SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 2321

SWIG - Poravnanje struktura korištenjem iterativne primjene Smith-Waterman algoritma

Bruno Rahle

Zagreb, lipanj 2012.

Umjesto ove stranice umetnite izvornik Vašeg rada.
Da bi ste uklonili ovu stranicu obrišite naredbu `\izvornik`.

Zahvaljujem se obitelji, mentoru i kolegama.

SADRŽAJ

POPIS SLIKA

POPIS TABLICA

1. Uvod

Živimo u doba kada velikih promjena. Današnjim generacijama mladih nezamisliv je život bez Interneta i društvenih mreža. Internet, a kamoli društvene mreže, bili su nezamislivi njihovim roditeljima dok su bili mladi. Računala su bila (većinom i ostala) neshvatljiva čuda djedovima i bakama, dok je njihovim roditeljima čak i električna energija strana.

Nije samo računarstvo doživjelo ogromne promjene. Tek pred nešto više od stotinu godinu izumljen je automobil. Danas gotovo da ne postoji obitelj koja ne posjeduje barem jedan. Slično vrijedi i za perilicu rublja i televizore.

Još je veliki hrvatski pjesnik Petar Preradović u pjesmi "Mujezin" zapisao "Stalna na tom svijetu samo mijena jest." Ono što danas smatramo znanstvenom fantastikom za nekoliko bi godina moglo postati stvarnost. Kako se povećava ljudsko znanje, tako se povećava i broj pitanja na koje ljudi traže odgovore. Često problemi s kojima se znanost susreće prelaze granice isključivo jedne discipline. Zbog toga je postao običaj da ljudi različitih profila sudjeluju na istom istraživanju.

Postoji čitav niz problema iz biologije koji se danas više ili manje efikasno rješavaju uz pomoć kompjutera. Ti problemi spadaju u područje koje nazivamo bioinformatikom. Jedan od njih je i problem poravnanje proteina (koji su zapravo nizovi molekula). Za neka dva proteina, zanima nas koliko su, prema nekoj mjeri, slični. Mjera koju ćemo u ovom radu koristiti je fizička udaljenost između dvije molekule. Da bismo malo "zapaprili" stvari, dopustit ćemo jednostavne transformacije - rotaciju i translaciju - jednog od proteina. Taj problem dalje možemo apstrahirati na problem poravnanja bilo kakvih struktura.

Poglavlje 2 govorit će općenito o problemu s kojim se susrećemo i načinu na koji smo mu pristupili. Opisat će ideju našeg rješenja i koje novosti ono donosi u odnosu na dosadašnja.

Poglavlje 3 govorit će o načinu rješavanja problema pronalaska poravnanja. Prvo ćemo krenuti od algoritma za globalno poravnanje, potom ćemo opisati algoritam za traženje lokalnog poravnanja, te ćemo se baviti njihovim optimizacijama.

Poglavlje 4 govorit će o algoritmu simuliranog kaljenja i načinu na koji smo ga prilagodili za naše specifične potrebe, dok ćemo Poglavlje 5 iskoristiti za lagani uvod u CUDA tehnologiju te njene prednosti i mane.

Poglavlje 6 će nešto detaljnije opisati specifičnosti implementacije te kako su elementi

povezani. Pozabavit ćemo se rješenjima koje nam omogućuju da upregnemo snagu masivno paralelnih čipova.

Poglavlje 7 prikazat će dobivene rezultate i kako se naše rješenje eventualno može poboljšati, dok ćemo konačnu analizu napraviti ostaviti za Poglavlje 8.

2. Poravnavanje struktura

U ovom radu bavit ćemo se rješavanjem slijedećeg problema:

Zadana su nam dva proteina, tj. pozicije u prostoru svakog od molekula koje protein sadrži. Zanimaju nas samo atomi koji se nalaze u okosnici aminokiseline te od dva proteina napravimo dva niza takvih atoma. Želimo jednog od njih transformirati koristeći samo translacije i rotacije tako da se što je moguće više preklapa s drugim. Protein koji ćemo rotirati zvat ćemo protein B, a protein A bit će onaj s kojim ga želimo preklopiti.

Kao rješenje želimo dobiti preklapanje koje će poravnati neki podniz elemenata iz proteina A i B i koje će biti maksimalno za ta dva proteina. Zbog toga nas zanima i rekonstrukcija rješenja.

Da bismo dobili rješenje, koristit ćemo simulirano kaljenje zajedno sa Smith-Watermanovim algoritmom za proračun energije. Taj par algoritama trebao bi moći u "pristojnom" vremenu pronaći neko poravnanje koje je relativno blisko optimalnom.

Ovaj problem riješen je mnogo već mnogo puta - postoje gotovi alati poput CE-a (Shindyalov i Bourne (1998)), FATCAT-a (Ye i Godzik (2003)), DALI-a (Holm et al. (1992)) i MUSTANG-a (Konagurthu et al. (2006)).

Inovativnost ovog rada temelji se na tome da ćemo kao arhitekturu koristi GPU, tj. grafičku procesnu jedinicu. Prednost GPU arhitekture u odnosu na CPU jest to što je GPU masivno paralelan, što znači da imamo velik broj dretvi (broje se u stotinama!) koje nam omogućavaju da stvari rješavamo paralelno. Najzrelija tehnologija koja nudi korištenje grafičkih kartica u svrhu procesiranja podataka je CUDA. Unatoč nekim ograničenjima (npr. radi samo na Nvidijinim grafičkim karticama), odlučili smo koristiti upravo nju za potrebe implementacije rješenja ovog problema. Da bismo upregnuli svu moć koju nam taj čip za masivno paralelno računanje daje, nećemo uspoređivati samo jedan po jedan niz, već ćemo to raditi na više njih paralelno.

3. Smith-Watermanov algoritam

Smith-Watermanov algoritam služi nam da bismo pronašli lokalno poravnanje. Osmislili su ga Temple F. Smith i Michael S. Waterman (Smith i Waterman, 1981). Temelji se na Needleman-Wunschevom algoritmu (Needleman i Wunsch, 1970) te i sam spada u kategoriju algoritama dinamičkog programiranja. Glavna razlika između ta dva algoritma jest što Needleman-Wunschov algoritam pronalazi globalno poravnanje. Unatoč toj razlici, ulaz i izlaz oba algoritma možemo definirati na jednak način.

3.1. Ulazni i izlazni podaci

3.1.1. Ulazni podaci

1. Dva niza (A i B) proteina ili nukleotida. Zbog jednostavnosti, pretpostavit ćemo da su to nukleotidi iz DNK - adenin (A), timin (T), gvanin (G) i citozin (C). U primjeru ćemo koristiti $A = "ATGCCGTA"$ i $B = "TGCACTA"$. Dužinu niza A označit ćemo s N, a dužinu niza B s M.

2. Supstitucijska matrica S, koja nam daje bodove koje dobijemo kada jedan nukletoid preklopimo s drugim. U našem će slučaju imati dimenzije 4×4, budući da ćemo razmatrati slučaj kada imamo samo četiri nukleotida. U principu će na dijagonali imati pozitivne brojeve, a na ostalim poljima negativne. To znači da nam se najviše isplati preklapati nukletide istog tipa, jer za to dobivamo bodove, a inače ih gubimo. Tablica 3.1 prikazuje jednu takvu matricu.

	A	C	G	T
A	10	-3	-9	-1
C	-5	8	-8	-7
G	-5	-4	7	-5
T	-4	-11	-8	9

Tablica 3.1: Primjer supstitucijske matrice koju koristimo

3. Negativan broj d, koji označava bodove koje dobijemo (tj. izgubimo) kada nukleotid preklopimo s prazninom. U primjeru ćemo koristiti $d = -5$.

3.1.2. Izlazni podaci

1. Broj H, ocjena najboljeg globalnog poravnanja.

2. Dva nova niza jednake dužine, A' i B', nastala ubacivanjem praznina (označenih najčešće sa '-') u nizove A i B koja predstavljaju najbolje pronađeno poravnanje.

3.2. Needleman-Wunschov algoritam

Algoritam, kao što je već rečeno, traži globalno poravnanje. To znači da se svi članovi ulaznih nizova moraju poravnati. Dopuštene operacije kada tražimo poravnanje su preklapanje s elementom iz suprotnog niza i ubacivanje praznina u neki od nizova. Sve parove elemenata u dobivenom preklapanju bodujemo i na osnovu te ocjene određujemo sličnost nizova. Konačan rezultat ovog algoritma jest poravnanje koje maksimizira takvu ocjenu, tj. daje maksimalno globalno poravnanje.

Zanimljivost je da je to prvi algoritam dinamičkog programiranja ikada primijenjen u bioinformatici.

3.2.1. Algoritam

Neka nam matrica F služi za računanje poravnanja. Tada će nam $F_{i,j}$ označavati maksimalan broj bodova koje možemo dobiti kada poravnamo prvih i članova niza A i prvih j članova niza B. $F_{i,j}$ možemo računati rekurzijom na slijedeći način:

$$F_{i,j} = \begin{cases} 0 & \text{ako je } i = 0 \text{ i } j = 0 \\ F_{i-1,j} + d & \text{ako je } i > 0 \text{ i } j = 0 \\ F_{i,j-1} + d & \text{ako je } i = 0 \text{ i } j > 0 \\ max \begin{pmatrix} F_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \\ F_{i-1,j} + d \\ F_{i,j-1} + d \end{pmatrix} & \text{ako je } i > 0 \text{ i } j > 0 \end{cases}$$

U $F_{N,M}$ će nam stoga pisati maksimalno globalno poravnanje. Primijetite da u niti jednom slučaju nećemo poravnati dvije praznine. Ako bismo to učinili, samo bismo izgubili bodove, budući da je d nužno negativan broj.

Da bismo znali rekonstruirati rješenje, koristit ćemo matricu R. U polju $R_{i,j}$ pisat će koje smo polje matrice F koristili da bi došli u polje $F_{i,j}$. Kako su jedine mogućnosti $F_{i-1,j}$,

$F_{i,j-1}$, $F_{i-1,j-1}$ i da nismo došli iz nikojeg polja (to vrijedi jedino za polje $F_{0,0}$), koristit ćemo redom oznake A, B, O i X.

$$R_{i,j} = \begin{cases} X & \text{ako je } \begin{pmatrix} i = 0 \\ j = 0 \end{pmatrix} \\ O & \text{ako je } \begin{pmatrix} i > 0 \\ j > 0 \\ F_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \geq F_{i-1,j} + d \\ F_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \geq F_{i,j-1} + d \end{pmatrix} \\ A & \text{ako je } \begin{pmatrix} i > 0 \\ j = 0 \end{pmatrix} \text{ili } \begin{pmatrix} i > 0 \\ j > 0 \\ F_{i-1,j} + d > F_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \\ F_{i-1,j} + d \geq F_{i,j-1} + d \end{pmatrix} \\ B & \text{ako je } \begin{pmatrix} i = 0 \\ j > 0 \end{pmatrix} \text{ili } \begin{pmatrix} i > 0 \\ j > 0 \\ F_{i,j-1} + d > F_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \\ F_{i,j-1} + d > F_{i-1,j} + d \end{pmatrix} \end{cases}$$

Rekonstrukciju provodimo tako krenemo iz polja $R_{N,M}$ i krećemo se po matrici unazad dok ne dođemo do polja na kojem piše X, tj. $R_{0,0}$. Ako na polju pročitamo O, pomičemo se po dijagonali, tj. u izlazni niz spremimo par (A_{i-1}, B_{j-1}) te smanjimo i i i j. Ako pročitamo A, spremamo par $(A_{i-1}, -)$ te smanjimo samo i za jedan. U slučaju da pročitamo B, spremamo par $(-, B_{j-1})$ te smanjujemo j za jedan. Ako smo pročitali X, došli smo do kraja i generirali smo izlazni niz, ali u obrnutom redoslijedu.

3.2.2. Primjer

Tablica 3.2 i Tablica 3.3 sadrže potpuno izračunate matrice F i R. Iz tih podataka lagano je napraviti potpunu rekonstrukciju (polja označena sivom bojom). Stoga zaključujemo da je traženo globalno poravnanje A' = "ATGCCGTA" i B' = "-TGCACTA".

3.2.3. Analiza složenosti

Trivijalno je vidljivo da su memorijska i vremenska složenost opisanog algoritma jednake $O(NM)$.

	-	T	G	C	A	C	T	A
-	0	-5	-10	-15	-20	-25	-30	-35
A	-5	-1	-6	-11	-5	-10	-15	-20
T	-10	4	-1	-6	-10	-15	-1	-6
G	-15	-1	11	6	1	-4	-6	-6
C	-20	-6	6	19	14	9	4	-1
C	-25	-11	1	14	14	22	17	12
G	-30	-16	-4	9	9	17	17	12
T	-35	-21	-9	4	5	12	26	21
A	-40	-26	-14	-1	14	9	21	36

Tablica 3.2: Potpuno izračunata matrica F za Needleman-Wunschov algoritam

	-	T	G	C	A	C	T	A
-	X	B	B	B	B	B	B	B
A	A	O	B	B	O	B	B	O
T	A	O	B	B	A	A	O	B
G	A	A	O	B	B	B	A	O
C	A	A	A	O	B	O	B	B
C	A	A	A	O	O	O	B	B
G	A	A	O	A	O	A	O	O
T	A	O	A	A	O	A	O	B
A	A	A	A	A	O	B	A	O

Tablica 3.3: Potpuno izračunata matrica R za Needleman-Wunschov algoritam

3.3. Smith-Watermanov algoritam

Razlika njega i prethodno opisanog Needleman-Wunschevog algoritma jest u tome što ovaj algoritam traži najbolje lokalno poravnanje. To znači da ne koristi nužno cijele nizove proteina ili nukleotida već samo najsličnije uzastopne podnizove. U praksi se koriste nešto poboljšane verzije ovog algoritma.

3.3.1. Algoritam

U ovom ćemo algoritmu koristiti matricu H za računanje poravnanja. Za razliku od Needleman-Wunschevog algoritma, $H_{i,j}$ ovaj će put označavati rezultat najboljeg lokalnog poravnanja koje koristi A_{i-1} i B_{j-1}. Matricu ćemo popuniti na slijedeći način:

$$H_{i,j} = \begin{cases} 0 & \text{ako je } i = 0 \text{ ili } j = 0 \\ \max \begin{pmatrix} 0 \\ H_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \\ H_{i-1,j} + d \\ H_{i,j-1} + d \end{pmatrix} & \text{ako je } i > 0 \text{ i } j > 0 \end{cases}$$

Primijetite bitnu razliku u odnosu na Needleman-Wunschev algoritam - u ovom slučaju matrica H neće sadržavati negativne brojeve. Rješenje, tj. najbolje lokalno poravnanje više neće pisati na polju $H_{N,M}$. Ono će biti na polju na kojem se nalazi najveći broj u matrici, a to je a njega ćemo nazvati $H_{t,u}$.

Da bismo znali rekonstruirati takvo lokalno poravnanje, koristit ćemo matricu R koju gradimo na sličan način kao i kod prethodnog algoritma:

$$R_{i,j} = \begin{cases} X & \text{ako je} H_{i,j} = 0 \\ O & \text{ako je} \begin{pmatrix} i > 0 \\ j > 0 \\ H_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \geq H_{i-1,j} + d \\ H_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \geq H_{i,j-1} + d \end{pmatrix} \\ A & \text{ako je} \begin{pmatrix} i > 0 \\ j = 0 \end{pmatrix} \text{ili} \begin{pmatrix} i > 0 \\ j > 0 \\ H_{i-1,j} + d > H_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \\ H_{i-1,j} + d \geq H_{i,j-1} + d \end{pmatrix} \\ B & \text{ako je} \begin{pmatrix} i = 0 \\ j > 0 \end{pmatrix} \text{ili} \begin{pmatrix} i > 0 \\ j > 0 \\ H_{i,j-1} + d > H_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \\ H_{i,j-1} + d > H_{i-1,j} + d \end{pmatrix} \end{cases}$$

	-	T	G	C	A	C	T	A
-	0	0	0	0	0	0	0	0
A	0	0	0	0	10	5	0	10
T	0	9	4	0	5	0	14	9
G	0	4	16	11	6	1	9	9
C	0	0	11	24	19	14	9	4
C	0	0	6	19	19	27	22	17
G	0	0	7	14	14	22	22	17
T	0	9	4	9	10	17	31	26
A	0	4	0	4	19	14	26	41

Tablica 3.4: Potpuno izračunata matrica H za Smith-Watermanov algoritam

	-	T	G	C	A	C	T	A
-	X	X	X	X	X	X	X	X
A	X	X	X	X	O	B	B	O
T	X	O	B	X	A	A	O	B
G	X	A	O	B	B	O	A	O
C	X	X	A	O	B	O	B	O
C	X	X	A	O	O	O	B	B
G	X	X	O	A	O	A	O	O
T	X	O	B	A	O	A	O	B
A	X	A	O	A	O	B	A	O

Tablica 3.5: Potpuno izračunata matrica R za Smith-Watermanov algoritam

Rekonstrukcija se, slično kao i kod Needleman-Wunschevog algoritma, izvodi tako da krenemo iz polja $R_{t,u}$ i krećemo se po matrici R dok ne dođemo do polja na kojem piše X koje ovaj put ne mora nužno biti polje $R_{0,0}$. Dobiveni niz okrenemo i dobit ćemo traženo poravnanje.

3.3.2. Primjer

Tablica 3.4 i Tablica 3.5 prikazuju potpuno izračunate matrice H i R nakon izvršavanja Smith-Watermanovog algoritma. Iz njih je lako očitati traženo poravnanje: $A' =$ "TGC-CGTA" i $B' =$ "TGCAC-TA".

3.3.3. Analiza složenosti

Trivijalno je vidljivo da su memorijska i vremenska složenost i ovog algoritma jednake $O(NM)$.

3.4. Procjepi

Do sada smo razmatrali samo slučaj kada je cijena otvaranja procjepa i njegova proširivanja jednaka (d). Taj slučaj nazivamo *linearnom ocjenom procjepa*, budući da, ako je k njegova dužina, dk će biti njegova cijena. U praksi se primjenjuju još dva načina ocjenjivana.

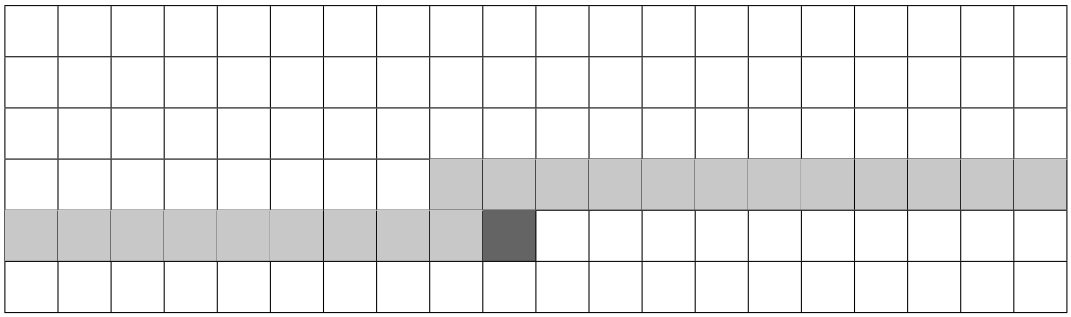
Jedan je da za svaki procjep, bez obzira na njegovu dužinu, platimo fiksnu cijenu (c), a nazivamo ga *konstantnom ocjenom procjepa*.

Drugi je kombinacija prethodna dva načina: za svaki procjep plaćamo fiksnu cijenu (c), ali za njegovo produženje plaćamo neku drugu cijenu (d). Tu ideju prvi je uveo Gotoh, 1982. Takvu funkciju ocjene nazivamo *Afinom ocjenom procjepa*. Prema tome, za procjep dužine k, platit ćemo cijenu jednaku $c + (k-1)d$. Kako je empirijski pokazano da je ta funkcija u biti parabola, ovakva je ocjena ujedno i najpreciznija. Nažalost, ona otežava računanje matrice H (i R). Problem je najlakše riješiti tako da uvedemo dvije nove matrice H^A i H^B koje će nam pomoći u računanju cijene procjepa. Matrice ćemo onda računati na slijedeći način:

$$H_{i,j} = \begin{cases} 0 & \text{ako je } i = 0 \text{ ili } j = 0 \\ \max \begin{pmatrix} 0 \\ H_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \\ H^A_{i-1,j} + c \\ H^B_{i,j-1} + c \end{pmatrix} & \text{ako je } i > 0 \text{ i } j > 0 \end{cases}$$

$$H^A_{i,j} = \begin{cases} 0 & \text{ako je } i = 0 \text{ ili } j = 0 \\ \max \begin{pmatrix} 0 \\ H_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \\ H^A_{i-1,j} + d \\ H^B_{i,j-1} + c \end{pmatrix} & \text{ako je } i > 0 \text{ i } j > 0 \end{cases}$$

$$H^B_{i,j} = \begin{cases} 0 & \text{ako je } i = 0 \text{ ili } j = 0 \\ \max \begin{pmatrix} 0 \\ H_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \\ H^A_{i-1,j} + c \\ H^B_{i,j-1} + d \end{pmatrix} & \text{ako je } i > 0 \text{ i } j > 0 \end{cases}$$



Tablica 3.6: Pod pretpostavkom da polja računamo iterativno po recima, matrica prikazuje koja polja moramo čuvati u memoriji (svjetlo-siva boja) da bismo mogli izračunati preostala. Polje koje trenutno računamo pobojamo je tamno-sivom bojom. U praksi ćemo, zbog jednostavnosti, čuvati cijeli prošli redak.

Rekonstrukcijska matrica vrlo je slična onoj u izvorno opisanom algoritmu, a kako ćemo u idućem potpoglavlju maknuti potrebu za njom, nećemo je posebno navoditi.

3.5. Memorijska optimizacija

S obzirom da nizovi nukleotida mogu sadržavati i do nekoliko milijuna elemenata, matrica dimenzija NM fizički ne stane u memoriju. Međutim, čim primijetimo da za potrebe generiranja matrice H (a ujedno i H^A odnosno H^B) koristimo samo polja iz trenutnog i prethodnog retka, postaje jasno da nam ostali reci koje smo obradili ranije njih nisu potrebni. Stoga možemo pamtiti samo ta dva retka i time korištenu memoriju za proračun matrice H smanjimo na samo $O(\min(N, M))$. Tablica 3.6 prikazuje koja polja minimalno moramo pamtiti. Ipak, problem se javlja kada trebamo rekonstruirati rješenje pomoću matrice R, budući da se prilikom rekonstrukcije potencijalno vraćamo se kroz sva njena polja. U nastavku ćemo opisati rješenje tog problema.

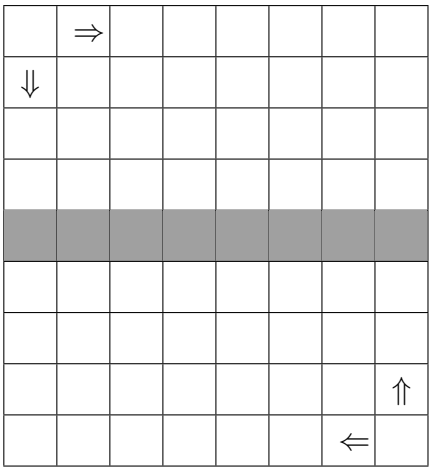
3.5.1. Hirschbergov algoritam

Dan Hirschberg osmislio je algoritam koji rješava problem rekonstrukcije globalnog poravnanja u $O(NM)$ vremena koristeći $O(\min(N, M))$ memorije (Hirschberg, 1975). Njegova ideja temeljena je na metodi *podijeli-pa-vladaj*.

No, prije nego je objasnimo, promotrimo što će se dogoditi s globalnim poravnanjem ako okrenemo nizove A i B. Okrenute nizove nazivat ćemo A^R i B^R, a matrice F^R i R^R. Očito je da se ocjena poravnanja neće promijeniti, a prilikom rekonstrukcije dobit ćemo obrnuti niz. Uz sitne modifikacije, možemo izračunati matricu F^R bez da fizički okrećemo nizove.

Hirschbergov algoritam radi na slijedeći način:

1. Bez smanjenja općenitosti, možemo pretpostaviti da je niz A duži od niza B. Neka



Tablica 3.7: Računanje Smith-Watermanovog algoritma na dvije polovice niza ($A_{0..p}$ i $A^R_{p..N}$). Svijetlo siva linija predstavlja red p, prvi odnosno posljednji red za koji računamo lokalno poravnanje.

nam p označava polovicu dužine niza A, zaokruženu na dolje. Podijelimo niz A na dvije polovice, dužine p.

2. Na obje polovice niza ($A_{0..p}$ i $A^R_{p..N}$) pokrenemo gore opisani algoritam za pronalaženje lokalnog poravnanja koje koristi $O(\min(N, M))$ memorije, ali bez rekonstrukcije. Nad drugom polovicom niza radimo algoritam za obrnuto nizove, tako da je za obje polovice posljednji izračunati red onaj odabran u prethodnom koraku.

3. Definirajmo funkciju $\phi(j)$ kao sumu lokalnih poravnanja obje matrice u redu p, tj. kao:

$$\phi(j) = F_{p,j} + F^R_{p,j}$$

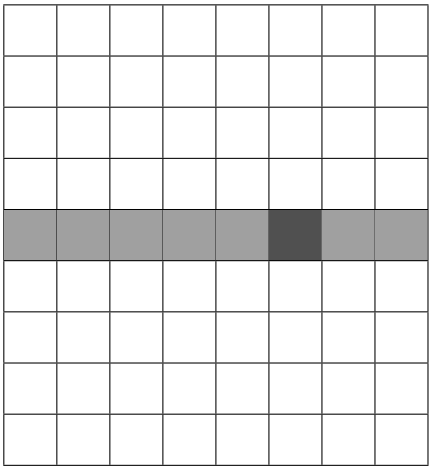
Barem će jedan član globalnog poravnanja, biti u retku p, a Hirschberg je pokazao da će to biti onaj u stupcu j za kojeg je $\phi(j)$ najveći. Time smo pronašli jedan član koji je sigurno u globalnom poravnanju.

4. S obzirom na svojstva globalnog poravnanja, intuitivno je jasno da polja koja se nalaze gore-desno i dolje-lijevo od pronađenog polja sigurno nisu u točnom globalnom poravnanju, stoga ta polja možemo izbaciti iz daljnjeg razmatranja. Ako rekurzivno primijenimo algoritam na parove podnizova $A_{0..p}$ i $B_{0,j}$ te $A_{p..N}$ i $B_{j..M}$, možemo ponavljajući postupak napraviti rekonstrukciju cijelog niza.

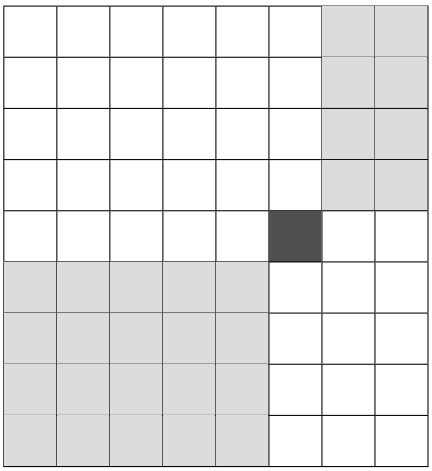
Ono što možda nije jasno jest vremenska složenost gore opisanog algoritma. Pokazuje se da pri odbacivanju polja odbacimo u pravilu polovicu trenutno razmatranih polja. Prema tome, broj operacija koje napravimo možemo napisati u obliku geometrijskog niza:

$$NM + NM/2 + NM/4 + ... < 2NM \in O(NM)$$

Time dolazimo do zaključka da je složenost i dalje ostala jednaka do razlike u konstantni, ali je znatno smanjena količina iskorištene memorije, stoga se ova ušteda isplatiti koristiti.



Tablica 3.8: Nakon što smo završili s računanjem poravnanja na oba dijela niza, trebamo pronaći polje u kojem je $\phi(j)$ maksimalno. To polje označeno je tamno-sivom bojom.



Tablica 3.9: Budući da smo sigurni da polja gore-desno i dolje-lijevo nisu u traženom globalnom poravnanju, njih ne trebamo dalje promatrati. Na slici su označena svijetlo sivom bojom.

3.6. Odstranjivanje supstitucijske matrice

U ovom ćemo radu umjesto supstitucijske matrice za usporedbu dva elementa koristiti fizičku udaljenost između dva atoma. Stoga će nizovi A i B biti zapravo nizovi koordinata iz kojih ćemo moći za svaki par elemenata izračunati koordinate.

Neka nam d označava udaljenost između dva atoma. Želimo napraviti funkciju koja će za dvije molekule koje su fizički bliske vratiti pozitivnu vrijednost, a za one koje su udaljene negativnu. Nadalje, odlično bi bilo da skok između jednake razlike u udaljenostima u početku budu malen, ali što je veća udaljenost, da time i on raste. To nas navodi na negativnu eksponencijalnu funkciju. Primjer jedne takve funkcije je $d_1 - e^{d/d_0}$, gdje su d_1 i d_0 konstante. U praksi su se $d_0 = 10$ i $d_1 = 1000$ pokazale kao dobar izbor.

4. Algoritam simuliranog kaljenja

Algoritam simuliranog kaljenja generička je metoda bazirana na vjerojatnosti koja traži ekstrem neke funkcije u velikom prostoru traženja. Zbog toga što nam ne garantira da će pronaći najbolje rješenje, obično se koristi kada nam je želimo dobiti rješenje koje je prihvatljivo u relativno kratkom vremenu.

Inspiracija za algoritam dolazi iz metalurgije. Metali se prvo grubo obrađuju na visokoj temperaturi, a, kako se hlade, tako obrada postaje sve finija i finija. Drugim riječima, *željezo se kuje dok je vruće*, kao što prikazuje Slika 4.1. Taj proces pokušavamo simulirati u ovom algoritmu.

Slika 4.1: Kovanje željeza

4.1. Algoritam

Uvedimo nekoliko definicija:

- S - trenutno stanje.
- S' - stanje koje je susjedno trenutnom.

– $E(s)$ - funkcija koja računa energiju nekog stanja. Želimo pronaći stanje s najmanjom (ili najvećom) energijom.

– S_n - najbolje pronađeno stanje, tj. stanje u kojem je energija najmanja.

– e - energija trenutnog stanja, tj. $E(S)$.

– t - vrijeme proteklo od početka pokusa.

– T - trenutna temperatura. Definiramo je kao $T_0 T_1^t$, gdje je T_0 početna temperatura, a T_1 faktor koji govori koliko se temperatura brzo smanjuje.

– $P(e, e', T)$ - vjerojatnost da prijeđemo u stanje koje ima energiju e' iz stanja koje ima energiju e u trenutku T.

Algoritam se sastoji od ponavljanja slijedećeg niza operacija dok nismo dobili stanje koje ima traženu energiju ili dok vrijeme nije isteklo.

1. Odaberi stanje S' koje je susjedno stanju S.
2. Izračunaj energiju novog stanja $e' = E(S')$.
3. Provjeri ima li novo stanje veću energiju od najboljeg do sada pronađenog stanja. Ako ima, onda je $S_n = S'$.
4. Izračunaj vjerojatnost napredovanja u stanje S': $p = P(e, e', T)$.
5. Odaberi nasumičan broj iz intervala $[0, 1]$ i ako je manji od p, pomakni trenutno stanje u S'.
6. Povećaj vrijeme t.

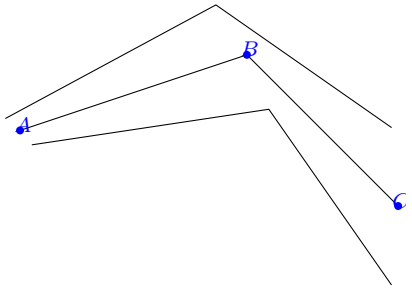
4.1.1. Odabir susjeda

U našem slučaju, odabir susjeda radimo tako uzmemo koordinate atoma te im translatiramo sve tri koordinati, a cijeli niz potom rotiramo oko sve tri osi. Time smo osigurali da možemo konstruirati bilo koji drugi izomorfan niz u prostoru.

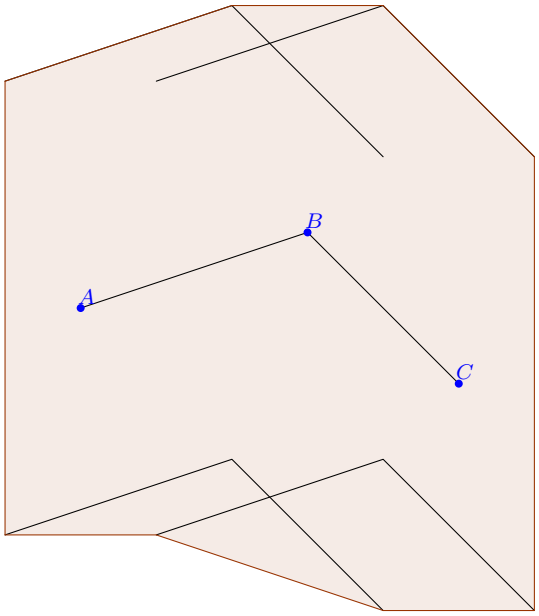
Da bismo bili sigurni da ne radimo prevelike skokove, svaka translacija ili rotacija događa se u određenom intervalu, to jest, nećemo raditi translacije koje za više od x, kao što prikazuje Slika 4.3, niti rotacije za više od α stupnjeva, što, pak, prikazuje Slika 4.2.

4.1.2. Računanje energije

Za računanje energije, koristimo prethodno opisani Smith-Waterman algoritam. Jedan ulazni niz je onaj koji želimo dobiti transformacijama, dok je drugi onaj za kojeg računamo energiju. Budući da nam rekonstrukcija nije potrebna jer nas zanima samo energija, nju niti ne računamo. Time ćemo dobiti ubrzanje od nekoliko puta.



Slika 4.2: Slika prikazuje maksimalnu i minimalnu rotaciju nekog niza. Rotacija se radi oko izvorišta. Iako radimo u 3D prostoru, za primjer je dana 2D rotacija.



Slika 4.3: Slika prikazuje maksimalnu i minimalnu translaciju nekog niza. Iako radimo u 3D prostoru, za primjer je dana 2D translacija.

4.1.3. Računanje vjerojatnosti prihvaćanja

Rješavanje ovog problema srž je algoritma simuliranog kaljenja. Jasno je da želimo otići u stanje koje ima bolju energiju, pa će nam ova funkcija kada je nova energija veća od trenutne, uvijek vratiti 1, što znači da ćemo sigurno otići u to stanje. Ono što razlikuje ovaj algoritam od klasičnog algoritma *penjanja na brdo* (engl. *hill-climbing*) jest ponašanje u slučaju da je nova energija manja od trenutne.

Simulirano kaljenje, kao što je već rečeno, ponekad ode i u lošija stanja. Hoće li se to dogoditi, ovisi o implementaciji funkcije P. Kako nam inspiracija dolazi iz obrade metala, vjerojatnost da prihvatimo loše stanje veća je na početku nego pri kraju. Također, ako je rješenje puno lošije, ne isplati nam ga se prihvatiti kao trenutno.

Razmotrimo način na koji možemo modelirati vrijeme. Pretpostavimo da nam q označava trenutnu vrijednost funkcije P. Također, uvedimo ograničenje da q mora biti iz intervala $[0, 1]$. Želimo, dakle, da P pada kako vrijeme odmiče. Najjednostavnije što možemo napraviti jest da od q oduzmemo t. Međutim, kako P mora biti u granicama $[0, 1]$ to nije dobro rješenje. Alternativno, ako oduzmemo t/t_max, dobili smo rješenje koje je potencijalno u intervalu $[-1, 1]$ i linearno pada ako mijenjamo samo vrijeme. Pokazalo se, međutim, da u "prirodi" nije baš tako. Vjerojatnost, naime, pada eksponencijalno.

Slično razmišljanje vrijedi i za modeliranje razlike u energiji. Što je razlika veća, to nam vjerojatnost prijelaza treba biti manja.

Obično se uzima slijedeća funkcija:

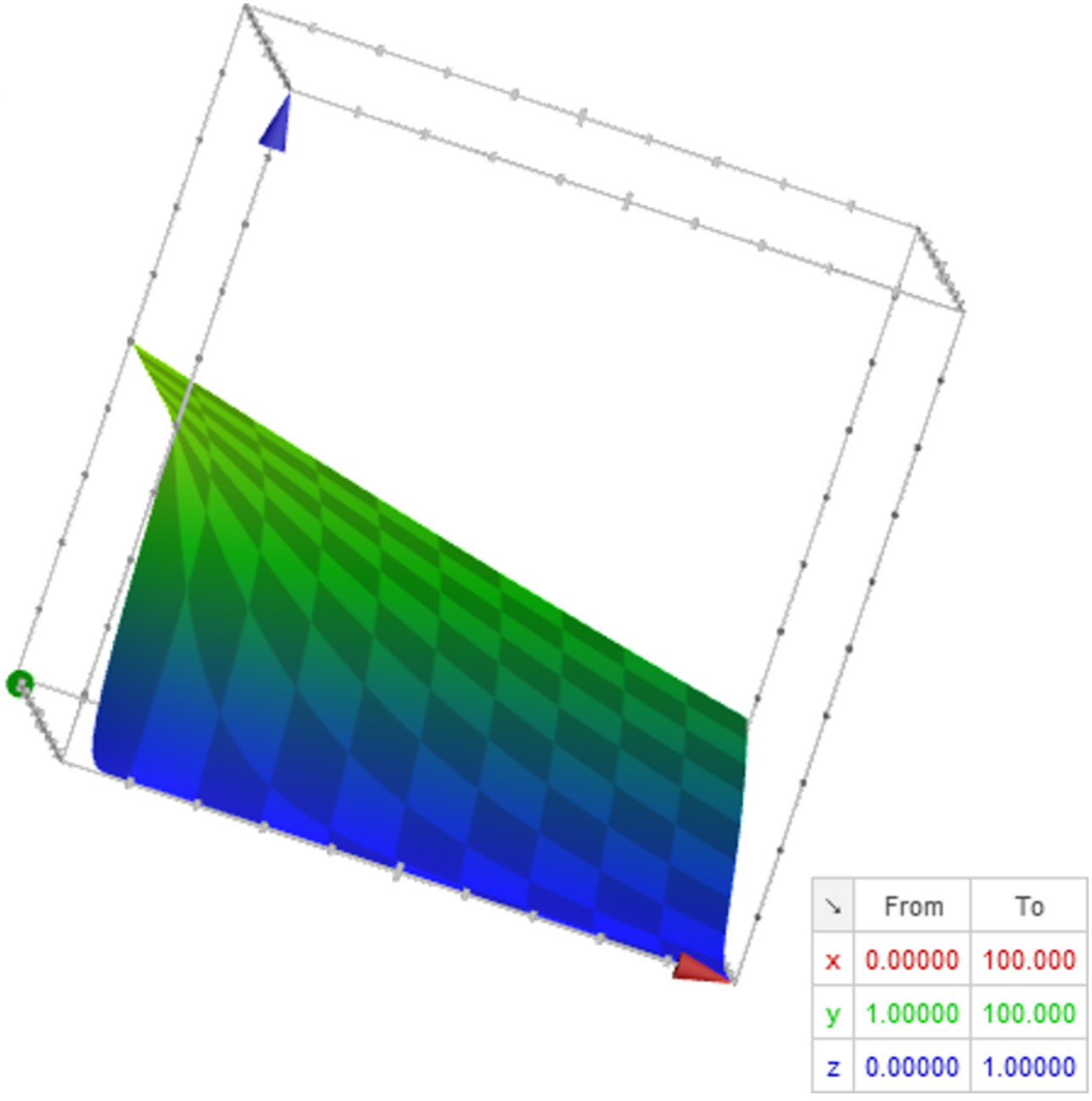
$$P(e, e', T) = \begin{cases} 1 & \text{ako je } e' > e \\ \frac{1}{1+\exp((e'-e)/T)} & \text{inače} \end{cases}$$

Primijetite da se, umjesto vremena, u njoj govori o temperaturi, koja eksponencijalno pada kako vrijeme odmiče.

Slika 4.4 daje graf funkcije $P(\Delta e, T)$, gdje je $\Delta e = e' - e$. Os x predstavlja Δe, os y je T (bitno je ponoviti da vremenom ova vrijednost pada) te os z prikazuje vrijednost P. Budući da 3D graf prikazujemo na 2D površini, pobojali smo ga u gradijent s obzirom na njegovu vrijednost kako bismo ga lakše interpretirali.

4.2. Modifikacije

Ovaj algoritam smo nešto prilagodili našoj upotrebi, kako bismo dobili prihvatljivo rješenje što je moguće prije. Budući da je cijeli algoritam baziran na heuristici, a izmjene koje smo napravili ne utječu na točnost, smijemo ih koristiti jer jedina stvar na što utječu su brzina algoritma.



Slika 4.4: Graf funkcije vjerojatnosti prihvaćanja P.

4.2.1. Više susjeda

Klasičan algoritam, kako je to gore objašnjeno, koristi samo jedan susjed u svakom koraku. Mi ćemo u našem rješenju koristiti više (k) njih. Od tako dobivenih, uzimamo onaj koji je najbolji i uspoređujemo ga, kako je gore opisano, s trenutnim stanjem. Na primjer, Slika 4.5 prikazuje situaciju moguću situaciju kada imamo 20 susjeda za koje smo izračunali energiju. Od njih biramo stanje s najboljom energijom i nju uspoređujemo s energijom trenutnog stanja. U ovom slučaju, s obzirom da je energija susjednog stanja veća od one trenutnog, trenutno stanje postane susjedno.

S obzirom da nam je svaki korak ocjenjivanja relativno skup, time se možemo osigurati rijetko odlazimo u slabija stanja. Međutim, takav pohlepan način odabira može nas dovesti u stanje lokalnog minimuma, iz kojeg se nećemo izvući. Da bismo pokušali otkloniti taj problem, u 10% generirat ćemo samo jednog susjeda u kojeg ćemo se, ako zadovoljava uvjete, proširiti.

Također, pokazat će se u nastavku, ovime možemo nešto uštedjeti i na korištenju memorije i time povećati efikasnost koju dobijemo kada koristimo grafički procesor za računanje traženih podataka.

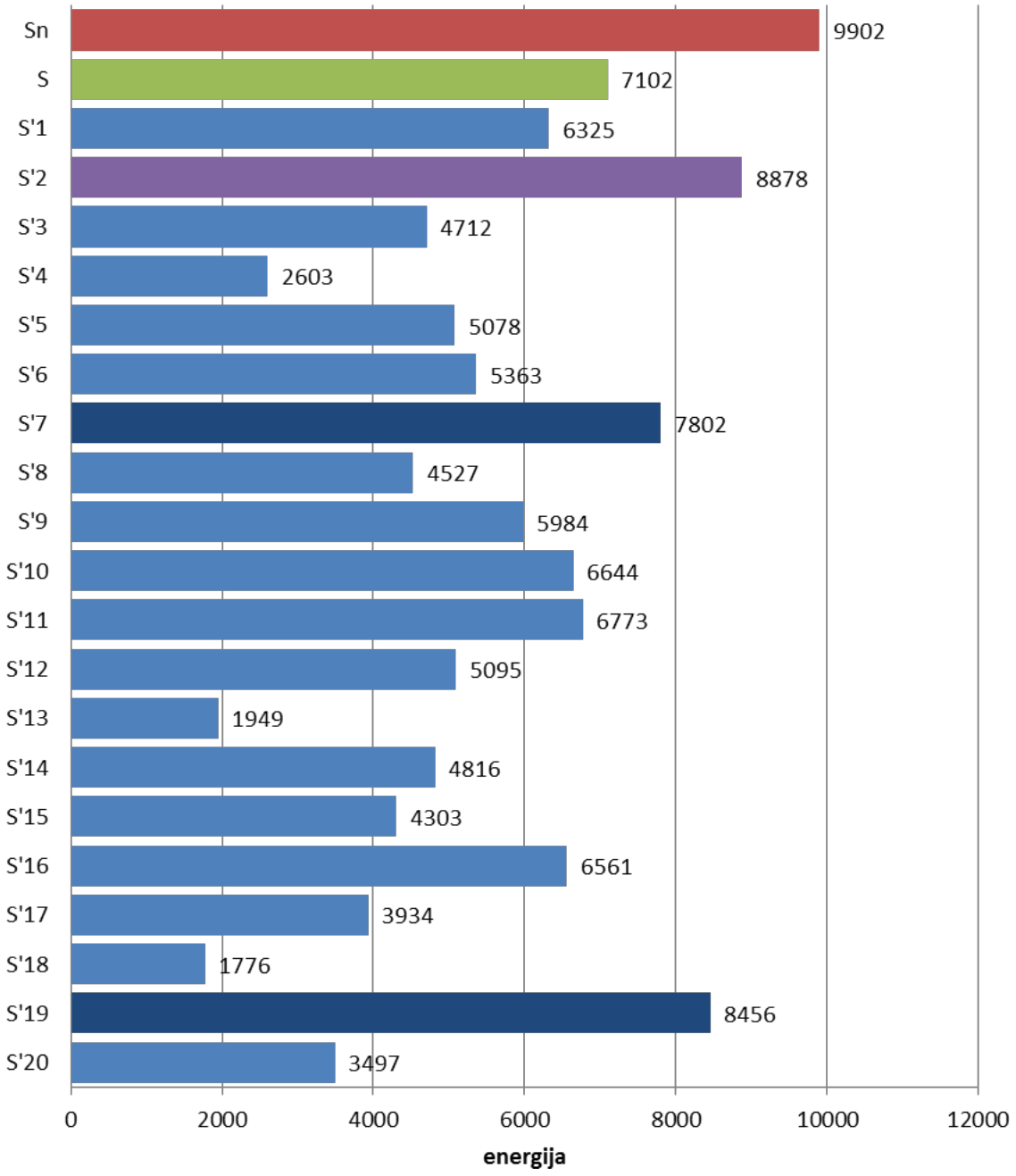
4.2.2. Ponovo pokretanje

Jedna od metoda da se izvučemo iz potencijalnog lokalnog ekstrema jest da ponovo pokrenemo algoritam iz neke prethodne točke. Često se to implementira na način da se promijeni vrijeme (smanji se, tj. postavi se u prošlost). Time se poveća vjerojatnost da izađemo iz nekog lokalnog ekstrema koristeći neka lošija stanja.

U slučaju da se najbolje rješenje nije promijenilo u zadnjih 15 koraka, vrijeme ćemo vratiti na početnu vrijednost da bismo se pokušali pomaknuti iz mogućeg lokalnog ekstrema.

4.3. Parametari

S obzirom da naglasak ovog rada nije bio isključivo na ovom algoritmu, a i na vremenske rokove postavljene na pisanje rada, parametri koji su se koristili u implementaciji nisu previše istraženi.



Slika 4.5: Prikazan je slučaj kada imamo 20 susjeda. S prikazuje trenutnu energiju (zelena boja), Sn prikazuje energiju najboljeg do sada poznatog stanja, a energije susjeda prikazana su plavom bojom. Energije svih susjeda koji imaju energiju veću od trenutnog niza su označene tamno-plavom bojom, a jedino je energija najvećeg susjeda, u koju ćemo premjestiti trenutno stanje, pobojana u ljubičasto.

5. CUDA tehnologija

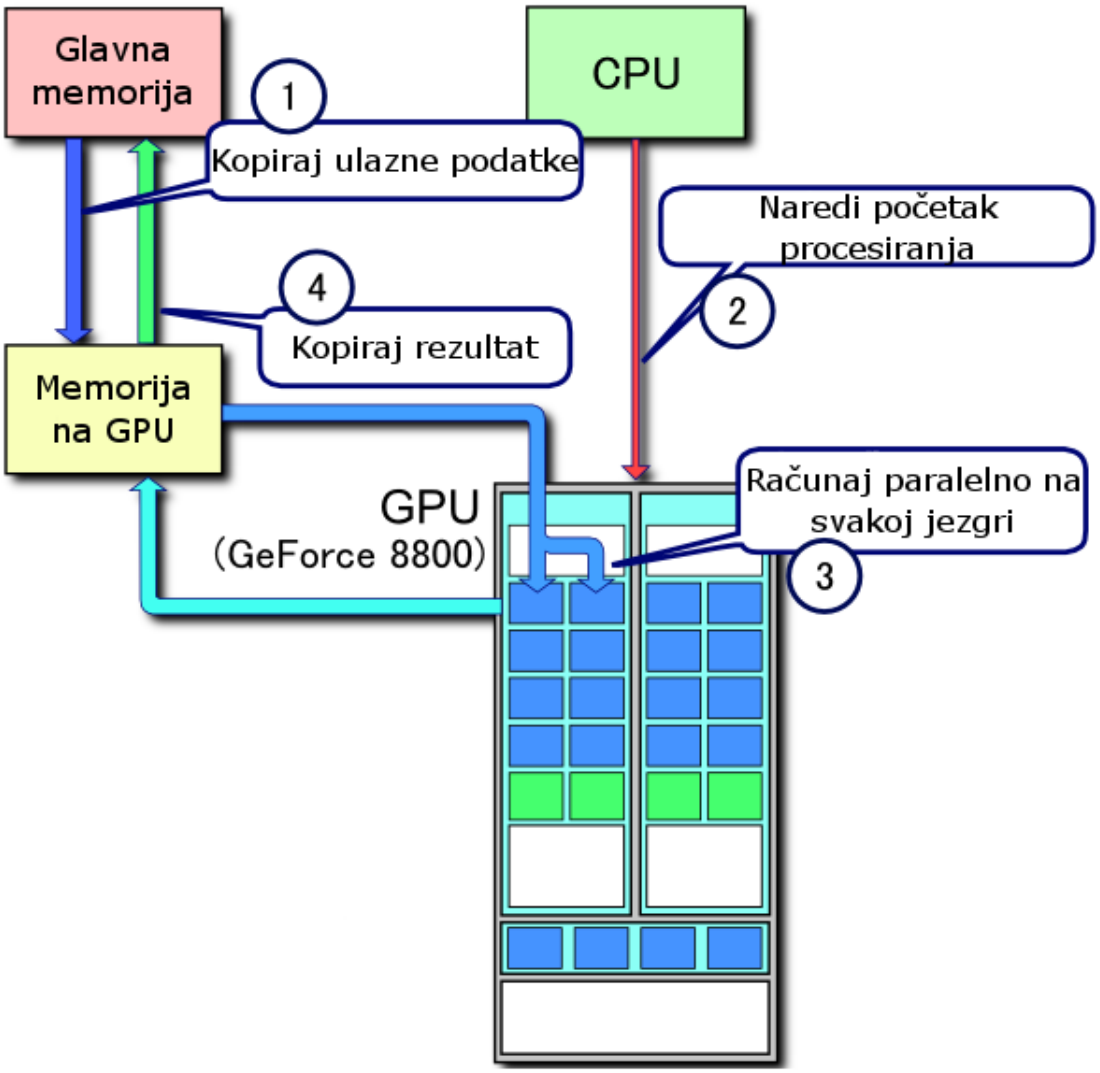
Compute Unified Device Architecture (engl.; računski objedinjena arhitektura uređaja) ili skraćeno CUDA, tehnologija je koja nam omogućuje da upregnemo moć grafičkih čipova na problemima za koje se tipično koristio CPU (odnosno procesor). Kompanija Nvidija razvila ju je za svoje grafičke kartice.

CUDA se smatra najnaprednijim alatom za opće-namjensku uporabu grafičke jedinice (engl. GPGPU - ***g****eneral* ***p****urpose* ***g****raphics* ***p****rocessing* ***u****nit*). S obzirom da se grafičke kartice nalaze u gotovo svim računalima, a vrlo se rijetko koriste u svojoj punoj moći (obično samo u računalnim igrama), inženjerima i znanstvenicima palo je na pamet da ne bi bilo loše iskoristiti njihovu snagu u rješavanju općenitijih zadataka.

Prva verzija CUDA-inog SDK-a (engl. ***s****oftware* ***d****evelopment* ***k****it* - oprema za razvoj programske podrške) izdana je 15. veljače 2007. Ona je omogućila korištenje resursa grafičkih kartica jedino u programskom jeziku C. Od tada do danas izdane su još dvije velike iteracije i nekoliko manjih te danas možemo koristiti i podskup jezika C++. Osim službenog SDK-a, za čitav niz programskih jezika, poput Fortrana, Jave, Haskella, Perla, Pythona, Matlaba, napisane su neslužbene poveznice.

Pri radu na CUDA-i, moramo razumjeti slijedeće pojmove:

- Domaćin (engl. *host*) - računalo u kojem se nalazi grafička kartica. Kada kažemo, npr. memorija domaćina mislimo na RAM.
- Uređaj (engl. *device*) - grafička kartica.
- Blok (engl. *block*) - skup dretvi koje se izvršavaju paralelno.
- Mreža (engl. *grid*) - skup blokova koji se izvršavaju paralelno.
- Dretva (engl. *thread*) - najmanja jedinica izvršavanja koja se može izvršiti. Sve naredbe u njoj izvršavaju se nužno slijedno, ali se više dretvi može izvršavati paralelno.
- Osnova (engl. *warp*) - skup dretvi unutar bloka koje se izvršavaju istodobno. Za njih možemo pretpostaviti da su u svakom trenutku na istom mjestu u kodu, pa ih međusobno ne treba sinkronizirati.



Slika 5.1: Protok podataka na CUDA grafičkim karticama. Prvo iz glavne memorije računala (RAM-a) kopiramo podatke u memoriju na GPU-u. Potom procesor naređuje GPU-u da pokrene odgovarajući kernel na nekom broju jezgri. Kada one završe, rezultat se kopira nazad u glavnu memoriju. CUDA SDK nas traži da ručno napravimo korake 1, 2 i 4, a on se brine da se pravilno izvrši korak 3.

Prednosti koje ova tehnologija (odnosno grafičke kartice koje su sposobne koristiti ju) ima nad ostalim GPGPU tehnologijama su slijedeće:

- Nudi pristup bilo kojem dijelu memorije.
- Dretve mogu koristiti brzu zajedničku dijelnu memoriju.
- Podrška za račun s cjelobrojnim tipovima podataka, uključujući i operacije s bitovima.

Nažalost, ništa u životu nije savršeno, pa tako nije niti CUDA. Treba imati na umu da i ona ima i neka ograničenja:

- Učestalo kopiranje memorije između uređaja i domaćina ima loš utjecaj na brzinu izvođenja programa.
- U praksi se pokazalo da trebamo koristiti barem 32 dretve paralelno da bismo dobili brži program nego onaj na CPU-u. Ukupan broj dretvi koje koristimo trebao bi se brojati u tisućama.
- Sklopovlje je dostupno isključivo od jednog proizvođača - Nvidije.
- Zbog nezrelosti prevoditelja i optimizatora valjani C/C++ kôd ne radi nužno i na CUDA-i.
- Računanje s brojevima s posmačnim zarezom nije implementirano posve po standardima, ali u većini slučajeva to ne igra nikakvu ulogu.

6. Implementacija

6.1. Implementacija simuliranog kaljenja

Simulirano kaljenje temelji se na postupku koji je identičan za sve vrste podataka. To nas navodi da implementaciju riješimo koristeći obrazac okvirna metoda. Ideja tog obrasca jest da imamo jednu metodu (funkciju) koja izvodi neki postupak pozivajući generičke korake. Svaki korak može biti drugačije implementiran, ovisno o postupku koji želimo.

Izdvojimo apstraktne korake iz algoritma:

1. Odabir susjeda.
2. Računanje energije stanja.
3. Računanje vjerojatnosti prihvaćanja.

Kako točno rješavamo svaki od tih koraka, napisalo smo u prethodnom poglavlju.

Implementacijski gledano, sva tri postupka su funkcije. Prvi je funkcija koja prima jedan parametar, trenutno stanje te vraća transformirano stanje koje dobijemo primjenjujući translaciju i/ili rotaciju. Drugi prima takvo transformirano stanje i računa njegovu energiju. To radi tako da primjeni prethodno opisani Smith-Waterman algoritam nad transforimarnim i ciljnim nizom, ali ne radimo rekonstrukciju. Treći prima energiju početnog i krajnjeg niza te trenutnu temperaturu te iz toga računa vjerojatnost prihvaćanja novog niza.

Zahvaljujući genericima, tj. predlošcima, u C++ je lagano ostvariti ortogonalnost. Njihovom upotrebom, omogućavamo da kôd algoritma napišemo samo jednom i da ga potom možemo koristiti na više mjesta, za više tipova podataka, što upravo i želimo.

Takva implementacija omogućuje nam da ovaj dio programskog rješenja testiramo i na nekom jednostavnijem problemu, što uvelike olakšava razvoj programske potpore jer ne moramo čekati za završetak cijelog sustava da bismo vidjeli radi li. Osim toga, prilikom bubolova (engl. *debugging*) nam je lakše tražiti pogreške ako primjer možemo vizualizirati. Nažalost, kako ovo nije problem u kojem je vizualizaciju lagano napraviti, autor je koristio neke jednostavnije probleme da bi ispravio pogreške u ovom dijelu koda. Primjeri tih problema su traženje specifične točke u prostoru, traženje težišta poligona i optimizacija puta

0	1	1	1	1
1	3	3	3	3
1	3	3	3	3
1	3	3	3	3
1	3	3	3	3

$\Rightarrow$

0	0	1	1	1
0	2	3	3	3
1	3	3	3	3
1	3	3	3	3
1	3	3	3	3

$\Rightarrow$

0	0	0	1	1
0	0	2	3	3
0	2	3	3	3
1	3	3	3	3
1	3	3	3	3

Tablica 6.1: Matrica W na polju $W_{i,j}$ prikazuje broj polja koja trebamo izračunati prije nego izračunamo polje $H_{i,j}$. Tamno-sivom bojom prikazana su izračunata polja, a svijetlo-sivom ona koja možemo izračunati.

trgovačkog putnika. Prilagodba algoritma za tu svrhu svela se na reimplementaciju funkcija za računanje energije i odabir susjeda.

6.2. Implementacija Smith-Watermanovog algoritma

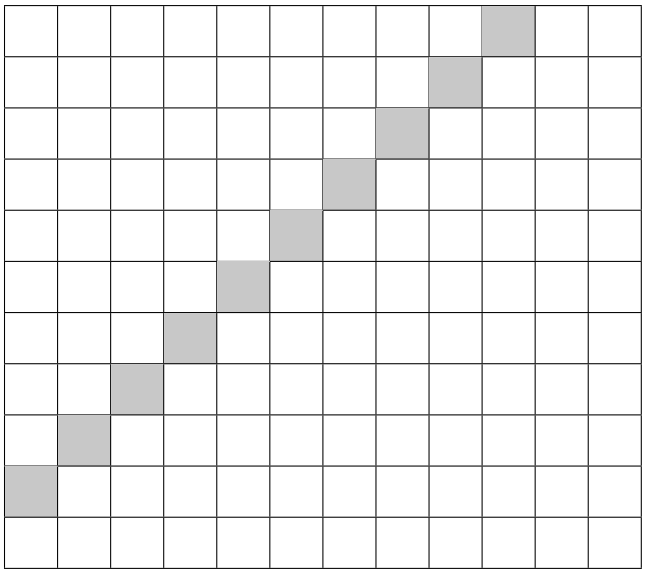
Do sada smo razmatrali samo situaciju kada računamo jedno po jedno polje matrice R. Pretpostavka je bila da su sva polja u prošlim recima i trenutnom retku do trenutnog stupca izračunata i da ih možemo koristiti u daljnjem računu.

Međutim, kada koristimo masivno paralelne procesore, želimo odjednom računati više od jednog polja jer inače ne upregnemo svu snagu koju nam oni nude.

Ideja na kojoj nam se temelji rješenje jest da primijetimo koja su polja nezavisna, tj. koja možemo koristiti u računu ako smo do sada već izračunali neki dio rješenja. Definirajmo novu matricu W u kojoj nam $W_{i,j}$ označava broj polja matrice R koje moramo izračunati prije nego izračunamo polje $R_{i,j}$. Na početku će vrijediti slijedeće:

$$W_{i,j} = \begin{cases} 0 & \text{ako je } i = 0 \text{ i } j = 0 \\ 3 & \text{ako je } i > 0 \text{ i } j > 0 \\ 1 & \text{inače} \end{cases}$$

Jasno je da možemo izračunati vrijednosti samo onih polja za koje je $W_{i,j} = 0$. Na početku je takvo samo jedno polje: $W_{0,0}$. Međutim, kada njega izračunamo, smanjit ćemo vrijednosti tri polja u matrici W: $W_{1,0}$, $W_{0,1}$ i $W_{1,1}$. To će uzrokovati da $W_{1,0}$ i $W_{0,1}$ postanu jednaki 0, što znači da sada smijemo i njih izračunati. Kada to učinimo, i osvježimo vrijednosti u matrici W, vidjet ćemo da smo dobili tri nove nule, na pozicijama $W_{2,0}$, $W_{0,2}$ i $W_{1,1}$. Tablica 6.1 prikazuje popunjavanje matrice W. Pravilnost se već lagano počinje nazirati. Naime, sve elemente koji se nalaze na istoj sporednoj dijagonali moći ćemo izračunati u istom koraku. Stoga ćemo paralelizaciju temeljiti upravo na tome da u pojedinom koraku računamo sve elemente na sporednoj dijagonali (Tablica 6.2).



Tablica 6.2: Prikazana je jedna od sporednih dijagonala matrice. Sva polja na njoj možemo računati u istom trenutku.

Vremenska složenost takvog bi algoritma u optimalnom slučaju bila bi $O(M + N)$. Nažalost, realan je slučaj daleko od optimalnog. Problem nam stvara to što nemamo dovoljno procesora da bismo mogli pokrenuti računanja nad svim elementima dijagonale, ako je ona prevelika. Na sreću, CUDA nam ne stvara prevelike probleme ako koristimo stvorimo više blokova (do najviše 65536) nego što grafička kartica fizički može istovremeno izračunati. S obzirom da dužina nizova vjerojatno neće biti puno veća od 15000 elemenata, takvo ograničenje je sasvim prihvatljivo.

Isto tako, ako u memoriju grafičke kartice ne stanu oba niza (što će se dogoditi, na sreću, samo za iznimno duge nizove koje nećemo promatrati), morat ćemo prilagoditi algoritam. Jedna efikasna ideja za rješavanje tog problema jest da podijelimo nizove na manje dijelove koje možemo u potpunosti riješiti te da kombiniramo tako dobivena rješenja. Međutim, to izlazi izvan teme ovog rada, pa nećemo ovdje detaljno objasniti kako se to radi.

Također, moramo biti svjesni da će konstanta koja je sakrivena u O notaciji biti puno veća nego kod obične implementacije na procesoru, budući da moramo koristiti barijere za sinkronizaciju. Pozivanje funkcija na uređaju također troši vrijeme, a i pristup globalnoj memoriji uređaja jest skuplji nego pristup radnoj memoriji računala.

Što se memorijske složenosti tiče, i nju možemo ovdje ostvariti kao $O(\min(M, N))$. Dovoljno nam je da, umjesto posljednja dva reda, pamtimo posljednje tri sporedne dijagonale. Rekonstrukcija, iako nam neće biti potrebna, može se riješiti Hirschbergovim algoritmom.

6.2.1. Uspoređivanje više nizova odjednom

Do sada je opisan problem uspoređivanja jednog niza s drugim. S obzirom na zahtjeve ovog rada (traženje transformacije jednog od nizova tako da maskimiziramo ocjenu preklapanja), valja razmotriti mogućnost da uspoređujemo više nizova odjednom. Zanemarimo na trenutak činjenicu da smijemo transformirati samo jedan od dva dana niza.

Neka nam je K broj nizova koje napravimo transformacijom prvog, a L broj koji dobijemo transformacijom drugog. Ako spojimo sve transformacije prvog niza u jedan super-niz, tako da među dvije transformacije ubacimo atom koji se nalazi u $+\infty$, te isto napravimo s drugim nizom (ali atome na međi stavimo u $-\infty$), možemo napraviti Smith-Watermana nad tim super-nizovima. Budući da će rezultat pri usporedbi bilo koja dva atoma s onima na međi biti $-\infty$, dobili smo resetiranije algoritma na početku svakog para nizova i time računom samo jednog Smith-Watermana možemo odjednom dobiti najbolje lokalno poravnanje među $K \times L$ nizova. Najbolji par transformacija je onaj koji je napravio par nizova na čijem je presjeku nastalo najbolje lokalno poravnanje. Kako smo i do sad od više mogućih transformacija birali samo najbolje, to ćemo i ovaj put učiniti.

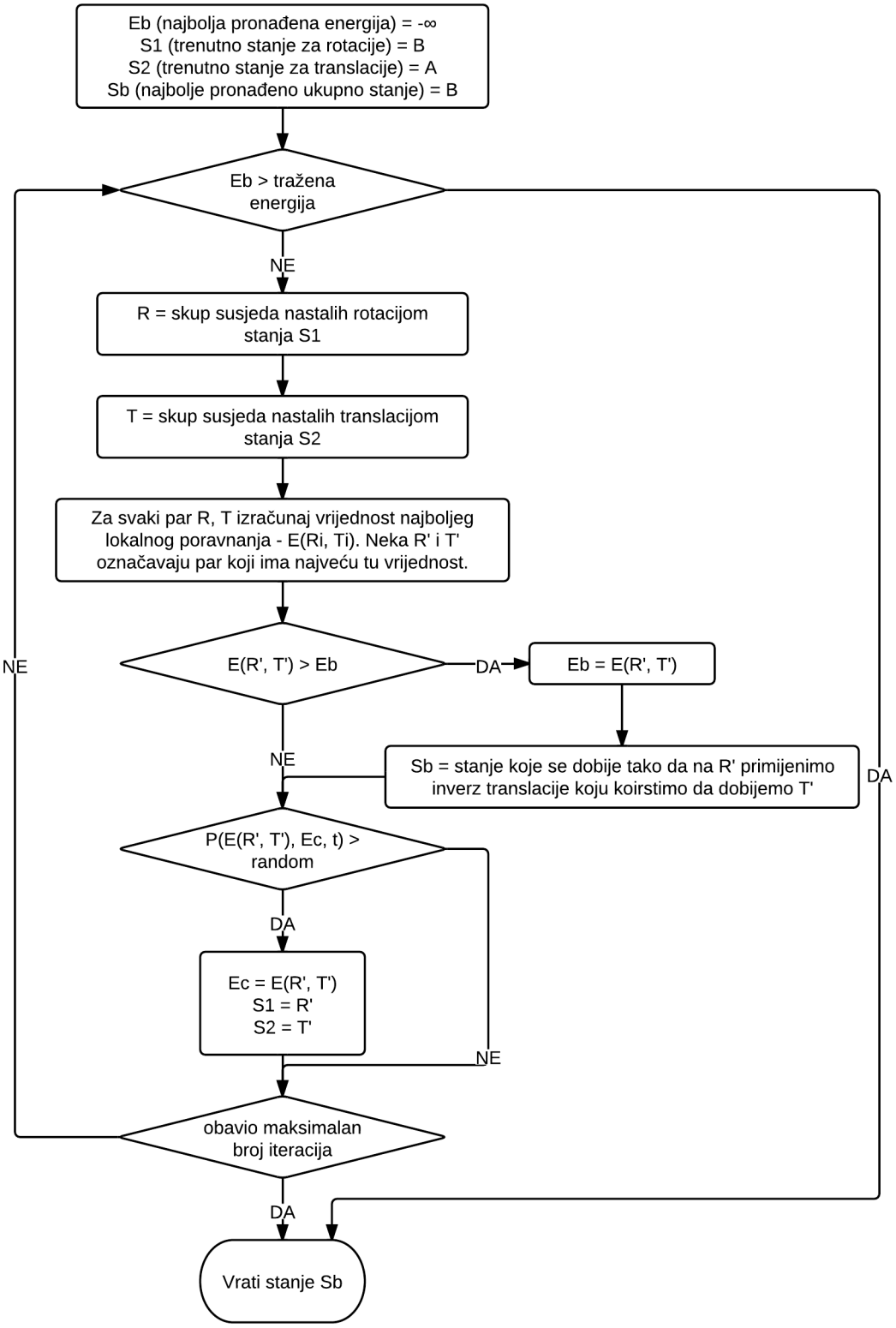
Prednost ovakvog pristupa jest da se broj elemenata na dijagonali znatno povećava, čime se povećava i broj paralelnih računa. Osim toga, vrijeme prebacivanja iz jedne memorije na drugu i vrijeme potrebno da se kernel pokrene znatno se smanjuju. Preciznije, relativan odnos tih vremena s vremenom potrebnim da se izračuna Smith-Watermanov algoritam na danim nizovima smanjuje se za čak i do dva reda veličine! Kako je kod kraćih nizova to vrijeme često i duže od vremena za računanje algoritma, postižemo dosta dobro ubrzanje.

Glavni nedostatak ove metode jest povećavanje zahtjeva za memoriju. Međutim, budući da je korištenje memorije linearno u ovisnosti o dužini kraćeg niza, a nizovi nisu pretjerano dugački (oko 15,000 elemenata svaki u najgorem slučaju), to ne predstavlja veliku manu.

Preostaje nam još jedino opisati način na koji transformiramo oba niza da možemo jednostavno pretvoriti pretvoriti te transformacije u transformaciju isključivo drugog niza. Odgovor na to pitanje vrlo je jednostavan - ako onaj niz koji prije nismo dirali sada translatiramo, a onaj koji smo prije transformirali sada samo rotiramo, konstrukcija transformacije isključivo drugog niza je trivijalna. Naime, na dobivenu rotaciju primijenimo inverznu translaciju prvog niza i tako dobijemo potpunu transformaciju isključivo drugog niza.

6.3. Konačna implementacija

Pogledajmo kako to u konačnici izgleda kada spojimo sve o čemu smo do sada pričali. Slika 6.1 prikazuje dijagram toka algoritma simuliranog kaljenja. Pseudokod algoritma koji računa lokalno poravnanje za više nizova odjednom prikazuje Slika 6.2, a Slika 6.3 prikazuje funkciju za računanje jednog elementa na dijagonali.



Slika 6.1: Dijagram toka implementacije simuliranog kaljenja.

$A \leftarrow$ prazan protein
$B \leftarrow$ prazan protein
for $i \in [1, K]$ **do**
    $A \leftarrow A + T_i+$ atom koji se nalazi na (∞, ∞, ∞)
end for
for $j \in [1, L]$ **do**
    $B \leftarrow B + R_j+$ atom koji se nalazi na $(-\infty, -\infty, -\infty)$
end for
$A_{device} \leftarrow$ kopija proteina A na uređaju
$B_{device} \leftarrow$ kopija proteina B na uređaju
$N \leftarrow$ dužina proteina A
$M \leftarrow$ dužina proteina B
$H \leftarrow$ posljednje tri dijagonale matrice ukupnog rješenja (na početku sve 0)
$H^A \leftarrow$ posljednje tri dijagonale matrice procjepa u A (na početku sve 0)
$H^B \leftarrow$ posljednje tri dijagonale matrice procjepa u B (na početku sve 0)
$res \leftarrow$ vrijednost najboljeg poravnanja (na početku 0)
$p \leftarrow$ par transformacija koji daje najbolje poravnanje (na početku $(1, 1)$)
for $d \in [1, N + M - 1]$ **do**
    $izracunajCijeluDijagonaluNaGPU(d, H, H^A, H^B, A_{device}, B_{device}, N, M)$
    $b \leftarrow$ indeks polja zadnje dijagonale koje daje najveću vrijednost
    if $H_b > res$ **then**
        $res \leftarrow H_b$
        $p \leftarrow$ par transformacija koji se nalazi na b-tom polju dijagonale d
    end if
end for
return (res, p)

Slika 6.2: Pseudokod modificiranog Smith-Waterman algoritma koji koristimo u implementaciji. Grafička kartica prilikom poziva funkcije $izracunajCijeluDijagonaluNaGPU$ zapravo pozove $izraunajElementDijagonaleNaGPU$ (Slika 6.3) za svaki element te dijagonale. Kao prvi element doda indeks elementa na dijagonali kojeg taj poziv treba izračunati.

function IZRAČUNAJELEMENTDIJAGONALENAGPU($i, d, H, H^A, H^B, A_{device}, B_{device}, N, M$)

$tren \leftarrow d$ mod 3

$prosla \leftarrow (d + 1)$ mod 3

$pretprosla \leftarrow (d + 2)$ mod 3

$x \leftarrow$ x koordinata i. elementa d. dijagonale

$x \leftarrow$ y koordinata i. elementa d. dijagonale

if $x = 0$ or $y = 0$ **then**

$H_{tren} \leftarrow 0$

$H^A_{tren} \leftarrow 0$

$H^B_{tren} \leftarrow 0$

return

end if

$gore \leftarrow$ indeks elementa $d - 1$. dijagonale koji se nalazi na $(x - 1, y)$

$lijevo \leftarrow$ indeks elementa $d - 1$. dijagonale koji se nalazi na $(x, y - 1)$

$oba \leftarrow$ indeks elementa $d - 2$. dijagonale koji se nalazi na $(x - 1, y - 1)$

$tmp_1 \leftarrow \max(0, H_{pretprosla,oba} + cijena(A_x, B_y))$

$tmp_2 \leftarrow H^A_{prosla,gore} + C$

$tmp_3 \leftarrow H^B_{prosla,lijevo} + C$

$H_{tren,i} \leftarrow \max(tmp_1, tmp_2, tmp_3)$

$tmp_4 \leftarrow H^A_{prosla,gore} + D$

$H^A_{tren,i} \leftarrow \max(tmp_1, tmp_4, tmp_3)$

$tmp_5 \leftarrow H^B_{prosla,lijevo} + D$

$H^B_{tren,i} \leftarrow \max(tmp_1, tmp_2, tmp_5)$

end function

Slika 6.3: Pseudokod funkcije koja računa jedno polje dijagonale na uređaju. Cilj nam je da na kraju funkcije imamo izračunate vrijednosti za i. element d. dijagonale, tj. $H_{tren,i}$, $H^A_{tren,i}$, i $H^B_{tren,i}$

7. Rezultati

Za provjeru rada programa koristili smo primjere za koje već znamo neka dobra preklapanja. Uzeli smo proteine 1d0nA i 2d8bA. Situaciju prije preklapanja prikazuje Slika 7.1. Nakon što naš algoritam odradi posao, imamo situaciju koju prikazuje Slika 7.2. Obje slike smo dobili koristeći PyMOL (2010). Iz te je dvije slike očigledno da smo uspjeli poboljšati pronađeno poravnanje, ali i da rješenje nije nužno optimalno. Jedan od načina kojim potencijalno možemo poboljšati ovaj algoritam jest da smanjujemo pomake i rotacije susjeda kako se smanjuje i temperatura, tj. što smo bliži lokalnom maksimumu. To bi nam omogućilo da radimo finije pomake što smo bliže rješenju, čime bismo dobili i preciznije poravnanje. Osim toga, tako bismo riješili i problem koji imamo kada su nam proteini na početku jako udaljeni. U tom slučaju je udaljenost između atoma nekoliko redova veličina manja nego udaljenost dva proteina, što znači da će postupak do pronalaska prvog preklapanja dugo trajati. Ako tada radimo velike korake, brže ćemo se približiti rješenju, pa ćemo ga potom i brže pronaći.

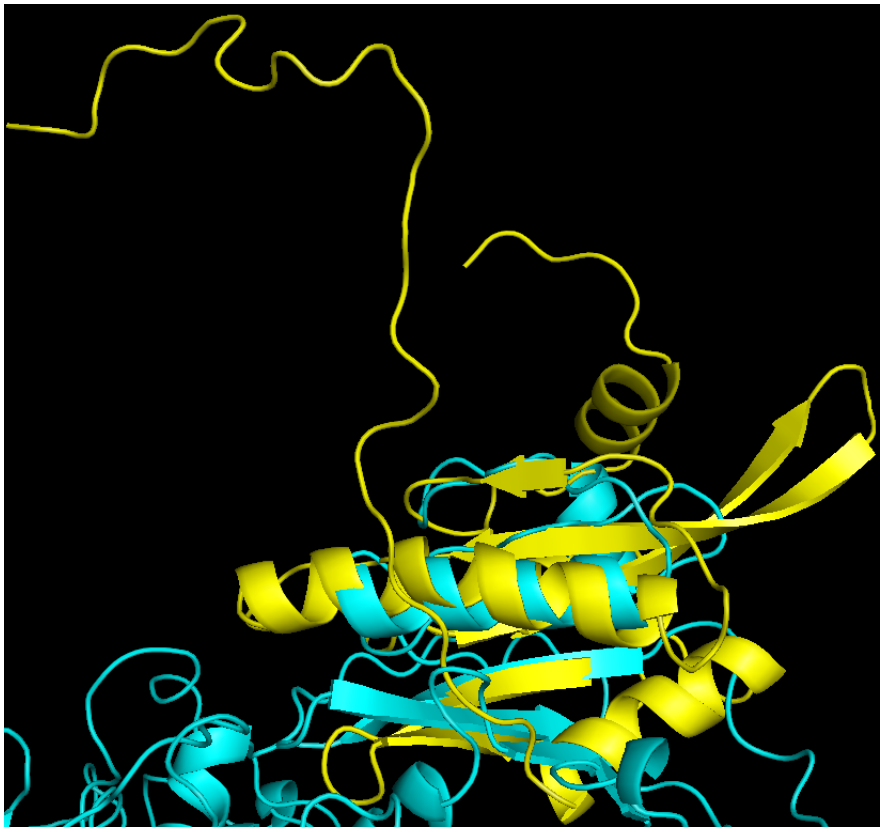
Kada prije pokretanja programa niti približno ne znamo točna rješenja, za neku jednostavnu provjeru vjerodostojnosti možemo izračunati energiju koju bismo imali ako bismo pokušali poravnati prvi niz sa samim sobom. Budući da je to ujedno i maksimalna energija koju potencijalno možemo pronaći, iz nje i najbolje pronađene energije možemo razumjeti koliko nas je algoritam simuliranog kaljenja doveo blizu rješenja.

Alternativan način provjere jest da promatramo mijenjanje energije kroz vrijeme, budući nju želimo maksimizirati. U nastavku su dani grafovi (Slika 7.3 i Slika 7.4) koji prikazuju tri energije: energiju najboljeg susjeda, energiju najboljeg do sada pronađenog elementa i energiju trenutno aktivnog elementa. Osim toga, prikazuju i trenutnu temperaturu, da bi nam bilo jasno koliko često prihvaćamo rješenja slabija od trenutnog.

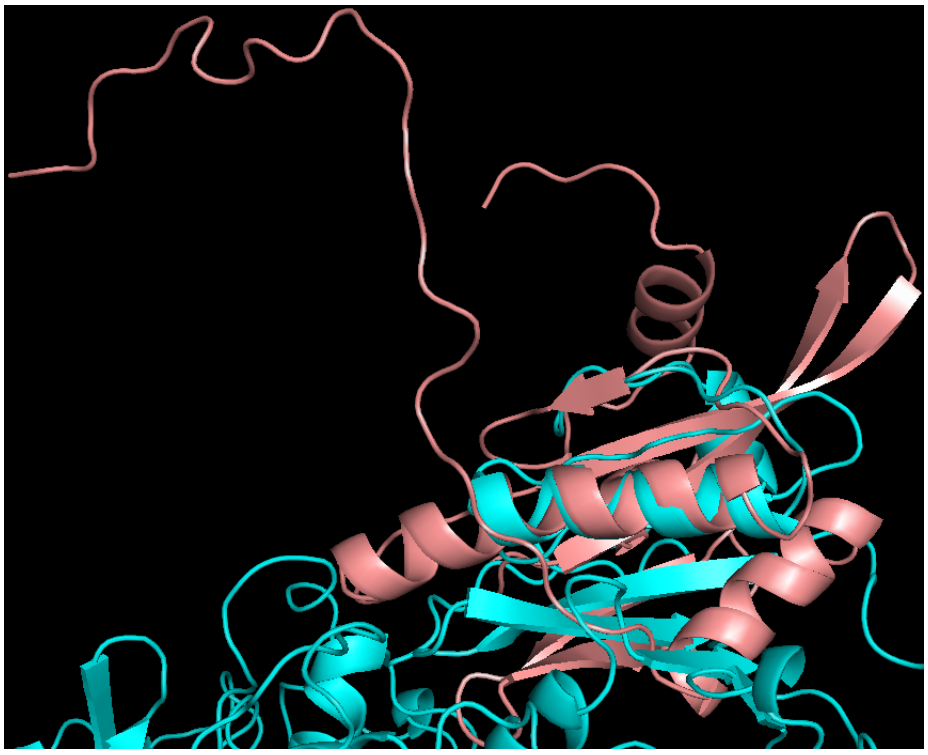
7.1. Potencijalna unaprijeđenja

Osim prethodno opisanog poboljšanja koje bi nam omogućilo traženje finijeg poravnanja, možemo još štošta poboljšati da dobijemo kvalitetnija rješenja.

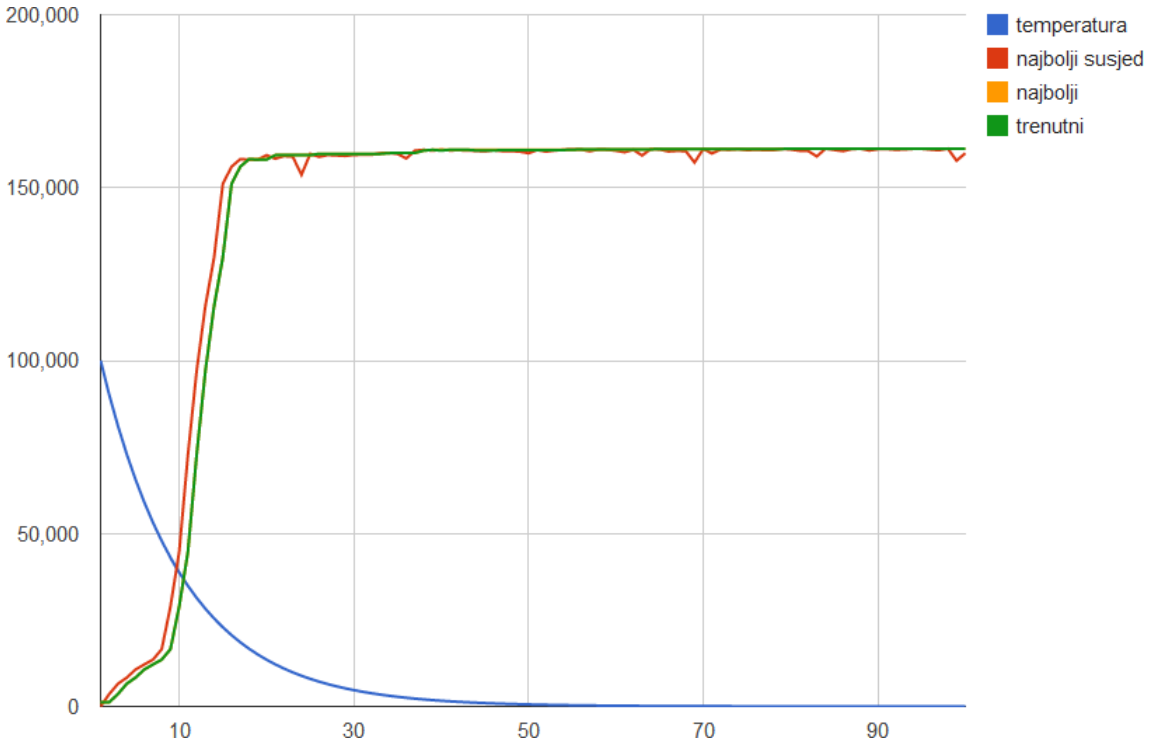
Traženjem boljih i preciznijih konstanti možemo, kao i u svakom drugom optimizacijskom problemu baziranom na heuristici učiniti čuda i naći nekoliko puta bolja rješenja.



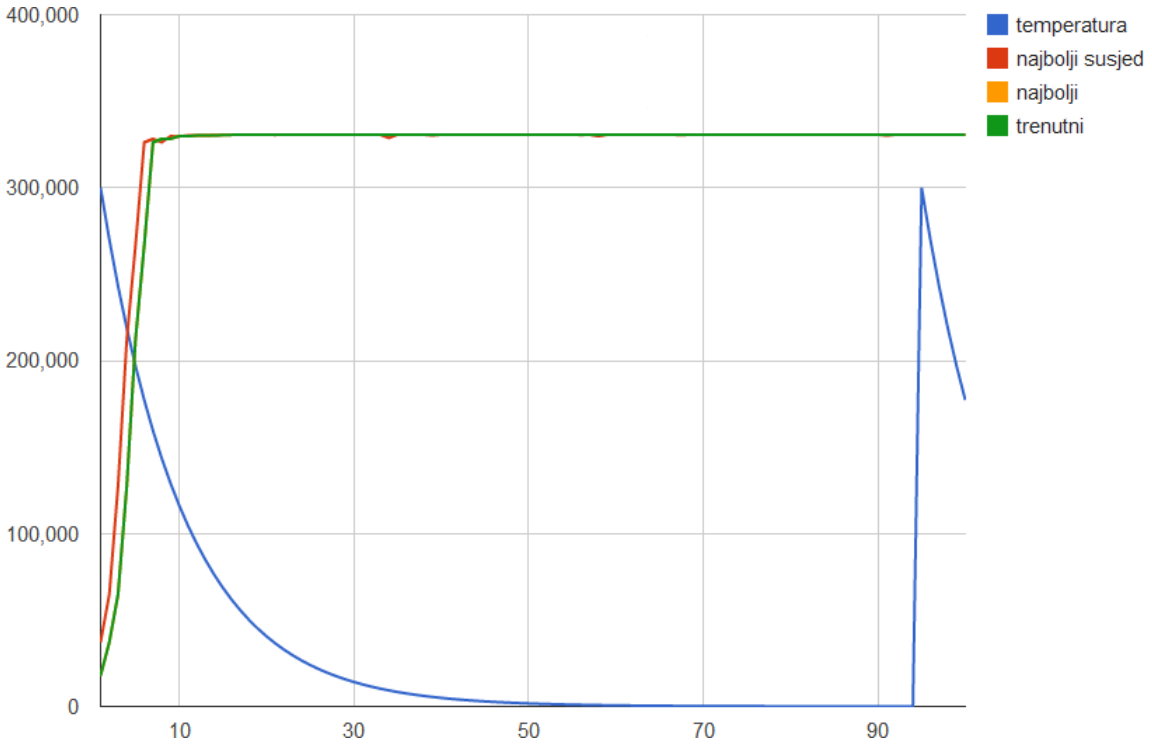
Slika 7.1: Pozicije u prostoru proteina 1d0nA i 2d8bA prije nego što su poravnati našim algoritmom. Obratite pažnju na plavu i žutu spiralu i kako se preklapaju.



Slika 7.2: Pozicije u prostoru proteina 1d0nA i 2d8bA nakon što su poravnati. Crvena i plava spirala sada se preklapaju puno više nego plava i žuta koje prikazuje Slika 7.1



Slika 7.3: Izvođenje programa na proteinima 1d0nA i 2d8bA. Uspjeli smo ostvariti otprilike 20% maksimalne moguće vrijednosti preklapanja (nešto manje od 730000).



Slika 7.4: Uzeli smo protein 1a0iA i iz njega generirali nasumičnu transformaciju. Potom smo pokrenuli algoritam i dobili gotovo 100% preklapanje u manje od 10 koraka. Budući da se u 15 koraka trenutno riješenje nije promijenilo, nešto nakon 90. koraka smo vratili temperaturu na početnu vrijednost, kako bismo pokušali pronaći bolje rješenje micanjem iz lokalnog maksimuma.

Kako se to efikasno je dosta velika tema i prelazi granice ovog rada pa ćemo tu diskusiju preskočiti.

Nasuprot generalnom poboljšanju, stoji poboljšanje specifično za našu primjenu. U ovom radu koristimo samo fizičke pozicije za usporedbu, pa očito postoji prostor za unaprjeđenje u vidu prilagođavanja funkcije ocjenjivanja dvije molekule tako da počnemo koristiti biološka i kemijska svojstva atoma i molekula, odnosno nukleotida i proteina. Time bi rezultati koje dobijemo mogli postati puno smisleniji, barem što se prirode tiče.

Alternativno, programsko bi rješenje sigurno bilo privlačnije kada bi se dodatno ubrzalo, a to se sigurno može postići raznim niskim optimizacijama i pažljivijom uporabom memorije. Drugi, elegantniji, način ubrzanja jest da poboljšamo algoritam, a jedan od načina kako bismo to napravili jest da smanjimo broj malih dijagonala. To potencijalno možemo u nekoj mjeri efikasno napraviti ako odjednom paralelno računamo Smith-Waterman algoritam na O parova nizova, pri čemu u svakom koraku računamo istu dijagonalu za sve parove. Time, unatoč tome što nam je prva dijagonala dužine 1, krenemo odmah računati O njih, što smanjuje relativnu cijenu korištenja CUDA arhitekture (pozivi kernel funkcija, kopiranje u memoriju, sinkronizacija...). Osim toga, u svakom bi koraku broj elemenata koje možemo izračunati rastao za O, čime bismo puno brže došli do trenutka kada se trošak pokretanja paralelnog računa isplati.

8. Zaključak

Jedan od najvećih problema s kojim se bioinformatika danas susreće je problem poravnanja struktura. Pojavom grafičkog sklopovlja koje omogućava opće-namjensko računanje, znanstvenici su odmah prionuli poslu i počeli raditi na projektima kojima je za cilj iskoristiti moć tih čipova za rješavanje problema poravnanja struktura. No, tehnologija koja se koristi za rješavanje tih problema još uvijek je u povojima. Svakih nekoliko mjeseci izlaze čipovi koji su mnogostruko jači u odnose na prethodne, ali i nešto promijenjenom arhitekturom koja obično omogućava pisanje efikasnijih programa. Zbog toga programi vrlo brzo zastarijevaju, a kako se povećava broj fizičkih jezgri koje kartice sadržavaju, mnoge pretpostavke koje su nam bile temelj osmišljavanja algoritma više ne vrijede. Za nekoliko godina moglo bi se dogoditi da grafička kartica ima i nekoliko stotina tisuća dretvi, što bi značilo da u većini slučajeva imamo dovoljno dretvi na raspolaganju da sav posao koji možemo radimo paralelno. Time bismo mogli napraviti mnogo efikasnije algoritme. Također, pomoćni alati, biblioteke, pa čak i prevoditelj, su još u razvoju.

U ovom radu smo prikazali kako implementirati algoritam Smith-Waterman na CUDA grafičkoj kartici. On se koristi da pronađemo lokalno poravnanje dva niza. Specifičnost implementiranog je u tome što ne koristi supstitucijsku matricu kao osnovu za ocjenjivanje poravnanja, već fiziče udaljenosti između molekula. Ta funkcija sigurno se može još unaprijediti i time bi se rezultati također mogli poboljšati. Ako takvu implementaciju Smith-Watermanovog algoritma ukomponiramo kao sredstvo izračuna energije u simuliranjom kaljenju, možemo pronaći takvu transformaciju drugog niza koja maksimizira vrijednost poravnanja. Drugim riječima, pomoću njega možemo ocijeniti sličnost dva niza, bez obzira na njihovu početnu poziciju.

LITERATURA

Michael Downes. *Short Math Guide for LaTeX*. American Mathematical Society, 2002. URL ftp://ftp.ams.org/pub/tex/doc/amsmath/short-math-guide.pdf.

Google. Graf funkcije p, 6 2012. URL https://www.google.hr/search?hl=en&biw=1857&bih=995&noj=1&sclient=psy-ab&q=1%2F%281%2Be%5E%28x%2Fy%29%29&oq=1%2F%281%2Be%5E%28x%2Fy%29%29&aq=f&aqi=&aql=&gs_l=serp.3...274395.277370.0.277780.6.6.0.0.0.0.176.515.5j1.6.0...0.0.h3zpB1qjAkE.

Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162 (3):705–8, 1982.

Dan S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18:341–343, 1975.

L. Holm, C. Ouzounis, C. Sander, G. Tuparev, i G. Vriend. A database of protein structure families with common folding motifs. *Protein Science*, 1:12, 1992.

Šime Ungar. *Uvod u TeX s naglaskom na LaTeX2ϵ*. Odjel za matematiku, Sveučilište J.J. Strossmayera u Osijeku, 2002.

Arun S. Konagurthu, James C. Whisstock, Peter J. Stuckey, i Arthur M. Lesk. Mustang: A multiple structural alignment algorithm. *Proteins: Structure, Function, and Bioinformatics*, 64:559–574, 2006.

Matija Korpar. Implementacija smith waterman algoritma koristeći grafičke kartice s cuda arhitekturom. Završni rad, 6 2011.

Saul B. Needleman i Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48: 443–453, 1970.

T. Oetiket, H. Partl, Hyna, i E. Schlegl. *The not-so-short introduction to Latex*, 2007. URL http://www.ctan.org/tex-archive/info/lshort/english/lshort.pdf.

Shrodinger LLC. PyMOL. The pymol molecular graphics system, 2010. URL http://www.pymol.org/pymol.

TeX/LaTeX savjeti. Tex/latex savjeti, 2012. URL https://www.facebook.com/pages/TeXLaTeX-savjeti/261129223970236.

I. N. Shindyalov i P. E. Bourne. Protein structure alignment by incremental combinatorial extension (ce) of the optimal path. *Protein Engineering*, 11(9):739–747, 1998.

Temple F. Smith i Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

Mike tn. Crushing fire (photo), 2011. URL http://www.flickr.com/photos/beginasyouare/6121659469/sizes/l/in/photostream/.

Goran Žužić. Usporedba dna sekvenci: Smith-waterman algoritam. Seminarski rad, 4 2011.

Yuzhen Ye i Adam Godzik. Flexible structure alignment by chaining aligned fragment pairs allowing twists. *Bioinformatics*, 19:ii246–ii255, 2003.

SWIG - Poravnanje struktura korištenjem iterativne primjene Smith-Waterman algoritma

Sažetak

Algoritam dinamičkog programiranje Smith-Waterman služi nam da deterministički pronađemo lokalno poravnanje neka dva proteina. Možemo ga prilagoditi da, umjesto supstitucijske matrice, kao sredstvo ocjene koristi fizičke udaljenosti između dvije molekule proteina u prostu. Metoda simuliranog kaljenja omogućava nam da pronađemo lokalni maksimum neke funkcije. Ako kao funkciju energije koristimo algoritam Smith-Waterman, a za pronalazak susjeda rotiramo i translatiramo drugi protein, možemo provjeriti koliko su dva proteina fizički slična bez obzira na njihovu početnu poziciju. Implementacijom tih algoritama na grafičkim karticama koristeći CUDA tehnologiju, možemo dobiti nekoliko redova veličine brži kod, pogotovo kod većih proteina. Kod manjih proteina, efekt je nažalost obratan i dobivamo sporiji kod jer je konstanta skrivena u O notaciji relativno velika.

Ključne riječi: Smith-Waterman, CUDA, simulirano kaljenje, bioinformatika, paralelizacija, poravnanje struktura

SWIG - Aligning structures using iterative application of Smith-Waterman algorithm

Abstract

Dynamic programming algorithm Smith-Waterman is used to deterministically find local alignment of two proteins. We can modify it to, instead of using substitution matrix, use physical distances between two molecules as a rating function. Simulated annealing method is used to find a local extreme (maximum) of a given function. If we use Smith-Waterman algorithm as energy function, and for finding neighbors we rotate and translate one of the proteins, we can check how physically similar two proteins are regardless of their initial positions. Implementing those algorithms on graphic cards using CUDA technology, we can get code that is orders of magnitude faster, especially with larger proteins. With smaller proteins, however, the effect is, unfortunately, adversary, and we get slower code because the constant hidden in O notation is relatively large.

Keywords: Smith-Waterman, CUDA, simulated annealing, bioinformatics, parallelization, aligning structures