Next Generation Cluster Editing

Thomas Bellitto[1⋆], Tobias Marschall[2],
Alexander Schönhuth[2], and Gunnar W. Klau[2]

[1] Department of Computer Science and Telecommunications,
ENS Cachan/Rennes, France
[2] Life Sciences, Centrum Wiskunde & Informatica (CWI),
Amsterdam, The Netherlands

Abstract. This work aims at improving the quality of structural variant prediction from the mapped reads of a sequenced genome. We suggest a new model based on cluster editing in weighted graphs and introduce a new heuristic algorithm that allows to solve this problem quickly and with a good approximation on the huge graphs that arise from biological datasets.

Keywords. bio-informatics; next-generation sequencing; graph theory; combinatorial optimisation; cluster editing; approximation algorithm

Introduction

Structural variations in the human genome play a key role in genetic diversity. Many diseases have been associated with genetic alterations and locating them can be valuable for developing appropriate treatments. Thus, analysing and understanding human genetic variations has become a key issue in many fields of medicine and biology, and the advent of next-generation sequencing enabled the creation of huge databases on human genomes. However methods for exploiting the data and detecting the structural variations have not kept up with sequencing. There is ample evidence that existing datasets contain variations that the current methods cannot discover.

Many variant finders were developed in recent years to address this problem, but discovering small insertions and deletions is still a very challenging problem. Our works are based on Clever (CLique Enumerating Variant findER), a tool introduced in 2012 in [12] that achieves very good results and compares favourably to the other state-of-the-art tools. However, Clever is based on clique enumerating, a model that leads to problems we know how to solve exactly but that shows some flaws. In order to improve the results of Clever, we designed a new approach based on a different optimisation problem: cluster editing. The new model seems more appropriate but leads to a difficult problem that has to be solved on huge data structures.

In the first part of the report, we will describe the principle and the work scheme of Clever and point out the problems we want to solve and that justify

⋆ This work was done during an internship of the first author at CWI.

the cluster editing model. We will then describe the state of the art on cluster editing and expand on a particular class of graphs: point graphs.

The second part presents our contribution to the new model. We study exact and heuristic algorithms to solve cluster editing in weighted point graphs. We then use it to design a fast heuristic algorithm suited to biological instances. We finally present experimental results to validate the model and the heuristic.

We believe that the validation of this new model and the theoretical study that has been undertaken can give a new direction for future research regarding Clever and can lead to significant improvements of the results.

1 Context

1.1 Clever

Principle of Clever A paired-end read is a fragment of DNA whose extremities have been sequenced. We then look for those sequences in a reference genome to align the paired-end read on the genome (see figure 1).

According to their sizes, alignments are more or less likely to indicate insertions or deletions, and the positions of the reads on the genome will help us locating the variations.

Fig. 1. The sequenced fragments we align are displayed in red. According to their sizes, alignments may indicate deletions (left) or insertions (right).

Given a dataset that usually consists of millions of reads, Clever finds variations by processing as described:

1. Clever organises those reads in what we call a read alignment graph. Clever then draws edges between reads who are likely to stem from the same allele and enumerates maximal cliques, that represent contradiction-free groups of reads. More details about read alignment graphs and the clique enumerating algorithm are given in the next part.
2. At this stage, we have many cliques that we assume to contain reads that stem from the same allele. We compute for each clique a p-value that indicates how likely it is to observe those reads under the hypothesis that there were no insertions and a p-value with the hypothesis that there were no deletions. Predictions will therefore come from the lowest p-values.
3. Finally, Clever uses the Benjamini-Hochberg process [3] to control false discovery rate: if we have c clusters, to control the false discovery rate at

r, we will return the biggest number p of predictions such that all their p-values are under $\frac{rp}{c}$. As we use different formulas to compute the p-values of insertions and deletions, Clever will apply this process for the insertions and the deletions separately. In practice, we often choose a 10% false discovery rate.

Read Alignment graph The read alignment graph is a graph where each vertex represents a read and where we draw edges between statistically "close" reads. To draw an edge, we want the two following conditions to hold:

- the reads must be close in position (once they are aligned on the reference genome), meaning that the overlap between the reads must be big enough
- the sizes of the reads must be close enough.

Formally, we call $I(A)$ the length of a read A. In practice, we can assume that the length of the reads are normally distributed. We call μ the mean value and σ the standard deviation.

Let A and B be two reads:

- We call $\Delta(A,B) = |I(A) - I(B)|$ the length difference
- We call $O(A,B)$ the size of the overlap
- We call $\overline{I}(A,B) = \dfrac{I(A)+I(B)}{2}$ the mean lengths
- We define $U(A,B) = \overline{I}(A,B) - O(A,B)$

Let $X \sim \mathcal{N}_{(0,1)}$. If A and B are two different overlapping reads, we will draw an edge between A and B iff:

$$\begin{cases} \mathbb{P}\left(|X| \geqslant \dfrac{1}{\sqrt{2}}\dfrac{\Delta(A,B)}{\sigma}\right) \leqslant T & \text{(size criteria)} \\ \mathbb{P}\left(X \geqslant \sqrt{2}\dfrac{U(A,B)-\mu}{\sigma}\right) \leqslant T & \text{(overlap criteria)} \end{cases}$$

where T is a threshold value that will determine the graph density.

We can also use those criteria to weight our edges. The weight $w(A,B)$ of the edge between A and B is given by the formula:

$$w_{\text{size}}(A,B) = \ln T - \ln\left(\mathbb{P}\left(|X| \geqslant \frac{1}{\sqrt{2}}\frac{\Delta(A,B)}{\sigma}\right)\right)$$

$$w_{\text{overlap}}(A,B) = \begin{cases} -\infty & \text{if } O(A,B) = 0 \\ \ln T - \ln\left(\mathbb{P}\left(X \geqslant \sqrt{2}\dfrac{U(A,B)-\mu}{\sigma}\right)\right) & \text{if } O(A,B) \geqslant 0 \end{cases}$$

$$w(A,B) = \min(w_{\text{size}}(A,B), w_{\text{overlap}}(A,B))$$

As most of the reads do not overlap at all (actually, the proportion of overlapping reads is $O(\frac{1}{n})$), the presence/absence of most of the edges (unweighted case) or their weight will often be determined by the overlap. This makes unweighted read alignment graphs look like interval graphs, a well-studied class of graphs.

Definition 1 (interval graph). *Given a set $\mathcal{I}$ of intervals, the associated interval graph is the graph $G = (\mathcal{I}, \{(A, B) \in \binom{\mathcal{I}}{2} \mid A \cap B \neq \varnothing\})$. We represent each interval with a vertex and we draw edges between overlapping intervals.*

There are $O(n \times k)$-time exact algorithms for the maximum clique enumerating problem in interval graphs, where k is a bound on the size of the neighbourhood of the vertices and can in practice be assumed constant. As reads that overlap do not have to be connected in read alignment graphs because of size difference, read alignment graphs are only interval graphs with missing edges, which is a trivial property as cliques are interval graphs. However, even if it is difficult to write it formally, at low coverage, read alignment graphs will be "close" to interval graph and exact algorithms that run in polynomial time for interval graphs will still behave polynomially.

Here is the algorithm used by Clever: (we use the notation $N(A)$ to denote the neighbourhood of A)

Algorithm 1: Clique enumerating [12]

```
let L be the sorted list of the 2n endpoints of the reads
𝒞 ← ∅ will be the set of active cliques
for p ∈ L do
    if p is the left endpoint of a read A then
        if ∀C ∈ 𝒞, N(A) ∩ C = ∅ then
            𝒞 ← 𝒞 ∪ {A}
        else
            for C ∈ 𝒞 do
                if C ∩ N(A) = C then
                    C ← C ∪ {A}
                if C ∩ N(A) = ∅ then
                    𝒞 ← 𝒞 ∪ {(C ∩ N(A)) ∪ {A}}
    else
        p is the right endpoint of a read A: for C ∈ 𝒞 do
            if A ∈ C then
                𝒞 ← 𝒞 \ C
                return 𝒞
```

It is established in [12] that this algorithm runs in time $O(n(\log n + kc^2) + s)$ where k is an upper bound on the size of the cliques, c is an upper bound on

the size of the set of active cliques, and s is the size of the output. As we said previously, every graph could be seen as an interval graph with missing edges, but in the general case, c cannot be bounded with a polynomial expression. However, with interval graphs, $c \leqslant k$ and in practice, we can assume that with our read alignment graphs, $c = O(k)$.

Limits of Clever First of all, let us take a closer look at the model. We say that two sets of vertices overlap iff their intersection is non empty. Clever makes predictions on groups of reads that are supposed to stem from the same allele, thus, it makes no sense for two different groups to overlap, but we form those groups by enumerating maximal cliques that will always overlap.

Just like reads, predictions are said to overlap if and only if both starts before the other end. Thus, it does not make sense to predict, for example, two overlapping insertions in a genome. However, the predictions of Clever may contain overlapping insertions or deletions.

We still can use a post-processing script that deletes predictions that overlap with others until all the predictions are disjoint. However, dropping information will damage the quality of the predictions: what Clever optimises is the file of overlapping predictions and not the post-processed file. Ultimately, we would like to find a model that avoids overlapping predictions or at least, minimises them so that the optimised file (with overlap) is as close as possible to the final post-processed file.

Another completely different problem arises when we try to run Clever on higher coverage data, e.g. to analyse viral populations (see [1]). The $O(n(\log n + kc^2) + s)$ complexity will no longer be acceptable as k and c will reach values that are too high.

1.2 Optimal cluster editing problem

Definition

Unweighted case

Given an undirected unweighted graph $G = (V, E)$, the cluster editing problem consists of transforming G into a clustered graph G', *i.e.* into a union of disjoint cliques by adding or removing edges.

A solution to this problem can be represented by the set $R \subset E$ of edges to remove and the set $A \subset \binom{V}{2} \setminus E$ of edges to add, or alternatively by a partition of V showing which clusters have been made.

The cost of a solution is the number of modifications made on G: $|A| + |R|$.

The optimal cluster editing problem consists in finding the solution with the smallest cost.

Example 1. Let G be the graph shown in figure 2.

The solution presented in figures 3 and 4 can be written $(\{(4,5)\}, \{(1,4)\})$ or $\{\{1,2,3,4\}, \{5,6,7\}\}$ and has cost 2.

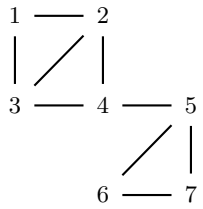

Fig. 2. Initial graph G

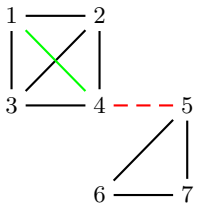

Fig. 3. Transformations on G

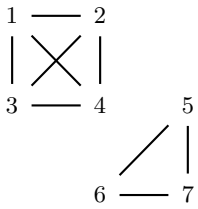

Fig. 4. Clustered graph G'

Weighted case

Definition 2 (weighted graph).
A weighted graph is an ordered pair (V, w), where V is the set of vertices of the graph and $w : \binom{V}{2} \to \overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$ is the weight function. We consider that there is an edge between vertices a and b iff $w(a, b) \geqslant 0$.

Note that with this definition, even non-edges have a weight.

Definition 3 (positive/negative part of a function). *The positive and negative part of a function f, respectively written f^+ and f^- are defined on the same domain as f by $f^+(x) = \max(f(x), 0)$ and $f^-(x) = \max(-f(x), 0)$.*

The weighted optimal cluster editing problem still consists of transforming the initial graph $G = (V, w)$ into an unweighted clustered graph G', but this time, the edges have a deletion/insertion cost given by their weight. Thus, the cost of inserting an edge in G' between a and b is $w^-(a, b)$ and the cost of removing an edge is $w^+(a, b)$. Note that the unweighted case is a special case of the weighted problem where every edge has weight 1 and every non-edge has weight -1.

State of the art The problem of cluster editing can be seen as a compromise between high edge-density community searching and low cut searching, two NP-hard problems, and as we can expect, the cluster editing problem itself has been proven NP-hard by Jan Kratochvíl and Mirko Krivánek in [8]. The problem is even APX-hard, that is, it is NP-hard to approximate the optimal solution with a given approximation factor $k > 1$ [4], and the best deterministic approximation factor reached today with a polynomial time algorithm is 2.5 [16]. Deciding whether the cluster editing problem is NP-complete in the case of interval graphs is still an open question.

The cluster editing problem has numerous applications in various fields, like data processing, visualisation and bio-informatics. Returning to Clever, we were looking for a way to divide our reads into non-overlapping communities and cluster editing is a natural way to do so (note that clustered graphs are the only ones whose maximal cliques do not overlap). As for avoiding overlapping

predictions, cluster editing is not completely sufficient: two clusters that do not overlap (i.e. that are vertex-disjoint) still may contain reads that overlap. However, this new model is more proper, reduces the number of overlapping predictions and therefore makes the post-processing step less damaging for the quality of the predictions. Moreover, this problem provides us a natural way to relax the clique definition (one missing edge, which can be due to a misread, can have decisive consequences on cliques, while clusters can support it). Cluster editing also has the advantage of being compatible with weighted graphs.

Indeed, each pair of reads will overlap more or less and be more or less close in size. It is therefore appreciable that the weighted cluster editing problem allows us to make some edges heavier than others, while clique enumerating requires a less adequate "all or nothing" model for edges.

However, cluster editing is a difficult problem to solve, even approximately, and the size of the biological data to be dealt with only allows linear time and space algorithms. We will therefore need large approximations to solve cluster editing while the clique enumerating problem still can be solved exactly on the read alignment graphs.

There are already many heuristic cluster algorithms, well known examples are CAST [2], CLICK [14] or HCS [6], but the extreme size of the data and the special structure of the read alignment graphs, which allows solving clique enumerating in linear time, lead us to design a new specific algorithm.

A special case: one dimensional point graph

Definition 4 (point graph).

- *Given a set of points V in an n-dimensional metric space (M, d) and a threshold distance $l \in \mathbb{R}^+$, the induced point graph is the unweighted graph* $G = \left(V, E = \{(a, b) \in \binom{V}{2} \mid d(a, b) \leqslant l\}\right)$
- *An unweighted graph G is said to be an n-dimensional point graph iff there exists a n-dimensional metric space (M, d), a threshold distance l and a set of points $V \subset M$ which induces G.*

Given a one dimensional point graph, being able to place those vertices on a line provides us an order on the vertices (from left to right e.g.) and allows us to talk about consecutive vertices. The main result about cluster editing on point graphs is the following:

Theorem 1 (Mannaa [11]). *Given a one dimensional point graph, there exists an optimal clustering where all the clusters are made of consecutive vertices.*

This result considerably reduces the size of the set of potential solutions to investigate and makes this set much easier to enumerate. Instead of enumerating the whole set of partitions of V, we only scan the points from left to right and have to determine whether we end the cluster or not after each vertex.

A dynamic programming algorithm [11] follows: for $j \in [\![0, |V|]\!]$, we call $\mathrm{opt}(j)$ the cost of optimally clustering vertices 0 to j and for $j \in [\![0, |V|]\!]$, $i \in [\![0, j]\!]$, we call $\mathrm{opt}'(j, i)$ the cost of the best clustering for vertices 0 to j, where the last cluster has size $i + 1$. We thus have:

$$\mathrm{opt}(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min\limits_{0 \leqslant i \leqslant j} \mathrm{opt}'(j, i) & \text{if } j > 0 \end{cases}$$

$$\mathrm{opt}'(j, i) = \begin{cases} \mathrm{opt}(j-1) + \deg^+(v_j) & \text{if } i = 0 \\ \mathrm{opt}'(j-1, i-1) + |i - \deg^+(v_j)| & \text{if } 0 < i \leqslant j \end{cases}$$

Where $\deg^+(v)$ denotes the out degree of a vertex v *i.e.* the number of arcs starting at v.

We can compute all opt' value in $O(n^2)$ time and space:

Algorithm 2: Computing the values of opt'

```
opt'(0,0) ← 0
for j from 1 to n − 1 do
    opt'(j,0) ← deg+(v_j) + min_{0⩽i⩽j−1} opt'(j−1,i)
    for i from 1 to j do
        opt'(j,i) ← |i − deg+(v_j)| + opt'(j−1,i−1)
```

We then know which cluster to make by searching where the function $\mathrm{opt}'(j, \cdot)$ reaches its minimum. Once we know opt', it can be processed in $O(n)$.

Application

As previously said, our read alignment graphs fail to be interval graphs because of size differences between the reads: if the size is too different, there will not be an edge between the two vertices no matter how much they overlap. If we only take into account reads whose sizes are close enough to allow edges, we would have an interval graph, but, as NP-completeness of cluster editing on interval graphs still is an open problem, there is no guarantee that interval graphs would be of any help. But by considering only reads whose sizes are close, we end up having not only an interval graph, but a one dimensional point graph whose points set is for example the left endpoint of the intervals, and whose threshold distance is their length. If we sort the reads by their left endpoint, we would then be able to use the 1D point graph algorithm of Mannaa.

2 Contribution

2.1 Algorithm on weighted point graphs

Exact algorithm We define a weighted point graph as follows:

Definition 5 (weighted point graph).

- *Given a set of points V in an n-dimensional metric space (M, d) and a decreasing function $f : \mathbb{R}^+ \to \overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$, the induced point graph is the weighted graph $G = (V, w : (a, b) \mapsto f(d(a, b)))$*
- *A weighted graph G is said to be an n-dimensional weighted point graph iff there exists a n-dimensional metric space (M, d), a decreasing function $f : \mathbb{R}^+ \to \overline{\mathbb{R}}$ and a set of point $V \subset M$ which induce G.*

Remark 1.

- An unweighted point graph is a special case of weighted point graph with

$$f : \begin{cases} \mathbb{R}^+ & \to & \overline{\mathbb{R}} \\ a & \mapsto & \begin{cases} 1 & \text{if } a \leqslant l \\ -1 & \text{if } a > l \end{cases} \end{cases}$$

- Suppose now we create a weighted read alignment graph with only reads of the same size. The closer the left endpoint of two reads will be, the more they will overlap and the heavier the edge will be. If all the reads have the same size, weighted read alignment graphs are one dimensional weighted point graphs, like in the unweighted case.

  With this definition, the proof of theorem 1 (see [11]) still holds:

Theorem 2. *Given a one dimensional weighted point graph, there exists an optimal clustering where all the clusters are made of consecutive vertices.*

We now have:

$$\text{opt}(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min\limits_{0 \leqslant i \leqslant j} opt'(j, i) & \text{if } j > 0 \end{cases}$$

$$\text{opt}'(j, i) = \begin{cases} \text{opt}(j-1) + \sum\limits_{k=0}^{j-1} w^+(j, k) & \text{if } i = 0 \\ \text{opt}'(j-1, i-1) + \sum\limits_{k=0}^{j-1} w^+(j, k) + \sum\limits_{k=j-i}^{j-1} w^-(j, k) & \text{if } 0 < i \leqslant j \end{cases}$$

The formula for each of the $O(n^2)$ values of opt$'$ now includes a sum that takes $O(n)$ time to compute. Still, those sums are often alike and by computing only one of them, we can deduce the next one in constant time and we still end up with an $O(n^2)$ algorithm:

Algorithm 3: Computing the values of opt' in a weighted graph

```
1 opt'(0,0) ← 0
2 for j from 1 to n − 1 do
3 |   X ← Σ_{0⩽k⩽j−1} w⁺(j,k)
4 |   opt'(j,0) ← X + min_{0⩽i⩽j−1} opt'(j−1,i)
5 |   for i from 1 to j do
6 |   |   X ← X − w(j,j−i)
7 |   |   opt'(j,i) ← X + opt'(j−1,i−1)
```

Remark 2. If we do not initialize X with $\sum_{0\leqslant k\leqslant j-1} w^+(j,k)$, $\mathrm{opt}(j)$ will no longer give the cost of the optimal clustering for vertices 0 to j. However, X is nothing but an additive constant in $\mathrm{opt}'(j,\cdot)$ and what we are really interested in is to determine where the minimum of $\mathrm{opt}'(j,\cdot)$ is reached. If we initialize X with any other value, for example, zero, we will still find the optimal clustering.

Heuristic algorithms When the number of nodes of the input graph reaches hundreds of millions, even $O(n^2)$ algorithms are not suitable anymore. However, we can be much faster with an acceptable approximation by noticing the following: if the optimal clustering for nodes 0 to j is reached with a last cluster of size i, chances are very low that the optimal clustering for nodes 0 to $j+1$ is reached with a last cluster of size strictly bigger than $i+1$ and we do not need to compute $\mathrm{opt}'(j+1,k)$ for $k \in [\![i+2, j+1]\!]$. There are of course counter examples, especially with small clusters, but this approximation is generally safe when clusters exceed hundreds of vertices.

We therefore obtain a first heuristic algorithm by making the following change on line 5 of the exact algorithm:

```
5 for i from 1 to argmin(opt'(j,·)) + 1 do
```

We can however, make our heuristic more accurate by noticing that most of the time when errors happen, there was still a positive edge between the vertex j and the vertex $j-i$.

Line 5 of the second algorithm is:

```
5 for i from 1 to max(argmin(opt'(j,·)) + 1, max{i | w(j,j−i) > 0}) do
```

If we initialize X with 0 as suggested in note 2, our two algorithms now run in $O(n \times k)$ where k is the size of the clusters (we will not, however, be able to determine the cost of the provided solution in less than $O(n^2)$). Finally, we can note that to determine the clustering, all we need to know at the end are the $\mathrm{argmin}(\mathrm{opt}'(j,\cdot))_{0\leqslant j\leqslant n}$. Once we are done computing $\mathrm{opt}'(j,\cdot)$, we then can erase the values of $\mathrm{opt}'(j-1,\cdot)$ as long as we keep memory of $\mathrm{argmin}(\mathrm{opt}'(j-1,\cdot))$. This reduces the space complexity of the algorithm from $O(n^2)$ to $O(n)$.

Experimental results Our tests generate random weighted graphs of size n by placing n points randomly on $[0, 1]$. We then choose a threshold distance l, the distance between two points a and b is of course $|a - b|$ and we create the graph by composing the distance with the function:

$$f_l : \begin{cases} \mathbb{R}^+ & \to & \overline{\mathbb{R}} \\ a & \mapsto & \dfrac{l^2 - a^2}{la} \end{cases}$$

This function was chosen because of three intuitive properties we expected from the weight: $f_l(0) = +\infty$, $f_l(l) = 0$ and $\lim_{x \to +\infty} f_l(x) = -\infty$.

The values of the threshold distance l will determine the edge density of the graph and therefore the size c of the clusters. All the results in the following table are average results on 1,000 measures on graphs of 10,000 vertices. Those results aim at showing the speed (through the number of opt$'$ values it computes) and the accuracy (through the cost of the answer it returns) of both heuristics but also the impact of the size of the clusters.

$\underline{l = 0.1:}$
average number of clusters (exact solution): 6.121
average size of clusters: 1633.7

	Cost of the solution	Number of opt$'$ values calculated
Exact algorithm	5,269,293.579	49,995,000.000
Second heuristic algorithm	5,269,293.579	14,734,267.491
First heuristic algorithm	5,269,293.579	14,733,439.917

$\underline{l = 0.01:}$
average number of clusters (exact solution): 62.809
average size of clusters: 159.21

	Cost of the solution	Number of opt$'$ values calculated
Exact algorithm	535,093.568	49,995,000.000
Second heuristic algorithm	535,094.783	1,525,308.148
First heuristic algorithm	535,283.994	1,515,427.532

$\underline{l = 0.001:}$
average number of clusters (exact solution): 644.327
average size of clusters: 15.520

	Cost of the solution	Number of opt$'$ values calculated
Exact algorithm	40,964.600	49,995,000.000
Second heuristic algorithm	40,968.290	139,936.991
First heuristic algorithm	41,378.066	132,794.486

We can see that the number of calculations the heuristics do is proportional to the size of the clusters. As it was mentioned previously, the chances that errors are made are getting very low when the clusters grow bigger. We can also see that for the price of few additional calculations, the second heuristic avoids most of the mistakes made by the first one.

2.2 Back to the read alignment graph

Comparison with the one dimensional case As knowing the left endpoint and the length of every read gives us the required information to determine the weights of the edges, we can place our reads in a two-dimensional plane and look for a metric and a weight function that would make it a point graph. However, dynamic programming was only possible because of the very special structure of the set of possible solutions theorem 2 leaves to investigate. As this theorem no longer holds in two dimensions, there is no easy way to adapt the one dimensional point graph algorithm.

Still, we saw that in most of the cases, the weight of the edges will be determined by the position of the reads and not by the size difference. Then, even though we are in two dimensions, points are much more staggered over one dimension than over the other and there is still hope that our one dimensional study helps.

Best clustering for a given order Looking back at algorithm 3, we realise that it solves a more general problem than just optimal clustering for one dimensional point graphs: given any kind of graph and an order on the vertices, it will find the optimal clustering where clusters only contain consecutive vertices. Theorem 2 only ensures that an optimal clustering with the left-to-right order is a general optimal clustering.

Regarding the first heuristic algorithm, if two vertices with a very heavy edge are far from each other in the order, the algorithm may not even try to put them in the same clusters and will pay a high cost for pulling those vertices apart. However, this will only happen if the order is bad, but with good orders, the first heuristic algorithm will return a good solution too and will still be decisively faster than the exact one.

The small refinement brought by heuristic two makes no sense in our case since a positive edge between the vertex $j-i$ and the vertex j does not necessarily imply a positive edge between the vertex $j-i$ and the vertex $j-1$ and the heuristic may lead us to calculate $\mathrm{opt}'(j,i)$ without even knowing $\mathrm{opt}'(j-1,i-1)$.

In what follows, we will therefore use the first heuristic to address this problem.

Best order for clustering This leads us to studying a new problem: the optimal order for cluster editing.

Given an order on the vertices, we define its cost as the cost of the optimal clustering where clusters are made of consecutive vertices. The purpose is now

to find a fast heuristic for finding the optimal order and to combine it with the previous heuristic for best clustering for a given order.

First of all, let us see why the previous approach no longer works. Imagine we have 10 reads stemming from two different alleles (reads 1 to 5 from the first one and read 6 to 10 from the other) and that the position of the left endpoint of the reads on the reference genome happen to be close even if the reads stem from different alleles. If reads stemming from different alleles have different sizes, we should still be able to distinguish them, and this is precisely what we expect from our cluster editing algorithm.

Figure 5 illustrates this example; reads are organised on a plane according to their positions and lengths. Such a configuration of the reads is very likely to happen in practice.

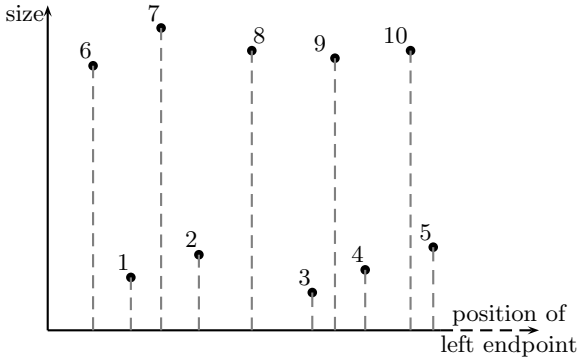

Fig. 5. Example of flaw in the left-to-right approach

Here, we should look for optimal clustering with vertices 1, 2, 3, 4 and 5 and with vertices 6, 7, 8, 9 and 10 separately . However, the left-to-right order is [6,1,7,2,8,3,9,4,10,5] and the arising clustering would clearly be sub-optimal.

A good way to solve this problem is the following: start with the left vertex, and at each step, insert in the order the unassigned vertex which is the closest from the previous one (*i.e.* the one with the heaviest edge). In the previous example, the resulting order would be $[6, 7, 8, 9, 10, 5, 4, 3, 2, 1]$. There will be n steps and at each step, the program has to investigate the $O(k)$ vertices whose associated reads overlap with the read of the previous vertex (the edges we do not investigate have a $-\infty$ weight). If all such vertices are already assigned, choose arbitrarily which vertex will follow in the order.

This algorithm is quite simple and gives significantly better results than the previous one, but it still is far from being flawless.

The reads configuration depicted in figure 6 highlights a typical problem:

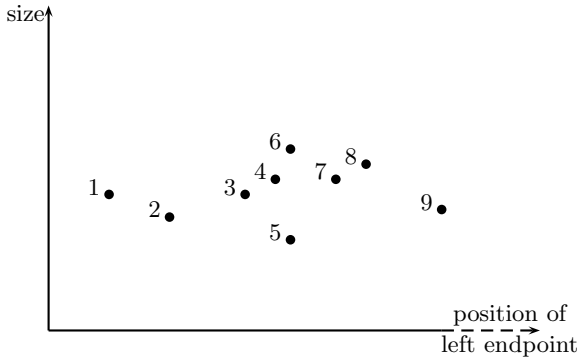

Fig. 6. Example of flaw in the closest-neighbour approach

If we apply the previous algorithm with the example of figure 6, the resulting order will be $[1, 2, 3, 4, 6, 7, 8, 9, \cdots]$ and the vertex 5 will be skipped. The algorithm will go back to vertex 5 only long after and vertex 5 will probably end up in a singleton.

A generalisation of this process is to choose $h \geqslant 1$ (3 or 5 already give good results) and to look at the unassigned vertex that maximises the sum of the weight of the edges with the h previously assigned vertices. The complexity is now $O(hkn) = O(kn)$, just like the clustering algorithm we will run on the provided order.

Now, let us see what will happen when, in the clustering, we find for the j first vertices, the vertex j ends up in a singleton. In this case, we know before choosing which vertex will be at rank $j+1$ that the vertex j and $j-1$ will not end up in the same cluster in the final clustering (our heuristic will not even investigate those solutions). Thus, it makes no sense to take into account the weight of the edges between the vertex $j-1$ and the candidates for rank $j+1$ as none of those vertices will end up in the same cluster than $j-1$.

In our final algorithm, we will run simultaneously the heuristic for best order and for best clustering with a given order: as soon as we know the vertex j, we compute $opt'(j, \cdot)$ and we will use $h = \text{argmin}(opt'(j, \cdot))$ to choose the vertex $j+1$. Our algorithm runs in $O(n \times k^2)$ which is acceptable since k can be assumed constant.

Final cluster editing algorithm We are given a set of vertices that we number from 0 to $n-1$ from left to right, and a weight function. Computing once every value of w for non overlapping reads can be done in $O(nk)$. We then can store those values in a hashtable to determine $w(a, b)$ in constant time. If the pair (a, b) has no entry in the hashtable, we know that the reads do not overlap and that $w(a, b) = -\infty$.

We then create the following objects:

- an integer array `order` of size n. Ultimately, order(i) will be the i^{th} vertex of the order we will use for cluster editing.
- a boolean array `assigned` of size n initialized with false. It will indicate whether vertex i has already been assigned in order.
- an (integer list) array `neighbours` of size n, such that neighbours(a) contains a vertex b iff their associated reads overlap. In other words, neighbours(a) is the list of vertices b such that $w(a, b) \neq -\infty$. We can also create this array and the weight function w simultaneously.
- functions `argmin` and `min`: given an integer list l and a function $f : \mathbb{N} \to \mathbb{R}$, they respectively return the element of l that minimises f and the values of the minimum.
- functions `argmax` and `max`: given an integer list l and a function $f : \mathbb{N} \to \mathbb{R}$, they respectively return the element of l that maximises f and the value of the maximum.
- a function `select`: given an integer list l and a predicate $p : \mathbb{N} \to \{\text{true}, \text{false}\}$, it returns the list of all elements of l that satisfy the predicate p.
- A function `anyvertex` which returns a vertex that has not been assigned yet. A good thing to do is to return the unassigned vertex with the lowest index and to keep memory of the last vertex the function returned so that you know where to start the investigation next time the function is called. It is correct as you will never unassign a previously assigned vertex and guarantee that no matter how many times you use anyvertex, the total computation time will always be $O(n)$.
- A matrix `opt'` of size $n \times n$ and an array `opt` of size n whose purpose will be the same as previously. We will also use an array `size` of size n to keep argmin($opt'(j, \cdot)$) in memory. Thus, size(j)+1 will indicate the size of the last cluster in the optimal clustering of the first j vertices.

Algorithm 4: Final cluster editing algorithm

1 $\text{opt}'(0,0) \leftarrow 0$
2 $\text{opt}(0) \leftarrow 0$
3 $\text{size}(0) \leftarrow 0$
4 $\text{order}(0) \leftarrow 0$
5 $\text{assigned}(0) \leftarrow \text{true}$
6 **for** j *from 1 to* $n-1$ **do**
7 $\quad X \leftarrow 0$
8 $\quad \text{opt}'(j,0) \leftarrow X + \min_{0 \leqslant i \leqslant j-1} \text{opt}'(j-1,i)$
9 $\quad \text{candidates} = \text{select}(\text{neighbours}(\textit{order}(j-1)), p : i \mapsto \textit{not}(\text{assigned}(i)))$
10 $\quad$ **if** candidates $= [\,]$ **then**
11 $\quad\quad \text{order}(j) \leftarrow \text{anyvertex}()$
12 $\quad$ **else**
13 $\quad\quad \text{order}(j) \leftarrow \text{argmax}(\text{candidates}, f : i \mapsto \sum_{k=0}^{\text{size}(j-1)} w(i, \text{order}(j-1-k))$
14 $\quad \text{assigned}(\text{order}(j)) \leftarrow \text{true}$
15 $\quad$ **for** i *from 1 to* $\text{size}(j-1)$ **do**
16 $\quad\quad X \leftarrow X - w(j, j-i)$
17 $\quad\quad \text{opt}'(j,i) \leftarrow X + \text{opt}'(j-1, i-1)$
18 $\quad \text{opt}(j) \leftarrow \min([\![0, \text{size}(j-1)]\!], f : i \mapsto \text{opt}'(j,i))$
19 $\quad \text{size}(j) \leftarrow \text{argmin}([\![0, \text{size}(j-1)]\!], f : i \mapsto \text{opt}'(j,i))$
20 $\text{clusters} \leftarrow [\,]$
21 $j \leftarrow n-1$
22 **while** $j > 0$ **do**
23 $\quad \text{clusters} \leftarrow \text{clusters} \cup \left\{ \bigcup_{k=j-\text{size}(j)}^{j} \text{order}(k) \right\}$
24 $\quad j \leftarrow j - \text{size}(j) - 1$
25 **return** clusters

As we said previously, it is possible to erase $\text{opt}'(j-1,\cdot)$ after the for loop from line 14 to 16. To avoid creating an $n \times n$ array, one can also create a $2 \times n$ array and replace all the $\text{opt}'(i,j)$ in the algorithm with $\text{opt}'(i \bmod 2, j)$.

The final algorithm runs in $O(nk)$ in time and $O(n)$ in space.

2.3 Experimental Results

Protocol To compare our new algorithm with existing state-of-the-art tools, we ran experiments based on Craig Venter's genome [10] and used UCSC's Simseq to simulate reads of mean size $\mu = 112$ and standard deviation $\sigma = 15$, in accordance to biological datasets. The reference genome was downloaded from UCSC genome browser. The main advantage of using simulated data is that we know exactly what are the insertions and deletions we expect our program

to find while real data would highlight discordance between existing tools but would not be able to determine which ones give the best results.

The test consists of searching insertions and deletions on the first chromosome of the genome. It is the largest one and the number of insertions and deletions is large enough to draw relevant conclusion about the quality of the predictions. We compare the results provided by our new tool to the one provided by the former clique enumerating version of Clever and to the other existing state-of-the-art tools: GASV [15], Variation Hunter [7], Breakdancer [5], Hydra [13], MoDIL [9], Pindel [17] and SV-seq2 [18]. Note that GASV and SV-Seq2 only predict deletions.

We compared those tools on two criteria: the precision of the predictions, *i.e.* the rate of predictions that were actual insertions and deletions, and recall, *i.e.* the rate of insertions and deletions that were predicted. We also indicate the number of exclusive predictions each tool makes *i.e.* the number of actual variations only this tool detects. Note that the proximity between the two versions of Clever will tend to lower their rates of exclusive predictions. We computed scores separately on insertions and deletions and we distinguished 5 length ranges.

Results The graph that arises from this dataset contains $n = 41,437,854$ vertices and the number of edges whose weight is not $-\infty$ is $nk = 588,570,096$. Our program was given the files containing the vertices and the edges, which are respectively 3.3GB and 15GB. An implementation in ocaml of the main algorithm (including the creation of the weight hashtable) ran in 55 minutes on a single core and peaked at 72.2GB. The interpretation of the results leading from the clustering to the predictions took less than 20 minutes. With the other tools, the computation could last up to several hundreds of hours.

Results we present in this report were obtained with a threshold of 0.4 for the weight of the edges and a false discovery rate of 0.1 when applying Benjamini-Hochberg process to the p-values file. We also used a post-process file to eliminate overlapping predictions.

Comparative results of all the tools we tested are presented in the following table.

	Insertions			Deletions		
Length range: 20-49 - 644 true insertions - 618 true deletions						
Tool	Precision	Recall	Excl.	Precision	Recall	Excl.
Clever (cluster)	86.7	50.0	0.5	65.2	63.8	0.2
Clever (cliques)	93.4	46.7	0.6	95.0	65.0	0
BreakDancer	–	7.0	0	89.7	7.4	0
GASV	-	-	-	9.3	44.7	0.2
HYDRA	0	0	0	–	0.2	0
VH	52.8	10.4	0	83.1	13.3	0
PINDEL	95.3	67.9	18.3	68.8	76.1	4.9

	Insertions			Deletions		
SV-seq2	-	-	-	–	0	0
MoDIL	62.4	61.8	5.3	73.4	80.3	2.9

	Insertions			Deletions		
Length range: 50-99 - 153 true insertions - 125 true deletions						
Tool	Precision	Recall	Excl.	Precision	Recall	Excl.
Clever (cluster)	53.8	79.1	0	55.2	77.6	0.8
Clever (cliques)	67.0	73.2	0	69.2	76.8	0
BreakDancer	93.7	57.5	0	96.8	51.2	0
GASV	-	-	-	57	43.2	0.8
HYDRA	0	0	0	–	4.8	0
VH	60.4	80.4	0	76.4	73.6	0
PINDEL	88.9	27.5	0	74.4	37.6	0
SV-seq2	-	-	-	–	0	0
MoDIL	50.9	92.2	5.9	55.3	84.8	5.6
Length range: 100-249 - 90 true insertions - 63 true deletions						
Tool	Precision	Recall	Excl.	Precision	Recall	Excl.
Clever (cluster)	80.0	27.8	0	68.8	57.1	1.6
Clever (cliques)	50.0	27.8	1.1	77.5	54.0	0
BreakDancer	49	25.6	3.3	63.7	49.2	0
GASV	-	-	-	100	42.9	1.6
HYDRA	0	0	0	78.0	42.9	0
VH	50	52.2	3.3	36.7	60.3	3.2
PINDEL	–	3.3	0	65.0	15.9	0
SV-seq2	-	-	-	–	0	0
MoDIL	40.6	55.6	7.8	58.8	34.9	3.2
Length range: 250-999 - 106 true insertions - 113 true deletions						
Tool	Precision	Recall	Excl.	Precision	Recall	Excl.
Clever (cluster)	–	0	0	92.9	69.9	0
Clever (cliques)	–	0	0	91.9	70.8	0
BreakDancer	–	0	0	71.8	68.1	0
GASV	-	-	-	0.5	61.1	0
HYDRA	–	0	0	82.1	73.5	1.8
VH	–	0	0	95.2	71.7	0.9
PINDEL	–	0	0	94.8	62.8	0
SV-seq2	-	-	-	–	0	0
MoDIL	–	0	0	–	0	0
Length range: 1000-50000 - 27 true insertions - 21 true deletions						
Tool	Precision	Recall	Excl.	Precision	Recall	Excl.
Clever (cluster)	–	0	0	77.8	66.7	0
Clever (cliques)	–	0	0	87.5	66.7	0
BreakDancer	–	0	0	76.5	61.9	0
GASV	-	-	-	66.7	66.7	0

	Insertions			Deletions		
HYDRA	–	0	0	37.2	66.7	0
VH	–	0	0	70.0	66.7	0
PINDEL	–	0	0	60.0	57.1	0
SV-seq2	-	-	-	–	0	0
MoDIL	–	0	0	–	0	0

The first thing we can notice is that although the new version of Clever uses a heuristic algorithm, the predictions are comparable to the ones the previous version provided, with even some exclusive predictions. This model also gives good results for every length range, for both insertions and deletions. Moreover, our predictions are as good as the predictions of considerably slower tools, giving hope that Clever can be used to investigate data that the other tools could not handle.

3 Conclusion

In this report, we introduced a new model based on cluster editing for next-generation sequencing. This model has many advantages, helps relax the definition of cliques and is an essential step toward algorithms which avoid overlapping predictions. However, the underlying problem is very difficult and requires large approximation to be solved on real biological data while the previous approach was based on enumerating cliques, a problem that could be solved exactly. We therefore introduced a $O(nk^2)$ time and $O(nk)$ space heuristic for cluster editing where k is a bound on the size of the clusters and can be assumed constant. The resulting tool provides good predictions for both insertions and deletions of various length and can stand up to comparison with other state-of-the-art tools. Moreover, there is surely considerable room for improvements, in the heuristic for example, and there is hope that the results of this method improve faster and ultimately get decisively better than the results provided by the clique enumerating method which is already the results of years of work and use exact algorithms.

In addition, besides the fact that it helped validate the clustering model, the $O(nk^2)$ heuristic for cluster editing can be seen as a contribution in itself, as the problems it solves have many applications in various fields.

Finally, the fact that our algorithm runs in $O(nk^2)$ makes it possible to deal with high coverage data, which will be more and more common in the coming years.

Acknowledgements

The authors thank COST action BM1006 SeqAhead for supporting this work and Murray Patterson for his advice on an early version of this manuscript.

References

1. Niko Beerenwinkel and Osvaldo Zagordi. Ultra-deep sequencing for the analysis of viral populations. *Current Opinion in Virology*, 1(5):413 – 418, 2011.
2. Amir Ben-Dor, Ron Shamir, and Zohar Yakhini. Clustering gene expression patterns. *Journal of Computational Biology*, 6(3/4):281–297, 1999.
3. Yoav Benjamini and Yosef Hochberg. Controlling the false discovery rate: A practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society Series B (Methodological)*, 57(1):289–300, 1995.
4. Moses Charikar, Venkatesan Guruswami, and Anthony Wirth. Clustering with qualitative information. *J. Comput. Syst. Sci.*, 71(3):360–383, 2005.
5. Ken Chen, John W Wallis, Michael D McLellan, David E Larson, Joelle M Kalicki, Craig S Pohl, Sean D McGrath, Michael C Wendl, Qunyuan Zhang, and Devin P Locke. Breakdancer: an algorithm for high-resolution mapping of genomic structural variation, 2009.
6. Erez Hartuv, Armin O. Schmitt, Jörg Lange, Sebastian Meier-Ewert, Hans Lehrach, and Ron Shamir. An algorithm for clustering cDNAs for gene expression analysis. In *RECOMB*, pages 188–197, 1999.
7. Fereydoun Hormozdiari, Can Alkan, Evan E. Eichler, and Süleyman Cenk Sahinalp. Combinatorial algorithms for structural variation detection in high throughput sequenced genomes. In *RECOMB*, pages 218–219, 2009.
8. Jan Kratochvíl and Mirko Krivánek. On the computational complexity of codes in graphs. In *MFCS*, pages 396–404, 1988.
9. Seunghak Lee, Fereydoun Hormozdiari, Can Alkan, and Michael Brudno. Modil: detecting small indels from clone-end sequencing with mixtures of distributions, 2009.
10. Samuel Levy, Granger Sutton, Pauline C Ng, Lars Feuk, Aaron L Halpern, Brian P Walenz, Nelson Axelrod, Jiaqi Huang, Ewen F Kirkness, Gennady Denisov, Yuan Lin, Jeffrey R MacDonald, Andy Wing Chun Pang, Mary Shago, Timothy B Stockwell, Alexia Tsiamouri, Vineet Bafna, Vikas Bansal, Saul A Kravitz, Dana A Busam, Karen Y Beeson, Tina C McIntosh, Karin A Remington, Josep F Abril, John Gill, Jon Borman, Yu-Hui Rogers, Marvin E Frazier, Stephen W Scherer, Robert L Strausberg, and J. Craig Venter. The diploid genome sequence of an individual human. *PLoS Biol*, 5(10):e254, 09 2007.
11. Bassel Mannaa. Cluster editing problem for points on the real line: A polynomial time algorithm. *Inf. Process. Lett.*, 110(21):961–965, 2010.
12. Tobias Marschall, Ivan G. Costa, Stefan Canzar, Markus Bauer, Gunnar W. Klau, Alexander Schliep, and Alexander Schönhuth. Clever: clique-enumerating variant finder. *Bioinformatics*, 28(22):2875–2882, 2012.
13. Aaron R. Quinlan, Royden A. Clark, Svetlana Sokolova, Mitchell L. Leibowitz, Yujun Zhang, Matthew E. Hurles, Joshua C. Mell, and Ira M. Hall. Genome-wide mapping and assembly of structural variant breakpoints in the mouse genome. *Genome research*, 20(5):623–635, May 2010.
14. Roded Sharan, Adi Maron-Katz, and Ron Shamir. Click and expander: a system for clustering and visualizing gene expression data. *Bioinformatics*, 19(14):1787–1799, 2003.
15. Suzanne S. Sindi, Elena Helman, Ali Bashir, and Benjamin J. Raphael. A geometric approach for classification and comparison of structural variants. *Bioinformatics*, 25(12), 2009.

16. Anke van Zuylen and David P. Williamson. Deterministic algorithms for rank aggregation and other ranking and clustering problems. In *WAOA*, pages 260–273, 2007.
17. Kai Ye, Marcel H. Schulz, Quan Long, Rolf Apweiler, and Zemin Ning. Pindel: a pattern growth approach to detect break points of large deletions and medium sized insertions from paired-end short reads, 2009.
18. Jin Zhang, Jiayin Wang, and Yufeng Wu. An improved approach for accurate and efficient calling of structural variations with low-coverage sequence data. *BMC Bioinformatics*, 13(S-6):S6, 2012.