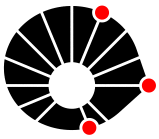

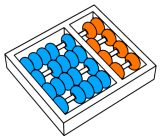

Sorting Permutations by Reversals with Reinforcement Learning

Guilherme Bueno Andrade *Andre Rodrigues Oliveira*
Zanoni Dias

Relatório Técnico - IC-PFG-18-13
Projeto Final de Graduação
2018 - Julho

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Sorting Permutations by Reversals with Reinforcement Learning

Guilherme Bueno Andrade* Andre Rodrigues Oliveira† Zanoni Dias†

Abstract

Finding the minimum number of mutations necessary for one genome to transform into another is a major problem in molecular biology. If genomes are represented as numeric permutations, this problem can be reduced to sorting such permutations using certain genome rearrangements operations, where, in this work, reversals operations are the main focus. We present two different techniques using reinforcement learning to address that. Our results show that this approach is competitive for permutations of size $n < 11$. However, as the permutations grow, converging gets trickier.

1 Introduction

Finding the minimum number of mutations necessary for one genome to transform into another using genome rearrangements, a concept usually referred to as the *distance* between genomes, is a major problem in molecular biology. A common operation is a reversal. It happens when a fragment of the DNA filament gets reversed in the final replica.

Formally, in this work we treat a genome of size n as being equal to a permutation p of integers from 0 to $n-1$. Also, we define the *composition* of permutations p and q as being $p \cdot q = (p_{q_0}\ p_{q_1}\ \ldots\ p_{q_{n-1}})$. In addition, we assume that a reversal operation $f_r(i,j)$ applied to permutation $p = (p_0\ \ldots\ p_{i-1}\ \underline{p_i\ \ldots\ p_j}\ \ldots\ p_{n-1})$ generates the permutation $p \cdot f_r(i,j) = (p_0\ \ldots\ p_{i-1}\ \underline{p_j\ \ldots\ p_i}\ \ldots\ p_{n-1})$.

Finally, as Phylogenetics (the study of evolutionary history and relationships between species) relies strongly on the Principle of Parsimony, the idea that, given a set of possible explanations for a fact, the simplest explanation is most likely to be correct [5], the distance $d(p,q)$ between permutations p and q is the minimum number of operations needed to transform p into q.

Note that finding $d(p,q)$ is equivalent to the problem of finding the distance between some permutation α and the identity permutation ι, where $\alpha = q^{-1} \cdot p$. That is the case because $d(p,q) = d(q^{-1} \cdot p, q^{-1} \cdot q) = d(\alpha, \iota)$. Furthermore, Caprara proved that finding $d(p,\iota)$ is NP-hard [4]. That being so, this work proposes using Reinforcement Learning to come up with a heuristic for finding $d(p,\iota)$ for any arbitrary permutation p.

*gbuenoandrade@gmail.com

†Institute of Computing, University of Campinas, Brazil.

In the next section, we start by introducing several Reinforcement Learning ideas that were used during the project conception. Later, in Section 3, we conclude by talking about implementation details and discussing our findings.

2 Reinforcement Learning

This section introduces the main ideas and concepts from Reinforcement Learning that have been explored during the implementation of the project. It starts by introducing basic ideas from the field and ends talking about the algorithms which supported our agents.

2.1 Overview

Reinforcement Learning (RL), like other branches of Machine Learning, has been drawing a lot of attention from the community in recent years. Google DeepMind's AlphaGo victory over Lee Sedol [2], world champion of the game Go, is one out of many examples of recent astonishing applications of the technique. It is based on an agent learning how to accomplish a certain goal based on interactions with the environment.

RL can be thought of as a sequence of episodes. Each of which consists of the agent at an initial state S_0. Then, based on a policy π, where a policy is a function that maps states to actions, it takes an action a, ending up at state S_1 and receiving some reward R_1. This process keeps going until the agent reaches a terminal state. Its goal is to find a function $\pi*$, known as optimal policy, that maximizes the cumulative discounted reward at a given time step t:

$$G(t) = \sum_{\tau=0}^{\infty} \gamma^{\tau} R_{t+\tau+1}, \tag{1}$$

Where:

$$\gamma \in [0, 1) : \text{is a discount rate to highlight most likely short-term rewards}$$

2.2 Exploitation vs. Exploration

A major concern in RL is the exploitation/exploration trade-off. Exploration is about exploring new possibilities within the environment and finding out more information about it. Exploitation, on the other hand, is related to exploiting already known information to maximize the total reward.

Initially, the agent has no other option but to randomly explore the environment; however, as it learns about its surroundings, it can fall into the trap of sticking to known safe actions and miss larger rewards that depend on exploring unknown states.

This work uses the Epsilon-greedy strategy to address that problem. It specifies an exploration rate ϵ, which is set to 1 initially. This rate defines the ratio of the steps that will be done randomly. Before selecting an action, the agent generates a random number

x. If $x > \epsilon$, then it will select the best-known action (exploit); otherwise, it will select an action at random (explore). As the agent acquires more knowledge about the environment, ϵ is progressively reduced.

2.3 Q-table and the Bellman Equation

In value-based RL, the branch being considered in this work, the efforts are concentrated on maximizing the value function V_π. It tells the agent the expected cumulative discounted reward it will get if it is at state s during time step t:

$$V_\pi(s) = \mathbb{E}_\pi[G(t) \mid S_t = s], \tag{2}$$

Where:

$$\mathbb{E}_\pi : \text{expected value given that policy } \pi \text{ is being followed}$$

V_π can be generalized so as to also consider the action a taken at time step t, which is known as Q-table:

$$Q^\pi(s, a) = \mathbb{E}_\pi[G(t) \mid S_t = s,\ A_t = a] \tag{3}$$

The previous definition is convenient because it allows the agent to pick the best action that can be performed from state s by simply finding $\arg\max_a Q^\pi(s, a)$.

Furthermore, Q can be expressed in terms of itself. An expression known as the Bellman Equation [14]:

$$Q^\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma \sum_{a'} Q^\pi(s', a')], \tag{4}$$

Where:

s' : is the state reached after action a is taken from state s
a' : is an action that can be taken from s'

The above equation can be used alongside dynamic programming to develop iterative approaches to solve the problem [14].

If one can calculate the Q-table from Equation 3, they could trivially come up with a great policy. At each state s, the agent should simply be greedy and select the action a that maximizes $Q^\pi(s, a)$. As mentioned above, finding the Q-table can be easily done with dynamic programming.

However, as the number of states grows, dynamic programming and other iterative approaches become infeasible due to space and time limitations. Fortunately, it turns out that the Q-table can be approximated instead of having its exact values determined, and it will still produce great results [3]. This work tries to achieve that using both linear regression and deep neural networks.

2.4 Monte Carlo and Temporal Difference Learning

In Monte Carlo Approaches, the agent plays an entire episode, keeping track of the rewards received at each time step so it can calculate the cumulative discounted reward. After that, it updates the value function for each visited stated based on the expression [14]:

$$V(S_t) \leftarrow V(S_t) + \alpha(G(t) - V(S_t)), \tag{5}$$

Where:

$$\alpha : \text{is the learning rate}$$

Therefore, the agent only learns after an entire episode has been played. In Temporal Difference Learning, on the other hand, the value of V is updated after each time step. At time $t+1$, the observations made during time t are already being considered. In its simplest form, the method is called TD(0) or 1-step TD, and its update equation is as follows:

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{6}$$

The right-hand side of the previous equation is referred to as the TD(0) error.

TD(0) is biased as it relies on information from a single time step to perform its updates. It does not take into account the fact that the action that caused a reward might have happened several time steps earlier, which can lead to slow convergence [12]. Monte Carlo methods, although not biased, have a lot of variance since they use the rewards of an entire episode in order to perform updates [12].

To overcome that, from Equation 6, we can define the 1-step return, $G_t^{(1)} = R_{t+1} + \gamma V(S_{t+1})$. We can extend the concept further to 2-step return, $G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2})$, and, generically, to, $G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^n V(S_{t+n})$.

TD-Lambda methods use a mathematical trick to average all the possible n-step returns into a single one. This is done by introducing a factor $\lambda \in [0, 1]$ and weighting the nth-return with γ^{n-1}. It can be shown that when $\lambda = 0$, the method is equivalent to TD(0), and when $\lambda = 1$, equivalent to Monte Carlo [3]. So, intuitively, by setting $0 < \lambda < 1$, we can get a mixture of both methods.

2.5 Q-learning

Q-learning is another technique based on Temporal Difference Learning to learn the Q-table. The main difference between it and the previously shown techniques is that Q-learning is off-policy, while TD(0) and TD-Lambda are on-policy [14]. This is reflected in its update equation, derived from the Bellman Equation [17]:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R_{t+1} + \gamma \max_{a'} Q(s', a') - Q(s, a)) \tag{7}$$

The fact that there is no constraint acting upon action a', only that it must optimizes Q, makes it an off-policy method [17].

2.5.1 Deep Q-learning

In order to approximate the Q-table to make using it feasible even when the number of states is very large, since Google AlphaGo's paper [13], it has become common the use of deep neural networks. Even though standard Q-learning is proven to converge when there are finite states and each pair state-action is repeatedly presented [18], the same proof does not hold when neural networks are being used to calculate Q. To face with this issue, this work makes use of several ideas found in the literature [16] to stabilize the training. They are presented in the following subsections.

2.5.2 Experience Replay

During each step of an episode, our estimation can be shifted according to Equation 7. It is reasonable to think that as more truth is being introduced into the system, it will eventually converge; however, this is often not true. One of the reasons is that samples arrive in the order they are experienced and as such are highly correlated. Also, by throwing away our samples immediately after we use it, we are not extracting all the knowledge they carry.

Experience replay address that by storing samples in a queue, and, during each learning step, it samples a random batch and performs gradient descend on it. As samples get old, they are gradually discarded.

2.5.3 Target Network

The points $Q(s, a)$ and $Q(s', a')$ in Equation 7 are very close together as s' is a state reachable from s. That being so, updates to one of them influence the other, which in turn can lead to instabilities.

To overcome that, a second network, called target network, can be used to provide more stable $\widetilde{Q}$ values. This second network is a mere copy of the first one; however, it does not get updated every simulation step. Instead, it stays frozen in time, and only after several steps it is updated by copying the weights from the first network. The introduction of the target network changes our update equation to the following:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R_{t+1} + \gamma \max_{a'} \widetilde{Q}(s', a') - Q(s, a)) \tag{8}$$

2.5.4 Double Learning

Due to the max term presented in both equations 7 and 8, the approximation tends to overestimate the Q function value, which can severely impact on the stability of the algorithm [1]. A solution proposed by Hado van Hasselt [8], called Double Learning, consists of using two Q functions, Q_1 and Q_2, that are independently learned. One function is then used to determine the maximizing action and second to estimate its value. As we are already making use of two different networks, Hado van Hasselt's approach can be easily introduced to the update Equation 8. Its augmented version is as follows:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R_{t+1} + \gamma \widetilde{Q}(s', \operatorname*{argmax}_{a'} Q(s', a')) - Q(s, a)) \tag{9}$$

3 Experiment

Initially, in Section 3.1.1, we discuss different ways to represent the permutations so we can use function approximation techniques. Later, in Section 3.1.2, we talk about the agents' architecture and about a technique that was used to speed up the convergence in Section 3.1.3. Lastly, we present and discuss our results (Section 3.2) and conclusions (Section 3.3).

3.1 Modeling

The agents were implemented so as to perform not only reversals, but also another important genome rearrangement: transposition. However, due to limited computer resources available, we decided later to limit our agents to only perform reversals to make training faster. Their goal was to reach the identity permutation ι while maximizing total reward.

3.1.1 State Representation

Three different state representations were considered. They are as follows.

- *One-Hot Encoding*: Each number is treated as a category. By doing so, each p is mapped to a matrix m, where $m_{ij} = 1$ if $p_i = j$, and 0 otherwise.
- *Min-Max Normalization*: p is mapped to an array v, where $v_i = p_i/(|p| - 1)$.
- *Permutation Characterization*: p is mapped to an array v of 30 features, where the features are the ones described in the work by Silva, Oliveira, and Dias [6].

3.1.2 Proposed Architecture

The experiment was based on two different architectures. The first one consisted of a Double Deep Q-Network (DDQN), with a main network and a secondary target network. Also, it included experience replay (see Section 2.5.2) and double learning (see Section 2.5.4) optimizations.

Both main and secondary networks used the following architecture:

Layer	Number of Units	Activation	Type
Input layer	|Input|	Linear	-
Hidden layer 1	256	ReLU	Fully-connected
Hidden layer 2	256	ReLU	Fully-connected
Output layer	|Actions|	Linear	Fully-connected

Table 1: Main and secondary networks architecture

Furthermore, after tuning, the hyperparameters selected were as follows:

Hyperparameter	Value
γ	0.99
α	0.001
|Batch|	32
ϵ_{min}	0.1
ϵ_{decay}	0.993
|Replay queue|	2000
Loss	logcosh
Optimizer	Adam
Step reward	-1

Table 2: Hyperparameters used in the DDQN

Nevertheless, we tried a second approach. An agent based on the TD-Lambda algorithm was also built. In order to approximate the Q-table, it relied on a much simpler linear regressor. Also, since Q is highly nonlinear, the radial basis function kernel was used as a pre-processing step. The agent's hyperparameters were the following:

Hyperparameter	Value
γ	0.999
α	0.01
λ	0.25
ϵ_{min}	0.05
ϵ_{decay}	0.99
RBF Components	4
RBF γ array	$[5.0, 2.0, 1.0, 0.5]$
Step reward	-1

Table 3: Hyperparameters used to implement TD-Lambda

3.1.3 Speeding up the Convergence

During our initial tests, the Q-table took a long time to converge to optimal values. Being that the case, several techniques were tried so as to decrease training time. The one that seemed more promising is described below:

- *Greedy Pre-Training*: Kececioglu and Sankoff [10] presented a 2-approximation for sorting a permutation using reversals, its output is represented below:

$$\widetilde{d}(p) = (p,\ a(p),\ a'(a(p)),\ \ldots,\ \iota), \tag{10}$$

Where:

a : is the resultant permutation after applying action a on p
ι : is the identity permutation

From Equation 10, let $\hat{V}$:

$$\hat{V}(s) = (\gamma^{|\widetilde{d}(s)|} - 1)/(\gamma - 1) \tag{11}$$

Furthermore, we can define function $\hat{Q}$, which can be shown to be lower bound on Q:

$$\hat{Q}(s, a) = -1 + \gamma \hat{V}(s'), \tag{12}$$

Where:

s' : is the state reached by taking action a at state s

Having defined those, we can talk about our pre-training strategy. It consisted of partially fitting our neural network and linear regressor to function $\hat{Q}$ before start learning from proper episodes.

3.2 Results and Discussion

The models were trained experiencing 10000 episodes, each of which had S_0 set to a random permutation of size $n = 10$ using NumPy's random permutation generator [15] (see Figure 1). After that, in order for the models to effectively sort a permutation p, the exploration rate ϵ was kept constant and equal to 0.2, and then the models were run for another 100 episodes, where S_0 was always equal to p. The best score among those simulations was considered the model's answer $\hat{d}(p, \iota)$. Finally, we used Kececioglu and Sankoff's greedy algorithm output $|\widetilde{d}(p)|$ and ours for 1000 random permutations as a way to measure the model's performance.

As we can see in Table 4, one-hot encoding state representation is slightly better than our Min-Max approach. The former beat Kececioglu and Sankoff's in 28.0% of the simulations, while the latter only in 21.7%. Nonetheless, investigating more effective representations so as to lead to better generalizations is a pending task. Also, we did not manage to reach convergence using either DDQN and Permutation Characterization or the TD-Lambda agent for permutations of size $n > 8$. Regarding our best RL architecture, DDQN and One-Hot Encoding, it consistently outperformed Kececioglu and Sankoff's. However, converging starts getting tricky and time-consuming as n gets bigger, convergence was not achieved using any of our methods for permutations of size $n > 10$. So, it is still early to define our approach as being *practical*.

Configuration	$\hat{d} < \widetilde{\lvert d \rvert}$	$\hat{d} = \widetilde{\lvert d \rvert}$	$\hat{d} > \widetilde{\lvert d \rvert}$	Avg $\hat{d}/d$
DDQN and One-Hot Encoding	**28.0%**	**64.7%**	**7.3%**	**1.0255**
DDQN and Min-Max Normalization	21.7%	61.2%	17.1%	1.0574
DDQN and Permutation Characterization	0.0%	0.0%	100.0%	∞
TD-Lambda	0.0%	0.0%	100.0%	∞

Table 4: Summary of the performance of different model configurations

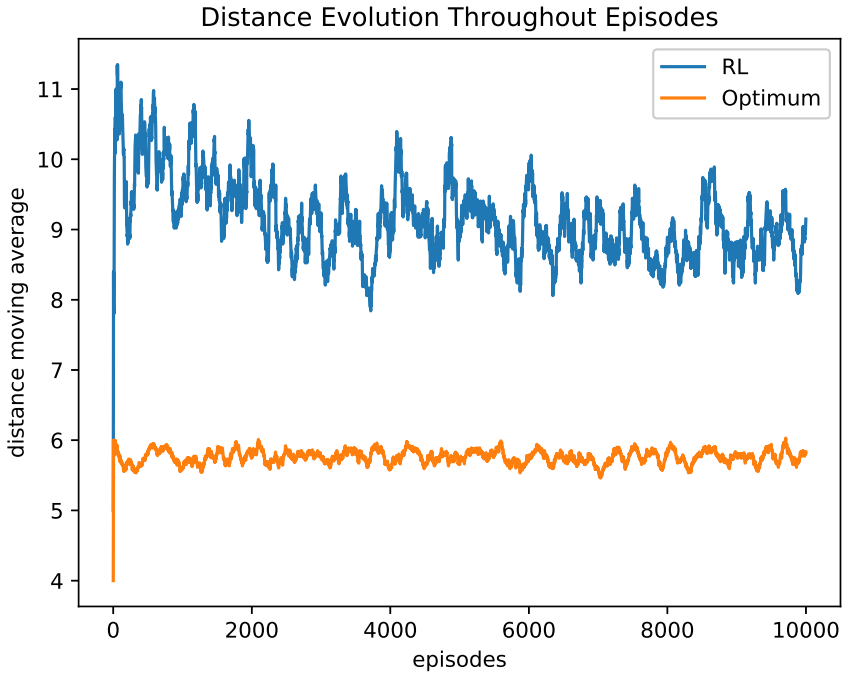

Figure 1: Distance estimation evolution for DDQN and One-Hot Encoding

3.3 Conclusions

This work involved building different RL agents to sort permutations by reversals. Interesting results were obtained for small permutations, but performance was a bottleneck when they got bigger. Future work could try to address that by using distributed deep Q-learning methods [9, 11] and considering recent findings related to large discrete action spaces [7].

References

[1] O. Anschel, N. Baram, and N. Shimkin. Deep reinforcement learning with averaged target DQN. *CoRR*, abs/1611.01929, 2016.

[2] S. Borowiec. AlphaGo seals 4-1 victory over Go grandmaster Lee Sedol. `https://www.theguardian.com/technology/2016/mar/15/googles-alphago-seals-4-1-victory-over-grandmaster-lee-sedol`, 2016. [Online; accessed 15-June-2018].

[3] L. Busoniu, R. Babuska, B. D. Schutter, and D. Ernst. *Reinforcement Learning and Dynamic Programming Using Function Approximators (Automation and Control Engineering)*. CRC Press, 2010.

[4] A. Caprara. Sorting permutations by reversals and eulerian cycle decompositions. *SIAM Journal on Discrete Mathematics*, 12(1):91–110, 1999.

[5] S. M. Carr. Principles of Parsimony Analysis: an example with molecular data. `https://www.mun.ca/biology/scarr/2900_Parsimony_Analysis.htm`, 2002. [Online; accessed 24-June-2018].

[6] F. A. M. da Silva, A. R. Oliveira, and Z. Dias. Machine Learning Applied to Sorting Permutations by Reversals and Transpositions. Technical Report IC-PFG-17-03, Institute of Computing, University of Campinas, July 2017.

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

[8] H. V. Hasselt. Double q-learning. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2613–2621. Curran Associates, Inc., 2010.

[9] D. Horgan, J. Quan, D. Budden, G. Barth-Maron, M. Hessel, H. van Hasselt, and D. Silver. Distributed prioritized experience replay. In *International Conference on Learning Representations*, 2018.

[10] J. Kececioglu and D. Sankoff. Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement. *Algorithmica*, 13(1):180, Feb 1995.

[11] H. Y. Ong, K. Chavez, and A. Hong. Distributed deep q-learning. *CoRR*, abs/1508.04186, 2015.

[12] A. Reis. Reinforcement Learning: Eligibility Traces and TD(lambda). `https://amreis.github.io/ml/reinf-learn/2017/11/02/reinforcement-learning-eligibility-traces.html`, 2017. [Online; accessed 15-June-2018].

[13] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, Oct. 2017.

[14] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. A Bradford Book, 1998.

[15] The SciPy community. NumPy Documentation. `https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.random.permutation.html`, 2018. [Online; accessed 26-June-2018].

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

[17] C. J. C. H. Watkins and P. Dayan. Q-learning. In *Machine Learning*, pages 279–292, 1992.

[18] C. J. C. H. Watkins and P. Dayan. Technical note: q-learning. *Mach. Learn.*, 8(3-4):279–292, May 1992.