VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

EFEKTIVNÍ VARIANTY DYNAMICKÉHO PROGRAMOVÁNÍ V BIOINFORMATICE

EFFECTIVE DYNAMIC PROGRAMMING IN BIOINFORMATICS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **JAROMÍR FRANĚK**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. IVANA BURGETOVÁ, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Franěk Jaromír**
Program: Informační technologie
Název: **Efektivní varianty dynamického programování v bioinformatice**
Effective Dynamic Programming in Bioinformatics
Kategorie: Algoritmy a datové struktury

Zadání:

1. Seznamte se s technikou dynamického programování a s bioinformatickými problémy, které lze řešit s využitím této techniky.
2. Po dohodě s vedoucí vyberte bioinformatické problémy, kterými se budete podrobně zabývat.
3. Prostudujte existující algoritmy pro řešení vybraných problémů. Zaměřte se na algoritmy, které řeší problémy efektivním způsobem.
4. Po dohodě s vedoucí navrhněte aplikaci, která bude demonstrovat řešení zvolených problémů různými algoritmy.
5. Navrženou aplikaci implementujte.
6. Aplikaci otestujte a zhodnoťte dosažené výsledky.

Literatura:

- Mareš, M., Valla, T.: *Průvodce labyrintem algoritmů,* CZ.NIC, 2017, ISBN 978-80-88168-19-5
- Zvelebil, M. J., Baum, J. O.: Understanding Bioinformatics, Garland Science, 2007, ISBN 9780815340249
- Andreas D. Baxevanis, B. F. Francis Ouellette: Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins, Wiley-Interscience, 2005, ISBN: 0-471-47878-4

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 4.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Burgetová Ivana, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 28. května 2020
Datum schválení: 16. října 2019

Abstrakt

Cílem této práce je nastudovat princip efektivních algoritmů využívajících dynamické programování. S pomocí těchto znalostí vytvořit aplikaci demonstrující princip efektivních algoritmů dynamického programování v bioinformatice a sepsat zprávu shrnující výsledky. Algoritmy, obsažené v této práci, řeší zarovnání sekvencí DNA, nebo predikci sekundární struktury RNA. Tyto algoritmy jsou zde porovnávány mezi sebou pro různé hodnoty vstupů. Pro samotné zarovnání sekvencí jsou zde použity algoritmy jako Needleman-Wunch a X-drop. Pro predikci sekundární struktury RNA je použit Zukerův algoritmus, který by měl odstraňovat některé nedostatky Nussinin algoritmu a samotný Nussinin algoritmus. Rekurze je zde představována pomocí rekurzivních stromů, dynamické programování pomocí skórovací matice. Uživatel má možnost také porovnat rychlosti obou přístupů pro zadané sekvence. Pro zajištění jednoduché dostupnosti se jedná o webovou aplikaci běžící na straně klienta.

Abstract

Purpose of this thesis is to study principle of effective algorithms, that are using dynamic programming. Using this knowledge to create application demonstrating principle of effective algorithm of dynamic programming in bioinformatics and write a report summarizing results. Algorithms used in this thesis are solving DNA sequence alignment or RNA secondary structure prediction. These algorithms are compared between themselves based on different input values. For DNA sequence alignment are used algorithms such as Needleman-Wunch and X-drop. For prediction of secondary RNA structure is used Zuker algorithm, that should remove some of Nussin algorithm weaknesses and Nussin algorithm itself. Recursion is showed by recursion trees. Dynamic programming is showed by score matrix. User also have ability to compare speed of both approaches for given sequences. It is programmed as web application, that run on client's side. This ensure easy availability.

Klíčová slova

dynamické programování, rekurze, bioinformatika, zarovnání sekvencí DNA, predikce sekundární struktury RNA

Keywords

dynamic programing, recursion, bioinformatics, DNA sequencing, prediction of secondary RNA structure

Citace

FRANĚK, Jaromír. *Efektivní varianty dynamického programování v bioinformatice*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Ivana Burgetová, Ph.D.

Efektivní varianty dynamického programování v bioinformatice

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením paní Ing. Ivany Burgetové, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jaromír Franěk
28. května 2020

Poděkování

Rád bych poděkoval paní Ing. Ivaně Burgetové, Ph.D. za cenné rady a připomínky k vypracování této práce, bez kterých by tato práce nemohla být v této podobě vypracována.

Obsah

Kapitola 1

Úvod

Tato práce se zabývá využitím efektivních variant dynamického programování v bioinformatice, tedy algoritmy využívajícími dynamické programování a jejich výhodám oproti méně efektivním algoritmům. Samotná bioinformatika je vědní disciplína, která se zabývá zejména analýzou velkého množství biologických dat na molekulární úrovni, jejich účelem a strukturou. Pro řešení úloh využívá poznatků a principů biologie, matematiky a informatiky. Je stále ve stavu rozvoje, a proto přibývá spousta nových problémů, pro které potřebujeme nalézt správné algoritmické řešení. I když nalezneme správné algoritmické řešení, tak nám stále bude trvat ještě spoustu let, než kompletně porozumíme výsledkům, proto se dá předpokládat, že její rozvoj bude ještě dlouho pokračovat. Prozatím, i přes její rozvoj a rozšíření pole působnosti, je stále jedním ze základních kamenů sekvenční analýza. Ta potřebuje většinou zpracovat veliké množství dat, a proto dbáme na rychlost a efektivnost zvoleného algoritmu. Sekvenční analýza nám pomáhá zjistit primární strukturu DNA nebo RNA. Po zjištění primární struktury bychom rádi zjistili i sekundární strukturu. To je ovšem komplikovanější a přináší méně přesné výsledky.

Je zde také představeno rekurzivní řešení a porovnáno s algoritmy Needleman-Wunch, X-drop, Greedy x-drop pro zarovnání sekvencí DNA. Dále je zde demonstrováno, proč samotné dynamické programování má vysokou časovou složitost a je porovnáno s variantami, které jsou rychlejší na úkor přesnosti. Ty jsou také porovnány s algoritmem FOGSAA nevyužívající dynamické programování. Dále jsou prezentovány algoritmy pro predikci sekundární struktury RNA. Jedná se o algoritmy Nussinov a Zuker, které jsou převážně porovnávány podle časové složitosti. Webová aplikace, která je výsledkem této práce je dostupná na adrese http://www.stud.fit.vutbr.cz/~xfrane16/.

Nakonec bude provedeno zhodnocení rychlosti a způsobu provedení jednotlivých algoritmů a jejich výsledků. Pro správné porovnání jsou algoritmy napsané ve stejném programovacím jazyce a testování je provedeno na stejném hardwaru. Dále tyto výsledky porovnáme mezi jednotlivými algoritmy.

Kapitola 2

Efektivita algoritmu

Tato kapitola se zabývá rychlostí algoritmu a jejím měřením. Jsou zde rozebírány složitosti algoritmu, zejména časová složitost, které je zde pro efektivní algoritmus nejdůležitějším kritériem.

Existuje mnoho způsobů jak vyřešit daný problém, ať se jedná o problém v oblasti informatiky nebo běžného života. Všechna řešení nejsou však stejně kvalitní a už vůbec nejsou stejně rychlá. Kritérií kvality algoritmu je hodně, my se zde zaměříme hlavně na rychlost. Řešení problému v rámci sekund a v rámci minut je veliký rozdíl. Samozřejmě pokud budeme řešit problém několik minut a kvůli našemu pomalejšímu způsobu řešení si přidáme pár sekund, potom zdržení není natolik veliké, aby nás to trápilo. Ke ztrátě by docházelo tehdy, kdybychom tento problém řešili opakovaně, to už by se nám začalo zdržení nepříjemně nasčítávat. Častým problémem v bioinformatice bývá také veliký vstup. Právě proto se snažíme dosáhnout co nejefektivnějších algoritmů. Převážně pomocí kvalitnější metody. Někdy jsme ale nuceni vyměnit rychlost za potencionální kvalitu výsledku.

2.1 Měření rychlosti

Pokud se budeme bavit o měření rychlosti, většinu lidí nejspíše napadne si vzít stopky a měřit ručně. Pokud bychom to převedli do informatiky, potom by to bylo způsobem, že si daný algoritmus naprogramujeme v určitém programovacím jazyce a následně změříme na počítači rychlost provedení. Takový způsob měření je možné v některých situacích použít, z teoretického hlediska je ve většině případů však nevhodný. S tím, že se výsledky budou lišit podle velikosti vstupních dat, se dá počítat a říct, že pro takto veliký vstup běžel algoritmus takhle dlouho. Výsledná rychlost se však bude lišit podle programovacího jazyka, hardwaru, nebo třeba i podle procesů, které poběží na pozadí.

Podstatně efektivnější způsob, který se nabízí, je měření pseudokódem. Tedy počtem provedených elementárních operací. Nejdříve se tedy musí definovat co je elementární operace. Povětšinou se jedná o operaci, kterou vykoná námi používaný procesor pomocí jedné, výjimečně několika instrukcemi. Jedním z příkladů může být sčítání nebo například podmíněný skok. Třeba takové odmocňování by se skládalo z více instrukcí, proto bychom to nepokládali za elementární operaci, ale museli bychom to na elementární operace rozložit. Čas vykonání elementární operace bude označen jako jednotkový. Tím se nám povede oprostit od situací, kdy bychom vyhodnotili rychlost horší, než ve skutečnosti je, například kvůli procesům běžícím na pozadí. Pro výpočet je ale také potřeba se přizpůsobit danému pro-

cesoru a jeho architektuře, protože co je a není elementární operace, se může lišit, zejména pokud pracujeme se speciálně navrženými procesory pro řešení daného problému. [7]

Takovýto způsob teoretického měření však z praktického hlediska nelze aplikovat vždy. Existují totiž algoritmy, které redukují počet provedených operací na základě dosud získaných výsledků. Jejich časová složitost se tedy bude lišit nejen po změně velikosti vstupu, ale také podle kvality vstupu, nebo třeba podle předchozího provedení. Takovéto algoritmy se v této práci vyskytují, příkladem je X-drop, kterým se budeme zabývat později.

2.2 Složitost algoritmu

Jednotlivé algoritmy mají různou složitost, která závisí na implementaci daného algoritmu. Kritérií složitosti algoritmu existuje více, může se jednat například o časovou, prostorovou nebo průměrnou složitost.

Časová složitost

Skutečná složitost algoritmu je závislá jak na implementaci algoritmu, vlastně se nedá v obecném případě přesně spočítat. Začali se proto používat odhady složitosti až na multiplikativní konstantu. Tyto odhady popisují růst složitosti vzhledem ke zvětšujícím se vstupům, ale neurčují konkrétní číslo. [10]

Abychom vyjádřili časovou složitost algoritmu, musíme nejdříve rozhodnout, co určuje velikost vstupu. Pokud se jedná o posloupnost čísel, potom velikost určuje jejich počet. V dalších případech může velikost vstupu určovat třeba délka řetězce, nebo v některých případech hodnota vstupního čísla. [7]
Pro určení časové složitosti tedy potřebujeme:

- Rozhodnout jak určit velikost vstupu.
- Určit maximální počet elementárních operací algoritmu.
- Ve výsledné formuli ponecháme pouze nejrychleji rostoucí člen a ostatní zanedbáme.
- Seškrtáme konstanty, kterými se zbytek funkce násobí.

Ve výsledku dostaneme časovou funkci. Podle algoritmu se liší od velmi komplikovaných po jednodušší. Nejčastěji se setkáváme s následujícími složitostmi:

- Konstantní - O(1).
- Lineární - O(n).
- Kvadratická - O(n^2).
- Kubická - O(n^3).
- Logaritmická - O($\log n$).
- Exponenciální - O(2^n).

Konstantní složitost vypadá ideálně, ale pro různé velikosti vstupu je v podstatě nedosažitelná, pokud tedy algoritmus se vstupem pracuje. Ze vzorců je vidět, že lineární složitost

je nejlepší pro větší velikost vstupu. Je to tím, že se zvětšujícím se vstupem její časová složitost roste nejpomaleji. Naopak exponenciální nám naroste dramaticky i pro malé zvýšení velikosti vstupu. [7]

Je možné namítnout, že se trh s počítači a jejich hardwarem velmi rychle vyvíjí. Podle Mooreova zákona se dokonce výkon hardwaru každé dva roky zdvojnásobí. To je naprostá pravda. Pokud ale náš algoritmus vyřeší problém za 2 roky, tak mu to po vylepšení hardwaru bude trvat pouze rok. To ale nestačí, pokud bychom tento problém rádi řešili opakovaně, přinejlepším denně. [7]

Vždy se snažíme o co nejefektivnější algoritmy, které mají nejpřijatelnější časovou složitost a hlavně u algoritmů pro které tato složitost neroste se vstupem drasticky. Je velmi veliký rozdíl mezi algoritmem, u kterého pro velikost vstupu o pěti znacích bude doba řešení pět sekund. Pokud se velikost vstupu zvýší o jeden znak, potom se i doba řešení zvýší o jednu sekundu a mezi algoritmem, pro který se doba řešení za stejných podmínek zdvojnásobí. Proto, pokud je to možné, tak se snažíme algoritmům s exponenciální složitostí vyhýbat. [7]

Prostorová složitost

Krom časové složitosti je možné zhodnotit efektivitu algoritmu pomocí prostorové složitosti. Ta se zaměřuje na využitou paměť. Opět má základní jednotku, kterou je elementární paměťová buňka. Příkladem této paměťové buňky může být byte nebo ukazatel. Prostorová složitost se vyjádří množstvím spotřebovaných buněk v závislosti na velikosti vstupu.[7] Vzhledem k dostupnému hardwaru nebývá prostorová složitost podstatná pro návrh algoritmu. V bioinformatice je důležitější časová a průměrná složitost.

Průměrná složitost

Zatímco v případě časové složitosti se bavíme o nejhorším možném případu, v případě průměrné složitosti se jedná o aritmetický průměr časových nebo prostorových nároků algoritmu přes všechny vstupy dané velikosti. Průměrná složitost může být tedy praktičtější, protože mohou existovat případy, kdy je algoritmus efektivní, krom několika vstupů, pro které je pomalý. [7] Tato složitost bude využívána u měření prováděných v této práci.

Kapitola 3

Dynamické programování

Tato kapitola se zabývá zejména dynamickým programováním. Na úvod je definována rekurze a její princip. Následně je zde uvedeno dynamické programování, jeho princip a výhody oproti rekurzi. V poslední je dynamické programování představeno na základních algoritmech.

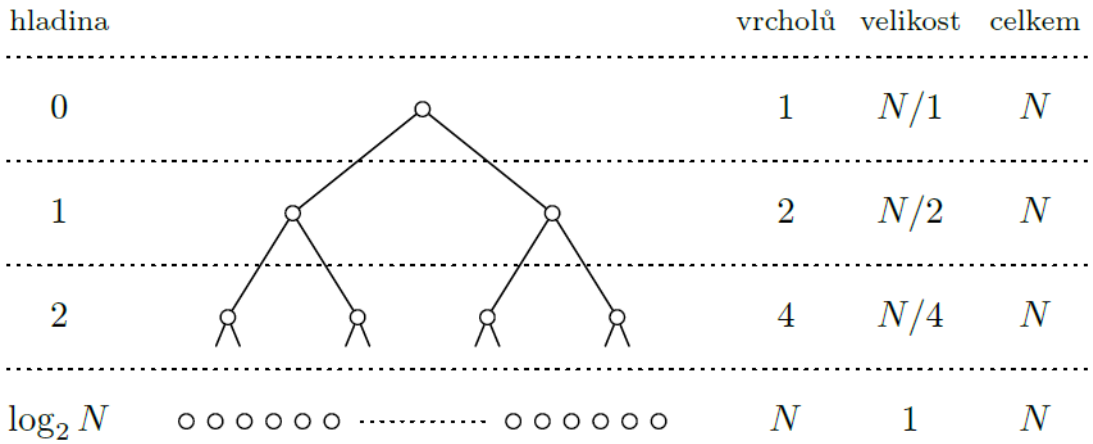
3.1 Rekurze

Je metodou definování určitého objektu pomocí sebe sama. V programování se jedná o funkci, která volá sama sebe, dokud není dosaženo ukončovací podmínky. Také platí, že každou rekurzi lze převést na iteraci a naopak.

S využitím rekurze je možné napsat kratší program než bez využití rekurze. Pokud je program správně napsaný, potom se jedná o přehlednější a čistší algoritmus. Rekurze má však úskalí. Pokud je napsaná nevhodně, může se stát program nepřehledným. Hledání a odstraňování chyb v takovýchto rekurzivních programech bývá obtížnější než v programech bez rekurze. Špatně použitá rekurze navíc vede k dalším podstatným nebezpečím. Často vede k programům, které jsou sice správné, ale jsou příliš pomalé. To je způsobeno tím, že je funkce volána pro stejné vstupy vícekrát. [11]
Existují dva základní typy rekurze:

- Přímá - Funkce volá sama sebe.
- Nepřímá - Více funkcí utvoří kruh volání.

Rekurze se dá například vyjádřit pomocí rekurzivních stromů. Přiklad takového stromu je na obrázku 3.1. [7]



Obrázek 3.1: Příklad rekurzivního stromu algoritmu MergeSort [7]

3.2 Princip dynamického programování

Dynamické programování je metoda řešící problémy znovupoužitím výsledků z už vyřešených podproblémů. [4] Na rozdíl od rekurze eliminuje opětovné řešení stejných podproblémů, díky pomocné struktuře, kterou může být například pole už získaných hodnot. Název vymyslel Richard Bellman, když zkoumal vícekrokové plánování, kde optimální volba kroku je závislá na krocích předchozích. [7]
Základní princip, který dynamické programování využívá:

- Vezmeme rekurzivní algoritmus, který je pomalý a který opakovaně řeší stejné podproblémy (většinou se bude jednat o rekurzivní algoritmus, může se ale také jednat o iteraci).
- Identifikujeme opakované výpočty podproblémů.
- Vytvoříme si strukturu, do které si budeme po provedeném výpočtu ukládat hodnoty.
- Před provedením výpočtu se podíváme do struktury, zda už nemáme spočítaný výsledek.

Takovémuto postupu se říká kešování. Jedná se o termín používaný v informatice pro paměť pro často používaná data. V souvislosti se zrychlováním rekurze se tomuto postupu také může říkat memoizace neboli kešování. [7]

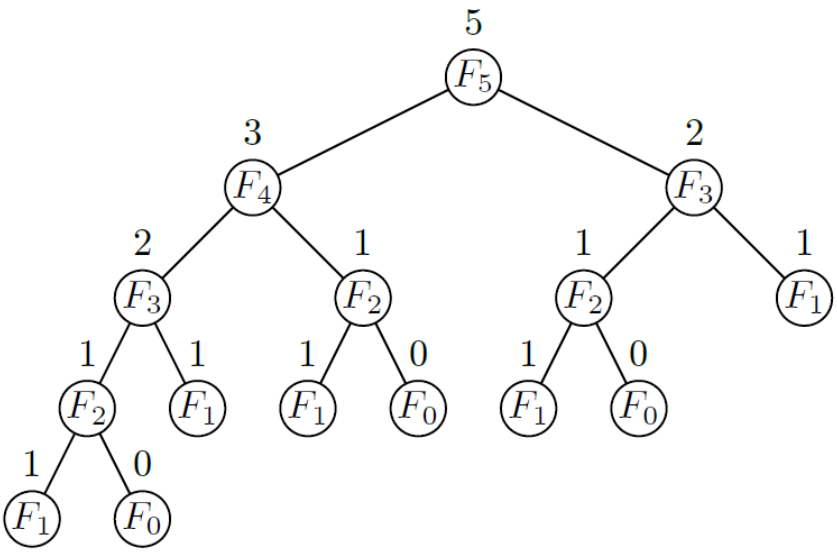
3.3 Příklady algoritmů

Zde budou představeny příklady algoritmů využívajících rekurze nebo dynamické programování. Prvně je řešení problému představeno rekurzí a poté dynamickým programováním. Řešenými problémy budou Fibonacciho čísla a editační vzdálenost.

3.3.1 Fibonacciho čísla rekurzí

Výpočet n-tého Fibonacciho čísla F_n. Budeme předpokládat, že n značící velikost čísla, které budeme počítat je větší nebo rovno nule.
Algoritmus Fibo(n):

- Pokud $n <= 1$, vrátíme n.
- Jinak vrátíme Fibo($n-1$) + Fibo($n-2$).



Obrázek 3.2: Rekurzivní strom pro výpočet Fibonacciho čísel [7]

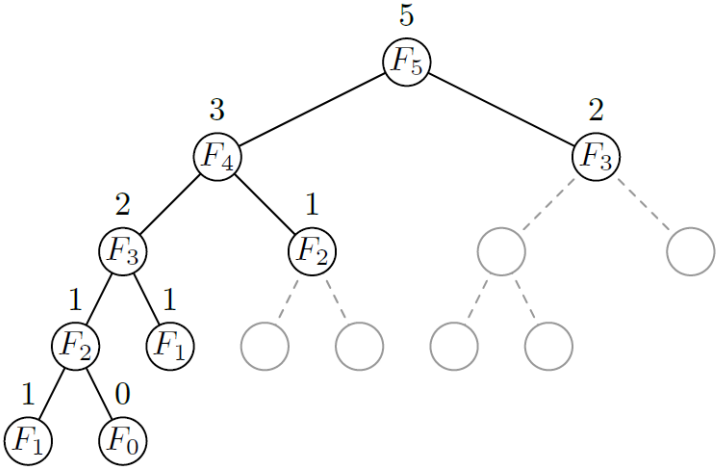
Pomocí stromu rekurze, vyobrazeném na obrázku 3.2, zjistíme, že pro výpočet libovolného vnitřního vrcholu potřebujeme součet z jeho synů. Přitom takovýto výpočet provádíme opakovaně, tedy i pro stejné vrcholy nacházející se v jiné části stromu. Složitost algoritmu je tedy přinejmenším exponenciální, přestože voláme funkci pouze pro argumenty v rozsahu 0 - n. Pokud bychom však nepočítali opakovaně totéž, potom by mohla být složitost algoritmu přijatelnější. [7]

3.3.2 Fibonacciho čísla dynamickým programováním

Vylepšení předchozího algoritmu pomocí tabulky pro vypočítané hodnoty. Nazveme ji T. Opět n musí být větší než nula.
Algoritmus Fibo2(n):

- Pokud $n <= 1$, vrátíme n.
- Pokud je $T[n]$ definováno, vrátíme $T[n]$.
- Jinak do $T[n]$ uložíme Fibo2($n-1$) + Fibo2($n-2$).
- Vrátíme $T[n]$.

Při každém volání se tedy podíváme do tabulky, zda se zde již nenachází vypočítaná hodnota. Pokud ano, pak ji nebudeme znovu počítat, ale vezmeme si ji z tabulky, jinak tuto hodnotu vypočítáme a uložíme do tabulky. K rekurzi dochází pouze tehdy, pokud se snažíme vypočítat dosud neznámou hodnotu. Strom bude mít tedy pouze $3n$ vrcholů. Tento algoritmus by šlo ještě upravit tak, aby rekurze nebyla vůbec přítomná. Tabulky bychom vyplňovali postupně. Tohle je však jako příklad dostatečné a pokračovat není třeba. [7]



Obrázek 3.3: Strom pro výpočet Fibonacciho čísel po využití dynamického programování [7]

3.3.3 Editační vzdálenost

Editační vzdálenost je definována jako minimální počet operací, které musí být provedeny mezi dvěma řetězci, aby byly totožné. Samotnou operací je vložení, mazání nebo změna znaku. Můžeme si představit, že procházíme řetězec x od začátku do konce a postupně ho přetváříme na řetězec y. Editační vzdálenost budeme značit $L(x; y)$.

	p	o	t	e	m	n	í	k	
p	3	4	4	4	4	4	5	6	7
o	4	3	3	3	3	3	4	5	6
u	4	3	3	2	2	2	3	4	5
t	4	3	2	2	1	1	2	3	4
n	5	4	3	2	1	0	1	2	3
í	6	5	4	3	2	1	0	1	2
k	7	6	5	4	3	2	1	0	1
	8	7	6	5	4	3	2	1	0

Obrázek 3.4: Tabulka pro slova potemník a poutník [7]

3.3.4 Editační vzdálenost rekurzí

Možnosti pro první symbol: [7]

- Pokud $x_1 = y_1$, potom znak nebudeme měnit. Pak $L(x; y) = L(x_2...x_n; y_2...y_m)$.
- Pokud znak x_1 zaměníme za y_1. Pak $L(x; y) = 1 + L(x_2...x_n; y_2...y_m)$.
- Pokud znak x_1 smažeme. Pak $L(x; y) = 1 + L(x_2...x_n; y_1...y_m)$.
- Pokud na začátek vložíme y_1. Pak $L(x; y) = 1 + L(x_1...x_n; y_2...y_m)$.

Vstupem boudou řetězce $x_i...x_n, y_j...y_m$.
Algoritmus Edit(i, j): [7]

- Pokud $i > n$, vrátíme $m - j + 1$.
- Pokud $j > m$, vrátíme $n - i + 1$.
- $R_z = \text{Edit}(i + 1, j + 1)$.
- Pokud $x_i \neq y_j$, $R_z = R_z + 1$.
- $R_s = \text{Edit}(i + 1, j) + 1$.
- $R_v = \text{Edit}(i, j + 1) + 1$.
- Vrátíme min (R_z, R_s, R_v).

Časová složitost algoritmu Edit je exponenciální. Ta je způsobena opětovným voláním funkce pro stejné hodnoty parametrů. [7]

3.3.5 Editační vzdálenost dynamickým programováním

Vytvoříme si tabulku pro ukládání hodnot výsledků podproblémů. Příklad takovéto tabulky je uveden na obrázku 3.4. Dále otočíme směr výpočtu, budeme tedy postupovat od nejkratších sufixů k delším.
Vstupem boudou řetězce $x_i...x_n, y_j...y_m$.
Algoritmus Edit2(i, j): [7]

- Pro $i = 1, ..., n + 1$, bude $T[i, m + 1] = n - i + 1$.
- Pro $j = 1, ..., m + 1$, bude $T[n + 1, j] = m - j + 1$.
- Pro $i = n, ..., 1$:
- Pro $j = m, ..., 1$:
- Pokud $x_i = y_j$, pak $d = 0$, jinak $d = 1$.
- $T[i, j] = \min(d + T[i + 1, j + 1]; 1 + T[i + 1, j]; 1 + T[i, j + 1])$.

Algoritmus Edit2 už neprovádí opětovně stejné výpočty. Jedná se tedy o podstatně rychlejší algoritmus běžící v čase O(nm).

Grafové řešení editační vzdálenosti

Editační vzdálenost je možné popsat pomocí orientovaného grafu. Vrcholy jsou pozice znaků v řetězcích. Hrany budou možné operace. Každá cesta z vrcholu (1, 1) do vrcholu (n + 1, m + 1) je jednou z možných posloupností operací. [7]

Kapitola 4

Bioinformatika

Tato kapitola popisuje základní funkcionalitu DNA a RNA a samotný význam zarovnání a predikce struktury. První část se soustředí zejména na funkcionalitu. V druhé části jsou popsány způsoby zarovnání sekvencí a predikce struktury, které zde budou podrobněji probírány.

4.1 Úvod do DNA a RNA

DNA (deoxyribonucleic acid) slouží hlavně k uchování informací o genetických instrukcích. Vyskytuje se v buňkách všech živých organismů. Všechny informace potřebné k rozmnožení a fungování organismu jsou uchovány ve velmi malém množství molekul DNA. Tyto molekuly se nazývají genom organismu. Každá molekula DNA je zkopírována než dojde k buněčnému dělení a její kopie zdědí nově vytvořené buňky. Také proteiny jsou vytvářeny pomocí DNA. Jedná se o molekuly, které jsou nezbytné pro mezibuněčnou komunikaci, metabolismus a život samotný. Pro některé procesy jsou využívány také molekuly RNA.

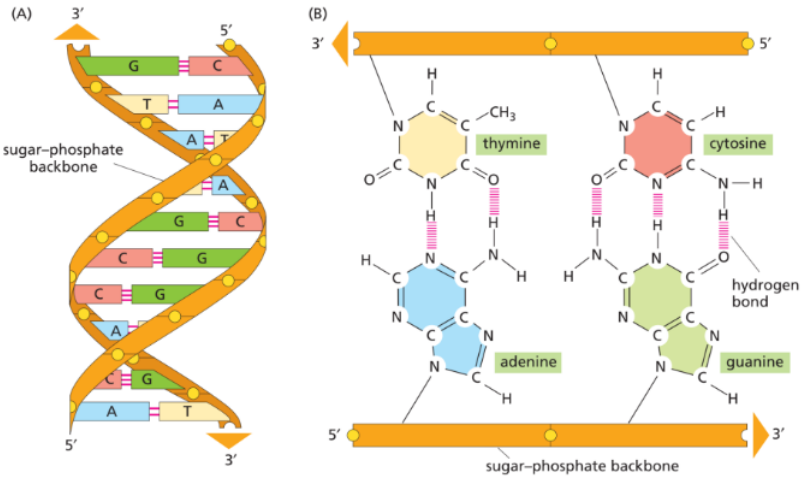
4.1.1 Struktura DNA a RNA

Základní chemická struktura je jednoduchá, hlavně pokud bereme v potaz celkovou roli DNA. Obsahují dusíkatou bázi, molekulu cukru a fosfátovou skupinu. Podle použité báze rozlišujeme adenin, guanin, thymin a cytosin (A, G, T, C), kde první dva patří mezi puriny a další mezi pyrimidi. V případě RNA existuje uracil (U), který slouží jako náhrada za thymin (T). Nukleotidy obsahují bázi, které je spojená s pěti uhlíkovými cukry a fosfátovou skupinou. V DNA je cukrem deoxyribóza a v případě RNA se jedná o ribózu. [16]

Dvoušroubovice DNA

Nobelovu cenu dostal James Watson a Francis Crick za kresbu dvoušroubovice DNA, podle výzkumu Rosalind Franklin a dalších. V dvoušroubovici se dvě vlákna DNA otáčí okolo společné osy. Tato dvoušroubovice je ilustrována na obrázku 4.1 Všechny báze jsou umístěny uvnitř a fosfátově spojené cukry tvoří oporu zvenku. Důležité je, že se báze vážou mezi nukleotidy z různých vláken podle jistých pravidel, známých jako Watson-Crick base-pairing. Určitý purin se váže na určitý pyrimid. Tedy cytosin se váže na guanin a adenin na thymin. Každé vlákno obsahuje bázovou sekvenci, která je komplementární k té druhé. Vlákna jdou opačnými směry. Báze jsou spojeny pomocí vodíkových můstků. Tedy pokud

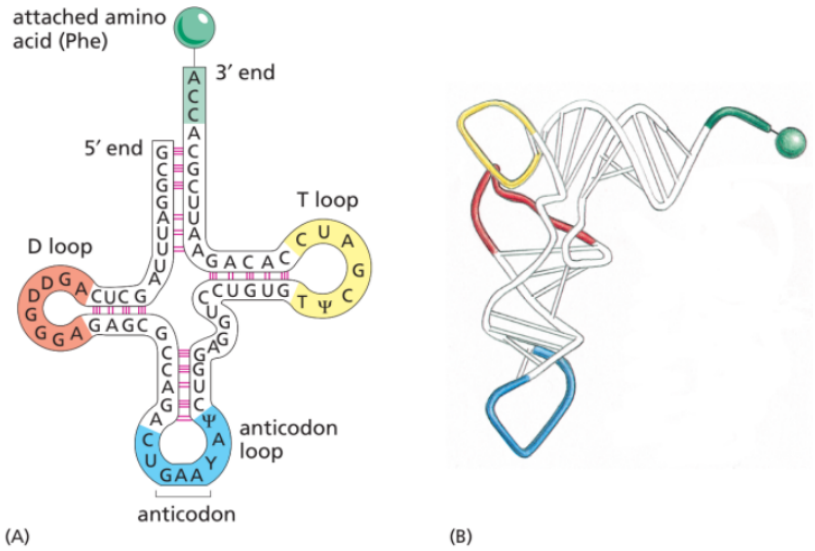
máme jednu sekvenci, tak jsme schopni odvodit i sekvenci druhého vlákna. V případě nutnosti jsou vlákna schopny se rozvinout a vytvořit dvě nové DNA molekuly a tím zachovají genetickou informaci. [16]



Obrázek 4.1: Dvoušroubovicová struktura DNA [16]

Struktura RNA

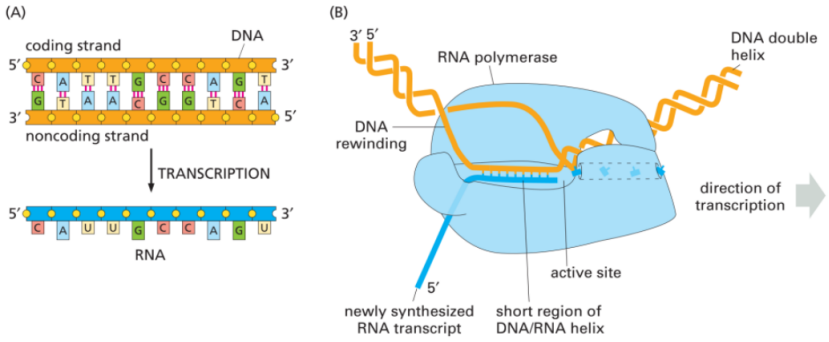
RNA molekuly jsou pouze jedno vláknové, neobsahují tedy dvoušroubovici a jsou kratší než molekuly DNA. RNA je daleko flexibilnější. Vlákno molekuly RNA je obvykle sbaleno do tvaru, kdy jsou jeho části ohnuty tak, aby mohly interagovat spolu s jinými částmi vlákna. Záleží tedy taky na tom, jak se vlákno vlní. [16]



Obrázek 4.2: Struktura RNA [16]

Funkce DNA a RNA

Za centrální dogma je považován vztah mezi DNA, RNA a proteiny. Udává směr toku genetických informací z DNA do RNA pro vytvoření proteinů. Proteiny jsou hlavní fungující součástí organismu a jeho procesů. Sekvence v DNA udává, jak bude vypadat sekvence v proteinech. Vytváření nové molekuly je podobné tvoření nových molekul DNA, ale v tomto případě není vytvořená nová molekula DNA, ale RNA. K vytvoření je použita pouze potřebná část jednoho vlákna, jak je představeno na obrázku 4.3. RNA vytvořená z genu pro kódování proteinů se nazývá messenger RNA (mRNA). Dalšími typy RNA jsou ribosomální RNA (rRNA) a transfer RNA (tRNA). [16]



Obrázek 4.3: Vytváření RNA z DNA [16]

4.2 Zarovnání sekvencí DNA

Míra shody znaků na zarovnaných sekvencích představuje míru příbuznosti, pomáhá nám identifikovat stejné části dvou nebo více DNA sekvencí. Identifikace podobných sekvencí nám může například pomoci zkonstruovat fylogenetický strom a vztahy mezi jednotlivými druhy v tomto stromu. Metoda zarovnání není přímočará. Musíme vzít v potaz nejen mnoho dostupných informací, ale i možnost, že se jedna ze sekvencí nebo obě změnili v průběhu evoluce. Tedy znak mohl být přidán nebo odebrán. Tyto změny mohou dobře maskovat podobnost sekvencí a komplikovat naši snahu o určení podobnosti. [16]

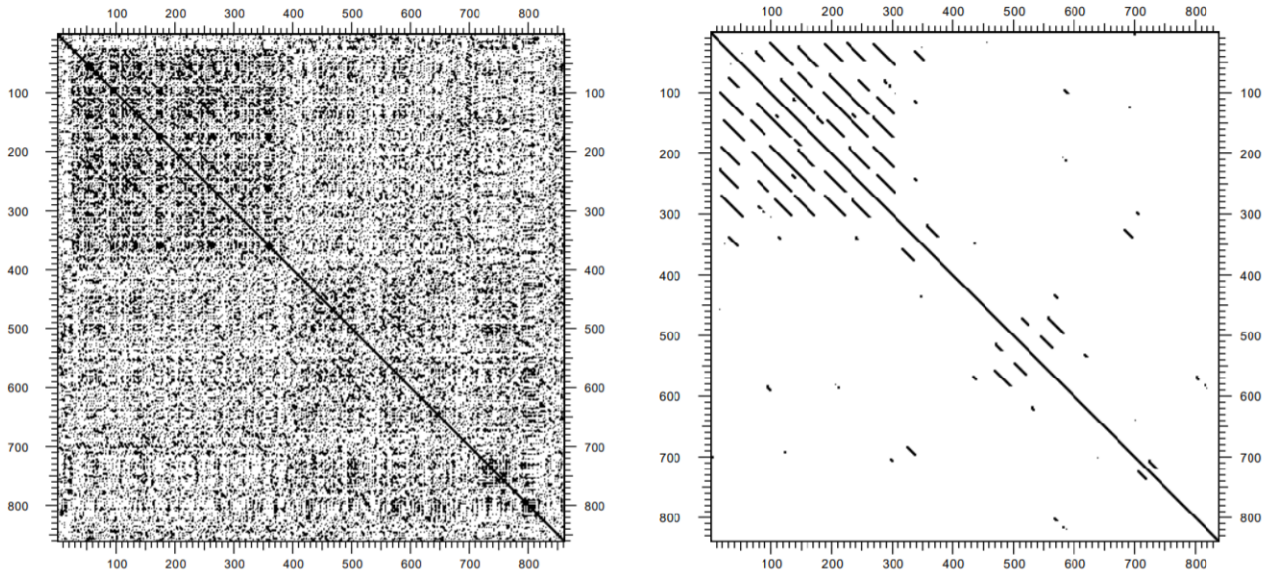
Možností zarovnání je mnoho, potřebujeme ale co nejefektivnější metody pro určení optimálního skóre. Dnes existují algoritmy, které nalezení optimálního skóre mezi dvěma sekvencemi garantují. K vyhledávání zarovnání ve frontě sekvencí s celou databází sekvencí, se používají převážně aproximační metody urychlující vyhledávání. [16] Mezi tyto metody patří například FASTA a BLAST. V této kapitole se později budeme zabývat efektivními algoritmy pro zarovnání sekvencí, mezi ty patří X-drop, Greedy X-drop, FOGSAA. Nejdříve se však musíme seznámit se základními principy zarovnání dvou sekvencí.

4.2.1 Dot plot

Jedná se o bodovou matici. Je to jedna z nejzákladnějších metod pro porovnání dvou sekvencí. Jedná se však pouze o grafické porovnání. Základem je matice, kde jeden řetězec je veden po ose X, zatímco druhý je veden po ose Y. Znaky X a Y jsou porovnány na jejich průsečících. Pokud jsou znaky na pozici průsečíku řádku a sloupce shodné, potom se zde

vloží bod. Výsledné diagonály značí úseky, ve kterých jsou sekvence podobné. Další body mimo diagonály pouze poukazují na náhodnou shodu znaků, nedochází však k podobnosti. Používá se například pro nalezení opakování úseků v sekvencích. Nevýhodou je, že se pouze jedná o orientační grafickou reprezentaci, nevíme tedy, jak jsou výsledné sekvence zarovnány. Dále nejsme schopni přesně ohodnotit podobnost sekvencí. Časová Složitost algoritmu je $O(n^2)$. V základní verzi dochází ke vzniku šumu. [16]

Nevýhoda šumu se dá eliminovat pomocí filtru v podobě posuvného okénka. Okénko o námi zvolené velikosti se posouvá jak ve směru sloupců, tak ve směru řádků. V okénku se porovnávají všechny znaky, a pokud je z nich shodných alespoň tolik, kolik je námi určeno, potom je na pozici začátku okénka vložen bod. Pomocí toho jsou vykresleny pouze diagonály, které přesahují námi zvolenou minimální velikost. [16]



Obrázek 4.4: Dot plot s posuvným okénkem [8]

4.2.2 Jednoduchá metoda zarovnání

Metoda bere v potaz pouze mutace (záměnu nukleotidů, ke které dochází při evoluci), nebere v potaz vložení nebo odstranění znaku. Dále se předpokládá, že menší z řetězců vznikne z většího vložením mezer na začátek nebo konec. [8] Principem je tedy to, že kratší řetězec posunujeme a porovnáváme s delším, kde každé porovnání ohodnotíme počtem shodujících a neshodujících se znaků. Příklad tohoto principu je představen na obrázku 4.5, není ale použitelný pro reálné případy.

$$S = \sum_{i=1}^{n} \begin{cases} \text{odměna} & \text{pro shoda} \\ 0 & \text{pro neshoda} \end{cases} \tag{4.1}$$

Zarovnané sekvence								Skóre
C	C	G	A	G	C	G	C	3
C	C	T	A	C	G	-	-	
1	1	0	1	0	0			

Zarovnané sekvence								Skóre
C	C	G	A	G	C	G	C	3
-	C	C	T	A	C	G	-	
	1	0	0	0	1	1		

Zarovnané sekvence								Skóre
C	C	G	A	G	C	G	C	0
-	-	C	C	T	A	C	G	
		0	0	0	0	0	0	

Obrázek 4.5: Příklad zarovnání sekvencí jednoduchou metodou CCGAGCGC a CCTACG

4.2.3 Metoda zarovnání s vložením mezer

Pro reálné případy je potřeba brát v úvahu vložení a odstranění znaku, ke kterému může dojít během evoluce. To ovšem zvyšuje počet možností. Některé z těchto možností jsou zobrazeny na obrázku 4.6. Mezery navíc musí být rozmístěny tak, aby bylo dosaženo co nejvyšší skóre. Snažit se porovnat dvě sekvence pomocí vložení spousty mezer je nesmyslné a výsledkem nebude validní informace. [8]

$$S = \sum_{i=1}^{n} \begin{cases} \text{odměna} & \text{pro shoda} \\ 0 & \text{pro neshoda} \\ \text{penalizace} & \text{pro } S_1[i] = \text{'-'} \ || \ S_2[i] = \text{'-'} \end{cases} \tag{4.2}$$

Zarovnané sekvence								Skóre
C	C	G	A	G	C	G	C	1
C	C	T	A	C	G	-	-	
1	1	0	1	0	0	-1	-1	

Zarovnané sekvence								Skóre
C	C	G	A	G	C	G	C	-1
-	C	C	T	-	A	C	G	
-1	1	0	0	-1	0	0	0	

Zarovnané sekvence								Skóre
C	C	G	A	G	C	G	C	3
C	C	T	A	-	C	G	-	
1	1	0	1	-1	1	1	-1	

Obrázek 4.6: Příklad zarovnání sekvencí s vložením mezer CCGAGCGC a CCTACG (pouze 3 z 28 možných kombinací)

Dále je tedy potřeba rozlišovat typy mezer. Strukturální analýza ukázala, že menší počet delších mezer je pravděpodobnější, než veliký počet menších. Správně tedy musíme rozlišovat mezi počáteční mezerou a mezerou rozšiřovací. Lépe tedy budou hodnoceny zarovnání s menším počtem delších mezer. Kvalita výsledku pak bude záviset na vybrané skórovací funkci. [16]

Řešením je rozdílná penalizace různě velikých mezer podle principu:

$$S = \sum_{i=1}^{n} \begin{cases} \text{odměna} & \text{pro shoda} \\ 0 & \text{pro neshoda} \\ \text{penalizace} & \text{pro pokračující mezera} \\ \text{větší penalizace} & \text{pro počáteční mezera} \end{cases} \tag{4.3}$$

4.2.4 Skórovací matice

Pro správné zarovnání je kromě mezer také potřeba brát v úvahu záměnu mezi znaky. Proto byly vytvořeny skórovací matice, které definují ohodnocení těchto záměn. Záměna mezi puriny a pyrimidiny jsou pravděpodobnější, než záměna mezi purinem a pyrimidinem. Matice, které tuto skutečnost berou v úvahu, se označují jako matice s traverzí. Příklad takové matice je na obrázku 4.7. [8] Nejúspěšnější z těchto matic jsou založeny na znalostech o evoluci. Rozhodnout, jakou matici zvolit pro daný případ, není jednoduché, protože žádná není vytvořená pro všechny případy. Někdy potřebujeme hodnotit evolučně velice příbuzné sekvence, jindy se zase snažíme najít podobnosti ve velice vzdálených sekvencích. V prvním případě bychom rádi hodnotili přísně, tedy dávali vysoké skóre pouze v perfektní shodě, v druhém případě se snažíme nalézt alespoň podobnost.

	A	T	C	G
A	1	-5	-5	-1
T	-5	1	-1	-5
C	-5	-1	1	-5
G	-1	-5	-5	1

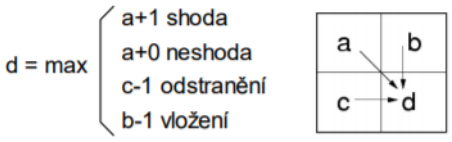
Obrázek 4.7: Příklad skórovací matice s traverzí

Často používané typy matic jsou PAM (Point Accepted Mutation) matice. Ty byly navrženy Margaret Dayhoff a jejími spolupracovníky. Jsou sestaveny tak, že jejich vzájemným vynásobením získáme matice pro různě odlišné sekvence. Jsou ohodnoceny podle předpokládaného počtu mutací. PAM-1 je určena pro velmi blízké sekvence, zatímco PAM-1000 pro sekvence značně rozdílné. Jedna z nejčastěji používaných je PAM-250. [8]

Dalším často používaným typem matic jsou BLOSUM matice. Jsou modernější a používají více lokálních zarovnání místo globálního zarovnání. Jsou získány pomocí změny procenta, podle kterého se shlukují sekvence do podobných skupin. Tedy BLOSUM-XX, kde XX označuje procento míry podobnosti sekvencí. Jedna z nejpoužívanějších je BLOSUM-62. [8]

4.2.5 Needleman-Wunch

Jedná se o algoritmus používaný pro globální zarovnání dvou sekvencí. Tedy sekvence jsou zarovnány jako celek a každý výskyt mezer je penalizován. Mezery nesmí být zarovnány k sobě. Algoritmus je založen na principu dynamického programování a byl pojmenován po svých autorech. I přes to, že byl publikován v roce 1970, je stále důležitým jádrem moderních zarovnávacích algoritmů a metod. Jedná se o problém podobný editační vzdálenosti, pracuje tedy na stejném principu. Problém editační vzdálenosti je popsán v kapitole 3.3.3. Sekvence jsou poskládány v tabulce tak, že řádky reprezentují jednu sekvenci a sloupce druhou. První řádek a sloupec je nastaven tak, že se postupně zvyšuje penalizace za vložení mezery. Vnitřní buňka se nastaví jako maximum ze tří možností ve vzorci 4.8: [8]



Obrázek 4.8: Příklad penalizace [8]

- Převzetím hodnoty zleva s penalizací za přidání mezery.
- Převzetím hodnoty shora s penalizací za přidání mezery.
- Převzetím hodnoty z levého horního pole s inkrementací hodnoty pokud dojde ke shodě znaků.

Tímto získáme tabulku jako na obrázku 4.9. Hodnota v pravém spodním rohu je hodnota skóre ideálního zarovnání sekvencí.

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

Obrázek 4.9: Příklad vyplnění tabulky

Zpětným průchodem z pravého spodního rodu k levému hornímu získáme tvar ideálního zarovnání sekvencí. Zpětný průchod je zobrazen na obrázku 4.10.

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

Obrázek 4.10: Příklad zpětného průchodu tabulkou

Při zpětném průchodu se rozhodujeme podle těchto možností: [8]

- Posun zpět po diagonále.
- Posun nahoru vložením mezery do horizontálního řetězce.
- Posun doleva vložením mezery do vertikálního řetězce.

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

Obrázek 4.11: Výsledné zarovnání sekvencí ze zpětného průchodu na obrázku 4.10

Vybereme tu buňku, pomocí které byla vypočtena hodnota této buňky. Ve výsledku můžeme dojít k více ekvivalentním možnostem. Samotná metoda má $O(n^2)$ časovou složitost za předpokladu, že $n >= m$ a zpětný průchod má složitost $O(n + m)$.

Implementovaný algoritmus na obrázku 4.12: Na vstupu jsou sekvence M a N o délce m a n. Dále máme pole S. Penalizace mezery se značí g, $s(M_i, N_j)$ představuje ohodnocení zarovnání znaků na pozicích i a j.

Inicializace prvního řádku a sloupce
for i v rozsahu 1 .. m **do**
  for j v rozsahu 1 .. n **do**

$$S_{i,j} \leftarrow Max \begin{cases} S_{i-1,j} + g \\ S_{i,j-1} + g \\ S_{i-1,j-1} + s(M_i, N_j) \end{cases}$$

  end for
end for
výsledné skóre je $S_{n,m}$

Obrázek 4.12: Needleman-Wunch algoritmus

Needleman-Wunch s penalizací rozšiřující mezery

V této verzi algoritmu musí být prozkoumány všechny možné velikosti mezer. První sekvence označené Q_{seq} a druhé sekvence označené Y_{seq}. Budeme předpokládat, že penalizaci mezer vypočítáme funkcí $g(n_{gap})$ o délce n. Penalizaci rozšíření mezery budeme označovat jako e. Jedním ze způsobů je úprava vzorce.

$$F(i,j) = Max \begin{cases} F_{i-1,j-1} + S(Q_m, Y_n) \\ [F_{i-n_{gap1},j} + g(n_{gap1})]_{1 \leq n_{gap1} \leq i} \\ [F_{i,j-n_{gap2}} + g(n_{gap2})]_{1 \leq n_{gap2} \leq j} \end{cases} \tag{4.4}$$

Složitost algoritmu se změní na mn^2 za předpokladu, že $n > m$. Další možností je inicializace dalších matic, kde si budeme ukládat mezery.

$$Q(i,j) = Max \begin{cases} F(i-1,j) - g \\ Q(i-1,j) - e \end{cases} \tag{4.5}$$

$$Y(i,j) = Max \begin{cases} M(i,j-1) - g \\ Y(i,j-1) - e \end{cases} \tag{4.6}$$

$$M(i,j) = Max \begin{cases} F(i-1,j-1) + S(Q_i, Y_j) \\ Q(i-1,j-1) + S(Q_i, Y_j) \\ Y(i-1,j-1) + S(Q_i, Y_j) \end{cases} \tag{4.7}$$

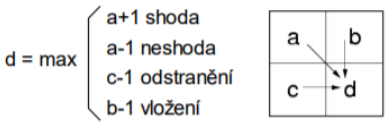
Matice F představuje skóre zarovnání. Matice Q si uchovává všechny mezery nalezené v Q_{seq} a matice Y v Y_{seq}. Algoritmus by měl opět běžet se složitostí $O(n^2)$ za předpokladu, že $n >= m$. [16]
Tyto úpravy však podstatně komplikují zpětný průchod.

4.2.6 Smith-Waterman

Jedná se o algoritmus určený pro lokální zarovnání dvou sekvencí. Hledáme tedy všechny možné podsekvence a hodnotíme kvalitu zarovnání. Je opět založen na principu dynamic-

kého programování. Nese název po svých autorech a tvoří jeden ze základních kamenů bioinformatiky. Byl publikován v roce 1981.

Sekvence jsou opět poskládány v tabulce tak, že řádky reprezentují jednu sekvenci a sloupce druhou. První řádek a sloupec je nastaven na nuly. Vnitřní buňka se nastaví jako maximum podle pravidel na obrázku 4.13: [8]



Obrázek 4.13: Příklad penalizace

- Převzetím hodnoty zleva s penalizací za přidání mezery.
- Převzetím hodnoty shora s penalizací za přidání mezery.
- Převzetím hodnoty z levého horního pole s inkrementací hodnoty pokud dojde ke shodě znaků, nebo penalizací pokud dojde k záměně.
- Pokud by maximum z předchozích možností vedlo na zápornou hodnotu, potom je hodnota buňky nastavena na nulu.

Zpětný průchod se spouští od maximálních hodnot skóre v tabulce podle stejného principu jako u Smith-Waterman. Příklad této matice a zpětného průchodu je na obrázku 4.14.

		A	A	C	C	T	A	T	A	G	C	T
	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	0	1	0	0
C	0	0	0	1	1	0	0	0	0	0	2	1
G	0	0	0	0	0	0	0	0	0	1	1	1
A	0	1	1	0	0	0	1	0	1	0	0	0
T	0	0	0	0	0	1	0	2	1	0	0	1
A	0	1	1	0	0	0	2	1	3	2	1	0
T	0	0	0	0	0	1	1	3	2	2	1	2
A	0	1	1	0	0	0	2	2	4	3	2	1

Obrázek 4.14: Příklad zarovnání sekvencí AACCTATAGCT a GCGATATA

Smith-Waterman s penalizací rozšiřující mezery

S-W metoda s penalizací rozšiřující mezery byla představena v roce 1982. Je považována za model, který lépe vystihuje podobnost sekvencí přiřazováním větší penalizace za první nalezenou mezeru a menší penalizaci za mezery, které za ní následují. Pro správnou funkci metody jsou potřeba tři různé matice. Skórovací matice $F(M+1, N+1)$, kde M je velikost první sekvence označené Q_{seq} a N představuje velikost druhé sekvence označené Y_{seq}. Další dvě matice o velikosti (M, N) Q a Y. [12] Za předpokladu, že smazání nemůže následovat za vložením a naopak, skóre zarovnání je vypočítáno:

$$Q(i,j) = Max \begin{cases} 0 \\ F(i-1,j) - g \\ Q(i-1,j) - e \end{cases} \tag{4.8}$$

$$Y(i,j) = Max \begin{cases} 0 \\ M(i,j-1) - g \\ Y(i,j-1) - e \end{cases} \tag{4.9}$$

$$M(i,j) = Max \begin{cases} 0 \\ F(i-1,j-1) + S(Q_i, Y_j) \\ Q(i-1,j-1) + S(Q_i, Y_j) \\ Y(i-1,j-1) + S(Q_i, Y_j) \end{cases} \tag{4.10}$$

Matice F představuje skóre zarovnání. Matice Q si uchovává všechny mezery nalezené v Q_{seq} a matice Y v Y_{seq}. [12]

Optimalizace pro ušetření prostoru

Klasická Needleman-Wunch nebo Smith-Waterman metoda počítá celou matici, pokud mají ale obě sekvence příliš velikou délku, potom může samotná matice zabírat příliš mnoho paměti. Algoritmus tedy můžeme modifikovat, aby ukládal dva řádky matice. Na výpočet jednoho pole nám totiž stačí pouze znalost výsledků předchozího řádku. Toto ovšem komplikuje získání zarovnání. [16]
Postup:

1. Inicializujeme první řádek podle pravidel zvoleného algoritmu.
2. Nyní vypočítáme všechny elementy druhého řádku.
3. Následně je první řádek uložen a nahrazen druhým. A pokud nejsme na konci matice, pak se vrátíme na druhý bod.

Díky tomuto postupu získáme hodnotu zarovnání a ušetříme paměť, získání samotného zarovnání ale není možné, pokud si nebudeme hodnoty řádků ukládat. Existuje více způsobů jak ušetřit paměť. Většina ostatních pracuje s aproximací. [16]

4.2.7 FASTA

Jedná se o heuristickou metodu sloužící pro prohledávání rozsáhlých databází. FASTA zrychluje na úkor přesnosti. Samotné dynamické programování využívá až v posledním

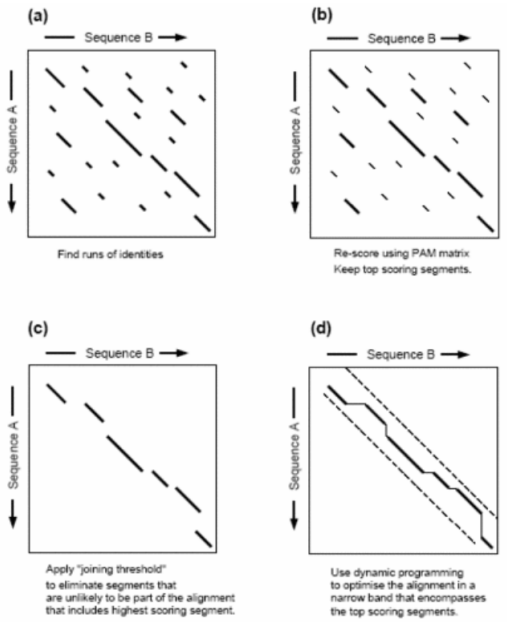
kroku metody, kde už dojde k porovnání sekvencí, které byly zhodnoceny předchozími kroky jako nejpodobnější.

FASTA v prvním kroku rozdělí sekvenci na slova. Parametr k udává velikost slov, většinou se jedná o hodnotu 6. Čím menší hodnota, tím bude větší přesnost. Základním cílem je vytvořit list všech začínajících pozic různých slov v sekvenci. Prvně tedy přiřadíme jednotlivým slovům hodnotu. To je možné udělat tak, že každá báze dostane hodnotu.

Pro sekvenci o délce L zde bude $L - k + 1$ slov. Ty seřadíme podle hodnot. Abychom zjistili, kde každé slovo začíná, použijeme zřetězení. Potřebujeme dvě pole a o velikosti všech možných slov a b o velikosti $L - k + 1$. Položka v a ukazuje na první výskyt daného slova v sekvenci, nebo značí, že se slovo v sekvenci nevyskytuje. Položka v b ukazuje na následný výskyt slova, nebo značí, že se slovo v sekvenci už neopakuje. Zřetězená pole zjednodušují vyhledání všech identických slov. Všechna seřazená slova jsou pozitivně ohodnocena. Zbytky mezi nimi jsou penalizovány poměrem jejich délky a délky diagonály $g(l)$. Algoritmus si drží ohodnocení každé diagonály. [9] Ohodnocení je vždy aktualizováno podle vzorce:

$$S_{i-j} = S + \max \begin{cases} S_{i-j} + g(l) \\ 0 \end{cases} \tag{4.11}$$

V druhém kroku vezmeme 10 nejlépe hodnocených zarovnání. A přehodnotíme je pomocí PAM250. Tak získáme kvalitnější ohodnocení. V třetím kroku spojíme nejlépe ohodnocené zarovnání do jednoho. Kde sčítáme hodnoty zarovnání a penalizujeme mezery. V posledním kroku jsou dobře hodnocené sekvence procházeny, aby se získalo konečné zarovnání. Využijeme Smith-Waterman algoritmus, který bude hodnotit pouze část původní matice. Bude se jednat o pásmo o velikosti 16 se středem v původním zarovnání. Tyto kroky jsou ilustrovány na obrázku 4.15. Takto je FASTA schopna urychlit databázové vyhledávání oproti klasickému Smith-Waterman algoritmu. Cenou je pouze malá ztráta citlivosti. [16]



Obrázek 4.15: Jednotlivé kroky metody FASTA. [9]

4.2.8 BLAST

BLAST je jedna z nejčastěji používaných heuristik pro prohledávání databází. Základní princip je stejný jako u metody FASTA. Také pracuje se slovy, ale místo hešování využívá sufixové stromy. Opět je forma dynamického programování využita až v posledním kroku.

V prvním kroku je hledaná sekvence rozdělena na slova. Z těchto slov je vytvořen strom sufixů. Strom je vytvářen přímočaře. Všechny pozice v sekvenci jsou shromážděny podle typu počáteční báze. Tyto skupiny jsou vyvedeny z kořene stromu. Dále se každá skupina rozdělí podle následujícího typu báze. Pro sekvenci o délce L tato metoda vykoná $L \log L$ kroků. Při vyhledání slov v sekvenci jsou povoleny pouze přesné shody v případě nukleotidů a samotný princip je podobný jako v metodě FASTA.

V následujícím kroku jsou tato slova rozšiřována na obě strany bez mezer. Tyto zarovnání jsou označována jako HSPs. Rozšiřování přestane, když se ohodnocení zmenší o x_u z nejlepšího nalezeného ohodnocení tohoto zarovnání.

V posledním kroku je aplikováno dynamické programování. Začátkem je střed nejlépe ohodnoceného HSP. Matice je vyplňována oběma směry. Dokud se ohodnocení nesníží o x_g. [16]

4.2.9 X-drop

Zarovnání s vysokým ohodnocením většinou nemívají moc vkládání nebo mazání. Pro matici to znamená, že zarovnání většinou jsou blízko diagonály. To způsobilo, že se lidé pokoušeli o algoritmus, který by šetřil čas počítáním pouze elementů okolo diagonály. Takovéto algoritmy však nemusí dojít ke správnému zarovnání. [16]

Jeden z těchto algoritmů je právě X-drop a využívá ho program prohledávání databází BLAST. Zmenšuje počet prvků matice podle toho jak moc je ohodnocení menší, než současné nejlepší ohodnocení. Čím větší hranici zvolíme, tím větší počet elementů bude prohledáván a tím je větší šance nalezení optimálního zarovnání. [16]

Algoritmus X-drop prochází matici po anti-diagonálách. Anti diagonála d je definována jako $d = i + j$, tedy pro každou hodnotu i značící pozici v první sekvenci, hodnota j, která značí pozici v druhé sekvenci, může být získána $j = d - i$. Prvky, které budeme na diagonále hodnotit jsou ohraničené pomocí dvou proměnných i_L a i_U, kde i_L je spodní hranice a i_U je horní hranice. Hodnotu nejlepšího skóre uložíme jako S_{best}. Algoritmus bude začínat elementem $S_{i_0 j_0}$. Nastaví se tedy $S_{best} = S_{i_0 j_0}$ a další proměnné $d = i_0 + j_0$ a $i_l = i_u = i_0$. Postupně hodnotíme každou diagonálu. Po vyhodnocení elementů diagonály se prvně zkontroluje, zda je hodnota některého z elementů větší, než hodnota v S_{best}, pokud ano tak je jeho hodnota uložena do S_{best}. Poté se projdou elementy na anti diagonále, a pokud je hodnota některého o X menší, než S_{best}, pak se jeho hodnota nastaví na $-\infty$, takže se s ním nepočítá při dalších výpočtech. Hodnoty i_L a i_U jsou přehodnoceny podle pravidla $S_{i,j} \neq -\infty$, tak že i_L se nastaví na nejmenší prvek, kde toto platí, zatímco i_U na největší. Díky tomu je možné snížit počet elementů, které budeme hodnotit v další diagonále. Pokud elementy na okraji diagonály nesplňují $S_{i,j} \neq -\infty$, potom je nutné diagonálu rozšířit. Pokud $i_U + 1 \leq i_L$ pak tu nejsou žádné další elementy v regionu a výpočet je ukončen, jinak se posuneme na další diagonálu. [16]

Implementovaný algoritmus na obrázku 4.16: Na vstupu jsou sekvence M a N o délce m a n. Dále máme pole S. Penalizace mezery se značí g, $s(M_i, N_j)$ představuje ohodnocení zarovnání znaků na pozicích i a j. X představuje hranici snížení.

$T \leftarrow T' \leftarrow S_{0,0} \leftarrow 0$
$k \leftarrow L \leftarrow U \leftarrow 0$
while $L \leq U + 1$ **do**
  $k \leftarrow k + 1$
  for j v rozsahu L .. $U + 1$ **do**
    $i \leftarrow k - j$

$$S_{i,j} \leftarrow Max \begin{cases} S_{i-1,j} + g \text{ pro } i \geq 0 \\ S_{i,j-1} + g \text{ pro } j \geq 0 \\ S_{i-1,j-1} + s(M_i, N_j) \text{ pro } i \geq 0, j \geq 0 \end{cases}$$

    $T' \leftarrow \max\{T', S_{i,j}\}$
    if $S_{i,j} < T - X$ **then**
      $S_{i,j} \leftarrow -\infty$
    end if
  end for
  $L \leftarrow$ nejmenší i z hodnot $S(i, k - i) > -\infty$ získaných v cyklu
  $U \leftarrow$ největší i z hodnot $S(i, k - i) > -\infty$ získaných v cyklu
  $L \leftarrow \max\{L, k + 1 - n\}$
  $U \leftarrow \min\{U, m - 1\}$
  $T \leftarrow T'$
end while
výsledné skóre je T

Obrázek 4.16: X-drop algoritmus

4.2.10 Greedy x-drop

Tento algoritmu ohodnocuje jak moc je zarovnání sekvencí odlišné, odlišnosti tedy zvyšují skóre. Ohodnocení odlišnosti nacházející se na pozicích i a j budeme značit $D(i, j)$. Čím menší hodnota, tím jsou sekvence podobnější. Počítáme tedy neshody a mezery. Pro správnou funkci se předpokládá, že $ind = mis - mat/2$. Kde ind značí penalizaci za mezeru, mis značí neshodu a mat značí shodu. Protože pro správnou funkci algoritmu potřebujeme taky klasické ohodnocení shody zarovnání, musíme ho být schopni získat z rozdílu. To získáme podle vzorce $S(i, j) = S'(i+j, D(i, j))$. Tedy $S'(i+j, d) = (i+j) * mat/2 - d * (mat - mis)$. Pokud tedy známe $D(i, j)$, pak známe také $S(i, j)$. Pozice, na kterých je $D = 0$, se nachází na hlavní diagonále a všechny předchozí pozice byly shody. Algoritmus prochází matici po diagonálách, je tedy vhodný pro podobné sekvence, u kterých se optimální zarovnání bude nacházet blízko prostřední diagonály. [15]

Pokud bychom algoritmus vysvětlovali podrobněji, potřebujeme funkci $R(d-1, k)$, vracející x-pozici poslední hodnoty $d-1$ na diagonále k pro $-d < k < d$. Budeme předpokládat, že jsme získali všechny pozice (i, j), kde $D(i, j) = d-1$, přesněji všechny hodnoty $R(d-1, k)$. Diagonála k se skládá z bodů (i, j), kde $k = i - j$. Nakonec potřebujeme způsob jak zjistit zda $S(i, j) < T - X$. Kde X představuje hranici snížení a T je maximum $S(p, q)$ přes všechna (p, q), kde $p + q < i + j$. Problém je, že oproti klasickému X-drop algoritmu už neprocházíme matici po anti diagonálách. [15]

Předpokládejme, že fáze x představuje fázi algoritmu, kde $D(i, j) = x$ a v $T[x]$ bude uložena největší nalezená hodnota od 0 do x. Porovnáním $S(i, j)$ s $T[d'] - X$ zjistíme, zda má $S(i, j)$ být ořezané. Tento test je potřeba provést pouze pokud poprvé najdeme bod

(i, j) na diagonále k, kde $D(i, j) = d$. Pokud tento bod projde, poté projdou i všechny následující body na diagonále. d' $= d - [(x + mat/2)/(mat - mis)] - 1$

Zpětný průchod nezískává klasické zarovnání, ale minimální počet operací potřebných pro změnu první sekvence na druhou. Přístup pro získání změny je následující. Nejdříve potřebujeme získat $d_{best}ak_{best}$. Protože jsme pro každé $d \geq 0$, nechali U_d a L_d označit horní a dolní hranice $k : R(d, k) > -\infty$ a uložili si všechny $R(d, k)$kde$L_d - 2 \leq k \geq U_d + 2$, můžeme provést zpětný průchod. Ten je proveden rekurzivně a budeme ho značit script(d, k). Získáme maximální hodnotu z $R(d - 1, k - 1), R(d - 1, k), R(d - 1, k + 1)$, získáme zarovnání bází a podle toho, které R bylo největší, poté zavoláme script(d, k). [15]

Implementovaný algoritmus na obrázku 4.17: Na vstupu jsou sekvence M a N o délce m a n. Dále máme pole S. X představuje hranici snížení.

$i \leftarrow 0$
while $i < min\{m, n\}$ a $a_{i+1} = b_{i+1}$ **do**
  $i \leftarrow i + 1$
end while
$R_{0,0} \leftarrow i$
$T' \leftarrow T[0] \leftarrow S'(i + i, 0)$
$d \leftarrow L \leftarrow U \leftarrow 0$
while $L \leq U + 2$ **do**
  $d \leftarrow d + 1$
  $d' \leftarrow d - [\frac{X+mat/2}{mat-mis}] - 1$
  for k v rozsahu $L - 1$ .. $U + 1$ **do**
    $i \leftarrow k - j$
    $i \leftarrow Max \begin{cases} R(d-1, k-1) + 1 \text{ pro } L < k \\ R(d-1, k) + 1 \text{ pro } L \leq k \leq U \\ R(d-1, k+1) \text{ pro } k < U \end{cases}$
    $j \leftarrow i - k$
    if $i > -\infty$ a $S'(i + j, d) \geq T[d'] - X$ **then**
      while $i < m$ a $j < n$ a $a_{i+1} = b_{i+1}$ **do**
        $i \leftarrow i + 1$
        $j \leftarrow j + 1$
      end while
      $R(d, k) \leftarrow i$
      $T' \leftarrow \max\{T', S(i + j, d)\}$
    else
      $R(d, k) \leftarrow -\infty$
    end if
  end for
  $L \leftarrow$ nejmenší k z hodnot $> -\infty$ získaných v cyklu
  $U \leftarrow$ největší k z hodnot $> -\infty$ získaných v cyklu
  $L \leftarrow \max\{L, (\text{nejmenší } k \text{ z hodnot } R(d, k) = n + k \text{ získaných v cyklu}) + 2\}$
  $U \leftarrow \min\{U, (\text{nejmenší } k \text{ z hodnot } R(d, k) = m \text{ získaných v cyklu}) - 2\}$
  $T[d] \leftarrow T'$
end while
výsledné skóre je T'

Obrázek 4.17: Greedy x-drop algoritmus

4.2.11 FOGSAA

V základu se jedná o algoritmus, který od kořene rozšiřuje větev na základě chamtivého výběru. Sestavujeme strom, kde každá cesta od kořene k listu představuje možné zarovnání páru sekvencí. Pokud je při průchodu větví jiná větev shledána více slibnou z hlediska ohodnocení, potom bude rozšířena. Tato procedura se opakuje, dokud není nalezena žádná větev s lepšími předpoklady. FOGSAA vypočítá optimální zarovnání nalezením optimální větve ve stromě. Nakonec vrátí optimální skóre. Výsledným zarovnáním je cesta od kořene k listu. [3]
Každý uzel stromu se skládá z prvků:

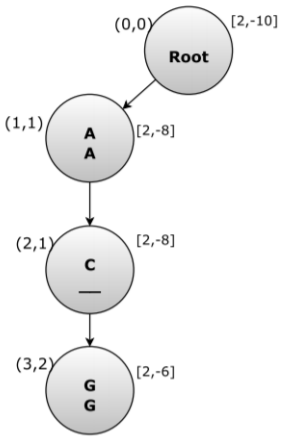
- (P1,P2) dvojice zarovnaných prvků.
- Typ zarovnání.
- Současné skóre větve.
- Předpokládané maximální a minimální skóre celé větve.

Algoritmus začíná v kořenu stromu a pokračuje podle jedné ze tří možností, tedy zarovnání dvou prvků, vložení mezery do první sekvence, vložení mezery do druhé sekvence. Příklad tohoto stromu je na obrázku 4.18. Který uzel bude expandován, se rozhodne podle hodnotící funkce. Zbytek bude vložen do prioritní fronty. Vybereme nejlépe tedy ohodnoceného potomka a vyvedeme z něj další větev. Ohodnocení je složené ze současného skóre a předpokládaného maximálního skóre. [3] Předpokládané budoucí skóre se získá:

$$F_{\min} = \begin{cases} x_2 * Ms + G * (x_1 - x_2), & x_2 < x_1 \\ x_1 * Ms + G * (x_2 - x_1), & \text{otherwise} \end{cases} \tag{4.12}$$

$$F_{\max} = \begin{cases} x_2 * M + G * (x_1 - x_2), & x_2 < x_1 \\ x_1 * M + G * (x_2 - x_1), & \text{otherwise} \end{cases} \tag{4.13}$$

Kde M je ohodnocení shody, Ms neshody a G je ohodnocení mezery. Dále $x_1 = (n - P1)$ a $x_2 = (m - P2)$. Porovnávané sekvence mají velikost n a m. [3]



Obrázek 4.18: Příklad stromu, uzlů a jejich prvků. [3]

Implementovaný algoritmus na obrázku 4.19: Na vstupu jsou sekvence M a N o délce m a n. $C[i][j]$ představuje prvek stromu, kde $P1 = i, P2 = j$.

$P1 \leftarrow P2 \leftarrow PrS \leftarrow 0$
$C[0][0]$
$optimal \leftarrow C[0][0]$
repeat
  while $P1 \leq (m-1)$ nebo $P2 \leq (n-1)$ **do**
    Vyber nejlepšího zbývajícího potomka podle T_{max}
    if Některý z potomků stále nebyl vybrán **then**
      Ulož současný prvek stromu do prioritní fronty podle T_{max} nejlepšího ze zbývajících potomků
    end if
    if $potomek.PrS \leq C[potomek.P1][potomek.P2].PrS$ **then**
      Utni současnou větev
    else
      $C[potomek.P1][potomek.P2] \leftarrow$ nové ohodnocení
      $P1 \leftarrow potomek.P1$
      $P2 \leftarrow potomek.P2$
      if $potomek.T_{max} \leq optimal$ **then**
        Utni současnou větev
      end if
    end if
  end while
  if $C[P1][P2].T_{max} \geq optimal$ **then**
    $optimal \leftarrow C[P1][P2].T_{max}$ a nastav větev jako otimální
  end if
  Vyber vrchní prvek z fronty u aktualizuj T_{max}
  if Pokud T_{max} vrchního prvku nedosahuje 30% shody **then**
    ukonči a navrať skóre
  end if
until $optimal \geq tMax$

Obrázek 4.19: FOGSAA algoritmus

4.3 Predikce sekundární struktury RNA

RNA je důležitá nejen pro předání genetické informace jako pošta, ale také pro imunitní systém, nebo signal recognition particle (česky: signál rozpoznávající částice) a další... . Pro pochopení jednotlivých mechanismů a funkcí, musíme prvně porozumět struktuře. Samotnou strukturu je možné rozdělit do tří úrovní. Primární strukturou rozumíme lineární. Sekundární strukturou rozumíme kolekci bázových párů. Terciální strukturu představuje trojrozměrné uskupení atomů v RNA. Zde se budeme zabývat sekundární strukturou.

Palindromy v sekvencích RNA

Palindrom je řetězec, který se čte stejně, nezávisle na směru. Existují tedy dva druhy palindromů, sudé a liché. V obou případech se jedná o dva podřetězce, které jsou vzájemně

reverzní a komplementární. Sudý je rozdělen pomyslně mezi znaky. Lichý je rozdělen středovým znakem. [2]

V případě RNA jsou palindromy také označovány jako vlásenky. Mohou být takové, jak již bylo definováno, ale mohou být také komplementární, zde je druhý podřetězec nejen reverzní, ale i komplementární k prvnímu řetězci. Komplementární palindromy jsou pro sekundární strukturu RNA velmi důležité, protože umožňují interakci vzdálených nukleotidů, která stabilizuje celkovou strukturu proteinu. Informace o tom, které nukleotidy spolu vzájemně interagují, představuje popis sekundární struktury RNA. Mohou obsahovat smyčky a chyby. Chyby se dělí na dva typy, kde první je porušení komplementarity v některé z párů bází a vzniká vnitřní smyčka, druhá vznikne přidáním báze do jednoho z řetězců, tím je vytvořena vyboulená smyčka. Bývají zakončeny smyčkou obsahující nespárované nukleotidy.

Termodynamická struktura RNA

Existují metody pro predikci sekundární struktury, které počítají s volnou energií (Gibbsova volná energie), čím nižší je, tím je větší stabilita struktury. Pro výpočet se pracuje s parametrem nejbližšího souseda, protože stabilita každého páru závisí pouze na sousedních párech. Výsledná volná energie je součtem všech volných energií jednotlivých párů. Volná energie se mění s teplotou, zde budeme předpokládat hodnoty volné energie pro pevně danou teplotu. Tato metoda je například využívána v Zukerově algoritmu. [1]

	A-U	C-G	G-C	U-A	G-U	U-G
A-U	-0.9	-1.8	-2.3	-1.1	-1.1	-0.8
C-G	-1.7	-2.9	-3.4	-2.3	-2.1	-1.4
G-C	-2.1	-2.0	-2.9	-1.8	-1.9	-1.2
U-A	-0.9	-1.7	-2.1	-0.9	-1.0	-0.5
G-U	-0.5	-1.2	-1.4	-0.8	-0.4	-0.2
U-G	-1.0	-1.9	-2.1	-1.1	-1.5	-0.4

Počet bází	1	5	10	20	30
Vnitřní	--	+5.3	+6.6	+7.0	+7.4
Vyboulené	+3.3	+4.8	+5.5	+6.3	+6.7
Vlásenková	--	+5.9	+5.3	+6.1	+6.5

Obrázek 4.20: Tabulka hodnot volné energie párů nahoře a volné energie smyček dole [2]

Pravidla pro tvorbu sekundární struktury

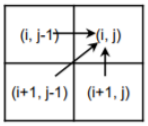
- Spárované báze stabilizují výslednou strukturu (snižují volnou energii).
- Nespárované části ji destabilizují (zvyšují volnou energii).
- Pokud se změní jedna báze z páru, obvykle se změní i druhá tak, aby byl pár zachován.
- Reálné molekuly RNA mohou mít více stabilních struktur, mezi kterými přechází.

Sekundární struktura RNA je složena z vlásenkových struktur pospojovaných k sobě, proto pro predikci budeme hledat vlásenkové struktury a následně jejich spojení. [2]

Detekce vlásenkových struktur pomocí dynamického programování

Protože RNA palindromy mohou obsahovat i chyby v podobě záměny, vložení nebo odstranění znaku, tak je výhodné použít dynamické programování. Postup: [2]

- Sestrojení 2D matice, kde sloupce i řádky reprezentují analyzovaný vstupní řetězec.
- Dvojici hlavních diagonál inicializujeme na nuly. Tedy hlavní diagonálu a diagonálu pod ní.
- Vypočteme trojúhelníkovou matici ve směru od hlavní diagonály směrem vpravo nahoru podle obrázku 4.21, kde využijeme vzorce 4.14.



Obrázek 4.21: Možnosti výpočtu dané buňky [2]

Vzorec pro výpočet:

$$M(i,j) = Max \begin{cases} M(i+1,j) - g \\ M(i,j-1) - g \\ M(i+1,j-1) + d(S[i], S[j]) \end{cases} \tag{4.14}$$

Kde i a j jsou souřadnicové osy matice.
Příklad skórovací funkce textu:

- $d(S[i], S[j]) = +1$ pro $S[i] = S[j]$
- $d(S[i], S[j]) = -1$ pro $S[i] \neq S[j]$

Příklad skórovací funkce RNA:

- $d(S[i], S[j]) = +1$ pro $S[i] =$ comp($S[j]$)
- $d(S[i], S[j]) = -1$ pro $S[i] \neq$ comp($S[j]$)
- Penalizace za mezeru je $g = -2$

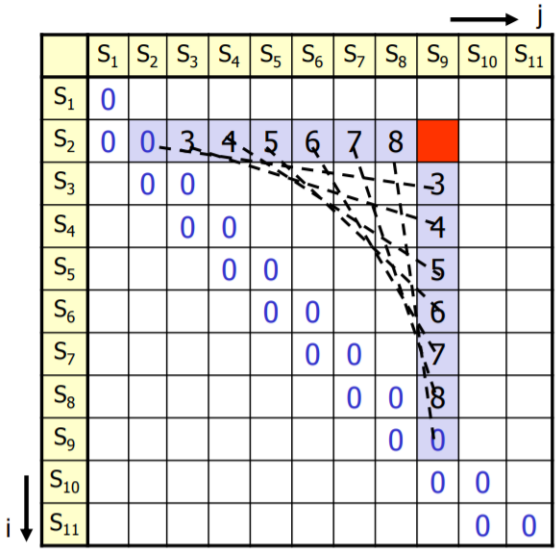
Spojování vlásenek

Pravidla pro spojení:

- Palindromy se nesmí překrývat $j_1 < i_2$
- Sečtením hodnot skóre palindromů získáme celkové skóre spojené oblasti.

- Ze všech přípustných kombinací pro danou oblast ohraničenou indexy (i, j), vybereme tu, která dosahuje nejvyššího skóre.
- Vede na příliš mnoho kombinací a tedy i výpočtů.

Vede to na veliký počet kombinací proto, že uvažujeme libovolnou mezeru mezi pozicemi j_1 a i_2. Pokud odstraníme penalizaci za mezeru, potom spojujeme pouze sousední palindromy.



Obrázek 4.22: Příklad spojování pouze sousedních dvou palindromů [2]

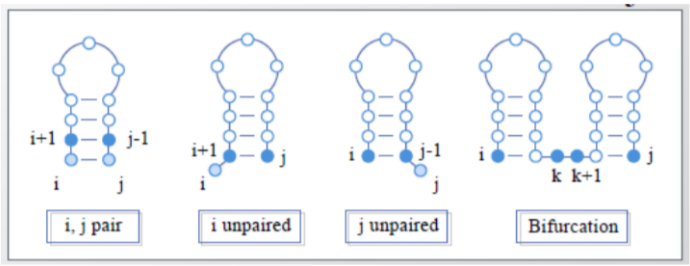
4.3.1 Nussinov algoritmus

Algoritmus řeší problém predikce sekundární struktury maximalizováním párů. Toho je dosaženo přiřazením ohodnocení vstupu v matici. Pro každý pár se ohodnocení zvýší, v ostatních případech ohodnocení zvyšovat nebudem. Nakonec provedeme zpětný průchod pro největší ohodnocení. Hlavní část Nussinov algoritmu je rekurzivní a počíta sekundární strukturu pro malé podsekvence, než dosáhne větších.[6]
Postup:

- Inicializujeme matici, kde záhlaví sloupců a řádku obsahuje analyzovanou sekvenci.
- Dvojici hlavních diagonál inicializujeme na nuly. Tedy hlavní diagonálu a diagonálu pod ní.
- Pro každou buňku vypočteme hodnotu jako maximum ze čtyř možností uvedených ve vzorci 4.15.
- Následně si musíme pamatovat, jakou z těchto možností jsme si vybrali.
- Zpětnou rekonstrukcí z pravého horního rohu získáme predikovanou strukturu.

Vzorec pro výpočet:

$$M(i,j) = Max \begin{cases} M(i+1,j) \\ M(i,j-1) \\ M(i+1,j-1) + d(S[i],S[j]) \\ Max[M(i,k) + M(k+1,j)] \quad \forall k : i \leq k < j \end{cases} \tag{4.15}$$



Obrázek 4.23: Čtyři možnosti přidání bází [2]

Zde se nepenalizují mezery. Časová složitost algoritmu je $O(n^3)$. Výsledky však nejsou dostatečně kvalitní. Algoritmus dává stejnou váhu bázovým párům, i když různé bázové páry přispívají ke stabilitě různou vahou. Neuvažuje vliv sousedních bázových párů. Smyčky, neshody a další defekty nejsou penalizovány. Akceptuje smyčky s nulovým počtem resudií, což je fyzikálně nemožné. [2]
Implementovaný algoritmus na obrázku 4.24: Na vstupu je sekvence N o délce n. Dále máme pole M. $M(N_i, N_j)$ představuje ohodnocení zarovnání znaků na pozicích i a j.

inicializace diagonál
for k v rozsahu 1 .. $n-1$ **do**
  for i v rozsahu k .. $n-1$ **do**
    $j \leftarrow i - k$

$$M(i,j) = Max \begin{cases} M(i-1,j) \\ M(i,j+1) \\ M(i-1,j+1) + d(S[i],S[j]) \\ Max[M(i,k) + M(k-1,j)] \quad \forall k : j < k \leq i \end{cases}$$

  end for
end for
výsledné skóre je $M_{n-1,0}$

Obrázek 4.24: Nussinov algoritmus

4.3.2 Zukerův algoritmus

Opět založen na principu dynamického programování. Odstraňuje některé nedostatky Nussinova algoritmu. Pracuje s volnou energií tak, že započítává její vliv na sousední páry. Zukerův algoritmus tedy pracuje se sekundární strukturou jako s grafem. Cílem je nalézt uspořádání s minimální celkovou volnou energií. [2] Algoritmus pracuje se dvěma maticemi.

Kde obě matice slouží pro získání celkové volné energie. První matice přiřazuje hodnotu volné energie do pole, pokud se báze párují, jinak jí nastaví na $-\infty$. [13]

Zde bude $eh(i,j)$ představovat ukončovací smyčku (hairpin loop) uzavřenou (i, j). Dále $es(i,j)$ bude představovat energii párů (i, j) a (i + 1, j - 1). $Ebi(i,j,i',j')$ představuje energii vnitřní smyčky uzavřené páry (i, j) a (i', j'). Nakonec a představuje energii multismyčky.[5] Vzorec pro výpočet první matice:

$$W(i,j) = \min \begin{cases} W(i+1,j) \\ W(i,j-1) \\ V(i,j) \\ \min[W(i,n) + W(n+1,j)] \text{ kde } i < n < j \end{cases} \tag{4.16}$$

Vzorec pro výpočet druhé matice:

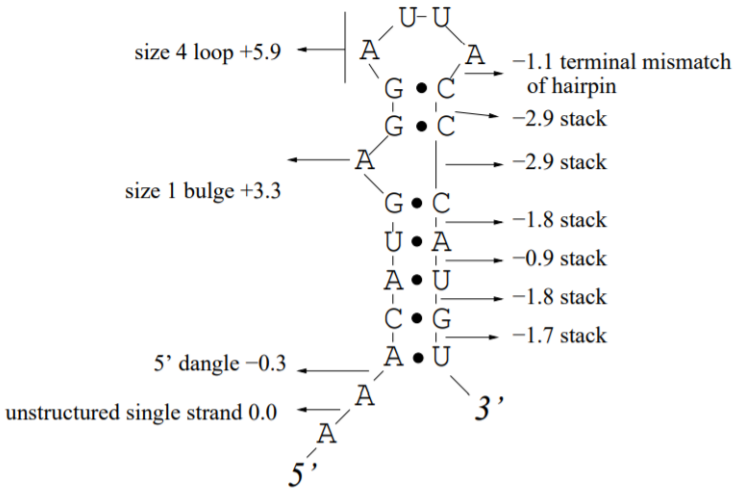
$$V(i,j) = \min \begin{cases} eh(i,j) \\ es(i,j) + V(i+1,j-1) \\ VBI(i,j) \\ VM(i,j) \end{cases} \tag{4.17}$$

kde

$$VBI(i,j) = \min_{\substack{i<i'<j<j' \\ i'-i+j-j'>2}} \{Ebi(i,j,i',j') + V(i',j')\} \tag{4.18}$$

$$VM(i,j) = \min_{i<k<j-1} \{W(i+1,k) + W(k+1,j-1)\} + a$$

Časová složitost výpočtů se liší pro fukci. Pro W je $O(L^3)$. Pro V je $O(L^2)$. Pro VBI je $O(L^4)$. Pro VM je $O(L^3)$. Celková časová složitost algoritmu je $O(L^4)$.[5]



Obrázek 4.25: Příklad predikce využitím volné energie [5]

Implementovaný algoritmus na obrázku 4.26: Na vstupu je sekvence N o délce n. Dále máme pole W a V. $W(N_i, N_j)$ a $V(N_i, N_j)$ představuje ohodnocení zarovnání znaků na pozicích i a j.

inicializace diagonál
for k v rozsahu 1 .. $n-1$ **do**
  for i v rozsahu k .. $n-1$ **do**
    $j \leftarrow i - k$
    $min \leftarrow 0$
    if $W(i-1, j) < min$ **then**
      $min \leftarrow W(i-1, j)$
    end if
    if $W(i, j+1) < min$ **then**
      $min \leftarrow W(i, j+1)$
    end if
    if $V(i, j) < min$ **then**
      $min \leftarrow V(i, j)$
    end if
    for k v rozsahu i .. $j-1$ **do**
      if $W(i, k) + W(k-1, j) < min$ **then**
        $min \leftarrow W(i, k) + W(k-1, j)$
      end if
    end for
    $W(i, j) \leftarrow min$
  end for
end for
výsledné skóre je $M_{n-1,0}$

Obrázek 4.26: Zuker algoritmus

Funkce pro výpočet energií smyček a párů $eh(i, j)$, $es(i, j)$ a $Ebi(i, j, i', j')$, se kterými pracuje matice V, byli převzaty z projektu RNA-Google: Implementing a prototype for fast search in large RNA repositories.[14]

Kapitola 5

Návrh a Implementace

V této kapitole je popis aplikace a princip implementace jednotlivých implementovaných algoritmů. První část popisuje návrh aplikace a jednotlivé požadavky na její funkčnost. V druhé kapitole je demonstrována implementace aplikace a rozhraní. Dále je zde demonstrováno vykreslení průběhu algoritmů a výstup.

5.1 Návrh

Cílem této práce bylo vytvořit aplikaci, která by měřila rychlost zvolených algoritmů. Tato aplikace také musí být schopná tyto algoritmy krokovat. Dále by měla být schopna ukládat jednotlivé statistiky a umožňovat vyhledávání mezi jednotlivými statistikami. Nakonec musím samotná aplikace být jednoduše dostupná.

Hlavním vstupem pro vykonání algoritmu je způsob, jakým bude probíhat, tedy jestli jen budeme měřit, nebo i krokovat postup. Dalším podstatným vstupem budou sekvence, jejich počet se samozřejmě liší podle účelu algoritmu, tedy jestli slouží pro zarovnání sekvencí nebo predikci sekundární struktury RNA. Dále jsou podstatné parametry algoritmu, tedy hodnotící matice, pro zarovnání sekvencí penalizace mezer a hranice snížení, pokud s ní algoritmus pracuje.

Výstupem musí být statistika obsahující parametry vstupu, která musí také hlavně obsahovat zarovnané sekvence nebo sekundární strukturu. V případě měření algoritmů musí být také přítomna doba průběhu, která se dělí na ohodnocování a zpětný průchod. Statistiky by měli následně být ukládány a mělo by být možné je mezi sebou porovnat.

5.2 Implementace

Jedná se o webovou aplikaci. Velikou výhodou je tedy její dostupnost. Programovacím jazykem použitým pro implementaci aplikace byl Javascript. Jedná se o interpretovaný jazyk, který je dynamicky typovaný. Využívá funkcionální paradigma, to umožňuje do běžné proměnné uložit funkci. Při implementaci bylo využito objektově orientovaného programování pomocí syntaxe představené v rozšíření ES6. Pro tvorbu uživatelského rozhraní byly použity jazyky HTML a CSS s pomocí frameworku Bootstrap. Bootstrap obsahuje zejména šablony pro HTML a CSS, zjednodušující implementaci rozhraní.

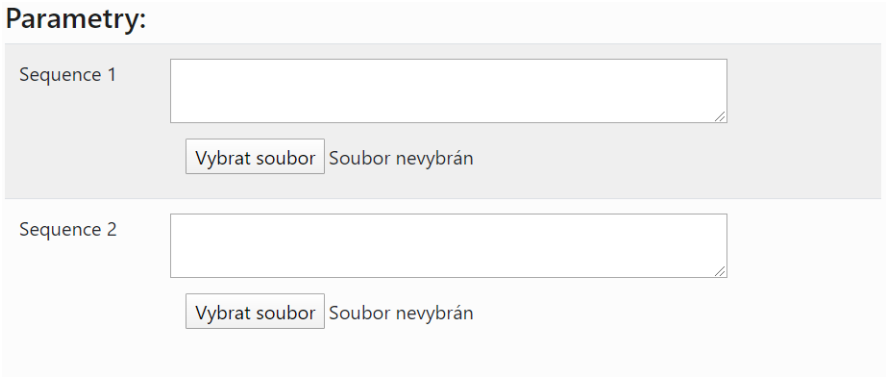
Realizace

Jednotlivé stránky a jejich obsah je generován pomocí Javasciptu. Každá stránka má vlastní třídu, která dědí od třídy Page. Při spuštění algoritmu třída AlgorithmStarter vytvoří třídu daného algoritmu a předá mu potřebné parametry. Podle parametrů je proveden algoritmus a výsledky, které si ukládal, předá třídě, která udržuje statistiky. V případě, že je během průběhu algoritmu nutné něco vykreslit, potom to zajišťuje třída AlgorithmDrawer.

Důležité bylo oddělit vykreslování a provádění algoritmu, aby nebylo měření ovlivňováno a tedy zpomalováno vykreslením matice nebo statistik. To bylo bez větších problémů vyřešeno pomocí návrhu. Problém byl ale u algoritmu FOGSAA, u kterého bylo nutné udělat menší ústupy, aby došlo k přehlednému vykreslování algoritmu. Vykreslování je sice odděleno a nedochází k němu při průběhu algoritmu, nicméně některé konstrukce, se kterými vykreslování pracovalo, byly ponechány. Důvodem byl nedostatek času. Dále byl menší problém u Zuker algoritmu, jehož princip nebylo obtížné naprogramovat. Funkce hodnotící jednotlivé smyčky ale nebyly uvedené ve zdrojích, ze kterých bylo čerpáno. Nakonec byly tyto funkce převzaty. U Greedy x-dropu bylo nutné rozhodnou, jak vykreslit průběh algoritmu. Algoritmus nepracuje s klasickou hodnotící maticí. Nakonec je vykreslováno ohodnocení zarovnání, které algoritmus vypočítá během svého průběhu. K tomu ale nedochází během každého kroku, proto během některých kroků není aktualizována matice. Jinak nedocházelo k větším problémům. Věci jako přidání možnosti opakovaného měření, nebo průměrná doba průběhu algoritmu nebyli součástí návrhu, ale byly přidány po otestování.

Rozhraní

Rozhraní bylo navrhnuté s důrazem na rozšiřitelnost a přehlednost. Po spuštění se otevře hlavní stránka, kde jsou základní informace o algoritmech a stručný návod k použití. Menu přesměrovává jak na jednotlivé algoritmy, tak na samotné statistiky. Každá stránka jednotlivého algoritmu obsahuje textovou oblast pro vložení sekvencí. Sekvence obsahující DNA znaky se vkládají pro algoritmy pro zarovnání sekvencí, RNA znaky pro predikci sekundární struktury. Sekvence je taky možné nahrát ze souboru. Pokud bude vložen chybný znak, algoritmus nebude spuštěn. Na obrázku 5.1 jsou textové oblasti, do kterých se ze souboru sekvence nahrají. Je možné tedy tyto sekvence ještě po nahrání a mezi porovnáváním upravovat.



Obrázek 5.1: Vložení sekvencí

Dále se zvolí způsob provedení, tedy jestli chceme průběh algoritmu krok po kroku vykreslit, nebo měřit rychlost. Jednotlivé možnosti jsou:

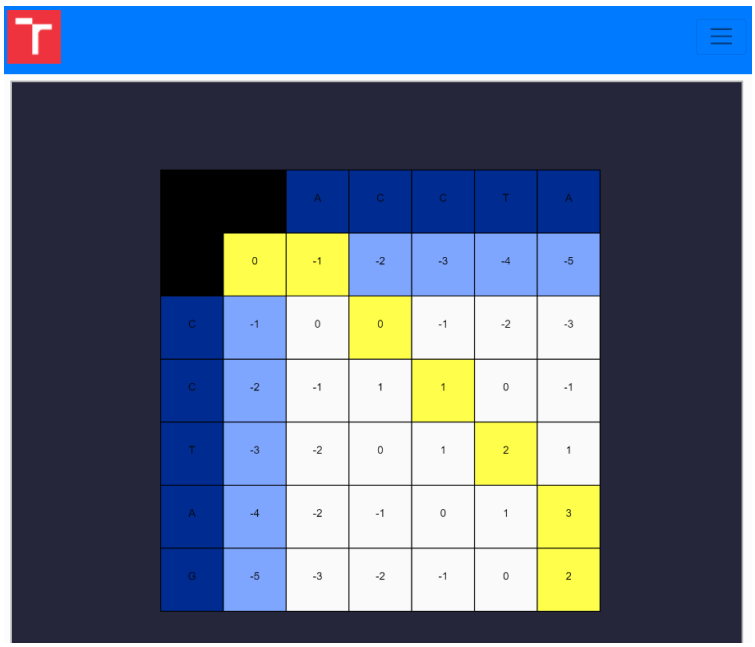
- Provedení kroku po stisknutí klávesy.
- Provedení kroku po uplynutí 2 sekund.
- Okamžité provedení.
- Měření rychlosti algoritmu.

Nakonec je zde možné zadat parametry algoritmu, mezi kterými je třeba hodnotící matice, popřípadě parametry jako penalizace mezery, pokud s ní algoritmus pracuje. V případě hodnotících matic je možný výběr z několika přednastavených matic, pokud ale budou pro uživatele nedostatečné, potom je možné nahrát vlastní. S hodnotícími maticemi nepracují Greedy x-drop a Zuker algoritmus. Další potřebné parametry jsou nastavovány v číselném vstupu.

5.3 Vykreslení průběhu algoritmů

Jednotlivé algoritmy jsou vykreslovány různými způsoby, povětšinou se jedná o matici, kde je v každém kroku aktualizována hodnota v matici. Hodnoty v této matici představují dosažené skóre zarovnání. Výjimkou je rekurze, kde se vykreslí rekurzivní strom. V případě Zukerova algoritmu se vykreslují obě matice, se kterými algoritmus pracuje, a v jednom kroku jsou aktualizovány hodnoty v obou maticích. U Greedy x-dropu se vykreslují hodnoty dosaženého skóre, pouze pokud jsou v kroku vypočítány.

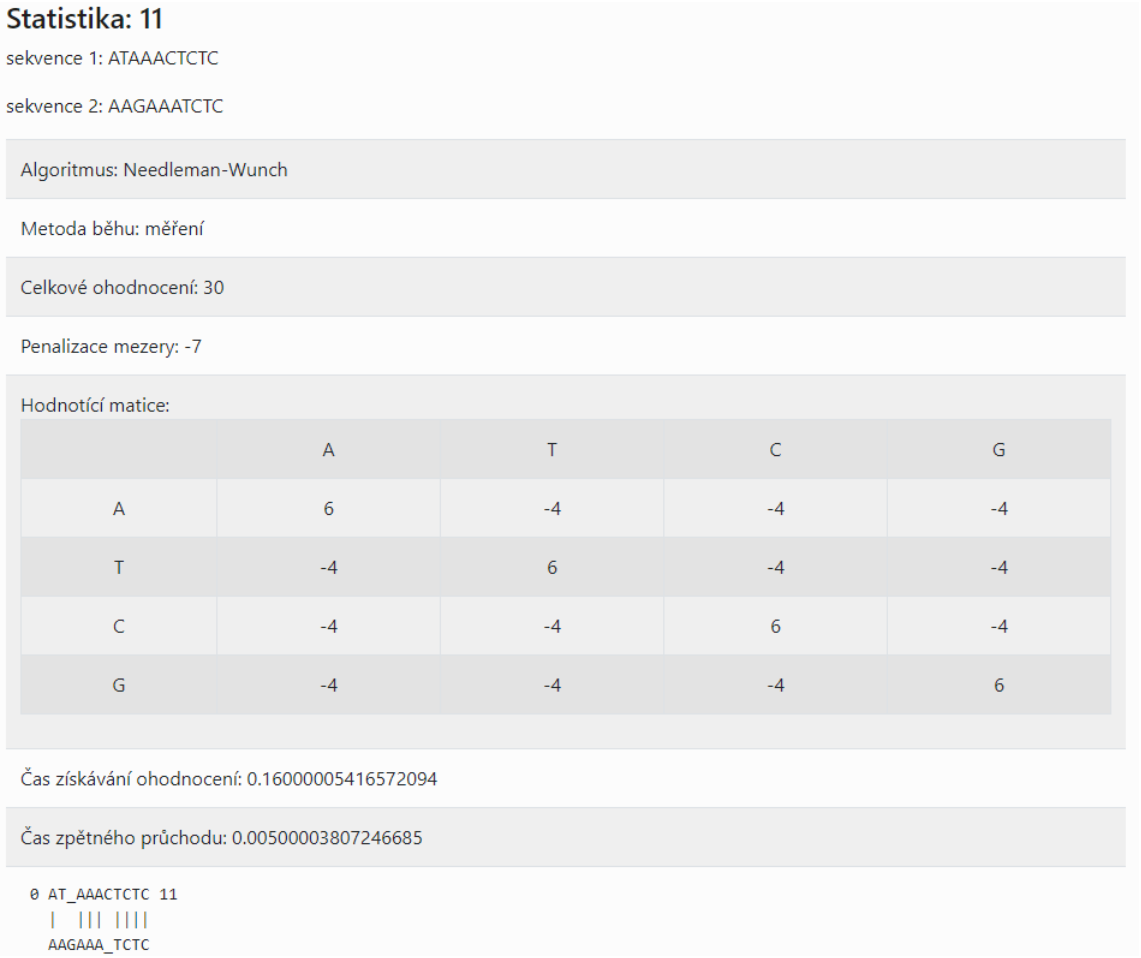
Na obrázku 5.2 je představeno vykreslení zpětného průchodu maticí pro Needleman-Wunch algoritmus při porovnávání dvou krátkých sekvencí.



Obrázek 5.2: Vykreslení matice pro Needleman-Wunch algoritmus

5.4 Výstupy a statistiky

Po provedení algoritmu jsou vypsány parametry, se kterými byl spuštěn, mezi nimi je například ohodnocení nebo zvolený způsob průběhu. Pokud bylo jako způsob průběhu zvoleno měření, potom je také vypsána doba průběhu algoritmu. Vždy je vypsáno výsledné zarovnání nebo výsledná sekundární struktura. Po provedení algoritmu je jeho statistika uložena do karty se statistikami. Zde jsou uložené všechny provedené statistiky. Každá statistika má tlačítko, které zobrazí její JSON pro zkopírování. Je také možné nahrát statistiky ve formátu JSON. Ve statistikách je možné vyhledávat podle algoritmu, velikosti sekvencí, ohodnocení nebo průběhu algoritmu. Ve výsledku je vypsána průměrná doba průběhu ze zobrazených statistik algoritmů. Na obrázku 5.3 je příklad výpisu statistiky běhu pro Needleman-Wunch algoritmus.



Obrázek 5.3: Vykreslení statistiky pro Needleman-Wunch algoritmus

Kapitola 6

Testy a jejich výsledky

Tato kapitola popisuje jednotlivé testy provedené s algoritmy pomocí samotné aplikace. Je zde popsáno, se kterými parametry jednotlivé algoritmy pracují a za jakých podmínek jsou efektivnější, než ostatní algoritmy, které aplikace obsahuje. Samotným účelem těchto testů je srovnat implementované algoritmy podle rychlosti, které dosahují a určit za jakých podmínek je dosahují.

6.1 Způsob testování

Testy běžely na zařízení s Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz 2.40 GHz a 8 GB RAM. Testy jsou rozděleny podle délky zarovnávaných sekvencí, která je v rozsahu délky od 10 znaků do 600 znaků, tak že sekvence o 600 znacích byli rozděleny na úseky 10, 60, 120, 250, 350, 600 znaků. V případě zarovnání sekvencí byly testovány i sekvence o 1000 znacích. Zde jsou menší sekvence podsekvencemi větších.

Zarovnání sekvencí

Testovalo se zarovnání různých sekvencí. Testy byly prováděny s nastavenými parametry:

- Délka sekvence.
- Ohodnocení shody [6], neshody [-4] a mezery [-7]. (ty se během testů neměnili)
- Hranice snížení. (pokud s ní algoritmus pracoval)

Všechny testy měli nastavené stejné ohodnocení. Porovnávala se tedy rychlost zarovnání algoritmu podle velikosti sekvence, nastavené hranice snížení, podobnosti zarovnávaných sekvencí. Dále bylo srovnáno dosažené ohodnocení zarovnání. Rychlost provádění byla měřena jak pro získávání skóre, tedy ohodnocení sekvence, tak pro zpětný průchod. Celkový čas představuje dobu získávání skóre sečtenou s dobou zpětného průchodu. Ve statistikách je uvedená rychlost ohodnocení a celková rychlost provádění.

Při testování byly porovnávány dvě sady sekvencí. Sekvence s malou podobností, označíme sada A a s větší podobností označíme sada B. Při porovnávání sekvencí o velikost 1000 znaků, byla podobnost sekvencí podstatně zvýšena, aby bylo možné porovnat závislost růstu doby provádění algoritmů na podobnosti zarovnávaných sekvencí. V případě porovnávání sekvencí s malou podobností je podobnost velice malá, ohodnocení dosahuje i záporných hodnot.

Predikce sekundární struktury

Testy byly prováděny s nastavenými parametry:

- Délka sekvence.

Vzhledem k principu obou algoritmů, byla délka sekvencí jediným společným parametrem. Porovnávala se tedy rychlost zarovnání algoritmu podle velikosti sekvence.

6.2 Naměřené hodnoty

Rekurze

V tabulce 6.1 jsou uvedeny statistiky základních testů provedených na rekurzivním zarovnání. Hodnoty zde jsou převážně pro představení, jak dlouho se může špatně navržený algoritmus provádět. Jen při porovnávání o trochu větších sekvencí bude doba provádění skoro třikrát větší.

Velikost	Rychlost hodnocení	Celková rychlost	Dosažené ohodnocení
10 znaků	716.70 ms	716.70 ms	10
12 znaků	22140.79 ms	22140.79 ms	2

Tabulka 6.1: Testy provedené na rekurzi

Needleman-Wunch

V tabulce 6.2 jsou uvedeny statistiky základních testů provedených na algoritmu Needleman-Wunch. Výsledky tohoto algoritmu budou sloužit jako výchozí hodnoty, se kterými budou porovnávány výsledky následujících testů.

Velikost	Rychlost hodnocení	Celková rychlost	Dosažené ohodnocení
10 znaků	0.24 ms	0.25 ms	10
60 znaků	1.34 ms	1.38 ms	-40
120 znaků	2.02 ms	2.03 ms	-90
250 znaků	5.11 ms	5.12 ms	-70
350 znaků	14.12 ms	14.18 ms	-80
600 znaků	37.66 ms	37.72 ms	30
1000 znaků	117.24 ms	117.60 ms	2190

Tabulka 6.2: Testy provedené na Needleman-Wunch algoritmu

V tabulce 6.3 jsou uvedeny statistiky základních testů provedených na algoritmu Needleman-Wunch pro sadu B. Algoritmus prochází celou matici nehledě na sekvence, není důvod tedy přeměřovat rychlost provedení, jelikož by měla být stejná, ohodnocení sekvencí zde uvedu, protože na něj budu odkazovat při porovnávání výsledků testů dalších algoritmů.

Velikost	Dosažené ohodnocení
10 znaků	30
60 znaků	150
120 znaků	310
250 znaků	720
350 znaků	1010
600 znaků	1440
1000 znaků	3840

Tabulka 6.3: Testy provedené na Needleman-Wunch algoritmu

X-drop

V tabulce 6.4 jsou uvedeny statistiky základních testů provedených X-drop algoritmu, s velikou hranicí snížení. Protože byly sekvence hodně odlišné, tak ani s vysokou hranicí snížení nebylo dosaženo optimálního skóre a navíc byl průběh testů delší, než u Needleman-Wunch algoritmu.

Velikost	Rychlost hodnocení	Celková rychlost	Dosažené ohodnocení	Hranice snížení
10 znaků	0.29 ms	0.31 ms	14	14
60 znaků	0.66 ms	0.67 ms	14	84
120 znaků	1.81 ms	1.81 ms	14	168
250 znaků	6.99 ms	6.99 ms	14	350
350 znaků	23.40 ms	23.41 ms	14	490
600 znaků	57.39 ms	57.48 ms	89	840
1000 znaků	127.46 ms	127.60 ms	2190	1400

Tabulka 6.4: Testy provedené na X-drop algoritmu s velikou hranicí snížení

V tabulce 6.5 jsou uvedeny statistiky základních testů provedených na X-drop algoritmu, se čtvrtinou předchozí hranice snížení. Protože hranice byla nižší, tak se procházela menší část matice a výpočet byl rychlejší, ale stejně jako v předchozím testu nebylo dosaženo optimálního skóre. To je samozřejmě dáno tím, že čím větší je hranice, tím větší část matice se prochází a u odlišných sekvencí je tedy daleko větší šance najít optimální zarovnání. V předchozím testu nebylo optimální skóre nalezené a v tomto testu byla ještě snížena hranice, proto optimální řešení nemohlo být nalezeno.

Velikost	Rychlost hodnocení	Celková rychlost	Dosažené ohodnocení	Hranice snížení
10 znaků	0.06 ms	0.07 ms	0	3.5
60 znaků	0.35 ms	0.35 ms	14	21
120 znaků	0.21 ms	0.22 ms	14	42
250 znaků	1.17 ms	1.19 ms	14	87.5
350 znaků	4.44 ms	4.50 ms	14	122.5
600 znaků	26.09 ms	26.12 ms	89	210
1000 znaků	168.97 ms	170.24 ms	2190	350

Tabulka 6.5: Testy provedené na X-drop algoritmu s menší hranicí snížení

V tabulce 6.6 jsou uvedeny statistiky základních testů provedených na X-drop algoritmu u podobnějších sekvencí, s velikou hranicí snížení. Zde je vidět, že pro sadu B algoritmus proběhne rychleji. Doba běhu je jen o trochu pomalejší, než Needleman-Wunch. Podle hodnot získaného skóre se dá předpokládat, že se algoritmus blíží zarovnání získané Needleman-Wunch, akorát usekne konec, protože by došlo ke snížení skóre.

Velikost	Rychlost hodnocení	Celková rychlost	Dosažené ohodnocení	Hranice snížení
10 znaků	0.18 ms	0.19 ms	30	14
60 znaků	0.40 ms	0.42 ms	165	84
120 znaků	1.60 ms	1.61 ms	310	168
250 znaků	6.61 ms	6.62 ms	720	350
350 znaků	11.57 ms	11.58 ms	1010	490
600 znaků	41.72 ms	41.76 ms	1470	840
1000 znaků	89.17 ms	89.31 ms	3840	1400

Tabulka 6.6: Testy provedené na X-drop algoritmu s velikou hranicí snížení

V tabulce 6.7 jsou uvedeny statistiky základních testů provedených na X-drop algoritmu, se čtvrtinou předchozí hranice snížení. Při zmenšení hranice se podstatně snížila doba běhu a u větších sekvencí nedochází ke změně dosaženého ohodnocení.

Velikost	Rychlost hodnocení	Celková rychlost	Dosažené ohodnocení	Hranice snížení
10 znaků	0.02 ms	0.02 ms	0	3.5
60 znaků	0.72 ms	0.73 ms	84	21
120 znaků	0.33 ms	0.36 ms	234	42
250 znaků	2.92 ms	3.01 ms	720	87.5
350 znaků	5.57 ms	5.60 ms	1010	122.5
600 znaků	9.39 ms	9.42 ms	1470	210
1000 znaků	33.03 ms	33.33 ms	3840	350

Tabulka 6.7: Testy provedené na X-drop algoritmu s menší hranicí snížení

Greedy x-drop

V tabulce 6.8 jsou uvedeny statistiky základních testů provedených na Greedy x-drop algoritmu. Co se ohodnocení týče, Greedy x-drop je lehce nepřesný a bude se tedy lehce lišit oproti klasickému X-dropu. Důležitá je však doba běhu, zde pro velice odlišné sekvence je horší, než u klasického X-drop algoritmu.

Velikost	Rychlost hodnocení	Celková rychlost	Dosažené ohodnocení	Hranice snížení
10 znaků	0.13 ms	0.14 ms	0	3.5
60 znaků	0.91 ms	0.93 ms	8	21
120 znaků	2.45 ms	2.49 ms	8	42
250 znaků	4.02 ms	4.05 ms	8	87.5
350 znaků	10.14 ms	10.15 ms	8	122.5
600 znaků	85.46 ms	85.90 ms	83	210
1000 znaků	168.97 ms	170.24 ms	2190	350

Tabulka 6.8: Testy provedené na Greedy x-drop algoritmu

V tabulce 6.9 jsou uvedeny statistiky základních testů provedených na Greedy x-drop algoritmu pro sadu B. Když se podíváme na statistiky a porovnáme s klasickým X-dropem, potom si všimneme, že pro 10-350 znaků je rychlost podstatně větší a pro 600 znaků je pomalejší. Zde bychom mohli dojít k mylnému závěru, že pro větší sekvence je tento algoritmus pomalejší, to ale není důvodem. Greedy x-drop je náchylnější na podobnost sekvencí a pokud se podíváme na rozdíl ohodnocení 350 znaků a 600 znaků lze poznat, že u sekvencí o 600 znacích je v druhé polovině více rozdílů, to podstatně zpomaluje algoritmus.

Velikost	Rychlost hodnocení	Celková rychlost	Dosažené ohodnocení	Hranice snížení
10 znaků	0.05 ms	0.05 ms	0	3.5
60 znaků	0.57 ms	0.59 ms	78	21
120 znaků	0.28 ms	0.30 ms	228	42
250 znaků	2.59 ms	2.72 ms	720	87.5
350 znaků	2.55 ms	2.76 ms	1010	122.5
600 znaků	12.21 ms	12.40 ms	1464	210
1000 znaků	30.39 ms	31.02 ms	3840	350

Tabulka 6.9: Testy provedené na Greedy x-drop algoritmu

FOGSAA

U následujících testů bych rád podotknul, ačkoli to nerad přiznávám, že by se moje implementace FOGSAA algoritmu dala urychlit. I kdybych nebral v potaz menší úpravu pro vykreslení, která zpomaluje i měření, tak je několik možných míst, kde je možné tuto implementaci optimalizovat. Proto by se zdejší hodnoty měli přiřazovat mé implementaci a ne algoritmu při maximální optimalizaci.
V tabulce 6.10 jsou uvedeny statistiky základních testů provedených na FOGSAA algoritmu s hranicí podobnosti 30. Jedná se o hranici, pokud větev má menší potenciál, než je tato

hranice, potom je zahozena. Tato hodnota byla zvolena autory intuitivně a byla doporučena jako výchozí.

Velikost	Rychlost hodnocení	Celková rychlost	Dosažené ohodnocení
10 znaků	0.50 ms	0.51 ms	10
60 znaků	6.51 ms	6.54 ms	-50
120 znaků	10.81 ms	10.85 ms	-100
250 znaků	67.42 ms	67.44 ms	-220
350 znaků	170.06 ms	170.09 ms	-290
600 znaků	964.09 ms	964.14 ms	-270
1000 znaků	37585.57 ms	37585.90 ms	2190

Tabulka 6.10: Testy provedené na FOGSAA algoritmu

V tabulce 6.11 jsou uvedeny statistiky základních testů provedených na FOGSAA algoritmu s nastavenou větší hranicí podobnosti, tedy 60. Zde je vidět, že můžeme za cenu přesnosti nalezeného ohodnocení algoritmus podstatně urychlit.

Velikost	Rychlost hodnocení	Celková rychlost	Dosažené ohodnocení
10 znaků	0.33 ms	0.34 ms	10
60 znaků	4.14 ms	4.17 ms	-80
120 znaků	3.90 ms	3.95 ms	-130
250 znaků	33.81 ms	34.40 ms	-350
350 znaků	56.35 ms	56.38 ms	-510
600 znaků	302.73 ms	302.78 ms	-640
1000 znaků	9152.80 ms	9153.13 ms	1830

Tabulka 6.11: Testy provedené na FOGSAA algoritmu s větší hranicí podobnosti

Statistiky základních testů provedených na FOGSAA algoritmu pro sekvence s větší podobností s hranicí podobnosti 30.

Velikost	Rychlost hodnocení	Celková rychlost	Dosažené ohodnocení
10 znaků	0.33 ms	0.34 ms	30
60 znaků	2.82 ms	2.84 ms	150
120 znaků	13.92 ms	14.03 ms	310
250 znaků	66.93 ms	66.95 ms	720
350 znaků	152.16 ms	152.19 ms	1010
600 znaků	943.79 ms	943.85 ms	1440
1000 znaků	1628.75 ms	1629.09 ms	3840

Tabulka 6.12: Testy provedené na FOGSAA algoritmu

V tabulce 6.13 jsou uvedeny statistiky základních testů provedených na FOGSAA algoritmu s nastavenou větší hranicí podobnosti pro sekvence s větší podobností, opět 60. Zde jsme opět za cenu ohodnocení podstatně zkrátili dobu běhu algoritmu.

Velikost	Rychlost hodnocení	Celková rychlost	Dosažené ohodnocení
10 znaků	0.53 ms	0.54 ms	30
60 znaků	1.02 ms	1.05 ms	50
120 znaků	8.35 ms	8.46 ms	300
250 znaků	45.43 ms	45.55 ms	700
350 znaků	91.81 ms	91.83 ms	1000
600 znaků	385.09 ms	385.14 ms	1220
1000 znaků	1669.45 ms	1669.77 ms	3840

Tabulka 6.13: Testy provedené na FOGSAA algoritmu s větší hranicí podobnosti

Nussinov

V tabulce 6.14 jsou uvedeny statistiky základních testů provedených na Nussinov algoritmu. Protože Nussinov i Zuker ohodnocují podle jiného principu, nebudu samotná ohodnocení uvádět a porovnávat.

Velikost	Rychlost hodnocení	Celková rychlost
10 znaků	0.13 ms	0.15 ms
60 znaků	1.37 ms	1.39 ms
120 znaků	1.72 ms	1.77 ms
250 znaků	12.32 ms	12.35 ms
350 znaků	32.81 ms	33.00 ms
600 znaků	219.60 ms	219.93 ms

Tabulka 6.14: Testy provedené na Nussinov algoritmu

Zuker

V tabulce 6.15 jsou uvedeny statistiky základních testů provedených na Zukerově algoritmu. Zde je podstatné navýšení doby běhu algoritmu. Zuker by však měl přinášet přesnější zarovnání, než Nussinov.

Velikost	Rychlost hodnocení	Celková rychlost
10 znaků	0.63 ms	0.66 ms
60 znaků	157.21 ms	157.24 ms
120 znaků	2528.44 ms	2528.50 ms
250 znaků	71365.19 ms	71365.40 ms
350 znaků	263000.39 ms	263000.69 ms

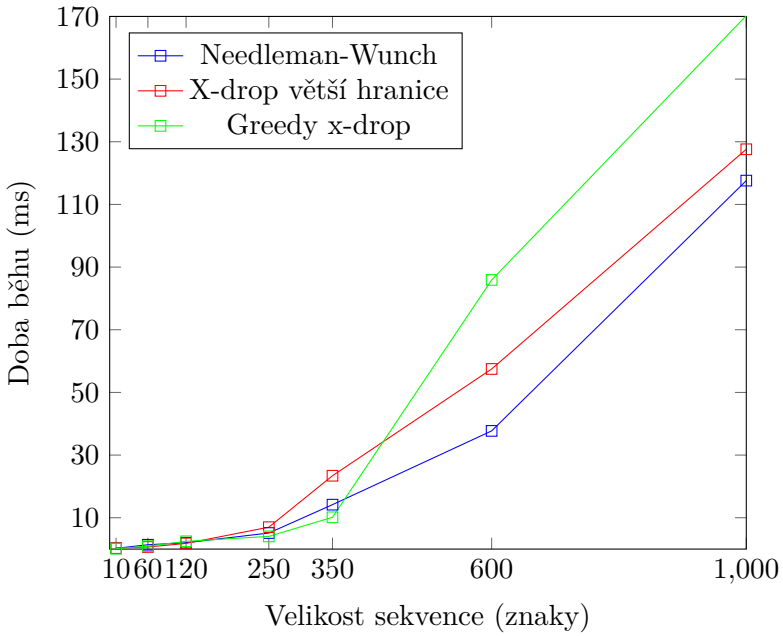
Tabulka 6.15: Testy provedené na Zuker algoritmu

Shrnutí rychlosti

Ze statistik všech algoritmů vidíme, že rychlost zpětného průchodu je v podstatě zanedbatelná.

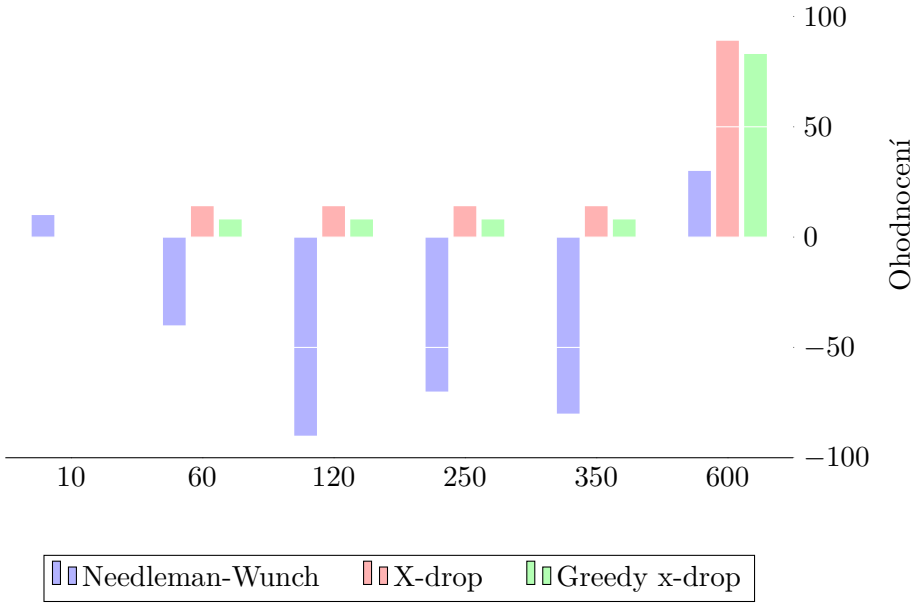
Porovnání hodnot zarovnání sekvencí

V této podkapitole porovnáme statistiky z provedených testů pro zarovnání sekvencí.



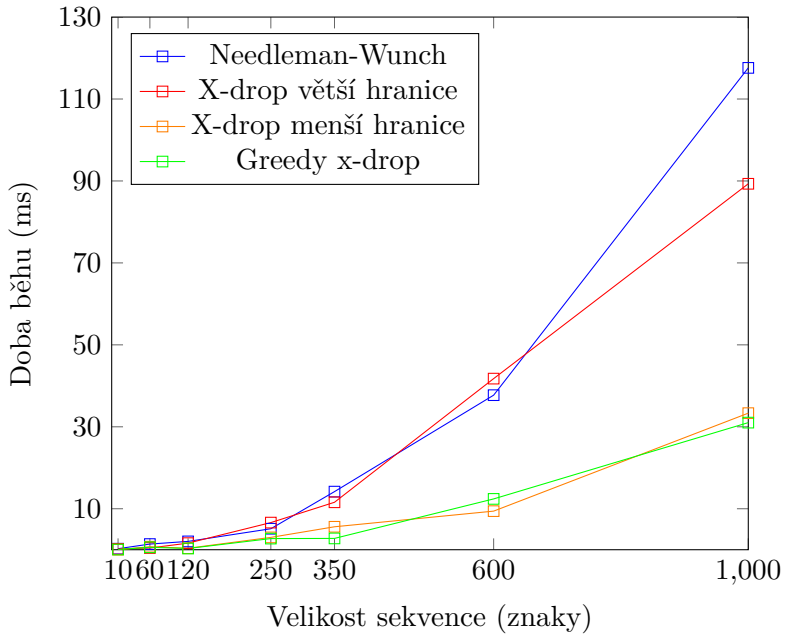
Obrázek 6.1: Doba běhu algoritmů u rozdílných sekvencí

V grafu 6.1 vidíme, že X-drop i Greedy x-drop mají daleko větší nárůst, něž Needleman-Wunch pro rozdílné sekvence. Tento nárůst se snížil pro 1000 znaků, protože se podstatně zvýšila podobnost sekvencí. I přes zvýšení podobnosti sekvencí má však Greedy x-drop podstatně větší nárůst.

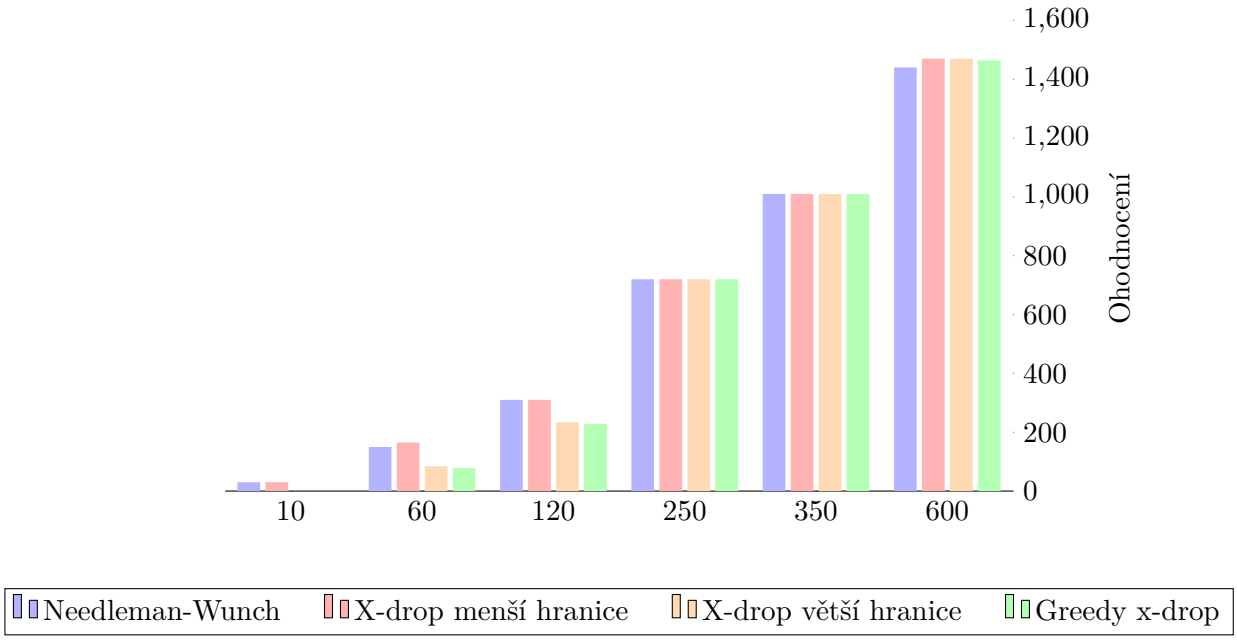


Obrázek 6.2: Dosažené ohodnocení algoritmů u rozdílných sekvencí

Pokud v grafu 6.2 bereme skóre získané Needleman-Wunch algoritmem jako optimální, potom je vidět, skóre získané zbylými algoritmy se optimálnímu ani zdaleka neblíží. Výjimkou je velikost sekvence o 1000 znacích, kde při vyšší podobnosti dosáhli všechny algoritmy stejného ohodnocení, nebylo nutné je tedy uvádět v grafu.



Obrázek 6.3: Doba běhu algoritmů u podobných sekvencí



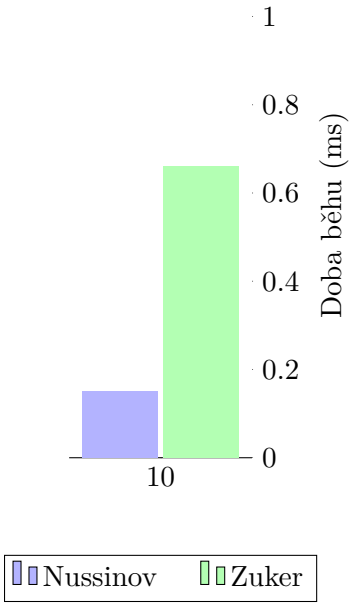
Obrázek 6.4: Dosažené ohodnocení algoritmů u podobných sekvencí

Pokud porovnáváme testy pro podobné sekvence uvedené v grafu 6.3, potom je vidět, že X-drop a Greedy x-drop jsou vhodnější, za předpokladu, že je dobře zvolena hranice snížení. Nárůst doby výpočtu je podstatně menší se správně zvolenou hranicí snížení.

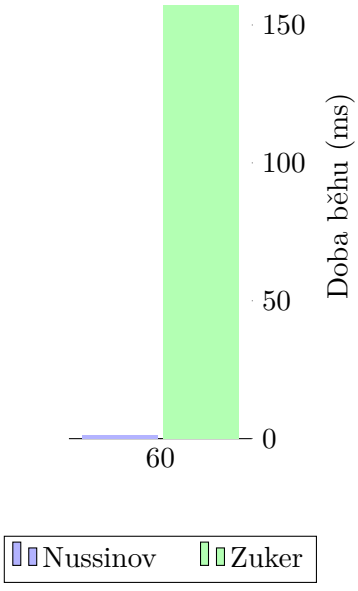
V grafu 6.4 porovnáváme ohodnocení získané pro podobné sekvence. Zde je vidět, že všechny algoritmy dosahují optimálního skóre. Výjimkou jsou zde menší sekvence, kde by bylo třeba pro dosažení ideálního skóre zvýšit hranici. Opět pro sekvence o 1000 znacích dosáhli všechny algoritmy stejného ohodnocení, nebylo nutné je tedy uvádět v grafu.

Porovnání hodnot predikce sekundární struktury

V této podkapitole porovnáme statistiky z provedených testů pro predikci sekundární struktury.



Obrázek 6.5: Rychlost predikce sekundární struktury



Obrázek 6.6: Rychlost predikce sekundární struktury

Pokud porovnáme nárůst doby z grafu 6.6 běhu oproti grafu 6.5. Je poznat, že nárůst při zvětšení délky sekvence u Zuker algoritmu mnohonásobně větší, než u Nussinov algoritmu. To je způsobeno tím, že algoritmus prověřuje více možností, pouze ve výjimečných ideálních případech je časová složitost těchto algoritmů rovna.

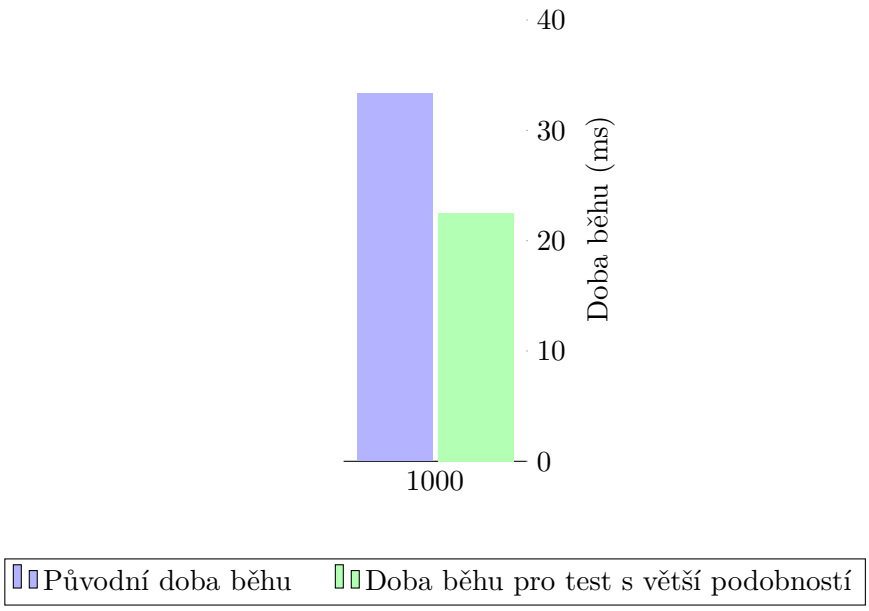
Dodatečné otestování zarovnání sekvencí DNA

Vzhledem k tomu, že předchozí testy zarovnání sekvencí nezobrazovali dostatečně závislost X-drop algoritmu a Greedy x-drop algoritmu na podobnosti sekvence, je zde uveden test pro zarovnání dvou sekvencí o 1000 znacích. Tyto sekvence jsou téměř identické. V těchto sekvencích se po každých 95 znacích liší 5 znaků. Z této informace se dá odvodit, že se výsledné zarovnání bude pravděpodobně nacházet přímo na diagonále. Nastavené parametry jako například ohodnocení mezery jsou identické s předchozími testy. Hranice snížení nebyla měněna a byla tedy nastavena na 350 pro Greedy x-drop i X-drop algoritmus, i když by v tomto případě mohla být menší, což by ještě zvýšilo rychlost průběhu algoritmů.

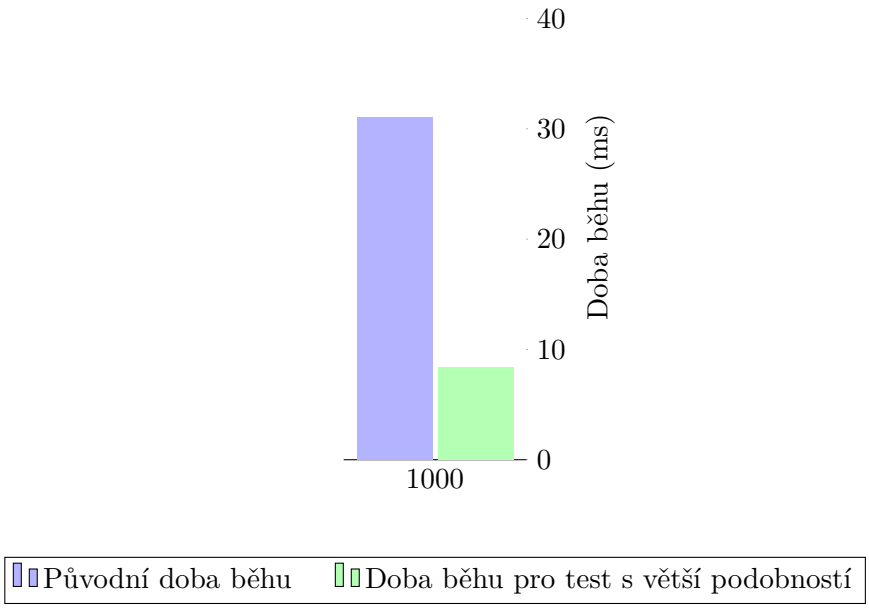
Algoritmus	Rychlost hodnocení	Celková rychlost	Dosažené ohodnocení
Needleman-Wunch	111.77 ms	112.00 ms	5500
X-drop	22.22 ms	22.50 ms	5520
Greedy x-drop	8.02 ms	8.35 ms	5514

Tabulka 6.16: Testy provedené na velmi podobných sekvencích

Needleman-Wunch algoritmus je zde uveden hlavně pro porovnání. Důležité je se však zaměřit na X-drop algoritmus a porovnat hodnotu rychlosti v tabulce 6.7 a novou hodnotu uvedenou v tabulce 6.16. To samé můžeme udělat pro Greedy x-drop a porovnat hodnotu v tabulce 6.9 a zde v tabulce 6.16. Tyto hodnoty jsou porovnány na následujících grafech. X-drop je uveden v grafu 6.7 a Greedy x-drop je uveden v grafu 6.8.



Obrázek 6.7: Porovnání rychlosti X-drop algoritmu podle podobnosti sekvence



Obrázek 6.8: Porovnání rychlosti Greedy x-drop algoritmu podle podobnosti sekvence

Z grafu 6.7 a 6.8 můžeme vypozorovat, že i když rychlost X-drop algoritmu se zvýší znatelně, Greedy x-drop je schopen za těchto podmínek ohodnotit sekvence podstatně rychleji. Pro velmi podobné sekvence je tedy výhodnější použít Greedy x-drop, jeho rychlost ohodnocení však klesá daleko rychleji s odlišnostmi v sekvencích, než u klasického X-drop algoritmu.

Kapitola 7

Závěr

Tato práce se zaměřuje na efektivní algoritmy využívající dynamické programování pro řešení bioinformatických úloh. V rámci této práce byly nastudovány algoritmy určené pro zarovnání sekvencí a predikce sekundární struktury. Úvodní kapitoly popisují efektivitu algoritmu, zarovnání sekvencí DNA a predikci sekundární struktury RNA, nakonec také jednotlivé algoritmy použitelné pro řešení daných úloh. Uvedené jsou standardně používané algoritmy jako X-drop, ale také méně známé varianty jako Greedy x-drop.

Dále byla vytvořena aplikace implementující efektivní algoritmy, která je dostupná na adrese http://www.stud.fit.vutbr.cz/~xfrane16/. Pomocí této aplikace je algoritmy možné jak krokovat jednotlivé algoritmy, tak i měřit čas potřebný pro provedení algoritmu. Jednotlivé kroky je možné provádět při stisknutí klávesy, nebo se provedou po určitém čase. Pokud potřebujeme, můžeme vykreslit matici okamžitě, to je možné provést jak během krokování, tak zvolením před spuštěním samotného algoritmu. Algoritmy jsou založené na dynamickém programování, které pracuje většinou s polem, proto je v jednom kroku aktualizována hodnota pole. Výjimkou je FOGSAA algoritmus, který není založen na principu dynamického programování. Samotná měření se ukládají do statistik a je možné je po ukončení měření procházet. Také je možné nahrát statistiky uložené ve formátu JSON.

Jedná se o webovou aplikaci zejména pro její dostupnost. Jazykem použitým pro implementaci byl Javascript. Pro tvorbu uživatelského rozhraní byly použity jazyky HTML a CSS. Pro zjednodušení vytváření rozhraní byl využit framework Bootstrap.

Na algoritmech byly provedeny jednotlivé testy, které sloužily pro získání časové složitosti. Testovány byly odlišně dlouhé sekvence. Výsledky těchto testů byly porovnány mezi jednotlivými algoritmy, pro určení jejich efektivity. Porovnání bylo také vizualizováno graficky. Pro zarovnání sekvencí bylo zjištěno, že čím podobnější sekvence, tím je výhodnější použít Greedy x-drop, při porovnání méně podobných sekvencí je výhodnější použít klasický X-drop. Pro odlišné sekvence je výhodnější použít Needleman-Wunch algoritmus. Při predikování sekundární struktury je rychlejší Nussinov, ale obecně podává horší výsledky.

Literatura

[1] Baxevanis, A.; Ouellette, F.: *Bioinformatics a practical guide to the analysis of genes and proteins*. 111 River Street, New Jersey: John Wiley & Sons, třetí vydání, 2005, ISBN 0-471-47878-4, 144–148 s.

[2] Burgetová, I.; Martínek, T.: Predikce struktury RNA. online.
URL https://wis.fit.vutbr.cz/FIT/st/cfs.php?file=%2Fcourse%2FBIF-IT%2Flectures%2Fpredikce_RNA.pdf

[3] Chakraborty; Bandyopadhyay, A. .: FOGSAA: Fast Optimal Global Sequence Alignment Algorithm. *Scientific Reports*, ročník 3, 04 2013.

[4] Guan, D.: Introduction to Dynamic Programming with Examples. online, 05 2018.
URL https://medium.com/@davidguandev/introduction-to-dynamic-programming-with-examples-bc04dca3ccee

[5] Huson, D.; Gropl, C.: RNA Secondary Structure. online.
URL http://www.mi.fu-berlin.de/wiki/pub/ABI/RnaAnalysis/rna.pdf

[6] M: Nussinov algorithm to predict secondary RNA fold structures. online, 02 2019.
URL https://bayesianneuron.com/2019/02/nussinov-predict-2nd-rna-fold-structure-algorithm/

[7] Mareš, M.; Valla, T.: *Průvodce labyrintem algoritmů*. Milešovská 5, 130 00 Praha 3: CZ.NIC, z. s. p. o., první vydání, 2017, ISBN 978-80-88168-22-5.

[8] Martínek, T.: Zarovnání sekvencí. online.
URL https://wis.fit.vutbr.cz/FIT/st/cfs.php?file=%2Fcourse%2FBIF-IT%2Flectures%2Fzarovnani_sekvenci.pdf

[9] PBworks: Fast Sequence Similarity Search Algorithms. online.
URL http://compbio.pbworks.com/w/page/16252890/Fast%20Sequence%20Similarity%20Search%20Algorithms

[10] Richta, K.: Složitost algoritmů. online.
URL https://cw.fel.cvut.cz/b182/_media/courses/b6b36dsa/dsa-3-slozitostalgoritmu.pdf

[11] Töpfer, P.: Rekurze. online.
URL https://ksvi.mff.cuni.cz/~topfer/Texty/TextRekurze.pdf

[12] Urgese, G.; Paciello, G.; Acquaviva, A.; aj.: Dynamic Gap Selector: A Smith Waterman Sequence Alignment Algorithm with Affine Gap Model Optimisation.

online, 04 2014.
URL http://iwbbio.ugr.es/2014/papers/IWBBIO_2014_paper_143.pdf

[13] Yu, L.: Study of RNA Secondary Structure Prediction Algorithms. online.
URL http://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1022&context=etd_projects

[14] Zhang, L.: Implementation of basic RNA structure prediction algorithms. online.
URL http://ad-publications.informatik.uni-freiburg.de/student-projects/rna-google/

[15] Zhang, Z.; Schwartz, S.; Wagner, L.; aj.: A Greedy Algorithm for Aligning DNA Sequences. *Journal of computational biology*, ročník 7, 02 2000.

[16] Zvelebil; Marketa: *Understanding bioinformatics*. New York: Garland Science, 2008, ISBN 978-0-8153-4024-9.