Optimal Genome Paired-End Alignments Downsampling by Solving Interval Multicover Problem

Marek Mytkowski Borys Kurdek Michał Okurowski

June 29, 2024

Contents

1 Problem of Short Paired-End Alignments Downsampling

Due to technical reasons methods for studying biological genomes are producing a significant number of short sequences from alphabet $\Sigma = \{A, C, T, G\}$. These sequences, also called reads, are often paired. An element from the alphabet Σ in the genome context is called a base pair. So reads can consist of tens of base pairs. Reads need to be aligned to the reference genome, and alignment software grades read with *MAPping Quality* [7], later called MAPQ score. The reference genome can be divided into *amplicons*, which are specific regions of the genome. The volume of data in sequencing tasks is huge, but computational resources are always limited. This is the reason, why it is a common practice to filter out part of reads before further processing. This process we would call a *downsampling of genome paired-end reads*. The problem is how to filter out only the least useful data but with close to even coverage of the reference genome. The proposed downsampling method should also optimize time and memory complexity, for the same reason downsampling is not being used at all. During downsampling, we would prioritize:

1. Pairs where both component sequences are contained within a single amplicon,
2. Pairs of better MAPQ.

The goal of the downsampling is to obtain a set of paired reads such that every index from the reference genome is covered at least m times while minimizing the number of paired reads. If the index is covered less than m times its initial coverage should not change after downsampling.

1.1 Real-world example

The real-world problem presented to us, based on which we have tested solutions, involves reducing the number of paired-end RNA sequences for the SARS-CoV-2 virus. Reads have a length of $n \approx 150bp$. The reference genome is a sequence containing approximately $N = 3 \cdot 10^4 bp$ of RNA code. As an example, there might be $M = 10^7$ short paired reads, resulting in $2 \cdot 10^7$ reads of n *bp* each. Generally, we can assume that $M >> N$. This means that when searching for an algorithm to solve the problem, it would be best if it would minimize the impact of M on computational and memory complexity.

2 Problem definition

Let *read* R be a finite, continuous subsequence of length $l(R)$ from finite sequence $S = 0, 1, \ldots, N$ that represents a *genome sequence*. Depending on the context S and R will be treated either as a set or as a sequence, for example when $R_0 \leq i$ and $R_{l(R)-1} \geq i$, it means that $i \in R$.

Let $Q = \{R^{(0)}, R^{(1)}, \ldots R^{(M)}\}$ be a *samples set* of reads for S and let $C_Q : S \to \mathbb{N}$ denote the coverage of $i \in S$ by the set Q defined as number of reads in Q such that the $i \in R$.

Real datasets very often provide paired reads. We will denote $\mathcal{Q} = \{I_0, I_1, \dots I_m\}$ as a partition of Q, where every set I_i known as *2-interval* contains at most 2 elements and there is at least one j such that $|I_j| = 2$. We define $C_{\mathcal{Q}} : S \to \mathbb{N}$ as the coverage C_Q, where $Q = \bigcup_{I \in \mathcal{Q}} I$.

Let $b : S \to \mathbb{N}$ be a function that is defined as $b(i) = \min\{m, C_Q(i)\}$ for any $i \in S$. We say that $Q' \subseteq Q$ *covers* S with respect to function b if and only if condition $C_{Q'}(i) \geq b$ is satisfied for any $i \in S$. The $\mathcal{Q}' \subseteq \mathcal{Q}$ covers S if and only if $Q = \bigcup_{I \in \mathcal{Q}} I$ covers the S.

Definition 2.1. *We define the Multi Cover Problem (MCP) as the problem of finding a minimum subset $Q' \subseteq Q$ such that Q' covers S with respect to b.*

Definition 2.2. *We define the Paired Multi Cover Problem (2MCP) as the problem of finding a minimum subset $\mathcal{Q}' \subseteq \mathcal{Q}$ such that $\mathcal{Q}'$ covers S with respect to b.*

More general definitions of the problems described here were proposed in [3] (c-Interval Cover Problem) and [1] (c-Interval Multicover Problem).

Let *amplicon* A be continuous subsequence from S starting at position k and ending at position l of the S as $A = (k, k+1, ..., l-1, l)$. R is contained within amplicon A if $k \leq i$ and $i + n - 1 \leq l$, it means that $R \in A$. To prioritize pairs of reads contained within a single amplicon, for any pair of reads I, so $R \in I$ and a set of amplicons $\{A_1, A_2, ..., A_n\}$, we define $q : Q \to \mathbb{N}$ a quality function

$$q(R) = \begin{cases} q_r(R) - 1, & \text{if } \ R \in I \wedge \exists_{i \in \{1,2,\dots,n\}} \ \forall_{R_I \in I} \ R_I \in A_i \\ q_r(R), & \text{otherwise} \end{cases}.$$

where $q_r(R)$ is MAPQ score value (we assume normalized MAPQ $q_r : Q \to (0, 1]$) of the read R. Then $q : Q \to (-1, 1]$.

For MCP we define QMCP (Quality Multi Cover Problem) as a problem of finding solution Q' in all possible MCP solutions with minimium $\sum_{i \in S} \sum_{R \in Q', i \in R} q(R)$. We define the 2QMCP by analogy.

As many previous works already stated ([1], [3], [5]) the 2MCP is NP-hard. For that reason our work will focus on 2-approximation of 2MCP which utilizes MCP solvers. The approach is based on solving the MCP and then including additional pairs in the solution that have not already been included. This method was proposed in [5].

3 Algorithms

3.1 Solving MCP with circulation

Let $\text{arc}(R) := R_0(R_{l(R)-1} + 1)$ and $G := (V, E)$ be a directed graph, where

$$\begin{aligned} V &:= \{0, 1, \dots, N, N+1\}, \\ F &:= \{\text{arc}(R) : R \in Q\}, \\ B &:= \{v(v-1) : v \in V \setminus \{0\}\}, \\ E &:= B \cup F. \end{aligned}$$

We will refer to the arcs from B as *backward arcs* and to F as *forward arcs*. Let $d : V \to \mathbb{N}$ be a demand function defined for any $v \in V$ as

$$d(v) = \begin{cases} -b(0) \text{ if } v = 0 \\ b(v) - b(v+1) \text{ if } v \in [1, N] \\ b(N) \text{ if } v = N+1 \end{cases} .$$

Let $c : E \to \mathbb{N}$ be a capacity function and $k : E \to \infty$ a cost function, for any $e \in E$ they are defined as follows

$$c(e) = \begin{cases} 1 \text{ if } e \in F \\ \infty \text{ if } e \in B \end{cases} ,$$

$$k(e) = \begin{cases} 1 \text{ if } e \in F \\ 0 \text{ if } e \in B \end{cases} .$$

We will refer to the described network (G, d, c, k) as *MPC network*. MPC network is a special case of a network that was introduced in [5] for solving multi-shift scheduling problem.

Let f be a circulation on any MPC network, we define

$$Q_f := \{R : f(\text{arc}(R)) = 1\}.$$

Observation 1. *From the k and c definitions the cost of a circulation f is equal to the $|Q_f|$.*

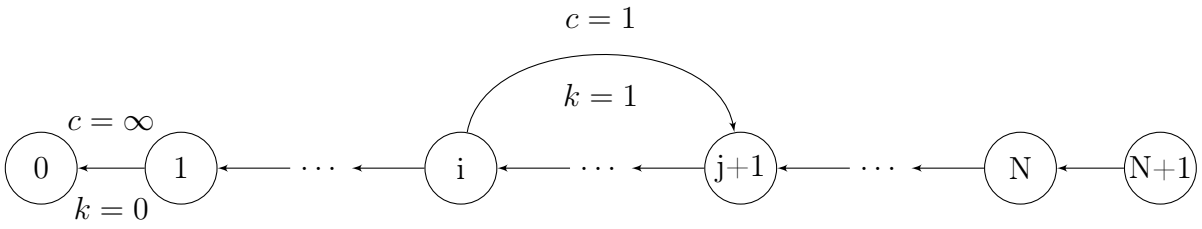

Figure 1: MCP network with read $R = i, i+1, \ldots, j$, represented as the forward arc $\text{arc}(R) = i(j+1)$.

Theorem 3.1. *If f is a circulation in (G, d, c) network then $\forall_{i \in S} C_{Q_f}(i) \geq b(i)$.*

Proof. Let $x \in S$ be any vertex.

1. Let p be a number of arcs $uv \in E$, such that $u \leq x$ and $v \geq x+1$ and $f(uv) = 1$, then $C_{Q_f}(x) = p$.

2. Let $\alpha := \sum_{v \leq x} d(v)$ and $\beta := \sum_{v \geq x+1} d(v)$. Note that

$$\alpha = d(0) + d(1) + \ldots + d(x) = -b(0) + b(0) - \ldots b(x-1) - b(x) = -b(x) \leq 0.$$

3. Since $\alpha + \beta = 0$, we have $\beta = -\alpha$. Based on step 2, $p \geq -\alpha$. If this were not the case, we wouldn't transport enough units from the left side of pivot x to the right side, and the demand β wouldn't be satisfied.

4. From 1., 2. and 3. we may imply that

$$C_{Q_f}(x) = p \geq -\alpha = b(x).$$

□

The 3.1 shows that any Q_f, where f is a circulation in (G, d, c) network fulfills the MCP necessary condition i.e. Q_f always covers the genome sequence.

Lemma 3.1. *For any $Q' \subseteq Q$ that covers the genome with respect to b a corresponding circulation f exists in the (G, d, c) network such that $Q' = Q_f$.*

Proof. Let $Q' \subseteq Q$ be any solution of MCP. We define $f_F : E \to \mathbb{N}$ such that for each read $R \in Q$

$$f_F(\text{arc}(R)) = \begin{cases} 1 \text{ if } R \in Q' \\ 0 \text{ otherwise} \end{cases} .$$

In other words we saturate all arcs from F that are corresponding to the reads from Q'. Note that f_F is not a circulation, since there exists a node v for which $e_{f_F}(v) := f_F^{\text{in}}(v) - f_F^{\text{out}}(v) - d(v) \neq 0$.
Let's define $f : E \to \mathbb{Z}$ such that for each arc $uv \in F$, $f(uv) = f_F(uv)$ and for each arc $(v+1)v \in B$,

$$f(v+1, v) = e_{f_F}(v+1) + e_{f_F}(v+2) + \ldots + e_{f_F}(N+1).$$

We claim that f is a circulation. To show that we need to prove that

1. for any arc $(v+1)v \in B$, $f(v+1, v) \geq 0$,
2. for any node $u \in V$, $e_f(u) := f^{\text{in}}(u) - f^{\text{out}}(u) - d(u) = 0$.

From the definition of f for any $(v+1)v \in B$ we get

$$f(v+1, v) = e_{f_F}(v+1) + \ldots + e_{f_F}(N+1) =$$

$$= \sum_{x \in [v+1, N+1]} f_F^{\text{in}}(x) - \sum_{x \in [v+1, N+1]} f_F^{\text{out}}(x) - \sum_{x \in [v+1, N+1]} d(x) =$$

$$= \sum_{x \in [v+1, N+1]} f_F^{\text{in}}(x) - \sum_{x \in [v+1, N+1]} f_F^{\text{out}}(x) - b(v).$$

Note that $\sum_{x\in[v+1,N+1]} f_F^{\text{in}}(x) - \sum_{x\in[v+1,N+1]} f_F^{\text{out}}(x) \geq 0$ as there is no arc $xy \in F$ such that $y < x$. Since the Q' covers the genome with respect to b we get

$$\sum_{x\in[v+1,N+1]} f_F^{\text{in}}(x) - \sum_{x\in[v+1,N+1]} f_F^{\text{out}}(x) \geq b(v),$$

so the 1. must be true.
For any node $v \in V \setminus \{0, N+1\}$ from the definition of e_f we know that

$$e_f(v) = f(v, v-1) - (e_{f_F}(v) + f(v+1, v)) = 0.$$

When the $v = 0$,

$$e_f(v) = e_{f_F}(0) - (\sum_{x\in[1,N+1]} f_F^{\text{in}}(x) - \sum_{x\in[1,N+1]} f_F^{\text{out}}(x) - b(0))$$

from the fact that $\sum_{x\in[0,N+1]} f_F^{\text{in}}(x) + \sum_{x\in x\in[0,N+1]} f_F^{\text{out}}(x) = 0$, we get

$$e_f(v) = f_F^{\text{in}}(0) - f_F^{\text{out}}(0) - b(0) - \sum_{x\in[1,N+1]} f_F^{\text{in}}(x) + \sum_{x\in[1,N+1]} f_F^{\text{out}}(x) + b(0) = 0.$$

When the $v = N + 1$,

$$e_f(N+1) = e_{f_F}(N+1) - f(N+1, N) = e_{f_F}(N+1) - e_{f_F}(N+1) = 0.$$

We proved that 2. must also be true, thus f is a circulation for which $Q' = Q_f$. □

Theorem 3.2. *For any minimum cost circulation f in MCP network the Q_f is MCP solution.*

Proof. Let f be any minimum cost circulation and let Q' be any MCP solution.

1. From the lemma 3.1 we know that Q' has corresponding circulation f' such that $Q' = Q_{f'}$.
2. If $|Q_f| > |Q_{f'}|$ then the f would not be a minimum cost circulation since the f' would have better cost than f (observation 1).
3. If $|Q_f| < |Q_{f'}|$ then the $Q_{f'}$ would not be a MCP solution since according to the theorem 3.1, Q_f covers the genome sequence, indicating that Q_f is better MCP solution than Q'.
4. Thus $|Q_f| = |Q_{f'}|$.
5. From the theorem 3.1 and 4th step the Q_f must be a MCP solution.

□

From 3.2 we derive that the algorithm based on finding the minimum cost circulation in the MCP network will result in the MCP solution.

Since the theorem 3.1, we decided to test algorithms that find any circulation in the graph (G, d, c). We call such algorithms *quasi-MCP solvers*. Although this approach does not solve the MCP problem, it produces satisfactory results, which we believe is caused by the fact that the input reads have similar lengths.

According to our experiments we suspect that the maximum flow based quasi-MCP algorithm is actually MCP solver for the case when all reads are of the same length. However this conclusion needs a proof.

The gain from quasi-MCP solvers is that they are faster as they are based on maximum flow problem solvers while solving MCP requires minimum cost flow solver. We have developed CUDA and CPU quasi-MCP solvers, both based on the push-relabel method.

3.1.1 CPU quasi-MCP maximum flow solver

The CPU quasi-MCP solver is based on finding any circulation in the (G, d, c) graph which is equivalent of solving maximum flow problem. In order to convert the circulation problem to the maximum flow problem we add vertices $s = N + 2$ and $t = N + 3$, which results in new set of vertices $V' = V \cup \{s, t\}$. We also create a new set of arcs E', by adding new arcs to E:

- sv for any node $v \in V$ for which $d(v) < 0$,
- vt for any node $v \in V$ for which $d(v) > 0$.

This way a graph $G' = (V', E')$ is created and the maximum flow algorithm may be executed for the (G', c, s, t) network. The circulation exists if and only if all of the arcs from s are saturated. Based on the lemma 3.1 we know that if there is any MCP solution, there must also be a circulation.

For our analysis we've used push-relabel based maximum flow solver implemented in OR-tools [4] library.

3.1.2 CPU MCP minimum cost circulation solver

To solve the MCP, we have constructed the MCP circulation network and utilized the OR-Tools [4] circulation minimum cost solver, which is based on cost-scaling method, for our analysis.

3.1.3 CUDA quasi-MCP maximum flow solver

Push-relabel maximum flow method is known for its potential to be parallelized. We tested CUDA based parallel implementation of the push-relabel maximum flow solver for the quasi-MCP solver. The input graph data is the same as in CPU quasi-MCP solver – it's the (G', c, s, t) network.

Let $l : V' \to \mathbb{N}$ be a *label function*. A function $f : E' \to \mathbb{N}$ is said to be *a preflow* if and only if for any arc $uv \in E'$ and for any node $x \in V$ conditions $f(uv) \leq c(uv)$ and $e_f(x) := f^{\text{in}}(x) - f^{\text{out}}(x) \geq 0$ are met. We call the e_f function *an excess function* for f. If $\forall_{x \in V} e_f(x) = 0$ the f is said to be *feasible flow* or just *flow*. A node $x \in V$ for which $e_f(x) > 0$ is said to be *active*. Let E'_f denote the arcs of the residual graph with respect to the preflow f and let $c_f : E'_f \to \mathbb{N}$ be a capacity in that residual graph.

In the push-relabel method a preflow is initialized by saturating all arcs coming out of s and setting $l(s) = |V|$. During the algorithm the f is always a preflow and for any arc $uv \in E'_f$ the $l(u) \leq l(v) + 1$ is satisfied.

The push-relabel method is based on the following loop. While there is an active node $u \in V$

- do *push*(u) if there is an admissible arc $uv \in E'_f$. This operation sends flow $\delta := \min\{c_f(uv), e_f(u)\}$ to the node v (excess of u decreases and excess of v increases).
- do *relabel*(u) if all arcs uv out of u are either not in E'_f or the $l(v) \geq l(u)$. This operation sets $l(u) = 1 + \min\limits_{v:uv \in E'_f} l(v)$.

We have implemented the method presented in [8] which assigns each thread to a node from the V. Every thread runs the push or relabel operation *KERNEL_ CYCLES* times before transferring the graph data to CPU memory in order to check whether the $e_f(v) = 0$ for every node $v \in V$. To perform the push operation *AtomicAdd* and *AtomicSub* are being used.

Algorithm 1 Push-relabel CUDA kernel (GPU)

Require: $u \in V$

```
1: cycle ← 0
2: while cycle < KERNEL_CYCLES do
3:     if e(u) > 0 and l(u) < |V| then
4:         l_min ← min_{v:uv∈E'_f} {l(v)}
5:         v ← arg min_{v:uv∈E'_f} {l(v)}
6:         if l_min ≥ l(u) then
7:             l(v) ← l_min + 1
8:         else
9:             δ ← min{c_f(uv), e_f(u)}
10:            AtomicAdd(c_f(vu), δ)
11:            AtomicSub(c_f(uv), δ)
12:            AtomicAdd(e_f(v), δ)
13:            AtomicSub(e_f(u), δ)
14:        end if
15:    end if
16:    cycle ← cycle + 1
17: end while
```

To make the method coverage faster the global-relabel heuristic from [2] is used. This heuristic performs backward BFS from the sink and assigns the BFS level to the label l for each node. The global-relabel heuristic is invoked on CPU each time after the graph data is transferred to the CPU memory.

The initial step in the global-relabel function is called *violation-cancellation*. At any point, the residual graph might contain a violating arc uv where $l(u) > l(v) + 1$. It is proven in [6] that this violation arc will be canceled by a push operation from the vertex u to v in the next kernel cycle. However, since these canceling push operations may not have been performed when the kernel stops, the violation-cancellation step is required.

Algorithm 2 Global-relabel heuristic (CPU)

1: **for** $uv \in E'_f$ **do**
2: **if** $l(u) > l(v) + 1$ **then**
3: $\delta \leftarrow \min\{c_f(uv), e_f(u)\}$
4: $c_f(vu) \leftarrow c_f(vu) + \delta$
5: $c_f(uv) \leftarrow c_f(uv) - \delta$
6: $e(v) \leftarrow e(v) + \delta$
7: $e(u) \leftarrow e(u) - \delta$
8: **end if**
9: **end for**
10: Let L be a FIFO list
11: Let A be an array of booleans of size $|V|$
12: Initialize A with *false*
13: L.Push$(t, 0)$
14: $A[t] = true$
15: **while** $L \neq \emptyset$ **do**
16: $(u, h) \leftarrow L$.Pop()
17: $l(u) \leftarrow h$
18: **for** $vu \in E'_f$ **do**
19: **if** $A[v] = false$ **then**
20: $A[v] = true$
21: L.Push$(v, h + 1)$
22: **end if**
23: **end for**
24: **end while**

The main loop of the algorithm invokes the CUDA kernels, then performs global-relabel method and checks whether the $e_{s,t} := -e_f(s) - e_f(t) > 0$ (note that excess of s is always negative), if that condition is not met, the algorithm finishes its execution.

Experiments show that for our graph the push-relabel heuristic significantly improves the convergence of the algorithm but due to necessity of copying the graph data from GPU to CPU the real time execution is much worse. For that reason after reaching the arbitrary value $USE\_GLOBAL\_RELABEL\_MIN$ of $e_{s,t}$, we turn off the global-relabel heuristic.

Algorithm 3 CUDA push-relabel main loop (CPU)

Initialize the preflow f
Copy graph data from CPU to GPU
$e_{s,t} \leftarrow -e_f(s)$
while $e_{s,t} > USE\_GLOBAL\_RELABEL\_MIN$ **do**
    Call push-relabel CUDA kernel
    Copy c_f, e_f and l from GPU to CPU
    Call global-relabel
    $e_{s,t} \leftarrow -e_f(s) - e_f(t)$
    Copy c_f, e_f and l from CPU to GPU
end while
while $e_{s,t} > 0$ **do**
    Call push-relabel CUDA kernel
    Copy $e(s)$ and $e(t)$ from GPU to CPU
    $e_{s,t} \leftarrow -e_f(s) - e_f(t)$
end while

3.2 Solving QMCP with circulation

To solve the QMCP, we have constructed the MCP circulation network with different cost function k_q and utilized the OR-Tools [4] circulation minimum cost solver, which is based on cost-scaling method, for our analysis. Let

$$k_q(e) = \begin{cases} 1 + q(e) \text{ if } e \in F \\ 0 \text{ if } e \in B \end{cases},$$

so $k_q : E \to [0, 2]$ and define $q(e) = q(R)$ of read R, which is represented by arc e in network.

3.3 Solving MCP with linear programming

Definition 3.1. ***Linear programming problem*** *refers to any problem that can be reduced to the so-called standard form: Find the vector* $\boldsymbol{x}$ *that maximizes* $\boldsymbol{c}^\intercal \boldsymbol{x}$ *subject to* $A\boldsymbol{x} \leq \boldsymbol{b}$ *and* $\boldsymbol{x} \geq \boldsymbol{0}$*, where* $\boldsymbol{c}$ *and* $\boldsymbol{b}$ *are any vectors, and* A *is any matrix.*

Let A be a matrix with the number of rows equal to $|S|$ (the number of indices to cover) and the number of columns equal to $|Q|$ (the number of reads). Each read R is represented as a single column in matrix A, where each covered index by this read is marked with a one and all other indices are marked with zeros.

The vector $\boldsymbol{b}$ corresponds to the function b that assigns the required coverage to each index.

The vector $\boldsymbol{x}$ assigns 1 to each read if the corresponding read is part of the solution and 0 otherwise.

The 1-MCP problem can be solved by invoking any algorithm that solves the linear programming problem in standard form (e.g., simplex). By finding the vector $\boldsymbol{x}$ that solves

$$\begin{bmatrix} I \\ -A \end{bmatrix} \boldsymbol{x} \leq \begin{bmatrix} \mathbf{1} \\ -\boldsymbol{b} \end{bmatrix}$$

and $\boldsymbol{x} \geq \mathbf{0}$.

3.3.1 CUDA quasi-MCP solver

In order to solve this problem on CUDA we've used Biconjugate gradient stabilized method method for solving linear problems. For the implementation of the algorithms, we used the sample offered by NVIDIA as an example for working with the cuSPARSE and cuBLAS libraries.

The BiCGSTAB method requires the matrix A to be square and non-singular, so we had to slightlyś modify our matrix to meet these requirements by adding additional columns with nonzero elements on the diagonal, thus we construct matrix:

$$\begin{bmatrix} I_{q\times q} & \boldsymbol{\Theta}_{n\times n} \\ -A & B \end{bmatrix},$$

where $q = |Q|$, $n = |S|$ and $B = I_{n\times n} * \text{MAX\_INT}$.

By constructing our left side matrix like this we ensure that it is square not singular and added matrix B doesn't affect the output because of it's giant value.

So the final form of the linear problem we are solving is:

$$\begin{bmatrix} I_{q\times q} & \boldsymbol{\Theta}_{n\times n} \\ -A & B \end{bmatrix} \boldsymbol{x} \leq \begin{bmatrix} \mathbf{1} \\ -\boldsymbol{b} \end{bmatrix}.$$

.

3.4 Solving QMCP with linear programming

In order to solve the QMPC problem we need to act accordingly to MCP solution and perfom addtional step.

Let $\boldsymbol{x}$ is the output of MPC solution.

The vector $\boldsymbol{c}$ corresponds to quality of given read, which is product of it's length and mapping quality.

The second call aims to find the solution to 1-QMCP: Find a vector x that maximizes $\boldsymbol{c}^\intercal \boldsymbol{x}$, assuming that

$$\begin{bmatrix} \mathbf{1}^\intercal \\ I \\ -A \end{bmatrix} \boldsymbol{x} \leq \begin{bmatrix} C \\ \mathbf{1} \\ -\boldsymbol{b} \end{bmatrix}$$

oraz $\boldsymbol{x} \geq \mathbf{0}$.

3.4.1 CUDA QMCP solver

As for the MCP problem, we used the BiCGSTAB method to solve our problem. Therefore, we had to modify our matrix to meet the requirements, and we did so in the same way as in the MCP solution, obtaining:

$$\begin{bmatrix} I_{q\times q} & \mathbf{\Theta}_{n\times n} \\ -A & B \\ \mathbf{1}^\intercal & \end{bmatrix},$$

where $q = |Q|$, $n = |S|$, and $B = I_{(n+1)\times(n+1)} \cdot \text{MAX\_INT}$.
So the final form of the linear problem we are solving is:

$$\begin{bmatrix} I_{q\times q} & \mathbf{\Theta}_{n\times n} \\ -A & B \\ \mathbf{1}^\intercal & \end{bmatrix} \boldsymbol{x} \leq \begin{bmatrix} \mathbf{1} \\ -\boldsymbol{b} \end{bmatrix}$$

and $\boldsymbol{x} \geq 0$.

4 Experiment results

4.1 Efficiency comparison

To evaluate the efficiency of algorithms and their implementations, a comprehensive test suite was designed as outlined in Table 1. The test data was generated using a consistent pseudo-random seed to ensure comparability across test executions. Additionally, the coverage distribution of the reference genome can be configured within the generator to align with specified functions. Visual representations of these coverage distributions are depicted in Figure 5.

Name	$\|S\|$	$\|Q\|$	m
uniform small	3000	100000	400
with hole small	3000	100000	400
low coverage on sides small	3000	100000	400
zero coverage on sides small	3000	100000	400
uniform medium	15000	500000	2000
with hole medium	15000	500000	2000
low coverage on sides medium	15000	500000	2000
zero coverage on sides medium	15000	500000	2000
uniform large	30000	1000000	4000
with hole large	30000	1000000	4000
low coverage on sides large	30000	1000000	4000
zero coverage on sides large	30000	1000000	4000

Table 1: Table describing test cases used for hardware execution time comparison.

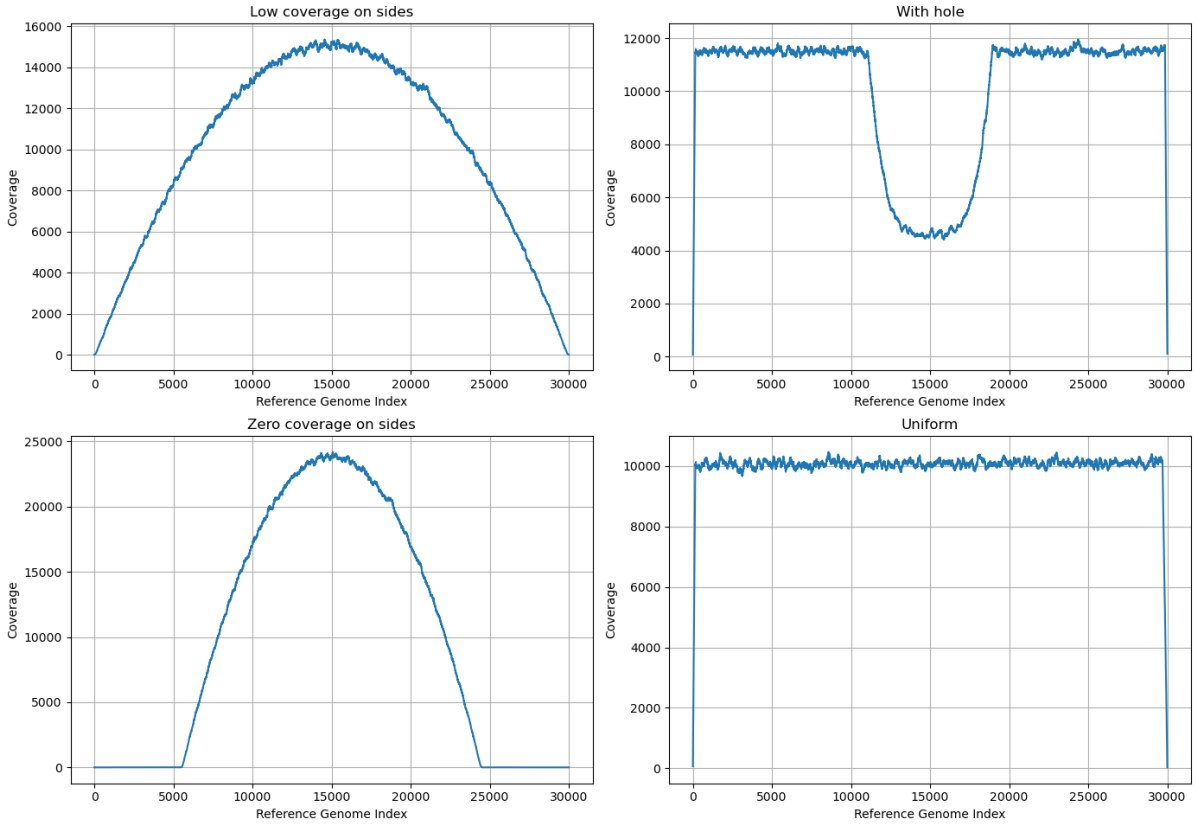

Figure 2: Example plots of 4 different coverage distributions used in test suite. $|Q| = 2000000, |S| = 30000$

The tests are categorized into three size groups: small (including uniform small, with hole small, low coverage on sides small, and zero coverage on sides small), medium (including uniform medium, with hole medium, low coverage on sides medium, and zero coverage on sides medium), and large (including uniform large, with hole large, low coverage on sides large, and zero coverage on sides large).

4.1.1 Performance of algorithms and its implementations

In this section, we aim to evaluate the performance of various algorithms and their implementations on a single machine equipped with an Intel i5-4460 processor, NVIDIA GeForce GTX 960 graphics card, and 32GB of RAM. Our evaluation encompasses four test groups: *small*, *medium*, *large* (refer to Table 1), and an additional *real* group detailed in Table 2. Unlike the pseudo-randomly generated data in other groups, the *real* group consists of real-world datasets comprising short paired-end reads of RNA from the SARS-CoV-2 virus. Standard deviation of read length for this dataset equals 15.93.

Name	$\|S\|$	$\|Q\|$	m
real M 1000	29903	2253780	1000
real M 2000	29903	2253780	2000
real M 3000	29903	2253780	3000

Table 2: Table describing *real* group test suite. It is real-world dataset of short paired-end reads of RNA from SARS-CoV-2 virus.

The execution time for the *real* group includes the time required to import the dataset from the BAM file. The tests were conducted five times, and the average execution time, along with the sample standard deviation, is depicted with error bars in Figure 3.

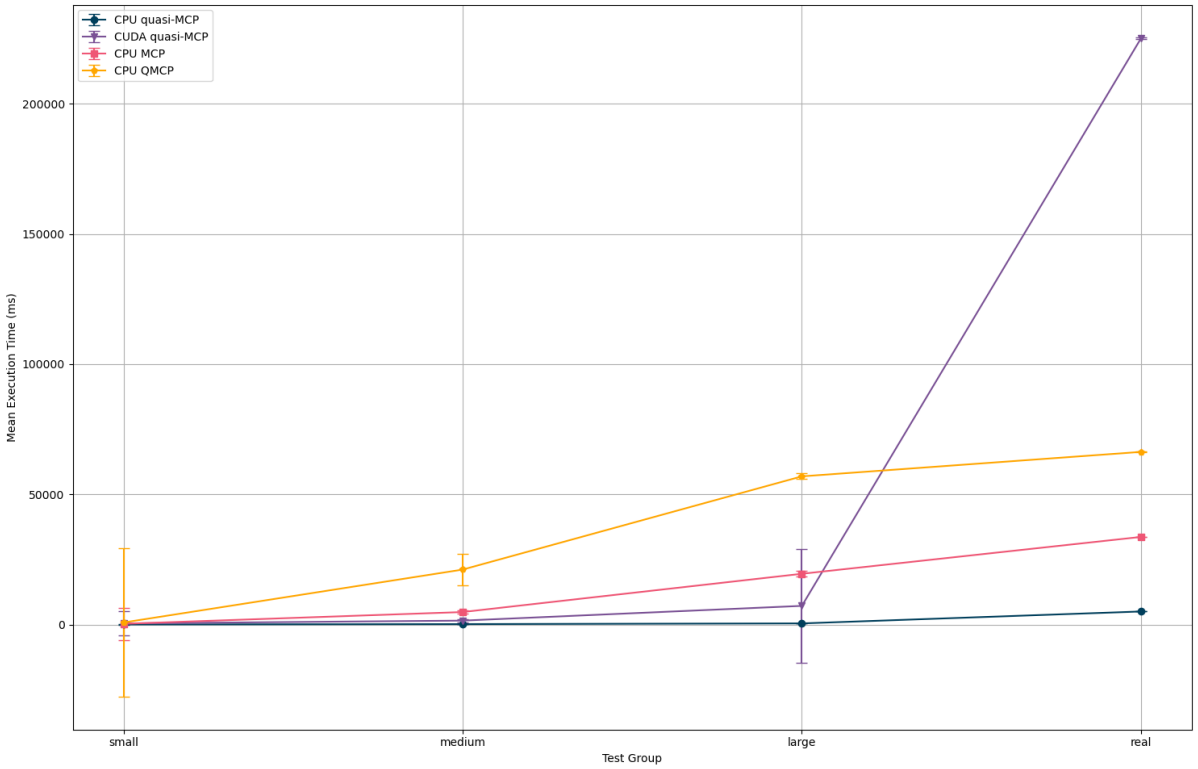

Figure 3: Comparison of execution time for every implementation for each test group.

Based on our experimental findings and after testing with sufficient quasi-MCP coverage and downsampling of read counts, we assert that using the CPU-based quasi-MCP algorithm offers the most optimal solution for real-world applications. It should be noted, however, that this approach represents an approximation of the MCP problem.

In contrast, the CUDA-based quasi-MCP algorithm demonstrates notably poorer efficiency, particularly evident in the *real* test group, and it also lacks portability. The significant increase in execution time observed with CUDA quasi-MCP for the *real* group warrants further investigation. We hypothesize that this inefficiency may be attributed to a high initial coverage value in a small subsequence of sequence S within the *real* dataset.

4.1.2 Influence of hardware

Tests were performed on 3 different machines

1. α - Intel Core i5-4460, NVIDIA GeForce GTX 960 4GB, 32GB RAM,
2. β - Intel Core i5-9400F, NVIDIA GeForce GTX 1060 6GB, 16 GB RAM,
3. γ - AMD Ryzen 9 5900HS, NVIDIA GeForce GTX 1650 Mobile 4GB, 16 GB RAM

The mean execution time of the 3 executions of each test group is considered. Tests are grouped by size (*small*, *medium*, *large*) and described in Table 1. Results of hardware influence are presented in Figure 4.

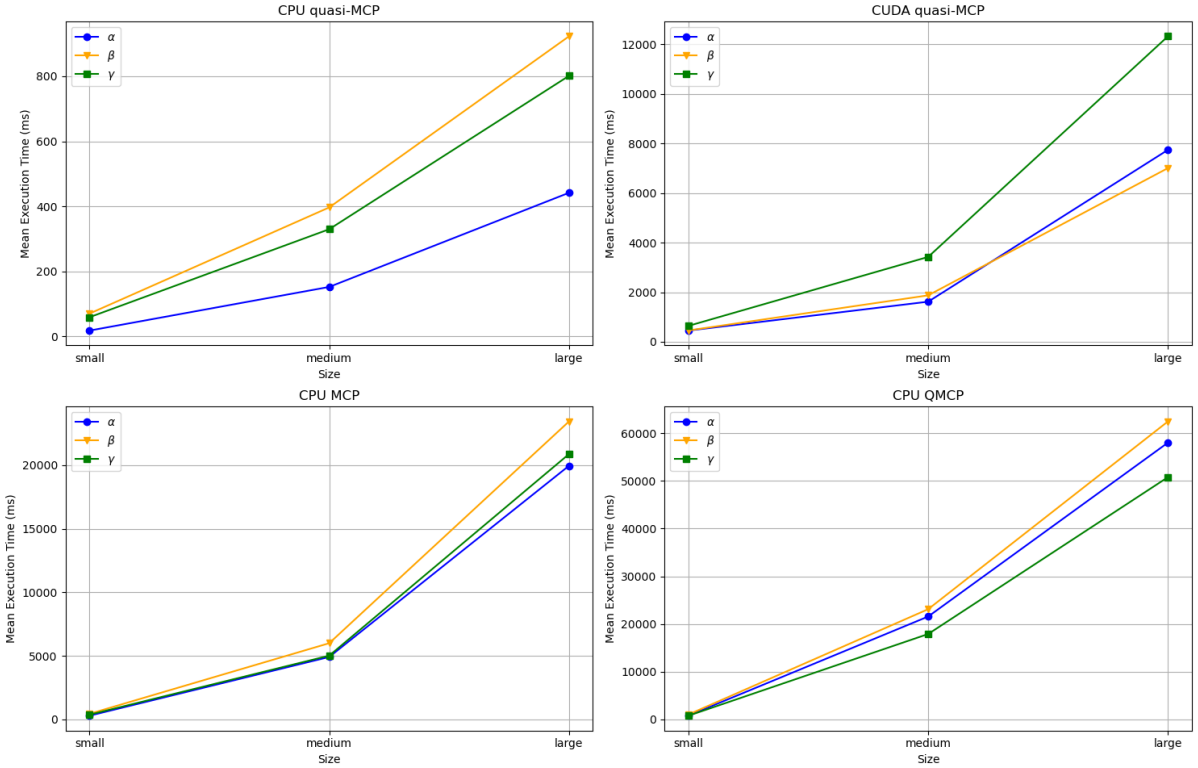

Figure 4: Comparison of execution time for every implementation for machines α, β and γ

Based on experimental data we suppose that the biggest gain in execution time by choosing better-suited hardware would be available for quasi-MCP solutions.

4.1.3 Influence of m

We conducted pseudo-randomly generated tests with a uniform distribution for the CPU quasi-MCP with $|Q| = 2000000$ and $|S| = 30000$, as well as for CPU MCP, CPU QMCP,

and CUDA quasi-MCP with $|Q| = 400000$ and $|S| = 3000$. The tests were run 10 times on the same machine, equipped with an Intel i5-4460 processor, an NVIDIA GeForce GTX 960 GPU, and 32GB of RAM. The graphs showing the average execution time along with the standard deviation are presented in Figure 5.

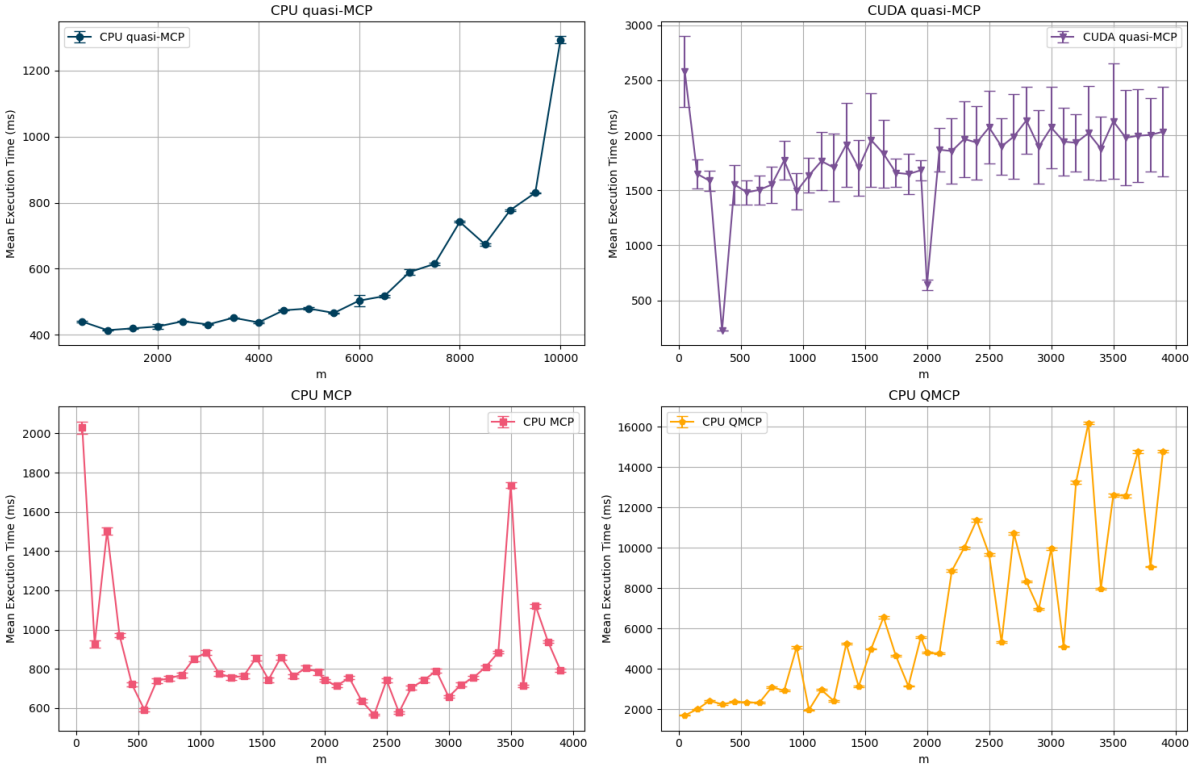

Figure 5: Comparison of execution time dependant of m value. For CPU quasi-MCP $|Q| = 2000000, |S| = 30000$, for CPU MCP, CPU QMCP and CUDA quasi-MCP $|Q| = 400000, |S| = 3000$. Pseudo-randomly generated uniform distribution of reference genome coverage.

In the implementation of CPU quasi-MCP, the execution time almost monotonically rises with the increase in the m value. A similar trend is observed with CPU QMCP, though it is not strictly defined. This may be attributed to the smaller sizes of $|Q|$ and $|S|$. The relationship between the m value and execution time for CPU MCP and CUDA quasi-MCP is not specified. It is important to conduct further analysis of the CUDA quasi-MCP results in future work, given that our experimental results showed a significant standard deviation in execution time.

4.1.4 Influence of input coverage distribution

In this section, we aim to evaluate the performance of various algorithms based on input distribution type on a single machine equipped with an Intel i5-4460 processor, NVIDIA GeForce GTX 960 graphics card, and 32GB of RAM. Our evaluation encompasses four

test groups: *low coverage on both sides*, *uniform dist*, *with hole* and *zero coverage on both sides* (they are visualized in Figure 2). The tests were conducted five times, and the average execution time, along with the sample standard deviation, is depicted with error bars in Figure 6.

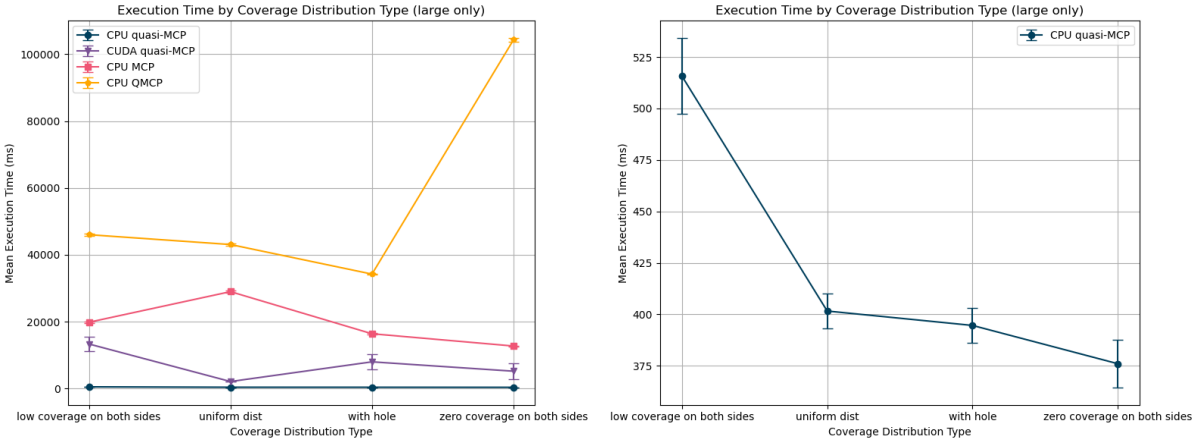

Figure 6: Comparison of execution times based on input coverage distribution type values for the *large* test group for all implementations.

Our experimental results indicate that coverage distribution significantly impacts the execution time of every method. However, the reasons why specific implementations perform significantly slower for certain distributions remain unclear and require further investigation.

4.2 Circulation based solutions

4.2.1 Quasi-MCP vs MCP

We have reviewed the experimental outcomes regarding the coverage of the reference genome for both CPU quasi-MCP and CPU MCP implementations. Each technique greatly enhances the number of reads exceeding the coverage threshold m. In figure 7, the coverage data for the extitreal group's dataset is illustrated. CPU quasi-MCP yielded 448217 reads compared to 434553 reads from CPU MCP. This indicates that our dataset was reduced by approximately 3.05% fewer reads with CPU MCP, signifying a notable improvement. It is important to note that CPU MCP exhibits a less uniform distribution of output coverage compared to CPU quasi-MCP.

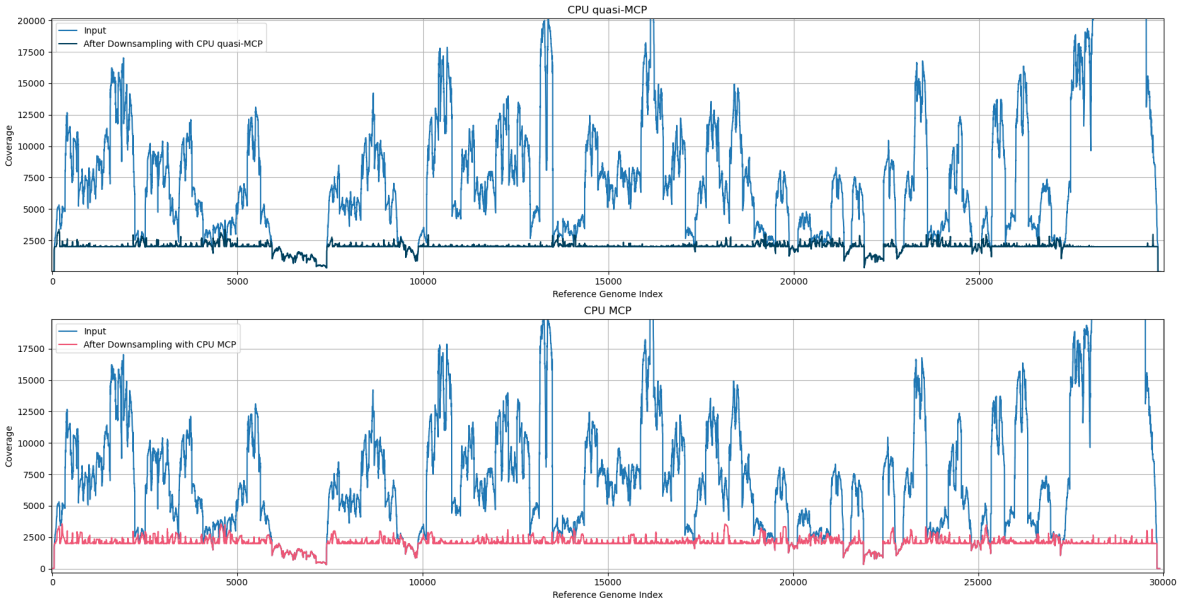

Figure 7: Comparison of coverage of results obtained using CPU quasi-MCP and CPU MCP algorithms on *real* group's dataset. CPU quasi-MCP returned 448217 reads. CPU MCP returned 434553 reads. $|Q| = 2253780, |S| = 29903, m = 2000$.

In our analysis, we conducted the identical experiment using pseudogenerated data characterized by a extitwith hole distribution. Notably, the read length of our pseudo-generated data remains constant at $|R| = 150$. The findings are illustrated in figure 8.

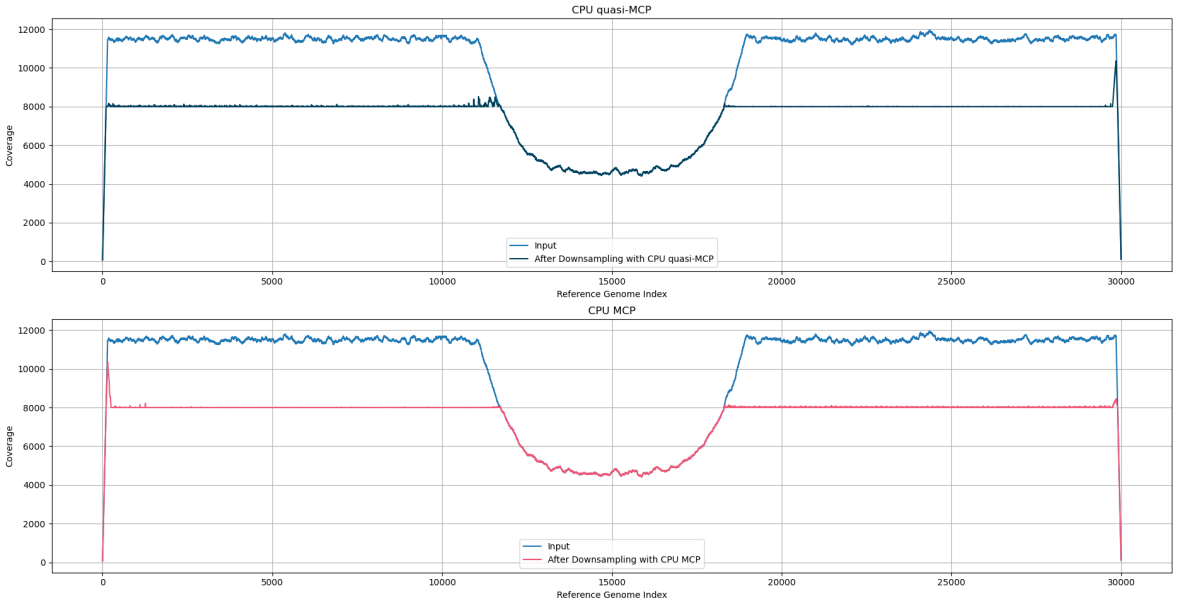

Figure 8: Comparison of coverage of results obtained using CPU quasi-MCP and CPU MCP algorithms on pseudo-generated data. Both methods returned 1477124 reads. $|Q| = 2000000, |S| = 30000, m = 8000$.

Both methods yielded an identical number of reads, suggesting that quasi-MCP might be equivalent to MCP if $\forall_{R \in Q}\ |R| = const$. This hypothesis requires further validation or refutation, which should be addressed in future studies.

4.2.2 CUDA quasi-MCP maximum flow solver convergence

In this section, we compare the convergence of $e_{s,t}$ in the CUDA quasi-MCP algorithm between two approaches: with and without the global-relabel heuristic. The experiments show (Figure 9) that running the algorithm without global-relabel heuristic requires more main-loop iterations in order to achieve the $e_{s,t} = 0$ than the variant which uses global-relabel heuristic in every main loop iteration.

Despite the lower number of iterations, the global-relabel-based method has worse execution times (Table 3), which is caused by the necessity of copying the graph data from GPU memory to CPU. That's the reason why the $USE\_GLOBAL\_RELABEL\_MIN$ value is used.

Better approach to solve that issue is presented in [9], where the algorithm changes between the CUDA kernel and CPU implementation of the push-relabel algorithm basing on arbitrator. However, the execution time, portability, and simplicity of the CPU quasi-MCP algorithm seem to overwhelmingly favor it over any GPU solution.

Test	With global-relabel [s]	No global-relabel [s]
With hole	56.0167	11.5542
Uniform	9.8260	8.4621
Low coverage on both sides	652.978	22.9461

Table 3: Execution time for the CUDA quasi-MCP algorithm with and without the global-relabel heuristic. Tested on AMD Ryzen 9 5900HS, NVIDIA GeForce GTX 1650 Mobile 4GB, 16 GB RAM.

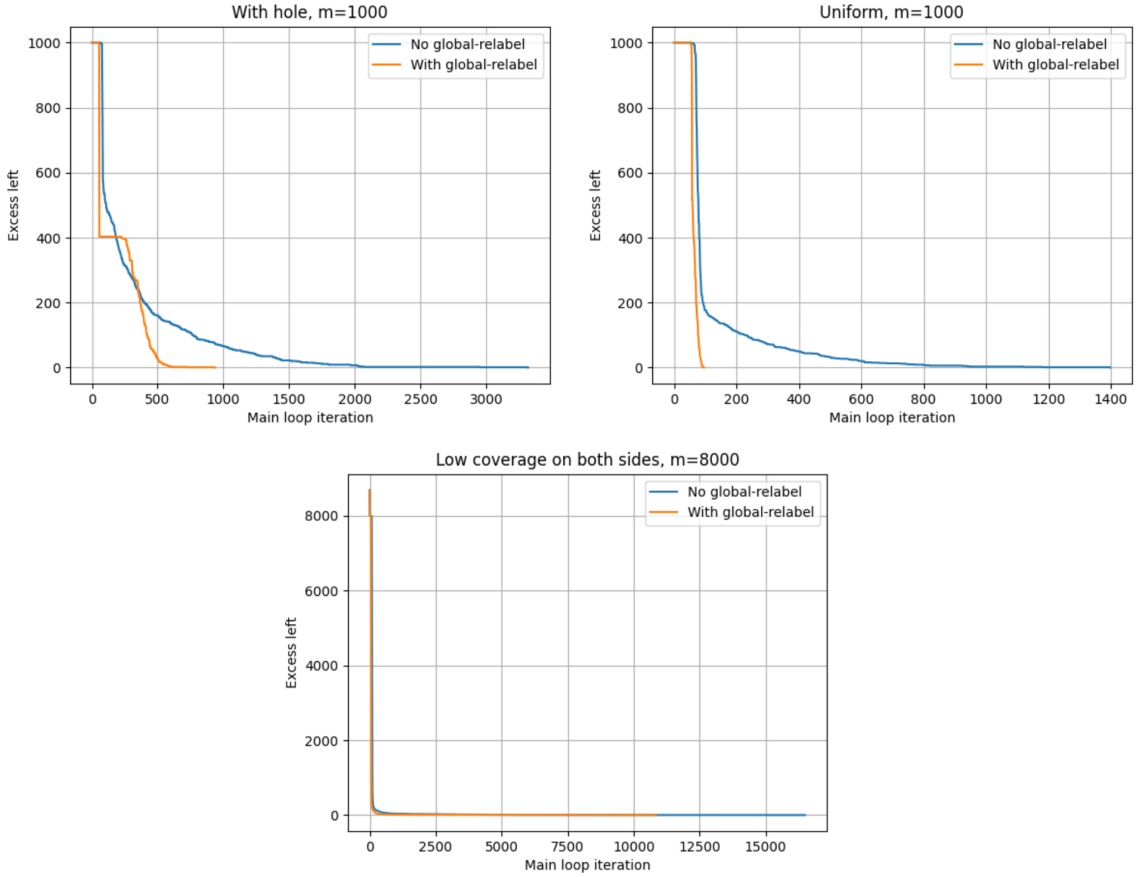

Figure 9: Convergence of the CUDA quasi-MCP algorithm: Analyzing the relationship between the number of main loop iterations and the $e_{s,t}$ value.

4.3 Linear programming based solutions

The results of our experiments indicate that methods based on linear programming require a different approach. The critical issue is that the $\boldsymbol{x}$ vector must consist solely of ones and zeros. The linear programming solution doesn't take it into account at treats it as vector of floats. Therefore, the algorithm does not deterministically specify whether we should take given read or not.

Another limitation of the linear programming approach is its substantial memory usage. Despite our efforts to minimize memory consumption by employing the *cuSPARSE* interface for storing sparse matrices, the process still demanded at least 8GB of VRAM to handle sample data.

5 Conclusions and Future Remarks

The real-world problem that forms the basis of our work is quite specific, focusing on the SARS-CoV-2 virus genome and its short paired-end alignments. It is crucial to test our implementations on different datasets and to consult bioinformaticians for additional constraints in the downsampling process, particularly in QMCP. From a computer science perspective, it is essential to test the limits of our approximations to determine their applicability and identify scenarios where they might fail to produce sufficient results. If such research is conducted, downsampling software could dynamically switch between implementations based on the provided input data. This would greatly enhance user experience and save researchers' time.

Our experiments demonstrate that the CPU quasi-MCP maximum flow-based solution, although approximate, yields satisfactory results with significant execution time improvements. The data suggest (we need a formal proof) that this algorithm may be optimal when all reads are of uniform length. Future research should investigate the influence of read length on the accuracy of the algorithm.

Maximum flow based solutions have been tested with push-relabel approach, and the minimum cost with cycle-cancelling and cost-scaling. It may be valuable to test other flow algorithms that might be beneficial for the graph structure we have proposed. For example augmentation path methods might be faster for the sparse graphs.

Despite the challenges with linear programming, we still believe it can be used for solving QMCP efficiently on CUDA. However, it is crucial to explore algorithms suited for discrete problems and develop a custom interface for cuBLAS based on cuSPARSE, better suited for our sparse matrix structure, to minimize the memory usage of our algorithm.

References

[1] René van Bevern et al. *Approximability and parameterized complexity of multicover by c-intervals*. 2015. URL: https://www.sciencedirect.com/science/article/pii/S0020019015000423?ref=pdf_download&fr=RR-2&rr=862c48e089113494.

[2] Boris V. Cherkassky and Andrew V. Goldberg. "On implementing push-relabel method for the maximum flow problem". en. In: *Integer Programming and Combinatorial Optimization*. Ed. by Gerhard Goos et al. Vol. 920. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 157–171. ISBN: 978-3-540-59408-6 978-3-540-49245-0. DOI: 10.1007/3-540-59408-6_49. URL: http://link.springer.com/10.1007/3-540-59408-6_49 (visited on 06/26/2024).

[3] Liang Ding, Bin Fu, and Binhai Zhu. *Minimum Paired-End Interval Cover and Its Application*. 2011. URL: https://www.gsoc.montana.edu/bhz/doc/tcbb12-1.pdf.

[4] Google. *OR-Tools*. https://developers.google.com/optimization. Version 9.9.3963. Algorithms used: Max Flow, Min Cost Max Flow. 2024. URL: https://developers.google.com/optimization.

[5] Dorit S. Hochbauma and Asaf Levin. *Cyclical scheduling and multi-shift scheduling: Complexity and approximation algorithms*. 2006. URL: https://www.sciencedirect.com/science/article/pii/S1572528606000454?ref=pdf_download&fr=RR-2&rr=864c81ce09a63512.

[6] Bo Hong. "A lock-free multi-threaded algorithm for the maximum flow problem". en. In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. ISSN: 1530-2075. Miami, FL, USA: IEEE, Apr. 2008, pp. 1–8. ISBN: 978-1-4244-1693-6. DOI: 10.1109/IPDPS.2008.4536352. URL: http://ieeexplore.ieee.org/document/4536352/ (visited on 06/15/2024).

[7] Heng Li, Jue Ruan, and Richard Durbin. *Mapping short DNA sequencing reads and calling variants using mapping quality scores*. 2008. DOI: https://doi.org/10.1101%2Fgr.078212.108. URL: https://pubmed.ncbi.nlm.nih.gov/18714091/.

[8] Jiadong Wu, Zhengyu He, and Bo Hong. "Efficient CUDA Algorithms for the Maximum Network Flow Problem". en. In: *GPU Computing Gems Jade Edition*. Elsevier, 2012, pp. 55–66. ISBN: 978-0-12-385963-1. DOI: 10.1016/B978-0-12-385963-1.00005-8. URL: https://linkinghub.elsevier.com/retrieve/pii/B9780123859631000058 (visited on 06/12/2024).

[9] Zhengyu He and Bo Hong. "Dynamically tuned push-relabel algorithm for the maximum flow problem on CPU-GPU-Hybrid platforms". In: *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)* (Apr. 2010). Conference Name: Distributed Processing (IPDPS) ISBN: 9781424464425 Place: Atlanta, GA Publisher: IEEE, pp. 1–10. DOI: 10.1109/IPDPS.2010.5470401. URL: http://ieeexplore.ieee.org/document/5470401/ (visited on 06/12/2024).