synthetic graphs *rmat_er* and *rmat_sd1* was observed to be somewhat similar to that depicted in Figure 2 for the real-world graphs. In contrast, for the DIMACS

graphs, the number of vertices pruned in steps Pruning 1, 3, and 5 were observed to be zero; the numbers in the step Pruning 2 were nonzero but relatively modest.

As a result of the differences seen in the effects of the pruning steps, as discussed below, the runtime performance of our algorithm (seen in Table III) compared to the other three algorithms varied in accordance with the differences in the structures represented by the different categories of graphs in the testbed.

Real-world graphs. For most of the graphs in this category (Table III), it can be seen that our algorithm runs several orders of magnitude faster than the other three, mainly due to the large amount of pruning the algorithm attained. These numbers also illustrate the great benefit of hierarchical pruning. For the graphs *Fault_639*, *audikw_1*, and *af_shell10*, Prunings 1, 3, and 5 have only minimal impact, whereas Pruning 2 makes a big difference, resulting in impressive runtimes. The number of vertices pruned in steps Pruning 1 and 3 varied among the graph *within* the category, ranging from 0.001% for *af_shell* to a staggering 97% for *as-Skitter* for the step Pruning 1. For the graphs in Table 4.3 as well, one can observe that our algorithm performs much better than the others.

Synthetic graphs. For the synthetic graph types *rmat_er* and *rmat_sd1*, our algorithm clearly outperforms the other three by a few orders of magnitude in all cases. This is also primarily due to the high number of vertices discarded by the new pruning steps. In particular, for *rmat_sd1* graphs, between 30% to 37% of the vertices are pruned just in the step Pruning 1. For the *rmat_sd2* graphs, which have relatively larger maximum clique and higher maximum degree than the *rmat_sd1* graphs, our algorithm is observed to be faster than CP but slower than *cliquer*.

DIMACS graphs. The runtime of our exact algorithm for the DIMACS graphs is in most cases comparable to that of CP and higher than that of *cliquer* and MCQD+CS. For these graphs, only Pruning 2 was found to be effective, and thus the performance results agree with one's expectation. The timings on a much larger collection of DIMACS graphs are presented in Table X in the Appendix.

It is to be noted that the DIMACS graphs are intended to serve as challenging test cases for the maximum clique problem, and graphs with such high edge densities and low vertex counts are rare in practice. Most of these have between 20 to 1024 vertices, with an

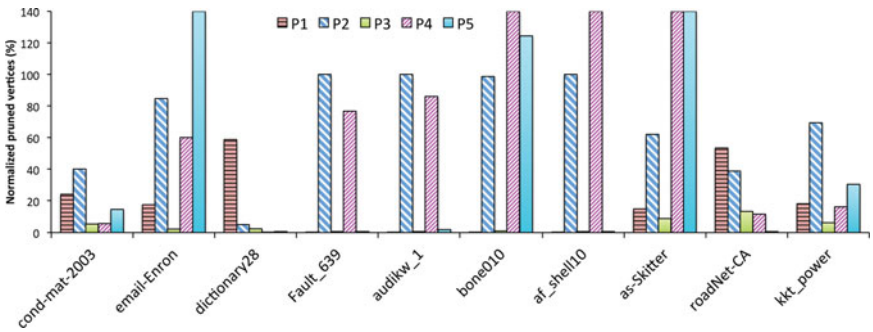

Figure 2 Number of "pruned" vertices in the various pruning steps normalized by the number of edges in the graph (in percents) for the UF collection graphs (we cut few bars reachining 140% as their correspnding values are much higher).

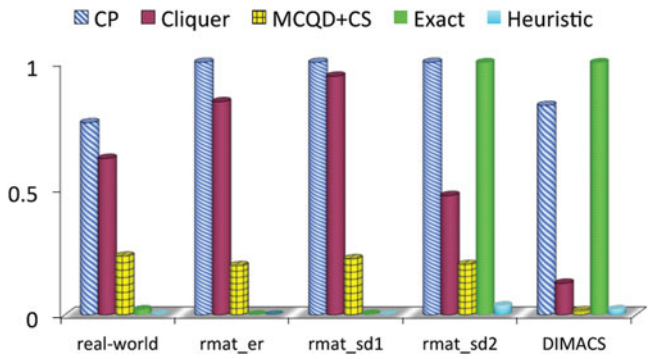

Figure 3 Runtime (normalized, mean) comparison between various algorithms. For each category of graph, first, all runtimes for each graph were normalized by the runtime of the slowest algorithm for that graph, and then the mean was calculated for each algorithm. Graphs were considered only if the runtimes for at least three algorithms were less than the 25,000 seconds limit set.

average edge density of roughly 0.6, whereas, most real-world graphs are often very large and sparse, as exemplified by the real-world graphs in our testbed. Other good examples of instances with similar nature include Internet topology graphs [15], the web graph [29], and social network graphs [13].

4.3.2. The heuristic. It can be seen that our heuristic runs several orders of magnitude faster than our exact algorithm, while delivering either optimal or very close to optimal solution. It gave the optimal solution on 25 out of the 30 test cases. On the remaining 5 cases where it was suboptimal, its accuracy ranges from 83% to 99% (on average 93%).

Additionally, we run the heuristic by choosing a vertex randomly in Line 6 of Algorithm 2 instead of the one with the maximum degree. We observe that on average, the solution is optimal only for less than 40% of the test cases compared to 83% when selecting the maximum degree vertex.

4.3.3. Further analysis. Figure 3 provides an aggregated visual summary of the runtime trends of the various algorithms across the five categories of graphs in the testbed.

To give a sense of runtime growth rates, we provide in Figure 4 plots of the runtime of the new exact algorithm and the heuristic for the synthetic and real-world graphs in the testbed. In addition to the curves corresponding to the runtimes of the *exact* algorithm and the *heuristic*, the figures also include a curve corresponding to the number of *edges* in the graph divided by the clock frequency of the computing platform used in the experiment. This curve is added to facilitate comparison between the growth rate of the runtime of the algorithms with that of a linear-time (in the size of the graph) growth rate. It can be seen that the runtime of the heuristic by and large grows somewhat linearly with the size of a graph. The exact algorithm's runtime, which is orders of magnitude larger than the heuristic, exhibited a similar growth behavior for these test cases even though its worst-case complexity suggests exponential growth in the general case.

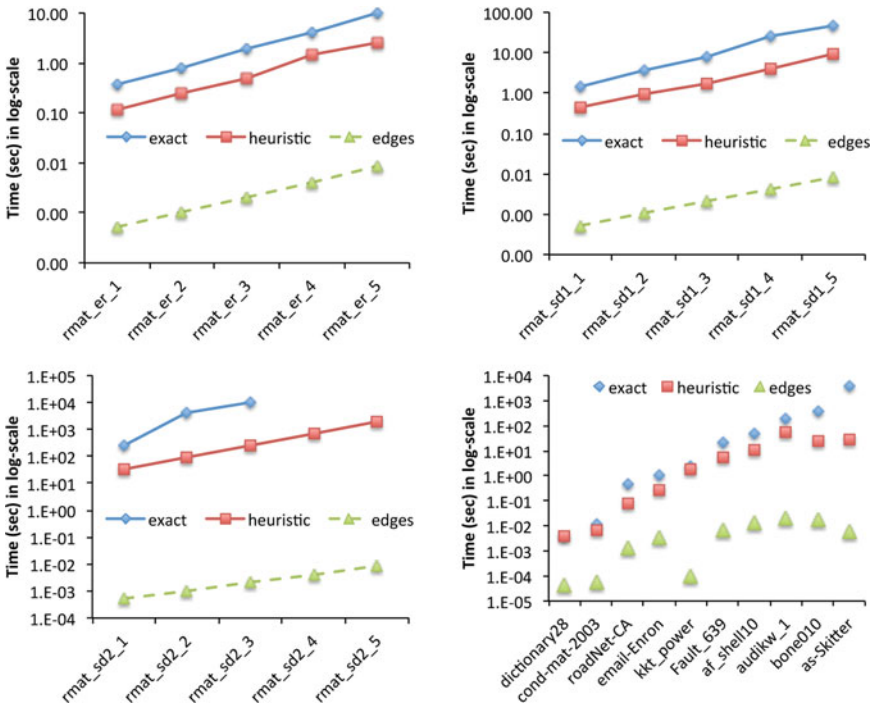

Figure 4 Runtime plots of the new exact and heuristic algorithms. The third curve, labeled *edges*, shows the quantity number of edges in the graph divided by the clock frequency of the computing platform used in the experiment.

5. PARALLELIZATION

We demonstrate in this section how our exact algorithm can be parallelized and show performance results on a shared-memory platform. The heuristic can be parallelized following a similar procedure.

As explained in Section 3, the ith iteration of the *for* loop in the exact algorithm (Algorithm 1) computes the size of the largest clique that contains the vertex v_i. Since our algorithm does not impose any specific order in which vertices have to be processed, these iterations can, in principle, be performed concurrently. During such a concurrent computation, however, different processes might discover maximum cliques of different sizes—and for the pruning steps to be most effective, the current globally largest maximum clique size needs to be communicated to all processes as soon as it is discovered. In a shared-memory programming model, the global maximum clique size can be stored as a shared variable accessible to all the processing units, and its value can be updated by the relevant processor at any given time. In a distributed-memory setting, more care needs to be exercised to keep the communication cost low.

We implemented a shared-memory parallelization based on the procedure described using OpenMP. Since the global value of the maximum clique is shared by all processing units, we embed the step that updates the maximum clique, i.e., Line 4 of Algorithm 1, into a *critical* section (an OpenMP feature that enforces a lock, thus allowing only one thread

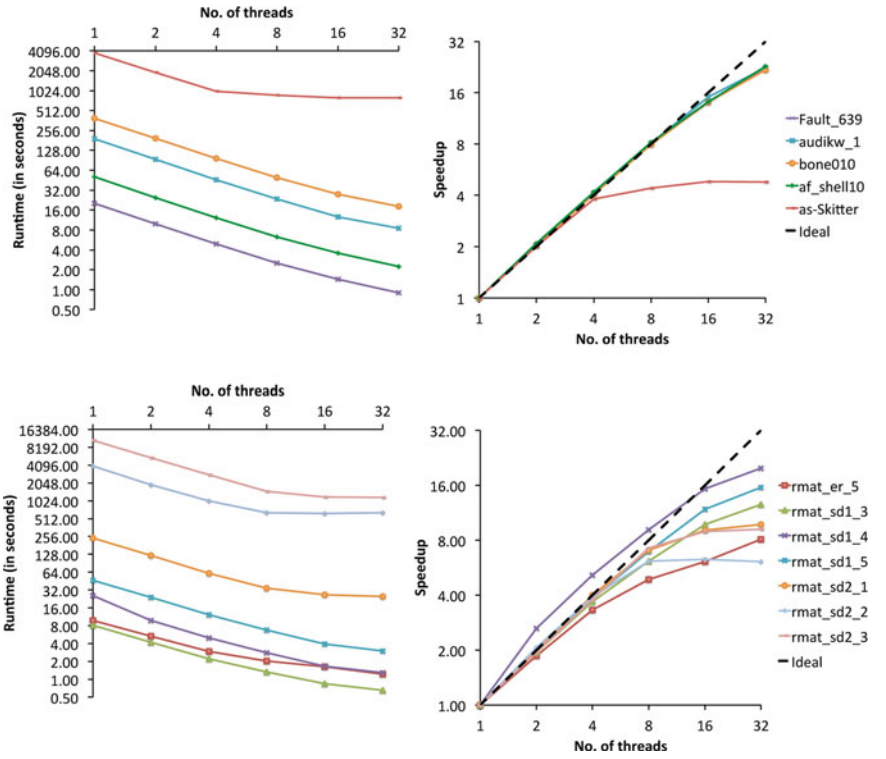

Figure 5 Performance (timing and speedup plots) of shared-memory parallelization on graphs in the test bed. The top set of figures show performance on the real-world graphs, whereas the bottom set show results for the RMAT graphs. Graphs whose sequential runtime is less than 5 seconds were omitted.

at a time to make the update). Further, we use dynamic load scheduling, because different vertices might return different sizes of maximum clique, resulting in different work loads.

We performed experiments on the same graphs listed in the testbed described in Section 4. For these, we used a Linux workstation with six 2.00 GHz Intel Xeon E7540 processors. Each processor has six cores, each with 32 KB of L1 and 256 KB of L2 cache, and each processor shares an 18 MB L3 cache.

Figure 5 shows the timings and speedups we obtained for the real-world and RMAT graphs separately. We omitted graphs whose sequential runtimes were less than 5 seconds because they were too low to measure to do meaningful assessment of parallelization performance. For most of the real-world graphs, one can see from the figure that we obtained near-linear scaling of runtimes and speedups when up to 16 threads are used. The only exception is the graph *as*-Skitter. The relatively poorer scaling there is likely because the instance has a relatively large maximum clique, and hence, the core (thread) that computes it spends a relatively large amount of time computing it while other cores (threads) that have completed their processing of the remaining vertices remain idle. For the

other instances of the real-world graphs, the maximum speedup we obtained while using 32 cores/threads is around 22×.

For the RMAT graphs, it can be seen that the scaling of the runtime and speedups vary with the structures (RMAT parameters) of the instances. We observed superlinear speedups for a couple of the instances, which happens as a result of some unfruitful searches in the branch-and-bound procedure being discovered early as a result of parallel processing. This phenomenon is better exploited and more fully explored in other works in the literature, such as [37]. For the other instances, the algorithm scales fairly well up to eight threads, and begins to degrade thereafter. The speedups we obtained range between 6× to 20× when 32 threads are used.

6. CLIQUE ALGORITHMS AND COMMUNITY DETECTION

In this section we demonstrate how clique-finding algorithms can be used for detecting overlapping communities in networks.

Background. Most community detection algorithms are designed to identify mutually independent communities in a given network and, therefore, are not suitable for detecting overlapping communities. Yet, in many real-world networks, it is natural to find vertices (or members) that belong to more than one group (or community) at the same time.

The Clique Percolation Method (CPM; [41]) is one effective approach for detecting overlapping communities in a network. The basic premise in CPM is that a typical community is likely to be made up of several cliques that share many of their vertices. We recall a few notions defined in [41] to make this more precise. A clique of size k is called a *k-clique*, and two k-cliques are called *adjacent* if they share $k-1$ nodes. A *k-clique community* is a union of all k-cliques that can be reached from each other through a series of adjacent k-cliques. With these notions in hand, a method was devised in [41] to extract such k-clique communities of a network. Note that, by definition, k-clique communities allow for overlaps, i.e., common vertices can be shared by the communities.

The CPM algorithm is illustrated in Figure 6. Given a graph, we first extract all cliques of size k; for this example we choose $k = 3$. This is followed by generating the *clique graph*, in which each k-clique in the original graph is represented by a vertex. An edge is added between any two k-cliques in the clique graphs that are adjacent. For the case $k = 3$, this means an edge is added between any two 3-cliques in the clique graph that share two common vertices (of the original graph). The connected components in the clique graph represent a community, and the actual members of the community are obtained by gathering the vertices of the individual cliques that form the connected component. In our example in Figure 6, we obtain two communities, which share a common vertex (vertex 2), forming an overlapping community structure.

A large clique of size $q \geq k$ contains $\binom{q}{k}$ different *k-cliques*. An algorithm that tries to locate the k-cliques individually and examine the adjacency between them can, therefore, be very very slow for large networks. Two observations are made in [41] that help one come up with a better strategy. First, a clique of size q is clearly a k-clique connected subset for

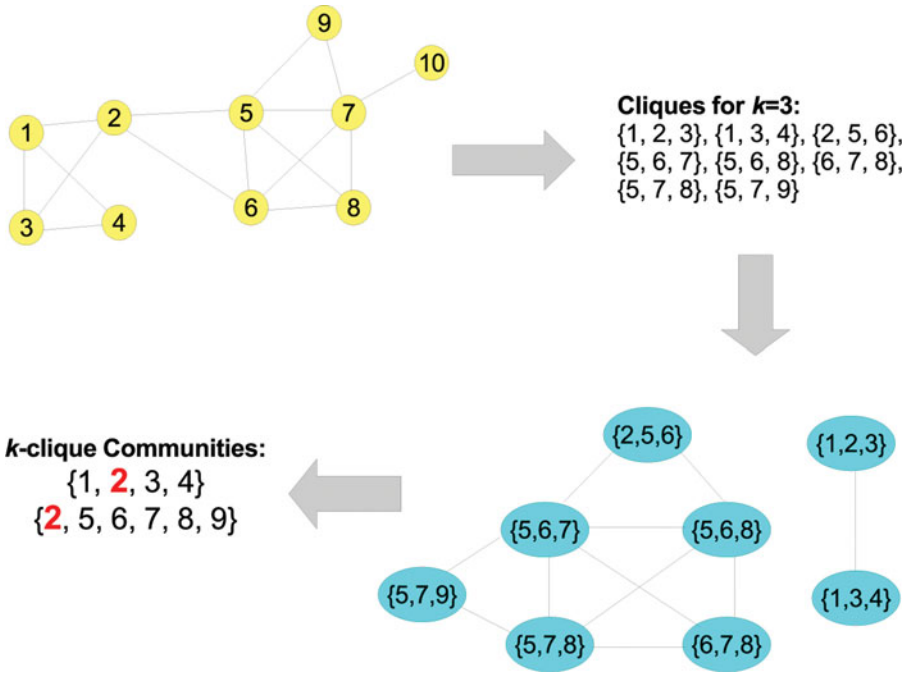

Figure 6 Illustration of overlapping community detection by the Clique Percolation Method (CPM) algorithm on a sample graph. Steps involved are 1) Detecting the k-cliques 2) Forming the clique-graph 3) Merging members of the connected components in the clique-graph to obtain the k-clique communities. In this example, node 2 is shared by the two communities formed, resulting in an overlapping structure.

any $k \leq q$. Second, two large cliques that share at least $k-1$ nodes form one k-clique connected component as well. Leveraging these, the strategy of [41] avoids searching for k-cliques individually and instead first locates the large cliques in the network and then looks for the k-clique-connected subsets of given k (that is, the k-clique communities) by studying the overlap between them. More specifically, their algorithm first constructs a symmetric *clique-clique overlap matrix*, in which each row represents a large (to be precise a maximal) clique, each matrix entry is equal to the number of common nodes between the two corresponding cliques, and each diagonal entry is equal to the size of the clique. The k-clique communities for a given k are then equivalent to connected clique components in which the neighboring cliques are linked to each other by at least $k-1$ common nodes. These components are then found [41] by erasing every off-diagonal entry smaller than $k-1$ and every diagonal entry smaller than k in the matrix, replacing the remaining elements by one, and then carrying out a component analysis of this matrix. The resulting separate components are equivalent to the different k-clique communities.

Our method. We devised an algorithm based on a similar idea as the previous procedure, but using a variant of our heuristic maximum clique algorithm (Algorithm 2) for the core clique detection step.

The standard CPM in essence presupposes finding *all* maximal cliques in a network. The number of maximal cliques in a network can, in general, be exponential in the number

of nodes n in the network. In our method, we work instead with a smaller set of maximal cliques. In particular, the basic variant of our method considers exactly one clique per vertex, the largest of all the maximal cliques that it belongs to. Clearly, this can be too restrictive a requirement and might fail to add deserving members to a k-clique community. To allow for more refined solutions, we include a parameter c in the method, which tells us how many additional cliques per vertex would be considered. The case $c = 0$ corresponds to no additional cliques other than the largest maximal clique containing the vertex. The case $c \geq 1$ corresponds to c additional maximal cliques (each of size at least k) per vertex. Hence, in total, $n(c + 1)$ cliques will be collected. For a particular vertex v, the largest maximal clique containing it can be heuristically obtained by exploring the neighbor of v with *maximum degree* in the corresponding step in Algorithm 2. In contrast, to obtain the other maximal cliques involving v (case $c \geq 1$), we explore a *randomly* chosen neighbor of v, regardless of its degree value.

Test on synthetic networks. We tested our algorithm on the Lancichinetti-Fortunato-Radicchi (LFR) benchmarks[5] proposed in [30]. These benchmarks have the attractive feature that they allow for generation of synthetic networks with known communities (ground truth). We generated graphs with $n = 1000$ nodes, average degree $K = 10$, and power law exponents $\tau_1 = 2$ and $\tau_2 = 1$ in the LFR model. We set the maximum degree K_{max} to 50, and the minimum and maximum community sizes C_{min} and C_{max} to 20 and 50, respectively. We used two far-apart values each for the mixing parameter μ, the fraction of overlapping nodes O_n, and the number of communities each overlapping node belongs to O_m. Specifically, for μ we used 0.1 and 0.3, for O_n we used 10% and 50%, and for O_m we used 2 and 8. All together, these combinations of parameters resulted in eight graphs in the testbed.

We evaluated the performance of our algorithm against the ground truth using *Omega Index* [10], which is the overlapping version of the Adjusted Rand Index (ARI) [24]. Intuitively, Omega Index measures the extent of agreement between two given sets of communities by looking at node pairs that occur the same number of times (possibly none) in both. We used three values for the parameter c of our method: 0, 2, and 5. Recall that c set to zero corresponds to picking only the largest clique for each node. As c is increased, more and more large cliques (each of size at least k) are considered for each node.

We compare the performance of our method with that of CFinder,[6] an implementation of CPM [41]. We used the command line utility provided in the package for all experiments. For this study, we used a MacBook Pro running OS X v.10.9.2 with 2.66GHz Intel Core i7 processor with 2 cores, 256KB of L2 cache per core, 4MB of L3 cache and 8GB of main memory.

Table V shows the experimental results we obtained. The first three columns of the table list the various parameters used to generate each test network. The fourth column lists the number of ground truth communities ($C(GT)$) in each network. The remaining columns in the table show performance in terms of the number of communities detected

[5]http://sites.google.com/site/andrealancichinetti/files
[6]http://www.cfinder.org

							Our Method								
				CFinder			$c = 0$			$c = 2$			$c = 5$		
μ	O_m	O_n	$C(GT)$	C	S	Ω	C	S	Ω	C	S	Ω	C	S	Ω
0.1	2	10%	34	37	55	0.842	60	64	0.758	49	76	0.825	40	59	0.840
0.1	2	50%	44	62	153	0.402	87	116	0.280	83	158	0.363	68	158	0.395
0.1	8	10%	51	58	30	∼0.0	76	46	∼0.0	65	39	∼0.0	61	35	∼0.0
0.1	8	50%	130	74	68	0.030	74	56	0.020	79	66	0.028	79	72	0.030
0.3	2	10%	35	50	53	0.612	69	67	0.461	64	73	0.577	55	60	0.605
0.3	2	50%	44	60	95	0.193	92	93	0.109	85	102	0.152	77	106	0.169
0.3	8	10%	48	64	35	0.304	79	59	0.239	74	55	0.293	69	46	0.301
0.3	8	50%	142	57	43	0.019	58	46	0.013	61	49	0.017	59	45	0.018

Table V Results of communities detected using our new method and CFinder on LFR benchmarks [30]. All networks have $n = 1000$ nodes, and the parameters used to generate the networks are listed in the first three columns: μ is the mixing parameter, O_m is the number of communities each overlapping node is assigned to, O_n is the fraction of overlapping nodes, $C(GT)$ is the number of communities in the input graph (ground truth), C denotes the number of communities, S the number of shared nodes, and Ω the omega index.

(C), total number of shared nodes (S) and the Omega Index (Ω) for CFinder and our method with different values for the parameter c. In our experiments, for CFinder as well as all variants of our algorithm, we found that the value of $k = 4$ gives the best Omega Index value relative to the ground truth. All results reported in Table V are therefore for $k = 4$.

In general, we observe that there is a close correlation between our method and CFinder in terms of all three of the quantities C, S, and Ω. It can be seen that as we increase the value of c in our method, the Omega Index values get closer and closer to that of CFinder. For our algorithm run with $c = 0$, the Omega Index is about 75% of that of CFinder. When run with $c = 2$, it is about 92% and for $c = 5$, it is about 99%. When we increased the value of c even further to 10, we observed that the Omega Index was almost identical to that obtained by CFinder. From this, one can see that we can get almost exactly the same results as the CPM method by using our algorithm, which uses only a small set of maximal cliques, as opposed to all the maximal cliques in the graph.

Table VI shows the time taken by CFinder and our method, run with $c = 0$, 2, and 5. For our method, the table lists the total runtime τ as well as the time τ_c spent on just the clique detection part and the ratio τ_c/τ expressed in percents (τ_c%). The remaining time $\tau - \tau_c$ is spent on building the clique-clique overlap matrix, eliminating cliques, component analysis, and generating the communities. As a side note, we point out that our immediate goal in the implementation of these later phases has been quick experimentation, rather than efficient code, and therefore, there is very large room for improving the runtimes. This is also reflected by the numbers; one can see from the table that the average percentage of time our algorithm spends on clique finding decreases as c is increased; the quantity is about 30% when $c = 0$ and about 16% when $c = 5$. Yet, looking only at the total time taken in Table VI, and comparing CFinder and our algorithm run with $c = 5$ (the case where the Omega Index values match most closely with CFinder), we see that our algorithm is at least 4× faster on average. When c is set to 0 and 2, the mean speedups are 51× and 13×, respectively.

				Our Method								
			CFinder	$c = 0$			$c = 2$			$c = 5$		
μ	O_m	O_n	τ	τ_c	τ	τ_c%	τ_c	τ	τ_c%	τ_c	τ	τ_c%
0.1	2	10%	0.913	0.005	0.031	17.3%	0.025	0.255	9.7%	0.050	0.912	5.5%
0.1	2	50%	0.855	0.004	0.016	22.0%	0.010	0.089	10.8%	0.017	0.254	6.8%
0.1	8	10%	0.732	0.004	0.025	17.4%	0.012	0.159	7.4%	0.024	0.550	4.4%
0.1	8	50%	0.358	0.002	0.005	42.8%	0.006	0.016	36.1%	0.011	0.042	26.7%
0.3	2	10%	0.592	0.004	0.013	26.0%	0.011	0.090	11.8%	0.027	0.321	8.3%
0.3	2	50%	0.434	0.002	0.006	38.1%	0.006	0.024	27.2%	0.013	0.068	18.4%
0.3	8	10%	0.472	0.003	0.011	25.7%	0.008	0.064	12.5%	0.016	0.205	7.6%
0.3	8	50%	0.326	0.002	0.005	50.0%	0.006	0.011	54.9%	0.017	0.034	51.3%

Table VI Timing results of our new method and CFinder on the LFR benchmarks. All networks have $n = 1000$ nodes, and the parameters used to generate the graphs are given by the first three columns: μ is the mixing parameter, O_m is the number of communities each overlapping node is assigned to and O_n is the fraction of overlapping nodes, τ_c is the time spent (in seconds) on computing the cliques, τ is the total time (in seconds), and τ_c% is the percentage of time spent for clique computation.

Because the source code for CFinder is not publicly available, we were not able to measure exactly the proportion of time it spends on clique finding. We suspect it constitutes a vast proportion of the total runtime.

Test on real-world networks. We also tested our method and CFinder on four of the real-world graphs from our testbed in Section 4—*cond-mat-2003*, *email-Enron*, *dictionary28*, and *roadNet-CA*—and a user-interest-based graph generated using data collected from Facebook. We briefly explain here how this graph was generated. Every user on Facebook has a *wall*, which is a the user's profile space that allows the posting of messages, often short or temporal notes or comments by other users. We generate a graph with the *walls* as vertices, and assign an edge between a pair of vertices, if there is at least one user who has commented on both walls. We assign edge weights proportional to the number of common users, because we consider this to be an indicator of the strength of the connection. A more elaborate explanation of the data collection method and network generation for

				OurMethod							
	CFinder			$c = 0$				$c = 5$			
Name	C	S	τ	C	S	τ_c	τ	C	S	τ_c	τ
cond-mat-2003	4132	5180	68.46	3081	3937	0.130	14.000	3070	4460	0.917	417.151
email-Enron	—	—	—	3947	4229	0.096	7.505	2518	3899	0.819	353.236
dictionary28	3675	4308	9.67	2185	1799	0.040	1.132	3222	4154	0.300	95.596
roadNet-CA	41	0	312.21	42	0	1.788	2.290	42	0	13.287	21.620
facebook	29	29	3.42	38	46	0.002	0.011	19	27	0.023	0.252

Table VII Results of our new algorithm and CFinder on three real-world graphs from Table II and one additional graph generated using data collected from Facebook. The structural properties of the former three graphs are listed in Table II. The Facebook graph has 1144 vertices and 2561 edges after thresholding. C is the total number of communities found, S is the total number of shared nodes, τ_c is the time spent (in seconds) on computing the cliques, and τ, the total time (in seconds). A '-' denotes that the algorithm did not complete in 30 minutes.

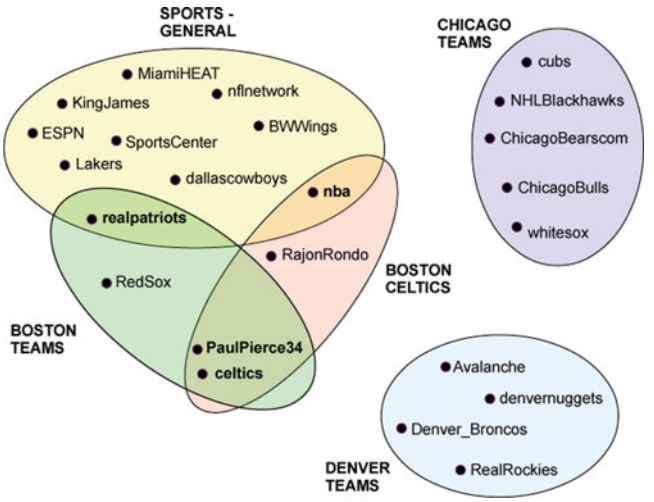

Figure 7 Some Facebook communities detected by our algorithm.

the Facebook graph is discussed in [42]. As this is a weighted graph, we used a threshold (of 0.009) to include only strong links, resulting in a network with 1144 vertices and 2561 edges.

The results of our method for $c = 0$ and 5, $k = 4$, and CFinder for the case $k = 4$ on all the five real-world graphs is shown in Table VII. The table lists the number of communities found by each algorithm, total number of shared nodes, and the total time taken. For our algorithm, it also specifies the time spent on clique finding. These graphs do not have any known community structure, so we were not able to measure the Omega Index values. For the graph *email-Enron*, CFinder was not able to complete within 30 minutes. For the $c = 0$ case, our algorithm comfortably outperforms CFinder, even in terms of total time, on average performing about 115× faster. For the $c = 5$ case, the total time taken by our algorithm turns out to be higher for some cases, and lower for others. However, it should be noted that the time taken for clique finding is incomparably smaller than the post-clique-finding processing time.

We also present some of the sports-related communities detected by our algorithm in the Facebook graph in Figure 7, which is self-explanatory.

7. CONCLUSION

We presented a new exact algorithm and a new heuristic algorithm for the maximum clique problem. We performed extensive experiments on three broad categories of graphs and compared the performance of our algorithms to the algorithms (CP) [8], (*cliquer*) [40], (MCQ, MCS; [49, 50]), (MCQD; [28]), and (BBMC; [48]). For DIMACS benchmark graphs and certain dense synthetic graphs (*rmat_sd2*), our new exact algorithm performs comparably with the CP algorithm, but slower than the others. For large sparse graphs, both synthetic and real-world, our new algorithm runs several orders of magnitude faster than the others, and its general runtime is observed to grow nearly linearly with the size of the

graphs. The heuristic, which runs orders of magnitude faster than our exact algorithm and the others, delivers an optimal solution for 83% of the test cases, and when it is suboptimal, its accuracy ranges between 0.83 and 0.99 for the graphs in our testbed. We also showed how the algorithms can be parallelized. Finally, as a demonstration of the applications of clique-finding algorithms, we presented a novel and efficient alternative based on our algorithms to the Clique Percolation Method [41], to detect overlapping communities in networks.

Maximum clique detection is often avoided by practitioners from being used as a component in a network analysis algorithm on the grounds of its NP-hardness. The results shown here suggest that this is not necessarily true, because maximum cliques can, in fact, be detected rather quickly for most real-world networks that are characterized by sparsity and other structures well suited for branch-and-bound type algorithms.

FUNDING

This work is supported in part by the following grants: NSF awards CCF-0833131, CNS-0830927, IIS-0905205, CCF-0938000, CCF-1029166, and OCI-1144061; DOE awards DE-FG02-08ER25848, DE-SC0001283, DE-SC0005309, DESC0005340, and DESC0007456; AFOSR award FA9550-12-1-0458. The work of Assefaw Gebremedhin is supported in part by the NSF award CCF-1218916 and by the DOE award DE-SC0010205.

REFERENCES

1. J. G. Augustson and J. Minker. “An Analysis of Some Graph Theoretical Cluster Techniques.” *Journal of the ACM* 17:4 (1970), 571–588.
2. L. Babel and G. Tinhofer. “A Branch and Bound Algorithm for the Maximum Clique Problem.” *Mathematical Methods of Operations Research* 34:3 (1990), 207–217.
3. V. Batagelj and A. Mrvar, *Pajek Datasets*. Available online (`http://vlado.fmf.uni-lj.si/pub/networks/data/`), 2006.
4. V. Boginski, S. Butenko, and P. M. Pardalos. “Statistical Analysis of Financial Networks.” *Computational Statistics & Data Analysis* 48:2 (2005), 431–443.
5. I. M. Bomze, M. Budinich, P. M. Pardalos, and M. Pelillo. “The Maximum Clique Problem.” In *Handbook of Combinatorial Optimization*, pp. 1–74. Kluwer Academic Publishers, pp. 1–74, 1999.
6. R. E. Bonner. “On Some Clustering Techniques.” *IBM Journal of Research and Development* 8:1(1964), 22–32.
7. A. E. Brouwer, J. B. Shearer, N. J. A. Sloane, and W. D. Smith. “A New Table of Constant Weight Codes.” *IEEE Transactions on Information Theory* 36:6 (1990), 1334–1380.
8. R. Carraghan and P. Pardalos. “An Exact Algorithm for the Maximum Clique Problem.” *Operations Research Letters* 9:6 (1990), 375–382.
9. D. Chakrabarti and C. Faloutsos. “Graph Mining: Laws, Generators, and Algorithms.” *ACM Computing Surveys* 38:1 (2006) p.4.
10. L. M. Collins and C. W. Dent. “Omega: A General Formulation of the Rand Index of Cluster Recovery Suitable for Non-Disjoint Solutions.” *Multivariate Behavioral Research* 23:2 (1988), 231–242.

11. S. R. Corman, T. Kuhn, R. D. Mcphee, and K. J. Dooley. "Studying Complex Discursive Systems: Centering Resonance Analysis of Communication." *Human Communication Research* 28:2 (2002), 157–206.
12. T. A. Davis and Y. Hu. "The University of Florida Sparse Matrix Collection." *ACM Transactions on Mathematical Software (TOMS)* 38:1 (2011), 1–25.
13. P. Domingos and M. Richardson. "Mining the Network Value of Customers." In *Proceedings of the 7th ACM SIGKDD (KDD'01)*, pp. 57–66. New York, NY, USA: ACM, 2001.
14. D. Eppstein and D. Strash.Listing all maximal cliques in large sparse real-world graphs, in *Proceedings of the 10th International Conference on Experimental Algorithms (SEA'11)*, edited by P. M. Pardalos and S. Rebennack pp. 364–375. Berlin, Heidelberg: Springer-Verlag, 2011.
15. M. Faloutsos, P. Faloutsos, and C. Faloutsos. "On Power-Law Relationships of the Internet Topology." In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '99*, pp. 251–262. New York, NY, USA: ACM, 1999.
16. M. Ferronato, C. Janna, G. Gambolati. "Mixed Constraint Preconditioning in Computational Contact Mechanics." *Computer Methods in Applied Mechanics and Engineering* 197:45–48 (2008), 3922–3931.
17. S. Fortunato. "Community Detection in Graphs." *Physics Reports* 486:3 (2010), 75–174.
18. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.
19. *Google programming contest*. Available online (`http://www.google.com/programming-contest/`).
20. G. Gutin, J. Gross, J. Yellen. "Discrete Mathematics & Its Applications." *Handbook of Graph Theory*. CRC Press, 2004.
21. B. H. Hall, A. B. Jaffe, and M. Trajtenberg. "The NBER patent citation data file: Lessons, insights and methodological tools." Technical Report, NBER Working Paper, 8498, 2001.
22. R. Horaud and T. Skordas. "Stereo Correspondence Through Feature Grouping and Maximal Cliques." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 11:11(1989), 1168–1180.
23. D. Howe. "Foldoc: Free On-Line Dictionary of Computing." Available online (`http://foldoc.org/`).
24. L. Hubert and P. Arabie. "Comparing Partitions." *Journal of Classification* 2:1 (1985), 193–218.
25. D. Johnson and M. A. Trick, Editors, *Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge*. In DIMACS Series on Discrete Mathematics and Theoretical Computer Science 26, 1996.
26. *KDD Cup*. Available online (`http://www.cs.cornell.edu/projects/kddcup/index.html`), 2003.
27. G. R. Kiss, C. Armstrong, R. Milroy, and L. Piper. *An Associative Thesaurus of English and Its Computer Analysis*. The Computer and Literary Studies. Edinburgh University Press, 1973.
28. J. Konc and D. Janežič. "An Improved Branch and Bound Algorithm for the Maximum Clique Problem." *MATCH Communications in Mathematical Computer Chemistry* 58 (2007), 569–590.
29. R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. "Extracting Large-Scale Knowledge Bases from the Web." In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB'99)*, edited by M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, pp. 639–650. San Francisco, CA, USA: Morgan Kaufmann pp. 639–650,1999.
30. A. Lancichinetti, S. Fortunato, and F. Radicchi. "Benchmark Graphs for Testing Community Detection Algorithms." *Physical Review E* 78:4 (2008), 046110.

31. J. Leskovec, *Stanford Large Network Dataset Collection*. Avaiable online (`http://snap.stanford.edu/data/index.html`).
32. J. Leskovec, L. Adamic, and B. Adamic. "The Dynamics of Viral Marketing." *ACM Transactions on the Web* 1:1 (2007).
33. J. Leskovec, D. Huttenlocher, and J. Kleinberg. "Predicting Positive and Negative Links in Online Social Networks." In *Proceedings of the 19th International Conference on World Wide Web (WWW'10)*, pp. 641–650. New York, NY, USA: ACM, 2010.
34. J. Leskovec, J. Kleinberg, and C. Faloutsos . "Graphs Over Time: Densification Laws, Shrinking Diameters and Possible Explanations." In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD'05)*, pp. 177–187. New York, NY, USA: ACM, 2005.
35. J. Leskovec, J. Kleinberg, and C. Faloutsos. "Graph Evolution: Densification and Shrinking Diameters." *ACM Transactions on Knowledge Discovery from Data* 1:1 (2007).
36. T. Matsunaga, C. Yonemori, E. Tomita, and M. Muramatsu. "Clique-Based Data Mining for Related Genes in a Biomedical Database." *BMC Bioinformatics* 10 (2010), 205.
37. C. McCreesh and P. Prosser. "Multi-Threading a State-of-the-Art Maximum Clique Algorithm." *Algorithms* 6:4 (2013), 618–635.
38. M. E. J. Newman. "Coauthorship Networks and Patterns of Scientific Collaboration." In *Proceedings of the National Academy of Sciences of the United States of America* 101(2004), 5200–5205.
39. S. Niskanen and P. R. J. Östergård. "Cliquer user's guide, version 1.0." Technical Report T48, Communications Laboratory, Helsinki University of Technology, Espoo, Finland, 2003.
40. P. R. J. Östergård. "A Fast Algorithm for the Maximum Clique Problem." *Discrete Applied Mathematics* 120:1-3 (2002), 197–207.
41. G. Palla, I. Derényi, I. Farkas, and T. Vicsek. "Uncovering the Overlapping Community Structure of Complex Networks in Nature and Society." *Nature* 435 (2005), pp. 814–818.
42. D. Palsetia, M. M. Patwary, A. Agrawal, and A. Choudhary. "Excavating Social Circles via User Interests." *Social Network Analysis and Mining* 4:1 (2014), 1–12.
43. P. M. Pardalos and J. Xue. "The Maximum Clique Problem." *Journal of Global Optimization* 4 (1994), 301–328.
44. M. Pavan and M. Pelillo. "A New Graph-Theoretic Approach to Clustering and Segmentation." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'03)*, pp. 145–152. Washington, DC, USA: IEEE Computer Society, 2003.
45. P. Prosser. "Exact Algorithms for Maximum Clique: A Computational Study." *Algorithms* 5:4 (2012), 545–587.
46. M. Richardson, R. Agrawal, and P. Domingos. "Trust Management for the Semantic Web." In *Proceedings of the Second International Semantic Web Conference (ISWC)*, pp. 351–368. Berlin, Heidelberg: Springer-Verlag, 2003.
47. S. Sadi, S. Öğüdücü, and A. S. Uyar. "An Efficient Community Detection Method Using Parallel Clique Finding Ants." In *Proceedings of the IEEE Congress on Evolutionary Computation*, pp. 1–7. IEEE, 2010.
48. P. San Segundo, D. Rodríguez-Losada, and A. Jiménez. "An Exact Bit-Parallel Algorithm for the Maximum Clique Problem." *Computers & Operations Research* 38:2 (2011), 571–581.
49. E. Tomita and T. Seki. "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*, pp. 278–289. Berlin, Heidelberg: Springer-Verlag, 2003.

50. E. Tomita, Y. Sutani, T. Higashi, S. Takahashi, and M. Wakatsuki. "A Simple and Faster Branch-and-Bound Algorithm for Finding a Maximum Clique." In *Proceedings of the 4th International Workshop on Algorithms and Computation (WALCOM)*, edited by M. Rahman and S. Fujita, pp. 191–203. Berlin, Heidelberg: Springer, 2010.
51. B. van Rietbergen, H. Weinans, R. Huiskes, and A. Odgaard. "A New Method to Determine Trabecular Bone Elastic Properties and Loading Using Micromechanical Finite-Element Models." *Journal of Biomechanics* 28:1 (1995), 69–81.
52. L. Wang, L. Zhou, J. Lu, and J. Yip. "An Order-Clique-Based Approach for Mining Maximal Colocations." *Information Sciences* 179:19 (2009), 3370–3382.

APPENDIX

G	$\|V\|$	$\|E\|$	Δ	ω	τ_{A1}	τ_{ES} [14]
foldoc	13,356	91,470	728	9	0.05	0.13
eatRS	23,219	304,938	1090	9	1.81	1.03
hep-th	27,240	341,923	2411	23	4.48	1.70
patents	240,547	560,943	212	6	0.13	1.65
days-all	13,308	148,035	2265	28	75.37	5.18
roadNet-TX	1,393,383	1,921,660	12	4	0.26	4.00
amazon0601	403,394	2,247,318	2752	11	0.20	6.03
email-EuAll	265,214	364,481	7636	16	0.92	1.33
web-Google	916,428	4,322,051	6332	44	0.35	9.70
soc-wiki-Vote	8,297	100,762	1065	17	4.31	1.14
soc-slashdot0902	82,168	504,230	2252	27	25.54	2.58
cit-Patents	3,774,768	16,518,947	793	11	19.99	58.64
soc-Epinions1	75,888	405,740	3044	23	15.01	4.78
soc-wiki-Talk	2,394,385	4,659,565	100029	26	6885.13	216.00
web-berkstan	685,230	6,649,470	84230	201	44.70	20.87

Table VIII Comparison of runtimes of our new exact maximum clique-finding algorithm (τ_{A1}) and the maximal clique enumeration algorithm by [14] (τ_{ES}) for the Pajek and Stanford data sets in our testbed. The runtimes under column τ_{ES} are directly quoted from [14]. The column Δ denotes the maximum degree, and ω the maximum clique size of the graph.

G	ω	$P1$	$P2$	$P3$	$P4$	$P5$
cond-mat-2003	25	29,407	48,096	6,527	2,600	17,576
email-Enron	20	32,462	155,344	4,060	110,168	8,835,739
dictionary28	26	52,139	4,353	2,114	542	107
Fault_639	18	36	13,987,719	126	10,767,992	1,116
audikw_1	36	4,101	38,287,830	59,985	32,987,342	721,938
bone010	24	37,887	34,934,616	361,170	96,622,580	43,991,787
af_shell10	15	19	25,582,015	75	40,629,688	2,105
as-Skitter	67	1,656,570	6,880,534	981,810	26,809,527	737,899,486
roadNet-CA	4	1,487,640	1,079,025	370,206	320,118	4,302
kkt_power	11	1,166,311	4,510,661	401,129	1,067,824	1,978,595
rmat_er_1	3	780	1,047,599	915	118,461	8,722
rmat_er_2	3	2,019	2,094,751	2,351	235,037	23,908
rmat_er_3	3	4,349	4,189,290	4,960	468,086	50,741
rmat_er_4	3	9,032	8,378,261	10,271	933,750	106,200
rmat_er_5	3	18,155	16,756,493	20,622	1,865,415	212,838
rmat_sd1_1	6	39,281	1,004,660	23,898	151,838	542,245
rmat_sd1_2	6	90,010	2,004,059	56,665	284,577	1,399,314
rmat_sd1_3	6	176,583	4,013,151	106,543	483,436	2,677,437
rmat_sd1_4	6	369,818	8,023,358	214,981	889,165	5,566,602
rmat_sd1_5	6	777,052	16,025,729	455,473	1,679,109	12,168,698
rmat_sd2_1	26	110,951	853,116	88,424	1,067,824	614,813,037
rmat_sd2_2	35	232,352	1,645,086	195,427	81,886,879	1,044,068,886
rmat_sd2_3	39	470,302	3,257,233	405,856	45,841,352	1,343,563,239
rmat_sd2_4	43	*	*	*	*	*
rmat_sd2_5	N	*	*	*	*	*
hamming6-4	4	0	704	0	583	0
johnson8-4-4	14	0	1855	0	136,007	0
keller4	11	0	9435	0	8,834,190	0
c-fat200-5	58	0	8473	0	70449	0
brock200_2	12	0	9876	0	349,427	0

Table IX $P1$, $P2$, $P3$, $P4$, and $P5$ are the number of vertices pruned in steps Pruning 1, 2, 3, 4, and 5 of Algorithm 1. An asterisk (*) indicates that the algorithm did not terminate within 25,000 seconds for that instance; ω denotes the maximum clique size. For some of the graphs, none of the algorithms computed the maximum clique size in a reasonable time; the entry for the maximum clique size marked with N, stands for "Not Known."

G	$\|V\|$	$\|E\|$	ω	τ_{CP}	$\tau_{cliquer}$	$\tau_{MCQD+CS}$	τ_{MCQ1}	τ_{MCSa1}	τ_{BBMC1}	τ_{A1}	ω_{A2}	τ_{A2}
brock200_1	200	14,834	21	*	10.37	0.75	7.11	5.48	1.7	*	18	0.02
brock200_2	200	9,876	12	0.98	0.02	0.01	0.1	0.1	<0.01	1.1	10	<0.01
brock200_3	200	12,048	15	14.09	0.16	0.03	0.35	0.4	0.1	14.86	12	<0.01
brock200_4	200	13,089	17	60.25	0.7	0.12	0.88	1.11	0.2	65.78	14	<0.01
brock400_1	400	59,723	27	*	*	671.24	4145.68	2873.91	762.14	*	20	<0.01
brock400_2	400	59,786	29	*	*	272.31	2848.1	2123.73	546.84	*	20	<0.01
brock400_3	400	59,681	31	*	*	532.77	2186.3	1523.55	431.92	*	20	<0.01
brock400_4	400	59,765	33	*	*	266.43	1038.33	881.31	211.29	*	22	<0.01
brock800_1	800	207,505	N	*	*	*	*	*	*	*	17	0.3
brock800_2	800	208,166	N	*	*	*	*	*	*	*	18	0.4
brock800_3	800	207,333	N	*	*	*	*	*	*	*	17	0.4
brock800_4	800	207,643	26	*	*	*	*	*	3455.58	*	17	0.4
c-fat200-1	200	1,534	12	<0.01	<0.01	<0.01	<0.01	<0.01	0.02	<0.01	12	<0.01
c-fat200-2	200	3,235	24	<0.01	<0.01	<0.01	<0.01	<0.01	0.02	<0.01	24	<0.01
c-fat200-5	200	8,473	58	0.6	0.33	0.01	0.03	0.03	0.03	0.93	58	0.04
c-fat500-1	500	4,459	14	<0.01	<0.01	<0.01	0.02	0.02	0.05	<0.01	14	<0.01
c-fat500-2	500	9,139	26	0.02	<0.01	0.01	0.02	0.02	0.04	0.01	26	0.01
c-fat500-5	500	23,191	64	3.07	<0.01	<0.01	0.03	0.03	0.05	*	64	0.11
hamming6-2	64	1,824	32	0.68	<0.01	<0.01	<0.01	0.01	<0.01	0.33	32	<0.01
hamming6-4	64	704	4	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	4	<0.01
hamming8-2	256	31,616	128	*	0.01	0.01	0.04	0.04	0.04	*	128	0.67
hamming8-4	256	20,864	16	*	<0.01	0.1	0.4	0.45	0.23	*	16	0.03
hamming10-2	1,024	518,656	512	*	0.31	-	0.36	0.371	0.13	*	512	95.24
johnson8-2-4	28	210	4	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	4	<0.01
johnson8-4-4	70	1,855	14	0.19	<0.01	<0.01	0.01	0.02	0.01	0.23	14	<0.01
johnson16-2-4	120	5,460	8	20.95	0.04	0.42	0.85	0.95	0.41	22.07	8	<0.01
keller4	171	9,435	11	22.19	0.15	0.02	0.21	0.33	0.12	23.35	11	<0.01
keller5	776	225,990	N	*	*	*	*	*	*	*	22	0.6
keller6	3361	4,619,898	N	*	*	*	*	*	*	*	45	99.21
MANN_a9	45	918	16	1.73	<0.01	<0.01	<0.01	<0.01	<0.01	2.5	16	<0.01
MANN_a27	378	70,551	126	*	*	3.3	5.75	6.03	0.97	*	125	1.74
MANN_a45	1035	533,115	345	*	*	*	4612.29	3431.67	250.12	*	341	59.96
p_hat300-1	300	10,933	8	0.14	0.01	<0.01	0.07	0.08	0.06	0.14	8	<0.01
p_hat300-2	300	21,928	25	831.52	0.32	0.03	0.38	0.23	0.09	854.59	24	0.03
p_hat300-3	300	33,390	36	*	578.58	4.31	69.91	13.53	3.2	*	26	<0.01
p_hat500-1	500	31,569	9	2.38	0.07	0.04	0.33	0.35	0.12	2.44	9	0.02
p_hat500-2	500	62,946	36	*	159.96	1.2	63.89	3.87	0.96	*	34	0.14
p_hat500-3	500	93,800	50	*	*	324.23	*	1428.02	311.06	*	39	0.27
p_hat700-1	700	60,999	11	12.7	0.12	0.13	0.96	0.92	0.23	12.73	9	0.04
p_hat700-2	700	121,728	44	*	*	12.28	675.72	29.36	6.76	*	26	0.15
p_hat1000-1	1,000	122,253	10	97.39	1.33	0.41	1.89	2.29	0.69	98.48	10	0.11
p_hat1000-2	1,000	244,799	46	*	*	406.71	*	1359.88	382.31	*	33	0.57
san200_0.7_1	200	13,930	30	*	0.99	<0.01	0.05	0.47	0.1	*	16	0.01
san200_0.7_2	200	13,930	18	*	0.02	<0.01	0.072	0.04	0.03	*	14	<0.01
san200_0.9_2	200	17,910	60	*	13.4	0.8	18.5	5.85	1.42	*	34	<0.01
san200_0.9_3	200	17,910	44	*	561.64	3.16	134.67	119.27	28.02	*	31	<0.01
san400_0.5_1	400	39,900	13	*	<0.01	0.1	0.11	0.11	0.1	*	8	<0.01
san400_0.7_1	400	55,860	40	*	*	0.35	1.59	2.94	0.74	*	22	0.1
san400_0.7_2	400	55,860	30	*	*	0.1	12.71	19.51	4.79	*	18	<0.01
san400_0.7_3	400	55,860	22	*	5.04	2.1	9.59	10	2.77	*	16	<0.01
san1000	1000	250,500	15	*	0.09	0.45	43.12	8.48	2.55	*	10	0.5

Table X Comparison of runtimes of algorithms: [8] (*CP*), [40] ($\tau_{cliquer}$), [28] ($\tau_{MCQD+CS}$), [45, 49] (τ_{MCQ1}), [45, 50] (τ_{MCSa1}), and [45, 48] (τ_{BBMC1}), with that of our new exact algorithm (τ_{A1}) for DIMACS graphs. An asterisk (*) indicates that the algorithm did not terminate within 7,200 seconds for that instance; ω denotes the maximum clique size, ω_{A2} the maximum clique size found by our heuristic, and τ_{A2}, its runtime. For some of the graphs, none of the algorithms computed the maximum clique size in a reasonable time; the entries for the maximum clique size is marked with N, stand for "Not Known."