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

LEANDER DE KRAKER & nINA CIALDELLA – HEURISTICS 2017

**absttract**

**Titel misschien veranderen misschien permutation sorting erin? Als laatste doen . what are permutation sorting problems hierin opnemen ?? dit is onzin maar dan weet ik hoe lang het ongeveer gaat worden en eruit gaat zien. We extended the BFS algorithm with a priority queue. Instead of sorting the queue based on which genome was created last, the queue is ordered based on how well the genomes are sorted. 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.**

**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*.*

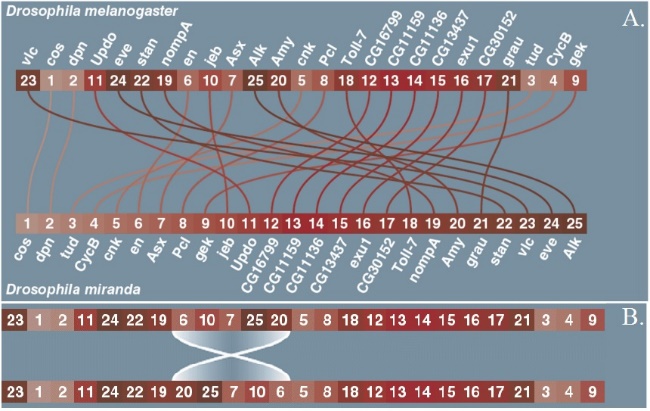
Species in a genus often show high genetic resemblance, as is the case for this study as well. Genetic research has shown that the genomes of *D. melanogaster* and *D. miranda* consist of 25 identical genes but in a different order. Therefore, we will simplify our model into a series of 25 numbers in a different order, 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 5, which is how changes occur in the algorithms.

The 25 genes of the *Drosophila* genome can be put in any random order. The first place can have 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.When the genome has changed twice there is only a one in 90 000 chance the second change reverses the first change.

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. Uitwijden over permutation sorting and bekende dingen ??? literatuur zoeken….

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.

**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 firstly 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**

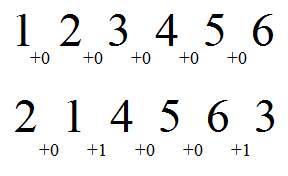
We extended the BFS algorithm with a priority queue. Instead of sorting the queue based on which genome was created last, the queue is ordered based on how well the genomes are sorted. This results in a queue in which relatively well sorted genomes move to the front, thus being treated first. How well 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 not doing this and 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. (misschien is dit discussie of conclusie weet niet of dit hier hoort).

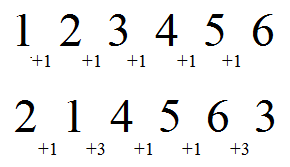
**Waaarom moesten we ook al weer een heap gebruiken in onze implementatie???vergeten ineens. Prunen bij bepaalde grootte van queue dit staat hier nog niet in. Sws een heleboel dingetjes over prunen er nog in.**

**Cost Functions**

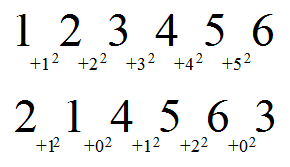
We used a variety of cost functions to decide on priority. To improve the priority given by the cost functions, we added a 0 to the beginning of the sequence and a 26 to the end of the sequence, by doing this we also prevent that the genome is being sorted the wrong way around. All cost functions are 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 (!!! 3 is 5 in cost.py)**

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 (!!!!4 is Functie 3 in cost.py, 5 = 4 wilde niet verwarren)**

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 (the exponent is 2) is depicted.

**Mutation Scores & Weighing**

Weet niet zo goed wat ik over weighing moet opschrijven omdat het vooral gwn random was…

**results**

**al die plaatjes van de goeie en dan nog een beetje tekst en het getimede gedeelte en de uitslag van de 100-random genomen.**

**Flip Sorter**

The Flip Sorter sorted the genome 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 2 shows this solution.

Plaatje.

**DFS & BFS**

**discussion**

………………………………………………………….. conclusion, are our solutions the best or just approaching it?. Broader conclusion on 100 random genomes > permutation sorting. Critique on how we solved it. alternatives way of solving