Algorithm Engineering
WS 22/23

Cedric Wiese

Maximizing Graph Cuts: An Analysis of Exact, Approximation, and Heuristic Algorithms for Max Cut

Abstract

In this paper, we present a study of three different algorithms for the Maximum Cut problem: a heuristic algorithm, a two-approximation algorithm as well as an exact algorithm based on an ILP formulation. Furthermore, we explore the parallelization of the approximation algorithm in order to improve their practical performance and to exploit the power of modern parallel architectures. We evaluate the algorithms by using self generated, random graphs as benchmarks. These graphs are of various sizes and densities, providing a good variety of test sets. Our experimental results will demonstrate the strengths and weaknesses of each algorithm regarding exactness, efficiency and scalability for solving the Max Cut problem.

1 Introduction

Graph partitioning is a fundamental problem in computer science with numerous real world applications including, but not limited to, clustering, network analysis and data mining. The Maximum Cut (decision) problem is well known and, despite its simple formulation, known to be an NP-complete problem in graph partitioning, which aims to divide the vertices of an undirected graph into two disjoint sets, maximizing the number of edges between them. This problem is of great interest due to its wide range of applications, such as for example image segmentation, social network analysis and more.

2 Preliminaries

In this section we provide the preliminaries for the Max Cut problem. We define the problem, as well as its mathematical formulation.

Given an undirected graph $G(V, E)$ with $n = |V|$ nodes and $m = |E|$ edges, the Maximum Cut aims to partition the vertex set V into two disjoint sets S and T in such a way, that the number of edges with one vertex in S and one vertex in T is maximized.

Formally, the problem can be described as follows:

$$\text{maximize } |\{(u, v) \in E | u \in S \wedge v \in T\}|$$

2.1 Heuristic Algorithm

There are many possible ways to implement a heuristic algorithm (2.1) for Max Cut. We decided to use an algorithm, which sorts the nodes of a graph by degree in a descending order. The algorithm then partitions the vertices in change. Since ordering the vertices by degree takes us $\mathcal{O}(n^2)$, the heuristic algorithm has a worst case running time of $\mathcal{O}(n^2 + n) = \mathcal{O}(n^2)$.

Although this algorithm does not guarantee an approximation ratio and its result can be arbitrarily bad compared to the optimal solution, our experiments show that this algorithm can find reasonable results efficiently.

2.2 2-Approximation Algorithm

The Two-Approximation Algorithm (2.1) for Max Cut is a simple algorithm that provides a guaranteed approximation ratio of 2 for the Max Cut problem. The algorithm starts by partitioning the nodes of a graph into two groups at random. It then iteratively improves the partition by moving nodes between the groups in a way that increases the number of edges that cross the cut. This happens with the help of two For-Loops (one for each set), checking how the cut changes when moving one vertex from one set to another.

Algorithm 1 Heuristic Algorithm

```
1: procedure HEURISTIC(graph, size)
2:     result ← bool[]
3:     ordered ← nodes_ordered_by_degree()
4:     switch ← true
5:     for i ← 0 to ordered.len() do
6:         result[ordered[i].get_node()] ← switch
7:         switch ← !switch
8:     end for
9:     return(result, size)
10: end procedure
```

2.3 Solving Max Cut with Integer Linear Programming

Integer Linear Programming (ILP) is a mathematical optimization method that can be used to solve the various optimization problems, including Max Cut. In ILP, a problem is modeled as a linear program and solved using linear optimization techniques, with the added constraint that the solution variables must be integers.

To solve Max Cut using ILP, the problem is modeled as a linear program with variables created for edges as well as for vertices. Each edge can either have a value of 0 or 1, representing whether or not the edge is part of the cut or not. Regarding the vertices, each variable can have values of 0 or 1, representing the set in which the vertex belongs.

For each vertex v we create a variable x_v:

$$x_v = 0 \vee x_v = 1 \tag{1}$$

with: $x_v = 0 \rightarrow x_v \in S$ and $x_v = 1 \rightarrow x_v \in T$
For every edge e_{ij} we create a variable x_{ij}:

$$\{x_{ij} = 0\} \rightarrow \{i \in S \wedge j \in S \vee i \in T \wedge j \in T\} \tag{2}$$
$$\{x_{ij} = 1\} \rightarrow \{i \in S \wedge j \in T \vee i \in T \wedge j \in S\} \tag{3}$$

Algorithm 2 Approximation Algorithm

```
    procedure APPROXIMATION(graph, size)
2:      partition1 ← Vec¡int¿
        partition2 ← Vec¡int¿
4:      for i ← 0 to ⌊size/2⌋ do
            partition1.push(i)
6:      end for
        for i ← ⌊size/2 + 1⌋ to size do
8:          partition2.push(i)
        end for
10:     improved ← true
        while improved do
12:         improved ← false
            for i ← 0 to partition1.len() do
14:             if move partition1[i] to partition2 reduces cut then
                    partition2.add(partition1[i])
16:                 partition1.remove(i)
                    improved ← true
18:             end if
            end for
20:         for i ← 0 to partition2.len() do
                if move partition2[i] to partition1 reduces cut then
22:                 partition1.add(partition2[i])
                    partition2.remove(i)
24:                 improved ← true
                end if
26:         end for
        end while
28:     return get_cut(partition1, partition2)
    end procedure
```

Our objective is to maximize the number of edges between vertex i and j, so that $x_i \neq x_j$ (which means that the vertices are in different sets.

$$\text{Maximize:} \quad |\{(i,j)|x_i \neq x_j\}| \tag{4}$$

For each edge we need to add the following constraints:

$$\text{Constraint 1:} \quad x_{ij} \leq x_i + x_j \tag{5}$$
$$\text{Constraint 2:} \quad x_{ij} \leq 2 - x_i - x_j \tag{6}$$

Constraint 1 ensures that if an edge is cut, at least one of its endpoints is in the first set, while Constraint 2 ensures that if an edge is cut, not both endpoints can be in the first set. This results in a guarantee that both endpoints are in different sets.

3 Algorithm & Implementation

In this chapter, we describe modifications to the previously described alrgorithms, as well as our parallelization approach for the Two-Approximation Algorithm. We further discuss the graph datastructure we used and its effects on the implementation.

3.1 Graph Datastructure

There a numerous datastructures open for consideration when it comes to storing graphs, each with its own advantages and disadvantages. We decided to implement a modified adjacency matrix, which is stored in a one dimensional array. The size of the array can be calculated with

$$size = n^2 - \frac{n \cdot (n+1)}{2} \tag{7}$$

where n is the number of nodes in our graph. By using this datastructure, we only have entries for every edge e_{ij} where $i < j$. This way, we are able to save memory space, as the size of our matrix is not n^2.

3.2 (Parallel) Approximation

The Two-Approximation algorithm (2.1) can be parallelized efficiently. Since the iterations of every For-Loop are independent from each other, we can distribute the iterations over multiple cores, which can lead to a significant decrease with regarding the running time.

We do so by spawning multiple threads, where each thread covers a certain number of iterations, checking for each iteration, whether or not the cut is improved and if so, for how much the cut has increased. When a thread finishes, it compares its best swap of vertices to the current best swap of the main thread. If the child's swap leads to a bigger cut, the main thread takes over the child's swap. After every For-Loop, the main thread waits for each child to be finished. This results in the main thread having the best swap of all child threads and now being able to perform the swap.

Since memory has to be shared between the threads, we utilized Read-Write-Locks, in which every thread could acquire a read-lock to mutually read from the same memory, but only one thread is able to acquire a write-lock, to ensure exclusive writing to memory.

We can use this implementation both for parallel approximation as well as a simple approximation, by increasing the chunk size to the size of the graph. This leads to only one thread being spawned, thus simulating a simple approximation algorithm.

Our initial implementation calculated the cut from scratch after every single For-Loop. By changing the implementation to only adding the calculated difference to the old cut, we were able to siginificantly increase the performance of our algorithm. Additionally, we needed to decide which chunk size (meaning how many nodes will be assigned to each thread) is optimal, which is why we conducted experiments in order to resolve that question.

3.3 Algorithm Engineering Principles & Techniques

Regarding the AE-principles, we strongly relied on the AE-cycle. We started off by creating an abstract

idea of how our code could look like. We then started to implement it step by step, fixing it along the way until it worked as expected.

As for libraries, we used the good_lp library, which uses various solvers (from which one can choose). For our tasks, we found that the coin_cbc solver fits best our needs. For our experimental evaluation, we also used the Rng-library, which helped us in creating random graphs. In order to implement concurrency in the Two-Approximation algorithm, we used the Arc-library to allow concurrent read, as well as the Rw-library for ensuring exclusive writing on memory.

4 Experimental Evaluation

In this section, the experimental setup is described and the results are presented.

4.1 Data and Hardware

Experiments were conducted on a Dell Inc. Latitude 5491 with 8 GiB memory and a Intel Core i5-8400H CPU @ 2.50GHz × 8. The implementations were tested on graphs provided by the Big Mac Library, as well as randomly self generated graphs of various sized and densities. To achieve good results, we tested these algorithms on different random graphs, and plotted the average running time. We found this to be an advantage, as one implementation could work very well for certain densities or random graphs, while performing poorly on others.

4.2 Results

4.2.1 Heuristic Algorithm

The running time of the Heuristic Algorithm is shown in Figure 1. One can clearly see that the expected running time of $\mathcal{O}(n^2)$ matches with the measured running time.

4.2.2 Approximation Algorithm

Regarding the (Parallel) Approximation Algorithm, the running times are shown in Figure 2 and Figure 3. It comes at no surprise that both the parallel and the simple Approximation Algorithm have an quadratically increasing running time. However, interestingly the speed up of the concurrent algorithm increases with an increasing number of nodes (Figure 4). While at first, the simple algorithm is even faster than the concurrent one, the concurrent algorithm outperforms the simple algorithm by a factor, which is increasing with the number of nodes. This factor seems to converge at some point.

Figure 5 and Figure 6 show the running time of the concurrent Approximation Algorithm depending on the chunk size. Interestingly, we found the running times to be minimal when choosing a chunk size with the following formula:

$$\text{chunksize} = \frac{n}{2 \cdot c} \tag{8}$$

where n is the number of nodes in the graph and c representing the number of cores available for the computation.

That is because this chunk size evenly distributes the work of each For-Loop across the threads, while still keeping the number of threads to a minimum. If the chunk size is bigger, not all cpu cores are used efficiently, whereas if the chunk size is smaller, the number of spawned threads increases without enhancing performance, thus increasing the running time.

4.2.3 ILP solver

Figure 7 shows the running time of the ILP solver. Even though it is possible that, despite graphs being bigger in size, the running time decreases, a clear exponential trend is visible. This matches our expectations of a worst case running time of $\mathcal{O}(2^n)$

Figure 5 shows the performance of the Heuristic Algorithm compared to the concurrent Approximation Algorithm. We measured the solution of our algorithms, as well as their running times. While the Approximation Algorithm continuously outperformed the Heuristic with regard to the solution, the Heuristic was significantly faster. It is also important to consider, that the Heuristic only utilizes one core, while the concurrent Approximation utilized eight. Despite it, the Heuristic was 3 orders of magnitude faster than the Approximation.

5 Discussion and Conclusion

The results indicate that both the Heuristic and the Two-Approximation Algorithm yield great results. One advantage of the Approximation is that it guarantees us an approximation of 2. However, we found that the Heuristic find very good estimates quickly, but it is still important to keep in mind that the Heuristic can create arbitrarily bad solutions. This is not possible for the Approximation.

To come to a conclusion, it is difficult to find an exact solution for the Max Cut Problem. The running times of the ILP solver quickly exceeds reasonable orders of magnitudes. Yet, we have seen efficient ways to approximate the solutions.

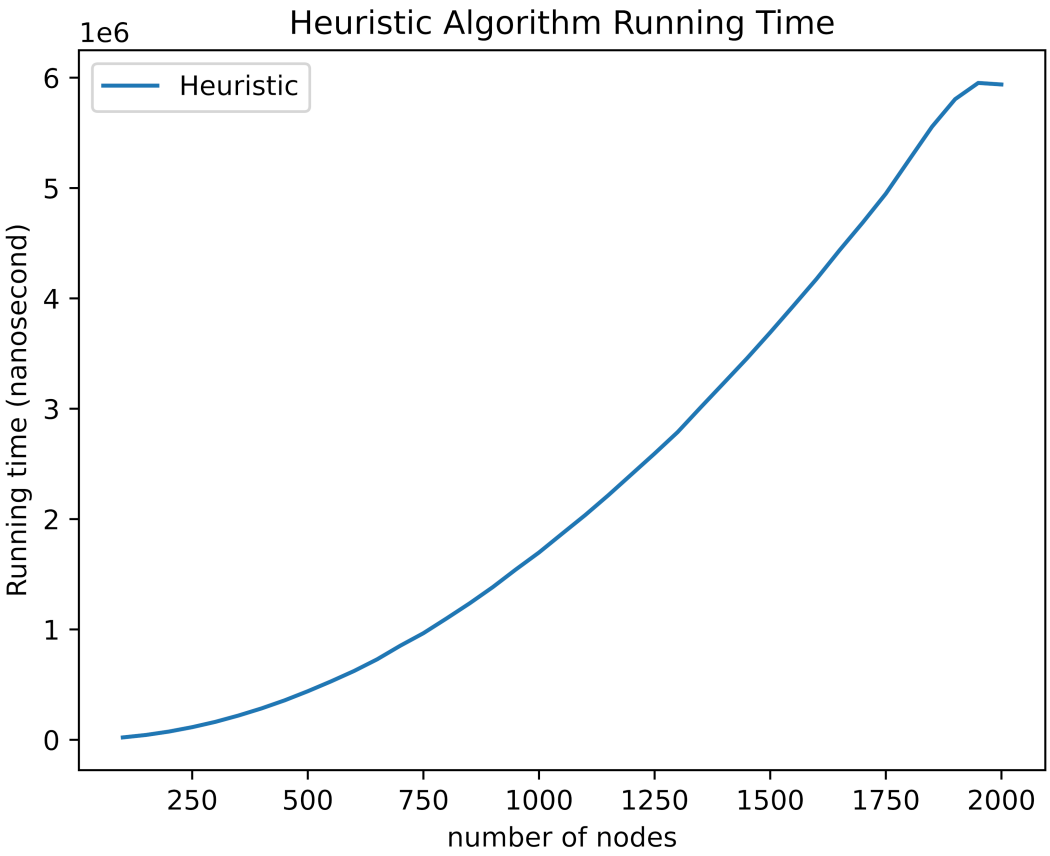

Figure 1: Running Time of Heuristic Algorithm for increasing number of nodes

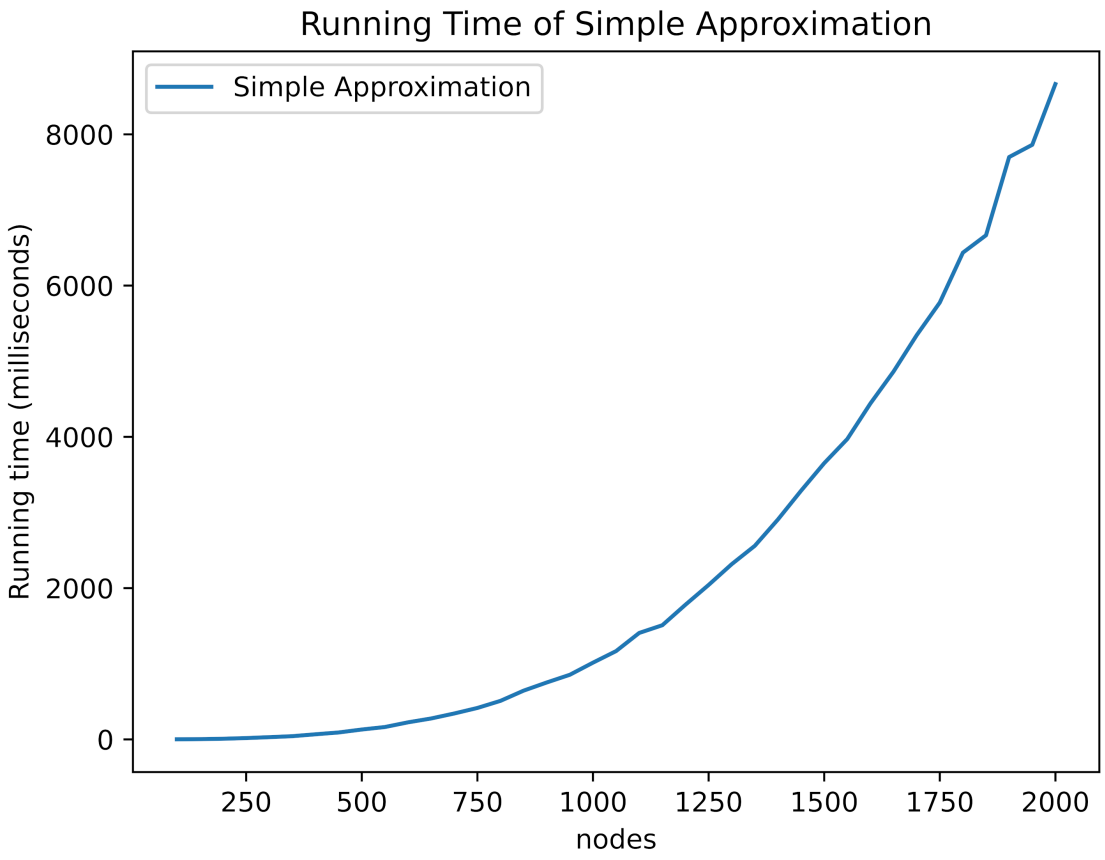

Figure 2: Running Time of Approximation Algorithm (simple) for increasing number of nodes

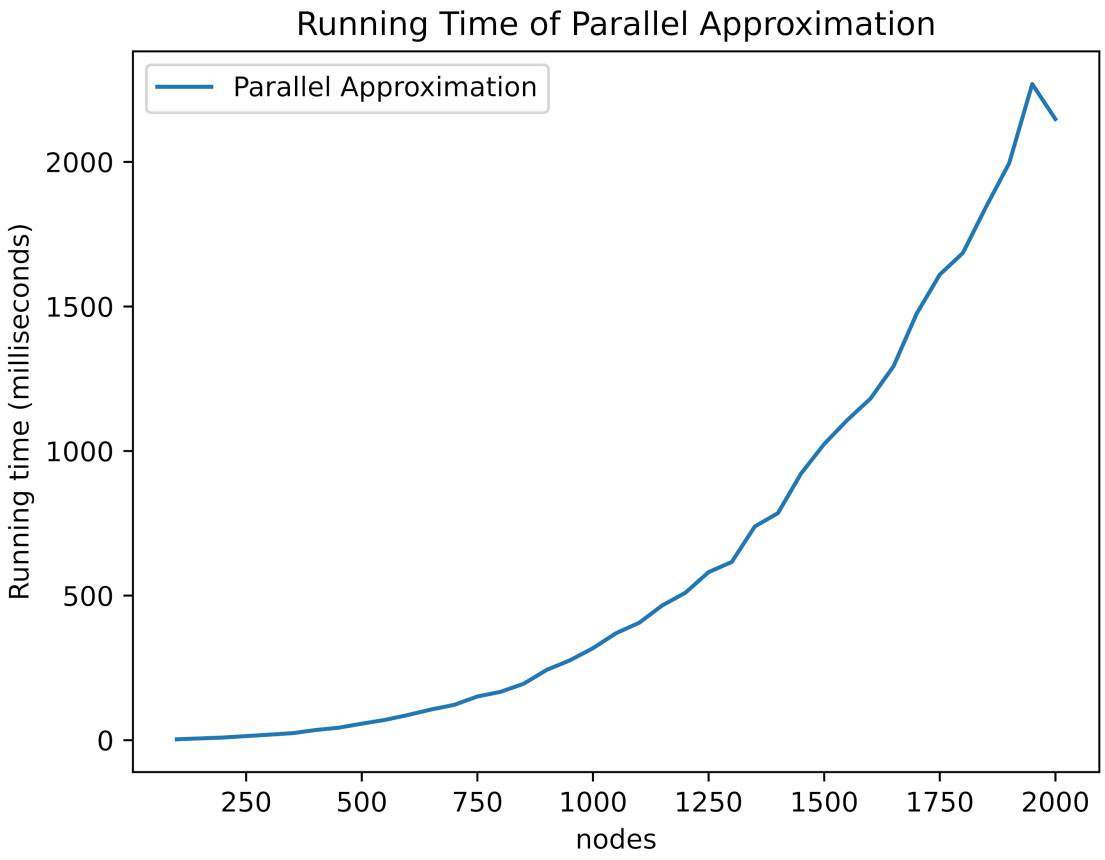

Figure 3: Running Time of Approximation Algorithm (parallel) for increasing number of nodes

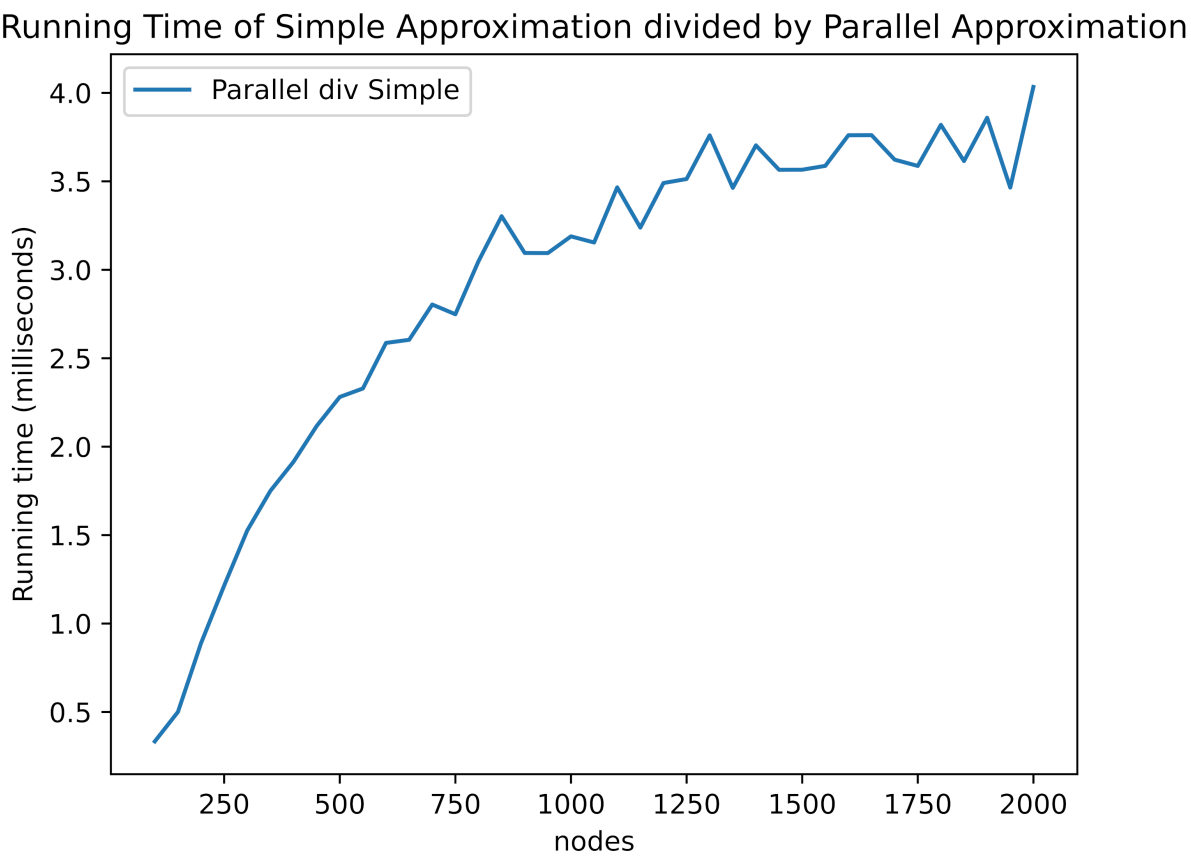

Figure 4: Running Time of Simple Approximation Divided by Running Time of Parallel Approximation

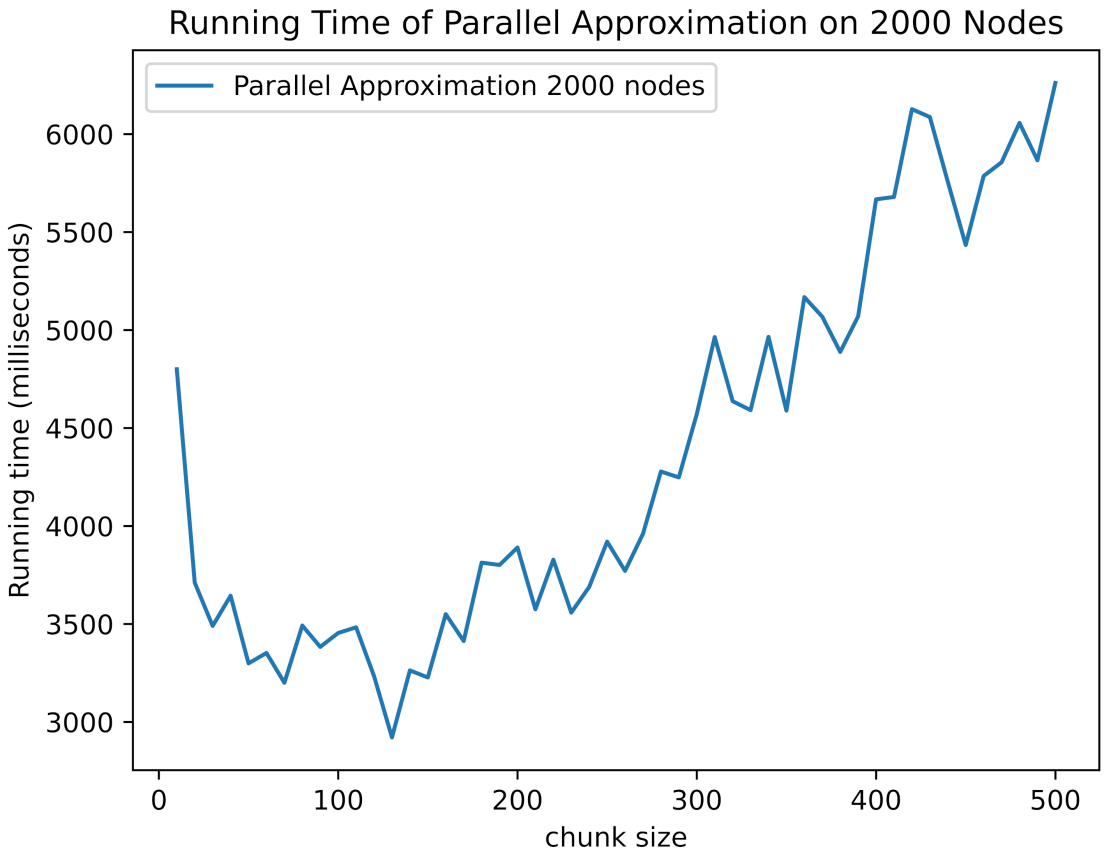

Figure 5: Running Time of Parallel Approximation for 2000 nodes with increasing chunk size

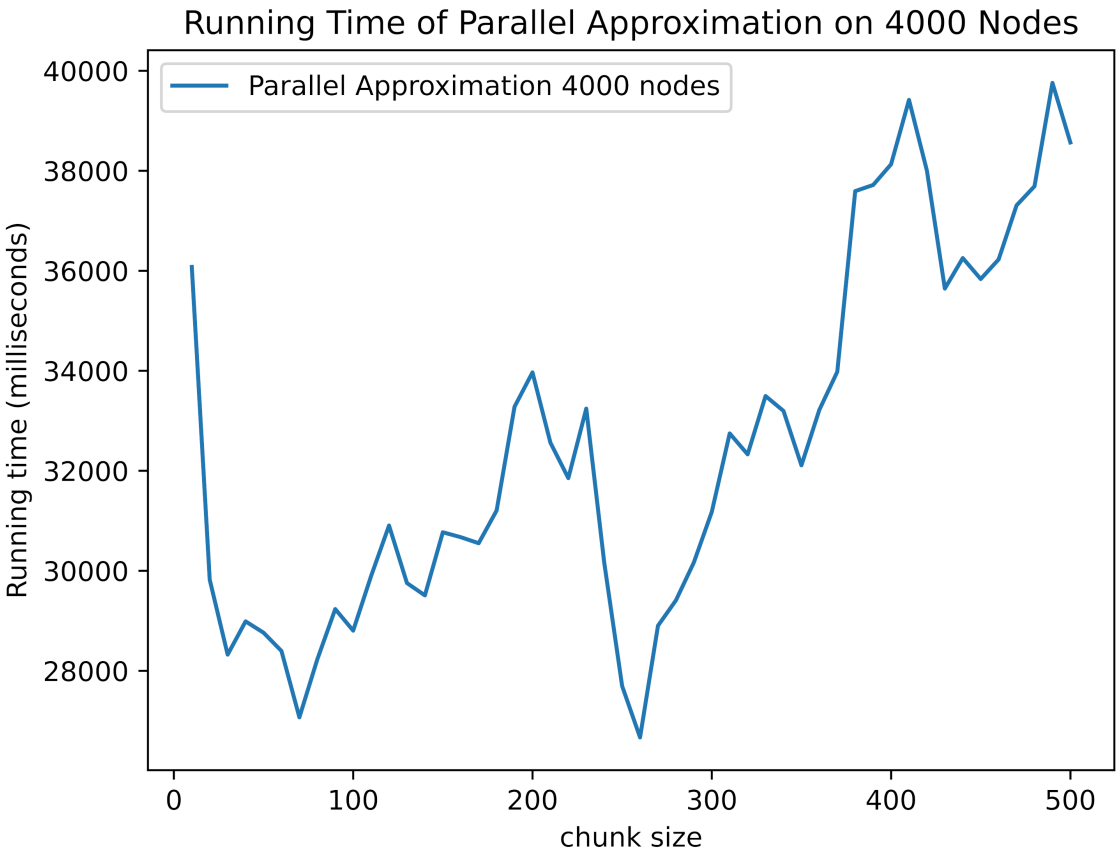

Figure 6: Running Time of Parallel Approximation for 4000 nodes with increasing chunk size

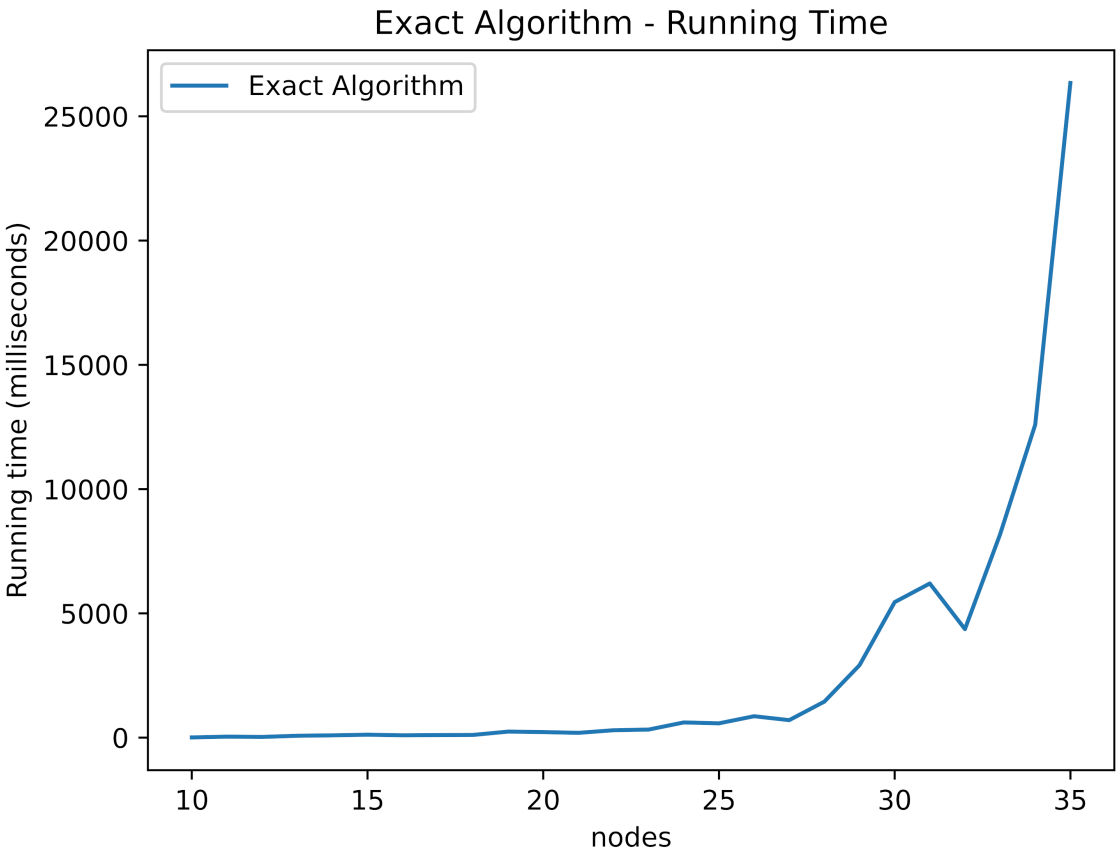

Figure 7: Running Time of Parallel Approximation for 4000 nodes with increasing chunk size

Table 1: Performance of Heuristic and (Parallel) Approximation Algorithm

graph	optimal solution	heuristic result	heuristic time	approx. result	approx. time
g05_60.0	536	451	9070	512	3331429
g05_60.1	532	446	8348	513	2957816
g05_60.2	529	472	8394	517	2393429
g05_60.3	538	451	8357	511	1756240
g05_60.4	527	432	8421	513	2155023
g05_60.5	533	434	8375	526	1931483
g05_60.6	531	450	8287	522	1655328
g05_60.7	535	431	8535	516	2327056
g05_60.8	530	453	8222	517	2222200
g05_60.9	533	475	8506	522	2660169
g05_80.0	929	791	14047	913	3298165
g05_80.1	941	825	13990	921	2901475
g05_80.2	934	766	14172	912	3364149
g05_80.3	923	810	14151	903	2327058
g05_80.4	932	791	13850	922	2885386
g05_80.5	926	786	13940	910	3688258
g05_80.6	929	809	14011	913	4517699
g05_80.7	925	797	14190	916	3281184
g05_80.8	923	823	13920	908	3720891
g05_80.9	1430	793	14083	907	3220666
g05_100.0	1425	1242	21306	1404	5980510
g05_100.1	1432	1253	20750	1406	4947027
g05_100.2	1424	1265	24903	1395	3289219
g05_100.3	1440	1238	20547	1404	3559857