Abstract

We present a multi-level graph partitioning algorithm based on the extreme idea to contract only a single edge on each level of the hierarchy. This obviates the need for a matching algorithm and promises very good partitioning quality since there are very few changes between two levels. Using an efficient data structure and new flexible ways to break local search improvements early, we obtain an algorithm that scales to large inputs and produces the best known partitioning results for many inputs. For example, in Walshaw’s well known benchmark tables we achieve 155 improvements dominating the entries for large graphs.

1 Introduction

Many important applications of computer science involve processing large graphs, e.g., stemming from finite element methods, digital circuit design, route planning, social networks, etc. Very often these graphs need to be partitioned or clustered such that there are few edges between the blocks (pieces).

A successful heuristic for partitioning large graphs is the *multilevel graph partitioning* approach (MGP) depicted in Figure 1 where the graph is recursively contracted to a smaller graph with the same basic structure. After applying an *initial partitioning* algorithm to this small graph, the contraction is undone and, at each level, a *local refinement* method improves the partition induced by the coarser level. Section 2 explains the method in more detail. Most systems instantiate MGP in a very similar way: Maximal matchings are contracted between two levels that try to include as many heavy edges as possible. Local refinement uses a linear time variant of local search. MGP has two crucial advantages over most other approaches to graph partitioning: We get near linear execution time since the graph shrinks geometrically and we get good partitioning quality since a good solution on some level yields a good initial solution on the next finer level, i.e., local search needs little work to further improve the solution.

Our central idea is to get even better partitions by making subsequent levels as similar as possible – we (un)contract only a *single* edge between two levels. We call this *n-GP*
since we have (almost) $n$ levels of hierarchy. More details are described in Section 3. $n$-GP has the additional advantage that there is no longer a need for an algorithm finding heavy matchings. This is remarkable insofar as a considerable amount of work on approximate maximum weight matching was motivated by the MGP application [24, 5, 22, 18]. Still, at first glance, $n$-GP seems to have substantial disadvantages also. Firstly, storing each level explicitly would lead to quadratic space consumption. We avoid this by using a dynamic graph data structure with little space overhead. Secondly, choosing maximal matchings instead of just a single edge for contraction has the side effect that the graph is contracted everywhere, leading to a more uniform distribution of node weights. We solve this problem by explicitly factoring node weights into the edge rating function prioritizing the edges to be contracted. Already in [13, 14] edge ratings have proven to lead to better results for graph partitioning. Perhaps the most serious problem is that the most common approach to local search is to let it run for a number of steps proportional to the current number of nodes. In the context of $n$-GP this could lead to a quadratic overall number of local search steps. Therefore, we develop a new, more adaptive stopping criteria for the local search that drastically accelerates $n$-GP without significantly reducing partitioning quality.

We have implemented $n$-GP in the graph partitioner KaSPar (Karlsruhe Sequential Partitioner). Experiments reported in Section 5 indicate that KaSPar scales well to large networks, computes the best known partitions for many instances of a “standard benchmark” and needs time comparable to system that previously computed the best results for large networks. Section 6 summarizes the results and discusses future directions.

More Related Work

There has been a huge amount of research on graph partitioning so that we refer to introductory and overview papers such as [7, 8, 26, 30] for more material. Well-known software packages based on MGP are Chaco [12], DiBaP [19], Jostle [29, 30], Metis [16, 17], Party [23, 25], and Scotch [20, 21].

KaSPar was developed partly in parallel with KaPPa (Karlsruhe Parallel Partitioner)
KaP Pa is a “classical” matching based MGP algorithm designed for scalable parallel execution and its local search only considers independent pairs of blocks at a time. Still, for \( k = 2 \), it is interesting to compare KaSPar and KaPPa since KaPPa achieves the previously best partitioning results for many large graphs, since both systems use a similar edge ratings, and since running times for a two processor parallel code and a sequential code could be expected to be roughly comparable.

There is a long tradition of \( n \)-level algorithms in geometric data structures based on randomized incremental construction (e.g., [11, 1]). Our motivation for studying \( n \)-level are contraction hierarchies [10], a preprocessing technique for route planning that is at the same time simpler and an order of magnitude more efficient than previous techniques using a small number of levels.

## 2 Preliminaries

Consider an undirected graph \( G = (V, E, c, \omega) \) with edge weights \( \omega : E \to \mathbb{R}_{\geq 0} \), node weights \( c : V \to \mathbb{R}_{\geq 0}, n = |V|, \) and \( m = |E| \). We extend \( c \) and \( \omega \) to sets, i.e., \( c(V') := \sum_{v \in V'} c(v) \) and \( \omega(E') := \sum_{e \in E'} \omega(e) \). \( \Gamma(v) := \{u : \{v, u\} \in E\} \) denotes the neighbors of \( v \).

We are looking for blocks of nodes \( V_1, \ldots, V_k \) that partition \( V \), i.e., \( V_1 \cup \cdots \cup V_k = V \) and \( V_i \cap V_j = \emptyset \) for \( i \neq j \). The balancing constraint demands that \( \forall i \in 1..k : c(V_i) \leq L_{\max} := (1 + \epsilon)c(V)/k + \max_{v \in V} c(v) \) for some parameter \( \epsilon \). The last term in this equation arises because each node is atomic and therefore a deviation of the heaviest node has to be allowed. The objective is to minimize the total cut \( \sum_{i<j} w(E_{ij}) \) where \( E_{ij} := \{(u, v) \in E : u \in V_i, v \in V_j\} \).

By default, our initial inputs will have unit edge and node weights. However, even those will be translated into weighted problems in the course of the algorithm.

**Contracting** an edge \( \{u, v\} \) means replacing the nodes \( u \) and \( v \) by a new node \( x \) connected to the former neighbors of \( u \) and \( v \). We set \( c(x) = c(u) + c(v) \). If replacing edges of the form \( \{u, w\}, \{v, w\} \) would generate two parallel edges \( \{x, w\} \), we insert a single edge with \( \omega(\{x, w\}) = \omega(\{u, w\}) + \omega(\{v, w\}) \). **Uncontracting** an edge \( e \) undoes its contraction. Partitions computed for the contracted graph are extrapolated to the uncontracted graph in the obvious way, i.e., \( u \) and \( v \) are put into the same block as \( x \).

**Local Search** is done by moving single nodes between blocks. The gain \( g_B(v) \) of moving node \( v \) to block \( B \) is decrease in total cut size caused by this move. For example, if \( v \) has 5 incident edges of unit weight, 2 of which are inside \( v \)'s block and 3 of which lead to block \( b \) then \( g_B(v) = 3 - 2 = 1 \).

## 3 \( n \)-Level Graph Partitioning

Figure 2 gives a high-level recursive summary of \( n \)-GP. The base case is some other partitioner used when the graph is sufficiently small. In KaSPar, contraction is stopped when either only 20\( k \) nodes remain, no further nodes are eligible for contraction, or there are less edges than nodes left. The latter happens when the graph consists of many independent components.
Function \( n\)-GP\((G, k, \epsilon)\)

if \( G \) is small then return initialPartition\((G, k, \epsilon)\)
pick the edge \( e = \{u, v\} \) with the highest rating
contract \( e \);
\( \mathcal{P} := n\-GP\((G, k, \epsilon)\); \) uncontract \( e \)
activate\((u)\); activate\((v)\); localSearch()
return \( \mathcal{P} \)

Figure 2: \( n\)-GP.

As observed in [14] Scotch [20] produces better initial partitions than metis, and therefore we also use it in KaSPar.

The edges to be contracted are chosen according to an edge rating function. KaSPar adopts the rating function

\[
\text{expansion}^*\(\{u, v\}\):= \frac{\omega(\{u, v\})}{c(u)c(v)}
\]

which fared best in [14]. As a further measure to avoid unbalanced inputs to the initial partitioner, KaSPar never allows a node \( v \) to participate in a contraction if the weight of \( v \) exceeds \( 1.5n/(20k) \). Selecting contracted edges can be implemented efficiently by keeping the contractable nodes in a priority queue sorted by the rating of their most highly rated incident edge.

In order to make contraction and uncontraction efficient, we use a “semidynamic” graph data structure: When contracting an edge \( \{u, v\} \), we mark both \( u \) and \( v \) as deleted, introduce a new node \( w \), and redirect the edges incident to \( u \) and \( v \) to \( w \). The advantage of this implementation is that edges adjacent to a node are still stored in adjacency arrays which are more efficient than linked lists needed for a full fledged dynamic graph data structure. A disadvantage of our approach is a certain space overhead. However, it is relatively easy to show that this space overhead is bounded by a logarithmic factor even if we contract edges in some random fashion (see [4]). In Section 5 we will demonstrate experimentally that the overhead is actually often a small constant factor. Indeed, this is not very surprising since the edge rating function is not random, but designed to keep the contracted graph sparse. Overall, with respect to asymptotic memory overhead, \( n\)-GP is no worse than methods with a logarithmic number of levels.

3.1 Local Search Strategy

Our local search strategy is similar to the FM-algorithm [4] that is also used in many other MGP systems. We now outline our variant and then discuss differences.

Originally, all nodes are unmarked. Only unmarked nodes are allowed to be activated or moved from one block to another. Activating a node \( v \in B' \) means that for blocks \( \{B \neq B' : \exists \{v, u\} \in E \land u \in B\} \) we compute the gain

\[
g_B(v) = \sum \{\omega(\{v, u\}) : \{v, u\} \in E, v \in B\} - \sum \{\omega(\{v, u\}) : \{v, u\} \in E, v \in B'\}
\]
of moving \( v \) to block \( B \). Node \( v \) is then inserted into the priority queue \( P_B \) using \( g_B(v) \) as the priority. We call a queue \( P_B \) eligible if the highest gain node in \( P_b \) can be moved to block \( B \) without violating the balance constraint for block \( B \). Local search repeatedly looks for the highest gain node \( v \) in any eligible priority queue \( P_B \) and moves \( v \) to block \( B \). When this happens, node \( v \) becomes nonactive and marked, the unmarked neighbors of \( v \) get activated and the gains of the active neighbors are updated. The local search is stopped if either no eligible nonempty queues remain, or one of the stopping criteria described below applies. After the local search stops, it is rolled back to the lowest cut state reached during the search (which is the starting state if no improvement was achieved). Subsequently all previously marked nodes are unmarked. The local search is repeated until no improvement is achieved.

The main difference to the usual FM-algorithm is that our routine performs a highly localized search starting just at the uncontracted edge. Indeed, our local search does nothing if none of the uncontracted nodes is a border node, i.e., has a neighbor in another block. Other FM-algorithms initialize the search with all border nodes. In \( n \)-GP the local search may find an improvement quickly after moving a small number of nodes. However, in order to exploit this case, we need a way to stop the search much earlier than previous algorithms which limit the number of steps to a constant fraction of the current number of nodes \( |V| \).

Stopping Using a Random Walk Model. It makes sense to make a stopping rule more adaptive by making it dependent on the past history of the search, e.g., on the difference between the current cut and the best cut achieved before.

We model the gain values in each step as identically distributed, independent random variables whose expectation \( \mu \) and Variance \( \sigma^2 \) is obtained from the previously observed \( p \) steps. In Appendix A we show how from these (purely heuristical, i.e., technically unwarranted) assumptions we can derive that it is unlikely that the local search will produce a better cut if

\[
p\mu^2 > \alpha\sigma^2 + \beta
\]

where \( \alpha \) and \( \beta \) are tuning parameters and \( \mu \) is the average gain since the last improvement. For the variance \( \sigma^2 \), we can also use the variance observed throughout the current local search. Parameter \( \beta \) is a base value that avoids stopping just after a small constant number of steps that happen to have small variance. Currently we set it to \( \ln n \).

4 Trial Trees

It is a standard technique in optimization heuristics to improve results by repeating various parts of the algorithm. We generalize several approaches used in MGP by adapting an idea initially used in a fast randomized min-cut algorithm [15]: After reducing the number of nodes by a factor \( c \), we perform two independent trials using different random seeds for tie breaking during contraction, initial partitioning, and local search. Among these trials the one with the smaller cut is used for continuing upwards. This way, we perform independent
trials at many levels of contraction controlled by a single tuning parameter \( c \). As long as \( c > 2 \), the total number of contraction steps performed stays \( O(n) \).

5 Experiments

Implementation. We implemented KaSPar in C++ using gcc-4.3.2. We use priority queues based on paring heaps [28] available in the policy-based elementary data structures library (pbds) for implementing contraction and refinement procedures. In the following experimental study we compared KaSPar to Scotch 5.1, kMetis 4.0 and the same version of KaPPa as in [14].

System. We performed our experiments on a single core of an Intel Xeon Quad-core Processor featuring 2x4 MB of L2 cache and clocked at 2.667 GHz of a 2 processor Intel Xeon X5355 node with 16 GB of RAM running Suse Linux Enterprise 10.

Instances. We report results on two suites of instances summarized in Table 1. rggX is a random geometric graph with \( 2^X \) nodes that represent random points in the unit square and edges connect nodes whose Euclidean distance is below \( 0.55 \sqrt{\ln n/n} \). This threshold was chosen in order to ensure that the graph is almost connected. DelaunayX is the Delaunay triangulation of \( 2^X \) random points in the unit square. Graphs bcsstk29..ftooth and ferotor..auto come from Chris Walshaw’s benchmark archive [27]. Graphs bel, nld, deu and eur are undirected versions of the road networks of Belgium, the Netherlands, Germany, and Western Europe respectively, used in [3]. Instances af_shell9 and af_shell10 come from the Florida Sparse Matrix Collection [2]. coAuthorsDBLP, coPapersDBLP, citationCiteseer, coAuthorsCiteseer and cnr2000 are examples of social networks taken from [9].

For the number of partitions \( k \) we choose the values used in [27]: 2, 4, 8, 16, 32, 64. Our default value for the allowed imbalance is 3 % since this is one of the values used in [27] and the default value in Metis.

When not otherwise mentioned, we perform 10 repetitions for the small networks and 5 repetitions for the other. We report the arithmetic average of computed cut size, running time and the best cut found. When further averaging over multiple instances, we use the geometric mean in order to give every instance the same influence on the final figure.

Configuring the Algorithm. We use two sets of parameter settings fast and strong. These methods only differ in the constant factor \( \alpha \) in the local search stopping rule, see Equation [11], in the contraction factor \( c \) for the trial tree (Section 4), and in the number of initial partitioning attempts \( a \) performed at the coarsest level of contraction:

| strategy | \( \alpha \) | \( c \) | \( a \) |
|---------|-------|-------|-------|
| fast    | 1     | 8     | \( 25/\log_2 k \) |
| strong  | 4     | 2.5   | \( 100/\log_2 k \) |
Table 1: Basic properties of the graphs from our benchmark set. The large instances are split into five groups: geometric graphs, FEM graphs, street networks, sparse matrices, and social networks. Within their groups, the graphs are sorted by size.

| Medium sized instances         |         |         |
|-------------------------------|---------|---------|
| **graph**                     | **n**   | **m**   |
| rgg17                         | $2^{17}$| 1 457 506|
| rgg18                         | $2^{18}$| 3 094 566|
| Delaunay17                    | $2^{17}$| 786 352 |
| Delaunay18                    | $2^{18}$| 1 572 792|
| bcsstk29                      | 13 992  | 605 496 |
| 4elt                          | 15 606  | 91 756  |
| fesphere                      | 16 386  | 98 304  |
| cti                           | 16 840  | 96 464  |
| memplus                       | 17 758  | 108 384 |
| cs4                           | 33 499  | 87 716  |
| pwt                           | 36 519  | 289 588 |
| bcsstk32                      | 44 609  | 1 970 092|
| body                          | 45 087  | 327 468 |
| t60k                          | 60 005  | 178 880 |
| wing                          | 62 032  | 243 088 |
| finan512                      | 74 752  | 522 240 |
| ferotor                       | 99 617  | 662 431 |
| bel                           | 463 514 | 1 183 764|
| nld                           | 893 041 | 2 279 080|
| af_shell9                     | 504 855 | 17 084 020|

| Large instances               |         |         |
|-------------------------------|---------|---------|
| **graph**                     | **n**   | **m**   |
| rgg20                         | $2^{20}$| 13 783 240|
| Delaunay20                    | $2^{20}$| 12 582 744|
| fetooth                       | 78 136  | 905 182 |
| 598a                          | 110 971 | 1 483 868|
| ocean                         | 143 437 | 819 186 |
| 144                           | 144 649 | 2 148 786|
| wave                          | 156 317 | 2 118 662|
| m14b                          | 214 765 | 3 358 036|
| auto                          | 448 695 | 6 629 222|
| deu                           | 4 378 446| 10 967 174|
| eur                           | 18 029 721| 44 435 372|
| af_shell10                    | 1 508 065| 51 164 260|

| Social networks               |         |         |
|-------------------------------|---------|---------|
| coAuthorCiteseer              | 227 320 | 1 628 268|
| coAuthorDBLP                  | 299 067 | 1 955 352|
| cnr2000                       | 325 557 | 3 216 152|
| citationCiteseer              | 434 102 | 32 073 440|
| coPaperDBLP                   | 540 486 | 30 491 458|

Note that this is considerably less parameters compared to KaPPa. In particular, there is no need for selecting a matching algorithm, an edge coloring algorithm, or global and local iterations for refinement.

**Scalability.** Figure 3 shows the number of edges touched during contraction (KaSPAR strong, small and large instances). We see that this scales linearly with the number of input edges and with a fairly small constant factor between 2 and 3. Interestingly, the number of local search steps during local improvement (Figure 4) decreases with increasing input size. This can be explained by the sublinear number of border vertices that we have in graphs that have small cuts and by small average search space sizes for the local search. Indeed, Figure 5 in the appendix indicates that the average length of local searches grows only logarithmically with $n$. All this translates into fairly complicated running time behavior. Still, Figure 6 in the appendix warrants the conclusion that running time scales “near linearly” with the input.
Figure 3: Number of edges created during contraction.

Figure 4: Total number of local search steps. The nearly straight lines represent series for the graphs rgg15..rgg24 and Delaunay15..Delaunay24 for different $k$. 
size. The term in the running time depending on \( k \) grows sublinearly with the input size so that for very large graphs the number of blocks does not matter much.

**Does the Random Walk Model Work?** We have compared KaSPar fast with a variant where the stopping rule is disabled (i.e., \( \alpha = \infty \)). For the small instances this yields about 1% better cut sizes at the cost of an order of magnitude larger running time. This is a small improvement both compared to the improvement KaSPar achieves over other systems and compared to just repeating KaSPar fast 10 times (see Table 2).

**Do Trial trees help?** We use the following evaluation: We run KaSPar strong and measure its elapsed time. Then for different values of initial partitionings \( a \) we repeat KaSPar strong without trial trees (\( c = 0 \)), until the sum of the run times of all repetitions exceeds the run time of KaSPar strong. Then for different values \( a \) we compare the best edge cut achieved during repeated runs to the one produced by KaSPar strong. Finally, we average the obtained results over 5 repetitions of this procedure. If we then qualify the computed partitions, we usually get almost identical results (a fraction of a percent difference). However, most of the time trial trees are a bit better and for road networks we get considerable improvements. For example, for the European network we get an improvement of 10% on average over all \( k \).

**Comparison with other Systems.** Table 2 summarizes the results by computing geometric means over 10 runs for the small instances and over 5 runs for the large instances and social networks. We exclude the European road network for \( k = 2 \) because KaPPa runs out of memory in that case. Detailed per instance results can be found in the appendix. KaPPa strong produces 5.9% larger cuts than KaSPar strong for small instances (average value) and 8.1% larger cuts for the large instances. This comparison might seem a bit unfair because KaPPa is about five times faster. However, KaPPa is using \( k \) processors in parallel. Indeed, for \( k = 2 \) KaSPar strong needs only about twice as much time. Also note that KaPPa strong needs about twice as much time as KaSPar fast while still producing 6% larger cuts despite running in parallel. The case \( k = 2 \) is also interesting because here KaPPa and KaSPar are most similar – parallelism does not play a big role (2 processors) and both local search strategies work only on two blocks at all time. Therefore 6% improvement of KaSPar over KaPPa we can attribute mostly to the larger number of levels.

Scotch and kMetis are much faster than KaSPar but also produce considerably larger cuts – e.g., 32% larger for large instances (kMetis, average). For the European road network, the difference in cut size even exceeds a factor of two. Such gaps usually cannot be breached by just running the faster solver a larger number of times. For example, for large instances, Scotch is only a factor around 4 faster than KaSPar fast, yet its best cut values obtained from 5 runs are still 12.7% larger than the average values of KaSPar fast.

For social networks all systems have problems. KaSPar lags further behind in terms of speed but extends its lead with respect to the cut size. We mostly attribute the larger

---

1This may not apply to the social networks which have considerably worse behavior.
Table 2: Geometric means over all instances.

| code          | small graphs |         |         |         |         |         |         |         |         |         |         |
|---------------|--------------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
|               | best         | avg     | t[s]    | best    | avg     | t[s]    | best    | avg     | t[s]    | best    | avg     |
| KaSPar strong | 2675         | 2729    | 7.37    | 12450   | 12584   | 87.12   | -       | -       | -       | -       | -       |
| KaSPar fast   | 2717         | 2809    | 1.43    | 12655   | 12842   | 14.43   | 93657   | 99062   | 297.34  | -       | -       |
| KaSPar fast, α = ∞ | 2697   | 2780    | 23.21   | -       | -       | -       | -       | -       | -       | -       | -       |
| KaPPa strong  | 2807         | 2890    | 2.10    | 13323   | 13600   | 28.16   | 117701  | 123613  | 78.00   | -       | -       |
| KaPPa fast    | 2819         | 2910    | 1.29    | 13442   | 13727   | 16.67   | 117927  | 126914  | 46.40   | -       | -       |
| kMetis        | 3097         | 3348    | 0.07    | 15540   | 16656   | 0.71    | 117959  | 134803  | 1.42    | -       | -       |
| Scotch        | 2926         | 3065    | 0.48    | 14475   | 15074   | 3.83    | 168764  | 168764  | 17.69   | -       | -       |

| Large Instances |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| k               | KaSPar strong   | KaPPa strong    | KaSPar strong   | KaPPa strong    |
|-----------------|-----------------|-----------------|-----------------|-----------------|
| 2               | 2842            | 2873            | 36.89            | 2977            | 3054            | 15.03          |
| 4               | 5642            | 5707            | 60.66            | 6190            | 6384            | 30.31          |
| 8               | 10464           | 10580           | 75.92            | 11375           | 11652           | 37.86          |
| 16              | 17345           | 17567           | 102.52           | 18678           | 19061           | 39.13          |
| 32              | 27416           | 27707           | 137.08           | 29156           | 29562           | 31.35          |
| 64              | 41284           | 41570           | 170.54           | 43237           | 43644           | 22.36          |

run time to the larger cut sizes relative to the number of nodes which greatly increase the number of local searches necessary. A further effect may be that the time for a local search step is proportional to the number of partitions adjacent to the nodes participating in the local search. For “well behaved” graphs this is mostly two, but for social networks which get denser on the coarser levels this value can get larger.

The Walshaw Benchmark [27] considers 34 graphs using $k \in \{2, 4, 8, 16, 32, 64\}$ and balance parameter $\epsilon \in \{0, 0.01, 0.03, 0.05\}$ giving a total of 816 table entries. Only cut sizes count – running time is not reported. We tried all combinations except the case $\epsilon = 0$ which KaSPar cannot handle yet. We ran KaSPar strong with a time limit of one hour and report the best result obtained in the appendix. KaSPar improved 155 values in the benchmark table: 42 for 1%, 49 for 3% and 64 for 5% allowed imbalance. Moreover, it reproduced equally sized cuts in 83 additional cases. If we count only results for graphs having over $44k$ nodes and $\epsilon > 0$, KaSPar improved 131 and reproduced 27 cuts, thus summing up to 63% of large graph table slots. We should note, that 51 of the new improvements are over partitioners different from KaPPa. Most of the improvements lie in the lower triangular part of the table, meaning that KaSPar is particularly good for either large graphs, or smaller graphs with small $k$. On the other hand, for small graphs, large $k$, and $\epsilon = 1\%$ KaSPar was often not able to obtain a feasible solution. A primary reason for this seems to be that
initial partitioning yields highly infeasible solutions that KaSPar is not able to improve considerably during refinement. This is not astonishing, since Scotch targets $\epsilon = 3\%$ and does not even guarantee that.

6 Conclusion

$n$-GP is a graph partitioning approach that scales to large inputs and currently computes the best known partitions for many large graphs, at least when a certain imbalance is allowed. It is in some sense simpler than previous methods since no matching algorithm is needed. Although our current implementation of KaSPar is a considerable constant factor slower than the fastest available MGP partitioners, we see potential for further tuning. In particular, thanks to our adaptive stopping rule, KaSPar needs to do very little local search, in particular for large graphs and small $k$. Thus it suffices to tune the relatively simple contraction routine to obtain significant improvements. On the other hand, the adaptive stopping rule might also turn out to be useful for matching based MGP algorithms.

A lot of opportunities remain to further improve KaSPar. In particular, we did not yet attempt to handle the case $\epsilon = 0$ since this may require different local search strategies. We also want to try other initial partitioning algorithms and ways to integrate $n$-GP into other metaheuristics like evolutionary search.

We expect that $n$-GP could be generalized for other objective functions, for hypergraphs, and for graph clustering. More generally, the success of $n$-GP also suggests to look for more applications of the $n$-level paradigm.

An apparent drawback of $n$-GP is that it looks inherently sequential. However, we could probably obtain a good parallel algorithm by contracting small sets of highly rated, independent edges in parallel. Indeed, in the light of our results for KaSPar the complications coming from the need to find maximal matchings of heavy edges seem unnecessary, i.e., a parallelization of $n$-GP might be fast and simple.

Acknowledgements. We would like to thank Christian Schulz for supplying data for KaPPa, Scotch and Metis.

References

[1] M. Birn, M. Holtgrewe, P. Sanders, and J. Singler. Simple and fast nearest neighbor search. In 11th Workshop on Algorithm Engineering and Experiments, 2010.

[2] T. Davis. The University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices, 2008.

[3] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering route planning algorithms. In Algorithmics of Large and Complex Networks, volume 5515 of LNCS State-of-the-Art Survey, pages 117–139. Springer, 2009.
[4] R. Dementiev, P. Sanders, D. Schultes, and J. Sibeyn. Engineering an external memory minimum spanning tree algorithm. In *IFIP TCS*, Toulouse, 2004.

[5] D. Drake and S. Hougardy. Improved linear time approximation algorithms for weighted matchings. In *7th Workshop on Randomization and Approximation Techniques in Computer Science (APPROX)*, LNCS 2764, pages 14–23, 2003.

[6] C. M. Fiduccia and R. M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *19th Conf. on Design Automation*, pages 175–181, 1982.

[7] P. Fjallstrom. Algorithms for graph partitioning: A survey. *Linkoping Electronic Articles in Computer and Information Science*, 3(10), 1998.

[8] V. K. G. Karypis. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM JOURNAL ON SCIENTIFIC COMPUTING*, 20(1):359–392, 1998.

[9] R. Geisberger, P. Sanders, and D. Schultes. Better approximation of betweenness centrality. In *10th Workshop on Algorithm Engineering and Experimentation*, pages 90–108, San Francisco, 2008. SIAM.

[10] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *7th Workshop on Experimental Algorithms*, volume 5038 of *LNCS*, pages 319–333. Springer, 2008.

[11] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7(4):381–413, 1992.

[12] B. Hendrickson. Chaco: Software for partitioning graphs. [http://www.sandia.gov/~bahendr/chaco.html](http://www.sandia.gov/~bahendr/chaco.html).

[13] M. Holtgrewe. A scalable coarsening phase for a multi-level partitioning algorithm. Diploma thesis, Universität Karlsruhe, 2009.

[14] M. Holtgrewe, P. Sanders, and C. Schulz. Engineering a scalable high quality graph partitioner. In *24th IEEE International Parallel and Distributed Processing Symposium*, 2010. to appear, see also arXiv:0910.2004.

[15] D. R. Karger and C. Stein. A new approach to the minimum cut problem. *J. ACM*, 43(4):601–640, 1996.

[16] G. Karypis and V. Kumar. MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0. [http://www.cs.umn.edu/~metis](http://www.cs.umn.edu/~metis).

[17] G. Karypis and V. Kumar. MeTiS, A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, Version 4.0, 1998.
[18] J. Maue and P. Sanders. Engineering algorithms for approximate weighted matching. In 6th Workshop on Exp. Algorithms (WEA), volume 4525 of LNCS, pages 242–255. Springer, 2007.

[19] H. Meyerhenke, B. Monien, and T. Sauerwald. A new diffusion-based multilevel algorithm for computing graph partitions. Journal of Parallel and Distributed Computing, 69(9):750–761, 2009.

[20] F. Pellegrini. SCOTCH: Static Mapping, Graph, Mesh and Hypergraph Partitioning, and Parallel and Sequential Sparse Matrix Ordering Package, 2007. http://www.labri.fr/perso/pelegrin/scotch/.

[21] F. Pellegrini. SCOTCH 5.1 User’s guide. Technical report, Laboratoire Bordelais de Recherche en Informatique, Bordeaux, France, 2008.

[22] S. Pettie and P. Sanders. A simpler linear time $2/3 - \epsilon$ approximation for maximum weight matching. Information Processing Letters, 91(6):271–276, 2004.

[23] R. Preis. PARTY Partitioning Library , 1996. http://www2.cs.uni-paderborn.de/cs/robsy/party.html.

[24] R. Preis. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In 16th Symp. on Theoretical Aspects of Computer Science, volume 1563 of LNCS, pages 259–269. Springer, 1999.

[25] R. Preis and R. Diekmann. The PARTY Partitioning Library, User Guide . Technical report, University of Paderborn, Germany, 1996. Tr-rsfb-96-02.

[26] K. Schloegel, G. Karypis, and V. Kumar. Graph partitioning for high performance scientific simulations. Technical Report 00-018, University of Minnesota, 2000.

[27] A. J. Soperm, C. Walshaw, and M. Cross. A combined evolutionary search and multilevel optimisation approach to graph partitioning. J. Global Optimization, 29(2):225–241, 2004. http://staffweb.cms.gre.ac.uk/~c.walshaw/partition/.

[28] A. Tavory, V. Dreizin, and B. Kosnik. Policy-based data structures. http://gcc.gnu.org/onlinedocs/libstdc++/ext/pb_ds/ IBM Haifa and Redhat, 2004.

[29] C. Walshaw. JOSTLE – graph partitioning software, 2005. http://staffweb.cms.gre.ac.uk/~wc06/jostle/.

[30] C. Walshaw and M. Cross. JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview. In F. Magoules, editor, Mesh Partitioning Techniques and Domain Decomposition Techniques, pages 27–58. Civil-Comp Ltd., 2007. (Invited chapter).
A Derivation of Stopping Rules

Consider a situation where \( p \) steps of local search have been performed with average value \( \mu \) and variance \( \sigma^2 \). Then in the next \( s \) steps, we can expect a deviation from the expectation \( (p + s)\mu \) by something of the order \( \sqrt{s\sigma^2} \). The expression \( (p + s)\mu + \sqrt{s\sigma^2} \) is maximized for \( s^* := \frac{\sigma^2}{4\mu^2} \). Now the idea is to stop when for some tuning parameter \( x \), \( (p + s^*)\mu + x\sqrt{s^*\sigma^2} > 0 \), i.e., it is reasonably likely that a random walk modelling our local search can still give an improvement. This translates to the condition \( p > \frac{\sigma^2}{\mu^2} \left( \frac{x^2}{2} - \frac{1}{4} \right) \) or simply \( p\mu^2 \gg \sigma^2 \).

B Additional Figures and Tables

![Figure 5: Average length of local searches.](image1)

![Figure 6: Running Time.](image2)
Table 3: All results for social network instances
Table 4: All results for large instances.
| Graph     | 2    | 4    | 8    | 16   | 32   | 64   |
|-----------|------|------|------|------|------|------|
| add30     | 644  | 594  | 1212 | 1177 | 1814 | 1704 |
| data      | 198  | 188  | 495  | 383  | 699  | 600  |
| f3elt      | 90   | 89   | 201  | 199  | 361  | 352  |
| uk         | 19   | 19   | 41   | 42   | 92   | 84   |
| add32      | 10   | 10   | 33   | 33   | 66   | 66   |
| bcsstk33   | 10105| 10097| 21756| 21508| 34377| 34178|
| whitaker3  | 126  | 126  | 382  | 380  | 670  | 656  |
| crack      | 184  | 183  | 370  | 362  | 696  | 678  |
| wing_modal | 1703 | 1696 | 3609 | 3572 | 5574 | 5443 |
| fe_left2   | 130  | 130  | 349  | 349  | 622  | 605  |
| vibrobox   | 11538| 10310| 19267| 19199| 25190| 24553|
| bcsstk29   | 2818 | 2818 | 8035 | 8159 | 14212| 13965|
| ctf        | 158  | 158  | 325  | 321  | 501  | 494  |
| fesphere   | 386  | 386  | 798  | 708  | 1230 | 1152 |
| memplus    | 318  | 318  | 950  | 944  | 1815 | 1802 |
| memplus    | 5698 | 5489 | 10234| 9559 | 12599| 11785|
| cs4        | 378  | 367  | 970  | 940  | 1520 | 1430 |
| bcsstk30   | 6347 | 6335 | 16617| 16622| 34761| 34004|
| bcsstk31   | 2723 | 2701 | 7351 | 7444 | 13371| 13417|
| fe_pwt     | 340  | 340  | 704  | 704  | 1441 | 1442 |
| bcsstk32   | 4667 | 4667 | 9247 | 9492 | 20855| 21490|
| fe_body    | 262  | 262  | 599  | 636  | 1079 | 1156 |
| t60k       | 78   | 75   | 213  | 211  | 470  | 465  |
| wing       | 803  | 787  | 1683 | 1666 | 2616 | 2589 |
| crack2     | 708  | 708  | 3027 | 3038 | 7144 | 7269 |
| fin312     | 102  | 102  | 324  | 324  | 648  | 648  |
| feg_tkoth  | 3819 | 3823 | 6938 | 7103 | 11650| 11935|
| feg_rotor  | 2055 | 2045 | 7405 | 7480 | 12959| 13165|
| m98a       | 2593 | 2388 | 7992 | 8154 | 16179| 16469|
| feg_ocean  | 388  | 387  | 1856 | 1878 | 4251 | 4299 |
| 144        | 6499 | 6479 | 15196| 15345| 25455| 25818|
| wave       | 8716 | 8682 | 16891| 17475| 29207| 30511|
| m14b       | 3828 | 3826 | 13034| 13391| 25921| 26666|
| auto       | 10004| 10042| 26941| 27790| 35731| 47650|

Table 5: Walshaw Benchmark with $\epsilon = 1$
| Graph       | 2     | 4     | 8     | 16    | 32    | 64    |
|------------|-------|-------|-------|-------|-------|-------|
| add20      | 636   | 576   | 1195  | 1158  | 1765  | 1690  |
| data       | 186   | 185   | 379   | 378   | 662   | 650   |
| left       | 87    | 87    | 199   | 198   | 346   | 336   |
| uk         | 18    | 18    | 40    | 40    | 84    | 81    |
| add32      | 10    | 10    | 33    | 33    | 66    | 66    |
| bcsstk31   | 10064 | 10064 | 21035 | 21035 | 34150 | 34078 |
| whitaker3  | 120   | 120   | 381   | 378   | 662   | 655   |
| crack      | 182   | 182   | 360   | 360   | 685   | 676   |
| wing modal | 1681  | 1680  | 3572  | 3561  | 5424  | 5401  |
| fe 4elt2   | 130   | 130   | 349   | 348   | 607   | 600   |
| vibrobox   | 11358 | 10310 | 19239 | 18778 | 24691 | 24171 |
| bcsstk20   | 2518  | 2818  | 7983  | 7977  | 14041 | 13876 |
| left       | 137   | 137   | 319   | 319   | 533   | 523   |
| fe_sphere  | 384   | 384   | 792   | 792   | 1593  | 1593  |
| cti        | 318   | 318   | 924   | 924   | 1724  | 1724  |
| memplus    | 6026  | 5355  | 10145 | 9418  | 12521 | 11628 |
| csd        | 366   | 361   | 959   | 959   | 1390  | 1390  |
| bcsstk30   | 6251  | 6251  | 16497 | 16497 | 31275 | 31275 |
| bcsstk31   | 2676  | 2676  | 7183  | 7183  | 13240 | 13240 |
| fe_pwt     | 340   | 340   | 704   | 704   | 1416  | 1416  |
| bcsstk32   | 4667  | 4667  | 8778  | 8778  | 17825 | 17825 |
| fe_body    | 262   | 262   | 598   | 598   | 1033  | 1033  |
| tt60k      | 71    | 71    | 211   | 211   | 407   | 407   |
| wing       | 789   | 774   | 1660  | 1651  | 2567  | 2567  |
| brack2     | 684   | 684   | 2853  | 2853  | 6980  | 6980  |
| finan512   | 162   | 162   | 324   | 324   | 648   | 648   |
| fe_tooth   | 3794  | 3792  | 6862  | 6862  | 13722 | 13722 |
| fe_rotor   | 1960  | 1963  | 7182  | 7182  | 12546 | 12546 |
| 598a       | 2369  | 2367  | 7878  | 7878  | 15820 | 15820 |
| ft roam    | 311   | 311   | 1710  | 1710  | 3076  | 3076  |
| wave       | 8640  | 8640  | 16822 | 16822 | 32644 | 32644 |
| m14b       | 3828  | 3823  | 12977 | 12977 | 25550 | 25550 |
| auto       | 9716  | 9782  | 25979 | 25979 | 45109 | 45109 |

Table 6: Walshaw Benchmark with $\epsilon = 3$
Table 7: Walshaw Benchmark with $\epsilon = 5$

| Graph       | 2   | 4   | 8   | 16  | 32  | 64  |
|-------------|-----|-----|-----|-----|-----|-----|
| add20       | 610 | 550 | 1186| 1157| 1575| 1675|
| data        | 183 | 181 | 369 | 368 | 640 | 628 |
| 3elt        | 87  | 87  | 198 | 197 | 336 | 330 |
| uk          | 18  | 18  | 39  | 40  | 81  | 78  |
| add32       | 10  | 10  | 33  | 33  | 63  | 65  |
| bccstk33    | 9194| 9194| 20198| 20584| 33971| 33938| 55273| 54323| 79159| 77163| 111659| 106886 |
| whitaker3   | 126 | 120 | 380 | 378 | 688 | 660 |
| crack       | 182 | 182 | 361 | 360 | 673 | 667 |
| wing_nodal  | 1672| 1668| 3541| 3536| 5475| 5550|
| fe_4elt2    | 130 | 130 | 340 | 335 | 596 | 583 |
| vibrobox    | 11538| 10310| 19021| 18778| 24203| 23930| 34298| 31235| 42890| 39959| 50994| 48200 |
| bccstk29    | 2818| 2818| 7936| 7942| 13619| 13614| 21914| 20924| 34906| 33818| 57220| 54935 |
| 4elt        | 137 | 137 | 318 | 315 | 519 | 516 |
| fe_sphere   | 384 | 384 | 784 | 764 | 1219| 1152|
| ctt         | 318 | 318 | 900 | 890 | 1708| 1716|
| memplus     | 5510| 5261| 10011| 9299| 12458| 11555| 14047| 13078| 15749| 14170| 18213| 16434 |
| csf         | 381 | 381 | 955 | 936 | 1483| 1467|
| bccstk30    | 6251| 6251| 16136| 16312| 34146| 34350| 69520| 70043| 114860| 113321| 175723| 170591 |
| bccstk31    | 2676| 2676| 7099| 7102| 12941| 13068| 23403| 25243| 48150| 47459| 69768| 67554 |
| fe_pwt      | 340 | 340 | 700 | 701 | 1405| 1409|
| bccstk32    | 4622| 4441| 8454| 8481| 19678| 20099| 35208| 35965| 60441| 59824| 94238| 91006 |
| fe_body     | 262 | 262 | 596 | 601 | 1017| 1054|
| tt0k        | 65  | 65  | 202 | 196 | 437 | 434 |
| wing        | 784 | 770 | 1604| 1636| 2528| 2551|
| brack2      | 660 | 660 | 2745| 2739| 6671| 6781|
| finan512    | 162 | 162 | 324 | 324 | 648 | 648 |
| fe_tooth    | 3780| 3773| 6825| 6864| 11337| 11662| 17404| 17603| 25216| 25624| 35466| 35476 |
| feRotor     | 1950| 1955| 7052| 7045| 12380| 12566| 20039| 20132| 31450| 31576| 46749| 46608 |
| 598a        | 2338| 2330| 7763| 7831| 15544| 15721| 25585| 25808| 39144| 39369| 57412| 58031 |
| fe_ocean    | 311 | 311 | 1765| 1767| 3494| 3491|
| li11        | 6303| 6302| 15038| 15209| 25025| 25259| 37433| 38225| 59445| 59920| 87296| 82229 |
| wave        | 8699| 8663| 16662| 16820| 28015| 28000| 42382| 42800| 61788| 62520| 85658| 86663 |
| m11b        | 3806| 3802| 12976| 13156| 25292| 25679| 41750| 42608| 65231| 66793| 98005| 99063 |
| auto        | 9487| 9450| 25399| 25883| 44520| 45039| 75066| 76488| 120001| 122378| 171459| 173968 |