ISSN 1995-0802, Lobachevskii Journal of Mathematics, 2018, Vol. 39, No. 9, pp. 1217–1227. © Pleiades Publishing, Ltd., 2018.

Linear Algorithm for a Cyclic Graph Transformation

V. A. Lyubetsky[1,2*], E. Lyubetskaya[1], and K. Gorbunov[1***]**

(Submitted by A. V. Lapin)

[1] Institute for Information Transmission Problems of the Russian Academy of Sciences (Kharkevich Institute), Bol'shoi Karetnyi per. 19, str. 1, Moscow, 127051 Russia

[2] Faculty of Mechanics and Mathematics, Lomonosov Moscow State University, Moscow, 119991 Russia

Received June 4, 2018

Abstract—We propose a linear time and linear space algorithm that constructs a minimal (in the total cost) sequence of operations required to transform a directed graph consisting of disjoint cycles into any graph of the same type. The following operations are allowed: double-cut-and-join of vertices and insertion or deletion of a connected fragment of edges; the latter two operations have the same cost. We present a complete proof that the algorithm finds the corresponding minimum. The problem is a nontrivial particular case of the *general* problem of transforming a graph into another, which in turn is an instance of a hard optimization problem in graphs. In this general problem, which is known to be NP-complete, each vertex of a graph is of degree 1 or 2, edges with the same name may repeat unlimitedly, and operations belong to a well-known list including the above-mentioned operations. We describe our results for the general problem, but the proof is given for the cyclic case only.

DOI: 10.1134/S1995080218090147

Keywords and phrases: *graph, cycle, graph rearrangement, operation cost, combinatorial problem, optimization in graphs, linear algorithm.*

1. INTRODUCTION

In the general case, we consider the *graph transformation problem*: we are given directed graphs, a and b, with vertex degrees 1 or 2, each edge having an assigned name (a natural number); well-known operations over graphs, which retain this condition on the vertex degrees; and a cost (or weight) of each operation. The costs are strictly positive rational numbers. Such graphs are referred to as *structures*. It is required to find a *shortest* sequence of operations transforming a into b, i.e., a sequence with a minimum overall cost, which will be referred to as the *minimum cost*. The above-mentioned operations over such graphs are said to be *standard*, and there are also operations of *insertion* and *deletion* of a connected sequence of edges (called *fragment*), referred to as *additional*. A detailed description of standard and additional operations is given, e.g., in [1, Fig. 1].

We obtain linear time and linear space algorithms for solving the problem in the following cases: (1) All operations have equal costs [2]; (2) Cost of any operation other than insertion is equal of the same value w, and the cost of the latter lies in the segment from w to $2w$, [1]; (3) Costs of insertions and deletions are the same and are not greater than equal costs w of other operations [3]. We also obtain a precise algorithm for optimal structure reconstruction (expansion) from leaves of a tree to its internal vertices [4]; "precise" means that the algorithm outputs an optimal decision. Solutions have two versions: a direct precise algorithm and reduction to linear programming [5, 6]; "direct" means that no reduction to another problem is used.

[*] E-mail: lyubetsk@iitp.ru
[**] E-mail: liin@math-study.ru
[***] E-mail: gorbunov@iitp.ru

In this paper we consider a particular case of the problem, which will be referred to as *cyclic*: a graph consists of disjoint cycles, including loops, names in a graph do not repeat, and costs are always assumed to satisfy condition (3) *without any constraint on w*, which is arbitrary now. We refer to this condition as (4). Such graphs are called *cyclic* structures; in arbitrary structures, besides cycles, chains are also allowed.

For cyclic structures, standard and additional operations reduce to *double-cut-and-join* (DCJ), *deletion*, and *insertion*. The first of them consists in cutting two vertices of a graph and joining the four thus obtained extremities in a new way. The second is deleting a connected fragment of edges with names belonging to the initial structure but not belonging to the final one; the third is inserting a connected fragment of edges with names belonging to the final structure but not belonging to the initial one. These constraints on the operations do not lose the generality of a solution.

Both the original problem and its cyclic case have a long history and find various applications; see, e.g., references in [5–10]. Originally, the problem was posed in bioinformatics as a problem of finding the most low-cost transformation of one chromosome structure into another. In the corresponding terminology, edges of a structure are referred to as *genes*, and its connected components, as *chromosomes* (linear or cyclic). Application of various developed algorithms is also mainly related with needs of bioinformatics. At present, a considerable amount of data on chromosome structures of genomes has been collected. This made it possible to propose the above-described model of a chromosome structure, where genes are contained in cyclic and linear chromosomes and each gene belongs to one of two DNA chains. Specifying a chain is equivalent to specifying a direction on edges (genes) of our graph. Sequencing results in exponential growth of the number of available genomes, which makes indispensable parallel processing of, in particular, their chromosome structures using a supercomputer. Our linear time and linear space algorithms enable fast processing of structures with a large number of genes, up to many thousands.

Numerous studies on this subject were also performed within the bioinformatics framework. For unequal gene content, the graph transformation problem (also called the reduction problem) was stated in [11, 12]. It was widely studied in the context of applied issues, where its setting assumed considerable restrictions on a graph and operations, and heuristic algorithms for its solution were proposed; for detailed references, see, e.g., [11, 12]. As far as the authors are concerned, mathematical results related to this problem in the above setting are contained in [13, 14] only. In [13], a solution to this problem was proposed for a particular case of equal operation costs. In conference theses [14], a solution plan was proposed for unequal costs in the case where all chromosomes are cyclic. To the best of our knowledge, the corresponding proof has never been published; perhaps, the proposed plan could not be realized; the authors have obtained another algorithm and a proof of its correctness following another way, see Theorem 1 below. The algorithms in [13, 14] are essentially other in kind than ours.

Other publications analyze particular cases of the reduction problem. For instance, [15] considered genomes consisting of a single linear chromosome. Only the inversion operation was allowed, i.e., reversal of a chromosome fragment at the same place. An algorithm was described that constructs a shortest sequence of inversions transforming one chromosome to another, these chromosomes having equal gene content, i.e., the same set of gene names. The algorithm has polynomial runtime of $O(n^4)$. In [16], a linear time algorithm was proposed, which computed the minimum number of operations (but not the operations themselves) in the reduction problem for two structures with equal gene content and consisting of linear chromosomes. Only the translocation operation was allowed, which consists in exchanging heads of two chromosomes. In [17] this result was extended to the case where not only translocations but also inversions, fusions, and fissions are allowed. In [18] a linear time algorithm was described that solved the reduction problem for general structures with equal gene content and equal operation costs. Here the set of operations coincides with ours (except for insertions and deletions). In [19], an original construction of a breakpoint graph was proposed; we generalized it to the case of unequal gene content. Note that in [2] we refer to the obtained graph as *common* and denote it by $a + b$; here we will use the standard term "breakpoint graph" (its definition is given below). In [19], a method was proposed to reduce the graph transformation problem to the problem of finalizing the breakpoint graph; the method is analogous to the reduction executed by our algorithm for the graph $a + b$ [2].

2. DEFINITIONS NEEDED TO DESCRIBE OUR ALGORITHM

For two arbitrary structures a and b, their *breakpoint graph*, denoted by $a+b$, is an undirected graph consisting of connected *components*, which are also chains (including isolated vertices) and cycles, including loops. A detailed definition of $a+b$ is given in [1] immediately after Fig. 1, but we will give an essential description of $a+b$ in the next paragraph. In what follows, we refer to edges in graphs a and b and in intermediate graphs between a and b as *arcs*, and for edges in $a+b$ we keep the standard term. An arc is said to be *ordinary* if it is represented in both a and b; otherwise, it is said to be *special*.

Vertices and edges in $a+b$ can also be *ordinary* or *special*: an ordinary vertex is an extremity of an arc represented in a and b; an ordinary edge connects two ordinary vertices if they are joined in a or in b; a special vertex is a connected maximal fragment consisting of special arcs in a or in b (referred to as a *block*; the vertex is labeled by a sequence of these arcs); a special edge connects such a fragment with an adjacent extremity of an ordinary arc, i.e., connects a special vertex with an ordinary one. A *loop* in $a+b$ corresponds to a cycle of special arcs, i.e., connects a special vertex with itself.

Any edge or any special vertex has a source in a or b and is accordingly called an a- or b-edge, or an a- or b-vertex, or an a- or b-loop. A *pendant* edge is an edge incident to a special vertex of degree 1; such an edge is located at an end of a chain. Nonpendant special edges occur in $a+b$ in pairs: these are edges incident to a common special vertex. The *size* of a component in $a+b$ or of $a+b$ itself is the sum of the number of its ordinary edges and half the number of its special nonpendant edges. For isolated ordinary vertices and loops, the size is assumed to be 0; for isolated special vertices (not loops), it is assumed to be -1. A breakpoint graph is said to be final (*finalized*, in *final form*) if each of its components is either an isolated ordinary vertex or a cycle without special vertices of size (or, which is the same in this case, of length) 2, one of its edges being an a-edge and the other a b-edge; such cycles will be called *final*. Cycles of size 2 will be referred to as 2-*cycles*. In the cyclic case, a breakpoint graph consists of cycles and loops only; it has no pendant edges and no isolated vertices. Our algorithm starts its operation with forming a breakpoint graph $a+b$ with all labels mentioned above; clearly, this requires linear time and space.

In [1], five operations over a breakpoint graph were defined: analogs of standard operations, which are also referred to as *standard*, and one *additional* operation. The latter is called a *deletion* and consists in deleting a special vertex: when deleting it, we join two special edges incident to it into one ordinary edge with the same label. In the cyclic case, the standard operations are only one, an analog of the DCJ: delete two identically labeled edges (both a- or both b-edges) in $a+b$ and join the four obtained endpoints by two new edges with the same label. If an edge with special endpoints (both a- or both b-vertices) is formed, it is replaced with one special vertex with the joined label. A DCJ operation preserves the size of each component in $a+b$ and of $a+b$ itself. Indeed, if no join of special vertices occurs, then the numbers of special and ordinary edges do not change; otherwise, the former reduces by 2, and the latter increases by 1.

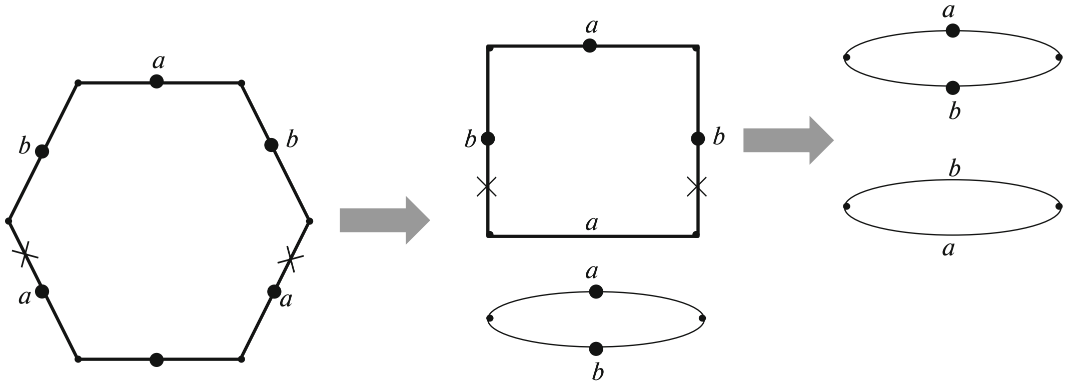

Fig. 1. Application of the 4th step to an (a,b)-cycle of size 6. Special vertices are shown by bold dots; each such vertex is incident to two special edges. Label a or b indicates the structure from which this vertex and these edges are taken. The first arrow corresponds to the first part of this step, the second arrow, to the second. Both DCJ-operations are special.

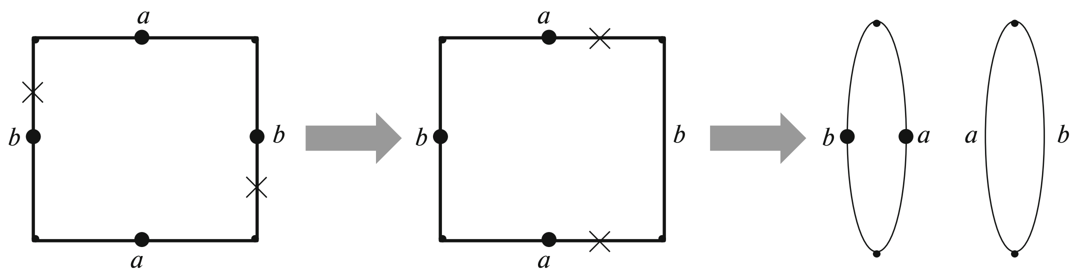

Fig. 2. Application of the 5th step to an (a,b)-cycle of size 4. Arrows correspond to two parts of the step; in both parts, special DCJ-operations are applied.

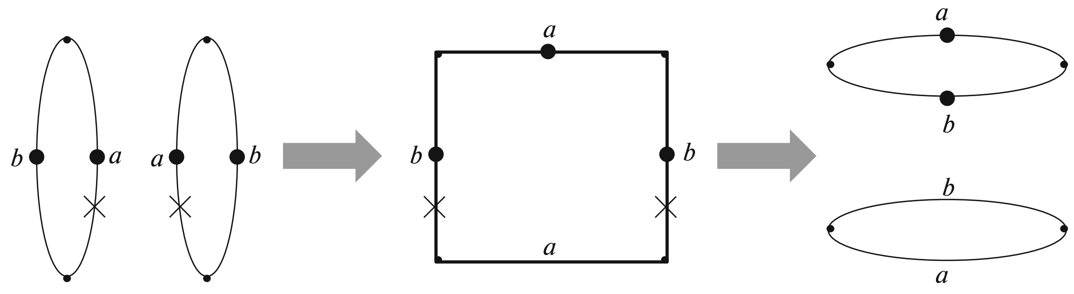

Fig. 3. Application of the 6th step to two (a,b)-cycles of size 2. Arrows correspond to two parts of the step; in both parts, special DCJ-operations are applied.

Cutting out an ordinary edge (say with label a) from a cycle of size strictly greater than two consists in the following: replace two b-edges adjacent to the a-edge with two other b-edges: one of them together with the a-edge forms a final 2-cycle, and the other joins two "far" endpoints of the replaced edges. Clearly, this is an analog of a DCJ; for more details, see [2], Section 3.2. Analogously, *cutting* out a pair of special incident edges (say, aa) from a cycle of size strictly greater than 2 means the following: take two identically labeled b-edges nearest to aa and replace them with two other b-edges so that the cycle splits into two cycles, one of them being a 2-cycle, [2, Fig. 2a]. Let us have, for instance, a fragment $baab$ in a cycle; applying the DCJ-operation, we delete left and right b-neighbors of the special edges aa and replace them with two edges, resulting in two smaller cycles, one of them being aab.

In [2] it is proved (corollary of Theorem 1) that if all operations have equal costs, then transforming a into b using six operations over structures is equivalent, with the same total cost, to finalizing $a+b$ using the five operations given above, where operation costs are the same as those over structures: deletion cost for an a-special vertex is equal to the cost of deleting a block from current a, and deletion cost for a b-special vertex is equal to the cost of inserting a block into current b. The proof remains literally the same if costs of all standard operations are only equal but costs of deletion and insertion are arbitrary. Therefore, we will consider finalizing the breakpoint graph $a+b$ instead of reducing a to b. If a transformation of $a+b$ to any final form (finalization) is obtained, then an obvious linear algorithm transforms it into a solution of the original transformation problem.

Let us explain the linearity of time required to construct $a+b$. Each component $k_1,\ldots,k_n$ (cycle, loop, or chain) of the original structures is stored as its list; names of oppositely directed arcs are recorded with the minus sign. Looking through the structures a and b, we obtain a list $(*)$ of indices, i.e., names used for a and b together with marks indicating whether the name belongs to a, b, or both structures. Looking through these lists, we create a list of blocks; in a separate list, we record the initial and final arcs of each block. The list $(*)$ trivially specifies arcs as ordinary, a-special, or b-special. For each arc in a we indicate which arc is immediately before and after it in a; the same for b. This allows to perform a DCJ-operation in a prescribed fixed time and a deletion operation in linear time, since the number of deleted arcs is not greater than the total number of arcs. The breakpoint graph $a+b$ is stored in a sequence M_i, where absolute values of indices are indices of arcs in a and b; negative indices indicate heads of arcs and positive, their tails. The value of M_i is a pair of arc extremities in a and b with which the extremity of i is joined in a and independently in b (or 0 if it is not joined). This ensures linear time and space when constructing $a + b$ from structures a and b, fast run-through of its components (without scanning the blocks), and performing any operation over the breakpoint graph or transition between operations over the structures and the breakpoint graph. Since indices of blocks are also used as indices or list values only and are natural numbers, we conclude that in the Random Access Machines model under the uniform weight criterion the runtime (and hence, the size of the memory used) of the algorithms below depends linearly on the total number of arcs in given structures.

3. THEOREM AND DESCRIPTION OF THE ALGORITHM FOR FINALIZING A BREAKPOINT GRAPH

Throughout what follows, we consider the *cyclic case* (4): cyclic graphs a and b, an arbitrary fixed cost w of insertion and deletion, assuming the cost of a DCJ operation to be equal to 1. In this case the breakpoint graph $G = a + b$ consists of cycles and loops only. The sign □ marks the end of each essential part of the proof. The description of the algorithm is given below, inside the proof, between the ► and ■ signs.

Theorem 1. *In the cyclic case, the algorithm given below constructs a shortest sequence of operations (with the minimum total cost). Its runtime and required space depend linearly on the aggregate size of any two input structures.*

Proof. Consider three conditions on w.

First, **let $0 < w \leq 1$**. The description of the algorithm is as follows.

► For the breakpoint graph $G = a + b$ the algorithm independently deletes all loops, then from any cycle of size greater than 2 it cuts out all ordinary edges, and then cuts out all special edges until only 2-cycles remain. In all 2-cycles it deletes all special vertices, if any, and thus obtains a set of final cycles; see [2, Section 2.2]. ■

The described actions require linear time.

Denote by $T(G)$ the total cost of operations in a sequence *constructed by our algorithm* (it is called the *finalizing cost*). For G we use the following *notation*: B is the total number of special vertices in G, and S is the sum of integral parts of halved lengths of maximum connected fragments in G consisting of ordinary edges (which will be referred to as *segments*) minus the number of cyclic segments. The *quality* of G is the number of cycles in it (loops are not counted).

Lemma 1. *Let d be the aggregate size of components of a breakpoint graph G, and let c be its quality. Then*

$$T(G) = (1 - w)(0.5d - c) + w(B + S). \tag{1}$$

Proof. For G, the number of operations in a finalizing sequence output by our algorithm is $P(G) = B + S_0$, where B is the number of usages of deletions and DCJ operations that reduce the number of special vertices, and S_0 is the number of usages of DCJ operations that do not reduce this number. The latter usages occur only when cutting out an ordinary edge if one of its neighboring edges is ordinary. This implies that $S_0 = S$, which is proved in detail in [2].

Denoting by U and Q, respectively, the number of deletions and the number of DCJ operations in a finalizing sequence, we obtain $U + Q = P(G) = B + S$. Hence, $T(G) = Q + w(B + S - Q) = (1 - w)Q + w(B + S)$.

When finalizing a cycle K of size d_0, deletion and DCJ operations preserve the size of any breakpoint graph G, and the final form of the cycle consists of $0.5 \cdot d_0$ final cycles. The DCJ operation increases the quality of G by 1, and the deletion operation does not change it. Thus we obtain $Q = 0.5d_0 - 1$. Summing over all cycles, we obtain $Q = 0.5d - c$. □

Let $c(G)$ be the *minimum total cost* of all sequences finalizing a breakpoint graph G.

Let us show that $T(G) = c(G)$ for any breakpoint graph G; it suffices to check that

$$T(G) \leq c(G). \tag{2}$$

Consider the shortest sequence that finalizes G. In it, let o be the first operation with cost $c(o)$ applied to G, and let $o(G)$ be the result of its application. Then $c(o(G)) < c(G)$, and we may proceed by induction on $c(G)$.

Below we will check the key inequality for any operation o and any G:

$$c(o) \geq T(G) - T(o(G)). \tag{3}$$

Then inequality (3) implies (2). Indeed, by the induction hypothesis we have $T(o(G)) \leq c(o(G))$. Using (3), we obtain $T(G) \leq T(o(G)) + c(o) \leq c(o(G)) + c(o) = c(G)$, i.e., (2).

Now we pass to checking inequality (3); the check essentially exploits equality (1). We use the fact that in the cyclic case any operation over a breakpoint graph changes $B + S$ by at most 1. This is implied by [2, Theorem 5] or can easily be checked directly, as follows.

We say that a segment is *adjacent* to an edge a of a breakpoint graph if it contains a or an edge neighboring with a; if a has no adjacent segment, we assume that an empty segment is adjacent to a. If a DCJ operation reduces B by 1, then two segments adjacent to the deleted edges are joined into one of length greater by 1 than the sum of lengths of the original segments; if the new segment is cyclic, it has an even length. The value of S does not reduce. If a DCJ operation does not change B, such two segments are transformed into two new segments, their aggregate length being equal to the sum of lengths of the original segments. The value of S can change by at most 1. Similarly, for the deletion of a special vertex we have the following: S either does not change (if both segments adjacent to the deleted vertex are even or if the deleted vertex is the only special vertex in a cycle) or increases by 1. Consider all possible operations o in (3).

1. Deleting a loop: $T(G)$ reduces by w.

2. Deleting a special vertex in a cycle: $T(G)$ either is unchanged or reduces by w.

3. Inversion inside a cycle: $T(G)$ is changed by at most w.

4. Splitting a cycle into two cycles or into a cycle and a loop: $T(G)$ changes by $(w-1) \pm w$ or $\pm w$, which is not greater than 1 in absolute value.

5. Merging two cycles, a cycle and a loop, or two loops: $T(G)$ changes by $(1-w) \pm w$ or $\pm w$. □

Now **let $1 < w \leq 2$**. By an *a-cycle* we call a cycle in $G = a + b$ that contains special a-vertices but no b-vertices; a *b-cycle* is defined similarly; an *(a, b)-cycle* contains both a- and b-special vertices. By a *special* operation, considered together with its arguments, we call an operation that results in joining two special vertices into one.

► Our algorithm consists in seven steps.

1) Cut out all ordinary edges.

2) While possible, apply the DCJ-operation to special edges of a loop and of another component. (At most two loops can remain: one a- and one b-loop.)

3) Delete the remaining loops.

4) While there is an (a, b)-cycle of size strictly greater than 4, apply the DCJ-operation to special edges in this cycle separated by four edges so as to form an (a, b)-cycle of size 2, and then cut out the ordinary edge thus formed; Fig. 1.

5) While there is an (a, b)-cycle of size 4, apply the DCJ-operation (inversion) to special edges in this cycle separated by three edges so as to preserve an (a, b)-cycle of size 4, and then cut out the ordinary edge thus formed; Fig. 2.

6) While there are two or more (a, b)-cycles, apply the DCJ-operation to special edges from these different cycles so as to form an (a, b)-cycle of size 4, and then cut out the ordinary edge thus formed, Fig. 3. By the beginning of this step, there are only cycles of size 2 remaining. At this step, if we had at least one (a, b)-cycle, we will have exactly one (a, b)-cycle of size 2.

7) Delete the remaining special vertices. The obtained graph is of a final form. At each step of the algorithm, the first pair of edges, the first loop, and the first vertex in a fixed traversal over all components in $a + b$ should be taken. ■

Linear runtime of the algorithm is obvious.

Let us prove correctness of the algorithm. Let C_a and C_b be the number of a- and b-cycles in $a + b$, respectively. The number of deletions of special vertices in the algorithm equals $C_a + C_b + 2I_{ab} + I_{pa} + I_{pb}$, where $I_{ab} = 1$ if $a + b$ contains an (a, b)-cycle (and 0 otherwise), $I_{pa} = 1$ if there is an a-loop and no cycle with a special a-vertex (and 0 otherwise), and I_{pb} is a similar indicator for a b-loop. Operations, except for those performed at Step 1, are special. The number of special operations in the algorithm is B, the number of nonspecial ones is S. Then

$$\begin{aligned}T(G) &= w(C_a + C_b + 2I_{ab} + I_{pa} + I_{pb}) + (B + S - C_a - C_b - 2I_{ab} - I_{pa} - I_{pb}) \\ &= (w-1)(C_a + C_b + 2I_{ab} + I_{pa} + I_{pb}) + B + S.\end{aligned}$$

For the second condition on w, the proof of the algorithm correctness follows the same scheme, i.e., we examine all possible operations o applied to G. Below, in items 1.1—1.5, 2.2.2—2.2.6, and 2.3.2, we mention only one of two symmetric (with respect to a and b) cases. As above, we only mention the quantities that can change when applying an operation o.

1. o is deletion of a special vertex. Consider the cases.

1.1. Deletion of an a-loop. Upon applying o, B reduces by 1, and I_{pa} either does not change or reduces by 1. Thus, $T(G)$ reduces by either 1 or w.

1.2. Deletion of a unique special a-vertex in an a-cycle. B and C_a reduce by 1, and I_{pa} either does not change or increases by 1. Thus, $T(G)$ reduces by either w or 1.

1.3. Deletion of a nonunique special a-vertex in an a-cycle. Only B and S can change, and the sum $B + S$ changes by at most 1.

1.4. Deletion of a unique special a-vertex in an (a, b)-cycle. B reduces by 1, and S does not change, since both segments adjacent to the deleted vertex are even. C_b increases by 1, I_{ab} does not change or reduces by 1, and I_{pa} does not change or increases by 1. Therefore, possible increments for $T(G)$ are the following: $w - 2$, $2w - 3$, -1, or $-w$. They are not greater than w in absolute value.

1.5. Deletion of a nonunique special a-vertex in an (a, b)-cycle. Only B and S can change.

2. o is a DCJ. Consider the cases.

2.1. Inversion inside a cycle. Only B and S can change.

2.2. Splitting a cycle into two cycles or into a cycle and a loop. Consider the cases.

2.2.1. Splitting an (a, b)-cycle into two (a, b)-cycles, or into an (a, b)-cycle and a loop, or into an (a, b)-cycle and a cycle without special vertices. Only B and S can change.

2.2.2. Splitting an (a, b)-cycle into an (a, b)-cycle and an a-cycle or splitting an a-cycle into two a-cycles. C_a increases by 1. Let us show that the sum $B + S$ cannot increase by 1. If B reduces by 1, then, since d is unchanged under a DCJ, it follows that the aggregate length of the two original segments adjacent to the cuts is less by 1 than that of the two new segments (note that in this case one of the new segments is empty). Then S cannot increase by more than 1. If B is unchanged, then, since d is preserved, it follows that the aggregate length of the two original segments adjacent to the cuts is equal to the aggregate length of the two new segments. S increases by 1 if the original segments are odd and the two new are even. But in this case both original segments lie between a-vertices. Hence, the new segments are lie between special a-vertices, i.e., are odd; a contradiction. Thus, in all the cases, possible increments for $T(G)$ are $w - 1$ or $w - 2$, which are not greater than 1 in absolute value.

2.2.3. Splitting an (a, b)-cycle into an a-cycle and a b-cycle. C_a and C_b increase by 1, and I_{ab} either does not change or reduces by 1. The sum $B + S$ can only reduce by 1. Indeed, both original segments adjacent to the cuts are even, since they lie between a special a-vertex and a special b-vertex. If B reduces by 1, then a new segment is odd and S does not change. If B does not change, then both new segments are odd and S reduces by 1. In all the cases, possible increments for $T(G)$ are -1 or $2w - 3$, which are not greater than 1 in absolute value.

2.2.4. Splitting an (a, b)-cycle into an a-loop and a b-cycle. C_b increases by 1, and I_{ab} either does not change or reduces by 1; in the last case, I_{pa} can increase by 1. B and S do not change, since a pair of incident special a-edges in the cycle is replaced with one ordinary edge, and two even segments are joined into an odd one of length greater by 1 than the sum of their lengths. In all the cases, $T(G)$ changes by at most 1.

2.2.5. Splitting an a-cycle into an a-cycle and an a-loop, or into an a-cycle and a cycle without special vertices, or splitting a cycle without special vertices. Only B and S can change.

2.2.6. Splitting an a-cycle into an a-loop and a cycle without special vertices. C_a reduces by 1, I_{pa} either does not change or increases by 1, and S does not change. In all the cases, $T(G)$ changes by at most 1.

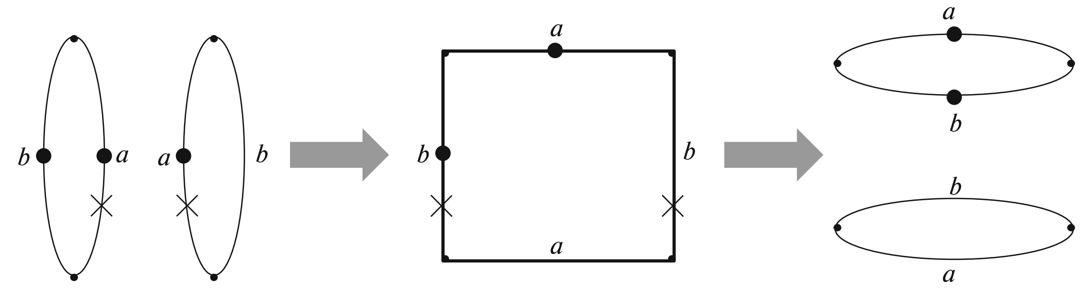

Fig. 4. Application of the 7th step to an (a, b)-cycle and an a-cycle of size 2. Arrows correspond to two parts of this step; in the first part, a special DCJ-operation is applied, in the second part, a nonspecial one.

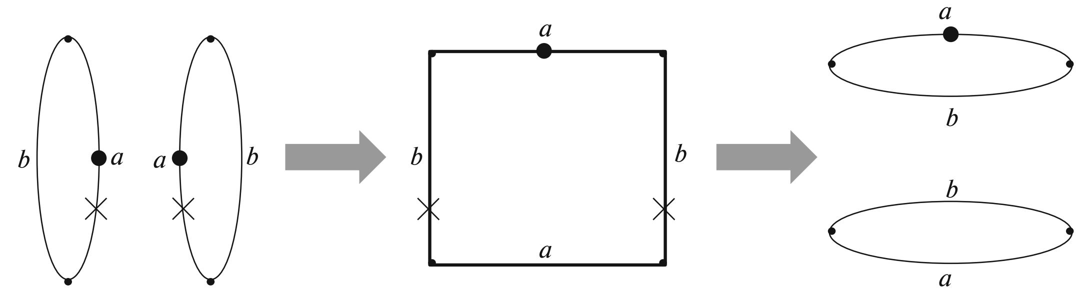

Fig. 5. Application of the 8th step to two a-cycles of size 2. Arrows correspond to two parts of this step; in the first part, a special DCJ-operation is applied, in the second part, a nonspecial one.

2.3. Merging two cycles into one. Consider the cases.

2.3.1. Merging two (a, b)-cycles or two cycles one of which has no special vertices. Only B and S can change.

2.3.2. Merging an (a, b)-cycle and an a-cycle or merging two a-cycles. C_a reduces by 1, and $B + S$ cannot reduce by 1. Indeed, if B does not change, then S cannot reduce by 1, since the original segment adjacent to the cut in the a-cycle is odd. If B reduces by 1, then S increases by 1 by the same reason. Then $T(G)$ changes by $1 - w$ or $2 - w$, which are not greater than 1 in absolute value.

2.3.3. Merging an a-cycle and a b-cycle. Since no joining of two special vertices into one occurs, this operation is inverse to the one considered in item 2.2.3.

2.4. Merging a cycle and a loop into a cycle. If no joining of two special vertices into one occurs, then this operation is inverse to the one considered in item 2.2. Otherwise, it is equivalent to deletion of a loop. □

Now **let $\mathbf{w > 2}$**.

► The first six steps of the above algorithm are in this case continued by the following three.

7) While there are an (a, b)-cycle and an a-cycle, apply the DCJ-operation to special edges from these different cycles so as to form an (a, b)-cycle of size 4, and then cut out any of the two ordinary edges thus formed, Fig. 4 (the latter operation is not special, and therefore it increases by 1 the number $B + S$ of operations in the algorithms for the two above-considered conditions on w). Perform the same for b-cycles.

8) While there are two or more a-cycles, apply the DCJ-operation to special edges from these different cycles so as to form an (a, b)-cycle of size 4, and then cut out any of the three ordinary edges thus formed, Fig. 5 (the latter operation is not special). Perform the same for b-cycles.

9) Delete the remaining special vertices. The obtained graph is of a final form. ■

Let us prove correctness of the algorithm. The total number of operations in it is $B + S + C_a - I_{ca}(1 - I_{ab}) + C_b - I_{cb}(1 - I_{ab}) = B + S + C_a + C_b - (I_{ca} + I_{cb})(1 - I_{ab})$; here $I_{ca} = 1$ (respectively, $I_{cb} = 1$) if $a + b$ contains an a-cycle (respectively, a b-cycle) and 0 otherwise. Among them, there are

$I_a + I_b$ deletions; here $I_a = 1$ (respectively, $I_b = 1$) if $a + b$ has a special a-vertex (respectively, a b-vertex) and 0 otherwise. Then

$$\begin{aligned} T(G) &= w(I_a + I_b) + (B + S + C_a + C_b - (I_{ca} + I_{cb})(1 - I_{ab}) - I_a - I_b) \\ &= (w - 1)(I_a + I_b) + B + S + C_a + C_b - (I_{ca} + I_{cb})(1 - I_{ab}). \end{aligned}$$

For the third condition on w, the proof of the algorithm correctness follows the same scheme as above.

1. Deleting a special vertex. Consider the cases.

1.1. Deleting an a-loop. B reduces by 1, and I_a either does not change or reduces by 1. Thus, $T(G)$ reduces by either 1 or w.

1.2. Deleting a unique special a-vertex in an a-cycle. B and C_a reduce by 1, and I_a either does not change or reduces by 1, but the latter reduction is partly compensated by simultaneous reduction of I_{ca} by 1 (taking into account that $I_{ab} = 0$ in this case). Thus, $T(G)$ reduces by 1, 2, or w.

1.3. The same as in the preceding proof.

1.4. Deleting a unique special a-vertex in an (a, b)-cycle. B reduces by 1, S does not change, C_b increases by 1, and I_a either does not change or reduces by 1; in the latter case I_{ab} also reduces by 1 (and $I_{ca} = 0$ does not change); I_{cb} either does not change or increases by 1. The quantity $V = (I_{ca} + I_{cb})(1 - I_{ab})$ either does not change or increases by 1 or 2, but if I_a reduces, then V increases by 1. Thus, $T(G)$ either does not change or reduces by 1, 2, or w.

1.5. The same as in the preceding proof.

2. DCJ. Consider the cases.

2.1. The same as in the preceding proof.

2.2. Splitting a cycle into two cycles or into a cycle and a loop. Consider the cases.

2.2.1. The same as in the preceding proof.

2.2.2. Splitting an (a, b)-cycle into an (a, b)-cycle and an a-cycle or splitting an a-cycle into two a-cycles. C_a increases by 1, I_{ca} either does not change or increases by 1, and $B + S$, as was shown in the preceding case, either does not change or reduces by 1. In all the cases, $T(G)$ changes by at most 1.

2.2.3. Splitting an (a, b)-cycle into an a-cycle and a b-cycle. C_a and C_b increase by 1, I_{ca} and I_{cb} either do not change or increase by 1, and I_{ab} either does not change or reduces by 1. The sum $B + S$, as was shown above, reduces by 1. In all the cases, $T(G)$ changes by at most 1.

2.2.4. Splitting an (a, b)-cycle into an a-loop and a b-cycle. C_b increases by 1, and I_{ab} either does not change or reduces by 1, B and S do not change. The quantity $V = (I_{ca} + I_{cb})(1 - I_{ab})$ either does not change or increases by 1 or 2. In all the cases, $T(G)$ changes by at most 1.

2.2.5. The same as in the preceding proof.

2.2.6. Splitting an a-cycle into an a-loop and a cycle without special vertices. C_a reduces by 1, S does not change, and I_{ca} either does not change or reduces by 1. In all the cases, $T(G)$ changes by at most 1.

2.3. Merging two cycles into one. Consider the cases.

2.3.1. The same as in the preceding proof.

2.3.2. Merging an (a, b)-cycle and an a-cycle or merging two a-cycles. C_a reduces by 1, I_{ca} either does not change or reduces by 1, and $B + S$, as was shown above, either does not change or increases by 1. In all the cases, $T(G)$ changes by at most 1.

2.3.3. The same as in the preceding proof.

2.4. The same as in the preceding proof. □

The space and time linearity of the algorithm is obvious now.

Theorem 1 is proved. □

A separate difficult question is how the topology of intermediate structures occurring when passing from a to b differs from the topology of a and b themselves. Let, for example, a and b consist of a single cycle each. In what cases can a be transformed to b by a shortest sequence in such a way that all intermediate structures also consist of a single cycle? Consider a weaker requirement: a second cycle may appear in an intermediate structure, but in this case again a single cycle must remain at the next step. Theorem 2 gives a partial result in this direction.

Theorem 2. *Let a and b contain no special arcs, both structures consist of a single cycle, and only the DCJ operation be allowed. There exists a shortest sequence transforming a to b such that any intermediate structure in it consists of no more than two cycles; if a second cycle appears in the sequence of structures after some DCJ operation, then the next DCJ operation results in a single cycle.*

Proof. In what follows, we refer to a cycle in a structure as a *component*, keeping the standard term for a cycle in a breakpoint graph. We use a result from [1, Section 3]: under these conditions, a sequence of operations that finalizes $a + b$ is shortest if and only if the quality of $a + b$ increases by 1 at every step; the quality is upper bounded by the number of arcs in a or b. At the first step, apply a DCJ operation to $a + b$ so that the quality increases by 1 and b does not change; in our conditions, this is possible. Let at some step a breakpoint graph $a_0 + b_0$ appear such that a_0 contains a second component (the other case is symmetric). For brevity, we omit the index 0 in $a_0 + b_0$. Consider two cases.

1) There exists a cycle in $a + b$ that contains an a-edge e_1 from the first component and an a-edge e_2 from the second. Apply to the pair (e_1, e_2) a DCJ operation splitting the cycle in $a + b$ into two cycles. Then the two components merge into one, and the quality of $a + b$ increases by 1, as required.

2) Each cycle in $a + b$ contains only extremities of arcs from the first component or only those from the second. Then there is no b-edge joining an arc extremity from the first component with an extremity from the second, which contradicts the condition that b consists of a single component. □

Let us show how one can derive from the proof of Theorem 2 a quadratic time algorithm constructing the described shortest sequence (the authors do not know whether a linear time algorithm is possible). Let us prove that in the first case the two above-mentioned edges e_1 and e_2 can be chosen neighboring, i.e., separated by exactly one b-edge. Indeed, there exists a b-edge joining the extremity of an arc from the first component with an extremity from the second. The a-edges adjacent to it form a desired pair. After each operation over a (and the same for b), by traversing the components in a we mark which arcs belong to the first components and which belong to the second. Then by traversing all cycles in $a + b$ we find the above-described b-edge and the pair (e_1, e_2). The search for the next argument of a DCJ-operation is performed in linear time, so that the total runtime of the algorithm is quadratic.

We can ensure an additional property of the described algorithm: if there exists a DCJ operation (say over a) that increases the quality of $a + b$ and does not produce a second component in a, then the algorithm chooses such an operation. Let us prove that in this case two edges that are arguments of an operation can be chosen neighboring. We will traverse the unique component in a and simultaneously traverse the corresponding vertices and edges in $a + b$, orienting the edges in the traversal direction. It is easily seen that a required DCJ exists if and only if in some cycle in $a + b$ there are two oppositely oriented edges. Then there also exist two neighboring oppositely directed edges that are arguments of a desired DCJ. Before choosing an operation, we make the above-described orientation of edges in $a + b$, then by traversing all cycles in $a + b$ we find a pair of oppositely directed neighboring a-edges or b-edges, and apply a DCJ to them. Clearly, the runtime of such an algorithm remains to be quadratic.

ACKNOWLEDGMENTS

This research was carried out at the expense of the Russian Science Foundation, project no. 14-50-00150. It was also carried out using supercomputers at Joint Supercomputer Center of the Russian Academy of Sciences (JSCC RAS). The authors are deeply grateful for a reviewer, whose remarks helped them to considerably improve the text.

REFERENCES

1. K. Yu. Gorbunov and V. A. Lyubetsky, "A linear algorithm for the shortest transformation of graphs with different operation costs," J. Commun. Technol. Electron. **62**, 653–662 (2017). doi 10.1134/S1064226917060092
2. K. Yu. Gorbunov and V. A. Lyubetsky, "Linear algorithm for minimal rearrangement of structures," Probl. Inform. Transmiss. **53**, 55–72 (2017). doi 10.1134/S0032946017010057
3. K. Yu. Gorbunov and V. A. Lyubetsky, "A linear algorithm of graph reconfiguration," Autom. Remote Control, No. 12 (2018, in press).
4. K. Yu. Gorbunov, R. A. Gershgorin, and V. A. Lyubetsky, "Rearrangement and inference of chromosome structures," Mol. Biol. (Moscow) **49**, 327–338 (2015). doi 10.1134/S0026893315030073

5. V. A. Lyubetsky, R. A. Gershgorin, A. V. Seliverstov, and K. Yu. Gorbunov, "Algorithms for reconstruction of chromosomal structures," BMC Bioinform. **17**, 40.1–40.23 (2016). doi 10.1186/s12859-016-0878-z
6. V. A. Lyubetsky, R. A. Gershgorin, and K. Yu. Gorbunov, "Chromosome structures: reduction of certain problems with unequal gene content and gene paralogs to integer linear programming," BMC Bioinform. **18**, 537.1–537.18 (2017). doi 10.1186/s12859-017-1944-x
7. Z. Yin, J. Tang, S. W. Schaeffer, and D. A. Bader, "Exemplar or matching: modeling DCJ problems with unequal content genome data," J. Combinat. Optimiz. **32**, 1165–1181 (2016). doi 10.1007/s10878-015-9940-4
8. *Models and Algorithms for Genome Evolution,* Ed. by C. Chauve, N. El-Mabrouk, and E. Tannier, Comput. Biol. Series (Springer, London, 2013).
9. K. Yu. Gorbunov and V. A. Lyubetsky, "The minimum-cost transformation of graphs," Dokl. Math. **96**, 503–505 (2017). doi 10.1134/S1064562417050313
10. R. A. Gershgorin, K. Yu. Gorbunov, O. A. Zverkov, L. I. Rubanov, A. V. Seliverstov, and V. A. Lyubetsky, "Highly conserved elements and chromosome structure evolution in mitochondrial genomes in ciliates," Life **7**, 9.1–9.11 (2017). doi 10.3390/life7010009
11. M. D. V. Braga, E. Willing and J. Stoye, "Double cut and join with insertions and deletions," J. Comput. Biol. **18**, 1167–1184 (2011). doi 10.1089/cmb.2011.0118
12. P. H. da Silva, R. Machado, S. Dantas, and M. D. V. Braga, "DCJ-indel and DCJ-substitution distances with distinct operation costs," Algorithms Mol. Biol. **8**, 21.1–21.15 (2013). doi 10.1186/1748-7188-8-21
13. P. E. C. Compeau, "DCJ-indel sorting revisited," Algorithms Mol. Biol. **8**, 6.1–6.9 (2013). doi 10.1186/1748-7188-8-6
14. P. E. C. Compeau, "A generalized cost model for DCJ-indel sorting," in *Proceedings of 14th International Workshop on Algorithms in Bioinformatics, Wroclaw, Poland, Sept. 8–10, 2014,* Lect. Notes Comput. Sci. **8701**, 38–51 (2014). doi 10.1007/978-3-662-44753-6_4
15. S. Hannenhalli and P. A. Pevzner, "Transforming men into mice (polynomial algorithm for genomic distance problem)," in *Proceedings of the 36th Annual Symposium on Foundations of Computer Science—FOCS 1995, Milwaukee, USA, Oct. 23–25, 1995,* pp. 581–592.
16. G. Li, X. Qi, X. Wang, and B. Zhu, "A linear-time algorithm for computing translocation distance between signed genomes," in *Proceedings of 15th Annual Symposium on Combinatorial Pattern Matching—CPM 2004, July 5–7, 2004, Istanbul, Turkey,* Lect. Notes Comput. Sci. **3109**, 323–332 (2004). doi 10.1007/978-3-540-27801-6_24
17. A. Bergeron, J. Mixtacki, and J. Stoye, "A new linear time algorithm to compute the genomic distance via the double cut and join distance," Theor. Comput. Sci. **410**, 5300–5316 (2009). doi 10.1016/j.tcs.2009.09.008
18. A. Bergeron, J. Mixtacki, and J. Stoye, "A unifying view of genome rearrangements," in *Proc. of 6th International Workshop on Algorithms in Bioinformatics, Zurich, Switzerland, Sept. 8–10, 2006,* Lect. Notes Bioinform. **4175**, 163–173 (2006). doi 10.1007/11851561_16
19. M. A. Alekseyev and P. A. Pevzner, "Multi-break rearrangements and chromosomal evolution," Theor. Comput. Sci. **395**, 193–202 (2008). doi 10.1016/j.tcs.2008.01.013