An Efficient Approach of Computing *Double Cut and Join Distance for Genomes* with Duplicate Genes

Md. Ishat-E-Rabban[1], Shibbir Ahmed[2], and Md. Nazmul Hoq[3]

[1]**1014052029, ieranikg@gmail.com**
[2]**1015052005, shibbirahmedtanvin@gmail.com**
[3]**1015052068, mnsalim.cse@gmail.com**

Department of Computer Science and Engineering,
Bangladesh University of Engineering and Technology

Abstract Genome Rearrangement refers to large-scale mutations in genomes which is responsible for complex changes and structural variations. Computing the edit distance between two genomes is a fundamental problem in the study of genome evolution. The Double Cut and Join (DCJ) model forms the basis for most algorithmic research on Genome Rearrangements. Edit distance under theDCJ model can be computed in linear time for genomes without duplicate genes but DCJ computation for two genomes with duplicate genes is NP-Hard. All the existing solutions to the problem of finding DCJ distance between two genomes containing duplicate genes construct an adjacency graph and find the permutation with minimum number of cycles. We propose a new approach to the problem that involves A* search. We provide a tight heuristic to accelerate the search and also a preprocessing technique to farther improve the performance of our solution. Our solution works efficiently and correctly if there are no duplicate genes but might not find the optimum answer if the genomes contain duplicate genes.

1. Introduction

Genome rearrangement means major genomic mutation due to erroneous cell division after meiosis or mitosis. The emergence of entire genome sequencing has provided us with huge amount of data on which to study genomic rearrangements. The research of genomic rearrangements has been emerging since the problem was formulated two decades ago. A fundamental problem in genome rearrangements is to compute the edit distance between two genomes. Edit distance refers to the minimum number of operations needed to transform one genome into another. Under the inversion model, Hannenhalli and Pevzner gave the first polynomial-time algorithm to compute the edit distance for unichromosomal genomes [1], which was later improved to linear time [2].

Genome rearrangements have been modeled by a variety of operations such as inversions, translocations, fissions, fusions, transpositions and block interchanges. All these rearrangements can be represented by the double cut and join (DCJ) operation [3], which basically consists of cutting a genome in two distinct positions (possibly in two distinct chromosomes) and joining the four resultant open ends in a different way. A simple approach to apply this operation to the most general type of genomes with a mixed collection of linear and circular chromosomes was proposed in [4].

Most of the algorithms assume genomes contain no duplicate genes. However, gene duplications are widespread events and have long been recognized as a major driving force of evolution [5, 6]. For example, in human genomes segmental duplications are hotspots for non-allelic homologous recombination leading to genomic disorders, copy-number polymorphisms, and gene and transcript innovations [7]. Chen et al. [8] studied the problem of computing the inversion distance for genomes in the presence of duplicate genes. Moreover, for genomes with duplicate genes, computing the rearrangement distance is NP hard [9] even when the genomes have the same content and only DCJ operations are allowed to do so [10].

All the prevalent solutions to the problem of finding DCJ distance between two genomes containing duplicate genes normally construct an adjacency graph and find the permutation with minimum number of cycles. In this research study, we propose a new approach to the problem which is based on A* search. We provide a tight heuristic to accelerate the search and also an effective preprocessing technique to improve the performance to the extent of our solution. Our proposed solution works efficiently and precisely if there are no duplicate genes. Although it may not find the optimum solution if the genomes are comprised of duplicate genes.

2. Preliminaries

Genome is the entire DNA of a living organism. Gene is a segment of DNA that is involved e.g. in producing a protein and its orientation depends on the DNA-strand that it lies on. Genome consists of Chromosomes. Chromosomes are linear or circular list of Genes. To sum up, genome is a set of chromosomes and Chromosome is a linear or circular list of genes which is known as synteny blocks.

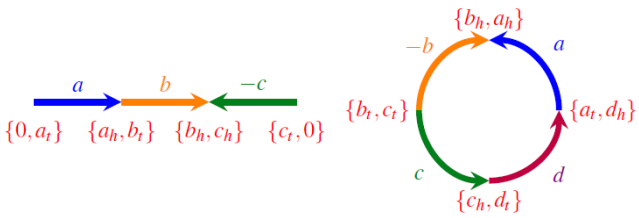

Figure 1. Extremities, Adjacency and Telomere of a gene

Extremities are two ends (head and tail) of a gene as illustrated in the Figure 1. Adjacency refers to two consecutive extremities which is also clearly indicated in the Figure 1. Another important term is Telomere which is explained as an extremity that is not adjacent to any other gene is called a telomere. In Figure 1, a_t , c_t are two telomeres.

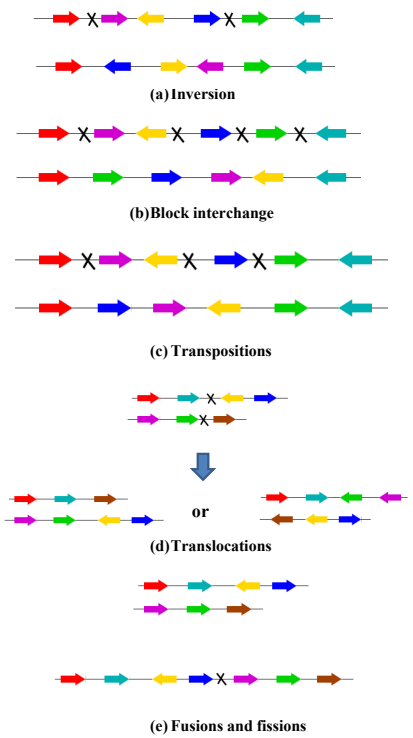

Figure 2. Variety of operations of Genome Rearrangement

Genome rearrangements have been modeled by a variety of operations such as inversions, translocations, fissions, fusions, transpositions and block interchanges. Now, Inversions reverse the order and the orientation of a segment as illustrated in Figure 2(a). Block interchanges exchange two segments which is also clearly shown in Figure 2(b). Transpositions are block interchanges whose exchanged segments are adjacent which has been demonstrated in Figure 2 (c). Translocations exchange two chromosome ends as it has been demonstrated in the Figure 2 (d). Finally, Fusions and fissions (shown in Figure 2 (e)) are translocations involving or creating empty chromosomes.

3. Related Work

The solutions of sorting multi-chromosomal genomes depend on what kind of rearrangement operations is allowed. Given their prevalence in eukaryotic genomes [11], the usual choices of operations include translocations, fusions, fissions and inversions. However, there are some indications that transpositions should also be included in the set of operations [12], but the lack of theoretical results showing how to include transpositions in the models led to algorithms that simulate transpositions as sequences of inversions. In the paper[3], the authors describe a general framework in which circular and linear chromosomes can coexist throughout evolving genomes. They model inversions, translocations, fissions, fusions, transpositions and block interchanges with a single operation, called the double cut and join operation. This general model accounts for the genomic evidence of the coexistence of both linear [13] and circular chromosomes or plasmids in many genomes [14].

With respect to the DCJ operation, the first problem is to investigate formal properties of graphs that are unions of paths and cycles. These graphs also give a firm starting point to explore difficult rearrangement problems that involve either gene duplications [15] or missing information about the actual order of genes in a genome [16]. Again, the Hannenhalli-Pevzner distance, that allows only translocations and inversions on linear chromosomes [17], can be recast as avoiding all DCJ operations that create a circular chromosome in either genomes. Another kind of restriction has recently been studied in [18], where operations are fusions and fissions between circular unsigned chromosomes, and block interchanges within a circular unsigned chromosome. The authors assign equal weight to the three operations, even if a block interchange requires two DCJ operations. Their algorithm first applies fusions to both source and target genome, until they have two genomes whose chromosomes have equal gene content. These fusions can be identified in linear time by a search of the adjacency graph. Then they sort the resulting genomes by block interchanges using an $O(N^2)$ time algorithm described in [19]. The combinatorics and algorithmics of genomic rearrangements have been the subject of

much research since the problem was formulated in the 1990s [20]. For the the multichromosomal genomes, the edit distance under the Hannenhalli-Pevzner model (inversions and translocations) has been studied through all these studies [21, 22, 23], culminating in a fairly complex linear-time algorithm [24]. Under the DCJ model, the edit distance can be computed in linear time for two multi-chromosomal genomes in a simple and elegant way [4].

A *DCJ* operation makes two cuts in the genome, either in the same chromosome or in two different chromosomes, producing four cut ends, then rejoins the four cut ends. Genomic rearrangements include inversions, transpositions, circularizations, and linearizations, all of which act on a single chromosome, and translocations, fusions, and fissions, which act on two chromosomes. These operations can all be described in terms of the single double-cut-and-join (DCJ) operation [25, 26]], which has formed the basis for most algorithmic research on rearrangements over the last few years [27, 28].

It has been studied [8] that the problem of computing the inversion distance for genomes in the presence of duplicate genes. It has been proved that the problem is NP-hard and designed heuristics to solve it, which thus packaged into the SOAR software system. They applied SOAR to assign orthologs on a genome wide scale. Later, they extended SOAR to unite rearrangements and single-gene duplications as a new software package, called MSOAR, which can be applied to detect inparalogs in addition to orthologs [29]. Recently, they incorporated tandem duplications into their model, and demonstrated that the new system achieved a better sensitivity and specificity than MSOAR [30]. In the paper [31], authors focus on the problem of computing the edit distance for two genomes with duplicate genes under the DCJ model. In [32], they described a capping method to remove telomeres by introducing null extremities. The problem is also NP-hard, which can be proved by a reduction from the NP-hard problem of breakpoint graph decomposition [33]. In another paper [34] authors recently studied the problem of computing the DCJ distance between two genomes with the same content and possibly duplicate genes, with the restriction that they have exactly the same number of copies of each gene in each genome.

4. Methodology

As discussed in the related works section, it is evident that all existing methods to compute the double cut and join distance for genomes with duplicate gene use a particular approach based on graph theory. The previous approaches form an adjacency graph between the source and destination genome and try to minimize the number of circles over all possible bijections and/or permutations. In order to find the bijection that minimizes the number of circles, they use several techniques. The first work mentioned in the related works section uses an integer linear programming (ILP) formulation and conducts a preprocessing step involving an iterative discharge step to find the optimum solution. The second approach mentioned in the related work section uses a variant of the minimum common string partition problem to develop a linear time approximation algorithm where the number of duplicates is assumed to be bounded. We propose a new solution to the problem of calculating the double cut and join distance between two genomes with duplicate genes that, unlike all the previous works, does not leverage on the idea of adjacency graph. Instead, we adopt a different paradigm, A* search, which is frequently used in the field of artificial intelligence. A* search can be used to find the optimum path between two nodes in a graph. The performance of the A* search can be improved by using a tight heuristic function. In our solution we use A* search to find the distance between the given source and destination genome and use a heuristic function that provides a tight lower bound to the predicted distance to the destination. We also use some preprocessing on the source and destination genomes to achieve farther speedup in the performance of the A* search. In Section 3.1, we discuss the general procedure of the A* search and how our problem is translated to an instance of a graph search problem, where A* search can be used to find the optimum solution. Next in Section 3.2, we introduce the heuristic that we have used to drive the A* search. Finally in Section 3.3, we describe the preprocessing step that we have used to improve the performance of the heuristic driven A* search.

4.1. A* Search

In this section, first we discuss some preliminary ideas regarding the A* search and then we present how the problem of finding the double cut and join distance between two genomes containing duplicate genes can be translated to an instance of the A* search. As a result solving the A* search problem eventually results in finding the DCJ distance between the source and destination genomes. A* is an informed search algorithm for finding the optimum path between two nodes of a graph. It uses best first approach to select the node to expand next. It solves problems by searching among all possible paths to the destination node for the one that incurs the smallest cost. Among these paths it first considers the ones that appear to lead most quickly to the solution. It is formulated in terms weighted graph. It starts from the source node and it constructs a tree of paths starting from that node, expanding paths one step at a time, until one of its paths ends at the destination node. At each iteration, the A* algorithm needs to determine which of its partial paths to expand into a longer path. It does so based on an estimate of the cost still to go to the destination node. Specifically, A* selects the path that minimizes,

$$f(n) = g(n) + h(n)$$

where n is the last node on the path, $g(n)$ is the cost of the path from the source node to n, and $h(n)$ is a heuristic that estimates the cost of the cheapest path from n to the destination. The heuristic is problem-specific. For the algorithm to find the actual shortest path, the heuris-

tic function must be admissible, meaning that it should never overestimate the actual cost to get to the destination node. A* considers fewer nodes than any other admissible search algorithm with the same heuristic. This is because A* uses an optimistic estimate of the cost of a path through every node that it considers-optimistic in the sense that the true cost of a path through that node to the destination will be at least as great as the estimate.

4.2. Our Approach

Now we describe how the DCJ problem can be converted to an A* search problem. Consider an undirected graph G, where there exists one node in G for each different permutation of the genes in the source genome (or equivalently the destination genome, as the source and destination contain the same set of genes). There exists an undirected edge between two nodes, u and v, of G, if the genome represented by u can be translated to the genome represented by v by a single DCJ operation or vice versa. Each edge of G has a weight of 1. There is a one-to-one correspondence between the actual DCJ problem and the problem of determining the distance between two nodes in G, where the two nodes represent the source and destination genome. Thus an instance of a DCJ problem can be translated to an instance of A* search in an undirected graph. So if we are provided with an instance of the DCJ problem, first we construct the graph G as stated above. Then we perform A* search on G to find out the distance between the nodes representing the source and destination genome and this distance is the desired DCJ distance. The algorithm depicting the whole procedure is given in Algorithm1.

Algorithm 1: computeDCJ(src, dst)

Input: Source Genome, Destination Genome
Output: DCJ Distance

```
1  construct G from src and dst
2  open_list = set containing the node for src
3  closed_list = empty set
4  src.g = 0
5  src.f = src.g + heuristic(src, dst)
6    while open_list != Empty
7          cur = open_list element with lowest f cost
8          if cur = dst
9            return cur.f //path found
10         remove cur from open_list
11         add cur to closed_list
12         for each n in neighbors(cur)
13           if n not in closed_list
14             n.f = n.g + heuristic(n, dst)
15             if n is not in open_list
16               add n to open_list
17             else
18               open_n = neighbor in open_list
19               if n.g < open_n.g
20                 open_n.g = n.g
21                 open_n.parent = n.parent
22 return NULL // no path exists
```

In the Algorithm 1, $computeDCJ$, we maintain two lists. $open\_list$ and $closed\_list$. $open\_list$ consists of nodes that have been visited but not expanded (meaning that successors have not been explored yet). This is the list of pending tasks. $closed\_list$ consists on nodes that have been visited and expanded (successors have been explored already and included in the $open_list$, if this was the case).

4.3. The Heuristic

In this section, we describe the heuristic function that we have used to drive the A* search. Note that in lines 5 and 14 of the algorithm $computeDCJ$, we have calculated the heuristic distance to the destination node. The heuristic value of a node, n, denoted by $h(n)$, serves as an estimation of the distance between n and the destination node. For the algorithm to find the actual shortest path, the heuristic function must be admissible, meaning that it never overestimates the actual cost to get to the nearest destination node. The heuristic value of a node n provides a lower bound on the predicted distance between n and the destination node. Thus the actual distance between n and the destination is at least as large as $h(n)$. The performance of the A* search is largely dependent on the choice of the heuristic function. If a heuristic function is able to provide a very tight lower bound, the A* search will converge to the destination node with fewer expansions. In this way a tight heuristic function helps to improve the performance of the A* search. Now we discuss the heuristic function used in our solution to estimate the distance to the destination node from a given node.
In order to devise an efficient heuristic for the A* search, we make some observations. Figure 3 lists all possible types of DCJ operations.

We intend to track the changes in adjacencies and telomeres and the changes in the number of adjacencies and telomeres. Table 1 lists these changes for each of six types of DCJ operation. Here the types are assumed to be numbered from 1 in a row-major order from the top-left.
Table 1 allows us to make the following four key observations.

1. One DCJ operation changes the number of adjacencies at most by 1
2. One DCJ operation changes the number of telomeres at most by 2
3. One DCJ operation changes at most 2 adjacencies
4. One DCJ operation changes at most 1 telomere

The above key observations allow us to provide a lower bound on the distance between two given genomes and thus devise a heuristic function. Given two genomes g and h, we make a list of their adjacencies and telomeres. Then we determine the changes in adjacencies and telomeres and the changes in the number of adjacencies and telomeres. The four key observations provide a lower bound on the number of DCJ steps required to make

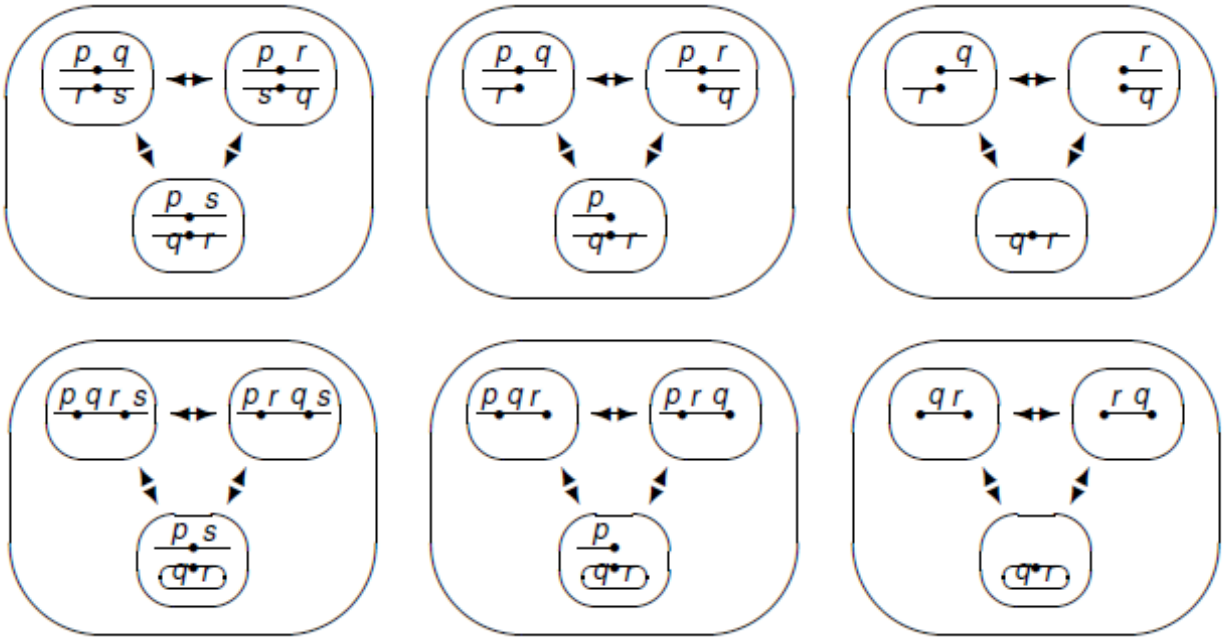

Figure 3. List of all possible types of DCJ operations

Table 1. Changes in adjacencies and telomeres for each of six types of DCJ operation

Type	Change in number of adjacencies	Change in number of telomeres	Change in adjacencies	Change in telomeres
1	0	0	2	0
2	0	0	1	1
3	1	2	0	0
4	0	0	2	0
5	0	0	1	1
6	1	2	0	0

the changes. Figure 4 illustrates with an example how the heuristic distance between two genomes can be computed.

Here two adjacencies remain unchanged, namely (B, C) and (D, E). But the adjacency (A, B) get changed. Also the number of adjacencies has increased by 1 as the destination genome has 4 adjacencies while the source genome has 3. From Table 1 and the key observations, we can deduce that a single DCJ operation cannot bring about all these changes in the adjacencies. At least two DCJ operations are required. Similarly we can list the telomeres and derive another lower bound and we can use the maximum of these two lower bounds as the heuristic distance between two genomes.

The heuristic function discussed above provides a very tight lower bound. For example, in Figure 4, the actual DCJ distance between the source and destination genome is 2 which equals the estimated heuristic distance. Thus we perform an A* search with the above formulated heuristic to determine the DCJ distance between two genomes.

4.4. The Preprocessing Step

In this section, we discuss a preprocessing step which can be used to improve the performance of the basic A* search solution. First we mention a bottleneck that slows down the A* search. Then we discuss the intuition behind developing a preprocessing step that can remove the bottleneck. Finally we describe the preprocessing phase in details.

In the basic A* solution discussed thus far, there is an overhead that degrades the performance despite the use of a good heuristic function. In line 12 of the Algorithm 1 $computeDCJ$, we expand all the neighbors of the current node and compute the heuristic values of the neighboring nodes. But note that if the number of genes in the genome is n, the number of neighbor nodes of a given node is $O(n^2)$. This is because from a genome comprising n genes, we can go the n^2 other nodes by making a single DCJ operation. For a DCJ operation we need to choose two locations in a genome and two locations from a genome of n genes can be selected in n^2 ways. So at each step of $computeDCJ$, we need to expand all these neighboring nodes and compute their heuristic distance to the destination. Computing the heuristic value for all these neighbors in each step of the algorithm is a prohibitively

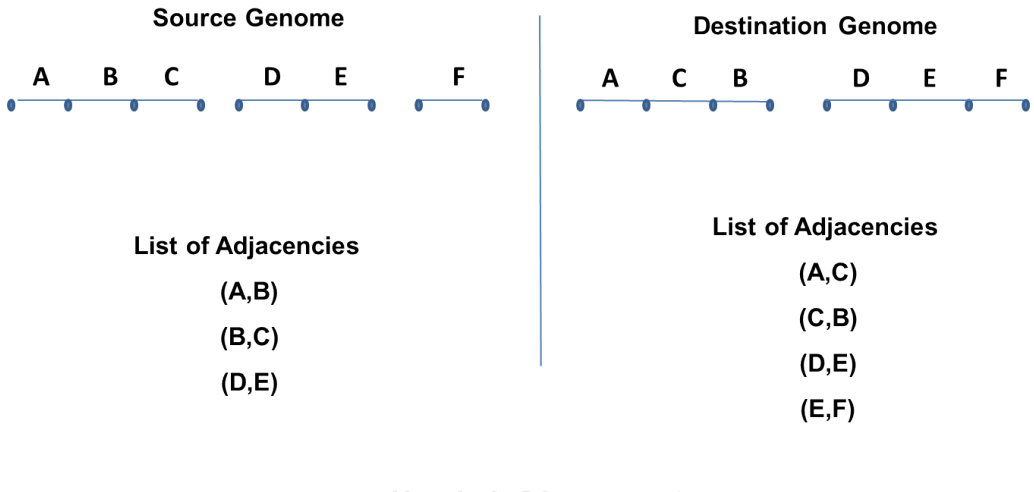

Figure 4. An Example

expensive task and can hamper the performance of the algorithm $computeDCJ$ to a great extent. So we conduct a preprocessing step on the given source and destination genomes before the $A*$ search and limit the set of neighbors to be expanded with a view to removing this bottleneck.

Figure 5 illustrates the intuition behind the preprocessing step. In this figure two DCJ operations are shown. In the first operation, the yellow and red segments got inverted. In the second operation, the selected locations to split the genome were the red-yellow junction and the purple-green junction.

Observe that the order of genes were unchanged in the single color segments. There can thousands of genes in these segments. So we can deduce that if a contiguous segment of genes in the source genome can be found unchanged in the destination genome, then no DCJ cut locations were selected within the segment. Otherwise it cannot be found in the same order in the destination genome. In this way we can restrict the potential cut locations in a genome, which will result in the decrease of the size of the set of neighbors of a node which represents the number of potential cut locations. This intuitive idea allows us to develop the following preprocessing step to improve the performance.

The preprocessing phase has the following steps:

1. Determine the maximal contiguous segment staring at each index in the source genome that can be found in the destination genome.

2. Determine a subset of the segments found in Step 1 of minimum cardinality where the segments in the subset are mutually disjoint and cover the whole genome.

3. The boundaries of the segments found in Step 2 are the potential cut locations.

Figure 6 illustrates the ideas developed above. The left portion of the figure shows the output of step 1 and the right portion shows a disjoint subset of minimum cardinality that spans the whole genome and the corresponding cut locations.

Thus we are able to restrict the potential cut locations. This helps us achieve the desired speedup and remove the bottleneck at line 12 of the Algorithm 1 $computeDCJ$.

5. Results & Discussion

Based on the methodology proposed in Section 4, we arrive at the following results. The A* search method described up to Section 4.2 inclusive preforms correctly, but has the overhead of expanding all the neighbors of a node. Section 4.4 describes a preprocessing phase to eliminate the overhead. But the techniques developed in the preprocessing step are only applicable in case where the source and destinations contain no duplicate genes. Otherwise the preprocessing step may not yield the correct output. If the source and destination genomes contain duplicate genes the following situations may arise:

1. The segments in the source genome might overlap in the destination genome and thus not cover the whole destination genome.

2. A segment in the source genome can be found at multiple places in the destination genome.

3. We used minimum number of segments to imply the cut locations which might not be the case.

Thus the existence of duplicate genes in the source or destination genome can invalidate the preprocessing step. To alleviate this problem, we can derive an order of the cut locations and impose the neighbors to be expanded in the given order. Thus the more probable cut locations will be explored early and the other branches will be pruned faster.

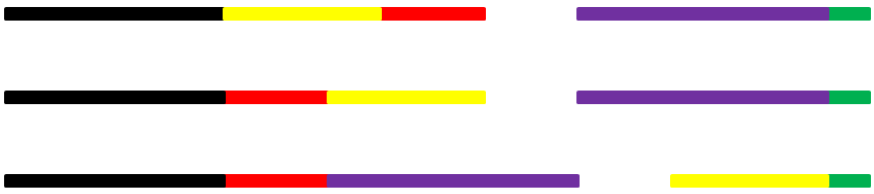

Figure 5. Intuition behind the preprocessing step

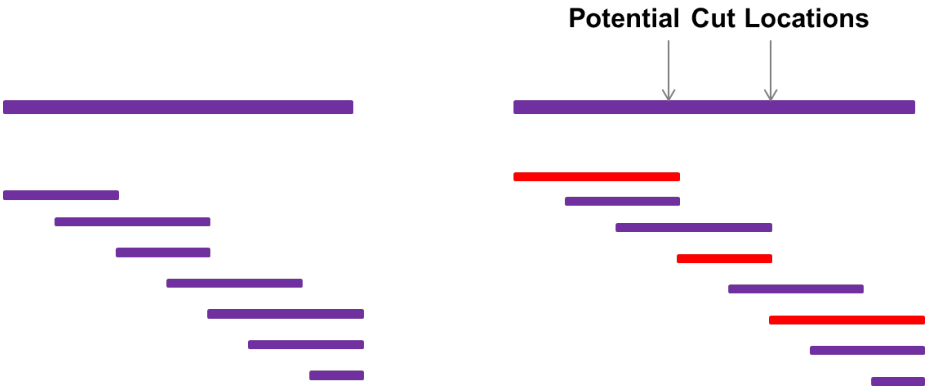

Figure 6. Determining potential cut locations

6. Conclusions

In order to solve the problem of computing the edit distance for two genomes with duplicate genes under the *DCJ* model, at first we perform A* search to find the destination genome from source genome. At each step consider all possible pair of adjacencies as well as telomeres for DCJ and expand the node with smallest heuristic value. The heuristic must be admissible, i.e., it should never overestimate the cost to the goal node. The heuristic must be tight enough to ensure better performance. At the preprocessing step, all contiguous segments of source and destination genome are formed. Then, the task is to identify the segments of source genome that are found in the destination genome as a contiguous segment. The segments in the source genome might overlap in the destination genome and thus not cover the destination genome. A segment in the source genome can be found at multiple places in the destination genome. We have used minimum number of segments to imply the cut locations which might not be the case.

From literature review, it is evident that all other existing solutions to the problem of finding DCJ distance between two genomes containing duplicate genes construct an adjacency graph and find the permutation with minimum number of cycles. Here, we have proposed a new approach the problem that involves A* search along with a tight heuristic to extend the search and also a preprocessing technique to farther enhance the performance of our solution. Our proposed solution works accurately and much more effectively if there are no duplicate genes but might not find the optimum result if the genomes contain duplicate genes.

Acknowledgements

The authors would like to thank Dr. Atif Hasan Rahman for providing fruitful suggestions and directions to complete the project. As this project is part of the course CSE 6406 (Bioinformatics Algorithms) of M.Sc. in CSE, BUET, authors would also like to acknowledge the suggestions of the classmates during the presentation session of the project progress .

References

[1] Sridhar Hannenhalli and Pavel A Pevzner. Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. *Journal of the ACM (JACM)*, 46(1):1–27, 1999.

[2] David A Bader, Bernard ME Moret, and Mi Yan. A linear-time algorithm for computing inversion distance between signed permutations with an experimental study. *Journal of Computational Biology*, 8(5):483–491, 2001.

[3] Sophia Yancopoulos, Oliver Attie, and Richard Friedberg. Efficient sorting of genomic permutations by translocation, inversion and block interchange. *Bioinformatics*, 21(16): 3340–3346, 2005.

[4] Anne Bergeron, Julia Mixtacki, and Jens Stoye. A unifying view of genome rearrangements. In *International Workshop on Algorithms in Bioinformatics*, pages 163–173. Springer, 2006.

[5] Jeffrey A Bailey and Evan E Eichler. Primate segmental duplications: crucibles of evolution, diversity and disease. *Nature Reviews Genetics*, 7(7):552–564, 2006.

[6] Michael Lynch and Bruce Walsh. *The origins of genome architecture*, volume 98. Sinauer Associates Sunderland, 2007.

[7] Zhaoshi Jiang, Haixu Tang, Mario Ventura, Maria Francesca Cardone, Tomas Marques-Bonet, Xinwei She, Pavel A Pevzner, and Evan E Eichler. Ancestral reconstruction of segmental duplications reveals punctuated cores of human genome evolution. *Nature genetics*, 39 (11):1361–1368, 2007.

[8] Xin Chen, Jie Zheng, Zheng Fu, Peng Nan, Yang Zhong, Stefano Lonardi, and Tao Jiang. Assignment of orthologous genes via genome rearrangement. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 2(4):302–315, 2005.

[9] Sébastien Angibaud, Guillaume Fertin, Irena Rusu, Annelyse Thévenin, and Stéphane Vialette. On the approximability of comparing genomes with duplicates. *Journal of Graph Algorithms and Applications (JGAA)*, 13(1):19–53, 2009.

[10] Sébastien Angibaud, Guillaume Fertin, Irena Rusu, and Stéphane Vialette. A pseudo-boolean framework for computing rearrangement distances between genomes with duplicates. *Journal of Computational Biology*, 14(4):379–393, 2007.

[11] Matthew Mazowita, Lani Haque, and David Sankoff. Stability of rearrangement measures in the comparison of genome sequences. *Journal of Computational Biology*, 13 (2):554–566, 2006.

[12] Anne Bergeron and Jens Stoye. On the similarity of sets of permutations and its applications to genome comparison. In *International Computing and Combinatorics Conference*, pages 68–79. Springer, 2003.

[13] Sherwood Casjens, Nanette Palmer, René Van Vugt, Wai Mun Huang, Brian Stevenson, Patricia Rosa, Raju Lathigra, Granger Sutton, Jeremy Peterson, Robert J Dodson, et al. A bacterial genome in flux: the twelve linear and nine circular extrachromosomal dnas in an infectious isolate of the lyme disease spirochete borrelia burgdorferi. *Molecular microbiology*, 35(3):490–516, 2000.

[14] Jean-Nicolas Volff and Josef Altenbuchner. A new beginning with new ends: linearisation of circular chromosomes during bacterial evolution. *FEMS microbiology letters*, 186 (2):143–150, 2000.

[15] Chunfang Zheng, Aleksander Lenert, and David Sankoff. Reversal distance for partially ordered genomes. *Bioinformatics*, 21(suppl 1):i502–i508, 2005.

[16] David Sankoff. Genome rearrangement with gene families. *Bioinformatics*, 15(11):909–917, 1999.

[17] Sridhar Hannenhalli and Pavel A Pevzner. Transforming men into mice (polynomial algorithm for genomic distance problem). In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, pages 581–592. IEEE, 1995.

[18] Chin Lung Lu, Yen Lin Huang, Tsui Ching Wang, and Hsien-Tai Chiu. Analysis of circular genome rearrangement by fusions, fissions and block-interchanges. *BMC bioinformatics*, 7(1):295, 2006.

[19] Ying Chih Lin, Chin Lung Lu, Hwan-You Chang, and Chuan Yi Tang. An efficient algorithm for sorting by block-interchanges and its application to the evolution of vibrio species. *Journal of Computational Biology*, 12(1):102–112, 2005.

[20] Guillaume Fertin. *Combinatorics of genome rearrangements*. MIT press, 2009.

[21] Cedric Chauve, Nadia El-Mabrouk, and Eric Tannier. *Models and algorithms for genome evolution*. Springer, 2013.

[22] Géraldine Jean and Macha Nikolski. Genome rearrangements: a correct algorithm for optimal capping. *Information Processing Letters*, 104(1):14–20, 2007.

[23] Michal Ozery-Flato and Ron Shamir. Two notes on genome rearrangement. *Journal of Bioinformatics and Computational Biology*, 1(01):71–94, 2003.

[24] Glenn Tesler. Efficient algorithms for multichromosomal genome rearrangements. *Journal of Computer and System Sciences*, 65(3):587–609, 2002.

[25] Xin Chen. On sorting permutations by double-cut-and-joins. In *International Computing and Combinatorics Conference*, pages 439–448. Springer, 2010.

[26] Xin Chen, Ruimin Sun, and Jiadong Yu. Approximating the double-cut-and-join distance between unsigned genomes. *BMC bioinformatics*, 12(9):1, 2011.

[27] Sophia Yancopoulos and Richard Friedberg. Sorting genomes with insertions, deletions and duplications by dcj. In *RECOMB International Workshop on Comparative Genomics*, pages 170–183. Springer, 2008.

[28] Bernard ME Moret and Tandy Warnow. Advances in phylogeny reconstruction from gene order and content data. *Methods in enzymology*, 395:673–700, 2005.

[29] Zheng Fu, Xin Chen, Vladimir Vacic, Peng Nan, Yang Zhong, and Tao Jiang. Msoar: a high-throughput ortholog assignment system based on genome rearrangement. *Journal of Computational Biology*, 14(9):1160–1175, 2007.

[30] Guanqun Shi, Liqing Zhang, and Tao Jiang. Msoar 2.0: Incorporating tandem duplications into ortholog assignment based on genome rearrangement. *BMC bioinformatics*, 11 (1):1, 2010.

[31] Mingfu Shao, Yu Lin, and Bernard Moret. An exact algorithm to compute the dcj distance for genomes with duplicate genes. In *International Conference on Research in Computational Molecular Biology*, pages 280–292. Springer, 2014.

[32] Mingfu Shao and Yu Lin. Approximating the edit distance for genomes with duplicate genes under dcj, insertion and deletion. *BMC bioinformatics*, 13(Suppl 19):S13, 2012.

[33] John Kececioglu and David Sankoff. Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement. *Algorithmica*, 13(1-2):180–210, 1995.

[34] Diego P Rubert, Pedro Feijão, Marília DV Braga, Jens Stoye, and Fábio V Martinez. A linear time approximation algorithm for the dcj distance for genomes with bounded number of duplicates. In *International Workshop on Algorithms in Bioinformatics*, pages 293–306. Springer, 2016.