Bioinformatički algoritmi

Analiza DNK sekvenci

Petar Veličković

Matematička gimnazija, Nedelja informatike

30. mart 2015.

Bioinformatika

- **Bioinformatika** (*bioinformatics*): polje računarstva koje razvija algoritme, strukture podataka i softverske alate za analizu bioloških podataka.

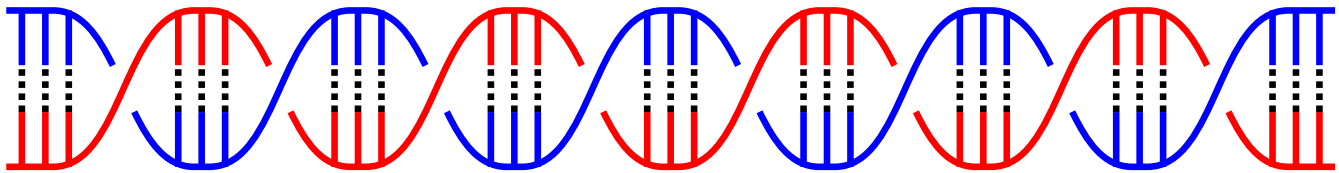
- Pored računarskih nauka i biologije, obuhvata i elemente statistike, matematike i inženjerstva.

Bioinformatički algoritmi

- Bioinformatički algoritmi pripadaju raznim oblastima:
  - Teorija grafova;
  - Veštačka inteligencija;
  - Analiza podataka (*data mining*);
  - Obrada slika;
  - Računarska simulacija;
  - ...
- Bioinformatički algoritmi proučavaju pojave na gotovo svim *nivoima organizacije* unutar nekog organizma:
  - Tkiva;
  - Ćelije;
  - Putevi (*Pathways*);
  - Biohemijske reakcije;
  - Biomolekuli (**DNK**, RNK, proteini...)

DNK

- **Dezoksiribonukleinska kiselina** (*deoxyribonucleic acid*, skraćeno DNK/*DNA*): molekul koji u sebi sadrži genetičke instrukcije neophodne za razvoj i funkcionisanje gotovo svih živih organizama.
- Sastoji se od dve niti isprepletane u formi **dvostrukog heliksa**:



- Niti se sastoje od niza osnovnih baza, tj. **nukleotida**:
  - Adenin (`A`);
  - Timin (`T`);
  - Citozin (`C`);
  - Guanin (`G`).
  - Nukleotidi su uvek **komplementarni** (`A-T`, `C-G`), tako da je za analizu *dovoljno znati jednu nit DNK*.

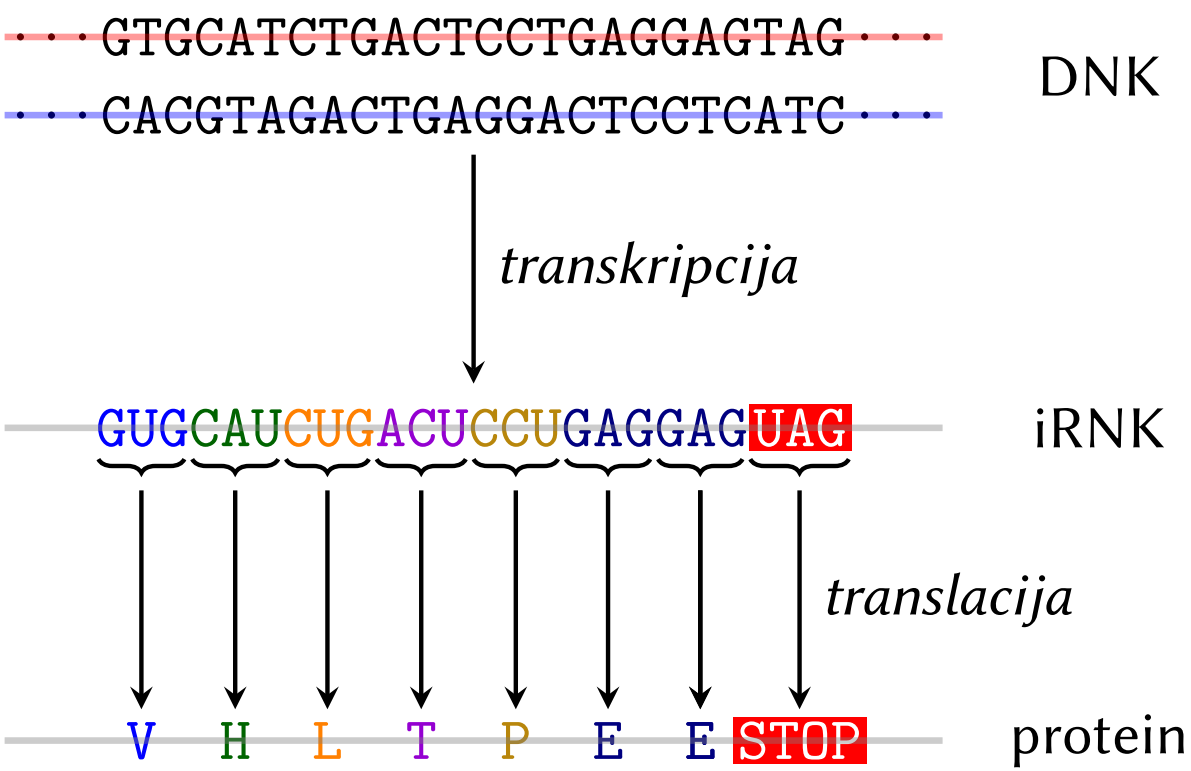
Transkripcija gena

- Glavna funkcija DNK je čuvanje "recepta" za sintezu proteina.
- Proteini se dobijaju 3D ‘uvrtanjem’ nekog niza amino-kiselina; deo DNK koji sadrži informacije neophodne za sintezu jednog takvog niza se naziva **gen**.
- Sinteza proteina počinje prepisivanjem (*transkripcijom*) neophodnog gena u RNK molekul (informacioni RNK/*messenger RNA* (iRNK/*mRNA*)).
- **Ribonukleinska kiselina** (*ribonucleic acid*, skraćeno RNK/*RNA*) je molekul sličan DNK, s tim što je timin zamenjen uracilom (U) i sastoji se od samo jedne niti.

Translacija u protein

- Unutar iRNK, nukleotidi se dele u grupe od po tri susedna, koje se zovu **kodoni**.
- Svaki kodon ili odgovara jednoj amino-kiselini, ili je specijalan *stop* kodon. Stop kodoni su `UAG`, `UAA` i `UGA`.
- Prevođenje (*translacija*) iRNK u niz amino-kiselina se radi kodon po kodon, sve dok se ne dođe do stop kodona.

Sinteza proteina – sumarizovano



DNK mutacije

- Struktura DNK nije konstantna, nego je podložna **mutacijama**; ovo je deo *prirodne selekcije* i način kojim nove, prilagođenije vrste organizama nastaju od starijih.

- Neki tipovi mutacija:
  - *Supstitucija:*
    TGCATTGCGTAGGC
    TGCATTCCGTAGGC
  - *Insercija/delecija:*
    TGCATT---TAGGC
    TGCATTCCGTAGGC
  - *Inverzija:*
    TGCATTGCGTAGGC
    TGCGCGTTATAGGC

Problem poravnanja

- Ovo nas dovodi do glavnog problema koji ćemo razmatrati: problem **poravnanja DNK sekvenci** (*DNA sequence alignment*).
- Pretpostavimo da imamo dve DNK sekvence, koje pripadaju dvema različitim vrstama. Zadatak je da identifikujemo slične ($\sim$ nemutirane) regione između njih.
- Razlikujemo dva odvojena problema:
  - **Globalno poravnanje**: poravnanje celokupnih sekvenci;
  - **Lokalno poravnanje**: pronalaženje regiona visoke sličnosti.
- Poravnanjem takođe odredjujemo i koliko su dve DNK sekvence "udaljene" jedna od druge.

Hamming distanca

- Najjednostavniji algoritam za poravnanje razmatra **samo supstitucije** kao moguće mutacije.
- Udaljenost se onda računa kao *Hamming* distanca između sekvenci; broj mesta na kojima se nukleotidi razlikuju.
- `Hamming("ATCG", "ACCG") = 1`
- $O(n)$ algoritam za računanje trivijalan (uz potencijalne optimizacije ako više puta računamo distancu nad podstringovima iste sekvence... *Quals 2015: Retrovirus*)

Dinamičko programiranje

- Hamming distanca nije dobra metrika u opštem slučaju, npr: `Hamming("ATATATAT", "TATATATA") = 8`

- Za korektniji rezultat, neophodno je preći na algoritme **dinamičkog programiranja**.

Najduži zajednički podniz (*LCS*)

- Najjednostavniji ovakav algoritam je najduži zajednički podniz (*Longest Common Subsequence, LCS*), koji dozvoljava samo insercije i delecije. *Podniz* je niz koji se dobije potencijalnim brisanjem nekih elemenata originalnog niza.

- `LCS("AGCATT", "TGTCAT") = "GCAT"`
  `LCS("ATATATAT", "TATATATA") = "ATATATA"`

- $\Longrightarrow$ Optimalna poravnanja:

  ```
  A-G-CATT        -ATATATAT
  -TGTCA-T        TATATATA-
  ```

- LCS je standardni takmičarski algoritam (*IOI 2000: Palindromes*).

LCS

LCS algoritam:

- **Ulaz:** sekvence X (dužine n) i Y (dužine m).
- Definišimo $LCS(i, j)$ $(i \leq n, j \leq m)$ kao dužinu najdužeg zajedničkog podniza od prefiksa $X[1..i]$ i $Y[1..j]$. Optimalna dužina je onda $LCS(n, m)$; sam podniz se izvlači backtrack-om kroz tabelu.
- **Bazni slučajevi:** $LCS(i, 0) = LCS(0, j) = 0$.
- $$LCS(i,j) = \begin{cases} LCS(i-1, j-1) + 1 & X[i] = Y[j] \\ \max(LCS(i-1, j), LCS(i, j-1)) & X[i] \neq Y[j] \end{cases}$$
- Vremenska i memorijska složenost: $O(n \cdot m)$.

Needleman-Wunsch algoritam

Needleman-Wunsch algoritam:

- Modifikacija LCS algoritma u kojoj su dozvoljene supstitucije; dati su i "skorovi" za svaku akciju:
  - m za isti nukleotid;
  - s za supstituciju;
  - d za inserciju/deleciju.
- Optimalno poravnanje postiže najviši skor.
- Definišemo $NW(i, j)$ slično kao i $LCS(i, j)$.
- Vremenska i memorijska složenost algoritma je i dalje $O(n \cdot m)$.

Needleman-Wunsch, cont'd

- **Bazni slučajevi:** $NW(i,0) = NW(0,i) = -i \times d.$
- $$NW(i,j) = \max \begin{cases} NW(i-1,j-1) + score(i,j) \\ NW(i-1,j) - d \\ NW(i,j-1) - d \end{cases}$$
- $$score(i,j) = \begin{cases} m & X[i] = Y[j] \\ -s & X[i] \neq Y[j] \end{cases}$$
- Primetiti: $LCS(i,j)$ je $NW(i,j)$ sa parametrima:
  - $m = 1$;
  - $s = +\infty$;
  - $d = 0$.

Needleman-Wunsch, primer

`NeedlemanWunsch("CATGT", "ACGCTG")`

$$m = 2$$

$$s = d = 1$$

			A	C	G	C	T	G
		0	**1**	**2**	**3**	**4**	**5**	**6**
	0	0	-1	-2	-3	-4	-5	-6
C	**1**	-1						
A	**2**	-2						
T	**3**	-3						
G	**4**	-4						
T	**5**	-5						

Needleman-Wunsch, primer

`NeedlemanWunsch("CATGT", "ACGCTG")`

$$m = 2$$

$$s = d = 1$$

			A	C	G	C	T	G
		0	**1**	**2**	**3**	**4**	**5**	**6**
	0	0	-1	-2	-3	-4	-5	-6
C	**1**	-1	-1	1	0	-1	-2	-3
A	**2**	-2	1	0	0	-1	-2	-3
T	**3**	-3	0	0	-1	-1	1	0
G	**4**	-4	-1	-1	2	1	0	3
T	**5**	-5	-2	-2	1	1	3	2

Needleman-Wunsch, primer

`NeedlemanWunsch("CATGT", "ACGCTG")`

$$m = 2$$

$$s = d = 1$$

			A	C	G	C	T	G
		0	**1**	**2**	**3**	**4**	**5**	**6**
	0	0	-1	-2	-3	-4	-5	-6
C	**1**	-1	-1	1	0	-1	-2	-3
A	**2**	-2	1	0	0	-1	-2	-3
T	**3**	-3	0	0	-1	-1	1	0
G	**4**	-4	-1	-1	2	1	0	3
T	**5**	-5	-2	-2	1	1	3	2

```
-C-ATGT
ACGCTG-
```

Needleman-Wunsch, primer

`NeedlemanWunsch("CATGT", "ACGCTG")`

$$m = 2$$

$$s = d = 1$$

			A	C	G	C	T	G
		0	**1**	**2**	**3**	**4**	**5**	**6**
	0	0	-1	-2	-3	-4	-5	-6
C	**1**	-1	-1	1	0	-1	-2	-3
A	**2**	-2	1	0	0	-1	-2	-3
T	**3**	-3	0	0	-1	-1	1	0
G	**4**	-4	-1	-1	2	1	0	3
T	**5**	-5	-2	-2	1	1	3	2

```
-CA-TGT
ACGCTG-
```

Needleman-Wunsch, primer

`NeedlemanWunsch("CATGT", "ACGCTG")`

$$m = 2$$

$$s = d = 1$$

			A	C	G	C	T	G
		0	**1**	**2**	**3**	**4**	**5**	**6**
	0	0	-1	-2	-3	-4	-5	-6
C	**1**	-1	-1	1	0	-1	-2	-3
A	**2**	-2	1	0	0	-1	-2	-3
T	**3**	-3	0	0	-1	-1	1	0
G	**4**	-4	-1	-1	2	1	0	3
T	**5**	-5	-2	-2	1	1	3	2

```
CATG-T-
-ACGCTG
```

Smith-Waterman algoritam

Smith-Waterman algoritam:

- Modifikacija N-W algoritma koja računa optimalna *lokalna poravnanja*; **skor nikad ne pada ispod nule**.
- Definišimo $SW(i,j)$ kao optimalan skor lokalnog poravnanja koje se završava na i-toj poziciji prve i j-toj poziciji druge sekvence.
- Uglavnom rekonstruišemo sva poravnanja sa $SW(i,j) > t$.
- **Bazni slučajevi:** $SW(i,0) = SW(0,j) = 0$.
- $$SW(i,j) = \max \begin{cases} 0 \\ SW(i-1,j-1) + score(i,j) \\ SW(i-1,j) - d \\ SW(i,j-1) - d \end{cases}$$

Optimizacija memorijske složenosti

- Sve dosadašnje metode su vremenske i memorijske složenosti $O(n \cdot m)$.
- Nepraktično za dugačke sekvence $\longrightarrow$ dužina ljudskog genoma $\sim 3 \cdot 10^9$ nukleotida!
- Memorijska složenost postaje problem daleko pre vremenske!
- Ispostavlja se da možemo optimizovati memorijsku složenost na $O(n)$ bez da žrtvujemo vremensku složenost.

Ključne opservacije

- Ukoliko nas interesuje samo optimalan **skor** globalnog poravnanja (polje u donjem-desnom ćošku *Needleman-Wunsch* matrice), a ne i samo poravnanje, nije potrebno više od $O(n)$ memorije. (Zašto?)

- Definišimo funkciju $NWScore(X, Y)$ koja vraća **poslednji red** *Needleman-Wunsch* tabele za sekvence X i Y.

Hirschberg-ov algoritam

Hirschberg-ov algoritam:

- Ukoliko je $|X| \leq 2$ ili $|Y| \leq 2$, odraditi regularan *Needleman-Wunsch* algoritam nad X i Y.
- Definisati sledeće parametre:
  - $x_{mid} = \frac{|X|}{2}$;
  - $ScoreL[] = NWScore(X[1..x_{mid}], Y)$;
  - $ScoreR[] = NWScore(\text{reverse}(X[x_{mid}+1..|X|]), \text{reverse}(Y))$;
  - $y_{mid} = \underset{i=0}{\overset{m}{\text{argmax}}} (ScoreL[i] + ScoreR[m-i])$.
- Primetiti da jedno od optimalnih poravnanja sigurno prolazi kroz polje (x_{mid}, y_{mid}) u matrici, tako da sada možemo rekurzivno podeliti upit na dva dela:
  - `Hirschberg(X[1..x_mid], Y[1..y_mid]);`
  - `Hirschberg(X[x_mid+1, |X|], Y[y_mid+1, |Y|]);`

Hirschberg-ov algoritam – složenost

- Memorijska složenost *Hirschberg*-ovog algoritma je $O(n)$; uvek u memoriji pamtimo najviše dva reda *Needleman-Wunsch* matrice.
- U jednom pozivu algoritma, radimo dva N-W upita nad polovinama prve sekvence i celom drugom, što je ukupne složenosti $2 \cdot \left(\frac{n}{2} \cdot m\right) = nm$; u najgorem slučaju, ukupan broj operacija nije veći od:

$$T(n,m) \leq nm(1+\frac{1}{2}+\frac{1}{4}+\dots) \leq \frac{1}{1-\frac{1}{2}}nm = 2nm = \underline{O(n \cdot m)}$$

- Dakle, *Hirschberg*-ov algoritam uspešno snižava memorijsku složenost *Needleman-Wunsch* algoritma ne žrtvujući pri tom vremensku složenost.

Kratko skretanje – *affine gap penalty*

- Dosadašnji modeli su davali **fiksnu** cenu inserciji i deleciji *svakog nukleotida pojedinačno*.
- U praksi, insercije/delecije se uglavnom odvijaju na većim blokovima nukleotida odjednom, tako da je ova cena **neadekvatna**.
- Realnija je tzv. *afina cena*, gde dajemo "početnu" cenu d za početak neke insercije/delecije, a zatim neku (jeftiniju) cenu e za svaku susednu inserciju/deleciju: $\gamma(n) = d + (n-1)e$ za inserciju/deleciju n susednih nukleotida.

Heuristički pristup

- Dinamičko programiranje je vremenski i memorijski optimalan metod ukoliko nas zanima **optimalno** poravnanje dve DNK sekvence.
- Kao što je prethodno navedeno, vremenska složenost $O(n \cdot m)$ je **nepraktična** za velike sekvence (iako se neki algoritmi poput *Smith-Waterman*-a uspevaju poprilično dobro paralelizovati na vektorskim mašinama).
- Radi ubrzanja, primorani smo da pređemo na **heurističke algoritme**, koji će dati suboptimalna poravnanja.

BLAST

- Prva heuristika koju ćemo razmatrati je *Basic Local Alignment Search Tool (BLAST)*—trenutno **verovatno najkorišćeniji algoritam u bioinformatici**.
- Najčešće se koristi kada želimo da pretražimo poravnanja neke *query* sekvence unutar baze podataka sekvenci (npr. ukoliko želimo saznati kojoj vrsti pripada neka nepoznata sekvenca);
- Ključna pretpostavka: uglavnom nas zanimaju jedino lokalna poravnanja koja su *skoro savršena* (95+%!).
- $\Longrightarrow$ Ovakva lokalna poravnanja će verovatno imati neki podstring koji je savršeno sparen (bez mutacija)!

BLAST, cont'd

BLAST algoritam:

- Podeliti *query* sekvencu na sve podstringove dužine w (gde je w dat parametar—za DNK, $w = 12$ uglavnom).
  npr. sekvenca `"ATCG"`, $w = 2$ generiše:
  `{"AT", "TC", "CG"}`
- Pronaći sva mesta unutar baze podataka gde se neki od ovih podstringova nalazi (tako što se preračunaju lokacije svih podstringova dužine w u bazi podataka, ili koristeći npr. *Knuth-Morris-Pratt (KMP)* algoritam).
- Proširiti sve lokacije gde je došlo do poklapanja, u oba smera, sve dok ukupan skor poravnanja ne padne ispod neke definisane granice, T.
- Očekivana vremenska složenost algoritma je $O(n + m)$, gde je n dužina *query* sekvence, a m dužina baze podataka.

Problematičnosti sa *BLAST*-om

- Zbog predugačke vrednosti w, može se namestiti "nesrećan" primer koji je očigledno skoro savršeno poravnat sa *query* sekvencom, ali ga *BLAST* i dalje neće razmatrati:

```
GAGTACTCAACACCAACATTAGTGGGCAATGGAAAAT
||  |||||||||  ||||||  |  ||||||   ||||||
GAATACTCAACAGCAACATCAATGGGCAGCAGAAAAT
```

- Jedna opcija: smanjiti w (na 9 za gorenavedeni primer); ovo bi imalo negativan efekat na brzinu algoritma (jer bi daleko više poklapanja moralo da se proširuje).

- Druga opcija...

PatternHunter

- *PatternHunter* je algoritam koji je gotovo identičan *BLAST*-u, osim u jednom ključnom detalju: **nezahtevanje potpunog poklapanja podstringa u bazi podataka**.
- Traže se poklapanja koristeći (pažljivo odabrane) *šablone sa rupama (spaced seeds)*—nizovi nula i jedinica koji određuju na kojim indeksima želimo da se poklope podstring i baza podataka.
  - Na primer: 111010010100110111 — podstringovi dužine $w = 18$, ali se traži poklapanje samo na datih 11 mesta!
  - *BLAST* bi za istu količinu poklapanja koristio šablon 11111111111: generiše više podstringova!
- *PatternHunter* kombinuje nekoliko ovakvih šablona odjednom; sa samo 4 šablona dobija se algoritam slične osetljivosti kao *Smith-Waterman*, ali ≈ 3700 puta brži!

Višestruko poravnanje

- Često nas interesuje da u isto vreme poravnamo *više od dve sekvence*.
- Optimalno poravnanje k sekvenci je u opštem slučaju jako teško naći.
  - DP pristup zahteva izgradnju k-dimenzione hipermatrice (za svako polje treba proveriti $2^k - 1$ suseda);
  - Ukupna vremenska složenost: $O((2n)^k)$!
- Ponovo moramo koristiti heurističke principe.
- Popularan kao *challenge* takmičarski problem (*BubbleRun 2014: DNA Alignment*).

Progresivno poravnanje

- Većina heuristika svodi problem višestrukog poravnanja na problem poravnanja dve sekvence.
- Jedan od najpopularnijih takvih algoritama je *CLUSTAL*; ovo je algoritam *progresivnog* poravnanja, koji u svakom momentu poravnava po dva elementa (od čega *element* može biti ili **sekvenca** ili **poravnata grupa sekvenci**).
- Kada poravnavamo grupe sekvenci, računamo ukupan skor neke pozicije kao neku kombinaciju ukupnih doprinosa skoru od svake sekvence iz grupe.
- Insercije/delecije su fiksirane u konačnom poravnanju nakon što ih jednom odredimo.

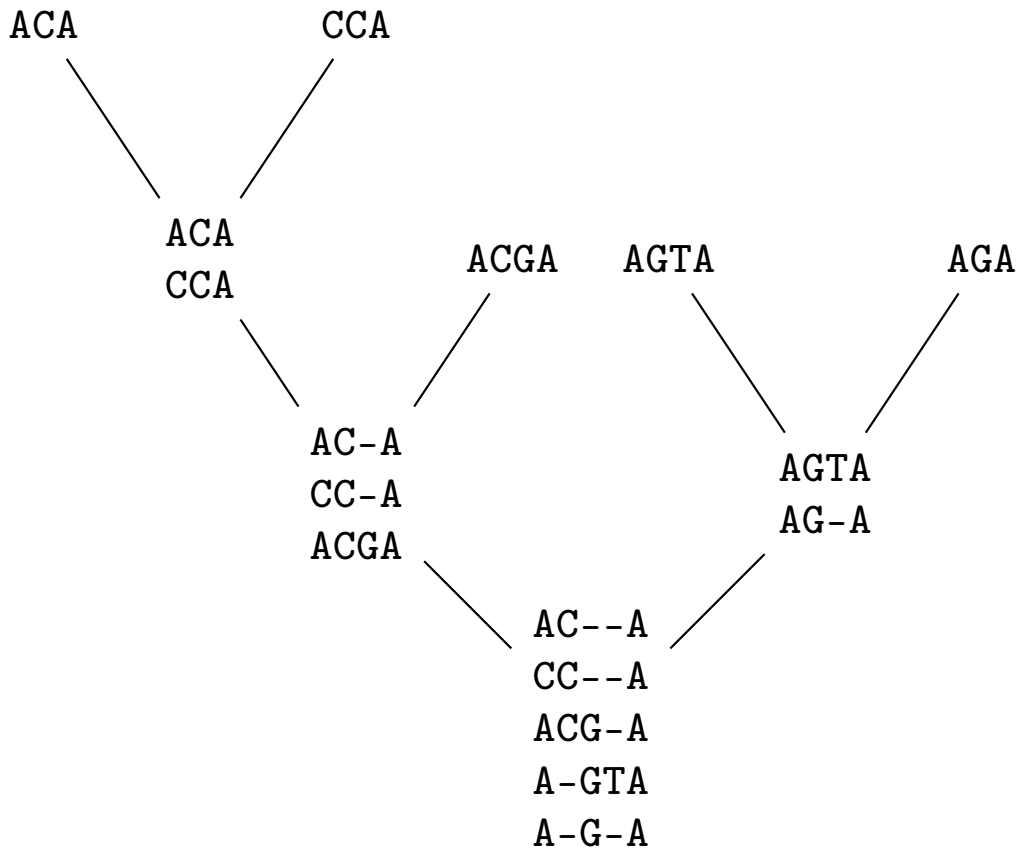
CLUSTAL

CLUSTAL algoritam:

- Izračunati skor optimalnog poravnanja za svaki par sekvenci iz ulaza (dobija se *matrica udaljenosti*);
- Upotrebiti ovu matricu da se izgradi binarno *stablo navođenja* (ili korisnik može zadati svoje stablo — u tom slučaju se preskače prvi korak);
- Progresivno graditi poravnanje koristeći se stablom, uvek poravnajući po dve elementa.
- Vremenska složenost: $O((nk)^2)$, memorijska složenost: $O(nk + n^2 + k^2)$.
- Metodama izgradnje stabla navođenja ćemo se baviti kasnije.

CLUSTAL, primer

- Primer *CLUSTAL* algoritma nad sekvencama `"ACA"(1)`, `"CCA"(2)`, `"ACGA"(3)`, `"AGTA"(4)` i `"AGA"(5)`, i stablom navođenja `(((1, 2), 3), (4, 5))`:



Kompresija DNK sekvenci

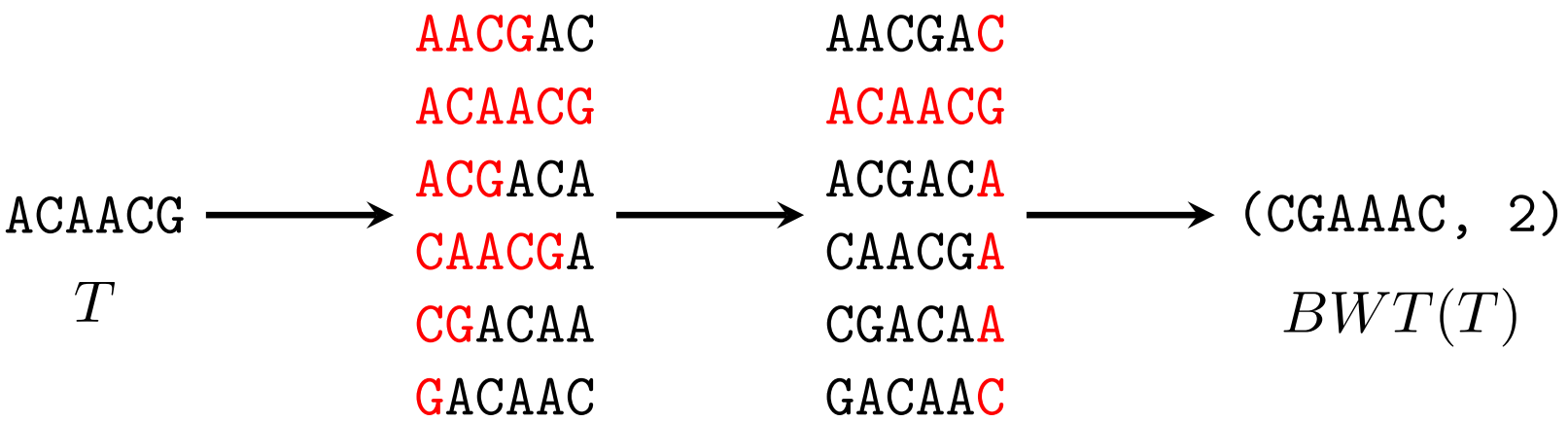
- Rad sa DNK sekvencama je često otežan **memorijskom potrošnjom** čuvanja sekvenci u radnoj memoriji.
- Posao nam umnogome olakšavaju **algoritmi za kompresiju**; kao primer ćemo navesti *Burrows-Wheeler Transform (BWT)*, transformaciju koja je centralna u `bzip2` programu za kompresiju, kao i brojnim algoritmima u bioinformatici.
- Algoritmi za kompresiju se ponekad pojavljuju unutar takmičarskih zadataka:
  (Mala Olimpijada 2005: Agenti) – traži inverziju *BWT*-a.

Burrows-Wheeler Transform (BWT)

Burrows-Wheeler Transform:

- **Ulaz:** DNK sekvenca T, dužine n, koju treba transformisati.
- Generisati svih n **cikličnih rotacija** od T, **sortirati** ih, i **poredjati u** $n \times n$ **matricu**.
- Vratiti *poslednju kolonu matrice* i *indeks reda* u kom se nalazi originalni string T.
- Vremenska složenost: naivno $O(n^2 \log n)$;
  postoje $O(n)$ metode.

BWT, primer


ACAACG
T
AACGAC
ACAACG
ACGACA
CAACGA
CGACAA
GACAAC
AACGAC
ACAACG
ACGACA
CAACGA
CGACAA
GACAAC
(CGAAAC, 2)
$BWT(T)$

Prednosti *BWT*-a

- Postoje brze ($O(n)$) metode za računanje;
- Uglavnom zbija ista slova zajedno, npr. za string `TATATATA` vraća string `TTTTAAAA`; ovakvi stringovi se daleko lakše kompresuju.
- Nema gubitka podataka; transformacija ima *inverz* (koji se takođe može efikasno izračunati).

Inverz *BWT*-a

- $BWT(T)$ nam daje poslednju kolonu matrice; prvu kolonu $C_1(T)$ možemo dobiti **sortirajući** $BWT(T)$.
- *LF Property*: i-ta pojava nekog karaktera u prvoj koloni je na istoj poziciji kao i-ta pojava tog karaktera u poslednjoj koloni!
- Definišimo $LF(i)$ kao mapiranje pozicija iz poslednje kolone na prvu kolonu, tako da važi:

$$BWT[i] = C_1[LF(i)]$$

$$BWT[i] = BWT[j] \wedge i < j \implies LF(i) < LF(j)$$

- Iterativni algoritam za računanje T:
  - Inicijalizacija: $i = x$ (gde je x indeks reda koji sadrži T);
  - Iteracija: $T = BWT[LF(i)] + T, i = LF(i)$

Izgradnja genoma

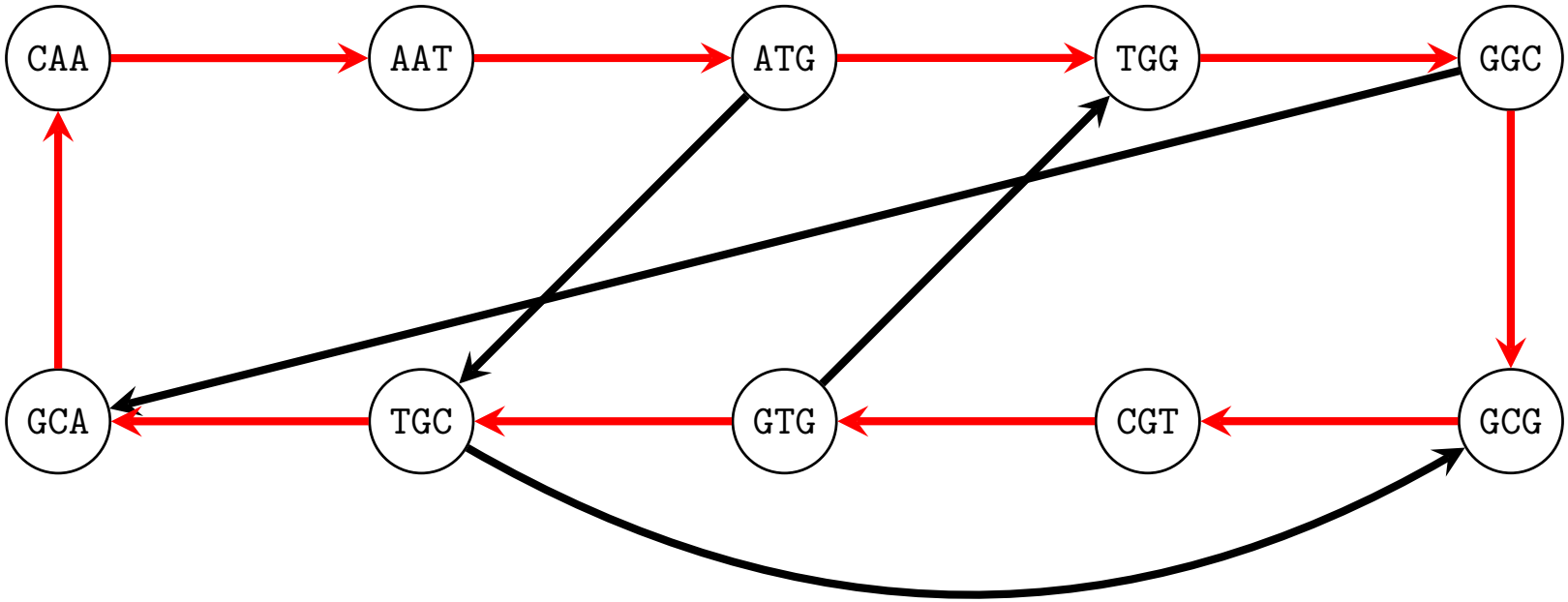
- Sada želimo da konstruišemo **celokupnu DNK sekvencu nekog organizma**.
- Vrlo je skupo odjednom izvući celu sekvencu sa velikom preciznošću; daleko jednostavnije je napraviti veliki broj ($\sim 10^7$) izvlačenja sitnih podstringova.
- Problem: kako rekonstruisati ceo genom koristeći ove sitnije podstringove?

Hamiltonov graf

- Najpre generišemo sve podstringove dužine w od pročitanih podstringova (slično kao u *BLAST* algoritmu).
- Prvih $w-1$ karaktera stringa dužine w nazvaćemo njegovim **prefiksom**, a poslednjih $w-1$ karaktera njegovim **sufiksom**. npr. prefiks od `"ATCG"` je `"ATC"`, a sufiks je `"TCG"`.
- Napravimo graf takav da svaki uočeni podstring dužine w ima jedan čvor; dva čvora spajamo usmerenom ivicom ukoliko se sufiks prvog poklapa sa prefiksom drugog; npr. `"ATCG"` $\rightarrow$ `"TCGG"`.
- Genom dobijamo iz *Hamiltonovog ciklusa* nad ovim grafom—ciklusa koji obilazi svaki **čvor** tačno jednom.
- Vremenska složenost: $O(n^2 2^n)$, *worst-case*

Hamiltonov graf, primer

Primer Hamiltonovog grafa nad podstringovima:
`"ATG", "TGG", "GGC", "GCG", "CGT", "GTG", "TGC", "GCA", "CAA", "AAT".`



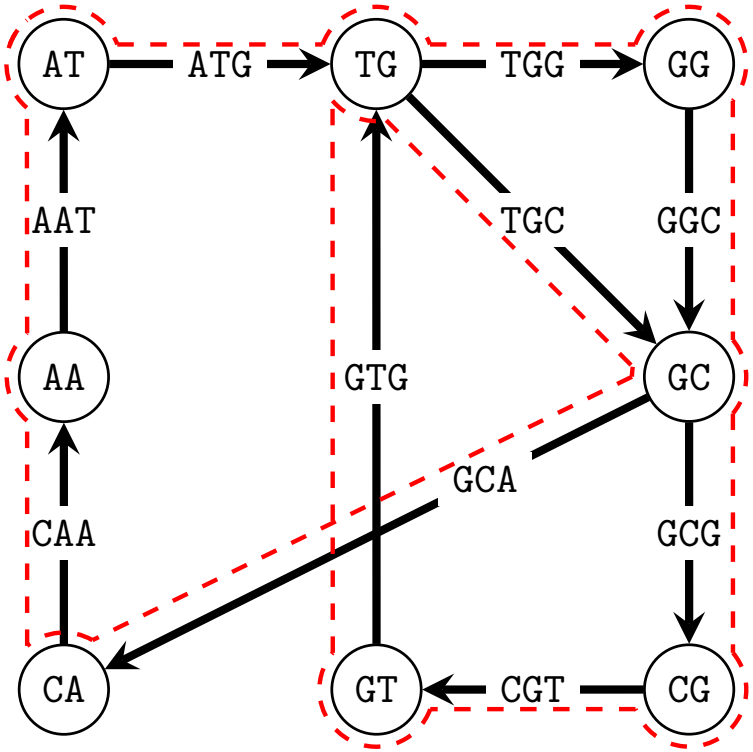
Hamiltonov ciklus daje genom `ATGGCGTGCAATG`.

de Bruijn-ov graf

- Ponovo generišemo sve podstringove dužine w od pročitanih podstringova.
- Međutim, ovoga puta čvorove grafa vezujemo za **prefikse** i **sufikse** svih ovih stringova.
- Dva čvora u ovom grafu se spajaju ivicom ukoliko postoji podstring kome je prvi čvor prefiks a drugi čvor sufiks; npr. `"ATC"` $\rightarrow$ `"TCG"` ukoliko imamo podstring `"ATCG"`.
- Genom dobijamo iz *Ojlerovog ciklusa* nad ovim grafom—ciklusa koji obilazi svaku **ivicu** tačno jednom.
- Vremenska složenost: $O(|V| + |E|)$.

de Bruijn-ov graf, primer

Primer *de Bruijn*-ovog grafa nad podstringovima:
"ATG", "TGG", "GGC", "GCG", "CGT", "GTG", "TGC", "GCA", "CAA", "AAT".



Ojlerov ciklus daje genom ATGGCGTGCAAT.

Filogenetska stabla

- Pretpostavimo da imamo DNK sekvence od nekoliko vrsta za koje sumnjamo da su *povezane* (tj. da su nastale od zajedničkog pretka).
- *Filogenetska analiza* se bavi **rekonstrukcijom** ovih evolucionih stabala.
- Problem: preci su (uglavnom) izumrle vrste, tako da ne možemo lako doći do njihovog DNK (osim fosila).
- Dva pristupa problemu:
  - **Štedljivost** (*parsimony*) – minimizacija broja mutacija;
  - **Razdaljine** (*distances*) – koristi *matricu rastojanja*, dobijenu iz međusobnih poravnanja.

Fitch-ov algoritam

Fitch-ov algoritam:

- Jednostavan algoritam koji implementira pristup štedljivosti; ulaz za algoritam je niz DNK sekvenci istih dužina n, i binarno stablo koje treba popuniti sa DNK sekvencama predaka (date DNK sekvence su listovi stabla).

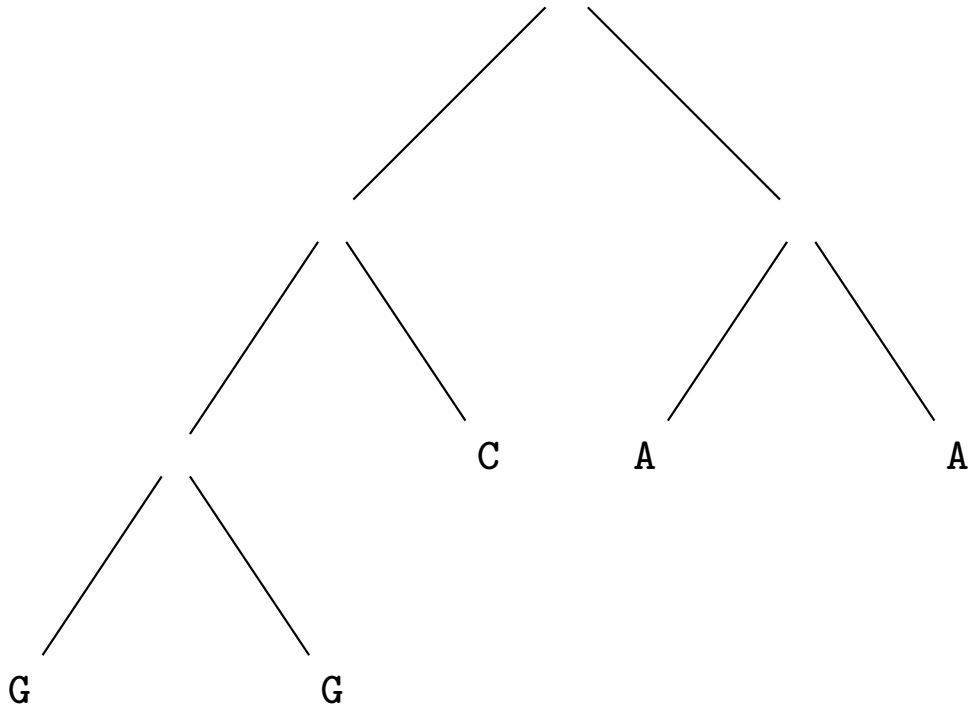
- Algoritam se odvojeno pokreće za svaku poziciju u sekvencama, tako da ćemo analizirati samo slučaj $n = 1$.

Fitch, cont'd

- *Downpass* prolaz: računanje skupa R_i, koji sadrži potencijalne vrednosti nukleotida u čvoru i:
  - Ukoliko je i list, $R_i = \{val_i\}$;
  - U suprotnom, prvo rekurzivno izračunaj R_j i R_k od dece čvora i;
  - Ukoliko $R_j \cap R_k \neq \emptyset$, $R_i = R_j \cap R_k$;
  - U suprotnom, $R_i = R_j \cup R_k$.
- *Uppass* prolaz: biranje vrednosti nukleotida r_i za svaki čvor koristeći izračunate skupove:
  - Ukoliko je i koren, $r_i = x$ tako da $x \in R_i$.
  - U suprotnom, neka je p roditelj od i;
  - Ukoliko $r_p \in R_i$, $r_i = r_p$;
  - U suprotnom, $r_i = x$ tako da $x \in R_i$.
  - Rekurzivno izračunaj r_j i r_k od dece čvora i.
- Vremenska složenost algoritma je $O(n \cdot |V|)$, gde je $|V|$ broj čvorova stabla.

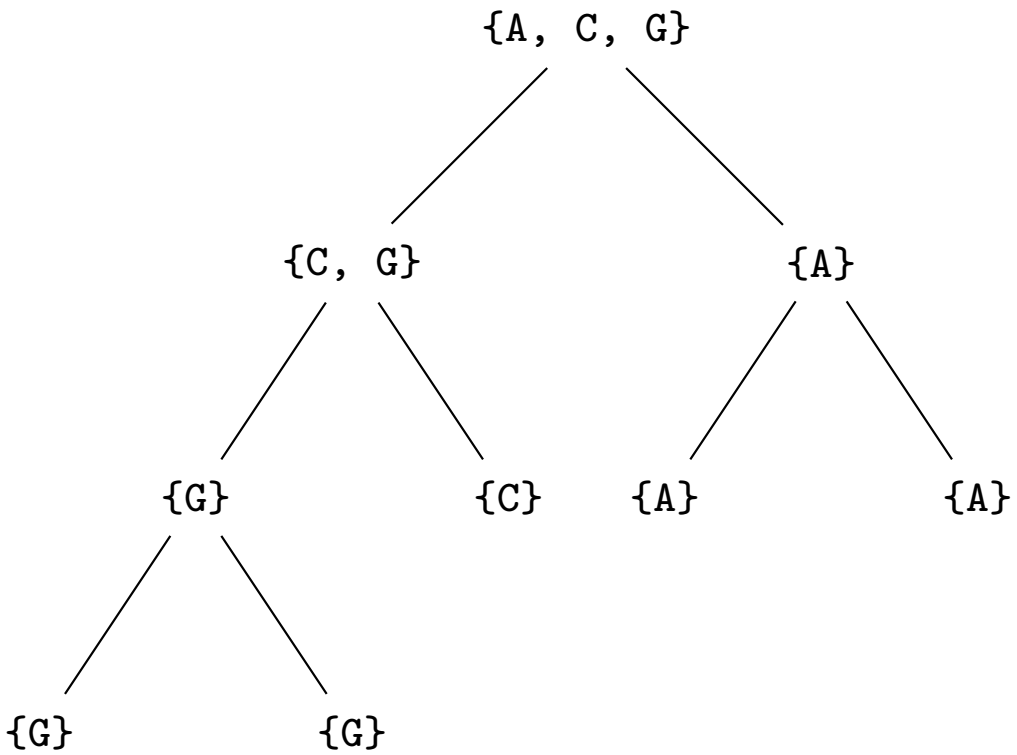
Fitch, primer

Primer *Fitch*-ovog algoritma nad nukleotidima `G, G, C, A, A` i stablom:



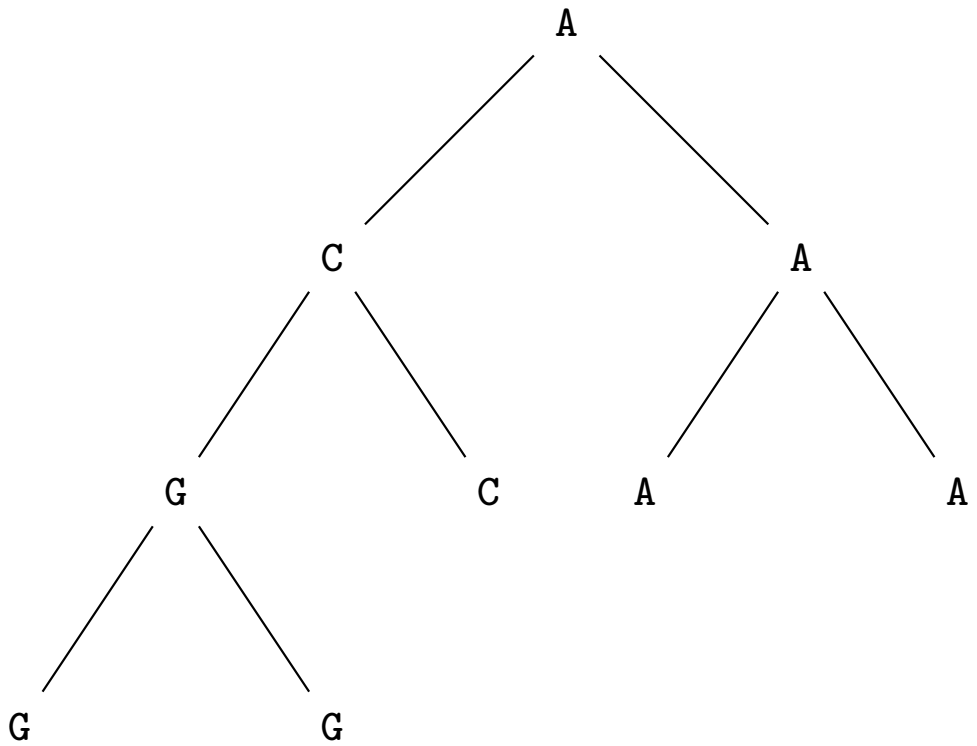
Fitch, primer

Uppass:



Fitch, primer

Downpass: (jedno od mogućih optimalnih stabala)



UPGMA algoritam

Unweighted Pair Group Method with Arithmetic mean (UPGMA):

- Algoritam koji implementira pristup rastojanja; ulaz za algoritam je *matrica udaljenosti* koja drži 'udaljenosti' između svih parova sekvenci. Izlaz je binarno stablo koje predstavlja potencijalnu strukturu zajedničkih predaka.
- *UPGMA* algoritam se često koristi za izradu *stabla navođenja* za prethodno spomenuti *CLUSTAL* algoritam.
- Algoritam je zasnovan na aritmetičkoj sredini svih udaljenosti kao metrici za razdaljinu između dva skupa, $\mathcal{A}$ i $\mathcal{B}$:

$$d(\mathcal{A}, \mathcal{B}) = \frac{1}{|\mathcal{A}| \cdot |\mathcal{B}|} \sum_{x \in \mathcal{A}} \sum_{y \in \mathcal{B}} d(x, y)$$

gde je $d(x, y)$ udaljenost sekvenci x i y.

UPGMA, cont'd

- Na početku, svaka sekvenca x je u svom (jediničnom) skupu $C_x = \{x\}$, i svaki skup je vezan za neki list izlaznog stabla;
- U svakoj iteraciji, biramo C_i i C_j tako da je $d(C_i, C_j)$ *minimalno*.
- Definišemo novi skup $C_k = C_i \cup C_j$ i računamo njegova rastojanja od svih preostalih skupova. Takođe, C_k vezujemo za novi čvor u stablu, koji je roditelj od čvorova vezanih za C_i i C_j.
- Brišemo skupove C_i i C_j.
- Iteriramo dok ne ostane samo jedan skup (*koren stabla*).
- Najefikasnija poznata implementacija ima vremensku složenost $O(n^2)$.

Napredniji zadaci za vežbu

Zanimljivi takmičarski zadaci inspirisani bioinformatičkim algoritmima (otprilike sortirani po težini):

- *ACM Mid Central Regionals 1995: DNA Translation*;
- *11th Iran Internet Contest: RNA Molecules*;
- *TUD Programming Contest 2004: DNA Laboratory*;
- *BubbleCup Quals1 2012 (SPOJ): Segment Flip*;
- *BubbleCup Quals2 2014 (SPOJ): Soccer Choreography*.