UNIVERSITÀ DEGLI STUDI DI SALERNO

Strumenti Formali per la Bioinformatica
Dipartimento di Informatica

GA-DPAMSA

Algoritmo genetico e Deep Reinforcement learning per l'allineamento multiplo di sequenze

AUTORE
Gabriele Tuozzo

DOCENTI
Clelia De Felice
Rocco Zaccagnino
Rosalba Zizza

ANNO ACCADEMICO 2023 - 2024

Abstract

L'allineamento multiplo di sequenze viene utilizzato in bioinformatica come tecnica per comparare due o più sequenze biologiche simultaneamente. Il problema principale dell'allineamento multiplo di sequenze è la sua complessità computazionale, che cresce esponenzialmente con il numero di sequenze e la loro lunghezza. Questo rende l'allineamento esatto impraticabile per un gran numero di sequenze lunghe. Inoltre, le sequenze possono variare significativamente fra loro, con inserzioni, delezioni e mutazioni che complicano ulteriormente il processo. Per superare queste sfide, vengono utilizzati metodi euristici e algoritmi approssimativi. Negli ultimi anni però si sono considerate anche tecniche basate sul machine learning, come ad esempio l'utilizzo del reinforcement learning. Tali tecniche sembrano promettenti ma anche qui vi sono diverse sfide, come ad esempio, il crescere del tempo di training dei modelli e delle risorse hardware richieste, all'aumentare della lunghezza delle sequenze e del numero. Per superare questo problema viene proposto un algoritmo genetico, applicato al reinforcement learning, con l'obiettivo di riutilizzare un modello addestrato per problemi più piccoli (in termini di lunghezza delle sequenze e numero), su problemi più grandi, senza dover riaddestrare il modello.

Indice

Capitolo 1

Introduzione

L'obiettivo di questo lavoro è mostrare se è possibile utilizzare un modello di Reinforcement Learning (RL) addestrato per risolvere un problema P_1, con sequenze sequenze lunghe massimo L_1, ed un numero di sequenze N_1, per risolvere un problema P_2, con sequenze lunghe massimo L_2 ed un numero di sequenze pari ad N_2, dove risulta $L_2 > L_1$ ed $N_2 > N_1$. Per affrontare questa sfida, proponiamo una metodologia che combina il reinforcement learning con un algoritmo genetico. Durante la fase di mutazione dell'algoritmo genetico, l'agente di reinforcement learning viene impiegato per operare sul problema che ha la stessa grandezza di P_1, ovvero quello su cui è stato inizialmente addestrato. L'obiettivo è verificare se questa combinazione possa migliorare l'efficacia e l'accuratezza dell'allineamento delle sequenze nel problema P_2, che presenta una maggiore complessità sia in termini di lunghezza delle sequenze che di numero delle sequenze da allineare, senza dover riaddestrare l'agente per la risoluzione del problema P_2. Addestrare l'agente di reinforcement learning sul problema P_2, può richiedere diverso tempo, così come notevoli capacità di computazione. Nella sezione successiva vedremo un'analisi dello stato attuale della letteratura e del modello di reinforcement learning usato come test per validare il nostro approccio. Va sottolineato che la metodologia adottata e l'algoritmo genetico sono indipendenti dal modello di reinforcement learning e il codice dell'applicazione può essere facilmente adattato per supportare un diverso modello (come dettagliato nella sezione 4.2.6, occorre semplicemente cambiare la chiamata al modello fatta nella funzione di mutazione dell'algoritmo genetico). Nei successivi capitoli invece, verrà visto in dettaglio l'architettura di DPAMSA (il modello per il test del nostro approccio), per poi passare a spiegare tutti i dettagli dell'algoritmo genetico, su come è stato implementato e come opera sulle sequenze da allineare. Infine, saranno mostrati i risultati dell'esperimento, con una comparazione con altri tool di allineamento noti in letteratura, vedremo poi i possibili sviluppi futuri del lavoro e infine le conclusioni. Il codice dell'applicazione realizzata e i dataset utilizzati per il test sono disponibili su Github al seguente link: https://github.com/strumenti-formali-per-la-bioinformatica/GA-DPAMSA.

Capitolo 2

Modelli di Reinforcement Learning in lettaratura

Per implementare il nostro algoritmo genetico, avevamo bisogno di utilizzare un modello di Reinforcement Learning da cui partire che offrisse già buone prestazioni e soprattutto con codice disponibile su GitHub con licenza open, in modo da poter riutilizzare il lavoro. Dall'analisi della letteratura, cinque lavori sono stati giudicati particolarmente interessanti. Nel paper [4] si utilizza l'algoritmo di Q-learning, integrato con il modello Actor-Critic, per migliorare le prestazioni dell'MSA. In particolare, il *Q-learning* è un algoritmo di apprendimento per rinforzo che consente a un agente di apprendere una politica, cioè una strategia per scegliere le azioni, che massimizza la ricompensa cumulativa futura. È un tipo di apprendimento senza modelli (model-free) che utilizza una funzione di valore, chiamata funzione Q, per valutare l'utilità di eseguire una certa azione in un determinato stato. L'*Actor-Critic* invece, è una famiglia di algoritmi di apprendimento per rinforzo che combina elementi dell'approccio basato su politiche (policy-based) e dell'approccio basato su valori (value-based). Si compone di due parti principali: l'actor e il critic.

- Actor: l'actor è responsabili di aggiornare la politica, ovvero la probabilità di selezionare una certa azione in uno stato specifico.
- Critic: il critic valuta la politica dell'actor calcolando una funzione di valore. Il critic fornisce un feedback al'actor su quanto sia buona una determinata azione in un certo stato.

Nel paper il Q-learning viene utilizzato per stimare la funzione di valore delle azioni in uno stato, mentre il modello Actor-Critic viene utilizzato per migliorare iterativamente la politica di allineamento delle sequenze. Questo approccio ibrido permette di sfruttare i vantaggi di entrambi i metodi, migliorando l'efficienza e l'accuratezza dell'allineamento delle sequenze multiple. Seppur il metodo applicato in questo lavoro risultava interessante, purtroppo non è stato trovato alcun riferimento a dove poter reperire il codice per utilizzare il modello realizzato.

Il lavoro presentato nel paper [1] invece, utilizza sempre il Q-learning e l'algoritmo Actor-Critic, ma aggiungendo le reti Long Short-Term Memory (LSTM) che risultano particolarmente utili grazie alla loro capacità di catturare dipendenze a lungo termine nelle sequenze. I risultati sperimentali discussi nel paper evidenziano miglioramenti significativi rispetto ai metodi tradizionali, validando l'efficacia del loro approccio. Tuttavia, anche in questo caso non viene fornito il codice dell'implementazione.

Nel paper [2] si fornisce un modello di RL progettato per essere compatibile con dispositivi edge embedded. Questo metodo denominato EdgeAlign lavora su una finestra mobile, applicando il RL su questa finestra e non sull'intera sequenza da allineare. Il concetto applicato in questo paper utilizza tecniche di preprocessing come la Longest Common Substring (LCS)[1], in particolare, dividendo il compito dell'allineamento in due sotto-allineamenti diretti a sinistra e a destra della LCS. La metodologia utilizzata risulta essere molta vicina al nostro obiettivo (cioè il modello lavora su una finestra ridotta anzichè sulle intere sequenze), ma la limitazione del loro modello è che funziona solo su coppie di sequenze. Poichè a corredo del paper è stato fornito un link a GitHub del codice del modello realizzato, è stata investigata la possibilità di estendere il loro modello per lavorare su sequenze multiple. Tuttavia, cioè comportava riscrivere gran parte del modello, in quanto non è stato progettato per una futura semplice estensione, tuttavia il paper è risultato utile per apprendere la metodologia per far operare un agente di RL su una finestra più piccola anziché sulle intere sequenze.

Il paper [7] presenta un tool chiamato RLALIGN, che presenta una soluzione utilizzando il RL insieme al framework Asynchronous Advantage Actor Critic (A3C). Quest'ultimo è un algoritmo di RL avanzato che utilizza un'architettura multi-thread per aggiornare un'unica rete neurale condivisa attraverso esperienze raccolte in parallelo da più agenti. Questo approccio migliora la stabilità e la velocità di convergenza rispetto agli algoritmi di RL tradizionali. RLALIGN è stato addestrato su sequenze di lunghezza moderata, e grazie a euristiche sviluppate, può scalare a sequenze più lunghe, senza dover rifare training del modello, mantenendo risultati accettabili. Il problema però che RLALIGN scala su sequenze più lunghe, ma non nel loro numero. Cioè se il modello ad esempio è stato addestrato per allineare 3 sequenza di lunghezza L, lo stesso può essere utilizzato per ottenere l'allineamento su sequenze lunghe L_2 con $L_2 > L$, ma non può essere utilizzato per l'allineamento di 4 o più sequenze. Poiché è stato fornito il link GitHub al codice dell'applicazione, inizialmente si è partiti da tale modello per realizzare l'algoritmo genetico, tuttavia analizzando il codice, è risultato incompleto ed i vari script forniti per il training del modello, non funzionavano per eseguire il training della rete con dataset contenenti più di 2 sequenze di DNA.

Alla fine, tra le varie proposte viste in letterature, si è deciso di utilizzare il modello presentato nel paper [6], che sarà dettagliato nella sezione successiva.

[1] è un algoritmo utilizzato per trovare la sottostringa più lunga che è presente in due sequenze (o stringhe) diverse

Capitolo 3

DPAMSA

Nel paper [6] viene presentato *DPAMSA* (Deep reinforcement learning-Positional encoding-self Attention per il Multi Sequence Alignment), un tool che utilizza il Deep Reinforcement Learning, Positional encoding e self-Attention per risolvere il problema dell'allineamento multiplo delle sequenze. In primo luogo, il positional encoding codifica la posizione della sequenza per evitare la perdita di informazioni sulla posizione dei nucleotidi. In secondo luogo, il modello di self-Attention viene utilizzato per estrarre le features chiave della sequenza. Quindi le features chiave vanno in input ad un multi-layer perceptron, che può calcolare la posizione dell'inserzione del gap in accordo alle features. Essenzialmente, vengono utilizzate tecniche di processing che provengono dal natural language processing insieme al Deep Reinforcement Learning (DRL). Il DRL combina il RL con le Deep Learning networks per creare agenti che possono prendere decisioni intelligenti in ambienti complessi e dinamici. In particolare, si utilizza una Deep Q-Network per approssimare la funzione Q, si utilizza il Q-learning per aggiornare la rete e si introduce un replay buffer per memorizzare le esperienze passate e il target network per stabilizzare l'apprendimento. Le politiche sono apprese direttamente tramite ottimizzazione di gradiente. L'ambiente di apprendimento del RL si basa sull'allineamento progressivo delle colonne e il sotto-allineamento di ciascuna colonna viene calcolato passo dopo passo. Successivamente, tutti i sotto-allineamenti vengono uniti in un allineamento completo.

3.1 Come funziona DPAMSA

Dato un dataset S contenente s sequenze, l'allineamento corrispondente è CA. Tuttavia, non esiste un modello di evoluzione biologica completamente corretto e non possiamo giudicare se l'allineamento CA sia conforme alla legge dell'evoluzione naturale. Pertanto l'allineamento CA del dataset S potrebbe non essere unico per la ricerca attuale. Potrebbero esserci più risultati o persino infiniti risultati, indicati come $\{CA_i\}_{i=1}^{\infty}$. Nell'allineamento multiplo di sequenze abbiamo un criterio di valutazione che possiamo usare per ognuno degli allineamenti CA_i. Il criterio è il calcolo del punteggio della Sum-of-Pairs (SP). Di seguito la formula

utilizzata per il calcolo.

$$SP(CA) = \sum_{k=1}^{len(CA)} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} p(CA^{i,k}, CA^{j,k})$$

dove $len(CA)$ rappresenta la lunghezza di una singola sequenza nell'allineamento CA, e $CA^{i,k}$ rappresenta il $k - esimo$ nucleotide nella sequenza $i - esima$. La funzione $p(x, y)$ viene utilizzata per calcolare il punteggio di matching dei nucleotidi x e y, nell'equazione seguente viene mostrato nel dettaglio il punteggio assegnato dalla funzione.

$$p(x, y) = \begin{cases} -4 & x = gap | y = gap \\ 4 & x = y \\ -4 & x \neq y \end{cases}$$

La base di calcolo di questa funzione include principalmente i seguenti tre tipi:

1. Penalità per un gap: quando o x o y è un gap, lo score è -4.
2. Matching score: quando x e y sono uguali, lo score è 4.
3. Penalità per miasmatch: quando x e y sono diversi, lo score è -4.

Pertanto, il SP score, è la somma dei punteggi di tutti nucleotidi accappiati nell'allineamento. L'obiettivo è mirare a trovare l'allineamento con la SP più alta. Cioè:

$$max(SP(CA)), CA \in \{CA_i\}_{i=1}^{\infty}$$

Nella figura 3.1 possiamo vedere l'architettura del tool DPAMSA proveniente dal paper di riferimento.

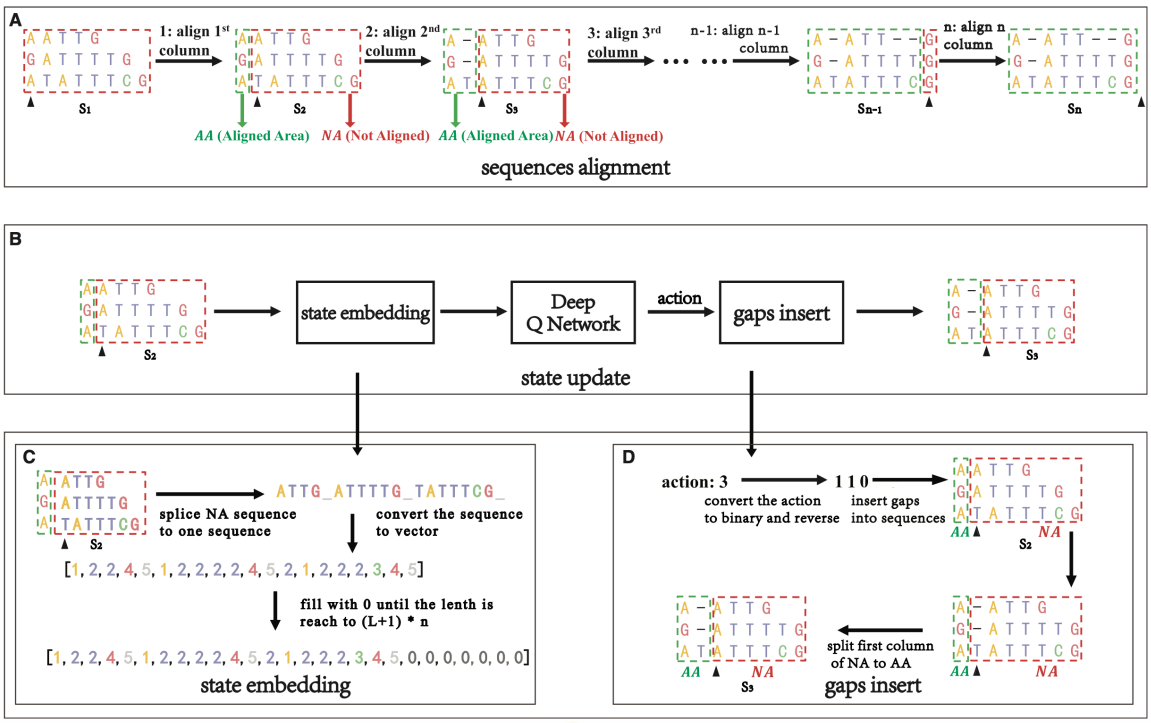

Figura 3.1: L'architettura di DPAMSA

Nella figura 3.1(A) viene mostrato il processo di allineamento delle sequenze fatto colonna per colonna, e lo stato della sequenza viene aggiornato ogni volta che l'allineamento di una colonna viene completato. Nella figura 3.1 (B) viene mostrato nel dettaglio come avviene l'aggiornamento dello stato da S_2 a S_3. Infine le figure 3.1 (C) e 3.1 (D) mostrano rispettivamente una descrizione dettagliata dello state embedding e dell'inserimento dei gap.

1. *Allineamento delle sequenze*: nella figura 3.1(A) viene mostrato il processo di allineamento delle sequenze. Data una sequenza S, l'allineamento avviene colonna per colonna. Lo stato della sequenza cambia quando una colonna viene allineata, come ad esempio per S_1,S_2 ed S_3, fino a quando non viene completato l'allineamento dell'ultima colonna. Durante il processo di allineamento, le colonne che hanno completato l'allineamento delle sequenze vengono incluse in un box verde, denotato con AA (Aligned Area). Al contrario, le colonne che non hanno completato l'allineamento sono circondate da un riquadro rosso, indicato come NA (Not Aligned Area). La freccia nera punta alla colonna di allineamento corrente.

2. *Aggiornamento dello stato*: nella figura 3.1(B) viene mostrato il processo di aggiornamento dello stato della sequenza da S_2 ad S_3. Per prima cosa la sequenza non allineata in S_2 viene incorporata in un embedding sequence vector, viene passato il vettore alla deep Q network, viene restituita l'azione dalla deep Q network, e poi vengono inseriti i gap nella colonna attualmente allineata in accordo al valore dell'azione restituita.

3. State embedding: nella figura 3.1(C) viene mostrato il processo di embedding dello stato. Tutte le sequenze in NA vengono concatenate in una lunga sequenza e poi

convertite nel corrispondente vettore di interi della sequenza. Quindi, ogni sequenza nella NA viene concatenata in una lunga sequenza espressa come

```
[ATTG_ATTTTG_TATTTCG_]
```

Successivamente questa lunga sequenza viene convertita nel vettore di stato

```
[1,2,2,4,5,2,2,2,2,4,5,2,1,2,2,2,3,4,5]
```

Occorre notare che dopo ogni allineamento di una colonna, la lunghezza del vettore di stato si accorcia, che non è adatto per l'input della rete neurale (la lunghezza deve mantenersi identica). Pertanto, vi è un metodo che fissa la lunghezza del vettore di stato a $n \cdot (L + 1)$, con la parte vuota del vettore che viene riempita di tutti 0. L è la lunghezza massima delle tre sequenze allineate ed n è il numero di sequenze allineate.

4. Inserimento dei Gap: nella figura 3.1(D) viene mostrato il processo di inserimento dei gap nella sequenza. Prima di tutto, il valore di azione 3 viene codificato in binario come 001 e invertito in 110. Otteniamo i valori corrispondenti dall'alto al basso (1,1,0). Quando il valore è 1, si inserisce il gap, quando è 0 non occorre nessun cambiamento. Quindi, nella colonna indicata dalla freccia nello stato S_2, nella prima e seconda riga inserisci gap, mentre lascia inalterata la terza riga, arrivando poi nello stato S_3

L'ambiente di reinforcement learning include principalmente l'*agente*, lo *stato*,l'*azione* e le *funzioni di ricompensa*. Vediamo ora nel dettaglio come sono definite queste componenti in DPAMSA.

- *L'agente*: in generale l'agente è considerato come un'entità con una specifica capacità di prendere decisioni. Nel RL, gli agenti possono interagire continuamente con l'ambiente per regolare o ottimizzare le loro strategie.

- Lo *stato*: rappresentato da s, è un vettore di interi, come ad esempio in figura 3.1(C) lo stato S_2 è il vettore

```
[1,2,2,4,5,2,2,2,2,4,5,2,1,2,2,2,3,4,5,0,0,0,0,0,0,0]
```

- *L'azione*: Rappresentata da a. L'azione è un valore intero usato per controllare l'inserimento dei gap nella posizione corrispondente della colonna di allineamento corrente nella sequenza. Nella figura 3.1(D), lo stato dell'allineamento delle sequenze è cambiato da S_2 a S_3 tramite l'azione dell'inserimento dei gap. Il valore dell'azione è 3, che usando la codifica in binario e facendone il reverse diventa 110, e seguendo la regola dove 1 rappresenta l'inserimento di gap e 0 indica nessuna operazione, l'operazione

di inserimento di gap viene eseguita sulla prima e sulla seconda colonna della colonna di allineamento corrente indicata dalla freccia, mentre la terza colonna non viene modificata.

- *Episodio*: La sequenza che viene generata dall'iterazione tra l'agente e l'ambiente viene chiamato episodio, il quale registra tutti i processi di transizione di stato dall'inizio allo stato finale e viene mostrato di seguito:

$$episode = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ...s_t... \xrightarrow{a_{T-2}} s_{T-1} \xrightarrow{a_{T-1}} s_T$$

Dove s_t è lo stato dell'ambiente al passo temporale t ed a_t è l'azione da eseguire nello stato s_t dell'ambiente. T è il passo temporale di terminazione, il che significa che dopo il passo temporale T, l'agente e l'ambiente non interagiscono più, segnando la fine dell'episodio corrente. Vi sono due differenti tipi di casi di terminazione in questo studio. *normal termination* e *abnormal termination*.

La *Normal termination* avviene quando tutte le sequenze sono allineate, la lunghezza di tutte le sequenze in NA_T è quindi 0. L'espressione matematica usata per indicare la *Normal termination* è

$$\sum_{i=1}^{n} len(NA_T^i) = 0$$

dove NA_T^i significa l'$i - esima$ sequenza in NA_T

Si parla invece di *abnormal termination*, quando c'è una sequenza di lunghezza 0 in NA_T e il suo vettore di azioni corrispondete è 0. A causa di una sequenza vuota in NA_{T-1}, significa che non è più possibile separare completamente la colonna da NA_{T-1}. In questo momento, l'episodio corrente dovrebbe essere terminato per evitare errori imprevedibili. L'espressione matematica usata per indicare la *abnormal termination* è la seguente:

$$\exists seq \in NA_{T-1},\ len(seq) = 0$$

- *Reward*: rappresentata con r. La reward che viene usata per l'agente è direttamente rappresentata dal punteggio della SP nella prima colonna della sequenza non allineata (NA). La formula è la seguente:

$$r_t = \begin{cases} -r_{max} & \exists seq \in NA_{T-1},\ len(seq) = 0 \\ \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} p(NA_t^{i,1}, NA_t^{j,1}) & altrimenti \end{cases}$$

dove r_{max} è il punteggio della SP più grande in una singola colonna, tutti i nucleotidi in questa colonna sono gli stessi. Il valore sarebbe $\frac{4n\cdot(n-1)}{2}$. Quando viene raggiunto il passo temporale di terminazione T, l'episodio corrente dovrebbe essere terminato immediatamente. Restituendo il valore $-r_{max}$, è possibile ridurre il valore cumulativo della ricompensa, quando si arriva alla *abnormal termination*, questo al fine di ridurre

il peso dell'azione non corretta. Negli altri casi, incluso quando si arriva alla *normal termination*, la reward r_t è determinata dalla formula $\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} p(NA_t^{i,1}, NA_t^{j,1})$, dove $NA_t^{i,1}$ rappresenta il primo nucleotide della $i-esima$ sequenza in NA_t

In figura 3.2 è rappresentato il modello di Deep Reinforcment Learning utilizzato da DPAMSA.

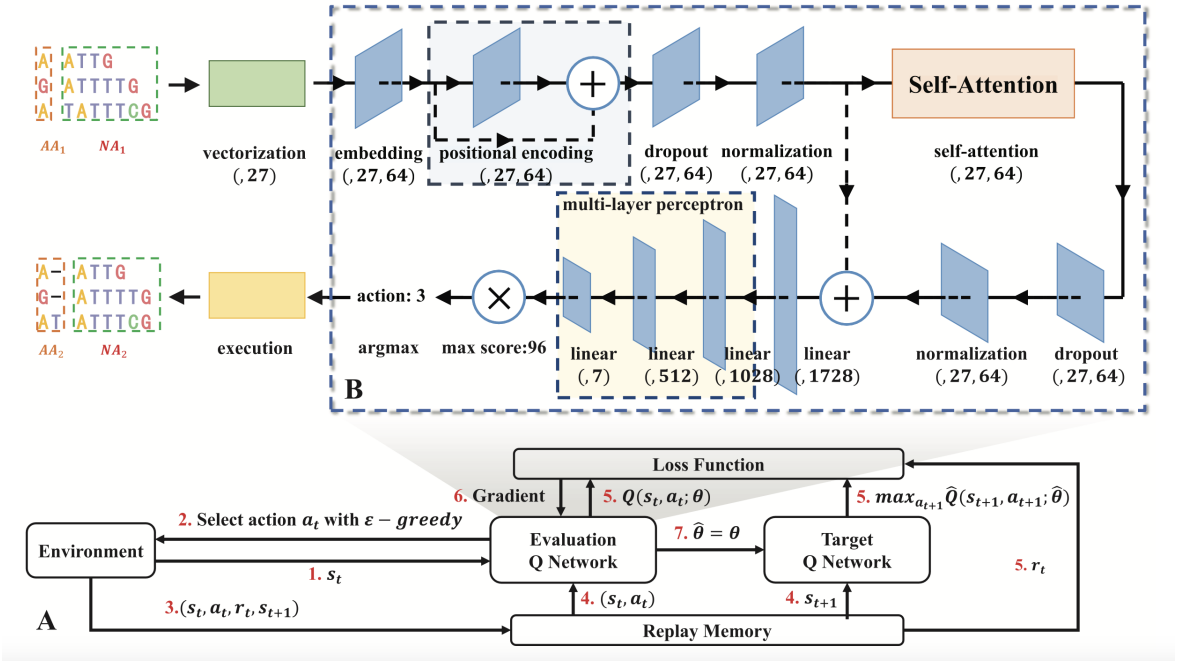

Figura 3.2: Il modello di DRL di DPAMSA

Per prima cosa, la sequenza originale S viene trasferita nell'ambiente. L'ambiente otterrà lo stato iniziale S_1 basato sulla sequenza S. Poi, in accordo alle $\epsilon-$greedy policy, riceve il valore dell'azione corrispondente e la restituisce all'ambiente; questo viene ripetuto fino a quando viene completato l'allineamento della sequenza. La strategia ϵ-greedy prende il valore dell'azione in modo casuale con una probabilità 1-ϵ ed ottiene il valore dell'azione attraverso la rete di validazione Q (Q network) con probabilità ϵ. La figura 3.2(B) mostra come è fatta la deep Q network, che essenzialmente è formata da tre pezzi: *Self-Attention mechanism*, *Positional encoding* e *Multi-Layer perceptron*.

- *Self-Attention mechanism*: la funzione principale di questa parte è di estrarre le fratures chiave dallo stato corrente delle sequenze, che è molto importante per aumentare l'accuratezza dell'allineamento e l'affidabilità del modello. Il meccanismo di attenzione può mappare il vettore query, key e value nel vettore di output per estrarre le caratteristiche. Questo componente è stato quindi introdotto con lo scopo di far "porre attenzione" al modello alla correlazione di ogni nucelotide nella sequenza di input.

- *Positional encoding*: poiché il meccanismo di Self-Attention comporta la perdita delle informazioni sulla posizione dei nucleotidi nella sequenza, lo scopo del positional encoding è di mantenere le informazioni sulla posizione prima di estrarre le features. Infatti,

nel modello non ci sono meccanismi di convoluzione e ricorrenza, quindi il modello non può catturare le informazioni sulla struttura della sequenza e le informazioni sulla posizione dei nucleotidi. Il metodo classico che tipicamente viene usato in questo caso per calcolare le informazioni sulla posizione è basato sull'apprendimento. Ma questo andrebbe ad aumentare il tempo di addestramento ed inferenza del modello, oltre che ad aumentare anche l'utilizzo della memoria. Pertanto, per migliorare l'efficienza del modello, viene utilizzata una formula per codificare le informazioni sulla posizione e vengono aggiunti i dati codificati al vettore embedded. La formula che viene utilizzata è la seguente:

$$PE_{pos} = \begin{cases} \sin(\frac{pos}{10000^{\frac{2i}{d\ model}}}) & i = 2k \\ \cos(\frac{pos}{10000^{\frac{2i}{d\ model}}}) & i = 2k+1 \end{cases}$$

La formula proviene dal paper [9]. Nella formula, PE rappresenta le informazioni sulla posizione assoluta dopo la codifica, pos è la posizione, e $2i$ è la dimensione del vettore embedded. Utilizzando questo metodo di calcolo, non solo è possibile ottenere la posizione assoluta PE_{pos}, ma è anche possibile ottenere le informazioni sulla posizione relativa PE_{pos+k} attraverso la trasformazione lineare di PE_{pos}. Nel paper [9] viene dimostrato che l'effetto delle informazioni sulla posizione apprese e delle informazioni sulla posizione calcolate non è molto differente.

- *Multi-Layer perceptron*: questa parte si trova dopo il meccanismo di *Self-Attention*. Calcola principalmente i valori Q di tutte le azioni nello stato attuale attraverso uno strato di connessione completa a più livelli basato sulle features estratte. Quindi questo componente viene usato per calcolare il vettore finale dei valori Q nella rete Q (il valore Q può essere considerato come uno score previsto della SP). Questa rete è composta da tre livelli lineari. Poiché il vettore di output della rete è la forma normalizzata del valore Q, è necessario moltiplicarlo per il massimo valore teorico del valore Q per convertirlo nel vettore finale dei valori Q. L'indice del vettore dei valori Q rappresenta l'azione effettiva.

Il *training del modello* avviene essenzialmente facendo in modo che quando lo stato della sequenza nell'ambiente cambia, le informazioni ambientali vengono trasmesse alla replay memory. Quando il numero di quadruple salvate nella replay memory raggiunge il numero massimo di batches (128), s_t e a_t vengono trasferiti dalla replay memory alla rete Q di validazione (verification Q network, da notare che le due reti Q hanno la stessa struttura di rete), e poi vengono calcolati i valori Q_t e Q_{t+1}. I due valori Q calcolati ed i corrispondenti r_t dalla replay memory vengono trasferiti alla loss function per calcolare la perdita. Quando la rete di validazione Q è stata aggiornata 128 volte, si sincronizzano i valori dei parametri alla rete Q target. La vera rete che viene addestrata nella fase di training è la rete Q di valutazione (evaluation Q network). In contrasto, la rete Q target (Target Q network) copia solamente gli ultimi parametri dalla rete di valutazione Q ad intervalli regolari. In un certo

senso, la struttura composta dalla rete di valutazione Q, la rete target Q e la loss function sono considerate un agente perché possono fare delle decisioni ed aggiustare le strategie.

Riassumendo quindi quello che fa DPAMSA è introdurre un meccanismo di self-attention per estrarre le feature dell'allineamento corrente e poi utilizza un positional encoding per migliorare l'affidabilità delle feature estratte. Infine, le feature vanno in input in un multi-layer perceptron per predire il peso dell'azione nel reinforcement learning. Nella sezione successiva verrà illustrato l'algoritmo genetico costruito per poter permettere al modello DPAMSA di operare su sequenze più lunghe e in quantità anche maggiore rispetto a quelle viste nella fase di training, senza necessità di riaddestrare la rete.

Capitolo 4

L'algoritmo Genetico

4.1 Introduzione agli algoritmi genetici

Gli algoritmi genetici (AG) sono algoritmi di ricerca euristica adattivi. Gli algoritmi genetici si basano sulle idee della selezione naturale e della genetica. Si tratta di uno sfruttamento intelligente delle ricerche casuali per dirigere la ricerca nella regione di migliori prestazioni nello spazio delle soluzioni. Sono comunemente utilizzati per generare soluzioni di alta qualità per problemi di ottimizzazione e di ricerca. Gli algoritmi genetici simulano il processo di selezione naturale, per cui le specie che riescono ad adattarsi ai cambiamenti del loro ambiente possono sopravvivere, riprodursi e passare alla generazione successiva. In parole povere, simulano la "sopravvivenza del più adatto" tra gli individui di generazioni consecutive per risolvere il problema. Ogni generazione consiste in una popolazione di individui e ogni individuo rappresenta un punto dello spazio di ricerca e una possibile soluzione. Ogni individuo è rappresentato tipicamente come una stringa di caratteri, interi, float o bit. Gli algoritmi genetici si basano su un'analogia con la struttura genetica e il comportamento dei cromosomi della popolazione. In particolare: gli individui di una popolazione competono per le risorse e si accoppiano. Gli individui che hanno successo (fittest) si accoppiano per creare più prole degli altri. I geni del genitore "più adatto" si propagano per tutta la generazione, cioè a volte i genitori creano una progenie migliore di entrambi i genitori. In questo modo, ogni generazione successiva è più adatta all'ambiente in cui vive.

La popolazione di individui viene mantenuta all'interno dello spazio di ricerca. Ogni individuo rappresenta una soluzione nello spazio di ricerca per un determinato problema. Ogni individuo è codificato come un vettore di lunghezza finita di componenti (analoghe al cromosoma). Queste componenti variabili sono analoghe ai geni. Quindi un cromosoma (individuo) è composto da diversi geni (componenti variabili). In figura 4.1 viene mostrata la struttura di popolazione, cromosoma e gene.

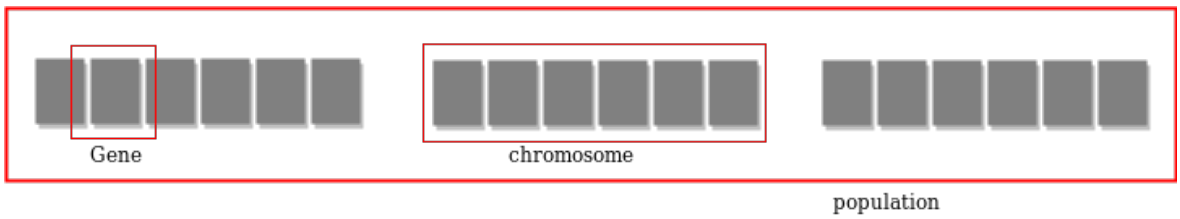

Figura 4.1: Popolazione, cromosoma e geni

Ad ogni individuo viene assegnato un punteggio di fitness (*fitness score*) che indica la capacità dell'individuo di "competere". Si cercano gli individui che hanno un punteggio di fitness ottimale (o quasi). La funzione di fitness viene scelta poi in base al problema. Gli AG mantengono una popolazione di n individui (cromosomi/soluzioni) con il loro fitness score. Gli individui con fitness score migliori hanno più possibilità di riprodursi rispetto agli altri. Vengono selezionati gli individui con punteggi di fitness migliori che si accoppiano e producono una prole migliore combinando i cromosomi dei genitori. La dimensione della popolazione è statica, quindi è necessario creare spazio per i nuovi arrivi. Pertanto, alcuni individui muoiono e vengono sostituiti dai nuovi arrivati, creando infine una nuova generazione quando tutte le opportunità di accoppiamento della vecchia popolazione sono esaurite. Si spera che nelle generazioni successive arrivino soluzioni migliori, mentre quelle meno adatte muoiono. Ogni nuova generazione ha in media più "geni migliori" rispetto agli individui (soluzioni) delle generazioni precedenti. Pertanto, ogni nuova generazione ha "soluzioni parziali" migliori delle generazioni precedenti. Una volta che la progenie prodotta non presenta differenze significative rispetto alla progenie prodotta dalle popolazioni precedenti, la popolazione è convergente. Si dice che l'algoritmo converge a un insieme di soluzioni per il problema.

Una volta creata la generazione iniziale, l'algoritmo fa evolvere la generazione utilizzando i seguenti operatori:

- *Operatore di selezione*: l'idea è quella di dare la preferenza agli individui con un buon fitness score e di permettere loro di passare i propri geni alle generazioni successive.
- *Operatore di crossover*: rappresenta l'accoppiamento tra individui. Due individui vengono selezionati con l'operatore di selezione e i siti di crossover vengono scelti in modo casuale. I geni in questi siti di crossover vengono scambiati, creando un individuo completamente nuovo (progenie). Nella figura 4.2 vi è un esempio dell'operazione di crossover.

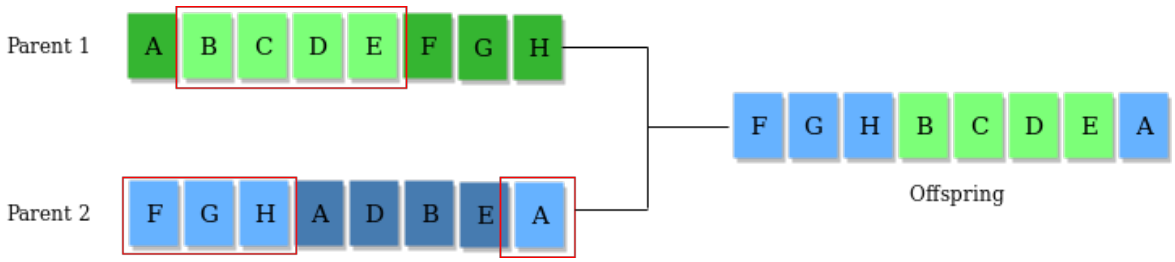

Figura 4.2: Operazione di crossover

- *Operatore di mutazione*: l'idea chiave qui è quella di inserire geni casuali nella prole per mantenere la diversità nella popolazione ed evitare una convergenza prematura. Nella figura 4.3 in verde chiaro ci sono i geni mutati.

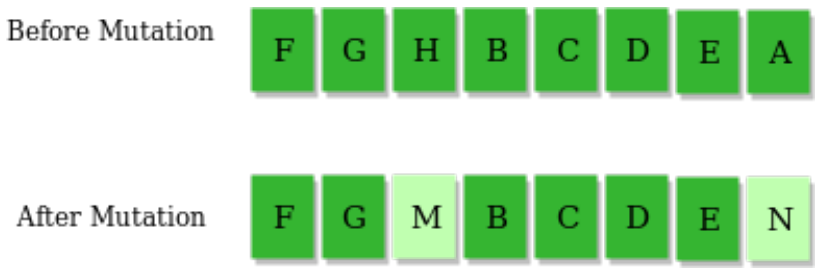

Figura 4.3: Operazione di mutazione

Possiamo quindi riassumete i passaggi fondamentali di un algoritmo genetico come:

1. Inizializza in maniera random la popolazione p
2. Determina la fitness della popolazione p
3. Fino a convergenza ripetere i seguenti passaggi:
   (a) Selezionare i genitori dalla popolazione
   (b) Esegui crossover per generare una nuova popolazione
   (c) Eseguire la mutazione sulla nuova popolazione
   (d) Calcolare il fitness della nuova popolazione

Gli algoritmi genetici sono particolarmente efficaci per risolvere problemi complessi che presentano spazi di ricerca molto vasti e sono molto flessibili, cioè possono essere formulati per essere applicati ad una vasta gamma di problemi di ottimizzazione. Un altro vantaggio del loro utilizzo è che a differenza di molti algoritmi di ottimizzazione tradizionali che posso rimanere bloccati nei minimi locali, gli algoritmi genetici tendono ad esplorare l'intero spazio delle soluzioni, aumentando le possibilità di trovare il minimo (o massimo) globale.

4.2 Algoritmo genetico e RL per l'allineamento multiplo di sequenze

Dopo un'introduzione al RL e agli AG, passiamo a dettagliare come sono stati usati insieme per risolvere il problema dell'allineamento multiplo di sequenze.

Supponiamo di avere 4 sequenze da allineare, ognuna di lunghezza 8, come quella in figura 4.4. Nell'algoritmo questa viene codificata come una matrice di dimensione $4x8$ e viene chiamata $board$.

Figura 4.4: Board delle sequenze da allineare

L'obiettivo è ora fare in modo, grazie all'utilizzo di un algoritmo genetico, di utilizzare un agente di RL per allineare la board, ma avendo un agente che è stato addestrato su una board $2x4$, quindi sostanzialmente saprebbe risolvere il problema solo su una $sub - board$, cioè quella nel quadrato verde in figura 4.5.

Figura 4.5: Sub-board su cui può operare l'agente di RL

Nelle sezioni successive dettaglieremo come è stato creato l'algoritmo genetico per risolvere tale problema, spiegando come sono state implementate le operazioni di: *generazione della popolazione*, *fitness score*, *selection*, *crossover* e *mutation*.

4.2.1 Generazione della popolazione

Date delle sequenze da allineare, come quelle in 4.4, viene creata una matrice (nel caso specifico della figura avremo una matrice $4x8$) e poi per poter applicare il RL, ogni nucleotide viene mappato ad un numero, rispettando la seguente regola:

```
A:1, T:2, C:3, G:4, a:1, t:2, c:3, g:4, -:5
```

Da notare che il gap viene codificato con 5. Quindi dopo questa fase avremo una matrice di interi $4x8$. A questo punto viene generata la popolazione dell'algoritmo genetico, essenzialmente ogni individuo sarà una matrice (da qui in poi ci riferiremo alla matrice anche come $board$). In figura 4.6 possiamo vedere come appare una popolazione di 4 individui per l'allineamento di 4 sequenze con ognuna avente 8 basi.

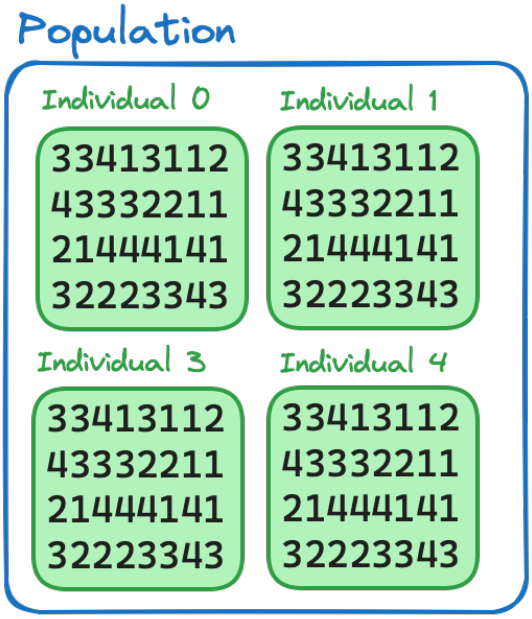

Figura 4.6: Popolazione dell'algoritmo genetico

Il numero di individui sarà poi configurabile dall'utente in base alle esigenze. In questa fase di generazione, vengono anche calcolati tutti i possibili range di dimensione $2x4$ sulla board (cioè proprio quei range per cui l'agente è addestrato). Ogni range viene calcolato in modo che siano range tutti diversi e che non ci sia overlap, per evitare che l'agente di RL vada ad operare in zone in cui ha già eseguito un allineamento su un altro individuo.

4.2.2 Fitness score

Il calcolo del fitness score non lo facciamo subito dopo la generazione della popolazione, poiché la popolazione nella fase iniziale è composta da individui tutti uguali, sono la stessa board replicata n volte (dove n può essere personalizzato dall'utente). La introduciamo ora perché è utile per capire come opererà la fase di mutazione. Il fitness score consiste essenzialmente nel calcolare la sum-of-pairs su ciscun individuo della popolazione, la formula utilizzata in questo caso è la stessa usata dal modello di RL su cui abbiamo testato il nostro approccio (quella definita in 3.1). E' importante notare che se si utilizza l'algoritmo con un modello di RL che utilizza pesi diversi per gap, match e mismatch, occorre modificare anche il calcolo della fitness score, impostando gli stessi pesi, per evitare incongruenze fra il peso che da l'agente di RL e quello che viene dato dall'algoritmo (questi valori sono facilmente modificabili dal file *config.py* dell'applicazione).

4.2.3 Mutazione

Dopo la generazione delle popolazione andiamo ad eseguire l'operazione di mutazione, e qui entra in gioco l'algoritmo di RL. Abbiamo previsto due differenti tipi di mutazioni, per valutarne le performance e dare la libertà all'utente di utilizzare quella più adatta in base al

caso d'uso. La prima è la *random mutation*, mentre la seconda è la mutazione sui *worst fitted individuals*.

1. *Random mutation*: la random mutation consiste nel selezionare in maniera random l'individuo nella popolazione da mutare. In base al numero di possibili sub-board diverse che abbiamo, andiamo ad eseguire lo stesso numero di mutazioni. Cioè, supponiamo di trovarci nel caso di esempio di una board $4x8$ come quella in figura 4.4 e di avere un agente di RL che è stato addestrato per operare su board $2x4$, quello che facciamo è generare tutti i range univoci sulla board $4x8$ (nel caso dell'esempio è facile vedere che sono 4) e per ogni range, selezioniamo a caso un individuo nella popolazione ed andiamo ad eseguire l'agente di RL su quell'individuo, in uno di quei range (anche il range su cui operare viene scelto casualmente). Eseguita la mutazione, tale range viene eliminato e si prosegue alla mutazione su un altro individuo su una sub-board che sarà quindi diversa. La fase di mutazione terminerà quando avremo esaurito tutti i possibili range univoci delle sub-board.

2. Mutazione sui *worst fitted individuals*: in questo caso, anziché selezionare in modo random l'individuo su cui operare, viene scelto quello che ha il punteggio della sum-of-pairs più alto. In particolare, prima di operare la mutazione, viene calcolata la sum-of-pairs per tutti gli individui, a questo punto vengono selezionato gli n individui migliori su cui fare la mutazione (cioè quelli con sum-of-pairs più alta). In questo caso il numero n di individui su cui eseguiamo la mutazione, non sarà in base al numero delle diverse sub-board che abbiamo, come nel caso precedente, ma sarà su una certa percentuale di individui nella popolazione (che può essere personalizzata opportunamente dall'utente in $config.py$). Per ciascuno di questi n individui selezionati che hanno sum-of-pairs più alta, la sub-board in cui andrà ad operare l'agente di RL, sarà quella dove la sum-of-pairs risulta più bassa. In altre parole, in questa fase di mutazione, dato un individuo (che è stato scelto perché aveva sum-of-pairs più alta su tutta la board), viene calcolata la sum-of-pairs anche per ognuna delle sub-board univoche, la sub-board che risulta avere sum-of-pairs più bassa, è quella in cui l'agente di RL andrà ad operare per cercare di migliorare quella soluzione. Da notare che sicuramente tale mutazione è molto più onerosa in termini computazionale, ma come vedremo nella sezione 5, porta notevoli benefici, portandoci ad avere un allineamento delle sequenze generalmente migliore (in termini di sum-of-pairs).

L'obiettivo di fornire questi due metodi diversi nasce dalla curiosità di vedere di quanto miglioriamo nel secondo caso rispetto al primo, ma anche per fornire all'utente la libertà di prediligere una maggiore velocità di calcolo dell'allineamento (e quindi usare il primo metodo) oppure una qualità dell'allineamento più alta a discapito delle prestazioni. In figura 4.7 possiamo vedere l'agente di RL che opera sull'individuo, eseguendo la mutazione.

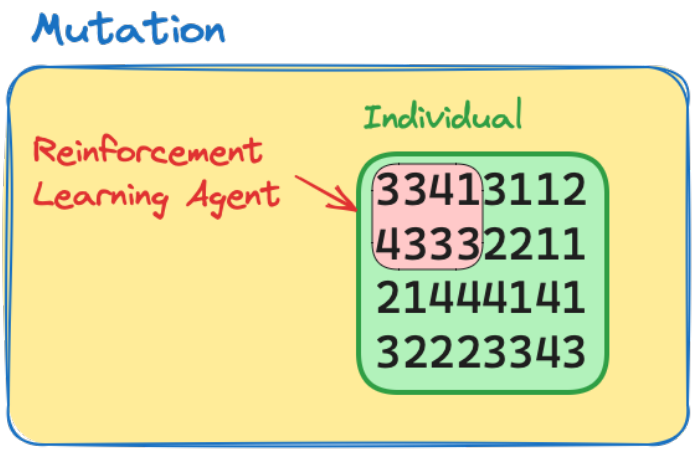

Figura 4.7: Mutazione

4.2.4 Selection

La fase di *Selection* del nostro algoritmo prevede il calcolo della sum-of-pairs per ogni individuo e di ordinarli in base al valore ottenuto. A questo punto solamente gli n individui, che hanno sum-of-pairs più alta saranno selezionati per generare dei nuovi individui e passare all'iterazione successiva (il valore n può essere personalizzato dall'utente in base all'esigenza ed a come risponde l'algoritmo al suo problema).

4.2.5 Crossover

Per la fase di crossover, abbiamo previsto due metodi diversi per la generazione del nuovo individuo, che sono *Vertical crossover* ed *Horizontal crossover*. Per entrambi i metodi, vengono scelti a coppie due individui (fra quelli che hanno passato la fase di *Selection*), chiamati *Parent 1* e *Parent 2*, e da ogni coppia verrà creato un nuovo individuo, fino a quando non arriviamo di nuovo al numero n di individui che abbiamo impostato all'inizio dell'algoritmo. La scelta del *Parent 1* e *Parent 2* è casuale per entrambi i metodi, fra gli individui che hanno passato la fase di selection.

- *Vertical crossover*: nel *vertical crossover*, presi *Parent 1* e *Parent 2*, vengono tagliati a metà verticalmente (la zona dove tagliare viene scelta facendo la media della lunghezza di ogni riga e si va a tagliare esattamente alla lunghezza media). Per generare il nuovo individuo, la prima metà viene presa dal *Parent 1* e la seconda metà dal *Parent 2*. Nella figura 4.8 viene mostrato un esempio di tale operazione.

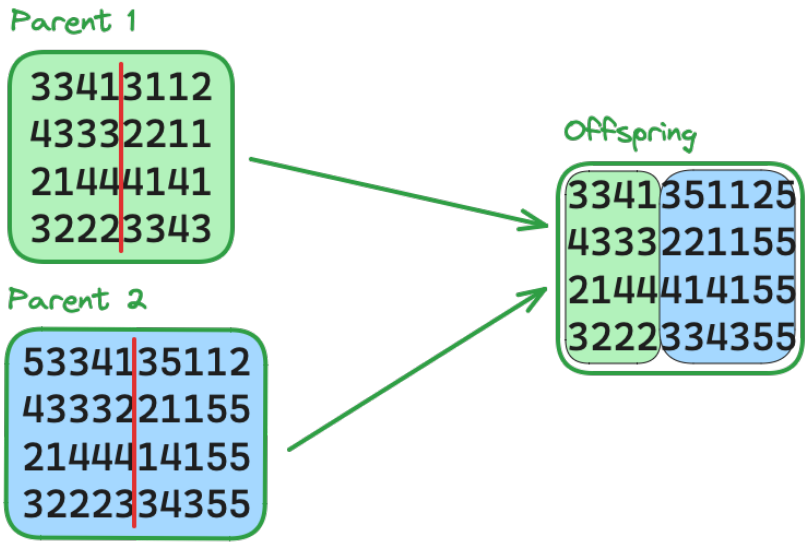

Figura 4.8: Vertical crossover

- *Horizontal crossover*: nell'*horizontal crossover* invece, il taglio dell'individuo avviene lungo le righe. In particolare, vengono contate il numero di righe della board, ed eseguita una divisione a metà della board lungo le righe (se dispari, dal *Parent 1* prenderemo una riga in più rispetto al *Parent 2*). A questo punto uniamo, prendendo la prima metà da *Parent 1* e la seconda metà da *Parent 2*. Nella figura 4.9 viene mostrato un esempio di questo crossover.

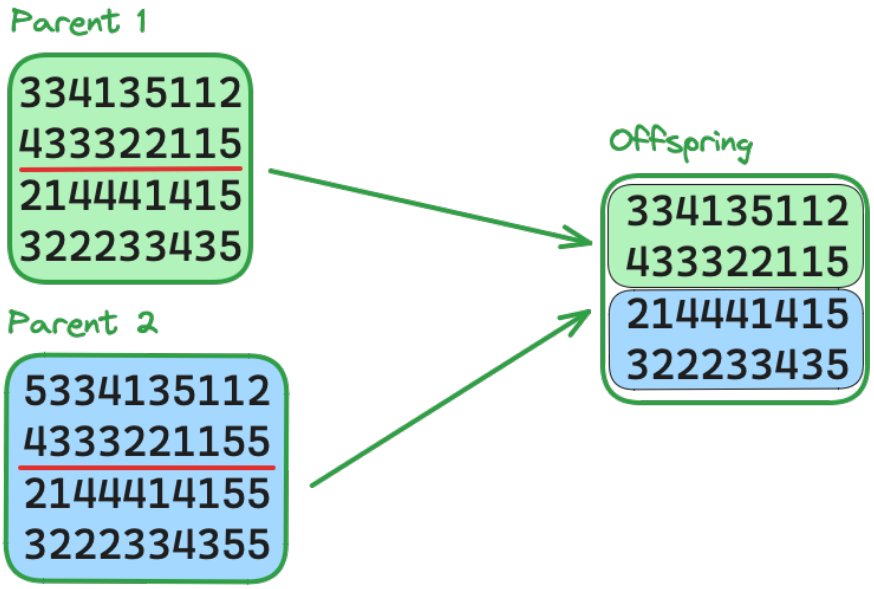

Figura 4.9: Horizontal crossover

4.2.6 Configurazione dell'algoritmo

Dal codice Python dell'applicazione è possibile personalizzare i seguenti parametri dell'algoritmo

```
GAP_PENALTY = -4
MISMATCH_PENALTY = -4
MATCH_REWARD = 4
GA_POPULATION_SIZE = 5
GA_NUM_ITERATION = 3
GA_NUM_MOST_FIT_FOR_ITER = 2
GA_PERCENTAGE_INDIVIDUALS_TO_MUTATE_FOR_ITER = 0.20 #20%

#This depend from the training dataset given to the DQN
AGENT_WINDOW_ROW = 3
AGENT_WINDOW_COLUMN = 30

DATASET_ROW = 6
DATASET_COLUMN = 30
```

Come possiamo vedere dal codice, è possibile personalizzare i valori usati per calcolare la sum-of-pairs (da notare che modificati questi, andrebbe riaddestrato il modello di RL). E' possibile modificare la taglia della popolazione e il numero di iterazioni dell'algoritmo genetico (dopo tale numero, l'algoritmo si arresta e viene restituito l'individuo con sum-of-pairs più alta). Con *GA_NUM_MOST_FIT_FOR_ITER* possiamo impostare quanti individui prendere dalla generazione attuale e farli passare alla generazione successiva (saranno anche quelli usati per generare nuovi individui). Con il paramentro *GA_PERCENTAGE_INDIVIDUALS_TO_MUTATE_FOR_ITER* inseriamo la percentuale di individui da mutare nella fase di mutazione (da notare che tale parametro viene utilizzato solo nel caso si usi la mutazione di tipo *worst fitted individuals*, vedi 4.2.3). Infine, possiamo modificare i parametri *AGENT_WINDOW_ROW* ed *AGENT_WINDOW_COLUMN* , che come si può intuire dal nome, corrisponde alla size della sub-board su cui l'agente di RL andrà a lavorare, quindi occorre modificarla in accordo al dataset di training dato al modello di RL. I valori *DATASET_ROW* ed *DATASET_COLUMN* vanno impostati sui valori del dataset che stiamo dando per fare inferenza sul modello, in modo da poter calcolare tutte le possibili differenti *sub-board* che ci possono essere.

Nella fase di mutazione come detto viene utilizzato il RL ed in particolare per il nostro esperimento abbiamo utilizzato il modello descritto in 3.1. Tuttavia il nostro algoritmo è agnostico rispetto al modello, ciò significa che l'unica modifica che occorre fare al codice nel caso si voglia utilizzare l'algoritmo con un altro modello, è relativa alla fase di mutazione, il resto del codice resta identico (nella repository GitHub maggiori informazioni).

4.2.7 Descrizione del flusso dell'algoritmo

Per l'esecuzione dell'algoritmo, dopo averlo opportunamente configurato, occorre lanciare lo script *mainGA.py*. Occorre fornire un dataset all'applicazione, che può essere inserito

nella cartella dataset, seguendo opportunamente la struttura degli altri dataset già presenti. Avviata l'applicazione, la prima cosa che viene fatta è generare la popolazione, dopodiché verranno eseguite un numero *n* di iterazioni in cui saranno eseguite le seguenti operazioni (dove *n* è il numero di iterazioni indicate nella configurazione dell'algoritmo):

1. Viene eseguita la mutazione sugli individui.
2. Si calcola il fitness score per ogni individuo.
3. Selection, vengono selezionati i migliori individui da far passare alla generazione successiva per generare i nuovi individui.
4. Crossover, creazione dei nuovi individui.

Al termine del numero di iterazioni *n*, si riesegue il calcolo del fitness score per ogni individuo (il ciclo termina con il crossover, quindi occorre vedere il fitness della generazione creata all'ultima iterazione). A questo punto dalla popolazione viene estratto l'individuo con la sum-of-pairs più alta, la sua board viene riconvertita da interi a caratteri (quindi in nucleotidi), e sia a video che in un file nella cartella *result* del progetto viene mostrata la sequenza allineata insieme al valore della sum-of-pairs.

Capitolo 5

Risultati

Per valutare la bontà del nuovo approccio proposto, sono stato eseguiti diversi test su diversi dataset, comparando i risultati ottenuti con alcuni dei tool più noti in letteratura che eseguono l'allineamento multiplo di sequenze. I tool che sono stati utilizzati per la comparazione sono:

- ClustalΩ[8]: utilizza un metodo di allineamento progressivo migliorato con guide-tree (albero guida) e modelli di Markov nascosti (HMM). Questo permette una maggiore precisione nell'allineamento di sequenze multiple.
- MSAProbs[5]: fa parte della famiglia degli algoritmi che utilizzano metodi probabilistici ed utilizza i Modelli di Markov nascosti per calcolare il livello di probabilità di allineamento fra sequenze.
- ClustalW[3]: fa parte della famiglia di algoritmi di allineamento progressivo usando una matrice di distanze e la costruzione di un albero guida.

E' stato eseguito anche un test di comparazione con il tool DPAMSA, con e senza l'utilizzo dell'algoritmo genetico, usando in entrambi i casi un agente addestrato sulla board della stessa dimensione, questo significa che nell'algoritmo genetico, nella fase di mutazione, l'agente lavorerà sull'intera board, in modo da analizzare i vantaggi che vengono ottenuti solo dall'applicazione dell'algoritmo genetico.

Prima di eseguire il testing dell'algoritmo genetico sul modello si è eseguito il training del modello DPAMSA, su due dataset diversi, con all'interno sequenze sintetiche. I dataset di training utilizzati si trovano su GitHub nella cartella

`dataset/training_dataset/`

Ogni dataset, contiene 50 test, dove ogni test rappresenta un allineamento che il modello deve eseguire (i parametri per il dataset di training sono stati ripresi dal paper di riferimento del modello DPAMSA [6], non è stato possibile utilizzare gli stessi dataset di training in quanto non sono stati forniti). Nella tabella 5.1 sono riportati tutti i dettagli sui due dataset utilizzati per addestrare il modello. Mentre nella tabella 5.2 sono riportati gli iperparametri usati per il training del modello (anche questi sono stati ripresi dal paper [6])

Nome dataset	Numero di test	Numero di sequenze per ogni test	Lunghezza di ogni sequenza (bp)
training_dataset1_3x30bp	50	3	30
training_dataset1_6x30bp	50	6	30

Tabella 5.1: Informazioni sul dataset usato per il training

Hyperparameter	Valore
bath size	128
replay memory size	1000
episode, number of iterations of training	6000 (max)
update iteration of target Q network	128

Tabella 5.2: Iperparametri DPAMSA

Nella tabella 5.3 e5.4 vengono riportati rispettivamente i tempi per addestrare i due modelli e l'hardware utilizzato:

Nome dataset	Tempo impiegato per il training (ore)
dataset1_3x30bp	26
dataset1_6x30bp	51

Tabella 5.3: Tempo impiegato per addestrare DPAMSA

CPU	GPU	RAM
Intel(R) Xeon(R) Gold 5218 CPU @ 2.30 GHz	NVIDIA Quadro RTX 4000	232 GB

Tabella 5.4: Hardware utilizzato per il training

Infine, sia per la fase di training, che nella fase di inferenza per valutare la bontà dell'algoritmo, vengono utilizzati i valori della sum-of-pairs in tabella 5.5. Da notare che se questi vengono cambiati dopo il training, il modello dovrebbe essere riaddestrato, perchè vengono utilizzati anche come reward per l'agente di RL.

Match	Mismatch	Gap
4	-4	-4

Tabella 5.5: Valori della sum-of-pairs

Dopo la fase di training, abbiamo utilizzato dataset differenti, sempre con sequenze sintetiche, per eseguire l'inferenza sul modello e testare la bontà del risultato. Ogni dataset generato, come per quelli di training, contengono 50 test, dove ogni test è formato da delle sequenze da allineare. Per ogni dataset, sono stati generati anche i relativi file fasta, in modo da poterli dare in input ai tool *ClustalΩ*, *ClustalW* ed *MSAProbs* per comparare i risultati. Per

valutare la qualità dell'allineamento, viene utilizzata la stessa funzione di calcolo della sum-of-pairs (SP), sia nel caso in cui l'output proviene dai tool menzionati precedentemente, sia nel caso degli allineamenti in output da DPMSA e da DPMSA con algoritmo genetico (da adesso in avanti indicato come GA-DPAMSA). Il valore, in generale, più è alto più l'allineamento è migliore (quindi in presenza di valori negativi, più siamo vicino allo 0, migliore sarà l'allineamento). Nella tabella 5.6 sono presenti i parametri utilizzati per l'algoritmo genetico durante tutti i test. Sono stati eseguiti anche dei test con il vertical crossover e con la random-mutation, che è stato visto danno tendenzialmente allineamenti con punteggio della SP più basso rispetto all'horizontal crossover con worst-fitted-individuals-mutation, per questo la comparazione con i tool in letteratura è stata fatta con quest'ultimi. Tuttavia nella sezione 5.4 vedremo una comparazione tra i diversi metodi, concentrandoci solo su GA-DPAMSA.

Parametro algoritmo genetico	Valore
Size popolazione	5
Numero di iterazioni	3
Numero di individui Most fitted da propagare nell'iterazione successiva	2
Percentuale di individui da mutare per iterazione[1]	20%
Tipo di crossover usato	horizontal crossover
Tipo si mutazione usato	worst-fitted-individuals

Tabella 5.6: Parametri algoritmo genetico

Dopo vari tentativi, i parametri illustrati nella tabella 5.6, sembrano quelli che offrono un buon bilanciamento fra tempo di ottenimento del risultato e score finale della sum-of-pairs. Tuttavia, vale la pena considerare che modificandoli, è possibile ottenere anche valori della sum-of-pair migliori (in alcuni casi) o anche peggiori, se non si bilancia bene la size della popolazione con il numero di iterazioni. Infatti, avendo un elevato numero di iterazioni e un numero di individui basso, quello che accade è che la soluzione potrebbe peggiorare nelle diverse iterazioni, in quanto si potrebbe arrivare ad un allineamento considerato "buono" dal modello di RL, ma se quell'individuo viene selezionato per la mutazione (ad esempio nella random mutation la scelta è casuale) l'agente di RL potrebbe peggiorare la soluzione. Nella mutazione di tipo worst-fitted questo fenomeno è meno presente, in quanto l'individuo migliore tende a non essere mutato e quindi propagato attraverso le iterazioni. I dataset utilizzati per ottenere i risultati che illustreremo nelle prossime sezioni, sono tutti disponibili su GitHub nella directory *datasets/inference_dataset*. Mentre nella cartella *datasets/fasta_files* ci sono tutti i file fasta, divisi per directory in base al dataset, in modo da poterli dare in input a *ClustalΩ*, *ClustalW* ed *MSAProbs*

[1] solo nel caso della mutazione worst-fitted

5.1 Risultati su dataset 3x30bp

I primi risultati che mostreremo, riguardano il dataset *dataset1_3x30bp*, cioè ogni test nel dataset avrà 3 sequenze, dove ogni sequenza è formata da 30 basi. Da notare che in questo caso, sia per DPAMSA che per GA-DPAMSA utilizzano il modello di RL addestrato su un dataset $3x30bp$, quindi in questo caso vogliamo solamente testare i benefici che ci vengono dati dal solo algoritmo genetico, senza testare il meccanismo di lavoro su una sub-board (l'agente di RL in questo caso lavorerà su tutta la board). Nella figura 5.1 possiamo vedere lo score medio della SP, in rosso il modello con il nostro algoritmo genetico. I valori utilizzati per il calcolo della SP sono quelli in tabella 5.5 (più siamo vicino allo 0, migliore è l'allineamento).

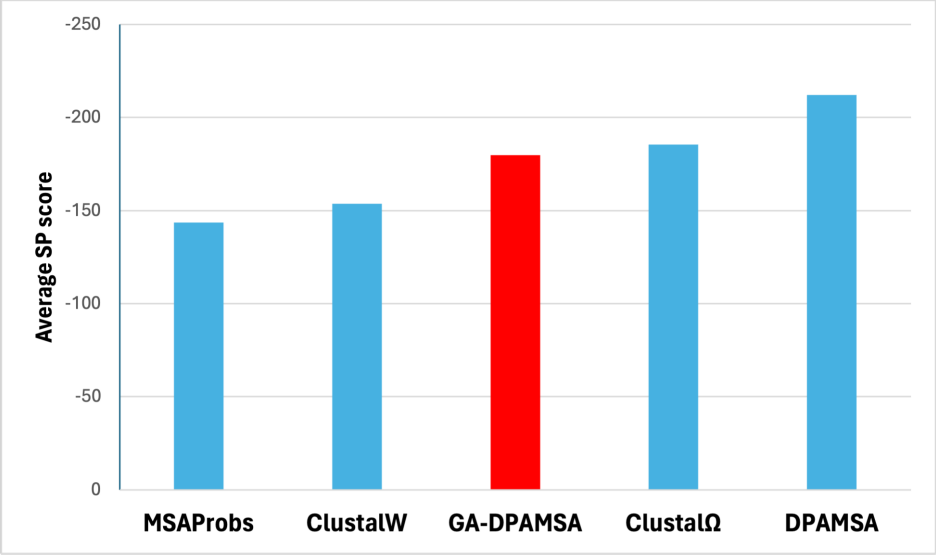

Figura 5.1: Punteggio medio della SP

Mentre in figura 5.2 è presente un box-plot con i valori calcolati

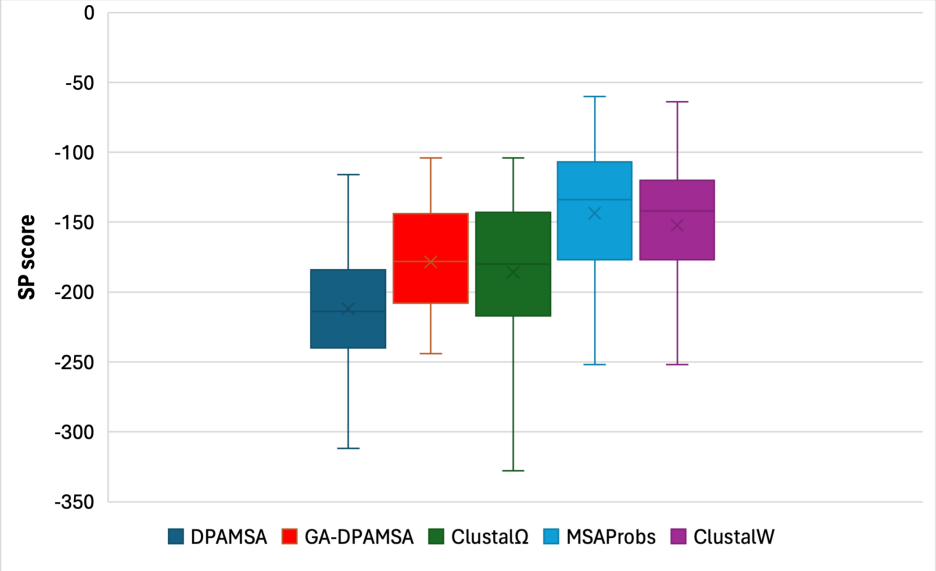

Figura 5.2: Boxplot della SP

Infine nella tabella 5.7 sono presenti i dettagli sulla media, massimo, minimo, mediana e deviazione standard sullo score della SP sui 50 test presenti nel dataset *dataset1_3x30bp*.

Tool	Media	Mediana	Deviazione standard	Minimo	Massimo
MSAProbs	-143,52	-134	47,17	-252	-60
ClustalW	-152,24	-142	43,24	-252	-64
GA-DPAMSA	-179,75	-178	37,88	-244	-104
ClustalΩ	-185,76	-180	54,96	-328	-104
DPAMSA	-211,84	-214	38,94	-312	-116

Tabella 5.7: Risultati dataset1_3x30b

Dai dati raccolti da questo primo test, possiamo sicuramente affermare che con il nostro algoritmo genetico, abbiamo ottenuto dei miglioramenti rispetto al tool DPAMSA e ci avviciniamo di molto alle prestazioni di ClustalΩ, infatti lo score medio della SP è più alto rispetto a DPAMSA e con ClustalΩ abbiamo solamente una differenza di $-3,52$.

5.2 Risultati su dataset 6x30bp

I prossimi risultati riguardano il dataset *dataset1_6x30bp*, dove ogni test ha 6 sequenze da allineare ed ognuna è fata da 30 nucleotidi. In questo caso dapprima vedremo le prestazioni di GA-DPAMSA usando il modello di RL addestrato su dataset 3x30 (quindi in questo caso utilizzerà il concetto di sub-board) e poi lo compareremo con DPAMSA addestrato su un dataset 6x30 e GA-DPAMSA con dataset 6x30 (quindi senza uso di sub-board).

Consideriamo in figura 5.3 il caso in cui per GA-DPAMSA usiamo il modello addestrato su un dataset 3x30bp, quindi in questo caso, utilizzerà la sub-board. Il grafico mostra il punteggio medio della SP totalizzato dai vari tool.

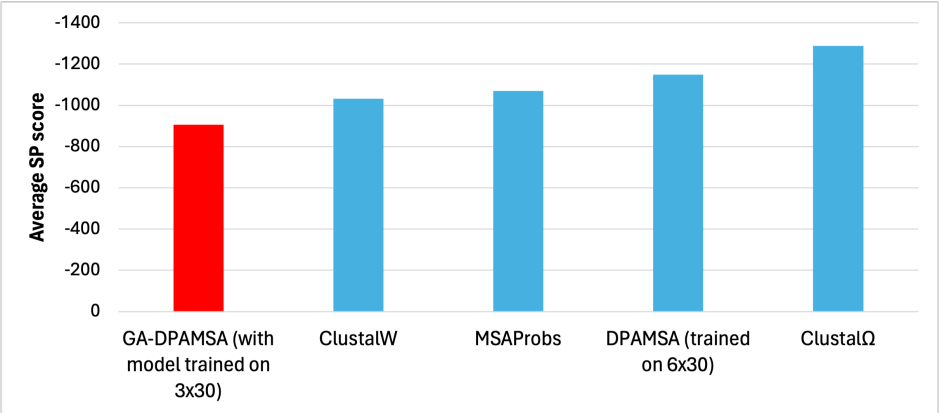

Figura 5.3: Punteggio medio della SP - Dataset 6x30bp

Vediamo come il nostro tool in questo caso risulta il migliore, con poca differenza rispetto a *ClustalW*. Nella tabella 5.8 vengono riportati i dettagli sulla media, massimo, minimo, mediana e deviazione standard del punteggio della SP.

Tool	Media	Mediana	Deviazione standard	Minimo	Massimo
GA-DPAMSA (with model trained on 3x30)	-906,80	-904	-83,16	-1112	-704
ClustalW	-1032,24	-1012	189,37	-1644	-636
MSAProbs	-1070,40	-1036	146,48	-1692	-900
DPAMSA (with model trained on 6x30)	-1149,52	-1140	146,48	-1692	-900
ClustalΩ	-1287,52	-1232	-305,23	-2264	-876

Tabella 5.8: Risultati dataset1_6x30bp

Inoltre, è importante notare che risulta essere migliore anche di DPAMSA, addestrato su un dataset 6x30. Questo è molto importante, perché significa che possiamo risparmiare tempo e risorse per l'addestramento del modello, ed otteniamo comunque ottimi risultati (in questo caso addirittura migliori).

In figura 5.4 vediamo anche dal box-plot come DPAMSA con il nostro algoritmo genetico risulta essere migliore di tutti gli altri tool testati.

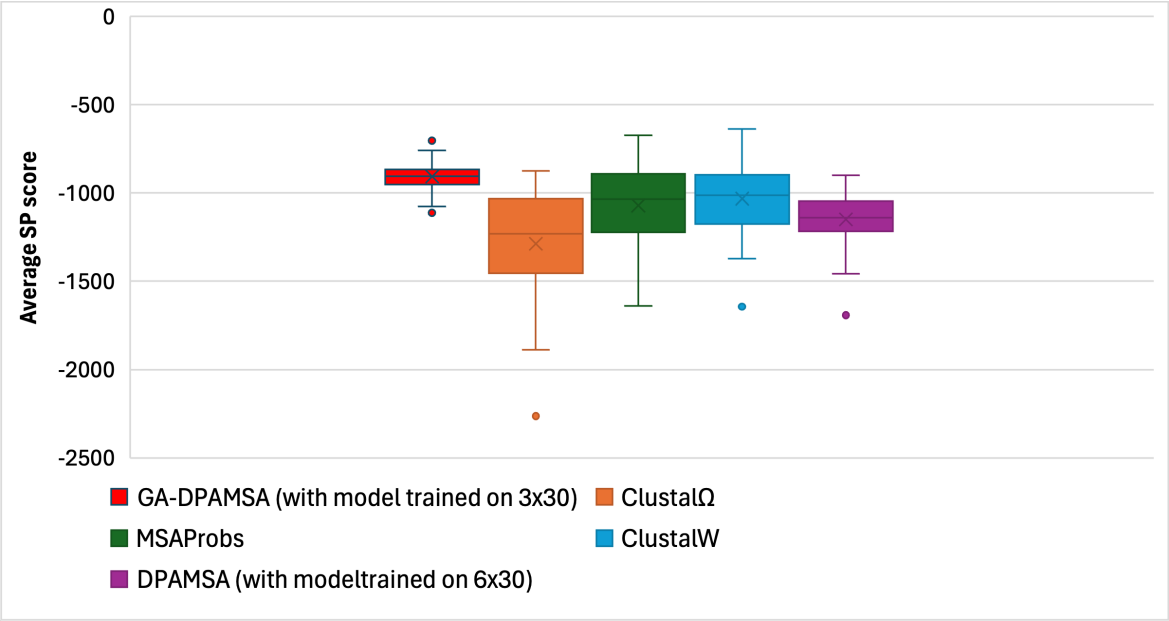

Figura 5.4: Box plot - Dataset 6x30bp

Visto le ottime performance del nostro tool, abbiamo deciso quindi di vedere cosa succede se applichiamo l'algoritmo genetico al modello DPAMSA addestrato su un dataset 6x30, per vedere le performance rispetto a GA-DPAMSA addestrato su 3x30. In figura 5.5 vediamo il punteggio medio della SP calcolata rispettivamente su: GA-DPAMSA con modello di RL addestrato su dataset 3x30, GA-DPAMSA addestrato su dataset 6x30 e DPAMSA addestrato su dataset 6x30.

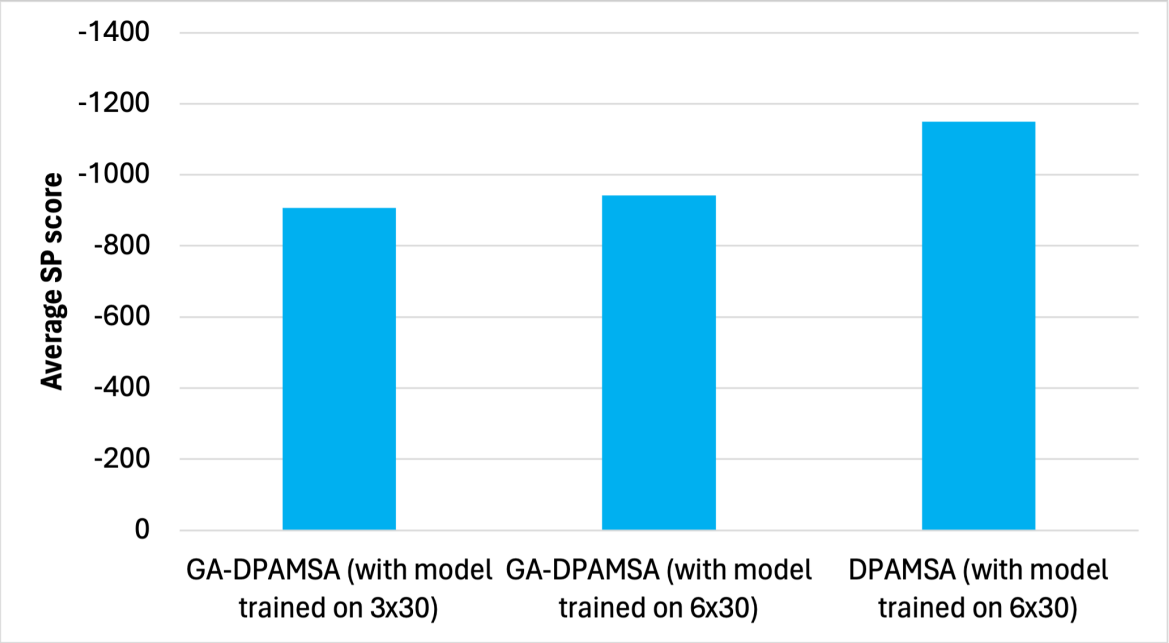

Figura 5.5: DPAMSA vs GA-DPAMSA

Nella tabella 5.9 vediamo nello specifico i valori per la media, mediana, deviazione standard, massimo e minimo.

Tool	Media	Mediana	Deviazione standard	Minimo	Massimo
GA-DPAMSA (with model trained on 3x30)	-906,80	-904	83,16	-1112	-704
GA-DPAMSA (with model trained on 6x30)	-941,20	-916	130,57	-1268	-688
DPAMSA (with model trained on 6x30)	-1149,52	-1140	146,48	-1692	-900

Tabella 5.9: DPAMSA vs GA-DPAMSA

Come possiamo notare, utilizzando l'algoritmo genetico, otteniamo un miglioramento rispetto a DPAMSA, quindi questo conferma anche quanto visto in 5.1. La cosa interessante è che GA-DPAMSA è in grado di avere le stesse prestazione su un dataset su 6x30bp, sia se in fase di training è stato addestrato con un dataset 3x30bp sia con uno 6x30bp, anzi, in questo test, GA-DPAMSA addestrato con dataset 3x30 è risultato essere leggermente migliore, anche se di poco. In figura 5.6 viene riportato infine anche il box plot, che conferma che quest'ultima configurazione risulta essere la migliore.

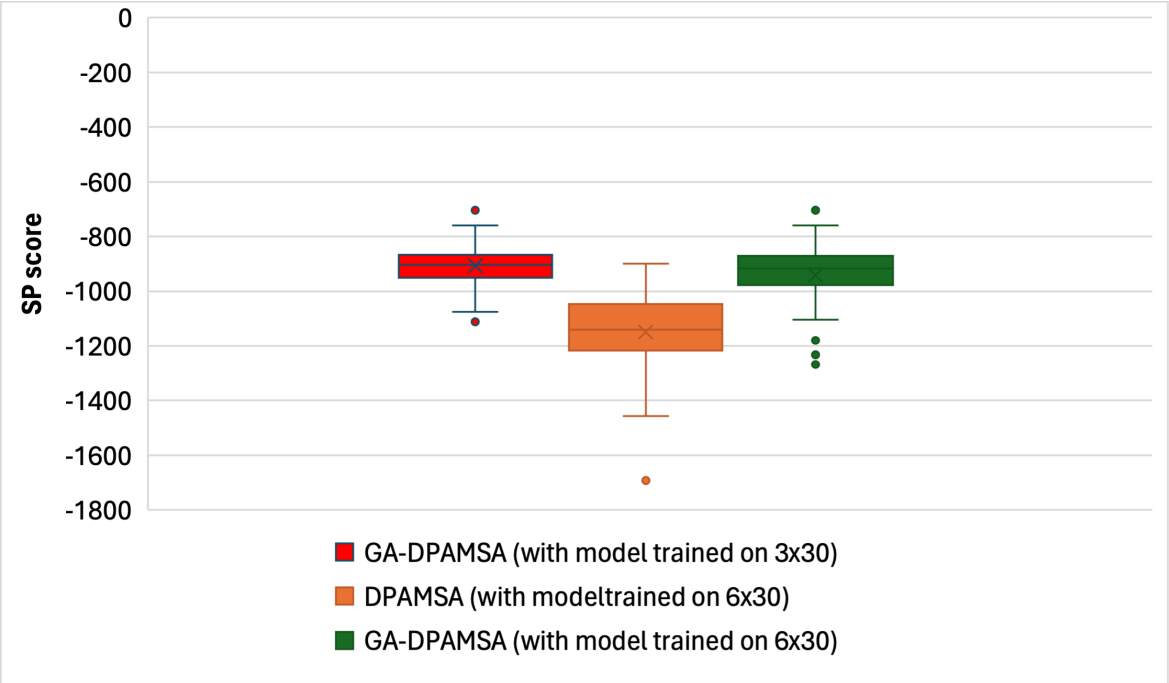

Figura 5.6: Box plot - DPAMSA vs GA-DPAMSA

5.3 Risultati su dataset 6x60bp

In questo tipo di test, quello che si voleva verificare è la bontà dell'algoritmo al crescere della grandezza del problema, restando con un modello di RL addestrato per problemi piccoli. Infatti nei risultati che ora illustreremo, il modello di RL utilizzato da GA-DPAMSA è quello addestrato su dataset 3x30 (che abbiamo visto prima essere anche leggermente migliore rispetto a quello addestrato su 6x60). Nella figura 5.7 è presente il grafico che ci mostra lo score medio della SP sui 4 tool, mentre in tabella 5.10 i dati più nel dettaglio.

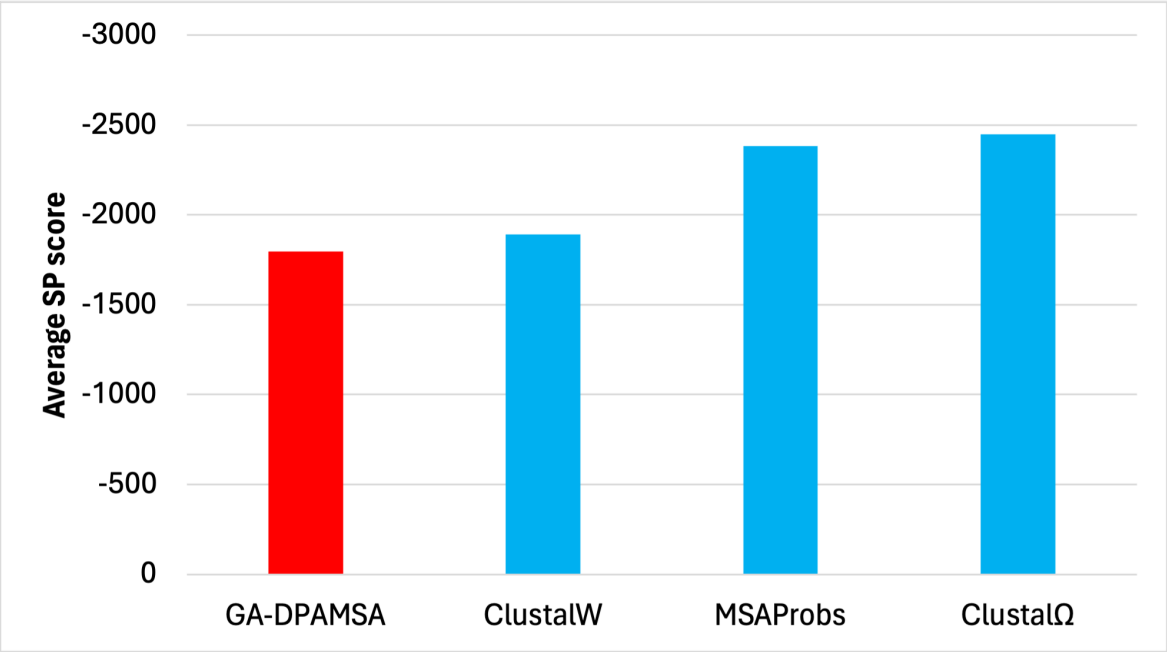

Figura 5.7: Risultati dataset 6x60

Tool	Media	Mediana	Deviazione standard	Minimo	Massimo
GA-DPAMSA	-1795,52	-1784	104,70	-1984	-1536
ClustalW	-1890,56	-1858	263,96	-2544	-1360
MSAProbs	-2384,32	-2322	413,79	-3380	-1748
ClustalΩ	-2449,28	-2424	428,97	-3336	-1600

Tabella 5.10: Risultati dataset1_6x60bp

Infine, anche in questo caso viene mostrato il box-plot in figura 5.8.

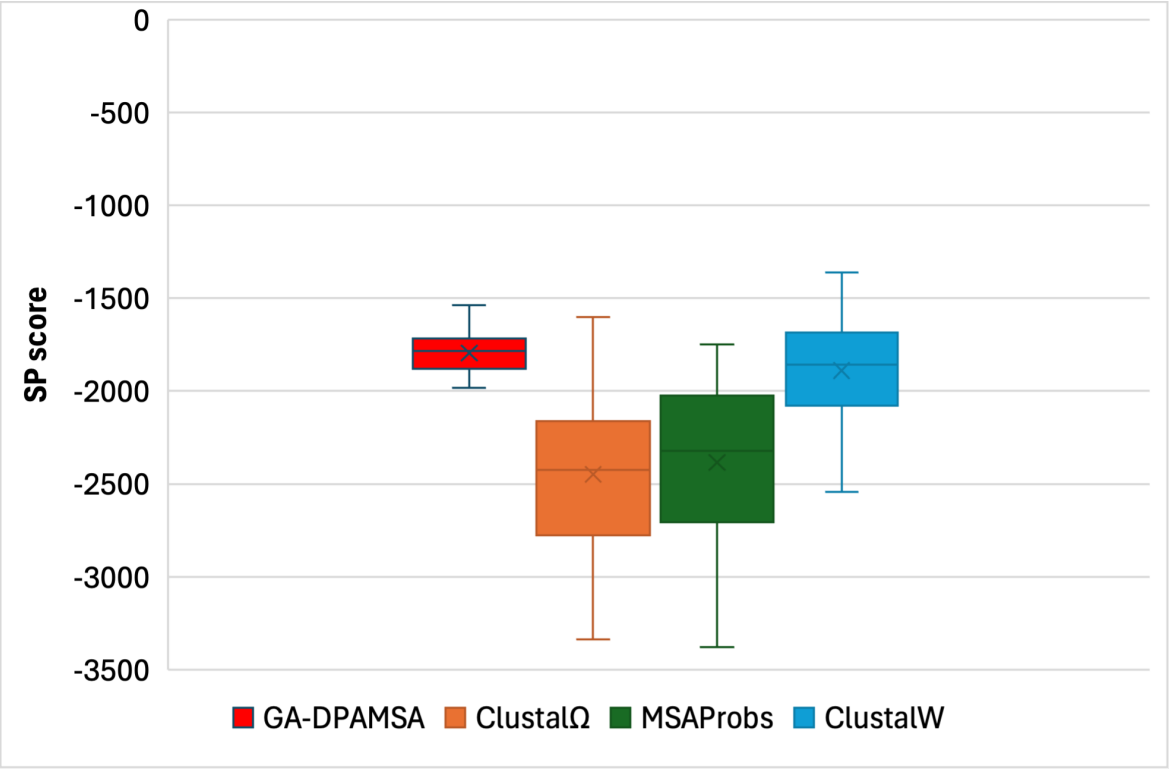

Figura 5.8: Box-plot dataset 6x60

Analizzando i dati nella tabella e nei grafici, risulta chiaro come anche in questo caso, il nostro tool risulta essere ancora il migliore, con ClustalW molto vicino in termini di qualità dell'allineamento.

5.4 Risultati sulle varie configurazioni GA-DPAMSA

In questa sezione, verranno illustrati i risultati ottenuti con GA-DPAMSA al variare del meccanismo di crossover (horizontal-crossover o vertical-crossover) e della mutazione (random mutation e worst-fitted mutation). I dati che illustreremo di seguito sono relativi ad un test eseguito con il dataset *dataset2_6x60bp*, con modello di RL addestrato su dataset 3x30. Anche per questo test vengono riportati: il grafico del punteggio medio della SP 5.9, la tabella con i dati di media, mediana, deviazione standard, minimo e massimo 5.11 ed infine il box plot con i valori della SP 5.10. Nella figura 5.9, le barre celesti fanno riferimento al vertical crossover, mentre le rosse all'horizontal (al variare del tipo di mutazione).

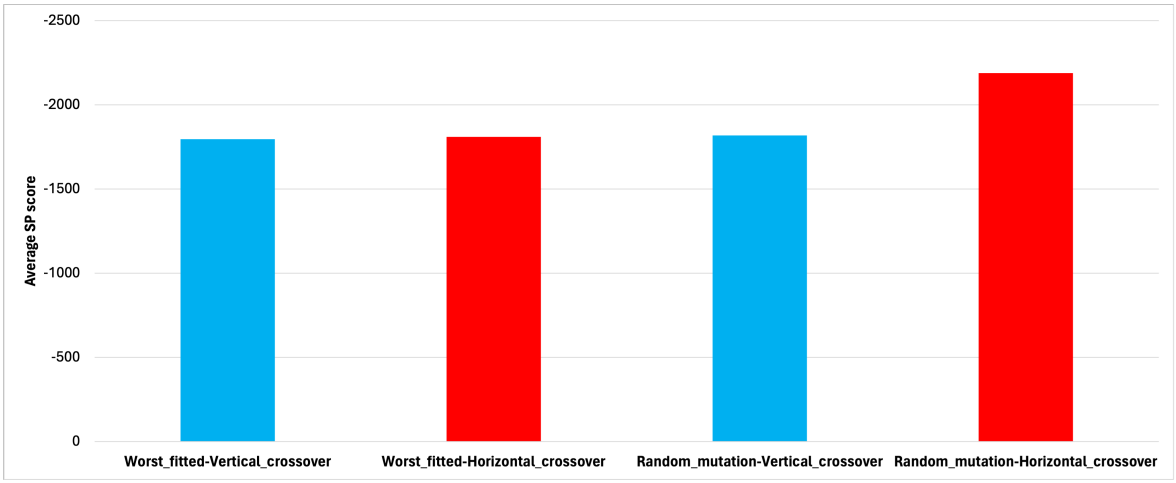

Figura 5.9: GA-DPAMSA score SP medio sulle differenti configurazioni

Tool	Media	Mediana	Deviazione standard	Minimo	Massimo
Worst fitted-Vertical crossover	-1795,52	-1784	104,70	-1984	-1536
Worst fitted-Horizontal crossover	-1809,44	-1800	108,65	-1984	-1588
Random mutation-Vertical crossover	-1817,76	-1812	100,92	-2068	-1604
Random mutation-Horizontal crossover	-2187,04	-2090	352,73	-3456	-1780

Tabella 5.11: GA-DPAMSA - risultati sulle diverse configurazioni

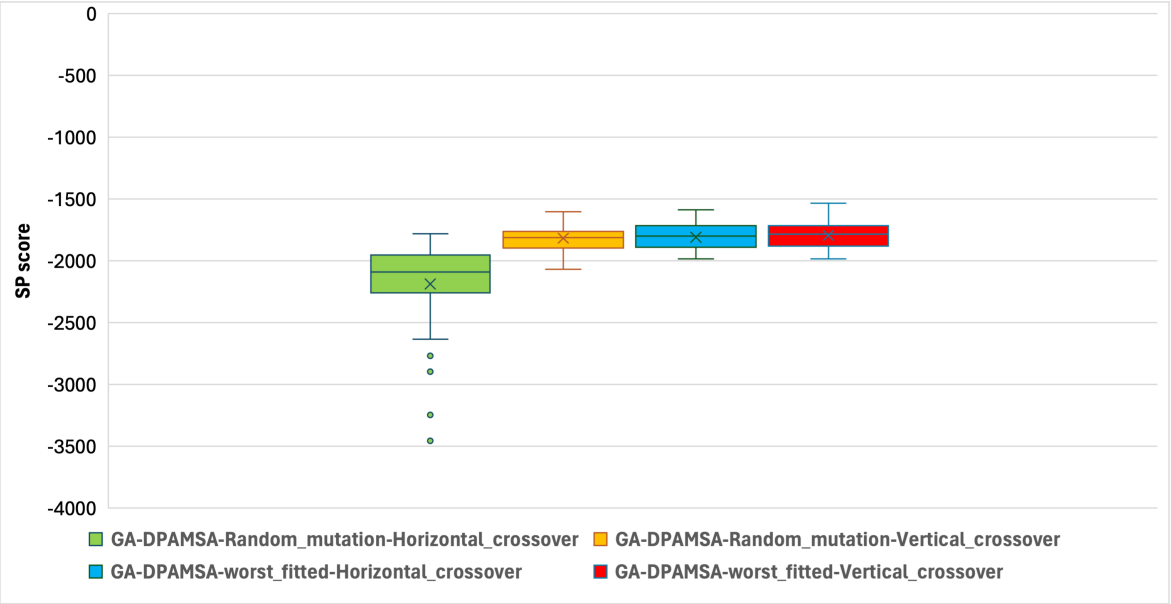

Figura 5.10: GA-DPAMSA box-plot sulle differenti configurazioni

Analizzando i risultati, possiamo concludere che come ci aspettavamo, la worst fitted mutation ci permette di avere un allineamento con un punteggio della SP più alto. Questo perché ad ogni iterazione, l'agente di RL lo facciamo operare sull'individuo con fitness score più basso, nella sub-board con fitness score più basso, quindi cerchiamo sempre di

migliorare le varie soluzioni. Tuttavia, la random mutation risulta essere comunque un buon compromesso, perché permette di ottenere i risultati più velocemente, in quanto non abbiamo bisogno del calcolo della SP su tutte le sub-board per capire dove operare, ed offre comunque dei risultati che non sono molto lontani dalla worst fitted mutation, infatti risulta esserci uno scarto solamente di circa 38 nel valore medio del punteggio della SP. Per quanto riguarda i differenti tipi di crossover, non ci aspettavamo delle differenze marcate, ed infatti, considerando il caso della worst fitted mutation, tra i due tipi di crossover abbiamo uno scarto di solamente 17 punti nello score della SP. Ovviamente ripetendo il test e cambiando i parametri dell'algoritmo genetico i risultati possono variare significativamente, infatti con un numero di interazioni più alto è stato osservato che la random mutation tende a diventare più distante in termini di SP rispetto all'altro tipo, come spiegato nel dettaglio nell'introduzione del capitolo 5.

5.5 Considerazioni

Alla luce del test eseguito, guardando i risultati ottenuti, possiamo sicuramente concludere che l'esperimento è risultato efficace e l'obiettivo è stato raggiunto. Infatti, con l'algoritmo genetico riusciamo a migliorare il modello base, ma riusciamo anche con un modello addestrato su un problema più piccolo, a risolvere un problema più grande, in maniera più precisa in termini di score della SP rispetto al modello base addestrato per risolvere il problema con le stesse dimensioni. Questo probabilmente accade perchè l'agente di RL può limitarsi a lavorare in una finestra ristretta e quindi tende ad essere più preciso nelle azioni, mentre avendo l'intera board, potrebbe avere troppe scelte da considerare e quindi portare ad un allineamento peggiore.

Non solo, possiamo concludere che all'aumentare della grandezza del problema, il nostro approccio risulta essere anche migliore in confronto ai 3 tool con cui ci siamo confrontati. Solamente nel caso del dataset 3x30bp non siamo risultati i migliori, ma comunque eravamo molto vicini a $Clustal\Omega$. Poi però al crescere del numero di sequenze nel problema e al numero di basi in ogni sequenza, il nostro tool riesce a calcolare un allineamento con SP più alta rispetto a tutti gli altri 3 tool.

Capitolo 6

Sviluppi futuri

Come lavoro futuro, c'è sicuramente l'obiettivo di provare ad utilizzare l'algoritmo genetico su un modello di RL diverso, per confermare i buoni risultati ottenuti. Inoltre, sarebbe interessante testare il modello su sequenze biologiche reali con molte basi e confrontarsi con gli altri tool in letteratura, per vedere dove riusciamo a posizionarci e se ci confermiamo essere i migliori su problemi grandi. Un altro esperimento interessante sarebbe testare l'algoritmo genetico utilizzando un approccio ibrido per la mutazione ed il crossover, ad esempio, potrei scegliere a caso, con probabilità $\frac{1}{2}$, se in una determinata iterazione utilizzare il vertical crossover o l'horizontal, stessa cosa per la mutazione, se usare la random mutation o la worst-fitted. Infine, l'approccio si potrebbe estendere non solo per l'allineamento di DNA, ma anche di proteine ed RNA.

Capitolo 7

Conclusioni

Eravamo partiti con l'idea di realizzare un algoritmo genetico per un modello di RL, in modo da verificare se questo ci desse un miglioramento in termini di qualità dell'allineamento. Ma non solo, l'obiettivo era anche di utilizzarlo per far in modo che l'agente di RL, se addestrato su una board $M \cdot N$, potesse lavorare anche su una board $M_1 \cdot N_1$ dove risulta $M_1 \cdot N_1 > M \cdot N$, senza dover riaddestrare il modello, questo perché se la matrice $M_1 \cdot N_1$ è molto grande, ciò richiede molto tempo e risorse di computazione molto elevate. Alla luce dei risultati ottenuti possiamo sicuramente affermare che entrambi gli obiettivi sono stati raggiunti e ciò ci suggerisce che l'uso di algoritmi genetici insieme al RL, può essere un ambito di ricerca promettente per l'allineamento multiplo di sequenze, che vale la pena continuare ad approfondire.

Bibliografia

[1] Reza Jafari, Mohammad Masoud Javidi, and Marjan Kuchaki Rafsanjani. Using deep reinforcement learning approach for solving the multiple sequence alignment problem. *SN Applied Sciences*, 1:1–12, 2019.

[2] Aryan Lall and Siddharth Tallur. Deep reinforcement learning-based pairwise dna sequence alignment method compatible with embedded edge devices. *Scientific Reports*, 13(1):2773, 2023.

[3] M.A. Larkin, G. Blackshields, N.P. Brown, R. Chenna, P.A. McGettigan, H. McWilliam, F. Valentin, I.M. Wallace, A. Wilm, R. Lopez, J.D. Thompson, T.J. Gibson, and D.G. Higgins. Clustal W and Clustal X version 2.0. *Bioinformatics*, 23(21):2947–2948, 09 2007.

[4] Yarong Li. Sequence alignment with q-learning based on the actor-critic model. *Transactions on Asian and Low-Resource Language Information Processing*, 20(5):1–7, 2021.

[5] Yongchao Liu, Bertil Schmidt, and Douglas L Maskell. Msaprobs: multiple sequence alignment based on pair hidden markov models and partition function posterior probabilities. *Bioinformatics*, 26(16):1958–1964, 2010.

[6] Yuhang Liu, Hao Yuan, Qiang Zhang, Zixuan Wang, Shuwen Xiong, Naifeng Wen, and Yongqing Zhang. Multiple sequence alignment based on deep reinforcement learning with self-attention and positional encoding. *Bioinformatics*, 39(11):btad636, 2023.

[7] Ramchalam Kinattinkara Ramakrishnan, Jaspal Singh, and Mathieu Blanchette. Rlalign: a reinforcement learning approach for multiple sequence alignment. In *2018 IEEE 18th International Conference on Bioinformatics and Bioengineering (BIBE)*, pages 61–66. IEEE, 2018.

[8] Fabian Sievers and Desmond G Higgins. Clustal omega. *Current protocols in bioinformatics*, 48(1):3–13, 2014.

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.