A Parallel Framework for Approximate Max-Dicut in Partitionable Graphs

Nico Bertram
Department of Computer Science, Technische Universität Dortmund, Germany

Jonas Ellert
Department of Computer Science, Technische Universität Dortmund, Germany

Johannes Fischer
Department of Computer Science, Technische Universität Dortmund, Germany

Abstract

Computing a maximum cut in *undirected* and weighted graphs is a well studied problem and has many practical solutions that also scale well in shared memory (despite its NP-completeness). For its counterpart in *directed* graphs, however, we are not aware of practical solutions that also utilize parallelism. We engineer a framework that computes a high quality approximate cut in directed and weighted graphs by using a graph partitioning approach. The general idea is to partition a graph into k subgraphs using a parallel partitioning algorithm of our choice (the first ingredient of our framework). Then, for each subgraph in parallel, we compute a cut using any polynomial time approximation algorithm (the second ingredient). In a final step, we merge the locally computed solutions using a high-quality or exact parallel Max-Dicut algorithm (the third ingredient). On graphs that can be partitioned well, the quality of the computed cut is significantly better than the best cut achieved by any linear time algorithm. This is particularly relevant for large graphs, where linear time algorithms used to be the only feasible option.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms; Theory of computation → Design and analysis of algorithms

Keywords and phrases maximum directed cut, graph partitioning, algorithm engineering, approximation, parallel algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.10

Supplementary Material *Software (Source Code)*: `https://github.com/NicoBertram/par-max-dicut`

Funding This work has been supported by the German Research Association (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis", project A6.

1 Introduction

A *directed* and *weighted graph* G is a tuple (V, E, w) with the set of *vertices* $V = \{1, ..., n\}$, the set of *edges* $E \subseteq V^2$ and nonnegative *weights* $w\colon V^2 \to \mathbb{R}_{\geq 0}$. If $(u, v) \notin E$, we set $w(u, v) = 0$. A *cut* in a directed and weighted graph $G = (V, E, w)$ is defined by a partitioning of V into two complementary subsets $S \subseteq V$ and $T = V \setminus S$. The value of a cut with respect to S and T is defined by $C(S, T) = \sum_{i \in S, j \in T} w(i, j)$, i.e. we sum the weights of the edges with origin in S and target in T. The maximum cut is then defined by $C_{max} = \max_{S \subseteq V, T = V \setminus S} C(S, T)$.

The problem of finding a maximum cut in a directed and weighted graph is denoted by Max-Dicut. It can be seen as a generalization of its well-studied counterpart Max-Cut in *undirected* graphs, which is one of the classical NP-complete problems listed by Karp [13]. In fact, Max-Dicut is at least as hard as Max-Cut, since every instance of Max-Cut can be trivially reduced to an instance of Max-Dicut (by replacing each undirected edge (u, v) with two directed edges (u, v) and (v, u) of the same weight). This reduction also shows that

Max-Dicut is NP-hard. Additionally, it is known that Max-Dicut is APX-hard, even when restricted to *directed acyclic graphs (DAGs)* [16]. Thus (assuming P $\neq$ NP), we cannot hope for more than a constant factor approximation algorithm.

One of the many practical applications of approximate Max-Dicut is as a subroutine in grammar-based text compression [12], where we encounter large graphs that may contain millions of nodes. In this setting, we are particularly interested in algorithms that not only obtain a good approximation ratio, but also utilize shared memory parallelism to accelerate the computation. To the best of our knowledge, there currently exists no such algorithm.

1.1 Related Work

Computing an *undirected* Max-Cut is a well-studied problem in theory as well as in practice. Among the practical solutions are a variety of exact Max-Cut solvers like BiqMac [20] and BiqCrunch [15]. However, these solvers do not use parallelism to accelerate the computation. Recently, new exact parallel solvers were introduced, e.g., BiqBin [9] and MADAM [11]. Unfortunately, these solvers cannot easily be modified to compute a *directed* maximum cut instead. They are also only practical for relatively small graphs.

There is a variety of sequential Max-Dicut approximation algorithms. A naive randomized algorithm assigns each node independently with probability $\frac{1}{2}$ either to S or T, which results in a cut with an expected performance guarantee of $\frac{1}{4}$. (As usual, we say that an algorithm has *performance guarantee* $\alpha \in (0, 1)$ if it always computes a cut of value at least $\alpha \cdot C_{max}$, where C_{max} is the value of a maximum cut.) By using the method of *conditional expectations* [22, 19], we can derandomize this algorithm. A linear time algorithm for unweighted graphs with a performance guarantee of $\frac{9}{20}$ was described in [10]. In [6], a set of algorithms with a performance guarantee in $[0.25, 0.5)$ was described. In these algorithms, the decision whether a node is assigned to S or T only depends on the in-degree and out-degree of the node. Max-Dicut can be seen as a maximization of a *submodular function*. In [4], a linear time algorithm with an expected performance guarantee of $\frac{1}{2}$ and one with deterministic ratio $\frac{1}{3}$ were described, both using maximization of a submodular function as their main ingredient. Max-Dicut was also considered in an online model [2] and shown to have an online algorithm with a performance guarantee of $\frac{1}{3}$.

By solving a relaxation of an integer linear program (ILP) and using a simple rounding scheme, we can achieve an expected performance guarantee of $\frac{1}{2}$ [18]. The currently best known performance guarantee uses an idea that was first described by Goemans and Williamson [7]. It relaxes an integer program into a *semidefinite program*, and then uses an interesting rounding technique to achieve a performance guarantee of 0.79607. The performance guarantee was subsequently improved to 0.859 [23] and 0.874 [17]. It has been shown that it can only be improved up to 0.878 in case that the *Unique Games Conjecture* [14] is true. These algorithms have a polynomial running time and do not perform well in practice. Also, they cannot easily be parallelized.

Summing up, when faced with graphs of nonhomeopathic size, the only practical option to date is to use one of the (randomized) linear time algorithms [4, 6], resulting possibly in poor quality directed cuts. Better performing algorithms such as [7] cannot be applied to dense graphs of more than a few thousand nodes.

1.2 Our Contributions

To bridge the gap between the linear time algorithms and the expensive ILP-based solutions, we propose a practical framework that computes high quality Max-Dicut approximations in well partitionable graphs using shared memory parallelism. In recent years, practical

improvements for a variety of graph problems were achieved by using graph partitioning [5, 1]. The general approach is to partition a graph into multiple subgraphs (usually aiming at minimizing the number of edges between the subgraphs), solve the problem locally on each subgraph, and then merge the local solutions into a global solution. Since we can independently compute a solution for each subgraph, this approach is well suited to be parallelized in shared (and potentially also distributed) memory.

This is also the idea behind our framework, which consists of three main algorithmic components and an optional post-processing step:

S1 A *graph partitioner* is used to split the input graph G into $k \ll n$ subgraphs G_i of roughly equal size, such that the dependency between the subgraphs is small, i.e. the sum of the edge-weights between the subgraphs is small.

S2 An *approximation algorithm for* Max-Dicut is used to compute a cut S_i, T_i for each subgraph G_i (processing up to p subgraphs simultaneously, using p processors).

S3 A *merger* is used to integrate all the local solutions S_i, T_i into a global cut. This is achieved by defining a new contracted graph with $2k$ nodes, where each node represents a partition set S_i or T_i. Merging the local cuts then corresponds to computing a Max-Dicut in the contracted graph. Thus, the merger is just another Max-Dicut algorithm, which can either be a high-quality approximation algorithm or even an exact solver, if k is sufficiently small.

S4 (optional) A *local search* is performed in order to improve the current solution by swapping nodes between the partitions until we cannot further increase the value of the cut.

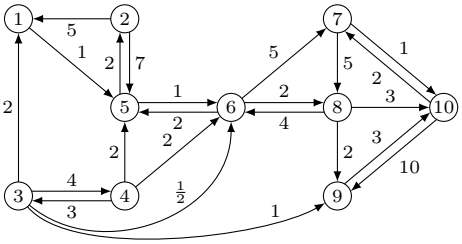
The C++ implementation of the framework is publicly available on GitHub (see supplementary material on the title page). At the time of writing, the framework provides three different partitioners, four sequential Max-Dicut approximation algorithms, and four mergers. It has been designed with extendability in mind, such that it should be little effort for the user to add new algorithms.

We evaluated the different options for each component on various real world graphs of up to 2^{23} nodes and 2^{24} edges. If we choose the right components, then the framework scales well in practice, while computing high quality cuts. The quality of the cut depends heavily on how well the graph can be partitioned in the first step. For graphs that can be partitioned well, we obtain a cut that is significantly better than the cut achieved by linear time algorithms. This is particularly relevant for large graphs, where linear time algorithms used to be the only feasible option.

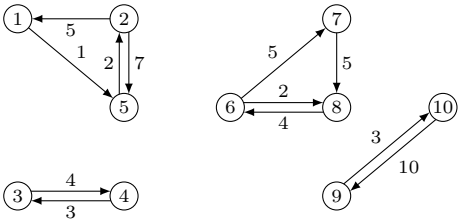
The remainder of the paper is structured as follows. First, we describe the main steps of the framework in more detail, providing examples of each step (Section 2). Then, we give implementation details, focusing on the actual algorithms that we use as components of the framework (Section 3.1). Finally, we provide practical results (Section 3.2), and discuss future work and open problems (Section 4).

2 Framework

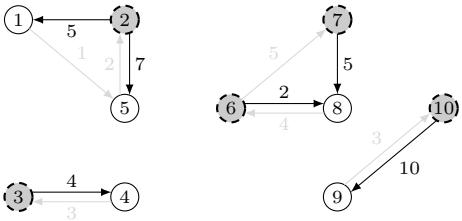
In this section, we describe the framework in more detail. The description is accompanied by a full example in Figure 1.



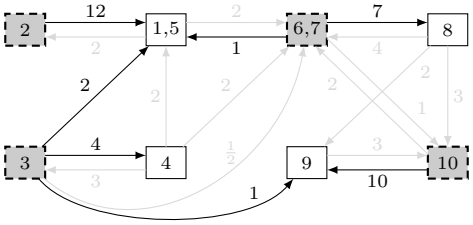
(a) The directed and weighted input graph G for which we want to compute a maximum directed cut.



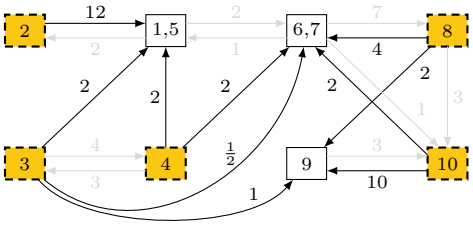
(b) Step 1: We partition the input graph into subgraphs G_1, G_2, G_3, G_4 induced by the node sets $V_1 = \{1, 2, 5\}$, $V_2 = \{6, 7, 8\}$, $V_3 = \{3, 4\}$ and $V_4 = \{9, 10\}$. The partition aims to minimize the sum of edge weights between subgraphs.



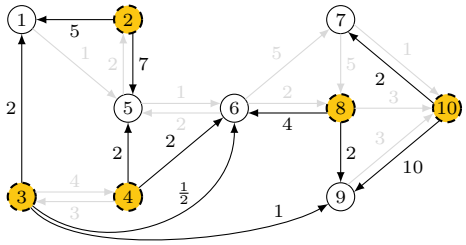
(c) Step 2: We compute a directed cut S_i, T_i on each G_i. The source nodes, i.e., the elements of any S_i, are filled and dashed.



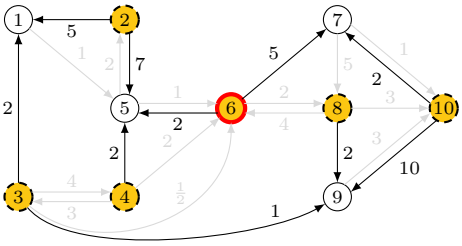
(d) Step 3.1: We obtain a contracted graph H from G by contracting each S_i and each T_i into a single node. The number of nodes is twice the number of subgraphs. The value of the naive cut $S = \bigcup S_i$, $T = V \setminus S$ is 37.



(e) Step 3.2: We compute an exact Max-Dicut S_H, T_H on H. Nodes from S_H are filled and dashed. The value of the cut is 37.5.



(f) Step 3.3: We obtain the final cut S, T on G by defining $S = \bigcup_{X \in S_H} X$ and $T = V \setminus S$. The unchanged value of the cut is 37.5.



(g) Step 4 (optional): We perform the local search and repeatedly identify nodes that have a positive gain, i.e., nodes that can be moved to the other partition set such that the overall cut improves. After Step 3.3, the only node with positive gain is node 6 with gain$(6) = (5 + 2) - (2 + \frac{1}{2} + 4) = \frac{1}{2}$. After moving node 6 from T to S (left part of the drawing), the cut has value 38. Now the only node with positive gain is node 4 with with gain$(4) = 4 - 2 = 2$. After moving it from S to T, (right part of the drawing), there is no node with positive gain, and the value of the cut is 40.

Figure 1 Running our framework on a small example.

2.1 Graph Partitioning

The first step of our framework is to partition the input graph $G = (V, E, w)$ into k disjoint subgraphs $G_1, \dots, G_k$ (where k is a freely choosable parameter), such that we can run on each subgraph independently an algorithm for approximating Max-Dicut. In order to improve the quality of the computed cut, we want to retain as much information as possible in each subgraph, while losing as little information as possible *between* the subgraphs. This can be achieved by maximizing the sum of the edge-weights in each subgraph, or, vice versa, minimizing the sum of the edge-weights between the subgraphs, i.e. we want to minimize $\sum_{i,j \in \{1,\dots,k\}, i \neq j} E_{ij}$, where E_{ij} is the sum of the edge-weights between subgraph G_i and G_j.

We use partitioning algorithms that split G into subgraphs $G_i = (V_i, E_i, w)$ of roughly equal size, i.e. we allow for a multiplicative error $\epsilon > 0$ such that $|V_i| \leq (1 + \epsilon) \left\lceil \frac{|V|}{k} \right\rceil$. Although graph partitioning is a hard problem itself, there are high-quality parallel graph partitioners that scale well on multiple processors [1, 8]. In Figures 1a and 1b, we see an example of an input graph partitioned into four subgraphs.

2.2 Compute Local Solutions

In the next step, we run on each subgraph G_i an algorithm that computes an approximation S_i, T_i for Max-Dicut. We process up to p subgraphs at the same time, where $p \leq k$ is the number of processors that we want to use. The choice of the approximation algorithm depends on the available time and the size of the subgraphs. If the subgraph are rather small, e.g., 100 nodes each, then we may be able to compute an optimal solution for each of them. If the subgraphs are slightly bigger, e.g., around 1000 nodes each, then a superlinear approximation algorithm with a relatively high performance guarantee might be feasible. If the subgraphs are even bigger, then we may have to use a simple linear time approximation algorithm with a lower performance guarantee. Depending on our choice of k, we have the flexibility to choose the algorithm such that we achieve the best trade-off between the quality of the cut and the runtime of the framework. As an example, we show in Figure 1c the optimal Max-Dicut for all subgraphs that were computed in Figure 1b.

2.3 Merging

In a final step, we have to merge the computed local cuts into a global cut. A naive approach is to define $S = \bigcup_i S_i$ and $T = \bigcup_i T_i$ as the trivial cut. The problem with this approach is that we did not consider the edges between the subgraphs. For some $i \in \{1, \dots, k\}$, it might be more advantageous to swap the subsets S_i and T_i in the global graph, or even to put S_i and T_i into the same partition. To consider the dependencies between the subgraphs, we reduce the problem of merging the local solutions to another Max-Dicut instance. We build a complete graph H with $2k$ nodes, where each node represents a locally computed partition set S_i or T_i. For every pair X, Y of nodes in H, we add an edge (X, Y) to H with weight $\sum_{i \in X, j \in Y} w(i, j)$ (see Figure 1d). Since the graph H has only $2k$ nodes, we can use an expensive algorithm to compute an exact Max-Dicut defined by S_H and T_H (see Figure 1e). Finally, the global cut is defined by $S = \bigcup_{X \in S_h} X$ and $T = \bigcup_{X \in T_h} X$ (see Figure 1f).

2.4 Optimization by Local Search

To further optimize the computed cut, we use in an optional fourth step, a *local search*. The idea of the local search is to check if we can improve the cut by swapping the partitioning of a node. We repeatedly swap the partitioning of nodes to improve the cut until we can no longer swap any nodes.

More precisely, for a node u we calculate the *gain* for swapping the partitioning of u. If $u \in S$, then we calculate the gain by $\text{gain}(u) = \sum_{(v,u)\in E, v\in S} w(v,u) - \sum_{(u,v)\in E, v\in T} w(u,v)$, and if $u \in T$ by $\text{gain}(u) = \sum_{(u,v)\in E, v\in T} w(u,v) - \sum_{(v,u)\in E, v\in S} w(v,u)$. We repeatedly find a node u with $\text{gain}(u) > 0$ and swap its partitioning, until for all nodes u we have $\text{gain}(u) \leq 0$. In this case, we have arrived at a local maximum. An example is provided in Figure 1g.

There are multiple ways of choosing a node u with $\text{gain}(u) > 0$. We use a simple *first candidate* strategy, which chooses the first node u with $\text{gain}(u) > 0$ that it encounters. We also implemented a *best candidate* strategy, which chooses the node u with maximal gain. However, in practice taking the best candidate rather than the first candidate improves the cut only by a small margin, while it significantly increases the running time.

We point out that this step is hard to parallelize, as multiple iterations of the local search are dependent of each other.

3 Experimental Evaluation

In this section, we present practical results of our framework. We implemented the framework in C++17 and used OpenMP for parallelization.

We evaluate our framework on the input graphs as summarized in Table 1. The graphs `great-britain`, `luxembourg`, `flixster`, `flickr`, `Stanford3`, and `web-sk-2005` were taken from *Network Repository* [21]. The graphs `luxembourg` and `great-britain` are road networks of Luxembourg and Great Britain, respectively. The graph `flixster` is a graph which represents all links between users of the website Flixster and the graph `flickr` is a graph where an edge represents if one user added another user as a contact on the website Flickr. The graph `Stanford3` is a Facebook graph with all users from Stanford University and the graph `web-sk-2005` is a crawl from 2005 for the .sk domain by using UbiCrawler [3]. For a comparison of our results with the optimal cut, we also took 121 small graphs from Network Repository on which we can compute the optimal cut in the time limit of one hour by doing three runs with different random seeds each (see Appendix A for a list).

To specify how well we can partition a weighted graph $G = (V, E, w)$, we additionally provide the *coverage* measure of a weighted graph for a partitioning $P = \{G_1, ..., G_k\}$. Intuitively, the coverage describes how many edges are covered by the partitioning. It is defined as $\text{cov}(G, P) = \frac{W(P)}{W}$, where $W(P) = \sum_{G_i=(V_i,E_i,w)\in P} \sum_{e\in E_i} w(e)$ and W is simply the sum of all edge-weights in G. The more the coverage of a graph tends towards 1, the less influence the merging step of our framework has on the computed cut. In case that G is clear from the context, we omit it in the notation. When we want to emphasize that there exists a partitioning P into k subgraphs with $\text{cov}(P) = x$, we also write $\text{cov}(k) = x$ (so x is a lower bound on the coverage of a graph with a partitioning into k subgraphs).

3.1 Implementation Details

For partitioning the input graph, we use the shared memory algorithm KaMinPar [8], which aims at minimizing the number of edges between subgraphs. To use this algorithm, we first have to transform our directed graph into an undirected graph by constructing a new graph $G' = (V, E', w')$ where $E' = \{(u,v) \mid (u,v) \in E \vee (v,u) \in E\}$ and $w'(u,v) = w(u,v) + w(v,u)$. We configured KaMinPar with an imbalance value of 0.01 and use a random seed for each computation.[1]

[1] We also tried two naive partitioning algorithms: simply slicing the list of nodes ore edges into equal-size parts, but, despite being fast, these turned out to produce cuts of inferior quality.

Table 1 A summary of our large input graphs. We provide for each graph the number of nodes n, the number of edges m and the coverage with four different partitionings. We partitioned the graph using KaMinPar into $k \in \{64, 256, 1024, 4096\}$ subgraphs and calculated for each the coverage measurement $\text{cov}(k)$ (in percent).

Graph	n	m	cov(64)	cov(256)	cov(1024)	cov(4096)
great-britain	7,733,822	16,313,034	99.96	99.92	99.82	99.58
flixster	2,523,386	15,837,602	55.26	49.26	44.41	39.83
flickr	513,969	6,380,904	69.78	53.68	37.14	27.50
luxembourg	114,599	239,332	99.62	99.10	97.64	92.83
web-sk-2005	121,422	668,838	99.59	98.45	93.98	77.07
Stanford3	11,586	1,136,618	27.76	15.56	6.20	1.69

We implemented several sequential algorithms that approximate Max-Dicut. As discussed earlier, they can be used both for computing the local solutions, as well as for merging them into the global solution. Our most naive implementations are the randomized algorithm that puts every node independently with probability $\frac{1}{2}$ either in S or T, and its derandomized counterpart. We call these $\frac{1}{4}$-approximation algorithms Random and Derandomized respectively. Additionally, we implemented the algorithm by Buchbinder et al. [4] with performance guarantee $\frac{1}{3}$, which we call Buchbinder. We also implemented the algorithm by Goemans and Williamson [7] with expected performance guarantee 0.79607, which we call Goemans. For solving semidefinite programs (which is needed as a subroutine of Goemans), we use the MOSEK Fusion API[2] with enabled parallelism. Lastly, we also implemented an algorithm that computes the exact Max-Dicut. This algorithm uses an *Integer Linear Program (ILP)* representation of Max-Dicut which was described in [10] and solves it using Gurobi[3] using only a single thread. We call this algorithm ILP.

To denote which of the sequential Max-Dicut algorithms we use in step 2, we use notations like Buchbinder_{S2}, meaning that we used the algorithm Buchbinder for computing the local solutions. (The merging step might still use other algorithms; see the following paragraph.)

When k gets large, the complete graph H as defined in Section 2.3 can get very large. For example, if we set k to 2048, then H has 4096 nodes and up to 16 million edges. In this case, the approximation algorithms with high performance guarantee are not practical anymore and the time to construct H can get too slow. To deal with this problem, we also implemented a tree-like merging algorithm. More precisely, we first divide the subgraphs $G_1, ..., G_k$ computed by the graph partitioner (with their local cuts) into $\frac{k}{l}$ groups of l subgraphs each (for some fixed $l < k$) where group i consists of the graphs $G_{(i-1)l+1}, ..., G_{il}$. Then, for each group we merge its subgraphs using the merge algorithm described in Section 2.3, resulting in $\frac{k}{l}$ subgraphs G'_i, each with a directed cut. We repeat this process with this new partitioning until the number of partitions gets smaller than l, where a final merging step produces the ultimate cut of G. We parallelized this algorithm with the exception of the Max-Dicut algorithm used to merge the directed cuts.

We denote this merge algorithm as Merge-Tree. In our experiments we use Merge-Tree with Goemans and $l = 256$ as our default configuration.

[2] https://www.mosek.com/
[3] https://www.gurobi.com/

3.2 Results

We now present experimental results on the running times and quality of our new algorithms in comparison with other existing approaches. When running an algorithm on a particular instance, we run it three times and for each execution we stop its execution after a time limit of 10 hours. Then, we take the average of the computed cuts and running times. This reflects the main research question that motivated our research: *Which quality can we achieve for directed cuts under given time and hardware constraints?*

We conducted our experiments on a Linux machine running Ubuntu version 18.04.6 with two AMD EPYC 7452 processors with 32 physical cores each and 1 TB of RAM. Thus when running the framework, we can compute the local cuts of up to $p = 64$ subgraphs at the same time. The code was compiled using GCC 7.5.0 with flag `-O3` enabled using Gurobi version 9.1.2 and Mosek version 9.3.

3.2.1 Overview

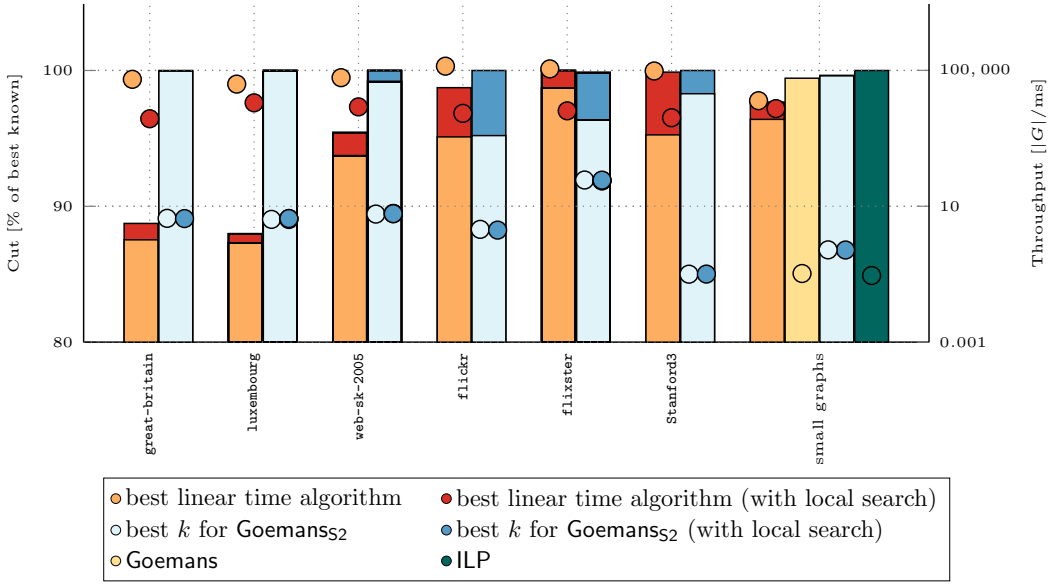


Figure 2 Overview of our results. For each algorithm, we visualize the quality of the cuts by a bar plot, and the throughput (size of the graph divided by the running time) by circles. The graphs from Table 1 are sorted in descending order by the value cov(64) to visualize the effect of the coverage on the computed cut quality. (Best viewed in color.)

Figure 2 provides an overview of what can be achieved with our framework. As we will see in Section 3.2.3 that the best performing configuration of our framework is to use the Goemans algorithm in step 2 (be it for varying k), we compare this configuration with other existing algorithms.

First look at the bar plots for the large graphs from Table 1, where currently the only feasible option is to use a linear time algorithm (Buchbinder, Derandomized, Random): we can observe that our new algorithms compute significantly better cuts than the best linear time algorithm on well partionable graphs which is indicated by their coverage. This is especially the case for `great-britain`, `luxembourg`, and `web-sk-2005`. Also, the computed cuts are

never worse for the other graphs (`flickr`, `flixster`, and `Stanford3`). We can also see that the final local search step (Section 2.4) pays off for all algorithms, in particular for the graphs that are not particularly well partitionable.

All this comes, of course, at the price of a lower throughput (cf. the circles in Figure 2), by several orders of magnitude. Nonetheless, we can conclude that within the given time and hardware constraints our new algorithms compute often better (and never worse) directed cuts than all previous existing approaches.

The final 4 bars show the cut quality of the small graphs in relation to the best possible cut. We calculated for each small graph the percentage and provide an (unweighted) average over the percentages of all small graphs since these values are independent of a graph. In addition to the linear time algorithms and our new parallel ones we could also run the Goemans algorithm on the entire graph, and the exact solution with ILP for each small graph. We see that our new algorithms again give better results than the best linear time algorithm with a final local search, and also slightly better results than Goemans. However, the difference between Goemans and our algorithms is not very large, which might be due to the fact that both of them are already very close to the exact solution (see final bar). On the positive side, our algorithms have a higher throughput than Goemans for the small graphs.

3.2.2 Scaling on Multiple Processors

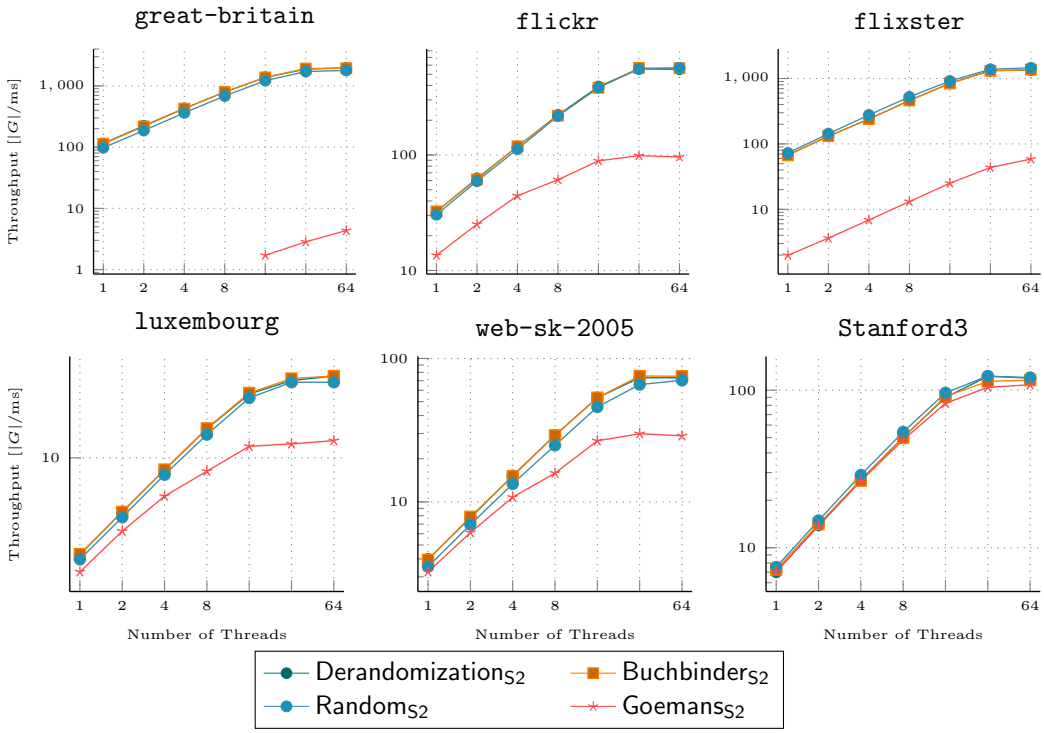


Figure 3 Scaling experiments on all graphs in Table 1. In all experiments, we partition the graph into 8192 subgraphs and run our framework with up to 64 threads. For this experiments we compare the algorithms Derandomized$_{S2}$, Buchbinder$_{S2}$, Random$_{S2}$ or Goemans$_{S2}$. On the x-axis we show the number of used threads and on the y-axis we show the throughput (size of graph divided by running time) of our framework.

Since our algorithms employ shared memory parallelism, we now briefly evaluate their scalability. Figure 3 shows the throughput for different choices of the approximation algorithms for the local solutions ($\mathsf{Derandomized}_{S2}$, $\mathsf{Buchbinder}_{S2}$, Random_{S2} or $\mathsf{Goemans}_{S2}$). For a better comparison, in all experiments we partition the graph into 8192 subgraphs and run our framework with up to 64 threads. We chose the value 8192 because our framework calculates for that value in all experiments a solution before the time limit is exceeded. We observe almost perfect scalability for up to 32 threads, with only slightly less good results for 32 threads, and significantly worse results for 64 threads. There are two possible explanations for the non-optimal scaling with a large number of threads. First, the used Max-Dicut algorithm in the final merging step is always performed sequentially. If we have many threads available, then the computation of the partition and the local solutions becomes faster, and thus the sequential merging step becomes more relevant for the total execution time. Second, our test system has two CPUs with 32 cores each, where each CPU is part of a separate NUMA node. When using close to or even more than 32 threads, the system will use cores from both CPUs. Some cores will therefore inevitably access memory outside their local NUMA node, which is slower than accessing local memory[4].

Note that for `great-britain` (the largest graph that we considered) we require at least 16 threads in order to use Goemans. With fewer threads, the framework will not finish within the given time limit.

3.2.3 Number of Subgraphs vs. Quality

Finally, we evaluate the influence of the number k of subgraphs on the quality of the computed dicut. This will also allow us to draw conclusions on how the framework should be best configured when running on different graphs. As we have seen already, the simple local search sometimes significantly improves the quality of the computed cut. For the following experiments, we therefore omit the local search. This allows us to see the true effect of the choice of k on the cut quality. Also, we always use 64 threads in order to get the best performance that is possible with our test system.

Figure 4 shows plots for both the quality and the achieved throughput for k between 64 and 8192. For $\mathsf{Derandomized}_{S2}$, Random_{S2}, and $\mathsf{Buchbinder}_{S2}$, the throughput decreases for increasing k. This is due to the fact that we use the superlinear Goemans algorithm for merging, and the graph used for merging has $2k$ nodes. On the other hand, the throughput increases with k when running $\mathsf{Goemans}_{S2}$. The reason for this is that a large value of k implies smaller subgraphs, for which the local solutions can then be computed faster. As seen for the graphs `flickr`, `luxembourg` and `web-sk-2005`, there is a range of k for which the throughput of $\mathsf{Goemans}_{S2}$ is in an equilibrium; increasing k within this range seems to equally accelerate the local solutions and decelerate the merging into a global solution. However, once k becomes too large, the throughput of $\mathsf{Goemans}_{S2}$ appears to decrease with k. We could only observe this for our smallest graph `Stanford3`.

Now let us focus on the quality of the cut. For $\mathsf{Goemans}_{S2}$, the quality generally decreases for larger values of k. This can be seen for all graphs except for `great-britain`, where $k = 8192$ is the only configuration that finished in time, and for `Stanford3`, where the cut quality first decreases, and then increases again once k exceeds 1024. This anomaly is likely due to the fact that `Stanford3` only has around 10000 nodes, and for large k the

[4] For more information on NUMA architectures see `https://uefi.org/specs/ACPI/6.4/17_NUMA_Architecture_Platforms/NUMA_Architecture_Platforms.html`

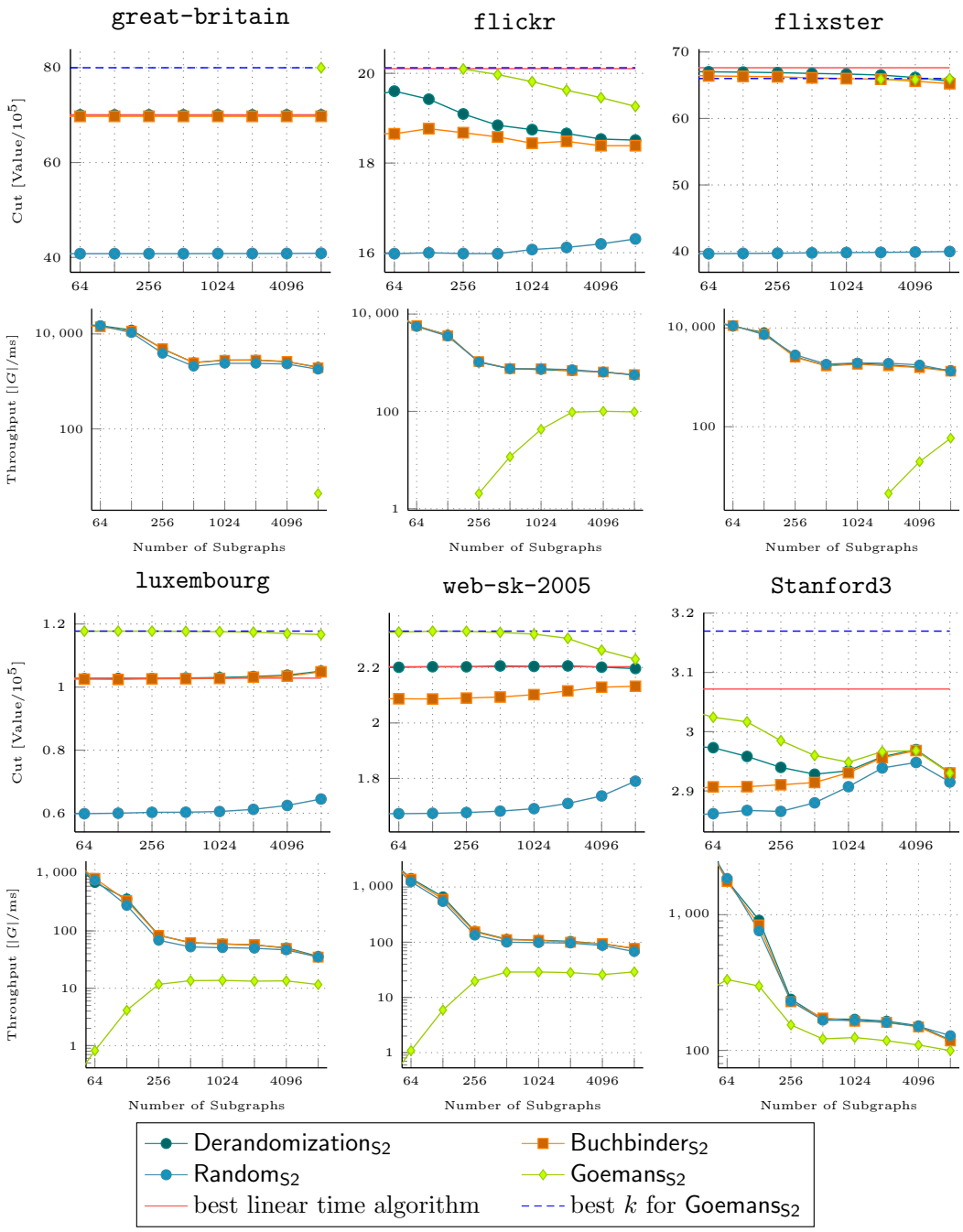


Figure 4 Scaling experiments for our framework where we visualize the effect of partitioning different graphs into a varying number of subgraphs on the computed cut quality and the running time of our framework. We partition our graph into between 64 to 8192 subgraphs while the number of used threads is fixed to 64. For this experiments we compare the algorithms Derandomized$_{S2}$, Buchbinder$_{S2}$, Random$_{S2}$ or Goemans$_{S2}$. In the plot above we see the computed cut and in the plot below the throughput (size of graph divided by running time) of our framework for each configuration. We additionally provide the cut quality of the best linear time algorithm (as continuous red line) that does not use our framework and the best computed cut for some $k \in \{2, 4, 8, ..., 8192\}$ by using Goemans$_{S2}$ in our framework as dashed blue line.

computation becomes more and more similar to a simple sequential execution of Goemans on the entire graph. The cuts computed by Derandomized$_{S2}$, Random$_{S2}$ and Buchbinder$_{S2}$ are generally worse than the one computed by Goemans$_{S2}$. The overall order of quality is (from worst to best): Random$_{S2}$, Buchbinder$_{S2}$, Derandomized$_{S2}$, Goemans$_{S2}$. The only exception is `flixster`, where the cut quality of most algorithms is almost the same, and only Random$_{S2}$ performs very poorly. With increasing k, the cut quality of Derandomized$_{S2}$, Random$_{S2}$ and Buchbinder$_{S2}$ on the smaller graphs (`luxembourg`, `web-sk-2005`, `Stanford3`) increases. This is because for large k, a significant fraction of the total edge weights is between the subgraphs. Thus, the simple linear time algorithms used for the local solutions loose relevance, while the better Goemans algorithm used for merging becomes more relevant. However, the quality does not reach the one of Goemans$_{S2}$.

We conclude that Goemans$_{S2}$ produces the best cuts, and that the best choice of k is the smallest k that allows the framework to finish within the given time limit. Increasing k further than that will only decrease the cut quality, and not even necessarily accelerate the computation.

4 Conclusion

We described a parallel framework for computing an approximate Max-Dicut that scales well for large graphs and produces high quality cuts for graphs with high coverage. It is also extendable, such that it is easy to add new algorithms.

The experiments showed that the best quality cuts are obtained by dividing the graphs into as few partitions as the Goemans algorithm finishes within the time limit on each of the partitions. While the sizes of resulting partitions can be used as a first indicator for choosing the right number of partitions, it remains an open problem to find an exact predictor for this. Also, our algorithms only sensibly scale for $\approx \sqrt{n}$ processors, as with more processors the sequential merging part becomes too expensive. Maybe a recursive (multilevel) approach can be used to fill this gap.

References

1 Y. Akhremtsev, P. Sanders, and C. Schulz. High-Quality Shared-Memory Graph Partitioning. *IEEE Transactions on Parallel and Distributed Systems*, 31(11):2710–2722, 2020. doi:10.1109/TPDS.2020.3001645.

2 Amotz Bar-Noy and Michael Lampis. Online maximum directed cut. *Journal of combinatorial optimization*, 24(1):52–64, 2012. doi:10.1007/s10878-010-9318-6.

3 Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. UbiCrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004. doi:10.1002/spe.587.

4 N. Buchbinder, M. Feldman, J. Naor, and R. Schwartz. A Tight Linear Time (1/2)-Approximation for Unconstrained Submodular Maximization. In *2012 IEEE 53rd Annual Symposium on Foundations of Computer Science*, pages 649–658, 2012. doi:10.1137/130929205.

5 A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. *Recent Advances in Graph Partitioning*, pages 117–158. Springer International Publishing, Cham, 2016. doi:10.1007/978-3-319-49487-6_4.

6 U. Feige and S. Jozeph. Oblivious Algorithms for the Maximum Directed Cut Problem. *Algorithmica*, 71(2):409–428, February 2015. doi:10.1007/s00453-013-9806-z.

7 M. X. Goemans and D. P. Williamson. Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming. *J. ACM*, 42(6):1115–1145, 1995. doi:10.1145/227683.227684.

8 Lars Gottesbüren, Tobias Heuer, Peter Sanders, Christian Schulz, and Daniel Seemaier. Deep Multilevel Graph Partitioning. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms (ESA 2021)*, volume 204 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 48:1–48:17, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2021.48.

9 N. Gusmeroli, T. Hrga, B. Luvzar, J. Povh, M. Siebenhofer, and A. Wiegele. BiqBin: a parallel branch-and-bound solver for binary quadratic problems with linear constraints. *arXiv: Optimization and Control*, 2020. doi:10.48550/arxiv.2009.06240.

10 E. Halperin and U. Zwick. Combinatorial Approximation Algorithms for the Maximum Directed Cut Problem. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '01, pages 1–7, USA, 2001. Society for Industrial and Applied Mathematics.

11 T. Hrga and J. Povh. MADAM: A parallel exact solver for Max-Cut based on semidefinite programming and ADMM. *Computational Optimization and Applications*, 80:347–375, 2021. doi:10.1007/s10589-021-00310-6.

12 A. Jez. Recompression: A Simple and Powerful Technique for Word Equations. *J. ACM*, 63(4):1–51, 2016. doi:10.1145/2743014.

13 R. M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, 1972. doi:10.1007/978-1-4684-2001-2_9.

14 S. Khot, G. Kindler, E. Mossel, and R. O'Donnell. Optimal Inapproximability Results for MAX-CUT and Other 2-Variable CSPs? *SIAM J. Comput.*, 37(1):319–357, 2007. doi:10.1137/S0097539705447372.

15 N. Krislock, J. Malick, and F. Roupin. BiqCrunch: A Semidefinite Branch-and-Bound Method for Solving Binary Quadratic Problems. *ACM Trans. Math. Softw.*, 43(32):1–23, 2017. doi:10.1145/3005345.

16 M. Lampis, G. Kaouri, and V. Mitsou. On the Algorithmic Effectiveness of Digraph Decompositions and Complexity Measures. In Seok-Hee Hong, Hiroshi Nagamochi, and Takuro Fukunaga, editors, *Algorithms and Computation*, pages 220–231, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-92182-0_22.

17 M. Lewin, D. Livnat, and U. Zwick. Improved Rounding Techniques for the MAX 2-SAT and MAX DI-CUT Problems. In W. J. Cook and A. S. Schulz, editors, *Integer Programming and Combinatorial Optimization*, pages 67–82, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. doi:10.1007/3-540-47867-1_6.

18 Michael Luby and Noam Nisan. A parallel approximation algorithm for positive linear programming. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 448–457, 1993. doi:10.1145/167088.167211.

19 P. Raghavan. Probabilistic construction of deterministic algorithms: Approximating packing integer programs. *Journal of Computer and System Sciences*, 37(2):130–143, 1988. doi:10.1016/0022-0000(88)90003-7.

20 F. Rendl, G. Rinaldi, and A. Wiegele. Solving Max-Cut to optimality by intersecting semidefinite and polyhedral relaxations. *Mathematical Programming*, 121:307–335, 2010. doi:10.1007/s10107-008-0235-8.

21 R. A. Rossi and N. K. Ahmed. The Network Data Repository with Interactive Graph Analytics and Visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015. URL: http://networkrepository.com.

22 J. Spencer. *Ten Lectures on the Probabilistic Method*, volume CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, 2nd edition, 1994. doi:10.1137/1.9781611970074.

23 U. Zwick. Analyzing the MAX 2-SAT and MAX DI-CUT approximation algorithms of Feige and Goemans, 2000. manuscript.

A Small Graphs

Table 2 Our used small graphs.

Graph	n	m
08blocks.mtx	300	592
3elt.mtx	4,720	27,444
BA-1_10_60-L5.edges	805	46,410
BZR.edges	14,480	31,070
COIL-RAG.edges	11,758	23,588
CSphd.mtx	1,882	1,740
California.mtx	9,664	16,150
DD199.edges	842	1,902
DD21.edges	5,749	14,267
DD242.edges	1,285	3,303
DD244.edges	292	822
DD349.edges	898	2,087
DD497.edges	904	2,453
DD6.edges	4,153	10,320
DD68.edges	776	2,093
DD687.edges	726	2,600
D_11.mtx	461	2,952
ENZYMES118.edges	97	121
ENZYMES123.edges	91	127
ENZYMES295.edges	125	139
ENZYMES296.edges	127	141
ENZYMES297.edges	123	149
ENZYMES8.edges	89	133
EPA.mtx	4,772	8,965
EVA.mtx	8,497	6,726
Franz3.mtx	2,800	11,520
G11.mtx	800	3,200
G12.mtx	800	3,200
G13.mtx	800	3,200
G32.mtx	2,000	8,000
G33.mtx	2,000	8,000
G34.mtx	2,000	8,000
G48.mtx	3,000	12,000
G49.mtx	3,000	12,000
G57.mtx	5,000	20,000
G62.mtx	7,000	28,000
G65.mtx	8,000	32,000
G66.mtx	9,000	36,000
G67.mtx	10,000	40,000
Letter-high.edges	10,508	20,250
Letter-low.edges	10,523	14,092
Letter-med.edges	10,519	14,426

Graph	n	m
MSRC-21C.edges	8,419	40,380
OHSU.edges	6,480	31,546
PTC-FM.edges	4,926	10,110
PTC-FR.edges	5,111	10,532
PTC-MM.edges	4,696	9,624
PTC-MR.edges	4,916	10,108
Peking-1.edges	3,342	13,150
SW-10000-6-0d3-L2.edges	10,001	30,000
SW-10000-6-0d3-L5.edges	10,001	30,000
TerroristRel.edges	882	8,592
aves-sparrow-social.edges	53	516
aves-weaver-social.edges	446	1,426
aves-wildbird-network.edges	203	11,900
bio-CE-GN.edges	2,220	53,683
bio-CE-GT.edges	924	3,239
bio-CE-HT.edges	2,617	2,985
bio-CE-LC.edges	1,387	1,648
bio-CE-PG.edges	1,871	47,754
bio-DM-HT.edges	2,989	4,660
bio-DM-LC.edges	658	1,129
bio-HS-HT.edges	2,570	13,691
bio-HS-LC.edges	4,227	39,484
bio-SC-CC.edges	2,223	34,879
bio-SC-GT.edges	1,716	33,987
bio-SC-LC.edges	2,004	20,452
bio-SC-TS.edges	636	3,959
bio-celegans.mtx	453	4,050
bio-celegansneural.mtx	297	2,345
bio-diseasome.mtx	516	2,376
bio-grid-fission-yeast.edges	2,031	25,274
bio-grid-mouse.edges	1,455	3,272
bio-grid-plant.edges	1,745	6,196
bio-grid-worm.edges	3,518	13,062
bio-yeast.mtx	1,458	3,896
bn-mouse-kasthuri_graph_v4.edges	1,029	1,700
c-fat200-1.mtx	200	3,068
c-fat200-2.mtx	200	6,470
c-fat500-1.mtx	500	8,918
chesapeake.mtx	39	340
citeseer.edges	3,244	4,536
cora.edges	2,709	5,429
delaunay_n10.mtx	1,024	6,112
delaunay_n11.mtx	2,048	12,254
delaunay_n12.mtx	4,096	24,528

Table 3 Our used small graphs.

Graph	n	m
eco-florida.edges	129	2,106
eco-foodweb-baydry.edges	129	2,137
eco-foodweb-baywet.edges	129	2,106
eco-mangwet.edges	98	1,492
eco-stmarks.edges	55	356
email-dnc-corecipient.edges	2,030	12,085
email-dnc.edges	2,029	39,264
email-enron-only.mtx	143	1,246
email-univ.edges	1,134	5,451
fb-forum.edges	900	33,720
gene.edges	1,094	1,672
ia-contacts_hypertext2009.edges	114	20,818
ia-dnc-corecipient.edges	2,030	12,085
ia-hospital-ward-proximity-attr.edges	1,661	32,424
ia-hospital-ward-proximity.edges	1,661	32,424
ia-workplace-contacts.edges	876	9,827
inf-euroroad.edges	1,175	1,417
insecta-ant-trophallaxis-colony1.edges	42	308
insecta-ant-trophallaxis-colony2.edges	40	330
internet-industry-partnerships.edges	218	631
mammalia-primate-association.edges	26	1,340
mammalia-raccoon-proximity.edges	25	1,997
mammalia-voles-bhp-trapping.edges	1,687	5,324
mammalia-voles-kcs-trapping.edges	1,219	4,258
mammalia-voles-plj-trapping.edges	1,264	3,863
mammalia-voles-rob-trapping.edges	1,481	4,569
reptilia-tortoise-network-bsv.edges	137	554
reptilia-tortoise-network-cs.edges	74	258
reptilia-tortoise-network-fi.edges	788	1,713
reptilia-tortoise-network-hw.edges	17	22
reptilia-tortoise-network-lm.edges	46	134
reptilia-tortoise-network-mc.edges	16	45
reptilia-tortoise-network-pv.edges	36	104
reptilia-tortoise-network-sg.edges	25	29
reptilia-tortoise-network-sl.edges	12	18