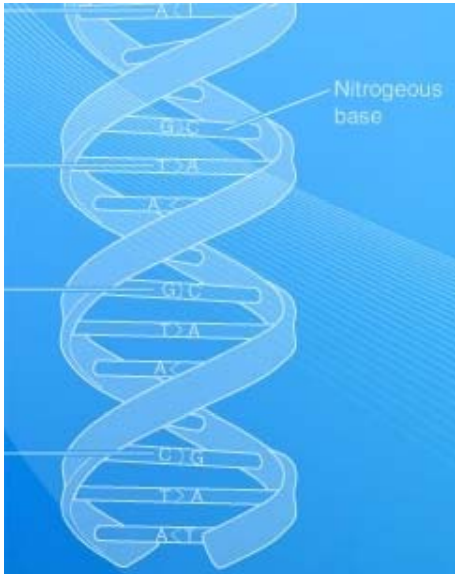
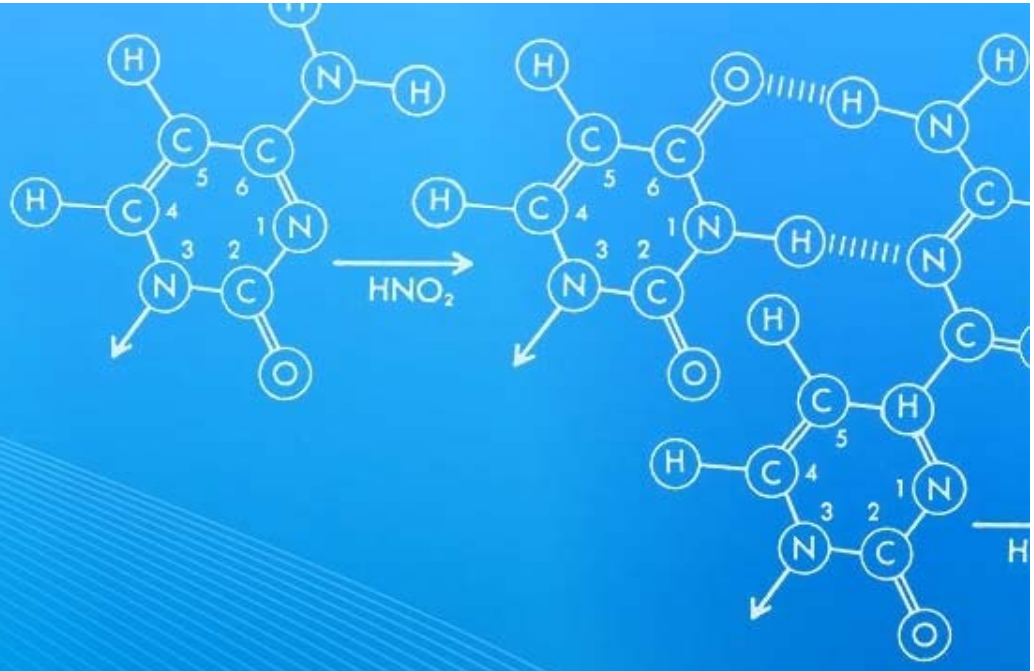
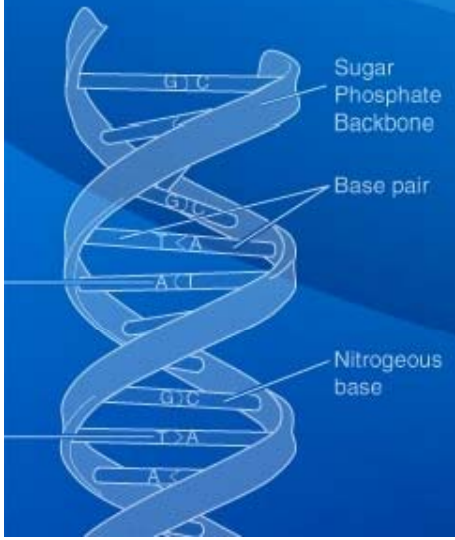




Consecutive Suffix Alignment



Cocco Gabriele

Bioinformatica - A.A. 2008-2009

Indice

- Introduzione
  - Presentazione del problema
  - Analogie con problemi noti
  - Stato dell'arte
- Soluzione standard
- Primo algoritmo
  - Premesse e definizioni
  - Idee generali
  - Implementazione ed analisi della complessità
- Secondo algoritmo
  - Premesse e definizioni
  - Idee generali
  - Implementazione ed analisi della complessità
- Conclusioni

Introduzione

Presentazione del problema

- **Problema noto**: date due stringhe A e B determinarne il (un) miglior allineamento globale secondo una certa metrica (es. edit distance) per stabilirne la similarità
- **Variante**: date due stringhe A e B determinare, per ogni suffisso C di A, il miglior allineamento globale fra C e B

- Esempio: A = autore, B = tre.

- **tre** vs **e**
- …
- **tre** vs **tore**
- **tre** vs **utore**
- **tre** vs **autore**

- **Utilizzi**: *Cyclic String Comparison (Approximate Overlap for DNA Sequencing*)

Introduzione

Presentazione del problema

- **Longest Common Subsequence** (LCS(A, B)): utilizzata come metrica per l'allineamento.

Gap penality = 0

Match = +1

		C	T	G	C
	0	0	0	0	0
C	0	1	1	1	1
G	0	1	1	2	2
T	0	1	2	2	2

LCS(A,B) = CG

Miglior allineamento:

C T G – C

C – G T –

Introduzione

- **Consecutive Suffix Alignment**: date due stringhe A (lunghezza m) e B (lunghezza n), determinare, per ogni suffisso C di A, il miglior allineamento globale fra C e B secondo una certa metrica

- **Con metrica LCS**: date due stringhe A e B, determinare, per ogni suffisso C di A, LCS(C, B)

- **Stato dell'arte:**

- *Landau, Myers (1998):* algoritmo con complessità O(nk) se il numero di differenze nell'allineamento è limitato da un parametro k
- *Kim, Park (2000):* algoritmo in grado di calcolare allineamento fra B e suffissi/estensioni di A una volta allineate A e B. Complessità $O(n^2 + nm)$

Soluzione standard

- **Soluzione standard**: calcolare LCS fra ogni prefisso di A e B "ripartendo da 0"

		C
C		1
G		1
T		1

→

		G	C
C		0	1
G		1	1
T		1	1

→

		T	G	C
C		0	0	1
G		0	1	1
T		1	1	1

- **Costo**

  O(mn) per allineare B (n) con A (m)

  ↓

  O(n) + O(2n) + O(3n) + …. O(mn) = O(m^2 n) = **O(n^3)** se m = O(n)

Primo algoritmo

Premesse: DP

- DP^k: matrice di programmazione dinamica per allineare A_k^m e B

  $DP^k[i,j] = |LCS(A_k^j, B_1^i)|$

	T	G	C
C	0	0	1
G	0	1	1
T	1	1	1

- **Valori monotoni** non decrescenti verso destra e verso il basso:

  Per $i' \geq i$ e $j' \geq j$: $DP[i', j'] \geq DP[i, j]$

- **Unit steps**

  $DP[i + 1, j] \neq DP[i, j] \rightarrow DP[i + 1, j] = DP[i, j] + 1$

  $DP[i, j + 1] \neq DP[i, j] \rightarrow DP[i, j + 1] = DP[i, j] + 1$

Primo algoritmo

Premesse: partition point

- **Partition point**: elemento di una colonna in cui si ha un salto di valore rispetto all'elemento sovrastante

	T	G	T
C	0	0	0
G	0	1	1
T	1	1	2

- **P^k**: rappresentazione di DP^k per partition points (partition encoding)

	m-2	m-1	m
1	3	2	2
2	NULL	NULL	3
3	NULL	NULL	NULL

m-2	m-1	m
3	2	2
		3

Primo algoritmo

Premesse: generazione

- G^k (**Generazione k**): struttura dati (DP^k, P^k) per all'allineamento di B e A_k^m

  "Ci troviamo alla generazione k": stiamo risolvendo il problema di allineare B con il suffisso A_k^m

Primo algoritmo

- Corrispondenza DP - P

	T
C	0
G	0
T	1

	G	T
C	0	?
G	1	?
T	1	?

Come aggiornare le colonne?

If (A[j] = B[i])
then DP[i-1 ,j-1] + 1
else
max(DP[i-1, j], DP[i, j-1])

m
3

m-1	m
2	?
	?

Come aggiornare le colonne?
Aggiungere partition points.

Esiste una regola anche per questo

Primo algoritmo

- Esempio: da P^2 a P^1

	G	G
C	0	0
G	1	1
T	1	1

→

	C	G	G
C	1	1	1
G	1	2	2
T	1	2	2

DP^k

	m-1	m
1	2	2
2	NULL	NULL
3	NULL	NULL

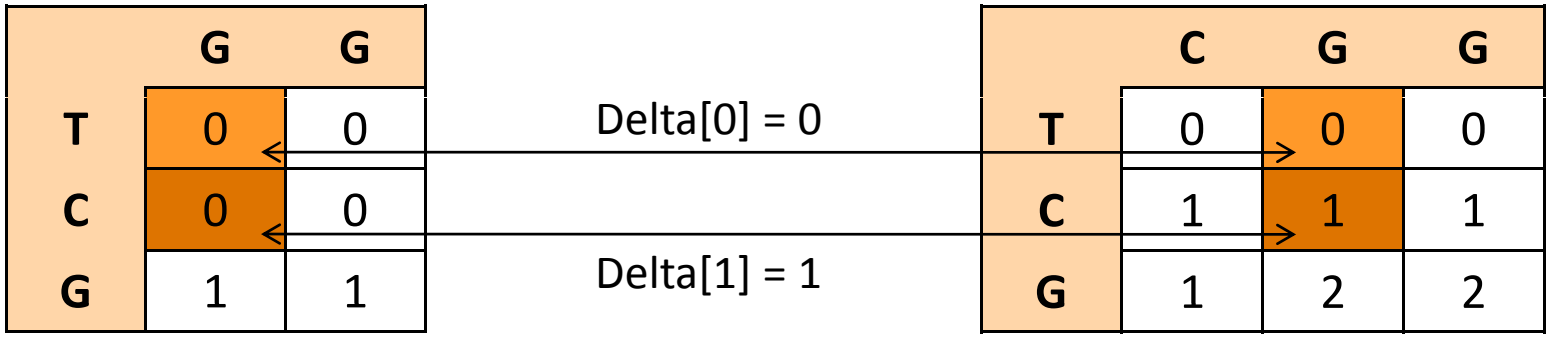
→

	m-2	m-1	m
1	1	1	1
2	NULL	2	2
3	NULL	NULL	NULL

P^k

Primo algoritmo

- Come cambia una colonna j da DP^{k+1} a DP^{k}?
- **Tutti gli elementi con riga ≥ I incrementati di 1**



- **Delta[i]** = $DP^{k}[i, j] - DP^{k+1}[i, j]$

Primo algoritmo

- Colonna j di DP^{k+1} = Colonna j di DP^{k} a parte il fatto che tutti gli elementi con riga **≥** I sono incrementati di 1
- Come si riflette in P^{k}?
- **Colonna j di P^{k} = Colonna j di P^{k+1} con, al più, un partition point aggiuntivo**: il più piccolo indice I tale che $DP^{k}[I, j] – DP^{k+1}[I, j] = delta[I] = 1$

	G	G
T	0	0
C	0	0
G	1	1

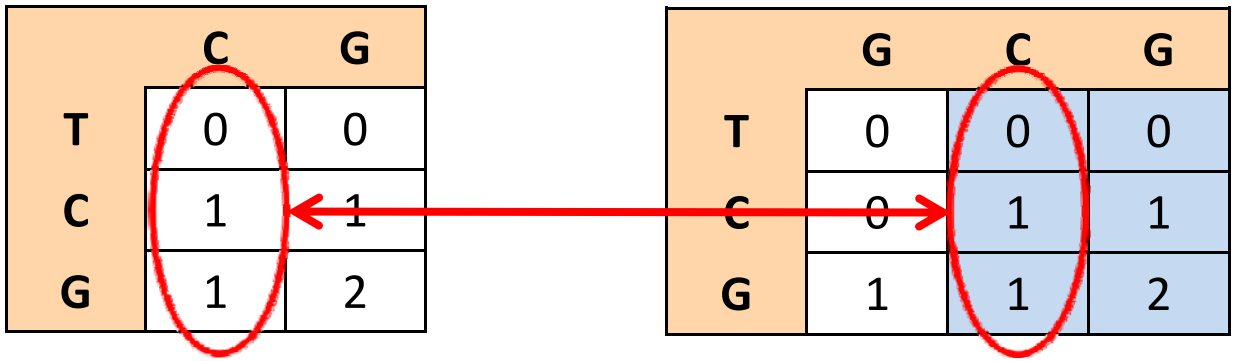
	C	G	G
T	0	0	0
C	1	1	1
G	1	2	2

	m-1	m
1	3	3
2	NULL	NULL
3	NULL	NULL

	m-2	m-1	m
1	2	2	2
2	NULL	3	3
3	NULL	NULL	NULL

Primo algoritmo

- Ogni elemento di DP^k calcolato sulla base della colonna a sinistra e della riga sopra: **se una colonna non cambia da DP^{k+1} a DP^k neanche le successive cambiano**
- Come si riflette in P^k?
- Se colonna j non cambia (non inserisco partition point) allora anche le colonne j' con j' > j non cambiano: **l'iterazione si conclude**



Primo algoritmo

- Più efficiente calcolare P^k di DP^k ?
- Si, perché:
  - Si utilizza P^{k+1}
  - Per ogni colonna attraversata, **l'unica operazione** che, al più, si deve compiere **è l'aggiunta di un partition point**
  - Se una colonna non viene aggiornata l'algoritmo termina l'iterazione

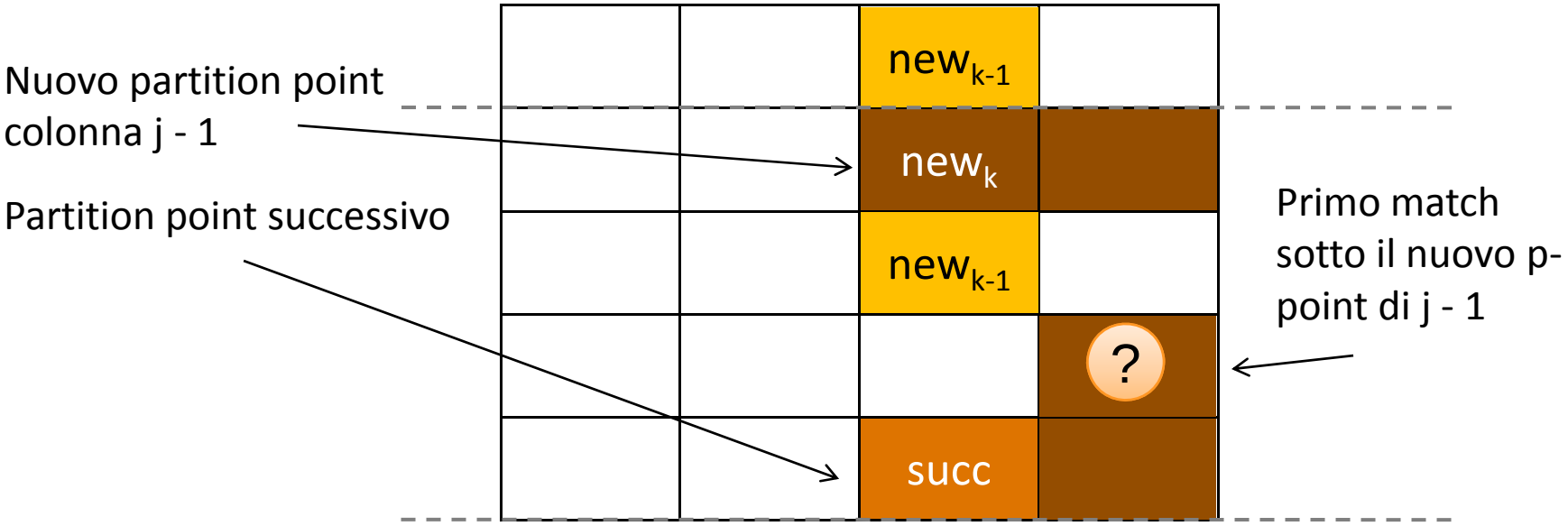
Primo algoritmo

- **Preprocessing + m iterazioni**
- **All'iterazione k si aggiunge la colonna k** (si passa dal suffisso A^m_{k+1} al suffisso A^m_k)
- **P^k viene calcolata partendo da P^{k+1}**
  - Viene aggiunto un partition point alla colonna k (il primo match point)
  - Si attraversano le altre colonne da sinistra a destra e si aggiornano i partition points
- Ad ogni iterazione si è calcolato l'allineamento di B con un suffisso di A

Primo algoritmo

Idee generali: algoritmo

- Problema: **calcolare dove aggiungere partition point** per ogni colonna j attraversata
- Possiamo restringere il range:
  - $l_j = l_{j-1}$
  - $l_j = \min \{ Succ(l_{j-1}), NextMatch(l_{j-1}, j) \}$



Primo algoritmo

Implementazione

- **NextMatch(i, j):** presa una colonna j, l'indice di riga del primo match point sotto la riga i
- **NextMatch table** costruita in $O(n|\sum|)$ ed utilizzata per trovare il punto di inserzione partition point in O(1)
- Inserzioni successive in O(1)
- **Costo: O(nL)**
  - Preprocessing: **$O(n|\sum|)$**
  - Numero partition points in $DP^1 \leq \mathbf{nL}$, $L = |LCS(A, B)| < n$
  - Durante l'intero algoritmo inserisco al più **nL** partition points, ognuno dei quali richiede O(1)

Secondo algoritmo

Premesse: chain

- Allineamento ottimo fra A e B

		G	T	C	C	G	T
		0	1	1	1	1	1
T		0	1	2	2	2	2
C		1	1	2	2	3	3
G		1	1	2	3	3	3
C							

- **La chain** di match points **di lunghezza massima**
- Chain: sequenza ordinata di match points tale che, per ogni coppia (x, y) e (x', y'):
  - x' > x e y' > y

  oppure
  - x > x' e y > y'

Secondo algoritmo

Premesse: tail

- Consideriamo i partition points per righe

	G	C	G
T	0	0	0
C	0	1	1
G	1	1	2

Numero di partition points $= L^k = |LCS(A_k^m, B)|$

- Ultima riga: n-esimo partition point è uguale all'indice j più piccolo tale che $|LCS(A_k^j, B)| = n$
- **Ultimo partition point in DP^k** $= |LCS(A_k^m, B)|$ (**quello che cerchiamo**)
- **Tail generazione k**: indice partition point dell'ultima riga di DP^k
- **TAILS**: struttura dati per memorizzare i tail delle varie generazioni
  - TAILS[k, j] = indice j-esimo tail di DP^k

Secondo algoritmo

Premesse: tail

- Se la colonna k di DP contiene almeno un match point:
  - TAILS[k, 1] = k
  - Tutte le altre entry di TAILS nella riga k ereditate dalla k + 1 tranne, al più, una.
    Una tail non ereditata sse $|LCS(A^m_k, B)| = |LCS(A^m_{k+1}, B)|$

	C	G
T	0	0
C	1	1
G	~~1~~	2

	G	C	G
T	0	0	0
C	0	1	1
G	1	~~1~~	2

TAILS[k, 1] = k

Secondo algoritmo

Idee generali

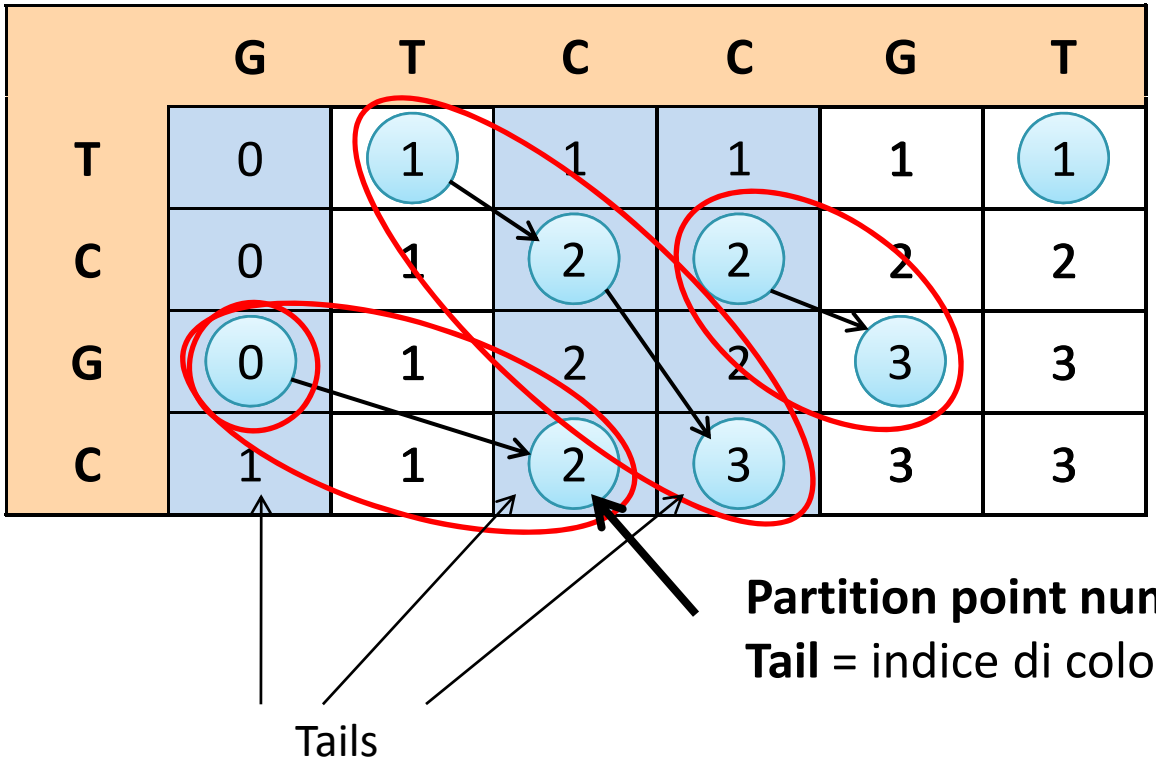
- Come nel primo algoritmo, **possiamo usufruire di quanto calcolato** nella generazione precedente
- Prima: fra generazioni successive una colonna acquisisce un partition point
- Ora: fra generazioni successive l'ultima riga perde un partition point (un tail)

- Come trovare la tail che scompare (se esiste)?
- **Ragionamento sull'evoluzione delle chain** fra generazioni successive

Secondo algoritmo

Idee generali: chain & tail

- Ultima riga: n-esimo partition point è uguale all'indice j più piccolo tale che $|LCS(A_k^j, B)| = n$
- Tail = indice colonna di un partition point



Partition point numero 2 -> valore 2
Tail = indice di colonna = **3**

Secondo algoritmo

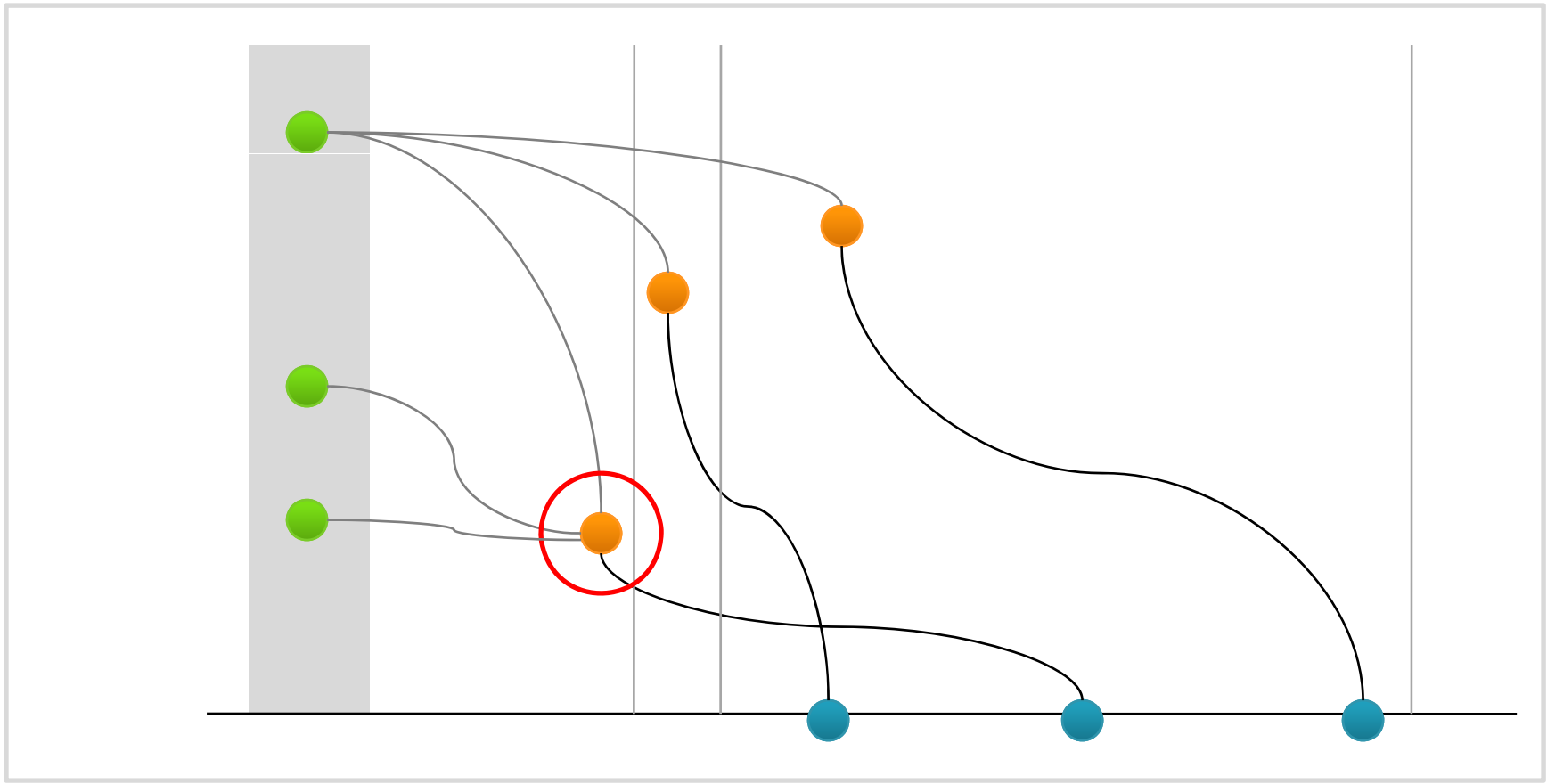
Idee generali: leftmost

- Ci interessano tutte le chains? Sono molte…
- TAILS[k, j] = t se t è l'indice della **colonna più a sinistra** che segna la fine di una chain lunga j
- **Leftmost-ending chains**: le catene che terminano prima (a parità di lunghezza)
- Ad ogni iterazione compaiono nuovi match: le leftmost chains cambiano
- **Tenere traccia delle chain più promettenti per diventare leftmost** nell'iterazione successiva

Secondo algoritmo

Idee generali: best chains

- Più promettenti: **iniziano più in basso e terminano prima**

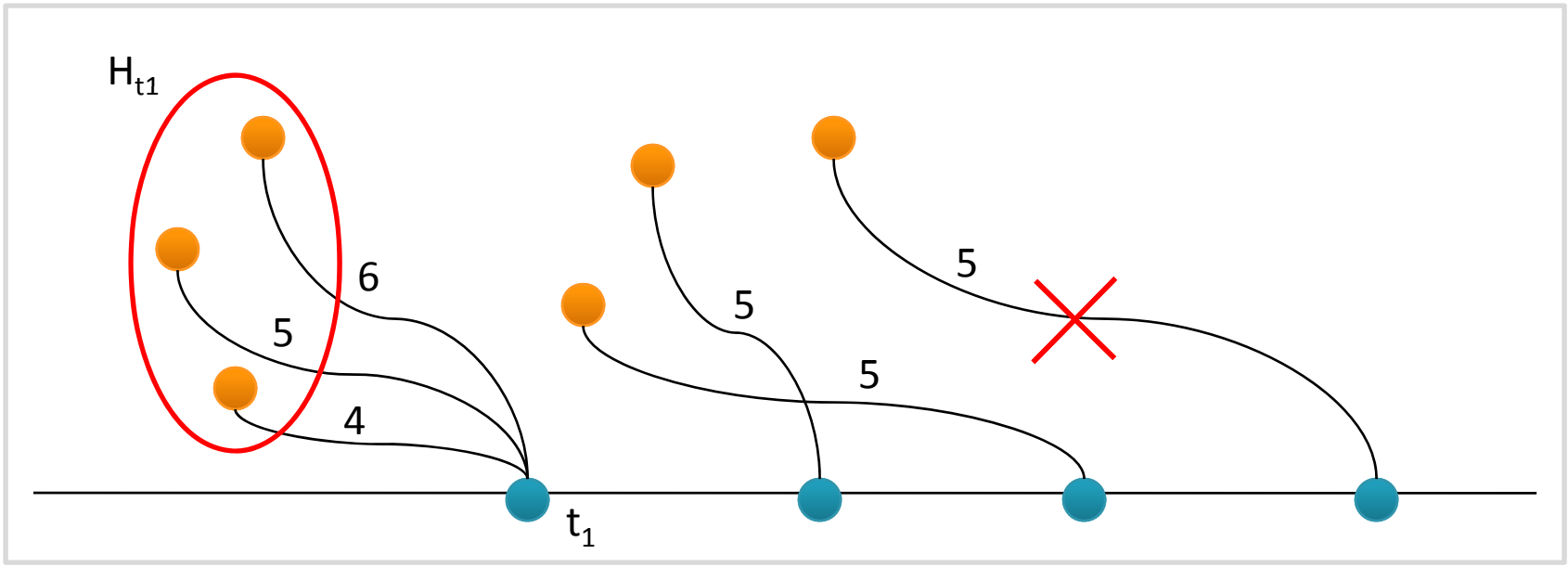


Secondo algoritmo

Idee generali: active chains

- **Active chain** lunga j: catena tale che non ne esiste alcun altra lunga j che inizia più in basso e termina prima
- **head/endpoint**: primo/ultimo match point di un'active chain

- H_t: insieme di head che terminano nella tail t
- Per ogni tail t: chain più lunghe corrispondono a head più alte



Secondo algoritmo

Idee generali: pairs

- **Come cambiano le active chains fra generazioni?**
  H_t può perdere al più una head (per ogni tail t), ed **è sempre la head della chain più corta ad essere rimossa**
- Se una tail ha una sola chain e questa viene rimossa? **E' la tail che non viene ereditata della nuova generazione**
- Per scoprire questa tail basta **tenere le active chain più corte, una per ogni tail**

- **Accoppiamento head/tail**: per ogni tail t ci teniamo la head h a capo della chain più corta (che termina in t) : (h,t) è denominata coppia head-to-tail
- Per ogni generazione k al più L^k tail -> **$O(L^k)$ coppie**
- **$PAIRS^k$**: l'insieme delle coppie (h, t) ordinata per indice descrescente di riga (di h)

Secondo algoritmo

Idee generali

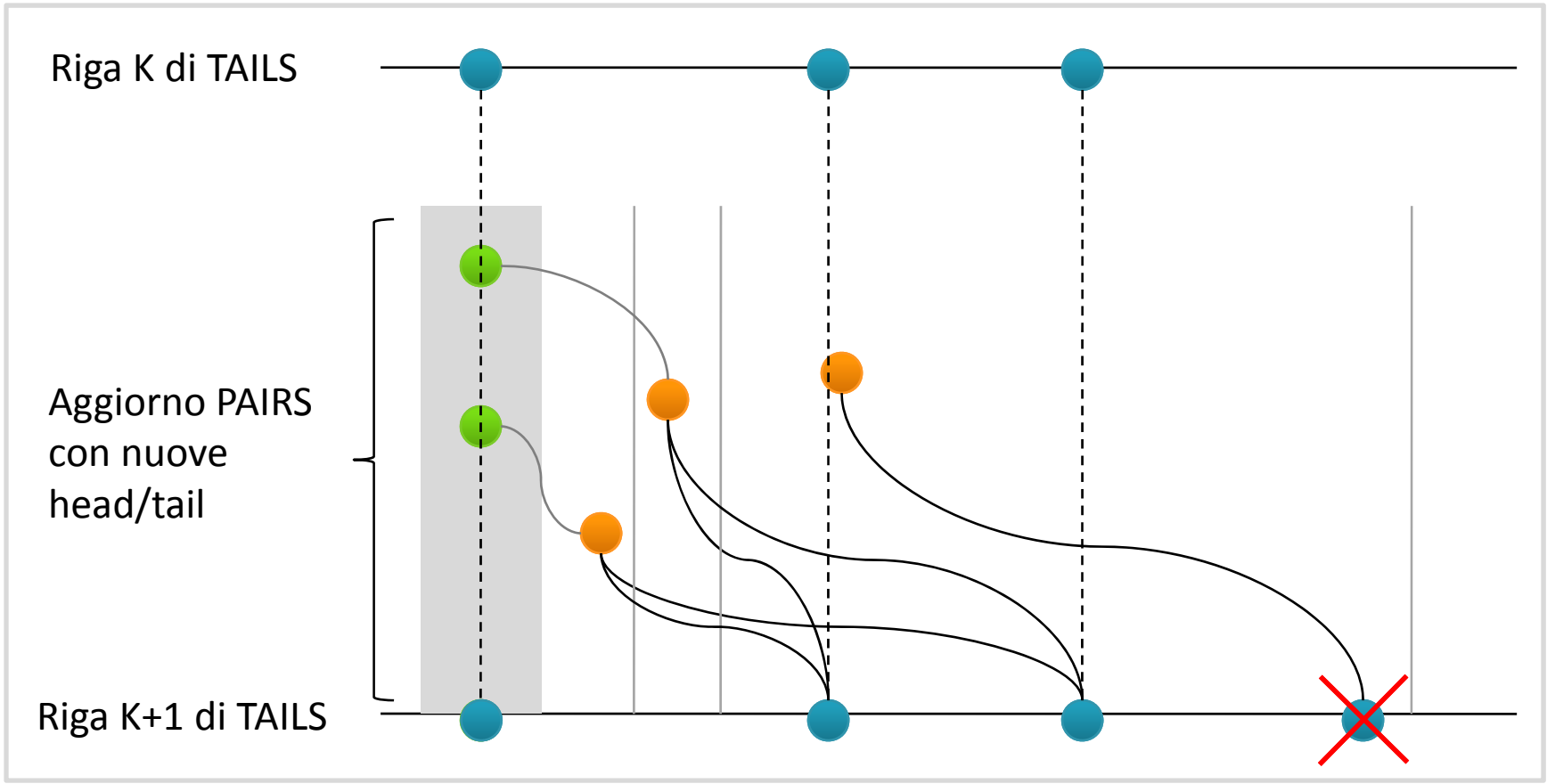
- **Problema:**
  - Supponiamo che la chain più corta di t venga rimossa, se non ci teniamo le altre come facciamo a sapere che ne aveva solo una?
  - Vedremo che **l'avere una sola chain corrisponde al non poter accoppiare la tail con un nuova head**

- **Riduzione del problema:**
  - Chains -> Leftmost chains -> Active chains -> Pairs
  - Trovare tail che scompare ed aggiornare PAIRS per la generazione successiva: **si fa tutto aggiornando PAIRS scorrendola per head decrescente**

Secondo algoritmo

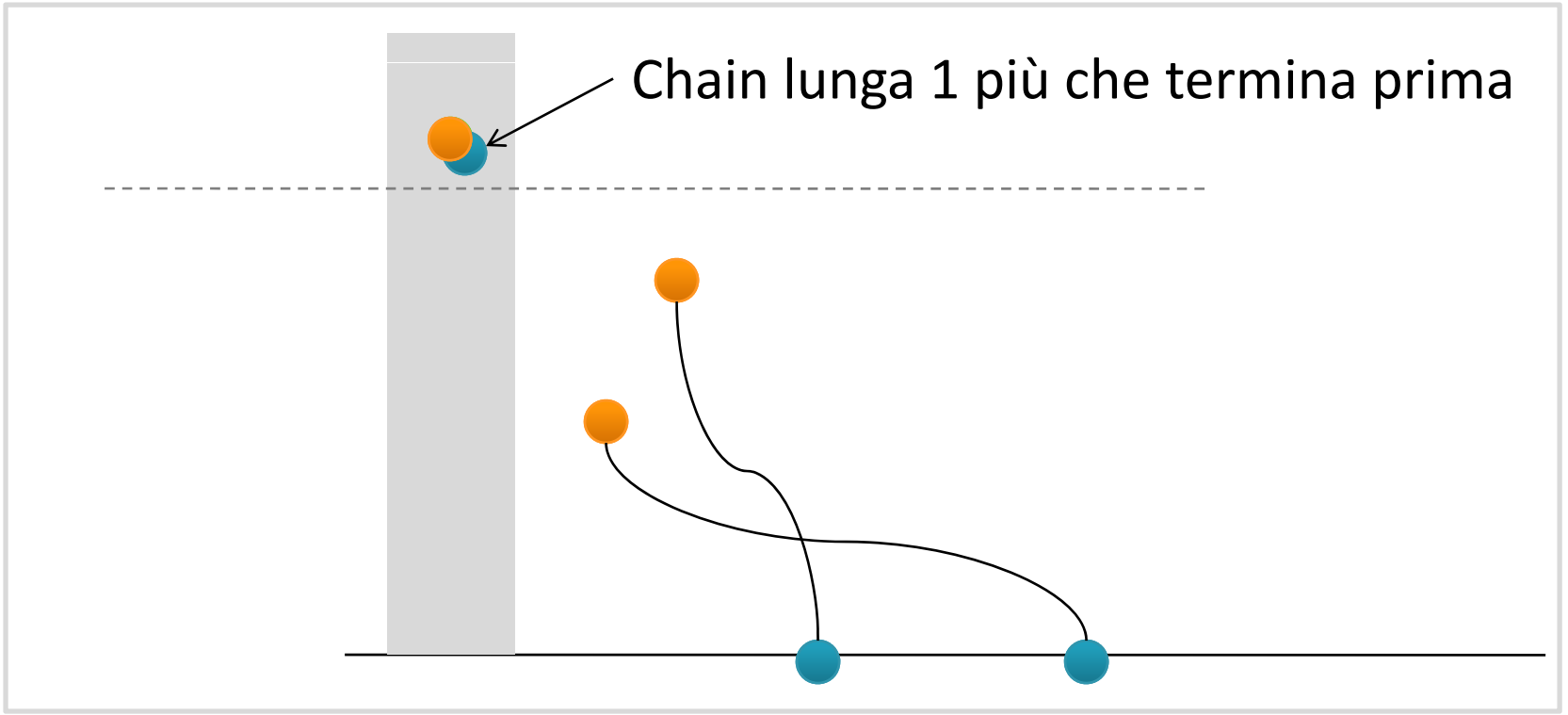
Idee generali: pairs

- **PAIRS^k**: utilizzata per scovare la tail da rimuovere



Secondo algoritmo

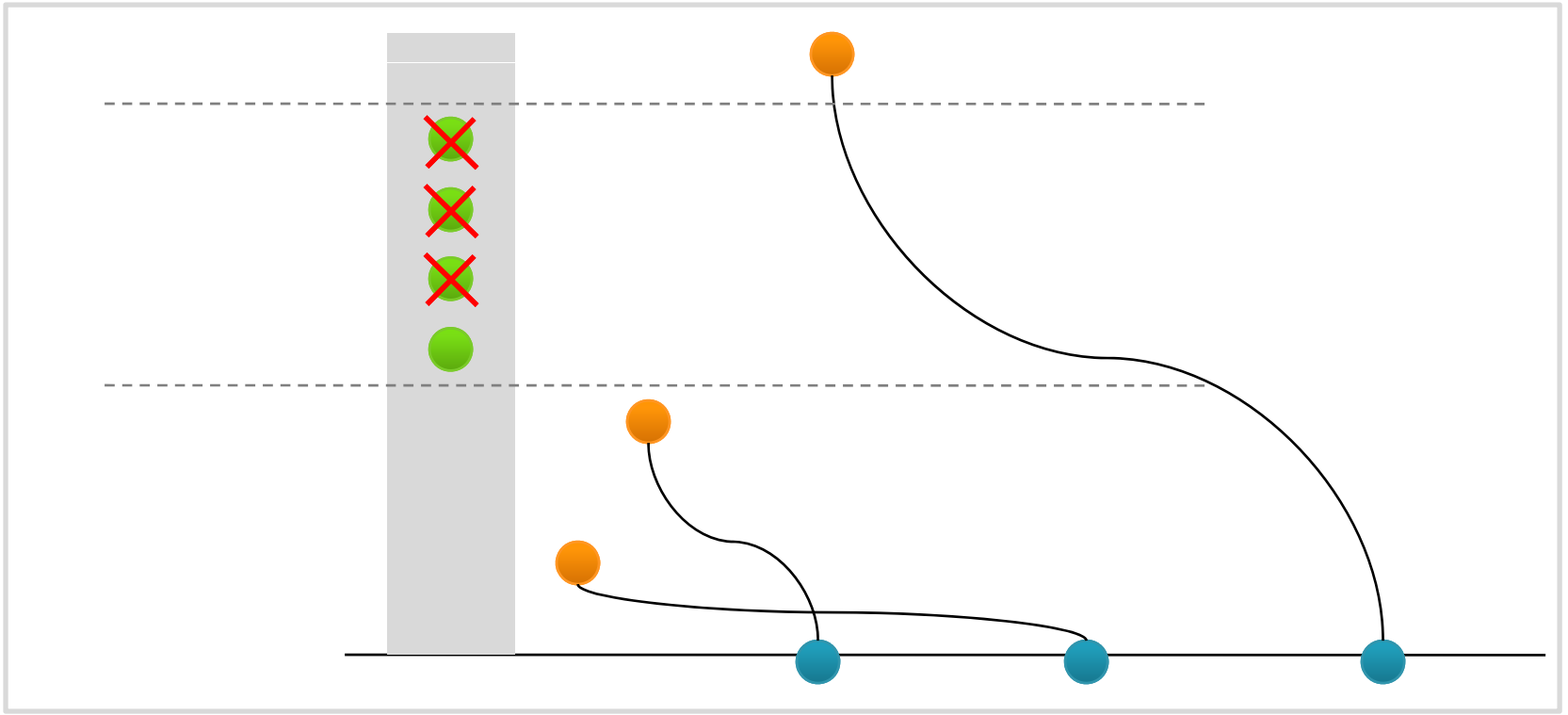
- Aggiornamento di $PAIRS^{k+1}$
  - Caso 1: head sotto l'ultimo match point



Secondo algoritmo

Idee generali: pairs

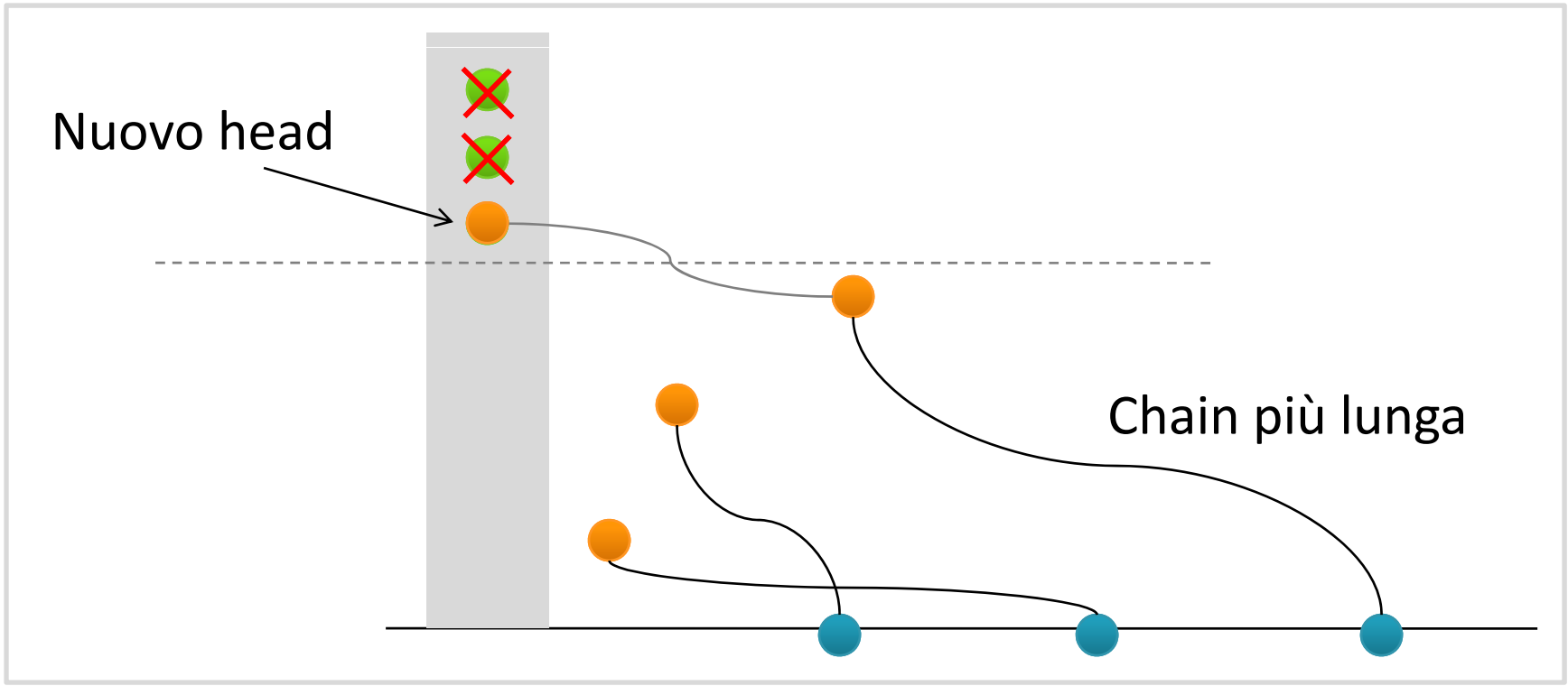
- Aggiornamento di $PAIRS^{k+1}$
  - Caso 2: sequenza di match points senza head nel mezzo



Secondo algoritmo

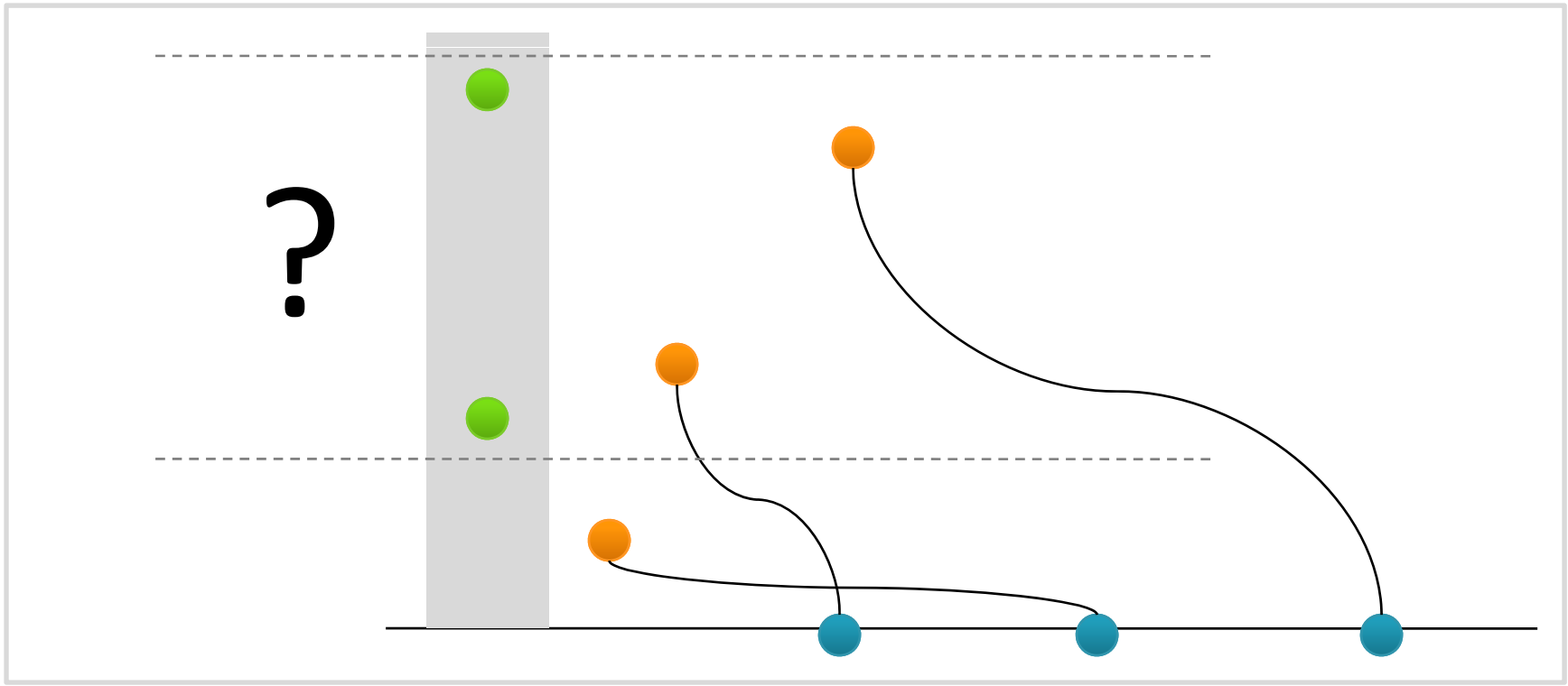
Idee generali: pairs

- Aggiornamento di $PAIRS^{k+1}$
  - Caso 3: match points sopra la head più alta (con indice di riga più basso)



Secondo algoritmo

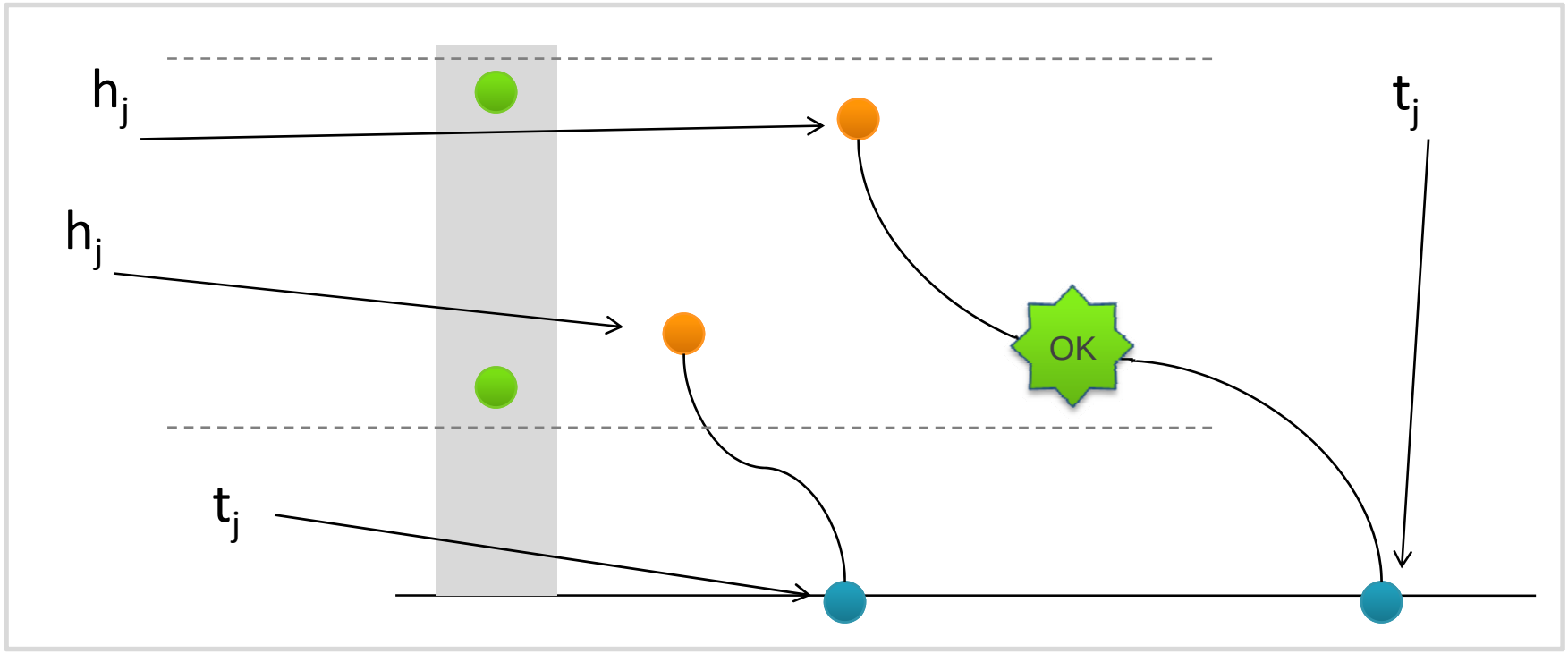
- Aggiornamento di $PAIRS^{k+1}$
  - Caso 4: sequenza di head fra due match points consecutivi



Secondo algoritmo

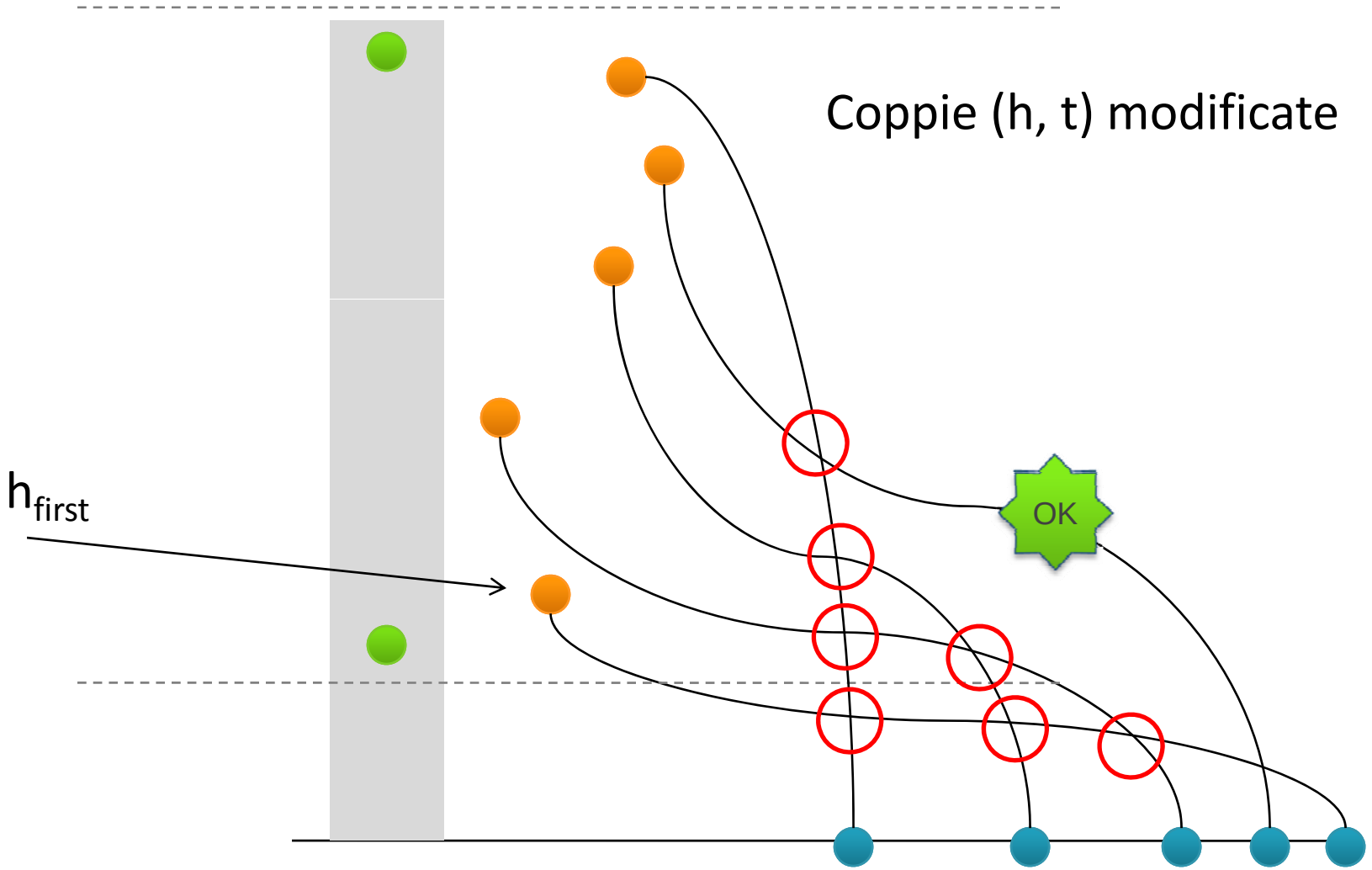
Idee generali: pairs

- Date due coppie (h_i, t_i) e (h_j, t_j), se non si intersecano allora (h_j, t_j) non viene modificata nella generazione k
- **Le coppie modificate formano una serie di chain che si incrociano** l'una con l'altra



Secondo algoritmo

Idee generali: updated



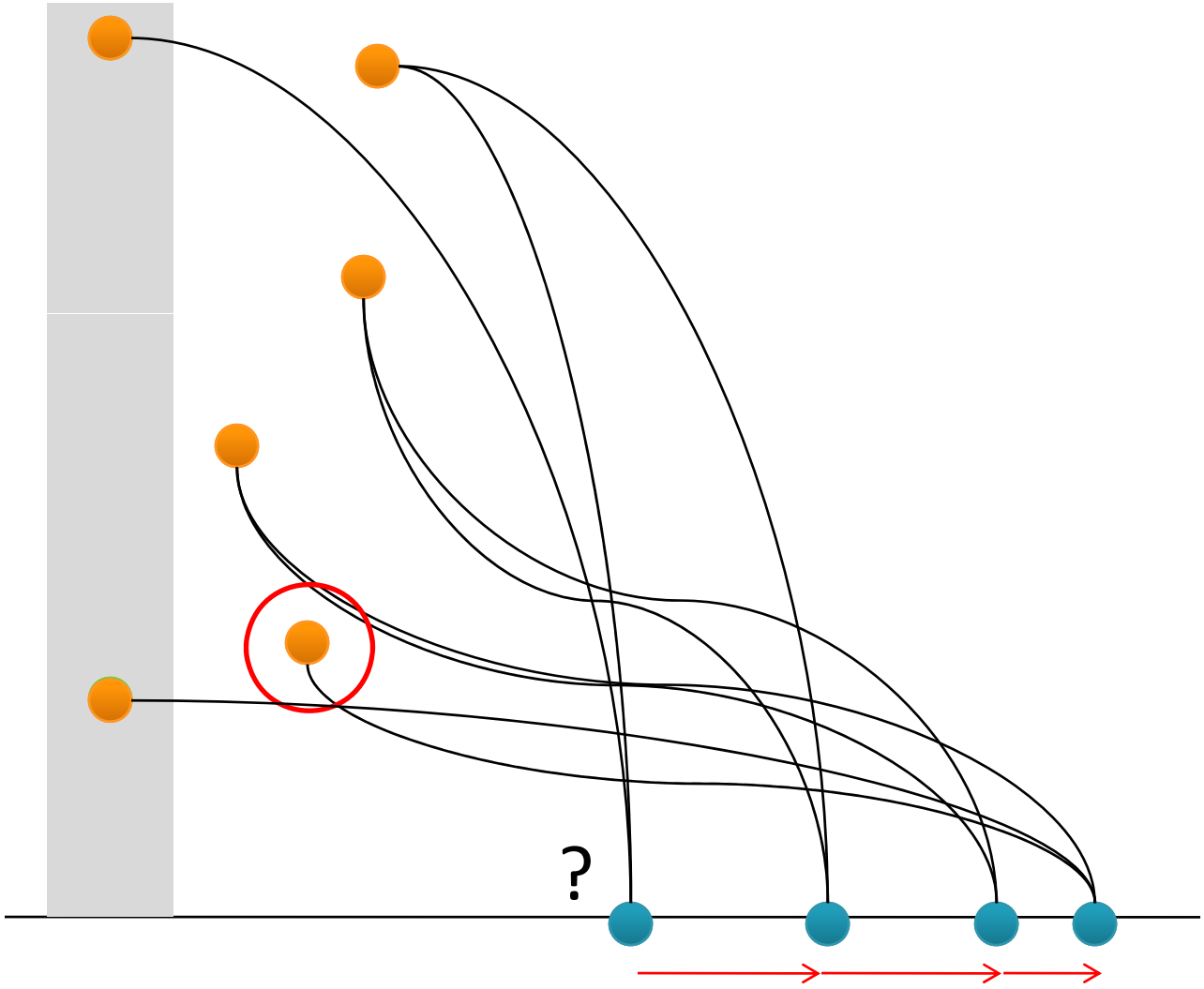
Secondo algoritmo

Idee generali: updated

- Le coppie modificate formano una serie di chain che si intersecano l'una con l'altra
- Come vengono modificate queste coppie da una generazione all'altra?
- Se **(h_i, t_i) e (h_{i+1}, t_{i+1})** sono coppie consecutive modificate, nella generazione k **sostituite con (h_{i+1}, t_i)**
- Problema: $\mathbf{h_{first}}$ e $\mathbf{t_{last}}$?
- **H_{first} sostituito dal match point più basso**
- **T_{last} accoppiata al match point più alto**

Secondo algoritmo

Idee generali: updated



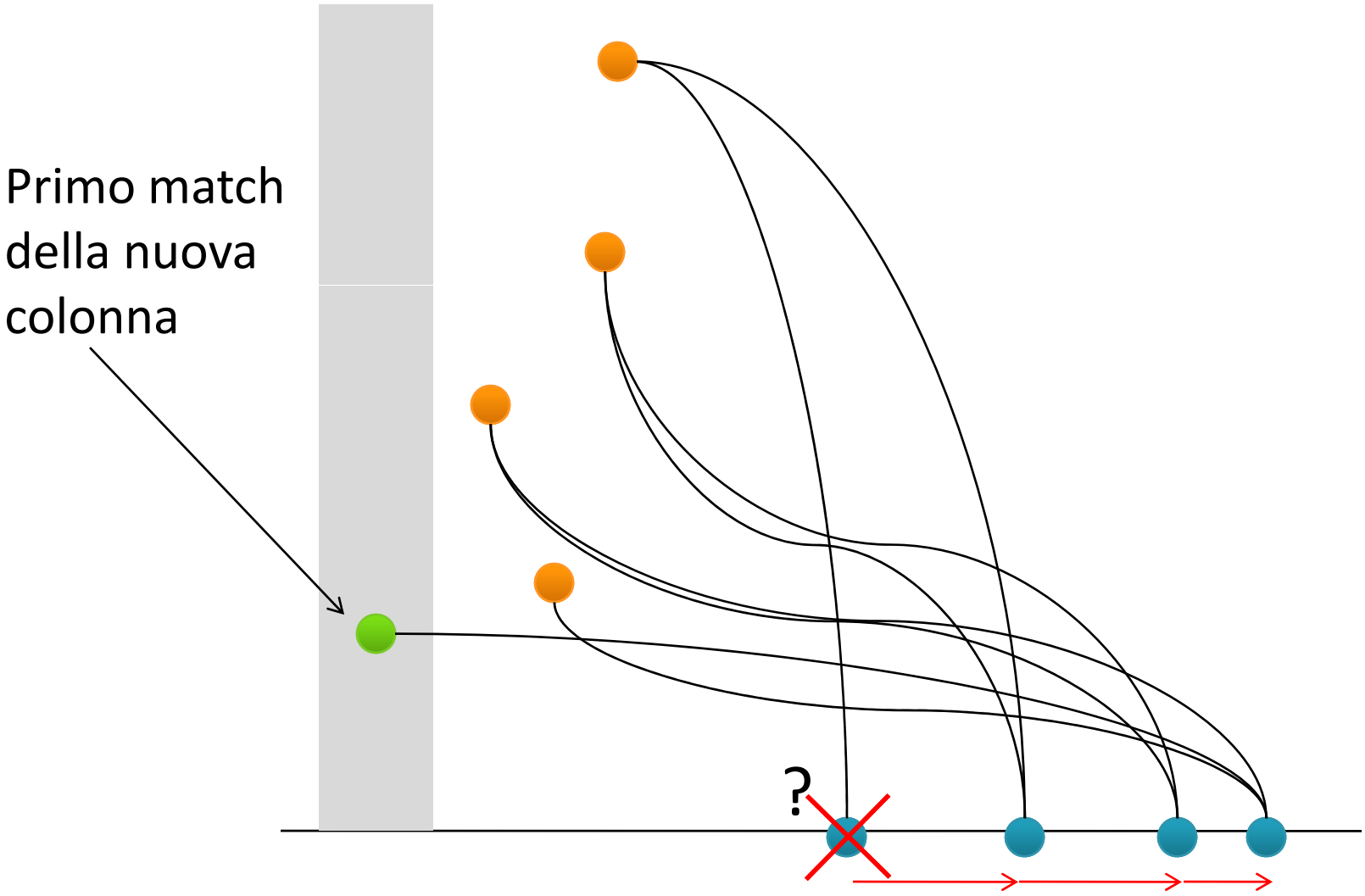
Secondo algoritmo

Idee generali: tail orfana

- t_{last} viene accoppiata al match point più alto
- **Può non esistere tale match point?** Si, se le head stanno sopra il primo match point della nuova colonna (essendo il primo non ce n'è uno più alto)
- **Allora t_{last} non viene accoppiata con nessun head e sparisce**
- Abbiamo trovato la tail che non viene ereditata dalla generazione k+1….
- …**e abbiamo anche una regola deterministica** per aggiornare PAIRS per la generazione successiva

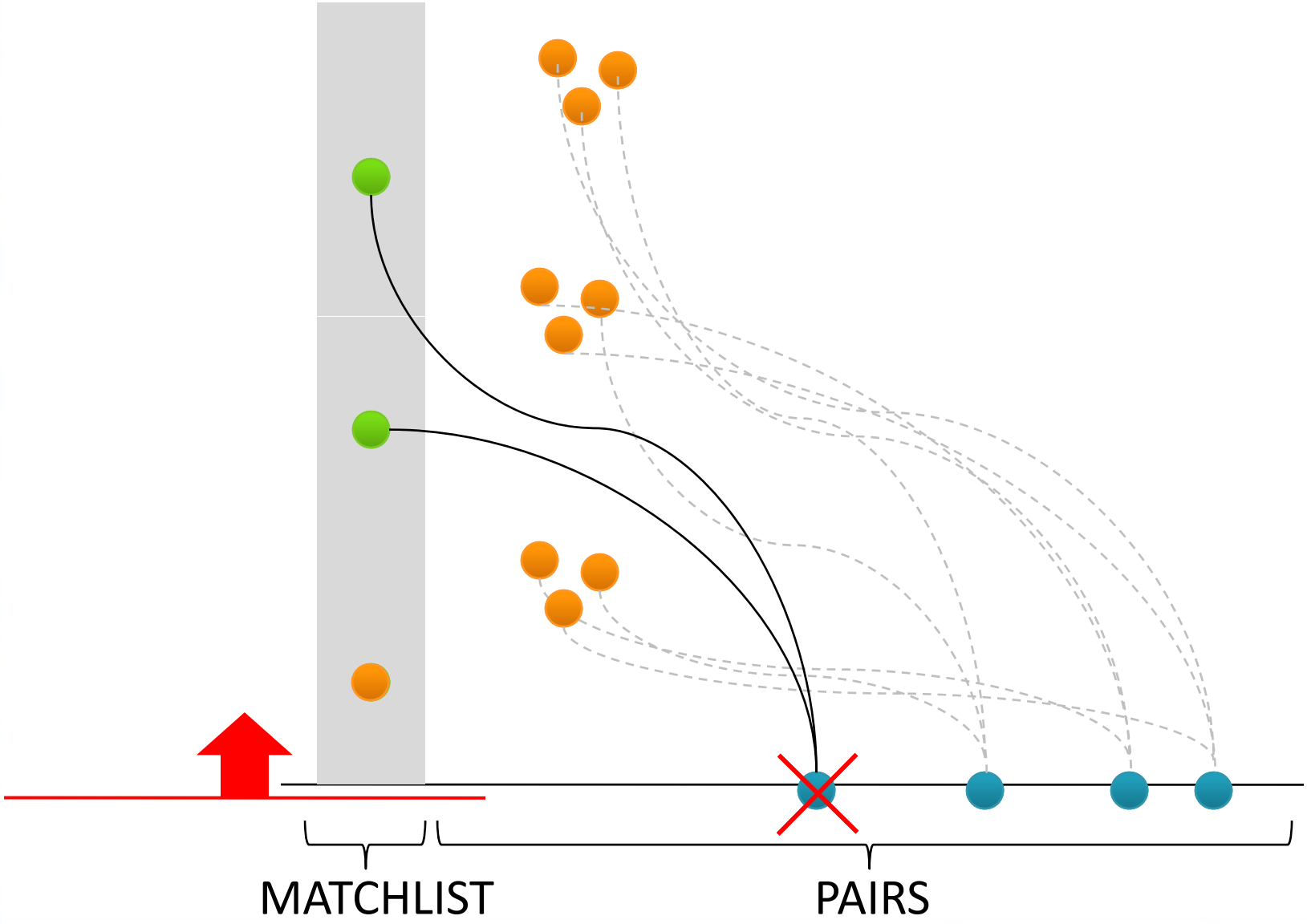
Secondo algoritmo

Idee generali: tail orfana



Secondo algoritmo

Idee generali: matchlist



Secondo algoritmo

Implementazione

- **TAILS**: partition points dell'ultima riga di DP^k, per ogni k
- **PAIRS**: coppie head-to-tail (h, t) ordinate per indice di riga decrescente

- **Preprocessing + m iterazioni**
- Ad ogni iterazione k:
  - Si considerano i match points della colonna k (MATCHLIST) e si aggiorna PAIRS calcolando le nuove head-to-tail pairs
  - Si inserisce la nuova head/tail
  - Si determina tail da rimuovere usando PAIRS aggiornata: aggiornamento di TAILS per esclusione

Secondo algoritmo

Implementazione

- Aggiornamento coppie in PAIRS scorrendo una sola volta (coppie per indice decrescente di riga)
- Quando si considerano le head che stanno sopra il primo match point si determina la tail che scompare
- **Costo: O(nL)**
  - **Numero di pair** nella generazione k = numero di tail nella generazione k = $\mathbf{L^k} < L$ (LCS fra A e B)
  - **Ogni pair controllato una sola volta** scorrendo PAIRS
  - **n iterazioni** per considerare tutti i suffissi

Conclusioni

- Punto chiave: riutilizzare l'output dell'allineamento precedente
- Algoritmi molto simili
  1. Scorrimento colonne, aggiornamento partition points di j utilizzando informazioni su j nella generazione precedente
  2. Scorrimento delle chain attive, determinazione partition point non ereditato utilizzando info sulle chain della generazione precedente
- Stesso costo computazionale O(nL)
- Il primo ha costo in spazio O(nL), il secondo O(L)
- **O(nL) vs O(n^3)**