

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

FIIT-5208-46165

Bc. Juraj Šimek

Skladanie DNA sekvencií

Diplomová práca

Študijný program: Informačné systémy

Študijný odbor: 9.2.6 Informačné systémy

Miesto vypracovania: Ústav informatiky, informačných systémov a softvérového inžinierstva

Vedúci práce: Mgr. Gabriela Grmanová, PhD.

máj 2016

Anotácia

Slovenská technická univerzita v Bratislave

FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLOGIÍ

Študijný program: Informačné systémy

Autor: Bc. Juraj Šimek

Diplomová práca: Skladanie DNA sekvencií

Vedúci diplomovej práce: Mgr. Gabriela Grmanová, PhD.

máj 2016

Výskum DNA má v súčasnosti veľký význam. Na to, aby bolo možné DNA skúmať, je ju potrebné osekvenovať pomocou prístrojov, ktoré sa nazývajú sekvencery DNA. Tieto prístroje vedia prečítať DNA molekulu a previesť ju na textovú reprezentáciu postupnosti nukleotidov, ktoré danú DNA molekulu tvoria. DNA organizmu však nie je prečítaná ako celok, ale sú prečítané len jej malé, rôzne sa prekrývajúce časti, nazývané krátke sekvencie. Kvôli výskumu DNA treba tieto krátke sekvencie poskladať, o čo sa starajú programy zvané skladače DNA sekvencií. V tejto práci opisujeme štruktúru DNA a proces sekvenovania a skladania DNA. Opisujeme aj viaceré algoritmy umelej inteligencie, ktoré boli aplikované na skladanie DNA sekvencií. Ďalej sme navrhli a implementovali skladač DNA sekvencií používajúci Paralelný imunitný algoritmus s pamäťou, klonovaním, výberom a distribuovanou operáciou zrenia protilátok (skr. PIMCSA-DMO), ktorý využíva Lin-Kernighanovu heuristiku a Vylepšený Iver-over operátor. Nakoniec sme vykonali testovanie nášho riešenia a zhodnotili jeho prínos.

Annotation

Slovak University of Technology Bratislava

FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES

Degree Course: Information Systems

Author: Bc. Juraj Šimek

Master's thesis: DNA short reads assembly

Supervisor: Mgr. Gabriela Grmanová, PhD.

2016, May

Research of DNA is a very important thing nowadays. For the purpose of a DNA research, it is important to sequence DNA using machines called DNA sequencers. These machines are able to read DNA molecule and transform it to the text representation of molecule's nucleotides. Nucleotides are building blocks of DNA. DNA of an organism is not read as a single long sequence, but only small parts of DNA, called short reads, are read. It is important to assemble reads to the original sequence for the purpose of DNA research. Assembly is performed by programs called DNA assemblers. We described the structure of DNA and the process of DNA sequencing and assembly in this work. We also described several algorithms of artificial intelligence which were applied to the fragment assembly problem. Then we designed and implemented DNA sequence assembler using Parallel Immune Memory Clonal Selection Algorithm with Distributed Maturation Operation (PIMCSA-DMO), which uses Lin-Kernighan heuristic and Improved Inver-over operator. Finally we tested our assembler and reviewed its benefits.

POĎAKOVANIE

Fakulta informatiky a informačných technológií bola v uplynulých piatich rokoch každodennou súčasťou môjho života. Jednou z najväčších výziev, ktorým som na nej čelil bolo práve riešenie tohto diplomového projektu. No jeho úspešné dokončenie nebolo samozrejmosťou. Chcem sa preto poďakovať predovšetkým Pánu Bohu za múdrosť, bez ktorej by som túto prácu nemohol dokončiť. Chcem sa tiež poďakovať Mgr. Gabriele Grmanovej, PhD. za jej vedenie najmä v prvých dvoch semestroch riešenia tohto diplomového projektu. Veľká vďaka patrí aj RNDr. Anne Bou Ezzeddine, PhD. za ochotu konzultovať postup na tejto diplomovej práci v posledných dvoch semestroch môjho štúdia, počas neprítomnosti pani Grmanovej. Ďakujem tiež Klárke za každé jej povzbudenie a mojej rodine, lebo bez ich podpory a pomoci by som sa nemohol venovať škole tak, ako som chcel. Veď nie každý má to šťastie, že mu výdavky spojené so školou môžu zaplatiť rodičia, niektorí z nás sú nútení pracovať, len aby mali z čoho financovať svoje štúdium. A práca stojí čas, ktorý som ja mohol venovať škole. Ďakujem aj profesorovi Davidovi Johnsonovi, ktorý si ako jediný z kontaktovaných zahraničných odborníkov našiel čas odpísať na môj mail. Vďaka patrí aj iným nemenovaným ľuďom, ktorí sa viac, či menej zaslúžili o to, že som mohol dokončiť túto diplomovú prácu v jej dnešnej podobe. A na záver ďakujem Fakulte informatiky a informačných technológií za všetky ťažké chvíle, lebo v nich som spoznal, že dokážem viac, ako som si myslel.

ČESTNÉ PREHLÁSENIE

Čestne prehlasujem, že záverečnú prácu som vypracoval samostatne s použitím uvedenej literatúry a na základe svojich vedomostí a znalostí.

.....

Bc. Juraj Šimek

Obsah

1	Úvod	1
1.1	Použité pojmy a skratky	2
2	Analýza	3
2.1	DNA	3
2.1.1	Primárna štruktúra DNA	4
2.1.2	Sekundárna štruktúra DNA	4
2.1.3	Terciárna štruktúra DNA	5
2.1.4	Zabalenie DNA a štruktúra chromozómov	6
2.1.5	Replikácia DNA	7
2.1.6	Vnútorná stavba DNA	8
2.1.7	Kódujúca DNA	8
2.1.8	Nekódujúca DNA	8
2.2	Sekvenovanie DNA	10
2.2.1	Typy DNA sekvencií	11
2.2.2	Sangerovo sekvenovanie	12
2.2.3	Pyrosekvenovanie	14
2.2.4	Sekvenovanie s použitím reverzibilných terminátorov	15
2.2.5	Sekvenovanie pomocou ligázy	16
2.2.6	Formáty pre ukladanie DNA sekvencií	17
2.3	Problém skladania DNA sekvencií	19
2.3.1	Najkratší spoločný nadreťazec (SCS)	20
2.4	Tradičné metódy skladania krátkych DNA sekvencií	21
2.4.1	Overlap-Layout-Consensus	21
2.4.2	Riešenie SCS hľadaním Hamiltonovskej kružnice	23
2.4.3	Riešenie SCS hľadaním Eulerovského ťahu	23
2.4.4	k-Reťazcové grafy	26
2.5	Umelá inteligencia v riešeníach problému skladania DNA sekvencií	27
2.5.1	Skladanie DNA sekvencií a TSP	27
2.5.2	Genetický algoritmus (GA)	28
2.5.3	Simulované žihanie (SA)	30
2.5.4	Optimalizácia kolóniou mravcov (ACO)	31
2.5.5	Umelé spoločenstvo včiel (ABC)	33
2.5.6	Modifikovaný genetický algoritmus inšpirovaný včelou kráľovnou (QEGA)	35
3	Návrh	37
3.1	Návrh skladača DNA sekvencií	37
3.2	Načítanie sekvencií a transformácia FAP na TSP	38

3.2.1	Redukcia dimenzionality problému obchodného cestujúceho	41
3.3	Riešenie problému obchodného cestujúceho	42
3.3.1	Paralelný optimalizačný model inšpirovaný imunitným systémom	42
3.3.2	Aplikácia PIMCSA-DMO na riešenie TSP	45
3.3.3	Lin-Kernighanova heuristika	46
3.3.4	Iterovaná Lin-Kernighanova heuristika	48
3.3.5	Vylepšený Inver-over operátor	49
3.4	Konečné spájanie sekvencií	50
4	Implementácia	53
4.1	Implementácia skladača DNA sekvencií	53
4.1.1	Základné implementačné súbory skladača DNA sekvencií	53
4.1.2	Implementačné súbory pre riešenie TSP	56
4.1.3	Implementačné súbory Lin-Kernighanovej heuristiky	61
4.1.4	Implementačné súbory testovacieho režimu	62
4.2	Framework pre skladanie DNA sekvencií pomocou riešenia TSP	62
4.3	Použitie skladača DNA sekvencií	65
5	Testovanie	67
5.1	Testovanie PIMCSA-DMO s použitím TSPLIB inštancií	67
5.2	Testovanie PIMCSA-DMO pomocou GenFrag inštancií	72
5.3	Testovanie aplikácie PIMCSA-DMO na skladanie DNA sekvencií	73
5.3.1	Testovacie dáta	74
5.3.2	Správnosť skladania sekvencií	74
5.3.3	Časy behu skladača DNA sekvencií	78
5.4	Vplyv modulu Compactor na redukciu dimenzionality TSP	78
5.5	Porovnanie algoritmov PIMCSA-DMO a PIMCSA	79
6	Zhodnotenie a budúca práca	81
A	DVD príloha	87
B	Inštalčná príručka	88
C	Opis nastavení skladača DNA sekvencií	89
D	Článok publikovaný na IIT.SRC 2016	92

Zoznam obrázkov

- 1 Na obrázku je zachytená štruktúra DNA. Každý nukleotid sa skladá zo zvyšku kyseliny fosforečnej, deoxyribózy a dusíkatej bázy, jeden z nukleotidov je na obrázku zvýraznený. Jednotlivé nukleotidy sú pospájané fosfodiesterovými väzbami. Dve komplementárne vlákna sú spárované pomocou vodíkových väzieb (prerušované čiary). Každý DNA reťazec má smer od 5' do 3', pričom reťazce DNA molekuly sú antiparalelné. Obrázok taktiež znázorňuje číslovanie uhlíkov deoxyribózy, odkiaľ pochádza označenie dopredného smeru DNA (od 5' do 3'). Prevzaté z [22]. 5
- 2 Spôsob zabalenia DNA do chromozómov. Chromozóm tvorený dvoma totožnými chromatidovými vláknami sa stáča okolo histónov a tvorí nukleozómy. Tie potom po ďalšom zvinutí tvoria chromozómy. Miesto, kde sa chromatidové vlákna krížia sa nazýva centroméra. Konce chromozómov sú teloméry. Prevzaté z [51]. 6
- 3 Obrázok znázorňuje činnosť jednotlivých enzýmov v procese replikácie DNA. Ich bližší opis je v časti 2.1.5. Prevzaté z [29]. 8
- 4 Obrázok znázorňuje orientáciu jednotlivých častí párovej sekvencie. Jednotlivé zložky (označené sivou), vždy lokalizované na dvoch antiparalelných reťazcoch, môžu smerovať buď k sebe, alebo od seba. Priestor medzi týmito zložkami sa nazýva vloženie. 12
- 5 Vrchný riadok predstavuje sekvenovaný reťazec DNA, pod ním sa pre zjednodušenie nasledujúceho opisu nachádza jeho reverzný komplement, keďže DNA polymeráza "kopíruje" reťazec tak, že vystavuje jeho reverzný komplement. Menšie reťazce predstavujú reťazce, ktoré vznikli ako výsledok činnosti DNA polymerázy. Kurzívou a sivou farbou sú označené dideoxynukleotidy, ktoré, keď sa pripojili k reťazcu, spôsobili ukončenie syntézy DNA. Posledný riadok predstavuje situáciu, kedy nebol pripojený žiadny dideoxynukleotid. Pri dostatočnom počte kópií bude každá z možností zastúpená v dostatočnom počte. 13
- 6 Obrázok ukazuje, ako môže vyzeráť film, ktorý vznikol ako výsledok Sangerovho sekvenovania. Rádioaktívne markery, ktorými boli označené dideoxynukleotidy vypálili na film tmavé miesta. Kratšie úseky (viac obrázok 5) prešli v géle dlhšiu cestu. Výslednú sekvenciu získame postupným čítaním gélu od kladného konca k zápornému, v tomto prípade ide o sekvenciu AGTACCCTGAC. Keďže sme sekvenovaný reťazec klonovali, výsledná sekvencia predstavuje jeho reverzný komplement, sekvenovaný reťazec bol teda reťazec GTCAGGGTACT. 14
- 7 Na obrázku sú znázornené použitia farbív pre dve rôzne kombinácie nukleotidov v oktaméroch. Prevzaté z [22]. 16

8	Na obrázku je zachytený proces sekvenovania pomocou ligázy. Sekvenovaný fragment je pripojený k DNA čipu, je naň naviazaný primer a ligáza postupne pripája k primeru oktaméry, z ktorých prvé dva nukleotidy sú komplementárne k nukleotidom na sekvenovanom reťazci. Pripojenie oktaméru vyžiari merateľné svetlo, potom sa z neho odstráni posledné 3 nukleotidy a pokračuje sa pripojením ďalšieho oktaméru. Prevzaté z [15].	17
9	V grafe prekrytí reprezentujú uzly dané sekvencie DNA. Hrana medzi uzlami vznikne, ak je medzi dvoma uzlami (sekvenciami) dostatočné prekrytie. Na obrázku sú prekrytia vyznačené na hranách. Prevzaté z [49].	22
10	Sekvenovaním kruhového chromozómu ATGGCGTGCA sme získali sekvencie ATG-GCGT, GGCGTGC, CGTGCAA, TGCAATG a CAATGGC. Ich pospájaním do grafu prekrytí, nájdením Hamiltonovskej kružnice (postupnosť jej hrán značia čísla) a prečítaním prvých dvoch nukleotidov z každého uzla získame pôvodne sekvenovaný chromozóm. Prevzaté a upravené z [8].	23
11	Rozložením sekvencií na kratšie k -tice získame graf prekrytí na obrázku. Nájdením Hamiltonovskej kružnice (je vyznačená hrubo) a prečítaním prvého nukleotidu každej sekvencie získame osekvenovaný chromozóm. Prevzaté a upravené z [8].	24
12	Uvažujeme príklad z kapitoly 2.4.2. Obrázok a) znázorňuje de Bruijnov graf pre $k = 4$, obrázok b) de Bruijnov graf pre $k = 3$. Hrany predstavujú dané k -tice a uzly k nim prislúchajúce $(k - 1)$ -tice. Nájdením Eulerovskej kružnice získame sekvenovaný chromozóm.	25
13	Obrázok znázorňuje všetky štyri spôsoby, ako môžu byť spojené uzly v dvojito usmernenom k -reťazcovom grafe. Prevzaté z [22].	27
14	Ukážka grafu prekrytí pre riešenie FAP pomocou TSP. Hrany sú ohodnotené veľkosťou prekrytí medzi dvoma sekvenciami predstavujúcimi mestá.	28
15	Obrázok vľavo udáva, ako sa správala najľpšia fitness v populácii pre jednotlivé porovnávané algoritmy. Ako vidíme, simulované žihanie výrazne dominuje. Obrázok vľavo udáva najlepší počet kontigov, ktorý boli dané algoritmy schopné dosiahnuť pre 2 testované inštancie A a B. Prevzaté z [26].	31
16	Na obrázku je znázornený postup výpočtu prekrytia sekvencie A (horná sekvencia) so sekvenciou B (spodná sekvencia) v rámci operácie overlap. Obe sekvencie majú veľkosť 32 nukleotidov. Sekvenciu B posunieme vzhľadom na sekvenciu A o 1 nukleotid vpravo a porovnáme ich prekrývajúce časti. Ak sa nezhodujú pokračujeme s posúvaním až kým nenarazíme na zhodu. V tomto prípade veľkosť prekrytia predstavuje 28 nukleotidov.	39

- 17 Obrázok zachytáva asymetrickú vzdialenostnú maticu a jej transformáciu na symetrickú maticu. Neplatné prechody sú označené ako x a predstavujú nejaké veľké číslo. Nastavením diagonály matice C na $-\infty$ zvýhodňujeme cesty ako $AA'BB'CC'$, alebo $AB'BC'CA'$, ale cesty ako $A'BC'AB'C$ budú znevýhodnené. Neplatný prechod zase veľmi znevýhodní cesty ako $ABCA'B'C'$ alebo $AB'BCC'A'$. Najlepšou cestou je cesta $AA'BB'CC'$ predstavujúca cestu ABC v asymetrickej verzii s cenou 9. 40
- 18 Obrázok znázorňuje spôsob spájania DNA sekvencií do predčasných kontigov v moduli Compactor. Tri sekvencie A, B a C zdieľajú veľkú časť spoločných nukleotidov. Keďže sekvencia A má parameter `free` nastavený na `true`, spájanie začneme od nej. Na základe veľkostí prekrytí v parametroch `overlap` vieme určiť výsledné spojenie týchto troch sekvencií na $TCGACCACGTACGAATCT$, čím sme zredukovali veľkosť TSP inštancie o 2 mestá. 42
- 19 Obrázok znázorňuje schému double-bridge presunu. Túto schému možno vykonať náhodným výberom 4 miest c_0, c_1, c_2, c_3 v poradí, v akom sa nachádzajú v ceste T a následným odstránením hrán $(c_2, next(c_2)), (c_3, next(c_3)), (c_0, next(c_0)), (c_1, next(c_1))$ a vytvorením hrán $(c_2, next(c_0)), (c_3, next(c_1)), (c_0, next(c_2)), (c_1, next(c_3))$. Prevzaté a upravené z [3]. 48
- 20 Obrázok zachytáva časť cesty v TSP s uzlami (mestami) 0 - 4. S každým uzlom je spätá príslušná DNA sekvencia. Uzol 1 je uzol, do ktorého vedie hrana s najmenšou veľkosťou prekrytia, preto je zvolený za počiatočný uzol skladania. Ku prvému skladanému kontigu $C1$ teda pridáme celú sekvenciu prislúchajúcu k uzlu 1. Za ním nasledujú uzly 2 a 3, ktorých sekvencie ku kontigu $C1$ pridáme s ofsetom s veľkosťou rovnou veľkosti prekryvu, ktorý zachytávajú ich vstupné hrany. Na obrázku sú časti sekvencií 2 a 3 v kontigu $C1$ oddelené zvislou čiarou. Kritickú hodnotu pre delenie kontigov sme v tomto prípade nastavili na 2 a preto sekvencia 4, ktorá má so sekvenciou 3 spoločný 1 nukleotid už do sekvencie $C1$ pridaná nebude, ale vytvorí sa nový kontig $C2$, do ktorého bude skopírovaná celá a ďalej sa bude postupovať podobným spôsobom. 51
- 21 Obrázok znázorňuje jednotlivé moduly *TSP Solver*. Farebne odlišené sú triedy reprezentujúce algoritmus PIMCSA-DMO, ktorý je hlavným algoritmom riešiacim TSP použitým v našom skladači DNA sekvencií. Triedu `stsp_from_atsp` možno nahradiť za `symmetric_tsp_instance` podľa definície v `STSP_INST`. 64

Zoznam tabuliek

1	Konfigurácie Lin-Kernighanovej heuristiky použité pri testovaní riešenia TSP.	68
2	Výsledky riešenia TSP pomocou Lin-Kernighanovej heuristiky.	68
3	Konfigurácie Iterovanej Lin-Kernighanovej heuristiky použité pri testovaní riešenia TSP.	69
4	Výsledky riešenia TSP pomocou Iterovanej Lin-Kernighanovej heuristiky.	69
5	Konfigurácie PIMCSA-DMO použité pri testovaní riešenia TSP.	69
6	Výsledky riešenia TSP pomocou algoritmu PIMCSA-DMO.	70
7	Časy behov jednotlivých algoritmov pri riešení TSPLIB inštancií.	71
8	Porovnanie PIMCSA-DMO s ostatnými algoritmami pri riešení GenFrag inštancií. .	72
9	Konfigurácie PIMCSA-DMO použité pri testovaní riešenia TSP pomocou GenFrag inštancií.	73
10	Konfigurácie generovaných testovacích DNA sekvencií.	74
11	Výsledky skladania DNA sekvencií.	75
12	Bodovanie jednotlivých konfigurácií algoritmov na riešenie TSP.	77
13	Časy behov skladača DNA sekvencií pre rôzne vzorky.	78
14	Rozmery inštancií TSP s použitím a bez použitia modulu Compactor.	79
15	Porovnanie času behov algoritmov PIMCSA-DMO a PIMCSA v ms.	79

Zoznam ukážok

1	Pseudokód algoritmu simulovaného žihania.	30
2	Pseudokód algoritmu ABC.	34
3	Pseudokód algoritmu QEGA.	35
4	Operácia $cost(a, b)$ vracajúca cenu cesty medzi mestami a a b	40
5	Pseudokód algoritmu MIA.	43
6	Pseudokód algoritmu PIA.	44
7	Pseudokód prehľadávania <code>lk_search</code>	47
8	Pseudokód Lin-Kernighanovej heuristiky (funkcie <code>lin-kernighan()</code>).	48
9	Pseudokód Iterovanej Lin-Kernighanovej heuristiky.	49
10	Pseudokód Inver-over operátora.	49
11	Pseudokód Vylepšeného Inver-over operátora.	50
12	Činnosť skladača DNA sekvencií	63

1 Úvod

Výskum DNA má v súčasnosti veľký význam. DNA je nositeľkou genetickej informácie každého organizmu. Zvyčajne je ukrytá v bunkovom jadre, kde je rozdelená na niekoľko častí – *chromozómov*. DNA môžeme zjednodušene chápať ako postupnosť *nukleotidov*, pričom poznáme 4 druhy nukleotidov: adenín, guanín, cytozín a tymín. Pomocou DNA vieme odhaliť mnohé genetické poruchy, vieme určiť pravdepodobnosť výskytu nejakej choroby, diagnostikovať rakovinové bunky, určiť rodičovstvo, či dokázať prítomnosť podozrivého na mieste činu. DNA nám vie odhaliť tajomstvo mnohých procesov prebiehajúcich v živých bunkách a jej dôkladné preskúmanie je nevyhnutné pre pochopenie toho, ako život funguje. Na to, aby sme mohli DNA skúmať je potrebné ju najskôr *osekvenovať*, čím získame textovú reprezentáciu poradia jej nukleotidov. Na sekvenovanie sa v súčasnosti používajú najmä *sekvencery novej generácie* (angl. next-generation sequencers), ktoré však kvôli limitáciám chemických reakcií, pomocou ktorých prebieha sekvenovanie, nevedia prečítať dané chromozómy súvisle, ale ich výstupom je veľké množstvo *krátkych sekvencií* (angl. short reads) dĺžky 35 až 1000 nukleotidov, ktorých vzájomnú polohu nepoznáme. Na detailné skúmanie DNA sú však tieto sekvencie nepostačujúce, je potrebné ich poskladať podobne ako puzzle, do dlhších súvislých celkov. Práve o to sa snažia programy, ktoré nazývame *skladače DNA sekvencií* (angl. DNA short read assemblers). *Skladanie DNA sekvencií* (angl. Fragment Assembly Problem) však patrí do triedy *NP-ťažkých* problémov, čo z neho robí výpočtovo náročnú úlohu. Tento problém je navyše komplikovaný chybami, ktoré vznikli pri sekvenovaní DNA a prítomnosťou opakovaných úsekov v DNA. Tradičné algoritmy skladania DNA sekvencií sú schopné produkovať celkom dobré zloženia, no pre veľké genómy trvá ich beh veľmi dlho i na výkonných paralelných počítačoch. Stále má preto význam skúmať rôzne *optimalizačné metódy* použitím algoritmov z oblasti umelej inteligencie. Na problém skladania DNA sekvencií sa podarilo viacerým autorom úspešne aplikovať genetické algoritmy i simulované žihanie, no sľubným sa ukazuje byť použitie biologicky inšpirovaných algoritmov. Z nich boli na problém skladania DNA sekvencií použité viaceré, ako napríklad optimalizácia kolóniou mravcov (ACO), optimalizácia umelým spoločenstvom včiel (ABC), či genetický algoritmus inšpirovaný včelou kráľovnou (QEGA). V našej práci sa preto zameriavame na aplikáciu *Paralelného optimalizačného modelu inšpirovanom imunitným systémom*, ktorého pôvodný algoritmus PIMCSA sme vhodne upravili a tak sme vytvorili algoritmus *PIMCSA-DMO*. Tento algoritmus patrí do triedy biologicky inšpirovaných algoritmov a je možné ho aplikovať na problém obchodného cestujúceho, na ktorý možno problém skladania DNA sekvencií pretypovať.

Naša diplomová práca pozostáva z niekoľkých kapitol. V *analýze* sa venujeme opisu DNA, opisu sekvenovania, problému skladania DNA sekvencií, tradičným metódam skladania DNA sekvencií, ako aj umelou inteligenciou inšpirovaným metódam skladania DNA sekvencií. V *návrhu* sme navrhli princíp činnosti nášho skladača DNA sekvencií, ktorý je založený na nami navrhnutom algoritme pre riešenie TSP - PIMCSA-DMO. Tento algoritmus využíva výhody Iterovanej Lin-Kernighanovej

heuristiky a Vylepšeného Inver-over operátora. Práve Lin-Kernighanova heuristika sa považuje za jednu z najlepších heuristik na poli hľadania riešení v TSP. Skladač sme implementovali v jazyku C++ s použitím knižnice MPI, nakoľko PIMCSA-DMO predstavuje paralelný algoritmus. V kapitole *implementácie* sa preto venujeme opisu implementačných detailov a frameworku pre skladanie DNA sekvencií, ktorý vznikol z nášho skladača pre potreby jeho testovania. Na záver sme skladač vhodne otestovali, pričom sme sa zamerali hlavne na testovanie sily algoritmu PIMCSA-DMO. Testovanie PIMCSA-DMO z hľadiska schopnosti riešiť veľké inštancie TSP sme vykonali na TSPLIB a GenFrag inštanciách. Samotný skladač sme testovali s použitím nami vygenerovaných krátkych DNA sekvencií.

1.1 Použité pojmy a skratky

afinita - miera úspešnosti protilátky vyjadrená kladným číslom, pričom vyššia hodnota znamená lepšiu protilátku

kontig - reťazec, ktorý vznikol poskladaním DNA sekvencií

kritické pokrytie - počet nukleotidov, ktoré musia mať dva kontigy spoločné na to, aby ich modul *Compactor* spojil vo finálnej fáze skladania do jedného superkontigu

kritický pomer - pomer $R \in (0, 1)$ prekrytia dvoch sekvencií ku dĺžke kratšej z nich, ktorý musí byť dosiahnutý na to, aby ich modul *Compactor* spojil do predčasného kontigu

nukleotid - základná stavebná jednotka DNA, pričom rozoznávame 4 nukleotidy: A, T, G a C

obsah GC báz - podiel počtu nukleotidov G a C ku počtu nukleotidov A a T v danom DNA reťazci

pokrytie genómu - podiel súčtu dĺžok všetkých krátkych DNA sekvencií, ktoré vznikli sekvenovaním ku dĺžke genómu, z ktorého boli sekvenované

predčasný kontig - spojenie dvoch alebo viacerých krátkych sekvencií pomocou modulu *Compactor* s cieľom znížiť počet miest v TSP

protilátka - člen populácie algoritmu v kontexte algoritmov inšpirovaných imunitným systémom

superkontig - spojenie dvoch alebo viacerých kontigov

vakcína - informácia použitá pri mutácii protilátky extrahovaná z inej protilátky

FAP - problém skladania DNA sekvencií

MIA - imunitný algoritmus nad pamäťovou populáciou, časť PIMCSA a PIMCSA-DMO

PIA - imunitný algoritmus nad podpopuláciami, časť PIMCSA a PIMCSA-DMO

PIMCSA - paralelný imunitný algoritmus s pamäťou, klonovaním a výberom

PIMCSA-DMO - nami navrhnutý paralelný imunitný algoritmus s pamäťou, klonovaním, výberom a distribuovanou operáciou zrenia protilátok

SCS - problém hľadania najkratšieho spoločného nadreťazca

TMSM - Tower Master-Slave Model, dvojúrovňový paralelný imunitný model s jednou pamäťovou populáciou protilátok a N podpopuláciami protilátok

TSP - problém obchodného cestujúceho

2 Analýza

V tejto kapitole sme sa zamerali na opis DNA a jej štruktúry. Pochopenie štruktúry DNA je nevyhnutné pre pochopenie toho, ako je DNA sekvenovaná a pre pochopenie procesu de novo skladania DNA sekvencií. Pri opise DNA sme vychádzali najmä z kníh *Úvod do molekulárnej biológie* od Stanislava Rosypala [40] [41] [42] a knihy *Molekulová biológia* od Evy Mišúrovej a Petra Solára [31]. Ďalej nasleduje opis sekvenovania DNA, ktorého výstupom je veľké množstvo krátkych DNA sekvencií. Z dôvodu limitácie chemických reakcií používaných pri sekvenovaní DNA, je dĺžka jednej sekvencie oproti dĺžke celého genómu takmer zanedbateľná. Zamerali sme sa hlavne na opis najčastejšie používaných metód sekvenovania novej generácie (angl. next-generation sequencing). V tejto časti opisu sme vychádzali najmä z dizertačnej práce Benjamína Jacksona [22]. Porozumenie dátam, ktoré sú výstupom sekvencerov je dôležité pre vývoj skladačov DNA sekvencií, ktorých úlohou je zo získaných krátkych sekvencií poskladať pôvodne sekvenované molekuly DNA. V ďalšom texte sme sa preto zamerali na opis problému skladania DNA sekvencií, na tradičné prístupy k riešeniu tohto problému a na aplikácie algoritmov z umelej inteligencie v skladačoch DNA sekvencií.

2.1 DNA

Každá živá bunka sa skladá z niekoľkých organel. *Organela* je samostatne funkčná jednotka bunky zabezpečujúca nejakú z jej základných funkcionalít. Jednou z týchto organel je aj *bunkové jadro*, v ktorom sa nachádza DNA (*deoxyribonukleová kyselina*) všetkých živých eukaryotických organizmov. DNA teda patrí do triedy nukleových kyselín¹. DNA je *polymér*, čo je pojem označujúci chemickú zlúčeninu skladajúcu sa z viacerých jednoduchších jednotiek – *monomérov*. Ďalšími organelami, ktoré obsahujú DNA sú *mitochondrie*, ktoré sú okrem iného zodpovedné aj za výrobu chemickej energie pre chemické procesy, ktoré sa odohrávajú v prostredí bunky [1]. DNA je nositeľkou genetickej informácie a je priamo zodpovedná za kódovanie štruktúry bunky. Okrem toho, že určuje stavebnú štruktúru bunky, predstavuje aj plán pre riadenie jej rastu, delenia a regenerácie.

V prípade DNA sú monoméry predstavované nukleotidmi. Každý *nukleotid* sa skladá z *cukru*, *kyseliny fosforečnej* a organickej *dusíkatej bázy*. Vo vnútri DNA sa nachádza päťuhlíkatý cukor – *deoxyribóza*. Stavbu DNA podrobnejšie zachytáva obrázok 1.

Celkovo rozpoznávame 5 druhov dusíkatých báz: *adenín (A)*, *tymín (T)*, *guanín (G)*, *cytozín (C)* a *uracil (U)*. V DNA sa nachádzajú 4 z nich a to adenín, tymín, guanín a cytozín, nukleotid uracil sa nachádza

¹Medzi nukleové kyseliny patrí taktiež RNA, ktorá je na rozdiel od DNA len jednovláknová, nukleotid tymín má nahradený uracilom a v úlohe mediátorovej RNA (mRNA) má význam pri prenose informácie z DNA ku ribozómu, ktoré podľa nej konštruujú bielkoviny. Niektoré sekvenovacie techniky, ktorých cieľom je sekvenovať kódujúcu DNA sekvenujú práve mRNA. Keďže sa v našej práci sústreďujeme na skladanie celého genómu, zameriavame sa len na DNA.

len v RNA, kde nahrádza tymín. Adenín s guanínom patria medzi *purínové bázy*² a tymín s uracilom medzi *pyrimidínové bázy*. V praxi sa ale často stretávame so zjednodušením, kedy celé nukleotidy pomenúvame podľa názvu ich dusíkatých báz. Toto zjednodušenie má význam, nakoľko sa rôzne nukleotidy odlišujú len v ich dusíkatých bázach, čo nám umožňuje zapisovať poradie nukleotidov v DNA alfabeticky, napr. ATGCGTCA.

Organizácia DNA nie je jednoduchá. DNA je organizovaná do pomerne zložitých štruktúr. Rozoznávame *primárnu, sekundárnu a terciárnu štruktúru* DNA.

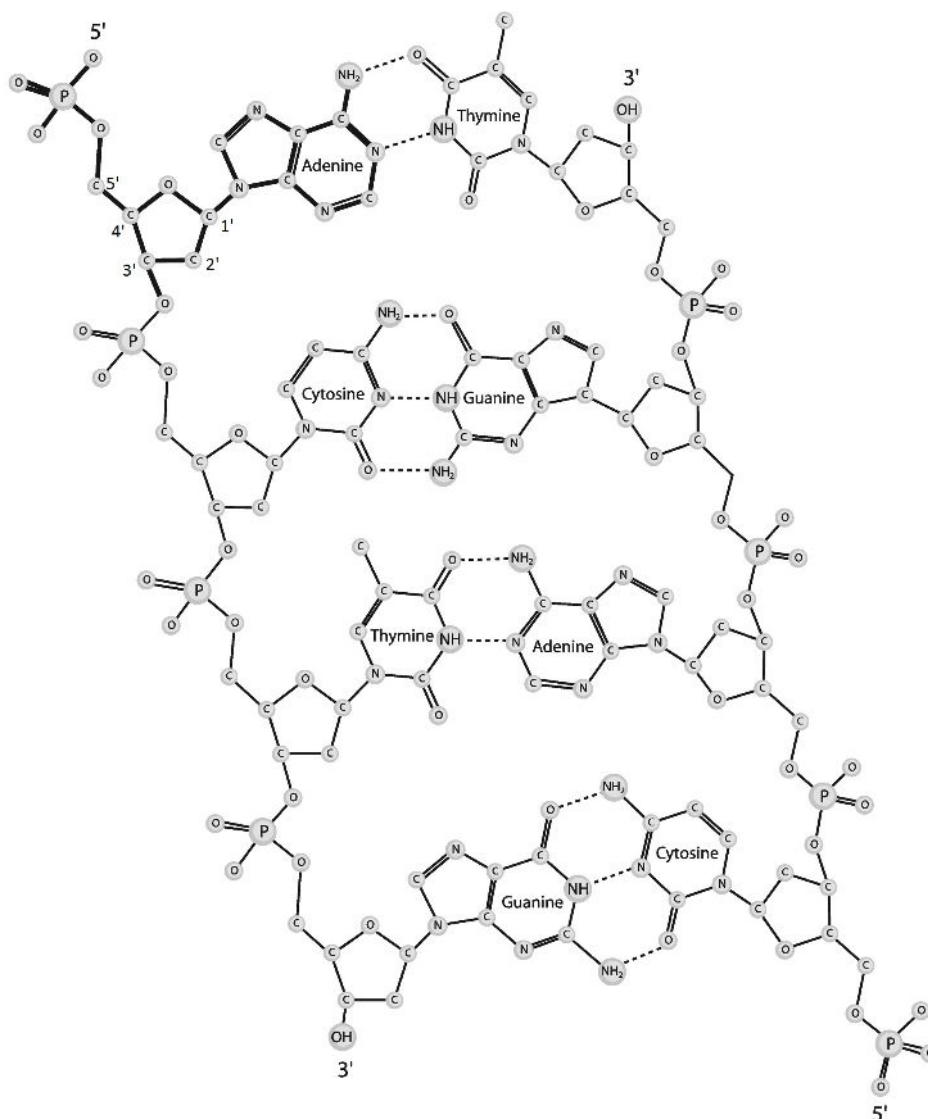
2.1.1 Primárna štruktúra DNA

Jednotlivým uhlíkom deoxyribózy sú pridelené čísla od 1' do 5', tak ako je možné vidieť na obrázku 1. Číslovanie poradia uhlíkov je v organickej chémii časté a je dôležité pre určenie názvu organickej zlúčeniny. Nukleotidy sa môžu ľubovoľne spájať, pričom spojenie vzniká medzi koncom jedného nukleotidu (3') a začiatkom druhého nukleotidu (5') prostredníctvom *fosfodiesterovej väzby*. Primárnu štruktúru predstavuje práve poradie nukleotidov, ktoré tvorí DNA reťazec. Z bioinformatického hľadiska si tak môžeme DNA reťazec predstaviť ako slovo vytvorené nad abecedou $\Sigma = \{A, T, G, C\}$ (napr. CGTGACGTGGATACAGCAG). Práve poradie nukleotidov udáva štruktúru bunky a kóduje aj informácie pre ostatné procesy, ktoré v bunke prebiehajú. Samotnú genetickú informáciu teda udáva primárna štruktúra DNA a preto je v prípade analýz DNA sekvencií jedinou skúmanou štruktúrou spomedzi ostatných štruktúr. Kvôli spôsobu, akým sa pripájajú nukleotidy, sme schopní rozlišovať smer DNA reťazca. *Dopredný smer* DNA reťazca predstavuje čítanie nukleotidov v smere od 5' do 3'.

2.1.2 Sekundárna štruktúra DNA

Dôležitou vlastnosťou reťazcov DNA je ich schopnosť párovať sa. Podľa *Watsonovho-Crickovho pravidla* sa páruje purínová báza s pyrimidínovou nasledovným spôsobom: adenín sa páruje s tymínom dvoma vodíkovými väzbami ($A = T$) a guanín s cytozínom tromi vodíkovými väzbami ($G \equiv C$). V literatúre sa toto pravidlo často označuje aj ako *princíp komplementarity*. Sekundárna štruktúra DNA je predstavovaná dvojzávitnicovým vinutím DNA molekuly. Dva reťazce DNA môžu vytvoriť DNA molekulu tak, že sa navzájom spoja ako na obrázku 1, pričom musia byť navzájom *antiparalelné* (mať opačný smer) a *komplementárne*. Komplementárny reťazec k reťazcu s (často označovaný aj pojmom *reverzný komplement*) značíme s' a zostrojíme ho tak, že otočíme poradie nukleotidov reťazca s a zameníme ich za k nim komplementárne nukleotidy podľa Watsonovho-Crickovho pravidla [22]. DNA molekula m je potom tvorená dvojicou $m = \{s, s'\}$. U DNA sa

²Pojem báza sa používa ako synonymum pre pojem nukleotid.



Obr. 1: Na obrázku je zachytená štruktúra DNA. Každý nukleotid sa skladá zo zvyšku kyseliny fosforečnej, deoxyribózy a dusikatej bázy, jeden z nukleotidov je na obrázku zvýraznený. Jednotlivé nukleotidy sú pospájané fosfodiesterovými väzbami. Dve komplementárne vlákna sú spárované pomocou vodíkových väzieb (prerušované čiary). Každý DNA reťazec má smer od 5' do 3', pričom reťazce DNA molekuly sú antiparalelné. Obrázok taktiež znázorňuje číslovanie uhlíkov deoxyribózy, odkiaľ pochádza označenie dopredného smeru DNA (od 5' do 3'). Prevzaté z [22].

vyskytujú 3 formy sekundárnej štruktúry, A-DNA, B-DNA a Z-DNA, z ktorých je najznámejšia B-DNA, všeobecne známa dvojzávitnica, objavená J. Watsonom a F. Crickom v roku 1953 [54].

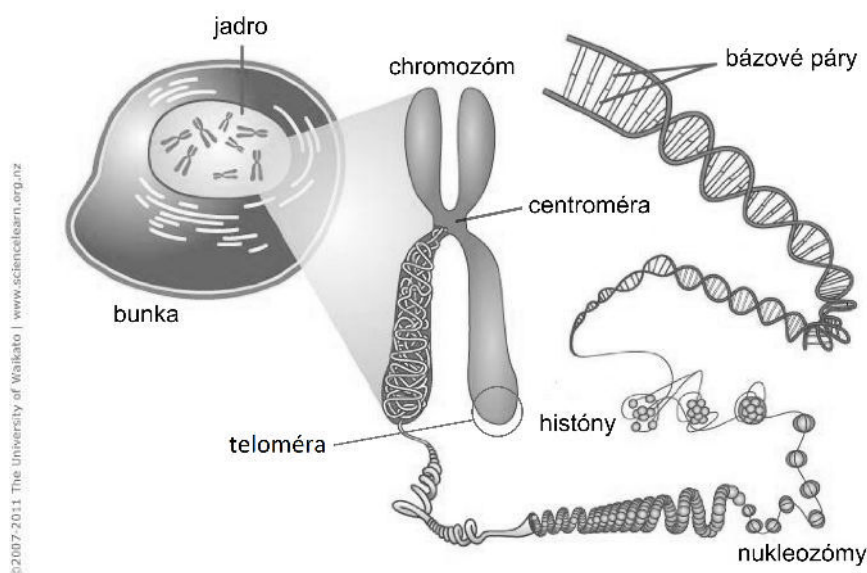
2.1.3 Terciárna štruktúra DNA

Terciárna štruktúra DNA predstavuje priestorové usporiadanie molekuly DNA, ktorá môže mať tvar *superhelixu*. Dvojzávitnica sa pomocou rôznych mechanizmov stáča do zložitejších tvarov, čo má za

následok zmenšenie plochy, ktorú zaberá. Toto stočenie je dôležité pre to, aby sa dlhá DNA molekula vošla do malého jadra bunky.

2.1.4 Zabalenie DNA a štruktúra chromozómov

Genetická informácia väčšiny živočíchov je tvorená viacerými DNA molekulami. Takéto samostatné molekuly nazývame *chromozómy*. Každý chromozóm je osobitne zvinutý až do podoby terciárnej štruktúry. Bunky *prokaryotických živočíchov* (živočíchy bez bunkového jadra) majú väčšinou jeden kruhový chromozóm uložený priamo v cytoplazme bunky. Bunky *eukaryotických živočíchov* (živočíchy, ktorých bunky majú aj bunkové jadro) obsahujú niekoľko *lineárnych chromozómov* uložených v bunkovom jadre. Chromozómy sú do terciárnej štruktúry zabalené tak, že dvojzávitnica (sekundárna štruktúra) sa obtáča okolo bielkovín zvaných *históny* s ktorými tvorí *nukleozómy*. Tieto nukleozómy sú potom ďalej stočené a tvoria samotné chromozómy (terciárna štruktúra). Lineárne chromozómy majú väčšiu časť bunkového cyklu lineárnu podobu (majú tvar čiarky), to však neplatí vo fáze pred delením bunky, kedy sa chromozóm skopíruje v procese replikácie a chromozóm je tak tvorený dvoma identickými *chromatidovými vláknami*, najčastejšie spojenými v tvare písmena X. Miesto, kde sa chromatidové vlákna spájajú sa nazýva *centroméra*. Konce lineárnych chromozómov sa nazývajú *teloméry* a ich úlohou je chrániť chromozómy pred stratou genetickej informácie, nakoľko sa chromozómy počas replikácie nepatrne skracujú. Zabalenie molekuly DNA do chromozómu možno vidieť na obrázku 2.



Obr. 2: Spôsob zabalenia DNA do chromozómov. Chromozóm tvorený dvoma totožnými chromatidovými vláknami sa stáča okolo histónov a tvorí nukleozómy. Tie potom po ďalšom zvinutí tvoria chromozómy. Miesto, kde sa chromatidové vlákna krížia sa nazýva centroméra. Konce chromozómov sú teloméry. Prevzaté z [51].

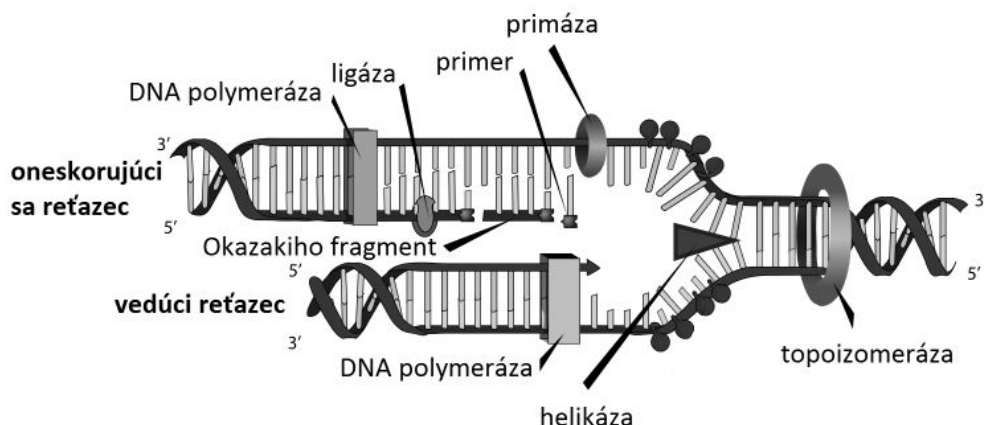
2.1.5 Replikácia DNA

Replikácia DNA je proces pri ktorom dochádza ku skopírovaniu chromozómov. Tento proces sa odohráva pred bunkovým delením. Z pohľadu sekvenovania DNA ide o pomerne dôležitý proces, nakoľko mnoho DNA sekvencerov využíva *enzýmy*, ktoré sa ho zúčastňujú.

Na začiatku replikácie sa postupne rozpletá terciárne i sekundárne vinutie DNA pričinením enzýmu *topoizomerázy* a súčasne enzým *helikáza* začne oddeľovať jednotlivé vlákna DNA molekuly. Úlohu zároveň preberá najdôležitejší enzým replikácie – *DNA polymeráza*. DNA polymeráza sa naviaže na každé z oboch vlákien a doplní ho na DNA molekulu podľa princípu komplementarity. Rozdelené vlákna sa stávajú tzv. *šablónovými reťazcami*. DNA polymeráza číta šablónový reťazec vždy len v smere od 3' do 5' a nad ním vystavuje jeho komplementárny doplnok vždy len v smere od 5' do 3'. Keďže DNA polymeráza nevie začať s výstavbou komplementárneho reťazca z ničoho, nad 3' koncom šablónového reťazca už musí byť časť jeho reverzného komplementu vytvorená, DNA polymeráza od tejto časti pokračuje ďalej a vytvára tak kompletnú molekulu.

Po rozstrihnutí DNA molekuly helikázou, má jeden šablónový reťazec 5' koniec pri aktuálnej pozícii helikázy. Tento reťazec sa nazýva *vedúci reťazec* (angl. leading strand) a DNA polymeráza ho môže priamo doplniť na DNA molekulu pomocou jeho reverzného komplementu. Stačí aby sa DNA polymeráza naviazala na *primer* (viac nižšie) vložený *primázou* na mieste, od ktorého začala helikáza strihať pôvodnú DNA molekulu (3' koniec vedúceho reťazca) a odtiaľ pokračovala s výstavbou komplementárneho reťazca v smere pohybu helikázy. Problém nastáva pri druhom šablónovom reťazci, ktorý má svoj 3' koniec na mieste kde sa aktuálne nachádza helikáza. Tento reťazec sa volá *oneskorujúci sa reťazec* (angl. lagging strand). DNA polymeráza v tomto prípade musí vystavovať komplementárny reťazec v smere od helikázy k počiatku rozdelenia pôvodnej DNA molekuly, enzým helikáza sa ale naďalej pohybuje opačným smerom, preto polymeráza vystavuje komplementárny reťazec po úsekoch. Začne neďaleko aktuálneho umiestnenia helikázy a pokračuje až po začiatok rozstrihnutia pôvodnej molekuly, keď sa dostane na tento začiatok, znova začína neďaleko aktuálnej pozície helikázy a pokračuje pozdĺž šablónového reťazca až kým nenarazí na úsek, z ktorého začínala v predošlom kroku. Výsledkom je množstvo úsekov, ktoré ale nie sú spojené fosfodiesterovou väzbou. Tieto nespojené oblasti sa nazývajú *Okazakiho fragmenty*. Keďže DNA polymeráza nevie začať s výstavbou komplementárneho reťazca bez toho, aby bola jeho časť už vystavaná, enzým *primáza* pripája neďaleko miesta výskytu helikázy k oneskorujúcemu sa šablónovému reťazcu v určitých rozostupoch *primer*, čo je sekvencia niekoľkých nukleotidov, ktorá je reverzne komplementárna vzhľadom na daný malý úsek šablónového reťazca. Keď je na šablónovom reťazci pripojený primer, DNA polymeráza môže od neho pokračovať v čítaní šablónového reťazca v smere od 3' do 5' a vystavovať tak nad ním jeho reverzný komplement až kým nenarazí na primer, ktorý bol na reťazec pripojený v predchádzajúcich krokoch replikácie zaostávajúceho reťazca. Enzým *ligáza* napokon pospája Okazakiho fragmenty fosfodiesterovou väzbou. Proces replikácie je znázornený na obrázku

3. Ligáza je ďalším z enzýmov, ktorý využíva špeciálne metódy sekvenovania.



Obr. 3: Obrázok znázorňuje činnosť jednotlivých enzýmov v procese replikácie DNA. Ich bližší opis je v časti 2.1.5. Prevzaté z [29].

2.1.6 Vnútoraná stavba DNA

Z hľadiska vnútornej stavby DNA rozoznávame dva typy úsekov DNA, tzv. *kódujúcu DNA* a *nekódujúcu DNA*.

2.1.7 Kódujúca DNA

Kódujúcu DNA tvoria nukleotidy, ktoré môžu byť prepísané do *bielkovín*. Tieto nukleotidy určujú štruktúru jednotlivých bielkovín, ktoré sa podieľajú na stavbe bunky. Kódujúca DNA sa do bielkovín prepisuje v procesoch známych ako *translácia* a *transkripcia*. Pretože tieto procesy nie sú z hľadiska skladania DNA sekvencií zaujímavé, bližšie ich neopisujeme. Ich bližší opis sme už uviedli v predchádzajúcej práci [47]. Kódujúca DNA, spolu s niektorými prvkami DNA spadajúcimi pod nekódujúcu DNA, tvorí *gény*. Práve gény sú úsekmi, ktoré medikov a biológov zaujímajú najviac, preto je potrebné ich osekvenovať a poskladať čo najpresnejšie.

2.1.8 Nekódujúca DNA

Nekódujúcu DNA tvoria úseky DNA, ktoré sa do bielkovín neprepisujú. Z hľadiska skladania DNA sekvencií je z nich najzaujímavejšia *repetitívna DNA*. Repetitívna DNA značne komplikuje a znep-

resňuje proces skladania DNA sekvencií. Táto DNA predstavuje úseky, ktoré sa v rámci genómu niekoľkokrát opakujú, pričom opakovania sa môžu vyskytovať v rámci rovnakého reťazca jedného chromozómu (paralelné opakovanie), v rámci oboch reťazcov rovnakého chromozómu (antiparalelné opakovanie), či medzi rôznymi chromozómami (rozptýlené opakovanie). Opakovania sa delia na *tandemové* a *rozptýlené* [31].

Autori knihy [52] zase rozoznávajú *priame opakovania* (angl. direct repeats), ktoré vlastne predstavujú tandemové opakovania. Medzi priame opakovania patria aj opakovania, kedy je opakovacia jednotka oddelená na tom istom DNA reťazci nejakým ľubovoľne dlhým slovom. Opakovania sa môžu taktiež vyskytovať ako *palindrómy* (angl. palindromic repeats), kde je reťazec s nasledovaný reťazcom s' na rovnakom vlákne DNA, pričom sa medzi nimi môže vyskytovať ľubovoľný úsek nukleotidov [17] [52]. Palindromické opakovania všeobecnejšie radíme medzi rozptýlené opakovania. Nakoniec sa v [52] rozoznávajú *zrkadlové opakovania*, kedy je opakovanie tvorené slovami ww^i , pričom w je slovo nad $\Sigma = \{A, T, G, C\}$ a $w^i = inv(w)$, kde $inv(w)$ predstavuje otočenie reťazca w (jeho čítanie v spätnom smere - od 3' do 5'). Štandardne sa tieto opakovania taktiež chápu ako rozptýlené opakovania.

Tandemové opakovania

Sú to úseky DNA, ktoré sa v DNA reťazci opakujú viac krát za sebou. Možno ich chápať ako slová w^n , kde $w = a_1a_2...a_k$ a $k, n \in \mathbb{N}^+$, pričom $a_i \in \Sigma = \{A, T, G, C\}$. Opakovanie však môže byť miestami narušené mutáciou, či nejakým vloženým úsekom DNA, na určitom úseku tak môže dôjsť k narušeniu *opakovacej jednotky*³, no i tak ide stále o tandemové opakovanie. Tandemové opakovania sa delia podľa veľkosti na mikrosatelity, minisatelity a satelity.

- *Mikrosatelite*. Ich celková dĺžka nepresahuje 150 bp⁴ Dĺžka opakovacej jednotky je 1 – 6 bp. Ak sa opakuje jeden nukleotid, hovoríme, že ide o *homopolymér* [4]. Mikrosatelity nájdeme v rôznych častiach genómu vrátane kódujúcej DNA.
- *Minisatelite*. Ich celková dĺžka je 1 kbp – 20 kbp. Dĺžka opakovacej jednotky je 9 bp – 80 bp. Vyskytujú sa prevažne v nekódujúcich oblastiach DNA. Do tejto kategórie patria aj tandemové opakovania v oblasti telomér. V prípade ľudskej DNA majú teloméry opakovaciu jednotu $(TTAGGG)_n$.
- *Satelite*. Ich celková dĺžka je od 100 kbp až po niekoľko Mbp. Väčšina satelitov sa nachádza v oblasti centroméry.

³Opakovacia jednotka je slovo w nad abecedou $\Sigma = \{A, T, G, C\}$, ktoré sa v DNA opakuje.

⁴Skratka *bp* znamená bázoý pár, ide o dvojicu komplementárnych nukleotidov spojených vodíkovou väzbou vyskytujúcich sa v protilahlých reťazcoch DNA. Skratku používame na označenie počtu nukleotidov v prípade DNA molekúl. Pre označovanie počtu nukleotidov na jednom reťazci DNA používame skratku *b* (báza) a jej násobky kb, Mb, Gb.

Rozptýlené opakovania

Rozptýlené opakovania sú tie úseky DNA, ktoré sa vyskytujú v rôznych častiach genómu, či už v rámci jedného chromozómu, alebo v rámci rôznych chromozómov. Takisto ako v prípade tandemových opakovaní, ani rozptýlené opakovania nemusia byť absolútne totožné. Rozlišujeme tri základné druhy rozptýlených opakovaní: LINEs, SINEs a LTR retrotransponzóny.

- *LINEs* (angl. *Long Interspersed Nuclear Elements*): Ide o dlhé úseky DNA, ktoré majú schopnosť sa samé skopírovať do inej časti genómu, čím vzniká rozptýlené opakovanie. Dĺžka LINEs elementov je 6 kbp – 7 kbp.
- *SINEs* (angl. *Short Interspersed Nuclear Elements*): Sú to kratšie úseky DNA, ktoré, na rozdiel od *LINEs*, nemajú schopnosť sa samé kopírovať v rámci daného genómu. Na kopírovanie používajú mechanizmy, ktoré sú syntetizované pomocou LINEs. Ich dĺžka je menej ako 500 bp, zvyčajne je v rozmedzí 100 bp až 400 bp.
- *LTR* (angl. *Long Terminal Repeats*) retrotransponzóny: Predstavujú neaktívnu DNA retrovirusov, ktoré sa do DNA začlenili v minulosti. LTR retrotransponzóny sú dlhé 7 kbp – 9 kbp a sú tvorené dvoma opakovanými LTR sekvenciami, medzi ktorými sa nachádzajú neaktívne, gény kódujúce bielkoviny. LTR sekvencie môžu byť orientované paralelne alebo antiparalelne [18].

2.2 Sekvenovanie DNA

Výskum DNA vyžaduje, aby bola DNA organizmu najprv prečítaná. To nám umožňuje proces zvaný *sekvenovanie DNA*. Keďže mnohé organizmy majú rozsiahlu DNA (napr. haploidný⁵ ľudský genóm obsahuje až 3.38 Gbp [11]), sekvenovanie DNA sa najčastejšie vykonáva pomocou DNA sekvencerov. *DNA sekvencer* je stroj, ktorý dokáže opakovane prečítať malé kúsky jedného vlákna DNA molekuly pozdĺž jeho dopredného smeru. DNA molekulu je možné získať z bunky za pomoci chemických reakcií, ktoré bunku rozložia tak, že ostane len samotná molekula DNA [33]. Na počiatku v oblasti sekvenovania dominovalo *Sangerovo sekvenovanie*, ktoré je schopné produkovať DNA sekvencie dĺžky 700 – 1000b [44]. Dĺžka procesu Sangerovho sekvenovania a vysoké náklady s ním spojené viedli neskôr k vývoju *sekvencerov novej generácie* (angl. *next-generation sequencers*). Proces sekvenovania na sekvenceroch novej generácie je podstatne kratší a rádovo lacnejší [2] [25]. Nevýhodou sekvencerov novej generácie je ale to, že dokážu prečítať len sekvencie dĺžky 35 – 400 b, čo značne komplikuje proces skladania DNA sekvencií [22]. V posledných rokoch sa pracuje na vývoji

⁵Haploidná sada chromozómov u pohlavných živočíchov predstavuje sadu, kde je každý typ chromozómu zastúpený len raz (buď je od otca, alebo od matky), takúto sadu predstavujú napríklad pohlavné bunky. Okrem haploidnej sady sa stretávame i s diploidnými sadami, kde je každý typ chromozómu zastúpený dvoma chromozómami (jeden je od otca, druhý od matky).

sekvencerov *tretej generácie* (angl. third generation sequencers), ktoré dokážu lacno produkovať dlhé sekvencie, ale s relatívne vysokou chybovosťou výstupných dát prevyšujúcou 5% [46].

Jeden z najdôležitejších parametrov v sekvenovaní, ale i skladaní DNA sekvencií je pojem *pokrytia genómu* (angl. coverage). Pokrytie genómu je podiel súčtu dĺžok všetkých DNA sekvencií, ktoré boli získané počas sekvenovania a celkovej dĺžky genómu. Matematicky možno pokrytie vyjadriť nasledovne. Nech S je množina všetkých krátkych sekvencií $s \in S$ a $|S| = n, n \in \mathbb{N}$, pričom $s_i \in S$ je podreťazcom genómu G pre $i = 1..n$ a funkcia $len : S \rightarrow \mathbb{N}$ vracia dĺžku vstupnej sekvencie, potom pokrytie genómu $cov(G)$ je definované ako:

$$cov(G) = \frac{\sum_{i=1}^n len(s_i)}{len(G)}$$

Všetky techniky používané na sekvenovanie DNA využívajú tieto tri koncepty [22]:

- *Replikácia*: Sekvenovaná DNA molekula sa najprv mnohonásobne zduplikuje pomocou chemickej reakcie zvanej *polymerázová reťazová reakcia* (PCR z angl. polymerase chain reaction), ktorá je bližšie opísaná v [20]. Na replikáciu sa využíva práve *DNA polymeráza*, ktorej činnosť bola opísaná v predchádzajúcej časti analýzy. Proces replikácie sa opätovne opakuje aj s nakopírovanými molekulami, takže počet molekúl, ktoré získame replikáciou exponenciálne rastie (počet získaných molekúl v závislosti od počtu opakovaní reakcie PCR (n), udáva exponenciálna funkcia 2^n). Počet opakovaní replikácie priamo vplýva na hodnotu pokrytia.
- *Náhodné strihanie*: Skopírované DNA molekuly sú rozstrihané na náhodných pozíciách na *DNA fragmenty*. Tieto DNA fragmenty sú ešte stále dvojzávitnicovej povahy. Rozstrihanie možno vykonať pomocou zvukových vĺn, enzymaticky, alebo inými prístupmi [32]. Pri dostatočnom počte fragmentov predpokladáme, že fragmentami bude pokrytý celý genóm.
- *Čítanie*: Všetky metódy sa spoliehajú na schopnosť prečítania jedného, či viacerých nukleotidov z DNA fragmentu.

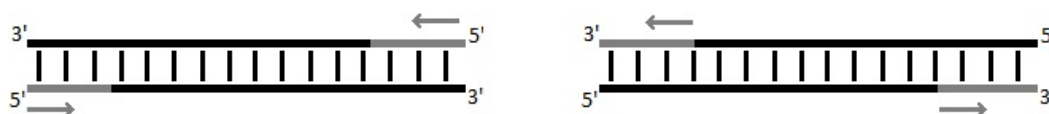
Sekvenovanie DNA však nie je bezchybný proces [22]. V procese sekvenovania môže dôjsť ku trom rôznym *chybám*. Do sekvencie môže byť omylom vložený nesprávny nukleotid, môže z nej byť omylom odstránený nukleotid, alebo nejaký nukleotid môže byť nahradený nesprávnym nukleotidom.

2.2.1 Typy DNA sekvencií

Rôzne DNA sekvencery dokážu produkovať rôzne druhy *DNA sekvencií* (angl. reads). Medzi najbežnejší typ DNA sekvencií patria *jednoduché sekvencie* (angl. single reads). Sekvencer sekvenuje

jeden z reťazcov DNA fragmentu. Tak po prečítaní jeho niekoľkých nukleotidov vznikne jednoduchá sekvencia.

Ďalším typom DNA sekvencií sú *párové sekvencie* (angl. pair-end reads). Sekvencer prečíta z jedného reťazca prislúchajúcemu určitému DNA fragmentu sekvenciu niekoľkých nukleotidov rovnako, ako v prípade jednoduchých sekvencií, následne však prečíta sekvenciu aj z druhého reťazca fragmentu podobným spôsobom. Tieto dve sekvencie tvoria párovú sekvenciu. Priestor medzi nimi sa nazýva *vloženie* (angl. insertion) a jeho dĺžka závisí od veľkosti sekvenovaného DNA fragmentu, pričom sa pre každú získanú párovú sekvenciu môže mierne líšiť. Keďže DNA možno čítať len v smere od 5' do 3' a reťazce fragmentu sú antiparalelné, aj zložky párovej sekvencie sú antiparalelné, môžu ale smerovať k sebe alebo od seba, ako je zobrazené na obrázku 4.



Obr. 4: Obrázok znázorňuje orientáciu jednotlivých častí párovej sekvencie. Jednotlivé zložky (označené sivou), vždy lokalizované na dvoch antiparalelných reťazcoch, môžu smerovať buď k sebe, alebo od seba. Priestor medzi týmito zložkami sa nazýva vloženie.

Spoločnosť Illumina rozoznáva ešte tretí typ sekvencií – tzv. *spriatelené sekvencie* (angl. mate-pair reads) [10]. V skutočnosti síce ide o párové sekvencie v zmysle vyššieho opisu, ale Illumina robí rozdiel medzi párovou sekvenciou, kedy zložky sekvencie smerujú k sebe a vloženie nie je dlhšie ako niekoľko stoviek nukleotidov a medzi spriatelenou sekvenciou, ktorej jednotlivé zložky smerujú od seba a vloženie má rozsah niekoľko tisíc nukleotidov.

2.2.2 Sangerovo sekvenovanie

Pred použitím samotného *Sangerovho sekvenovania* [44] sa sekvenovaný fragment najprv mnohonásobne skopíruje za použitia PCR. Tieto kópie sa *denaturujú*⁶, dôsledkom čoho získame oddelené reťazce DNA. Denaturovanú DNA rozdelíme do štyroch rôznych reakčných nádob, z ktorých každá obsahuje nasledovné chemické látky:

- *DNA primer*: Vytvára počiatočný bod pre kopírovanie reťazca pomocou DNA polymerázy (viac informácií v časti 2.1.5). Primer je umelo syntetizovaný aby sa zhodoval so začiatkom sekvenovaného fragmentu, ktorý je nám nejakým spôsobom známy.
- *DNA polymeráza*: vytvára komplementárny reťazec zo šablónového reťazca v zmysle opisu v 2.1.5.

⁶Dôjde k oddeleniu jednotlivých reťazcov DNA molekuly.

- *Deoxynukleotidy*: Deoxynukleotidy dATP, dTTP, dGTP, dCTP predstavujú nukleotidy adenín, tymín, guanín a cytozín.
- *Dideoxynukleotidy*: Jedná sa o chemicky upravené deoxynukleotidy tak, aby po ich naviazaní prostredníctvom DNA polymerázy došlo k prerušeniu vytvárania reverzného komplementu. Rozoznávame štyri dideoxynukleotidy a to ddATP, ddTTP, ddGTP a ddCTP, pričom v každej reakčnej nádobe je prítomný len jeden z nich.

Pre rozlíšenie jednotlivých nukleotidov v reťazci DNA je dôležité to, že primer alebo dideoxynukleotidy sú označené *rádioaktívnym markerom* alebo *farbivom*. DNA polymeráza počas reakcie pripája nukleotidy na šablónový reťazec, až kým nedosiahne jeho koniec, alebo sa nenaviaže jeden z dideoxynukleotidov. Možné pripojenia dideoxynukleotidu znázorňuje obrázok 5.

```

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

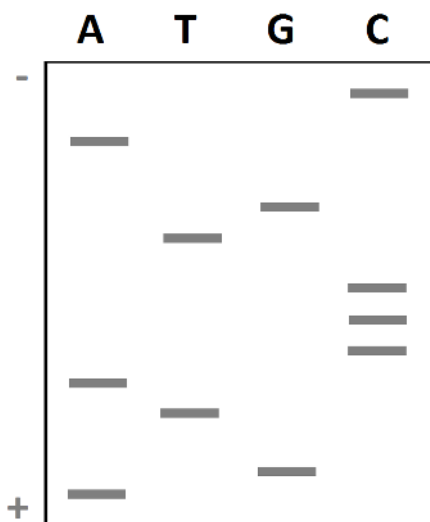
A
A G
A G T
A G T A
A G T A C
A G T A C C
A G T A C C C
A G T A C C C T
A G T A C C C T G
A G T A C C C T G A
A G T A C C C T G A C

```

Obr. 5: Vrchný riadok predstavuje sekvenovaný reťazec DNA, pod ním sa pre zjednodušenie nasledujúceho opisu nachádza jeho reverzný komplement, keďže DNA polymeráza “kopíruje” reťazec tak, že vystavuje jeho reverzný komplement. Menšie reťazce predstavujú reťazce, ktoré vznikli ako výsledok činnosti DNA polymerázy. Kurzívou a sivou farbou sú označené dideoxynukleotidy, ktoré, keď sa pripojili k reťazcu, spôsobili ukončenie syntézy DNA. Posledný riadok predstavuje situáciu, kedy nebol pripojený žiadny dideoxynukleotid. Pri dostatočnom počte kópií bude každá z možností zastúpená v dostatočnom počte.

Obsah každej z reakčných nádob sa neskôr vloží do osobitnej dráhy v géle v procese zvanom *gélová elektroforéza* [6]. Záporne nabitá molekula DNA putuje ku kladne nabitej elektróde, pričom kratšie úseky prejdú v géle dlhšiu cestu ako dlhšie úseky. V prípade označenia primeru alebo dideoxynukleotidu rádioaktívnym markerom ponecháme gél pôsobiť na film, do ktorého rádioaktivita vypáli značky. Na základe týchto značiek vieme prečítať výslednú DNA sekvenciu, ako to znázorňuje obrázok 6. Pri dostatočnom počte naklonovaných fragmentov je vysoko pravdepodobné, že sa dideoxynukleotid naviaže na každú pozíciu v sekvenovanom fragmente, čím vznikne množstvo rôzne dlhých reverzných komplementov, ktorých usporiadanie od najkratšieho po najdlhší umožní

zistiť výsledné poradie nukleotidov. Sekvencery využívajúce Sangerovo sekvenovanie nahrádzajú rádioaktívne markery niekoľkými kapilármi napustenými gélom. Dideoxynukleotidy sú zafarbené a nie je preto potrebné mať pre každý dideoxynukleotid osobitnú kapiláru. Sekvenovanie prebieha v kapilárach paralelne, výsledok číta laser a vyhodnocuje počítač.



Obr. 6: Obrázok ukazuje, ako môže vyzeráť film, ktorý vznikol ako výsledok Sangerovho sekvenovania. Rádioaktívne markery, ktorými boli označené dideoxynukleotidy vypálili na film tmavé miesta. Kratšie úseky (viac obrázok 5) prešli v géle dlhšiu cestu. Výslednú sekvenciu získame postupným čítaním gélu od kladného konca k zápornému, v tomto prípade ide o sekvenciu AGTACCCTGAC. Keďže sme sekvenovaný reťazec klonovali, výsledná sekvencia predstavuje jeho reverzný komplement, sekvenovaný reťazec bol teda reťazec GTCAGGGTACT.

2.2.3 Pyrosekvenovanie

Pyrosekvenovanie (angl. pyrosequencing) [12] je jednou zo sekvenačných metód novej generácie. Pyrosekvenovanie sa nespolieha na meranie dĺžok jednotlivých fragmentov (ako robí Sangerovo sekvenovanie), ale na schopnosť priamo merať priebeh chemickej reakcie. Prostredníctvom pyrosekvenovania teda vieme rozlíšiť, ktorý nukleotid bol v danom čase pripojený na šablónový reťazec v procese vytvárania komplementárneho reťazca. Do roztoku so sekvenovaným reťazcom DNA vložíme DNA polymerázu a enzým, ktorý vyžiari svetlo, keď polymeráza pripojí nejaký nukleotid. Následne sa do roztoku cyklicky, stále v tom istom poradí, pridávajú jednotlivé nukleotidy (v danom cykle len jeden typ), po ich pripojení pomocou polymerázy sa meria množstvo vyžiareného svetla a na konci každého cyklu sa nepoužité nukleotidy z roztoku odstránia pomocou enzýmu, ktorý sa nazýva *apyráza*. Ak DNA polymeráza pripojí na šablónový reťazec viacero nukleotidov daného typu, odrazí sa to na množstve vyžiareného svetla, homopolyméry sú tak osekvenované v jednom kroku.

Na *DNA čipe*⁷ je možné nechať súčasne prebiehať niekoľko takýchto reakcií. Takéto sparalizovanie sekvenovania výrazne znižuje náklady spojené so sekvenovaním. Pri použití DNA čipu sme za jednu hodinu schopní prečítať 40 až 60 miliónov nukleotidov [22].

Sekvencer *GS Junior System* spoločnosti 454 Life Sciences dokáže čítať jednoduché sekvencie dĺžky až 400 nukleotidov s presnosťou približne 99% [39]. Sekvencer *GS FLX* od rovnakej spoločnosti je taktiež schopný čítať párové sekvencie izolovaním fragmentov približnej dĺžky (napr. 3 kbp). K takýmto fragmentom sa pripojí *cirkularizačný adaptér*, ktorý z nich vytvorí kruhovú molekulu. Kruhovú molekulu potom rozrežeme na fragmenty dĺžky približne 400 bp a z nich vyberieme len tie, ktoré vo svojom strede obsahujú cirkularizačný adaptér. Z týchto fragmentov prečítame oba konce, čím získame párovú sekvenciu. *GS FLX* je schopný čítať párové sekvencie s dĺžkou vloženia 3kb, 8kb alebo 20kb [39].

2.2.4 Sekvenovanie s použitím reverzibilných terminátorov

Predstavuje ďalšiu metódu sekvenovania novej generácie. Sekvenovanie pomocou reverzibilných terminátorov (angl. reversible terminator sequencing) [5] využívajú spoločnosti Illumina a Helicos. Oproti Sangerovmu sekvenovaniu sú terminačné nukleotidy *reverzibilné*, je teda možné zrušiť ich blokovaciu činnosť a obnoviť proces syntézy reverzného komplementu.

Spoločnosť *Illumina* používa vo svojich sekvenceroch priesvitný DNA čip ku ktorému sa pripoja náhodne narezané jednovláknové DNA fragmenty. Na čipe sa tieto fragmenty naklonujú, čím vzniknú oblasti predstavujúce rovnaký fragment DNA [19]. Každý zo štyroch reverzibilných terminátorov je zafarbený rozdielnou farbou, pričom sa na čip vyleje roztok so všetkými štyrmi reverzibilnými terminátormi naraz. Vysokocitlivá kamera tak zachytí, ktorý z reverzibilných terminátorov bol v procese tvorby reverzného komplementu pripojený. Prebytočné nukleotidy sa zmyjú a proces pokračuje ďalším cyklom. Sekvencer spoločnosti Illumina dokáže produkovať aj párové sekvencie, pričom sa prečíta najskôr jedna strana molekúl, potom sa molekuly k čipu pripoja z druhého konca a prečítajú sa ich opačné konce. Tieto sekvencery produkujú párové sekvencie dĺžky 75 b s dĺžkou vloženia 200 b – 500 b, alebo spriatelené sekvencie s dĺžkou vloženia 2 – 3 kb. V jednom behu je sekvencer schopný osekvenovať až 20 Gb, pričom beh trvá zhruba týždeň. Sekvencer vie taktiež produkovať jednoduché sekvencie, taktiež dĺžky 75 b.

Spoločnosť *Helicos* nepoužíva metódu farbenia reverzibilných terminátorov, ale v danom čase je do vzorky pridaný len jeden typ reverzibilného terminátora. Sekvencer spoločnosti Helicos produkuje jednoduché sekvencie dĺžky 25 – 35 b.

⁷DNA čip je mriežka, ku ktorej možno pripevniť fragmenty DNA.

2.2.5 Sekvenovanie pomocou ligázy

Sekvencer SOLiD 2 od spoločnosti *Applied Biosystems*, na rozdiel od ostatných technológií sekvenovania novej generácie, využíva namiesto DNA polymerázy DNA ligázu [22]. Na začiatku procesu sa pripoja fragmenty DNA ku DNA čipu a pomocou PCR sa lokálne namnožia.

Sekvenuje sa jednovláknový reťazec DNA. Po jeho upevnení na DNA čip sa k nemu pripojí *primer* dĺžky n . Sekvenovanie pomocou ligázy sa spolieha na princíp postupného predlžovania sekvenovaného vlákna pripojením krátkych oligonukleotidov⁸ za použitia DNA ligázy [15]. Okrem DNA ligázy roztok obsahuje krátke úseky jednovláknovej DNA najčastejšie dĺžky 8 nukleotidov (tzv. *oktaméry*), pričom na ich prvých dvoch pozíciách 3' konca sa nachádzajú kombinácie rôznych nukleotidov. Spolu tak dostávame 16 kombinácií, kde sú dané oktaméry označené jedným zo 4 fluorescenčných farbív (v prípade sekvencera SOLiD) tak, ako to znázorňuje obrázok 7.

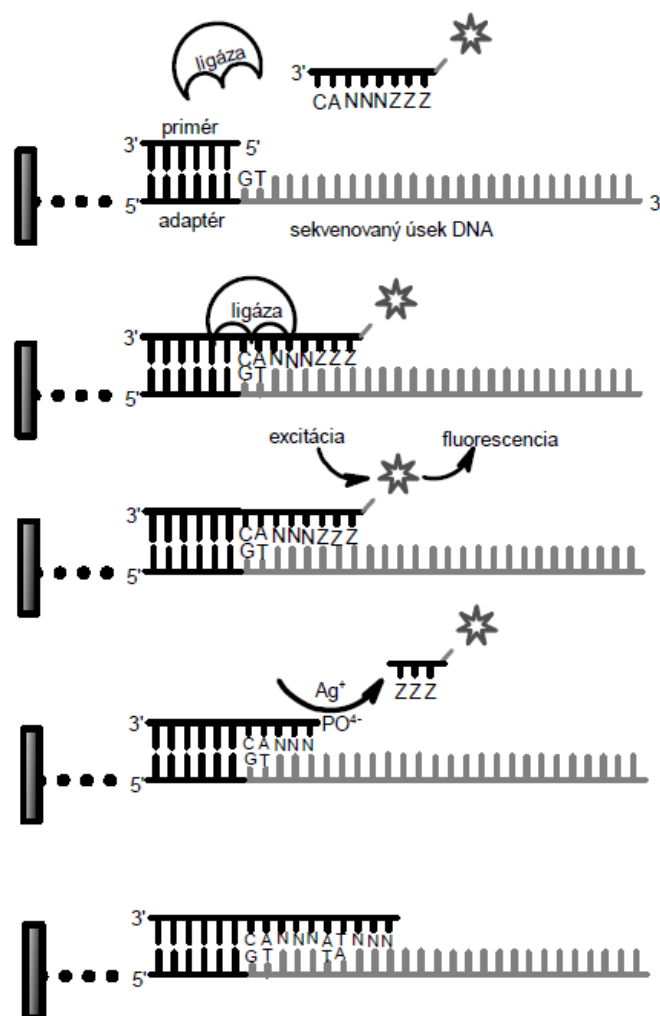
	A	C	G	T
A				
C				
G				
T				

Obr. 7: Na obrázku sú znázornené použitia farbív pre dve rôzne kombinácie nukleotidov v oktaméroch. Prevzaté z [22].

DNA ligáza vysoko pravdepodobnejšie pripojí k primeru oligonukleotid, ktorý má na 1. a 2. pozícii nukleotidy komplementárne s nukleotidmi sekvenovaného reťazca. Po fluorescencii sa posledné 3 nukleotidy oktaméru odstránia, čím sa primer predĺži o 5 nukleotidov a na 5' konci má voľný fosfátový zvyšok pre ďalšiu ligáciu. Tento proces sa opakuje niekoľko krát, pričom počet opakovaní ovplyvňuje dĺžku sekvencie. Na záver procesu sa molekula denaturuje, k šablónovému reťazcu sa pripojí primer o 1 nukleotid kratší a pokračuje sa rovnako. Takto pripojíme k jednej sekvencii (v prípade oktamérov) 5 primerov dĺžky n až $n - 4$ nukleotidov, čím získame rôzne ofsety a vďaka zvolenému farebnému kódovaniu a tomu, že každý nukleotid čítame dva krát, sme schopní previesť zachytenú postupnosť farebných svetiel na postupnosť nukleotidov. Proces sekvenovania pomocou ligázy znázorňuje obrázok 8.

System SOLiD využíva pri sekvenovaní oktaméry. Dĺžka sekvencie je 35 b. V *Applied Biosystems*

⁸Dva a viac nukleotidov spojených špecifickou chemickou väzbou. Jedná sa vlastne o krátky DNA reťazec.



Obr. 8: Na obrázku je zachytený proces sekvenovania pomocou ligázy. Sekvenovaný fragment je pripojený k DNA čipu, je naň naviazaný primer a ligáza postupne pripája k primeru oktaméry, z ktorých prvé dva nukleotidy sú komplementárne k nukleotidom na sekvenovanom reťazci. Pripojenie oktaméru vyžiari merateľné svetlo, potom sa z neho odstráni posledné 3 nukleotidy a pokračuje sa pripojením ďalšieho oktaméru. Prevzaté z [15].

zvolili tento prístup preto, lebo ligáza je menej náchylná na chyby a v tomto procese je každý nukleotid kvôli dlaždicovému sekvenovaniu každej bázy meraný dva krát, čo ešte viac zvyšuje presnosť.

2.2.6 Formáty pre ukladanie DNA sekvencií

Sekvencery ukladajú osekvenované krátke DNA sekvencie do rôznych výstupných formátov. Medzi týmito formátmi je možná konverzia, a aj z napohľad nezrozumiteľných formátov vieme získať človekom čitateľné dáta. Ďalej uvádzame zoznam niekoľkých najpoužívanějších formátov pre ukladanie DNA sekvencií [16].

- *DNA sekvencie ako čistý text* (angl. Plain Sequence Format): Tento formát môže obsahovať len IUPAC znaky [45]. Výstupom sekvencera pri sekvenovaní DNA sú však len znaky A, T, G a C. Okrem týchto znakov môže obsahovať aj medzery, no súbor môže obsahovať len jednu sekvenciu.

Príklad:

```
GTGGTTTGAGTGGACCTCCCAGGCCAGTGCCGGGCCCCTCATAGGAGAGGAAGCTCGGTATGGCTGCCTG
GAGGTGGCCAGGCGGCAGGAAGGCGCACCCCCCAGCAATCCGCGCGCCGGGACAGAATATCCCCAGAAT
```

- *EMBL formát*: Tento formát môže obsahovať viacero DNA sekvencií. Každá sekvencia začína identifikátorom 'ID', za ktorým nasledujú ďalšie anotácie. Začiatok každej sekvencie udáva značka 'SQ' a jej koniec je označený dvoma lomítkami '//'. Nukleotidy sú pre lepšiu viditeľnosť zoskupované po 10 znakoch, pričom na jednom riadku je najviac 60 nukleotidov. Kumulatívny počet nukleotidov sekvencie je na každom riadku označený číslom.

Príklad:

```
ID AB000263 standard; RNA; PRI; 368 BP.
XX
AC AB000263;
XX
DE Homo sapiens mRNA for prepro cortistatin like peptide.
XX
SQ Sequence 368 BP;
acaagatgcc attgtccccc ggcctcctgc tgctgctgct ctccggggcc acggccaccg 60
ctgccctgcc cctggagggt ggccccaccg gccgagacag cgagcatatg caggaagcgg 120
caggaataag gaaaagcagc ctcttgactt tcctcgcttg gtggtttgag tggacctccc 180
aggccagtgc cggggcccctc ataggagagg aagctcggga ggtggccagg cggcaggaag 240
gcgcaccccc ccagcaatcc gcgcgcccgg acagaatgcc ctgcaggaac ttcttctgga 300
agaccttctc ctcttgcaaa taaaacctca cccatgaatg ctcacgcaag ttttaattaca 360
gacctgaa 368
//
```

- *FASTA*: Spolu so svojím rozšírením FASTQ patrí medzi najobľúbenejšie a najpoužívanéjšie formáty na poli práce s DNA sekvenciami. Súbor môže obsahovať niekoľko sekvencií. Na začiatku každej sekvencie sa nachádza jednoriadkový opis začínajúci znakom '>'.

Príklad:

```
>AB000263 |acc=AB000263|descr=Homo sapiens mRNA for pclp|len=368
ACAAGATGCCATTGTCCCCGGCCTCCTGCTGCTGCTGCTCTCCGGGGCCACGGCCACCGCTGCCCTGCC
CCTGGAGGGTGGCCCCACCGGCCGAGACAGCGAGCATATGCAGGAAGCGGCAGGAATAAGGAAAAGCAGC
CTCCTGACTTTCTCGCTTGGTGGTTTGAGTGGACCTCCCAGGCCAGTGCCGGGCCCCTCATAGGAGAGG
AAGCTCGGGAGGTGGCCAGGCGGCAGGAAGGCGCACCCCCCAGCAATCCGCGCGCCGGGACAGAATGCC
CTGCAGGAACCTTCTTCTGGAAGACCTTCTCCTCCTGCAAATAAAACCTCACCCATGAATGCTCACGCAAG
TTTAATTACAGACCTGAA
```

- *FASTQ* [7]: Predstavuje rozšírenie formátu FASTA o hodnotenie kvality výstupu osekvenovaných nukleotidov. Niektoré DNA skladače vedú ku každému nukleotidu doplniť pravdepodobnosť

dobnosť toho, či je správne osekvenovaný. Na začiatku každej sekvencie je hlavička označená znakom '@', nasleduje DNA sekvencia, na novom riadku znak '+' a za ním, opäť na novom riadku znaky z ASCII tabuľky reprezentujúce kvalitu jednotlivých nukleotidov od ! (najnižšia kvalita) po ~ (najvyššia kvalita).

Príklad:

```
@AB000531 |acc=AB000531|descr=Homo sapiens mRNA for pclp|len=60
GATTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTGTTCAACTCACAGTTT
+
!&&*((( (** * +)) %%% ++)( %%%%) . 1 * * * - + * && ) * * 55CCF=====CCCCCCCC65
```

- **SAM** (angl. Sequence Alignment/Map Format): Ako oddeľovače sa v tomto formáte používajú tabulátory. Každá sekvencia (v jednom súbore ich môže byť opäť viac) pozostáva z voliteľnej hlavičky a zarovnanej sekcie. Hlavička začína znakom '@'. Formát predpisuje pomerne komplikovanú štruktúru, je vhodný najmä pre zarovnávanie sekvencií (viac 2.3). V praxi sa najčastejšie konvertuje na zrozumiteľnejší formát (napr. *FASTA*), až potom sa používa na skladanie DNA sekvencií. Jeden z nástrojov pre prácu so SAM formátom, ktorý umožňuje aj jeho konverziu do formátu *FASTA*, je nástroj *SAMtools* [24]. Formát SAM je bližšie opísaný v jeho špecifikácii [43].
- **BAM**: Predstavuje binárnu podobu formátu SAM. Binárna reprezentácia je pamäťovo šetrnejšia, špecifikácia tohto formátu je taktiež uvedená v rámci špecifikácie formátu SAM [43].

2.3 Problém skladania DNA sekvencií

DNA sekvencie môžu byť podkladom pre rôzne druhy geneticky zameraných analýz, avšak sekvencery produkujú len DNA sekvencie s náhodným rozmiestnením v rámci genómu. Pri danej sekvencii teda nevieme určiť chromozóm z ktorého pochádza, nevieme určiť jej pozíciu v nejakom chromozóme a ani vlákno DNA molekuly, odkiaľ krátka sekvencia pochádza [22]. Ak je teda našim cieľom skúmanie genómu nejakého organizmu, je potrebné tieto jednotlivé sekvencie pospájať, pokiaľ možno, do pôvodne osekvenovaných DNA molekúl.

Jedným z prístupov k skladaniu týchto sekvencií je *zarovnávanie sekvencií* (angl. DNA sequence alignment), ktoré ale vyžaduje, aby bol známy *referenčný genóm* daného organizmu. Sekvenovaním získané krátke sekvencie sú potom zarovnané vzhľadom na tento referenčný genóm, čo umožňujú malé rozdiely v DNA medzi jedincami (v prípade ľudí predstavujú rozdiely len cca 0.1% dĺžky genómu). Ak je známy referenčný genóm organizmu (napr. ľudí [21]), tento prístup je vo väčšine prípadoch najrýchlejší a najspoľahlivejší.

Úplne iný smer predstavuje tzv. *de-novo skladanie krátkych DNA sekvencií* (angl. de-novo short read

assembly), kedy sa snažíme poskladať sekvenovaný genóm bez akejkoľvek znalosti referenčného genómu, prostredníctvom skladačov DNA sekvencií. *Skladač DNA sekvencií* (angl. short reads assembler) je program, ktorého úlohou je z množiny krátkych sekvencií poskladať súvislý, pôvodne osekvenovaný genóm. Problém, ktorý riešime pomocou de-novo skladania nazývame *problém skladania DNA sekvencií* (FAP z angl. DNA Fragment Assembly Problem) a jeho cieľom je nájsť také spojenie krátkych sekvencií, ktoré čo najvernejšie zodpovedá sekvenovanému genómu. No ukazuje sa, že v praxi nie je možné nájsť také poskladanie, ktoré by stopercentne odrážalo sekvenovaný genóm, nakoľko skladanie komplikujú mnohé skutočnosti, ako chyby zakomponované do sekvencií počas procesu sekvenovania, opakovania v rámci DNA (či už tandemové, alebo rozptýlené), či kontaminácia sekvenovanej vzorky napr. DNA baktérií z okolitého prostredia [22] [10]. Pomocou de-novo skladania najčastejšie skladáme DNA organizmov, ktorých genóm nebol predtým osekvenovaný, s cieľom vytvoriť pre nich referenčnú vzorku. Ďalším využitím de-novo skladania je skladanie sekvencií rakovinových buniek, ktoré vykazujú značné odchýlky od referenčného genómu zdravých buniek.

2.3.1 Najkratší spoločný nadreťazec (SCS)

Problém skladania DNA sekvencií je vo svojej podstate definovaný príliš abstraktne. Na jeho riešenie potrebujeme konkrétnejšiu definíciu. Problém skladania DNA sekvencií preto môžeme chápať ako problém *najkratšieho spoločného nadreťazca* (SCS z angl. Shortest Common Superstring Problem), ktorý patrí do skupiny *NP-ťažkých* problémov.

Problém skladania DNA sekvencií možno definovať nasledovne. Nech reťazec r je vytvorený nad abecedou Σ . Nadreťazec q reťazca r definujeme ako $q = xry$, kde $x, y \in \Sigma^*$. Nadreťazec množín reťazcov $S = \{s_1, s_2, \dots, s_n\}$, $n \in \mathbb{N}$ je reťazec s obsahujúci všetky s_i , $1 \leq i \leq n$ ako podreťazce. *Problém nadreťazca* (angl. superstring problem) je potom definovaný nasledovne. Daná je množina reťazcov S a číslo $k \in \mathbb{N}$. Existuje pre množinu S nadreťazec dĺžky k ? Definovaním S ako množiny nestrácame všeobecnosť, pretože ak by S bola kolekcia reťazcov, kde sa niektoré reťazce opakujú aj viac ako raz, S má aj tak rovnakú množinu nadreťazcov ako $S' = \{s : s \text{ je v } S\}$ [14].

Problém najkratšieho spoločného nadreťazca má za úlohu nájsť pre množinu reťazcov S nadreťazec s s minimálnou dĺžkou k .

V prípade skladania DNA sekvencií abecedu Σ predstavuje množina $\Sigma = \{A, T, G, C\}$. SCS problém možno potom redukovať na nasledovné úlohy hľadania ciest v grafoch:

- nájdenie Hamiltonovskej kružnice (viac 2.4.2)
- nájdenie Eulerovského ťahu (viac 2.4.3)

- nájdenie cesty v probléme obchodného cestujúceho (viac 2.5.1)

2.4 Tradičné metódy skladania krátkych DNA sekvencií

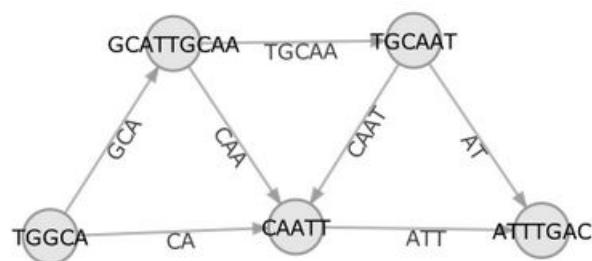
Tradičné metódy de-novo skladania krátkych DNA sekvencií sa spoliehajú predovšetkým na dostatočné pokrytie sekvenovaného genómu krátkymi sekvenciami. Kým pokrytie genómu (angl. coverage) v prípade Sangerovo sekvenovania nadobúdalo hodnoty 6 – 20x [22], v prípade krátkych sekvencií sa stretávame aj s hodnotami vyššími ako 100x [10]. Všetky tradičné metódy skladania krátkych DNA sekvencií využívajú prácu s rôznymi typmi grafov, z ktorých bližšie opíšeme najčastejšie používané *grafy prekrytí*, *de Bruijnove grafy* a *k-reťazcové grafy*.

2.4.1 Overlap-Layout-Consensus

Overlap-Layout-Consensus (OLC) [49] je jednou z najstarších metód skladania DNA sekvencií. Pre svoje vlastnosti je určená najmä na skladanie genómov s nižším pokrytím, ale s väčšou dĺžkou sekvencií. Táto metóda je jedna z najvhodnejších metód pre skladanie sekvencií získaných pomocou *Sangerovho sekvenovania*, no na de-novo skladanie krátkych sekvencií produkovaných pomocou sekvenovania novej generácie nie je dostatočná [22].

Metóda OLC pozostáva z troch krokov: (1) prekryvanie, (2) usporiadanie a (3) konsenzus, pričom v prvom kroku sa vytvorí základný graf prekrytí pre skladanie, v druhom kroku je tento graf komprimovaný na menšiu veľkosť a v poslednom kroku sa určí pravdepodobný reťazec DNA, z ktorého pochádzajú skladané sekvencie.

Vo fáze *prekryvania* (angl. overlap) sa porovná každá sekvencia s každou inou. Porovnáva sa ich dopredný smer i reverzný komplement, nakoľko nevieme, z ktorého vlákna DNA daná sekvencia pochádza. Tento krok preto zaberá veľa času a v praxi je ho potrebné vykonávať paralelne. Uzly v *grafe prekrytí* (angl. overlap graph) predstavujú sekvencie DNA. Orientovaná hrana sa medzi dvoma uzlami vytvorí, keď je medzi nimi dostatočne veľké prekrytie $\alpha w \beta$, kde α predstavuje prefix uzla, z ktorého hrana vychádza (αw), β sufix uzla, do ktorého hrana vchádza ($w \beta$) a w je ich spoločné podslovo. Rôzne skladače vyžadujú rôzne dĺžky prekrytia pre vytvorenie hrany v grafe prekrytí. Výsledkom sú dva typy prekrytí, *skutočné prekrytia* (angl. true overlaps) a *prekrytia opakovaní* (angl. repeat overlaps). Prekrytia opakovaní vzniknú vtedy, keď majú dve sekvencie spoločné požadované množstvo nukleotidov, ale v sekvenovanom genóme sú lokalizované na dvoch rôznych vzdialených častiach. Nakoľko nevieme v procese tvorby grafu prekrytí tieto prekrytia rozpoznať, musíme graf skonštruovať i s prekrytiami opakovaní. Cieľom je nájsť takú cestu grafom, ktorá navštívi každý uzol práve raz (túto vlastnosť spĺňa *Hamiltonovská kružnica*). Príklad grafu prekrytí znázorňuje obrázok 9.



Obr. 9: V grafe prekrytí reprezentujú uzly dané sekvencie DNA. Hrana medzi uzlami vznikne, ak je medzi dvoma uzlami (sekvenciami) dostatočné prekrytie. Na obrázku sú prekrytia vyznačené na hranách. Prevzaté z [49].

Nájsť cestu rozsiahlym grafom prekrytí ale nie je jednoduchá záležitosť. Preto aby bolo vôbec možné graf prekrytí prehľadávať je potrebné ho výrazne zjednodušiť v kroku *usporiadania* (angl. layout). Jednotlivé segmenty grafu sú zlúčené do *kontigov* (angl. contigs)⁹. V rámci grafu prekrytí sú kontigy spočiatku predstavované podgrafmi s veľkým množstvom vzájomne pospájaných uzlov. Ako náhle tieto podgrafy identifikujeme, zlúčime ich do jedného uzla - kontigu, čím sa graf prekrytí zjednoduší. Uzly sa zlučujú do kontigov, pokiaľ nenarazíme na rozdvojenie. *Rozdvojenia* (angl. forks) predstavujú uzly, ktoré sú spojené s dvoma alebo viacerými inými uzlami, ktoré nezdieľajú žiadny spoločný úsek. Rozdvojenia zvyčajne označujú hranice medzi opakovanými a neopakovanými časťami genómu. Závažnosti ohľadom opakování v grafoch prekrytí bližšie vysvetľuje článok [49].

V skomprimovanom grafe existujú 2 druhy kontigov, *unikátne kontigy* (angl. unique contigs), ktoré môžu byť zložené bez nejednoznačností a *kontigy opakování* (angl. repeat contigs). Kontigy opakování majú zvyčajne abnormálne veľké hodnoty pokrytia alebo sú spojené s extrémne veľkým množstvom iných kontigov. Na základe týchto skutočností vedú algoritmy tieto zloženia identifikovať. Unikátne kontigy sa spoja do väčších celkov – *kostier* (angl. scaffolds). Najčastejšie sa to udeje s použitím informácií z párových sekvencií, pomocou ktorých sú skladače schopné určiť, ako asi sú dané zloženia ďaleko. Po tom ako sa spoja zloženia do kostier, algoritmy sa pokúsia vyplniť medzery s použitím opakovaných kontigov.

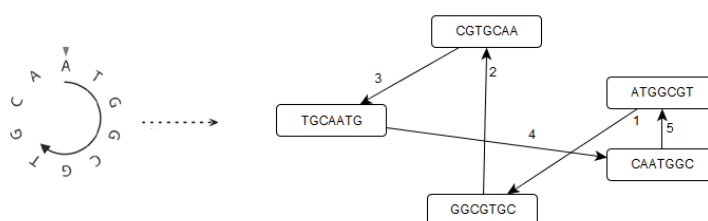
V poslednom kroku - *konsenzus* (angl. consensus) vytvárame ucelenú sekvenciu. V tomto bode je už graf prekrytí zredukovaný na menší počet kostier (ideálne jedinú). OLC algoritmus vypočíta konsenzus všetkých sekvencií, ktoré sa podieľajú na stavbe kostier. Ak sú medzi kostrami medzery, povedie to k viacerým výsledným sekvenciám, ktoré sú podmnožinami sekvenovaného genómu.

Príkladom skladačov, ktoré využívajú metódu OLC sú *TIGR*, *Celera*, *Phrap*, *CAP3*, *Atlas* a *ARACHNE*.

⁹Kontig je kolekcia sekvencií, ktoré sa zjavne prekrývajú a tvoria ucelenú supersekvenciu.

2.4.2 Riešenie SCS hľadaním Hamiltonovskej kružnice

Ak chceme riešiť problém nájdenia najkratšieho spoločného nadreťazca pomocou nájdenia *Hamiltonovskej kružnice*, musíme zostaviť *graf prekrytí*, ako v prípade metódy OLC. Metóda OLC teda predstavuje vylepšenie prístupu hľadania Hamiltonovskej kružnice. Nájdením Hamiltonovskej kružnice v grafe prekrytí sme zároveň našli najkratší spoločný nadreťazec [8]. Pôvodnú sekvenciu získame prečítaním začiatkovej časti každého uzla, ktorá nie je spoločná s iným, s ním spojeným uzlom. Ako príklad môžeme uviesť sekvenovanie kruhového chromozómu ATGGCGTGCA, ktorý bližšie zachytáva obrázok 10.



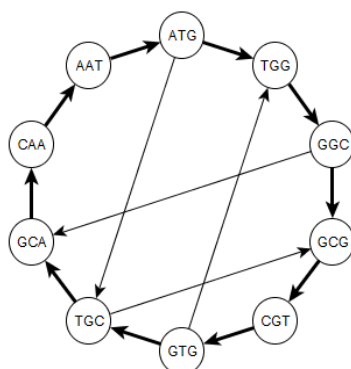
Obr. 10: Sekvenovaním kruhového chromozómu ATGGCGTGCA sme získali sekvencie ATGGCGT, GGCGTGC, CGTGCAA, TGCAATG a CAATGGC. Ich pospájaním do grafu prekrytí, nájdením Hamiltonovskej kružnice (postupnosť jej hrán značia čísla) a prečítaním prvých dvoch nukleotidov z každého uzla získame pôvodne sekvenovaný chromozóm. Prevzaté a upravené z [8].

Moderné skladače radšej pracujú s tzv. *k*-ticami (angl. *k*-mer), čo sú reťazce dĺžky *k*, ktoré sú kratšie ako skutočná sekvencia [8]. *K*-tice vytvoríme tak, že zvolíme číslo *k*, ktoré je kratšie ako získané sekvencie a každú sekvenciu (nech je jej dĺžka *n*) rozrežeme na $(n - k + 1)$ *k*-tic, kde prvá z nich začína jej prvým nukleotidom a ostatné sú vždy posunuté o jeden nukleotid v smere od 5' do 3'. To isté spravíme pre reverzný komplement sekvencie. Napríklad pre sekvenciu ATGGCG a $k = 3$ dostaneme *k*-tice $\{ATG, TGG, GGC, GCG, CGC, GCC, CCA, CAT\}$. Týmto spôsobom nám vznikne graf zobrazený na obrázku 11 (kvôli zjednodušeniu uvažujeme len jednovláknovú DNA), ktorého Hamiltonovská kružnica predstavuje, po prečítaní prvého nukleotidu každého uzla, riešenie problému skladania DNA sekvencií.

Nakoľko je hľadanie Hamiltonovskej kružnice *NP-úplný* problém a neexistuje žiadny rýchly algoritmus na jej nájdenie, bolo potrebné nájsť vhodnejšiu reprezentáciu pre riešenie problému hľadania najkratšieho spoločného nadreťazca, ktorú predstavuje riešenie pomocou *Eulerovského ťahu*.

2.4.3 Riešenie SCS hľadaním Eulerovského ťahu

V roku 1946 sa matematik Nicolaas de Bruijn zaoberal riešením všeobecného problému *najkratšieho spoločného nadreťazca*, ktorý obsahuje všetky možné podreťazce dĺžky *l* (*l*-tice) vytvorené nad danou



Obr. 11: Rozložením sekvencií na kratšie k -tice získame graf prekrytí na obrázku. Nájdením Hamiltonovskej kružnice (je vyznačená hrubo) a prečítaním prvého nukleotidu každej sekvencie získame osekvencovaný chromozóm. Prevzaté a upravené z [8].

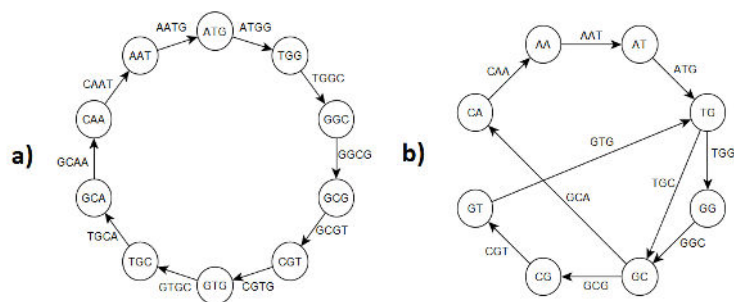
abecedou. Vo svojej práci sa inšpiroval Eulerom a výsledkom bolo predstavenie tzv. de Bruijnovho grafu.

De Bruijnov graf je graf B , ktorého uzly sú všetky $(l - 1)$ -tice zostrojené nad nejakou abecedou Σ . Orientovaná hrana medzi dvoma uzlami existuje práve vtedy, keď existuje l -tica, ktorej prefix predstavuje výstupný uzol a sufix vstupný uzol hrany. Nájdením Eulerovskej kružnice nájdeme aj najkratší spoločný nadreťazec všetkých reťazcov dĺžky l [8].

V rámci problému skladania DNA sekvencií však pracujeme s *neúplným de Bruijnovým grafom* pričom ho definujeme nasledovne. Daná je množina sekvencií $S = \{s_1, s_2, \dots, s_n\}$. De Bruijnov graf $G(S_k)$ definujeme ako graf s množinou vrcholov S_{k-1} (množina všetkých $(k - 1)$ -tíc z S) takto. Každá $(k - 1)$ -tica $v \in S_{k-1}$ je spojená orientovanou hranou s inou $(k - 1)$ -ticou $w \in S_{k-1}$, práve vtedy keď S_k obsahuje k -ticiu ktorá sa v prvých $k - 1$ nukleotidoch zhoduje s v a v posledných $k - 1$ nukleotidoch sa zhoduje s w . Každá k -tica z S_k tak predstavuje hranu v G [36]. Z definície vyplýva, že každá $(k - 1)$ -tica sa v grafe ako vrchol vyskytuje len raz.

Uvažujme príklad z kapitoly 2.4.2, v ktorom bol sekvenovaný kruhový chromozóm ATGGCGTGCA. Skonstruovaním de Bruijnového grafu sme problém hľadania Hamiltonovskej kružnice zjednodušili na problém hľadania Eulerovského cyklu. Euler dokázal, že graf má Eulerovskú kružnicu práve vtedy, keď je *balancovaný* (každý uzol má rovnaký počet vstupných a výstupných vrcholov). Z Eulerovej vety vyplýva, že de Bruijnov graf je balancovaný ak obsahuje všetky k -tice prítomné v genóme. Pre nájdenie Eulerovskej kružnice potom existuje efektívny algoritmus. Obrázok 12 b) znázorňuje konštrukciu de Bruijnového grafu pre uvedený príklad s $k = 3$. Obrázok 12 a) zase ukazuje konštrukciu de Bruijnového grafu pre $k = 4$ a znázorňuje rozdiel medzi de Bruijnovým grafom a grafom prekrytí.

V skutočnom svete sú veci ešte komplikovanejšie. Základným predpokladom pre existenciu Eule-



Obr. 12: Uvažujeme príklad z kapitoly 2.4.2. Obrázok a) znázorňuje de Bruijnov graf pre $k = 4$, obrázok b) de Bruijnov graf pre $k = 3$. Hrany predstavujú dané k -tice a uzly k nim prislúchajúce $(k - 1)$ -tice. Nájdením Eulerovskej kružnice získame sekvenovaný chromozóm.

rovskej kružnice je to, že v grafe musia byť prítomné všetky podreťazce dĺžky l daného genómu. Sekvencery väčšinou neprodukujú sekvencie, ktoré by predstavovali všetky l -tice sekvenovaného genómu a to ani v prípade veľkých *pokrytí genómu* (viac ako 100x). Práve preto sa namiesto samotných sekvencií dĺžky l pracuje s kratšími k -ticami, čo vedie k tomu, že pre dostatočne malé k budú v grafe prítomné takmer všetky k -tice z genómu [8].

Ďalší problém predstavujú *chyby* v procese sekvenovania, ktoré spôsobujú v grafe rôzne slučky, čo komplikuje skladanie. Ešte horšie je to, že v DNA sú nepresné opakovania, ktoré sa môžu líšiť len o 1 nukleotid, čo má za následok rovnaké typy slučiek, ktoré nevieme rozlíšiť. Preto je potrebné vykonať korekciu chýb, pričom rôzne skladače ju vykonávajú inak. Bližší opis toho, ako ju vykonáva skladač EULER-SR je v článku [36].

Ďalší problém predstavuje to, že častejšie sa sekvenujú *lineárne chromozómy*. Tak nie je graf uzavretý a nenašli by sme v ňom Eulerovskú kružnicu. Problém preto chápeme ako problém hľadania *Eulerovského ťahu*, čo nevyžaduje aby bol ťah uzavretý.

Asi najväčší problém predstavujú *opakovania*. Tie sa v procese hľadania Eulerovskej kružnice skrátia. Cestu $v_1 \dots v_n$ v de Bruijnovom grafe voláme *opakovanie* práve vtedy keď $\text{indegree}(v_1) > 1$, $\text{outdegree}(v_n) > 1$ a $\text{indegree}(v_i) = \text{outdegree}(v_i) = 1$ pre $1 < i \leq n - 1$. Funkcia *indegree* vracia počet vstupných hrán do vrcholu a funkcia *outdegree* počet výstupných hrán z vrcholu. Opakovania spôsobujú, že niektoré hrany grafu je potrebné navštíviť viac krát, čo predstavuje problém *čínskeho poštára* (angl. Chinese Postman Problem). Tento problém však vieme previesť naspäť na problém hľadania Eulerovskej kružnice tak, že do miest, kde sa nachádzajú opakovania vložíme viacnásobné hrany. To si vyžaduje, aby sme vedeli, koľko krát je každá z k -tic prítomná v genóme. Ak je k -tica v genóme prítomná m krát, potom bude jej odpovedajúca hrana spojená s uzlom prefixu a sufixu m hranami. V praxi ale nevieme nič o počtoch k -tic v genóme, na to by sme potrebovali mať genóm už zložený. No práve pomocou párových sekvencií sme schopní aspoň približne určiť dĺžku opakovania a na základe toho vytvoriť násobné hrany [8]. Prítomnosť opakovaní a chýb sekvenovania tak vedú

k zovšeobecneniu na problém *Eulerovskej supercesty* (angl. Eulerian Superpath Problem), ktorý pre daný Eulerovský graf a kolekciu ciest v tomto grafe hľadá Eulerovskú cestu grafom obsahujúcu všetky cesty z danej kolekcie [36].

I napriek detekcii chýb a snahe vysporiadať sa s opakovaniami nebudeme schopní zrekonštruovať genóm do jeho pôvodnej podoby. Niektoré úseky DNA mohli byť neosekvenované, čo má za následok zvýšený počet kontigov. Taktiež na počet kontigov vplývajú aj samotné opakovania, chyby a voľba k . Najlepšia voľba k je pre každý genóm rôzna a je potrebné vyskúšať rôzne zloženia s rôznymi k a z nich vybrať to najlepšie [8].

Medzi skladače využívajúce de Bruijnove grafy patria skladače EULER-SR, EULER-DB, Velvet, ALL-PATHS, ABySS a SOAPdenovo. Aj keď Eulerovský ťah vieme nájsť v polynomiálnom čase, bolo dokázané, že riešenie problému Eulerovskej supercesty je *NP-ťažký* problém.

2.4.4 k -Reťazcové grafy

Benjamin Jackson v práci [22] vytvoril skladač YAGA (z angl. Yet Another Genome Assembler), ktorý využíva k -reťazcové grafy (angl. k -string graphs). Pre reprezentovanie DNA molekuly $\{s, s'\}$ je použitá tzv. *kanonická reprezentácia DNA molekuly*, kde molekula m je chápaná ako usporiadaná dvojica $m = (m^+, m^-)$, pričom m^+ značí lexikograficky väčší reťazec a m^- lexikograficky menší reťazec z dvojice $\{s, s'\}$. Keďže m^- vieme určiť z m^+ , molekula môže byť reprezentovaná len reťazcom m^+ , ktorý nazývame *reprezentatívny reťazec*. Ďalej je definovaná operácia *pozitívneho posunu* ako $\overrightarrow{mc}$, kedy na koniec pozitívneho reťazca m^+ je pripojený nukleotid c a z jeho začiatku je jeden nukleotid odobraný. Podobne je definovaný *negatívny posun* značený $\overleftarrow{mc}$ pre reťazec m^- .

Dvojito usmernený graf je graf $G = \{V, E\}$, kde hrany predstavujú dvojicu $((d, d_u), (v, d_v))$, pričom $d_u \in \{out, in\}$ predstavuje usmernenie hrany pri uzle u . Pre potreby skladania sekvencií autor reprezentuje sekvencie ako cesty v *dvojito usmernenom de Bruijnovom grafe* (v práci nazvanom aj ako k -reťazcový graf). Uzly predstavujú podmolekuly dĺžky k (tzv. k -spektrum) a hrany predstavujú podmolekuly dĺžky $k + 1$ (tzv. $(k + 1)$ -spektrum). Hrana $((u, d_u), (v, d_v))$ medzi uzlom u a v existuje len vtedy, ak existuje taký posun, ktorý transformuje molekulu u na molekulu v (a v na u). Ak sa u transformuje na v pozitívnym posunom, potom $d_u = out$. Ak u je transformované na v negatívnym posunom, potom $d_u = in$. Podobne pre d_v .

Prechod dvojito usmerneným grafom musí rešpektovať usmernenia hrán pri jednotlivých vrchoch. Ak do uzla vstupujeme cez hranu *out*, musíme z neho vystúpiť pomocou hrany *in* a naopak. Každá hrana v grafe je označená dvoma znakmi. Každé toto značenie predstavuje prvý znak oboch reťazcov $(k + 1)$ -tice, ktorá je predstavovaná touto hranou. Hrany značíme ako c_{uv}/c_{vu} , kde c_{uv} predstavuje

prvý znak m_u^+ , ak $d_u = out$, inak predstavuje prvý znak m_u^- a c_{vu} predstavuje prvý znak m_v^+ ak $d_v = out$, inak prvý znak m_v^- . Ukážka grafu je znázornená na obrázku 13. Kľúčovou vlastnosťou k -reťazcového grafu je, že každá sekvencia môže byť získaná z grafu prechodom cestou, ktorá jej zodpovedá, spájaním korešpondujúcich označení hrán pozdĺž cesty a pridaním sufixu dĺžky $k - 1$ z posledného uzla na ceste. Ideálne tak pre každý chromozóm existuje v grafe cesta.

Autor sa v práci rozsiahlo venuje spôsobu paralelnej konštrukcie grafu, jeho komprimácii, identifikácii a odstraňovaniu rôznych typov chýb a použitiu párových sekvencií. Záverečným prechodom grafom získame množinu kontigov, ktoré zodpovedajú sekvenovanému genómu.



Obr. 13: Obrázok znázotňuje všetky štyri spôsoby, ako môžu byť spojené uzly v dvojito usmernenom k -reťazcovom grafe. Prevzaté z [22].

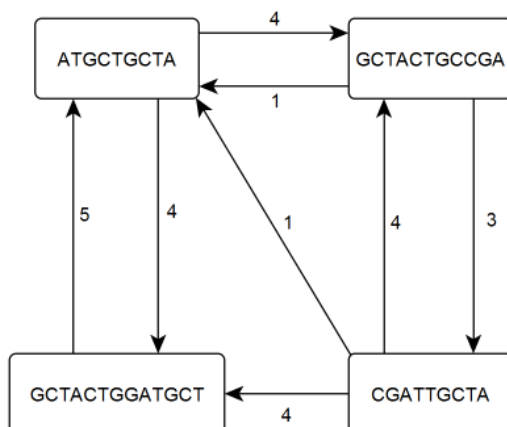
2.5 Umelá inteligencia v riešeníach problému skladania DNA sekvencií

Keďže problém skladania krátkych DNA sekvencií je pri použití akéhokoľvek tradičného prístupu *NP-tažký* problém, do pozornosti sa dostáva čoraz väčší počet skladačov, ktoré sa snažia o optimalizáciu skladania použitím algoritmov z umelej inteligencie. V posledných rokoch bolo predstavených mnoho riešení, ktoré využívajú rôzne prístupy od simulovaného žihania, cez evolučné algoritmy až po biologicky inšpirované algoritmy. Mnoho z týchto algoritmov rieši problém skladania DNA sekvencií jeho transformovaním na *problém obchodného cestujúceho*.

2.5.1 Skladanie DNA sekvencií a TSP

Problém skladania DNA sekvencií (FAP) úzko súvisí s problémom *obchodného cestujúceho* (TSP z angl. traveling salesman problem) [35]. Rovnako ako TSP, aj FAP sa obmedzuje len na riešenia, kde je každá sekvencia (resp. mesto) použitá len jedenkrát. Vzdialenosti medzi uzlami (resp. mestami v TSP) predstavujú v prípade FAP *mieru prekrytí* jednotlivých sekvencií, ktoré predstavujú dané uzly (resp. mestá). Transformáciou FAP na TSP však získame *asymetrickú verziu TSP*, v ktorej sú vzdialenosti medzi dvoma uzlami m a n rôzne v závislosti od toho, či prechádzame z uzla m do n , alebo z uzla n do m . Hlavný rozdiel medzi TSP a FAP je v tom, že každá platná cesta v prípade TSP predstavuje kružnicu a všetky štartovacie body na tejto kružnici predstavujú ekvivalentné riešenia. V prípade FAP máme určený počiatočný i koncový bod skladania, nie je jedno, odkiaľ čítame poskladanú sekvenciu.

V štandardnom TSP taktiež hľadáme najkratšiu vzdialenosť, no v prípade FAP chceme maximalizovať prekrytia medzi jednotlivými sekvenciami. Príklad grafu, ktorý získame transformáciou FAP na TSP udáva obrázok 14.



Obr. 14: Ukážka grafu prekrytí pre riešenie FAP pomocou TSP. Hrany sú ohodnotené veľkosťou prekrytí medzi dvoma sekvenciami predstavujúcimi mestá.

2.5.2 Genetický algoritmus (GA)

Jeden z prvých pokusov o aplikovanie *genetického algoritmu* na skladanie DNA sekvencií publikovali autori Parsons, Forrest a Burks [35] už v roku 1993. Princíp fungovania genetického algoritmu je nasledovný. Na začiatku je náhodne vygenerovaná *populácia* jedincov, z ktorých každý reprezentuje nejaké riešenie problému. Títo jedinci sú reprezentovaní pomocou bitového reťazca (*chromozómu*) predstavujúceho ich genetickú informáciu. Potom daní jedinci v jednotlivých iteráciách s určitou pravdepodobnosťou vstupujú do *reprodukcie*, kde sa *krížia*, pričom môže dochádzať aj k ich *mutáciám*, čím sa vytvárajú nové populácie. Kríženie jedincov má za následok vytváranie lepších jedincov a mutácie zabezpečujú, že nedôjde k zaseknutiu v lokálnom extréme. Schopnosť jedinca prežiť do ďalšej populácie udáva *fitnes funkcia*, ktorá určuje to, ako je jedinec dobrý z hľadiska riešenia problému i to s akou pravdepodobnosťou sa bude krížiť s ostatnými jedincami. Výber jedincov do reprodukcie zabezpečuje algoritmus *selekcie*, ktorý pri výbere jedincov zohľadňuje hodnotu fitnes jedincov v populácii. Fitnes funkcia je vlastne jediným prepojením medzi genetickým algoritmom a riešeným problémom.

Autori v článku použili transformáciu z FAP na TSP. Genetické algoritmy ale nemožno priamočiaro aplikovať na TSP. Problém predstavuje to, ako namapovať binárny reťazec na riešenie TSP tak, aby kríženie produkovalo *platné usporiadania sekvencií* (angl. legal layout). Autori si spomedzi dvoch možností, buď vytvoriť špecializovanú funkciu kríženia, ktorá by produkovala len platné usporiadania, alebo vytvoriť binárnu reprezentáciu jedinca tak, aby štandardne používané funkcie kríženia produkovali legálne usporiadania, vybrali možnosť druhú. Na kríženie potom použili štandardné

dvojbodové kríženie, ale akékoľvek štandardné kríženie by produkovalo platné usporiadania. Reprézenciu chromozómu stanovili nasledovne. Majme n sekvencií, zvolíme $k \in \mathbb{N}$ také, že $2^k \geq n$. Požadovaný bitový reťazec potom bude mať $m = k \times (n + 1)$ bitov, chromozóm bude teda predstavovať pole $(n + 1)$ kľúčov. Na získanie možného riešenia problému sa najskôr zoznam prvých n kľúčov zoradí. Ak sa kľúč na pozícii j chromozómu jedinca objaví na pozícii i zoradeného zoznamu, potom sa sekvencia j nachádza na pozícii i prechodného usporiadania. Posledný kľúč v jedincovom chromozóme špecifikuje začiatočnú pozíciu v prechodnom usporiadaní. Záverečné usporiadanie je získané len jednoduchou rotáciou prechodného usporiadania tak, aby bola prvá sekvencia tá, ktorú určuje začiatočná pozícia. Začiatočná pozícia je potrebná na to, aby kríženie mohlo prehodiť sekvencie z jedného konca usporiadania na druhý. Ako príklad môžeme uvažovať nasledovný bitový reťazec ($s \ k = 3$): 2-7-1-5-3-2. Tento reťazec predstavuje usporiadanie sekvencií 1-5-4-2-3 s prechodným usporiadaním 3-1-5-4-2. Keďže najmenšia hodnota 1 je na treťom mieste v reťazci, prvý znak v prechodnom usporiadaní je 3. Druhá najnižšia hodnota 2 je na prvej pozícii v reťazci a preto druhé pole v prechodnom usporiadaní je 1. Posledný kľúč je 2 a preto finálne usporiadanie začína druhou pozíciou prechodného usporiadania.

Posledným krokom je vytvorenie vhodnej *fitness funkcie*. Autori použili a porovnávali dve fitness funkcie. Prvá z nich ($F1$) predstavuje súčet prekrytí jednotlivých sekvencií pre každé susedné sekvencie:

$$F1 = \sum_{i=1}^{n-1} w(i, i + 1)$$

Túto funkciu sa snaží genetický algoritmus maximalizovať. Funkcia $w(m, n)$ vracia hodnotu prekrytia fragmentov m a n . Druhá funkcia ($F2$) navyše skúma prekrytia aj medzi vzdialenejšími sekvenciami. Genetický algoritmus sa ju snaží minimalizovať. Minimalizovanie tejto funkcie spôsobí presun sekvencií s väčším prekrytím bližšie k sebe.

$$F2 = \sum_{i=1}^n \sum_{j=1}^n |i - j| \times w(i, j)$$

Algoritmus bol testovaný s pravdepodobnosťou kríženia 0.6 a s pravdepodobnosťou mutácie 0.005 pre každý bit. Maximálny počet iterácií bol 400000. Autori zistili, že usporiadania, ktoré vznikli použitím funkcie $F2$ sú lepšie ako tie, čo vznikli použitím $F1$. Autori taktiež odhalili, že zvolená reprezentácia chromozómu nevytvára stále len lepšie riešenia. Autori algoritmus porovnali s *greedy algoritmom*, ktorý v niektorých prípadoch dosahoval lepšie výsledky, ale v iných výrazne zaostával.

2.5.3 Simulované žihanie (SA)

Autori v [26] zhrnuli rôzne optimalizačné metódy použiteľné pre problém skladania DNA sekvencií. Jedným z prístupov je riešenie FAP pomocou simulovaného žihania. *Simulované žihanie* (angl. Simulated Annealing) je stochastická optimalizačná metóda s pôvodom v štatistickej mechanike. Je založená na princípe procesu žihania v priemysle. Materiál sa zahreje na vysokú teplotu, kým neskvapalnie, potom sa nechá teplota pomaly klesať. Ak je materiál ochladzovaný dostatočne pomaly, atómy dosiahnu najstabilnejšiu (optimálnu) konfiguráciu.

V ukážke 1 je znázornený priebeh algoritmu. Na začiatku sa inicializuje teplota T a počiatočné riešenie. Riešenie s_1 je akceptované ako nové riešenie ak $\gamma = f(s_1) - f(s_0) < 0$. Zotrvanie v lokálnom optime je ošetrené akceptovaním aj riešení, ktoré majú $\gamma > 0$ s pravdepodobnosťou $\exp(-\gamma/T)$. Tento proces sa niekoľko krát opakuje, pričom počet opakovaní udáva parameter *dĺžka_markovovej_reťaze*. Následne sa zníži teplota T a celý proces sa iteratívne opakuje, kým sa nedosiahne stav zmrazenia v bode T_{min} .

```
t = 0
inicializuj(T)
s0 = inicializuj_pociatocne_riesenie()
v0 = vyhodnot(s0)
while T > T_min do
  while t mod dlzka_markovovej_reťaze == 0 do
    t = t + 1
    s1 = generuj(s0, T)
    v1 = vyhodnot(s0, T)
    if akceptuj(v0, v1, T) then
      s0 = s1
      v0 = v1
    end
  end
  T = aktualizuj_teplotu(T)
end
```

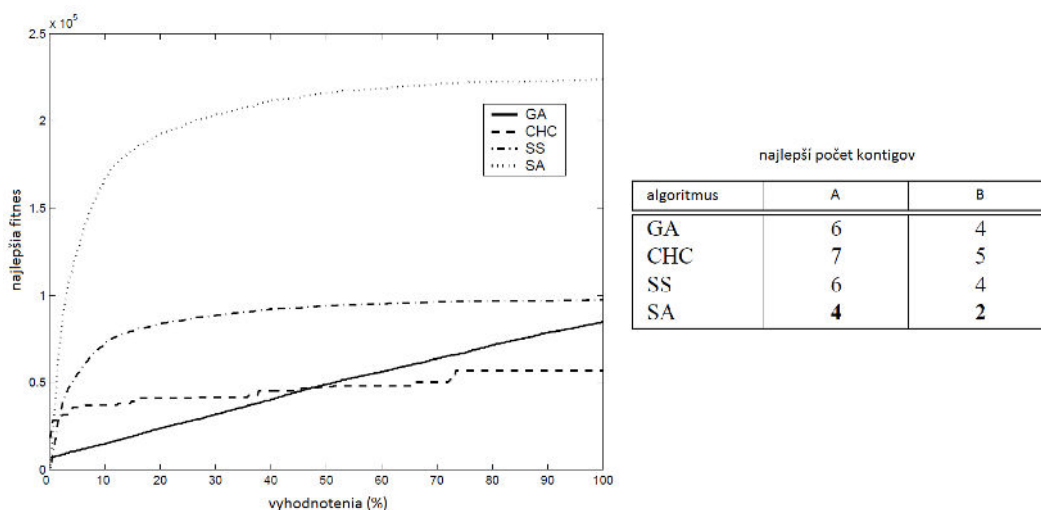
Ukážka 1: Pseudokód algoritmu simulovaného žihania.

Autori reprezentovali riešenie ako *celočíselnú permutáciu*. Každéj sekvencii je priradené číslo, pričom sa čísluje od 0 do $n - 1$, kde n predstavuje počet sekvencií. Reprezentácia pomocou permutácie vyžaduje špeciálne operátory, ktoré určia, či ide o *platné usporiadanie*. Pre udržanie platného riešenia je nutné, aby všetky sekvencie boli v usporiadaní prítomné, pričom sekvencie sa nesmú opakovať.

Ako fitness funkcie boli použité rovnaké funkcie ako v prípade vyššie opísaného genetického algoritmu (viac 2.5.2). Tieto funkcie slúžili na vyhodnotenie toho, ako dobré riešenia sú v danej iterácii prítomné. Program sa ukončil, keď bolo dosiahnuté maximum iterácií algoritmu, alebo keď sa riešenie už ďalej nezlepšovalo. Pre kontrolu poklesu teploty v nasledovných krokoch bola zvolená funkcia *proporčného simulovaného žihania* $T_k = \alpha \cdot T_{k-1}$, kde $\alpha \in (0, 1)$. Autori po viacerých pokusoch zvolili $\alpha = 0.99$ a

parameter $dĺžka_markovovej_reťaze = počet\ vyhodnotení / 100$. Nové riešenia boli generované tak, že sa v permutácii náhodne vybrali dva body a časť permutácie medzi nimi sa invertovala. Maximálny počet iterácií (vyhodnotení) bol nastavený na 512000.

Autori nad rovnakou reprezentáciou navrhli aj *genetický algoritmus* (GA), algoritmus *CHC* (Cross generational elitist selection, Heterogenous recombination, and Cataclysmic mutation), ktorý predstavuje určité vylepšenie genetického algoritmu a *Scatter Search* algoritmus (SS), čo je metaheuristika využívajúca referenčnú množinu pre kombináciu riešení a konštrukciu nových riešení. Medzi týmito algoritmami výrazne dominovalo *simulované žihanie*, ktoré našlo výsledok v priemere 5 – 12 krát rýchlejšie ako ostatné algoritmy. Porovnanie nárastu najlepšej fitness v populácii v rámci jednotlivých algoritmov udáva obrázok 15. Algoritmus simulovaného žihania taktiež pre testované problémy našiel najmenší počet kontigov spomedzi ostatných algoritmov.



Obr. 15: Obrázok vľavo udáva, ako sa správala najlepší fitness v populácii pre jednotlivé porovnávané algoritmy. Ako vidíme, simulované žihanie výrazne dominuje. Obrázok vpravo udáva najlepší počet kontigov, ktorý boli dané algoritmy schopné dosiahnuť pre 2 testované inštancie A a B. Prevzaté z [26].

2.5.4 Optimalizácia kolóniou mravcov (ACO)

Autori článku [30] predstavili riešenie FAP pomocou *optimalizácie kolóniou mravcov* (ACO z angl. Ant Colony Optimization). Zlepšenie efektívnosti prehľadávania v rámci GA môže byť dosiahnuté ak eliminujeme redundancie z reprezentácií riešení. Keďže FAP je kombinatorický optimalizačný problém, je potrebné ho reprezentovať pomocou permutácie. Rôzne zoradenia v permutáciách však môžu produkovať rovnaké zloženie DNA. Možný prístup je eliminácia efektu redundancie z prehľadávajúcich častí v GA. Jedným zo stochastických algoritmov, ktorý nie je redundanciou nijako zaťažovaný predstavuje práve *systém mravej kolónie* (ACS z angl. Ant Colony System).

Skutočné mravce sú schopné nájsť najkratšiu cestu medzi *potravou* a *mraveniskom* bez akejkoľvek vizuálnej informácie, len na základe *feromónovej stopy*. Počas chôdze mravce vypúšťajú na zem feromóny a s určitou pravdepodobnosťou nasledujú feromónové stopy iných mravcov. Spôsob, akým mravce používajú feromóny je nasledovný. Predstavme si, že mravec sa dostane do bodu, keď si musí zvoliť cestu, keďže nevie nič o tom, ktorá cesta je lepšia, zvolí si ju náhodne. Môžeme predpokladať, že polovica mravcov si zvolí nesprávnu cestu a polovica správnu cestu. Nakoľko je správna cesta kratšia, feromónová stopa sa na nej bude akumulovať rýchlejšie. Po niekoľkých takýchto prechodoch bude feromónová stopa na kratšej ceste dostatočne silná na to, aby ovplyvnila rozhodovanie ostatných mravcov.

ACS je algoritmus založený na *kolektívnom správaní hmyzu* a je vhodný na riešenie kombinatorických úloh ako TSP. Mravec sa v prípade TSP o prechode medzi mestami rozhoduje na základe *vzdialeností* medzi nimi a na základe *feromónovej stopy*. Úroveň feromónov medzi dvoma mestami koreluje s počtom mravcov, ktorí danou cestou prešli. Po tom, ako všetky mravce prešli všetkými mestami, cesty na najkratšej globálnej ceste obdržia dodatočnú dávku feromónov. V prípade TSP má teda každá hrana medzi mestami r a s okrem svojej dĺžky $\delta(r, s)$ priradenú aj feromónovú stopu $\tau(r, s)$, ktorá je počas behu programu aktualizovaná mravcami [9]. Jednotlivé mestá mravce navštevujú na základe tzv. *prechodového pravidla*. Podrobnosti o aplikovaní ACS na TSP uvádza článok [9].

Autori opisovaného článku kvôli vlastnostiam ACS previedli FAP na asymetrickú verziu TSP. Asymetrickosť spôsobuje relatívna poloha každého fragmentu (môže byť sprava doľava alebo zľava doprava) a to, že nevieme určiť, z ktorého vlákna dvojzávitnice fragment pochádza. Pre dva fragmenty teda existuje 8 možností, ako môžu byť voči sebe orientované (2 možnosti vzájomného pôvodu fragmentov, 2 možnosti orientácie prvého fragmentu a 2 možnosti orientácie druhého fragmentu). Jednotlivé možnosti, nazývané *konfigurácie*, majú priradené čísla a presne identifikujú orientáciu fragmentov a a b . Algoritmus najskôr zistí maximálne prekrytie medzi každými dvoma sekvenciami a vylúči všetky fragmenty, ktoré predstavujú podreťazce iných fragmentov, čím sa problém zjednoduší. Každá konfigurácia ale vytvára inú hodnotu maximálneho prekrytia. Po výpočte prekrytí je možné skonštruovať zoradenie sekvencií, ktoré bude viesť ku zjednotenej supersekvencii.

Na zostavenie usporiadania sekvencií sa používa *zoznam kandidátov* (angl. candidate list), ktorý pre danú sekvenciu s udáva najlepšie kandidátske sekvencie usporiadané z hľadiska veľkosti prekrytí spolu s dodatočnými informáciami o ich vzájomných orientáciách. Počet najlepších sekvencií pre sekvenciu s , ktoré budeme uchovávať, udáva veľkosť zoznamu cl . Pre každú sekvenciu tak existuje zoznam kandidátov danej veľkosti cl . Štruktúra tohto zoznamu je bližšie opísaná v článku [30], pričom veľkosť zoznamu pre riešenie FAP s n sekvenciami je kvôli uchovaniu dodatočných informácií rovná $2 \times 2n \times cl$. Zoznam sa používa v priebehu aplikovania *prechodového pravidla* na zjednodušenie TSP s veľkým množstvom uzlov [9].

Tri základné komponenty ACS predstavujú *prechodové pravidlo* (angl. state transition rule), *lokálne pravidlo aktualizovania feromónovej stopy* (angl. local pheromone-updating rule) a *globálne pravidlo aktualizovania feromónovej stopy* (angl. global pheromone-updating rule). Keďže lokálne pravidlo aktualizovania feromónovej stopy vyžaduje, aby jeho *faktor aktualizovania feromónov* bol nepriamo úmerný k dĺžke TSP cesty, po konštrukcii zoznamov kandidátov algoritmus postupuje s použitím *heuristiky najbližšieho suseda* (angl. Nearest Neighbour Heuristic Rule), pomocou ktorej sa sekvencie zoradia, čím získame odhad danej dĺžky. Najskôr sa náhodne vyberie sekvencia r spolu s jej orientáciou (napr. sprava doľava). Následne sa zo zoznamu kandidátov vyberie sekvencia s , ktorá má najväčšie prekrytie so sekvenciou r , pričom pomocou zoznamu kandidátov vieme určiť aj ich konfiguráciu (teda ich vzájomnú polohu). Proces sa potom opakuje pre sekvenciu s , ktorá je už zo zoznamov kandidátov odstránená. Ak sa nejaký zoznam kandidátov vyprázdni ešte pred skončením spájania sekvencií, vyberie sa taká nepoužitá sekvencia, ktorá má s naposledy vybratou sekvenciou najväčšie prekrytie. Na konci vznikne cyklická sekvencia, ktorá sa v prípade lineárnych chromozómov rozdelí v mieste najmenšieho prekrytia medzi dvoma sekvenciami a získa sa jej dĺžka.

Na aplikovanie ACS na FAP treba upraviť aj *prechodové pravidlo* pomocou ktorého sa mravce rozhodujú, ktorú sekvenciu pripoja ako ďalšiu v procese prechádzania grafom. Štandardne sa v prípade TSP robí rozhodnutie na základe *inverznej vzdialenosti miest*. Nakoľko FAP je maximalizačný problém, túto vzdialenosť je možné vyjadriť priamo pomocou veľkosti prekrytia medzi dvoma sekvenciami. Podobne, ďalší atribút, ktorý treba upraviť predstavuje *faktor aktualizovania feromónov* ($\Delta\tau$), ktorý určuje prírastok k feromónovej stope v procese lokálnej aktualizácie feromónovej stopy. V článku [9] ho odvodili pre TSP s počtom miest n pomocou heuristiky najbližšieho suseda ako inverziu vzdialenosti, ktorú táto heuristika vrátila (vzdialenosť L_{nn}) takto: $\Delta\tau(r, s) = (n \cdot L_{nn})^{-1}$, no v prípade FAP je kvôli maximalizačnej povahe problému tento faktor nastavený na $\Delta\tau(r, s) = L_{nn}/n$.

Autori pri testovaní použili v ACS 10 mravcov a 15 položiek v zozname kandidátov (hodnota cl). Autori zhodnotili, že výkony ACS a priameho aplikovania heuristiky najbližšieho suseda sú v prípade riešenia FAP takmer rovnaké. ACS však prekonáva prehľadávanie pomocou heuristiky najbližšieho suseda v prípade množín sekvencií s viacerými kontigmi.

2.5.5 Umelé spoločenstvo včiel (ABC)

Autori článku [13] použili na riešenie problému skladania DNA sekvencií algoritmus *umelého včelieho spoločenstva* (ABC z angl. Artificial Bee Colony). Algoritmus predstavuje metaheuristiku založenú na inteligentnom správaní roja včiel. Včely používajú rôzne techniky, ako napr. tanec, ktoré môžu byť použité na vývoj nových inteligentných algoritmov prehľadávania. Spoločenstvo včiel v ABC algoritme obsahuje tri typy včiel: *zamestnané včely* (angl. employed bees), *prizerajúce sa včely* (angl. onlookers) a *prieskumníkov* (angl. scouts). *Prizerajúce sa včely* čakajú v tanečnej miestnosti úľa, kde

sa na základe tanca zamestnaných včiel rozhodnú, ktorý zdroj potravy navštívia. Včely, ktoré putujú za zdrojom potravy, ktorý už v minulosti navštívili sa nazývajú *zamestnané včely*. Prvá polovica spoločenstva (v zmysle ABC) pozostáva zo zamestnaných včiel a druhá polovica z prizeraúcich sa včiel. Na každý zdroj potravy v okolí úľa existuje len jedna zamestnaná včela. Zamestnaná včela, ktorej zdroj potravy je vyčerpaný sa stáva prieskumníkom.

Cieľom FAP je generovať permutáciu sekvencií, ktorá minimalizuje počet kontigov a maximalizuje hodnotu fitness. V predstavenom algoritme ABC, každý *zdroj potravy* reprezentuje permutáciu sekvencií. Pozícia zdroja reprezentuje možné riešenie a *množstvo nektáru* predstavuje kvalitu (fitness) asociovaného riešenia. V inicializačnom kroku sa poloha zdrojov potravy generuje náhodne. Po inicializácii sa zdroje potravy opakovane prehľadávajú zamestnanými včelami, prizeraúcimi sa včelami a prieskumníkmi. Zamestnané a prizerajúce sa včely produkujú modifikácie riešenia pre hľadanie novej potravy a skúmajú množstvo nektáru nového riešenia. Na generovanie modifikovaných riešení bolo použité *problému si vedomé lokálne prehľadávanie* (PALS z angl. Problem Aware Local Search) na permutácii μ predstavené v článku [27]. Ako *fitness funkcia* bola použitá funkcia F^1 pre genetický algoritmus skladania DNA sekvencií (pozri 2.5.2). Fitness funkcia vyhodnocuje množstvo nektáru. Ak je množstvo nektáru nového modifikovaného riešenia vyššie ako predchádzajúceho riešenia, včela si zapamätá nové riešenie a predchádzajúce zabudne, inak si pamätá predchádzajúce riešenie. Prizerajúce sa včely si vyberajú zdroj potravy s určitou pravdepodobnosťou závislou od množstva nektáru. Táto pravdepodobnosť je vypočítaná ako $p_\mu = F_\mu / global_{max}$, kde $global_{max}$ predstavuje maximálnu fitness spomedzi všetkých zdrojov potravy. Pseudokód algoritmu ABC je uvedený v ukážke 2.

```

1: Inicializuj potencialne zdroje jedla pre zamestnane vcely.
2: while nie su splnene podmienky ukoncenia do
3:   Kazda zamestnana vcela leti k zdroju jedla, ktory ma v pamati,
     urci susedne zdroje (okolie aktualneho riesenia), vyhodnoti
     mnozstvo ich nektaru a tieto informacie odtancuje v uli.
4:   Kazda prizerajuca sa vcela pozera na tanec zamestnanych vciel,
     na zaklade ktoreho si vyberie jeden zo zdrojov a leti k nemu.
     Po vybere suseda v okoli vyhodnoti mnozstvo jeho nektaru.
5:   Zisti sa, ktore zdroje jedla su opustene a nahradia sa novymi
     zdrojmi jedla objavenymi prieskumnikmi.
6:   Zapamata sa zatiaľ najlepší zdroj jedla.
7: end

```

Ukážka 2: Pseudokód algoritmu ABC.

Nakoľko algoritmy ABC a QEGA sú opísané v rámci jedného článku, výsledky algoritmu ABC aplikovaného na FAP uvádzame v nasledovnej kapitole. Na testovanie nastavili autori maximálny počet generácií na 4000, veľkosť potomstva na 256, polovicu úľa tvorili prizerajúce sa včely a polovicu zamestnané včely, prieskumník bol jeden.

2.5.6 Modifikovaný genetický algoritmus inšpirovaný včelou kráľovnou (QEGA)

Štandardné evolučné algoritmy sú niekedy neúspešné pri nájdení globálneho optima v rámci limitovaného počtu krokov. Túto nevýhodu má odstrániť *modifikácia genetického algoritmu inšpirovaná včelou kráľovnou* (QEGA z angl. Queen-bee Evaluation Based On Genetic Algorithm) [13]. Populácia QEGA pozostáva z niekoľkých permutácií sekvencií. Na výpočet fitness bola použitá funkcia F1 ako v prípade vyššie opísaného genetického algoritmu (viac 2.5.2). Pseudokód algoritmu QEGA je uvedený v ukážke 3.

input: čas t , veľkosť populácie k , populácia P , pomer σ , pravdepodobnosť normalnej mutácie p_m , pravdepodobnosť silnej mutácie p'_m , včelia kráľovna I_q , vyberte včely I_m
output: najlepší jedinec

```
1: inicializuj  $P(t)$ 
2: ohodnot  $P(t)$ 
3: while nie sú splnené podmienky ukončenia do
4:    $t = t + 1$ 
5:   vyber  $P(t)$  z  $P(t-1)$  tak, aby  $P(t) = \{I_q(t-1), I_m(t-1)\}$ 
6:   rekombinuj  $P(t)$ 
7:   kríž jedincov
8:   for  $i = 0$  to  $k$  do
9:     if  $i \leq (\sigma * k)$  then
10:      mutuj jedince  $k$  s  $p_m$ 
11:     else
12:      mutuj jedince  $k$  s  $p'_m$ 
13:     end
14:   end
15:   ohodnot  $P(t)$ 
16: end
```

Ukážka 3: Pseudokód algoritmu QEGA.

Medzi QEGA a štandardným GA je niekoľko kľúčových rozdielov. Rodičia $P(t)$ vstupujúci do kríženia v GA pozostávajú z k jedincov vybraných algoritmom selekcie. Rodičia $P(t)$ v prípade QEGA však pozostávajú z $k/2$ kópií včelej kráľovnej $I_q(t-1)$, kde $q = \arg \max(F_\mu, 1 \leq \mu \leq k)$ a $k/2$ kópií včiel $I_m(t-1)$ vybraných pomocou nejakého selekčného algoritmu. Ďalší rozdiel predstavujú mutácie, kde v GA sú všetci jedinci mutovaní s malou pravdepodobnosťou p_m , ale v QEGA sa s touto malou pravdepodobnosťou mutuje len časť jedincov, ostatní jedinci sú mutovaní s veľkou pravdepodobnosťou p'_m . Pomer medzi p_m a p'_m sa značí ako σ . Vo všeobecnosti p_m je menej ako 0.1 a $p'_m > p_m$.

Najlepší jedinec v generácii je potom krížený s vybranými včelami. Fitness potomstva sa teda hlavne spolieha na operátor kríženia a na najlepšieho jedinca, čo zvyšuje pravdepodobnosť skorej konvergenzie. Zároveň silné mutácie p'_m zabezpečujú, že bude prehľadávaný dostatočne veľký problémový priestor. Autori použili na kríženie potomkov v kroku 7 *dvojbodové kríženie*, kde sa vyberú 2 náhodné body a chromozómy rodičov reprezentujúce riešenia sa rozdelia v týchto bodoch každý na 3 časti.

Prvý potomok je vytvorený z ľavej a pravej časti rodiča A a strednej časti rodiča B, druhý potomok je vytvorený z ľavej a pravej časti rodiča B a strednej časti rodiča A. Pre silnú mutáciu bol použitý algoritmus PALS [27], normálna mutácia bola vykonávaná náhodne.

Autori na záver testovali algoritmy ABC aj QEGA. Keďže sekvencie môžu byť voči sebe orientované rôzne, na výpočet ich prekrytia sa zobrala najlepšia hodnota ich prekrytia spomedzi doprednej orientácie i reverzného komplementu. Autori sa pri testovaní zamerali na meranie maximálnej hodnoty fitness, ktorú sa daným algoritmom podarilo získať a na počet získaných kontigov. Algoritmy boli testované so sekvenciami bez chýb, aj so sekvenciami s chybami vzniknutými v čase sekvenovania. V prípade sekvencií bez chýb bol prah prekrytia (kedy ešte možno hovoriť o tom, že sa sekvencie prekrývajú) nastavený na 30, pričom sekvencie mali dĺžku 130 – 1200 b. Algoritmy ABC a QEGA boli porovnávané s algoritmami SA, PALS a GA, s ktorými dosahovali porovnateľné výsledky. Algoritmus QEGA sa však ukázal byť v porovnaní s algoritmom ABC výkonnejší. Pre sekvencie s chybami bol nastavený menší prah prekrytia. ABC a QEGA boli v tomto prípade vyhodnotené oproti genetickému algoritmu (GA), genetickému algoritmu so simulovaným žiňaním (GA+SA) a genetickému algoritmu s horolezeckým algoritmom (GA+HC). Algoritmy ABC a QEGA dosahovali pre sekvencie s chybami oveľa lepšie výsledky ako ostatné porovnávané algoritmy.

3 Návrh

V tejto časti sme navrhli skladač DNA sekvencií, ktorý rieši FAP pretypovaním na TSP. TSP sa následne rieši pomocou nami navrhnutej modifikácie Paralelného optimalizačného modelu inšpirovaného imunitným systémom, ktorý bol predstavený v práci [37]. Model možno aplikovať na širšiu škálu optimalizačných problémov, preto opisujeme spôsob jeho aplikácie pri riešení TSP. Model [37], z ktorého sme vychádzali, sme za účelom implementovania skladača DNA sekvencií vybrali po dlhom hľadaní medzi biologicky inšpirovanými algoritmami, pričom tento model poskytoval jedny z najlepších výsledkov na poli riešenia problému obchodného cestujúceho a to z hľadiska kvality výsledných ciest, ale aj času spracovania a schopnosti riešenia veľkých TSP inštancií. Následne sme daný model vhodne modifikovali s cieľom zlepšenia časov behu programu, čím sme vytvorili *Paralelný imunitný algoritmus s pamäťou, klonovaním, výberom a distribuovanou operáciou zrenia protilátok* (skr. PIMCSA-DMO).

3.1 Návrh skladača DNA sekvencií

Zjednodušene môžeme povedať, že každý skladač DNA sekvencií vykonáva tri hlavné činnosti: na vstupe prijíma množinu krátkych sekvencií (1), ktoré následne vhodne usporiada (2) a z usporiadania vytvorí množinu kontigov, ktoré uloží na výstup (3). Náš skladač DNA sekvencií sme preto navrhli tak, aby bol *modulárny* a jeho jednotlivé moduly boli čo najmenej previazané, pričom každý hlavný modul vykonáva jednu z hlavných činností skladača DNA sekvencií. Náš skladač teda pozostáva z troch hlavných modulov:

- *Parser*: načíta množinu DNA sekvencií a vykoná transformáciu na problém obchodného cestujúceho
- *TSP Solver*: rieši problém obchodného cestujúceho s cieľom nájsť čo najlepšiu cestu
- *Connector*: na základe cesty v TSP pospája sekvencie do kontigov a uloží ich do výstupného súboru

Keďže skladanie DNA sekvencií je časovo náročný problém, rozhodli sme sa navrhnúť skladač s použitím *paralelného modelu*, ktorý by nám umožnil paralelné spracovanie na viacerých výpočtových uzloch. Paralelizmus sme sa rozhodli aplikovať práve na časovo najkritickejšie hľadanie cesty v probléme obchodného cestujúceho.

Nakoľko sa mnohým biologicky inšpirovaným algoritmom darí práve na poli riešenia problému obchodného cestujúceho, rozhodli sme sa hľadať biologicky inšpirovaný algoritmus, ktorý by bol schopný riešiť veľké inštancie problému obchodného cestujúceho (s viac ako 1000 mestami), ktorý by bol dostatočne presný a rýchly a zároveň by naň bolo možné aplikovať paralelizmus. Po dôslednom výskume v tejto oblasti sa nám ako najvhodnejší kandidát javil práve *Paralelný optimalizačný model inšpirovaný imunitným systémom* opísaný ďalej v kapitole návrhu. Výber sme realizovali približne z 30 rôznych biologicky inšpirovaných algoritmov, medzi ktorými boli aj rôzne modifikované genetické algoritmy, algoritmy inšpirované tokom vody, algoritmy inšpirované ekosystémom, či rôzne algoritmy inšpirované správaním sa roja, ako napríklad algoritmus inšpirovaný správaním sa netopierov, včiel, pavúkov, rôznymi druhmi vtákov a iné.

3.2 Načítanie sekvencií a transformácia FAP na TSP

Prvým krokom, ktorý skladač realizuje je *načítanie dát*. Rozhodli sme sa načítavať sekvencie vo formáte *FASTA*, ktorý je používaný mnohými skladačmi DNA sekvencií a je ľahko čitateľný aj pre človeka. Načítavanie sekvencií z formátu *FASTA* je vďaka jeho vlastnostiam, ktoré sme opísali v kapitole 2.2.6 ľahko implementovateľné.

Sekvencie sa načítavajú v rámci modulu *Parser*. Základnou úlohou tohto modulu je transformovať FAP na TSP. Táto transformácia sa deje tak, že najskôr načítané sekvencie porovnáme každú s každou a hľadáme maximálnu veľkosť prekrytia medzi nimi. Keďže sekvencie sú čítané vždy v doprednom smere, nemusíme porovnávať oba ich konce, ale stačí porovnať koniec prvej sekvencie so začiatkom druhej sekvencie (a naopak), pričom predpokladáme, že žiadne dve sekvencie nie sú zhodné. V praxi preto buď treba sekvencie predspracovať a duplicitné sekvencie odstrániť, alebo je možné na filtrovanie duplicitných sekvencií použiť priamo funkcionálnu skladača, ktorú treba zapnúť v jeho nastaveniach. Implicitne je však funkcionálna filtrovania duplicitných sekvencií vypnutá, aby sa šetril čas potrebný na spracovanie sekvencií. Časová náročnosť hľadania prekrytí medzi všetkými možnými dvojicami sekvencií je teda $O(M \cdot N^2)$, kde N predstavuje počet sekvencií a M priemernú dĺžku sekvencie.

Prekrytie sekvencií počítame pomocou operácie *overlap* nasledovným spôsobom. Z množiny sekvencií vyberieme dve sekvencie A a B . Obe sekvencie zarovnáme zľava, pričom sekvenciu B posunieme vzhľadom na sekvenciu A o 1 nukleotid doprava (keďže sekvencie nemôžu byť rovnaké) a porovnáme ich prekrývajúce sa časti. Ak sa tieto časti nezhodujú, sekvenciu B posunieme vzhľadom na sekvenciu A o ďalší nukleotid vpravo a znova zisťujeme, či sa ich prekrývajúce sa časti zhodujú. Takto pokračujeme, až kým nezistíme zhodu, z ktorej potom, na základe dĺžok sekvencií a veľkosti posunu sekvencie B oproti sekvencii A , zistíme hodnotu ich maximálneho prekrytia. Ukážku tohto procesu zachytáva obrázok 16.

ATGTTTCAGCACTTTTCAGAGACGTACTACGTA
TTCAGCACTTTTCAGAGACGTACTACGTATTGAC

ATGTTTCAGCACTTTTCAGAGACGTACTACGTA
TTCAGCACTTTTCAGAGACGTACTACGTATTGAC

ATGTTTCAGCACTTTTCAGAGACGTACTACGTA
TTTCAGCACTTTTCAGAGACGTACTACGTATTGAC

ATGTTTCAGCACTTTTCAGAGACGTACTACGTA
TTCAGCACTTTTCAGAGACGTACTACGTAATTGAC

Obr. 16: Na obrázku je znázornený postup výpočtu prekrytia sekvencie A (horná sekvencia) so sekvenciou B (spodná sekvencia) v rámci operácie overlap. Obe sekvencie majú veľkosť 32 nukleotidov. Sekvenciu B posunieme vzhľadom na sekvenciu A o 1 nukleotid vpravo a porovnáme ich prekrývajúce časti. Ak sa nezhodujú pokračujeme s posúvaním až kým nenarazíme na zhodu. V tomto prípade veľkosť prekrytia predstavuje 28 nukleotidov.

Zistením hodnôt všetkých prekrytí teda získame *maticu prekrytí* O rozmerov $N \times N$, kde prvok $O(a, b)$ predstavuje veľkosť maximálneho prekrytia medzi sekvenciou na pozícii a a sekvenciou na pozícii b v súbore so sekvenciami, čo môžeme zapísať ako $O(a, b) = \text{overlap}(\text{seq}[a], \text{seq}[b])$. Takto vzniknutú maticu ale nemožno považovať za inštanciu TSP, nakoľko by sme dĺžku cesty museli maximalizovať a nie minimalizovať. V rámci zisťovania prekrytí medzi všetkými dvojicami sekvencií teda hľadáme aj veľkosť najväčšieho prekrytia medzi ľubovoľnými dvoma sekvenciami: $\text{overlap_max} = \max_{m, n} O(m, n)$. Z matice prekrytí následne vytvoríme maticu C reprezentujúcu vzdialenosti medzi mestami, ktoré predstavujú jednotlivé DNA sekvencie, podľa vzorca $C(m, n) = \text{overlap_max} - O(m, n)$. Matica C však predstavuje *asymetrickú inštanciu problému obchodného cestujúceho*, nakoľko prekrytie medzi sekvenciou A a B vo všeobecnosti nemusí byť rovnako veľké ako prekrytie medzi sekvenciou B a A , teda $\text{overlap}(A, B) \neq \text{overlap}(B, A)$.

Asymetrické TSP inštancie sú náročnejšie na riešenie a existuje len málo algoritmov na ich riešenie. Je to spôsobené i skutočnosťou, že väčšina algoritmov na riešenie TSP sa sústreďuje na riešenie *euklidovských inštancií TSP*, ktoré, okrem symetrickosti, spĺňajú aj trojuholníkovú nerovnosť medzi ľubovoľnými tromi mestami m_1, m_2 a m_3 : $\text{cost}(m_1, m_2) \leq \text{cost}(m_1, m_3) + \text{cost}(m_2, m_3)$. Asymetrickú inštanciu TSP ale vieme transformovať na symetrickú inštanciu tak, ako uvádzajú autori v článku [38]. Veľkosť výsledného problému je ale oproti pôvodnej asymetrickej verzii dvojnásobná.

Transformácia z asymetrickej inštancie na symetrickú sa vykoná nasledovnými krokmi. Vytvoríme maticu C' rozmerov $M \times M$, kde $M = 2 \cdot N$. Na diagonálu asymetrickej matice C vložíme nízku cenu cesty, označenú ako $-\infty$. Následne do ľavého dolného rohu matice C' skopírujeme maticu C a jej inverznú podobu do pravého horného rohu matice C' . Ostatné polia predstavujú neplatné prechody medzi jednotlivými mestami, preto do nich vložíme vysokú cenu, ktorú autori článku [38] navrhujú nastaviť na $2 \cdot M \cdot \text{overlap_max}$. Touto úpravou získame *symetrickú maticu vzdialeností* predstavujúcu symetrickú inštanciu TSP. Zoštvornásobením veľkosti matice dôjde aj k duplikácii

miest z asymetrickej inštancie. Mestá A a A' v symetrickej inštancii teda reprezentujú rovnaké mesto A v asymetrickej inštancii. Preto chceme vždy z mesta X prejsť do mesta X' , pre každé mesto X . Toto síce nevieme nijako vynútiť, ale nastavením nízkej ceny medzi mestom X a X' podstatne zvýhodníme cesty, kde susedné prechody medzi dvoma mestami predstavujú dvojice (X, X') alebo (X', X) . Podobne, neplatné prechody zakázať nevieme, ale nastavením vysokej ceny pre neplatné prechody znevýhodníme cesty, ktoré ich obsahujú. Potrebnú transformáciu na symetrickú verziu TSP zachytáva obrázok 17.

	A	B	C
A	0	1	2
B	4	0	3
C	5	6	0

→

	A	B	C	A'	B'	C'
A	x	x	x	$-\infty$	4	5
B	x	x	x	1	$-\infty$	6
C	x	x	x	2	3	$-\infty$
A'	$-\infty$	1	2	x	x	x
B'	4	$-\infty$	3	x	x	x
C'	5	6	$-\infty$	x	x	x

Obr. 17: Obrázok zachytáva asymetrickú vzdialenosnú maticu a jej transformáciu na symetrickú maticu. Neplatné prechody sú označené ako x a predstavujú nejaké veľké číslo. Nastavením diagonály matice C na $-\infty$ zvýhodňujeme cesty ako $AA'BB'CC'$, alebo $AB'BC'CA'$, ale cesty ako $A'BC'AB'C$ budú znevýhodnené. Neplatný prechod zase veľmi znevýhodní cesty ako $ABCA'B'C'$ alebo $AB'BCC'A'$. Najlepšou cestou je cesta $AA'BB'CC'$ predstavujúca cestu ABC v asymetrickej verzii s cenou 9.

Symetrická matica C' má síce výhodu rýchleho prístupu k jej prvkom v čase $O(1)$, ale jej udržiavanie, kvôli jej štvornásobnej veľkosti oproti asymetrickej matici C , by nebolo pamäťovo efektívne. V našom skladači preto udržiavame v pamäti len upravenú maticu C . Túto maticu sme upravili tak, ako sme už spomínali, nastavením diagonálny na nízku hodnotu, ktorú sme v našom skladači zvolili ako -1 . Za týmto účelom sme ale museli modifikovať operáciu $cost(a, b)$ vracajúcu cenu cesty medzi mestami a a b (v ponímaní skladania DNA sekvencií, inverznú veľkosť prekrytia medzi sekvenciou a a sekvenciou b) tak, ako znázorňuje ukážka 4. V prípade matice C' by operácia $cost(a, b)$ vracala jednoducho prvok $C'(a, b)$.

```

if (a >= dimension(C) and b < dimension(C))
    return C[a - dimension(C), b]
else if (a < dimension(C) and b >= dimension(C))
    return C[b - dimension(C), a]
else
    return illegal_edge_cost

```

Ukážka 4: Operácia $cost(a, b)$ vracajúca cenu cesty medzi mestami a a b .

Po získaní symetrickej inštancie TSP (či už priamo, pomocou matice C' , alebo nepriamo, pomocou matice C a modifikovanej operácie $cost(a, b)$) sme schopní riešiť problém obchodného cestujúceho ľubovoľným algoritmom určeným na riešenie symetrických TSP inštancií. Medzi takéto algoritmy patrí aj Paralelný optimalizačný model inšpirovaný imunitným systémom použitý v našej práci.

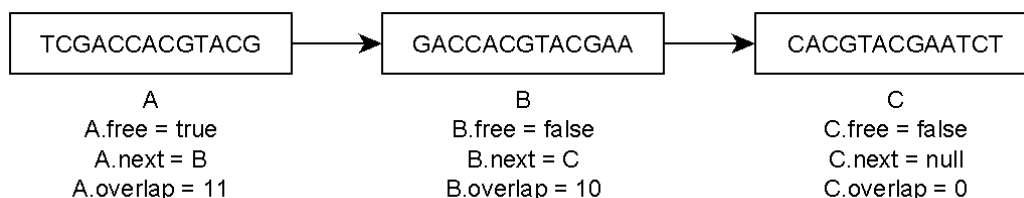
3.2.1 Redukcia dimenzionality problému obchodného cestujúceho

Keďže výstupom dnes najrozšírenejších DNA sekvencerov tretej generácie je veľké množstvo krátkych DNA sekvencií a každá sekvencia predstavuje jedno mesto v asymetrickom probléme obchodného cestujúceho, výsledná symetrická inštancia TSP predstavuje riešenie problému až s $2 \cdot n$ mestami, kde n predstavuje počet DNA sekvencií. Mnohé dvojice sekvencií sa však prekrývajú v tak veľkom množstve nukleotidov, že takmer s istotou môžeme povedať, že sa prekrývajú aj v skutočnosti (ich prekrytie môže presahovať 90% ich dĺžky). Takéto sekvencie môžeme navzájom pospájať, čím môžeme výrazne zredukovať počet sekvencií a teda aj veľkosť riešenej TSP inštancie. Za týmto účelom sme implementovali modul *Compactor*, ktorý je volaný modulom *Parser* po načítaní sekvencií z FASTA súboru, pred tým, ako sa vytvára matica prekrytí a vykoná sa s ňou súvisiaca transformácia na TSP.

Modul *Compactor* prijíma pole načítaných DNA sekvencií a tzv. *kritický pomer* R , ktorý predstavuje pomer prekrytia ku dĺžke kratšej z porovnávaných sekvencií, ktorý musí byť dosiahnutý na to, aby došlo k spojeniu práve porovnávaných sekvencií. Pomer prekrytia dvoch sekvencií a a b možno teda vypočítať ako $r = \text{overlap}(a, b) / \min(\text{len}(a), \text{len}(b))$. Modul uchováva pre každú sekvenciu inštanciu štruktúry *read_data*, ktorá má tieto parametre:

- *next*: identifikuje sekvenciu, ktorá nasleduje za sekvenciou prislúchajúcou k danej inštancii štruktúry *read_data* (ďalej *aktuálnou sekvenciou*), defaultne nastavené na *null*
- *overlap*: predstavuje veľkosť prekrytia medzi aktuálnou sekvenciou a sekvenciou identifikovanou pomocou *next*
- *free*: indikuje to, že aktuálna sekvencia nie je pripojená ku žiadnej inej sekvencii, defaultne nastavené na *true*

Najskôr zistíme veľkosť prekrytia $\text{overlap}(a, b)$ medzi každými dvoma sekvenciami a a b a vypočítame ich pomer prekrytia r podľa uvedeného vzorca. Ak $r > R$, tak $a.\text{next}$ nastavíme na b , $a.\text{overlap}$ nastavíme na $\text{overlap}(a, b)$ a $a.\text{free}$ nastavíme na *false*. Na záver pospájame sekvencie do *predčasných kontigov* tak, že spájanie začneme vždy od sekvencií, ktorých parameter *free* je ešte stále nastavený na *true* a teda neboli pripojené k nijakým iným sekvenciám. Samotné spájanie realizujeme na základe prekrytí, ktoré sú uložené v parametri *overlap* podobne, ako v prípade finálneho spájania sekvencií do kontigov v module Connector, ktoré opisujeme v kapitole 3.4. Príklad zrefazovania DNA sekvencií a ich spojenie do predčasných kontigov znázorňuje obrázok 18. Modul *Compactor* vracia vytvorenú množinu *predčasných kontigov*, ktoré predstavujú novú množinu sekvencií, s ktorou pracuje modul *Parser*. Ten opätovne nájde prekrytia medzi nimi, vytvorí maticu C a tú transformuje na symetrickú verziu TSP.



Obr. 18: Obrázok znázorňuje spôsob spájania DNA sekvencií do predčasných kontigov v moduli *Compactor*. Tri sekvencie A, B a C zdieľajú veľkú časť spoločných nukleotidov. Keďže sekvencia A má parameter *free* nastavený na *true*, spájanie začneme od nej. Na základe veľkostí prekrytí v parametroch *overlap* vieme určiť výsledné spojenie týchto troch sekvencií na TCGACCACGTACGAATCT, čím sme zredukovali veľkosť TSP inštancie o 2 mestá.

3.3 Riešenie problému obchodného cestujúceho

Riešeniu problému obchodného cestujúceho (a teda správneho zoradeniu sekvencií) sa venuje modul *Solver*. Tento modul sme navrhli tak, aby na vstupe prijímal symetrickú TSP inštanciu a na výstupe produkoval postupnosť prechodov medzi jednotlivými uzlami (mestami) tejto inštancie – cestu v TSP. O samotné riešenie TSP sa stará *Paralelný optimalizačný model inšpirovaný imunitným systémom*.

3.3.1 Paralelný optimalizačný model inšpirovaný imunitným systémom

Autori Yuato Qi, Fang Liu a Licheng Jiao predstavili *Paralelný optimalizačný model inšpirovaný imunitným systémom* (angl. A Parallel Artificial Immune Model for Optimization) v zmysle *Tower Master-Slave Modelu* (skr. TMSM) [37]. *Imunitný systém* obsahuje množstvo užitočných informácií vrátane rozpoznávania vzorov, učenia, pamäte a inherentného distribuovaného paralelizmu. Práve kvôli týmto vlastnostiam sa začal výskum na poli umelých imunitných systémov (skr. AIS z angl. Artificial Immune Systems). S nárastom veľkosti problému sa ale aj imunitné systémy stávajú neefektívne. Autori preto predstavili *Tower Master-Slave Model* a vytvorili nad ním *Paralelný imunitný algoritmus s pamäťou, klonovaním a výberom* (skr. PIMCSA z angl. parallel immune memory clonal selection algorithm), ktorý sme vhodne modifikovali a tak sme vytvorili *Paralelný imunitný algoritmus s pamäťou, klonovaním, výberom a distribuovanou operáciou zrenia protilátok* (skr. PIMCSA-DMO z angl. parallel immune memory clonal selection algorithm with distributed maturation operation).

TMSM je dvojúrovňový model imunitného systému s hrubozrnným paralelizmom, distribuovanou imunitnou odpoveďou a distribuovanou imunitnou pamäťou. V rámci PIMCSA, a teda aj PIMCSA-DMO, sa z populácie protilátok extrahujú vakcíny a tie sa posielajú medzi populáciami, čo je výhodnejšie ako posielanie celých protilátok. *Protilátka* predstavuje v pojmoch AIS člena populácie a vakcína informáciu extrahovanú z nejakého člena populácie, pomocou ktorej možno mutovať inú protilátku. Pri návrhu paralelného imunitného systému sa nesústredíme len na topológiu populácií, ich rozdelenie a komunikačnú stratégiu medzi nimi, ale sústredíme sa aj na to, ako simulovať dané

vlastnosti biologických imunitných systémov. TMSM je preto zložený z jednej pamäťovej populácie M a niekoľkých podpopulácií protilátok P_i .

Pamäťová populácia $M = (A_1, A_2, \dots, A_m)$ tvorí vrchnú vrstvu TMSM a tvorí ju m pamäťových protilátok. Každá protilátka A_i v populácii M je mapovaná na jednu podpopuláciu protilátok $P_i = (A_1^i, A_2^i, \dots, A_n^i)$. Spodná vrstva TMSM preto pozostáva z m podpopulácií veľkosti n . Diverzita podpopulácií potom vedie k rôznorodosti pamäťových protilátok.

Algoritmus PIMCSA-DMO vykonáva úlohy spojené s plánovaním úloh a komunikáciou medzi jednotlivými populáciami, pričom pamäťová populácia je *riadiacou populáciou*. Kvôli dvojúrovňovému charakteru TMSM sa algoritmus PIMCSA-DMO, tak ako algoritmus PIMCSA, delí na dve časti: *imunitný algoritmus nad pamäťovou populáciou* (skr. *MIA* z angl. Immune Algorithm on Memory Population) a *imunitný algoritmus nad populáciami* (skr. *PIA* z angl. Immune Algorithm on Subpopulations). *Afinita* protilátky vyjadruje silu interakcie jedného väzobného miesta protilátky s jedným antigénom, pričom *antigén* je látka, ktorú dokáže imunitný systém rozpoznať a zareagovať na ňu. Afinitu teda používame ako mieru účinnosti (sily) protilátky. Afinitu protilátky A označujeme ako *affinity(A)*, pričom vyžadujeme aby bola reprezentovaná *kladným číslom*. Algoritmy *MIA* a *PIA* sú potom popísané nasledovne:

MIA

```
ustanov podmienky ukončenia a nastav  $r = 0$ 
inicializuj pamäťovú populáciu  $M(r)$  a vypocitaj afinitu protilátok
nastav množinu vakcín  $V(r) = \emptyset$ 
while nie su splnene podmienky ukončenia do
  obdrž  $A'_i$  z každej  $P_i$ , ak  $\text{affinity}(A'_i) > \text{affinity}(A_i)$ , nahrad  $A_i$  s  $A'_i$ 
  vykonaj operáciu dynamickej extrakcie vakcín
     $V(R+1) = \text{extraction}(M(r+1))$ 
  vykonaj operáciu odoslania vakcín:
     $\text{dispatch}(P_1, P_2, \dots, P_m)$ 
   $r = r + 1$ 
end while
vypis výsledky a odosli ukončovaci signál populáciám  $P_i$ 
```

Ukážka 5: Pseudokód algoritmu *MIA*.

Algoritmus *MIA* sleduje kroky ukážky 5. Oproti algoritmu *MIA*, ktorý je súčasťou algoritmu PIMCSA, operátor *MIA* v PIMCSA-DMO vypúšťa *operáciu zrenia pamäťových protilátok*, ktorá mala byť aplikovaná nad každou protilátkou pamäťovej populácie, a ktorá mala nasledovať pred operáciou dynamickej extrakcie vakcín, pričom sme túto operáciu premiestnili do algoritmu *PIA*, vďaka čomu dôjde k značnému zrýchleniu celého programu. Operácie použité v algoritme *MIA* sú definované nasledovne:

- *Operátor dynamickej extrakcie vakcín*: Aktuálnu populáciu $M(r+1)$ rozdelíme na 2 časti,

extrakčnú množinu $M_1(r+1)$ a injekčnú množinu $M_2(r+1)$. Nech $k \in \mathbb{N}^+$ je najmenšie číslo menšie ako $m/2$, potom $V(r+1) = M_1(r+1)$ pozostáva z k náhodne vybraných protilátok z $M(r+1)$ a $M_2(r+1)$ zo zvyšných protilátok. Vakcinačné stratégie a spôsob extrakcie vakcín je potrebné navrhnuť pre rôzne optimalizačné problémy rôzne.

- *Operátor odoslania vakcín*: Vyberieme vakcínu v_i z množiny vakcín $V(r+1)$ a pamäťovú protilátku $A_i \in M_2(r+1)$. Podpopulácii P_i mapovanej na A_i potom odošleme vakcínu v_i .

PIA

```

nastav  $t = 0$  a  $\text{halt} = \text{FALSE}$ 
nahodne inicializuj  $P_i(t) = (A_1^i(t), A_2^i(t), \dots, A_n^i(t))$  a vypocitaj afinity protilátok
najlepsi protilátku z  $P_i(t)$  oznac ako  $A_i'(t)$  a nastav premennu  $\text{current\_best}$  na
     $\text{affinity}(A_i'(t))$ 
odosli  $A_i'(t)$  pamatovej populácii  $M$ 
nastav aktualnu množinu vakcin  $v(t) = \emptyset$ 
while( $\text{halt} \neq \text{TRUE}$ ) do
    skus obdržat ukoncovaci signal od  $M$ , v prípade uspechu prirad  $\text{halt} = \text{TRUE}$ 
    skus obdržat vakcinu z  $M$ , v prípade uspechu nahrad staru vakcinu vo  $v(t)$ 
    vykonaj operáciu klonovania  $P_i'(t) = \text{clone}(P_i(t))$ 
    vykonaj imunitnu geneticku operáciu:  $P_i''(t) = \text{immune\_operation}(P_i'(t))$ 
    vykonaj operáciu vyberu klonov  $P_i(t+1) = \text{selection}(P_i''(t))$ 
    najdi najlepsi protilátku v  $P_i(t+1)$  a oznac ju ako  $A_i'(t+1)$ 
    vykonaj operáciu zrenia protilátky  $A_i'(t+1) = \text{maturation}(A_i'(t+1))$ 
    ak  $\text{affinity}(A_i'(t+1)) > \text{current\_best}$ , nastav  $\text{current\_best}$  na  $\text{affinity}(A_i'(t+1))$ 
    odosli  $A_i'(t+1)$  pamatovej populácii  $M$ 
     $t = t + 1$ 
end while

```

Ukážka 6: Pseudokód algoritmu PIA.

Algoritmus PIA sleduje kroky ukážky 6 a rovnako, ako algoritmus MIA, pozostáva z niekoľkých operátorov:

- *Operátor klonovania*: Spôsobí niekoľkonásobné skopírovanie každej protilátky v podpopulácii. Počet klonov autori [37] definujú ako číslo q_j , ktoré závisí od $\text{affinity}(A_j(t))$, no možno ho nastaviť aj ako konštantu, rovnakú pre každú protilátku.
- *Imunitný genetický operátor*: Aplikuje vakcináciu (ktorú v pojmoch evolučných algoritmov môžeme chápať ako *kríženie*) a zároveň *mutáciu* na protilátkach, ktoré sú z podpopulácie vybrané náhodne s rovnakou pravdepodobnosťou. Na rozdiel od algoritmu PIA v pôvodnej verzii PIMCSA, je imunitný genetický operátor aplikovaný nad zvolenou protilátkou s použitím vakcín len s danou pravdepodobnosťou p_{vacc} , v opačnom prípade je tento operátor aplikovaný iba nad samotnými protilátkami. Z týchto protilátok môže byť potrebné získať vakcinačnú informáciu podobne, ako v prípade *operácie dynamickej extrakcie vakcín* v MIA. Týmto krokom zabezpečíme to, že kríženie a mutácia bude prebiehať nie len medzi podpopuláciami, ale aj

medzi jednotlivými protilátkami danej podpopulácie P_i . Na pravdepodobnosť p_{vacc} sa nekladú žiadne obmedzenia a je možné ju nastaviť na ľubovoľné hodnoty z intervalu $\langle 0, 1 \rangle$, vrátane hodnoty 0, kedy bude kríženie a mutácia prebiehať iba v rámci jednotlivých podpopulácií, alebo 1, kedy bude prebiehať kríženie s mutáciou iba medzi podpopuláciami navzájom.

- *Operátor výberu klonov*: Z každej množiny klonov, ktorá prislúcha nejakej pôvodnej protilátke $A_j(t)$ sa vyberie najlepší klon a ním sa nahradí príslušná protilátka v podpopulácii $P(t + 1)$.
- *Operátor zrenia protilátky*: Aplikuje mechanizmus lokálneho prehľadávania na najlepšiu protilátku v danej podpopulácii pred tým, ako bude odoslaná pamäťovej populácii. Tento operátor sme v rámci algoritmu PIMCSA-DMO presunuli z MIA (kde bol pôvodne prítomný) do PIA, čím dôjde k zrýchleniu behu programov, ktoré využívajú algoritmus PIMCSA-DMO a teda aj k zrýchleniu nášho skladáča DNA sekvencií. Keďže veľa algoritmov lokálneho prehľadávania vyžaduje značné množstvo času, ich aplikácia v každej iterácii MIA nad každou protilátkou pamäťovej populácie by nebola efektívna.

3.3.2 Aplikácia PIMCSA-DMO na riešenie TSP

TMSM model a algoritmus PIMCSA možno použiť na pestrú škálu optimalizačných problémov. Autori článku [37] ho úspešne aplikovali aj na euklidovské inštancie problému obchodného cestujúceho. Výhodou týchto inštancií je, že nie je potrebné mať v pamäti celú maticu vzdialeností, nakoľko vzdialenosť medzi dvoma mestami možno vypočítať na základe ich súradníc. Keďže algoritmus PIMCSA-DMO je vylepšením algoritmu PIMCSA, možno i tento algoritmus aplikovať na riešenie TSP. Pre potreby skladania DNA sekvencií sme potrebovali algoritmus aplikovať na neeuklidovskú symetrickú inštanciu TSP. Jednotlivé cesty v TSP vystupujú v PIMCSA-DMO ako *protilátky*. Afinitu sme definovali ako $affinity(T) = T_MAX - (cost(T) + cities)$, kde protilátka T predstavuje už spomínanú cestu v probléme obchodného cestujúceho, $cost(T)$ celkovú cenu tejto cesty a T_MAX vyjadruje najväčšiu možnú hodnotu dátového typu, ktorým je cena cesty reprezentovaná. Pomocou $cities$ kompenzujeme možné záporné hodnoty niektorých ciest, ktoré môžu vzniknúť kvôli prítomnosti -1 v matici C reprezentujúcej TSP. Ako operátor zrenia protilátok sme zvolili *Iterovanú Lin-Kernighanovu heuristiku*, ktorú možno i napriek tomu, že sa používa pre euklidovské TSP inštancie, aplikovať aj na neeuklidovské inštancie, o čom nás presvedčila i korešpondencia s jej autorom, profesorom Davidom S. Johnsonom. Ako vakcinačnú stratégiu sme použili *Vylepšený Inver-over operátor*, ktorý v sebe zahŕňa prvky kríženia i mutácie. Kvôli tomuto operátoru je však nutné ako vakcínu chápať celú protilátku z populácie M , ktorú odošleme zvoleným podpopuláciám protilátok P_i . Vylepšený Inver-over operátor potom na vakcináciu (čiže kríženie) protilátky v danej podpopulácii použije jednu z vakcín. Viac o Vylepšenom Inver-over operátore, ako aj o Lin-Kernighanovej heuristike uvádzame v nasledujúcich častiach tejto kapitoly. Algoritmus PIMCSA-DMO bude ukončený, ak v N po sebe nasledujúcich cykloch algoritmu MIA nedôjde k zmene najlepšej protilátky. Veľkosť

jednotlivých podpopulácií sme defaultne zvolili na 20, veľkosť vakcinačnej množiny na 8, pričom každý člen podpopulácie sa počas operácie *clone* klonuje 5 krát. Veľkosť pamäťovej populácie závisí od počtu podpopulácií. Všetky nastavenia možno meniť v konfiguračnom súbore *settings.txt*, ktorého jednotlivé položky opisujeme v prílohe C.

3.3.3 Lin-Kernighanova heuristika

Lin-Kernighanova heuristika patrí k jednej z najúspešnejších heuristik určených na riešenie problému obchodného cestujúceho, ako z hľadiska kvality výsledných ciest, tak aj z hľadiska rýchlosti nájdenia výsledného riešenia. Podrobný opis Lin-Kernighanovej heuristiky presahuje rozsah tejto diplomovej práce, no jej detailný opis a návod pre jej implementáciu uvádza kapitola *Finding Tours in the TSP* z knihy *The Traveling Salesman Problem* [3], z ktorej sme vychádzali pri jej implementácii¹⁰. Uvádzame preto len stručný opis základných vlastností tejto heuristiky.

Každá optimálna cesta v TSP spĺňa pre každé 2 mestá c_p, c_q také, že $0 \leq p < q < n$ nasledovnú podmienku: $cost(c_{p-1}, c_p) + cost(c_q, c_{q+1}) \leq cost(c_{p-1}, c_q) + cost(c_p, c_{q+1})$. Taký pár (p, q) , ktorý danú podmienku nespĺňa sa nazýva *pretínajúci pár* (angl. *intersectioning pair*). Dôležitý je fakt, že ak sa (p, q) v ceste $(c_0, \dots, c_{p-1}, c_p, \dots, c_q, c_{q+1}, c_{n-1})$ pretínajú, tento pár možno opraviť pomocou inverzie subsekvencie $(c_p, \dots, c_q)$, čím získame cestu: $(c_0, \dots, c_{p-1}, c_q, \dots, c_p, c_{q+1}, c_{n-1})$. Táto operácia sa nazýva *flip* (p, q) . Cesta v TSP sa nazýva λ -*optimal* ak nie je možné získať lepšiu cestu nahradením jej ľubovoľných λ hrán inými λ hranami. Každý λ -*opt* presun hrán vedúci k dosiahnutiu λ -*optimal* cesty potom možno realizovať sekvenciou flipov.

Lin a Kernighan vyvinuli algoritmus, ktorý sa niekedy nazýva aj ako *variabilný λ -opt*, no je známejší pod pojmom Lin-Kernighanova heuristika. Jadrom algoritmu je efektívna metóda hľadania sekvencie flipov, ktoré majú šancu na zlepšenie ceny výslednej cesty. Ak je hľadanie úspešné a našla sa zlepšujúca sekvencia flipov, táto sekvencia sa vykoná a začne sa s novým hľadaním, inak sa nájdená sekvencia flipov pred pokračovaním s novým hľadaním zamietne, pričom si dávame pozor na to, aby sme znova neskúšali neúspešné sekvencie flipov. Hľadanie končí, keď každý štartovací bod hľadania vedie k nezlepšujúcim sekvenciám flipov.

Predpokladajme TSP s n mestami. Nech T je cesta v TSP a *base* ľubovoľne zvolené mesto. Budeme vy-

¹⁰Pri implementácii Lin-Kernighanovej heuristiky sme okrem [3] vychádzali aj z jej neúplnej kostry v jazyku Java dostupnej na stránke <https://www.zib.de/borndorfer/Homepage/Documents/WS11/jLK.tgz>. Táto kostra bola súčasťou výučbového kurzu Computational Integer Programming na Zuse-Institute Berlin, pričom jeho študenti mali doimplementovať kľúčové kroky Lin-Kernighanovej heuristiky, najmä jej rekurzívny krok *step*, čo vyžadovalo značnú znalosť tejto heuristiky. Funkcia *step* zabezpečuje prehľadávanie problémového priestoru s cieľom nájsť zlepšujúce *flipy* a je bližšie opísaná v [3]. Tento kód nám pomohol vytvoriť si lepší obraz o tom, ako Lin-kernighanova heuristika funguje. Vychádzajúc z knihy [3] sme potom boli schopní heuristiku implementovať. Samotná implementácia heuristiky predstavovala asi najväčšiu výzvu tohto diplomového projektu.

tvárať sekvenciu operácií *flip* a aktuálnu cestu získanú aplikovaním danej sekvencie flipov označíme ako *current_tour*. Nech $prev(v)$ označuje mesto priamo pred mestom v a $next(v)$ mesto priamo za mestom v , pričom tieto operácie sú jednoznačné, nakoľko cesta v TSP je orientovaná. Jediné flipy, ktoré uvažujeme, sú flipy tvaru $flip(next(base), probe)$, kde vrchol *probe* je rozdielny od *base*, $next(base)$ a $prev(base)$. Takéto flipy nahradia hrany $(base, next(base))$ a $(probe, next(probe))$ hranami $(next(base), next(probe))$ a $(base, probe)$. Cesta *current_tour* bude takýmto flipom zlepšená, len ak $cost(base, next(base)) + cost(probe, next(probe)) > cost(next(base), next(probe)) + cost(base, probe)$.

Lin-Kernighanova heuristika však pred hľadaním flipov, ktoré spĺňajú vyššie uvedenú podmienku uprednostňuje flipy, pre ktoré platí $cost(base, next(base)) - cost(next(base), next(probe)) > 0$. Ide o *greedy prístup*, ktorý sa pokúša v danom čase vylepšiť len jednu hranu cesty. Nech *delta* je premenná na začiatku behu algoritmu nastavená na 0, ktorá sa inkrementuje o $cost(base, next(base)) - cost(next(base), next(probe)) + cost(probe, next(probe)) - cost(probe, base)$ po každom aplikovaní $flip(next(base), probe)$. *Delta* predstavuje množstvo lokálneho zlepšenia, ktoré sme zatiaľ s danou sekvenciou flipov získali. Cena *current_tour* potom môže byť vypočítaná ako $cost(T) - delta$. Vrchol *probe* voláme *sľubným vrcholom* (angl. promising vertex), ak spĺňa podmienku: $delta + cost(base, next(base)) - cost(next(base), next(probe)) > 0$. Prehľadávanie *lk_search* potom možno zjednodušiť chápať v zmysle ukážky 7.

```

sekvencia_flipov = ∅
delta = 0
while existuje sľubny vrchol do
    probe ← vyber sľubny vrchol
    delta = delta + cost(base, next(base)) - cost(next(base), next(probe)) +
        cost(probe, next(probe)) - cost(probe, base)
    sekvencia_flipov = sekvencia_flipov ∪ flip(next(base), probe)
end while
return sekvencia_flipov

```

Ukážka 7: Pseudokód prehľadávania *lk_search*.

Ak získame lacnejšiu cestu, danú sekvenciu flipov uložíme, ale s prehľadávaním pokračujeme, aby sme zistili, či nie je možné dosiahnuť ešte lepšiu sekvenciu flipov. Samotná heuristika nakoniec sleduje kroky ukážky 8.

Aplikovaním $flip(x, y)$ znova označíme vrcholy x a y , nakoľko je možné, že vďaka aplikovaniu sekvencie flipov niektoré vrcholy z tejto sekvencie môžu pri opätovnom použití v budúcich flipoch znova poskytovať zlepšujúce riešenia.

```

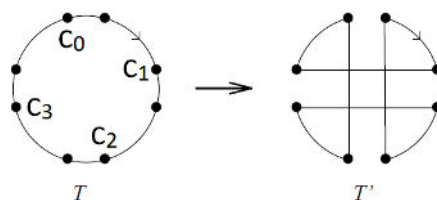
inicializuj lk_tour ako T
oznac vsetky vrcholy (mesta)
while existuje nejaký označený vrchol do
    vyber označený vrchol v
    lk_search(v, lk_tour)
    if bola najdená zlepšujúca sekvencia flipov then
        while sekvencia flipov nie je prázdna do
            flip(x, y) ← ďalší flip v sekvencii
            aplikuj flip(x, y) na lk_tour
            oznac vrcholy x a y
            vymaz flip(x, y) zo sekvencie flipov
        end while
    else
        zrus oznacenie v
    end if
end while
return lk_tour

```

Ukážka 8: Pseudokód Lin-Kernighanovej heuristiky (funkcie `lin-kernighan()`).

3.3.4 Iterovaná Lin-Kernighanova heuristika

Iterovaná Lin-Kernighanova heuristika je vylepšením štandardnej Lin-Kernighanovej heuristiky, ktoré navrhol David S. Johnson v článku *Local Optimization and the TSP* [23]. Ide o cyklické spúšťanie Lin-Kernighanovej heuristiky nad rovnakou cestou, ktorá sa pred každým prechodom cyklom nejako mutuje. Vhodnou mutáciou je aplikácia tzv. *double-bridge presunu*, ktorý Lin-Kernighanova heuristika nevie vykonať žiadnou sekvenciou flipov. Tento presun je špeciálnym prípadom tzv. *k-exchange presunu* (angl. *k-exchange move*) pre $k = 4$. *K-exchange* presun spočíva v náhodnom výbere k miest v poradí, v akom sa nachádzajú v danej TSP ceste. Následne sa odstránia všetky hrany $(c_{(i+2) \bmod k}, next(c_{(i+2) \bmod k}))$ a nahradia sa hranami $(c_{(i+2) \bmod k}, next(c_i))$ pre $i = 0 \dots k - 1$. Double-bridge presun znázorňuje obrázok 19 a algoritmus Iterovanej Lin-Kernighanovej heuristiky je opísaný v ukážke 9. Vylepšenie, ktoré predstavuje Iterovaná Lin-Kernighanova heuristika produkuje lepšie cesty ako základná Lin-Kernighanova heuristika, no za cenu zvýšených časových nárokov, ktoré sú zvýšené priamoúmerne počtu iterácií max_iter .



Obr. 19: Obrázok znázorňuje schému double-bridge presunu. Túto schému možno vykonať náhodným výberom 4 miest c_0, c_1, c_2, c_3 v poradí, v akom sa nachádzajú v ceste T a následným odstránením hrán $(c_2, next(c_2)), (c_3, next(c_3)), (c_0, next(c_0)), (c_1, next(c_1))$ a vytvorením hrán $(c_2, next(c_0)), (c_3, next(c_1)), (c_0, next(c_2)), (c_1, next(c_3))$. Prevzaté a upravené z [3].

```

best = NULL
for i = 0 to max_iter do
    current_tour = double_bridge(current_tour)
    current_tour = lin-kernighan(current_tour)
    if cost(current_tour) < cost(best) then
        best = current_tour
    end if
end for
return best

```

Ukážka 9: Pseudokód Iterovanej Lin-Kernighanovej heuristiky.

3.3.5 Vylepšený Inver-over operátor

Inver-over operátor predstavil Guo Tao v článku *Inver-over Operator for the TSP* [48]. Tento unárny operátor je založený na jednoduchej inverzii, ale jeho činnosť ovplyvňuje aj znalosť získaná od ostatných jedincov v populácii. Inver-over operátor zahŕňa kríženie, ale aj mutáciu, kedy mesto c' vyberáme z S' náhodne s malou pravdepodobnosťou p . Operátor je jednoduchý a ľahký na implementáciu, sleduje kroky ukážky 10.

```

 $S' = S$ 
nahodne vyber mesto  $c$  z  $S'$ 

while (TRUE) do
    if rand()  $\leq p$  then
        vyber mesto  $c'$  zo zostavajucich miest v  $S'$ 
    else
        nahodne vyber jedinca  $M$  z  $P$ 
        do  $c'$  prirad next( $c$ ) mesta  $c$  vo vybratom jedincovi  $M$ 
        end if
        if next( $c$ ) =  $c'$  or prev( $c$ ) =  $c'$  then
            break
        end if
        invertuj usek od  $c$  po  $c'$  v  $S'$  okrem  $c$ 
         $c = c'$ 
    end while

if cost( $S'$ )  $\leq$  cost( $S$ ) then  $S = S'$ 

```

Ukážka 10: Pseudokód Inver-over operátora.

Autori Y. Wang, J. Li, K. Tao a Q. Pan navrhli v článku [53] *Vylepšený Inver-over operátor*. Práve tento operátor používame v algoritme PIMCSA-DMO ako *imunitnú genetickú operáciu* v PIA. Štandardný Inver-over operátor neberie do úvahy smer cesty v TSP, tak sa môže stať, že mestá, ktoré sa dostali vedľa seba vďaka kríženiu, budú pri ďalšej iterácii operátora oddelené. Túto skutočnosť môžeme vyriešiť tak, že budeme uvažovať smer cesty a budeme invertovať úsek od $next(c)$ po c' v smere cesty S' . Inverzia úseku $next(c)$ po c' tak vlastne predstavuje operáciu $flip(next(c), c')$. Vylepšený Inver-over operátor taktiež zahŕňa dva druhy mutácie, jednu pomocou už opísaného double-bridge presunu s pravdepodobnosťou p_{rd} a jednu výberom mesta c' náhodne spomedzi α najbližších susedov

```

 $S' = S$ 
nahodne vyber mesto  $c$  z  $S'$ 

while (TRUE) do
  rnd = rand(0,1)
  if rnd  $\leq p_{rd}$  then
    vykonaj double-bridge presun nad  $S'$ 
     $c =$  nahodne mesto z  $S'$ 
    continue
  else if rnd  $\leq p_{rd} + p_{cs}$  then
    vyber mesto  $c'$  z  $\alpha$  najblizsich susedov mesta  $c$ 
  else
    nahodne vyber jedinca  $M$  z  $P$ 
    do  $c'$  prirad next( $c$ ) mesta  $c$  vo vybratom jedincovi  $M$ 
  end if
  if next( $c$ ) =  $c'$  or prev( $c$ ) =  $c'$  then
    break
  end if
  invertuj usek od next( $c$ ) po  $c'$  v  $S'$  pozdlz smeru cesty  $S'$ 
  if rand(0,1)  $< p_{uc}$  then
    ponechaj  $c$  nezmenene
  else
     $c = c'$ 
  end if
end while

if cost( $S'$ )  $\leq$  cost( $S$ ) then  $S = S'$ 

```

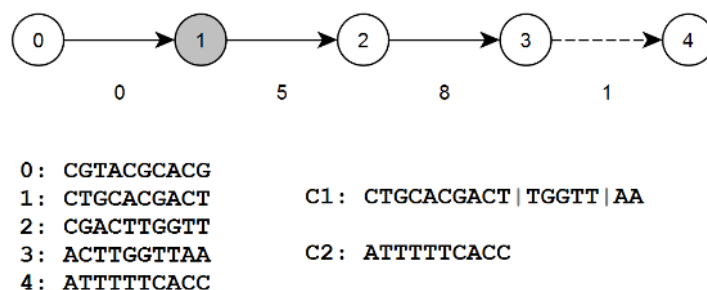
Ukážka 11: Pseudokód Vylepšeného Inver-over operátora.

mesta c s pravdepodobnosťou p_{cs} . Algoritmus taktiež ponecháva mesto c v ďalšej iterácii nezmenené s pravdepodobnosťou p_{uc} . Vylepšený Inver-over operátor dosahuje podstatne lepšie výsledky, ako základný Inver-over operátor za cenu miernejšieho nárastu času potrebného na jeho vykonanie. Vylepšený Inver-over operátor sleduje kroky ukážky 11.

3.4 Konečné spájanie sekvencií

Modul *Connector* prijíma cestu vytvorenú modulom *TSP Solver*. V tomto module dochádza ku vytváraniu *kontigov* na základe cesty v TSP. Modul najskôr zistí smer TSP cesty, nakoľko v asymetrickej verzii TSP nie je jedno, či cestu prechádzame od konca, či od začiatku (na rozdiel od symetrickej verzie TSP, kedy to jedno je). Tento smer sa určí na základe ceny cesty v prípade, že ju prechádzame od začiatku a ceny cesty v prípade prechodu od jej konca, nižšia cena potom indikuje správny smer cesty. Cena cesty sa pritom určuje na základe asymetrickej inštancie TSP, nie na základe symetrickej inštancie od nej odvodennej. Connector následne zistí, prekrytia medzi jednotlivými mestami vo finálnej ceste, uloží ich do poľa prekrytí medzi danými mestami v tejto ceste, pričom mesto B , ktorého predchodca je mesto A , také, že $cost(A, B)$ je najväčšou hodnotou všetkých cien v ceste (a teda medzi sekvenciami predstavovanými mestami A a B je najmenšie prekrytie) je zvolené za *počiatočné mesto* (a s ním

spojenú sekvenciu), od ktorého sa budú skladať ostatné sekvencie. To je nutné z toho dôvodu, že lineárne chromozómy majú svoj začiatok, ktorý je pravdepodobne v mieste najmenšieho prekrytia. Od tohto mesta potom modul postupuje v smere cesty a ku sekvencii prislúchajúcej k mestu B pripája ostatné sekvencie tak, že ich začne pripájať od miesta prekrytia, ako znázorňuje obrázok 20. Ak je počet nukleotidov prekrytia medzi dvoma mestami v ceste nižší ako zvolená kritická hodnota, dôjde k zastaveniu pripájania sekvencií a vytvorí sa *nový kontig*. Túto hodnotu možno nastaviť na ľubovoľné číslo. Delenie na kontigy sa deje z toho dôvodu, že dve sekvencie sa môžu prekrývať v určitom (najčastejšie malom) počte nukleotidov, aj keď v skutočnosti spolu nesusedia a netvorí spoločný kontig. Modul taktiež dbá na to, aby sa sekvencie prislúchajúce k mestu A a za ním nasledujúcemu mestu A' , ktoré predstavuje kópiu mesta A v symetrickej verzii TSP (resp. $A'-A$), chápali ako jedna sekvencia a nekopírovali sa dva razy. To zaistíme tak, že veľkosť prechodu medzi sekvenciami $A-A'$ (resp. $A'-A$) nastavíme na -1 a ak v priebehu vytvárania kontigov narazíme na túto hodnotu, ďalšiu sekvenciu preskočíme.



Obr. 20: Obrázok zachytáva časť cesty v TSP s uzlami (mestami) 0 - 4. S každým uzlom je spätá príslušná DNA sekvencia. Uzol 1 je uzol, do ktorého vedie hrana s najmenšou veľkosťou prekrytia, preto je zvolený za počiatočný uzol skladania. Ku prvému skladanému kontigu C1 teda pridáme celú sekvenciu prislúchajúcu k uzlu 1. Za ním nasledujú uzly 2 a 3, ktorých sekvencie ku kontigu C1 pridáme s ofsetom s veľkosťou rovnou veľkosti prekryvu, ktorý zachytávajú ich vstupné hrany. Na obrázku sú časti sekvencií 2 a 3 v kontigu C1 oddelené zvislou čiarou. Kritickú hodnotu pre delenie kontigov sme v tomto prípade nastavili na 2 a preto sekvencia 4, ktorá má so sekvenciou 3 spoločný 1 nukleotid už do sekvencie C1 pridaná nebude, ale vytvorí sa nový kontig C2, do ktorého bude skopírovaná celá a ďalej sa bude postupovať podobným spôsobom.

Keďže biologicky inšpirované algoritmy ako PIMCSA-DMO produkujú len suboptimálne cesty, často dochádza ku situácii, kedy výsledná TSP cesta nemá správne poradie všetkých miest, no napriek tomu je väčšina miest uvedených v správnom poradí. Môže preto dochádzať k situáciám podobným nasledovnej situácii. Predstavme si optimálnu cestu $A-B-C-D$, kde A , B , C , D predstavujú určité sekvencie miest. Každé tejto sekvencii zodpovedá určitá množina miest, ktoré sú v nej zoradené v správnom poradí tak, aby postupnosť sekvencií $A-B-C-D$ tvorila optimálnu cestu. Suboptimálne cesty ale môžu byť tvorené takýmito správnymi sekvenciami, no tie nemusia byť prítomné v správnom poradí, napr. $A-C-B-D$. Keďže medzi mestami $A_n - C_1$, $C_m - B_1$ a $B_l - D_1$ nie je dostatočný počet prekrývajúcich sa nukleotidov, dôjde k rozdeleniu na 4 kontigy. Preto je v záverečnej fáze možné znova aplikovať modul Compactor, ktorý na vstupe prijme kontigy a tak, ako sme opísali v kapitole 3.2.1, ich pospája do súvislých celkov podľa toho, koľko majú spoločných nukleotidov. Preto možno

modul Compactor nakonfigurovať tak, aby namiesto *kritického pomeru* pracoval s tzv. *kritickým prekrytím* predstavujúcim minimálny počet nukleotidov, ktoré musia byť pre dané dva kontigy spoločné na to, aby mohli byť zlúčené do jedného súvislého kontigu. Compactor tak v tejto fáze vracia menší počet dlhších kontigov. Túto funkcionálnosť je možné zapnúť v nastaveniach skladača DNA sekvencií.

Výstupom modulu Connector je teda *množina kontigov* vo forme reťazca. Následne je možné pomocou skladača tieto kontigy uložiť. O ukladanie kontigov sa stará modul *Saver*, ktorého vstupom je množina kontigov z modulu Connector a výstupom súbor vo formáte *FASTA* s uloženými kontigmi. Vďaka oddeleniu spájania sekvencií do kontigov od ich ukladania je možné skladač rozširovať o ďalšie moduly na spracovanie kontigov alebo o moduly na ich ukladanie do iných výstupných formátov.

4 Implementácia

V tejto kapitole sa venujeme implementačným detailom nami navrhnutého skladača DNA sekvencií. Skladač sme implementovali v jazyku C++ s použitím šablónových tried. Kvôli pamäťovým nárokom celého skladača je nutné aplikáciu kompilovať s použitím 64 bitových kompilátorov. Pre paralelnú komunikáciu sme použili rozhranie *MPI* sprostredkované knižnicou *MS MPI*.

V súčasnosti má celková implementácia 8605 riadkov zdrojového kódu v 80 súboroch, z čoho 4647 riadkov predstavuje čistý kód. Komentáre predstavujú 2027 riadkov a väčšinu z nich tvoria dokumentačné komentáre, vďaka ktorým možno exportovať technickú dokumentáciu pomocou nástroja *DoxyGen*. Túto dokumentáciu sme exportovali vo formáte $\text{\LaTeX}$ súborov, ktoré sme skompilovali do výstupného PDF súboru. Dokumentácia aj implementácia skladača sú umiestnené na priloženom DVD nosiči.

4.1 Implementácia skladača DNA sekvencií

Skladač DNA sekvencií je implementovaný v rámci projektu *DNA_ASSEMBLER* umiestneného na priloženom DVD. Skladač možno kompilovať v dvoch režimoch. Ak pri kompilácii uvedieme definíciu *BENCHMARK_MODE*, skladač bude skompilovaný v testovacom režime určenom na testovanie metódy riešenia TSP. V prípade kompilátora MinGW je možné túto definíciu uviesť v rámci prepínača *-D*. Testovanie je možné vykonávať na GenFrag alebo na TSPLIB inštanciách, pričom sledujeme cenu finálnej cesty a čas, za ktorý daný algoritmus pre riešenie TSP vyriešil vybraný problém obchodného cestujúceho. Ak definíciu *BENCHMARK_MODE* neuvedíme, skladač sa skompiluje v normálnom režime pre skladanie DNA sekvencií, pričom sleduje kroky opísané v kapitole návrhu. Skladač možno konfigurovať pomocou konfiguračného súboru *settings.txt*, ktorého jednotlivé položky opisujeme v prílohe C.

Opis jednotlivých súborov projektu *DNA_ASSMEBLER* sme rozdelili do niekoľkých kapitol podľa ich závislostí.

4.1.1 Základné implementačné súbory skladača DNA sekvencií

defines.h : Tento súbor zhromažďuje všetky symbolické konštanty potrebné pre správny beh skladača. Sú tu uložené predovšetkým ladiace konštanty, ktoré zapínajú príslušné ladiace výpisy. Sú tu definované aj riadiace správy pre komunikáciu v rámci PIMCSA a PIMCSA-DMO.

main.cpp : V súbore main.cpp sa nachádzajú všetky náležitosti pre volanie funkcionality skladača DNA sekvencií. Ak je skladač kompilovaný v režime *BENCHMARK_MODE*, volajú sa príslušné testovacie funkcie zo súboru *test.cpp*, v opačnom prípade sa volá funkcia *assembler_main()* definovaná v *main.cpp*. Funkcia *assembler_main()* vykonáva činnosti skladania DNA sekvencií v závislosti od toho, či bol prekladač skompilovaný s podporou MPI (pri kompilovaní treba uviesť definíciu *USE_MPI*, viac v prílohe B) alebo bez tejto podpory. Ak prekladač nebol skompilovaný s podporou MPI, alebo ak je spúšťaný len pre jeden výpočtový uzol, spustí sa skladanie vytvorením objektu triedy *dna_assembler* pomocou sekvenčného algoritmu na riešenie TSP. Tento algoritmus možno zvoliť v *settings.txt* spomedzi *Lin-Kernighanovej heuristiky*, *Iterovanej Lin-Kernighanovej heuristiky* a *memetického algoritmu* opísaného v [53]. V prípade spustenia skladača s podporou MPI na viac výpočtových uzloch je, v závislosti od toho, či ide o hlavný uzol (má MPI rank rovný 0), alebo vedľajšie uzly, vytvorený objekt triedy *dna_assembler*, ktorý v prípade PIMCSA-DMO predstavuje algoritmus *MLA*, alebo objekty triedy *pia_dmo* (resp. *pia*), ktoré predstavujú implementáciu PIA algoritmu. Každý vedľajší uzol tak vystupuje v úlohe podpopulácie P_i v zmysle TMSM. V nastaveniach možno taktiež zvoliť paralelný algoritmus pre riešenie TSP spomedzi *PIMCSA* a *PIMCSA-DMO*.

dna_assembler.h, dna_assembler.cpp : Trieda *dna_assembler* predstavuje samotný skladač DNA sekvencií a zabezpečuje jeho hlavné funkcionality v zmysle opisu uvedenom v kapitole 3.1. Skladaču sa pri jeho inicializácii poskytne názov súboru so sekvenciami a názov výstupného súboru, do ktorého budú uložené kontigy. Ďalším parametrom, ktorý prijíma konštruktor, je pointer na funkciu vracajúcu inštanciu abstraktnej triedy *tsp_solver*, ktorá predstavuje algoritmus na riešenie TSP. Táto funkcia musí prijímať jediný parameter, a to symetrickú TSP inštanciu, ktorou sa inicializuje daný algoritmus na riešenie TSP. Posledné dva parametre, ktoré konštruktor triedy *dna_assembler* prijíma je logická premenná *output* hovoriaca o tom, či bude na obrazovku produkován výstup o činnosti skladača a logická premenná *mc* značiaca to, či budú kontigy na záver spájané pomocou modulu *Compactor* tak, ako bolo opísané v kapitole 3.4. Po vytvorení objektu triedy *dna_assembler* možno volať jeho metódu *assembly()*, ktorá interne volá tieto súkromné členské metódy:

- *load_data()*: Načíta sekvencie a prevedie ich na symetrickú verziu TSP pomocou triedy *fasta_parser*.
- *solve()*: Na základe pointeru na funkciu vracajúcu inštanciu triedy *tsp_solver*, ktorú sme poskytli v konštruktoze skladača, získa objekt *solver* odvodený od tejto triedy. Volaním metódy *solve()* objektu *solver* následne zaháji hľadanie cesty v TSP. Po nájdení cesty nakoniec prostredníctvom triedy *connector* pospája sekvencie do kontigov a v prípade záverečného spájania kontigov pomocou modulu *Compactor* ešte volá metódu *merge_contigs()* triedy *connector*, ktorá danú akciu vykoná.
- *save_data()*: Uloží poskladané kontigy do výstupného súboru použitím objektu triedy *result_saver*.

asm_functions.h, asm_functions.cpp : Súbor *asm_functions.cpp* definuje funkciu *overlap()*, ktorá prijíma 2 reťazce predstavujúce DNA sekvencie a vracia počet nukleotidov, v ktorých sa prekrývajú. Táto operácia je definovaná v kapitole 3.2.

asm_settings.h, asm_settings.cpp : Súbor *asm_settings.cpp* poskytuje funkciu *load_settings()*, ktorá vytvorí inštanciu triedy *settings* a načíta nastavenia potrebné pre činnosť skladača DNA sekvencií. Následne možno pomocou funkcie *get_settings()* kdekoľvek v programe získať túto inštanciu a pomocou nej pristupovať k jednotlivým nastaveniam skladača.

settings.h, settings.cpp : Trieda *settings* parsuje súbor s nastaveniami, v ktorom majú jednotlivé nastavenia formát KLÚČ = hodnota. Triedu používa funkcia *load_settings()* definovaná v súbore *asm_settings.cpp*, ktorá načíta nastavenia skladača DNA sekvencií. Z objektu triedy *settings* možno získavať hodnoty jednotlivých nastavení pomocou metód *get_int()*, *get_double()*, *get_string()* a *get_bool()*. Všetky tieto funkcie prijímajú reťazec označujúci dané nastavenie (klúč) a defaultnú hodnotu, ktorá bude vrátená, ak sa dané nastavenie v súbore s nastaveniami nenachádza, alebo ak nie je možné pre daný klúč vyparsovať hodnotu požadovaného dátového typu.

tsp_parser.h, tsp_parser.cpp : Trieda *tsp_parser* predstavuje základnú triedu pre všetky parsery určené na prevod ľubovoľného vstupu na TSP. Trieda definuje abstraktnú metódu *parse()*, ktorá vracia symetrickú TSP inštanciu.

fasta_parser.h, fasta_parser.cpp : Trieda *fasta_parser* predstavuje modul *Parser* a slúži na parsovanie FASTA súborov a ich prevod na *symetrickú TSP inštanciu*. Trieda je odvodená od triedy *tsp_parser* a vyžaduje, aby sekvencie vo vstupnom súbore boli jedinečné. To možno dosiahnuť ich predspracovaním pomocou iného programu, alebo pomocou samotnej triedy *fasta_parser*, ak je táto možnosť zapnutá v nastaveniach *settings.txt*. Po volaní metódy *parse()* trieda najskôr načíta sekvencie vo formáte reťazcov do poľa sekvencií, v prípade zapnutého filtrovania duplicitných sekvencií odstráni sekvencie, ktoré už sú v tomto poli prítomné a potom, pomocou súkromnej metódy *compute_overlaps()* vypočíta maticu prekrytí tak, ako to bolo opísané v kapitole 3.2. Následne túto maticu prevedie na asymetrickú inštanciu TSP a tú prevedie na symetrickú inštanciu aplikovaním krokov, ktoré sme opísali v spomínanej kapitole. Keďže filtrovanie duplicitných sekvencií sa vykonáva ukladaním sekvencií do objektu STL triedy *set* a vyhľadávaním duplicit v tomto objekte pomocou metódy *find*, dochádza k zvýšeniu výpočtových nárokov a preto je filtrovanie implicitne vypnuté. Pre ďalšie šetrenie pamäťou je možné načítané sekvencie po vytvorení TSP inštancie uvoľniť a znova ich načítať, až keď budú potrebné pri spájaní sekvencií v module *Connector*.

compactor.h, compactor.cpp : Trieda *compactor* predstavuje modul *Compactor* opísaný v časti 3.2.1. Konštruktor tejto triedy prijíma pole DNA sekvencií (ako inštanciu STL triedy *vector*), číslo predstavujúce *kritický pomer* alebo *kritické prekrytie* a, voliteľne, názov súboru, do ktorého budú

uložené pospájané sekvencie (predčasné kontigy). Ak sa názov tohto súboru neuvedie, predčasné kontigy nebudú uložené. Toto uloženie je výhodné vtedy, ak chceme skladanie sekvencií spúšťať viac krát a nechceme zakaždým spúšťať modul *Compactor*, s cieľom skrátiť čas potrebný na načítanie DNA sekvencií. Volaním metódy `compactify()` dôjde k samotnému pospájaniu DNA sekvencií, pričom metóda vracia nové pole už pospájaných sekvencií.

connector.h, connector.cpp : Trieda `connector` predstavuje modul *Connector* a slúži na spájanie sekvencií do kontigov na základe optimálnej (resp. suboptimálnej) cesty v TSP, ktorú našiel modul *TSP Solver*. Najdôležitejšou metódou tejto triedy je metóda `connect()`, ktorá realizuje spájanie sekvencií do kontigov tak, ako sme to opísali v časti 3.4. Ak sme predtým načítané DNA sekvencie uvoľnili, metóda ich pred samotným spájaním znova načíta volaním metódy `load_sequences()` triedy `fasta_parser`, ktorá uchováva dané sekvencie. Nad objektom triedy `connector` je možné taktiež volať metódu `merge_contigs()`, ktorá vytvorí objekt triedy `compactor` a pomocou neho vykoná dodatočné zlučovanie kontigov tak, ako sme opísali v závere kapitoly 3.4.

result_saver.h, result_saver.cpp : Trieda `result_saver` slúži na ukladanie kontigov získaných triedou `connector` do výstupného FASTA súboru. Pred každý kontig sa uvedie komentár, ktorý možno konfigurovať v konfiguračnom súbore skladača.

4.1.2 Implementačné súbory pre riešenie TSP

tsp_instance.h : Šablónová trieda `tsp_instance` predstavuje základnú triedu pre všetky TSP inštancie. Poskytuje metódy pre prácu s maticou vzdialeností medzi jednotlivými mestami. Dôležitou je abstraktná metóda `set_cost()`, ktorá slúži na nastavenie vzdialenosti medzi mestom *a* a *b*. Metóda `cost()` potom vracia vzdialenosť medzi týmito dvoma mestami. Šablónová trieda je konkretizovaná dátovým typom `STORAGE_T`, ktorý slúži na uloženie vzdialenosti medzi dvoma mestami v matici vzdialeností a dátovým typom `COST_RET_T` pre vzdialenosť, ktorú vracia metóda `cost()`. Tieto typy môžu, ale nemusia byť rovnaké. V špecifických prípadoch je možné nastaviť `STORAGE_T` na dátový typ menšej veľkosti ako `COST_RET_T`. Jeden z týchto prípadov sa týka aj nášho skladača DNA sekvencií a bude opísaný neskôr v tejto kapitole. Matica vzdialeností je uložená ako jednorozmerné pole, čo uľahčuje jej posielanie pomocou MPI. Pointer na pole reprezentujúce túto maticu možno získať volaním metódy `get_cost_matrix()`.

asymmetric_tsp_instance.h : Šablónová trieda `asymmetric_tsp_instance` je odvodená od triedy `tsp_instance` a predstavuje asymetrickú inštanciu TSP. Pri vkladaní ceny cesty medzi dvoma mestami pomocou `set_cost()` sa táto cena neukladá do matice vzdialeností symetricky.

symmetric_tsp_instance.h : Šablónová trieda `symmetric_tsp_instance` odvodená od triedy

`tsp_instance` predstavuje symetrickú inštanciu TSP. Pri vkladaní cesty medzi dvoma mestami pomocou `set_cost()` sa cena ukladá symetricky, teda uloží sa ako $C[a, b] = C[b, a] = n$, kde n predstavuje cenu cesty medzi mestami a a b . Táto trieda tiež obsahuje konštruktor prijímajúci inštanciu `asymmetric_tsp_instance`, ktorý sa stará o prevod z asymetrickej verzie TSP na symetrickú verziu prevodom matice C na maticu C' tak, ako bolo opísané v kapitole 3.2. Pre šetrenie pamäťou sa hodnota *neplatného prechodu* iec síce poskytne v konštruktoze, ale táto hodnota sa neukladá do matice C' priamo, nakoľko cena neplatného prechodu môže byť o niekoľko rádov vyššia ako ceny platných prechodov. Preto treba iec reprezentovať väčším dátovým typom (`COST_RET_T` namiesto `STORAGE_T`). Keby sme túto hodnotu ukladali do matice C' priamo, musela by byť celá matica predstavovaná väčším dátovým typom, čo by viedlo k plytvaniu pamäte. Namiesto neplatných prechodov sa preto uloží hodnota `DIST_MAX + 1`, kde `DIST_MAX` predstavuje maximálnu možnú vzdialenosť medzi dvoma mestami. Metóda `cost()` potom namiesto tejto zástupnej hodnoty vráti samotnú hodnotu neplatného prechodu iec.

stsp_from_atsp.h: Šablónová trieda `stsp_from_atsp` predstavuje symetrickú inštanciu TSP, ktorá vznikla konverziou z asymetrickej verzie TSP s tým, že nedochádza ku kopírovaniu asymetrickej TSP inštalácie a vytváraniu novej matice C' . Namiesto toho inštalácia tejto triedy iba odkazuje na vzdialenostnú maticu asymetrickej inštalácie, ktorá je jej predaná v konštruktoze. Preto je potrebné zaručiť existenciu asymetrickej inštalácie počas celej životnosti symetrickej TSP inštalácie reprezentovanej touto triedou. Použitím objektu tejto triedy dochádza ku štvornásobnému šetreniu pamäte tak, ako sme to opísali v kapitole 3.2. Z dôvodu nevytvárania symetrickej matice C' , trieda prekrýva metódu `cost()` v zmysle ukážky 4. Cenou za ušetrnú pamäť je to, že pomocou objektu triedy `stsp_from_atsp` nie je možné použiť metódu `set_cost()`, a že musíme dbať na to, aby nedošlo k uvoľneniu asymetrickej TSP inštalácie zviazanej s týmto objektom. Naš skladáč DNA sekvencií z dôvodu pamäťovej efektívnosti používa pri pretypovaní z asymetrickej inštalácie TSP práve objekt tejto triedy.

cost_types.h: V tomto súbore sú definované dátové typy a s nimi súvisiace konštanty využívané pri reprezentovaní a riešení problému obchodného cestujúceho pre potreby skladania DNA sekvencií. Ide o tieto dátové typy a konštanty:

- `COST_T`: dátový typ reprezentujúci vzdialenosť medzi dvoma mestami v TSP, ktorá je vrátená pri použití metódy `cost()`, je nastavený na `int`
- `COST_SUM_T`: dátový typ reprezentujúci sumu vzdialeností medzi všetkými mestami v TSP určený na reprezentovanie ceny cesty, je nastavený na `long int`
- `COST_SUM_MAX`: konštanta reprezentujúca maximálnu cenu cesty, nastavená na `LONG_MAX`
- `DIST_T`: dátový typ reprezentujúci vzdialenosť medzi dvoma mestami v TSP, ktorá je uložená vo vzdialenostnej matici, je nastavený na `int16_t`

- `DIST_MAX`: maximálna vzdialenosť medzi ľubovoľnými dvoma mestami v TSP
- `MPI_DIST_T`: reprezentácia `DIST_T` v správach MPI, nastavená na `MPI_INT16_T`
- `MPI_COST_T`: obdoba `COST_T` v správach MPI, nastavená na `MPI_INT`

tsp_instance_type.h : V tomto súbore sa špecializujú šablóny inštancií TSP pre potreby skladania DNA sekvencií a definujú sa nimi dátové typy `STSP_INST` a `ATSP_INST` nasledovne:

```
typedef tsp_from_atsp<DIST_T, COST_T> STSP_INST;
typedef asymmetric_tsp_instance<DIST_T, COST_T> ATSP_INST;
```

Tento súbor tiež definuje konštantu `STSP_COPY_ATSP`, ktorá je nastavená na `true`, ak symetrická TSP inštancia vytvára vlastnú maticu vzdialeností pri pretypovaní z asymetrickej TSP inštancie (prípád `symmetric_tsp_instance`), alebo na `false`, ak symetrická TSP inštancie len odkazuje na maticu vzdialeností asymetrickej TSP inštancie (prípád `tsp_from_atsp`). Táto konštanta je používaná skladačom, ktorý na základe nej určí, či je možné asymetrickú inštanciu po jej pretypovaní na symetrickú verziu z pamäte uvoľniť, alebo nie. Nakoľko trieda `symmetric_tsp_instance` vytvára vlastnú maticu vzdialeností, asymetrickú inštanciu, z ktorej vznikla, možno odstrániť. To neplatí pre `tsp_from_atsp`.

tour.h, tour.cpp : Trieda `tour` reprezentuje cestu v TSP. Trieda je odvodená od triedy `std::vector` špecializovanej typom `int` a mestá ukladá ako postupnosť čísel, nakoľko každé mesto je identifikované vlastným číslom. Toto číslo je zhodné s poradím DNA sekvencie v súbore so sekvenciami. Trieda obsahuje metódu `cost()` vracajúcu celkovú cenu cesty a metódu `to_array()` pre prevod cesty na `int[]` pole postupnosti miest, ktoré je možné prenášať prostredníctvom MPI. Taktiež možno v čase $O(1)$ pristupovať k jednotlivým mestám v ceste na základe ich poradia v ceste pomocou operátora `[]` (napr. `t[0]` vracia prvé mesto v ceste, `t[1]` druhé, atď.). Mestá sa do cesty pridávajú pomocou metódy `add()`, pričom je nutné dávať pozor na to, aby sa neopakovali a aby boli v ceste prítomné všetky mestá spojené s inštanciou TSP, pre ktorú je cesta vytváraná. Trieda správnosť cesty z dôvodu efektívnosti nijako nekontroluje.

genetic_operators.h, genetic_operators.cpp : Súbor *genetic_operators.cpp* implementuje funkciu `k_exchange()` predstavujúcu *k*-exchange presun opísaný v kapitole 3.3.4 a `double_bridge()` funkciu predstavujúcu double-bridge presun.

inver_over.h, inver_over.cpp : Trieda `inver_over` implementuje *Inver-over* operátor definovaný v kapitole 3.3.5. V konštruktore prijíma množinu ciest, z ktorých sa bude vyberať mesto c' pri jednotlivých prechodoch hlavným cyklom operátora. Tieto cesty budú dodávať potrebnú informáciu pre kríženie. Taktiež prijíma pravdepodobnosť p , s ktorou bude mesto c' vybraté náhodne, spomedzi

zvyšných miest práve modifikovanej cesty. Nad zvolenou cestou T potom vykonáme Inver-over operátor volaním metódy `run()` triedy `inver_over`, ktorej poskytneme danú cestu T ako parameter.

improved_inver_over.h, improved_inver_over.cpp : Trieda `improved_inver_over` implementuje *Vylepšený Inver-over operátor* definovaný v kapitole 3.3.5. Táto trieda je odvodená od triedy `inver_over` a okrem parametrov, ktoré prijíma táto trieda, prijíma aj pravdepodobnosti, ktoré sme špecifikovali pri opise Vylepšeného Inver-over operátora a inštanciu triedy `neighborhoods` reprezentujúcu najbližších susedov jednotlivých miest. Podobne, ako Inver-over operátor, aj Vylepšený Inver-over operátor sa spúšťa pomocou metódy `run()`.

tsp_solver.h, tsp_solver.cpp : Abstraktná trieda `tsp_solver` predstavuje základ každého modulu *TSP Solver* a poskytuje rozhranie pre riešenie problému obchodného cestujúceho. Každá trieda odvodená od tejto triedy potom predstavuje nejaký algoritmus určený na riešenie TSP. Trieda `tsp_solver` deklaruje abstraktnú metódu `solve()`, ktorá vykonáva samotné riešenie problému obchodného cestujúceho. Chránená metóda `create_start_tour()` vytvorí počiatočnú cestu v tvare $1, 2, \dots, N$, kde jednotlivé čísla predstavujú identifikátory daných miest. Metóda `get_final_tour()` vracia výslednú cestu, ktorá vznikla vyriešením danej TSP inštancie. Pre potreby skladania DNA sekvencií sa požaduje, aby každá trieda odvodená od tejto triedy prijímala vo svojom konštruktore len symetrickú TSP inštanciu typu `STSP_INST`. Pre každú triedu odvodenú od tejto triedy potrebujeme definovať funkciu, ktorá prijíma práve túto inštanciu a vracia pointer na objekt danej odvodenej triedy. To je dôležité preto, aby sme vedeli objektu triedy `dna_assembler` poskytnúť meno funkcie, ktorá vytvorí inštanciu triedy `tsp_solver`, čím je možné modifikovať to, ktorý algoritmus bude použitý na riešenie TSP v procese skladania DNA sekvencií. Takto možno v skladači DNA sekvencií zamieňať moduly *TSP Solver* bez toho, aby to malo vplyv na ostatné časti skladača.

tsp_lk_solver.h, tsp_lk_solver.cpp : Trieda `tsp_lk_solver` odvodená od triedy `tsp_solver` predstavuje modul *TSP Solver* riešiaci TSP len pomocou *Lin-Kernighanovej heuristiky*, ktorú sme opísali v kapitole 3.3.3. Na riešenie TSP používa objekt triedy `lin_kernighan`. Modul zlepšuje len jednu cestu vytvorenú pomocou metódy `create_start_tour()` definovanú v triede `tsp_solver`.

tsp_ilk_solver.h, tsp_ilk_solver.cpp : Trieda `tsp_ilk_solver` odvodená od triedy `tsp_solver` predstavuje modul *TSP Solver* riešiaci TSP len pomocou *Iterovanej Lin-Kernighanovej heuristiky*, ktorú sme opísali v kapitole 3.3.4. Na riešenie TSP používa inštanciu triedy `iterated_lin_kernighan`. Tak ako v prípade `tsp_lk_solver`, aj teraz modul zlepšuje len jednu cestu vytvorenú pomocou metódy `create_start_tour()`.

iolk_ma.h, iolk_ma.cpp : Trieda `iolk_ma` odvodená od triedy `tsp_solver` predstavuje memetický algoritmus na riešenie TSP, ktorý využíva Lin-Kernighanovu heuristiku a Vylepšený Inver-over operátor. Tento algoritmus je bližšie opísaný v článku [53].

mia_dmo.h, mia_dmo.cpp : Trieda `mia_dmo` reprezentuje algoritmus MIA algoritmu PIMCSA-DMO a je odvodená od triedy `tsp_solver`. Algoritmus MIA predstavuje pamäťovú populáciu v algoritme PIMCSA-DMO a je spustený na hlavnom výpočtovom uzle. V rámci metódy `solve()` sú definované jednotlivé kroky algoritmu MIA tak, ako boli opísané v kapitole 3.3.1. Na riadenie komunikácie prostredníctvom MPI sa používajú riadiace správy definované v *defines.h*:

- `NOTHING_MSG`: MIA alebo PIA v danom okamihu nebude odosielať žiadne dáta (napr. sa tieto správy používajú na synchronizáciu výpočtových uzlov).
- `TERMINATE_MSG`: MIA odosiela túto správu PIA pre ukončenie činnosti PIA.
- `SENDING_BEST_MSG`: PIA bude v ďalšej správe pre MIA posílať svoju najlepšiu protilátku.
- `SENDING_VACC_MSG`: MIA bude v ďalšej správe pre PIA odosielať vakcínu.

Riadiace správy predstavujú MPI správy s tagom nastaveným na `MESSAGE_TAG`, v prípade odosielania dát (vakcín alebo protilátok) sa použije tag `DATA_TAG`. Trieda `mia_dmo` najskôr odošle symetrickú TSP inštanciu jednotlivým vedľajším uzlom, na ktorých beží algoritmus PIA, potom sa začne so samotným vykonávaním algoritmu PIMCSA-DMO. Trieda `mia_dmo` v spojitosti s triedou `pia_dmo` predstavuje modul *TSP Solver*, ktorý je v prípade paralelného výpočtu implicitne používaný našim skladačom DNA sekvencií.

pia_dmo.h, pia_dmo.cpp : Trieda `pia_dmo` predstavuje algoritmus PIA, ktorý je súčasťou algoritmu PIMCSA-DMO. Samotná akcia algoritmu PIA sa spúšťa metódou `launch()` a sleduje kroky opísané v časti 3.3.1. Trieda tiež definuje metódu `create_start_tour()`, ktorá náhodne inicializuje počiatočnú cestu v TSP. Ňou sa inicializujú jednotlivé protilátky v danej populácii. Algoritmus PIA pre svoju činnosť využíva *Vylepšený Inver-over operátor* a *Iterovanú Lin-Kernighanovu heuristiku*.

mia.h, mia.cpp : Trieda `mia` je odvodená od triedy `tsp_solver` a predstavuje algoritmus MIA v pôvodnom algoritme PIMCSA. V rámci tohto algoritmu sa aplikuje *Iterovaná Lin-Kernighanova heuristika* ako operácia zrenia pamäťových protilátok nad všetkými protilátkami v pamäťovej populácii. Podrobnosti o algoritme PIMCSA uvádzajú autori článku [37].

pia.h, pia.cpp : Trieda `pia` predstavuje algoritmus PIA pôvodného algoritmu PIMCSA. Podobne, ako algoritmus PIA, ktorý je súčasťou algoritmu PIMCSA-DMO sa aj činnosť tohto algoritmu spúšťa pomocou metódy `launch()`. Podrobnosti o algoritme PIMCSA uvádzajú autori článku [37].

pimcsa_functions.h, pimcsa_functions.cpp : Tieto súbory definujú funkciu `affinity()` používanú v algoritmoch PIMCSA-DMO a PIMCSA. Pre protilátku `T` je definovaná ako `COST_SUM_MAX - (T.cost + T.cities_count)`.

4.1.3 Implementačné súbory Lin-Kernighanovej heuristiky

lk_tour.h, lk_tour.cpp : Trieda `lk_tour` predstavuje cestu v TSP upravenú pre potreby Lin-Kernighanovej heuristiky. Cesta je reprezentovaná vo forme obojsmerne spájaného zoznamu, pričom jednotlivé mestá sú uložené v poli, kde index tohto poľa predstavuje identifikátor mesta. Vďaka tejto reprezentácii sme mohli v triede definovať operáciu `next(n)` vracajúcu nasledovníka mesta s id `n`, `prev()` vracajúcu predchodcu mesta s id `n` a `flip(a, b)` vykonávajúcu operáciu *flip* medzi mestami s id `a` a `b`. Operácia `cost()` vracia celkovú cenu cesty. Objekt triedy `lk_tour` možno inicializovať pomocou cesty typu `tour`, ktorá sa prevedie na reprezentáciu `lk_tour`. Pomocou triedy `tour` môžeme zistiť, na ktorej pozícii cesty sa nachádza ktoré mesto. Trieda `lk_tour` zase umožňuje zistiť predchodcu a nasledovníka daného mesta, ktoré je reprezentované svojím číselným identifikátorom.

lin_kernighan.h, lin_kernighan.cpp : Trieda `lin_kernighan` implementuje Lin-Kernighanovu heuristiku opísanú v kapitole 3.3.3. Inicializuje sa cestou `lk_tour`, samotný výpočet nad touto cestou sa spustí volaním metódy `run()` a výsledok možno získať pomocou metódy `get_current_tour()`. Metóda `run()` volá metódu `lk_search()`, ktorá pomocou rekurzívneho kroku `step()` vykonáva samotné lokálne prehľadávanie a vylepšovanie cesty.

iterated_lin_kernighan.h, iterated_lin_kernighan.cpp : Táto trieda predstavuje Iterovanú Lin-Kernighanovu heuristiku opísanú v kapitole 3.3.4. Inicializuje sa cestou `lk_tour`, výpočet sa spustí volaním metódy `iterate(i)`, ktorej parameter `i` je počet iterácií a výsledok možno získať pomocou metódy `get_best()`.

flip_sequence.h, flip_sequence.cpp : Trieda `flip_sequence` predstavuje sekvenciu flipov, do ktorej možno pridávať a odoberať flipy. Sekvencia flipov sa udržiava preto, aby sme vedeli odstrániť zlé flipy vykonané počas hľadania v `lk_search()`.

flip.h, flip.cpp : Predstavuje jeden flip v sekvencii flipov pre dvojicu miest `x` a `y`.

neighborhood.h, neighborhood.cpp : Trieda `neighborhood` pre dané mesto `c` vytvára a udržiava zoznam `n` najbližších miest. Len tieto najbližšie mestá budú použité pri prehľadávaní v `lk_search()`, čím sa výrazne zredukujú časové nároky, za cenu menšej straty kvality výslednej cesty. Parameter `n` možno špecifikovať v konštruktore triedy.

neighborhoods.h, neighborhoods.cpp : Trieda `neighborhoods` udržiava `neighborhood` zoznamy pre všetky mestá v danej inštancii TSP.

lk_ordering.h, lk_ordering.cpp : Trieda `lk_ordering` pre zvolené mesto `c` zhromažďuje zoradený zoznam miest, ktoré budú uvažované pri prehľadávaní v `lk_search()`. Ide o tzv. *sľubné vrcholy*, ktoré

sú zoradené podľa svojho potenciálu na zlepšenie cesty, nad ktorou aplikujeme Lin-Kernighanovu heuristiku. Tieto uzly potom možno využiť pri aplikácii jednotlivých flipov.

move.h, move.cpp : Trieda `move` predstavuje položku zoznamu `lk_ordering` a udržiava dvojicu $(s, prev(s))$ pre sľubný vrchol s a cenu, o ktorú bude cesta aplikovaním flipu $flip(next_base, prev(s))$ vylepšená. Táto cena je potrebná pre zoradenie v `lk_ordering`.

queue.h, queue.cpp : Trieda `queue` predstavuje front pre mestá, od ktorých bude ešte začaté prehľadávanie pomocou `lk_search()`. Je obdobou značenia vrcholov v Lin-Kernighanovej heuristike.

vertex.h, vertex.cpp : Predstavuje mesto v triede `queue`.

4.1.4 Implementačné súbory testovacieho režimu

genfrag_parser.h, genfrag_parser.cpp : Trieda `genfrag_parser`, ktorá je odvodená od triedy `tsp_parser` predstavuje parser GenFrag inštancií. Tieto inštancie boli použité pri testovaní nášho skladača DNA sekvencií a predstavujú symetrické TSP inštancie získané z DNA sekvencií na základe ich vzájomných prekrytí, ktoré boli získané aplikáciou Smith-Watermanovho algoritmu pre zarovnávanie DNA sekvencií. Z týchto prekrytí potom boli generované symetrické matice reprezentujúce TSP.

tsp_lib_parser.h, tsp_lib_parser.cpp : Tieto súbory definujú triedu `tsplib_parser` odvodenú od triedy `tsp_parser`. Objekt tejto triedy načítava euklidovské TSPLIB inštancie a prevádza ich do vnútornej reprezentácie definovanej pomocou triedy `symmetric_tsp_instance`.

tests.h, tests.cpp : Tieto súbory definujú funkcie `genfrag_benchmark()` a `tsplib_benchmark()`, ktoré načítajú príslušné symetrické TSP inštancie a vyriešia ich na základe zvoleného algoritmu pre riešenie TSP. Tento algoritmus možno zvoliť v nastaveniach skladača. Obe funkcie vypisujú cenu finálnej cesty, ktorá vznikla po riešení TSP a čas, za ktorý ju daný algoritmus našiel. V prípade GenGrag inštancií je vypísaná aj suma prekrytí DNA sekvencií pozdĺž finálnej cesty tak, ako uvádzajú autori článku [28]. Práve táto hodnota je používaná pri vyhodnocovaní algoritmov na riešenie TSP.

4.2 Framework pre skladanie DNA sekvencií pomocou riešenia TSP

Vedľajším produktom našej diplomovej práce je *Framework pre skladanie DNA sekvencií pomocou riešenia problému obchodného cestujúceho*, nakoľko naša implementácia skladača DNA sekvencií ponúka množstvo funkcionalít, ktoré možno využiť pri vývoji iných skladačov riešiacich FAP práve

pomocou TSP. Implementovali sme moduly *Parser*, *Compactor*, *TSP Solver*, *Connector* a *Saver*, ktoré tak možno opätovne použiť.

Z hľadiska riešenia TSP sme implementovali *Lin-Kernighanovu heuristiku*, *Iterovanú Lin-Kernighanovu heuristiku*, *Inver-over operátor*, *Vylepšený Inver-Over operátor*, *k-exchange presun*, *double-bridge presun* a v rámci triedy *1k_tour* aj operáciu *flip*. Tieto operácie môžu byť v spojení s našou implementáciou symetrickej TSP inštancie (či už pomocou *symmetric_tsp_instance* alebo *stsp_from_atsp*) použité pre riešenie ľubovoľnej symetrickej, či asymetrickej verzie TSP. Nami definované asymetrické i symetrické inštancie TSP sú implementované ako šablónové triedy a preto môžu byť použité aj v iných oblastiach ako je skladanie DNA sekvencií. Vzdialenosť medzi dvoma mestami je teda možné definovať pre ľubovoľný číselný základný dátový typ.

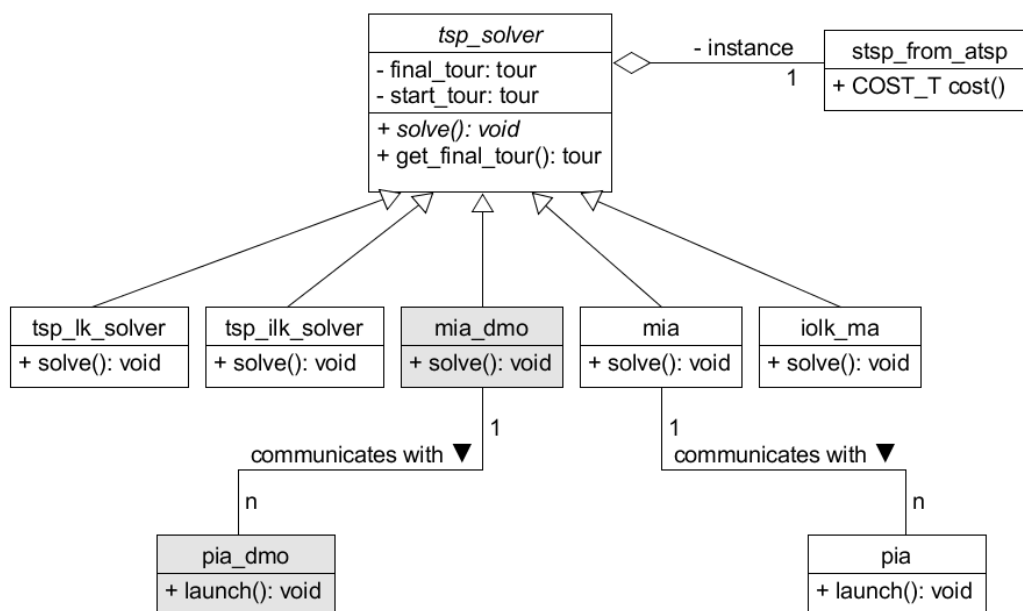
Implementovali sme tiež viacero parserov TSP inšancií, ako *genfrag_parser*, *tsplib_parser* a *fasta_parser*. Práve *FASTA Parser* načítava DNA sekvencie a transformuje ich na problém obchodného cestujúceho. Odvođením od triedy *tsp_parser* možno implementovať aj iné parseery, ktoré môžu konvertovať rôzne typy ťažkých problémov práve na TSP, čím teoreticky možno prostriedky frameworku použiť i na iné problémy ako skladanie DNA sekvencií. Jednotlivé moduly skladača sme sa snažili implementovať tak, aby boli čo najmenej previazané a aby bolo ich použitie čo najjednoduchšie. Každý základný modul (*FASTA Parser*, *TSP Solver*, *Connector*, *Compactor* a *Saver*) tak možno reimplementovať a vymeniť bez toho, aby to ovplyvnilo činnosť zvyšku skladača DNA sekvencií. I napriek značnej vnútornej zložitosti každého modulu a množstvu akcií, ktoré každý modul vykonáva, je ich použitie jednoduché. To, ako jednoducho možno použiť jednotlivé moduly na skladanie DNA sekvencií v mierne zjednodušenej podobe zobrazuje ukážka 12.

```
tsp_parser * parser = new tsp_parser('reads.fasta');
STSP_INST * instance = parser->parse();
tsp_solver * solver = create_pimcsa_dmo_solver(instance);
solver->solve();
tour final_tour = solver->get_final_tour();
connector con(instance, parser, final_tour);
con.connect();
result_saver saver(con, 'contigs.fasta');
saver.save();
delete solver;
delete parser;
```

Ukážka 12: Činnosť skladača DNA sekvencií

Najlepší príklad opísanej výmeny modulov možno vidieť práve pri module *TSP Solver*. V rámci nášho skladača sme implementovali niekoľko metód riešenia TSP, z ktorých každá predstavuje rôzny *TSP Solver*: riešenie TSP pomocou *Lin-Kernighanovej heuristiky*, riešenie TSP pomocou *Iterovanej Lin-Kernighanovej heuristiky*, pomocou *Memetického algoritmu* s *Lin-Kernighanovou heuristikou* a *Vylepšeným Inver-over operátorom*, pomocou algoritmu *PIMCSA* a riešenie TSP pomocou nami navrhnutej modifikácie *PIMCSA* algoritmu - *PIMCSA-DMO*. To, že z nášho skladača DNA sekvencií

vznikol framework umožňujúci skladanie DNA sekvencií pomocou riešenia TSP a to, že sme sami implementovali viacero modulov pre riešenie TSP, vystalo práve z potreby testovania nášho skladača a jeho kľúčového algoritmu PIMCSA-DMO. Aby nebolo potrebné neustále zdĺhavo meniť zdrojový kód skladača, bolo vhodné implementovať mechanizmus, ktorý by nám umožnil ľahko pridávať algoritmy na riešenie TSP a ľahko ich včleniť do samotného skladača reprezentovaného triedou `dna_assembler`. Práve preto sme definovali abstraktnú triedu `tsp_solver`, od ktorej musí byť každý algoritmus na riešenie TSP odvodený. Každý takýto algoritmus potom musí definovať metódu `run()`, ktorá vykoná samotné riešenie TSP a vo svojom konštruktore musí prijímať len jeden parameter, symetrickú TSP inštanciu podľa predpisu `STSP_INST`. Pre každý algoritmus riešenia TSP navyše musíme definovať funkciu, ktorá prijíma symetrickú TSP inštanciu a vracia inštanciu daného algoritmu. Práve táto funkcia sa predáva objektu skladača ako parameter konštruktora triedy `dna_assembler`, pričom v správny moment procesu skladania DNA sekvencií dôjde ku jej volaniu a vytvoreniu inštancie triedy `tsp_solver`. V prípade zahrnutia nového algoritmu na riešenie TSP do skladača DNA sekvencií stačí upraviť funkciu `assembler_main()` v súbore `main.cpp` a rozšíriť ju o vytvorenie skladača DNA sekvencií volaním jeho konštruktora s parametrom predstavujúcim funkciu pre vytvorenie inštancie nového algoritmu riešiaceho TSP, pričom sa možno inšpirovať už existujúcim kódom. Tak bude možné vybrať metódu riešenia TSP pomocou konfiguračného súboru skladača `settings.txt`. Diagram na obrázku 21 znázorňuje všetky implementované moduly *TSP Solver*.



Obr. 21: Obrázok znázorňuje jednotlivé moduly TSP Solver. Farebne odlíšené sú triedy reprezentujúce algoritmus PIMCSA-DMO, ktorý je hlavným algoritmom riešiacim TSP použitým v našom skladači DNA sekvencií. Triedu `stsp_from_atsp` možno nahradiť za `symmetric_tsp_instance` podľa definície v `STSP_INST`.

4.3 Použitie skladača DNA sekvencií

Skladač možno použiť na osobnom počítači, rovnako ako na klastri. Pre skladanie veľkých množín DNA sekvencií však skladač vyžaduje dostatočné množstvo operačnej pamäte, výpočtovej sily a času potrebného na beh programu. Odporúčame preto spúšťať skladač s použitím paralelizmu. Skladač možno skompilovať s podporou *MPI*, ak pri kompilácii uvedieme direktívu *USE_MPI*, alebo bez tejto podpory, ak danú direktívu vynecháme. Po skompilovaní možno meniť konfiguráciu skladača DNA sekvencií pomocou konfiguračného súboru *settings.txt*, ktorého položky sú opísané v prílohe C. Skladač možno potom spustiť príkazom:

```
mpirun -n <N> DNA_ASSEMBLER
```

Prepínač *n* označuje počet zúčastnených procesov *<N>* a teda počet podpopulácií algoritmu PIMCSA-DMO (resp. PIMCSA) zmenšený o 1.

5 Testovanie

V tejto kapitole sa venujeme testovaniu skladača DNA sekvencií, ktorého návrh a implementáciu sme opísali v predchádzajúcich kapitolách. Najprv sme sa zamerali na testovanie pomocou TSPLIB inštancií [50] a GenFrag inštancií [28], ktoré predstavujú symetrické inštancie problému obchodného cestujúceho v *testovacom režime* skladača. Zamerali sme sa pritom hlavne na testovanie presnosti hľadania cesty pomocou nášho algoritmu PIMCSA-DMO a na testovanie času, za ktorý algoritmus nájde výslednú optimálnu (resp. suboptimálnu) cestu. Algoritmus sme porovnali s inými algoritmami a na základe výsledkov sme usúdili, že algoritmus PIMCSA-DMO dosahuje značné úspechy pri riešení TSP, pričom kvalitou výsledných ciest prekonáva mnohé podobné algoritmy a preto je vhodný na aplikáciu v rámci skladania DNA sekvencií.

Následne sme testovali samotný skladač DNA sekvencií, ako z hľadiska presnosti skladania sekvencií, tak aj z hľadiska času potrebného pre skladanie týchto sekvencií. Na testovanie sme použili krátke DNA sekvencie generované *generátorom DNA sekvencií*, ktorý sme predstavili v predchádzajúcej práci [47]. Tento generátor najskôr, na základe nastavení, vygeneruje referenčný genóm, nad ktorým potom simuluje prácu sekvencera, jeho výstupom je teda vygenerovaný *referenčný genóm a súbor s krátkymi sekvenciami*. Pomocou referenčného genómu potom vieme vyhodnotiť správnosť skladania DNA sekvencií. Pri testovaní skladania DNA sekvencií sme sa sústredili na porovnanie PIMCSA-DMO s inými prístupmi riešenia problému obchodného cestujúceho, na ktorý je problém skladania DNA sekvencií prevedený. Porovnali sme preto výsledky skladania získané pomocou *PIMCSA-DMO* s *Lin-Kernighanovou heuristikou* a *Iterovanou Lin-Kernighanovou heuristikou*.

Kvôli tomu, ako je definovaná operácia *overlap* nebolo možné vykonať testovanie na reálnych DNA sekvenciách, nakoľko takéto sekvencie obsahujú chyby, ktoré boli do nich včlenené v priebehu sekvenovania. Tento problém možno v budúcnosti obísť reimplementáciou operácie *overlap* tak, aby na vyhodnotenie prekrytí používala Smith-Watermanov algoritmus zarovnávanie DNA sekvencií.

5.1 Testovanie PIMCSA-DMO s použitím TSPLIB inštancií

V tejto časti sa venujeme testovaniu algoritmu PIMCSA-DMO z hľadiska jeho schopnosti riešiť symetrické TSP inštancie. Testované inštancie z repozitára TSPLIB sú prístupné na webovej stránke [50]. Pri testovaní sme tiež merali čas, za ktorý testované algoritmy našli finálnu cestu. Algoritmus PIMCSA-DMO (PD) sme porovnávali s *Lin-Kernighanovou heuristikou* (LK) a *Iterovanou Lin-Kernighanovou heuristikou* (ILK). Algoritmus PIMCSA sme však do porovnávanie nezahrnuli, z dôvodu jeho veľkých časových nárokov.

Testovanie sme vykonávali na klastri so 7 výpočtovými uzlami. Na každom uzle je osadený 12 jadrový procesor taktovaný na frekvencii 3 GHz a každý uzol má k dispozícii 23 GB operačnej pamäte. Celkovo sme vykonali 2640 testov na 24 konfiguráciách daných algoritmov (4 konfigurácie pre LK, 8 pre ILK a 12 konfigurácií pre PIMCSA-DMO). Testy sme vykonávali nad 11 symetrickými inštanciami a každý test sa opakoval 10 krát. Celkové výsledky testovania, vrátane najlepších, najhorších a priemerných cien ciest, spolu s časmi behov jednotlivých algoritmov sú vo formáte excelovských tabuliek prítomné na priloženom DVD nosiči.

V nasledujúcich tabuľkách sme zaznamenávali len cenu najlepšej cesty, ktorú daný algoritmus s danou konfiguráciou v priebehu 10 behov vyprodukoval. Ak nejaký algoritmus dosiahol optimálnu cenu výslednej cesty, vyznačili sme príslušnú bunku tabuľky hrubým písmom. Pomocou tabuliek sme tiež uvádzali konfigurácie jednotlivých algoritmov, pre viac informácií o jednotlivých nastaveniach odkazujeme na prílohu C. Číslo v názve každej inštancie TSP označuje počet miest inštancie.

Lin-Kernighanova heuristika : Lin-Kernighanovu heuristiku sme testovali v 4 konfiguráciách. Jednotlivé konfigurácie Lin-Kernighanovej heuristiky (LK) zobrazuje tabuľka 1. Výsledky riešenia TSP pomocou LK uvádza tabuľka 2. Ako je možné vidieť, dosiahnuť optimálnu cestu sa podarilo len v prípade inštancie *berlin52* pre konfigurácie LK(1), LK(2) a LK(3). Základná Lin-Kernighanova heuristika preto sama o sebe nie je vhodná na riešenie väčších inštancií TSP.

Tabuľka 1: Konfigurácie Lin-Kernighanovej heuristiky použité pri testovaní riešenia TSP.

nastavenie	LK(1)	LK(2)	LK(3)	LK(4)
LK_MAX_DEPTH	20	30	30	50
LK_NEIGHBORHOOD_SIZE	20	30	50	80

Tabuľka 2: Výsledky riešenia TSP pomocou Lin-Kernighanovej heuristiky.

inštancia	optimum	LK(1)	LK(2)	LK(3)	LK(4)
eil51	426	436	431	427	429
eil101	629	631	630	632	636
berlin52	7542	7542	7542	7637	7542
a280	2579	2624	2624	2624	2624
lin318	42029	43455	42678	42999	43014
pr439	107217	108566	109358	107991	108679
pr1002	259045	263131	266268	268925	267426
rl1323	270199	282321	278860	278539	276271
pcb3038	137694	141314	140481	141060	140440
rl5915	565530	598004	586266	583183	582941
rl11849	923288	974100	955960	952682	953997

Iterovaná Lin-Kernighanova heuristika : Iterovanú Lin-Kernighanovu heuristiku (ILK) sme testovali celkovo v 8 konfiguráciách. Nastavenia jednotlivých konfigurácií sú zhrnuté v tabuľke 3.

Tabuľka 3: Konfigurácie Iterovanej Lin-Kernighanovej heuristiky použité pri testovaní riešenia TSP.

nastavenie	ILK(1)	ILK(2)	ILK(3)	ILK(4)	ILK(5)	ILK(6)	ILK(7)	ILK(8)
LK_MAX_DEPTH	20	30	30	50	20	30	30	50
LK_NEIGHBORHOOD_SIZE	20	30	50	80	20	30	50	80
ILK_ITERATIONS	10	10	10	10	20	20	20	20

Výsledky hľadania ciest v jednotlivých TSP inštanciách pomocou ILK uvádza tabuľka 4. Optimálnu cenu cesty sa pre inštanacie *berlin52* a *a280* podarilo dosiahnuť každej konfigurácii ILK. Mnohé z konfigurácií ILK boli úspešné i pri riešení inštancií *eil51* a *eil101*.

Tabuľka 4: Výsledky riešenia TSP pomocou Iterovanej Lin-Kernighanovej heuristiky.

inštančia	optimum	ILK(1)	ILK(2)	ILK(3)	ILK(4)	ILK(5)	ILK(6)	ILK(7)	ILK(8)
eil51	426	427	427	427	427	426	426	426	426
eil101	629	629	629	629	630	629	629	629	629
berlin52	7542	7542	7542	7542	7542	7542	7542	7542	7542
a280	2579	2579	2579	2579	2579	2579	2579	2579	2579
lin318	42029	42308	42297	42307	42154	42071	42133	42246	42222
pr439	107217	107500	107466	107409	107403	107340	107384	107305	107305
pr1002	259045	261278	261744	262302	261600	261513	261194	260655	261498
rl1323	270199	271695	271405	271354	271990	271323	271755	271237	271837
pcb3038	137694	139221	139197	139477	139182	139030	139065	139333	139049
rl5915	565530	575361	573588	574212	575112	571503	570920	572073	572573
rl11849	923288	943581	939944	937746	938333	937800	935606	936882	936861

PIMCSA-DMO : Algoritmus PIMCSA-DMO (PD) sme testovali celkovo v 4 konfiguráciách, pričom každú konfiguráciu sme spustili postupne s 8, 14 a 20 podpopuláciami P_i . Tabuľka 5 znázorňuje rozdiely jednotlivých nastavení algoritmu v daných konfiguráciách. Ostatné parametre boli rovnaké pre všetky konfigurácie, pričom boli nastavené nasledovne: PIA_POPULATION = 20, VACCINE_SET = 8, CLONE_SCALE = 5, PERTURBATE = TRUE a CROSS_VACC_P = 0.75

Tabuľka 5: Konfigurácie PIMCSA-DMO použité pri testovaní riešenia TSP.

nastavenie	PD(A)	PD(B)	PD(C)	PD(D)
LK_MAX_DEPTH	20	20	20	20
LK_NEIGHBORHOOD_SIZE	20	50	50	50
ILK_ITERATIONS	1	1	5	5
STOP_ITER	10	10	10	20

Výsledky riešenia TSP pomocou jednotlivých konfigurácií algoritmu PIMCSA-DMO zachytáva tabuľka 6. Čísla v hlavičke tabuľky za názvami jednotlivých konfigurácií označujú počet podpopulácií P_i , s ktorými bola daná konfigurácia testovaná. Z tabuľky je možné vidieť, že algoritmus PIMCSA-DMO

bol úspešný pri riešení inštancií *eil51*, *eil101*, *berlin52* a *a280* v každej konfigurácii. Pre konfigurácie PD(C) a PD(D) bol úspešný aj v prípade inštancie *lin318*, pričom sa mu dobre darilo aj s inštanciou *pr439*.

Tabuľka 6: Výsledky riešenia TSP pomocou algoritmu PIMCSA-DMO.

inštancia	optimum	PD(A)-8	PD(A)-14	PD(A)-20	PD(B)-8	PD(B)-14	PD(B)-20
eil51	426	426	426	426	426	426	426
eil101	629	629	629	629	629	629	629
berlin52	7542	7542	7542	7542	7542	7542	7542
a280	2579	2579	2579	2579	2579	2579	2579
lin318	42029	42050	42029	42029	42029	42029	42029
pr439	107217	107219	107222	107219	107224	107217	107224
pr1002	259045	259355	259596	259730	259675	259308	259380
rl1323	270199	270752	270597	270277	270254	270226	270307
pcb3038	137694	138441	138559	138239	138628	138506	138519
rl5915	565530	570850	569677	568827	569272	568965	569043
rl11849	923288	933000	933571	933303	933376	932207	932936

inštancia	optimum	PD(C)-8	PD(C)-14	PD(C)-20	PD(D)-8	PD(D)-14	PD(D)-20
eil51	426	426	426	426	426	426	426
eil101	629	629	629	629	629	629	629
berlin52	7542	7542	7542	7542	7542	7542	7542
a280	2579	2579	2579	2579	2579	2579	2579
lin318	42029	42029	42029	42029	42029	42029	42029
pr439	107217	107219	107217	107217	107222	107217	107217
pr1002	259045	259217	259235	259095	259731	259594	259406
rl1323	270199	270267	270281	270226	270226	270267	270254
pcb3038	137694	138249	138285	138227	138354	138316	138292
rl5915	565530	567180	566711	566859	568279	567735	568386
rl11849	923288	929848	930046	929640	932058	931344	931366

Zo získaných výsledkov vyplýva, že z hľadiska kvality výslednej cesty v TSP sa najlepšie darilo nášmu algoritmu - PIMCSA-DMO. Algoritmus našiel vo viacerých prípadoch optimálnu cestu, v ostatných prípadoch ju len mierne prekročil. V prípade najväčšej inštancie *rl11849* s 11849 mestami mala najlepšia cesta, ktorú sa nám podarilo pomocou PIMCSA-DMO získať, dĺžku 929640, čo je o 0,688% viac ako dĺžka optimálnej cesty. Túto dĺžku sa nám podarilo získať v konfigurácii algoritmu PD(C)-20. Najhoršia dĺžka cesty pre túto inštanciu mala veľkosť 939267, čo je 1,731% nad optimom. Túto cestu sme získali pomocou konfigurácie PD(A)-8. Aj najhoršia dĺžka cesty pre inštanciu *rl11849* získaná pomocou PIMCSA-DMO je však lepšia, ako najlepšia cesta získaná pomocou LK a tiež konkuruje i najlepším cestám získaným pomocou ILK. V ostatných prípadoch dĺžka najhorších ciest dosiahnutých pomocou PIMCSA-DMO bola približne 1% nad optimom a dĺžka najlepších ciest maximálne niekoľko desiatin percent. Z výsledkov tiež vyplýva, že najlepšie výsledky sme získali pomocou konfigurácie

PD(C), keď sme počet podpopulácií stanovili na 20. Táto konfigurácia dosahuje lepšie výsledky (nie len pre najlepšie cesty, ale i pre najhoršie a priemerné dĺžky ciest) ako konfigurácia PD(D), ktorej parameter STOP_ITER je nastavený na 20, čo je 2x väčšia hodnota, ako v prípade PD(C). Tento parameter značí to, po koľkých cykloch bez zmeny najlepšej protilátky v pamäťovej populácii dôjde k ukončeniu programu.

Algoritmus PIMCSA-DMO je schopný už pre jednoduchšie konfigurácie, akou je napríklad PD(A), dosahovať dostatočne dobré výsledky, no pri zložitejších konfiguráciách sme schopní, za cenu zvýšeného výpočtového času, dosiahnuť ešte lepšie výsledky, najmä z pohľadu znížených rozdielov medzi najlepšími a najhoršími cestami a z hľadiska priemernej ceny výslednej cesty.

Z časového hľadiska bol, prirodzene, najlepší algoritmus LK, ktorý je zo všetkých prezentovaných algoritmov výpočtovo najjednoduchší. V tabuľke 7 sme uviedli časy behov jednotlivých algoritmov v milisekundách. Pre každý algoritmus sme uviedli najnižší čas jeho behu (označený pomocou -) a najvyšší čas jeho behu (označený pomocou +) bez ohľadu na to, s akou konfiguráciou bol dosiahnutý. Najnižšie časy však boli dosiahnuté s najjednoduchšími konfiguráciami a najväčšie s najzložitejšími konfiguráciami daných algoritmov. Algoritmus PIMCSA-DMO je približne 5 - 17x pomalší ako Iterovaná Lin-Kernighanova heuristika. Dĺžka behu najjednoduchšej a najzložitejšej konfigurácie PIMCSA-DMO činí približne dvojnásobný časový rozdiel.

Tabuľka 7: Časy behov jednotlivých algoritmov pri riešení TSPLIB inštancií.

inštancia	LK -	LK +	ILK -	ILK +	PD -	PD +
eil51	0	3	8	26	111	160
eil101	4	2	19	84	206	324
berlin52	1	5	38	95	227	354
a280	20	28	95	209	612	1086
lin318	45	71	259	668	1699	3004
pr439	59	79	580	2178	5535	9788
pr1002	208	315	1197	3514	12800	29291
rl1323	281	534	1955	5864	14454	28034
pcb3038	1067	1268	3682	7321	45633	88250
rl5915	4742	6080	20567	53518	231461	504557
rl11849	28673	27053	65318	115183	946199	1962648

Časy v tabuľke sú uvádzané v milisekundách. Pre každý algoritmus sme uviedli najlepší (-) a najhorší (+) čas, ktorý sa nám podarilo získať, bez ohľadu na to, s akou konfiguráciou algoritmu sme daný čas získali.

Pre potreby skladania DNA sekvencií je potrebné zvážiť, ako kvalitnú výslednú cestu potrebujeme. Dosiahnuť dostatočne kvalitnú cestu je možné už v relatívne krátkych časoch, no dosiahnuť čo i len o kúsok lepšiu cestu môže trvať podstatne dlhší čas. Navyše, modul *Compactor* aplikovaný v procese vytvárania kontigov môže značne kompenzovať neoptimálnosť horšej cesty.

5.2 Testovanie PIMCSA-DMO pomocou GenFrag inštancií

V tejto kapitole opisujeme testovanie algoritmu PIMCSA-DMO pomocou GenFrag inštancií [28]. Tieto inštancie TSP vznikli z DNA sekvencií za účelom testovania skladačov DNA sekvencií, ktoré riešia FAP pretypovaním na TSP. Prekrytia jednotlivých sekvencií boli získané pomocou Smith-Watermanovho algoritmu, pričom z nich boli zostavené symetrické TSP inštancie. Ku každej inštancii je prítomná aj množina DNA sekvencií, z ktorej bola daná inštancia získaná, a preto možno pomocou GenFrag vzoriek vykonať i samotné skladanie DNA sekvencií. Nakoľko v našom skladači nepoužívame Smith-Watermanov algoritmus, nebolo by možné získané výsledky porovnať s inými algoritmi. Preto sme sa rozhodli použiť už pripravené TSP inštancie. Tieto inštancie sme načítali pomocou objektu triedy `genfrag_parser` a vykonali ich následné riešenie. Pomocou opisu v článku [28] sme previedli cenu výslednej cesty na súčet prekrytí medzi jednotlivými sekvenciami, pričom vyšší súčet značí lepší algoritmus. Testy pomocou GenFrag inštancií sme vykonali pre Lin-Kernighanovu heuristiku, Iterovanú Lin-Kernighanovu heuristiku, Memetický algoritmus a PIMCSA-DMO v niekoľkých konfiguráciách, pričom každú konfiguráciu sme testovali 5 krát. Výsledky jednotlivých testov sú vo formáte excelovskej tabuľky prítomné na priloženom DVD nosiči. V tabuľke 8 uvádzame len výsledky najlepšieho a najhoršieho behu algoritmu PIMCSA-DMO v jeho strednej konfigurácii PD(F), pričom ho porovnávame s inými algoritmi. Optimálne ceny ciest sme vyznačili hrubým písmom, druhý najlepší algoritmus sme vyznačili kurzívou.

Tabuľka 8: Porovnanie PIMCSA-DMO s ostatnými algoritmi pri riešení GenFrag inštancií.

inštancia	optimum	PPSO+DE	QEGA	SA	PALS	SAX	PD(F)+	PD(F)-
x60189_4	11478	11478	<i>11476</i>	11478	11478	11478	11478	11478
x60189_5	14161	13642	<i>14027</i>	<i>14027</i>	14021	<i>14027</i>	14161	14161
x60189_6	18301	18301	<i>18266</i>	18301	18301	18301	18301	18301
x60189_7	21271	20921	21208	21271	21210	<i>21268</i>	21271	21271
m15421_5	38746	38686	38578	38583	38526	<i>38726</i>	38746	38746
m15421_6	48052	47669	47882	<i>48048</i>	<i>48048</i>	<i>48048</i>	48052	48052
m15421_7	55171	54891	55020	55048	55067	<i>55072</i>	55171	55171
j02459_7	116700	114381	116222	<i>116257</i>	115320	115301	116700	116700
acin1	47618	47264	47115	46955	46876	46865	47618	<i>47609</i>
acin2	151553	147429	144133	144705	144634	144567	<i>151548</i>	<i>151544</i>
acin3	167877	163965	156138	156630	156776	155789	<i>167870</i>	<i>167859</i>
acin5	163906	161511	144541	146607	146591	145880	<i>163896</i>	<i>163886</i>
acin7	180966	180052	155322	157984	158004	157032	<i>180946</i>	<i>180934</i>
acin9	344107	335522	322768	324559	325930	314354	<i>344085</i>	<i>344081</i>

Jednotlivé hodnoty predstavujú sumu prekrytí pozdĺž cesty v TSP a teda celkovú sumu prekrytí daného zloženia DNA sekvencií. Väčšie hodnoty značia lepší algoritmus. Pre algoritmus PIMCSA-DMO v konfigurácii PD(F) uvádzame prekrytia pre najlepší beh programu PD(F)+ a najhorší beh PD(F)-. Hrubo sú vyznačené optimálne hodnoty, kurzívou druhé najlepšie hodnoty.

Konfigurácia PD(F), ktorú sme zahrnuli do porovnania síce produkovala horšie výsledky ako konfigurácia PD(G), ktorá mala najhoršie výsledky porovnateľné s najlepšimi výsledkami konfigurácie PD(F), ale už tieto výsledky sú lepšie, ako všetky uvedené algoritmy, s ktorými sme PIMCSA-DMO porovnávali. Voľbou strednej konfigurácie pritom ušetríme výpočtový čas. Autori článku [28] porovnávali ich algoritmus Particle Swarm Optimization and Differential Evolution (PPSO+DE) s algoritmami QEGA a SA, ktoré sme opisovali v analýze a s algoritmami PALS a SAX. Jednotlivé optimálne hodnoty vypočítali podľa Lin-Kernighan-Helsgaun algoritmu a overili ich pomocou programu Concorde. V tabuľke 8 sme k tomuto porovnaniu doplnili výsledky PIMCSA-DMO. Ako vidno, algoritmus PIMCSA-DMO dosiahol vždy buď optimálny výsledok, alebo druhý najlepší výsledok, dokonca aj vo svojom najhoršom behu, pričom sme v tabuľke uviedli len stredne výkonnú konfiguráciu PD(F). Nami navrhnutý algoritmus PIMCSA-DMO tak prekonal všetky porovnávané algoritmy.

Tabuľka 9: Konfigurácie PIMCSA-DMO použité pri testovaní riešenia TSP pomocou GenFrag inštancií.

nastavenie	PD(E)	PD(F)	PD(G)
LK_MAX_DEPTH	20	20	80
LK_NEIGHBORHOOD_SIZE	20	50	80
ILK_ITERATIONS	1	1	10

Ostatné parametre jednotlivých konfigurácií PIMCSA-DMO boli spoločné: $PIA_POPULATION = 20$, $MIA_STOP_ITER = 10$, $VACCINE_SET = 8$, $CLONE_SCALE = 5$, $PERTURBATE = TRUE$ a $CROSS_VACC_P = 0.75$, pričom všetky konfigurácie boli spustené so 14 podpopuláciami P_i .

5.3 Testovanie aplikácie PIMCSA-DMO na skladanie DNA sekvencií

Predchádzajúce testy preukázali schopnosť algoritmu PIMCSA-DMO riešiť veľké inštancie TSP, pričom algoritmus dosahuje dostatočne kvalitné cesty. V tejto kapitole sa teda venujeme testovaniu skladania DNA sekvencií pomocou nami implementovaného skladača, s dôrazom vyhodnotiť úspešnosť algoritmu PIMCSA-DMO. Všetky testy boli vykonané na už opísanom *klastri*. Hlavne kvôli časovej náročnosti transformácie FAP na TSP sme volili predovšetkým menšie množiny krátkych sekvencií. Skladač sme testovali s použitím *Lin-Kernighanovej heuristiky* v konfigurácii LK(1) a LK(4), *Iterovanej Lin-Kernighanovej heuristiky* v konfigurácii ILK(1) a ILK(4) a PIMCSA-DMO v konfigurácii PD(A)-8 a PD(C)-20. Použité konfigurácie sú rovnaké, ako v prípade testovania TSP inštancií a boli opísané v kapitole 5.1. Beh skladača sme pre každú konfiguráciu daných algoritmov opakovali 10 krát, pričom sme merali čas potrebný na skladanie a cenu výslednej cesty v TSP. Kontigy, ktoré vznikli pri poslednom behu skladača v danej konfigurácii sme následne použili pri porovnaní jednotlivých prístupov.

5.3.1 Testovacie dáta

Testovacie krátke sekvencie sme generovali použitím vlastného *generátora DNA sekvencií* [47]. Tieto sekvencie sú, spolu s ich referenčnými genómami a príslušnými nastaveniami generátora sekvencií, prítomné na priloženom DVD. Tabuľka 10 uvádza opis základných vlastností jednotlivých vzoriek so sekvenciami. Všetky sekvencie mali *obsah GC báz* nastavený na 50% a boli sekvenované ako *jednoduché sekvencie*. Dĺžka sekvencií bola vo všetkých prípadoch konštantná. Všetky súbory so sekvenciami sme pred testovaním upravili pomocou vlastného *preprocesora DNA sekvencií*, ktorý odstránil duplicitné sekvencie, čím sme ich mohli použiť s našim skladačom DNA sekvencií. Alternatívou k tomuto kroku by bolo použiť priamo funkcionality modulu *Parser*, ktorá odstraňuje duplicitné sekvencie, ale nakoľko sme skladanie vykonávali opakovane, bolo časovo výhodnejšie sekvencie predspracovať. Keďže FAP predstavuje asymetrickú inštanciu TSP, skladanie DNA sekvencií v danej vzorke vždy viedlo k riešeniu TSP s dvojnásobným počtom miest oproti počtu sekvencií v súbore.

Tabuľka 10: Konfigurácie generovaných testovacích DNA sekvencií.

vzorka	CHR	GS	T	REP	RL	ER	COV	CNT
A	1	100	V	N	30	0	40	133 (57)
B	1	1000	V	N	70	0	40	571 (408)
C	1	1000	V	N	70	0	100	1666 (759)
D	1	500	V	N	70	0.000001	50	357 (223)
E	2	200 + 100	V	N	40	0	50	377 (150)
F	1	200	M	N	40	0	50	250 (153)
G	1	300	V	N	40	0	5	37 (31)
H	1	1000	V	T	50	0	30	600 (408)
I	1	1000	V	C	50	0	30	600 (409)
J	1	10000	V	N	100	0	40	4000 (3256)
K	1	20000	V	N	100	0	40	8000 (6472)
L	1	100000	V	N	100	0	10	10000 (8644)
M	1	100000	V	N	100	0	30	30000 (19650)

Legenda: *CHR* - počet chromozómov, *GS* - veľkosť genómu, *T* - typ genómu (*V* - jednoduché vlákno, *M* - molekula), *REP* - typ opakovaní (*N* - žiadne, *T* - teloméry, *C* - centroméra), *RL* - dĺžka sekvencií (všetky sekvencie majú rovnakú veľkosť), *ER* - pravdepodobnosť vzniku chyby pri sekvenovaní, *COV* - pokrytie, *CNT* - počet sekvencií (v zátvorke je počet sekvencií po odstránení duplicitných sekvencií)

5.3.2 Správnosť skladania sekvencií

V tejto časti testujeme, či náš skladač DNA sekvencií je schopný skladať sekvencie do kontigov, ktoré sa zhodujú s pôvodne sekvenovanými vláknami DNA. Skladač DNA sekvencií bol pre každú testovanú konfiguráciu nastavený tak, aby nepoužíval modul *Compactor*, čím sme zachovali vyšší počet miest v každej TSP inštancii. Modulom *Compactor* neboli spájané ani kontigy, ktoré vznikli po skladaní

príslušných DNA sekvencií, čo nám umožnilo testovať len silu daného algoritmu pre riešenie TSP. V module *Connector* sme nastavili minimálny počet nukleotidov, ktorý musia dve sekvencie spoločné na to, aby boli zahrnuté do jedného kontigu na 8 (nastavenie `CONNECTOR_OVERLAP_LIMIT`). Správnosť skladania sme overili pomocou programu *FASTA* [34], ktorý nám umožnil výsledné kontigy zarovnať podľa referenčnej vzorky, z ktorej sme krátke sekvencie generovali a vyhodnotiť počet správne a nesprávne poskladaných nukleotidov. Tabuľka 11 uvádza výsledky skladania DNA sekvencií.

Tabuľka 11: Výsledky skladania DNA sekvencií.

vzorka	dĺžka	LK(1)	LK(4)	ILK(1)
A	100	3/100/100	1/100/100	1/100/100
B	1000	5/998/998	2/998/667(331)	1/998/998
C	1000	5/1000/1000	4/1000/1000	1/1000/1000
D	500	6/495/495	6/496/495(1)	4/495/495
E	300	5/299/299	4/299/299	3/299/299
F	400	3/396/396	3/448/280(168)	3/396/396
G	300	2/297/297	5/297/297	2/297/297
H	1000	4/914/914	2/917/913(4)	1/916/916
I	1000	4/947/947	2/1009/970(39)	3/967/944(23)
J	10000	9/10024/9999(25)	1/9999/9999	1/9999/9999
K	20000	3/20056/11810(8246)	3/20098/17982(2116)	1/19996/19996
L	100000	16/33472/22065(11407)	13/32865/32865	4/32865/32865

vzorka	dĺžka	ILK(4)	PD(A)-8	PD(C)-20
A	100	1/100/100	1/100/100	1/100/100
B	1000	1/998/998	1/998/998	1/998/998
C	1000	2/1000/1000	1/1000/1000	1/1000/1000
D	500	4/495/495(32)	4/495/495	4/495/495
E	300	2/299/299	2/299/299	2/299/299
F	400	2/396/396	2/396/396	2/396/396
G	300	2/297/297	2/297/297	2/297/297
H	1000	1/912/912	1/910/910	1/919/910
I	1000	1/920/881(39)	1/920/883(37)	1/920/883(37)
J	10000	2/9999/9999	1/9999/9999	1/9999/9999
K	20000	1/19996/19996	1/19996/19996	1/19996/19996
L	100000	4/32865/32865	1/32865/32865	1/32865/32865

Legenda: Tabuľka zobrazuje výsledky skladania DNA sekvencií pre 3 algoritmy na riešenie TSP. Dáta v tabuľke sú zapísané vo formáte $a/b/c(d)$, kde a značí počet kontigov generovaných skladačom, b značí celkovú dĺžku kontigov, pričom dbáme na to, aby prekrývajúce sa časti kontigov boli do tohto súčtu započítané len raz. Číslo c označuje celkový počet nukleotidov, ktoré sa zhodujú s referenčným genómom a číslo d označuje počet chybné zložených nukleotidov. Stĺpec dĺžka určuje dĺžku každého referenčného genómu v nukleotidoch.

V prípadoch, keď náš skladač poskladal viac ako jeden kontig (bez ohľadu na použitý algoritmus), sa tieto kontigy zvyčajne, aspoň z malej časti prekrývali. Väčšinou by tak bolo možné tieto kontigy spojiť

do jedného *superkontigu*, v ostatných prípadoch aspoň redukovať ich počet. Vyšší počet kontigov je spôsobený najmä neoptimálnosťou nájdenej TSP cesty (keďže na skladanie DNA sekvencií používame heuristické algoritmy, výsledné cesty sú len suboptimálne).

Pri vzorke *B* sme boli schopní poskladať vždy len 998 z 1000 nukleotidov, čo je spôsobené tým, že prvé 2 nukleotidy nie sú prítomné ani v jednej krátkej sekvencii, podobne to platí pre druhý chromozóm vzorky *E*, kde sme neboli schopní zrekonštruovať viac ako 99 zo 100 nukleotidov a jeden z reťazcov vzorky *F*, kde bolo osekvenovaných len 196 nukleotidov.

Zaujímavý je prípad vzorky *B*, kde konfigurácia LK(1) dosiahla z hľadiska skladania DNA sekvencií lepší výsledok, ako konfigurácia LK(4) i napriek tomu, že pre túto konfiguráciu sme získali cestu ceny 513 oproti LK(4), kde mala cesta cenu 249. Vyššia neoptimálnosť cesty spôsobila v prípade LK(1) vyšší počet kontigov, ktoré ale pokrývajú referenčný genóm lepšie, ako 2 chybne zložené kontigy konfigurácie LK(4). V tabuľke možno vidieť, že podobné situácie sa opakovali viac krát aj pre iné konfigurácie testovaných algoritmov.

V prípade vzorky *D*, kde boli sekvencie generované s možnosťou výskytu chyby počas sekvenovania, sme boli schopní zrekonštruovať vždy takmer celý reťazec DNA. Chyby sekvenovania ale mali za následok zvýšený počet kontigov pre každý testovaný algoritmus. V skladaní najlepšie dopadli konfigurácie PD(A) a PD(C), ktoré vygenerovali jeden kontig dĺžky 496 nukleotidov, pričom zvyšné 3 kontigy boli jeho podmnožinami. Algoritmus PIMCSA-DMO tak aj v tomto prípade preukázal svoju vhodnosť na skladanie DNA sekvencií.

Pri skladaní sekvencií s malým pokrytím, vo vzorke *G*, mala väčšie problémy len konfigurácia LK(4), ktorá zložila až 5 kontigov. Tieto však pokrývali 297 nukleotidov referenčného genómu rovnako, ako v prípade ostatných konfigurácií. V tomto prípade ale cena cesty konfigurácie LK(4) bola 372, oproti nižšej cene cesty LK(1), ktorá bola 235. Konfigurácia LK(4) preto poskytovala horšie výsledky ako LK(1) už v procese riešenia TSP.

Celková dĺžka pokrytia referenčného genómu kontigmi pri sekvenciách *H* bola znížená v dôsledku prítomnosti telomérov, ktorých opakovacie jednotky boli síce prítomné, ale neboli zopakované správny počet krát. To je spôsobené tým, že v prípade skladania DNA pomocou TSP nevieme správne určiť dĺžky jednotlivých opakovaní. I napriek tomu, že algoritmus PIMCSA-DMO dosahoval lepšiu cenu cesty v TSP, algoritmus Iterovanej Lin-Kernighanovej heuristiky v konfigurácii ILK(1) dosiahol z pohľadu skladania DNA sekvencií lepší výsledok.

V prípade skladania DNA sekvencií zo vzorky *I*, ktorá vo svojom strede obsahuje centromerické opakovanie, bolo potrebné v prípadoch, keď sme získali len 1 kontig, tento kontig rozdeliť na 2 časti, ktoré potom bolo možné bez závažnejších chýb mapovať na referenčný genóm vzorky *I*.

Vzorky J a K aj napriek svojej veľkosti nerobili väčšine konfigurácií pri skladaní problém. Najväčší prínos algoritmu PIMCSA-DMO vidno až na vzorke L , kde ako jediný dokázal vzorku poskladať do jedného kontigu, ktorý ale dosahoval len tretinu dĺžky referenčného genómu. S rovnakým problémom sme sa však v minulosti stretli aj pri skladaní dlhších chromozómov pomocou skladača ABySS, ktorý na skladanie DNA sekvencií používa De Bruijnove grafy.

Sekvencie zo vzorky M sme testovali len s konfiguráciou PD(A)-8 algoritmu PIMCSA-DMO. V tomto prípade sme získali 1 kontig dĺžky 32863 nukleotidov. Skladanie týchto sekvencií predstavovalo riešenie TSP s 39300 mestami, pričom algoritmus PIMCSA-DMO vyprodukoval suboptimálnu cestu s cenou -6437 za 2873 sekúnd. Celkové skladanie trvalo 3052 sekúnd.

Tabuľka 12 boduje úspešnosť jednotlivých testovacích konfigurácií pre dané vzorky. Každá konfigurácia mohla dosiahnuť maximálne 100 bodov. Body sme pre danú konfiguráciu počítali podľa vzorca $(a - b) * 100 / N - i$, kde a predstavuje počet správne určených nukleotidov, b počet nesprávne určených nukleotidov, N počet nukleotidov referenčného genómu a i penalizáciu za vyšší počet kontigov, pričom $i = 0$ pre konfiguráciu s najmenším počtom kontigov, $i = 10$ pre konfiguráciu s 2. najmenším počtom kontigov, $i = 15$ pre konfiguráciu s 3. najmenším počtom kontigov a $i = 20$ pre konfigurácie s najvyšším počtom kontigov. Najlepšie bodovania sme v tabuľke vyznačili hrubým písmom. Výsledné skóre danej konfigurácie predstavuje počet situácií, v ktorých bola daná konfigurácia v porovnaní s ostatnými najlepšia.

Tabuľka 12: Bodovanie jednotlivých konfigurácií algoritmov na riešenie TSP.

vzorka	LK(1)	LK(4)	ILK(1)	ILK(4)	PD(A)	PD(C)
A	90	100	100	100	100	100
B	84,8	23,6	99,8	99,8	99,8	99,8
C	80	85	100	90	100	100
D	89	88,8	99	92,6	99	99
E	79,7	84,7	89,7	99,7	99,7	99,7
F	89	18	89	99	99	99
G	99	89	99	99	99	99
H	76,4	80,9	91,6	91,2	91	91
I	74,7	83,1	77,1	84,2	84,6	84,6
J	84,7	100	100	90	100	100
K	7,8	69,3	100	100	100	100
L	0	17,9	22,9	22,9	32,9	32,9
skóre	1	2	8	5	11	11

Algoritmus PIMCSA-DMO sa v oboch svojich konfiguráciách umiestnil na prvom mieste. Práve tento algoritmus produkuje najmenší počet najdlhších kontigov s najväčším počtom správne umiestnených nukleotidov.

5.3.3 Časy behu skladača DNA sekvencií

Tabuľka 13 zobrazuje časy behov jednotlivých konfigurácií testovaných algoritmov pre riešenie TSP v rámci skladača DNA sekvencií. Časy v tabuľke sme vypočítali spriemerovaním časov z 10 behov každej konfigurácie. Z tabuľky možno vidieť, že najrýchlejší je algoritmus Lin-Kernighanovej heuristiky a najpomalší PIMCSA-DMO. Pomalosť PIMCSA-DMO je však vyvážená jeho vysokou presnosťou pri riešení TSP a teda i pri skladaní DNA sekvencií.

Tabuľka 13: Časy behov skladača DNA sekvencií pre rôzne vzorky.

vzorka	LK(1)	LK(4)	ILK(1)	ILK(4)	PD(A)	PD(C)
A	4(4)	15(15)	68(68)	44(43)	346(345)	765(765)
B	177(124)	106(47)	933(878)	783(731)	3032(2977)	8143(8089)
C	431(275)	502(327)	1805(1642)	3105(2941)	6991(6835)	16481(16313)
D	106(89)	114(101)	831(818)	1020(1004)	4024(4012)	8746(8730)
E	20(16)	41(38)	279(275)	258(253)	1151(1145)	2931(2927)
F	13(10)	14(13)	99(92)	107(104)	579(575)	1349(1342)
G	1(1)	1(1)	3(3)	9(9)	98(98)	120(120)
H	94(55)	87(51)	519(482)	421(383)	2112(2076)	5583(5545)
I	154(114)	104(65)	890(853)	619(578)	2456(2416)	6901(6860)
J	6865(2021)	6828(1980)	9241(4450)	10713(5933)	32509(27623)	44791(39654)
K	27672(8633)	25874(6909)	32187(13150)	34905(15920)	113639(94328)	135277(115628)
L	47088(13187)	47170(13152)	53837(19817)	59321(25473)	181836(147220)	217081(181169)

Legenda: Čas je udávaný v milisekundách vo formáte $a(b)$, kde a predstavuje celkový čas potrebný na skladanie DNA sekvencií a b čas hľadania cesty v TSP.

Časové rozdiely medzi PIMCSA-DMO a inými algoritmami sú najväčšie pri malých množinách DNA sekvencií, pri veľkých množinách (vzorky J , K a L) sa tento rozdiel pomaly stráca, najmä kvôli časovej náročnosti prevodu FAP na TSP v module *FASTA Parser*. Preto je vhodné použiť PIMCSA-DMO práve na skladanie väčších vzoriek, kde sú časové rozdiely oproti najrýchlejšej Lin-Kernighanovej heuristike len približne štvornásobné. Za cenu mierne zvýšených časových nárokov tak získame podstatne kvalitnejšie kontigy.

5.4 Vplyv modulu Compactor na redukcii dimenzionality TSP

V tejto kapitole sme otestovali, ako vplýva modul *Compactor* na celkový počet miest TSP. *Kritický pomer* sme nastavili na 0,9 a spustili sme skladač pre opísané vzorky krátkych DNA sekvencií. V tabuľke 14 sú uvedené rozmery inštancií TSP v počte miest, ktoré musel skladač riešiť v prípade, že sme ho spúšťali bez použitia modulu *Compactor* a v prípade, že sme ho spustili s modulom *Compactor*. Z riadku *kompresia* je zrejmé, že modul *Compactor* dokáže podstatne zredukovať počet

miest v jednotlivých inštanciách TSP, čo sa najviac odrazí pri genómoch, ktoré boli sekvenované s veľkým pokrytím. Vďaka modulu Compactor by sme tak boli schopní riešiť podstatne väčšie množiny DNA sekvencií, ako bez neho.

Tabuľka 14: Rozmery inštancií TSP s použitím a bez použitia modulu Compactor.

vzorka	A	B	C	D	E	F	G	H	I	J	K	L
bez Compactora	114	816	1518	446	300	306	62	816	818	6512	12944	17288
s Compactorom	4	24	4	8	12	52	44	70	68	202	460	1124
kompresia	28,5	34	379,5	55,75	25	5,9	1,4	11,7	12	32,2	28,1	15,4

5.5 Porovnanie algoritmov PIMCSA-DMO a PIMCSA

Algoritmy PIMCSA-DMO a PIMCSA sú si značne podobné, nakoľko algoritmus PIMCSA-DMO vychádza priamo z PIMCSA. Úpravy, ktoré sme uskutočnili v PIMCSA-DMO sa sústredili najmä na zlepšenie časov behu oproti algoritmu PIMCSA. Do PIMCSA-DMO sme tiež zahrnuli Vylepšený Inver-Over operátor, ktorý produkuje lepšie výsledky ako základný Inver-Over operátor, preto má algoritmus PIMCSA-DMO potenciál dosahovať lepšie výsledky ako PIMCSA pri riešení veľkých inštancií TSP. Tabuľka 15 zachytáva časy behov konfigurácií PD(A)-8 a PD(C)-20 algoritmu PIMCSA-DMO a algoritmu PIMCSA v konfigurácii odpovedajúcej konfigurácii PD(A).

Tabuľka 15: Porovnanie času behov algoritmov PIMCSA-DMO a PIMCSA v ms.

vzorka	PIMCSA	PD(A)	PD(C)
A	711(711)	346(345)	765(765)
E	3808(3803)	1151(1145)	2931(2927)
F	3353(3348)	579(575)	1349(1342)
G	161(161)	98(98)	120(120)

Z nameraných výsledkov vyplýva, že algoritmus PIMCSA je oproti PIMCSA-DMO výrazne pomalší. Zaostáva i za výrazne výpočtovo náročnejšou konfiguráciou PD(C) algoritmu PIMCSA-DMO. Algoritmus PIMCSA-DMO je teda oproti PIMCSA pre skladanie DNA sekvencií výhodnejší minimálne z časového hľadiska. Kvalitou ciest v TSP sú algoritmy buď rovnaké, alebo PIMCSA-DMO produkuje lepšie cesty. Preto je najvhodnejšie na skladanie DNA sekvencií používať algoritmus PIMCSA-DMO.

6 Zhodnotenie a budúca práca

V rámci diplomového projektu sme implementovali *skladač DNA sekvencií*, ktorý rieši *problém skladania DNA sekvencií* jeho pretypovaním na *problém obchodného cestujúceho* (skr. TSP). Pri implementácii algoritmu riešiaceho TSP sme vychádzali z algoritmu *PIMCSA* [37] inšpirovaným imunitným systémom, ktorý sme vhodne modifikovali, čím sme vytvorili algoritmus *PIMCSA-DMO*. Najvýraznejšou zmenou bolo presunutie *operácie zrenia protilátok* z pamäťovej populácie do jednotlivých podpopulácií protilátok. Súčasťou tohto algoritmu je použitie *Iterovanej Lin-Kernighanovej heuristiky* a *Vylepšeného Inver-over operátora*, ktoré sme tiež vhodne implementovali.

Testovanie preukázalo dostatočnú silu algoritmu *PIMCSA-DMO* pri riešení veľkých TSP inštancií z hľadiska kvality výsledných ciest, ale aj z hľadiska času potrebného na nájdenie týchto ciest. Algoritmus *PIMCSA-DMO* kvalitou ciest prekonal všetky algoritmy, s ktorými sme ho porovnávali. Keďže testovanie skladača vyžadovalo jeho porovnanie s inými algoritmami, vyústilo to k návrhu takej architektúry skladača, ktorá z neho spravila plnohodnotný *framework pre skladanie DNA sekvencií* pomocou algoritmov pre riešenie TSP. Skladač pozostáva z troch hlavných modulov (*Parser*, *TSP Solver* a *Connector*), ktoré možno reimplementovať a pri dodržaní predpísaného rozhrania zameniť bez toho, aby to malo vplyv na iné časti skladača DNA sekvencií. Práve moduly *TSP Solver*, ktoré predstavujú rôzne prístupy k riešeniu TSP možno v skladači ľahko vymieňať, v čom spočíva najväčšia sila frameworku. Modul *Compactor* navyše skladaču umožňuje skladať podstatne väčšie množiny krátkych DNA sekvencií, nakoľko pri genómoch, ktoré boli sekvenované s dostatočným pokrytím sme schopní redukovať počet sekvencií aj viac ako stonásobne. Modul *Compactor* tak redukuje aj rozmery TSP inštancie, ktorá musí byť riešená pomocou *PIMCSA-DMO*.

Jedna z časovo najkritickejších činností skladača je výpočet prekrytí medzi všetkými dvojicami DNA sekvencií. Tento výpočet nie je implementovaný paralelne a spôsob, akým je implementovaný nedovoľuje, aby v DNA sekvenciách boli prítomné chyby vzniknuté v dôsledku sekvenovania daného genómu. V budúcnosti je preto vhodné reimplementovať funkciu *overlap(a, b)*, ktorá vracia veľkosť prekrytia medzi sekvenciami *a* a *b* tak, aby používala *Smith-Watermanov algoritmus* zarovnávanie sekvencií a modul *FASTA Parser* tak, aby výpočet prekrytí vykonával paralelne s použitím MPI a prípadne OpenMP. Autori článku [28] tiež uvádzajú spôsob, ako implementovať *Smith-Watermanov algoritmus* efektívnejšie.

V ďalšej práci by sme tiež chceli vylepšiť implementáciu *Lin-Kernighanovej heuristiky* a to najmä z časového hľadiska. Keďže implementácia efektívnej *Lin-Kernighanovej heuristiky* nie je jednoduchá záležitosť, je vhodné zvážiť integráciu algoritmu *Lin-Kernighan-Helsgaun* (skr. LKH), ktorý v súčasnosti predstavuje najefektívnejšiu a kvalitou ciest najlepšiu implementáciu *Lin-Kernighanovej heuristiky*. Zdrojové kódy tohto algoritmu sú voľne dostupné. Algoritmus LKH je implementovaný v

jazyku C a poskytuje funkcie, ktoré možno volať v iných programoch. Pre potreby skladania DNA sekvencií pomocou nášho skladača by však bolo pravdepodobne potrebné modifikovať samotný zdrojový kód LKH algoritmu, nakoľko je potrebné, aby dokázal pracovať s objektom triedy `stsp_from_atsp`, ktorá neudržiava celú symetrickú maticu reprezentujúcu TSP.

Aby bol framework viacúčelovejší, chceli by sme tiež zmeniť implementáciu triedy `dna_assembler` tak, aby aj s modulmi *Parser*, *Connector* a *Saver* pracovala podobne, ako s modulom *TSP Solver*. Tak by bolo možné jednotlivé moduly vymieňať bez nutnosti meniť zdrojový kód triedy `dna_assembler`.

Pevne veríme, že nami vyvinutý skladač DNA sekvencií poslúži v budúcnosti nie len študentom Fakulty informatiky a informačných technológií Slovenskej technickej univerzity pri riešení ich diplomových projektov, ale aj výskumníkom v oblasti skladania DNA sekvencií pomocou algoritmov pre riešenie TSP. Z toho dôvodu sme rozhodnutí výsledky našej práce publikovať a tak sprístupniť nami vytvorený skladač DNA sekvencií.

Literatúra

- [1] ALBERTS, B., JOHNSON, A. LEWIS, J. et al.: The Mitochondrion. In: Molecular Biology of the Cell. 4th edition. New York: Garland Science, 2002. [Dostupné na <http://www.ncbi.nlm.nih.gov/books/NBK26894/>]
- [2] ANSORGE, W.: Next-generation DNA sequencing techniques. In: New Biotechnology, vol. 25, 2009, no. 4, pp. 195–203.
- [3] APPELEGATE, D., BIXBY, R., CHVÁTAL, V., COOK, W.: The Traveling Salesman Problem. 2006. ISBN: 0-691-12993-2.
- [4] Arachne Mediawiki. Homopolymer. <https://www.broadinstitute.org/crd/wiki/index.php/-Homopolymer>. [Navštívené 27.3.2015]
- [5] BENTLEY, D. R., BALASUBRAMANIAN, S., SWERDLOW P, et al.: Accurate Whole Human Genome Sequencing using Reversible Terminator Chemistry. In: Nature, vol. 456, 2008, no. 7218, pp. 53–59.
- [6] The Biotechnology Education Company.: Principles and Practice of Agarose Gel Electrophoresis. 2003. [Dostupné na: [http://www.wou.edu/las/physci/ch462/Gel Electrophoresis.pdf](http://www.wou.edu/las/physci/ch462/Gel%20Electrophoresis.pdf)]
- [7] COCK, P., FIELDS, CH., GOTO, N., et al.: The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. In: Nucleic Acids Research, vol. 28, 2010, no. 6, pp. 1767–1771.
- [8] COMPEAU, P., PEVZNER, P., TESLER, G.: How to apply de bruijn graphs to genome assembly. In: Nature Biotechnology, vol. 29, 2010, no. 11, pp. 987–991.
- [9] DORIGO, M., GAMBARDELLA L. M.: Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. In: IEEE Transactions on Evolutionary Computation, vol. 1, 1997, pp. 53–66.
- [10] EKBOLM, R., WOLF, J.: A field guide to whole-genome sequencing, assembly and annotation. In: Evolutionary Applications, 2014.
- [11] Ensembl. Whole Genome. http://useast.ensembl.org/Homo_sapiens/Location/Genome. [Navštívené 13.11.2015]
- [12] FAKRUDDIN, MD., CHOWDHURY, A., HOSSAIN, N. MD. et al.: Pyrosequencing - principles and applications. In: International Journal of Life science and Pharma Research, vol. 2, 2012, pp. L-65 – L-76.
- [13] FIROZ, J., RAHMAN, S., SAHA, T.: Bee Algorithms for Solving DNA Fragment Assembly Problem with Noisy and Noisless Data. In: Proceedings of GECCO, 2012.

- [14] GALLANT, J.: On Finding Minimal Length Superstrings. In: Journal of Computer and System Sciences, vol. 20, 1980, pp. 50–58.
- [15] GAPLOVSKÝ, M., KAPLOVSKÁ-KYSELÁ, K.: Princípy nových metód sekvenovania DNA. In: Chemické Listy, vol. 106, 2012, pp. 809–817.
- [16] Genomatix. DNA Sequence formats. http://www.genomatix.de/online_help/help/-sequence_formats.html. [Navštívené 1.4.2015]
- [17] GLICKMAN, B., RIPLEY, L.: Structural intermediates of deletion mutagenesis: A role for palindromic DNA. In: Genetics, vol. 81, 1984, pp. 512–516.
- [18] HRUBÍKOVÁ, K., BEŽO, M., ŽIAROVSKÁ, J., CANDRÁKOVÁ, A.: Retrotransponzóny, ich význam a využitie v mapovaní genómu organizmov. In: Acta fytotechnica et zootechnica, mimoriadne číslo, 2006, pp. 74–77.
- [19] Illumina, Inc. Illumina Sequencing Technology. http://res.illumina.com/documents/products/-techspotlights/techspotlight_sequencing.pdf. 2010.
- [20] Integrated DNA Technologies. The Polymerase Chain Reaction. <http://www.idtdna.com/pages/docs/educational-resources/the-polymerase-chain-reaction.pdf>. 2011.
- [21] International Human Genome Sequencing Consortium: Initial sequencing and analysis of the human genome. In: Nature, vol. 409, 2001, pp. 860–921.
- [22] JACKSON, B.: Parallel Methods for Short Read Assembly. Iowa: Iowa State University, 2009. Dizertačná práca.
- [23] JOHNSON, D. S.: Local optimization and the Traveling Salesman Problem. In: Lecture Notes in Computer Science, vol. 443, 2005, pp. 446 – 461.
- [24] LI, H., HANDSAKER, B., WYSOKER, A. et al.: The Sequence Alignment/Map format and SAMtools. In: Bioinformatics, vol. 25, 2009, no. 16, pp. 189–203.
- [25] LIN, L., YINHU, L., SILIANG, L. et al.: Comparison of Next-Generation Sequencing System. In: Journal of Biomedicine and Biotechnology, 2012.
- [26] LUQUE, G., ALBA, E.: Metaheuristics for the DNA Fragment Assembly Problem. In: International Journal of Computational Intelligence Research, vol. 1, 2005, no. 2, pp. 98–108.
- [27] LUQUE, G., ALBA, E.: A New Local Search Algorithm for the DNA Fragment Assembly Problem. In: Evolutionary Computation in Combinatorial Optimization, 2007, pp. 1–12.
- [28] MALLÉN-FULLERTON, G. M., HUGHES, J. A., HOUGHTEN, S., FERNÁNDEZ-ANAYA, G.: Benchmark Data Sets for the DNA Fragment Assembly Problem. In: International Journal of Bio-inspired Computation, vol. 5, 2013, no. 6, pp. 384–394.

- [29] Mediawiki. DNA replikácia. http://upload.wikimedia.org/wikipedia/commons/7/75/-DNA_replication_blank.svg. [Navštívené 27.3.2015]
- [30] MEKSANGSOUY, P., CHAIYARATANA, N.: DNA Fragment Assembly Using an Ant Colony System Algorithm. In: *Evolutionary Computation*, vol. 3, 2003, pp. 1756–1763.
- [31] MIŠÚROVÁ, E., SOLÁR, P.: *Molekulová biológia*. Košice: Univerzita Pavla Jozefa Šafárika, 2007. ISBN: 9788070976715.
- [32] New England Biolabs Inc. DNA Fragmentation. <https://www.neb.com/applications/library-preparation-for-next-generation-sequencing/dna-fragmentation>. [Navštívené 3.12.2015]
- [33] NISHIGUCHI, M., DOUKAKIS, P. et al.: DNA Isolation Procedures. In: *Methods and Tools in Biosciences and Medicin*, 2002, pp. 250–281.
- [34] PARSON, W. R., LIPMAN, D. J.: Improved tools for biological sequence comparison. In: *PNAS*, vol. 85, 1988, no. 8, pp. 2444 – 2448.
- [35] PARSONS, R., FOTTEST, S., BURKS, CH.: Genetic Algorithms for DNA Sequence Assembly. In: *ISMB-93 Proceedings*, 1993, pp. 310–318.
- [36] PEVZNER, P., TANG, H., WATERMAN, M.: An Eulerian path approach to DNA fragment assembly. In: *PNAS*, vol. 98, 2001, no. 17, pp. 9748–9753.
- [37] QI, Y., LIU, F., JIAO, L.: A Parallel Artificial Immune Model for Optimization. In: *International Conference on Computational Intelligence and Security*, 2009, pp. 20–24.
- [38] RATHESH, K., HAOMINI, L.: *On Asymmetric TSP: Transformation to Symmetric TSP and Performance Bound*. Kentucky: Department of Electrical Engineering University of Kentucky.
- [39] Roche Diagnostics Corporation. Whole Genome Sequencing. <http://454.com/applications/whole-genome-sequencing/>. [Navštívené 29.3.2015]
- [40] ROSYPAL, S. et al.: *Úvod do molekulární biologie. Díl I. Molekulární biologie prokaryotické buňky*. 3. preprac. vyd. Brno: S. Rosypal, 1998. ISBN: 8090256201.
- [41] ROSYPAL, S. et al.: *Úvod do molekulární biologie. Díl II. Molekulární biologie eukaryot*. 3. preprac. vyd. Brno: S. Rosypal, 1999. ISBN: 809025621X.
- [42] ROSYPAL, S. et al.: *Úvod do molekulární biologie. Díl III. Molekulární biologie virů*. 3. preprac. vyd. Brno: S. Rosypal, 2000. ISBN: 8090256228.
- [43] The SAM Format Specification Working Group: *The SAM Format Specification*. 2011.
- [44] SANGER, F., NICKLEN, S., COULSON, A. R.: DNA sequencing with chain-terminating inhibitors. In: *Biochemistry*, vol. 74, 1977, no. 12, pp. 5463–5467.

- [45] Scilico, LLC. IUPAC nucleotide codes. <http://www.bioinformatics.org/sms/iupac.html>. [Navštívené 1.4.2015]
- [46] SCHADT, E., TURNER, S., KASARSKIS, A.: A window into third-generation sequencing. In: Human Molecular Genetics, vol. 19, 2010.
- [47] ŠIMEK, J.: Generovanie DNA sekvencií. Bratislava: FIIT STU, 2014. Bakalárska práca.
- [48] TAO, G., MICHALEWICZ, Z.: Inver-over Operator for the TSP. In: Lecture Notes in Computer Science, vol. 1498, 2006, pp. 803–812.
- [49] TAYLOR, L. Overlap Layout Consensus. <http://gcat.davidson.edu/phast/olc.html>. [Navštívené 1.4.2015]
- [50] Universität Heidelberg. TSPLIB. <http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/>. [Navštívené 22.4.2016]
- [51] University of Waikato. Cell, chromosomes and DNA. <http://www.sciencelearn.org.nz/Contexts/Uniquely-Me/Sci-Media/Images/Cell-chromosomes-and-DNA>. [Navštívené 27.3.2015]
- [52] USSERY, D., WASSENAAR, T., BORINI, S.: Computing for Comparative Microbial Genomics: Bioinformatics for Microbiologists. Springer, 2009. ISBN: 978-1-84800-254-8.
- [53] WANG, Y., LI, J., GAO, K., PAN, Q.: Memetic Algorithm based on Improved Inver-over operator and Lin-Kernighan local search for the Euclidean traveling salesman problem. In: Computers and Mathematics with Applications, 2011, pp. 2743 – 2754.
- [54] WATSON, J., CRICK, F.: Molecular Structure of Nucleic Acids. In: Nature, vol. 171, 1953, no. 4356, pp. 737–738.