



Università degli Studi di Milano Bicocca

Scuola di Scienze

Dipartimento di Informatica, Sistemistica e Comunicazione

Corso di laurea in Informatica

Allineamento wavefront per l'individuazione di ricombinazioni

Relatore: Gianluca Della Vedova

Co-relatore: Paola Bonizzoni

Relazione della prova finale di:

Volpato Mattia

Matricola 866316

Anno Accademico 2022-2023

Abstract

L'**allineamento di sequenze genomiche** è una parte fondamentale della biologia molecolare moderna, che permette di individuare regioni di somiglianza all'interno delle sequenze nucleotidiche. I progressi degli ultimi anni nelle tecnologie di sequenziamento hanno permesso di avere a disposizione sequenze di lunghezza sempre maggiore, oltre a una quantità notevole di dati; per riuscire a sfruttare a pieno questi progressi tecnologici è necessario riuscire a sviluppare algoritmi di allineamento più veloci ed efficienti, che riescano a scalare con la crescita delle sequenze da allineare. In questo lavoro viene presentato l'**algoritmo *wavefront***, un nuovo approccio di programmazione dinamica per l'allineamento di sequenze basato sull'astrazione del *fronte d'onda*: nella prima parte viene riportata la letteratura sul tema dell'allineamento, partendo da coppie di sequenze e successivamente generalizzando a strutture a grafo; nella seconda parte viene presentato l'algoritmo ***wavefront***, partendo sempre dal caso più semplice di due sequenze per poi passare a estensioni sui grafi; infine, nella terza parte viene presentato un prototipo realizzato per l'implementazione dell'algoritmo, procedendo con un confronto con *Recgraph*, un tool sviluppato in precedenza dal laboratorio BIAS che effettua allineamento con metodologie standard.

Indice

Abstract

1	Introduzione	1
1.1	La bioinformatica	1
1.2	Applicazioni dell'allineamento di sequenze	1
1.3	Tecnologie di sequenziamento	2
1.4	Obiettivo dello stage	4
2	Allineamento di sequenze	5
2.1	Allineamento tra due sequenze	5
2.1.1	Equazioni di ricorrenza	6
2.1.2	Ricostruzione della soluzione	7
2.1.3	Complessità computazionale	7
2.1.4	Caso globale	8
2.1.5	Caso semiglobale	9
2.2	Distanza di edit tra due sequenze	10
2.2.1	Equazioni di ricorrenza	11
2.2.2	Ricostruzione della soluzione	12
2.2.3	Complessità computazionale	13
2.2.4	Caso pesato	13
2.3	Estensione all'allineamento tra un grafo e una sequenza	14
2.3.1	Grafi di sequenza	15
2.3.2	Partial Order Alignment (POA)	15
2.3.3	Grafi di variazione	19
2.3.4	Allineamento tra un grafo di variazione e una sequenza	20
2.4	Allineamento con ricombinazione	22
2.4.1	Definizioni	22
2.4.2	Equazioni di ricorrenza	23
2.4.3	Complessità computazionale	24
2.4.4	RecGraph	25
3	Algoritmo wavefront per la distanza di edit	26
3.1	Algoritmo wavefront per la distanza di edit tra due sequenze	26

3.1.1	Equazioni di ricorrenza	28
3.1.2	Condizioni al contorno	30
3.1.3	Pseudocodice	30
3.1.4	Complessità computazionale	33
3.1.5	Dimostrazione di correttezza	37
3.1.6	Caso pesato	40
3.2	Distanza di edit su grafi di variazione con wavefront	41
3.2.1	Equazioni di ricorrenza	41
3.2.2	Condizioni al contorno	42
3.2.3	Complessità computazionale	43
4	Progettazione e implementazione del prototipo	44
4.1	Progettazione	44
4.1.1	Organizzazione modulare	44
4.1.2	Integrazione in RecGraph	45
4.2	Implementazione	46
4.2.1	Interfaccia <i>wavefront</i>	47
4.2.2	Modulo <i>wf_implementation</i>	47
4.2.3	Modulo <i>wf_alignment</i>	48
4.3	Benchmark	50
4.3.1	RecGraph vs Wavefront	50
4.3.2	Analisi dell'andamento del prototipo su singole variabili	56
5	Conclusioni e sviluppi futuri	60

Capitolo 1

Introduzione

1.1 La bioinformatica

La **bioinformatica** è una disciplina che combina la **biologia**, l'**informatica** e la **statistica** per analizzare, elaborare e interpretare i dati biologici utilizzando *strumenti computazionali*. Con l'avvento delle tecnologie di sequenziamento di *DNA* e *RNA* ad alta velocità il volume di dati biologici generati è aumentato a dismisura, rendendo necessario integrare un approccio computazionale nelle metodologie di analisi biologica. La bioinformatica risulta quindi indispensabile per gestire, analizzare e trarre informazioni significative dall'enorme quantità di dati che viene generata.

Questa disciplina di recente nascita svolge un ruolo fondamentale in molteplici ambiti della biologia, trovando un'applicazione a diverse componenti della letteratura informatica sviluppata negli anni passati. Attraverso l'uso di algoritmi, modelli statistici e metodi computazionali, la **bioinformatica** consente, tra le altre cose, di analizzare sequenze genetiche, identificare geni, annotare genomi, studiare l'evoluzione e predire la struttura delle proteine.

In particolare, una componente che si dimostra cruciale in queste specifiche applicazioni è l'**allineamento di sequenze**, risultando uno strumento fondamentale per analizzare le similarità e le differenze tra le sequenze biologiche.

1.2 Applicazioni dell'allineamento di sequenze

Uno dei principi ampiamente riconosciuti nella **biologia molecolare**, che si pone come uno dei pilastri di questa disciplina, è il seguente:

Nelle **sequenze biomolecolari** (*DNA*, *RNA* o *sequenze amminoacide*), **un'alta similarità delle sequenze** di solito implica **significative similarità funzionali e/o strutturali**. [4]

Il fatto che due sequenze biomolecolari presentino un'alta similarità significa che esse condividono una notevole quantità di informazioni genetiche o strutturali. Questo

può indicare che le due sequenze derivino da organismi strettamente correlati o che svolgano funzioni simili.

In particolare, la **similarità funzionale** può indicare che le sequenze condividono una specifica attività biologica o svolgono un ruolo simile all'interno di un processo biologico, mentre la **similarità strutturale** si riferisce alla conservazione della struttura tridimensionale delle biomolecole: sequenze con alta similarità strutturale possono indicare conformazioni simili per le proteine corrispondenti. Tuttavia, è importante notare che non sempre la similarità delle sequenze garantisce una funzionalità o struttura identica: esistono infatti diversi casi in cui questo non accade.

Di conseguenza, l'allineamento di sequenze risulta fondamentale nella ricerca di **regioni conservate**, che sono segmenti di sequenze simili o addirittura identiche: esse spesso indicano l'esistenza di una funzione biologica comune o di una relazione evolutiva condivisa. Un esempio sono le sequenze proteiche, in cui regioni conservate possono rivelare la presenza di motivi strutturali o di siti attivi importanti per la loro funzione.

Un'altra applicazione importante dell'allineamento è l'**identificazione di variazioni o mutazioni**. Durante l'evoluzione, le sequenze nucleotidiche subiscono cambiamenti a livello genetico (ad esempio *mutazioni puntiformi*, *inserzioni* o *delezioni*). L'allineamento mette in evidenza le differenze tra le sequenze, permettendo di individuare queste mutazioni e di studiare il loro impatto sulla struttura e sulla funzione delle biomolecole. Un esempio di questa applicazione è il campo della ricerca sulle malattie genetiche, dove l'allineamento di sequenze genomiche può rivelare varianti genetiche associate a condizioni patologiche.

Inoltre, i risultati forniti dall'allineamento vengono utilizzati anche all'interno di altri campi della bioinformatica: nella costruzione degli **alberi filogenetici**, utilizzati per rappresentare storie e relazioni evolutive, vengono usati gli allineamenti di sequenze nucleotidiche per stimare la *distanza evolutiva* di specie diverse.

1.3 Tecnologie di sequenziamento

Uno dei principali motivi dei progressi nei campi della biologia e della genetica sono le **tecnologie di sequenziamento**, un insieme di metodi e strumenti utilizzati per determinare l'ordine esatto dei nucleotidi (A, C, G, T) all'interno di una molecola di *DNA* o *RNA*. Il loro sviluppo ha consentito la generazione di quantità sempre crescenti e di maggior qualità di dati genomici, in modo rapido ed efficiente.

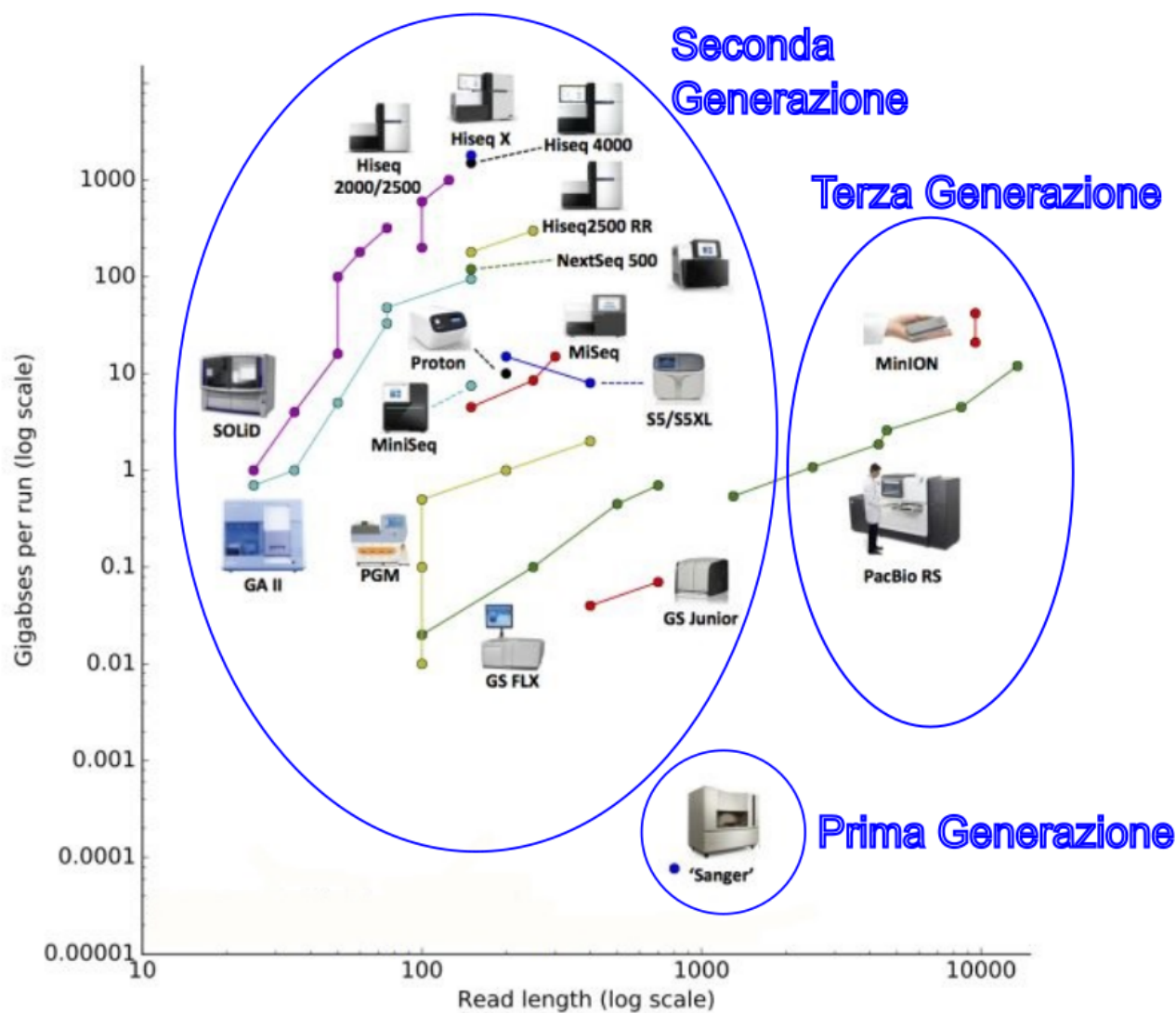


Figura 1.1: Suddivisione delle *tecnologie di sequenziamento* in generazioni.

Come si vede dalla figura 1.1, è possibile dividere le **tecnologie di sequenziamento** in **tre generazioni**:

- **Prima generazione:** tecnologie conosciute anche come *sequenziamento Sanger*, sono state usate nel progetto *Genoma Umano* per riuscire a ricostruire l'intero patrimonio genetico di un essere umano; producono sequenze di *lunghezza* pari a circa 1000 nucleotidi con un *throughput*¹ limitato, insieme a un *tasso di errore* del 6% circa;
- **Seconda generazione:** la classe di tecnologie più sviluppata, hanno come punto di forza un *elevato throughput*, mantenendo anche un *basso tasso di errore* (< 1%) e *lunghezza delle sequenze* abbastanza variabile, sempre però

¹Per *throughput* si intende la quantità di dati prodotta in un determinato lasso di tempo (nella figura 1.1 Gigabasi per esecuzione).

al di sotto delle 1000 bp² del *sequenziatore Sanger*; hanno però il limite di produrre *errori sistematici* sempre dello stesso tipo;

- **Terza generazione:** classe ancora in forte sviluppo, si concentrano sulla produzione di sequenze di *elevata lunghezza* (quasi 10000 nucleotidi per sequenza), mantenendo un *throughput* abbastanza elevato. Producono *errori* con un tasso piuttosto elevato (più del 10%), che sono però distribuiti in maniera *casuale*.

La tabella 1.1 riporta in maniera strutturata le caratteristiche delle varie generazioni di *tecnologie di sequenziamento*.

Tecnologie di sequenziamento				
Generazione	Sequenze	Throughput	Tasso di errore	Tipo di errore
Prima	≈ 1000	< 0.01	$\approx 6\%$	Casuale
Seconda	tra 50 e 1000	fino a 18000	$< 1\%$	Sistematico
Terza	≈ 10000	≈ 10	$> 10\%$	Casuale

Tabella 1.1: Caratteristiche delle varie generazioni di tecnologie di sequenziamento; il *throughput* è misurato in *Gb/day* e la *lunghezza* delle sequenze in *bp*.

1.4 Obiettivo dello stage

L'obiettivo dello stage era quello di effettuare uno studio approfondito dell'algoritmo **wavefront**, un nuovo approccio basato sulla *programmazione dinamica* che permette di calcolare la **distanza di edit** in tempo proporzionale al valore della distanza stessa (capitolo 3).

In particolare, l'obiettivo finale era quello di integrare l'approccio **wavefront** all'interno di **RecGraph** [2], un tool precedentemente sviluppato che permette di effettuare numerosi tipi di allineamenti tra un *grafo* e una *sequenza*, in particolare un allineamento che permette di individuare fino a una *ricombinazione* all'interno del genoma rappresentato nel grafo.

Nella prima parte del percorso ci si è concentrati sullo studio dell'algoritmo, partendo da una sua applicazione su due sequenze per poi cercare di generalizzarlo su strutture dati più complesse (**grafi di sequenza** e **grafi di variazione**).

In una seconda parte si è passati all'implementazione di un prototipo (nel linguaggio **Rust**) che utilizzasse l'algoritmo **wavefront** per effettuare un allineamento tra **grafi di variazione** e **sequenze** e che permettesse di migliorare le capacità computazionali di **RecGraph**, rendendo possibile l'utilizzo di istanze di grafi di dimensioni sempre maggiori.

²bp sta per *base pair*, che indica una coppia di nucleotidi; viene usata come unità di misura per la lunghezza delle sequenze generitiche.

Capitolo 2

Allineamento di sequenze

Dopo aver esposto alcuni dei principali motivi per cui l'allineamento di sequenze risulta fondamentale nella *bioinformatica*, si procede con una trattazione formale dell'argomento, partendo dal caso più semplice dell' **allineamento di due sequenze** fino ad arrivare a una sua generalizzazione tra un **grafo** e una **sequenza**.

2.1 Allineamento tra due sequenze

Definizione: Un *allineamento* di due stringhe S_1 e S_2 si ottiene inserendo inizialmente gli *spazi* (o *indel*) all'inizio, all'interno o alla fine di S_1 e S_2 , e successivamente allineando le due stringhe una sopra l'altra in maniera tale che, in entrambe le stringhe, ogni carattere o spazio risulti allineato a un singolo carattere o spazio dell'altra sequenza. Due spazi non possono risultare allineati tra loro. [4]

Un esempio di un allineamento delle due sequenze $S_1 = abbdcd$ e $S_2 = tbbqddc$ è il seguente:

a	-	b	b	-	d	c	d
t	b	b	q	d	d	c	-

Una delle possibili metodologie per approcciarsi al problema dell'allineamento è la *programmazione dinamica*: infatti esso può essere visto come un *problema di ottimizzazione*, descritto formalmente come segue:

Input:

- Alfabeto Σ
- Sequenza $X = \langle x_1, x_2, \dots, x_n \rangle$, $x_i \in \Sigma \quad \forall i \in \{1, \dots, n\}$
- Sequenza $Y = \langle y_1, y_2, \dots, y_m \rangle$, $y_j \in \Sigma \quad \forall j \in \{1, \dots, m\}$
- Matrice di score $d : (\Sigma \cup \{-\}) \times (\Sigma \cup \{-\}) \rightarrow \mathbb{Q}$

Output:

- Valore dell'allineamento $s = \sum_{i=1}^k d(x_i, y_i)$

In particolare, *l'insieme delle soluzioni ammissibili* è composto da tutti i possibili allineamenti di X e Y (con eventualmente spazi inseriti all'inizio, all'interno o alla fine delle due stringhe); la *soluzione ottimale* è quella che *massimizza* il punteggio dell'allineamento (**massima omologia**).

2.1.1 Equazioni di ricorrenza

Dimostrazione di correttezza

Si considerino i prefissi delle due stringhe X e Y

- $X_i = \langle x_1, x_2, \dots, x_i \rangle$,
- $Y_j = \langle y_1, y_2, \dots, y_j \rangle$

e sia $M[i, j]$ il *valore ottimo* dell'allineamento di X_i e Y_j ; è possibile distinguere solo i seguenti casi:

- x_i viene allineato con y_j :

X_{i-1}				X_i
x_0	x_1	\dots	x_{i-1}	x_i
y_0	y_1	\dots	y_{j-1}	y_j
Y_{j-1}				Y_j

in questo caso, il valore ottimo dell'allineamento sarà dato da

$$M[i, j] = M[i - 1, j - 1] + d(x_i, y_j)$$

- x_i viene allineato con '-':

X_{i-1}				X_i
x_0	x_1	\dots	x_{i-1}	x_i
y_0	y_1	\dots	y_{j-1}	-
Y_{j-1}				

in questo caso, il valore ottimo dell'allineamento sarà dato da

$$M[i, j] = M[i - 1, j] + d(x_i, -)$$

- y_j viene allineato con '-':

X_{i-1}				
x_0	x_1	\dots	x_{i-1}	-
y_0	y_1	\dots	y_{j-1}	y_j
Y_{j-1}				Y_j

in questo caso, il valore ottimo dell'allineamento sarà dato da

$$M[i, j] = M[i, j - 1] + d(-, y_j)$$

Non si riescono a individuare altre possibilità, quindi i tre casi elencati coprono tutte le casistiche; è possibile scrivere di conseguenza la seguente *equazione di ricorrenza*:

Needleman-Wunsch [8]

$$M[i, j] = \max \begin{cases} M[i - 1, j] + d(x_i, -) \\ M[i - 1, j - 1] + d(x_i, y_j) \\ M[i, j - 1] + d(-, y_j) \end{cases} \quad (2.1)$$

$$\forall i > 0, \forall j > 0$$

2.1.2 Ricostruzione della soluzione

Per ottenere anche l'allineamento effettivo delle due sequenze, è necessario salvare per ogni elemento di $M[i, j]$ la cella da cui è stato computato, in maniera da poter poi ricostruire la soluzione procedendo a ritroso. Formalmente, questo si traduce nel computare un'altra *matrice di programmazione dinamica* di dimensione $(n + 1) \times (m + 1)$ come segue:

$$T[i, j] = \begin{cases} \uparrow, & \text{se } M[i, j] = M[i - 1, j] + d(x_i, -), \\ \nwarrow, & \text{se } M[i, j] = M[i - 1, j - 1] + d(x_i, y_j), \\ \leftarrow, & \text{se } M[i, j] = M[i, j - 1] + d(-, y_j), \end{cases} \quad \forall (i, j) > \underline{0} \quad (2.2)$$

$$T[i, 0] = \uparrow \quad \forall i > 0$$

$$T[0, j] = \leftarrow \quad \forall j > 0$$

Una volta trovata la soluzione ottimale, è sufficiente seguire la sequenza delle celle salvate fino a $M[0, 0]$ per ricostruire l'allineamento.

2.1.3 Complessità computazionale

Complessità in tempo

Per calcolare l'allineamento ottimale, è necessario computare una matrice di dimensione $(n + 1) \times (m + 1)$, dove ogni elemento viene calcolato in $\Theta(1)$

(equazioni 2.1 e 2.2); di conseguenza, la *complessità temporale* dell'algoritmo sarà data da:

$$T(n, m) = \Theta(nm)$$

Per ricostruire la soluzione, invece, è necessario ripercorrere a ritroso il percorso ottimale all'interno della matrice; nel caso migliore questo significa muoversi sempre in diagonale fino alla cella iniziale, mentre nel caso peggiore si seguirà un percorso composto unicamente da passi orizzontali e verticali. In ogni caso, la complessità risulta minore della computazione della matrice: nel **caso migliore** è *limitata inferiormente* da

$$T(n, m) = \Omega(\max\{n, m\})$$

mentre nel **caso peggiore** risulta *limitata superiormente* da

$$T(n, m) = O(n + m)$$

Complessità in spazio

Ogni elemento della matrice occupa una quantità di memoria *costante*, quindi, in maniera analoga, la *complessità spaziale*¹ sarà:

$$M(n, m) = \Theta(nm)$$

2.1.4 Caso globale

Condizioni al contorno

Nel **caso globale**, l'obiettivo è quello di allineare completamente le due stringhe: come conseguenza, l'allineamento deve iniziare dal primo simbolo e finire all'ultimo, per entrambe le sequenze. Formalmente, questo si traduce nelle seguenti *condizioni al contorno*:

$$M[i, j] = \begin{cases} 0 & i = 0, j = 0 \\ M[i - 1, 0] + d(x_i, -) & i > 0, j = 0 \\ M[0, j - 1] + d(-, y_j) & i = 0, j > 0 \end{cases} \quad (2.3)$$

La **soluzione ottima** è data dall'allineamento di entrambe le sequenze nella loro interezza:

$$s = M[n, m] \quad (2.4)$$

¹Se si è interessati solo al *valore dell'allineamento* e non alla *ricostruzione della soluzione*, non è necessario salvare tutta la matrice, ma basta mantenere una sola riga (o colonna, a seconda di come la si computa) per volta. La **complessità spaziale** risulta quindi $\Theta(\min\{n, m\})$.

Esempio

Di seguito un esempio dell'**allineamento globale** ottimale di due sequenze nucleotidiche:

Input

- $\Sigma = \{A, G, C, T\}$
- $X = \text{"GAATTCAGTTA"}$
- $Y = \text{"GGATCGA"}$
- $d(x, y) = \begin{cases} 5 & \forall x, y \in \Sigma \mid x = y, \\ -3 & \forall x, y \in \Sigma \mid x \neq y, \\ -4 & \forall x \in \Sigma, y = -, \\ -4 & x = -, \forall y \in \Sigma \end{cases}$

Output

Score = 11

G	A	A	T	T	C	A	G	T	T	A
G	G	A	-	T	C	-	G	-	-	A
+5	-3	+5	-4	+5	+5	-4	+5	-4	-4	+5

2.1.5 Caso semiglobale**Condizioni al contorno**

Nel **caso semiglobale**, la seconda sequenza (chiamata solitamente *query* o *read*) può essere allineata a una qualsiasi *sottostringa* della prima (chiamata *testo*): in altre parole, è possibile inserire un numero qualsiasi di *spazi* all'inizio e alla fine della *read* senza penalità aggiuntive. Definita quindi X come la **sequenza di riferimento** e Y come la **query** da allineare, questo si traduce nelle seguenti *condizioni al contorno*:

$$M[i, j] = \begin{cases} 0 & i \geq 0, j = 0 \\ M[0, j - 1] + d(-, y_j) & i = 0, j > 0 \end{cases} \quad (2.5)$$

La **soluzione ottima** sarà data dall'allineamento che comprende tutta la *read* e una qualsiasi sottostringa del *testo* e che *massimizza il punteggio*, ovvero:

$$s = \max_{0 \leq i \leq n} M[i, m] \quad (2.6)$$

Esempio

Di seguito un esempio dell'**allineamento semiglobale** ottimale di due sequenze nucleotidiche:

Input

- $\Sigma = \{A, G, C, T\}$
- $X = \text{"GAATTCAGTTA"}$
- $Y = \text{"AAACGGT"}$
- $d(x, y) = \begin{cases} 5 & \forall x, y \in \Sigma \mid x = y, \\ -3 & \forall x, y \in \Sigma \mid x \neq y, \\ -4 & \forall x \in \Sigma, y = -, \\ -4 & x = -, \forall y \in \Sigma \end{cases}$

Output

Score = 23

G	A	A	T	T	C	G	G	T	T	A
-	A	A	-	A	C	G	G	T	-	-
0	+5	+5	-4	-3	+5	+5	+5	+5	0	0

2.2 Distanza di edit tra due sequenze

Un'altra possibilità per misurare quanto due sequenze sono simili tra loro è calcolare la **distanza** che le divide; in particolare, ci sono numerose maniere con cui viene formalizzata la *distanza tra due stringhe*: una delle più utilizzate è la **distanza di edit**, la quale fornisce il minimo numero di operazioni necessarie a trasformare una stringa nell'altra (oltre all'effettiva sequenza di operazioni necessarie).

Definizione: La *distanza di edit* tra due sequenze è definita come il *minimo* numero di *operazioni di edit* - **inserimenti, cancellazioni e sostituzioni** - necessarie a trasformare la prima sequenza nella seconda. [4]

Si noti che la **distanza di edit** riferita alla trasformazione della prima sequenza nella seconda è sempre uguale a quella che si riferisce alla trasformazione opposta: è inoltre possibile passare da un **edit transcript** al duale semplicemente invertendo le operazioni effettuate.

In maniera analoga all'**allineamento di sequenze**, anche il problema della **distanza di edit** può essere risolto in maniera efficiente tramite *programmazione dinamica*:

Input:

- Alfabeto Σ
- Sequenza $X = \langle x_1, x_2, \dots, x_n \rangle$, $x_i \in \Sigma \quad \forall i \in \{1, \dots, n\}$
- Sequenza $Y = \langle y_1, y_2, \dots, y_m \rangle$, $y_j \in \Sigma \quad \forall j \in \{1, \dots, m\}$

Output:

- Distanza di edit d
- Sequenza delle operazioni di edit (edit transcript)

L'insieme delle soluzioni ammissibili è composto da tutte gli **edit transcript** che trasformano X in Y ; la *soluzione ottimale* è quella che minimizza il numero delle operazioni di edit.

2.2.1 Equazioni di ricorrenza

Dimostrazione di correttezza

Si considerino i prefissi delle due stringhe $X_i = \langle x_1, x_2, \dots, x_i \rangle$, $Y_j = \langle y_1, y_2, \dots, y_j \rangle$ e siano noti

- $D[i, j]$, il *valore ottimo* della distanza di edit tra X_i e Y_j ;
- $S[i, j]$, l'insieme delle *operazioni di edit* per trasformare X_i in Y_j ;

è possibile distinguere 4 casistiche²:

- $x_i = y_j$: in questo caso nessuna operazione è necessaria e la *distanza di edit* resta invariata:

$$D[i + 1, j + 1] = D[i, j]$$

$$S[i + 1, j + 1] = S[i, j]$$

- $x_i \neq y_j$: in questo caso ci sono 3 possibilità:

²Le *operazioni di edit* sono rappresentate come

- $I(a)$: a viene inserito nella sequenza;
- $M(a, b)$: a viene sostituito da b nella sequenza;
- $D(b)$: b viene cancellato dalla sequenza.

- **Inserimento**, ovvero y_j viene aggiunto alla sequenza X_i : in questo caso la *distanza di edit* aumenta di uno:

$$D[i, j + 1] = D[i, j] + 1$$

$$S[i, j + 1] = S[i, j] \cup I(y_j)$$

- **Sostituzione**, ovvero x_i viene sostituito con y_j : in questo caso la *distanza di edit* aumenta di uno:

$$D[i + 1, j + 1] = D[i, j] + 1$$

$$S[i + 1, j + 1] = S[i, j] \cup M(x_i, y_j)$$

- **Cancellazione**, ovvero x_i viene cancellato dalla sequenza X_i : in questo caso la *distanza di edit* aumenta di uno:

$$D[i + 1, j] = D[i, j] + 1$$

$$S[i + 1, j] = S[i, j] \cup D(x_i)$$

Non esistono altre possibilità, quindi i quattro casi elencati coprono tutte le casistiche; è possibile scrivere di conseguenza la seguente *equazione di ricorrenza*

$$D[i, j] = \begin{cases} D[i - 1, j - 1] & \text{se } x_i = y_j \\ \min \begin{cases} D[i, j - 1] \\ D[i - 1, j - 1] \\ D[i - 1, j] \end{cases} + 1 & \text{se } x_i \neq y_j \end{cases} \quad (2.7)$$

$$\forall i > 0, \forall j > 0$$

Insieme alle seguenti *condizioni al contorno*

$$D[i, j] = \begin{cases} 0 & i = 0, j = 0 \\ D[0, j - 1] + 1 & i = 0, j > 0 \\ D[i - 1, 0] + 1 & i > 0, j = 0 \end{cases} \quad (2.8)$$

La *distanza di edit ottimale* è data da $D[n, m]$.

2.2.2 Ricostruzione della soluzione

Per la sequenza delle *operazioni di edit* si procede come per l'allineamento di sequenze (sezione 2.1.2), salvando per ogni elemento di $D[i, j]$ la cella da cui è stato computato, in maniera da poter poi ricostruire la soluzione procedendo all'indietro.

2.2.3 Complessità computazionale

Complessità in tempo

Per calcolare l'allineamento ottimale, è necessario computare una matrice di dimensione $(n + 1) \times (m + 1)$, dove ogni elemento viene calcolato in $\Theta(1)$ (equazione 2.7); di conseguenza, la *complessità in tempo* dell'algoritmo sarà data da:

$$T(n, m) = \Theta(nm)$$

Come per l'**allineamento di sequenze** (sezione 2.1.3), la ricostruzione della soluzione è per la **distanza di edit** è *limitata superiormente* da

$$T(n, m) = O(n + m)$$

Complessità in spazio

Ogni elemento della matrice occupa una quantità di memoria *costante*, quindi, in maniera analoga, la *complessità spaziale*³ sarà:

$$M(n, m) = \Theta(nm)$$

2.2.4 Caso pesato

E' possibile estendere il problema della *distanza di edit* a considerare penalità per le diverse operazioni non unitarie (e non necessariamente uguali), mantenendo comunque la *stessa complessità*. Si introducono

- i = penalità per un *inserimento*,
- m = penalità per una *sostituzione*,
- d = penalità per una *cancellazione*,

con $(i, m, d) \in \mathbb{Q}_+^3$; è possibile riscrivere le equazioni 2.7 e 2.8 come:

$$D[i, j] = \begin{cases} D[i - 1, j - 1] & \text{se } x_i = y_j \\ \min \begin{cases} D[i, j - 1] + i \\ D[i - 1, j - 1] + m \\ D[i - 1, j] + d \end{cases} & \text{se } x_i \neq y_j \end{cases} \quad (2.9)$$

$$\forall i > 0, \forall j > 0$$

³Se si è interessati solo alla *distanza di edit* e non alla *sequenza di operazioni di edit*, non è necessario salvare tutta la matrice, ma basta mantenere una sola riga (o colonna, a seconda di come la si computa) per volta. La **complessità spaziale** risulta quindi $\Theta(\min\{n, m\})$.

$$D[i, j] = \begin{cases} 0 & i = 0, j = 0 \\ D[0, j - 1] + i & i = 0, j > 0 \\ D[i - 1, 0] + d & i > 0, j = 0 \end{cases} \quad (2.10)$$

Esempio

Di seguito un esempio del calcolo della **distanza di edit pesata** di due sequenze nucleotidiche:

Input

- $\Sigma = \{A, G, C, T\}$
- $X = \text{"GAATTCAGTTA"}$
- $Y = \text{"GGATCGA"}$
- $i = 4$
- $m = 3$
- $d = 4$

Output

Weighted edit distance = 19

G	A	A	T	T	C	A	G	T	T	A
			d			d		d	d	
G	A	A		T	C		G			A
	m									
G	G	A		T	C		G			A
0	3	0	4	0	0	4	0	4	4	0

2.3 Estensione all'allineamento tra un grafo e una sequenza

I **grafi di pangenomi** sono strutture che rappresentano una collezione di *genomi* e codificano variazioni di sequenze al loro interno, permettendo lo studio di più genomi simili tra loro. Per riuscire a utilizzare queste strutture risulta necessario generalizzare il problema dell'allineamento: di seguito vengono riportate due estensioni a riguardo, rispettivamente su **grafi di sequenza** e **grafi di variazione**.

2.3.1 Grafi di sequenza

Un **grafo di sequenza** (o **sequence graph**) è una struttura dati utilizzata per rappresentare e analizzare le relazioni tra sequenze di DNA, RNA o proteine; i nodi rappresentano le diverse sequenze e gli archi rappresentano le relazioni tra di esse. Gli archi possono indicare la presenza di varianti, come mutazioni, inserzioni o delezioni, che possono differire tra le diverse sequenze rappresentate nel grafo. Formalmente, un **sequence graph** si definisce come segue:

Definizione: un *sequence graph* è una tripla $G = (V, E, \delta)$, dove

- $V = \{v_1, v_2, \dots, v_n\}$ è l'insieme dei **vertici** del grafo, che rappresentano le sequenze;
- $E \subseteq V \times V$ è l'insieme degli **archi** del grafo, che rappresentano le relazioni tra diverse sequenze;
- $\delta : V \rightarrow \Sigma^*$ è la **funzione di etichettatura**, che associa a ogni vertice la sequenza corrispondente.

In particolare, se a ogni **vertice** viene associato un solo carattere (e quindi la **funzione di etichettatura** è definita come $\delta : V \rightarrow \Sigma$), il *grafo* viene detto *canonico*.

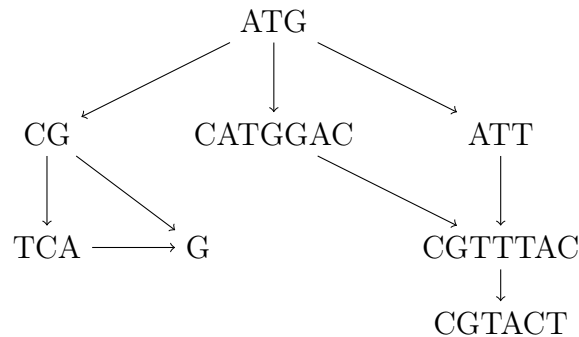


Figura 2.1: Esempio di **grafo di sequenza**.

2.3.2 Partial Order Alignment (POA)

Una delle possibilità per rappresentare un *allineamento multiplo di sequenze* è tramite una struttura chiamata **POA (Partial Order Alignment)** [5], che utilizza un **grafo di sequenza ordinato topologicamente** per rappresentare l'allineamento. Nella figura 2.2 è riportato un esempio:

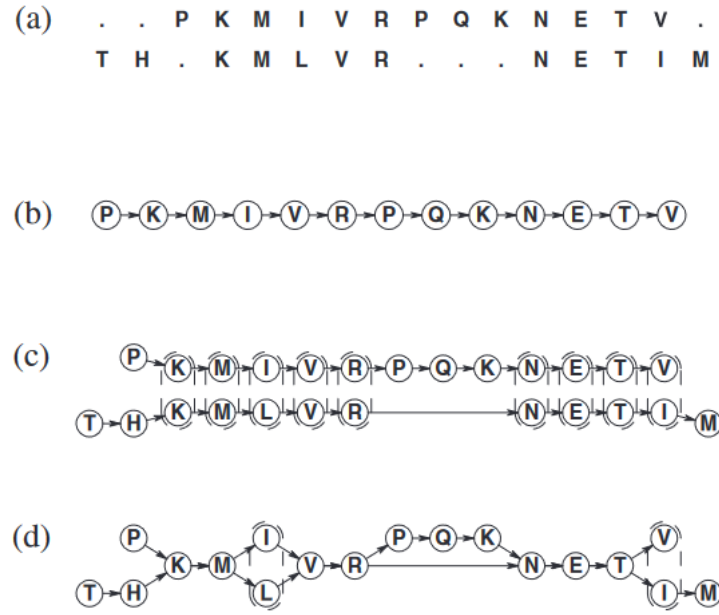


Figura 2.2: **MSA** rappresentato con un **POA**. (a) Rappresentazione in formato **RC-MSA** di un allineamento di sequenze proteiche. (b) Una singola sequenza in formato **PO-MSA**. (c) Due sequenze proteiche allineate tra loro in formato **PO-MSA**. I cerchi tratteggiati indicano che due nodi sono allineati. (d) Rappresentazione in formato **PO-MSA** di un allineamento di due sequenze proteiche. I cerchi tratteggiati indicano che due nodi sono allineati. [5]

Allineamento tra un POA e una sequenza

L'algoritmo di **programmazione dinamica Needleman-Wunsch** per calcolare l'allineamento tra sequenze (sezione 2.1.1) può essere esteso in maniera da funzionare anche su **ordini parziali**. In maniera informale, è possibile considerare una sequenza come un grafo degenere, in cui ogni vertice (tranne il primo e l'ultimo) hanno esattamente un predecessore e un successore: per estendere **Needleman-Wunsch** all'utilizzo di **POA**, è sufficiente che per ogni vertice non si consideri solo un nodo precedente, bensì tutti i predecessori di tale vertice. Formalmente, considerati in ingresso

- un **alfabeto** Σ ,
- un **grafo canonico** $G = (V, E, \delta)$ con
 - $V = \{v_1, v_2, \dots, v_n\}$ **ordinato topologicamente**, a cui si aggiunge un vertice v_0 per rappresentare il **grafo vuoto**,
 - $E : V \times V$, a cui si aggiunge (v_0, v_1) ,
 - $\delta : V \rightarrow \Sigma$, a cui si aggiunge $\delta(v_0) = \epsilon$,
- una **sequenza (o read)** $X = \langle x_1, x_2, \dots, x_m \rangle$, $x_j \in \Sigma \forall j \in \{1, 2, \dots, m\}$,
- una **matrice di score** $d : (\Sigma \cup \{-\}) \times (\Sigma \cup \{-\}) \rightarrow \mathbb{Q}$

questo si traduce nella seguente *equazione di ricorrenza*:

$$M[v, j] = \max \begin{cases} M[u, j] + d(\delta(v), -) \\ M[u, j-1] + d(\delta(v), x_j) \\ M[v, j-1] + d(-, x_j) \end{cases} \quad (2.11)$$

$$\forall u \text{ s.t. } (u, v) \in E, \forall j > 0$$

e nelle seguenti *condizioni al contorno*:

Caso globale

Caso base:

$$M[v, j] = \begin{cases} 0 & v = v_0, j = 0 \\ M[u, 0] + d(\delta(u), -) & \forall (u, v) \in E, j = 0 \\ M[v_0, j-1] + d(-, x_j) & v = v_0, j > 0 \end{cases} \quad (2.12)$$

Soluzione ottima:

$$M[v_n, m]$$

Caso semiglobale

Caso base:

$$M[v, j] = \begin{cases} 0 & \forall v \in V, j = 0 \\ M[v_0, j-1] + d(-, x_j) & v = v_0, j > 0 \end{cases} \quad (2.13)$$

Soluzione ottima:

$$\max_{v \in V} M[v, m]$$

Esempio

Come esempio si propone una visualizzazione grafica della *matrice di programmazione dinamica* costruita tramite l'algoritmo appena descritto:

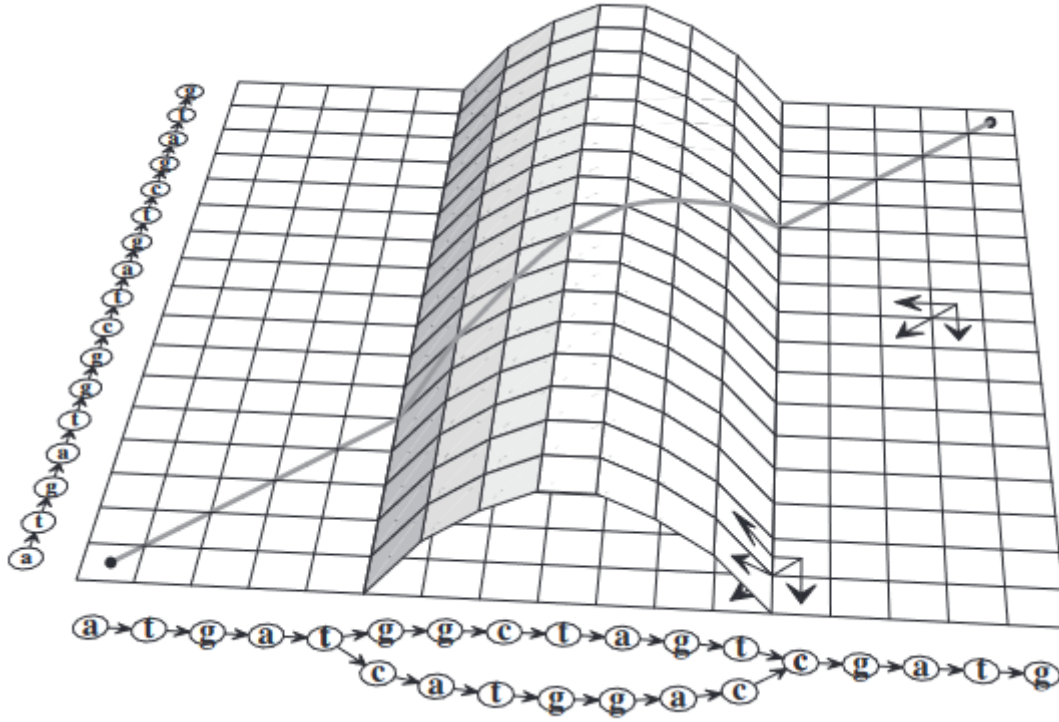


Figura 2.3: *Matrice di programmazione dinamica* di un allineamento tra un **POA** e una **sequenza**. [5]

Come si vede dalla figura 2.3, da ogni **vertice** con *successori multipli* viene computata una nuova *matrice di programmazione dinamica*; in maniera analoga, in ogni **nodo** con *più di un predecessore*, due diverse *matrici* si fondono in una sola.

Complessità computazionale

Come nell' **allineamento di sequenze** (sezione 2.1.3), è necessario computare una matrice di dimensione $(n + 1) \times (m + 1)$; la differenza risiede nel fatto che ogni elemento della matrice non è più computabile da un *numero finito* di casi, bensì dipende dal numero di *predecessori* di ogni vertice (equazione 2.11): di conseguenza, non è più possibile considerare *costante* il tempo con cui calcola ogni cella. Quindi, definito

- \bar{n}_p = *numero medio* di *predecessori* per vertice

La **complessità in tempo** dell'algoritmo risulta essere

$$T(n, m) = \Theta((2\bar{n}_p + 1) \cdot n \cdot m) = \Theta(\bar{n}_p \cdot n \cdot m)$$

lineare nel numero di *predecessori* di ogni vertice.

In particolare si noti che, fissando \bar{n}_p a 1, ci si riconduce all'allineamento di due sequenze, con la stessa **complessità temporale**

$$T(n, m) = \Theta(3nm) = \Theta(nm)$$

Per quanto riguarda la **complessità in spazio**, ogni elemento della matrice continua a mantenere una quantità *costante* di informazione: di conseguenza, essa rimane

$$M(n, m) = \Theta(nm)$$

2.3.3 Grafi di variazione

Un **grafo di variazione** (o **variation graph**) è un'ulteriore generalizzazione dei **grafi di sequenza**, che permette di rappresentare ancora con maggior efficacia genomi simili contenenti variazioni al loro interno: infatti, nei **grafi di variazione**, tutte le informazioni sugli aplotipi sono rappresentate come *percorsi distinti*, mentre nei **grafi di sequenza** non esiste distinzione sui diversi cammini. Questo è possibile in seguito all'aggiunta di una nuova componente del grafo, ovvero l'insieme di tutti i cammini e dei nodi che li compongono: in questa maniera, ogni variazione è rappresentata da un percorso diverso del grafo, che preso individualmente risulta "lineare" (ogni vertice al suo interno possiede soltanto un predecessore e un successore). Per fare in modo che tutti i cammini inizino e finiscano nello stesso nodo, vengono aggiunti un vertice iniziale e un vertice finali fittizi, appartenenti a tutti i cammini. Da punto di vista matematico, questo si traduce come segue:

Definizione: un *grafo di variazione* è una quadrupla $G = (V, E, \delta, P)$, dove:

- $V = \{v_0, v_1, v_2, \dots, v_n\}$ è l'insieme dei **vertici** del grafo;
- $E \subseteq V \times V$ è l'insieme degli **archi** del grafo;
- $\delta : V \rightarrow \Sigma^*$ è la **funzione di etichettatura**, che associa a ogni vertice la sequenza corrispondente,
- $P = \{p_1, p_2, \dots, p_k\}$, $p_i = \langle v_0 = v_{i_1}, v_{i_2}, \dots, v_{i_l} = v_n \rangle \forall i \in \{1, 2, \dots, k\}$ è l'**insieme dei percorsi del grafo**.

Anche per i **grafi di variazione**, se a ogni vertice viene associato un solo carattere (**funzione di etichettatura** definita come $\delta : V \rightarrow \Sigma$), il *grafo* viene detto *canonico*.

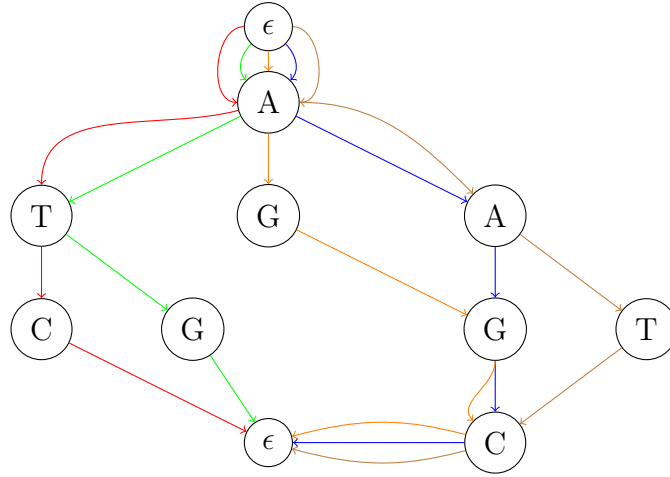


Figura 2.4: Esempio di **grafo di variazione canonico**: ogni *percorso* è rappresentato da un colore diverso.

2.3.4 Allineamento tra un grafo di variazione e una sequenza

E' possibile estendere ulteriormente le equazione di **Needleman-Wunsch** generalizzate a **grafi di sequenza** (2.11) per adattarle all'utilizzo di **grafi di variazione**. In particolare, è necessario aggiungere un'ulteriore dimensione alla *matrice di programmazione dinamica*, in modo da poter riuscire a memorizzare l'allineamento ottimo per ciascun cammino. Considerando un **grafo di variazione** $G = (V, E, \delta, P)$, questo viene espresso tramite le seguenti *equazioni*:

$$M[v, j, p] = \max \begin{cases} M[u, j, p] + d(\delta(v), -) \\ M[u, j - 1, p] + d(\delta(v), x_j) \\ M[v, j - 1, p] + d(-, x_j) \end{cases} \quad (2.14)$$

$$\forall u \text{ s.t. } (u, v) \in E, \forall j > 0, \forall p \in P$$

e dalle seguenti *condizioni al contorno*:

Caso globale

Caso base:

$$M[v, j, p] = \begin{cases} 0 & v = v_0, j = 0 \\ M[u, 0, p] + d(\delta(u), -) & \forall (u, v) \in E, j = 0 \\ M[v_0, j - 1, p] + d(-, x_j) & v = v_0, j > 0 \end{cases} \quad (2.15)$$

$$\forall p \in P$$

Soluzione ottima:

$$\max_{p \in P} M[v_n, m, p]$$

Caso semiglobale

Caso base:

$$M[v, j, p] = \begin{cases} 0 & \forall v \in V, j = 0 \\ M[v_0, j - 1, p] + d(-, x_j) & v = v_0, j > 0 \end{cases} \quad (2.16)$$

Soluzione ottima:

$$\max_{v \in V, p \in P} M[v, m, p]$$

Esempio

A seguire un esempio grafico dell'*allineamento* tra un **grafo di variazione** e una **sequenza**:

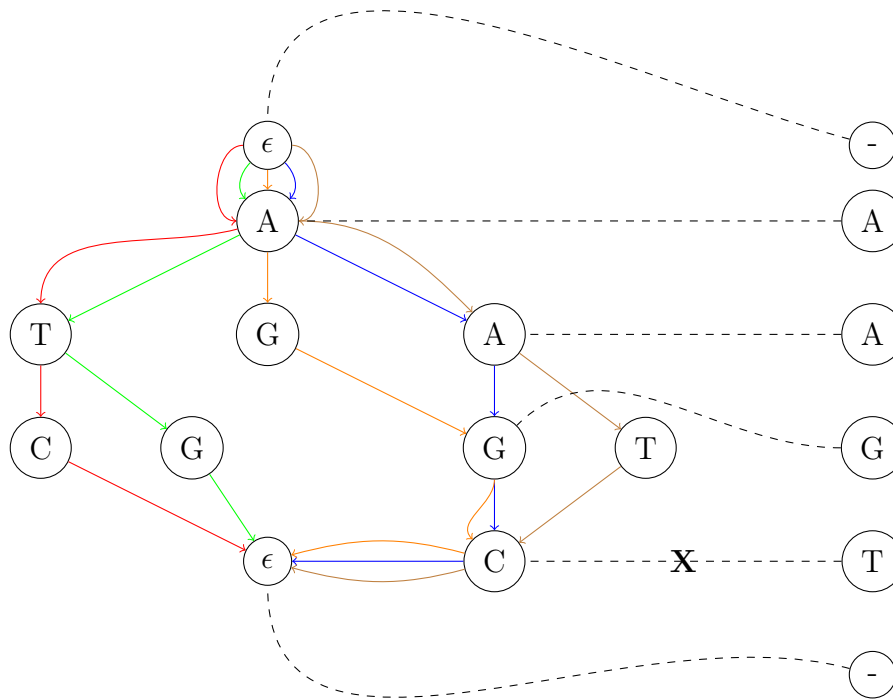


Figura 2.5: Esempio di allineamento tra un **grafo di variazione canonico** e una **sequenza**: il *percorso ottimale* risulta essere quello blu.

Complessità computazionale

A differenza dell'allineamento con un **grafo di sequenza** (sezione 2.3.2), ogni vertice possiede un unico *predecessore* e quindi, sebbene non immediatamente evidente dall'equazione 2.14, ogni cella della matrice viene computata in $O(1)$ (come nell'allineamento tra sequenze). E' tuttavia necessario aggiungere la dimensione relativa ai *percorsi* del grafo alla stessa matrice, che dunque assume

una dimensione totale $(n + 1) \times (m + 1) \times k$, dove $k = |P|$. Di conseguenza, la **complessità in tempo** risulta essere:

$$T(n, m, k) = O(n \cdot m \cdot k)$$

Per quanto riguarda la **complessità in spazio**, ogni elemento della matrice continua a mantenere una quantità *costante* di informazione: di conseguenza, essa è uguale a

$$M(n, m, k) = O(n \cdot m \cdot k)$$

2.4 Allineamento con ricombinazione

Uno dei processi biologici che viene osservato con molta frequenza durante le fasi di duplicazioni del DNA sono le **ricombinazioni**, le quali consistono nell'importazione totale o parziale di geni al posto di materiale genetico presente in regioni omologhe a quelle importate del genoma di interesse. Uno degli organismi in cui risulta più interessante lo studio delle ricombinazioni sono i *batteri*, nei quali questi processi avvengono con elevate frequenze: esse sono infatti state notate per la prima volta durante lo studio dei *emphloci* batterici che codificano per *antigeni* o per *resistenze agli antibiotici*.

Risulta interessante un approccio computazionale che estenda l'allineamento tra un **grafo di variazione** e una **sequenza** per riuscire a individuare una, o eventualmente più **ricombinazioni** avvenute all'interno del genoma rappresentato nel grafo: una trattazione approfondita dell'argomento è riportata in [2], mentre di seguito vengono riportate le idee principali dietro a questo tipo di allineamento.

2.4.1 Definizioni

Si considera un **grafo di variazione canonico** $G = \langle V, E, \delta, P \rangle$ (sezione 2.3.3); si definiscono:

- una **ricombinazione di due percorsi** come una quadrupla $R = \langle p_1, p_2, \rho, \psi \rangle$, dove
 - $p_1, p_2 \in P$ sono due *cammini* del grafo;
 - $\rho \in p_1$ e $\psi \in p_2$ sono due *vertici* dei cammini selezionati.
- i due vertici⁴ $\alpha_{p_1, p_2}(\rho, \psi) \in V$ e $\beta_{p_1, p_2}(\rho, \psi) \in V$ come due nodi per i quali:
 - $\alpha_{p_1, p_2}(\rho, \psi)$ precede ρ in p_1 e ψ in p_2 , e segue qualsiasi altro vertice u che precede entrambi ρ in p_1 e ψ in p_2 ;

⁴Quando chiari dal contesto, i cammini p_1, p_2 e i vertici ρ, ψ vengono omessi e i vertici indicati come α e β .

- $\beta_{p_1, p_2}(\rho, \psi)$ segue ρ in p_1 e ψ in p_2 , e precede qualsiasi altro vertice v che segue entrambi ρ in p_1 e ψ in p_2 ;

ricordando il concetto *Lower Common Ancestor (LCA)* di un albero;

- il **displacement di una ricombinazione** come la quantità

$$D = ||a_1| - |a_2| + 1| + ||b_1| - |b_2| - 1| \quad (2.17)$$

dove:

- a_1, a_2 sono i *sottopercorsi* che vanno rispettivamente da α a ρ e da α a ψ ;
- b_1, b_2 sono i *sottopercorsi* che vanno rispettivamente da ρ a β e da ψ a β ;

che vuole misurare il *peso* di una **ricombinazione**;

- un **allineamento con ricombinazione** come la concatenazione di due allineamenti senza ricombinazione:

- il primo che riguarda il *sottopercorso* a_1 e il *prefisso* $Y[: j]$,
- il secondo che riguarda il *sottopercorso* b_2 e il *suffisso* $Y[j + 1 :]$,

dove Y_m è la sequenza da allineare e $j \in \{1, \dots, m\}$ la posizione della sequenza in cui avviene la **ricombinazione**.

Un esempio delle definizioni introdotte è mostrato nella figura 2.6.

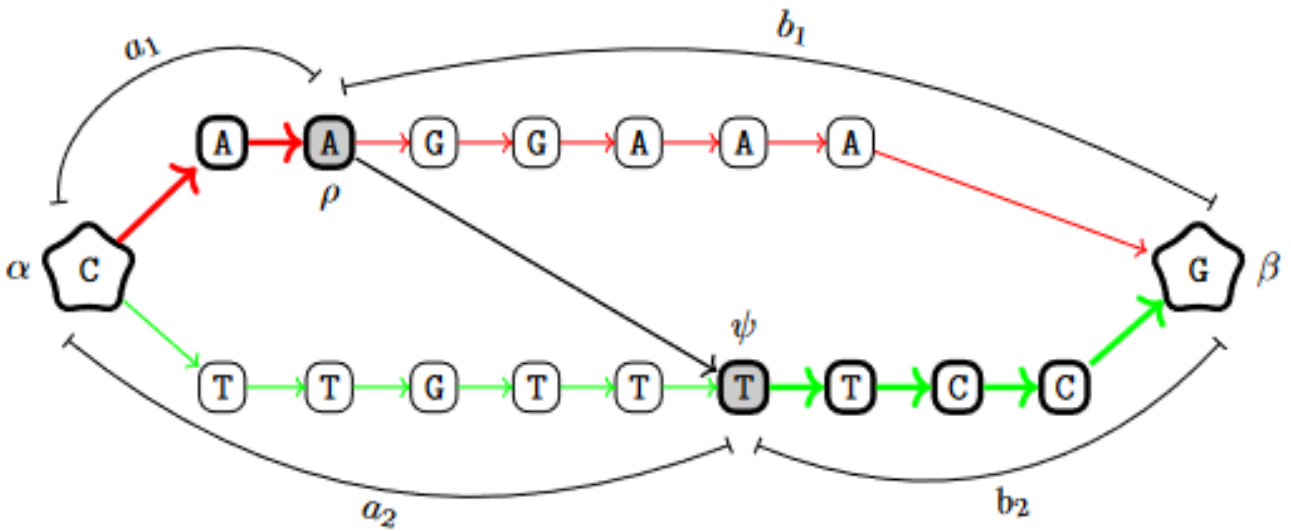


Figura 2.6: Esempio di **displacement** di una **ricombinazione**. [2]

2.4.2 Equazioni di ricorrenza

Come conseguenza della definizione, un **allineamento con ricombinazione** può essere computato attraverso due diversi **allineamenti senza ricombinazione**,

uno che comprenda la porzione iniziale di un cammino e il prefisso della read e l'altro che si riferisca alla porzione finale di un altro cammino e al suffisso della stessa read.

Formalmente, definita $M[v, j, p]$ come in 2.14 e $R[v, j, p]$ come il valore ottimale dell'allineamento globale tra la porzione finale di $p \in P$ che inizia nel vertice ψ e tra il *suffisso* $Y[: j]$, il valore ottimale dell'**allineamento con al più una ricombinazione** può essere descritto dalla ricorrenza 2.18:

$$\max \begin{cases} \max_{p \in P} M[v_0, m, p] & (a) \\ \max_{(v,w), j, p \neq q} M[v, j, p] + R[w, j+1, q] + d_o + d_e \cdot d(v, w) & (b) \end{cases} \quad (2.18)$$

dove:

- $d(v, w)$ è il **displacement** tra v e w (equazione 2.17);
- d_o è il **costo di apertura del displacement**;
- d_e è il **costo di estensione del displacement**;

Se il massimo è verificato dal caso (a), allora l' *allineamento ottimale* è **senza ricombinazione**; se invece è verificato dal caso (b), si trova un **allineamento con una ricombinazione**.

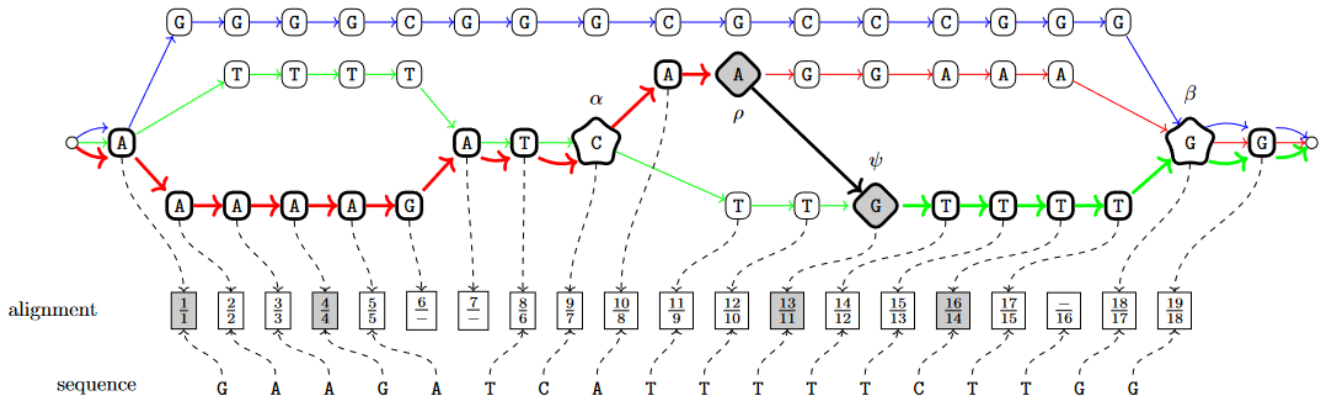


Figura 2.7: Esempio di un **allineamento ottimale con ricombinazione**. [2]

2.4.3 Complessità computazionale

Complessità in tempo

La **complessità temporale** per computare le due matrici $M[v, j, p]$ e $R[v, j, p]$ rimane la stessa di 2.3.4 (le due matrici possono anche essere calcolate in parallelo); di conseguenza, definiti

- $n = |V|$
- $m = |Y|$

- $p = |P|$

la **complessità in tempo** è data dal caso (b) dell'equazione 2.18 e vale:

$$T(n, m, p) = O(n^2 \cdot m \cdot p^2) \quad (2.19)$$

Complessità in spazio

Le due matrici $M[v, j, p]$ e $R[v, j, p]$ occupano la stessa quantità di memoria; di conseguenza la **complessità in spazio** rimane:

$$M(n, m, p) = O(n \cdot m \cdot p) \quad (2.20)$$

2.4.4 RecGraph

RecGraph è un tool sviluppato dal laboratorio **BIAS**, nel linguaggio di programmazione **Rust**, che permette di eseguire allineamenti tra un **grafo** e una **sequenza** in diverse modalità, tra cui la **recombination mode**, in cui si cerca di individuare fino a una **ricombinazione** avvenuta nel genoma corrispondente a uno dei percorsi del grafo. Un'interessante ottimizzazione per l'allineamento su **grafi di variazione** implementata all'interno del tool è il **reference path**, che permette di evitare di calcolare più di una volta il valore dell'allineamento di vertici appartenenti a più di un *cammino* (spiegazione formale contenuta in [2]).

L'obiettivo finale dello stage era quello di cercare di migliorare le capacità computazionali di **RecGraph**, concentrandosi sull'allineamento tra un **grafo di variazione** e una **sequenza**, in maniera che potesse poi essere utilizzato nella **recombination mode** su grafi di dimensioni maggiori (e, in futuro, possa essere usato per permettere più di una ricombinazione): per far questo, si è passati allo studio dell' **algoritmo wavefront**, di cui segue una trattazione formale nel capitolo successivo.

Capitolo 3

Algoritmo wavefront per la distanza di edit

La **complessità in tempo** (e **in memoria**) $\Theta(nm)$ rende poco pratico l'utilizzo dell'algoritmo di **Needleman-Wunsch** [8] su sequenze di grandi dimensioni; ciò risulta ancora più evidente nell'estensione dell'algoritmo sui **grafi di sequenza** (sezione 2.3.2) e sui **grafi di variazione** (sezione 2.3.4), limitando fortemente le dimensioni dei grafi che è possibile trattare in tempi realistici. Recentemente è stato ripreso e sviluppato [1] un algoritmo per calcolare l'allineamento di sequenze in modo esatto, chiamato *wavefront algorithm* (WFA) [9], che permette di computare la **distanza di edit** di *due sequenze* con una complessità in tempo lineare rispetto alla dimensione delle sequenze e al valore ottimale della stessa **distanza di edit**: in questo capitolo si propone una trattazione formale dell'algoritmo e una sua possibile estensione sui **grafi di variazione**.

3.1 Algoritmo wavefront per la distanza di edit tra due sequenze

Quando si calcola la **distanza di edit** tra due sequenze utilizzando le equazioni descritte nella sezione 2.2, la *matrice di programmazione dinamica* di dimensione $(n + 1) \times (m + 1)$ viene computata completamente, indipendentemente da quale sia il valore effettivo della **distanza di edit ottimale**:

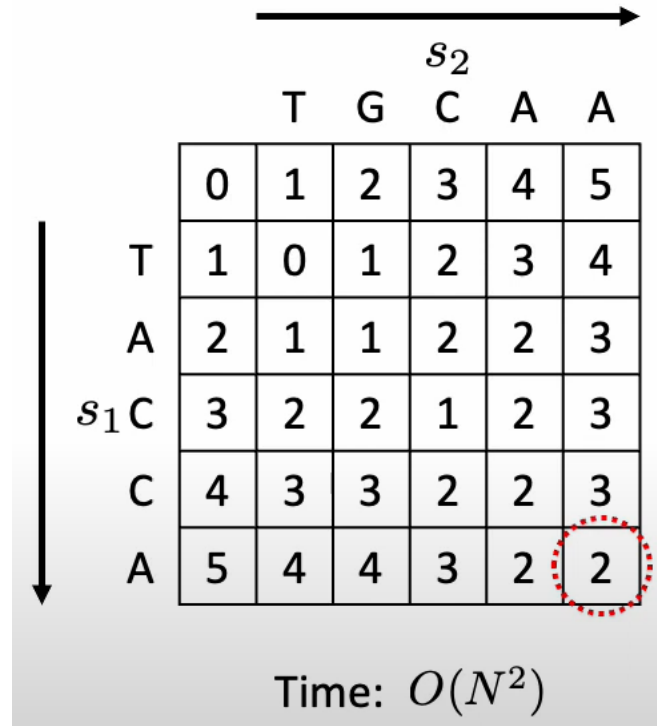


Figura 3.1: *Matrice di programmazione dinamica* per il calcolo della **distanza di edit** tra due sequenze. [10]

Tuttavia, è possibile notare che tutte le celle che hanno un valore strettamente maggiore della **distanza di edit ottimale** non contribuiscono al calcolo della soluzione ottimale e, di conseguenza, non si troveranno mai nel percorso che porta alla soluzione migliore. Questa è una conseguenza diretta del fatto che l'algoritmo di programmazione dinamica della **distanza di edit** risolve un *problema di minimizzazione* in cui le *penalità* utilizzate sono *strettamente positive*: di conseguenza, prese in ingresso le due sequenze $X_n = \langle x_1, x_2, \dots, x_n \rangle$ e $Y_m = \langle y_1, y_2, \dots, y_m \rangle$, vale la seguente proprietà:

$$D[i, j] \geq D[i - 1, j - 1]$$

e, in particolare

$$D[i, j] = D[i - 1, j - 1] \iff x_i = y_j$$

$$\forall i \in \{1, \dots, n\}, \forall j \in \{1, \dots, m\}$$

Detto a parole, le *diagonali* sono *crescenti in senso lato* su tutta la matrice, e *crescenti in senso stretto* se non sono presenti corrispondenze (*match*) nelle sottostringhe considerate. L'idea dietro all'algoritmo **wavefront** è di sfruttare questa proprietà per evitare di calcolare tutte le celle che corrispondono a una **distanza di edit** maggiore di quella ottimale: per far questo, gli indici e il contenuto della matrice di programmazione dinamica cambiano come segue:

- gli indici della matrice diventano la *diagonale* che si sta computando e lo **distanza di edit** a cui si è arrivati;

- il valore degli elementi della matrice diventa il maggior *offset* sulla *diagonale* di interesse per la *distanza di edit* considerata.

Si definisce **fronte d'onda** l'insieme composto dalle celle della matrice che hanno la stessa *distanza di edit*: esse vengono computate tutte durante la medesima iterazione dell'algoritmo. E' possibile suddividere **WFA** in due fasi principali:

- L'**estensione**, in cui tutte le diagonali appartenenti allo stesso *fronte d'onda* vengono appunto "estese" il più possibile (fino a quando ci sono dei **match** tra i simboli delle sequenze);
- L'**espansione**, in cui viene computato il *fronte d'onda successivo* a partire dalle diagonali del *fronte d'onda attuale*.

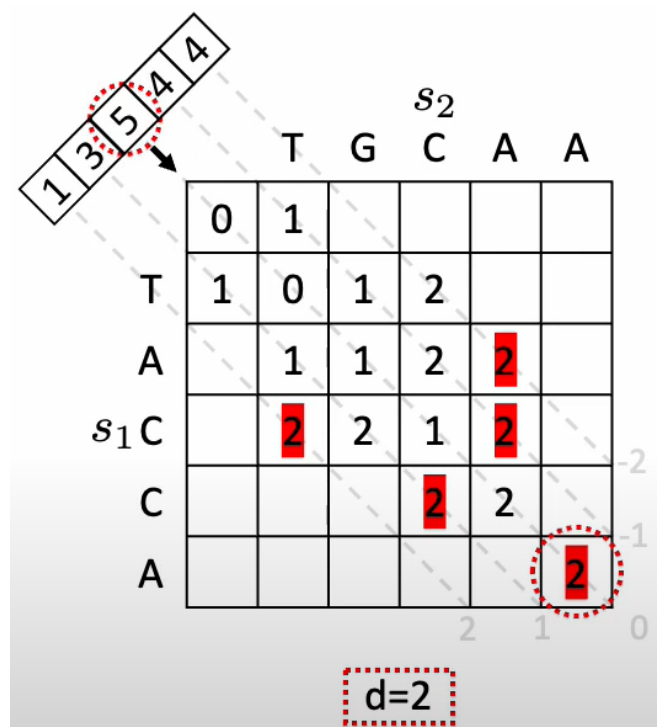


Figura 3.2: *Matrice di programmazione dinamica* virtualmente computata da **WFA**: i numeri in basso a destra rappresentano il numero della *diagonale*, mentre quelli in alto a sinistra il valore degli *offset* sulle rispettive *diagonali*. [10]

3.1.1 Equazioni di ricorrenza

Da un punto di vista matematico, questo viene espresso dalle seguenti equazioni:
siano

- Σ l'**alfabeto** di definizione,
- $X_n = \langle x_1, x_2, \dots, x_n \rangle$, $x_i \in \Sigma \forall i \in \{1, 2, \dots, n\}$ la **prima sequenza**,
- $Y_m = \langle y_1, y_2, \dots, y_m \rangle$, $y_j \in \Sigma \forall j \in \{1, 2, \dots, m\}$ la **seconda sequenza**,

•

$$D[i, j] = \begin{cases} D[i-1, j-1] & \text{se } x_i = y_j \\ \min \begin{cases} D[i, j-1] \\ D[i-1, j-1] \\ D[i-1, j] \end{cases} + 1 & \text{se } x_i \neq y_j \end{cases}$$

la *matrice di programmazione dinamica* computata dall'algoritmo standard per il calcolo della **distanza di edit** (sezione 2.2);

Si definiscono

• $k = j - i$ come la k -esima diagonale,

•

$$H[d, k] = \begin{cases} \max\{j \mid D[j-k, j] = d\}, & \text{se } \exists j \text{ s.t. } D[j-k, j] = d \\ -\infty, & \text{altrimenti} \end{cases} \quad (3.1)$$

come il valore del maggior *offset* per la k -esima diagonale tra le celle di programmazione dinamica con **distanza di edit** d (dopo una fase di **estensione**).

•

$$J[d, k] = \begin{cases} \min\{j \mid D[j-k, j] = d\}, & \text{se } \exists j \text{ s.t. } D[j-k, j] = d \\ -\infty, & \text{altrimenti} \end{cases} \quad (3.2)$$

come il valore del minor *offset* per la k -esima diagonale tra le celle di programmazione dinamica con **distanza di edit** d (dopo una fase di **espansione**).

Come conseguenza delle definizioni 3.1 e 3.2 vale la seguente implicazione:

$$D[j-k, j] = d \iff J[d, k] \leq j \leq H[d, k]$$

Ovvero l'*offset* di qualsiasi cella con **distanza di edit** d che si trova sulla diagonale k è compreso tra $J[d, k]$ e $H[d, k]$.

Le due fasi di **estensione** ed **espansione** vengono descritte dalle seguenti equazioni di ricorrenza:

Estensione del fronte d'onda

$$H[d, k] = j + LCP(X[j-k+1:], Y[j+1:]), \quad j = J[d, k] \quad (3.3)$$

dove

$$LCP(x, y) = |\text{Longest Common Prefix}(x, y)|$$

Espansione del fronte d'onda

$$J[d, k] = \max \begin{cases} H[d-1, k-1] + 1, \\ H[d-1, k] + 1, \\ H[d-1, k+1] \end{cases} \quad (3.4)$$

Le equazioni 3.3 e 3.4 esprimono l'alternarsi delle due fasi di **estensione** ed **espansione**: dopo aver computato tutti i **match** possibili, le diagonali vengono espanse per la **distanza di edit** successiva, per poi reiterare il procedimento.

3.1.2 Condizioni al contorno

Le **condizioni al contorno** vengono riformulate come segue:

Caso globale

Caso base:

$$H[0, 0] = 0 \quad (3.5)$$

Soluzione ottima:

$$d \text{ s.t. } H[d, m-n] = m$$

Caso semiglobale

Caso base:

$$H[0, -i] = 0, \quad \forall i \in \{0, \dots, n\} \quad (3.6)$$

Soluzione ottima:

$$d \text{ s.t. } H[d, m-i] = m, \quad \forall i \in \{0, \dots, n\}$$

3.1.3 Pseudocodice

Da un punto di vista algoritmico, le due procedure di **estensione** ed **espansione** vengono espresse tramite il seguente pseudocodice:

Extend

Algorithm 1: extend

Data: X_n, Y_m, d, H_d
for $k = -d$ **to** d **do**
 $i \leftarrow H[d][k] - k$;
 $j \leftarrow H[d][k]$;
 while $x_i = y_j$ **do**
 $i \leftarrow i + 1$;
 $j \leftarrow j + 1$;
 $H[d][k] \leftarrow H[d][k] + 1$;

Nell'estensione il numero di diagonali del d -esimo fronte d'onda non cambia.

Expand

Algorithm 2: expand

Data: X_n, Y_m, d, H_d
initialize H_{d+1} ;
 $k_{lo} \leftarrow -(d + 1)$;
 $k_{hi} \leftarrow d + 1$;
for $k = k_{lo}$ **to** k_{hi} **do**
 $H[d + 1, k] \leftarrow \max\{H[d, k - 1] + 1, H[d, k] + 1, H[d, k + 1]\}$;

Nella fase di espansione viene computato il fronte d'onda successivo: è necessario considerare le due nuove diagonali $-(d + 1)$ e $d + 1$.

A seguire lo pseudocodice per la procedura principale **WFA** nella **modalità globale**:

Caso globale

Algorithm 3: WFA_global

Data: X_n, Y_m
Result: $d = \text{edit_distance}(X_n, Y_m)$
initialize H_0 ;
 $H[0][0] \leftarrow 0$;
 $d \leftarrow 0$;
while true do
 $\text{extend}(X_n, Y_m, d, H_d)$;
 if $H[d, m - n] = m$ **then**
 break ;
 $\text{expand}(X_n, Y_m, d, H_d)$;
 $d \leftarrow d + 1$;
 $\text{traceback}(X_n, Y_m, d, H_d)$;
return d ;

E nella **modalità semiglobale**:

Caso semiglobale

Algorithm 4: WFA_semiglobal

Data: X_n, Y_m
Result: $d = \text{edit_distance}(X_n, Y_m)$
initialize H_0 ;
for $i = 0$ **to** n **do**
 $H[0][-i] = 0$
 $d \leftarrow 0$;
while true do
 $\text{extend}(X_n, Y_m, d, H_d)$;
 for $i = 0$ **to** n **do**
 if $H[d, m - i] = m$ **then**
 break ;
 $\text{expand}(X_n, Y_m, d, H_d)$;
 $d \leftarrow d + 1$;
 $\text{traceback}(X_n, Y_m, d, H_d)$;
return d ;

Si noti che nella **modalità semiglobale**, le sottoprocedure **extend** ed **expand** vanno modificate in maniera da considerare il range di diagonali $[-n, d]$ a ogni iterazione.

Infine un esempio (figura 3.3) che mostra la *matrice* computata da **WFA_global** e quella che verrebbe virtualmente calcolata dall'algoritmo standard:

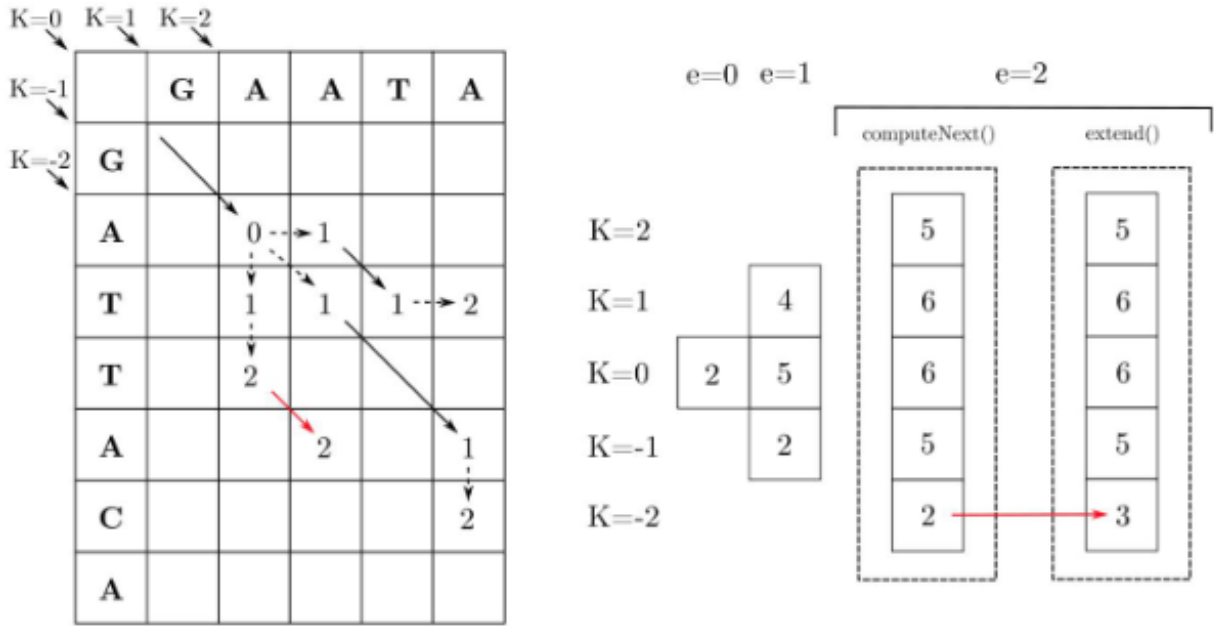


Figura 3.3: Matrice effettivamente calcolata da **WFA** rispetto a quella che verrebbe virtualmente computata dall'algoritmo standard. [1]

3.1.4 Complessità computazionale

Si consideri la seguente notazione:

- n = lunghezza della *sequenza più lunga (testo)*,
- m = lunghezza della *sequenza più corta (query)*,
- \bar{d} il valore ottimo della **distanza di edit** tra le due sequenze.

Si noti che la **distanza di edit** d è *limitata inferiormente* dalla differenza della lunghezza delle due sequenze ed è *limitata superiormente* dalla lunghezza maggiore tra le sequenze:

$$n - m \leq \bar{d} \leq n \quad (3.7)$$

Complessità in tempo

La **complessità in tempo** di **WFA** è determinata dalle operazioni presenti all'interno del ciclo principale, che viene eseguito una volta per ogni valore della **distanza di edit** $d \in \{0, 1, \dots, \bar{d}\}$ e contenente le tre sottoprocedure:

- **extend**,
- **controllo terminazione**,

- **expand**,

e dalla procedura **traceback**. Si procede a analizzare il tempo richiesto da ognuna delle sottoprocedure.

Extend

- **Caso globale**: nella sottoprocedura **extend** (algoritmo 1) il tempo di esecuzione è proporzionale al numero totale di **match** effettuato da tutte le diagonali, appartenenti all'insieme $\{-\bar{d}, -\bar{d} + 1, \dots, 0, \dots, \bar{d} - 1, \bar{d}\}$. E' possibile distinguere due casi:

– $m \leq \bar{d} \leq n \implies m = O(\bar{d})$, rappresentato dalla seguente figura

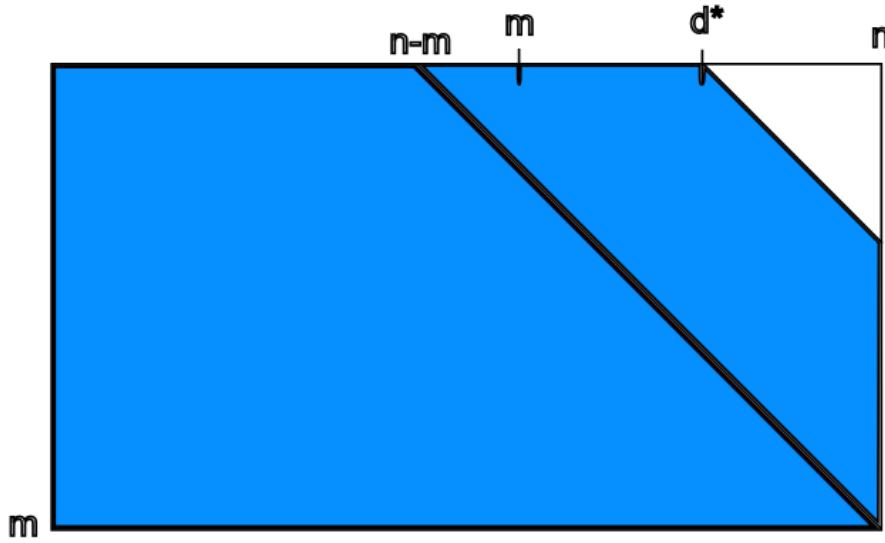


Figura 3.4: Sezione della *matrice di programmazione dinamica* in cui si possono verificare il massimo numero di **match** (caso $m \leq \bar{d} \leq n$).

In questo caso il *numero massimo* di **match** è dato da

$$\begin{aligned}
 T(n, m, \bar{d}) &= \\
 &= \sum_{j=0}^m m - j \quad + \quad \sum_{i=1}^{n-m} m \quad + \quad \sum_{i=1}^{\bar{d}-(n-m)} m - i = \\
 &= \frac{m(m+1)}{2} \quad + \quad m(n-m) \quad + \quad \frac{(\bar{d} - (n-m))(m + (n - \bar{d}) - 1)}{2}
 \end{aligned}$$

Per 3.7 si ha che

$$* \quad n - m \leq \bar{d} \implies n - m = O(\bar{d})$$

$$* \quad n - \bar{d} \leq m \implies n - \bar{d} = O(m)$$

e di conseguenza

$$T(n, m, \bar{d}) = O(m^2) + O(m) \cdot O(\bar{d}) + O(\bar{d}) \cdot O(m) = O(m \cdot \bar{d})$$

Si osservi che porre $n - m \leq m \leq \bar{d}$ non è limitante, in quanto è il caso in cui si considerano il maggior numero di diagonali (rispetto al caso $m \leq n - m \leq \bar{d}$ o al caso $m \leq \bar{d} \leq n - m$).

– $\bar{d} \leq m \leq n \implies m = O(\bar{d})$, rappresentato dalla seguente figura

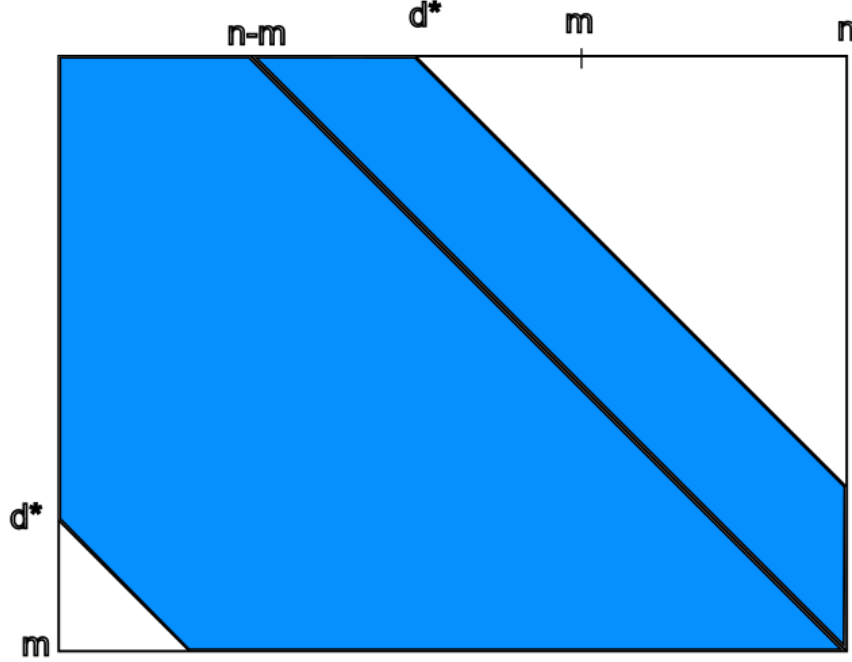


Figura 3.5: Sezione della *matrice di programmazione dinamica* in cui si possono verificare il massimo numero di **match** (caso $\bar{d} \leq m \leq n$).

Per questa altra possibilità si nota che viene computata una sezione di *matrice di programmazione dinamica* inferiore rispetto al caso precedente; quindi il tempo sarà sempre limitato da

$$T(n, m, \bar{d}) = O(m \cdot \bar{d})$$

Anche in questa casistica, porre $n - m \leq \bar{d} \leq m$ non è limitante.

Quindi, il tempo della sottoprocedura **extend** nel **caso globale** è *limitato superiormente* da

$$T(n, m, \bar{d}) = O(\min\{n, m\} \cdot \bar{d})$$

- **Caso semiglobale:** in questa modalità, vengono considerate tutte le diagonali $k \in \{0, -1, \dots, -n\}$ fin dalla prima iterazione; di conseguenza, il tempo di esecuzione risulta:

$$\begin{aligned} T(n, m, \bar{d}) &= \\ &= \sum_{j=0}^{\bar{d}} m - j \quad + \quad \sum_{i=1}^{n-m} m \quad + \quad \sum_{i=1}^{n-(n-m)} m - i = \\ &= \frac{(\bar{d} + 1) \cdot (2m - \bar{d})}{2} \quad + \quad m(n - m) \quad + \quad \frac{m(m - 1)}{2} \end{aligned}$$

ovvero

$$T(n, m, \bar{d}) = O(m \cdot (\bar{d} + m))$$

Controllo terminazione

- Per quanto riguarda il **caso globale** (algoritmo 3), il **controllo di terminazione** viene effettuato in tempo costante:

$$T(n, m, \bar{d}) = O(1)$$

- Nel **caso semiglobale** (algoritmo 4), invece, è necessario controllare a ogni iterazione gli offset di tutte le diagonali corrispondenti a una posizione della sequenza più lunga (chiamata **testo**); il tempo risulta quindi essere

$$T(n, m, \bar{d}) = O(\max\{n, m\} \cdot \bar{d})$$

Expand

- **Caso globale:** nella sottoprocedura **expand** (algoritmo 2) ad ogni iterazione vengono effettuate $O(2d + 1)$ operazioni: di conseguenza, la complessità temporale risulta essere limitata da

$$O(\sum_{d=0}^{\bar{d}} 2d + 1) = O(\bar{d} \cdot (\bar{d} + 2)) = O(\bar{d}^2), \text{ ovvero:}$$

$$T(n, m, \bar{d}) = O(\bar{d}^2)$$

- **Caso semiglobale:** come succede nella **extend**, anche qui è necessario considerare tutte le diagonali $k \in \{0, -1, \dots, -n\}$ fin dalla prima iterazione; il tempo risulta quindi $O(\sum_{d=0}^{\bar{d}} d + n) = O(\max\{n, m\} \cdot \bar{d}) + O(\bar{d}^2)$, ossia

$$T(n, m, \bar{d}) = O(\max\{n, m\} \cdot \bar{d})$$

Traceback

Come nel calcolo della **distanza di edit** standard, il tempo richiesto per ricostruire la soluzione è

$$T(n, m, \bar{d}) = O(n + m)$$

per entrambe le modalità.

La **complessità in tempo** totale risulta quindi essere:

- **Caso globale:**

$$\begin{aligned} T(n, m, \bar{d}) &= O(\min\{n, m\} \cdot \bar{d} + O(1) + O(\bar{d}^2) + O(n + m)) = \\ &= O(\min\{n, m\} \cdot \bar{d} + \bar{d}^2 + (n + m)) \end{aligned}$$

se interessati anche alla **ricostruzione della soluzione**, altrimenti

$$T(n, m, \bar{d}) = O(\min\{n, m\} \cdot \bar{d} + \bar{d}^2) \quad (3.8)$$

- **Caso semiglobale:**

$$T(n, m, \bar{d}) = O(m \cdot (m + \bar{d})) + 2 \cdot O(\max\{n, m\} \cdot \bar{d}) + O(n + m)$$

ossia

$$T(n, m, \bar{d}) = O(\max\{n, m\} \cdot \bar{d}) \quad (3.9)$$

Complessità in spazio

- **Caso globale:** per quanto riguarda la **complessità in spazio**, ogni *fronte d'onda* ha una dimensione proporzionale a $2d + 1$, con d che varia da 0 a \bar{d} ; di conseguenza, la **complessità spaziale**¹ nel **caso globale** è data da

$$M(n, m, \bar{d}) = O(\bar{d}^2) \quad (3.10)$$

- **Caso semiglobale:** in questa modalità, vale la stessa osservazione del **caso globale**; tuttavia, è necessario considerare anche le diagonali $k \in \{0, -1, \dots, -n\}$ fin dalla prima iterazione. La **complessità in spazio**² diventa quindi limitata da $O(\sum_{d=0}^{\bar{d}} d + n) = O(\max\{n, m\} \cdot \bar{d}) + O(\bar{d}^2)$, e di conseguenza

$$M(n, m, \bar{d}) = O(\max\{n, m\} \cdot \bar{d}) \quad (3.11)$$

3.1.5 Dimostrazione di correttezza

Si considerino due **sequenze** $X_n = \langle x_1, x_2, \dots, x_n \rangle$ e $Y_m = \langle y_1, y_2, \dots, y_m \rangle$; la **distanza di edit** tra qualsiasi coppia di **prefissi** $X[:i]$, $Y[:j]$ è descritta dalla seguente equazione

$$D[i, j] = \min \begin{cases} D[i-1, j] + 1 \\ D[i-1, j-1] + \Delta_{i,j} \\ D[i, j-1] + 1 \end{cases}$$

$$\Delta_{i,j} = \begin{cases} 0 & \text{se } x_i = y_j \\ 1 & \text{altrimenti} \end{cases}$$

che è **equivalente** alle seguenti equazioni

$$J[d, k] = \max \begin{cases} H[d-1, k+1] \\ H[d-1, k] + 1 \\ H[d-1, k-1] + 1 \end{cases} \quad (3.12)$$

$$H[d, k] = j + LCP(X[j-k+1:], Y[j+1:]), \quad j = J[d, k] \quad (3.13)$$

¹Se non interessati alla **ricostruzione della soluzione**, la complessità scende a $M(n, m, \bar{d}) = O(\bar{d})$.

²Se non interessati alla **ricostruzione della soluzione**, la complessità scende a $M(n, m, \bar{d}) = O(\max\{n, m\})$.

con

$$LCP(X, Y) = |Longest Common Prefix(X, Y)|$$

Dove $H[d, k]$ è il **maggior offset** sulla **diagonale** $k = j - i$ per lo **score** d , definito come segue:

$$H[d, k] = \begin{cases} \max\{j \mid D[j - k, j] = d\}, & \text{se } \exists j \text{ s.t. } D[j - k, j] = d \\ -\infty, & \text{altrimenti} \end{cases} \quad (3.14)$$

E $J[d, k]$ è il **minor offset** sulla **diagonale** $k = j - i$ per lo **score** d :

$$J[d, k] = \begin{cases} \min\{j \mid D[j - k, j] = d\}, & \text{se } \exists j \text{ s.t. } D[j - k, j] = d \\ -\infty, & \text{altrimenti} \end{cases} \quad (3.15)$$

Dimostrazione:

come conseguenza delle definizioni 3.14 e 3.15, vale la seguente implicazione:

$$D[j - k, j] = d \iff J[d, k] \leq j \leq H[d, k] \quad (3.16)$$

Si definisca

$$F[d, k, l] = \begin{cases} \max\{j \mid D[i, j] = d, j \leq l\}, & \text{se } \exists j \text{ s.t. } D[i, j] = d \\ -\infty, & \text{altrimenti} \end{cases} \quad (3.17)$$

come il **maggior offset** sulla **diagonale** k per lo **score** d *vincolato a essere minore o uguale* di l ; come conseguenza della definizione, valgono le seguenti proposizioni:

• Proposizione 1

$$D[j - k, j] = d \iff F[d, k, j] = j \quad (3.18)$$

analoga a 3.16 per $F[d, k, l]$

• Proposizione 2

$$F[d, k, l] \leq F[d, k, l + 1]$$

• Proposizione 3

$$J[d, k] = \min_l \{F[d, k, l]\}$$

$$H[d, k] = \max_l \{F[d, k, l]\}$$

E' quindi possibile riscrivere le equazioni 3.12 e 3.13 come

$$F[d, k, j] = \begin{cases} F[d, k, j - 1] + 1, & \text{if } x_{j-k} = y_j \\ \max \begin{cases} F[d - 1, k - 1, j - 1] + 1 \\ F[d - 1, k, j - 1] + 1 \\ F[d - 1, k + 1, j] \end{cases} & \text{if } x_{j-k} \neq y_j \end{cases} \quad (3.19)$$

Infatti, in caso di **mismatch** ($x_i \neq y_j$), $F[d, k, j]$ viene computato a partire dagli offset per lo score $d - 1$, come succede in 3.12; invece, in caso di **match** ($x_i = y_j$), $F[d, k, j]$ viene calcolato unicamente dagli offset per lo stesso score d ; questo continua ad accadere finchè $x[j - k + q] = y[j + q]$, con $q \in \{0, \dots, LCP(X[j - k + 1 :], Y[j + 1 :])\}$, in maniera analoga a 3.13. Per **ipotesi di induzione** siano noti

- $F[d, k, j] = j$, il maggior offset sulla diagonale k per lo score d vincolato a essere minore o uguale a j
- $D[i, j] = d$, il valore ottimo della distanza di edit tra $X[:i]$ e $Y[:j]$

Sono possibili in tutto 4 casi:

- **Match:** $x_{i+1} = y_{j+1}$

In questo caso, il valore della distanza di edit resta invariato

$$D[i + 1, j + 1] = D[i, j] = d$$

e l'offset sulla diagonale k per lo score d aumenta di uno

$$F[d, k, j + 1] = F[d, k, j] + 1 = j + 1$$

per 3.18 vale

$$F[d, k, j + 1] = j + 1 \implies D[(j + 1) - k, j + 1] = d$$

ossia

$$D[(j + 1) - k, j + 1] = D[(j - k) + 1, j + 1] = D[i + 1, j + 1] = d$$

- **Mismatch:** $x_{i+1} \neq y_{j+1}$

In questo caso ci sono 3 possibilità:

- **Sostituzione**, ovvero si sostituisce x_{i+1} con y_{j+1} , di conseguenza

$$D[i + 1, j + 1] = D[i, j] + 1 = d + 1$$

e l'offset sulla diagonale k per lo score $d + 1$ aumenta di uno

$$F[d + 1, k, j + 1] = F[d, k, j] + 1 = j + 1$$

per 3.18 vale

$$F[d + 1, k, j + 1] = j + 1 \implies D[(j + 1) - k, j + 1] = d + 1$$

ossia

$$D[(j + 1) - k, j + 1] = D[(j - k) + 1, j + 1] = D[i + 1, j + 1] = d + 1$$

- **Inserimento**, ovvero si inserisce y_{j+1} dopo $X[:i]$, di conseguenza

$$D[i, j+1] = D[i, j] + 1 = d+1$$

e l'offset sulla diagonale $k+1$ per lo score $d+1$ aumenta di uno

$$F[d+1, k+1, j+1] = F[d, k, j] + 1 = j+1$$

per 3.18 vale

$$F[d+1, k+1, j+1] = j+1 \implies D[(j+1) - (k+1), j+1] = d+1$$

ossia

$$D[(j+1) - (k+1), j+1] = D[(j-k) + 1 - 1, j+1] = D[i, j+1] = d+1$$

- **Cancellazione**, ovvero x_{i+1} viene cancellato, di conseguenza

$$D[i+1, j] = D[i, j] + 1 = d+1$$

e l'offset sulla diagonale $k-1$ per lo score $d+1$ resta invariato

$$F[d+1, k-1, j] = F[d, k, j] = j$$

per 3.18 vale

$$F[d+1, k-1, j] = j \implies D[j - (k-1), j] = d+1$$

ossia

$$D[j - (k-1), j] = D[(j-k) + 1, j] = D[i+1, j] = d+1$$

Non esistono altre possibilità, quindi i quattro casi elencati coprono tutte le casistiche.

3.1.6 Caso pesato

Risulta abbastanza immediato generalizzare l'algoritmo per l'utilizzo di *penalità generiche*. [3]

Considerate in ingresso

- ***ins*** = penalità per un *inserimento*,
- ***m*** = penalità per una *sostituzione*,
- ***del*** = penalità per una *cancellazione*,

con $(ins, m, del) \in \mathbb{Q}_+^3$, è possibile riscrivere l'equazione 3.4 come:

$$J[d, k] = \max \begin{cases} H[d - ins, k - 1] + 1 \\ H[d - m, k] + 1 \\ H[d - del, k + 1] \end{cases} \quad (3.20)$$

Per quanto riguarda lo pseudocodice, l'unica procedura a essere modificata è **expand** (algoritmo 2):

Algorithm 5: expand

Data: $X_n, Y_m, d, H_d, ins, m, del$

initialize H_{d+1} ;

$k_{lo} \leftarrow -(d + 1)$;

$k_{hi} \leftarrow d + 1$;

for $k = k_{lo}$ **to** k_{hi} **do**

| $H[d + 1, k] \leftarrow$
| $\max\{H[d + 1 - ins, k - 1] + 1, H[d + 1 - m, k] + 1, H[d + 1 - del, k + 1]\};$

3.2 Distanza di edit su grafi di variazione con wavefront

Sfuttando la proprietà dei **grafi di variazione** di presentare *percorsi lineari* (sezione 2.3.3), è possibile estendere l'approccio **wavefront** su queste strutture dati³, aggiungendo alle *matrici di programmazione dinamica* una dimensione per memorizzare i diversi percorsi.

3.2.1 Equazioni di ricorrenza

Si considerino in ingresso

- un **grafo di variazione canonico** $G = (V, E, \delta, P)$, dove
 - $V = \{v_0, v_1, v_2, \dots, v_n\}$ è l'insieme dei **vertici** del grafo;
 - $E \subseteq V \times V$ è l'insieme degli **archi** del grafo;
 - $\delta : V \rightarrow \Sigma$ è la **funzione di etichettatura**,
 - $P = \{p_1, p_2, \dots, p_k\}$, $p_i = \langle v_0 = v_{i_1}, v_{i_2}, \dots, v_{i_l} = v_n \rangle \forall i \in \{1, 2, \dots, k\}$ è l'insieme dei **percorsi** del grafo.
- una **sequenza** $Y_m = \langle y_1, y_2, \dots, y_m \rangle$;
- le **penalità** di **inserimento**, **sostituzione** e **cancellazione** (ins, m, del) .

³Diverse estensioni su **grafi di sequenza** (come [7]) e [10]) sono già state proposte e sviluppate da diversi autori.

e si definisca il *suffisso di un cammino* come

$$p[j:] = \delta(p_j)\delta(p_{j+1})\dots\delta(p_n)$$

$$\forall p \in P, \forall j \in \{0, \dots, |p|\}$$

Le equazioni 3.3 e 3.4 possono essere riformulate come segue:

Estensione del fronte d'onda

$$H[d, k, p] = j + LCP(p[j - k + 1:], Y[j + 1:]), \quad i = J[d, k] \quad (3.21)$$

dove

$$LCP(x, y) = |Longest Common Prefix(x, y)|$$

Espansione del fronte d'onda

$$J[d, k, p] = \max \begin{cases} H[d - ins, k - 1, p] + 1, \\ H[d - m, k, p] + 1, \\ H[d - del, k + 1, p] \end{cases} \quad (3.22)$$

3.2.2 Condizioni al contorno

Allo stesso modo, anche le condizioni al contorno 3.5 e 3.6 possono essere riscritte come:

Caso globale

Caso base:

$$H[0, 0, p] = 0, \quad \forall p \in P \quad (3.23)$$

Soluzione ottima:

$$\min_{p \in P} \left\{ d \text{ s.t. } H_{d, m-n, p} = m \right. \quad (3.24)$$

Caso semiglobale

Caso base:

$$H[0, -i, p] = 0, \quad \forall i \in \{0, \dots, n\}, \forall p \in P \quad (3.25)$$

Soluzione ottima:

$$\min_{p \in P} \left\{ d \text{ s.t. } H_{d, m-i, p} = m \quad \forall i \in \{0, \dots, n\} \right. \quad (3.26)$$

3.2.3 Complessità computazionale

Ogni percorso viene computato in maniera indipendente dagli altri, mantenendo la stessa complessità dell'allineamento tra due stringhe;

Considerando la notazione

- $n = \frac{\sum_{p \in P} |p|}{|P|}$, *lunghezza media dei percorsi del grafo di variazione*,
- $m =$ lunghezza della *sequenza (query)*,
- $\bar{d} =$ valore ottimo della **distanza di edit** tra uno dei cammini del grafo e la sequenza,
- $p =$ numero dei *percorsi* del **grafo di variazione**,

la complessità per questo tipo di allineamento risulta:

Caso globale

Tempo:

$$T(n, m, \bar{d}) = O((\min\{n, m\} + \bar{d}) \cdot \bar{d} \cdot p) \quad (3.27)$$

Spazio:

$$M(n, m, \bar{d}) = O(\bar{d}^2 \cdot p) \quad (3.28)$$

Caso semiglobale

Tempo:

$$T(n, m, \bar{d}) = O(n \cdot \bar{d} \cdot p) \quad (3.29)$$

Spazio:

$$M(n, m, \bar{d}) = O(n \cdot \bar{d} \cdot p) \quad (3.30)$$

Si noti che gli allineamenti tra i diversi cammini sono del tutto *indipendenti* e quindi possono essere facilmente *parallelizzati*.

Capitolo 4

Progettazione e implementazione del prototipo

In seguito allo studio dell'algoritmo *wavefront* si è passati a una fase di implementazione di un prototipo, che potesse andare a migliorare le prestazioni del tool *RecGraph* [2] nell'allineamento tra un *grafo di variazione* e una *sequenza*. In seguito vengono esposte la fasi di **progettazione** e **implementazione** del prototipo, per poi passare a un **benchmark** che va a confrontare le prestazioni dell'algoritmo *wavefront* con quelle della implementazione di *RecGraph*.

4.1 Progettazione

Nella seguente sezione vengono riportate le scelte prese in **fase di progettazione** del prototipo e la **struttura architettonica** a cui hanno portato. L'intera implementazione del prototipo è disponibile al seguente link:
https://github.com/iFoxz17/WF_Recgraph.

4.1.1 Organizzazione modulare

Si è scelto di organizzare il prototipo seguendo un approccio ad *elevata modularità*, in maniera da rendere più semplice l'integrazione di eventuali future modifiche o migliorie:

Come è possibile vedere dalla figura 4.1, viene definita un'interfaccia *wfa* che espone le quattro modalità di allineamento implementate:

- *wf_pathwise_alignment_global*, performa un allineamento in *modalità globale*;
- *wf_pathwise_alignment_semiglobal*, performa un allineamento in *modalità semiglobale*;
- *wf_pathwise_alignment_start_free_global*, performa un allineamento che può iniziare da qualsiasi vertice del grafo, ma vincolato a finire necessariamente nell'ultimo nodo;

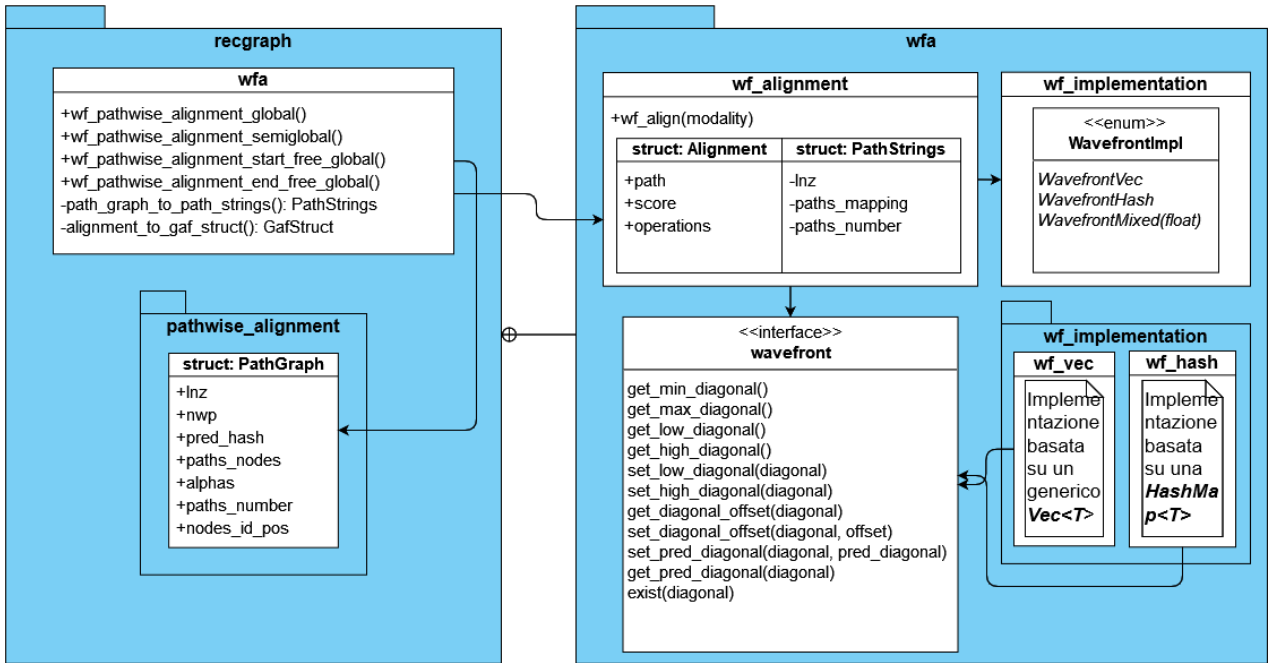


Figura 4.1: Organizzazione modulare del prototipo.

- ***wf_pathwise_alignment_end_free_global***, performa un allineamento che può finire in qualsiasi vertice del grafo, ma vincolato a iniziare necessariamente nel primo nodo.

Inoltre, all'interno del modulo sono contenute altre funzioni per convertire le *strutture dati* utilizzate da **RecGraph** in quelle utilizzate all'interno del package **wfa**, contenente le seguenti componenti:

- ***wavefront***: interfaccia per definire le operazioni che deve implementare una struttura volta a rappresentare un *fronte d'onda*;
- ***wf_implementation***: modulo che contiene le implementazioni disponibili per rappresentare un *fronte d'onda* (devono implementare l'interfaccia ***wavefront***);
- ***wf_alignment***: modulo contenente le funzioni e le strutture dati che effettuano l'allineamento utilizzando l'algoritmo ***wavefront***.

4.1.2 Integrazione in RecGraph

Per integrare il prototipo all'interno di **RecGraph** (al fine di utilizzare le già implementate funzionalità di lettura e scrittura di grafi) si è utilizzato il modulo **wfa**, utilizzando direttamente le funzioni di allineamento esposte al posto di quelle di **RecGraph**; lo stesso modulo si occupa della conversione delle differenti strutture dati utilizzate. La figura 4.2 mostra la *struttura modulare* di **RecGraph** e come è stato integrato il prototipo al suo interno.

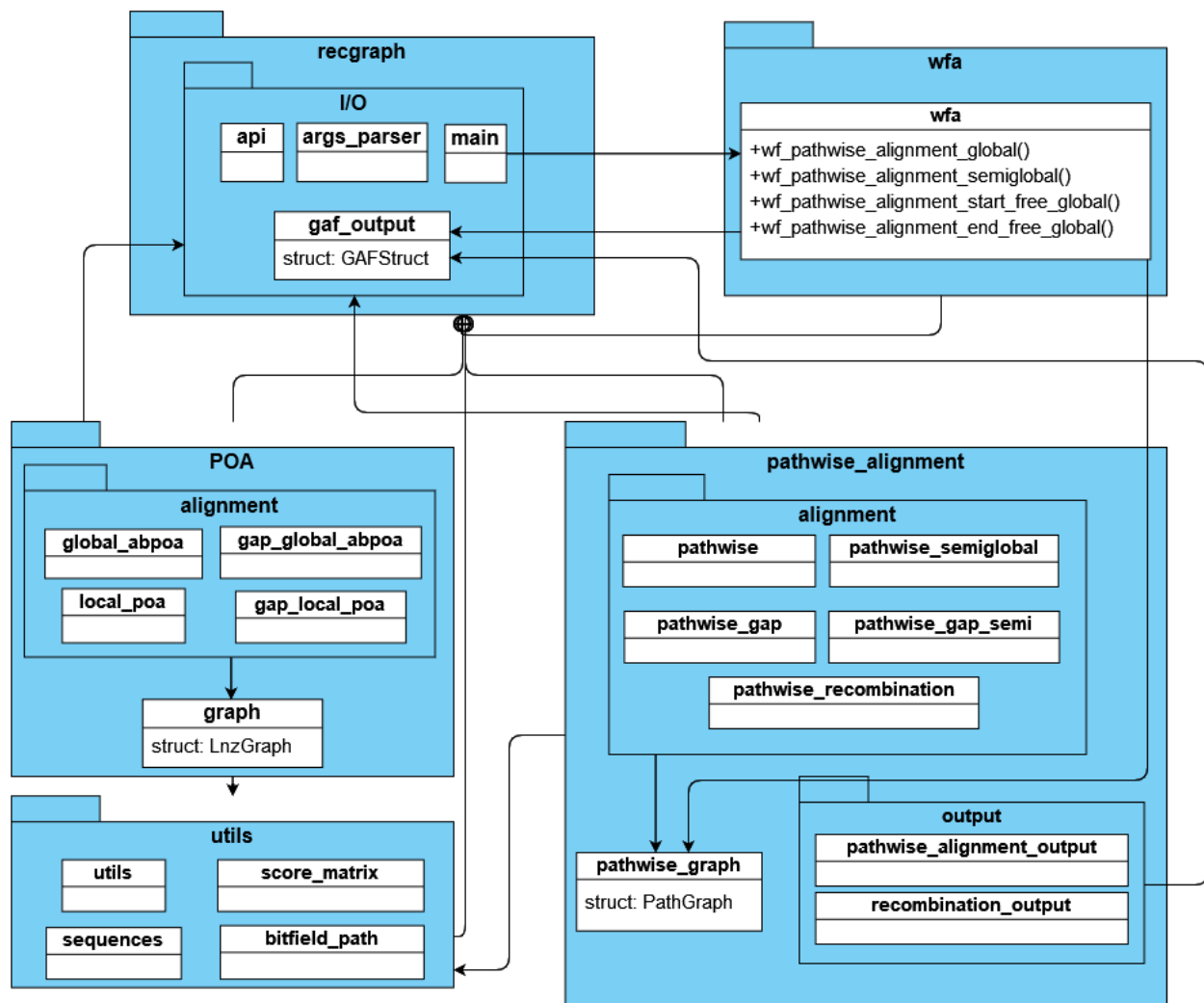


Figura 4.2: Integrazione del prototipo in *RecGraph*.

Come è possibile vedere dalle *relazioni di dipendenza*, il prototipo si serve soltanto delle due strutture dati *pathwise_graph* e *gaf_output*, utilizzate rispettivamente per rappresentare un *grafo di variazione* e per contenere le informazioni di un allineamento in formato *gaf*; inoltre, esso viene utilizzato direttamente all'interno del *main*. Si riesce così a ottenere un basso accoppiamento dei moduli, pratica consigliata da *design pattern* come *low coupling*, al fine di diminuire l'impatto di eventuali futuri cambiamenti sul sistema.

4.2 Implementazione

In questa sezione si discutono le **scelte implementative** prese nella costruzione del prototipo, sviluppato tramite il linguaggio di programmazione *Rust*.

4.2.1 Interfaccia *wavefront*

Modulo che definisce il *trait* **Wavefront** (struttura utilizzata in **Rust** per definire un'interfaccia), il quale specifica le operazioni che devono essere implementate per rappresentare un *fronte d'onda*; per mantenere la complessità in tempo dell'algoritmo *wavefront*, tutte le operazioni devono essere eseguite in $O(1)$.

4.2.2 Modulo *wf_implementation*

Modulo che definisce le implementazioni disponibili per rappresentare un *fronte d'onda*: le definizioni si trovano all'interno della *enum* **WavefrontImpl**, mentre le effettive implementazioni nel *sotto-package* **wf_implementation** (figura 4.1). In particolare, sono state fornite due implementazioni:

- **wf_vec**, memorizza il valore degli offsets delle diagonali all'interno di un **Vec**; risulta *più veloce* dell'altra implementazione, ma necessita di salvare anche gli offset delle diagonali che non vengono computate, generando uno *spreco di memoria* e portando la **complessità in spazio** a

$$T(n, m, p, \bar{d}) = O((n + m) \cdot \bar{d} \cdot p) \quad (4.1)$$

- **wf_hash**, salva il valore degli offsets delle diagonali all'interno di una **HashMap**; può risultare particolarmente *efficiente in memoria* quando il numero delle diagonali effettivamente computate è molto inferiore al totale, permettendo di evitare la memorizzazione di tutte le diagonali che non vengono computate e mantenendo l'*andamento in memoria* a

$$T(n, m, p, \bar{d}) = O(\bar{d}^2 \cdot p) \quad (4.2)$$

E' stata definita anche una terza implementazione, **WavefrontMixed**, che cerca di unire i *punti di forza* delle altre due: finché la *percentuale delle diagonali* che si sta computando è minore a una certa *soglia parametrizzata*, viene scelto l'approccio **WavefrontHash**, risparmiando così *memoria* senza perdere troppo in *tempo* (scegliendo soglie abbastanza basse, le diagonali effettivamente computate risultano molto inferiori a quelle totali, e la maggior parte del tempo dell'allineamento è richiesto quando il numero di diagonali che si stanno trattando cresce); superata la soglia, si passa invece all'implementazione **WavefrontVec**, di cui viene ridotto lo *spreco di memoria* (siccome il numero di diagonali che si stanno computando è vicino al massimo) e che risulta *più veloce*.

Inoltre, il *tipo degli interi* con cui vengono salvati gli offset risulta *generico* sia in **wf_vec** che in **wf_hash**: a seconda delle dimensioni del *grafo* e delle *reads* in ingresso viene scelto in automatico il tipo più piccolo tra quelli disponibili (**u8**, **u16**, **u32**, **u64** e **u128**), permettendo un significativo *risparmio di memoria* (si osservi che per grafi con fino a 65535 nodi è possibile utilizzare degli **u16**).

4.2.3 Modulo *wf_alignment*

Modulo che si occupa di effettuare effettivamente l'allineamento tra il *grafo di variazione* e la *sequenza*. Nella figura 4.3 viene fornito un diagramma che illustra le sue componenti più importanti:

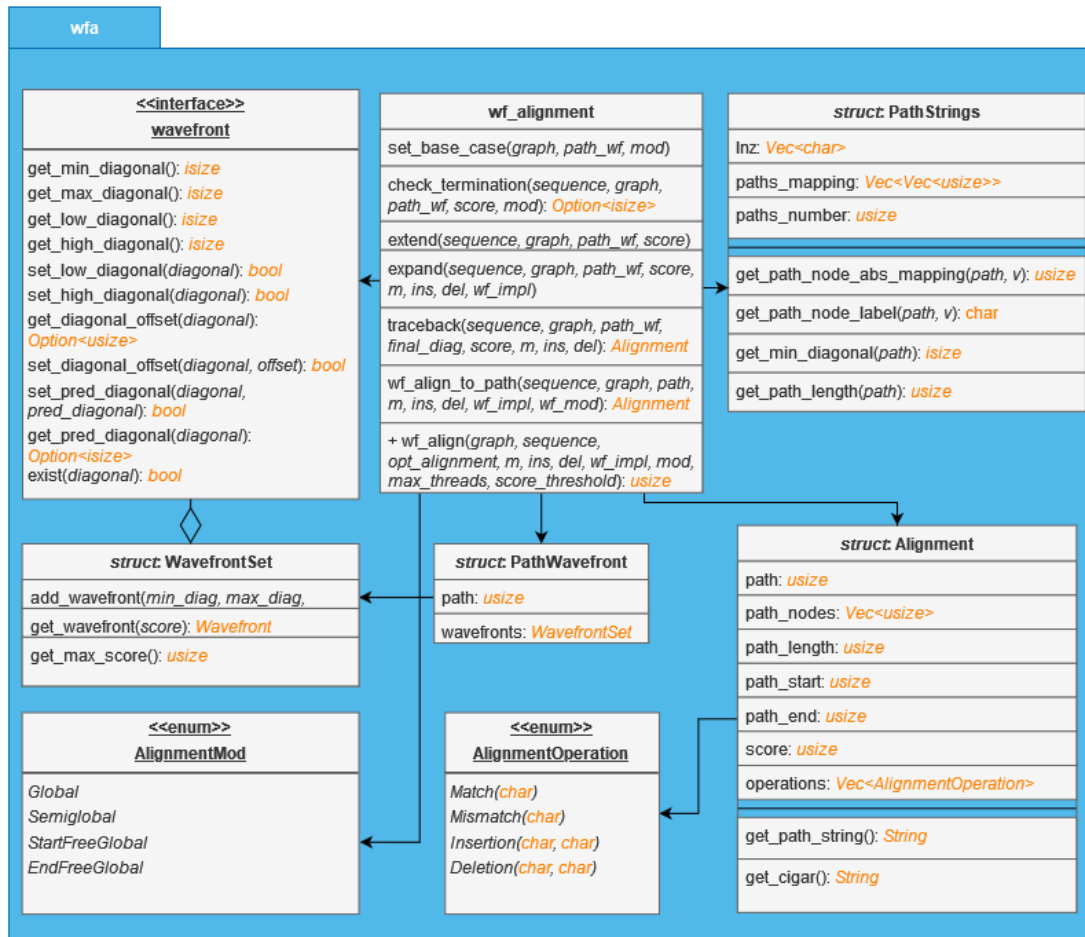


Figura 4.3: Modulo *wf_alignment* per la gestione dell'allineamento.

In ingresso, il *grafo di variazione* viene memorizzato tramite la ***struct PathStrings***, che essenzialmente estrae tutti i cammini del grafo; in output, invece, viene restituita un'istanza della ***struct Alignment***, contenente tutte le informazioni dell'allineamento effettuato. L'insieme dei *fronti d'onda* computati per ogni *punteggio* viene memorizzato nella ***struct WavefrontSet***: questa, insieme al *cammino* che si sta trattando, va a comporre la ***struct PathWavefront***, che contiene tutte le informazioni necessarie per effettuare l'allineamento a un percorso. A livello di interfaccia viene esposta solamente una funzione, ***wf_align***, che si occupa di gestire gli allineamenti tra i percorsi del grafo, lanciando un *thread* per ogni percorso e selezionando il miglior allineamento trovato. In particolare, riceve in ingresso¹

- il *grafo di variazione* (come ***PathStrings***),
- la *sequenza* da allineare,
- un *array* dove inserire gli allineamenti ottimali trovati (come ***Alignment***),
- le *penalità* di ***mismatch***, ***inserimento*** e ***cancellazione***.
- la *modalità di allineamento* (una voce della *enum AlignmentMod*),
- il numero massimo di *thread* da eseguire in parallelo,
- una *soglia* a cui troncare l'allineamento (facoltativa)

e restituisce il valore dell'allineamento prodotto. Inoltre, ogni volta che un thread termina un allineamento, viene impostata (o eventualmente aggiornata) una *soglia* sul *massimo punteggio raggiungibile*, condivisa da tutti i thread: ogni allineamento che supera tale soglia viene abbandonato. Ogni thread lanciato esegue la funzione ***wf_align_to_path***, che esegue l'algoritmo *wavefront* tra un percorso del grafo e una sequenza (quindi tra due stringhe), alternando le già discusse fasi di *estensione* ed *espansione*.

¹Per ragioni di leggibilità, sono stati riportati solo i parametri più importanti.

4.3 Benchmark

In seguito vengono riportate alcune sperimentazioni eseguite per osservare il comportamento del prototipo. In particolare, nella prima sezione si analizzano le performance dell'allineamento *wavefront* rispetto a **RecGraph**, mentre nella sezione successiva si tratta l'andamento delle prestazioni del prototipo rispetto a diversi **gradi di libertà** (*punteggio dell'allineamento, numero di thread utilizzati, numero di cammini del grafo di variazione*). La sperimentazione è stata eseguita su un server con 64 core e 256 GB di RAM. Per quanto riguarda le grandezze in input, si fa riferimento alla seguente notazione:

- *n*: lunghezza media dei cammini del *grafo di variazione*;
- *p*: numero di cammini del *grafo di variazione*;
- *m*: lunghezza della *sequenza*;
- *d*: valore ottimale dell'allineamento.

4.3.1 RecGraph vs Wavefront

Grafo di variazione

In questa prima fase della sperimentazione è stato usato il **grafo di variazione** descritto nella tabella 4.1:

Variation graph			
Linearization	Paths		
Length (bp)	Number	Paths mean length (bp)	Paths for node
49376	30	29744	18.1 (60%)

Tabella 4.1: Caratteristiche del *grafo di variazione* usato nel benchmark.

In questa istanza si nota un'importante *sovrapposizione dei cammini*: per ogni nodo, infatti, passano in media 18 cammini su 30 totali (più del 60%). Questa caratteristica va a penalizzare l'implementazione dell'algoritmo *wavefront* in quanto, basandosi sull'estrazione dei cammini dal grafo per permettere il *multithreading*, va a computare diverse volte i valori degli allineamenti per gli stessi nodi condivisi da più percorsi; al contrario, **RecGraph** sfrutta a suo vantaggio questa proprietà, memorizzando soltanto il punteggio di un solo percorso per i nodi condivisi (ottimizzazione del *reference path* (sezione 2.4.4)).

Si riportano i risultati ottenuti su **reads** di diversa lunghezza, sia in **modalità globale** che in **modalità semiglobale**.

Modalità globale

Per la sperimentazione in **modalità globale** sono state usate *sequenze* di lunghezza 150 bp, 1000 bp, 10000 bp e 25000 bp; i seguenti valori riguardano allineamenti eseguiti utilizzando **RecGraph**:

Modality: Global							
	Reads		Time (s)				Memory (GB)
Algorithm	N	L	Tot	Mean	Max	Min	Max
<i>RecGraph</i>	25	150	75	3.0	3.6	2.5	2.17
	10	1000 6.7×	214	21.4 7.1×	26.4 7.3×	20.0 8×	7.84 3.6×
	5	10000 66.7×	1155	231.1 77.0×	241.4 67.1×	216.2 86.5×	75.64 34.8×
	5	25000 167×	6909	690.9 230×	703.2 195×	678.5 271.4×	192.45 88.7×

Tabella 4.2: Benchmark per **RecGraph** in *modalità globale*.

I dati evidenziano una **complessità lineare** nella *lunghezza delle sequenze*, sia per quanto riguarda il **tempo** che per lo **spazio**, in linea con la complessità $O(n \cdot m \cdot p)$ attesa. Sulle **reads** di lunghezza 25000 bp inizia a essere necessaria anche un'importante quantità di memoria (≈ 200 GB).

A seguire invece i valori ottenuti dall'algoritmo **wavefront** con *distanza di edit unitaria*:

Modality: Global								
	Reads		Time (s)				Memory (GB)	Score
Algorithm	N	L	Tot	Mean	Max	Min	Max	Mean
<i>wf_edit-distance</i>	25	150	545	21.8	24.5	20.2	111.33	28879
	10	1000 6.7×	222	22.2 1.02×	24.9 1.02×	20.2 1×	110.32 0.99×	28029 0.97×
	5	10000 66.7×	139.7	28.0 1.28×	29.0 1.18×	26.8 1.34×	111.23 1.0×	19669 0.68×
	5	25000 167×	134	13.4 0.6×	15.1 0.6×	12.1 0.6×	64.42 0.58×	8149 0.28×

Tabella 4.3: Benchmark per **wf_edit-distance** in *modalità globale*.

Qui si nota un andamento del tutto diverso: l'allineamento risulta *meno performante* per le sequenze di lunghezza 150 bp, 1000 bp e 10000 bp (che producono punteggi di allineamento molto alti), mentre risulta più *efficiente* per le sequenze di 25000 bp, migliorando sia in **tempo di esecuzione** sia in **memoria occupata** in accordo con la complessità $O(\min\{n, m\} \cdot \bar{d} \cdot p)$ prevista.

Di seguito un confronto dei valori ottenuti dalle esecuzioni di *RecGraph* e di *wf_edit-distance* per sequenze della stessa lunghezza:

Modality: <i>Global</i>		Reads number: <i>25</i>		Reads length: <i>150 bp</i>	
	Time (s)				Memory (GB)
Algorithm	Tot	Mean	Max	Min	Max
<i>RecGraph</i>	75	3.0	3.6	2.5	2.17
<i>wf_edit-distance</i>	545	21.4	24.5	20.2	111.33
	7.3×	7.3×	7.5×	9×	51×
<i>wf_weighted_edit-distance</i>	1097	43.9	47.1	43.0	223.59
	14.6×	14.6×	13.1×	17.2×	103×

Tabella 4.4: Benchmark per *reads* di 150 bp in *modalità globale*. Nella modalità *wf_weighted-edit-distance*, sono state usate le seguenti penalità: **mismatch=1, insertion=2, deletion=2**.

Modality: <i>Global</i>		Reads number: 10		Reads length: 1000 bp	
	Time (s)				Memory (GB)
Algorithm	Tot	Mean	Max	Min	Max
<i>RecGraph</i>	214	21.4	26.4	20.0	7.84
<i>wf_edit-distance</i>	222	22.2	24.9	20.2	110.32
	1.04×	1.04×	0.94×	1×	14×
<i>wf_weighted_edit-distance</i>	464.0	46.4	46.9	46.0	223.61
	2.2×	2.2×	1.78×	2.3×	28.5×

Tabella 4.5: Benchmark per *reads* di 1000 bp in *modalità globale*. Nella modalità *wf_weighted-edit-distance*, sono state usate le seguenti penalità: **mismatch=1, insertion=2, deletion=2**.

Modality: <i>Global</i>		Reads number: 5		Reads length: 10000 bp	
	Time (s)				Memory (GB)
Algorithm	Tot	Mean	Max	Min	Max
<i>RecGraph</i>	1155	231.1	241.4	216.2	75.64
<i>wf_edit-distance</i>	139.7	28.0	29.0	26.8	111.23
	0.12×	0.12×	0.12×	0.12×	1.47×
<i>wf_weighted_edit-distance</i>	287.0	57.3	56.4	59.6	196.73
	0.25×	0.25×	0.23×	0.28×	2.6×

Tabella 4.6: Benchmark per *reads* di 10000 bp in *modalità globale*. Nella modalità *wf_weighted-edit-distance*, sono state usate le seguenti penalità: **mismatch=1, insertion=2, deletion=2**.

Modality: <i>Global</i>		Reads number: 5		Reads length: 25000 bp	
	Time (s)				Memory (GB)
Algorithm	Tot	Mean	Max	Min	Max
<i>RecGraph</i>	6909	690.9	703.2	678.5	192.45
<i>wf_edit-distance</i>	134.0	13.4	15.1	12.1	64.42
	0.02×	0.02×	0.02×	0.02×	0.33×
<i>wf_weighted_edit-distance</i>	190.0	19.0	20.9	17.7	95.16
	0.03×	0.03×	0.03×	0.03×	0.49×

Tabella 4.7: Benchmark per *reads* di 25000 bp in *modalità globale*. Nella modalità *wf_weighted-edit-distance*, sono state usate le seguenti penalità: **mismatch=1**, **insertion=2**, **deletion=2**.

e una **visualizzazione grafica** dei dati precedentemente riportati (figura 4.4):

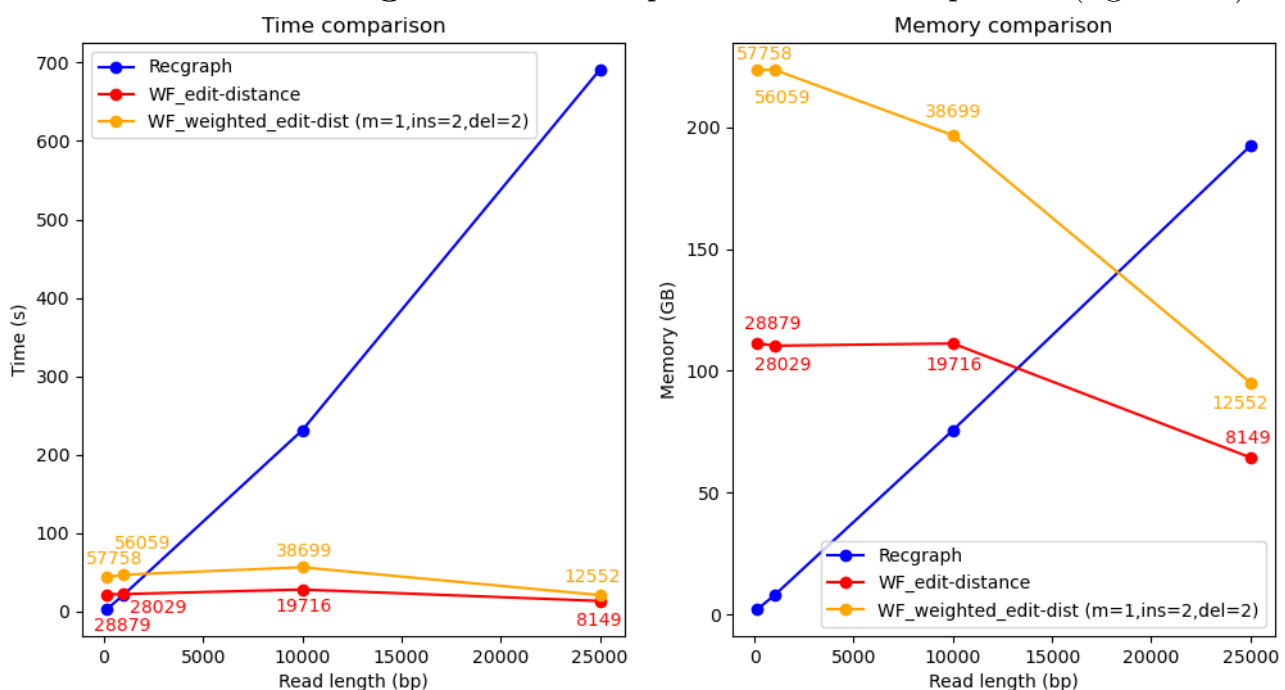


Figura 4.4: Confronto in termini di tempo e spazio per l'allineamento in **modalità globale** tra *RecGraph* e *wf_alignment_edit-distance*. I numeri riportati rappresentano le **distanze di edit** computate da *wf_alignment_edit-distance*.

Nella *modalità globale*, risulta poco sensato l'utilizzo di *wf_edit-distance* quando le sequenze sono molto *più corte* dei cammini del grafo; al contrario, diventa estremamente interessante il suo impiego (sia in termini di **tempo di esecuzione** che di **memoria occupata**) quando la *dimensione delle reads* diventa *paragonabile* a quella dei *cammini*, usando eventualmente anche *penalità non unitarie*.

Modalità semiglobale

Si riportano ora i risultati ottenuti in **modalità semiglobale**, per la quale sono state usate *reads* di lunghezza 150 bp, 1000 bp e 10000 bp; si mostrano per primi nuovamente i valori ottenuti utilizzando **RecGraph**:

Modality: <i>Semiglobal</i>							
	Reads		Time (s)				Memory (GB)
Algorithm	N	L	Tot	Mean	Max	Min	Max
<i>RecGraph</i>	100	150	270	2.7	3.1	2.3	2.67
	25	1000 6.7×	483	19.3 4.6×	22.3 7.2×	18.5 8×	7.84 2.9×
	5	10000 66.7×	1080	216 80×	223 71.9×	208.1 90.5×	75.66 28.3×

Tabella 4.8: Benchmark per *RecGraph* in *modalità semiglobale*.

Come da previsione, i risultati ottenuti sono molto simili a quelli della **modalità globale** (tabella 4.2): infatti, in entrambe le modalità, **RecGraph** computa allo stesso modo l'intera *matrice di programmazione dinamica*, mantenendo quindi la stessa *complessità*: le differenze stanno nelle diverse *condizioni iniziali* e nella selezione della *soluzione ottimale*.

Al contrario, i risultati ottenuti da **wf_edit_distance** cambiano radicalmente:

Modality: <i>Semiglobal</i>								
	Reads		Time (s)				Memory (GB)	Score
Algorithm	N	L	Tot	Mean	Max	Min	Max	Mean
<i>wf_edit-dist</i>	100	150	2.02	0.02	0.2	0.01	0.138	< 5
	25	1000 6.7×	2.869	0.1 5×	0.3 1.5×	0.1 10.0×	0.212 1.34×	22.5 1.5×
	5	10000 66.7×	18.4	3.7 185×	4.4 22.0×	3.2 320×	5.07 36.7×	1051 260×

Tabella 4.9: Benchmark per *wf_edit-distance* in *modalità semiglobale*.

L'**allineamento semiglobale** permette di allineare le sequenze a qualsiasi punto del grafo, ottenendo **distanze di edit** nettamente inferiori che aumentano in maniera proporzionale alla lunghezza delle reads. Come conseguenza, anche **wf_edit-distance** scala in maniera proporzionale alla lunghezza delle sequenze.

Di seguito un confronto dei valori ottenuti da **RecGraph** e da **wf_edit_distance** (sia con penalità unitarie che nel caso pesato) sulle sequenze di interesse:

Modality: <i>Semiglobal</i>		Reads number: <i>100</i>		Reads length: <i>150 bp</i>	
	Time (s)				Memory (GB)
Algorithm	Tot	Mean	Max	Min	Max
<i>RecGraph</i>	270	2.7	3.1	2.3	2.67
<i>wf_edit-distance</i>	2.02 < 0.01×	0.02 < 0.01×	0.2 0.06×	0.01 < 0.01×	0.138 0.05×
<i>wf_weighted_ed</i>	3.27 0.01×	0.024 0.01×	0.2 0.06×	0.01 < 0.01×	0.165 0.06×

Tabella 4.10: Benchmark per *reads* di 150 bp in *modalità semiglobale*. Nella modalità *wf_weighted-edit-distance*, sono state usate le seguenti penalità: **mismatch=1**, **insertion=2**, **deletion=2**.

Modality: <i>Semiglobal</i>		Reads number: 25		Reads length: 1000 bp	
	Time (s)				Memory (GB)
Algorithm	Tot	Mean	Max	Min	Max
<i>RecGraph</i>	483	19.3	22.3	18.5	7.84
<i>wf_edit-distance</i>	2.87 < 0.01×	0.1 < 0.01×	0.3 0.01×	0.1 < 0.01×	0.212 0.03×
<i>wf_weighted_edit-distance</i>	3.22 < 0.01×	0.1 < 0.01×	0.4 0.02×	0.1 < 0.01×	0.351 0.05×

Tabella 4.11: Benchmark per *reads* di 1000 bp in *modalità semiglobale*. Nella modalità *wf_weighted-edit-distance*, sono state usate le seguenti penalità: **mismatch=1**, **insertion=2**, **deletion=2**.

Modality: <i>Semiglobal</i>		Reads number: 5		Reads length: 10000 bp	
	Time (s)				Memory (GB)
Algorithm	Tot	Mean	Max	Min	Max
<i>RecGraph</i>	1080	216	223	208.1	75.66
<i>wf_edit-distance</i>	18.4	3.7	4.4	3.2	5.07
	0.02×	0.02×	0.02×	0.02×	0.08×
<i>wf_weighted_edit-distance</i>	18.8	3.8	4.3	3.4	5.68
	0.02×	0.02×	0.02×	0.02×	0.08×

Tabella 4.12: Benchmark per *reads* di 10000 bp in *modalità semiglobale*. Nella modalità *wf_weighted-edit-distance*, sono state usate le seguenti penalità: **mismatch=1**, **insertion=2**, **deletion=2**.

e un **plotting** dei risultati appena riportati (figura 4.5):

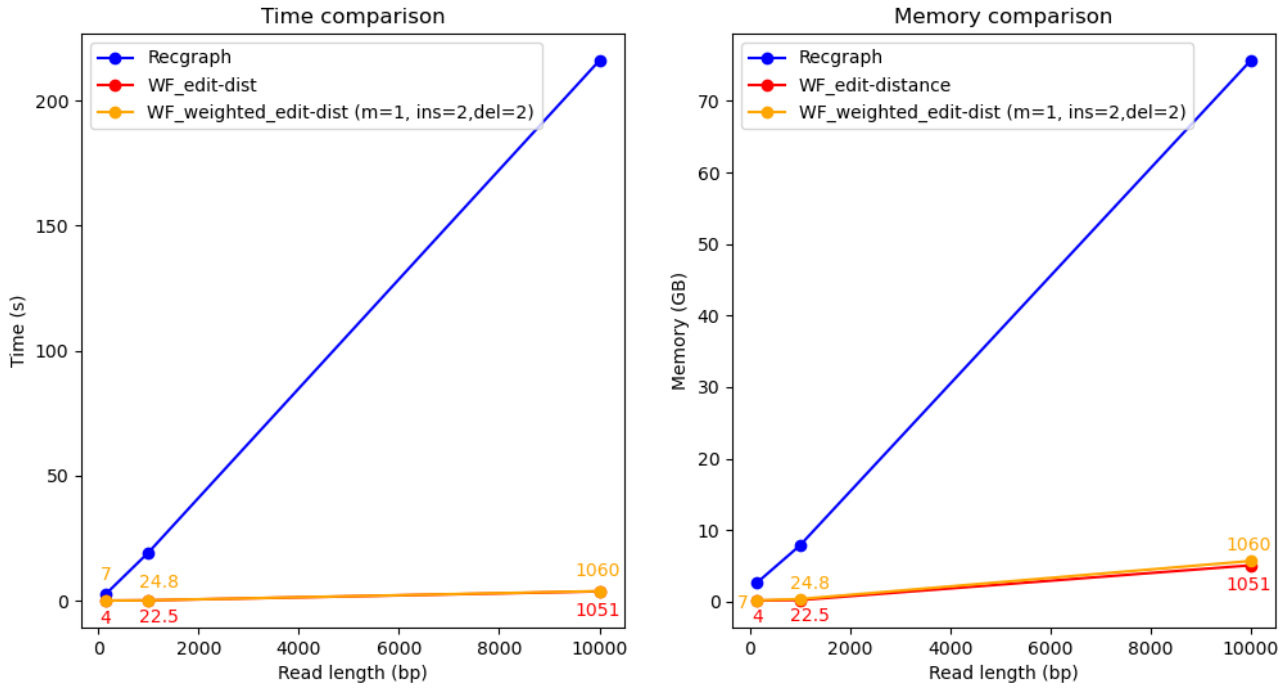


Figura 4.5: Confronto in termini di tempo e spazio per l'allineamento in **modalità semiglobale** tra *RecGraph* e *wf_alignment_edit-distance*. I numeri riportati rappresentano le **distanze di edit** computate da *wf_alignment_edit-distance*.

Nella **modalità semiglobale**, *wf_edit_distance* risulta superiore a *RecGraph* su tutti gli input, sia in **tempo di esecuzione** che in **memoria**.

4.3.2 Analisi dell'andamento del prototipo su singole variabili

In questa seconda parte del benchmark ci si è concentrati in maniera specifica sull'analisi dell'andamento asintotico di *wf_edit_distance*: prese come riferimento n variabili, si è andato a misurare l'andamento di ognuna di esse fissando la restanti $n - 1$ come costanti. In particolare, si sono considerate le seguenti dimensioni:

- *valore dell'allineamento*;
- *numero di threads* lanciati in parallelo;
- *numero di cammini* del grafo di variazione.

Si è utilizzato come input lo stesso grafo utilizzato in precedenza (tabella 4.1) e delle sequenze della stessa lunghezza dei suoi cammini, eseguendo allineamenti in **modalità globale**.

Dove non specificato diversamente, il **cammino ottimale** è stato inserito a metà di tutti i percorsi; inoltre è stato utilizzato un *singolo thread* (tranne ovviamente nella sperimentazione sul numero dei threads). Come implementazione del *fronte d'onda* è stato scelto *wf_vec*.

Si ricorda che la **complessità temporale** attesa dell'algoritmo è

$$T(n, m, p, \bar{d}) = O(\min\{n, m\} \cdot \bar{d} \cdot p) \quad (4.3)$$

mentre la **complessità in spazio** è

$$T(n, m, p, \bar{d}) = O((n + m) \cdot \bar{d} \cdot p) \quad (4.4)$$

a causa dell'implementazione ***wf_vec*** scelta (equazione 4.1).

A seguire i risultati ottenuti.

Numero di threads variabile

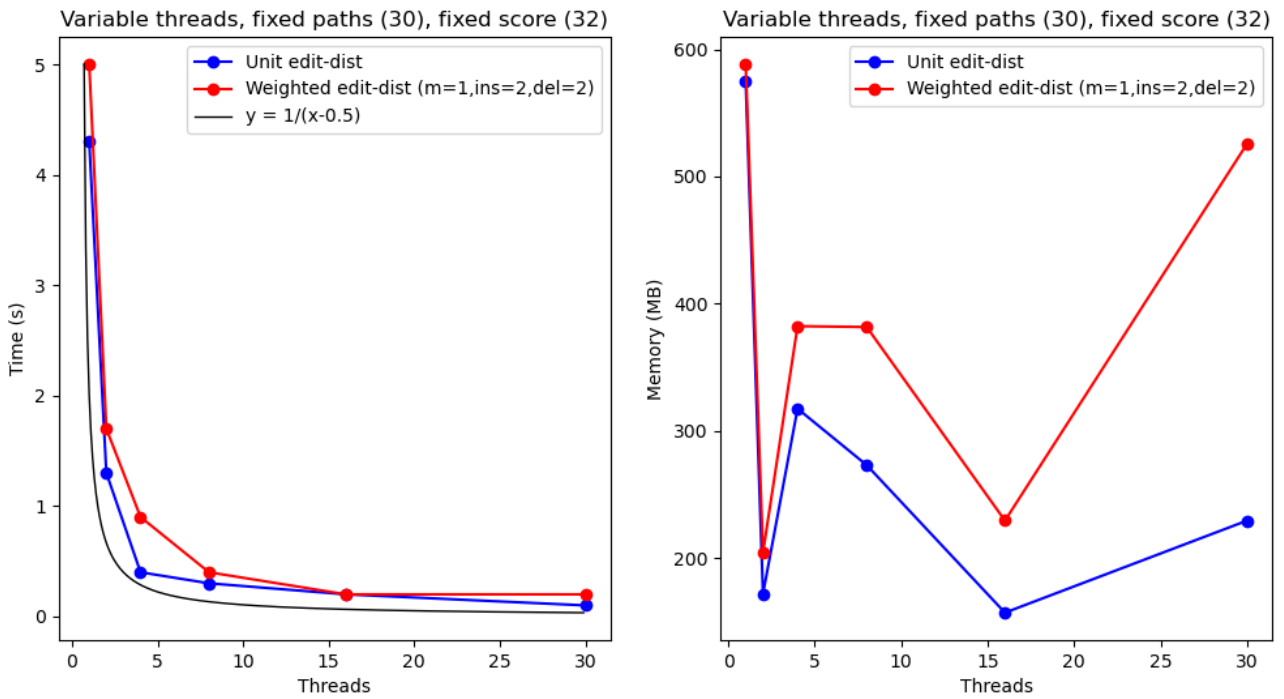


Figura 4.6: Andamento in *tempo* di ***wf_alignment*** rispetto a un *numero di threads* eseguiti in parallelo *variabile*.

Facendo variare il **numero di threads** eseguiti in parallelo si evidenzia un *andamento asintotico temporale* del tipo $y = \frac{1}{x}$, come da previsione. L'andamento in *memoria* risulta molto più imprevedibile, anche in questo caso come conseguenza del troncamento rispetto all'allineamento migliore trovato, spiegato nella sezione 4.2.3.

Punteggio dell'allineamento variabile

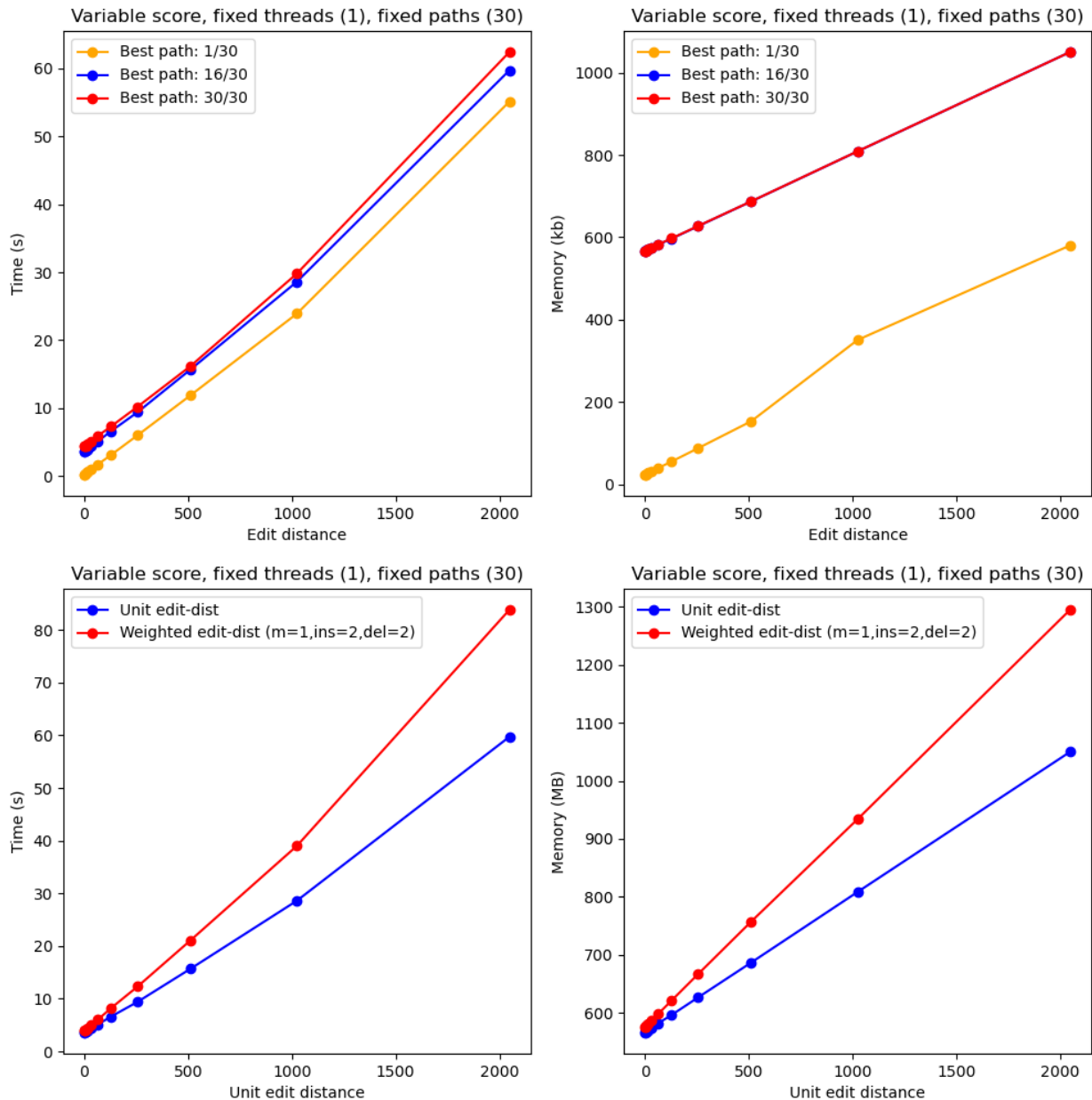


Figura 4.7: Andamento in *tempo* e in *memoria* di *wf_alignment* rispetto a un *punteggio* dell'allineamento *variabile*.

Nella sperimentazione con **distanza di edit variabile** si nota un *andamento lineare* sia in tempo che in memoria, in linea con le aspettative. Spostando la posizione del *percorso ottimale* le rette mantengono lo stesso **coefficiente angolare**, variando nell'**ordinata all'origine** (conseguenza del troncamento rispetto alla soglia trovata, sezione 4.2.3); aumentando le penalità, si nota un aumento dell'inclinazione della retta, conseguenza diretta della *dipendenza lineare* nella **distanza di edit**.

Numero di cammini del grafo variabile

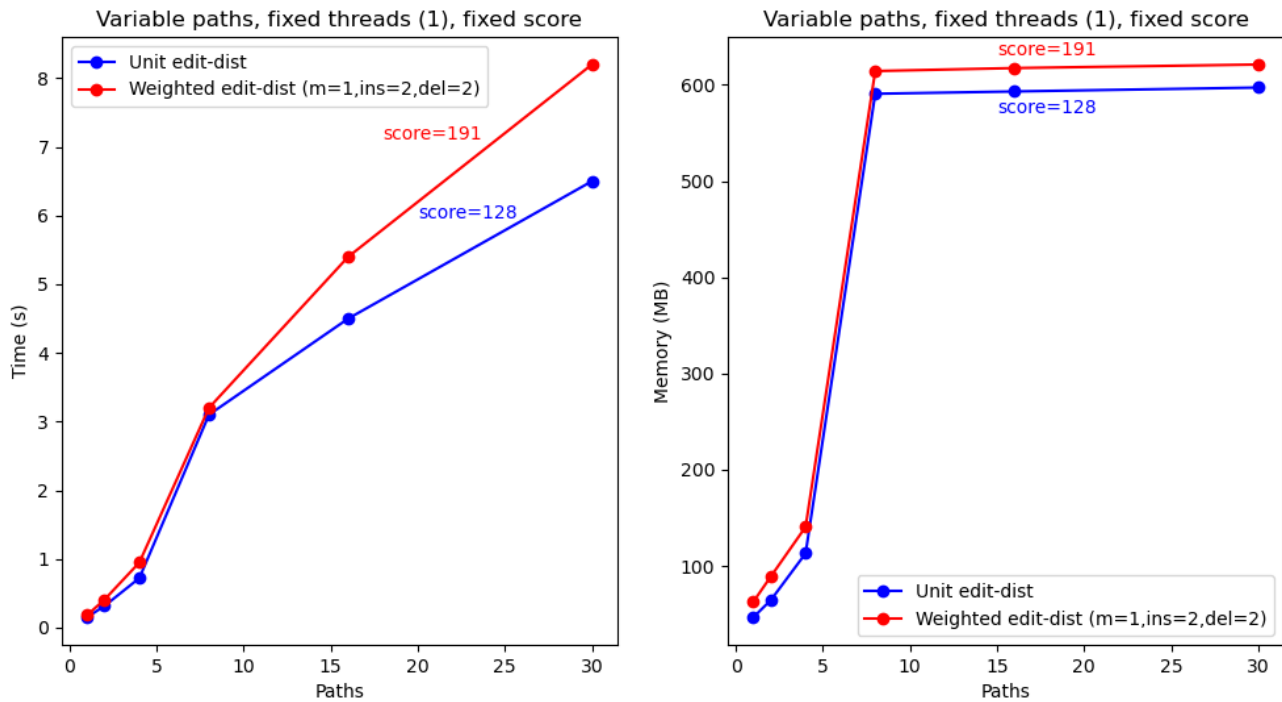


Figura 4.8: Andamento in *tempo* e in *memoria* di *wf_alignment* rispetto a un *numero di percorsi* del grafo *variabili*.

Nell'ultima sperimentazione, che considera il **numero di percorsi del grafo**, si ottengono risultati leggermente meno precisi dei precedenti, a causa dell'alta variabilità dell'input; infatti, il *percorso ottimale* viene sempre inserito a metà dei percorsi disponibili: come conseguenza, i percorsi non ottimali (variabili) computati prima di quello ottimale provocano un andamento non completamente in linea con le equazioni 4.3 e 4.4. Tuttavia, in assoluto è possibile comunque notare una **dipendenza lineare** nel **tempo di esecuzione**, mentre per quanto riguarda la **memoria** i risultati dipendono troppo dagli allineamenti trovati prima di quello ottimale.

Capitolo 5

Conclusioni e sviluppi futuri

Il prototipo sviluppato dimostra a pieno le maggiori potenzialità dell'algoritmo *wavefront* rispetto agli algoritmi di allineamento tradizionali, generando grandi risparmi sia in termini di **tempo** che in termini di **spazio** e permettendo di trattare istanze maggiori di diversi ordini di grandezza. Tuttavia, formalmente esso si basa sul calcolo della **distanza di edit** e non su un vero e proprio **allineamento**, producendo a volte risultati non in linea con quelli degli altri algoritmi; un importante futuro miglioramento è quindi sicuramente quello di estendere l'approccio *wavefront* su **grafi di variazione** e **sequenze** per effettuare *allineamenti con gap*¹, che forniscono risultati più coerenti rispetto a quelli basati sulla **distanza di edit (pesata)**.

Inoltre, nell'articolo [10] viene presentata un'**euristica** di **WFA** che permette di ottenere ottimi risparmi di tempo, al costo di non avere la certezza di trovare la *soluzione ottima*: l'**euristica** si integra molto bene con l'algoritmo, rendendo necessarie relativamente poche modifiche per implementarla all'interno del prototipo.

Infine, risulterebbe estremamente interessante riuscire a trovare un approccio che permetta di utilizzare l'algoritmo *wavefront* su **grafi di variazione canonici** senza dover estrarre tutti i **percorsi** di quest'ultimi: un simile approccio permetterebbe di risparmiare grandi quantità di tempo dal calcolare più volte i valori per vertici appartenenti a diversi cammini (in maniera simile a quello che avviene in *RecGraph*), rendendo tuttavia più complicato le computazioni in parallelo tramite **multithreading**.

In conclusione, l'**algoritmo wavefront** rappresenta un'importante novità nell'ambito della **bionformatica**, che potrà portare a importanti miglioramenti in tutte le diverse applicazioni dell'**allineamento di sequenze**.

¹Una implementazione per i **grafi di sequenza** è già stata fornita nell'articolo [6].

Bibliografia

- [1] Q. Aguado-Puig, S. Marco-Sola, J. C. Moure, D. Castells-Rufas, L. Alvarez, A. Espinosa, and M. Moreto. Accelerating edit-distance sequence alignment on gpu using the wavefront algorithm. *IEEE Access*, 10:63782–63796, 2022.
- [2] J. Avila, P. Bonizzoni, S. Ciccolella, G. D. Vedova, L. Denti, D. Monti, Y. Pirola, and F. Porto. Recgraph: adding recombinations to sequence-to-graph alignments. *bioRxiv*, 2022.
- [3] J. M. Eizenga and B. J. Paten. Improving the time and space complexity of the wfa algorithm and generalizing its scoring. *bioRxiv*, 2022.
- [4] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge, 1997.
- [5] C. Lee, C. Grasso, and M. F. Sharlow. Multiple sequence alignment using partial order graphs . *Bioinformatics*, 18(3):452–464, 03 2002.
- [6] S. Marco-Sola, J. C. Moure, M. Moreto, and A. Espinosa. Fast gap-affine pairwise alignment using the wavefront algorithm. *Bioinformatics*, 37(4):456–463, 09 2020.
- [7] N. M. Mwaniki, E. Garrison, and N. Pisanti. Fast exact string to d-texts alignments, 2022.
- [8] S. Needleman and C. Wunsch. A general method applicable to the search of similarities in the amino-acid sequence of two proteins. *JMB*, 48:443–453, 1970.
- [9] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1):100–118, 1985. International Conference on Foundations of Computation Theory.
- [10] H. Zhang, S. Wu, S. Aluru, and H. Li. Fast sequence to graph alignment using the graph wavefront algorithm, 2022.

*Un profondo ringraziamento a tutte le persone che hanno
contribuito al completamento di questo percorso accademico:
senza il loro sostegno, non avrei potuto raggiungere questo
importante traguardo.*

*Al Prof. Della Vedova e alla Prof. Bonizzoni,
per avermi introdotto alla disciplina della bioinformatica e per
avermi guidato nella stesura di questa tesi.*

*Ai miei genitori, a mia sorella e a tutta la mia famiglia,
per avermi supportato e incoraggiato per tutta la durata di
questi tre anni.*

*A tutti i miei amici, per avermi sostenuto durante questo
viaggio e per l'essere stati fondamentali nel raggiungimento di
questo traguardo.*

*Ai miei compagni di corso, per aver condiviso con me tutte le
sfide affrontate insieme nell'arco di questi tre anni.*