Karlsruhe Institute of Technology

Generating Graphs with Guarantees on Partition Costs

Study Thesis of

Florian Merz

At the Department of Informatics
Institute of Theoretical Informatics
Group Algorithmics I

Reviewer:	Prof. Dr. rer. nat. D. Wagner
Second reviewer:	Prof. Dr. rer. nat. Peter Sanders
Advisor:	Dipl.-Inform. A. Schumm
Second advisor:	Dr. rer. nat. Robert Görke

Duration:: 01. February 2011 – 31. July 2011

Abstract

Partitioning a graph is to divide its set of vertices into a fixed number of equally sized disjoint subsets minimizing the number of edges between them. In this work, we describe an algorithm to generate large random graphs such that an optimal solution to this problem is known. For this task, we first generate dense subgraphs such that we know a lower bound on the size of the minimal cut for each of these subgraphs. We then connect the subgraphs by few edges such that the partition induced by these subgraphs is known to be optimal. This algorithm has expected linear run-time.

Zusammenfassung

Graphpartitionierung ist das Problem, die Knotenmenge eines Graphen in disjunkte, gleich große Teilmengen aufzuteilen, welche die Anzahl der Kanten zwischen diesen Teilmengen minimiert. In dieser Arbeit beschreiben wir einen Algorithmus, um große Zufallsgraphen generieren, zu denen wir eine optimale Partitionierung kennen. Um dies zu erreichen, generieren wir zuerst dichte Teilgraphen, für welche wir die untere Schranke für die Größe deren minimalen Schnittes kennen. Diese Teilgraphen verbinden wir danach mit einigen wenigen Kanten, so dass die durch die Teilgraphen induzierte Partitionierung optimal ist. Die erwartete Laufzeit dieses Algorithmus ist linear.

Contents

1. Introduction

Partitioning a graph is to divide its set of vertices into a fixed number of equally sized disjoint subsets minimizing the number of edges between them. This is needed to decompose data for parallel computation or to optimize circuit layouts. Finding such a partition is NP-complete, which motivated the development of heuristics in the last decades.

Commonly, those algorithms are assessed by means of their performance on real world data. However, those data are limited and in most cases optimal partitions are unknown. Because of the latter, we focus on generating random graphs for which we can provide a guarantee on their partition costs. To achieve this we give an algorithm that generates such a graph bottom-up. Based on knowledge about minimal cuts during the generation process we can provide partitions, whose edge cuts are known to be optimal. We use an initial selected partition of the vertices to build up a graph step by step, keeping track of the minimal cut of all subgraphs which occur in the process.

This process limits the variety of graphs that can be generated and results in graphs in which the vertices have similar degrees. Also the traces of the building process are still visible but can be covered by modifications made in a post-processing step. Additional edges could be added in order to create more diverse graphs but we provide only the base upon which such modification can be designed.

Furthermore we implement our algorithm in order to test it against existing partitioning algorithms.

Related Work

The graph partitioning archive [9], which was set up as part of [8], provides a platform to test graph partitioning algorithm on selected graphs. Partitioners as METIS [5] and KaPPa [4] among many others compete in finding near to optimal partitions of those test graphs.

A large number of methods to generate random graphs have been developed over the last decades, as for example the $G(n, p)$ [3] and $G(n, m)$ [2] models, as well as the small-world [10] and preferential attachment [1] models.

In [6], a generator for graphs with known minimum cut structure is presented. To this end, in the first step, a random cactus representation of the minimum cuts is chosen and in the second step, a graph with this structure is generated. Our goal is to generate large random graphs that incorporate guarantees on their partition costs. To the best of our knowledge, this exact problem has not been addressed before.

2. Preliminaries

In this chapter we introduce the notation we use in the rest of this work and clarify the problem we solved.

2.1 Notation

Most of the notation we use is taken from [7].

Undirected graphs

An *undirected graph* or *graph* is a pair $G = (V, E)$, with a finite set of vertices V and a set of edges E, which are unorderd pairs in V. In this work we only consider *simple loopless* graphs, where each edge is unique and edges of the form $\{u, u\}$ don't exist.

An edge $e = \{u, v\}$ *connects* the vertices u and v and if such an edge e exists u and v are called *connected* or *adjacent*, and e is *incident* to u and v. In this case v is called a *neighbor* of u. Two edges e and e' are called *connected* or *adjacent* if they share a vertex, otherwise they are called *disjoint*.

Two graphs are called *disjoint* if their vertex sets are disjoint. Given two disjoint graphs $G = (V, E)$ and $G' = (V', E')$, we denote:

$$G + G' := (V \cup V', E \cup E').$$

For an edge $e = \{u, v\}$ with $e \notin E$, we denote:

$$G + e := (V, E \cup \{e\}),$$

similarly for a set of edges E'' with $E'' \cap E = \emptyset$, we denote:

$$G + E'' := (V, E \cup E'').$$

Subgraphs

Given a graph $G = (V, E)$, a subset $V' \subseteq V$ *spans* a set of edges:

$$E[V'] := \{\{u, v\} \in E \mid u, v \in V'\}$$

A graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$ is called subgraph of G. If $E' = E[V']$, G' is called the subgraph *induced* by V'. Another subset $E' \subseteq E$ *induces* $G_V[E'] := (V, E')$.

Degrees, Cuts and Partitions

Let $G = (V, E)$ be a graph, for a vertex $v \in V$ we denote by $\delta_G(v)$ or $\delta(v)$ the set of edges incident to v. And by $N_G(v) := N_E(v) := N(v)$ the set of neighbors of v.

The number of edges incident to v is called *degree* of v. We denote: $\deg_G(v) := |\delta_G(v)|$.

If $G = (V, E)$ is a graph and $U \subseteq V$, we denote:

$$\delta_G(U) := \{e = \{u, v\} \in E \mid u \in U, v \in V \setminus U\},$$

which are the edges connecting U and $V \setminus U$. Any subset F of E for which a $U \subseteq V$ with $F = \delta(U)$ exists, is called a *cut* of G. Further we denote for the *size* of F called *cut size*:

$$d_G(U) := |\delta(U)| = |F|.$$

The *minimal size* of a of cut of G is denoted by:

$$\operatorname{MinCut}(G) := \min(\{|\delta(U)| \mid U \subseteq V\}).$$

Any cut F of G with size $|F| = k$ is called a *k-cut* of G. If $k = \operatorname{MinCut}(G)$ it is called a *minimal cut.* For $s \in U$ and $t \notin U$, the cut $\delta(U)$ is called an *s-t cut.* Also if $S \subseteq U$ and $T \subseteq V \setminus U$, $\delta(U)$ is called an *S-T cut.* Moreover, for any subset $S \subseteq V$ and $T := V \setminus S$ we denote:

$$F_{ST} := \delta(S) = \delta(T).$$

For the subsets $U, U' \subset V$ and a cut $F = \delta(U)$, we denote:

$$F[U'] := \{e \in F \mid e \in E[U']\}.$$

$F[U']$ is called the *induced cut* of $E[U']$ by F.

A *k-partition* P of G is a mapping of V into disjoint subsets V_i with $\lfloor \frac{|V|}{k} \rfloor \leq |V_i| \leq \lceil \frac{|V|}{k} \rceil$ such that the union $\bigcup_{1 \leq i \leq k} V_i = V$. The set of edges between the subsets is denoted by E_P. A *minimal k-partition* P is a k-partition that minimizes E_P. In the following we will refer to the subsets V_i as *blocks.*

2.2 Problem Statement

The *graph partitioning problem* consists of dividing the set of vertices of a given graph into subsets, which are equally sized, and have few edges between them. Therefore we consider a graph $G := (V, E)$ and a given integer k. Our goal is to partition V into k disjoint subsets $V_1, \ldots, V_k$ such that all subsets have equal size and the number of edges between the subsets is minimized.

Our goal is to develop a fast algorithm, which generates a random graph and provides a minimal k-partition for that graph. Since the graph partitioning problem is NP-complete we cannot calculate a k-partition for the generated graph, instead we have to define a k-partition and generate the edges in a way that we can prove that this k-partition is minimal. How to generate those edges is the essential problem we address in this work. To make guarantees on the costs of a k-partition we abandon the idea of covering all possible graphs, and focus on performance and correctness.

Therefore the formal problem statement of this work:
Given a set of vertices V, an integer k and the edge cut l of an optimal k-partition, we want to generate a k-partition P and a set of edges E such that P is a minimal k-partition with an edge cut l of the graph $G := (V, E)$.

3. Graph Generator

The basic idea of our graph generator is to build a graph bottom-up. We generate the set of vertices and divide them into *blocks*, which will be the blocks of an optimal k-partition P of the graph we generate. Our goal is to generate a set of edges E_P of a predefined size between those blocks, such that this set of edges minimizes the size $|E_P|$ of the k-partition P. The idea is to generate E_P based on knowledge about the minimal cuts of the blocks. Hence we have to generate sets of edges for each block, such that we have guarantees on the minimal cut of each block.

To generate a block with a defined minimal cut size, we divide the block into small *subblocks* and generate edges, such that each of those subblocks has the same minimal cut size than the desired minimal cut size of the block. Then we insert edges between the subblocks of each block creating new subblocks with the same assertions on the minimal cut size. We repeat this step until only one subblock is left for each block. These final subblocks have a predefined minimal cut size and therefore represent the blocks we wanted to generate.

Note that theoretically, we could also use the graph generator presented in [6] to generate the blocks. However, as this involves iteratively computing the cactus representation of the graph, it would be too expensive for our purpose.

In the final step the edges between the blocks are generated. Our graph generator generates a graph and an optimal k-partition for this graph dependent on the number of vertices n and the minimal cut size of all blocks c. The size of the edge cut of the optimal k-partition will be $\frac{k \cdot c}{2}$ in this case.

In this chapter we first present an abstract of the algorithm, starting with a short description in prose, followed by a more detailed pseudo code. Further we introduce some data structures we use and explain details how the algorithm works.

3.1 Overview

As mentioned before the idea of our algorithm is to build a graph bottom-up, carrying along guarantees of minimal cuts. Suppose c is the desired minimal cut size of each block, we divide our Algorithm 1 into three phases.

1. Generating the initial subblocks of size $2c$.
2. Fusing the subblocks to k blocks.

3. Insert edges between the blocks.

In the following we explain those three phases based on Algorithm 1.

Phase 1

At first we generate vertices and divide them equally into the desired number of blocks k. Now we divide those blocks again, as done in Algorithm 1, this time they are grouped into subblocks of a size $2c$. In the following we refer to blocks and subblocks as graphs. Then we insert edges into the initial subblocks, until the degree of every node at least the desired minimal cut size c. After this step the initial subblocks have minimal cut size of at least c. We prove this in Proposition 1.

Phase 2

For the next step we connect the set of initial subblocks either by:

Method A: recursively joining two random subblocks by inserting c random edges between them and replacing the two joined subblocks in the set by the new one, see Algorithm 2.

Method B: grouping the subblocks into sets containing $2c$ subblocks. For each set we connect the subblocks of this set randomly until the minimal cut size of all the subblocks is at least c, similarly to the first phase. Now we have a new set of subblocks which we join the same way, see Algorithm 3.

The resulting subblocks contain all vertices of each block. According to Proposition 2 and Proposition 3, after this step the minimum cut size in each block is at least c.

Phase 3

After the first two phases, we have k blocks, which we join again by inserting random edges. We choose the edges in a way that we can assure that the degree of each block is less than $2c$ and the amount of edges we insert is smaller than $\frac{k \cdot c}{2}$. After this step the mapping into the blocks is a minimal k-partition of the resulting graph, as we prove in Proposition 4 and 5 of Chapter 4.

3.2 Building the Graph

In this section we will describe step by step how we generate a graph, starting by explaining the input required.

Input

The input needed by our algorithm is a set of vertices V, represented by the *number of vertices n*, the *number of blocks k* and the lower bound for the *minimal cut size c*. This minimal cut size will determine the number of edges between the block. For a given input the edge cut of a minimal k-partition will be of size $\frac{k \cdot c}{2}$, which we prove in Proposition 5.

How to Store a Graph

To store the graph structure of a block we use an *adjacency matrix*, more precise an *upper triangular matrix* which carries the degree of each vertex on its diagonal. The first idea is to use an own adjacency matrix for each subblock, which stores the structure of the subblock, but we achieved to decrease the memory cost and running-time by a factor of two by storing the subblocks directly in the adjacency matrix of the block (Figure 3.1).

Algorithm 1: Graph Generator

```
Data   : set of vertices $V$, number of blocks $k$, minimal cut size $c$
Result: set of edges $E$, and a minimal k-partition of $G = (V, E)$
1  $E \leftarrow \emptyset$;
2  $B \leftarrow \{b_1, \dots, b_k\}$, arbitrary k-partition of $V$;
3  foreach $b_i$ in $B$ do
       //phase 1
4      subblocks $\leftarrow \{s_1, \dots, s_o\}$, arbitrary $o$-partition of $b_i$ with $o = \lceil \frac{|b_i|}{2c} \rceil$;
5      foreach $S$ in subblocks do
6          while $\exists x$ in $S$ with deg$(x) < c$ do
7              $x \leftarrow$ choose uniformly at random among all v $\in S$ with deg$(v) < c$;
8              if $\exists y$ in subblock with deg$(y) < c$ then
9                  $y \leftarrow$ choose uniformly at random among all v $\in S$ with deg$(v) < c$;
10             else
11                 $y \leftarrow$ choose uniformly at random among all v $\in S$ with deg$(v) = c$;
12             end
13             $E \leftarrow E \cup \{x, y\}$;
14         end
15     end
       //phase 2
16     fuse subblocks(subblocks, $c$, $E$);
17 end
   //phase 3
18 while number of new edges added $< \frac{k \cdot c}{2}$ and $\exists b_i, b_j \in B$ with
   deg$(b_i) < 2c - 1 \wedge$ deg$(b_j) < 2c - 1$ do
19     pick two random elements $b_i$ and $b_j$ of $B$, with deg$(b_i) < 2c - 1 \wedge$ deg$(b_j) < 2c - 1$;
20     $e \leftarrow$ edge between random vertex in $b_i$ and random vertex in $b_j$, which is not an
       element of $E$;
21     $E \leftarrow E \cup \{e\}$;
22 end
```

Algorithm 2: `fuse subblocks` method A

```
Data   : set of sets of vertices subblocks, minimal cut size $c$, set of edges $E$
Result: set of edges $E$, such that MinCut$(G[\bigcup_{s \in \text{subblocks}} s] + E)$
1  while $|subblocks| > 1$ do
2      pick two random elements $a, b$ of subblocks;
3      $A \leftarrow$ set of $c$ unique edges between vertices in $a$ and vertices in $b$, chosen
       uniformly at random;
4      $E \leftarrow E \cup A$;
5      subblocks $\leftarrow \big(\text{subblocks} \cup \{a \cup b\}\big) \setminus \{a, b\}$;
6  end
```

Algorithm 3: `fuse subblocks` method B

```
Data  : set of sets of vertices subblocks, minimal cut size $c$, set of edges $E$
Result: set of edges $E$, such that MinCut($G[\bigcup_{s \in \text{subblocks}} s] + E$)
1 while $|subblocks| > 1$ do
2     new_subblock $\leftarrow$ select $2c$ random elements of $S$;
3     $A \leftarrow$ edges between the elements of new_subblock, in a way that $c \leq d(s) \leq c + 1$
      for all $s \in$ new_subblock;
4     subblocks $\big(\leftarrow subblocks \setminus new\_blocks\big) \cup \{\bigcup_{s \in \text{new\_subblock}} s\}$;
5     $E \leftarrow E \cup A$;
6 end
```

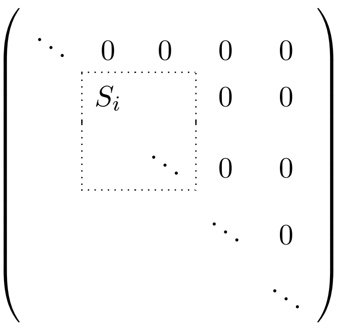


Figure 3.1: The adjacency matrix of a subblock S_i stored in an adjacency matrix of a block

This is done by a subblock data structure that keeps track of the location of the adjacency matrix of the subblock in the adjacency matrix of the block. This allows us to fuse two adjacent subblocks by simply adjusting the location of the new subblock we generated, which saves us the effort of transferring the entries of the two fused adjacency matrices into the new one.

In the following *adding* or *inserting* an edge means that we set the according entry in the adjacency matrix of the current block to one and increase the diagonal entry of the incident edges by one, to keep track of their degree. We start with the initial phase 1 of Algorithm 1.

Divide V into Blocks

The first step is to divide the set of vertices V into k blocks. Since the vertices are represented by numbers we divide them into sequences of size $\frac{n}{k}$ and $\frac{n}{k} + 1$. Thus the vertices of each block can be calculated by knowing an offset and its size. We store this data in a `Block` data structure.

Since we can build the k blocks independently the following steps can be done either sequentially or in parallel. However, keep in mind that we have to execute them for each block. The first step to build a block is to initialize an empty *adjacency matrix* to store the edges and the degree of each vertex. Now we continue with the next step of phase 1 of Algorithm 1.

Generate the $2c$ Sub Blocks

Given a block, we divide it equally into subblocks of size $2c$, in case $2c$ is a divider of the size of the block, else we also allow subblocks of size $2c - 1$. We do this in the same way as we divided the initial set of vertices, and store the offset, which is the number of the first vertex, and the size of the subblock for each subblock in a `SubBlock` data structure. To keep track of the subblocks of a block we store references to them in our `Block` data structure.

Next we build the $2c$ subblocks of the current block. The goal is to generate a graph with a minimal cut size c. We achieve this by iterating over the vertices inserting edges, which are chosen at random and are incident to this vertex until its degree equals c. Since it's not possible to satisfy the property that every vertex has a degree of exactly c we allow a degree of $c + 1$ if necessary. The pseudo code in Algorithm 4 clarifies this.

Algorithm 4: generate $2c$ subblock

```
Data  : set of $n'$ vertices $V'$, minimal cut size $c$
Result: set of edges $E$, such that the minimal cut size of the resulting graph is at
        least $c$
$E \leftarrow \emptyset$;
foreach $v$ in $V'$ do
    $P \leftarrow$ random permutation of $V$;
    foreach $u$ in $P$ do
        if $deg(v) < c$ and $u \neq v$ and $deg(v) < c$ then
            $E \leftarrow \{u, v\}$;
        end
    end
end
//This time allow a degree of $c + 1$ if necessary
foreach $v$ in $V'$ do
    $P \leftarrow$ random permutation of $V$;
    foreach $u$ in $P$ do
        if $deg(v) < c$ and $u \neq v$ and $deg(v) < c + 1$ then
            $E \leftarrow \{u, v\}$;
        end
    end
end
```

After initializing the $2c$ subblocks we can proceed with phase 2 of Algorithm 1.

Fusing the Sub Blocks

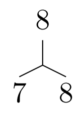
In our implementation we focus on Method A as in Algorithm 2, so we abandon Method B for the moment and explain in detail how we create the order in which we fuse the subblocks to obtain a block.

Due to our division of the block into the initial $2c$ subblocks the adjacency matrix of our resulting graph is a block matrix with the initial subblocks on its diagonal. It looks like this:

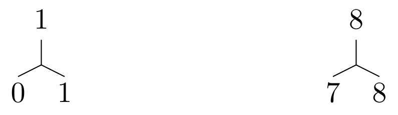
$$\begin{pmatrix} B_1 & 0 & 0 & 0 & 0 \\ & B_2 & 0 & 0 & 0 \\ & & \ddots & 0 & 0 \\ & & & \ddots & 0 \\ & & & & B_{\frac{n}{2c \cdot k}} \end{pmatrix}$$

To fuse the subblocks consecutively at random we generate a binary tree containing the indices of the subblocks as leaves, which represents the order we use to fuse the subblocks. The vertex labels, in this case the number of a vertex according to the adjacency matrix, are irrelevant for the output, because we only consider unlabeled graphs. We can use this to generate the binary tree to optimize the fusion process. In detail we can generate a binary tree that always fuses a subblock with his neighbor subblock according to its index. This assures that we can have subblocks with continuously numbered labels after each fusion step. Thus we save the overhead of keeping track of the scattered parts of the adjacency matrix in which the vertices of a subblock are being stored. We only have to set the offset and size of the new `SubBlock` data structure accordingly. Therefore we use this kind of binary tree for its impact on the run-time of the selection of a random vertex of a subblock, which can now be done by picking a random number of at most the size of the subblock and adding the offset.

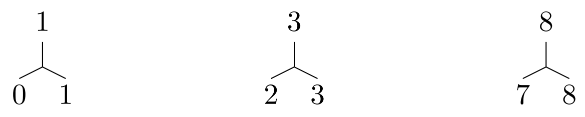
We use a random permutation of the indices of the subblocks to build a binary tree. In order to create a binary tree with n' leafs, each representing one subblock, we use a random permutation $p := \{i_1, \ldots, i_{n'-1}\}$ of the first $n' - 1$ indices of the subblocks. Those indices are by construction consecutive. Now we join the subblock with index i_1 with the subblock with index $i_1 + 1$ and index the resulting subblock with $i_1 + 1$. To illustrate this, consider a set of subblocks with indices from 0 to 9 and a random permutation of the first nine indices (702345186). After the first step we have the following tree:



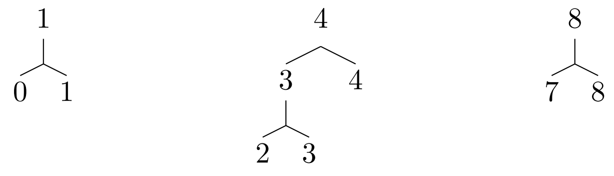
Next we take the index i_2 according to the permutation and join it with $i_2 + 1$. In general, we consecutively join i_j and $i_j + 1$. This next step results in:



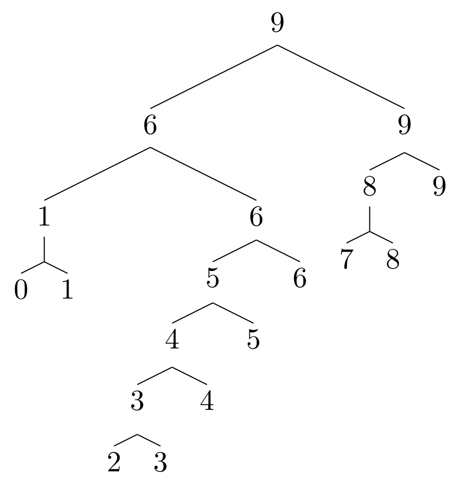
The next two steps are:



and:



And finally the resulting binary looks like this:



To fuse two subblocks we consecutively choose a random vertex from each subblock. If the edge between those two doesn't already exist, we add it. In case it already exists we have to choose two new vertices. This happens in Algorithm 2 as part of Algorithm 1. After c edges have been inserted we create a new `SubBlock` data structure for our new subblock. Then we add it to the list of subblocks of the current block and delete the two subblocks we used to create it.

After joining the subblocks according to the binary tree we created previously only one subblock remains. This subblock represents the block we wanted to build. As for the last step, we now have to generate the edges between the blocks.

Obtaining the Edges Between the Blocks

In phase 3 of Algorithm 1 we generate $\frac{k \cdot c}{2}$ edges between the k blocks, this is independent of the actual structure of the blocks. It only depends on the number of vertices in each block. Thus they can be obtained in a separate process. To do this we assume the vertices are labeled sequentially across the blocks, therefore all we have to do is pick $k \cdot c$ random pairs of vertices representing the edges and assure that:

1. each block has a degree of at most $2c - 1$,
2. the edges are unique.

This can be done by choosing two different blocks at random, and choose one random vertex of each. Now we add this pair to our selection if its not already in there. After we chose $2c - 1$ vertices of a block, this block is removed from the list of blocks to assure that the first condition. is satisfied. We repeat this until at most $\frac{k \cdot c}{2}$ pairs have been selected.

3.3 Run-Time

In this section we take a look at the *average case* run-time of our implementation given the input data:

- **n**: the number of vertices $|V|$
- **k**: the number of blocks
- **c**: the lower bound for any minimal cut size

At first we initialize the k blocks. To do this we have to calculate the offset and size for each of them. This takes $O(1)$ for each block. Next, we have to initialize the adjacency matrix for each block, which runs in $O((\frac{n}{k})^2)$ for one block. This sums up to $O(\frac{n^2}{k})$ to initialize all k blocks.

To build a block we first generate the $2c$ subblocks for this block. Generating the initial subblocks takes $O(\frac{n}{c \cdot k})$, since initializing one subblock, which means appointing the size and offset, runs in $O(1)$. Building a $2c$ subblock using Algorithm 4 runs in $O(c^2)$, so it takes $O(\frac{c \cdot n}{k})$ to build all initial subblocks of a block.

Next we generate the binary tree to obtain the order in which we fuse the subblocks, this runs in $O(\frac{n}{c \cdot k})$. Now we can fuse the subblocks in the calculated order, this takes $\frac{n}{c \cdot k}$ fusion steps, and each runs in $O(c)$, because on one hand we have to check for every edge we insert if it's unique, which runs in $O(1)$ and on the other hand the expected number of collisions is in $O(1)$, since the error probability is constant. This results in a run-time of $O(\frac{n}{k})$ to build a block.

To generate k blocks we have a run-time in $O(\frac{n^2}{k})$, which is dominated by the initialization. Now we have to generate the edges between them. To generate one edge we have to pick two random distinct blocks from a list of possible blocks, this can be done in $O(1)$, then we select one random vertex of both blocks, also in $O(1)$ and assure that the resulting edge has not already been added. This last step runs in $O(c \cdot k)$. We can assume that the occurrence of doubly chosen edges, can be discarded since in most cases $n >> c$. Anyway the error probability is as before constant, therefore expected value for the number of repetition to choose an edge is also constant. To insert $\frac{c \cdot k}{2}$ edges we have a run-time of $O(c^2 \cdot k^2)$.

Summing up the run-times of the three phases the algorithm can be implemented with an average run-time in $O(\frac{n^2}{k} + c^2 \cdot k^2)$ which is in $O(\frac{n^2}{k})$. In Section 5.3 we discuss the run-time of an improved version of Algorithm 1.

4. Proof of Correctness

In this chapter we prove that the k-partition our Algorithm 1 generates is a *minimal* k-partition of the corresponding generated graph. We prove propositions about the minimal cut sizes for each phase of Algorithm 1, and use those to finally prove the minimality of our generated k-partition.

4.1 A Simple $2c$ Subblock

In phase 1 of Algorithm 1, the initial subblocks are small graphs with at most $2c$ vertices. To assure that their minimal cut size is at least c we make sure that each vertex has a degree of at least c.

Proposition 1. *Given a* graph $G := (V, E)$ *with* $|V| \leq 2c$ *and for all* $v \in V$ *it holds that deg*$(v) \geq c$. *Then MinCut*$(G) \geq c$

Proof. Let F_{ST} be a *cut* of G, which divides V into S and T. Without loss of generality, it is $l := |S| \leq c$. For any vertex $s_i \in S$ the following holds:

$$\deg(s_i) \geq c \wedge \deg_{G[S]}(s_i) \leq l - 1 \implies \deg_{out} = \deg_{G[T \cup \{s_i\}]}(s_i) \geq c - l + 1$$

$$\implies |F| = \sum_{s_i \in S} \deg_{out}(s_i) \geq l \cdot c - l^2 + l \geq_{l \leq c} c$$

□

Thus the $2c$ graphs we generate in Phase 1 of our Algorithm 1 each have a minimal cut size of at least c.

4.2 Joining Sub Blocks

Method A

In phase 2 of Algorithm 1, using method A, we join two subblocks by inserting at least c random edges. According to the following proposition, this provides a new subblock with a minimal cut size of at least c.

Proposition 2. *Given two disjoint sets of vertices* V_1 *and* V_2. *Let* $G := (V_1 \dot\cup V_2, E)$ *with* $d_G(V_1) \geq c$, *MinCut*$(G[V_1]) \geq c$ *and MinCut*$(G[V_2]) \geq c$. *Then, the following holds:*

$MinCut(G) \geq c$.

Proof. If we take a look at the different cuts F_{ST} of G, there are three cases.

1. $S = V_1$: in this case $|F| \geq c$ because it contains exactly the c edges of $\delta_G(V_1)$.
2. $T \cap V_1 \neq \emptyset \wedge S \cap V_1 \neq \emptyset$: $\exists s, t \in V_1 : s \in S, t \in T$. In this case F induces an s-t cut F' in $G[V_1]$ with $|F| \geq |F'| \geq c$.
3. $T \cap V_2 \neq \emptyset \wedge S \cap V_2 \neq \emptyset$: analogous to 2.

□

Hence, the new subblock satisfies our only constraint, that its minimal cut size is at least c.

Method B

Using method B as in Algorithm 3 we basically perform the same task as in phase 1. We create new subblocks using at most $2c$ old ones.

Proposition 3. *Let $G = (V, E)$ be a graph and $B := \{V_1, \ldots, V_l\}$ be a partition of V. If:*

- *$l \leq 2c$,*
- *$\forall V_i \in B : MinCut(G[V_i]) \geq c$,*
- *$\forall V_i \in B : d_G(V_i) \geq c$.*

$MinCut(G) \geq c$ holds.

Proof. Let $F := F_{ST}$ be a minimal cut of G, we distinguish two cases:

1. $F \subset \bigcup_{V_i \in B} \delta(V_i)$: In this case the same arguments as in Proposition 1 apply and $|F| \geq c$.
2. $\exists V_i \in B : \exists s, t \in V_i : s \in S, t \in T$: Let $F[V_i]$ be the cut of $G[V_i]$ that is induced by F. Because of MinCut$(G[V_i]) \geq c$, $|F| \geq |F[V_i]| \geq c$.

□

Thus, we also achieve that the minimal cut size of our new graph is at least c.

4.3 Joining Blocks

Given k blocks, each with a minimal cut size of at least c, we want to connect those blocks by generating edges between them. As we described in Algorithm 1 we achieve this by inserting $\frac{k \cdot c}{2}$ edges between the blocks. At first we describe the connection between the minimal cut size of a graph and a k-partition of this graph.

Proposition 4. *Given a* graph $G := (V, E)$ *with MinCut$(G) = c$ and a* k-partition P *of G. For the size of E_P, it holds that:*

$$|E_P| \geq \frac{k \cdot c}{2}$$

Proof. Suppose P partitions V into disjoint sets P_i, $i = 1, \dots, k$. For each P_i, the cut C_{P_i} which separates P_i and $\bigcup_{j \neq i} P_j$ has a size of at least c because MinCut$(G) = c$. Also for each edge $e = (v, u) \in E_P$ with $v \in P_i$ and $u \in P_j$ it holds that $e \in C_{P_i}$,$e \in C_{P_j}$ and $e \notin C_{P_l}$ for $l \neq i$ and $l \neq j$. Thus the sum $\sum_{P_i} |C_{P_i}| \geq k * c$ counts all edges of E_P exactly two times. This implies

$$|E_P| = \frac{\sum C_{P_i}}{2} \geq \frac{k \cdot c}{2}$$

□

We use this minimal size of E_P as an upper bound for the edge cut of the k-partition. Therefore we insert in phase 3 of Algorithm 1 at most $\frac{k \cdot c}{2}$ edges.

In conjunction with Proposition 1 and Proposition 2 and 3, in which we prove that the blocks we build in phase 1 and phase 2 of Algorithm 1 have a minimal cut size of at least c, we now prove in Proposition 5 that the k-partition we generate is in fact a minimal k-partition of the corresponding graph.

Proposition 5. *Given a* graph $G := (V, E)$ *and a set* $\mathcal{V} := \{V_1, \dots, V_k\}$*, a* k-partition *of* V *with:*

- *MinCut*$(G[V_i]) \geq c$
- $d_G(V_i) \leq 2c - 1$

Then, $\mathfrak{P} := \bigcup_{i=1}^{k} \delta_G(V_i)$ *is a minimal* k-partition *of* G *with size* $|E_{\mathfrak{P}}| \leq \frac{k \cdot c}{2}$*. An example of such a partition is illustrated in Figure 4.1*

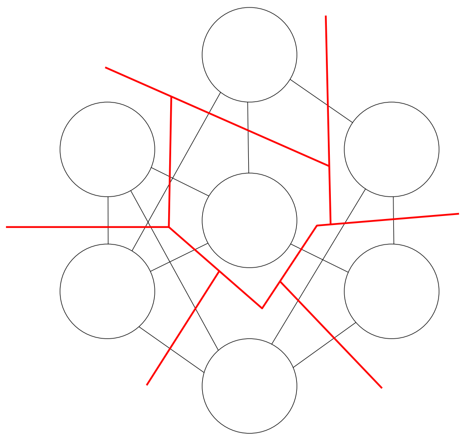


Figure 4.1: A graph $G = (V, E)$ and a 7-partition P

Proof. Suppose there is a k-partition P with $|E_P| < |E_{\mathfrak{P}}|$.

If $E_P \subset E_{\mathfrak{P}}$ there exists an edge $e = \{u, v\} \in E_{\mathfrak{P}}$ with $e \notin E_P$ and thus there exists $i \neq j$ such that $u \in V_i$ and $v \in V_j$. Since V_i and V_j are separated by $\mathfrak{P}$ but e connects them, P separates G into at most $k - 1$ subgraphs.

On the other hand, if $E_P \not\subseteq E_{\mathfrak{P}}$, there exists a $V_i \in \mathcal{V}$ and an edge $e_i \in E_P$ with $e_i \in E[V_i]$. Let us assume such an edge exists in $1 \leq l \leq k$ different $E[V_i]$. Each of those edges $e_i := \{s, t\}$ induces an *s-t cut* C_i of $G[V_i]$ (See 4.2). Because of MinCut$(G[V_i]) \geq c$, the size of any s-t cut is at least c, thus $|C_i| \geq c$.

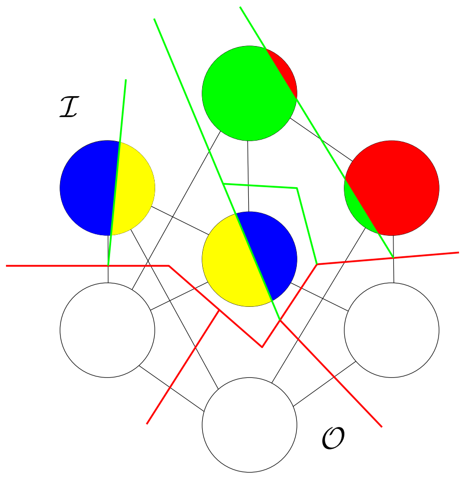


Figure 4.2: graph $G = (V, E)$ from Figure 4.1 and a candidate for another 7-partition $\mathfrak{P}$. The colored blocks belong to $\mathcal{I}$, and the color visualizes which parts belong together. The white blocks are the same as in Figure 4.1 and are element of $\mathcal{O}$.

Let $\mathcal{I} := \{V_i \in \mathcal{V} \mid E_P \cap E[V_i] \neq \emptyset\}$ be the set of V_i's which are separated by P. Let $\mathcal{O} := \mathcal{V} \setminus \mathcal{I}$ be the set of V_i's not separated by P. Further denote $V_{\mathcal{I}} := \bigcup_{V_i \in \mathcal{I}} V_i$ and $V_{\mathcal{O}:=\bigcup_{V_i \in \mathcal{O}} V_i}$.

Thus $E_l := \bigcup_{V_i \in \mathcal{I}} \delta_{G[V_{\mathcal{I}}]}(V_i)$ is the set of edges between the $V_i \in \mathcal{I}$ in the graph G reduced to $V_{\mathcal{I}}$. Since we assumed that $|\mathcal{I}| = l$ the size of E_l is at most $\frac{l \cdot (2c-1)}{2}$, as illustrated in Figure 4.2 for $l = 4$.

The remaining edges of E_P are $E_P \setminus E_{\mathcal{I}}$ and are a subset of $E_{\mathfrak{P}}$. This yields the following size of E_P:

$$|E_P| \geq |E_{\mathfrak{P}}| - |E_l| + l * c \geq |E_{\mathfrak{P}}| - \frac{l \cdot (2c-1)}{2} + l * c \geq |E_{\mathfrak{P}}| + \frac{l}{2}$$

This contradicts with $|E_P| < |E_{\mathfrak{P}}|$.

$\square$

If we allow $\lfloor \frac{|V|}{k} \rfloor \leq |V_i| \leq \lceil \frac{|V|}{k} \rceil$ we have to assure that for any V_i with $|V_i| = \lfloor \frac{|V|}{k} \rfloor + 1$ every vertex in V_i has at most $c - 1$ edges connecting itself to any other $V_j \in \mathcal{V}$.

5. Improvements

In this chapter we present two improvements of Algorithm 1 to cover the tracks of the block-building process. First we will introduce them in theory, second we will describe how they can be integrated into the algorithm and evaluate their impact on the run-time.

5.1 Breaking Locality

Due to the initial subblocks and their strong internal connectivity, we will now try to transport edges from the lower levels to the upper ones. We offer two techniques which can be combined.

5.1.1 Using Circles

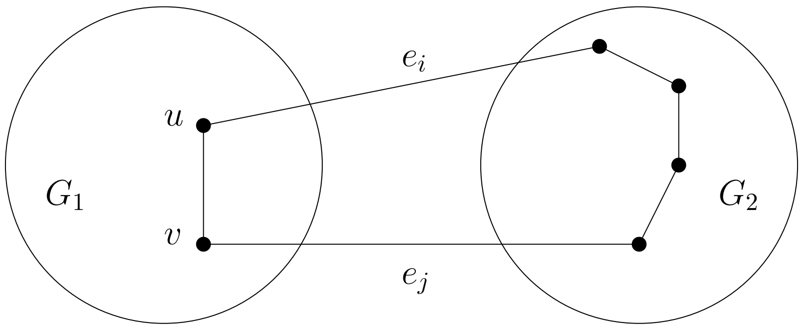


Figure 5.1: A circle generated in phase 2 of Algorithm 1.

In phase 2 of our Algorithm 1, when we join graphs G_1 and G_2 to a new graph G, a circle like in Figure 5.1 can occur. It is a circle of the form: $v, v' \in V_1$, $u, u' \in V_2$ and the edges $e = \{v, v'\}$, $e' = \{v, u\}$ and $e'' = \{v', u'\}$ exist. Furthermore a path from u to u' exists in G_2 because G_2 is connected. If such a circle occurs we may delete e from G, but only if no other edge has been deleted using a circle which contained e' or e''. The removal of such an edge does not interfere with the fact that $\mathrm{MinCut}(G) \geq c$.

Deleting Circles

Proposition 6. *Let $G = (V, E)$ and $\emptyset \neq V_1, V_2 \subset V$, with*

- $V_1 \dot{\cup} V_2 = V$
- $d(V_1) = d(V_2) = c$
- $MinCut(G[V_1]) \geq c$
- $MinCut(G[V_2]) \geq c$

Let us partition the edges in $\delta(V_1)$ into disjoint pairs , i.e. $M := \{\{e_i, e_j\} \mid e_i, e_j \in \delta(V_1)\}$ with $|M| = \frac{c}{2}$ and $\bigcup_{o \in M} o = \delta(V_1)$. This is a set of unordered tuples of edges between V_1 and V_2.
Furthermore we define a set of deletable edges

$$D := \{\{u, v\} \in E[V_1] \mid \exists \{e_i, e_j\} \in M : u \in e_i \wedge v \in e_j\}.$$

If we delete D, $MinCut(G) \geq c$ holds.

Proof. Let the set of *border nodes* of V_1 be $B := \{u \in V_1 \mid \exists e \in \delta(V_1) : u \in e\}$ and F_{ST} be a cut of G. W.l.o.g., it is $T \cap V_2 \neq \emptyset$. At first we observe two cases::

Case 1: $S \cap V_2 \neq \emptyset$ (see Fig.5.2 (*i*)): In this case $|F| \geq c$, because if we reduce F to $G[V_2]$ its a cut of $G[V_2]$ which has a size of at least MinCut($G[V_2]$), since we didn't delete any edges of $G[V_2]$. Therefore $|F| \geq \text{MinCut}(G[V_2]) \geq c$.

Case 2: $S \cap V_2 = \emptyset$: In this case we take a closer look at D:

1. $F \cap D = \emptyset$: Now either $S = V_1 \implies |F| = d(V_1) = c$ (see Fig.5.2 (*ii*)) or F can be reduced to it's cut F_{V_1} of $G[V_1]$ and since it doesn't contain any of the deleted edges $|F| \geq |F_{V_1}| \geq \text{MinCut}(G[V_1]) = c$ (see fig.5.2 (*iii*)).
2. $|F \cap D| = k \neq 0$: (see fig.5.2 (*iv*)) W.l.o.g. for a $\{u_i, v_i\} \in D$ let $u_i \in S$. Then because the cut F separates u_i and v_i, $v_i \in T$ applies. Further $\exists u_i' \in V_2 : \{u_i, u_i'\} \in \delta(V_1)$ and because of $u_i' \in V_2$, it follows that $u_i' \in T$. Hence, C also separates u_i and u_i'. Those edges $\{u_i, u_i'\} \in \delta(V_1) \cap F$ sum up to a size of k. F also induces a cut F' separating $V_1 \cap S$ and $V_1 \cap T$ in $G[V_1]$ with $|F'| \geq \text{MinCut}(G[V_1]) \geq c$. Now if we examine $F'[G[E \setminus D]]$ the cut is reduced by exactly the $|F \cap D| = k$ edges we deleted, thus $|F'[G[E \setminus D]]| \geq \text{MinCut}(G[V_1]) - k \geq c - k$. Now we see that F cuts at least k edges of $\delta(V_1)$ and $c - k$ edges of $E[V_1]$. Hence, because of $\delta(V_1) \cap E[V_1] = \emptyset$ it is $|F| \geq c - k + k = c$.

□

By deleting those edges we can decrease the internal connectivity of the joined graphs. It is also possible to additionally insert edges between G_1 and G_2 systematically to produce new circles, until we have deleted $\frac{c}{2}$ edges.

5.1.2 Triangles

Another way to break the locality after joining G_1 and G_2 to G is to create triangles like in Figure 5.3 between G_1 and G_2. To do so we pick a random edge $e = \{u, u'\}$ in G_1 and a random node v of G_2. Then we insert the edges $e' = \{u, v\}$ and $e'' = \{u', v\}$ and delete e. This modification also doesn't interfere with MinCut(G) $\geq c$.

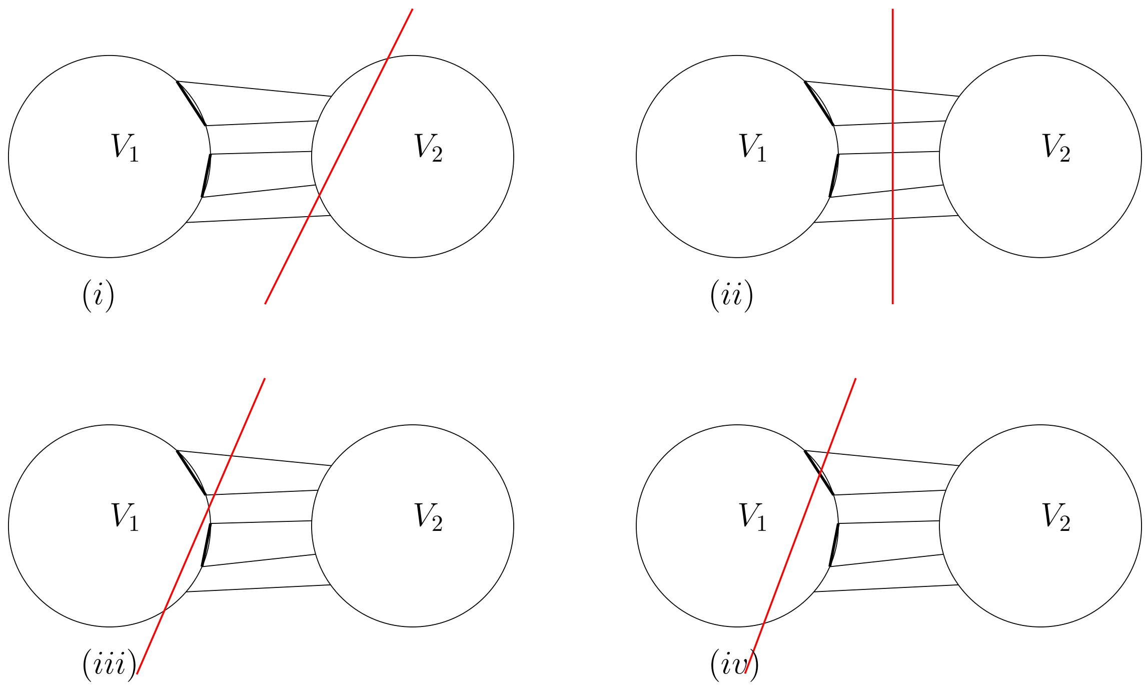


Figure 5.2: Graph $G = (V, E)$ with $V_1, V_2 \subset V$

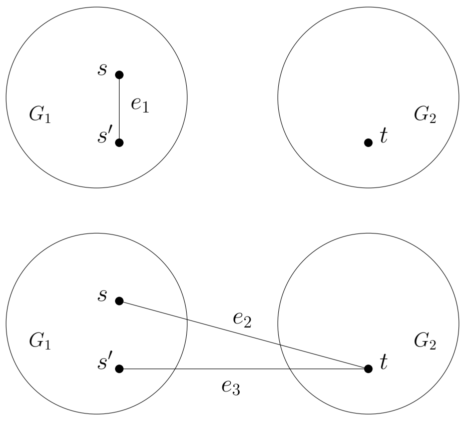


Figure 5.3: Triangle between G_1 and G_2

Inserting Triangles

Proposition 7. *Let $G = (V, E)$ be a graph, $V_1, V_2 \subset V$ with:*

(i) $V_1 \dot{\cup} V_2 = V$

(ii) $d_G(V_1) \geq c \wedge d_G(V_2) \geq c$

(iii) *MinCut*$(G) \geq c$

Given three edges $e_1 := \{s, s'\}$, $e_2 := \{s, t\}$ *and* $e_3 = \{s', t\}$ *with* $e_1 \in E$ *and* $e_2, e_3 \notin E$, $s, s' \in V_1$ *and* $t \in V_2$.

For $E' := (E \setminus \{e_1\}) \cup \{e_2, e_3\})$ *and the graph* $G' = (V, E')$ *MinCut*$(G') \geq c$ *holds.*

Proof. Let $G'' = (V, E'')$ be the graph with $E'' = E \cup \{e_2, e_3\}$. We have 2 cases:

1. **$\exists$ minimal cut F_{ST} of G'' with $e_1 \in F$:** F separates s and s'. Let $s \in S$ and $s' \in T$. Now either $t \in T$ or $t \in S$ holds. In the former case $|F| \geq c + 1$ because $d_G(S) \geq_{(iii)} c$ and $e_2 \notin E_1$. Else if $t \in S$, it follows that $e_3 \in F \implies |F| \geq c + 1$ for the same reason as before. In both cases $d_{G'}(S) \geq c$ because of $|E'| = |E''| - 1$. Therefore MinCut$(G') \geq c$ holds.
2. **$\neg\exists$ minimal cut F of G'' with $e_1 \in F$:** For any cut F' of G'' with $e_1 \in F'$ $|F'| \geq \text{MinCut}(G'') + 1$ holds. Because of $|E'| = |E''| - 1$, which means we only delete one edge, every cut is decreased by at most 1. Thus MinCut$(G') \geq c$ holds.

□

5.1.3 Combining Both

In Algorithm 5 we combine Proposition 6 and Proposition 7 by deleting an edge in each circles that occurs and using the rest of the inserted edges, which have not been incident to a deleted edge, to insert triangles. We use an existing edge between V_1 and V_2 to insert a triangle, therefore we only have to insert one additional edge. We prove that this graph still has a minimal cut size of at least c in the following two propositions.

Let $G := (V, E)$ be a graph, $V_1, V_2 \subset V$ with:

- $V_1 \dot{\cup} V_2 = V$
- $\delta(V_1) \geq c$
- $\text{MinCut}(G[V_1]) \geq c$
- $\text{MinCut}(G[V_2]) \geq c$

Furthermore, let D be a set of deletable edges as of Proposition 6 and let C be the set of edges, which have been involved in a circle.

$$C := \{e_i, e_j \in \delta(V_1) \mid \{e_i, e_j\} \in M \wedge \exists\{u, v\} \in E[V_1] : u \in e_i \wedge v \in e_j\}.$$

Let $R := \delta(V_1) \setminus C$ be the remaining set of edges between V_1 and V_2 and $G' := (V, E \setminus D)$.

Proposition 8. *In* G' *a cut* F *with* $F \cap R \neq \emptyset$ *and* $F \neq \delta(V_1)$ *has a size of at least* $c + |F \cap R|$.

Proof. Without loss of generality we assume $F \cap E'[V_1] \neq \emptyset$. We have two cases:

1. $\mathbf{F \cap D = \emptyset}$: Then $|F \cap E'[V_1]| \geq c$ since $\text{MinCut}(G[V_1]) \geq c$. Hence $|F| \geq c + |F \cap \delta(V_1)| \geq c + |F \cap R|$.

2. $|\mathbf{F} \cap \mathbf{D}| = \mathbf{k} \neq \mathbf{0}$: As we saw in the second part of Case 2 of the proof of Proposition 6 F contains at least k edges of $\delta(V_1)$ more precisely of C and $c - k$ edges of $E'[V_1]$. Therefore $|F| \geq |F \cap C| + |F \cap E'[V_1]| + |F \cap R| \geq k + (c - k) + |F \cap R| = c + |F \cap R|$.

□

Now we can combine the deletion of circles and insertion of triangles.

Proposition 9. *Let B be the set of edges eligible to insert a triangle. $B := \{\{u, v\} \in E[V_1] \mid e \in R : u \in e\}$ is a set of edges incident to an edge of R. Hence for each edge $e_i = \{u, v\} \in B$ exists an edge $e'_i = \{u, v'\} \in R$ with $u \in V_1$ and $v \in V_2$. Let $A := \{\{v, v'\} \mid \{u, v\} \in B\}$ be a set of edges, that contains an edge e''_i for every pair e_i, e'_i, so that e_i, e'_i and e''_i is a triangle as of Proposition 7.*

The graph $G'' := (V, (E \cup A) \setminus (C \cup B))$ has minimal cut size of at least c.

Proof. The graph $G' := (V, E \setminus C)$ has a minimal cut size of at least c according to Proposition 6. Therefore we have to show that this still holds if we delete B.

Suppose an *S-T cut* $F := F_{ST}$ of G'' and Let $B' := \delta_{G'}(S) \cap B$. If $B' = \emptyset$ then $F| \geq c$ since $\mathrm{MinCut}(G') \geq c$. On the other hand if $|B'| = k \neq 0$, F contains k edges of B. Since each of these edges is, by construction, part of a triangle in G'', F also contains either one edge of R or one edge of A for each of these edges.

We can divide the set of edges B' into the two subsets B'_R, which are the edges whose incident edge of the triangle that is contained in F is an element of R, and B'_A, which is defined analogously. For each edge of B'_A that has been deleted in G'' an incident edge contained in A has been added which is also contained in F, thus we decrease the size of F and increase it again. Therefore deleting the edges of B'_A doesn't alter the size of F. The edges B'_R imply that F contains $|B'_R|$ edges of R. This means that by Proposition 8 the size of $\delta_{G'}$ is at least $c + |B'_R|$, therefore if we delete B'_R the size of F is at least c. □

The last proposition proves that using the inserted edges of a fusion of two subblocks can be used to delete circles if possible and we can insert triangles using the edges, that were not involved to delete circles as we do in Algorithm 5.

5.2 Improved Algorithm

Both methods can be used in phase 2 of Algorithm 1. Since the deletion of circles (5.1.1) decreases the actual amount of edges we prefer it over inserting triangles (5.1.2), also we can use the existing edges between two newly fused subblocks to lower the amount of inserted edges to one. In Algorithm 5 we enhanced Algorithm 2.

5.3 (Improved) Run-Time

The deletion of circles and the insertion of triangles can have a huge impact on the run-time of our algorithm. While the deletion of circles, as in Section 5.1.1, only costs an additional lookup among the newly inserted edges of the current fusion process, which requires $O(c)$ steps, inserting triangles , as in Section 5.1.2, can cost much more. In our case we have to choose between run-time and required memory, which is needed to choose a random neighbor of a vertex. Either we search a neighbor by iterating through all possible neighbors in the adjacency matrix, or we have to carry along a list of neighbors for each vertex. We decided to use the first method, avoiding the overhead of keeping track of the current neighbors. Sadly, it adds a linear factor to our run-time, but our resulting

Algorithm 5: improved `fuse subblocks` method A

```
Data  : set of set of vertices subblocks, minimal cut size $c$, set of edges $E$
Result: set of edges $E$, such that MinCut($G[\bigcup_{s\in\text{subblocks}} s] + E$)
1  while $|subblocks| > 1$ do
2      pick two random elements $a, b$ of subblocks;
3      $A \leftarrow$ set of $c$ unique edges between vertices in $a$ and vertices in $b$, chosen
       uniformly at random;
4      $E \leftarrow E \cup A$;
5      foreach $e \in A$ do
6          if a circle involving $e$ exists then
7              $e' \leftarrow$ edge $e' \in A$ chosen at random, such that a circle containing $e$ and $e'$
               exists;
8              $f \leftarrow$ edge in $E$ incident to $e$ and $e''$;
9              $A \leftarrow A \setminus \{e, e''\}$;
10             $E \leftarrow E \setminus \{f\}$;
11         else
12             construct a triangle using $e$ and an edge $f \in a$ incident to $e$ chosen at
               random;
13             $e' \leftarrow$ third edge of the constructed triangle, which is incident to $e$ and $f$;
14             $A \leftarrow A \setminus \{e\}$;
15             $E \leftarrow (E \cup \{e'\}) \setminus \{f\}$;
16         end
17     end
18     subblocks $\leftarrow \big(\text{subblocks} \cup \{a \cup b\}\big) \setminus \{a, b\}$;
19 end
```

implementation still performs very well. However, if we aim to generate even larger graphs with several million vertices, we could gain a substantial speedup by implementing another data structure that keeps track of the neighbors of a vertex.

During the generation of test data we reevaluate our previous decision to store the graph structure in an *adjacency matrix*. When generating large thin graphs with few blocks our implementation of Algorithm 1 preforms very poorly. It takes long to allocate such a large *adjacency matrix* and also to choose a random neighbor of a vertex. So we now migrate to another data structure to store the graph (see 3.2). This time we chose an *adjacency list* along with an *array* tracking the *degrees* of the vertices. We implement the adjacency list as an array of linked lists, which provides looking up edges $\{u, v\}$ in $O(\max(\deg(u), \deg(v)))$ which is in $O(c)$, inserting edges in $O(1)$ and deleting edges in $O(c)$. We can still use all methods we developed using an adjacency matrix, since we only change the underlying data structure. The subblocks can still be stored in the adjacency matrix of its block, and we still profit from the fact that we fuse neighboring subblocks during the block-building phase.

Resources and Run-Time Using an Adjacency List

Using an adjacency list, we reduced the memory requirements from $O(\frac{n^2}{k})$ to $O(\frac{m \cdot n}{k})$. The impact on the run-time is also huge, especially for thin graphs. We summarize the differences compared to Section 3.3.

To initialize a block, we now have to initialize the adjacency list instead of the adjacency matrix. This runs in $O(\frac{n}{k})$ for each block. Additionally, we have to initialize the array, in which we want to store the degrees of the vertices, this also runs in $O(\frac{n}{k})$. Everything else works the same way as before, thus the initialization for all k blocks runs in $O(n)$. Generating the $2c$ graphs still runs in $O(\frac{c \cdot n}{k})$ for each block.

As before, the generation of the binary tree to obtain the fusion order runs in $O(\frac{c \cdot n}{k})$. In each of the $\frac{n}{c \cdot k}$ fusion steps the generation of the c edges runs expected in $O(c^2)$, since we have to assure uniqueness for each of the c edges we insert, as we did before. Furthermore the improvements of Algorithm 5 run in $O(c^2)$ to detect and delete circles and also $O(c^2)$ to insert the triangles. This sums up to $O(\frac{c \cdot n}{k})$ to build one block.

Therefore, the generation of the k blocks has an expected run-time in $O(c \cdot n)$, and the generation of the edges between the blocks still runs in $O(c^2 \cdot k^2)$. Thus our improved Algorithm has an *average run-time* in $O(k \cdot c \cdot n)$.

Most of the memory required by our implementation is used to store the adjacency list. Since we can generate the k blocks sequentially and independent only one block has to be stored in the main memory at a time. Such a block has a size of $\frac{n}{k}$ and the average degree of a vertex is in $O(c)$. Thus the adjacency list memory usage is in $O(\frac{n \cdot c}{k})$.

6. Implementation

In this chapter we will present details about the implementation of Algorithm 1. We implement Algorithm 1 and the improvements of Algorithm 5 in the C programming language using no external libraries. Next, we present the command line options needed by our implementation.

6.1 Input

The input needed by the implementation are the *number of vertices* n, the *number of partitions* k and the lower bound for the *minimal cut size* c. Further we supply a file name for the output file and the type of the output file.

To program is run by `./gnm n k c OUTPUT_MODE FILENAME` where:

- **n** is the number of vertices of the generated graph,
- **k** is the number of partitions of the generated graph,
- **c** is the minimal cut size of the blocks,
- **OUTPUT_MODE** is a bit mask with the following options:
  - 1: generate a .metis output file containing the graph in the metis format.
  - 2: generate a .tgf output file containing the graph in the trivial graph format.
  - 4: generate a .agr output file containing the graph in the agreed non-binary format
  - 8: generate a .clu output file containing the clustering data of the partition.
- **FILENAME** is the name of the output file.

7. Results

In this chapter we present the output of our implementation of Algorithm 1 using Algorithm 5 for phase 2. The computer on which we run our implementation has two Intel Xeon E5430 (2.66 GHz) Quad-Core processors and 32 GB RAM. We generate several graphs in order to visualize the output, present the performance and finally to test against METIS[5] and KaPPa[4]. First, introduce the programs we use.

yEd

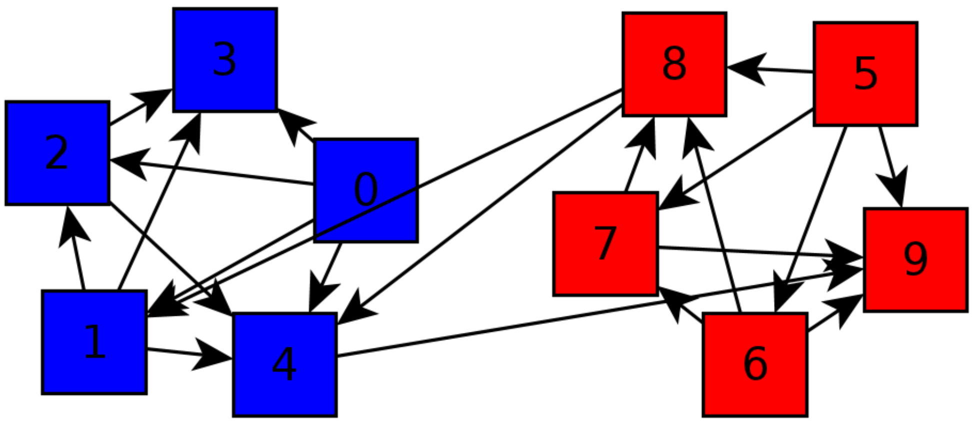
To visualize the generated graph we use the yEd Graph Editor ([11]). Furthermore to visualize the partitions we use a clustering extension for yEd, which layouts a graph according to an additional input file.

METIS and KaPPa

Due to its purpose we want to test our generated graph with a graph partitioning program. Since its one of the most common graph partitioner we test our graphs with METIS. We also text generated graphs with KaPPa which is a more advanced graph partitioner, which provides smaller edge cuts in most cases, but also requires more time to compute the partitions.

7.1 Example Output

If we generate a small graph using `./gnm 10 2 3 10 example1`, which means we generate a graph with 10 vertices, 2 blocks, and a minimal cut size of the blocks of at least 3. We visualize it using yEd, this results in the following image:



In this picture the blocks are colored red and blue. We see that each vertex has a degree of at least 3 and between the blocks are 3 edges.

The difference between the generated partition and the partition obtained from METIS has been visualized in the next example. First we see the log of the command line followed by the results created by yEd (Figures 7.1 and 7.2):

```
$ time ./gnm 5000 8 10 15 5000_8_10
generated graph with edge cut of 40
real    0m0.216s
user    0m0.180s
sys     0m0.007s

$ kMETIS 5000_8_10.METIS 8
**********************************************************************
  METIS 4.0.3 Copyright 1998, Regents of the University of Minnesota

Graph Information ---------------------------------------------------
  Name: 5000_8_10.METIS, #Vertices: 5000, #Edges: 28560, #Parts: 8

K-way Partitioning... -----------------------------------------------
  8-way Edge-Cut:     293, Balance:  1.03

Timing Information --------------------------------------------------
  I/O:                            0.010
  Partitioning:                   0.000   (KMETIS time)
  Total:                          0.010
**********************************************************************
```

In Figure 7.1 and 7.2 we can see the 8 blocks represented by the large circles, since we use a circular layout with yEd. The subblocks are the small dots inside the circles. We can see the big difference of the number of edges between the blocks. In Figure 7.1, which visualizes the 8-partition of our implementation there are 40 edges between the 8 blocks and in Figure 7.2, that is the 8-partition calculated by METIS, we can see 293 edges.

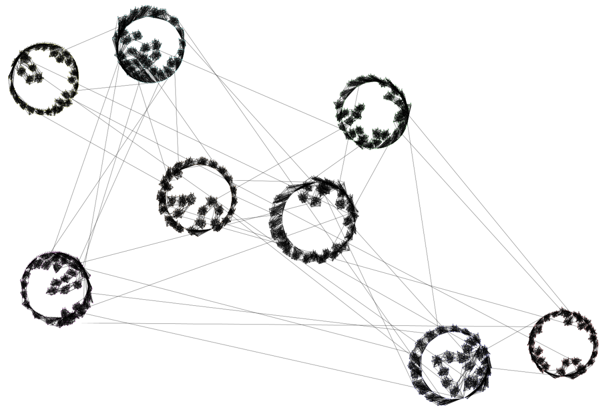


Figure 7.1: The generated graph clustered by the generated partition

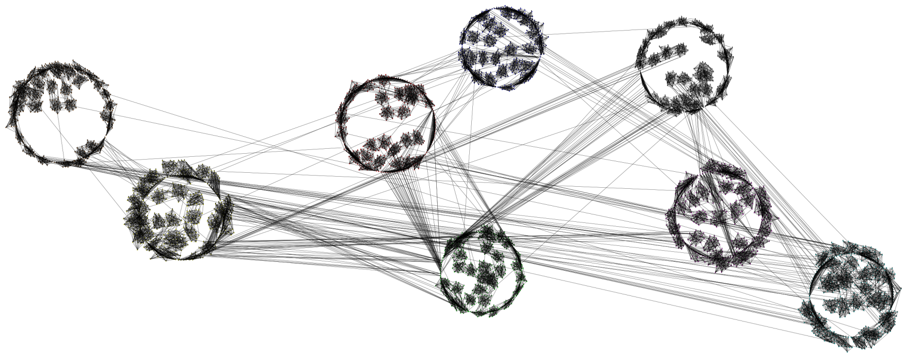


Figure 7.2: The generated graph clustered by METIS

7.2 Run-Time

The following two pictures illustrate the run-time of our implementation for the parameters $n = \{100, 1000, 10000, 100000\}$, $k = \{2, 4, 8, 16, 32, 64\}$ and $c = \{20, \sqrt{n}\}$. We run all valid combinations of the parameters 10 times. Figure 7.3 and 7.4 show the average time is takes to generate an instance. Notice that both axes have a logarithmic scale.

In the first chart we show the average time using a minimal cut size of 20 in the second chart we use a minimal cut size of $\sqrt{n}$.

n	c	2	4	8	16	32	64
100	20	0.004	-	-	-	-	-
1000	20	0.013	0.011	0.011	0.012	-	
10000	20	0.095	0.095	0.099	0.103	0.114	0.139
100000	20	1.142	1.104	1.081	1.046	1.293	1.379

n	c	2	4	8	16	32	64
100	10	0.003	0.003	-	-	-	
1000	31	0.017	0.017	0.016	0.018	-	-
10000	100	0.72	0.7	0.684	0.696	0.765	-
100000	316	56.06	56.31	56.721	57.439	59.112	59.151

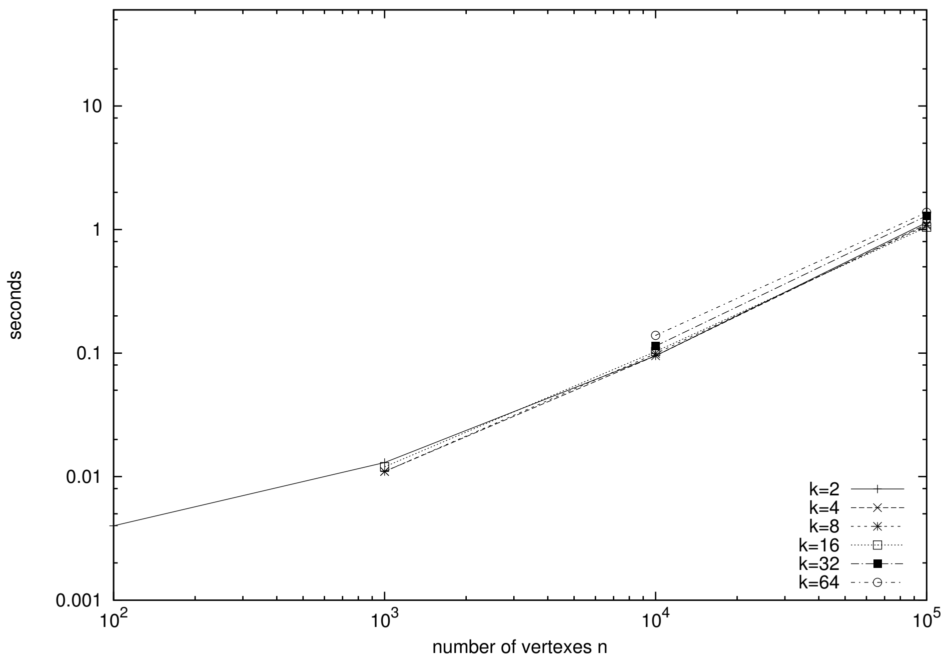


Figure 7.3: $c = 20$

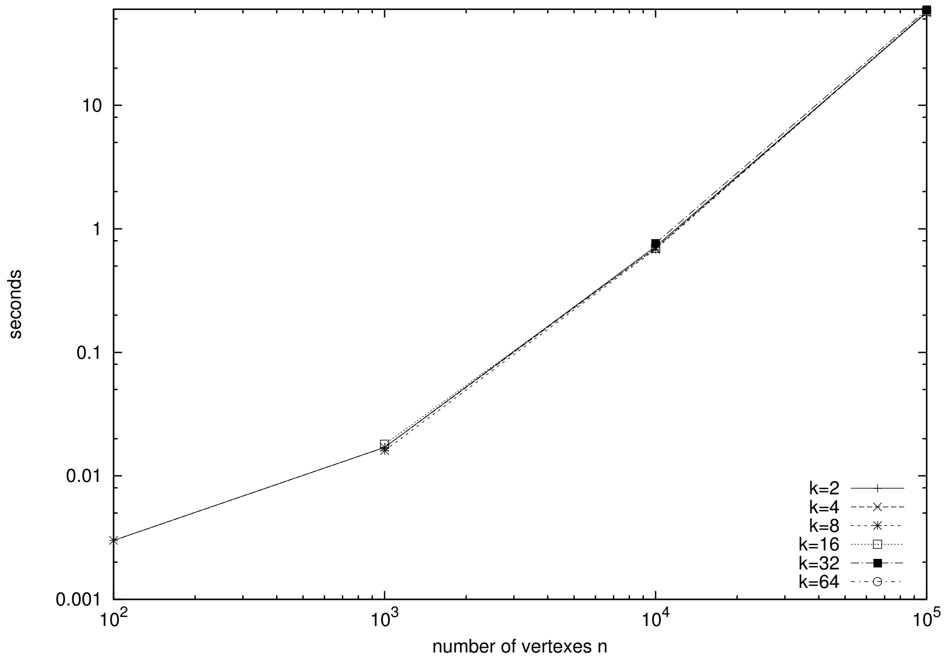


Figure 7.4: $c = \sqrt{n}$

In Figure 7.3 and 7.4 we see the linearity of the run-time.

7.3 Testing against METIS

To compare our calculated partitions against the partitions generated by METIS we generated several sets of 100 graphs, using varying parameters for each set. All valid combination of the following parameters have been used:

- $n = 100, 1000, 10000, 100000$
- $c = 10, 20, \sqrt{n}$
- $k = 2, 4, 8, 16, 32, 64$

Then we run METIS on the generated graphs, providing the number of their blocks. In the following tables we compared the results of METIS and the optimal partitions we calculated. The values in the columns labeled *opt* contain the edge cut $|E_p|$ provided by our implementation, the rows *max, min* and *avr* withhold the values of the *maximal edge cut*, the *minimal edge cut* and the *average edge cut* calculated by METIS. The columns labeled with % represent the success rate of METIS, which means in how many of the 100 test cases METIS was able to find an optimal partition.

		k = 2					**k = 4**				
n	**c**	**opt**	**max**	**min**	**avr**	**%**	**opt**	**max**	**min**	**avr**	**%**
100	**10**	*10*	26	**10**	10	95					
100	**20**	*20*	48	**20**	20	99	*20*	112	**20**	21	98
1000	**10**	*10*	81	**10**	42	19	*20*	199	**20**	51	25
1000	**20**	*20*	426	**20**	104	32	*40*	412	**40**	66	57
1000	**31**	*31*	1066	**31**	93	50	*62*	183	**62**	76	80
10000	**10**	*10*	55	**10**	29	1	*20*	223	29	108	0
10000	**20**	*20*	308	45	77	0	*40*	934	75	307	0
10000	**100**	*100*	5230	242	369	0	*200*	22985	387	2886	0
100000	**10**	*10*	67	21	35	0	*20*	183	54	102	0
100000	**20**	*20*	149	46	87	0	*40*	766	149	253	0
100000	**316**	*316*	3308	786	1962	0	*632*	5855	2396	4324	0
		k = 8					**k = 16**				
n	**c**	**opt**	**max**	**min**	**avr**	**%**	**opt**	**max**	**min**	**avr**	**%**
1000	**10**	*40*	277	**40**	66	36	*80*	315	**80**	112	28
1000	**20**	*80*	160	**80**	87	84	*160*	237	**160**	169	65
1000	**31**	*124*	185	**124**	125	97	*248*	367	**248**	255	88
10000	**10**	*40*	541	112	236	0	*80*	950	199	487	0
10000	**20**	*80*	1704	210	594	0	*160*	2267	456	1366	0
10000	**100**	*400*	1479	**400**	793	23	*800*	1583	**800**	872	76
100000	**10**	*40*	529	158	254	0	*80*	1325	349	617	0
100000	**20**	*80*	1770	305	668	0	*160*	4021	811	1573	0
100000	**316**	*1264*	64924	4336	13443	0	*2528*	310924	7194	34186	0
		k = 32					**k = 64**				
n	**c**	**opt**	**max**	**min**	**avr**	**%**	**opt**	**max**	**min**	**avr**	**%**
1000	**10**	*160*	402	**160**	198	13					
10000	**10**	*160*	1671	329	884	0	*320*	2054	660	1191	0
10000	**20**	*320*	2984	341	1533	0	*640*	4048	**640**	1489	3
10000	**100**	*1600*	**1600**	**1600**	**1600**	100					
100000	**10**	*160*	2058	796	1230	0	*320*	3273	1643	2334	0
100000	**20**	*320*	6497	1946	3344	0	*640*	10126	3864	6632	0
100000	**316**	*5056*	15127	**5056**	9589	1	*10112*	12316	**10112**	10439	8

We can see that METIS struggles with our graphs if the number of vertices reaches 10000, but manages to produce good partitions on smaller graphs with less than 1000 vertices. Also, the discrepancy of the edge cut for graphs generated with the same parameters can be large.

7.4 Testing against KaPPa

The output data of Section 7.3 we test with KaPPa. We use the algorithms *kaffpaeco* and *kaffpastrong* of KaPPa. Since *kaffpaeco* profits from the our continuous vertex labels it was able to calculate almost every optimal partition of the test data. A permutation of the vertex labels could be implemented in a future work to receive more significant conclusion. Therefore we present the result of *kaffpastrong* in the following chart. For graphs with less than 10000 vertices KaPPa was able to calculate the optimal partition, hence they don't appear in the chart.

		k = 2					**k = 4**				
n	**c**	**opt**	**max**	**min**	**avr**	%	**opt**	**max**	**min**	**avr**	%
10000	**20**	*20*	**20**	**20**	**20**	100	*40*	113	**40**	81	7
10000	**100**	*100*	252	**100**	171	51	*200*	669	**200**	368	31
100000	**10**	*10*	**10**	**10**	**10**	100	*20*	37	**20**	21	95
100000	**20**	*20*	49	**20**	21	95	*40*	92	**40**	49	73
100000	**316**	*316*	2700	781	1594	0	*632*	6165	3311	4554	0
		k = 8					**k = 16**				
n	**c**	**opt**	**max**	**min**	**avr**	%	**opt**	**max**	**min**	**avr**	%
10000	**10**	*40*	94	**40**	62	16	*80*	258	162	214	0
10000	**20**	*80*	416	212	416	0	*160*	662	390	548	0
10000	**100**	*400*	1169	**400**	595	32	*800*	993	**800**	804	98
100000	**10**	*40*	83	**40**	52	42	*80*	188	**80**	151	1
100000	**20**	*80*	261	**20**	168	3	*160*	645	347	544	0
100000	**316**	*1264*	10624	6842	9058	0	*2528*	21350	16321	19327	0
		k = 32					**k = 64**				
n	**c**	**opt**	**max**	**min**	**avr**	%	**opt**	**max**	**min**	**avr**	%
10000	**10**	*160*	528	292	440	0	data was not provided				
10000	**20**	*320*	1079	597	863	0	data was not provided				
10000	**100**	*1600*	**1600**	**1600**	**1600**	100					
100000	**10**	*160*	468	300	394	0	*320*	1081	786	926	0
100000	**20**	*320*	1673	1186	1368	0	*640*	3120	2512	2871	0
100000	**316**	*5056*	18583	11335	14943	0	*10112*	172951	**10112**	17518	27

We can see that KaPPa provides especially for $k = 2$ and $k = 4$ much better results than METIS, also for $k = 8$ KaPPa finds optimal partitions in several cases. In many cases KaPPa calculates partitions with a smaller edge cut than METIS does. But for large graphs with 100000 vertices and more than 8 blocks KaPPa fails to calculate optimal partitions.

8. Conclusion

We developed an algorithm whose run-time is linear in the number of edges, which generates a graph and an optimal partition of this graph. The graphs we are able to generate are random, but we cannot cover every possible graph. Our algorithm provides a skeletal structure to build a graph with guarantees on its optimal partition costs. However, even those graphs we are able to generate already provide a challenge for the well known partitioner METIS[5], at least if we consider large graphs with more than 10000 vertices. Also KaPPa[4] fails most the time with respect to graphs containing 100000.

Our implementation of the graph generator performs and scales very well. Even on a normal laptop we are able to generate graphs with 100000 vertices and several million edges in under a minute.

Future Work

Many enhancements can be done to vary the types of graphs that can be generated. The simplest way to achieve this would be to insert a random amount of edges chosen randomly into the blocks we create during the generation of a graph. Furthermore, the structure of the graphs we generate is very special in the sense that the blocks of the partition are very dense compared to the cut. Another future goal could be to be able to generate graphs, which are more similar to real world data.

Bibliography

[1] A.-L. Barabasi and R. Albert. Emergence of scaling in random networks. *ArXiv Condensed Matter e-prints*, October 1999.

[2] P. Erdos and A. Renyi. On random graphs i. *Publ. Math. Debrecen*, 6:290, 1959.

[3] E.N. Gilbert. Random graphs. *Ann. Math. Stat.*, 30:1141–1144, 1959.

[4] Manuel Holtgrewe, Peter Sanders, and Christian Schulz. Engineering a scalable high quality graph partitioner. *CoRR*, abs/0910.2004, 2009.

[5] George Karypis and Vipin Kumar. Multilevelk-way partitioning scheme for irregular graphs,. *Journal of Parallel and Distributed Computing*, 48(1):96 – 129, 1998.

[6] Manuel Krings. Simultane Schnitte in Graphen, September 2009.

[7] Alexander Schrijver. *Combinatorial optimization*, volume A: Paths, flows, matchings, chapters 1 - 38 of *Algorithms and combinatorics ; 24*. Springer, Berlin, 2003.

[8] A.J. Soper, C. Walshaw, and M. Cross. A combined evolutionary search and multilevel optimisation approach to graph-partitioning. *Journal of Global Optimization*, 29:225–241, 2004. 10.1023/B:JOGO.0000042115.44455.f3.

[9] C. Walshaw. The graph partitioning archive. `http://glaros.dtc.umn.edu/gkhome/views/metis`.

[10] D. J. Watts and S. H. Strogatz. Collective dynamics of ‘small-world’ networks. *Nature*, 393:440–442, June 1998.

[11] yWorks GmbH. yed graph editor. `http://www.yworks.com/en/products_yed_about.html`.