**Evolutionary relationships in the *drosophila-*genus:  
a computational analysis for *d. melanogaster* and *d. miranda***

LEANDER DE KRAKER & nINA CIALDELLA – HEURISTICS 2017

**abstract**

**Evolutionary relationships are commonly studied using advanced computational analysis. This study investigates the evolutionary relationship between two species in the *Drosophila-*genus by trying to develop a series of algorithms to sort genomes in the least possible number of mutations, the lowest total mutation length and the lowest mutation score (½N2, in which N is the mutation length). The algorithms developed in this study will also be used to draw a broader conclusion on permutation sorting problems in general. The best method to solve this type of problem, as proposed by the authors of this article, is a Best First Search algorithm with a priority queue based on a cost function in combination with advanced pruning methods.**

**Introduction**

*Drosophila* is a genus of small flies, most commonly referred to as fruit flies due to the habit of many of its species to circle around rotting fruit. The entire genus contains over 1,500 distinct species with a wide variety in (breeding) behaviour and appearance. One species of this genus, *D. melanogaster*, is used as a model organism in developmental biology and genetics, resulting in an abundance of knowledge about the genus and this species in particular. The entire genus, however, also holds valuable information for evolutionary biology and speciation. The genus can be found on all continents and its high number of species and diversity of characteristics in combination with a relatively small genome and knowledge about breeding the species, makes the genus a great candidate for evolutionary studies. This study will analyse the genomes of two related *Drosophila* species, namely: *D. melanogaster* and *D. miranda* by developing a number of algorithms suitable for investigating their evolutionary relationship*.*

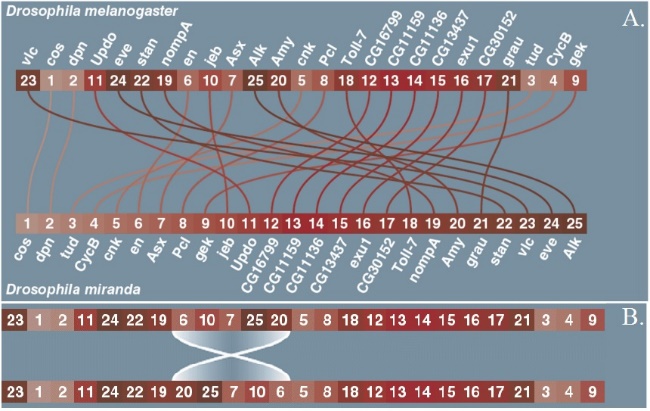
Species in a genus often show high genetic resemblance, as is the case for this study as well. Genetic research has shown that the genomes of *D. melanogaster* and *D. miranda* consist of 25 identical genes but in a different order. Therefore, we will simplify our model into a series of 25 numbers, as shown in figure 1.1a. Our model will also be a simplification of the real biology by not taking epigenetic marking, which is found in all eukaryotes, and variation in genetics coding for the same genes into account. Evolutionary changes happen through mutations. In this specific case the only possible type of mutation are flips, in which an entire subsequence is flipped as a whole (figure 1.1b). Evolutionary biology uses several methods, one of which is referred to as maximum parsimony. This method, used for deciding on the evolutionary relationship between species in a phylogenetic tree, focusses on finding the smallest number of evolutionary events to explain the genetic data. Another, more advanced method, is called Bayesian interference. This method relies on assumptions which are made beforehand based on the likelihood of certain mutations to occur, resulting in a more reliable phylogenetic tree. In this study, we will analyse several methods to find pathways of maximum parsimony, thus focussing on trying to find the least numbers of flips to transfer the genome of *D. melanogaster* into the genome of *D. miranda.* We will also focus on mutation scores which favour shorter mutations over longer mutations. This theory is based on biological research which has concluded that long mutations are less likely to occur than short mutations, due to the risks and high failure rates for longer mutations. We will use two different scores, one which is the total sum of all mutation lengths and another one which is calculated by the following formula, which makes longer mutations even more unlikely: ½n2.

Fig 1.1a-b: **Genomes sequences of *D. melanogaster* and *D. miranda* and flip visualisation**. 1.1a shows the genomes of *D. melanogaster* and *D. miranda.* 1.1b visualizes a flip of length 4, which is how changes occur in the algorithms.

**State space**

The 25 genes of the *Drosophila* genome can be put in any random order. The first place can be any of the 25 genes, the second place can be any of the 24 remaining genes, third place can be any of 23 remaining genes et cetera. So the number of possible states of the genome is 25\*24\*23\*21… \*2\*1 = 25!. 25! = 15511210043330985984000000 = 1.551 \* possible ways to order the genome of the *Drosophila.*

The length of the flip can be 2 to 25 genes long on gene 1, but if the flip starts at gene 2 the flip can only be 24 genes long at the most. A flip at gene 24 can only switch around gene 24 and 25, and a flip starting at gene 25 can’t happen. So the number of possible different flips in the genome is 24+23+22+ … +2+1 = 300.

The knowledge gained through this study is not only applicable for this specific case or evolutionary studies but is of great worth for computational science and heuristics in general since it seeks to optimize a solution for a permutation sorting problem with a big state space by using a variety of methods and heuristics. To gain a deeper understanding of permutation sorting problems and add to the already available knowledge we also use the algorithms designed for the *Drosophila* case for sorting 100-random sequences of length 25 and draw a broader conclusion.

To optimize the solution to the described permutation sorting problem, we will use a variety of algorithms combined with heuristics. We will define the upper bounds by a basic flip sorter which functions like a selection sort. Furthermore, we will try to constructively solve the case by depth first and breadth first algorithms. We will expand the breadth first algorithm with a priority queue, testing a variety of cost functions, to be able to handle the size of the state space.

**Materials and Methods**

**materials**

All code was written in Python 3. The packages used are: *numpy, matplotlib, random, heapq* and *time*. The algorithms were run on a Lenovo laptop with 8 GB-ram and a dual core, 2.5GHz Intel i-7-6500u processor. The analysis on 100 random genomes of length 25 were performed in Matlab 2016b.

**methods**

**Flip Sorter**

To identify the upper bound of the problem, we implemented a basic flip-sorter. This sorter sorts the sequence by moving the small numbers to the beginning of the sequence, in a way comparable to selection sort, but it uses flips instead of swapping selected numbers.

**Depth First Search**

After defining our upper bounds, we implemented an algorithm known as a Depth First Search (DFS). DFS algorithms are always constructive searching algorithms, which search for a solution to a problem by traversing down a tree. In our case, we try out a new mutation on a sequence and progress deeper into the branch by trying out another mutation on the new sequence.

Prior to running the algorithm, we define all possible mutations for the sequence of interest. When running the algorithm, we keep track of the mutations through a mutation tracker. This tracker assures that we are always able to retrieve information about all previous flips. Furthermore, the combination of defining all possible mutations and keeping track of them as well allows us to make sure that we never try the same mutation on a genome and helps saving time as the process continues because we are not randomly assigning mutations so we never have to test if our mutation is unique for a genome.

When investigating the least number of flips, all branches are pruned at the level found by the Flip Sorter. We extended the DFS with an archive. This archive keeps track of all sequences it encounters and the level on which it encounters this sequence. If the algorithm finds a sequence which is already in the archive on a level lower or equal to the level where it encounters the sequence at this point, the branch is pruned.

**Breadth First Search**

Like DFS-algorithms, Breadt First Search (BFS) algorithms are also constructive and work by traversing down a tree. Unlike DFS-algorithms, it explores all neighbour nodes of the tree root before moving to a deeper level in the tree thus using a queue instead of a stack, which is the case for DFS. Obviously, it first explores the ‘breadth’ before going ‘deep’, if the computer would have enough power to overcome the state space this would always lead to the solution with the least possible number of mutations. Therefore, this algorithm does not require any pruning as it only goes deeper when it has not found the solution on the previous level. The BFS algorithm used for this case also works with an archive and keeps track of the mutations to make sure we are not constantly performing the same mutation on a genome.

**Best First Search**

We extended the BFS algorithm with a priority queue. Instead of sorting the queue based on which genome was created last, the queue is sorted based on a costfunction. This results in a queue in which relatively well sorted genomes move to the front, thus being treated first. How well a genome is sorted is calculated by cost functions, which are described in the next section. The archive of this algorithm keeps track of the genomes as well as the level on which a genome was found and prunes when a genome was already found on the same or lower level. We also prune when the depth or mutation score found by the flip sorter is reached.

When trying to find a mutation sequence leading to low mutation scores, we combined the cost functions looking at how well genomes are sorted with the mutation score. These two scores are weighed and scaled to result in a priority which will lead to a solution but also takes the mutation score into account. When only focussing on the mutation score we will never reach a solution because genomes on which a low number of (short) mutations have been performed will always have a higher priority than genomes which are closer to the solution. Using the average mutation score will also not help, because when doing this, at some point the length of a mutation does not make a difference anymore. Resulting in solutions in which very short mutations are performed to begin with followed by long mutations later.

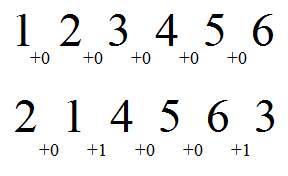
**Pruning**

To decrease memory load on the computer, genomes with a worse than average cost were removed from the priority queue, if the priority queue became bigger than 10,000 genomes. The average cost was determined per level in the search tree (depth level), otherwise all genomes at the beginning of the search tree would be removed, since cost decreases after mutations. If all genomes of low depth levels would be removed, the algorithm would not be able to search a large part of the search tree after finding multiple solutions. Because pruning everything above average results in a lower average, and thus more pruning next time, a linear function was added upon the average: by subtracting 0.01\*depth level + 0.125, from the average per level, the best genomes from low depth levels were preserved regardless of how many times pruning was done, while more than halve of the genomes from the higher search levels were removed.

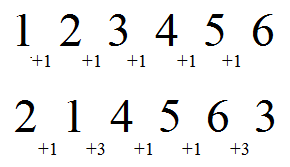
After a solutions was found, all genomes of depth levels close to the solutions level were deleted, this forced the algorithm to find a more different route. If this was not implemented, many very similar solutions were found.

**Cost Functions**

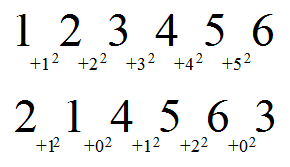
We used a variety of cost functions to decide on priority. All cost functions are described here:

**Cost Function 1**

This function checks if a gene is situated next to the gene it should be, if so no costs are added, if not the costs are always +1. It does not matter if the genes are in the good order, so 1-2 and 2-1 are both in good order.

**Cost Function 2 and 3**

This function adds the difference between two adjacent genes to the score. Function 3 works like function 2, but the differences between two adjacent genes are added with an exponent of 2. Function 2 is depicted.

**Cost Function 4, 5 and 6**

This function checks how many genes are sorted in row and gives the total. To make better sorted genomes a high priority, which is equal to a low score, the maximum possible score is divided by the current score to get the priority. Function 5 and 6 work exactly like function 4, but instead of only adding the number of sorted genes in a row, the number of sorted genes in a row are added with an exponent of 2 and 3 respectively. Function 5 (exponent is 2) is depicted.

**Padding**

To improve performance padding around the genome sequence is added when calculating the cost. Around a sequence of length 25 a 0 will be placed in front of the genome, and a 26 is placed at the end. This will cause the 1 and 25 to be positioned where they should be earlier in the sorting process, and it also prevents the genome from being sorted the wrong way around.

**Weighing**

To find the right balance between the mutation score and cost function we will do experimental test runs, adjusting the weight of the mutation score based on previous results. The cost of the mutation should be smaller than the possible improvement when improving the genome sequence. Otherwise short mutations are considered more important than getting a solution.

**Data from 100 runs**

To be able to draw solid conclusions, the Best First Search algorithm with the cost functions and weighing yielding the best results for the specific *Drosophila* case, will be used on 100 random genomes of length 25. These algorithms will run for 60 seconds on each randomly shuffled genome with a maximum of 15 solutions. The best outcomes for all three criteria and the begin cost will be used for further analysis. As a null-hypothesis, these genomes will also be sorted by the Flip Sorter.

Furthermore, we will also analyse the performance of the different algorithms by timing the duration until the algorithm finds the first solution, for random genome sequences of different lengths.

**results**

**Flip Sorter**

The Flip Sorter sorted the genome of *D. Miranda* in 18 flips, with the total sum of *n* being 147 and ½*n*2 being 963.5. These values have been used as the upper bounds for further algorithms. Figure 4a. shows this solution.

**Depth First Search and Breadth First Search**

The Depth First Search algorithms, without archive as well as with an archive, were not able to solve the 25-gene genome due to the size of the state space. Genomes until length 9 could be solved, but when the genomes grew, memory shortages forced the algorithms to stop. The same accounted for our Breadth First Search algorithms, which could solve genomes up until length 10. Figure 2 shows the time it took for these algorithms to find a solution. Outliners became more extreme when genome length increased. During the run with genome length 8 for example, almost all solutions took around a second to find, except one of the genomes, which took 90 seconds.

**Best First Search**

The Best First Search algorithm with cost function 1, no mutation function, could find a solution up to genomes with a length 200, which took 7 minutes to find. (longer not tested). During the run to solve the genome of length 200 the memory consumption almost became problematic, with less long sequences memory was not an issue.

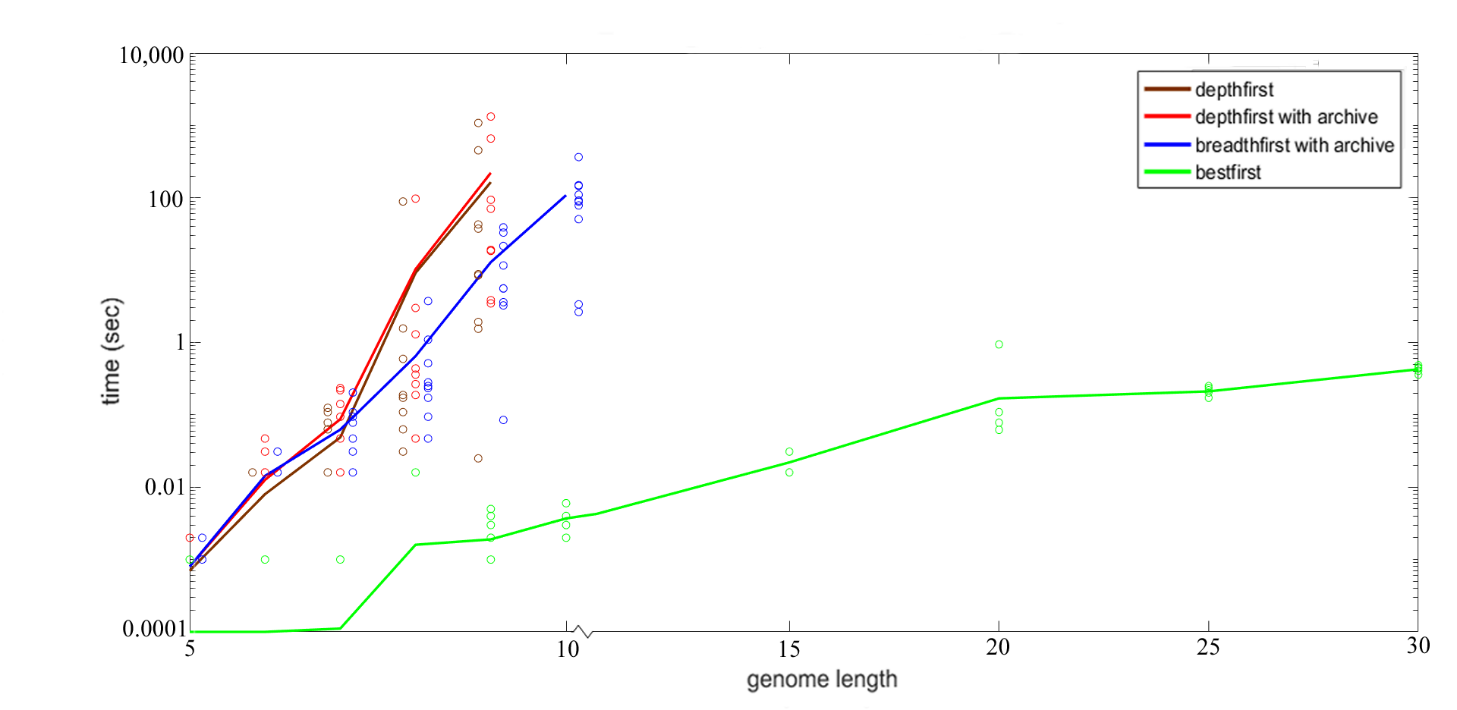
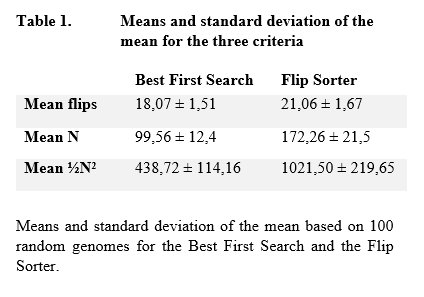


Fig 2: **Time (sec) to find the first solution with different algorithms for different genome lengths, ranging from 5 to 30.** The average time (sec) per algorithm is shown and all ten values of the sample are represented by the bubbles. Depth First is shown in black, Depth First with Archive in red, Breadth First with Archive in blue and Best First with cost function 1 in green.

**Least number of Flips**

The mutation sequence consisting of the least number of flips was found through the Best First Search algorithm using cost function 1 and mutation function 3. The number of flips was 13, the sum of N was 89 and ½N2was 489.5. Figure 4b. shows this mutation sequence.

**Lowest N**

The lowest sum of N was found in a different solution with the same parameters as the mutation sequence with the least number of flips (Best First Search with cost function 1 and mutation function 3). The number of flips was 14, the sum of N was 70 and ½N2was 315. Figure 4c. shows this mutation sequence.

**Lowest ½N2**

The lowest mutation score for ½N2 was not found by the cost function designed for this problem but in the same way as the two other best solutions. The number of flips was 14, the sum of N was 70 and ½N2was 315. Figure 4c. shows this mutation sequence.

**100 random genomes**

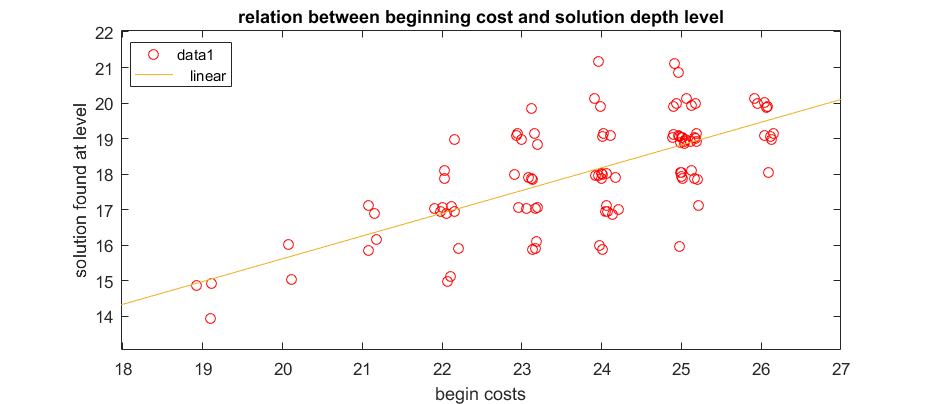
The 100-genome-test set showed a positive correlation between begin cost (by cost function 1) and the number of flips needed to put the genome in order (figure 3), implying that genomes with a low costs are ‘easier’ to sort. This relation was not found as strong for the other two measurements (N and ½N2). Further, the Best First Search algorithm found a better solution depth in 94%, an equal depth in 4% and a worse solution depth in 2% of the instances compared to the flip sorter. For both mutation scores the Best First Search algorithms found better solutions in 100% of the instances. Table 1 shows the mean and standard deviation of the mean for the 100 random genomes for the three criteria. The Best First Search algorithm found the best solution for the two mutation scores through the same mutation sequence for 73% of the genomes.

Fig 3: **Relation between begin costs and solution depth level found by the Best First Search.** All data points of the begin costs calculated by cost function 1 are plotted against the depth level on which the Best First Search found a solution. The data points are plotted with a linear fit presenting the positive correlation.

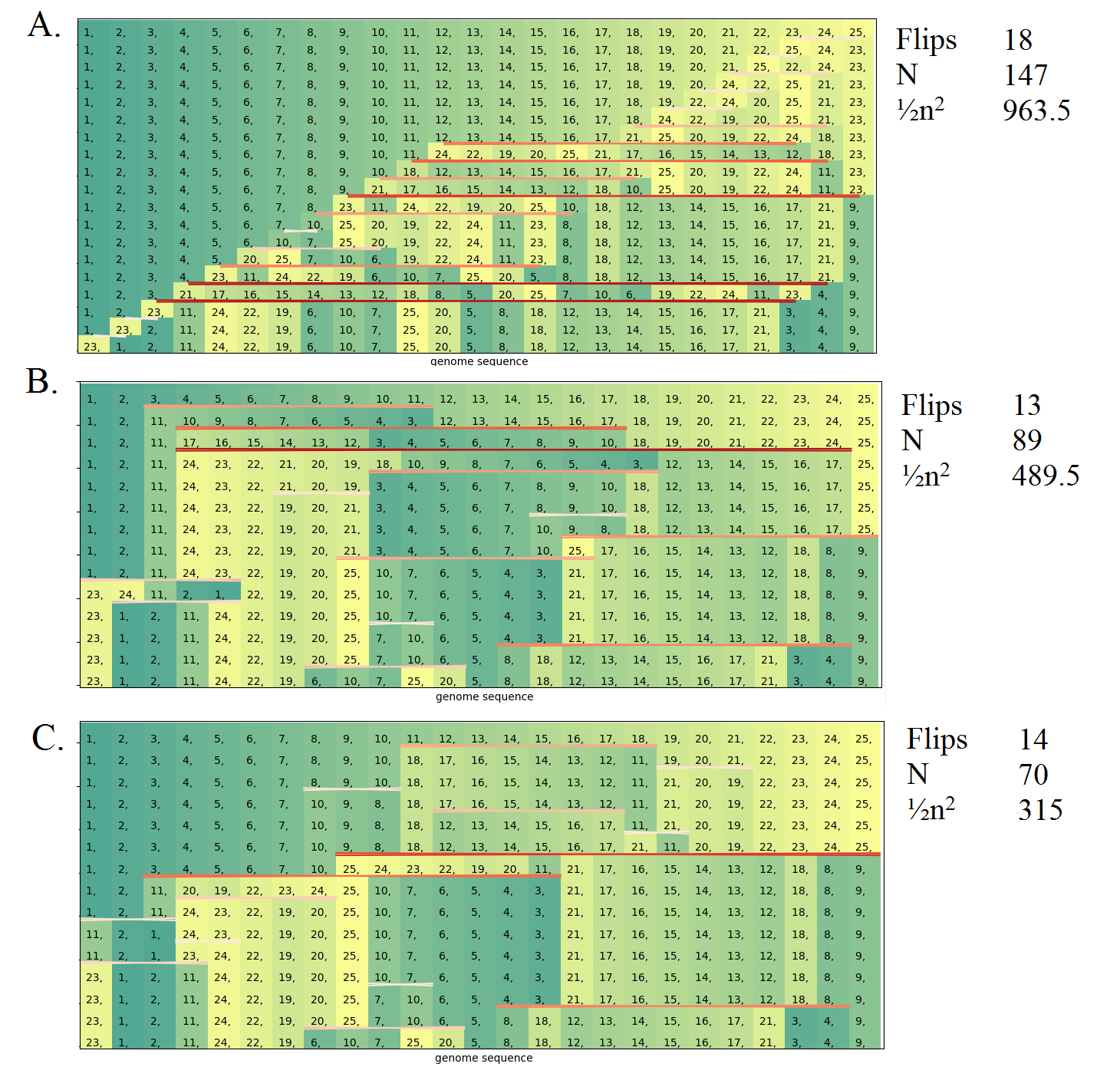


Fig 4a-c: **Mutation sequences to sort the genome of *D. miranda* to *D. melanogaster.*** 4a. shows how the flip sorter sorts the genome. 4b. and 4c. show the shortest mutation sequence and the lowest mutation scores respectively. Sorting goes from the bottom to the top, flip lengths are represented by colour shades; the darker the colour the longer the flip.

**Conclusion & discussion**

A flip is the most efficient when two genes are put at the right place by performing the flip, so the solution with the least number of flips has the highest number of efficient flips. By increasing the number of efficient flips, we lower the number of total flips, e.g.: the flip sorter only uses one efficient flip (the last one, which is always efficient) while the solution with the least number of flips uses six efficient flips. When in a specific instance it is not possible to perform an efficient flip, it is most efficient to perform a flip which facilitates an efficient flip later. Even though the number of flips found by the Best First Search are significantly lower than the number of flips found by the flip sorter, we cannot state that this is the shortest mutation sequence because we have not run through the entire state space. A possible method to find that 13 flips is the shortest way to solve the case, is by running the sequence in the depth first algorithm and prune at level 12. If there’s no solution 13 flips is indeed the shortest mutation sequence. Unfortunately, as shown in figure 2, we lack the computer power and memory to run this programme. In addition, the presented sequence of mutations is only a single solution out of a series of solutions with a mutation sequence length of 13. When considering the two mutation scores we cannot be sure that we have found the best solution, due to the reasons presented above. The higher variability in mutation score outcome make it even harder to draw a conclusion concerning these questions.

Surprisingly, all solutions with the best outcome for our problems were generated by the same algorithm and cost functions. This underlines how closely related the solutions are and how the factors (number of flips and length of flip) interact to get to optimal solutions. The two mutation scores are even closer related, in 73% of the cases from the 100 random genomes, the best outcome for these two criteria were yielded from the exact same mutation sequence. Furthermore, it also tells us how sensitive the algorithm is to small changes such as adjusting the weighing of the cost functions, e.g. the algorithm could fail in finding a single solution after a minimum change in weighing. To be able to find better cost functions for this specific case, and permutation sorting problems in general, more theoretical studies have to be performed. Theoretical evidence could also help in proving if the found mutation sequences are the best solutions to the problems. Even though we cannot conclude that our algorithms are able to find the best solutions, based on our comparison with the flip sorter we can conclude that we have at least found a way to find better solutions for all three criteria.

The 100-genome test set also showed us what genomes are relatively easy to sort (meaning in a relative low number of flips), namely genomes with a low begin cost as calculated by cost function 1. Cost function 1 counts the number of genes not adjacent to the gene it should be adjacent to, resulting in a reliable measurement for difficulty. The genome of interest in our specific case has a relatively low begin cost of 17, thus making it an easier genome to sort than most random genomes. Based on our result we can conclude that genomes with a start cost of 17 can be sorted in 13 flips.

Looking at the time it takes different algorithms to solve several genomes of increasing length, we can conclude that only the Best First Search is able to solve genomes from over a length of 10. The fact that the BFS algorithm can solve longer genomes than the DFS algorithm lies in how these searching algorithms explore the state space and how solutions to the problem are spread in depth and breadth. By only performing random mutations on a genome, there is a very low chance to eventually reach the desired genome, thus leading to high likeliness of branching growing to the maximum depth and thus demanding more from the computers’ memory than using a breadth first which reaches the solution faster. Furthermore, the spreading of the data shows us that for the DFS and BFS algorithms several genomes are harder to solve than average, due to their ‘position’ in the state space, while the data for the Best Fist Search is less spread. This is caused by the high increase in efficiency in going through the state space by prioritizing with a cost function.

Our study has no direct implications in relation to the evolutionary pathways in the *Drosophila* genus. The mutation sequences we have found through our algorithms might be the evolutionary pathway between the two species of interest, but due to the long term processes and the role of chance underlying evolutionary change, these type of evolutionary studies can never be sure about the exact sequence of mutations leading to the variation found in nature but only give an estimate about relatedness between species. The algorithms we have used do not allow any deterioration of the sequence, a gene only gets placed next to the one it needs to be in the end, while it is likely it would have moved more than once before being positioned where it is in *D. melanogaster.*