Tomáš Oberhuber

Faculty of Nuclear Sciences and Physical Engineering
Czech Technical University in Prague

Video na Youtube

Porovnávání sekvencí - *sequence alignment*

- pokud biologové objeví nový gen, většinou se nezná jeho funkce
- zkouší se zjistit, kterým jiným genům je nejpodobnější
- bereme tedy postupně geny z databáze a porovnáváme je se zkoumaným genem
- snažíme se pak nějakým způsobem určit, jak moc jsou si dva geny podobné
- úloha se řeší pomocí metody dynamického programování
- nejprve si ukážeme tuto metodu

Editační vzdálenost

- vraťme se k naší úloze porovnávání dvou řetězců
- otázka je, jak ohodnotit podobnost dvou řetězců
- 1966, Vladimir Levenshtein – zadefinoval *editační* nebo také *levenshteinovu vzdálenost*
- ta je definována pomocí *editačních operací*
  - smazání znaku
  - vložení znaku
  - přepis znaku

Editační vzdálenost

Definition 1

Levenshteinova nebo editační vzdálenost mezi dvěma řetězci obecně různé délky je minimální počet editačních operací, které je potřeba provést k přepsání jednoho řetězce na druhý.

- Levenshtein sám nenašel algoritmus pro výpočet této vzdálenosti
- algoritmus byl objeven později nezávisle v několika různých oborech

Zarovnání řetězců

- k výpočtu editační vzdálenosti nejprve musíme provést tzv. zarovnání řetězců – *string alignment*

Definition 2

Mějme dva řetězce *v* a *w*. Jejich zarovnání je dvouřádková matice, která v prvním řádku obsahuje řetězec *v* a ve druhém řádku řetězec *w* oba případně doplněné o mezery tak, že v žádném sloupci nejsou dvě mezery.

Example 3

- **v =** `ATGTTAT`
- **w =** `ATCCTAC`

v = `AT-GTTAT-`

w = `ATCCT-A-C`

Zarovnání řetězců

- pokud jsou v jednom sloupci oba znaky stejné, jde o shodu (*match*)
- pokud jsou v jednom sloupci oba znaky různé, jde o neshodu/přepis (*mismatch*)
- je-li v jednom sloupci mezera, jde o tzv. indel
  - mezera v prvním řádků je vložení znaku – *insertion*
  - mezera v druhém řádku je mazání znaku – *deletion*
- jde nám o to, jak najít zarovnání s co nejmenším počtem editačních operací
- to získáme pomocí dynamického programováni
- nejprve si ale ukážeme jednduší úlohu, kde nebudeme uvažovat přepisování znaků, tj. jenom vkládání a mazání

Nejdelší společná podposloupnost

Definition 4

Podposloupnost (*subsequence*) řetězce je posloupnost znaků jdoucích po sobě, ale ne nutně bezprostředně po sobě. Tj. pro $\mathbf{v}=v_1 v_2 \ldots v_n$ a libovolnou ostře rostoucí posloupnost indexů $1 \leq i_1 < i_2 \ldots i_k \leq n$ je řetězec $v_{i_1} v_{i_2} \ldots v_{i_k}$ podposloupnost **v.**

Example 5

- **v =** `ATTGCTA`
- `AGCA` **a** `ATTA` **jsou podposloupnosti v**
- `TGTT` **není podposloupnost v – po** `GT` **již nenásleduje T**

Nejdelší společná podposloupnost

Definition 6

Společná podposloupnost (*common subsequence*) řetězců $\mathbf{v}=v_1 \ldots v_n$ a $\mathbf{w}=w_1 \ldots w_m$ a je posloupnost pozic

$$1 \leq i_1 < i_2, \ldots i_k \leq n,$$

ve **v** a posloupnost pozic

$$1 \leq j_1 < j_2, \ldots, j_k \leq m,$$

ve **w** takových, že

$$v_{i_t} = w_{j_t},$$

pro všechna $t = 1, \ldots, k$.

Nejdelší společná podposloupnost

- pokud provedeme zarovnání dvou řetězců bez přepisů, stejné znaky pod sebou tvoří společnou podposlounpost

Example 7

- **v =** `ATGTTAT`
- **w =** `ATCCTAC`

v = `AT-G-TTAT-`

w = `ATC-CT-A-C`

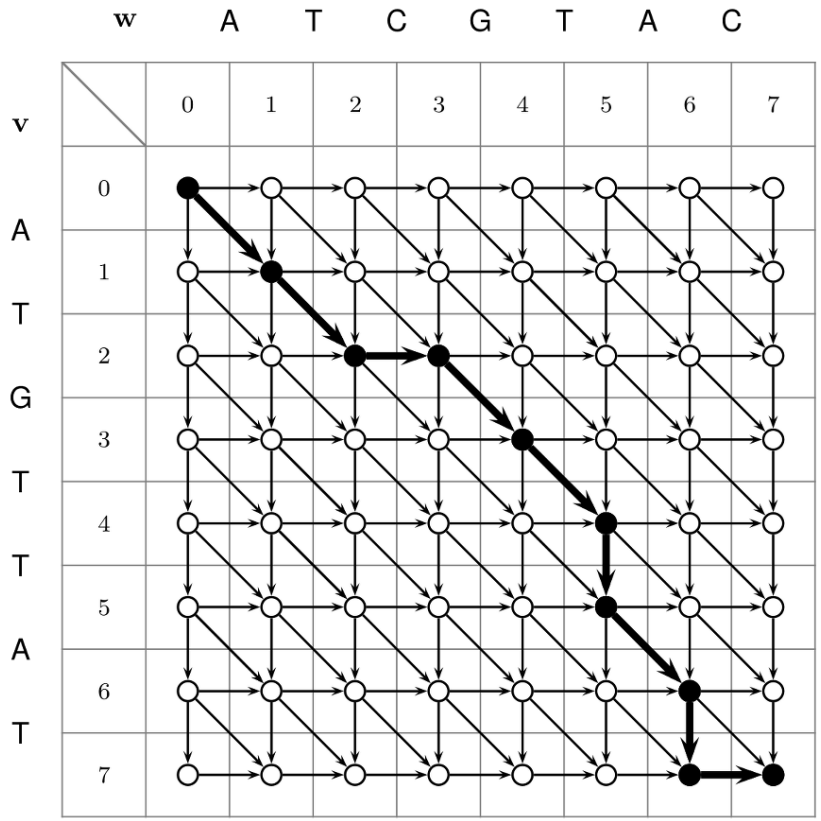
Nejdelší společná podposloupnost

- naším úkolem bude najít nejdelší společnou podposloupnost dvou řetězců **v** a **w**
- k výpočtu zarovnání použijeme algoritmus dynamického programování
- řetězce napíšeme podél horní a levé strany grafu, tak že znaky řetězců jsou zarovnané na hrany grafu
- tam, kde se řetězce shodují, uděláme šikmou hranu doprava dolu
- dolníme všechny svislé a vodorovné hrany

Nejdelší společná podposloupnost

		0	1	2	2	3	4	5	6	7	7
v	=		A	T	-	G	T	T	A	T	-
			|	|		|	|		|		
w	=		A	T	C	G	T	-	A	-	C
		0	1	2	3	4	5	5	6	6	7

	↘	↘	→	↘	↘	↓	↘	↓	→
	A	T	-	G	T	T	A	T	-
	A	T	C	G	T	-	A	-	C

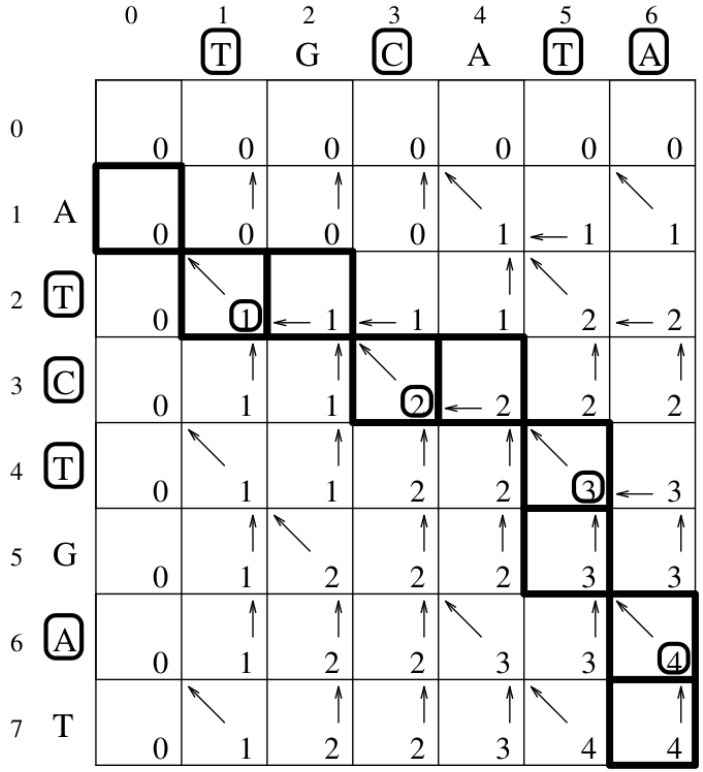


Zdroj: Jones, Pevzner, An introduction to bioinformatics algorithms

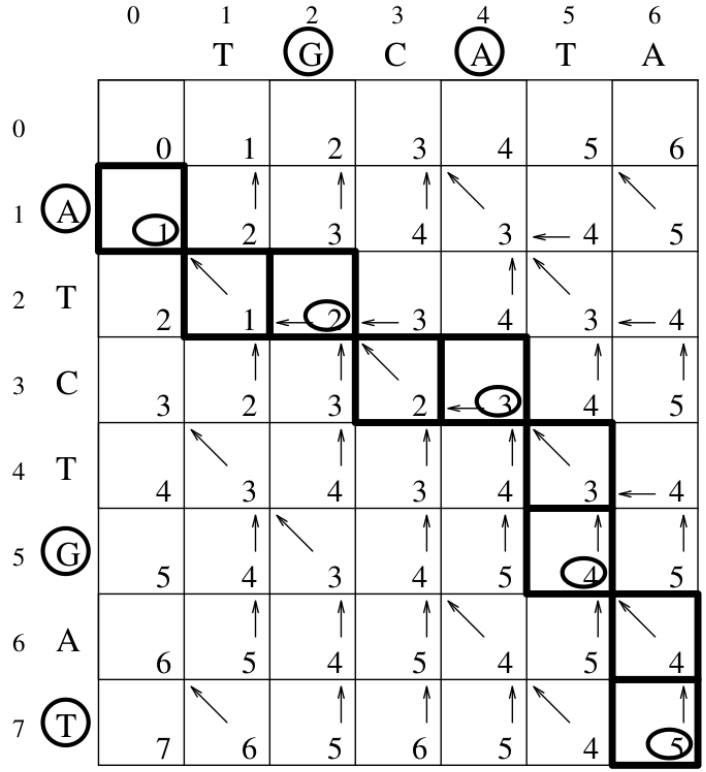
Nejdelší společná podposloupnost

- jakákoliv cesta z levého horního do pravého dolního rohu pak odpovídá nějakému zarovnání
  - krok po šikmé hraně odpovídá shodě
  - krok po svislé nebo vodorovné hraně odpovídá mezeře v jednom z řetězců
- pokud šikmé hrany ohodnotím jedničkou a ostatní nulou, stačí už jen najít nejdelší cestu v grafu
- její délka pak odpovídá délce nejdelší společné podposloupnosti

Nejdelší společná podposloupnost



Computing similarity s(V,W)=4
V and W have a subsequence TCTA in common



Computing distance d(V,W)=5
V can be transformed into W by deleting A,G,T and inserting G,A

Alignment:

A T - C - T G A T
- T G C A T - A -

Nejdelší společná podposloupnost

- řešíme tedy úlohu dynamického programování s předpisem

$$s_{ij} = \max \begin{cases} s_{i-1,j} + 0 & \\ s_{i,j-1} + 0 & \\ s_{i-1,j-1} + 1 & \text{pokud } v_i = w_j \end{cases}$$

- pokud chci následně získat LCS, jde pouze o rekonstrukci nejdelší cesty

Globální porovnávání řetězců

- nyní se vracíme k naší původní úloze porovnávání řetězců
- kromě vkládání a mazání znaků umožníme provádět i přepisy právě podle definice editační vzdálenosti
- do našeho grafu dynamického programování tak vložíme šikmé hrany i tam, kde se oba řetězce nerovnají
- tyto nové hrany ale musíme ohodnotit menší vahou než 1, aby algoritmus upřednostňoval aplikaci shody řetězců

Globální porovnávání řetězců

Obecně můžeme použít následující penalizace:

- $-\mu$ pro přepis
- $-\sigma$ pro vložení nebo mazání znaku
- 1 pro shodu znaků

Tím získáme následující pravidlo:

$$s_{ij} = \max \begin{cases} s_{i-1,j} - \sigma & \\ s_{i,j-1} - \sigma & \\ s_{i-1,j-1} - \mu & \text{pro } v_i \neq w_j \\ s_{i-1,j-1} + 1 & \text{pro } v_i = w_j \end{cases}$$

Example 8

http://demonstrations.wolfram.com/GlobalAndLocalSequenceAlignmentAlgorithms/

Globální porovnávání řetězců

- to lze zobecnit zavedením matice δ, která všechny možné dvojice znaků, které se mohou vyskytnout pod sebou v matici zarovnání.
- v našem případě by vypadala takto

$$\delta \equiv \left(\begin{array}{c|ccccc} & A & T & C & G & - \\ \hline A & 1 & -\mu & -\mu & -\mu & -\sigma \\ T & -\mu & 1 & -\mu & -\mu & -\sigma \\ C & -\mu & -\mu & 1 & -\mu & -\sigma \\ G & -\mu & -\mu & -\mu & 1 & -\sigma \\ - & -\sigma & -\sigma & -\sigma & -\sigma & -\infty \end{array} \right)$$

- matici δ nazýváme maticí skóre (*scoring matrix*)

Globální porovnávání řetězců

- algoritmus dynamického programování by se pak řídil pravidlem

$$s_{ij} = \max \begin{cases} s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \\ s_{i-1,j-1} + \delta(v_i, w_j) \end{cases}$$

- výhodou je, že nyní můžeme každý přepis ohodnotit jinak

Globální porovnávání proteinů

- to je výhodné při porovnávání proteinů
- zde vlastně porovnáváme řetězce tvořené abecedou aminokyselin
- biologové vědí, že v přírodě se díky mutacím některé aminokysleliny zaměňují poměrně často, zatímco jiné velmi zřídka
- hodnota $\delta(i,j)$ pak vyjadřuje pravděpodobnost, s jakou se zaměňují aminokyseliny i a j
- tyto pravděpodobnosti ale závisí na zkoumaném proteinu
- pokud budeme mít dostatečně velkou databázi porovnaných proteinů, můžeme tyto pravděpodobnosti snadno spočítat
- tuto databázi ale nedostaneme, dokud nenapočítáme porovnání proteinů, k čemuž tyto pravděpodobnosti potřebujeme
- postupuje se tak, že nejprve porovnáme proteiny s jednoduchou maticí

Lokální porovnávání řetězců

- v některých situacích nás více zajímá co nejlepší shoda určitých podřetězců obou sekvencí raději, než porovnání celých sekvencí

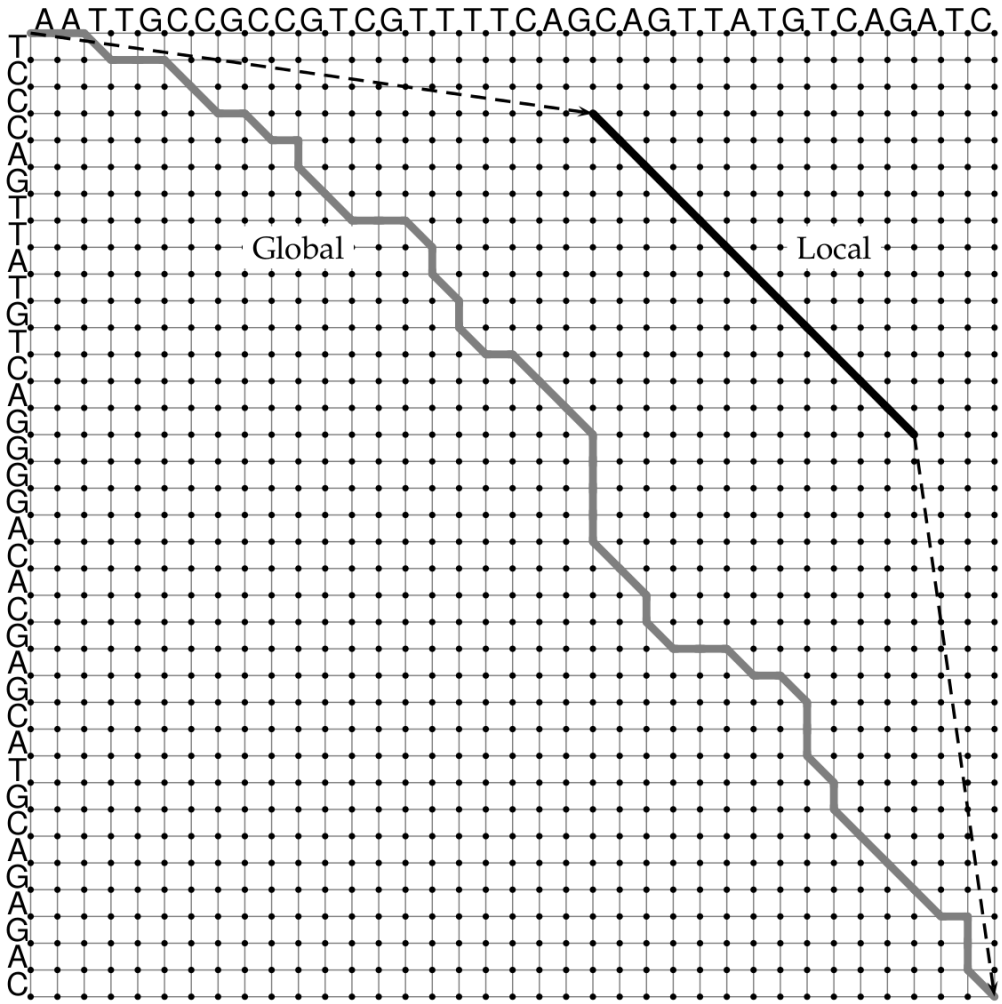
```
--T--CC-C-AGT--TATGT-CAGGGGACACG--A-GCATGCAGA-GAC
  |   || |   ||   | | | |||     || |   | |   | ||||    |
AATTGCCGCC-GTCGT-T-TTCAG----CA-GTTATG--T-CAGAT--C

                    tccCAGTTATGTCAGgggacacgagcatgcagagac
                       ||||||||||||
aattgccgccgtcgttttcagCAGTTATGTCAGatc
```

Zdroj: Jones, Pevzner, An introduction to bioinformatics algorithms

- v grafu dynamického programování takové zarovnání vypadá takto...

Lokální porovnávání řetězců

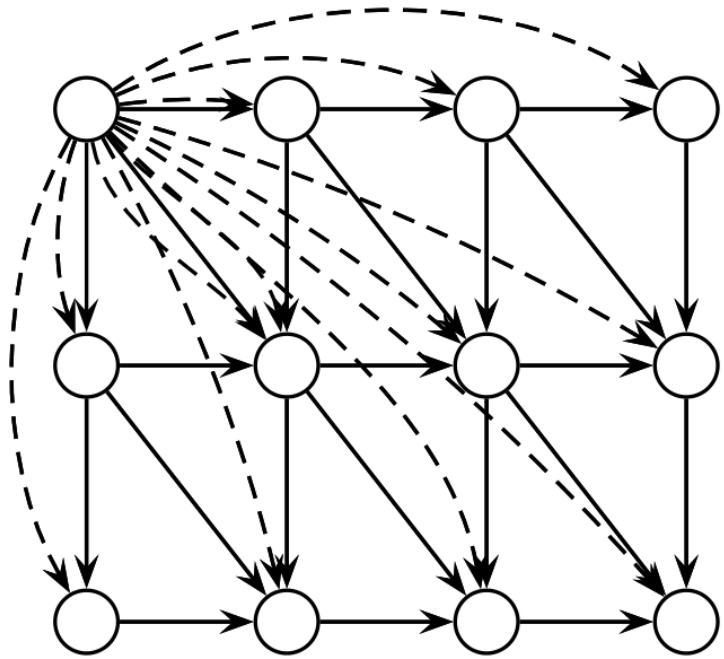


Zdroj: Jones, Pevzner, An introduction to bioinformatics algorithms

Lokální porovnávání řetězců

- co brání nalezení takové cesty?
- jsou to penalizace nabrané před a po požadovaném zarovnání
- rádi bysme našli způsob, jak je najít
- penalizací před zarovnáním se lze zbavit tak, umožníme přeskočit tu část zarovnání, kde získáváme záporné body
- v grafu to znamená, že ke každému vrcholu přidáme hranu vedoucí do levého horního rohu s nulovou vahou

Lokální porovnávání řetězců



Zdroj: Jones, Pevzner, An introduction to bioinformatics algorithms

Lokální porovnávání řetězců

- vztahy pro úlohu dynamického programování pak mají tvar

$$s_{ij} = \max \begin{cases} 0 \\ s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \\ s_{i-1,j-1} + \delta(v_i, w_j) \end{cases}$$

- jak se zbavit penalizací za lokálním zarovnáním?
- po ohodnocení všech vrcholů grafu algoritmem dynamického programování najdeme vrchol s nejvyšším skóre
- ten znamená konec nejlepšího lokálního zarovnání
- od něj pak běžným způsobem rekonstruujeme nejdelší cestu, dokud nenarazíme na vrchol s nulovou hodnotou
- jde o Smithův-Watermanův algoritmus

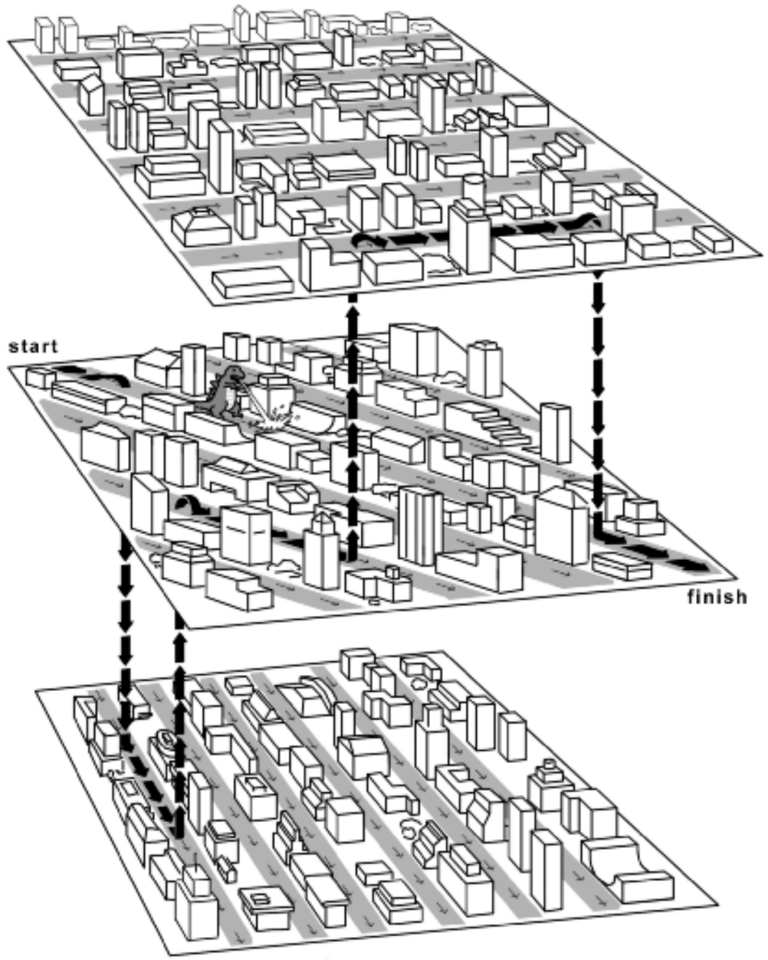
Porovnávání řetězců s dlouhými sekvencemi

- v přírodě jsou častejší podobnosti, kdy se záměny provádí po delších sekvencích a ne po jednotlivých nukleotidech
- k nalezení takových zarovnání je potřeba zavést vyšší penalizaci pro vložení první mezery
  - první mezeru v sekvenci mezer budeme navíc penalizovat hodnotou $-\rho$
  - další mezery hodnotou $-\sigma$
- problém je v tom, jak v daném bodě zjistit, na jak dlouhou sekvenci mezer napojujeme další mezeru
- mohli bysme zkusit opět přidat další hrany do grafu
- bylo by ale nutné přidat všechny možné svislé a vodorovné hrany, což by zpomalilo algoritmus

Porovnávání řetězců s dlouhými sekvencemi

- zavedeme si dva pomocné grafy $s^{\rightarrow}$ a $s^{\downarrow}$
- nyní budeme hledat nejdelší cestu po třech grafech
- v jednom se nám půjde nejlépe doprava, ve druhém dolu a ve třetím šikmo dolů

Porovnávání řetězců s dlouhými sekvencemi



Zdroj: Jones, Pevzner, An introduction to bioinformatics algorithms

Porovnávání řetězců s dlouhými sekvencemi

- rekurentní vztahy pak mají tvar

$$s^{\downarrow}_{ij} = \max \begin{cases} s^{\downarrow}_{i-1,j} - \sigma \\ s_{i-1,j} - (\rho + \sigma) \end{cases}$$

$$s^{\rightarrow}_{ij} = \max \begin{cases} s^{\rightarrow}_{i,j-1} - \sigma \\ s_{i,j-1} - (\rho + \sigma) \end{cases}$$

$$s_{ij} = \max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) \\ s^{\downarrow}_{i-1,j} \\ s^{\rightarrow}_{i,j-1} \end{cases}$$

Porovnávání více sekvencí

- nyní se pokusíme o porovávání více sekvencí najednou
- pro tři řetězce v, w, u to může vypadat například takto

```
--T--CC-C-AGT--TATGT-CAGGGGACACG--A-GCATGCAGA-GAC
  |   || |   ||   | | | |||     || |   | |   | ||||    |
AATTGCCGCC-GTCGT-T-TTCAG----CA-GTTATG--T-CAGAT--C
 ||||| |    X|||| |           || XXX|||   | ||| |   |
-ATTGC-G--ATTCGTAT------GGGACA-TGGATGCATGCAG-TGAC
```

Zdroj: Jones, Pevzner, An introduction to bioinformatics algorithms

Porovnávání více sekvencí

- nyní při sestavování každého dalšího sloupce matice můžeme provést:
  - vložení znaku z jednoho řetězce a dvě mezery
  - vložení různých znaků ze dvou řetězců a jedné mezery
  - vložení svou stejných znaků a jedné mezery
  - vložení tří různých znaků
  - vložení tří stejných znaků
- to vše musíme opět vhodně ohodnotit
- použijeme opět "matici" δ
- úloha dynamického programování se bude řešit ve třech dimenzích

Porovnávání více sekvencí

- rekurence pro úlohu dynamického programování má tvar:

$$s_{ijk} = \max \begin{cases} s_{i-1,j,k} + \delta(v_i, -, -) \\ s_{i,j-1,k} + \delta(-, w_j, -) \\ s_{i,j,k-1} + \delta(-, -, u_k) \\ s_{i-1,j-1,k} + \delta(v_i, w_j, -) \\ s_{i-1,j,k-1} + \delta(v_i, -, u_k) \\ s_{i,j-1,k-1} + \delta(-, w_j, u_k) \\ s_{i-1,j-1,k-1} + \delta(v_i, w_j, u_k) \end{cases}$$

Porovnávání více sekvencí

- snadno tak lze odvodit algoritmus pro porovnvání obecně k sekvencí
- problém je ale v tom, že složitost je $O(n^k)$, pokud budou mít všechny sekvence délku přiližně n
- pro $n = 1000$ a $k = 3$ jsou paměťové nároky $sizeof(int) \cdot 1000^3 = 4$ GB
- často tedy musíme použít heuristiky
- mohli bysme zkusit napočítat optimální zarovnání všech $\binom{k}{2}$ dvojic a ty pak skombinovat dohromady
- to se ale nemusí vždy podařit

Porovnávání více sekvencí

Zdroj: Jones, Pevzner, An introduction to bioinformatics algorithms

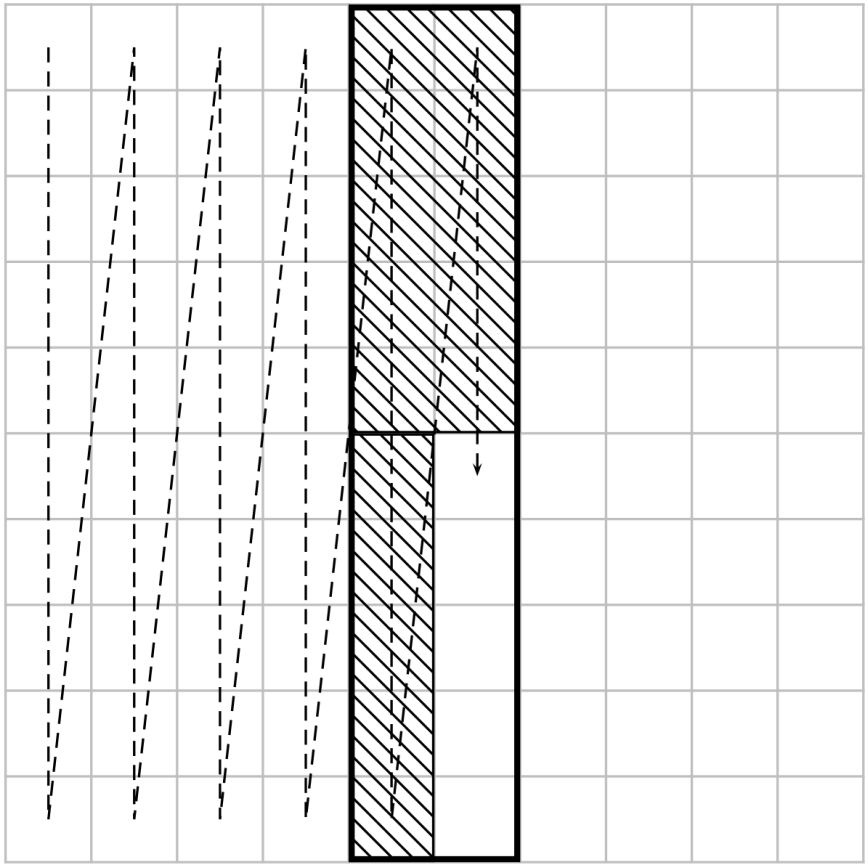
Porovnávání více sekvencí

- efektivním algoritmem je progresivní zarovnávání sekvencí
- vybere se dvojice sekvencí s nejlepším zarovnáním
- ty se pak sloučí do jednoho řetězce
- k němu se pak přidává další
- tento algoritmus používá známý program Clustal

Paměťově efektivní porovnání sekvencí

- problém algoritmu pro porovnávání sekvencí není ani tak v časové složitosti jako spíš ve velkém nároku na paměť
- 1975 - Daniel Hirschberg, navrhl algoritmus typu "rozděl a panuj", který paměťové nároky snižuje
- využívá faktu, že k napočítání j-tého sloupce matice/grafu s nám stačí znát pouze $j-1$-ní sloupec
- předchozí sloupce využívá pouze při rekonstrukci nejdelší cesty
- pokud by nám stačilo napočítat pouze délku nejdelší cesty, nemusíme si tyto sloupce už pamatovat

Paměťově efektivní porovnání sekvencí



Zdroj: Jones, Pevzner, An introduction to bioinformatics algorithms

Paměťově efektivní porovnání sekvencí

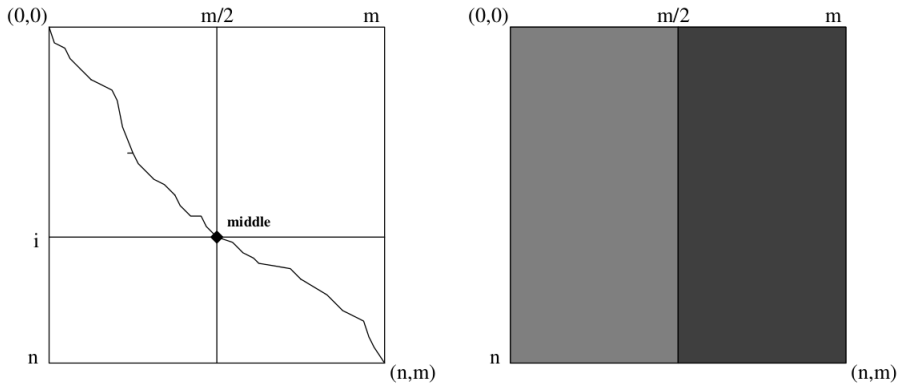
- buď j^* index prostředního sloupce
- nechť nejdelší cesta prochází jeho i^*-tým řádkem
- otázka je, jak tento index i^* najít?
- délky všech n cest z levého horního rohu do pravého dolního lze vyjádřit pro $i = 1, \ldots, n$ jako součet délky
  - z levého horního rohu do j^*-tého sloupce – $prefix_i$
  - z pravého dolního rohu do j^*-tého sloupce – $suffix_i$
    - řešímé úlohu dynamického programování se startem v pravém dolním rohu
- snadno pak vidíme, že

$$j^* = \arg \max_{i=1,\ldots,n} (preffix_i + suffix_i)$$

Paměťově efektivní porovnání sekvencí

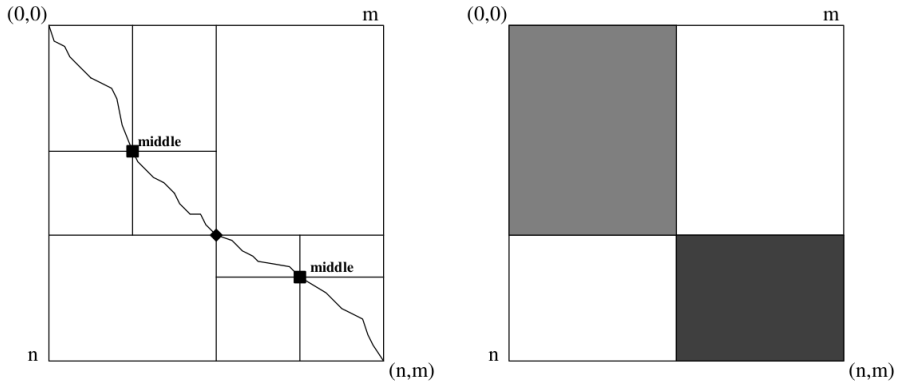
- k výpočtu $prefix_i$ řešíme úlohu dynamického programování se zahazováním nepotřebných sloupců s počátkem v **levém horním** rohu
- k výpočtu $suffix_i$ řešíme úlohu dynamického programování se zahazováním nepotřebných sloupců s počátkem v **pravém dolním** rohu
- celkem tedy potřebujeme ukládat jen $3n$ prvků
- pokud známe souřadnice (i^*, j^*) rozdělíme problém na dva
  - nalezení nejdelší cesty z levého horního bodu do (i^*, j^*)
  - nalezení nejdelší cesty z (i^*, j^*) do pravého dolního rohu

Paměťově efektivní porovnání sekvencí



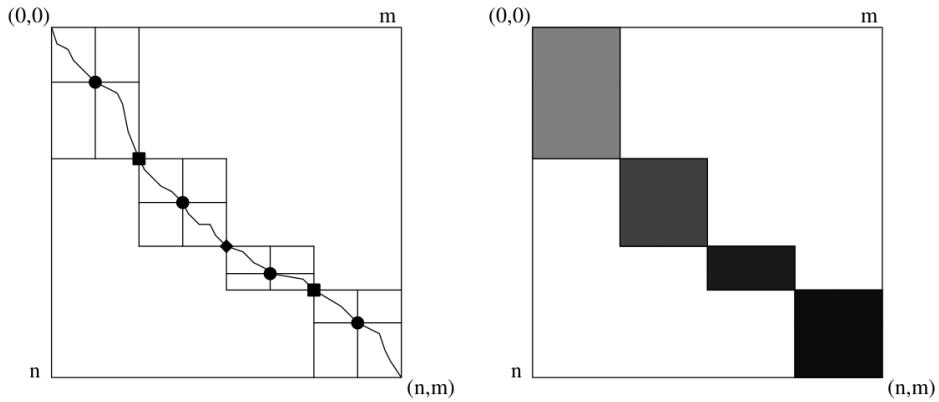
Zdroj: Jones, Pevzner, An introduction to bioinformatics algorithms

Paměťově efektivní porovnání sekvencí



Zdroj: Jones, Pevzner, An introduction to bioinformatics algorithms

Paměťově efektivní porovnání sekvencí



Zdroj: Jones, Pevzner, An introduction to bioinformatics algorithms

Paměťově efektivní porovnání sekvencí

- jaká je časová složitost?
- po každém dělení procházíme jen polovinu uzlů grafu s znovu, tj.

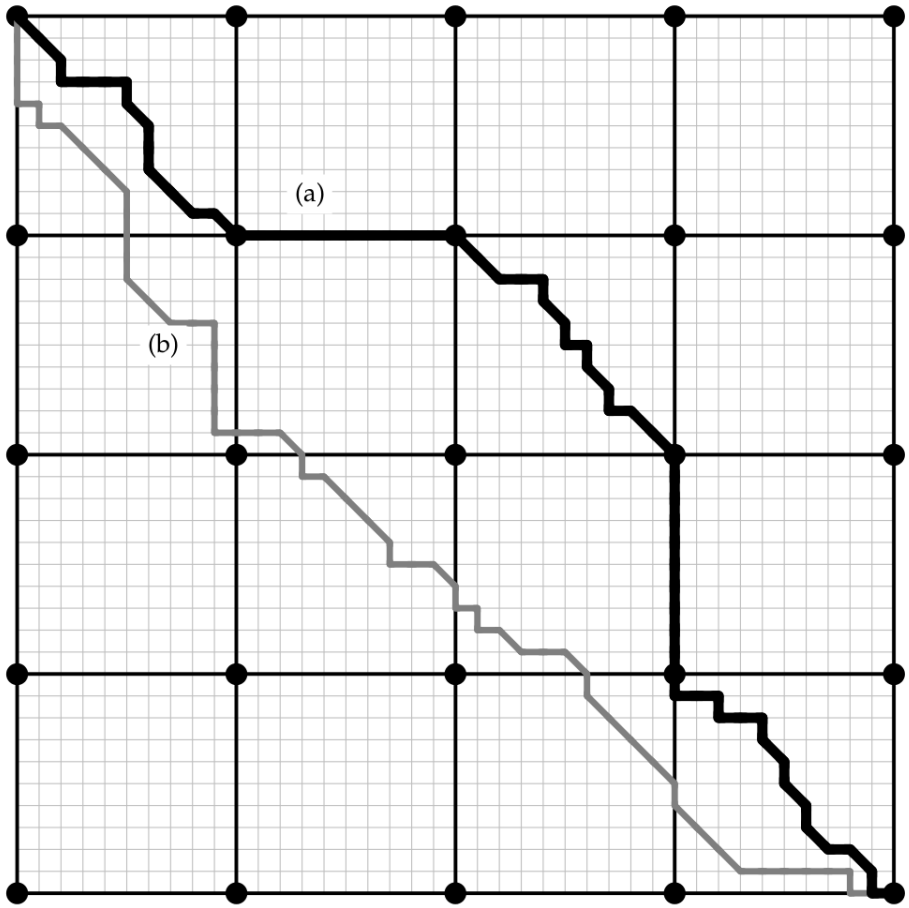
$$1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots \leq 2$$

- algoritmus tedy poběží maximálně dvakrát tak dlouho jako normální algoritmus
- protože ale máme větší šanci, že výpočet poběží v cache CPU, může být výpočet ještě výrazně rychlejší

Blokové porovnávání sekvencí

- zatím není známý žádný spodní odhad složitosti pro porovnávání sekvencí
- algoritmus dynamického programování vede na kvadratickou složitost
- ukážeme si, jak algoritmus urychlit za cenu horšího výsledku
- použijeme tzv. **blokové porovnávání**
- předpokládejme pro jednoduchost velikost úlohy $n \times n$, tj. $|v| = |w| = n$
- graf s rozdělíme na bloky o velikosti $t \times t$, a cesty omezíme tak, že musí procházet vždy pouze skrz rohy bloků

Blokové porovnávání sekvencí



Zdroj: Jones, Pevzner, An introduction to bioinformatics algorithms

Blokové porovnávání sekvencí

- jelikož nyní víme, že do každého bloku může každá cesta vstupovat jen levým horním rohem a vystupovat pravým dolním, můžeme si nejprve napočítat cesty skrz jednotlivé bloky
- získané skóre v každém bloku označíme jako β_{ij} tj. délka nejdelší cesty blokem
- následně řešíme úlohu dynamického programování pro bloky
- sestavíme tedy graf S tvořený jednotlivými bloky a řešíme úlohu

$$S_{ij} = \max \begin{cases} S_{i-1,j} - t\sigma \\ S_{i,j-1} - t\sigma \\ S_{i-1,j-1} + \beta_{ij} \end{cases},$$

Blokové porovnávání sekvencí

- složitost blokové úlohy je $O(n^2/t^2)$
- vyřešení jednoho bloku je $O(t^2)$
- bloků je n^2/t^2, což dává opět $O(n^2)$
- abychom získali rychlejší algoritmus, je potřeba si skóre pro jednotlivé bloky β_{ij} předpočítat
- každý blok je jednoznačně určen částí řetězce w nad jeho horní hranou a částí řetězce v podél jeho levé hrany
- tj. znaky $v_{it}, v_{it+1}, \ldots, v_{it+t-1}$ a $w_{jt}, w_{jt+1}, \ldots, w_{jt+t-1}$
- v abecedě se čtyřmi znaky je to celkem $4^t \times 4^t$ možností
- pokud vollíme $t = \log_4 n/2$ je

$$4^t \cdot 4^t = 4^{\log_4 n/2} \cdot 4^{\log_4 n/2} = n^{\frac{1}{2}} n^{\frac{1}{2}} = n$$

- to lze předpočítat do tabulky se složitostí $O(n \log^2 n)$

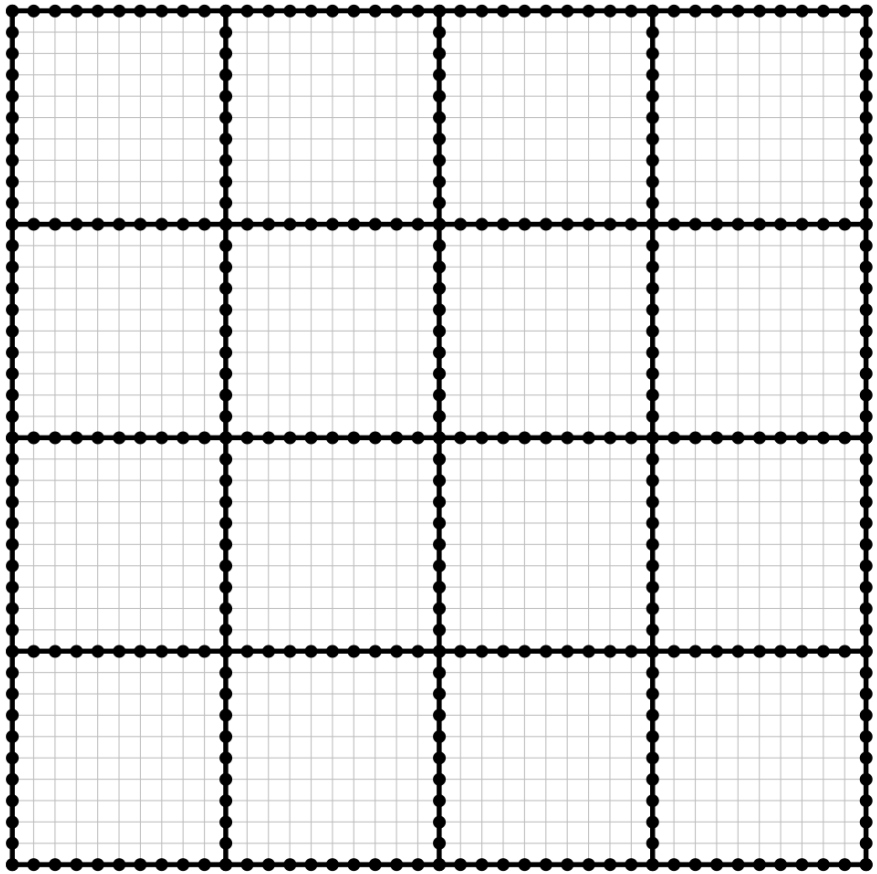
Blokové porovnávání sekvencí

- složitost blokového dynamického programování je $O(n^2 / \log^2 n)$
- celkem tedy $O(n^2 / \log^2 n)$
- trik je tedy v tom, že pokud si velikost bloků zvolíme dostatečně malou, musí se některé bloky opakovat
- tím, že je ale předpočítám do tabulky, napočítám každý blok jen jednou

Blokové porovnávání pro LCS

- pro úlohu nalezení nejdelší společné podposloupnosti lze pomocí bloků získat dokonce optimální řešení
- umožníme, aby libovolná cesta mohla křižovat hranici bloku kdekoliv, ne jen v protějších rozích

Blokové porovnávání pro LCS



Zdroj: Jones, Pevzner, An introduction to bioinformatics algorithms

Blokové porovnávání pro LCS

- řešíme tedy úlohu dynamického programování na všech uzlech na hranách bloků
- každý blok je nyní popsán
  - podřetězci $v_{it}, v_{it+1}, \ldots, v_{it+t-1}$ a $w_{jt}, w_{jt+1}, \ldots, w_{jt+t-1}$
  - hodnotami skóre s_{ij} podél své horní a levé hrany
- ze znalosti těchto vstupních dat dokážeme napočítat skóre s_{ij} v libovolném uzlu podél pravé a dolní hrany bloku
- pro úlohu globálního porovnávání řetězců je to příliš mnoho kombinací na to, aby se daly předpočítat do tabulky
- pro úlohu LCS to ale lze

Blokové porovnávání pro LCS

- úloha LCS je dána předpisem

$$s_{ij} = \max \begin{cases} s_{i-1,j} + 0 & \\ s_{i,j-1} + 0 & \\ s_{i-1,j-1} + 1 & \text{pokud } v_i = w_j \end{cases}$$

- skóre s_{ij} se tedy může podél horní a levé hrany měnit pouze o 0 nebo 1
- to lze zakódovat do posloupnosti nul a jedniček o délce t
- celkem tak máme

$$4^t \cdots 4^t \cdot 2^t \cdot 2^t = 2^{6t}$$

kombinací

- pro $t = \log_2 n/4 \rightarrow n^{1.5}$ kombinací

Blokové porovnávání pro LCS

- zmíněná technika se nazývá Four-Russians Speed-up po autorech
- V.L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzev. On economical construction of the transitive closure of an oriented graph. Soviet Math. Dokl., 11:1209– 1210, 1970.

Program diff

- je založen na algoritmu pro hledání LCS
- E.W.Myers, *An O(ND) Difference Algorithm and Its Variations*, Algorithmica, (1), 251–266, 1986.
  - N je součet délek řetězců v a w
  - D jejich editační vzdálenost
  - lze dosáhnout složitosti $O(N + D^2)$
  - paměťová náročnost je $O(N)$

Dijkstrův algoritmus

```
1:  procedure DIJKSTRA( V, E, w, s))
2:      V_F := {s} , V_T := ∅, l[s]
3:      for all v ∈ V \ V_F do
4:          if existuje hrana (s, v) then
5:              l[v] := w(s, v), V_T := V_T ⋃ {v}
6:          else
7:              l[v] := +∞
8:          end if
9:      end for
10:     while V_F ≠ V do
11:         najdi u ∈ V_T, že l[u] = min_{v∈V_T} l[v]
12:         V_F := V_F ⋃ {u}
13:         for all v ∈ N_V(u) \ V_F do
14:             l[v] := min{l[v], l[u] + w(u, v)}
15:             V_T := V_T ⋃ {v}
16:         end for
17:     end while
18:     return l
19: end procedure
```