Politechnika Warszawska
WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

Instytut Informatyki

Praca dyplomowa inżynierska

na kierunku Informatyka
w specjalności Inżynieria Systemów Informatycznych

Implementation of the de novo genome assembler in the Rust programming language

Łukasz Neumann
Numer albumu 261479

promotor
dr hab. inż. Robert M. Nowak

WARSZAWA 2017

Implementation of the de novo genome assembler in the Rust programming language

Abstract

This thesis describes the design and the implementation of the de novo sequence assembler, written in the Rust programming language. The assembler is designed with respect to future parallelization of the algorithms, run time and memory usage optimization, and exclusive RAM usage. Moreover the application uses new algorithms for the correct assembly of repetitive sequences.

Performance and quality tests were performed on various data, showing that the new assembler improves run time and memory usage in comparison to the `dnaasm`, `ABySS` and `Velvet` genome assemblers. Additionally, benchmarks indicate that the Rust-based implementation is comparable to the `dnaasm` project written in C++ concerning quality of created contigs, while outperforming it in terms of assembly time and memory usage.

Quality tests indicate that the new assembler creates more contigs than well-established solutions, but the contigs have better quality with regard to mismatches per 100kbp and indels per 100kbp. Furthermore N50 statistics of created contigs are similar between different assemblers. All assemblers yield the same longest contig for each dataset.

Keywords: DNA assembling, contigs, optimization, Rust

Implementacja asemblera genomu w języku programowania Rust

Streszczenie

Niniejsza praca opisuje projekt i implementację asemblera DNA dla sekwencerów nowej generacji. Asembler został napisany w języku Rust. Jego architekturę oparto o możliwość zrównoleglenia algorytmów, optymalizację czasu działania i używanej pamięci, oraz wyłączne użycie pamięci RAM. Dodatkowo aplikacja pozwala na poprawne odtwarzanie powtarzalnych sekwencji. Program inspirowany był istniejącym asemblerem `dnaasm`.

Przeprowadzone zostały testy jakości i wydajności dla kilku wybranych organizmów, porównując nowo powstały asembler z 3 innymi rozwiązaniami. Wyniki pokazują, że nowy asembler jest wydajniejszy od asemblera `dnaasm`, zarówno pod względem używanej pamięci, jak i czasu działania. Ponadto stwierdzono porównywalność jakości wyników aplikacji zaimplementowanej w języku Rust z aplikacją `dnaasm`, napisaną w języku C++, przy jednoczesnym mniejszym czasie działania i ilości zużytej pamięci.

Testy jakości dowiodły, że nowy asembler tworzy więcej kontigów niż wiodące rozwiązania na rynku, jednakże tworzone kontigi mają lepszą jakość pod względem kryteriów `mismatches` oraz `indels` na 100 tys. par zasad. Dodatkowo statystyki N50 tworzonych kontigów są podobne do wyników innych asemblerów. Wszystkie asemblery utworzyły identyczne najdłuższe kontigi dla wszystkich danych testowych.

Słowa kluczowe: asembler DNA, kontigi, optymalizacja, Rust

Contents

1 Introduction

1.1 Goals

The goal of this work was to propose redesign of the in-memory de novo genome assembler for next generation sequencers. Main assumptions about the new design are as follow:

- efficient memory layout — lowering the memory usage to the minimum;
- on-par speed — memory optimization should not hurt the performance of the assembler, as tested against previous implementation;
- support for parallelism/concurrency in the algorithms within assembler;
- modular design allowing user to easily change the internal layout of the assembler.

1.2 Structure

This thesis is divided into 7 chapters:

1. Introduction — goals and the structure of the thesis;
2. DNA Assembling — overview of the assembling techniques and de novo assemblers, depiction of the data structures used throughout the project;
3. Algorithms — definitions of new algorithms introduced in this thesis;
4. Implementation — language, tooling and layout of the created assembler. Contains statistics for the code as well as testing summary;
5. Optimization — description of memory and run-time optimizations implemented in the project;
6. Evaluation — quality and performance assessments of the implemented assembler on different datasets;
7. Summary — conclusions and future work sketch.

Additionally Appendix A is provided, which holds the user manual for the created assembler.

2 DNA assembling

Current technology does not allow for reading entire genome at once, but rather it reads shorter fragments of the DNA sequence, referred to as 'reads'. DNA assembling is a process of aligning and merging reads in order to create longer sequences, ideally representing entire chromosomes. There are two main approaches to reading DNA data.

Whole genome shotgun sequencing
Because of read length limitation, one of the most popular techniques used to obtain genome data is shotgun sequencing genomic DNA [1], [2]. This technique randomly divides DNA data into fragments small enough to get its symbols and create reads. Division and read is performed multiple times to statistically ensure that read data is of specific coverage. Coverage is defined as an average number of reads representing given nucleotide. In this thesis I will be describing assemblers using shotgun sequencing technique.

Hierarchical shotgun sequencing
Hierarchical shotgun sequencing first divides the DNA data into larger fragments (roughly 50-200 kb) of known order and then uses shotgun sequencing to sequence each fragment. It is worth noting that in order to sort these fragments, special libraries are used. While this technique relies less on the computational algorithms in the assembler, it requires already created libraries and is generally slower than whole genome shotgun sequencing technique.

Sequence reading techniques
Independently from DNA sequencing technique used, it is important to choose proper technique for reading the sequence from the organism. Currently there are three leading techniques, each with its own advantages and disadvantages:

- Sanger sequencing — older technique, which provides high accuracy reads over longer lengths, but is expensive and slow;
- Next Generation Sequencing (NGS) — fast and cost effective method, provides high output, with lower quality. Usually paired with high coverage to ensure fixed accuracy;
- Single Molecule, Real-Time (SMRT) — offers longer reads which come with a cost of more errors in the reads, when compared to NGS and Sanger.

2.1 DNA Assemblers

There are two main approaches to genome assembly.

2.1.1 Mapping assembly

Mapping assemblers use reference genome, against which they align data from the sequencers. Reads are independently assigned the most likely position in the genome. Mapping alignment does not assume any synergy between reads. While this technique can be efficient, it has some major downfalls, such as:

- inability to handle situations in which reference genome contains duplicate regions;
- different strategies when read has multiple, equally valid positions of placement;
- reads which vary greatly from the reference genome will not be aligned properly.

Despite these pitfalls, there are different implementations of mapping assemblers, most notably:

- BLAST [3];
- BWA [4];
- Bowtie 1 + 2 [5], [6].

2.1.2 De Novo assembly

De Novo assembly process does not require any reference material to assemble genetic material. Such assemblers use graph structures to merge reads of nucleotides sequences into bigger sequences called contigs. One of the most popular graph structures used by different implementations of de novo assembly process is De Bruijn graph. De Bruijn graph is a directed graph, which represents overlaps of sequences of nucleotides. De Bruijn graph is defined as a complete graph of the overlapping sequences, and so most assemblers use the subgraph of the De Bruijn graph, called the Pevzner graph. Despite this fact most of the sources refer to this subgraph as De Bruijn graph, and I will also follow this convention to provide consistency with the established terminology. Second important thing to note here is that canonically De Bruijn graph stores sequences in its nodes, whereas solution used by my implementation stores sequences in the edges of the graph.
Subsequences which are building blocks for De Bruijn graphs are called 'k-mer's, where k is the length of the subsequence, typically set by the user. They are created by a sliding window on the given read, as depicted on Figure 1. Single k-mer denotes an edge in the De Bruijn Graph, in which substring of length $k-1$ beginning at the offset 0 represents source node and substring of length $k-1$ starting at the offset 1 represents target node of the edge. Each edge has the counter denoting number of its occurrences in the input data. These counters are further used to eliminate erroneous reads, created by the imperfect nucleotide-reading technique. On average, after graph is fully built, edge should have the weight of the coverage of input data. Example of creating both source and target node, with the edge connecting the two are shown on the Figure 2. Red and yellow colors mark symbols exclusive to source and target nodes respectively, while blue color marks the common part of k-mer.

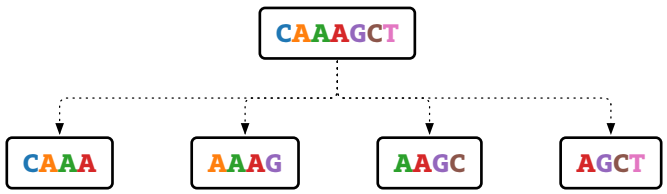

Figure 1. Creation of k-mers from a single read. K-mer size is 4.

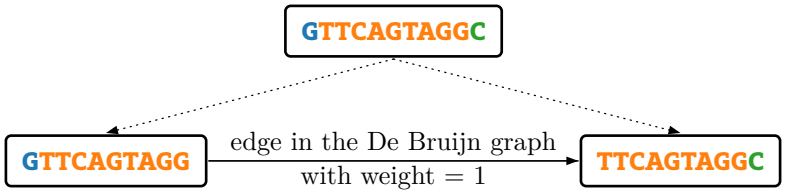

Figure 2. Source and target node, with respective edge, derived from the original k-mer of length 11.

Graph is created from reads, using the Algorithm 1.

Algorithm 1. De Bruijn graph creation

Require: *input* is the array of reads
Require: k is the length of *kmer*
Require: Length of each read in the *input* is greater than or equal to k

```
1  initialize graph
2  for each read in input do
3      n ← len(read)
4      for i ← 0 to (n − k + 1) do
5          kmer ← read[i..(i + k − 1)]
6          source_node ← kmer[0..(k − 2)]
7          target_node ← kmer[1..(k − 1)]
8          edge ← CREATE_EDGE(source_node, target_node)
9          if source_node not in graph then
10             ADD_NODE(graph, source_node)
11         if target_node not in graph then
12             ADD_NODE(graph, target_node)
13         if edge not in graph then
14             ADD_EDGE(graph, edge)
15         else
16             UPDAGE_WEIGHT(graph, edge)
```

Example graph created from the set of given reads can be seen on Figure 3.

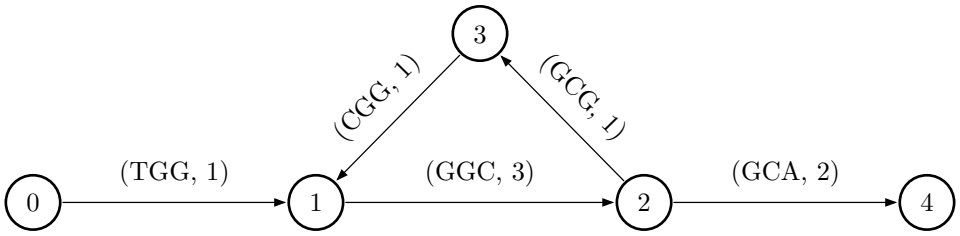

Figure 3. Example graph for k-mer of size 3, created from reads: 'TGGCG', 'CGGCA', 'GGCA'.

Main advantage of De Novo assembly, as opposed to reference mapping, is its ability to assemble DNA data of organisms, for which there are no known reference genomes created. Repeated sequences can also be assembled properly. The biggest problem associated with the De Novo assembly technique is its extensive memory usage.

2.2 Repetitive sequences

One area in which most well-known assemblers fail to deliver long, uncut contigs are organisms with repetitive sequences in their genome. This fact is a consequence of how graph is turned into contigs.
In the classic implementation of DNA assembler based on the De Bruijn graph, after the graph

has been built and corrected all nodes, which have incoming degree (number of incoming edges) or outgoing degree (number of outgoing edges) greater than 1, are called ambiguous nodes. Each ambiguous node marks a place in which contig is ended, and new contigs are started during the contig creation.
Repetitive sequences are represented in the De Bruijn graph as cycles. This means that even if such cycle is a loop, one of its nodes will be marked as ambiguous and in turn result in several contigs.
Solution to this problem was described by R. M. Nowak [7]. According to that paper it is possible to prevent artificial breakage of the contig by changing the definition of node's ambiguity.

2.3 *katome* DNA assembler

Based on the new definition of ambiguity the application called *dnaasm*[8] was developed on Warsaw University of Technology, created by Wiktor Kusmirek and Robert Nowak. While *dnaasm* correctly assembles genomes with repetitive regions it does however suffer from the extensive memory usage. While most of the De Novo assemblers use persistent, on-disk graph structures, *dmaasm* stores entire graph in RAM. This approach makes the application sufficiently faster, as it does not have to perform any I/O operations interfacing hard disks. However *dnaasm* is not optimized in terms of memory usage, which can be observed during assembling of organisms with complex genomes (for an example of its scaling refer to Section 4.2). It is also not designed to support parallelism or concurrency throughout assembling process.

In order to mitigate these issues I have implemented a new application called *katome*. It strives to be highly efficient both in terms of memory, as well as in terms of run time. Its design is modular, with rich API, allowing users to easily implement their own representation of graph, data filters and custom implementations of algorithms. It also enables easy creation of new assemblers using provided algorithms and collections as basic building blocks, effectively creating a custom pipeline of data transformation.

Moreover *katome* has its data representation guarded by a synchronization primitive, making it easy to share across multiple threads/processes. Application is implemented in the Rust programming language, which by design guarantees memory safety.

katome uses similar algorithms to the ones introduced by *dnaasm*, with two exceptions, which are described in the following sections.

2.4 Shrinking algorithm

After graph is built, pruned and standardized, it should be collapsed. Before collapsing can take place the graph should be reduced in size. Shrinking is a simple technique, which greatly simplifies underlying graph representation while maintaining information necessary to create contigs. In order to describe shrinking I introduce the concept of strand in graph. To define strand I present Property 2.4.1 and Property 2.4.2. strand is defined in Definition 2.4.1.

Property 2.4.1

We say a cycle C has Property 2.4.1 if it has at most 1 vertex which has an edge not included in C.

Property 2.4.2

Let X be a directed path of the form $v_0e_0v_1e_1\cdots e_{n-1}v_n$. We say that X has Property 2.4.2 if the following conditions are met:

- Each vertex in $v_0v_1...v_{n-1}$ has exactly one outgoing edge;
- Each vertex in $\{v_1, v_2, ..., v_n\}$ has exactly one incoming edge.

Definition 2.4.1

Let $G = (V, E)$ be a directed graph. We call directed path S in G a **strand** if it is a directed path such that the following holds:

- if S is a cycle, then S must satisfy Property 2.4.1;
- if S is not a cycle, then S must satisfy Property 2.4.2.

Examples of strands are shown in Figure 4.

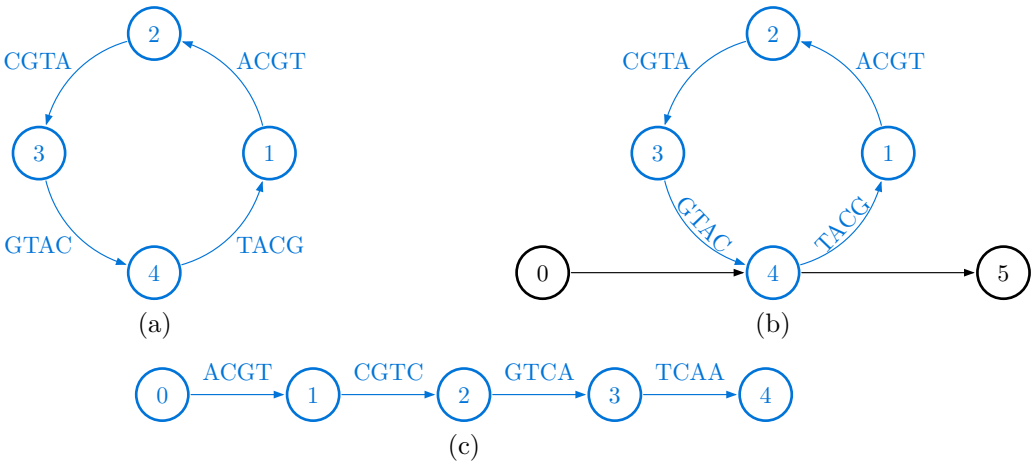

Figure 4. Examples of strands

Shrinking algorithm maps strands with at least 3 nodes into a single edge with source and target nodes. It is described in Algorithm 2. Shrunk representation of strands shown in Figure 4 is presented in Figure 5.

Algorithm 2. Graph shrinking

```
Require: graph is the built De Bruijn Graph
Require: Sequences are stored in edges
Require: k is the length of the k-mer
1  procedure SHRINK_GRAPH(graph)
2      k1_size ← k − 1
3      for each vertex in graph do
4          if OUT_DEGREE(vertex) = 1 and IN_DEGREE(vertex) = 1 then
5              base_edge ← vertex.in_edge
6              out_edge ← vertex.out_edge
7              source ← base_edge.source
8              target ← base_edge.target
9              if source ≠ target then
10                 tmp_seq ← base_edge.sequence
11                 tmp_seq.extend(out_edge.sequence[k1_size..])
12                 REMOVE_EDGE(base_edge)
13                 REMOVE_EDGE(out_edge)
14                 REMOVE_NODE(vertex)
15                 ADD_EDGE(source, target, tmp_seq)
```

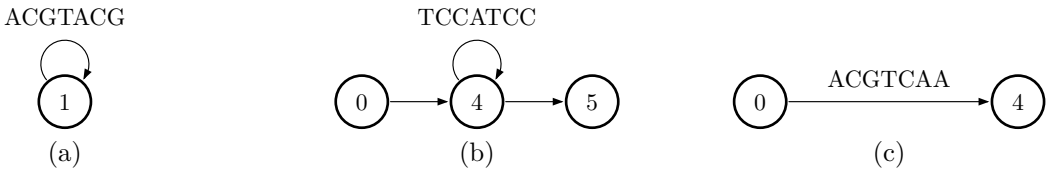

Figure 5. Examples of shrunk strands

2.5 Collapsing algorithm

To create the result of the assembly collapsing algorithm is used. It converts the shrunk graph to build a set of resulting contigs. In order to present collapsing algorithm I need to introduce definitions of two different loops, namely self-loop (Definition 2.5.1) and simple-loop (Definition 2.5.2).

Definition 2.5.1

Let $G = (V, E)$ be a directed graph. Let $v \in V$ and $e \in E$, we call the pair (v, e) a **self-loop** if the source and target of e is v.

Definition 2.5.2

Let $G = (V, E)$ be a weighted, directed graph, with edge weights $w : E \to \mathbb{R}$. Let X be a directed path of the form $v_0e_0v_1e_1v_0$. We call the path X in G a **simple-loop** if the following conditions are met:

- $w(e_0) > w(e_1)$;
- node v_1 has exactly one incoming edge and two outgoing edges;
- node v_0 has exactly two incoming edges and one outgoing edge.

Collapsing algorithm assumes that contigs in the graph are standardized (standardization algorithm introduced by W. Kusmirek [8]), and that the graph is shrunk prior to collapse. Algorithm 3 describes the process of collapse. In the core of the algorithm each considered node is checked, if it is ambiguous. If it has not been marked as one, then we need to determine whether it is a part of a simple-loop or a self-loop. If it is, then algorithm should collapse such loops without breaking the contig, otherwise the classic approach should be used.

Algorithm 3. Graph collapse

```
Require: graph is shrunk
Require: All contigs in graph are standardized
Require: All nodes are marked as non-ambiguous
1  function COLLAPSE_GRAPH(graph)
2      contigs ← []
3      loop
4          externals ← All nodes with in degree 0, or nodes in the highest topologically sorted cycle
           for any weakly connected component which doesn't contain nodes of in degree 0
5          if externals is empty then break
6          for each node in externals do
7              tmp ← CONTIGS_FROM_NODE(graph, node)
8              EXTEND(contigs, tmp)
9              REMOVE_SINGLE_NODES(graph)
10     return contigs
11 procedure ADD_CONTIG(contigs, contig)
12     if contig ≠ [] then
13         PUSH(contigs, contig) ▷ Add contig to contigs
14         contig ← []
Require: k is the size of k-mer
15 function CONGIGS_FROM_NODE(graph, node)
16     contig ← []
17     contigs ← []
18     current_node ← node
19     in_num ← in degree of node
20     loop
```

```
21      out_num ← out degree of current_node
22      if out_num = 0 then
23          ADD_CONTIG(contigs, contig)
24          return contigs
25      current_edge ← first outgoing edge of current_node
26      if current_edge is marked as ambiguous then
27          ADD_CONTIG(contigs, contig)
28      else
29          switch (in_num, out_num)
30              case (2, 1)
31                  if current_vertex hasn't got self-loop then
32                      ▷ get the e_0 from the simple-loop
33                      second_edge ← SIMPLE_LOOP(graph, current_edge_index)
34                      if second_edge ≠ ∅ then
35                          INSERT(ambiguous_nodes, current_vertex)
36                      ADD_CONTIG(contigs, contig)
37              case (1, 2) ‖ (2, 2)
38                  if current_vertex has self-loop then
39                      current_edge_index ← self_loop
40                  else
41                      INSERT(ambiguous_nodes, current_vertex)
42                  ADD_CONTIG(contigs, contig)
43              case (0, 1) ‖ (1, 1) ▷ do nothing
44              case default
45                  INSERT(ambiguous_nodes, current_vertex)
46                  ADD_CONTIG(contigs, contig)
47      ▷ Add remainder of the edge's symbols to the current contig.
48      EXTEND(contig, current_edge.sequence[k_size..])
49      target ← current_edge.target
50      in_num ← in degree of target
51      if second_edge ≠ ∅ then
52          EXTEND(contig, second_edge.sequence[k_size..])
53          DECREASE_WEIGHT(second_edge)
54      DECREASE_WEIGHT(current_edge)
55      current_node ← target
56  return contigs
```

Example of the collapsing algorithm usage is shown in the Figure 6. Notably assemblers without the support for repetitive sequences would yield 6 contigs instead of one.

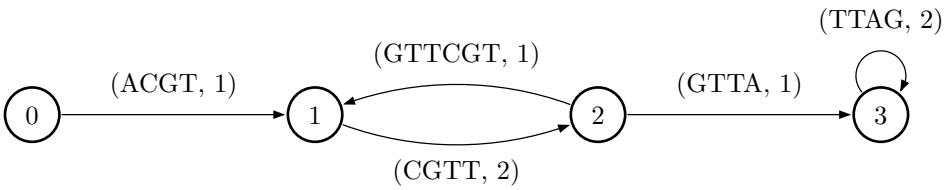

Figure 6. Example graph, each edge contains a sequence and a weight, collapsing it should yield the contig 'ACGTTCGTTAGG' for k-mer size 4.

3 Implementation

3.1 Language and tooling

katome project has been entirely written in the Rust language. Below I give a short overview on Rust, its main features and reasons why I chose it for my project.

3.1.1 Rust programming language

Rust is a new programming language introduced by Mozilla Research. It is built around three main concepts: memory safety, concurrency and speed. The first stable release of the Rust [9] compiler was introduced on May 15th, 2015. It is currently developed in 6-week cycle, in a 3-staged approach. At the end of each cycle, a new stable version of the compiler is released. The Rust language is moderated by the entire community via the process of RFCs (Request For Comments). The fast pace of development, together with the unique moderation approach, makes Rust less prone to stagnation. This is an unusual way of creating the language, as most well established languages are governed by special committees [10], [11] and new versions of standards are released less frequently. Despite a rapid release cycle, Rust is meant to be a stable language, which indicates stability and a lack of breaking changes between stable releases of the compiler.

3.1.2 Rust's main features

Memory safety

Memory safety has always been a problem in low-level languages like C or C++. It influences both security of the programs [12], as well as soundness of the written algorithms [13]. This problem can also be triggered by compilers themselves:

> (...) conventional compilers can, and do, on rare occasions, introduce data races, producing completely unexpected (and unintended) results without violating the letter of the specification.

as described by H.-J. Boehm and S. V. Adve [14]. Rust resolves these problems during compilation by introducing a new module called the 'borrow checker'. The Rust language enforces several axioms:

- all data is immutable by default;
- there is always exactly one owner of the piece of data;
- if there is an active mutable reference, then nobody else can have active access to the data;
- if there is an active shared, immutable reference, then every other active access to the data is also a shared, immutable reference.

These rules allow the borrow checker to satisfy memory safety (most notably it eliminates errors such as buffer overflow, use-after-free, double dereference, dereferencing null pointer). Notably, this solution is based purely on compile-time static analysis and does not introduce run-time overhead. Such an approach leads to very high performance implementations. It also enforces

a sound memory model of the written algorithm, because any error detected by the borrow checker will fail the compilation process.

Strict typesystem
Rust's type system is inspired by the ML family of languages. Notable mentions among its features are:

- algebraic data types;
- pattern matching;
- generics;
- traits;
- automatic type inference for variables.

Traits are similar to the concept of interface known in C/C++, but are baked into the typesystem itself. They allow programmer to enforce the specific functionality for a given type in generic functions/methods. User-defined types are called `structs` in the language, and are similar to the C structures. Rust does not have a concept of inheritance, but similar effects can be achieved by trait compositions.
The language's type system is also used by the borrow checker to enforce a lack of data races in the written code. It does so by introducing two marker traits: `Send` and `Sync` which indicate that any type implementing them is safe to share between threads in the described manner.

Performance
The Rust language is competitive against the C family of languages in terms of its performance [15], [16]. As described in Section 3.1.2, Rust's memory model makes it possible for the developer to use multi threading without the burden of typical problems that occur in multi threaded code bases - most notably data races.

3.1.3 Development environment

Cargo
Cargo is a package manager and build tool for Rust. It was created in order to standardize management of external libraries during compilation of the project. Unlike similar tools in C++ it can automatically fetch dependencies either from `Crates.io` or provided URLs. Once fetched, dependencies will get compiled and linked automatically against the project, using the local compiler. Dependencies are described by the local file `Cargo.toml` in TOML language [17]. Cargo supports versioning.

katome uses Cargo as it's building tool. It is also used to compile and run tests, format the code with respect to the specified style and perform a static analysis of the code. Cargo enables easy cross-compilation of both libraries and binary executables for officially supported platforms [18].

Crates.io
Libraries in Rust are called `crates`. Crates.io is the global repository of crates, which allows tools like Cargo for trivial resource management. It supports versioning. One of its main features

is immutability - once a crate is published it cannot be removed. This fact guarantees non-breakage of existing sources, which rely on specific crates being hosted on `Crates.io`.

3.1.4 Crates

In this section I list notable crates, without which implementation of *katome* would have been unachievable, or significantly harder, in the limited time span of this project.

petgraph

`petgraph` [19] is a graph structure library. It provides a clean API for graph creation/modification, as well as various graph algorithms. Its `Graph` structure is the core of `PtGraph` collection in *katome* sources.

fixedbitset

`fixedbitset` [20] provides simple implementation of the fixed-size set of bits. Its main features are low memory overhead and functional API. Some of the methods in the API have been implemented as the part of this thesis.

metrohash

To achieve the best possible performance for hashing compressed data (used in various hash maps and hash sets throughout modules and collections in *katome*) I settled upon `metrohash` [21] (Rust implementation [22]) hasher, as it provides the lowest run-time from all of the non-cryptographic hash functions that I tested. Tests have been devised on a fixed number of test organisms (see Section 4.2). Other tested hash functions are xxHash, SeaHash, FarmHash, SipHash, FnvHash.

parking_lot

To allow parallel/concurrent algorithms for being implemented within the *katome* library, the global vector of read data (detailed description in Section 3.2.1.1 and Section 3.5.1) must be synchronized between threads/processes. In Rust all synchronization primitives are wrappers around data to synchronize, rather than being standalone objects (although they can be used as classic synchronization primitives, by using them with `()` type). This means that global vector must be wrapped in some kind of synchronization primitive, which will be accessed every time the vector is being read or written to. Due to the fact that various algorithms heavily rely on the data in the global vector during graph creation and collapse, it is crucial that chosen synchronization primitive has the best possible performance, as even for single thread programs this primitive needs to be locked/unlocked on each access. I decided to use multiple readers/single writer lock (`RwLock`) to exploit the fact, that vector is more often read than written to. Rust's standard library contains an implementation of this lock, however I found it lackluster in terms of efficiency. `parking_lot` [23] crate provides its own implementation of `RwLock`, which has proven to be significantly faster.

3.2 Architecture

katome is divided into library and binary executable. This design allows users to create different front ends to the `katome` assembler, possibly using different implementers of the `Assemble` trait.

3.2.1 Library

katome has a modular design. The entire library is written around the concept of De Bruijn Graph and algorithms that modify it. Such a graph can be represented by a collection from the collections module Section 3.2.1.3. By design graph does not store read sequences, but uses specialized indices called `slices`, which represent given sequence. It therefore separates graph representation from genomic data, enabling efficient reallocation strategies for collections.
Modules layout of the *katome* crate is presented on the Figure 7.

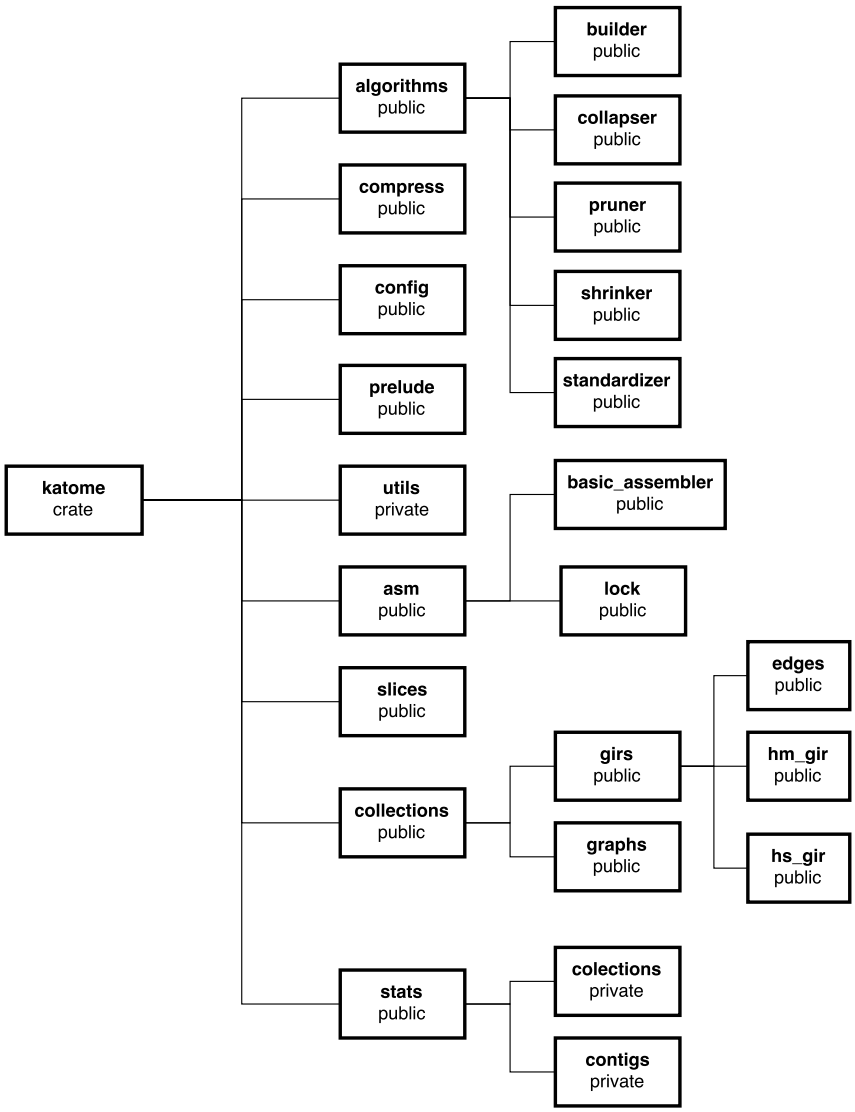

Figure 7. Modules layout of the *katome* crate.

3.2.1.1 slices

When graph representation is created each k-mer should be stored in memory. In order to achieve this *katome* stores read k-mers in the global vector of reads. A single read is represented as `boxed slice` – a special construct in the Rust language, which can be thought of as an array on the heap. This global vector should only be interacted with via `Slice`s. A `Slice` is a structure which wraps operations on global vector, most notably providing efficient hashing, comparison

and decompression for the underlying read. More detailed description of the global vector of reads is provided in Section 3.5.1.

katome provides two types of `Slices`: `NodeSlice` and `EdgeSlice`. During creation of De Bruijn graph `NodeSlices` are used to provide information about already seen nodes (reads of sequences with length $k-1$, where k is the length of k-mer). `EdgeSlices` represent edges in De Bruijn graph. It is important to note that these edges do not have any fixed size, but can be of any length greater than 2.

3.2.1.2 compress

Genomic data that is provided as an input to the compiler tends to use a lot of on-disk memory. When reading this data into RAM it is crucial to make it as small as possible. Small amount of data will allow for more data fitting in the cache, as well as making hashing and comparison significantly faster. Because of those reasons read data in *katome* is compressed. Compression assumes that reads exclusively comprise of 4 basic nucleotides: A, C, G, T. This assumption makes it possible to encode single nucleotide on 2 bits. Such an approach can approximately compress input data fourfold. More thorough overview of compression algorithms and compressed data representation is given in the Section 3.5.1.

3.2.1.3 collections

This module defines collections used in the assembler. Most notably there is distinction between two types of collections — `Graph` and `GIR`. `Graph` is a trait which describes De Bruijn Graph representation in the application. `GIR` stands for Graph's Intermediate Representation, and serves as a collection which is used to filter out noisy/incorrect data before creating `Graph`. All types of collection implement `Build` trait, which enables building these collections based on supplied files. Additionally `GIR` implementers can provide a method to convert `GIR` collection to `Graph` collection.
Currently *katome* provides one implementation of `Graph` for the representation based on `petgraph` library Graph, called `PtGraph`. It also provides two `GIR` implementations based on `HashMap` and `HashSet` from Rust's standard library, which are respectively called `HsGIR` and `HmGIR`. It is assumed that any `Graph` trait implementer will represent De Bruijn graph by storing `EdgeSlices` for the respective edges.

3.2.1.4 algorithms

All algorithms that are available in the process of DNA assembly are described in this module. Notably each of them is represented as a trait which allows for easy implementation of algorithms for new collections. All algorithms are implemented for `PtGraph`. Algorithms do not provide generic implementations based on `Graph` trait, because such approach would hurt the performance for new implementations of `Graph`. This can be observed in the implementation of `Clean` trait, which provides two methods — one for removal of single, disconnected vertices and one for removal of edges with weights below specified threshold. `Clean` trait is implemented for both `PtGraph` and `HmGIR` collections and those implementations significantly vary, as seen in the Listing 1 and Listing 2.

Listing 1. `remove_single_vertices()` implementation for `PtGraph`

```
fn remove_single_vertices(graph: &mut PtGraph) {
    graph.retain_nodes(|g, n| g.neighbors_undirected(n).next().is_some());
}
```

Listing 2. `remove_single_vertices()` implementation for `HmGIR`

```
fn remove_single_vertices(gir: &mut HmGIR) {
    let mut keys_to_remove: Vec<NodeSlice> = gir.iter()
        .filter(|&(_, val)| val.is_empty())
        .map(|(key, _)| *key)
        .collect();
    keys_to_remove = keys_to_remove.into_iter()
        .filter(|x| !has_incoming_edges(gir, x))
        .collect();
    for key in keys_to_remove {
        gir.remove(&key);
    }
}
```

3.2.1.5 asm

This module should be regarded as the entry point for the library, as it defines the trait `Assemble`. `Assemble` exposes a generic API for genome assembly, as shown in Listing 3.

Listing 3. Trait describing public API for genome assembly in *katome* library.

```
/// Public API for assemblers.
pub trait Assemble {
    /// Assembles given data using specified `Graph` and writes results into the
    /// output file.
    fn assemble<P: AsRef<Path>, G: Graph>(config: Config<P>);

    /// Assembles given data using specified `GIR` and `Graph`, and writes
    /// results into the output file.
    fn assemble_with_gir<P: AsRef<Path>, G, T: GIR>(config: Config<P>)
        where G: Graph + Convert<T>;
}
```

This trait should be used when creating custom assemblers to allow for a robust, unified API.

Currently there's only one struct which implements `Assemble` trait — `BasicAssembler`. This struct is used by the client application to provide genome assembling functionality.

3.2.1.6 stats

`stats` module provides trait `Stats` and implementation of this trait for all collections and `SerializedContigs`. Trait provides a simple API, which allows user for acquiring various statistics for collections and generated contigs. These statistics are used throughout log of the `BasicAssembler`.

3.2.1.7 prelude

Prelude holds definitions for all fundamental types used throughout library, as well as definition of global static variables and constants. It also provides a method which changes variable `K_SIZE` (size of the k-mer), along with other related global variables (`K1_SIZE` and `COMPRESSED_K1_SIZE`).

3.2.1.8 config

Struct used for configuration (`Config`) of the assembler is implemented in this module. Trait `RustcDecodable` is derived for this struct, which means that it can be deserialized from several different configuration files. Currently `TOML` [17] format is used for the client application. Example of the configuration file is shown in Section A.3.

3.2.2 Binary executable

Binary executable provides a thin wrapper around *katome* crate. It parses config file into the `Config` struct and runs assembler using the `BasicAssembler` implementation of the `Assemble` trait.

3.3 Code statistics

Following code statistics were generated by the 'loc' [24] tool.

```
-------------------------------------------------------------------------------
 Language             Files        Lines        Blank      Comment         Code
-------------------------------------------------------------------------------
 Rust                    32         5670          432          780         4458
 YAML                     1           26            1            0           25
 Toml                     2           70           11           45           14
-------------------------------------------------------------------------------
 Total                   35         5766          444          825         4497
-------------------------------------------------------------------------------
```

Statistics below show total number of insertions and deletions made in the project, as reported by `Git` version control system, excluding statistics for test data files.

```
16565 insertions(+), 10901 deletions(-), 5664 net
```

3.4 Testing

Tests were implemented using Rust's built-in testing features. Unit tests are placed on the bottom of the file they test, which is a standard pattern in Rust. Additionally each algorithm is tested in integration tests — one test file per algorithm. Integration tests are generally performed in the following steps:

1. collections are built on the various test data;
2. specific algorithm is run on the collection;
3. statistics are compared against correct results.

Due to trait-based design of algorithms (details in Section 3.2.1.4) it is possible to test different implementations of the algorithms — results for different implementations should not differ. In order to aid the process of testing each algorithm's implementations a set of macros were

created. Each test file provides two macros: `test_graph!()` and `test_gir()`. These are used to test collection which implements both the algorithm that file is testing and `Graph` or `GIR` traits accordingly.

Listing 4. Example usage of test macros

```
// import your collection into the scope of the file
use ::{CustomGraph, CustomGIR};
// each macro creates entire test suite for the specific
// implementer of the algorithm
test_graph!(CustomGraph, custom_graph);
test_gir!(CustomGIR, custom_gir);
```

During development continuous integration services were used to regularly build and test project on three tier 1 [18] 64 bit platforms – Linux, Windows and MacOS, with all stages of compiler – stable, beta, nightly. In total 66 unit tests and 43 integration tests were created. Code coverage is estimated to be 89%.

3.5 Efficient memory layout

The main goal of the thesis is the reduction of the memory used by the assembler. Important aspect of such optimization is that it should not hurt the time of assembly, but rather (where possible) make it faster. Because *katome* does not use physical disk to store any information, it is of utmost importance that it keeps as low memory profile as possible — lower memory usage allows user to assemble genomes of more complicated organisms.

3.5.1 Global vector of read data

Usually during creation of the graph there is at least one structure which keeps track of the already considered nodes/edges. In *katome* it is either hash map or hash set. If such structure does not have any place left for the new node/edge then it needs to allocate more memory. New size of of such structure is determined based on the capacity of the collection prior to the resize. Both hash map and hash sets in Rust have noticeable spikes during resize, it is therefore best to reduce the size of the single item in the collection.

Another aspect of presented algorithm is that during the creation of the graph assembler needs to keep track of the already seen nodes to properly create the graph. However once the graph is fully build, this information is no longer needed. Nodes are represented as strings containing FASTA symbols and due to the nature of SSR assembly they are highly redundant. This means that the standard way of tracking them is usage of hash maps/hash sets, which guarantees on average $\mathcal{O}(1)$ lookup, insert and remove operations. The cost here is storing the hash per each item, which influences problem mentioned in the first paragraph of this subsection.

katome solves both problems. I propose the usage of a HashMap to keep the track of nodes during creation of the graph, which is deallocated once the graph is created. Both the graph and the hash map use slices (better described in the Section 3.2.1.1). Resize of the hash map and the graph can be done easily, because both types of slices are lightweight (8 bytes per

node/edge). Graph does not store any `NodeSlice`s, as the full information is stored within the edge. To achieve that the representation of the read data is different during the graph building process and after it is fully built, which are described in this section.

Using global vector further reduces amount of memory used in the hash map/graph, as they don't need to store any additional reference to the vector of data itself. Memory model of Rust does not allow to store a single reference to the global vector in each item, as the vector needs to remain mutable during the process of building the graph. This could be achieved using 'fat' pointers, or more specifically their equivalent in Rust, but this implies higher memory usage per item ('fat' pointers use more than 8 bytes of memory).

3.5.1.1 Data representation during creation of the graph

In general global vector stores arrays of bytes. Slices are used to interpret these bytes into nodes/ edges. During the creation of the graph we store each unique edge as two binary representations of nodes, one after another. This situation is illustrated on Figure 8.

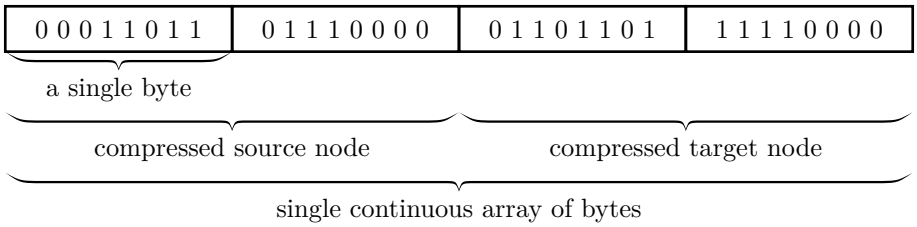

Figure 8. Single compressed edge 'ACGTCTT' representation during graph build with size of k-mer = 4.

Algorithm 4 describes the process of compression of a single k-mer. Compressed data is zero-padded for both k-mer and edge representations.

Algorithm 4. K-mer compression

```
Require: kmer is a string containing only letters A, C, G, T
Require: n is the length of kmer
function Compress K-mer(kmer, n)
    compressed_source ← compress_node(kmer[0..n − 1])
    compressed_target ← compress_node(kmer[1..n])
    ▷ return compressed value as a continuous array of bytes
    return merge_arrays(compressed_source, compressed_target)
function compress_sequence(sequence)
    compressed ← []
    len ← length of sequence
    i ← 0
    while i < len
        compressed.push(compress_byte(sequence[i..i + 4]))
        i ← i + 1
    if len mod 4 ≠ 0 then ▷ align last byte to the most significant bits
        shift_left(compressed.last_byte, 4 −(len mod 4))
    return compressed
Require: read contains at most 4 symbols
function compress_byte(read)
    carrier ← 0
    symbols ← {'A': 0, 'C': 1, 'G': 2, 'T': 3}
    for each symbol in read do
        carrier ← binary_or(carrier, symbols[symbol])
        carrier ← shift_left(carrier)
    return carrier
```

Notably, because both nodes share almost all symbols (except the first symbol for source node and the last symbol for target node) this representation could compress such edge even more. During my work I tried to exploit this fact. It turns out that creating such representation comes with a cost. I have not been able to create representation, which would give me a satisfying performance in terms of hashing and comparison. Most notably while it is easy to hash/compare the source node, to compare/hash the target node the whole representation needs to be shifted, to properly align the data. While methods for doing so are implemented in the application, it uses significantly more time. This is a good example of the trade off between memory usage and run time, the only one in the entire project in which I opted into run time reduction at the expense of memory usage.

3.5.1.2 Data representation after creation of the graph

After graph is built the information about nodes is unnecessary, and so for each edge both nodes get merged into the binary, compressed edge representation. It is worth noting that the first byte in the compressed edge representation is reserved for the size of padding in the last byte, calculated as shown in Equation 1

$$(4 - (edgelength \bmod 4)) \bmod 4 \tag{1}$$

This padding is necessary to properly decompress data, as during shrinking of the graph edges can have various length, as they get extended. Example compression of the edge after graph is built is shown on Figure 9.

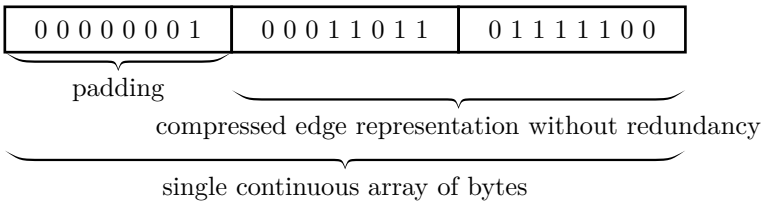

Figure 9. Single compressed edge 'ACGTCTT' representation after graph is fully build, size of k-mer = 4.

Algorithm 5 describes the process of compression of a single edge.

Algorithm 5. Edge compression

Require: *kmer* is a string containing only letters A, C, G, T
Require: n is the length of *edge*

```
function COMPRESS EDGE(kmer, n)
    padding ← 4 − (n mod 4)
    compressed_edge ← COMPRESS_SEQUENCE(kmer)
    return MERGE_ARRAYS([padding], compressed_edge)
```

3.5.2 Graph's Intermediate Representation (GIR)

The peak of the memory usage throughout assembly process resides in the graph creation phase. After all reads are considered algorithms only remove unnecessary nodes, there are no algorithms which add more nodes to the graph. Therefore in order to reduce the memory consumption of the entire application, the reduction during the graph building is needed. While in theory there is a possibility to further reduce the memory usage of the graph, many times it comes with too much cost in terms of usability further on during the assembly process. To help aid such situation *katome* uses collections called Graph's Intermediate Representation, or 'GIR' in short. 'GIR' is a collection created solely for the purpose of graph creation, it should implement a trait, which converts given GIR into some Graph representation. If the conversion step is performant enough in terms of memory, then user might opt-in to include GIR in the assembly process.

GIR can also serve as filters to the raw input data, and so they might implement basic traits used to reduce the number of nodes and edges in the graph. 'BFCounter' application, described in the following subsection, can be thought of as a GIR, which is implemented outside of *katome*. GIR trait gives the possibility to implement its features natively in the Rust language.

3.5.3 'BFCounter' input file type

Preprocessing of the data can be done outside of the assembler itself, and its results might be stored on the disk. Assembler can then use such file to construct the graph and assemble contigs. Currently *katome* supports raw FASTA and FASTAQ file formats, as well as output of the 'BFCounter' application [25]. It is a memory efficient k-mer counting software, which allows the user to count number of occurrences of each edge in the provided data. *katome* can use this information to build the graph more efficiently, especially if user sets up minimal threshold of the weight of the edge in the graph. Refer to the Chapter 4 to see the difference in memory usage / run time that 'BFCounter' introduces.

3.6 Run time minimalization

During implementation of the assembler numerous optimizations of the code were introduced. Below I describe three most notable optimizations.

3.6.1 Compression optimization

As described in Section 3.2.1.2 subsection basic FASTA nucleotides are mapped into their respective 2-bit representation. The implementation of this basic functionality can be found in the Listing 5

Listing 5. `encode_fasta_symbol()` implementation before optimization

```
pub fn encode_fasta_symbol(symbol: u8, carrier: u8) -> u8 {
    // make room for the new symbol in the carrier
    let x = carrier << 2;
    // encode the new symbol
    match symbol {
        b'A' => x,
        b'C' => x | 1,
        b'G' => x | 2,
        b'T' => x | 3,
        _ => unreachable!(),
    }
}
```

As depicted on the Figure 10 this approach has two main problems:

- neither `compress_node` nor `encode_fasta_symbol` are inlined
- compression of the FASTA symbol uses 17% of the application's time

The first problem means that each time the assembler compresses a single FASTA symbol it has to make a function call. Depending on the data it means that there are at least several million unnecessary function calls issued.

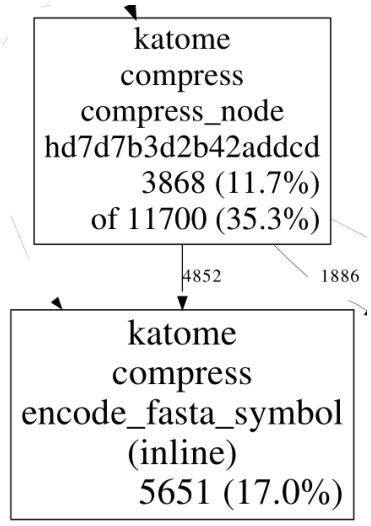

Figure 10. Part of the call graph before `encode_fasta_symbol` function optimization

Second problem means that processor cannot efficiently predict the if-else statement branch. While there is a phenomena called 'nucleotide bias', which shows that for some parts of the genome frequency of occurrence of basic nucleotides may not be evenly distributed (skew) [26], for the purpose of compression algorithm it can be assumed that nucleotides are evenly distributed.

In order to optimize this code `match` statement was removed and replaced with computation of two Boolean functions: Equation 2 and Equation 3, where C, D and A are bits, shown on the Figure 11.

$$\sim C \cdot D \tag{2}$$

$$C + A \tag{3}$$

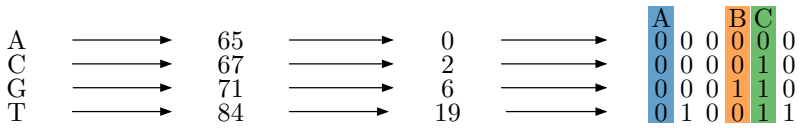

Figure 11. Bits used in the optimized compression algorithm

These functions were created based on the ASCII representation of the symbols. New implementation is described in the Listing 6.

Listing 6. `encode_fasta_symbol()` implementation after optimization

```
#[inline]
pub fn encode_fasta_symbol(mut symbol: u8, mut carrier: u8) -> u8 {
    // make room for the new symbol
    carrier <<= 2;

    // make 'A' 0
    symbol -= b'A';
    // shift so that second bit is first
    symbol >>= 1;
    let c_masked = (symbol & 2) >> 1;
    let a_masked = (symbol & 8) >> 3;
    let d_masked = symbol & 1;
    let first_bit = (c_masked ^ 1) & d_masked;
    let second_bit = c_masked | a_masked;
    carrier | ((second_bit << 1) | first_bit)
}
```

After optimization neither `compress_node` nor `encode_fasta_symbol` are found in the call graph, meaning that their run time is negligible, which is the goal of the optimization.

3.6.2 Node index calculation

During processing of BFCounter input only `NodeSlice`s are tracked. This is an exploit of the BFCounter's property — its output contains only unique edges. The lack of tracking of `NodeIndex`s (indices of nodes in the actual instance of `PtGraph`) allows for further memory usage reduction, but requires a method to map `NodeSlice` to `NodeIndex`. `get_node_index` is a function which implements this functionality. It does so by keeping the track of unique nodes in the global vector of read data. As described in Section 3.5.1 during creation of the graph edges are represented as two compressed nodes. It means that if during insertion of the node it already is present in the graph, the assembler will still compress it and put into the global vector — the edge is guaranteed to be unique and so it needs to be inserted. This creates situation where some of the nodes are duplicated throughout the global vector of sequences.

`NodeSlice` contains a single number, which is an offset on the global vector of reads. Given such number, if assembler is able to distinguish unique nodes from duplicates, then it also can create an index on the graph. To do this it needs to count all of the unique nodes up to the given offset. This algorithm uses the fact that both `NodeSlice`s and `NodeIndex`s are strictly, linearly increasing.

First implementation of `get_node_index` used bit set of fixed size to keep the track of unique nodes. Assembler would set the bit on the offset of uniquely inserted node. Code for the function is shown in the Listing 7.

Listing 7. `get_node_idx` implementation before optimization

```
fn get_node_idx(&self, node: NodeSlice) -> NodeIndex {
    let off = node.offset();
    // count set bits up to the given offset
    NodeIndex::new(self.fb.count_ones(..off))
}
```

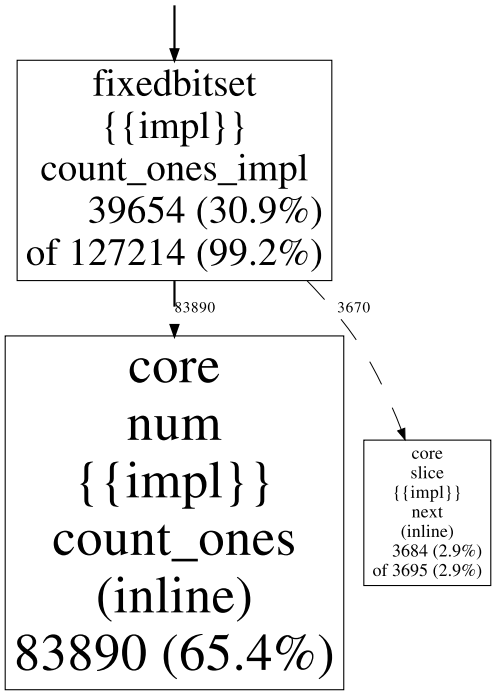

Figure 12. Call graph of the graph creation of assembler before `get_node_idx` optimization.

This implementation was a severe bottleneck of the entire assembler. As shown on the Figure 12 `get_node_idx` method is responsible for 99% of the time during graph building stage. Notably, function which runs for 65% of the time is the `core::num::count_ones` method for the primitive type `u32`. This function counts set bits in the binary representation of unsigned, 32-bit number. These numbers are elements which store bits in the `FixedBitSet` and will be further on referred to as 'blocks'. On architectures which support `POPCNT` assembly instruction (Intel's SSE4.2 or higher) `core::num::count_ones` is realized as a single machine instruction, however if such support is missing then generated assembly uses roughly 15 instructions, as seen in the Listing 8, which significantly hurts the performance of the program.

Both percentages are a clear indication that using the `FixedBitSet::count_ones` function should be avoided, or its usage should be minimized.

Listing 8. `core::num::count_ones` assembly representation

```
mov     eax, edi
shr     eax
and     eax, 0x55555555
sub     edi, eax
mov     eax, edi
and     eax, 0x33333333
shr     edi, 2
and     edi, 0x33333333
add     edi, eax
mov     eax, edi
shr     eax, 4
add     eax, edi
and     eax, 0xf0f0f0f
imul    eax, eax, 0x1010101
shr     eax, 24
```

Proposed solution to this problem is using a vector of numbers, in which each number, further referred to as a region, stores a count of set per multiple blocks. During insertion of the node, region representing block of the node is incremented. To get the desired index of the node algorithm needs to calculate special offset to which it can sum elements of the vector. The rest of the bits should be calculated by using `FixedBitSet::count_ones` function, as they can't be derived from the vector of regions due to its resolution. Changing the number of blocks represented by a single region gives the control over algorithm — the more blocks are represented by region, the faster part of the vector can be summed, but the more bits have to be counted via `FixedBitSet::count_ones`. On the other hand, setting the granularity too high will result in longer times needed to sum the part of the vector (especially for high offsets). Empirically chosen value of 512 blocks per region seems to be best fitting, as it offers the best performance. Implementation of the optimized `get_node_idx` function is presented in the Listing Listing 9. It is worth nothing that the new implementation is specifically fast on architectures with SIMD support, where the compiler is able to vectorize the sum of regions.

Listing 9. `get_node_idx` implementation after optimization

```
#[inline]
fn get_node_idx(&self, node: NodeSlice) -> NodeIndex {
    let off = node.offset();
    // get the index of the last region and number of bits that need to be
    // counted via `count_ones`
    let (last_region, last_block_offset) = (off / 512, off % 512);
    // sum regions
    let mut idx = self.regions[..last_region].iter().sum::<Idx>();
    // count remainder
    if last_block_offset != 0 {
        idx += self.fb.count_ones(last_region * 512..off) as Idx;
    }
    NodeIndex::new(idx as usize)
}
```

New implementation is a magnitude faster than the old one, which indicates a very successful optimization of the code. As depicted on the Figure 13 `get_node_idx` is still responsible for roughly 19% of the graph building process. It could be further optimized by introducing additional vectors, serving the similar role of the carrier vector, but with much higher number of blocks per carrier. It could also be completely removed, by introducing a memory overhead, namely 8 bytes per node. In this scenario `NodeIndex` would be stored as value in the hash map keyed by `NodeSlice`. This is similar to the current solution, but I decided not to use it in order to further reduce memory usage of `katome`, although it might be beneficial to provide it as an opt-in solution for the user, especially if the memory overhead is not an issue (e.g. assembling of organisms with relatively short genome length).

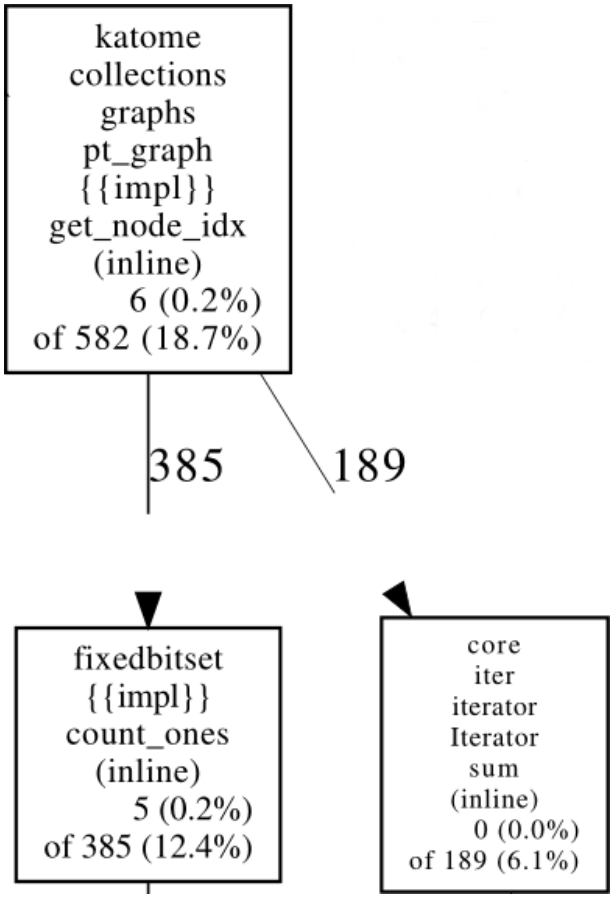

Figure 13. Part of the call graph of the graph creation of assembler after `get_node_idx` optimization

4 Evaluation

Evaluation of genome assemblers can be split into two categories:

- Quality
- Performance

I compare *katome* to three other assemblers, two assemblers chosen by me, as subjectively leading solutions on the current market:

- Abyss [27];
- Velvet [28];

and the assembler *dnaasm* [8]. Rough size of the original genome for all of the organisms is given, in both quality and performance evaluations.

4.1 Quality

Quality of genome assembly can be assessed by comparing four statistics:

- number of created contigs;
- N50 statistic — the shortest contig length at 50% of the genome;
- length of the longest contig;
- number of misassemblies

  The number of misassemblies, using Plantagora's definition. Plantagora defines a misassembly breakpoint as a position in the assembled contigs where the left flanking sequence aligns over 1 kb away from the right flanking sequence on the reference, or they overlap by >1kb, or the flanking sequences align on opposite strands or different chromosomes;

- mismatches per 100 kbp (kilo base pairs)

  The average number of mismatches per 100 000 aligned bases. (…) This metric does not distinguish between single-nucleotide polymorphisms, which are true differences in the assembled genome versus the reference genome, and single-nucleotide errors, which are due to errors in reads or errors in the assembly algorithm;

- indels per 100 kbp

  The average number of single nucleotide insertions or deletions per 100 000 aligned bases.

Definitions for No. of misassemblies, mismatches per 100kbp and indels per 100kbp after A. Gurevich, V. Saveliev, N. Vyahhi, and G. Tesler [29]. Generally assemblers strive to achieve the least number of long contigs. Ideal assembler running on perfect read data should yield one contig per chromosome, which would represent entire chromosome.

Currently both *katome* and *dnaasm* generate contigs and their reverse complement, meaning that generated data is redundant, as each contig can be found twice — once in the coding strand and once in the complementary strand. In the following quality assessment I will show the number of contigs created by both *katome* and *dnaasm* as divided by two, to fairly compare redundant data.

Evaluation of created contigs has been performed with the 'QUAST' — quality assessment tool for genome assemblers [29].

Presented results are generated using the following settings for all assemblers:

- k-mer size: 55;
- reverse complement sequence generation: yes;
- use of paired sequences: no;
- graph correction where possible: yes.

4.1.1 Synthetic data generated on the basis of natural data

Reads for assemblers have been randomly generated, with uniform distribution (standard deviation 20), based on the reference genome of the bacteria Escherichia Coli. All reads are perfect, meaning they don't contain any read errors. The dataset for the bacteria Escherichia coli is described in the Table 1 and the results of the quality evaluation are described in the Table 2.

Table 1. Dataset description for bacteria *Escherichia coli* genome

size of the genome [bp]	number of chromosomes	number of reads	mean length of read	coverage
4M	1	3180956	100	80

Table 2. Quality of assembly on synthetic reads based on *Escherichia coli* genome

assembler	number of contigs	N50 [bp]	longest[1] [bp]	misassemblies	mismatches[2]	indels[3]
katome	576	118966	265135	0	0.00	0.00
dnaasm	427	118966	265135	0	0.00	0.00
velvet	160	118966	265135	0	0.05	0.03
abyss	279	118966	265135	0	0.05	0.00

[1] Length of the longest contig. [2] Mismatches per 100 kbp. [3] Indels per 100 kbp.

4.1.2 Natural data

4.1.2.1 Saccharomyces cerevisiae

Data for strain `S288C` of the yeast genome. The dataset for the yeast is described in the Table 3 and the results of the quality evaluation are described in the Table 4.

Table 3. Dataset description for bacteria *Saccharomyces cerevisiae* genome

size of the genome [bp]	number of chromosomes	number of reads	mean length of read	coverage
12M	16	4862828	100	43

Table 4. Quality of assembly on synthetic reads based on *Saccharomyces cerevisiae* genome

assembler	number of contigs	N50 [bp]	longest[1] [bp]	misassemblies	mismatches[2]	indels[3]
katome	15584	38353	140369	0	0.22	0.08
dnaasm	6448	38353	140369	0	0.26	0.05
velvet	1982	38520	140369	0	0.58	0.02
abyss	3527	38520	140369	0	1.62	0.14

[1] Length of the longest contig. [2] Mismatches per 100 kbp. [3] Indels per 100 kbp.

4.1.2.2 Caenorhabditis elegans

The dataset for the Caenorhabditis elegans is described in the Table 5 and the results of the quality evaluation are described in the Table 6.

Table 5. Dataset description for bacteria *Caenorhabditis elegans* genome

size of the genome [bp]	number of chromosomes	number of reads	mean length of read	coverage
100M	6	40114554	100	42

Table 6. Quality of assembly on synthetic reads based on *Caenorhabditis elegans* genome

assembler	number of contigs	N50 [bp]	longest[1] [bp]	misassemblies	mismatches[2]	indels[3]
katome	380424	14051	130755	0	0.10	0.07
dnaasm	154391	14051	130755	0	0.15	0.04
velvet	43802	14363	130755	0	0.22	0.02
abyss	104271	13943	130755	0	2.30	0.28

[1] Length of the longest contig. [2] Mismatches per 100 kbp. [3] Indels per 100 kbp.

4.2 Performance

In this section I present performance results of *katome*, in comparison to other solutions. Unless stated otherwise size of the k-mer for these evaluations is 55. Each cell in the table contains mean time of execution, standard deviation of that time and peak memory usage of the application. Data that assemblers are tested on is too big to perfectly measure memory usage (e.g. via 'valgrind'), and so the memory usage of the application is probed every 1 millisecond and the peak is reported. While this approach is not ideal (it can miss the exact peak and report lower usage) in practice this method turns out to be good enough to estimate the memory usage of the assemblers.

All benchmarks have been performed using the hardware and software listed in the Table 7.

Table 7. Software and hardware used for performance evaluations of assemblers.

System	Kernel	3.16.0-4-amd64 x86_64 (64 bit)
	Distro	Debian GNU/Linux 8
CPUs		2 Octa core Intel Xeon E5-2630 v3s cache: 40 MB
Drives		2 WDC_WD1000DHTZ size: 1 TB, working in RAID 1
Memory		252 GB

4.2.0.1 Escherichia coli

Size of the original genome: 4M. Results are shown in the Table 8.

Table 8. Performance of assemblers for *E. coli* genome

		FASTQ		BFCounter	
		1	2	1	2
	PtGraph	104.9s std=0.67s 1392 MB	180.8s std=3.68s 1396 MB	12.0s std=0.16s 605 MB	35.6s std=0.00s 1194 MB
katome	HmGIR	136.1s std=1.01s 1592 MB	193.2s std=3.90s 1590 MB	—	—
	HsGIR	165.3s std=0.11s 1724 MB	293.7s std=0.04s 1724 MB	—	—
dnaasm		136.9s std=0.66s 1367 MB	226.6s std=0.19s 1364 MB	41.9s std=1.05s 1363 MB	41.9s std=0.21s 1364 MB
velvet		140.2s std=0.69s 582 MB	—	—	—
abyss		95.29s std=0.2687s 516 MB	—	—	—

1 Without reverse complement generation.

2 With reverse complement generation.

4.2.0.2 Saccharomyces cerevisiae

Size of the original genome: 12M. Results are shown in the Table 9.

Table 9. Performance of assemblers for *Saccharomyces cerevisiae* genome

		FASTQ		BFCounter	
		1	2	1	2
katome	PtGraph	170.1s std=0.12s 3594 MB	299.0s std=0.41s 3595 MB	36.8s std=0.16s 2054 MB	96.2s std=2.63s 3077 MB
	HmGIR	249.6s std=0.11s 4710 MB	370.2s std=1.88s 4502 MB	—	—
	HsGIR	288.4s std=0.43s 5192 MB	465.2s std=0.17s 5063 MB	—	—
dnaasm		231.5s std=0.57s 3923 MB	468.4s std=1.33s 3926 MB	132.0s std=0.45s 3928 MB	131.9s std=0.74s 3922 MB
velvet		218.0s std=0.566s 1026 MB	—	—	—
abyss		174.9s std=0.417s 822 MB	—	—	—

1 Without reverse complement generation.

2 With reverse complement generation.

4.2.0.3 Caenorhabditis elegans

Size of the original genome: 100M. Results are shown in the Table 10.

Table 10. Performance of assemblers for *Caenorhabditis elegans* genome

		FASTQ		BFCounter	
		1	2	1	2
	PtGraph	1622.0s std=14.9s 28784 MB	2776.0s std=8.1s 28788 MB	432.2 std=0.37s 16533 MB	1183.0s std=5.0s 25052 MB
katome	HmGIR	2416.0s std=40.06s 38277 MB	3683.0s std=10.47s 38244 MB	—	—
	HsGIR	3001.0s std=306.06s 41246 MB	4516.1s std=19.46s 41231 MB	—	—
dnaasm		2129.0s std=4.01s 32648 MB	3864.0s std=29.2s 32645 MB	1288.0s std=8.59s 32620 MB	1298.0s std=10.87s 32621 MB
velvet		3035.0s std=5.62s 7277 MB	—	—	—
abyss		1807.0s std=11.26s 3378 MB	—	—	—

1 Without reverse complement generation.

2 With reverse complement generation.

5 Summary

5.1 Conclusions

As shown in the Chapter 4, *katome* outperforms all of the assemblers on almost all of the data. While the overall difference in the run time is not big, it can be speculated that other assemblers are more mature than *katome* and their developers had more time to tune and optimize them accordingly. In my opinion there is still a possibility of significant speed up and memory usage reduction in the *katome* project.

Notably the new assembler is comparable to the *dnaasm* project in terms of quality — *dnaasm* and *katome* have overall lower number of mismatches per 100 bp and indels per 100 bp, while 'Velvet' and 'ABySS' assemblers create less contigs. Judging by the statistics for *katome* it creates a lot of small (less than 1000 bp) contigs, although created longer contigs are similar to other assemblers, thus other statistics are similar to other assemblers.

Additionally further speed-up is possible at the cost of the memory, if the user chooses to opt into that.

The goals, described in Section 1.1 have been successfully achieved. I was able to construct a modular, safe and well-performing library, which has been designed with parallelism in mind. Rust enables strong memory safety and parallel/concurrent safety, as well as the ease of extending the library, by introduction of implementers of the proposed traits.

Throughout the project Rust was a significant improvement to the usual workflow, as the language puts a strong emphasis on memory safety. I found its tooling simple, effective and intuitive. Furthermore the language's strong type system made refactoring notably easier than in the weakly typed languages.

5.2 Future work

Although *katome* is currently a functional and working assembler there are many ways to improve it. Below I list several ideas which might be expanded further to improve the quality of its output, as well as its performance, both in terms of speed and memory usage.

5.2.1 Quality improvements

Reduction of the number of created small contigs
In the number of assembled contigs it is clearly shown that *katome* creates more contigs than *dnaasm* (up to roughly 2 times more). Further analysis of the data shows, that most of these contigs are short (less than 1000 bp). I think that this difference can come up from the way *katome* handles subgraphs in which all nodes have at least one incoming edge. Reducing number of created contigs would lead to improving various statistics and would lead to the general improvement of the assembly.

Implementation of the bubble removal
Authors of 'Velvet' assembler introduced the Tour Bus algorithm, which removes so called 'bubbles' — two paths that start and end at the same node. By looking at the graphs created by *katome* for various organisms, it is clearly shown that bubbles make up for a lot of ambiguous

nodes, which in turn directly influence number of created contigs. I haven not implemented this algorithm due to the lack of time in the current project, but I am convinced that it could significantly reduce number of created contigs.

Merge of reverse complement k-mers representations
Currently *katome* stores k-mers and their reverse complements as distinct edges in the created graph. There exist methods to merge both representations into one, effectively reducing the memory usage of the entire application.

Support for paired-end reads
Using information about paired-end reads greatly improves the quality of assembly. While most assemblers, including *dnaasm* supports it, *katome* does not use this information during assembly. The challenge in using this technique is memory usage, as in the naive form, implemented in the *dnaasm* assembler, it doubles the memory used by the assembler. Efficient, RAM-only implementation of this method is an interesting problem both from in terms of algorithm design and optimization of the implementation, but in my opinion it would yield significant improvement of the quality of assembler.

5.2.2 Performance improvements

Introduction of parallelization into different algorithms
Application is written in a fashion that should allow parallelized algorithms to operate on the Graph and GIR implementers. The most important thing to note is that due to the nature of the algorithms (operating on a single, big collection) usually algorithms would get divided into the parallel section, which operates on non-mutable references to the collection and gathers information, and a single-threaded code which would mutate the collection accordingly, using information from the parallel part.

While in theory it would be possible to process input in parallel by building several separate collections and merge them, in my opinion this approach is not feasible. It would almost certainly use more memory due to the redundancy in the data it would potentially introduce. As such I think that parallelization of the graph creation is a challenging, and possibly unachievable goal, at least if the application will stay RAM-exclusive.

GIR based on Bloom filter
'BFCounter' application uses Bloom filter to effectively count k-mers in the given input, which appear more times than given threshold. Using the GIR trait it would be easy to implement such functionality within katome. This could prove beneficial to the application, because Bloom filter is known to be a memory-optimal solution to the input-tracking problem, which all assemblers need to solve to create contigs. Furthermore, 'BFCounter' uses disk to store the intermediate structures during counting, which most probably influences its run time. By storing these structures in memory we are looking at the potential speed-up of the input processing.

Run time optimization at the cost of memory usage
For small organisms users might opt-in into faster assembly, at the cost of memory usage, which for small algorithms could be negligible. I observed a speed-up of at least 30% of the entire assembly time in *katome* on varied data, after adding an already-seen nodes tracking hash map

to the ‘BFCounter’ input type graph building algorithm. I believe that there are places in the code, in which similar trade-offs could be beneficial to the user. It would therefore be a good idea to design and implement robust API for the user to choose between run time reduction and memory usage reduction.

Bibliography

[1] J. Adams, "Complex genomes: Shotgun sequencing," *Nature Education*, vol. 1, no. 1, p. 186, 2008.

[2] J. L. Weber and E. W. Myers, "Human whole-genome shotgun sequencing," *Genome Research*, vol. 7, no. 5, pp. 401–409, 1997.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[4] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows–Wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[5] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome," *Genome Biology*, vol. 10, no. 3, p. R25, 2009, doi: 10.1186/gb-2009-10-3-r25.

[6] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with Bowtie 2," *Nature methods*, vol. 9, no. 4, pp. 357–359, 2012.

[7] R. M. Nowak, "Assembly of repetitive regions using next-generation sequencing data," *Biocybernetics and Biomedical Engineering*, vol. 35, pp. 276–283, 2015.

[8] W. Kusmirek, "The DNA assembler for next-generation sequencing data," 2016.

[9] "The Rust Programming Language." Accessed: Nov. 27, 2016. [Online]. Available: https://www.rust-lang.org/en-US/

[10] "The C++ Committee." Accessed: Nov. 27, 2016. [Online]. Available: https://isocpp.org/std/the-committee

[11] "Java's Executive Committee." Accessed: Nov. 27, 2016. [Online]. Available: https://jcp.org/en/participation/committee

[12] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, in SP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 48–62. doi: 10.1109/SP.2013.13.

[13] H.-J. Boehm, "How to Miscompile Programs with "Benign" Data Races," in *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, in HotPar'11. Berkeley, CA: USENIX Association, 2011, p. 3. [Online]. Available: http://dl.acm.org/citation.cfm?id=2001252.2001255

[14] H.-J. Boehm and S. V. Adve, "Foundations of the C++ Concurrency Memory Model," *SIGPLAN Not.*, vol. 43, no. 6, pp. 68–78, Jun. 2008, doi: 10.1145/1379022.1375591.

[15] N. Myers, "Rust vs. C++: Fine-grained Performance." Accessed: Nov. 21, 2016. [Online]. Available: http://cantrip.org/rust-vs-c++.html

[16] I. Gouy, "The Computer Language Benchmarks Game." Accessed: Nov. 21, 2016. [Online]. Available: https://benchmarksgame.alioth.debian.org/u64q/rust.html

[17] T. Preston-Werner, "Tom's Obvious, Minimal Language." Accessed: Nov. 27, 2016. [Online]. Available: https://github.com/toml-lang/toml

[18] "Rust Platform Support." Accessed: Nov. 26, 2016. [Online]. Available: https://forge.rust-lang.org/platform-support.html

[19] "Graph data structure library for Rust." Accessed: Jan. 04, 2017. [Online]. Available: https://github.com/bluss/petgraph

[20] "A simple bitset container for Rust." Accessed: Jan. 04, 2017. [Online]. Available: https://github.com/bluss/fixedbitset

[21] "MetroHash: Faster, Better Hash Functions." Accessed: Jan. 04, 2017. [Online]. Available: http://www.jandrewrogers.com/2015/05/27/metrohash/

[22] "Rust implementation of MetroHash." Accessed: Jan. 04, 2017. [Online]. Available: https://github.com/arthurprs/metrohash-rs

[23] "Compact and efficient synchronization primitives for Rust." Accessed: Jan. 04, 2016. [Online]. Available: https://github.com/Amanieu/parking_lot

[24] "Tool for counting lines of code.." Accessed: Jan. 17, 2017. [Online]. Available: https://github.com/cgag/loc

[25] P. Melsted and J. K. Pritchard, "Efficient counting of k-mers in DNA sequences using a bloom filter," *BMC Bioinformatics*, vol. 12, no. 1, p. 333, 2011, doi: 10.1186/1471-2105-12-333.

[26] J. Lobry, "Asymmetric substitution patterns in the two DNA strands of bacteria.," *Molecular biology and evolution*, vol. 13, no. 5, pp. 660–665, 1996.

[27] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. Jones, and I. Birol, "ABySS: a parallel assembler for short read sequence data," *Genome research*, vol. 19, no. 6, pp. 1117–1123, 2009.

[28] D. R. Zerbino and E. Birney, "Velvet: algorithms for de novo short read assembly using de Bruijn graphs," *Genome research*, vol. 18, no. 5, pp. 821–829, 2008.

[29] A. Gurevich, V. Saveliev, N. Vyahhi, and G. Tesler, "QUAST: quality assessment tool for genome assemblies," *Bioinformatics*, vol. 29, no. 8, pp. 1072–1075, 2013.

List of Symbols and Abbreviations

self-loop: A pair of an edge and a vertex satisfying Definition 2.5.1.

simple-loop: A directed path satisfying Definition 2.5.2.

strand: A directed path satisfying Definition 2.4.1.

List of Figures

List of Tables

List of Appendices

Appendix A User Manual

A.1 Installation

katome currently supports Linux, MacOS and Windows operating systems. Note that internet connection is required to download and install necessary tools, as well as to compile the application — dependencies must be downloaded from `Crates.io`. For more information refer to Section 3.1.3. To install *katome* on any of the above systems follow these steps:

1. Install Git version control system from https://git-scm.com/
2. Install latest stable version of the Rust compiler and Cargo package manager. Recommended tool for this installation can be found at https://rustup.rs/
3. Clone *katome* project and change working directory into the cloned repository.

```
$ git clone https://github.com/fuine/katome
$ cd katome
```

4. Compile application.

```
$ cargo build --release
```

5. Built binary executable can be found in the `target/release` directory.

A.2 Testing

To run tests issue the following command in the repository:

```
$ cargo test
```

A.3 Configuration

Currently *katome* is configured via two files, which should be placed at `config` directory relative to the current working directory:

1. `settings.toml` — configuration of the assembler
2. `log4rs.yaml` — configuration of the logger

Both example files are provided to the user. To find more information about logging configuration please refer to the following url https://docs.rs/log4rs/0.5.2/log4rs/. Example `settings.toml` file is shown in the Listing A.1 with comments describing each field.

Listing A.1. Example `settings.toml` configuration file for *katome* assembler

```
# input files path, should contain at least one filename
input_files = ["/path/to/input/file", "/path/to/another/input/file"]

# output file path
output_file = "/path/to/output/file"

# original genome length
original_genome_length = 100

# minimal weight of the edge in De Bruijn Graph
minimal_weight_threshold = 0

# input file type, currently can have one of threes values:
# BFCounter, Fasta, Fastq
input_file_type = "Fastq"

# size of the k-mer
k_mer_size = 40

# whether or not katome should create reverse complementary sequences
# to the original reads. While this option noticeably slows down
# the process of assembly it usually will create higher quality output.
reverse_complement = true
```

A.4 Usage

To use the assembler configure it to your use case and simply execute the binary file. If your current working directory is within the repository for *katome* you may also issue following command to run the assembler:

```
$ cargo run --release
```

In case of any issues or suggestions regarding katome please open an issue or file a pull request at https://github.com/fuine/katome.