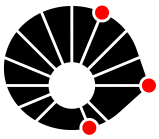

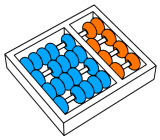

Using Reinforcement Learning on Genome Rearrangement Problems

Victor de Araujo Velloso *Andre Rodrigues Oliveira*
Zanoni Dias

Relatório Técnico - IC-PFG-18-19
Projeto Final de Graduação
2018 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Using Reinforcement Learning on Genome Rearrangement Problems

Victor de Araujo Velloso* Andre Rodrigues Oliveira† Zanoni Dias†

Abstract

Genome Rearrangement Problems compare different genomes considering events of mutations affecting a large segment of their DNA structure. Some examples of mutations are deletions, insertions, reversals, and transpositions. To make this comparison, genomes are considered as a group of blocks, formed by one or more genes, that are conserved between the genomes compared. By representing these blocks as numbers, and considering that there are no repeated blocks, we obtain permutations, and the distance between two genomes is the minimum number of rearrangements events required to transform one permutation into the other. Depending on the rearrangement events considered this problem becomes $\mathcal{NP}$-hard, so finding approximation algorithms with a low factor to solve this problem is of interest. In this work, we estimate the distance in permutations of length 10 and 15 using different Reinforcement Learning algorithms. The rearrangement events considered here were reversals and transpositions. We compare the performance and the results of each method with either the exact distance or the distance estimated by other approximation algorithms in the literature.

1 Introduction

One way to infer evolutionary relationships between different species is by comparing and finding similarities between the genomes of these species. Comparisons can be made both on low levels, by looking at punctual differences in the nucleotides of the DNA sequence of each genome, as well as in a higher scale, looking at differences on the segments of the DNA.

Genome rearrangements are mutation events that affect large portions of the DNA. These events are called *conservative* if there is no insertions and deletions to the DNA, and *non-conservative* otherwise. The *Parsimony Principle* says that we may choose the simplest scientific explanation that fits the evidence, so in genome comparison problems we always seek for the minimum number of operations that transforms the first into the second. A *model* defines the set of genome rearrangements allowed to be performed over genomes.

If a model allows only conservative events, we assume that the two genomes compared share the same set of genes and that there are no repeated genes. By representing one of them as an ordered sequence of numbers representing each gene, we only need to *sort* the second considering the genome rearrangements allowed by the model. Reversals and transpositions are the most studied conservative genome rearrangements. A *reversal* occurs when a segment of the genome is reversed. A *transposition* occurs when two consecutive segments exchange places.

Sorting by Reversals (**SbR**) is an $\mathcal{NP}$-hard problem [3], and the best known algorithm factor has an approximation factor of 1.375 [1]. *Sorting by Transpositions* (**SbT**) is also $\mathcal{NP}$-hard [2],

*v148131@dac.unicamp.br

†{andrero,zanoni}@ic.unicamp.br

and the best known algorithm has an approximation factor of 1.375 [5]. *Sorting by Reversals and Transpositions* (**SbRT**), on the other hand, has unknown complexity, and the best approximation algorithm has a factor of $2k$ [12], where k is the factor of an algorithm used in the cycle decomposition. The best k known is $1.4167 + \epsilon$ for any positive ϵ [9], so the approximation factor of **SbRT** becomes $2.8334 + \epsilon$.

Reinforcement learning refers to goal-oriented algorithms that after many steps learn how to attain a complex objective (goal) [10]. These algorithms may lack any prior knowledge at the start, but under the right conditions, they achieve superhuman performance. The *reinforcement* term is because these algorithms are penalized after performing wrong decisions and rewarded after making the right ones.

In this work, we explore the use of reinforcement learning on the Sorting by Reversals and Transpositions problem. We started by finding solutions using permutations of size ten, comparing with the optimal solution known. Then, we increased the size of permutations to 15 and compared the results with existent approximation algorithms in the literature. This report is organized as follows. Section 2 formulates the Sorting by Reversals and Transpositions problem. Section 3 details the concept behind Reinforcement Learning algorithms. Sections 4 and 5 explain two methods of Reinforcement Learning we will use. Section 6 describes our experiment. Section 7 shows the results we obtained. Section 8 concludes this report.

2 Problem Specification

In Genome Rearrangements Problems we represent genomes as a sequence of conserved blocks. By considering that there is no repetition, this sequence becomes a permutation. Then, a genome with n conserved blocks is represented by $\pi = (\pi_1\ \pi_2\ \ldots\ \pi_{n-1}\ \pi_n)$, with $1 \leq \pi_i \leq n$ for any $i \in [1..n]$ and $\pi_i = \pi_j$ if, and only if $i = j$.

A reversal $\rho(i, j)$, with $1 \leq i < j \leq n$, is an operation that reverts the segment $\pi[i..j]$, i.e., the segment going from the element at position i to the element at position j, as follows: $\pi \cdot \rho(i, j) = (\pi_1\ \pi_2\ \ldots\ \underline{\pi_j\ \pi_{j-1}\ \ldots\ \pi_{i+1}\ \pi_i}\ \ldots\ \pi_n)$. A transposition $\rho(i, j, k)$, with $1 \leq i < j < k \leq n + 1$, switches the position of segments $\pi[i..j-1]$ and $\pi[j..k-1]$, as follows: $\pi \cdot \rho(i, j, k) = (\pi_1\ \pi_2\ \ldots\ \pi_{i-1}\ \underline{\pi_j\ \pi_{j-1}\ \ldots\ \pi_{k-1}}\ \ \underline{\pi_i\ \pi_{i+1}\ \ldots\ \pi_{j-1}}\ \pi_{k+1}\ \ldots\ \pi_n)$.

The *rearrangement distance* between two genomes with n conserved blocks represented by permutations π and σ, denoted by $d(\pi, \sigma)$, is the size of a minimum-length sequence $S = \{\gamma_1, \gamma_2, \ldots, \gamma_n\}$ of allowed rearrangements such that $\pi \cdot \gamma_1 \cdot \gamma_2 \ldots \gamma_r = \sigma$ (in this case, $d(\pi, \sigma) = r$). Let σ^{-1} be the inverse permutation of σ. The rearrangement distance problem between permutations π and σ can be transformed into the sorting distance problem of finding a minimum-length sequence of rearrangements that transforms the permutation $\alpha = \pi \cdot \sigma^{-1}$ into the *sorted permutation* $\iota = (1\ 2\ \ldots\ n)$, since $d(\pi, \sigma) = d(\pi \cdot \sigma^{-1}, \sigma \cdot \sigma^{-1}) = d(\alpha, \iota)$.

The *extended permutation* of π is the permutation π with two new elements: $\pi_0 = 0$ and $\pi_{n+1} = (n+1)$. Given an extended permutation π, a *breakpoint* is a pair of consecutive elements where $|\pi_{i+1} - \pi_i| \neq 1$, for $0 \leq i \leq n$. In other words, if a pair of elements is not formed by sequential numbers, either in an increasing or decreasing order, then this pair is a breakpoint. The number of breakpoints in a permutation π is denoted by $b(\pi)$. Note that $b(\iota) = 0$, so we can say that once all breakpoints of a permutation have been removed through genome rearrangements, we reached the sorted permutation. The maximum number of breakpoints removed by a reversal is 2, and by a transposition is 3.

3 Reinforcement Learning

Reinforcement Learning (RL) is an area of Machine Learning that solves problems by focusing on rewards you get out of actions [10]. The main idea is that from a particular state you are, you have different possible actions to take, and each action is associated to a reward. By taking an action, you will receive its reward and move to a new state. You will repeat this process until it reaches an end state. The focus of RL is attempting to find the policy that chooses the actions leading to the highest total reward.

Generally, an RL model is made up of two parts: the agent and the environment. The agent is the algorithm that learns about the relations of state, actions, and rewards. This agent chooses the actions based on states and updates its values based on the received rewards. On the other hand, the environment changes the states and gives rewards based on agent' actions.

The environment also defines how the states, rewards, and actions are represented. These representations can be discrete or continuous; however some agents can only interact with one type of representation, so the pair agent and environment must be able to work together.

The agent and the environment interact cyclically. At step t the agent chooses an action A_t, that it expects will give the best reward for the current state S_t. The environment receives A_t, finds the new state S_{t+1} and gives the reward R_t for the received action A_t; the agent receives the new state S_{t+1} and the reward R_t, and updates its prediction. This process is repeated until the agent receives an ending state, or a step limit is reached.

A completed process is an *episode*. Multiple episodes are required for an agent to learn the best possible actions in an environment. The agent improves its prediction not only after each step of an episode but also through every episode, by finding better total rewards. Episodes are played until the total reward found converged into a satisfactory result or until an episode limit is reached.

The focus of RL is to find the agent policy θ that returns the highest total reward for a given environment. A policy θ is said to be the best when its expected total reward is better or equal to every other policy in the environment, and it is called the optimal policy θ_*.

For a problem at a state t, the sum of discounted rewards received after the state t is expressed as:

$$G_\theta(t) = \sum_{k=0}^{\infty} \gamma^k R_{t+k}, \tag{1}$$

where $\gamma \in [0, 1]$ is a discount value used to keep closer rewards more important than distant ones.

One issue with RL is the relation between exploration and exploitation: how can the agent learn enough about the environment while still taking the known good actions to improve the total reward. Exploration is when the agent takes unknown actions to learn more about the environment, this can lead to the agent finding better actions it can take on certain states. Exploitation is when the agent takes only the best-expected actions to find the best possible reward. The challenge is that by focusing entirely on one or the other will not allow the agent to achieve its goal, since only exploring may lead to the agent only learning about bad actions, and only exploiting will not allow the agent to discover better actions existent in the environment. There are some ways to deal with this issue, and each algorithm detailed on the next sessions will have one method for it.

Here we are going to work with two different RL algorithms, both using neural networks to estimate the best actions for each state. They are called Double Deep Q-Network [15] and Wolpertinger Policy [4].

4 Double Deep Q-Network

In Dynamic Programming RL methods, the agent learns the environment by exploring all the actions of every space and storing the rewards found. This method works well for small states and action spaces, but as spaces grow in size, the memory required to store the values becomes too large to be feasible to use them. For such spaces, an approximation function of state-action pairs is required. Q-learning [17] is one of the many methods that find an approximation function.

For the approximation problem we have:

$$Q(s, a) = \mathbb{E}[G(t)|S_t = s, A_t = a] \tag{2}$$

where $\mathbb{E}$ is the expected value at step t of $G(t)$ when the state is s and action is a. This is known as the Q-table, and we can find the best policy by maximizing Q or, in other words, by choosing the action that gives the highest expected value on each state.

Q-learning uses the equation (2) to choose an action at each step. However, to improve the approximation it updates the parameters in Q to converge the expectation to a better result. After taking an action A_t in the state S_t, receiving the reward R_{t+1} and moving to the state S_{t+1}, the parameter update is performed as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{3}$$

While the approximation function of Q-learning helps to deal with memory issues of dynamic programming, the time required to train larger environments is impracticable.

Deep Q-Network (DQN) is an implementation of a Q-learning [6] algorithm using neural networks to find the approximation function (2). The neural network makes the function approximation of Q-learning much more feasible for larger environment states.

The DQN uses a second network, called target network, to train the parameters of its main network. The target network has the same models as the main neural network and features the same starting weights. However, the target network does not get updated every step alongside the main network. Instead, it gets updated by copying the weights of the main network, only after a complete episode.

In the regular DQN, the target network is used to update the main model as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q\text{'}(S_{t+1}, a) - Q(S_t, A_t)], \tag{4}$$

where Q' is the target network.

The DQN uses experience replay training instead of training the network with the values received after each step. It stores the values in a batch and trains the network with mini batches of random samples (S_t, A_t, S_{t+1}, R_t) gotten from the batch. The use of a random distribution of samples can improve the results because values received in a sequence may feature some unwanted patterns that can be added to the network weights.

DQN deals with the exploitation and exploration problem by using a common approach called epsilon-greedy. The epsilon-greedy approach uses a variable ϵ with starting value $\epsilon \in [0, 1]$, and it picks a random $x \in (0, 1)$. If $x > \epsilon$ it selects the best-known action, and it chooses a random action otherwise. The ϵ starts with a high value, and it is decreased based on a decay parameter after each step. The motivation behind this decay is that during early episodes there are still a large number of unknown actions, but as more actions become known, it becomes better exploring its rewards and find even better total rewards.

While DQN has achieved some impressive results in practice, it can lead to overestimated values [6] because it uses the same values to select (2) and evaluate (4) an action. To avoid this problem we use the target network Q' to evaluate the actions, while still using the main network to select the action:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q'(S_{t+1}, a)) - Q(S_t, A_t)]. \quad (5)$$

This method is the Double Deep Q-Network (DDQN) [15] and it is a method we used in the Sorting by Rearrangements problem.

The way DDQN works in conjunction with the environment is as follows. It receives a state S_t from the environment. Then the neural network outputs the expected value for every action in the current state. The action with the highest expected reward A_t is sent to the environment, which returns the reward R_t for this action and a new state S_{t+1}. Values S_t, A_t, S_{t+1} and R_t are stored in the experience replay memory, and the cycle continues with the DDQN finding the action for S_{t+1}. Once the replay memory has more than a set number of samples stored, the DDQN is updated after every step t.

5 Wolpertinger Policy

Another method used was the Wolpertinger Policy (WP), built upon the Deep Deterministic Policy Gradient (DDPG) [8], which implements the actor-critic framework [13] in a deep neural network model. DDPG is a sequence of two networks, the actor-network $\mu(s)$, which maps states to actions based on the current policy, and the critic network $Q_c(s, a)$, that returns the expected value of a given action based on the current state. They work in a sequence where the actor-network finds an action, the critic network evaluates that action, then the critic network updates its estimations based on the reward gotten, and the actor-network is updated based on the critic network weights.

The DDPG is focused on solving problems with continuous actions spaces, and the value returned by the actor-network $\mu(s)$ is continuous. However, in the Sorting by Rearrangements problem the action space is discrete. We could use the action with the closest value of the continuous action found, but that would give us poor results since in many cases actions with low estimations can be close to the continuous action. The WP changes the DDPG algorithm to make it work with discrete actions spaces while also improving the learning for large discrete action spaces.

In most Q-Learning methods the policy is finding the action a that maximizes the Q-table (2). For the best action be found this policy requires every action to be estimated but estimating all actions in large action spaces can become quite expensive. The WP policy avoids this cost by using the actor-critic framework alongside a k-nearest-neighbor algorithm, allowing fewer actions to be evaluated.

In WP the actor-network outputs a proto action $\hat{a}$ after receiving a state s as input. This proto action $\hat{a}$ is not a working action for the environment, but it has a continuous value that approximates a discrete action value. To actually use this action, the WP uses a k-nearest-neighbor search on the proto action $\hat{a}$, and finds a group A of the closest discrete actions, where $|A| = k$. Actions $a \in A$ are then refined through the critic network: each action a is given as input alongside with the current state s, and the critic network returns an estimated value for that action in that state. The action with the best-estimated value is then chosen. By doing this WP decreases the required number of actions to be estimated from the size of the action space to the value of the parameter k.

The critic network $Q_c(s, a)$ is updated in the same way as the DQN (4). The actor-network is updated by using the policy gradient [13]:

$$\nabla_{\theta^\mu} G \approx \mathbb{E}[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=S_t, a=\mu(S_t)}] = \mathbb{E}[\nabla_a Q(s, a|\theta^Q)|_{s=S_t, a=\mu(S_t)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s=s_t}], \quad (6)$$

where θ^μ is the policy for the actor network, and θ^Q is the policy for the critic network.

Similarly to DDQN, the WP also features experience replay training and target network. In this case, there are two target networks, one for the actor and another for the critic. While a target network in DDQN is updated by simply copying the weights of the main network, a target network in the WP is updated through soft updates, as $\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$, with $\tau \ll 1$.

Concerning the exploration and exploitation problem, WP adds some noise value N to the proto action $\hat{a}$. This allows exploration since some different actions could be chosen by the k-nearest-neighbor search.

6 Experiment

We built an environment for the genome rearrangement problem featuring reversals and transpositions. The states of the environment are all permutations π, and the identity permutation $\iota = (1\ 2\ \ldots\ n)$ is the end state. The actions are all reversals and transpositions that can be applied over the state permutation. Since the goal is to have the lowest amount of actions until it reaches the end state, the value chosen for the reward for any action taken was -1 - there is no discrimination for different actions.

In the DDQN model we represented actions as integer values, ranging from zero to the number of actions, where each integer is the index in a list composed by four values. The first value stores the type of rearrangement, and the other three values stores the positions (i, j, and k) used by this rearrangement - since reversals uses only positions i and j, the value of k for any reversal was set to -1. However, since WP uses the idea of "closest actions", we had to represent actions in a better way to allow the close actions to be not just the neighbors on the list. We then encoded the action list using a One Hot Encoding method, making every action to be represented as a binary vector. Concerning the states, instead of representing them in the network as permutations themselves we also used on the implementation of both algorithms the One Hot Encoding method.

We implemented the RL algorithms with Python using the TensorFlow[1] library for the neural networks and the gym[2] library to create the genome rearrangement environment.

Our implementation of the neural networks are shown in Table 1, for the DDQN, and in Table 2, for the WP. We used a densely-connected layer for every layer in the networks. The optimizer used was Adam [7], with a learning rate of 10^{-3} on the DDQN and the critic network, and 10^{-4} on the actor-network. To generate the actor noise required for the exploration in the WP an Ornstein-Uhlenbeck process [14] was used.

We build two different methods to improve the convergence time, and to allow bigger permutations to reach the end state. The first method is a pre-training on the networks, where we gather a set number of random states and fit the weights of the networks based on the number of breakpoints removed by each possible action (genome rearrangement). This helps the network to start with decent parameter weights at the beginning of the training.

Two phenomenons occurred during the learning process of the network in our tests with the pre-training method. The first was that the weight parameters of the network obtained from the pre-training were quite different compared to the final weights parameters after a successful

[1] https://www.tensorflow.org/
[2] https://gym.openai.com/

Table 1: DDQN Layers

Layers	**Size**	**Activation**
Input	|State|	Linear
Hidden Layer 1	400	ReLu
Hidden Layer 2	300	ReLu
Output Layer	|Action|	Linear

Table 2: WP Layers

	Actor Network		**Critic Network**	
Layers	***Size***	***Activation***	***Size***	***Activation***
Input	|State|	Linear	|State|+|Action|	Linear
Hidden Layer 1	400	ReLu	400	ReLu
Hidden Layer 2	300	ReLu	300	ReLu
Output Layer	|Action|	Tanh	1	Linear

training. These parameter changes during the training could lead to a sequence of episodes having worse results than the ones using the pre-training weights before converging into better results.

The other phenomenon was that the results sometimes completely diverged during the sequence of "worse results" but this could be avoided after a high number of good episodes found with the pre-training weights stored in the experience replay. Then we had the idea of the second method that helps with the convergence: memory fill, a technique that adds the result of good episodes to the experience replay memory.

Before starting the training, the memory fill method iterates through some episodes choosing either a greedy action, where it takes the action that removes the highest amount of breakpoints in the current state, or a random action. After that, it saves the result in the replay memory that is used in the first few training steps of the RL algorithm.

7 Results

We trained both the DDQN and the WP implementation on environments with permutations of size $n = 10$ and $n = 15$. We also tried to train the WP with different values for the k parameter used in the k-nearest-neighbour. Here the k value is represented by a percentage value, meaning the percentage of the total action space chosen for the parameter. All RL algorithms training was performed in the same machine, with a CPU Inter Core i5-4690 3.50GHz.

Our first test was with permutations of size $n = 10$. Figure 1 shows the convergences during the training of the DDQN and the WP with different values of k. It shows the difference depending on the value of k used on the WP, and we can see that WP converges in fewer episodes. The DDQN took more than 600 episodes to converge and even after that the distance found was still too high. By setting the value of k as 10% the convergence decreases compared to the higher values of k tested, but it still took much less episodes than the DDQN. For values of k as 50%, 80%, and 100% the convergence took almost the same amount of episodes.

We measured the time each method took to go through 1000 episodes, and they are as follows:

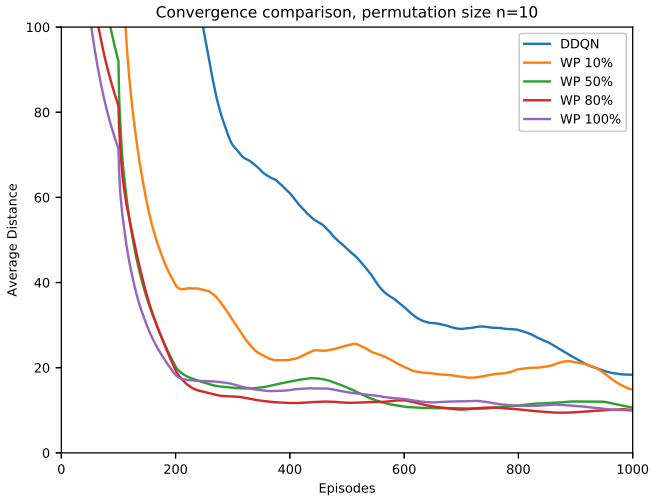

Figure 1: Distance convergence during the training of the DDQN algorithm and the WP with different k values, for permutations of size $n = 10$. This training used the memory fill method to improve convergence for all algorithms.

DDQN took 473.81 seconds, WP with $k = 10\%$ took 405.42 seconds, WP with $k = 50\%$ took 574.22 seconds, WP with $k = 80\%$ took 667.41 seconds, and WP with $k = 100\%$ took 806.01 seconds. This shows that, while WP requires fewer episode to converge, the time it takes to go through each step is higher than the time DDQN takes. It also shows that the lower the value of k, the less time each episode takes.

We also attempted to train WP with values of k as 5% and 1%. However, WP did not converge after 1000 training episodes. We concluded that while smaller values of k allow less training time, it will not allow the agent to learn enough information about the environment and converges to good results if this value is too small.

Although Figure 1 shows the convergence for the first 1000 episodes of training only, in the complete 10000 episodes training the DDQN converged to a good result, and the time difference between WP and DDQN considering 10000 episodes was even larger than the one shown on the first 1000 episodes. The WP with $k = 80\%$ took 3109.30 seconds to complete the training, and the DDQN took only 973.24 seconds, so the time each algorithm takes to go through a step becomes quite significant after the complete training.

We used the trained weights of both the WP with $k = 80\%$ and the DDQN, and computed the distance for 10000 different permutations of size $n = 10$. We compared the results of both RL method against the exact sorting distance of each permutation, and Figure 2 shows this comparison. We can see that both RL methods found similar results. However, after the 10000 episodes the WP presented better results.

Both DDQN and WP went through the same amount of training episodes. While the former was shown to be faster, the fact that it had worse results with the trained weights lead us to conclude that WP is a better choice for genome rearrangement problems.

We also trained both RL methods with permutations of size $n = 15$, going through 10000 episodes of training and using both memory fill and pre-training methods to speed up convergence. DDQN did not converge in the 36971.19 seconds it took to train, concluding the training after 5000 episodes with no significant results. WP with $k = 80\%$ converged quickly and concluded the 10000

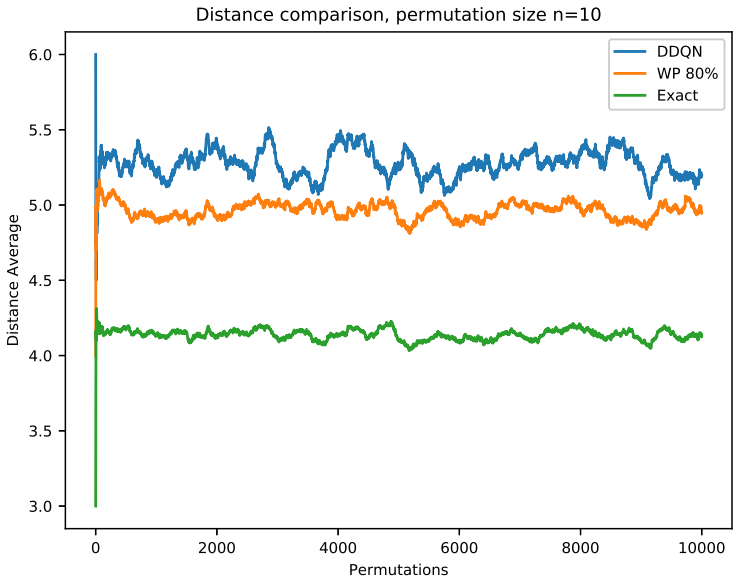

Figure 2: Average distance found with the DDQN, WP and the exact minimum distance, for permutations of size 10.

episode training in 10307.40 seconds, which shows that WP is better on dealing with higher action space values.

Figure 3 shows the training results of both RL algorithms with permutations of size $n = 15$, in the 5000 episodes the DDQN was able to go through. We can see that DDQN was getting very divergent distances, while the WP converged to good results in less than 1000 episodes.

After that we compared the trained WP with the following approximation algorithms for Sorting by Reversals and Transpositions:

- **Walter:** it is an approximation algorithm for **SbRT** developed by Walter et al. [16]. It has an approximation factor of 3.
- **Breakpoint:** it is a greedy algorithm for **SbRT** that tries to remove the maximum number of breakpoints at each step, and its approximation factor is 3.
- **Rahman:** It is an approximation algorithm for Sorting by Signed Reversals and Transpositions developed by Rahman et al. [12] with an approximation factor of 2. Since we are not working with signed permutations, we used the Lin and Jiang algorithm [9] on the cycle decomposition, so its approximation factor here is 2.8386.
- **DoD:** It is an approximation algorithm for **SbRT** developed by Oliveira et al. [11], with an approximation factor of $2k$. We also used the Lin and Jiang algorithm [9] on the cycle decomposition, so its approximation factor is 2.8386.

We used a total of 15000 permutations for the comparison, created by randomly performing reversals and transpositions on the identity permutations, starting with two operations (one reversal and one transposition) and ending with 30 operations (15 reversals and 15 transpositions). Permutations were separated by the number of operations applied with 1000 for each of them, and we found the distance for them using the algorithms described above.

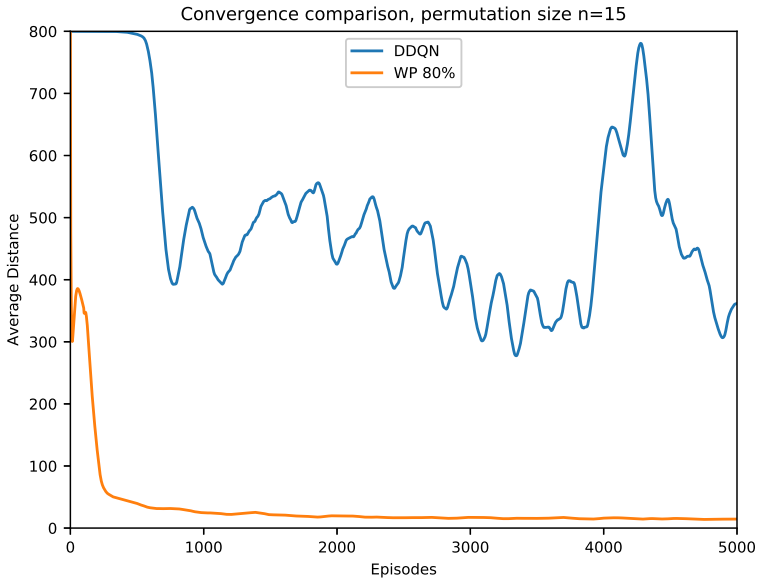

Figure 3: The complete convergence of the DDQN algorithm and the WP with $k = 80\%$, for permutations of size $n = 15$. This training used both memory fill and pre-training to help convergence.

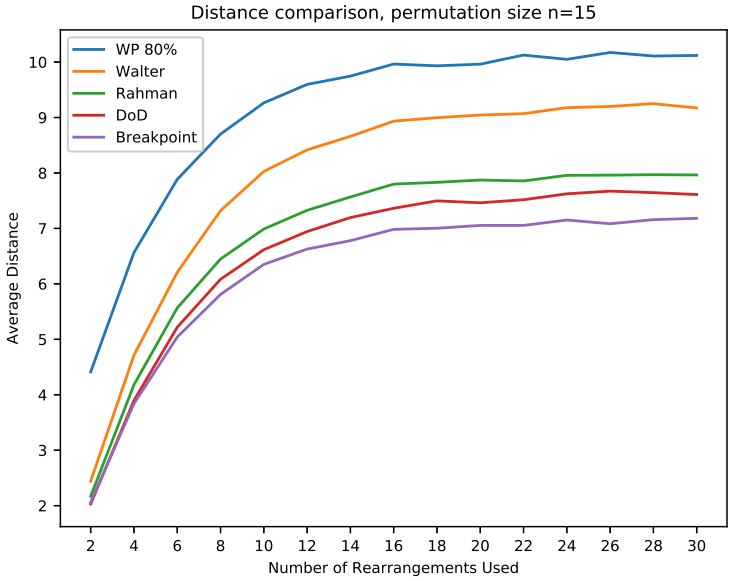

Figure 4: Comparison of WP results with the four other approximation algorithms. The x scale represents the number of random rearrangement operations the identity went through to generate the data. The y scales shows the average distance found by each algorithm.

Table 3: Comparison of the WP with four other genome rearrangement algorithms, over 15000 permutations of size $n = 15$.

Algorithm	$d_{WP} < d$	$d_{WP} = d$	$d_{WP} > d$	Avg d_{WP}/d
Walter	16.3%	21.1%	62.6%	1.215
DoD	2.1%	9.1%	88.8%	1.461
Breakpoint	0.7%	6.1%	93.2%	1.525
Rahman	5.3%	11.5%	83.2%	1.392

Figure 4 shows the obtained results. We can see that while WP found some good result on its own, it constantly found higher results for the majority of the permutations when compared to the other algorithms. However, we can notice that WP results had the same consistent difference when compared to the best algorithm. Meanwhile, other algorithms such as Walter got increasingly worse results as the number of rearrangements operations grows.

We also made a comparison between WP and the approximation algorithms permutation by permutation, presented in Table 3. It shows how many permutations WP returned a better result, equal results, and inferior results when compared to the other algorithms, and also shows the average distance factor for each of them. Something that we can notice is that even when compared to the best approximation algorithm, the trained WP algorithm was still able to find shorter distances for a few permutations.

An issue found was that for 0.1% of the permutations of length $n = 15$ tested, the WP was not able to reach the end state, which means that it did not return any value as the permutation distance. This means that either it requires more than 10000 episodes to learn all permutations, or it is not able to learn all of them. To overcome this issue we set a limit on the maximum number of operations WP algorithm could take: after $n - 1$ steps the algorithm stops. This limit of $n - 1$ was chosen because any permutation can be sorted with $n - 1$ operations using a variation of the Selection Sort algorithm.

We also attempted to train WP with permutations larger than 15, however, this was not possible. When we tested WP with 5000 episodes on permutations of size 20 the results did not even begin to converge.

8 Conclusion

In this work, we tested the use of Reinforcement Learning algorithms on the Sorting by Reversals and Transpositions problem. While RL algorithms return good results for small permutations, it gets harder to converge as the size of the permutations grows. For instance, the Double Deep Q-Network was not able to explore and learn using permutations of size greater 10.

The Wolpertinger Policy returned better results compared to Double Deep Q-Network. It got smaller distances on permutations of size 10 and also learned environments of permutations with sizes higher than 10, but it was not able to explore and learn using permutations of size greater than 15.

A natural extension of this work is to find a way to compute the distance with permutations of size greater than 15, since the possible actions on these permutations became too large for our algorithm to explore and learn.

References

[1] P. Berman, S. Hannenhalli, and M. Karpinski. 1.375-Approximation Algorithm for Sorting by Reversals. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA'2002)*, volume 2461 of *Lecture Notes in Computer Science*, pages 200–210. Springer-Verlag Berlin Heidelberg New York, 2002.

[2] L. Bulteau, G. Fertin, and I. Rusu. Sorting by Transpositions is Difficult. *SIAM Journal on Computing*, 26(3):1148–1180, 2012.

[3] A. Caprara. Sorting Permutations by Reversals and Eulerian Cycle Decompositions. *SIAM Journal on Discrete Mathematics*, 12(1):91–110, 1999.

[4] G. Dulac-Arnold, R. Evans, P. Sunehag, and B. Coppin. Reinforcement learning in large discrete action spaces. *CoRR*, abs/1512.07679, 2015.

[5] I. Elias and T. Hartman. A 1.375-Approximation Algorithm for Sorting by Transpositions. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 3(4):369–379, 2006.

[6] H. V. Hasselt. Double q-learning. In *Advances in Neural Information Processing Systems (NIPS'2010)*, volume 23, pages 2613–2621. Curran Associates, Inc., 2010.

[7] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[8] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

[9] G. Lin and T. Jiang. A Further Improved Approximation Algorithm for Breakpoint Graph Decomposition. *Journal of Combinatorial Optimization*, 8(2):183–194, 2004.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[11] A. R. Oliveira, U. Dias, and Z. Dias. On the sorting by reversals and transpositions problem. *Journal of Universal Computer Science*, 23(9):868–906, 2017.

[12] A. Rahman, S. Shatabda, and M. Hasan. An Approximation Algorithm for Sorting by Reversals and Transpositions. *Journal of Discrete Algorithms*, 6(3):449–457, 2008.

[13] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning (ICML'2014)*, volume 32, pages I–387–I–395, 2014.

[14] G. E. Uhlenbeck and L. S. Ornstein. On the theory of the brownian motion. *Physical Review*, 36:823–841, 1930.

[15] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.

[16] M. E. M. T. Walter, Z. Dias, and J. Meidanis. Reversal and Transposition Distance of Linear Chromosomes. In *Proceedings of the 5th International Symposium on String Processing and Information Retrieval (SPIRE'1998)*, pages 96–102, Los Alamitos, CA, USA, 1998. IEEE Computer Society.

[17] C. J. C. H. Watkins and P. Dayan. Technical note Q-learning. *Machine Learning*, 8:279–292, 1992.