UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Blaž Poje

Zaporedni in vzporedni genetski algoritmi za reševanje problema trgovskega potnika

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: doc. dr. Tomaž Dobravec

Ljubljana, 2013

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01951 / 2013
Datum: 2.9.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **BLAŽ POJE**

Naslov: **ZAPOREDNI IN VZPOREDNI GENETSKI ALGORITMI ZA REŠEVANJE PROBLEMA TRGOVSKEGA POTNIKA**
SEQUENTIAL AND PARALLEL GENETIC ALGORITHMS FOR SOLVING TRAVELLING SALESMAN PROBLEM

Vrsta naloge: DIPLOMSKO DELO UNIVERZITETNEGA ŠTUDIJA

Tematika naloge:

V delu preglejte področje evolucijskega računanja in natančneje opišite splošne genetske algoritme. Osredotočite se na problem trgovskega potnika in predstavite nekatere genetske algoritme za njegovo reševanje. Predstavljene algoritme realizirajte v obliki zaporednega programa (primerno za izvajanje na CPE) in vzporednega programa (primerno za izvajanje na GPE). Implmentirane programe izvedite na testnih primerih in primerjajte učinkovitosti delovanja.

Mentor:
doc. dr. Tomaž Dobravec

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:
prof. dr. Nikolaj Zimic

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani Blaž Poje,

z vpisno številko 63050086,

sem avtor diplomskega dela z naslovom:

Zaporedni in vzporedni genetski algoritmi za reševanje problema trgovskega potnika

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom
  doc. dr. Tomaža Dobravca
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 15.12.2013 Podpis avtorja:

Zahvala

Zahvaljujem se mentorju doc. dr. Tomažu Dobravcu, za izkazano strokovno pomoč, nasvete in usmerjanje pri izdelavi diplomskega dela.

Kazalo

Seznam uporabljenih kratic in simbolov

CPE - Centralna procesna enota
CUDA - Splošnonamenska visoko vzporedna računska arhitektura (angl. *Compute Unified Device Architecture*)
DM - Premestitvena mutacija (angl. *Displacement mutation*)
EA - Evolucijski algoritem
EM - Zamenjava (angl. *Exchange Mutation*)
ER - Evolucijsko računanje
GA - Genetski algoritem
GPE - Grafična procesna enota (angl. *graphics processing unit*)
IMV - Inverzija (angl. *Inversion Mutation*)
ISM - Vrivanje (angl. *Insertion Mutation*)
NBK - Izrek "ni brezplačnega kosila"(angl. *No Free Lunch theorem*)
OX - Križanje položajev (angl. *Order Crossover*)
PMX - Križanje z delno preslikavo (angl. *Partially Matched Crossover*)
PTP - Problem trgovskega potnika (angl. *travelling salesman problem*)
SIM - Preprosta inverzija (angl. *Simple Inversion Mutation*)
SM - Mutacija premešanja (angl. *Scramble mutation*)
XML - Razširljiv označevalni jezik

Povzetek

Osnovni namen diplomske naloge je primerjava vzporedne in zaporedne implementacije algoritma za reševanje problema trgovskega potnika z uporabo genetskih algoritmov. Vzporedni algoritem se izvaja na grafični procesni enoti s pomočjo platforme CUDA, zaporedni pa uporablja centralno procesno enoto. Med raziskovanjem smo slednji algoritem pretvorili tudi v vzporedno obliko s pomočjo procesorskih niti. Ukvarjamo se tudi s kvaliteto razvitih rešitev v odvisnosti od različnih naborov križanj in mutacij ter ob prisotnosti migracij.

V nasprotju s pričakovanji vzporedni algoritem na grafični procesni enoti ne doseže pohitritev. Analiza je razkrila, da so splošno uveljavljena križanja, ki smo jih implementirali, neprimerna za implementacijo na masovno vzporednih arhitekturah. Prizadevanja, da bi jih pretvorili v vzporedne, niso obrodila sadov.

Uspešno smo pokazali, da migracije pozitivno vplivajo na povprečno uspešnost rešitev. Vpliv raste z manjšanjem števila rešitev, ki se prenašajo med ločenimi populacijami.

Naš pregled vpliva različnih operatorjev na porazdelitev doseženih uspešnosti bi lahko nudil smernice za izbiro križanj, ki bi se jih v prihodnosti splačalo pretvoriti v obliko, primerno za vzporedno izvajanje.

Ključne besede:

problem trgovskega potnika, evolucijski algoritem, genetski algoritem, migracije, CUDA, CPE, GPE, pohitritev, pthreads, NVIDIA Visual Profiler

Abstract

The main aim of this thesis is the comparison of parallel and sequential algorithm implementation for solving Travelling Salesman Problem using Genetic algorithms. The parallel algorithm is being executed on a graphical processing unit and sequential algorithm on central processing unit. The latter algorithm has been converted into a parallel form by using processor threads. It also focuses on the quality of the developed solutions in co-dependence with different sets of crossovers and mutations with the presence of migrations.

Contrary to our expectations, the parallel algorithm does not reach speedups on the graphical processing unit. The analysis demonstrates the commonly used crossovers to be unfit for implementation on massively parallel architectures. We have not been successful in transforming them to parallel form.

We have successfully proven the positive effects of migrations on general solution fitness. The smaller the number of solutions being transferred between separate populations, the bigger the effect.

Distribution of average solution fitness could offer further guidelines for selection of crossovers that could be beneficial if transformed into parallel form.

Key words:

Travelling Salesman Problem, Evolutionary algorithm, Genetic algorithm, migrations, CUDA, CPU, GPU, speedup, pthreads, NVIDIA Visual Profiler

Poglavje 1

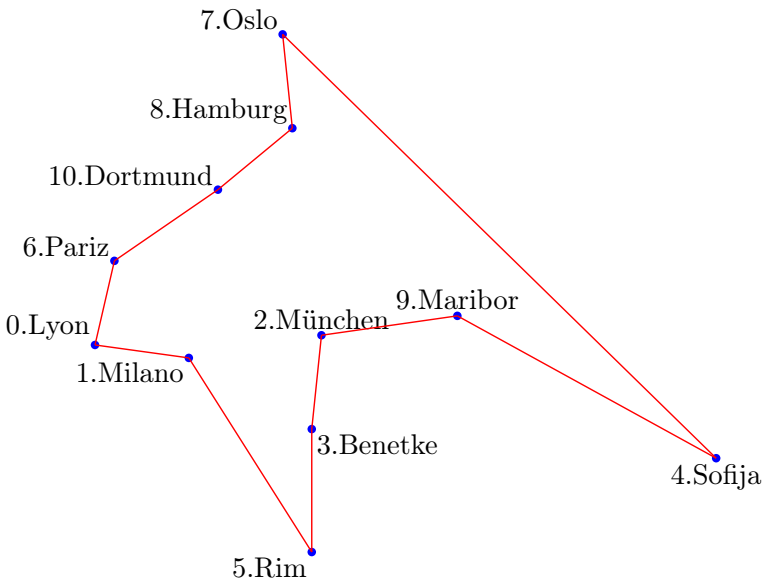
Uvod

1.1 Opis problema

Robič [8] in Mernik [6] opisujeta problem trgovskega potnika (PTP) (angl. *travelling salesman problem*).

Danih je nekaj mest, ki so med seboj vsa neposredno povezana z različno dolgimi cestami. Trgovski potnik želi narediti najkrajši obhod, v katerem bo obiskal vsako mesto natanko enkrat in se vrnil v začetno mesto. Nalogo problema sestavljata končna množica $\{c_1, c_2, \ldots, c_n\}$ mest in matrika $D = (d_{i,j})_{n \times n}$, kjer $d_{i,j} \in \mathbb{R}^+$ predstavlja razdaljo od mesta c_i do mesta c_j. Dopustna rešitev naloge je vsaka permutacija $p = (c_{i_1}, c_{i_2}, \ldots, c_{i_n})$ mest. Kakovost dopustne rešitve p je $m(p) = \sum_{k=1}^{n-1} d_{i_k, i_{k+1}} + d_{i_n, i_1}$. Cilj je poiskati optimalno rešitev, tj. dopustno rešitev (permutacijo) p^*, pri kateri je $m(p)$ minimalna.

PTP je NP-težek optimizacijski problem, saj je njegova odločitvena oblika NP-poln problem. V primeru, ko je N veliko število, je pomembno, da poznamo algoritme, ki najdejo dobro, vendar ne optimalno rešitev v sprejemljivem času. Za PTP obstaja veliko hevurističnih algoritmov, kot sta 2-opt in 3-opt, in drugih aproksimacijskih pristopov, kot so: simulirano ohlajanje, nevronske mreže in evolucijski algoritmi. Algoritem danes srečujemo vse pogosteje v najrazličnejših aplikacijah, npr. risanju grafov in načrtovanju, povečuje pa se tudi število mest N. Če na kratko podamo nekaj primerov, ki so jih reševali na različnih področjih: pri optimizaciji integriranih vezij so reševali problem s 17.000 mesti, pri kristalografiji s 14.000 mesti, pri VLSI-tehniki pa srečamo celo 1.200.000 mest. PTP prikažimo na manjšem primeru enajstih evropskih mest (slika 1.1). Vsako mesto opisuje število med 0 in 10. Položaj na sliki je določen s pomočjo zemljepisne širine in dolžine mest.



Slika 1.1: Optimalna pot med enajstimi evropskimi mesti.

1.2 Namen diplomskega dela

Osnovni namen diplomskega dela je primerjava vzporedne in zaporedne implementacije algoritma za reševanje PTP z uporabo genetskih algoritmov. Vzporedni algoritem se izvaja na grafični procesni enoti (GPE) (angl. *graphics processing unit*) in zaporedni na centralni procesni enoti (CPE). Opcijski del predstavlja raziskovanje vpliva različnih parametrov in strukture algoritma na izvajanje genetskih algoritmov (GA), določitev operatorjev križanja in mutacije, ki v povprečju vodijo do najugodnejših rešitev in paralelizacija zaporednega CPE algoritma.

1.3 Zgradba diplomskega dela

Diplomsko delo je deljeno v štiri večje tematske sklope. Prvi obsega drugo in tretje poglavje ter opisuje genetske algoritme v okviru evolucijskega računanja. Četrto poglavje predstavlja drugi tematski sklop, ki opiše vzporedno arhitekturo CUDA in njene značilnosti. Tretji sklop vključuje peto in šesto poglavje, kjer se poglobimo v realizacijo algoritmov. Četrti podaja opis izvajalnega okolja ter obrazložitev meritev in rezultatov.

Poglavje 2

Evolucijsko računanje

Spodnje besedilo je povzeto po [5, 6]. Evolucijsko računanje (ER) je danes pomembna raziskovalna disciplina v okviru področja računalniških znanosti. Predstavlja poseben način računanja, ki črpa ideje iz procesa naravne evolucije v kombinaciji s posebnim načinom reševanja problemov na osnovi napak (angl. *trial-and-error*). V najosnovnejši obliki si pod naravno evolucijo predstavljamo naslednje: v danem okolju je populacija posameznikov (angl. *individuals*), ki se borijo za obstanek in se ohranjajo z reprodukcijo. Okolje določa stopnjo prilagajanja posameznikov pri doseganju njihovega osnovnega cilja. Od tega je odvisna njihova sposobnost preživetja. Tabela 2.1 prikazuje analogije, ki veljajo v kontekstu reševanja problemov.

EVOLUCIJA	REŠEVANJE PROBLEMOV
Okolje	Problem
Posameznik	Možna rešitev (kandidat)
Prilagajanje	Kvaliteta

Tabela 2.1: Analogije v kontekstu reševanja problemov.

Seveda je bila Darwinova teorija tista, ki je ponudila razlago za razvoj življenja na zemlji, kjer igra naravni izbor osrednjo vlogo. V okolju, kjer je možna omejena eksistenca števila posameznikov z osnovnim instinktom reprodukcije, je selekcija neizbežna, da se populacija ne povečuje eksponentno. Selekcija, ki jo povzroča tekmovanje v prilagajanju okolju (naravi) med posamezniki, je naravna in predstavlja boj za preživetje.

Posamezniki v populaciji so torej elementi selekcije, katerih reprodukcija (in s tem njihovo obnašanje v okolju) je odvisna od njihove prilagoditve okolju.

Ker se bolj prilagodljivi posamezniki reproducirajo, njihovi mutirani potomci predstavljajo dodatno možnost za še boljše prilagajanje okolju.

Pri izvajanju procesa naravne evolucije igra pomembno vlogo molekularna genetika. Posamezniku pripisuje dualne entitete: zunanje fenotipske lastnosti in notranje genotipske strukturne značilnosti. Povedano drugače, genotip posameznika predstavlja kodni zapis za njegov fenotip. Pri tem so geni funkcijske enote genotipov. Kombinacija lastnosti dveh posameznikov pri potomcih se imenuje križanje (angl. *crossover*), naključna sprememba posameznika pa njegova mutacija.

Čeprav se zdi, da križanje ni bistveno pri naravni evoluciji, pa je bilo dokazano, da bistveno vpliva na konvergenco želenih lastnosti obnašanja, s tem pa seveda igra pomembno vlogo v hitro spreminjajočih se okoljih, kakršnega npr. predstavljajo tudi pogoji življenja na zemlji.

Poleg motivacije za uporabo evolucijskega računanja, omenjenega na začetku, obstajajo še drugi razlogi za njeno afirmacijo. Enega predstavlja poenoten način reševanja številnih različnih problemov, drugi pa se skriva v človeški radovednosti, saj je s pomočjo evolucijskega računanja mogoče izvajati eksperimente, ki nimajo podlage v tradicionalni biologiji. Tipičen primer predstavlja t.i. Lamarckov model evolucije, ki predpostavlja, da se pridobljene izkušnje prenašajo na potomce. Čeprav se ne ujema z lastnostmi naravne evolucije, pa ga je mogoče uporabiti pri reševanju različnih tehničnih problemov.

Danes velja prepričanje, da je prednost evolucijskega računanja pri reševanju problemov z najrazličnejših področij v tem, da je v mnogih ozirih univerzalno in torej neodvisno od narave problema. Vzemimo splošen model sistema z vhodnimi spremenljivkami, modelskimi enačbami in izhodnimi spremenljivkami, tako da je pri znanih modelskih enačbah sistema in znanih vrednostih vhodnih spremenljivk možno določiti vrednosti izhodnih spremenljivk. Tako lahko evidentiramo tri osnovne tipe problemov, ki so primerni za evolucijsko reševanje, glede na to, česa v sistemu ne poznamo:

- optimizacijski problem imamo takrat, kadar poznamo modelske enačbe in želene izhodne vrednosti, iščemo pa optimalne vhodne vrednosti,

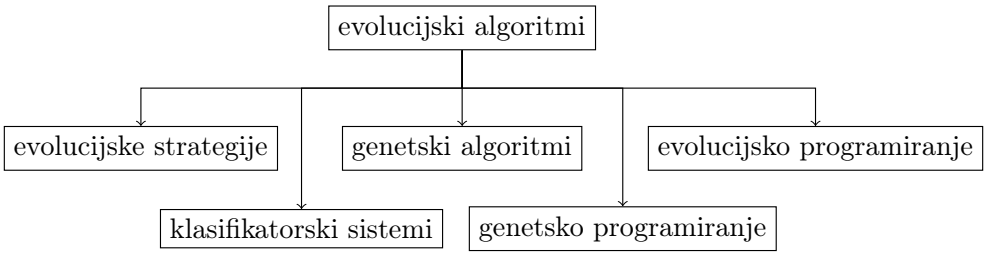
- identifikacijski problem išče modelske enačbe pri znanih parih vhodnih in izhodnih spremenljivk,

- simulacijski problem pa išče vrednosti izhodnjih spremenljivk, če poznamo vrednosti vhodnih spremenljivk in modelske enačbe.

2.1 Evolucijski algoritem

Evolucijsko računanje sledi evolucijskemu algoritmu (EA) [5]. Evolucijski algoritem [6] je skupni izraz za postopke reševanja problemov s pomočjo računalnikov, ki uporabljajo model in mehanizme biološke evolucije. Za vse te algoritme je značilno, da simulirajo evolucijo in njene mehanizme, kot so selekcija, križanje in mutacija.

K evolucijskim algoritmom (angl. *evolutionary algorithm*) (slika 2.1) prištevamo evolucijske strategije (angl. *evolutionary strategies*), genetske algoritme (angl. *genetic algorithms*) in evolucijsko programiranje (angl. *evolutionary programming*). Iz genetskih algoritmov se kasneje razvijejo še klasifikatorski sistemi (angl. *classifier systems*) in genetsko programiranje (angl. *genetic programming*). Skupno ime za te algoritme je tudi evolucijsko računanje (angl. *evolutionary computation*).



Slika 2.1: Delitev evolucijskih algoritmov.

Evolucija je spreminjanje živih bitij v geoloških obdobjih. Značilen je razvoj od enostavnih k bolj zapletenim, višjim oblikam. Evolucijski algoritmi uporabljajo populacijo struktur, ki se razvijajo s pomočjo selekcije in operatorjev iskanja (križanje in mutacija). Za lažje razumevanje evolucijskih algoritmov je potrebno razumeti biološke procese, na katerih temeljijo. V naravi lahko opazimo, da se velika množica organizmov razmožuje nespolno, npr. bakterije. V tem primeru so potomci identični staršem. Spolno razmnoževanje omogoča, da se genetska informacija premeša. Potomci vsebujejo neko poljubno genetsko kombinacijo, ki so jo podedovali od obeh staršev. Ta proces imenujemo križanje (angl. *crossover*) in je tipičen za okolje, kjer je izbira staršev v veliki meri funkcija uspešnosti (angl. *fitness function*) posameznika pred drugimi. Pogoj za uspeh evolucije je raznolikost. V naravi je pomemben vir raznolikosti mutacija (angl. *mutation*). Seveda je prisotna tudi naključnost. Evolucijski algoritem pa ni algoritem z naključnim iskanjem, temveč stohastičen proces.

Na tem mestu opozorimo na izrek "ni brezplačnega kosila"(NBK) (angl. *No Free Lunch theorem*). Izrek NBK dokaže, da ne obstaja algoritem za reševanje

vseh možnih problemov, ki bi bil boljši od naključnega iskanja. Tega izreka si ne smemo razlagati napačno in se zadovoljiti samo z naključnim iskanjem. Zmotno je meniti, da je iskanje boljšega algoritma brezpredmetno. Nikoli namreč ne iščemo algoritma, ki bi bil primeren za vse vrste problemov. Posledica izreka NBK je naslednja. V primeru uspešnejšega delovanja algoritma na določeni družini problemov (specializirani algoritem) se bo le-ta slabše odrezal na drugi družini problemov.

2.2 Razlike med evolucijskimi algoritmi

Med najpogosteje omenjene evolucijske algoritme sodijo genetski algoritmi, evolucijske strategije, evolucijsko programiranje in genetsko programiranje [6]. Vsem tem je skupen algoritem 2.1.

Algoritem 2.1 Splošni evolucijski algoritem

```
1: procedure SPLOŠNI_EVOLUCIJSKI_ALGORITEM()
2:     t ← 0;
3:     P(t) ← inicializiraj(μ);
4:     F(t) ← ovrednoti(P(t));
5:     while not ι(P(t), Θ_j) do
6:         P'(t) ← križanje(P(t), Θ_r, κ);
7:         P''(t) ← mutacija(P'(t), Θ_m, λ);
8:         F(t) ← ovrednoti(P''(t));
9:         P(t+1) ← selekcija(P''(t), F(t), Θ_s, μ);
10:        t ← t + 1
11:    end while
12: end procedure
```

Naj bo I poljuben prostor in a rešitev v tem prostoru ($a \in I$). S funkcijo $F : I \to \mathbb{R}$ ocenimo uspešnost rešitve. Parametri μ, κ in λ določajo velikost starševske in vmesnih populacij. Populacijo v času t zapišemo kot $P(t) = (a_1(t), \cdots, a_\mu(t)) \in I^\mu$. Križanje, mutacijo in selekcijo predstavimo kot operatorje:

- $r : I^\mu \to I^\kappa$ - križanje,
- $m : I^\kappa \to I^\lambda$ - mutacija,
- $s : I^\lambda \to I^\mu$ - selekcija,

ki spremenijo trenutno populacijo. Bodimo pozorni, da vsi našteti operatorji (r, m, s) delujejo nad celotno populacijo. Tako je algoritem dovolj splošen, da lahko opišemo različne podzvrsti evolucijskih algoritmov. Operator mutacije bi lahko definirali tudi na nivoju enega posameznika in pri tem večkrat uporabili operator $m' : I \to I$. Delovanje operatorjev je odvisno od krmilnih parametrov Θ_r, Θ_m, Θ_s (algoritem 2.1). Evolucijski algoritem se prične z inicializacijo začetne populacije. Običajno je le-ta generirana naključno. V kolikor o problemu vemo več, lahko to znanje vključimo v evolucijski algoritem. Sledi ovrednotenje populacije in zanka, v kateri izvajamo križanje, mutacijo in selekcijo. Kriterij zaustavitve $\iota\left(P(t), \Theta_j\right)$ je odvisen od trenutne populacije in od drugih parametrov, ki jih označimo s Θ_j. Tako dobimo splošni evolucijski algoritem (algoritem 2.1). Bodimo pozorni, da se lahko velikost populacije v zanki spreminja (μ, λ, κ). V primeru genetskih algoritmov velja $\mu = \lambda = \kappa$. Prav tako lahko v posameznih podzvrsteh izpustimo korak križanja ($P'(t) = P(t)$) ali mutacije ($P''(t) = P'(t)$).

	Genetski algoritem	Evolucijske strategije	Evolucijsko programiranje	Genetsko programiranje
Populacija	bitni nizi	realna števila	končni avtomati	programi
Iskalni operatorji	križanje, mutacija	mutacija (kasneje tudi križanje)	mutacija	križanje
Selekcija	stohastična	deterministična	stohastična	stohastična
Vrstni red operatorjev	selekcija križanje mutacija	križanje mutacija selekcija	mutacija selekcija	selekcija križanje
Krmilni parametri so del predstavitve	NE	DA	NE	NE

Tabela 2.2: Razlike med evolucijskimi algoritmi.

Čeprav je osnovni evolucijski algoritem za vse podzvrsti enak, se le-te razlikujejo v malenkostih, ki jih podaja tabela 2.2. Ena izmed razlik med posameznimi podzvrstmi je predstavitev populacije oz. kaj tvori populacijo. Pri genetskih algoritmih populacijo tvorijo bitni nizi, pri evolucijskih strategijah realna števila, pri evolucijskem programiranju so to končni avtomati in pri genetskem programiranju računalniški programi.

Poglavje 3

Genetski algoritmi

Poglavje povzema vire [1, 5, 6]. Za začetnika GA velja John Holland, ki je leta 1975 napisal knjigo "Adaptation in Natural and Artificial Systems" [6]. Za uspešno delovanje tako evolucijskih kot genetskih algoritmov je potrebno:

- rešitve primerno predstaviti (predstavitveni problem),
- določiti funkcijo uspešnosti, ki oceni uspešnost oz. kvaliteto rešitve,
- določiti genetske operatorje (npr. križanje in mutacija),
- določiti krmilne parametre evolucijskega (genetskega) algoritma.

Namesto izraza rešitev se pogosto uporablja izraz genom ali osebek. Rešitev je sestavljena iz posameznih delov, ki jih imenujemo tudi geni.

V primeru genetskih algoritmov možno rešitev kodiramo v obliki bitnega niza [6]. Takšen niz vsebuje samo enice in ničle, ki jim pravimo biti, ter omogoča enostavno izbedbo operacije križanja in mutacije. Različni primeri kodiranja so predstavljeni v podpoglavju 3.1. Ena izmed težjih nalog genetskih algoritmov je torej, kako predstaviti rešitev v obliki bitnega niza. To imenujemo tudi predstavitveni problem. Velikokrat je bolje, da rešitev podamo na naravnejši način in z bogatejšo podatkovno strukturo (npr. seznam, matrika, graf). Venda moramo tedaj primerno določiti tudi genetska operatorja križanja in mutacije.

Naslednja naloga, ki se lahko prav tako izkaže za težavno, je določitev funkcije uspešnosti. Poznamo probleme, kjer je predstavitev rešitve sorazmerno enostavna, težje pa je določiti funkcijo uspešnosti. Takšen je npr. problem iskanja tavtologije konjuktivne normalne oblike (podpoglavje 3.1). V kolikor

se izkaže, da ne znamo razlikovati med dobrimi in slabimi rešitvami, evolucijskega algoritma za dani problem ne moremo uporabiti. Funkcija uspešnosti določa uspešnost preživetja rešitve. Boljše rešitve bodo imele večjo verjetnost preživetja kot slabše.

Genetski algoritem (algoritem 3.1) se od splošnega evolucijskega algoritma (algoritem 2.1) razlikuje po vrstnem redu izvajanja operatorjev (tabela 2.2).

Algoritem 3.1 Genetski algoritem

```
1: procedure GENETSKI_ALGORITEM()
2:     t ← 0;
3:     P(t) ← inicializiraj(velikost_populacije);
4:     F(t) ← ovrednoti(P(t));
5:     while not ι(P(t), Θ_j) do
6:         P'(t) ← selekcija(P(t), F(t), Θ_s, velikost_populacije);
7:         P''(t) ← križanje(P'(t), Θ_r, velikost_populacije);
8:         P(t+1) ← mutacija(P''(t), Θ_m, velikost_populacije);
9:         F(t+1) ← ovrednoti(P(t+1));
10:        t ← t+1
11:    end while
12: end procedure
```

V začetni populaciji naključno generiramo bitne nize. Velikost populacije je določena s krmilnim parametrom *velikost_populacije* in se običajno skozi generacije ne spreminja. V kolikor problem bolje poznamo, lahko to znanje uporabimo pri gradnji začetne populacije. Tako bitni nizi ne bodo generirani povsem naključno. Ko je zgrajena začetna populacija, je naslednji korak v evolucijskem ciklu določitev funkcije uspešnosti (podpoglavje 3.6). Slednja je osnova za selekcijo, ki izbere uspešnejše nize v naslednjo generacijo. Različne oblike selekcije opisuje podpoglavje 3.3. Naslednji korak v genetskem algoritmu tvorita operaciji križanja (podpoglavje 3.4) in mutacije (podpoglavje 3.5). Ti dve spremenita populacijo, ki postane nova začetna populacija naslednjega evolucijskega cikla.

3.1 Predstavitveni problem

Problemi, ki jih rešujemo, so različni in vsakega lahko predstavimo na mnogo načinov. Iskalni algoritmi so odvisni od načina predstavitve, zato je še kako pomembno, izbrati pravilno predstavitev. Osnovni genetski algoritem je zelo

preprost, vendar zahteva predstavitev v obliki bitnega niza. Običajno je največji problem prav ta predstavitev, zato govorimo o predstavitvenem problemu. Predstavitev z bitnimi nizi je primerna za probleme, kjer ima rešitev binarno predstavitev. Taki problemi so npr. problem 0/1 nahrbtnika, psevdo-Boolovi optimizacijski problemi $f : \{0,1\}^m \rightarrow \mathbb{R}$, kjer je m velikost problemskega prostora, iskanje minimalno povezanega grafa itd. V vseh teh primerih rešitev predstavimo kot bitni niz, kjer z 1 označimo, da je element, vozlišče ali povezava prisotna, z 0 pa, da element ni prisoten v rešitvi. Vendar takšno predstavitev uporabljamo tudi za poljubne probleme oblike $f : \mathbb{S} \rightarrow \mathbb{R}$, kjer je iskalni prostor $\mathbb{S}$ bistveno različen od $\{0,1\}^m$ (npr. $\mathbb{R}^n$). Zakaj uporabljamo tako predstavitev tudi za te primere? Mnogi so prepričani, da je predstavitev z bitnimi nizi najprimernejša, saj v takšni predstavitvi najlažje maksimiziramo število različnih shem, s katerimi preiskujemo prostor. Vendar obstaja veliko primerov, kjer so se alternativne predstavitve izkazale za uspešnejše. Dandanes velja naslednje priporočilo: izberimo takšno predstavitev in kodiranje, ki sta za dani problem najnaravnejši. Pri preslikavi iz prostora rešitev v prostor predstavitev si želimo tako preslikavo, kjer se razdalja med elementoma v obeh prostorih ohranja. Empirične raziskave so potrdile, da je uporaba Grayjeve kode primernejša kakor standardno binarno kodiranje. Razlog je ta, da prvo kodiranje ohranja razdaljo v predstavitvenem prostoru. Na primer, imamo 5-bitni parameter, ki lahko zavzame vrednosti med 0 in 31. Število 15 binarno predstavimo z 01111 in število 16 z 10000. Opazimo, da majhen korak v prostoru rešitev zahteva veliko spremembo v predstavitvenem prostoru (Hammingova razdalja je 5 in potrebno je spremeniti vse bite). Medtem ko pri uporabi Grayjeve kode število 15 predstavimo z 01000 in število 16 z 11000 (Hammingova razdalja je 1 in potrebno je spremeniti le en bit).

Potem ko smo problem uspešno preslikali v bitno predstavitev, operaciji križanja in mutacije ne predstavljata večjih implementacijskih problemov.

Za določene probleme, kot je npr. iskanje tavtologije normalne konjuktivne oblike, izbira funkcije uspešnosti ni enostavno opravilo. V normalni konjuktivni obliki je izraz predstavljen kot konjunkcija logičnih podizrazov, ki so sestavljeni iz logičnih spremenljivk ali njihovih negacij, povezanih z Boolovim operatorjem ($\vee$). Primer konjuktivne normalne oblike za logične spremenljivke a, b, c, d in e podaja izraz 3.1.

$$f\,(a,b,c,d,e) = \left(a \vee \bar{b} \vee c\right) \wedge \left(a \vee \bar{c} \vee \bar{d}\right) \wedge \left(\bar{c} \vee \bar{d} \vee \bar{e}\right) \tag{3.1}$$

Pri iskanju tavtologije normalne konjuktivne oblike je potrebno poiskati vrednosti logičnih spremenljivk tako, da bo vrednost izraza *true*. V našem primeru je ena izmed možnosti, da bo izraz vračal logično vrednost *true* te-

daj, ko ima logična spremenljivka a vrednost $true$ in logična spremenljivka c vrednost $false$. Ta problem ima zelo enostavno predstavitev z nizi. Logične spremenljivke so med seboj neodvisne, zato lahko vsako logično spremenljivko predstavimo z enim bitom. V našem primeru, ko imamo pet logičnih spremenljivk, je dolžina niza pet. Tako v primeru niza 10101 izraz 3.2 vrača logično vrednost $true$.

$$f\,(1,0,1,0,1) = (1 \vee \bar{0} \vee 1) \wedge (1 \vee \bar{1} \vee \bar{0}) \wedge (\bar{1} \vee \bar{0} \vee \bar{1}) \tag{3.2}$$

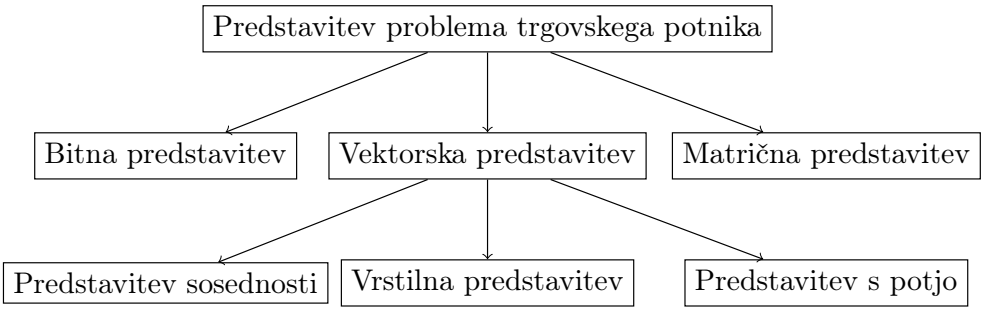
Po drugi strani izbira funkcije uspešnosti ni tako enostavno opravilo. Opazimo, da normalna konjuktivna oblika vrne logično vrednost $true$, ko najdemo rešitev, v vseh ostalih primerih pa logično vrednost $false$. Slednje ne omogoča razlikovanja med boljšimi in slabšimi rešitvami. Ena izmed možnih izbir funkcije uspešnosti je vračanje števila podizrazov z logično vrednostjo $true$. Ko vsi podizrazi vrnejo logično vrednost $true$, smo našli rešitev. V primeru iskanja tavtologije konjuktivne normalne oblike opazimo, da je predstavitev zelo naravna in enostavna, zahtevnejša pa je izbira funkcije uspešnosti.

Poznamo pa tudi probleme, kjer je izbira funkcije uspešnosti zelo naravna in enostavna, predstavitev z bitnimi nizi pa prinaša težave. Sem sodi tudi problem trgovskega potnika. Ta je zanimiv s stališča različnih predstavitev problema. Možne rešitve ne predstavimo v obliki bitnega niza, temveč z bogatejšo podatkovno strukturo, npr. z vektorjem ali matriko. Ker smo spremenili podatkovno strukturo, moramo ustrezno spremeniti tudi operatorja križanja in mutacije, kar pa ni vedno enostavno opravilo.

3.2 Predstavitev pri problemu trgovskega potnika

Običajni genetski algoritem predpostavlja, da je problem predstavljen v obliki bitnega niza, kar omogoča uporabo nespremenjenih operatorjev križanja in mutacije [6]. Posamezno mesto predstavimo v obliki bitnega niza dolžine $\lceil log_2(N) \rceil$ bitov, kjer je N število mest, cela pot pa je opisana z nizom dolžine $N \cdot \lceil log_2(N) \rceil$ bitov. Vendar v tem primeru problema ne moremo predstaviti v obliki bitnega niza, ne da bi dodatno uporabili še algoritem popravljanja. Sledje razložimo na naslednjem primeru, kjer je število $N = 20$. Za bitni opis enega mesta potrebujemo pet bitov, saj s štirimi biti lahko opišemo samo $2^4 = 16$ mest. Po drugi strani pa s petimi biti opišemo 32 različnih mest. Tako lahko s križanjem in mutacijo dobimo mesto, ki ga med 20 mesti sploh ni. Mesto se mora pojaviti v seznamu obiskov samo enkrat, kar je treba vedno

preverjati in po potrebi popravljati. Zato zasledimo primernejše predstavitve poti z vektorji in matrikami (slika 3.1). Predstavitev z vektorjem nadalje delimo še na: predstavitev sosednosti, vrstilno predstavitev in predstavitev s potjo.



Slika 3.1: Različne predstavitve problema trgovskega potnika.

Na tem mestu si natančneje oglejmo različne predstavitve in na njih vezane specifične operatorje križanja.

3.2.1 Vektorska predstavitev poti

Preprosta predstavitev poti [6] za problem trgovskega potnika z N mesti je predstavitev z vektorjem oz. s seznamom $(i_0, i_1, \ldots, i_{N-1})$. Delimo jo na:

- predstavitev sosednosti (angl. *adjacency representation*),
- vrstilno predstavitev (angl. *ordinal representation*),
- predstavitev s potjo (angl. *path representation*).

Vsaka od naštetih predstavitev zahteva specifične genetske operatorje. Pri teh predstavitvah je operator mutacije relativno lahka, medtem ko je operator križanja zahtevnejša operacija.

Predstavitev sosednosti

V predstavitvi sosednosti je mesto j na i-tem položaju, samo če vodi pot neposredno iz mesta i v mesto j.

Tako seznam (2 5 4 8 6 9 1 3 0 10 7) predstavlja pot 0-2-4-6-1-5-9-10-7-3-8 (tabela 3.1). Vsaka pot ima natanko eno predstavitev sosednosti, vsaka predstavitev sosednosti pa ne predstavlja pravilne poti. Primer takega seznama

je (8 0 6 10 7 2 5 4 9 1 3), ki pripelje do prehitre vrnitve v začetno mesto (cikel), saj opravimo samo delno pot 0-8-9-1-0. V takih primerih je potreben ustrezen algoritem popravljanja, ki popravi pot tako, da ne vsebuje ciklov.

Položaj	0	1	2	3	4	5	6	7	8	9	10
Seznam	2	5	4	8	6	9	1	3	0	10	7
Pot	0	2	4	6	1	5	9	10	7	3	8

Tabela 3.1: Primer poti s predstavitvijo sosednosti.

Predstavitev sosednosti ne podpira klasičnega operatorja križanja, zato se uporabljajo naslednja tri križanja:

- križanje z izmenjavo povezav (angl. *alternating edges*),
- križanje z deli poti (angl. *subtour chunks*),
- križanje s hevristično funkcijo.

Vrstilna predstavitev

Uporablja seznam, v katerem velja, da je i-ti element seznama število med 0 in $N - i$. Tako lahko za prvi element seznama izbiramo med vsemi mesti. Nato se izbira manjša iz koraka v korak. Mesta so podana v referenčnem seznamu. Vzemimo, da imamo referenčni seznam, ki ga prikazuje izraz 3.3. Pot, ki jo želimo opisati, npr. 0-2-4-6-1-5-9-10-7-3-8, opišemo z vrstilno predstavitvijo s seznamom (0 1 2 3 0 1 3 3 1 0 0). Tvorimo ga na način, i ga prikazuje tabela 3.2, kjer i pomeni položaj v seznamu. Glavna prednost vrstilne predstavitve je preprosta izvedba križanja. Vsaka pot, nastala s križanjem, predstavlja dovoljeno pot.

$$C = (0\,1\,2\,3\,4\,5\,6\,7\,8\,9\,10) \tag{3.3}$$

Predstavitev s potjo

Je najnaravnejša in najučinkovitejša predstavitev pri PTP. Tako pot 0-2-4-6-1-5-9-10-7-3-8 preprosto opišemo s seznamom (0 2 4 6 1 5 9 10 7 3 8). Za to predstavitev je bilo predlagano zelo veliko število različnih operatorjev [6], med katerimi omenimo naslednje:

Korak	Referenčni seznam	Element i	Dobljena pot
1	(0 1 2 3 4 5 6 7 8 9 10)	0	(0)
2	(1 2 3 4 5 6 7 8 9 10)	1	(0-2)
3	(1 3 4 5 6 7 8 9 10)	2	(0-2-4)
4	(1 3 5 6 7 8 9 10)	3	(0-2-4-6)
5	(1 3 5 7 8 9 10)	0	(0-2-4-6-1)
6	(3 5 7 8 9 10)	1	(0-2-4-6-1-5)
7	(3 7 8 9 10)	3	(0-2-4-6-1-5-9)
8	(3 7 8 10)	3	(0-2-4-6-1-5-9-10)
9	(3 7 8)	1	(0-2-4-6-1-5-9-10-7)
10	(3 8)	0	(0-2-4-6-1-5-9-10-7-3)
11	(8)	0	(0-2-4-6-1-5-9-10-7-3-8)

Tabela 3.2: Gradnja seznama pri vrstilni predstavitvi.

- križanje z delno preslikavo PMX (angl. *partially-mapped crossover*),
- križanje vrstnega reda OX (angl. *order crossover*),
- križanje vrstnega reda OX2 (angl. *order based crossover*),
- križanje ciklov CX (angl. *cycle crossover*),
- križanje s prerazporeditvijo povezav ERX (angl. *edge recombination crossover*).

Križanje vrstnega reda OX in križanje z delno preslikavo PMX podrobneje opisujeta podpoglavji 3.4.1 in 3.4.2.

3.2.2 Matrična predstavitev poti

Tako kot pri predstavitvi z vektorjem poznamo tudi več različnih predstavitev s pomočjo matrike. Dva različna pristopa predstavljata:

- predstavitev s pomočjo položaja,
- predstavitev s pomočjo sosednjih mest.

V sklopu diplomske naloge se s to predstavitvijo nismo podrobneje ukvarjali. Bralec si o njej lahko prebere v uporabljeni literaturi [6].

3.3 Selekcija

Poznamo različne oblike selekcije:

- proporcionalna selekcija,
- selekcija z rangiranjem,
- turnirska selekcija.

Slabost proporcionalne selekcije je možnost prehitre konvergence k lokalnemu optimumu [6]. To se zgodi v primeru, ko imamo v populaciji niz, ki je daleč najuspešnejši (t.i. super niz). Po drugi strani ta niz ne vodi h globalnemu optimumu. V tem primeru bo kaj hitro prevladal v celotni populaciji in našli bomo samo lokalni optimum. Slabosti se izognemo z uporabo drugih vrst selekcij, npr. selekcija z rangiranjem ali turnirske selekcije, ki sta opisani v naslednjih podpoglavjih.

3.3.1 Proporcionalna selekcija

Pri proporcionalni selekciji je uspešnost preživetja bitnih nizov sorazmerna njihovi uspešnosti. Večja je uspešnost, tem večja je njihova verjetnost preživetja. Za njeno implementacijo običajno izberemo pravilo rulete (angl. *roulette rule*), kjer je velikost polja rulete vsakega niza sorazmerna z njegovo uspešnostjo. Tako ruleto zgradimo na naslednji način [6]:

- izračunamo uspešnost($eval(v_i)$) za vsak niz v_i ($i = 1, \ldots,$ $velikost\_populacije$),
- seštejemo uspešnost vseh nizov po izrazu 3.4,
- za vsak niz izračunamo verjetnost izbire p_i po izrazu 3.5,
- za vsak niz izračunamo kumulativno vrednost q_i po izrazu 3.6.

$$F = \sum_{i=1}^{velikost\_populacije} eval(v_i) \tag{3.4}$$

$$p_i = \frac{eval(v_i)}{F} \tag{3.5}$$

$$q_i = \sum_{j=1}^{i} p_j \tag{3.6}$$

Proces selekcije nato poteka tako, da zavrtimo ruleto natanko *velikost populacije*-krat in vsakič izberemo niz za novo populacijo. Natančneje, *velikost populacije*-krat generiramo naključno število r iz intervala [0..1]. Če je $r < q_1$, izberemo prvi niz (v_1), sicer izberemo i-ti niz (v_i) $(2 \leq i \leq velikost\_populacije)$, za katerega velja $q_{i-1} < r \leq q_i$. Tako bodo uspešnejši nizi izbrani večkrat.

3.3.2 Selekcija z rangiranjem

Verjetnost preživetja v procesu selekcije odloča o uspešnosti posameznika. Pri selekciji z rangiranjem se ta verjetnost določa glede na rang posameznika. Populacijo uredimo po kriteriju uspešnosti. Mesto v tako urejenem seznamu določa rang posameznika. V tem primeru nizov ne izbiramo glede na njihovo absolutno uspešnost, temveč jih glede na njihovo uspešnost rangiramo. Tako bo imel najuspešnejši niz najvišji rang $(rang = 1)$ in najslabši niz najnižji rang $(rang = velikost\_populacije)$. Uspešnost niza nato določimo glede na njegov rang. Obstaja veliko različnih selekcij, ki temeljijo na rangiranju.

Preprosta Bakerjeva selekcija z rangiranjem izračuna verjetnost izbire, tako da je razlika verjetnosti med dvema rangoma konstantna. Uporabnik določi vrednost MAX, ki predstavlja uspešnost najuspešnejšega niza. Razlika med posameznimi uspešnostmi nizov je konstantna in se določi s pomočjo enačbe 3.7. Uspešnost niza v izračunamo po enačbi 3.8. Nadaljnji postopek je enak kot pri proporcionalni selekciji.

$$r = \frac{MAX}{velikost\_populacije} \tag{3.7}$$

$$eval\,(v) = MAX - (rang(v) - 1) \cdot r \tag{3.8}$$

3.3.3 Turnirska selekcija

Ideja učinkovito izrablja princip selekcije z rangiranjem. Metoda naključno izbere k elementov $(k \leq velikost\_populacije)$, kjer k imenujemo tudi velikost turnirja. Med izbranimi se izbere najboljši in se uvrsti v naslednjo generacijo.

Postopek se ponovi $velikost\_populacije$-krat. Logično lahko sklepamo, da večji kot je k, večji je selekcijski pritisk. Najpogostejša izbira velikosti turnirja je dva ($k = 2$).

Pri žrebu kandidatov za turnir si pomagamo z naključno izbranim številom $r \in [0, 1)$ in s pomočjo enačbe 3.9 izberemo niz v_i. V naslednjo generacijo se prenesejo zmagovalci turnirja. Turnirska selekcija je preprosta za implementacijo, saj ni potrebno urejati populacije.

$$i = trunc\,(r \cdot velikost\_populacije + 1) \tag{3.9}$$

3.4 Križanja

Breskvar in Rodič [1] opisujeta dve različni metodi križanja pri uporabi permutacijskega zapisa genomov.

3.4.1 Križanje položajev - Order Crossover (OX)

Nekateri avtorji uporabljajo tudi ime križanje vrstnega reda. V potomcu so združeni elementi med dvema izbranima točkama vključno iz izbranega genoma v enakem vrstnem redu, kot se pojavijo v začetnem izbranem genomu. Preostali elementi so podedovani iz drugega genoma v vrstnem redu, kot se v njem pojavijo, začenši s prvo pozicijo za zadnjim izbranim mestom križanja. Elementi, ki so že uporabljeni v potomcu, se preskočijo. Glavni namen operatorja je ohranjanje relativnega vrstnega reda elementov, soseščine in absolutne pozicije dela enega in drugega genoma. Vendar pri tem prihaja do motenj, ko želimo ohranjati vrstni red iz drugega genoma, pa so bili elementi izbrani že v prvem.

Naključno določimo dve mesti križanja. V prikazanem primeru sta mesti križanja na mestih 2 in 6 (slika 3.2).

Genom 1:	0	1	2	3	4	5	6	7	8	9
Mesto križanja:			|				|			
Genom 2:	2	5	0	9	7	3	8	6	1	4

Slika 3.2: Izbrani mesti križanja - OX.

Geni med točkama križanja se med genomoma zamenjajo. Iz genoma 1 se geni: 2, 3, 4, 5, 6 prenesejo v genom 2 na izbrana mesta. Nasprotno se

prepišejo geni iz genoma 2 v genom 1 (slika 3.3). Gene, ki niso določeni za zamenjavo, označimo z zvezdico.

Genom 1:	*	*	0	9	7	3	8	*	*	*
Mesto križanja:			|				|			
Genom 2:	*	*	2	3	4	5	6	*	*	*

Slika 3.3: Zamenjani geni med genomoma - OX.

Določimo gene iz genomov, ki še niso bili uporabljeni in bodo prišli na mesta, označena z zvezdico (slika 3.4). Pri tem je treba paziti na vrstni red, v katerem so geni zapisani v posameznem genomu. Gene začnemo pregledovati na zadnji točki križanja, to je na mestu 7. V prvem genomu je na mestu 7 gen 7 in ta gen je že uporabljen na mestu križanja. Zato gremo na naslednje mesto 8, kjer se nahaja gen 8 in tudi ta je že uporabljen. Mesto devet je tudi že uporabljeno, zato začnemo pregledovati na začetku genoma 1. Prvo mesto in gen, ki še ni uporabljen, je 1. Tega zapišemo na mesto 7 v prvem genomu. Postopek ponavljamo, dokler niso zapolnjena vsa mesta. Za genom 2 je postopek enak.

Neuporabljeni geni v genomu 1:	1	2	4	5	6
Neuporabljeni geni v genomu 2:	1	0	9	7	8

Slika 3.4: Prosti geni in njihov pravilni vrstni red - OX.

Postopek zapolnjevanja praznih mest s prostimi geni začnemo za zadnjo točko križanja, kot ponazarja slika 3.5.

Genom 1:	*	*	0	9	7	3	8	1	2	4
Mesto križanja:			|				|			
Genom 2:	*	*	2	3	4	5	6	1	0	9

Slika 3.5: Zapolnjevanje mest za zadnjo točko križanja - OX.

Nazadnje zapolnimo mesta na začetku genomov in dobimo končni rezultat, ki je prikazan na sliki 3.6.

Potomec 1:	5	6	0	9	7	3	8	1	2	4
Potomec 2:	7	8	2	3	4	5	6	1	0	9

Slika 3.6: Končni rezultat operatorja OX.

3.4.2 Križanje z delno preslikavo - Partially Matched Crossover (PMX)

PMX je nastal z razmišljanjem o slepem pristopu k problemu trgovskega potnika. Pri delno ujemajočem se križanju dveh izbranih genomov izberemo dve naključni točki med geni. Gene v posmeznem genomu, ki ležijo znotraj izbranih točk, definiramo kot enakovredne dele posameznega genoma (slika 3.7).

Genom 1:	0	1	2	3	4	5	6	7	8	9
Mesto križanja:				|			|		
Genom 2:	2	5	0	9	7	3	8	6	1	4

Slika 3.7: PMX - naključna izbira dveh mest križanja.

PMX zamenja izbrane gene med genomoma (slika 3.8).

Genom 1:	0	1	2	9	7	3	8	7	8	9
Mesto križanja:				|			|		
Genom 2:	2	5	0	3	4	5	6	6	1	4

Slika 3.8: PMX - zamenjava genov med genomoma.

Sedaj se nekateri geni v genomih ponovijo, kar pa pri razporejanju ni dovoljeno. Zato operator PMX pregleda, kako so se zamenjali posamezni geni in ugotovi naslednje prehode oziroma križanja: 3-9, 4-7, 5-3, 6-8 in 9-3, 7-4, 3-5, 8-6. Geni, ki v prehodih ne nastopajo (0, 1 in 2), ostanejo v potomcih na svojih mestih. Pozicije v genomih, ki jih je treba urediti, kjer se ponavljajo, so označena z zvezdicami (slika 3.9). Genoma uredimo s križanji, ki smo jih dobili pri zamenjavi genov. Vidimo, da je treba urediti gen na mestu 8, katerega vrednost je bila 7. Poiščemo križanje z vrednostjo 7 (7-4) in na mesto 8 postavimo vrednost 4. Na mestu 9 je bila vrednost 8 (8-6), kamor postavimo vrednost 6. Na mestu 10 je bila vrednost 9 (9-3) in tu zamenjamo zvezdico z vrednostjo 3. Prvi genom je tako urejen. Drugi genom uredimo na enak način.

Na mestu 2 je bila vrednost 5, poiščemo pripadajoče križanje (5-3) in zamenjamo zvezdico z vrednostjo 3. Na mestu 8 je bila vrednost 6 (6-8), zvezdico zamenjamo z vrednostjo 8. Preostane nam še mesto 10 (4-7), kjer zvezdico zamenjamo z vrednostjo 7.

Genom 1:	0	1	2	9	7	3	8	*	*	*
Mesto križanja:				|			|		
Genom 2:	2	*	0	3	4	5	6	*	1	*

Slika 3.9: PMX - zamenjava genov z zvezdicami.

Slika 3.10 prikazuje naslednjo rešitev. Če se katera od vrednosti, označenih z zvezdico, po zamenjavi še vedno podvaja, iščemo križanja vse dotlej dokler ne najdemo vrednosti, ki se v genomu ne podvaja. V danem primeru se v prvem in drugem genomu podvaja vrednost 3. Zato poiščemo nova križanja za vrednost 3: 3-5 in 9-3. Na mesto 10 zapišemo v prvem genomu 5. Na mesto dve zapišemo v drugem genomu 9.

Potomec 1:	0	1	2	9	7	3	8	4	6	3
Mesto križanja:				|			|		
Potomec 2:	2	3	0	3	4	5	6	8	1	7

Slika 3.10: PMX - delna rešitev.

Slika 3.11 prikazuje končno rešitev, kjer se noben gen ne podvaja.

Potomec 1:	0	1	2	9	7	3	8	4	6	5
Mesto križanja:				|			|		
Potomec 2:	2	9	0	3	4	5	6	8	1	7

Slika 3.11: Končni rezultat operatorja PMX.

3.5 Mutacije

Breskvar in Rodič [1] opisujeta šest različnih metod mutiranja pri uporabi permutacijskega zapisa genomov.

3.5.1 Zamenjava - Exchange Mutation (EM)

Naključno sta izbrana dva gena, ki jih med seboj zamenjamo (slika 3.12). Za operator zamenjave se uporablja še kar nekaj imen - swap mutation operator, point mutation operator, reciprocal exchange operator. Omenjeni operator je med mutacijami najbolj uporabljan.

Genom:	1	2	3	4	5	6	7	8	9	10
Mesto:	*				*					
Potomec:	5	2	3	4	1	6	7	8	9	10

Slika 3.12: Mutacija z zamenjavo izbranih genov.

3.5.2 Vrivanje - Insertion Mutation (ISM)

Naključno se izbere en gen (*) in mesto vrivanja (|). Izbrani gen se prestavi pred mestom vrivanja. Slika 3.13 prikazuje primer, kjer je izbrani gen za zamenjavo 3 in mesto vrivanja gen 5.

Genom:	1	2	3	4	5	6	7	8	9	10
Mesto:			*			|				
Potomec:	1	2	4	5	3	6	7	8	9	10

Slika 3.13: Mutacija z vrivanjem izbranega gena.

3.5.3 Preprosta inverzija - Simple Inversion Mutation (SIM)

Mutacija z inverzijo genov izbranim genom v genomu obrne vrstni red naključno izbranemu nizu elementov v genomu. Slika 3.14 predstavlja primer, kjer se vrstni red genov zamenja med mestoma 3 in 7.

3.5.4 Inverzija - Inversion Mutation (IMV)

Operator inverzije je podoben spodaj opisanemu premostitvenemu operatorju. Operator najprej izbere naključni niz elementov v genomu. Ta niz je odstranjen in vnesen na naključno izbrano mesto. Vendar je izbrani niz vnesen v nasprotnem vrstnem redu (slika 3.15).

Genom:	1	2	3	4	5	6	7	8	9	10
Mesto:			|			|				
Potomec:	1	2	6	5	4	3	7	8	9	10

Slika 3.14: Mutacija z obračanjem genov v izbranem območju.

Genom:	1	2	3	4	5	6	7	8	9	10
Mesto:			|			|			*	
Potomec:	1	2	7	8	6	5	4	3	9	10

Slika 3.15: Mutacija s prestavitvijo in obrnjenim vrstnim redom niza elementov.

3.5.5 Mutacija premešanja - Scramble mutation (SM)

Mutacija s premestitvijo genov deluje na enak princip kot inverzija, le da se izbrani geni med seboj naključno zamenjajo. Slika 3.16 prikazuje naključno menjavo genov med mestoma 3 in 7.

Genom:	1	2	3	4	5	6	7	8	9	10
Mesto:			|				|			
Potomec:	1	2	5	3	4	6	7	8	9	10

Slika 3.16: Mutacija s premestitvijo genov v izbranem območju.

3.5.6 Premestitvena mutacija - Displacement mutation (DM)

Operator najprej izbere naključni niz elementov v genomu. Ta niz je odstranjen in vnesen na naključno izbrano mesto. Slika 3.17 predstavlja uporabo operatorja DM. Z * je označeno mesto, kamor se prepiše izbrani niz elementov.

Genom:	1	2	3	4	5	6	7	8	9	10
Mesto:			|		*	|				
Potomec:	1	2	7	8	3	4	5	6	9	10

Slika 3.17: Mutacija s prestavitvijo niza elementov.

3.6 Vrednotenje

V podpoglavju 3.1 smo omenili, da pri PTP možne rešitve ne predstavimo v obliki bitnega niza temveč z bogatejšo podatkovno strukturo, npr. z vektorjem ali matriko. Tako je izbira funkcije uspešnosti zelo naravna in enostavna, saj jo lahko enačimo npr. z razdaljo med mesti, ki jih mora trgovski potnik obiskati.

Nekateri viri [9] za primerno funkcijo uspešnosti navajajo evklidsko razdaljo, ki je zaokrožena na najbližje celo število (enačba 3.10).

$$d = round\left(\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} + 0,5\right) \tag{3.10}$$

3.7 Migracije

Izmenjavo rešitev oz. genomov imenujemo migracije:

- migracijska stopnja določa, koliko genomov je določenih za izmenjavo,
- razpored migracije določa, kdaj se migracija zgodi,
- topologija povezav med posameznimi podpopulacijami.

Migracija vpliva na kakovost iskanja [1] in učinkovitost izvajanja algoritma. Pogosta migracija ima na primer za posledico veliko izmenjavo potencialno uporabnega genetskega materiala, vendar hkrati tudi negativno vpliva na izvedbo, saj so komunikacije časovno zahtevne. Podobno se zgodi z gosto povezanostjo, kjer vsaka podpopulacija komunicira s številnimi drugimi. Glavni cilj je hitro najti dobro rešitev in zato je nujno najti ravnotežje med stroški, ki so posledica migracije, in povečanjem možnosti za najdbo dobre rešitve.

Poglavje 4

Arhitektura CUDA

Poglavje povzema [3–5]. CUDA, splošnonamenska visoko vzporedna računska arhitektura (angl. *Compute Unified Device Architecture*), je platforma za vzporedno računanje in programski model, ki ga je razvilo podjetje NVIDIA [4]. Omogoča dramatično povečanje računske učinkovitosti z uporabo sredstev, ki jih nudijo GPE.

Njen razvoj je imel več ciljev:

- Zagotoviti majhen nabor razširitev k standardnim programskim jezikom, kot so C, ki omogoča enostavno implementacijo vzporednih algoritmov. Z razširitvijo CUDA C/C++ se programerji lahko posvetijo sami nalogi paralelizacije algoritma, namesto da bi izgubljali čas na njeni implementaciji.

- Podpirati heterogeno računanje, kjer aplikacije uporabljajo tako CPE kot GPE. Izrazito zaporedni deli programa se izvajajo na CPE, vzporedni deli pa se naložijo na GPE. To omogoča postopno dodajanje arhitekture CUDA k obstoječim aplikacijam. CPE in GPE se smatrata kot ločeni napravi, ki imata vsaka svoj pomnilniški prostor. Taka delitev omogoča sočasno računanje na CPE in GPU, brez da bi prihajalo do tekmovanja za pomnilniške vire.

GPE z CUDA platformo imajo na stotine jeder, ki lahko kolektivno izvajajo na tisoče računskih niti (angl. *computing threads*). Jedra imajo skupne vire, med drugim registre (angl. *register file*) in deljeni pomnilnik (angl. *shared memory*). Deljeni pomnilnik omogoča, da vzporedna opravila, ki se izvajajo na teh jedrih, delijo podatke brez prenosov prek sistemskega pomnilniškega vodila.

4.1 Kdaj uporabiti platformo CUDA

Programabilne GPE so se razvile v visoko vzporedne, večnitne, večjedrne procesorje z velikimi računskimi sposobnostmi [3] in zelo veliko pomnilniško pasovno širino. Razlog, da so GPE prehitele CPE v računskih sposobnostih s plavajočo vejico, je predvsem v tem, da so GPE posebej načrtovane za računsko intenzivne, visoko vzporedne računske probleme. Najbolj pogost problem te vrste je grafični izris slik (angl. *graphics rendering*). GPE so oblikovane tako, da je več tranzistorjev namenjenih procesiranju podatkov kot pa predpomnenju podatkov (angl. *data caching*) in krmiljenju pretoka (angl. *flow control*), kot je to običajno pri CPE.

GPE so še posebej primerne za probleme,

- v katerih lahko izrabimo podatkovno vzporednost (enak program se vzporedno izvaja na veliko podatkovnih elementih),
- ki so aritmetično zahtevni (visoko razmerje med aritmetičnimi in pomnilniškimi operacijami).

Ker se enak program izvaja za vsak podatkovni element, je manjša potreba po sofisticiranem krmiljenju pretoka. Izvajanje na večih podatkovnih elementih in aritmetična zahtevnost pa pripomoreta tudi k temu, da se zakasnitve pomnilniških dostopov skrijejo z izračunavanji namesto z velikimi podatkovnimi predpomnilniki.

Podatkovno vzporedno procesiranje mapira podatkovne elemente na vzporedne računske niti (angl. *parallel processing threads*). Veliko aplikacij, ki obdelujejo velike podatkovne zbirke lahko uporabijo podatkovno vzporedni programski model za pohitritev izračunov. V praksi se uporablja za zelo raznolike probleme, vse od obdelave slik in videa, procesiranje signalov, fizikalnih simulacij, do računske biologije.

4.2 Programski model

4.2.1 Ščepec

CUDA C razširja programski jezik C, tako da lahko programer definira funkcije, imenovane ščepci (kernels), ki se bodo izvajale na GPE. Ščepec je definiran z uporabo deklaracije `__global__`. Ob klicu je potrebno podati število CUDA niti, ki naj pripadajoč ščepec izvedejo. Tako se isti ščepec izvede vzporedno na N različnih CUDA nitih. V ta namen se uporablja posebna sintaksa `<<<...>>>`.

4.2.2 Hierarhija niti

Vsaka nit, ki izvaja ščepec, ima unikaten indeks, v obliki vektorja s tremi dimenzijami, do katerega lahko dostopamo prek vgrajene spremenljivke `threadIdx`. To nam omogoča identifikacijo niti s pomočjo ene, dveh ali treh komponent vektorja `threadIdx` in tako tvorimo eno-, dvo- ali tridimenzionalen blok niti (angl. *thread block*). To omogoča naravnejšo delitev problemov, saj lahko izvedemo računske operacije čez vse elemente domene, kot so vektor, matrika ali prostor. Število niti v bloku je omejeno, saj se pričakuje, da se bodo vse niti bloka nahajale na istem procesorskem jedru in si morajo deliti omejene pomnilniške vire tega jedra. Trenutno je ta omejitev do 1024 niti.

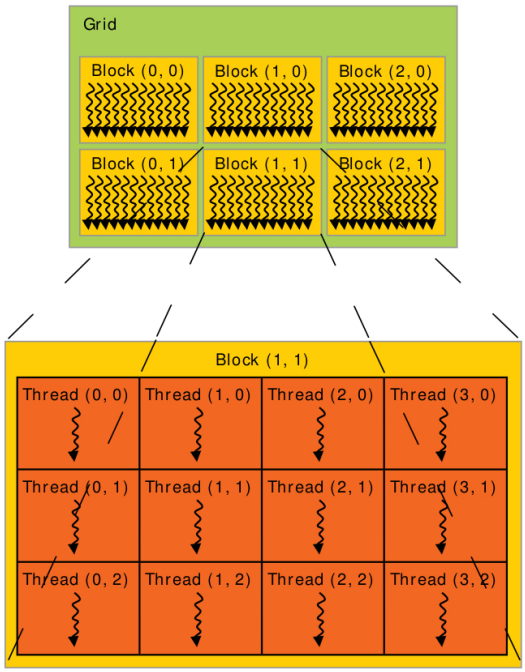
Kodni izsek 4.1 prikazuje ščepec za seštevanje matrik A in B v matriko C. Vse tri matrike so velikosti NxN.

Kodni izsek 4.1 Ščepec za seštevanje dveh matrik velikosti NxN

```
1: // Kernel definition
2: __global__ void MatAdd(float A[N][N], float B[N][N],
3:                         float C[N][N])
4: {
5:   int i = threadIdx.x;
6:   int j = threadIdx.y;
7:   C[i][j] = A[i][j] + B[i][j];
8: }
9:
10: int main()
11: {
12:   ...
13:   //Kernel invocation with one block of N * N * 1 threads
14:   int numBlocks = 1;
15:   dim3 threadsPerBlock(N, N);
16:   MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
17:   ...
18: }
```

Ščepec lahko izvede več blokov niti, ki imajo enake dimenzije. Celotno število niti je enako produktu števila niti v bloku in številu blokov.

Bloke lahko organiziramo v eno-, dvo- ali tridimenzionalno mrežo (angl. *grid*), kot to prikazuje slika 4.1. Število blokov v mreži običajno določa velikost podatkov, ki jih procesiramo.



Slika 4.1: Mreža sestavljena iz blokov niti.

4.2.3 Pomnilniška hierarhija

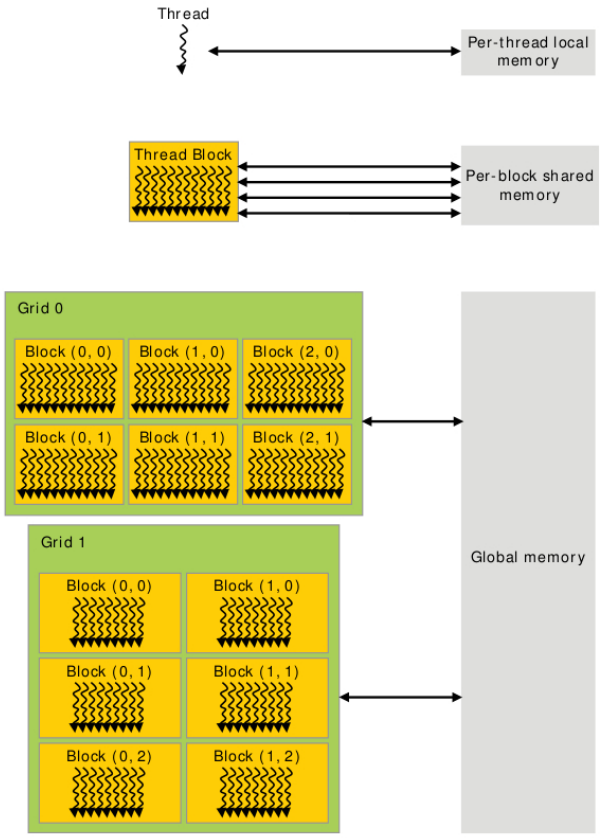
Niti CUDA lahko med izvajanjem dostopajo do podatkov iz različnih pomnilniških prostorov (slika 4.2). Vsaka nit ima lasten lokalni pomnilnik (angl. *private local memory*). Vsak blok niti ima deljeni pomnilnik, do katerega lahko dostopajo vse niti iz pripadajočega bloka. Vse niti imajo dostop do istega globalnega pomnilnika (angl. *global memory*).

Obstajata tudi dva dodatna pomnilniška prostora, ki dovolita samo branje. Gre za pomnilniški prostor za konstante (angl. *constant memory*) in teksture (angl. *texure memory*), do katerih lahko dostopajo vse niti.

Globalni, konstantni in teksturni pomnilniški prostor so optimizirani za različne vrste pomnilniških dostopov. Njihova vsebina se ohranja med klici ščepcev.

4.3 Analiza izvedbe

S pomočjo analize časa izvajanja vzporednega programa lažje razumemo ovire za doseganje boljših rezultatov [5] in spoznamo odvisnost časa izvajanja od



Slika 4.2: Pomnilniška hierarhija.

števila procesorjev. Operacije v vzporednem algoritmu lahko razvrstimo v tri skupine:

- operacije, ki se morajo izvajati zaporedno (v nadaljevanju zaporedne operacije),
- operacije, ki se lahko izvajajo vzporedno (v nadaljevanju vzporedne operacije) in
- operacije komunikacije in redundantnega računanja, ki predstavljajo strošek paralelizacije.

4.3.1 Pohitritev in uspešnost

Razvoj in izvedbo vzporednih programov spremlja pričakovanje, da bo njihovo izvajanje hitrejše od ustrezne zaporedne različice. Zato je pohitritev ψ definirana kot razmerje med časom izvajanja zaporednega algoritma in časom izvajanja vzporednega algoritma (enačba 4.1).

$$\psi = \frac{t_{zaporednega\ algoritma}}{t_{vzporednega\ algoritma}} \tag{4.1}$$

Uspešnost ε (angl. *efficiency*) vzporednega programa je mera za izkoriščenost procesorjev. Definirana je kot razmerje pohitritve in števila procesorjev p (enačba 4.2).

$$\varepsilon = \frac{\psi}{p} \tag{4.2}$$

4.3.2 Amdahlov zakon

Označimo z $\psi(n, p)$ pohitritev problema velikosti n s p procesorji in s f delež zaporednih operacij, ki jih ne moremo pohitriti. Potem velja izraz 4.3, ki ga imenujemo Amdahlov zakon. Določa največjo možno pohitritev algoritma s vzporednim računalnikom s p procesorji. Zasnovan je na predpostavki, da želimo rešiti problem določene velikosti čim hitreje in postavlja zgornjo mejo pohitritve. Na primer, pri $f = 10\%$ je pohitritev z dvema procesorjema $1,82$, z desetimi procesorji $5,26$, s sto procesorji pa $9,17$. Če gre število procesorjev p proti neskončnosti, gre pohitritev proti $1/f$ oziroma v našem primeru proti $1/0,1 = 10$. To tudi pomeni, da nujni zaporedni del računanja omejuje pohitritev.

Kadar pri konstantnem številu procesorjev pohitritev narašča z velikostjo problema, govorimo o Amdahlovem učinku.

$$\psi(n, p) \leq \frac{1}{f + \frac{1-f}{p}} \tag{4.3}$$

Poglavje 5

Vzporedni algoritem na GPE in implementacija

Vzporedni algoritem uporablja platformo za vzporedno računanje CUDA in se izvaja na GPE, pri operaciji redukcije pa sodeluje tudi CPE.

5.1 Delitev

Predpostavimo, da rešujemo dvodimenzionalni PTP (mesta imajo dve koordinati) velikosti N mest, z delitvijo $[2, 7]$ na b blokov, kjer vsak blok vsebuje t niti. Vsaka nit predstavlja enega izmed genomov v GA. Vseh genomov je tako $g = b \cdot t$. Vsak blok niti predstavlja neodvisno populacijo nad katero se izvaja GA. Z m označimo število genomov, ki se prenesejo ob migraciji, če je le-ta omogočena. Manjša ko je cena rešitve, bolj uspešna je, saj predstavlja hitrejšo oz. cenejšo pot.

5.2 Inicializacija

CPE s pomočjo funkcije `cudaMalloc` na GPE rezervira pomnilnik za več enodimenzionalnih tabel. Med njimi so:

- tabela (velikosti $2 * N$) koordinat mest,
- dve tabeli (velikosti $g * N$) za hranjenje rešitev,
- tabela (velikosti g) v kateri se hranijo uspešnosti rešitev,
- tabela (velikosti g), ki hrani stanja generatorja naključnih števil,

- tabela (velikosti $m*b$) za hranjenje indeksov genomov, ki bodo migrirali,
- tabela (velikosti $m*b*N$) za hranjenje genomov, ki bodo migrirali,
- tabela (velikosti b) za hranjenje najboljših najdenih cen po blokih.

Zatem CPE s funckijo `cudaMemcpy` sproži prenos koordinat mest v globalni pomnilnik GPE. S klicem ščepca `kernelInitSeed` nad b bloki in t nitmi, se inicializirajo generatorji naključnih števil (kodni izsek 5.1). Naključna števila se generirajo s pomočjo knjižnice cuRAND.

Kodni izsek 5.1 Ščepec za inicializacijo generatorjev naključnih števil

```
1: __global__ void kernelInitSeed(curandState * state,
2:                                unsigned long seed)
3: {
4:   int id = blockIdx.x * blockDim.x + threadIdx.x;
5:   curand_init (seed, id, 0, &state[id]);
6: }
```

Sledi klic ščepca `kernelInitGenomes` nad b bloki in t nitmi, ki poskrbi za inicializacijo genomov. Vsaka nit izračuna indeks, ki določa, kateri genom v globalnem pomnilniku bo inicializirala. V prvi fazi se vse genome nastavi na rešitev $0, 1, 2, ..., N-1$. V drugi fazi se generira naključno celo število z intervala $[0, 2*N]$. Gre za število zamenjav, ki jih bomo izvedli nad začetno rešitvijo. V sklopu vsake zamenjave naključno izberemo dve mesti znotraj rešitve in njuni vrednosti zamenjamo. V kolikor uporabljamo migracije, se prvih m genomov iz vsakega bloka prenese v tabelo, ki hrani migracijske genome. Sledi klic funkcije `evaluate` (kodni izsek 5.2), s katerim niti izračunajo, kako uspešne so njihove rešitve. Vrednosti se shranijo v pripadajočo tabelo v globalnem pomnilniku.

Kodni izsek 5.2 Funkcija za izračun cene povezave med mestom i in j

```
1: __device__ unsigned int evaluatePath(int *places, int i,
2:                                      int j)
3: {
4:   double xd = places[i * 2] - places[j * 2];
5:   double yd = places[i * 2 + 1] - places[j * 2 + 1];
6:   return (unsigned int) (sqrt(xd * xd + yd * yd) + 0.5);
7: }
```

5.3 Iskanje najboljše cene z redukcijo

Med izvajanjem algoritma je potrebno ugotoviti, kako uspešne rešitve so bile odkrite. Manjša, ko je cena poti, bolj uspešna je rešitev. Iščemo torej najmanjšo ceno, oz. največjo uspešnost. Uspešnost obravnavamo na ravni blokov, ki predstavljajo neodvisne populacije in globalno na ravni celotnega algoritma. Algoritem s pomočjo ščepca `kernelReductionBestPrice` (kodni izsek 5.3) poišče najboljšo rešitev v bloku. V izvajanje se pošlje b blokov in t niti. Gre za iskanje najmanjše cene poti z operacijo redukcije. Redukcija je splošen postopek [5, 10], pri katerem vzamemo tabelo vhodnih podatkov in s pomočjo določenih izračunov generiramo manjšo tabelo rezultatov. Redukcija se pogosto pojavi pri vzporednem računanju. Ko se ščepec konča, CPE prenese b najboljših cen v glavni pomnilnik in poišče najmanjšo ceno. CPE dokonča delo GPE, saj bi bila slednja neizkoriščena v zadnjih korakih redukcije. Velikost podatkov je premajhna, da bi lahko efektivno izkoristili vse aritmetično logične enote. Rezultat opisanega postopka je pridobitev najboljše uspešnosti iz vseh populacij.

Kodni izsek 5.3 Ščepec za izračun najboljše (najmanjše) cene v blokih

```
1:  __global__ void kernelReductionBestPrice(...)
2:  {
3:    __shared__ unsigned int shPrice[threadsPerBlock];
4:
5:    unsigned int id = blockIdx.x * blockDim.x +
6:                      threadIdx.x;
7:
8:    shPrice[threadIdx.x] = devPrices[id];
9:
10:   __syncthreads();
11:
12:   //Find best price with "reduction"
13:   int i = blockDim.x / 2;
14:   int shIndex = threadIdx.x;
15:   while (i != 0)
16:   {
17:     if (shIndex < i)
18:       shPrice[shIndex] = min(shPrice[shIndex],
19:                              shPrice[shIndex+i]);
20:     __syncthreads();
21:     i /= 2;
22:   }
```

```
23:
24:    if (shIndex == 0)
25:      devBestPrice[blockIdx.x] = shPrice[0];
26: }
```

5.4 Glavna zanka algoritma

Glavna zanka algoritma skrbi za izvajanje GA. Vsaka iteracija zanke s pomočjo ščepca `kernelEvolve` izvede n_{gen} generacij GA. V izvajanje se pošlje b blokov in t niti. Če so mutacije omogočene, sledi klic ščepca `kernelMigrate`, ki poskrbi za prenos predhodno izbranih dobrih genomov v tabelo za hranjenje migracijskih genomov. Pri tem blok i piše na lokacije, ki pripadajo bloku $(i+1)mod(b)$. Povedano drugače, dobri genomi iz trenutnega bloka se prenesejo tako, da jih bo ob naslednjem klicu ščepca `kernelEvolve` naslednji blok integriral v svojo populacijo. Sam prenos ni vezan na delitev genomov GA, zato lahko uporabimo drugačno število blokov in niti (npr. 32 blokov in 256 niti). V vsakem primeru je potrebno ugotoviti, kako uspešne so nove najboljše rešitve. Ponovno uporabimo postopek, ki ga opisuje podpoglavje 5.3.

5.4.1 Ščepec kernelEvolve

Deli se na tri večje sklope. Prvi in zadnji sklop sta omogočena le ob uporabi migracij.

Prvi sklop skrbi za prenos genomov iz tabele za hranjenje migracijskih genomov v populacijo bloka. Zamenjajo se slabše ocenjeni genomi, ki jih določimo s pomočjo turnirske selekcije. Tako dobro ocenjene rešitve iz prejšnjega bloka zamenjajo slabe rešitve v trenutem bloku.

Drugi sklop izvaja evolucijo rešitev. Gre za zanko, ki izvrši n_{gen} generacij. V vsaki iteraciji se izvrši sledeč postopek:

1. Izvrši se že predhodno omenjena funkcija `evaluate` (kodni izsek 5.2), s katerim niti izračunajo, kako uspešne so njihove rešitve. Vendar se tukaj vrednosti shranijo v deljeni pomnilnik bloka in ne v globalni pomnilnik. Sledi namreč turnirska selekcija, pri kateri vsaka nit potrebuje izračunano tabelo uspešnosti. Zagotoviti moramo, da so vse niti končale z izračunom, preden se začne izvajati selekcija. To dosežemo s sinhronizacijskim stavkom `__syncthreads()`.

2. Selekcija vsaki niti izbere dve rešitvi, ki bosta služili kot starša pri operaciji križanja.

3. S pomočjo generatorja naključnih števil zagotovimo, da je operacija križanja pogojena z verjetnostjo in se ne izvaja v vsaki iteraciji. Križanje poskuša združiti dobre lastnosti izbranih staršev. V implementaciji se izvede ena izmed vrst križanja, ki so opisane v podpoglavju 3.4. Katero vrsto uporabljamo, določa parameter, ki ga določimo pred prevajanjem programa. Zaradi sočasnega spreminjanja večih rešitev se pri operaciji križanja uporabljata dve tabeli. V prvi se nahaja populacija rešitev iz predhodnje iteracije, v drugo pa z operatorjem križanja gradimo nove rešitve. Vsaka nit pri križanju spreminja le svojo rešitev. Če do križanja ne pride, se rešitev enostavno prenese v drugo tabelo.

4. Sledi operacija mutacije, ki je prav tako pogojena z verjetnostjo. Vsaka nit mutira svojo rešitev, ki se nahaja v drugi tabeli, tako da sinhronizacija ni potrebna.

5. Kazalca na tabeli rešitev moramo zamenjati, saj sedaj druga vsebuje dokončane rešitve. V novi iteraciji se bodo tako nove rešitve gradile v prvo tabelo. Zaradi zamenjav mora biti število izvedenih generacij n_{gen} vedno sodo. Pred zamenjavo moramo počakati, da vse niti končajo s spreminjanjem rešitev, saj bo nova iteracija zahtevala ponovno ovrednotenje uspešnosti.

Ko se zanka konča, se rešitve ovrednotijo in uspešnosti shranijo v globalni pomnilnik.

Naloga tretjega sklopa je določitev indeksov dobrih rešitev danega bloka, ki jih nameravamo v prihodnosti poslati naslednjemu bloku. Ponovno si pomagamo s turnirsko selekcijo. Na tem mestu se samo pošiljanje še ne izvrši, saj ni zagotovila, da so vsi bloki ščepca `kernelEvolve` že končali z evolucijo svojih rešitev.

5.5 Zaključni del algoritma

Iz GPE prenesemo tabelo rešitev in tabelo uspešnosti nazaj v glavni pomnilnik. Sledi opcijski preizkus pravilnosti rešitev s ščepcem `kernelGenomesValid`. Rešitve se izpišejo v preprostem tekstovnem formatu in v razširljivem označevalnem jeziku (XML). CPE s pomočjo funkcije `cudaFree` na GPE sprosti pomnilnik, ki so ga zasedale tabele.

Poglavje 6

Vzporedni algoritem na CPE in implementacija

Z namenom primerjave je bil razvit tudi vzporedni algoritem, ki se izvaja na CPE. V osnovi je bila predvidena izgradnja zaporednega algoritma za CPE, vendar se je kasneje izkazalo, da je bolj priročno uporabiti implementacijo, ki omogoča nastavljanje števila niti in s tem nivoja vzporednosti. Po zgradbi in zaporedju operacij poskuša posnemati strukturo vzporednega algoritma, ki uporablja GPE. Vzporednost se doseže s uporabo niti, ki jih nudi knjižnica Pthreads.

6.1 Delitev

Uporabimo enake oznake kot v podpoglavju 5.1. Rešitve razdelimo med niti. Indeks prve in zadnje rešitve v niti $thread_{id} \in [0, t-1]$, podajata izraza 6.1 in 6.2. Pri tem velja opozoriti, da še vedno uporabljamo več blokov, ki predstavljajo neodvisne populacije. Niti morajo obravnavati vsako rešitev, glede na to, v katero populacijo spada. Tudi ob omogočenih migracijah se dodatno delo razdeli na že obstoječe niti.

$$first_{id} = \frac{g}{t} \cdot thread_{id} \tag{6.1}$$

$$last_{id} = \frac{g}{t} \cdot (thread_{id} + 1) - 1 \tag{6.2}$$

6.2 Inicializacija

Je zelo podobna postopku, ki je opisan za GPE v podpoglavju 5.2. Največja razlika je v tem, da se obdelovanje več rešitev razdeli na posamezno CPE nit. Tabele za hranjenje najboljših najdenih uspešnosti po nitih ne uporabljamo, saj se za ta namen uporablja sinhronizacija in ena sama spremenljivka.

6.3 Iskanje najboljše cene z redukcijo

Vsaka nit med rešitvami, ki so ji dodeljene, poišče najbolj uspešno. Sledi primerjava uspešnosti z drugimi nitmi. Ko več niti sočasno uporablja deljeno spremenljivko, lahko pride do tekmovalnih okoliščin [5] (angl. *race condition*), kar vodi v nedeterministično računanje. Zaradi tega je potrebna uporaba sinhronizacije. Vsaka nit med primerjanjem vstopi v kritično sekcijo `pthread_mutex_lock(mutex)`, tako da deljeno spremenljivko spreminja največ ena nit naenkrat. V deljeni spremenljivki je največja uspešnost šele takrat, ko vse niti končajo s primerjavo. Zaradi tega je potrebno počakati na vse niti, kar dosežemo s sinhronizacijskim stavkom `pthread_barrier_wait(barrier)`.

6.4 Glavna zanka algoritma

Uporabljajo se enaki sklopi kot pri algoritmu z GPE (podpoglavje 5.4). Tudi tukaj vsaka iteracija zanke izvrši funkcijo `kernelEvolve`, ki izvede n_{gen} generacij GA. Prav tako obstaja ekvivalent funkciji `kernelMigrate`, ki skrbi za migracije.

6.4.1 Funkcija kernelEvolve

Prvi in zadnji sklop sta omogočena le ob uporabi migracij. V prvem sklopu se delo, ki ga terjajo migracije, razdeli med obstoječe niti. Indeks prvega in zadnjega migranta, ki pripadata niti $thread_{id} \in [0, t-1]$, podajata izraza 6.3 in 6.4. Enako delitev uporablja tudi tretji sklop.

$$first\ migrant_{id} = \frac{m}{t} \cdot thread_{id} \tag{6.3}$$

$$last\ migrant_{id} = \frac{m}{t} \cdot (thread_{id} + 1) - 1 \tag{6.4}$$

Drugi sklop prav tako izvaja evolucijo s pomočjo zanke, ki izvrši n_{gen} generacij. Opišimo nekaj ključnih razlik v implementaciji:

1. Vsaka nit izvrši funkcijo `evaluate` le nad svojim delom populacije. Rezultati se shranjujejo v glavni pomnilnik. Ker se pri selekciji uporablja uspešnosti vseh rešitev, moramo z ukazom `pthread_barrier_wait (barrier)` počakati, da z izračuni končajo vse niti.

2. Treba se je zavedati, da še vedno uporabljamo več blokov, ki predstavljajo med seboj neodvisne populacije. Pri selekciji in izbiri rešitev mora vsaka nit upoštevati informacijo, h kateremu bloku populacije pripada določena uspešnost in selekcijo vršiti samo nad ustreznim blokom.

3. Niti lahko neodvisno izvajajo križanje nad svojimi rešitvami, saj uporabljamo dve ločeni tabeli.

4. Pri mutacijah ni bistvenih sprememb.

5. Tudi v zadnjem koraku ni sprememb.

Ko se zanka konča, se rešitve shranijo v glavni pomnilnik.

6.5 Zaključni del algoritma

Tudi ta algoritem nudi opcijsko preverjanje pravilnosti rešitev in shranjevanje rešitev. Na koncu CPE sprosti zasedeni glavni pomnilnik.

Poglavje 7

Meritve in rezultati

7.1 Izvajalno okolje

Vse meritve so se izvajale na operacijskem sistemu Linux (Ubuntu 12.04.3 LTS). Reševali smo vnaprej pripravljene probleme, ki so prosto dostopni na spletu [9]. Za merjenje sta se uporabljali dve različni CPE in ena GPE, kot je razvidno iz tabele 7.1. Pri meritvah nas je predvsem zanimalo:

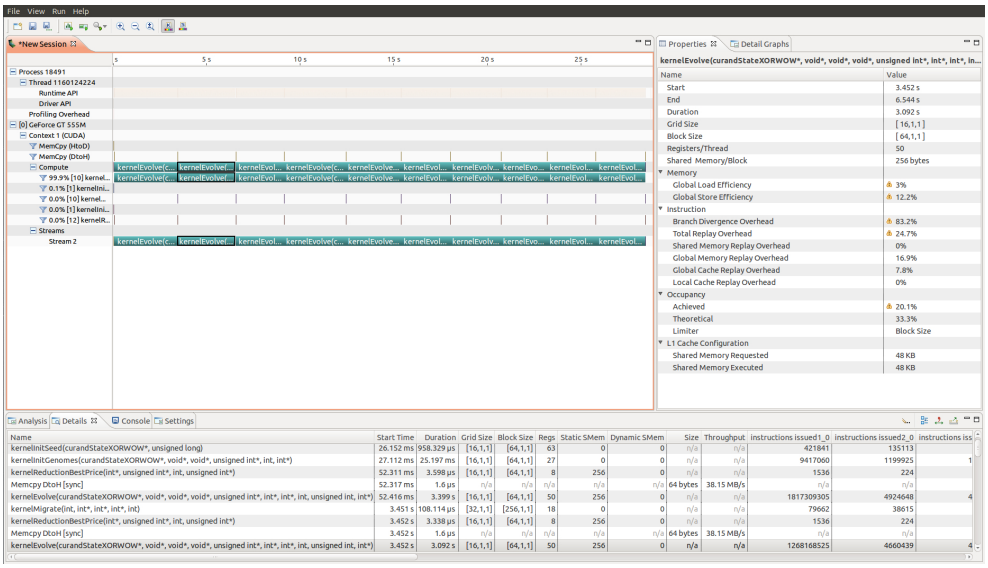
- kakšne pohitritve doseže vzporedni algoritem, ki se izvaja na GPE, v primerjavi s tistim na CPE,
- kako se obnaša CPE algoritem, če ga izvajamo na večjem številu niti,
- ali migracije res vpivajo na izboljšanje rešitev,
- kateri par križanja in mutacije v poprečju razvije najboljše rešitve.

NASTAVITVE	
CPE	
Model	Intel Core2 Quad CPU Q6600
Model	Intel Core2 CPU 6400
GPE	
Model	GeForce GT 555M 2GB

Tabela 7.1: Skupne nastavitve pri merjenju pohitritev.

7.2 Pohitritve

Potek meritve in numerične rezultate bolj podrobno opisuje dodatek A. Vzporedni algoritem na GPE v veliki večini primerov ne deluje hitreje kot algoritem na CPE. Pohitritev je večinoma manjša od ena ($\psi < 1$). Časovna analiza različnih delov algoritma je pokazala, da se večina časa porabi pri operacijah križanja. Program NVIDIA Visual Profiler (slika 7.1), ki se uporablja za analizo CUDA programov, je opozoril na zelo slabo izrabo globalnega pomnilnika. Pri trenutni delitvi, ki jo uporablja tudi članek [2], vsaka nit predstavlja rešitev in jo med izvajanjem spreminja z mutacijami in križanji. Genetski operatorji pa so v veliki meri pogojeni z verjetnostmi (npr. naključna izbira intervala križanja). Posledično vsaka nit samostojno dostopa do popolnoma drugega dela globalnega pomnilnika, kar povzroči serializacijo pomnilniških dostopov. Članek [2] je bil bolj uspešen pri paralelizaciji algoritma, saj so se odločili za relativno preprost operator križanja. V dveh izbranih starših so naključno izbrali tri zaporedna mesta in jih permutirali glede na vrstni red, ki so ga števila zavzela po velikosti.



Slika 7.1: Zgled analize algoritma z uporabo orodja NVIDIA Visual Profiler.

Za GPE bi bila primernejša delitev problema, pri kateri bi več niti sodelovalo pri mutacijah in križanjih. Ker so rešitve PTP, v resnici permutacije, kjer je vrednost vsakega gena v genomu unikatna, to ni enostavno doseči.

Ker naključna števila pogojujejo količino dela, ki se izvrši v genetskih ope-

ratorjih, prihaja do neuravnoteženosti števila in tipov ukazov, ki jih morajo izvršiti posamezne niti. Posledica je vrivanje čakalnih ciklov in seveda daljše izvajanje.

Paralelizacija zaporednega algoritma, ki ga izvaja CPE, je uspešnejša. S povečevanjem števila procesorskih niti čas izvajanja pada. Podrobnosti so navedene v dodatku D.

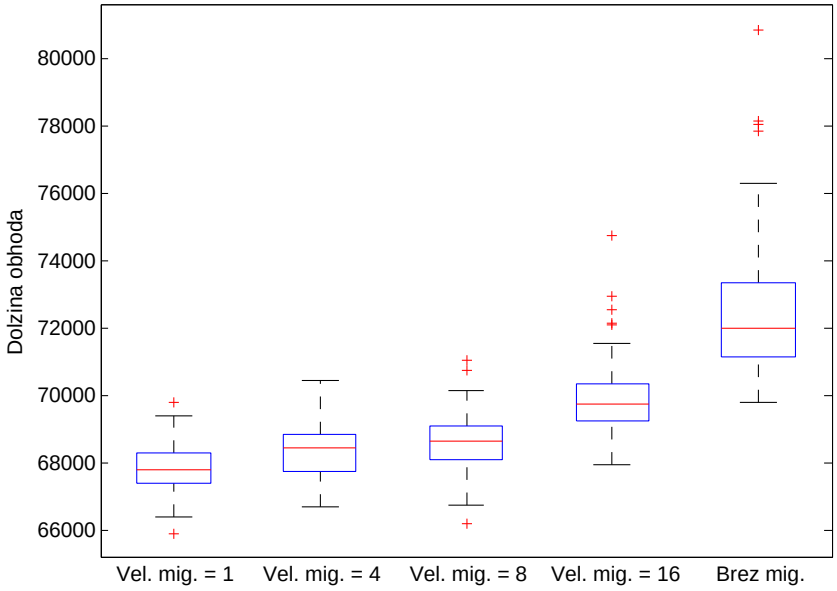
7.3 Vpliv migracij

Uspešnost rešitve podaja dolžina njenega obhoda. Uporabljamo vrednotenje z evklidsko razdaljo, ki je opisano v podpoglavju 3.6. Naš cilj je poiskati najkrajši obhod (glej poglavje 1), kar pomeni da uspešnost raste s padanjem dolžine obhoda.

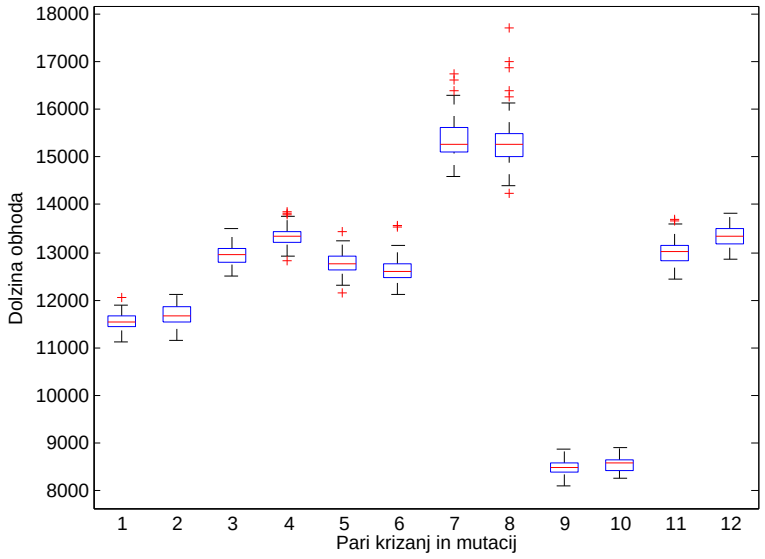
Pri meritvi smo spreminjali velikost migracij in opazovali vpliv na uspešnost končnih rešitev. Ker GA uporabljajo naključna števila, moramo upoštevati večje število meritev. Vsaka nastavitev se je merila na 100 izvedbah CPE algoritma. Tako smo prišli do množic končnih uspešnosti oz. dolžin obhodov za posamezne velikosti migracij. Njihove porazdelitve prikazuje slika 7.2. Izkazalo se je, da lahko delitev populacije na več neodvisnih blokov in simuliranje emigracij pozitivno učinkujeta na uspešnost rešitev. V našem primeru se uporablja topologija obroča, pri čemer vsak blok posreduje genetski material k naslednjemu bloku. Manjša ko je velikost emigracij, večja je dosežena uspešnost. Meritev bolj podrobno opisuje dodatek B.

7.4 Najboljši nabor operatorjev

O poteku meritve govori dodatek C. Spreminjali smo pare operatorjev križanja in mutacije ter s 100 izvedbami algoritma prišli do množic končnih uspešnosti oz. dolžin obhodov. Njihovo porazdelitev prikazuje slika 7.3, pripadajočo legendo pa podaja tabela C.2. Videti je, da najboljše rešitve razvijeta mutacija SIM in eno izmed križanj (OX ali PMX). Ker je mutacija SIM prisotna v obeh primerih, bi lahko upravičeno sklepali, da ima vodilno vlogo pri zagotavljanju konvergiranja k dobrim rešitvam. Treba je opozoriti, da zelo verjetno obstaja močna korelacija med nastavitvami, s katerimi smo poganjali algoritem, in rezultati. Obstaja verjetnost, da bi pri drugačnih vrednostih parametrov dobili povsem drugačne rezultate.



Slika 7.2: Vpliv velikosti migracij na porazdelitev dolžin obhodov.



Slika 7.3: Vpliv različnih naborov križanj in mutacij na porazdelitev dolžin obhodov.

Poglavje 8

Sklepne ugotovitve

V okviru diplomske naloge smo razvili dva algoritma, ki rešujeta problem trgovskega potnika s pomočjo genetskih algoritmov. Prvi je vzporeden in se izvaja na GPE s pomočjo platforme CUDA, drugi pa je zaporedne narave in za izvajanje uporablja CPE. Časovna primerjava tako različnih algoritmov včasih ni upravičena. Zaradi tega smo posebno pozornost namenili zagotovilu, da oba algoritma izvajata enako implementacijo operatorjev križanja in mutacije. Na to odločitev je v večji meri vplivala razširjenost literature [1, 6], ki nudi mnogo različnih implementacij zaporednih križanj in mutacij. Prav tako je pogosta delitev, pri kateri vsaki niti pripada samo ena rešitev [2], kar omogoča neposreden prepis implementacije operatorjev iz enega algoritma v drugega. V nasprotju s pričakovanji vzporedni algoritem ni dosegel pohitritev. Analiza in primerjava s člankom, ki je pri enaki delitvi problema dosegel pohitritev [2], je pokazala, da večino časa porabijo križanja. Drugi avtorji so jih namreč implementirali na preprostejši način. Naša prizadevanja, da bi obstoječe operatorje spremenili v vzporedne, niso uspela. Po navedenem je moč sklepati, da je implementacija vzporednih križanj za problem trgovskega potnika zahtevna naloga.

Uspešno smo pokazali, da migracije pozitivno vplivajo na povprečno uspešnost rešitev. Vpliv raste z manjšanjem števila rešitev, ki se prenašajo med ločenimi populacijami.

Prikazali smo vpliv različnih naborov operatorjev na porazdelitev doseženih uspešnosti. Raziskava bi lahko nudila smernice za paralelizacijo najuspešnejših križanj.

Zaporedni algoritem za CPE smo na koncu uspešno pretvorili v vzporedno obliko. S povečevanjem števila procesorskih niti čas izvajanja pada.

Dodatek A

Meritve pohitritev

Vsi časi so podani v sekundah. Algoritem se je izvajal na GPE in CPU pri:

- različnih parih operatorjev križanja in mutiranja,
- različnih velikostih problema (število mest),
- različnih delitvah problema (število blokov in niti na blok pri GPE),
- algoritem na CPE je vedno uporabljal samo eno nit.

Spodnje tabele podajajo čas, potreben za razvoj rešitev, pri nastavitvah algoritma, ki jih prikazuje tabela A.1. Podane so tudi pohitritve ψ.

NASTAVITVE	
Izvajanje evolucije	
Št. klicev ščepca kernelEvolve	10
Št. generacij na klic ščepca kernelEvolve	10
Selekcija	
Oblika selekcije	Turnirska selekcija
Velikost turnirja	16
Verjetnost izbire najboljšega	0.8
Križanja	
Verjetnost križanja	0.8
Mutacije	
Verjetnost mutacije	0.2
Migracije	
Omogočene	DA
Št. izvedenih migracijskih sklopov	10
Velikost migracij	4
Naključna števila	
Generator naključnih števil	Inicializiran na trenuten čas
Centralna procesna enota	
Model	Intel Core2 Quad CPU Q6600
Št. uporabljenih niti	1
Grafična kartica	
Model	GeForce GT 555M 2GB

Tabela A.1: Skupne nastavitve pri merjenju pohitritev.

A.1 Število blokov na mrežo je enako 16

256 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	3.3	2.1	1.57
1083	118.8	146.9	0.80
2036	426.6	595.9	0.71
2566	429.7	926.5	0.46
3649	908.8	1999.9	0.45
5087	1376.7	3132.2	0.43
10150	6249.0	14504.6	0.43
128 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	1.7	1.2	1.41
1083	39.1	78.4	0.49
2036	137.0	322.6	0.42
2566	270.4	486.5	0.55
3649	490.1	1111.1	0.44
5087	676.3	1750.9	0.38
10150	3918.0	7832.7	0.50
64 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	0.8	0.6	1.33
1083	18.8	38.9	0.48
2036	78.7	145.6	0.54
2566	130.4	228.2	0.57
3649	199.2	498.1	0.39
5087	330.0	899.9	0.36
10150	1326.6	3613.1	0.36

Tabela A.2: Mutacija ISM, križanje OX

256 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	2.8	1.5	1.86
1083	48.5	82.1	0.59
2036	226.0	411.8	0.54
2566	283.3	543.8	0.52
3649	845.7	2078.4	0.40
5087	723.9	2079.6	0.34
10150	3104.3	9588.5	0.32
128 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	1.4	0.9	1.55
1083	22.6	47.7	0.47
2036	77.2	194.9	0.39
2566	110.0	280.7	0.39
3649	280.0	944.0	0.29
5087	354.9	1045.7	0.33
10150	1593.5	4777.1	0.33
64 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	0.7	0.4	1.75
1083	11.2	26.0	0.43
2036	49.6	142.5	0.34
2566	53.8	187.8	0.28
3649	127.6	584.2	0.21
5087	218.8	580.2	0.37
10150	1059.7	2753.4	0.38

Tabela A.3: Mutacija ISM, križanje PMX

256 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	3.3	2.1	1.57
1083	81.9	147.7	0.55
2036	289.0	598.3	0.48
2566	494.6	929.3	0.53
3649	907.8	1985.6	0.45
5087	1885.2	3085.1	0.61
10150	9221.4	14375.9	0.64
128 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	1.7	1.2	1.41
1083	46.0	79.1	0.58
2036	135.7	321.8	0.42
2566	198.9	484.1	0.41
3649	456.5	1131.7	0.40
5087	696.2	1719.4	0.40
10150	2706.5	7740.5	0.34
64 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	0.8	0.6	1.33
1083	18.6	38.5	0.48
2036	74.6	142.2	0.52
2566	91.1	223.8	0.40
3649	196.9	493.4	0.39
5087	422.1	879.9	0.47
10150	1902.5	3551.9	0.53

Tabela A.4: Mutacija EM, križanje OX

256 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	2.8	1.4	2.00
1083	50.8	77.7	0.65
2036	128.1	272.1	0.47
2566	272.9	431.2	0.63
3649	530.8	879.9	0.60
5087	1005.7	1716.2	0.58
10150	3768.1	6720.2	0.56
128 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	1.4	0.9	1.55
1083	21.8	45.9	0.47
2036	63.5	157.1	0.40
2566	95.6	250.2	0.38
3649	180.0	507.3	0.35
5087	392.8	985.5	0.39
10150	1260.3	3994.7	0.31
64 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	0.7	0.4	1.75
1083	10.8	21.8	0.49
2036	40.6	76.6	0.53
2566	47.8	123.7	0.38
3649	90.5	247.5	0.36
5087	170.1	484.0	0.35
10150	631.1	1951.9	0.32

Tabela A.5: Mutacija EM, križanje PMX

256 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	3.4	2.7	1.25
1083	115.4	186.0	0.62
2036	316.2	744.4	0.42
2566	465.7	1130.4	0.41
3649	981.9	2459.2	0.39
5087	2279.3	4000.5	0.56
10150	9794.6	18001.2	0.54
128 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	1.7	1.6	1.06
1083	41.8	102.9	0.40
2036	161.7	414.2	0.39
2566	215.5	610.3	0.35
3649	697.0	1416.4	0.49
5087	956.1	2326.8	0.41
10150	4159.1	9504.7	0.43
64 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	0.9	0.7	1.28
1083	20.1	49.9	0.40
2036	103.3	192.0	0.53
2566	99.4	288.6	0.34
3649	222.2	645.3	0.34
5087	358.0	1105.7	0.32
10150	1971.0	4679.3	0.42

Tabela A.6: Mutacija IMV, križanje OX

256 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	3.0	2.4	1.25
1083	65.0	131.5	0.49
2036	240.2	575.0	0.41
2566	324.7	803.5	0.40
3649	676.7	2157.5	0.31
5087	1253.4	2817.5	0.44
10150	4454.6	12486.6	0.35
128 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	2.2	1.3	1.69
1083	30.0	72.0	0.41
2036	92.9	308.4	0.30
2566	168.4	425.1	0.39
3649	342.1	1181.8	0.28
5087	613.1	1551.0	0.39
10150	2261.6	6772.4	0.33
64 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	0.7	0.7	1.00
1083	13.0	40.7	0.31
2036	46.0	175.2	0.26
2566	61.6	260.2	0.23
3649	159.9	671.1	0.23
5087	211.7	937.8	0.22
10150	1014.1	3914.4	0.25

Tabela A.7: Mutacija IMV, križanje PMX

256 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	3.4	2.1	1.61
1083	81.4	149.8	0.54
2036	292.7	596.5	0.49
2566	432.9	892.5	0.48
3649	1009.1	1987.8	0.50
5087	1383.8	3097.8	0.44
10150	6187.5	14101.8	0.43
128 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	1.8	1.2	1.50
1083	39.2	78.3	0.50
2036	132.6	310.7	0.42
2566	189.0	455.3	0.41
3649	510.9	1100.3	0.46
5087	740.0	1856.7	0.39
10150	3917.1	7135.8	0.54
64 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	0.9	0.6	1.50
1083	18.8	38.2	0.49
2036	82.0	133.7	0.61
2566	89.3	211.7	0.42
3649	256.1	471.3	0.54
5087	352.6	873.0	0.40
10150	1684.9	3573.4	0.47

Tabela A.8: Mutacija SM, križanje OX

256 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	3.0	1.5	2.00
1083	45.6	78.5	0.58
2036	180.1	275.1	0.65
2566	195.7	436.2	0.44
3649	532.8	882.1	0.60
5087	681.0	1711.2	0.39
10150	3100.3	6836.5	0.45
128 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	1.5	0.9	1.66
1083	22.5	45.5	0.49
2036	92.2	159.8	0.57
2566	130.0	249.0	0.52
3649	183.1	506.2	0.36
5087	491.7	997.4	0.49
10150	1706.5	4007.3	0.42
64 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	0.7	0.4	1.75
1083	11.3	22.2	0.50
2036	32.3	77.3	0.41
2566	48.7	120.7	0.40
3649	91.6	247.0	0.37
5087	237.9	487.4	0.48
10150	634.0	1916.2	0.33

Tabela A.9: Mutacija SM, križanje PMX

256 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	3.3	2.1	1.57
1083	82.9	149.0	0.55
2036	428.0	616.2	0.69
2566	439.1	933.4	0.47
3649	1365.1	2028.8	0.67
5087	1431.6	3092.3	0.46
10150	9380.3	14701.9	0.63
128 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	1.7	1.2	1.41
1083	39.7	80.5	0.49
2036	139.4	334.3	0.41
2566	203.1	485.5	0.41
3649	442.2	1145.4	0.38
5087	1000.7	1764.2	0.56
10150	3227.9	7726.3	0.41
64 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	0.9	0.6	1.50
1083	19.2	38.9	0.49
2036	63.0	150.5	0.41
2566	92.2	227.4	0.40
3649	202.4	506.3	0.39
5087	399.0	868.1	0.45
10150	1854.7	3656.9	0.50

Tabela A.10: Mutacija SIM, križanje OX

256 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	2.9	1.8	1.61
1083	71.4	92.8	0.76
2036	231.0	398.3	0.57
2566	226.8	539.1	0.42
3649	824.5	1527.4	0.53
5087	716.9	1915.1	0.37
10150	4616.2	8694.2	0.53
128 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	1.4	0.9	1.55
1083	24.5	49.8	0.49
2036	80.3	212.7	0.37
2566	107.6	284.7	0.37
3649	266.4	761.5	0.34
5087	367.9	1068.4	0.34
10150	2108.0	4395.2	0.47
64 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	0.7	0.5	1.40
1083	11.3	27.8	0.40
2036	37.1	120.6	0.30
2566	52.6	162.0	0.32
3649	122.6	439.9	0.27
5087	181.7	577.9	0.31
10150	768.6	2824.3	0.27

Tabela A.11: Mutacija SIM, križanje PMX

256 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	3.4	2.7	1.25
1083	88.3	188.7	0.46
2036	314.2	738.5	0.42
2566	536.0	1136.6	0.47
3649	980.9	2452.4	0.39
5087	2102.2	3912.8	0.53
10150	9862.2	18002.9	0.54
128 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	1.7	1.5	1.13
1083	52.3	102.0	0.51
2036	148.3	416.0	0.35
2566	215.8	613.6	0.35
3649	689.0	1415.7	0.48
5087	1016.9	2297.3	0.44
10150	4290.4	9929.0	0.43
64 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	0.9	0.7	1.28
1083	20.0	48.9	0.40
2036	68.6	188.8	0.36
2566	100.3	291.6	0.34
3649	220.1	636.0	0.34
5087	460.0	1140.6	0.40
10150	1431.9	4609.1	0.31

Tabela A.12: Mutacija DM, križanje OX

256 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	3.0	2.3	1.30
1083	52.8	129.4	0.40
2036	193.9	605.0	0.32
2566	261.1	817.5	0.31
3649	1048.8	2498.7	0.41
5087	1201.0	2826.8	0.42
10150	5564.7	12966.7	0.42
128 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	1.5	1.3	1.15
1083	29.1	71.5	0.40
2036	97.4	333.7	0.29
2566	123.9	406.7	0.30
3649	523.8	1283.3	0.40
5087	519.2	1619.0	0.32
10150	2417.3	6503.7	0.37
64 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	0.7	0.8	0.87
1083	13.0	42.6	0.30
2036	47.2	201.3	0.23
2566	60.5	256.4	0.23
3649	169.0	729.0	0.23
5087	214.1	1030.1	0.20
10150	934.1	4305.2	0.21

Tabela A.13: Mutacija DM, križanje PMX

A.2 Število blokov na mrežo je enako 32

256 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	9.8	4.0	2.45
1083	243.6	283.7	0.85
2036	859.5	1165.9	0.73
2566	1298.3	1807.1	0.71
3649	2712.6	3893.0	0.69
5087	4181.5	6108.4	0.68
10150	18677.1	28407.4	0.65
128 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	3.4	2.0	1.70
1083	117.3	134.3	0.87
2036	409.1	531.9	0.76
2566	598.2	828.3	0.72
3649	1271.9	1805.1	0.70
5087	2042.6	2954.0	0.69
10150	8669.1	12922.3	0.67
64 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	1.7	1.1	1.54
1083	37.5	74.9	0.50
2036	163.0	285.3	0.57
2566	187.1	440.5	0.42
3649	448.5	974.8	0.46
5087	666.8	1681.6	0.39
10150	3975.3	6946.6	0.57

Tabela A.14: Mutacija ISM, križanje OX

256 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	8.1	2.9	2.79
1083	135.6	159.9	0.84
2036	469.0	783.4	0.59
2566	651.5	1026.6	0.63
3649	1859.1	3939.3	0.47
5087	2134.8	3645.5	0.58
10150	9238.6	18088.3	0.51
128 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	3.8	1.4	2.71
1083	66.9	77.0	0.86
2036	226.8	333.9	0.67
2566	317.1	482.7	0.65
3649	825.0	1517.6	0.54
5087	1088.3	1690.2	0.64
10150	4459.7	7587.6	0.58
64 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	1.4	0.8	1.75
1083	22.5	48.7	0.46
2036	74.9	210.3	0.35
2566	106.7	292.0	0.36
3649	357.1	882.2	0.40
5087	360.3	1129.3	0.31
10150	1593.8	5014.1	0.31

Tabela A.15: Mutacija ISM, križanje PMX

256 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	9.3	4.0	2.32
1083	242.0	280.0	0.86
2036	862.6	1154.8	0.74
2566	1300.0	1768.0	0.73
3649	2714.0	3921.4	0.69
5087	4204.3	6117.1	0.68
10150	18463.9	27822.9	0.66
128 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	4.2	1.9	2.21
1083	111.4	131.5	0.84
2036	404.1	525.1	0.76
2566	590.4	792.4	0.74
3649	1283.5	1818.3	0.70
5087	2022.5	2936.5	0.68
10150	8345.1	12323.9	0.67
64 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	1.7	1.1	1.54
1083	44.3	73.3	0.60
2036	123.1	276.0	0.44
2566	182.7	424.0	0.43
3649	391.8	974.9	0.40
5087	895.0	1677.9	0.53
10150	3285.9	6702.2	0.49

Tabela A.16: Mutacija EM, križanje OX

256 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	8.2	2.8	2.92
1083	128.1	151.9	0.84
2036	378.9	528.1	0.71
2566	574.4	839.0	0.68
3649	1079.4	1702.9	0.63
5087	2015.5	3358.4	0.60
10150	7568.9	13244.8	0.57
128 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	3.8	1.4	2.71
1083	63.5	72.9	0.87
2036	186.2	253.9	0.73
2566	282.7	402.8	0.70
3649	536.5	819.1	0.65
5087	1000.8	1611.5	0.62
10150	3771.7	6394.0	0.58
64 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	1.4	0.8	1.75
1083	22.4	41.4	0.54
2036	90.5	146.4	0.61
2566	95.0	233.2	0.40
3649	253.0	471.2	0.53
5087	471.8	925.1	0.50
10150	1889.1	3698.7	0.51

Tabela A.17: Mutacija EM, križanje PMX

256 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	8.9	5.3	1.67
1083	262.0	365.6	0.71
2036	945.1	1453.9	0.65
2566	1387.3	2196.2	0.63
3649	2939.2	4841.6	0.60
5087	4633.9	7765.0	0.59
10150	20219.3	34524.2	0.58
128 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	4.7	2.5	1.88
1083	123.0	167.0	0.73
2036	445.5	654.8	0.68
2566	643.4	994.6	0.64
3649	1405.7	2212.5	0.63
5087	2202.0	3638.0	0.60
10150	9107.8	15790.3	0.57
64 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	2.6	1.4	1.85
1083	57.7	94.6	0.60
2036	206.0	364.8	0.56
2566	298.8	548.5	0.54
3649	660.6	1260.3	0.52
5087	1078.2	2152.3	0.50
10150	4283.3	8664.7	0.49

Tabela A.18: Mutacija IMV, križanje OX

256 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	8.6	4.6	1.86
1083	160.8	254.4	0.63
2036	565.8	1078.9	0.52
2566	765.9	1536.2	0.49
3649	2026.9	4143.3	0.48
5087	2537.8	5407.5	0.46
10150	10612.5	23711.4	0.44
128 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	3.8	2.1	1.80
1083	78.1	114.1	0.68
2036	281.5	490.8	0.57
2566	372.0	700.2	0.53
3649	1010.1	1893.9	0.53
5087	1275.2	2509.3	0.50
10150	5115.3	10358.7	0.49
64 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	2.2	1.3	1.69
1083	37.8	71.5	0.52
2036	132.4	298.3	0.44
2566	181.1	419.4	0.43
3649	472.0	1150.3	0.41
5087	620.2	1562.8	0.39
10150	2458.7	6756.7	0.36

Tabela A.19: Mutacija IMV, križanje PMX

256 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	9.8	4.0	2.45
1083	242.8	279.7	0.86
2036	869.3	1160.8	0.74
2566	1291.3	1738.0	0.74
3649	2708.4	3858.9	0.70
5087	4187.1	6019.2	0.69
10150	17799.2	27150.1	0.65
128 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	3.5	1.9	1.84
1083	116.9	131.2	0.89
2036	391.3	513.0	0.76
2566	574.5	766.9	0.74
3649	1262.5	1772.8	0.71
5087	2024.7	2915.3	0.69
10150	8000.3	11824.9	0.67
64 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	2.2	1.1	2.00
1083	56.4	74.0	0.76
2036	172.1	269.1	0.63
2566	267.7	421.1	0.63
3649	539.7	897.3	0.60
5087	995.1	1713.2	0.58
10150	3791.8	6715.3	0.56

Tabela A.20: Mutacija SM, križanje OX

256 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	6.1	2.8	2.17
1083	134.8	152.0	0.88
2036	385.0	537.7	0.71
2566	584.9	849.9	0.68
3649	1093.5	1705.5	0.64
5087	2033.8	3322.5	0.61
10150	7589.6	13259.1	0.57
128 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	2.9	1.4	2.07
1083	66.2	74.1	0.89
2036	193.5	257.5	0.75
2566	288.6	406.6	0.70
3649	544.1	829.9	0.65
5087	1015.6	1623.7	0.62
10150	3781.4	6434.7	0.58
64 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	2.2	0.8	2.75
1083	32.6	41.9	0.77
2036	96.5	148.5	0.64
2566	144.4	235.1	0.61
3649	271.6	479.7	0.56
5087	504.7	943.9	0.53
10150	1905.9	3674.0	0.51

Tabela A.21: Mutacija SM, križanje PMX

256 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	7.9	4.1	1.92
1083	247.6	289.3	0.85
2036	878.3	1179.4	0.74
2566	1315.8	1805.4	0.72
3649	2754.2	3977.4	0.69
5087	4246.9	6164.8	0.68
10150	18746.0	27997.6	0.66
128 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	3.8	1.9	2.00
1083	117.6	134.9	0.87
2036	417.8	546.7	0.76
2566	599.2	810.1	0.73
3649	1310.4	1861.6	0.70
5087	2025.3	2887.4	0.70
10150	8326.9	12627.0	0.65
64 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	2.0	1.1	1.81
1083	55.5	75.1	0.73
2036	190.5	290.8	0.65
2566	273.8	438.0	0.62
3649	611.2	1017.0	0.60
5087	998.0	1684.3	0.59
10150	3875.7	6705.9	0.57

Tabela A.22: Mutacija SIM, križanje OX

256 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	8.4	3.3	2.54
1083	141.8	173.4	0.81
2036	480.5	762.0	0.63
2566	664.9	1039.2	0.63
3649	1626.4	2885.9	0.56
5087	2158.0	3552.8	0.60
10150	8893.8	16150.5	0.55
128 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	2.9	1.6	1.81
1083	69.6	79.6	0.87
2036	235.9	333.4	0.70
2566	322.7	466.6	0.69
3649	789.6	1248.4	0.63
5087	1064.0	1666.2	0.63
10150	4284.7	7133.1	0.60
64 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	1.7	0.9	1.88
1083	33.7	46.1	0.73
2036	109.8	194.9	0.56
2566	158.4	277.1	0.57
3649	371.1	763.3	0.48
5087	549.0	1058.0	0.51
10150	2083.7	4418.3	0.47

Tabela A.23: Mutacija SIM, križanje PMX

256 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	8.8	5.2	1.69
1083	260.6	361.9	0.72
2036	939.4	1434.1	0.65
2566	1396.9	2243.7	0.62
3649	2930.4	4837.1	0.60
5087	4537.6	7574.6	0.59
10150	20166.0	34698.6	0.58
128 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	4.9	2.5	1.96
1083	125.4	168.1	0.74
2036	445.8	661.8	0.67
2566	651.3	998.9	0.65
3649	1408.2	2227.1	0.63
5087	2180.4	3669.6	0.59
10150	9128.7	15523.4	0.58
64 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	2.3	1.4	1.64
1083	59.8	95.6	0.62
2036	203.4	358.3	0.56
2566	297.4	548.0	0.54
3649	659.2	1240.4	0.53
5087	1085.7	2131.8	0.50
10150	4212.3	8519.8	0.49

Tabela A.24: Mutacija DM, križanje OX

256 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	8.3	4.4	1.88
1083	155.8	250.6	0.62
2036	578.8	1150.4	0.50
2566	778.5	1594.3	0.48
3649	2107.0	4643.2	0.45
5087	2493.1	5482.8	0.45
10150	10750.7	24605.1	0.43
128 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	3.5	2.0	1.75
1083	74.4	116.8	0.63
2036	288.8	503.2	0.57
2566	371.6	690.6	0.53
3649	1076.6	2081.7	0.51
5087	1261.3	2556.6	0.49
10150	5352.9	10871.7	0.49
64 niti na blok			
Št. mest	t_{cpu}	t_{gpu}	ψ
131	1.5	1.3	1.15
1083	38.1	72.5	0.52
2036	136.9	326.1	0.41
2566	184.8	428.3	0.43
3649	498.5	1314.8	0.37
5087	647.9	1632.8	0.39
10150	2538.1	6911.9	0.36

Tabela A.25: Mutacija DM, križanje PMX

A.3 Interpretacija rezultatov

Ugotovitve:

- Zaradi relativno dolgih časov izvajanja, meritve pohitritev niso bile povprečene na več izvedb algoritma. Ker naključna števila pogojujejo količino dela, ki se izvrši v genetskih operatorjih, lahko prihaja do časovnih odstopanj.
- S povečevanjem števila blokov na mrežo se pohitritve povečujejo.
- Ob uporabi 16 blokov največje pohitritve dosega delitev 256 niti na blok. Pri 32 blokih je uspešnejša uporaba 128 niti na blok.
- S povečevanjem števila blokov in števila niti na blok se povečuje tudi čas izvajanja.

Dodatek B

Meritve migracijskega vpliva

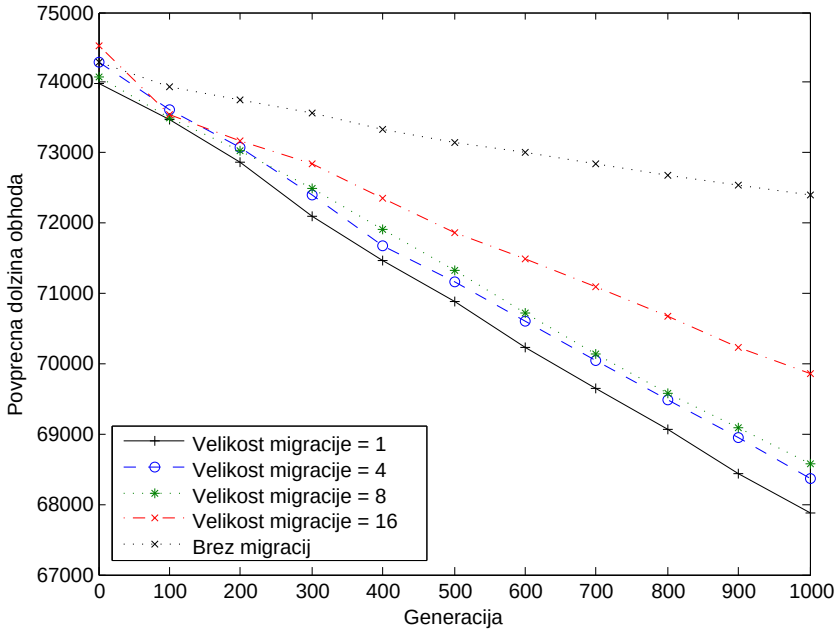
Vsi predstavljeni časi so v sekundah. Algoritem se je izvajal samo na CPE, in sicer pri:

- različnih velikostih migracij,
- pri omogočenih ali onemogočenih migracijah.

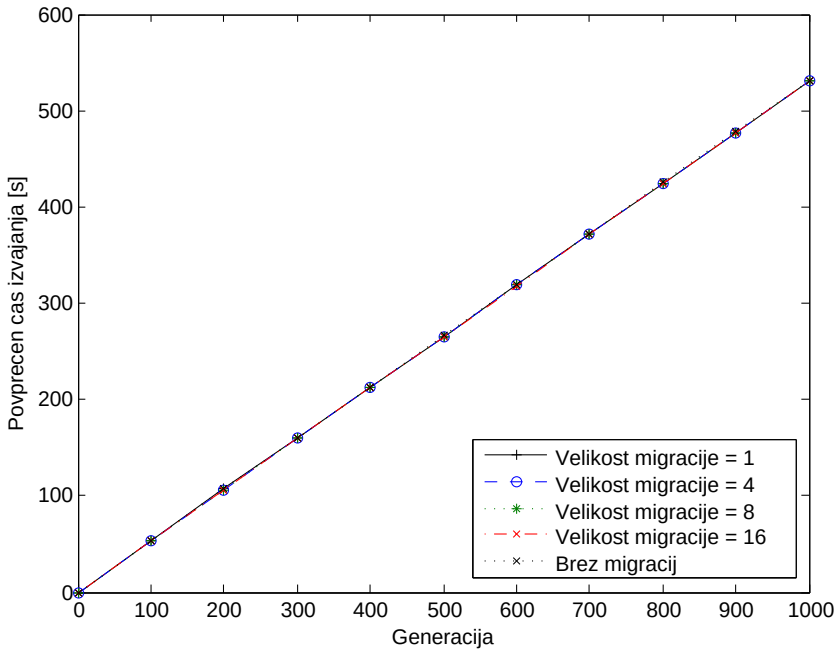
Meritve so povprečene na 100 izvedb algoritma. Nastavitve algoritma so prikazane v tabeli B.1. Cilj je ugotoviti, kako velikost migracij vpliva na uspešnost razvitih rešitev.

NASTAVITVE	
Izvajanje evolucije	
Velikost problema	2566 mest
Meritve, povprečene na št. izvedb	100
Št. klicev ščepca kernelEvolve	10
Št. generacij na klic ščepca kernelEvolve	100
Selekcija	
Oblika selekcije	Turnirska selekcija
Velikost turnirja	16
Verjetnost izbire najboljšega	0.8
Križanja	
Izbrano križanje	PMX
Verjetnost križanja	0.8
Mutacije	
Izbrana mutacija	SM
Verjetnost mutacije	0.2
Migracije	
Št. izvedenih migracijskih sklopov	10
Naključna števila	
Generator naključnih števil	Inicializiran na trenuten čas
Centralna procesna enota	
Model	Intel Core2 CPU 6400
Št. uporabljenih niti	1

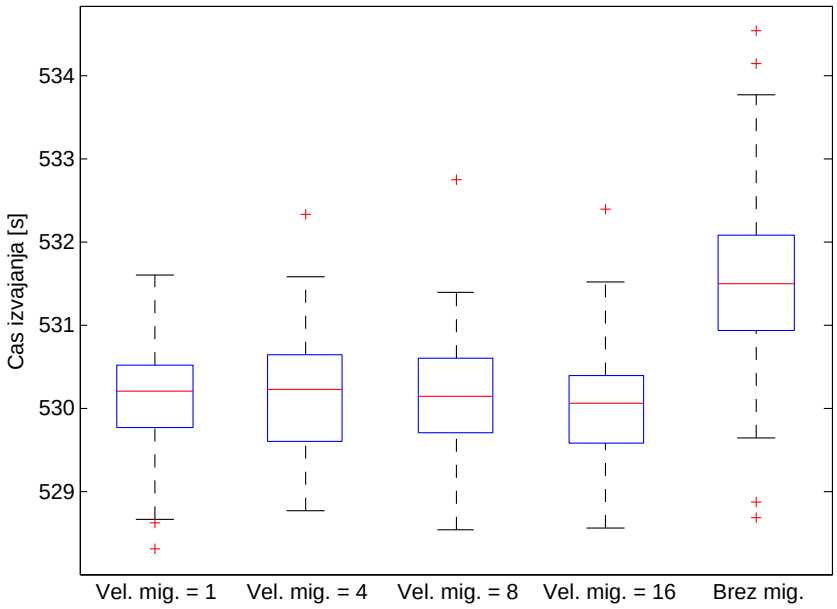
Tabela B.1: Skupne nastavitve pri merjenju migracijskega vpliva.



Slika B.1: Vpliv velikosti migracij na povprečno dolžino obhoda.



Slika B.2: Vpliv velikosti migracij na povprečen čas izvajanja.



Slika B.3: Vpliv velikosti migracij na porazdelitev časa izvajanja.

B.1 Interpretacija rezultatov

Ugotovitve:

- Izvajanje migracij na CPE porabi zanemarljivo majhen čas.
- Manjša kot je velikost emigracij, manjša je povprečna dolžina obhoda rešitev in s tem večja povprečna uspešnost.
- Z uporabo migracij v povprečju pridemo do boljših rešitev, v manj generacijah, kot če migracij ne bi uporabljali.

Dodatek C

Iskanje najuspešnejšega para operatorjev

Algoritem se je izvajal samo na CPE, in sicer pri:

- različnih parih operatorjev križanja in mutiranja

Meritve so povprečene na 100 izvedb algoritma. Nastavitve algoritma so prikazane v tabeli C.1. Cilj je ugotoviti, kateri par operatorjev križanja in mutacije v povprečju pripelje do najboljših rešitev.

NASTAVITVE	
Izvajanje evolucije	
Velikost problema	1083 mest
Meritve, povprečene na št. izvedb	100
Št. klicev ščepca kernelEvolve	10
Št. generacij na klic ščepca kernelEvolve	1000
Selekcija	
Oblika selekcije	Turnirska selekcija
Velikost turnirja	16
Verjetnost izbire najboljšega	0.8
Križanja	
Verjetnost križanja	0.8
Mutacije	
Verjetnost mutacije	0.2
Migracije	
Omogočene	DA
Št. izvedenih migracijskih sklopov	10
Velikost migracij	4
Naključna števila	
Generator naključnih števil	Inicializiran na trenuten čas
Centralna procesna enota	
Model	Intel Core2 CPU 6400
Št. uporabljenih niti	1

Tabela C.1: Skupne nastavitve pri iskanju najuspešnejšega para operatorjev.

LEGENDA SLIKE					
Št.	Mutacija	Križanje	Št.	Mutacija	Križanje
1	ISM	OX	7	SM	OX
2	ISM	PMX	8	SM	PMX
3	EM	OX	9	SIM	OX
4	EM	PMX	10	SIM	PMX
5	IMV	OX	11	DM	OX
6	IMV	PMX	12	DM	PMX

Tabela C.2: Legenda parov genetskih operatorjev.

Dodatek D

Meritve časov vzporednega algoritma na CPE

Vsi časi so podani v sekundah. Algoritem se je izvajal samo na CPE, in sicer pri:

- različnih parih operatorjev križanja in mutiranja,
- različnih velikostih problema (število mest),
- različnih delitvah problema (število niti pri CPE).

Spodnje tabele podajajo čas, potreben za razvoj rešitev, pri nastavitvah algoritma, ki jih prikazuje tabela D.1. Podane so tudi pohitritve ψ.

NASTAVITVE	
Izvajanje evolucije	
Št. klicev ščepca kernelEvolve	10
Št. generacij na klic ščepca kernelEvolve	10
Selekcija	
Oblika selekcije	Turnirska selekcija
Velikost turnirja	16
Verjetnost izbire najboljšega	0.8
Križanja	
Verjetnost križanja	0.8
Mutacije	
Verjetnost mutacije	0.2
Migracije	
Omogočene	DA
Št. izvedenih migracijskih sklopov	10
Velikost migracij	4
Naključna števila	
Generator naključnih števil	Inicializiran na trenuten čas
Centralna procesna enota	
Model	Intel Core2 Quad CPU Q6600

Tabela D.1: Skupne nastavitve pri merjenju časov CPE.

D.1 Število blokov na mrežo je enako 16

256 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	3.3	0.8	4.12
1083	118.8	20.9	5.68
2036	426.6	73.6	5.79
2566	429.7	110.2	3.89
3649	908.8	227.8	3.98
5087	1376.7	359.1	3.83
10150	6249.0	1576.3	3.96
128 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	1.7	0.4	4.25
1083	39.1	9.9	3.94
2036	137.0	34.6	3.95
2566	270.4	51.1	5.29
3649	490.1	108.5	4.51
5087	676.3	170.9	3.95
10150	3918.0	734.2	5.33
64 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	0.8	0.2	4.00
1083	18.8	4.8	3.91
2036	78.7	16.2	4.85
2566	130.4	23.9	5.45
3649	199.2	50.1	3.97
5087	330.0	83.7	3.94
10150	1326.6	337.0	3.93

Tabela D.2: Mutacija ISM, križanje OX

256 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	2.8	0.7	4.00
1083	48.5	11.7	4.14
2036	226.0	40.5	5.58
2566	283.3	55.9	5.06
3649	845.7	155.2	5.44
5087	723.9	183.2	3.95
10150	3104.3	808.9	3.83
128 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	1.4	0.3	4.66
1083	22.6	5.8	3.89
2036	77.2	20.0	3.86
2566	110.0	27.4	4.01
3649	280.0	70.5	3.97
5087	354.9	91.5	3.87
10150	1593.5	386.8	4.11
64 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	0.7	0.2	3.50
1083	11.2	2.9	3.86
2036	49.6	9.6	5.16
2566	53.8	13.7	3.92
3649	127.6	32.9	3.87
5087	218.8	44.3	4.93
10150	1059.7	181.6	5.83

Tabela D.3: Mutacija ISM, križanje PMX

256 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	3.3	1.0	3.30
1083	81.9	20.5	3.99
2036	289.0	73.1	3.95
2566	494.6	109.6	4.51
3649	907.8	227.8	3.98
5087	1885.2	352.0	5.35
10150	9221.4	1553.5	5.93
128 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	1.7	0.4	4.25
1083	46.0	10.0	4.60
2036	135.7	34.4	3.94
2566	198.9	50.2	3.96
3649	456.5	108.2	4.21
5087	696.2	170.1	4.09
10150	2706.5	710.2	3.81
64 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	0.8	0.2	4.00
1083	18.6	4.7	3.95
2036	74.6	15.5	4.81
2566	91.1	23.1	3.94
3649	196.9	49.1	4.01
5087	422.1	83.2	5.07
10150	1902.5	327.8	5.80

Tabela D.4: Mutacija EM, križanje OX

256 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	2.8	0.7	4.00
1083	50.8	11.2	4.53
2036	128.1	32.3	3.96
2566	272.9	48.5	5.62
3649	530.8	91.4	5.80
5087	1005.7	169.8	5.92
10150	3768.1	637.3	5.91
128 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	1.4	0.3	4.66
1083	21.8	5.6	3.89
2036	63.5	16.2	3.91
2566	95.6	24.2	3.95
3649	180.0	45.3	3.97
5087	392.8	84.5	4.64
10150	1260.3	317.7	3.96
64 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	0.7	0.2	3.50
1083	10.8	2.8	3.85
2036	40.6	8.2	4.95
2566	47.8	12.1	3.95
3649	90.5	22.8	3.96
5087	170.1	42.2	4.03
10150	631.1	160.9	3.92

Tabela D.5: Mutacija EM, križanje PMX

256 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	3.4	0.9	3.77
1083	115.4	22.0	5.24
2036	316.2	79.6	3.97
2566	465.7	117.3	3.97
3649	981.9	246.9	3.97
5087	2279.3	380.7	5.98
10150	9794.6	1706.9	5.73
128 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	1.7	0.5	3.40
1083	41.8	10.7	3.90
2036	161.7	37.7	4.28
2566	215.5	54.1	3.98
3649	697.0	118.8	5.86
5087	956.1	190.1	5.02
10150	4159.1	761.3	5.46
64 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	0.9	0.2	4.50
1083	20.1	5.2	3.86
2036	103.3	17.5	5.90
2566	99.4	25.4	3.91
3649	222.2	55.8	3.98
5087	358.0	90.5	3.95
10150	1971.0	357.9	5.50

Tabela D.6: Mutacija IMV, križanje OX

256 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	3.0	0.8	3.75
1083	65.0	13.7	4.74
2036	240.2	48.5	4.95
2566	324.7	65.5	4.95
3649	676.7	172.7	3.91
5087	1253.4	213.0	5.88
10150	4454.6	889.9	5.00
128 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	2.2	0.4	5.50
1083	30.0	6.8	4.41
2036	92.9	24.3	3.82
2566	168.4	31.8	5.29
3649	342.1	85.6	3.99
5087	613.1	106.1	5.77
10150	2261.6	445.5	5.07
64 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	0.7	0.2	3.50
1083	13.0	3.4	3.82
2036	46.0	11.6	3.96
2566	61.6	15.4	4.00
3649	159.9	40.1	3.98
5087	211.7	53.5	3.95
10150	1014.1	212.8	4.76

Tabela D.7: Mutacija IMV, križanje PMX

256 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	3.4	0.9	3.77
1083	81.4	21.0	3.87
2036	292.7	72.5	4.03
2566	432.9	108.7	3.98
3649	1009.1	227.0	4.44
5087	1383.8	351.2	3.94
10150	6187.5	1535.3	4.03
128 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	1.8	0.5	3.60
1083	39.2	10.1	3.88
2036	132.6	32.8	4.04
2566	189.0	48.8	3.87
3649	510.9	107.4	4.75
5087	740.0	173.6	4.26
10150	3917.1	668.8	5.85
64 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	0.9	0.2	4.50
1083	18.8	4.8	3.91
2036	82.0	15.0	5.46
2566	89.3	22.7	3.93
3649	256.1	45.5	5.62
5087	352.6	83.6	4.21
10150	1684.9	320.4	5.25

Tabela D.8: Mutacija SM, križanje OX

256 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	3.0	0.8	3.75
1083	45.6	11.6	3.93
2036	180.1	33.1	5.44
2566	195.7	49.2	3.97
3649	532.8	92.5	5.76
5087	681.0	171.6	3.96
10150	3100.3	641.1	4.83
128 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	1.5	0.4	3.75
1083	22.5	5.7	3.94
2036	92.2	16.5	5.58
2566	130.0	24.6	5.28
3649	183.1	46.1	3.97
5087	491.7	85.9	5.72
10150	1706.5	322.0	5.29
64 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	0.7	0.2	3.50
1083	11.3	2.8	4.03
2036	32.3	8.4	3.84
2566	48.7	12.5	3.89
3649	91.6	23.1	3.96
5087	237.9	43.1	5.51
10150	634.0	161.2	3.93

Tabela D.9: Mutacija SM, križanje PMX

256 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	3.3	0.8	4.12
1083	82.9	20.8	3.98
2036	428.0	74.6	5.73
2566	439.1	109.4	4.01
3649	1365.1	231.5	5.89
5087	1431.6	352.5	4.06
10150	9380.3	1563.2	6.00
128 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	1.7	0.4	4.25
1083	39.7	10.2	3.89
2036	139.4	34.9	3.99
2566	203.1	50.8	3.99
3649	442.2	110.1	4.01
5087	1000.7	176.1	5.68
10150	3227.9	712.9	4.52
64 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	0.9	0.2	4.50
1083	19.2	4.8	4.00
2036	63.0	16.2	3.88
2566	92.2	23.6	3.90
3649	202.4	51.1	3.96
5087	399.0	84.5	4.72
10150	1854.7	333.2	5.56

Tabela D.10: Mutacija SIM, križanje OX

256 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	2.9	0.7	4.14
1083	71.4	12.4	5.75
2036	231.0	41.9	5.51
2566	226.8	56.0	4.05
3649	824.5	141.6	5.82
5087	716.9	180.0	3.98
10150	4616.2	759.8	6.07
128 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	1.4	0.4	3.50
1083	24.5	6.0	4.08
2036	80.3	20.2	3.97
2566	107.6	27.1	3.97
3649	266.4	68.2	3.90
5087	367.9	90.5	4.06
10150	2108.0	374.6	5.62
64 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	0.7	0.2	3.50
1083	11.3	3.0	3.76
2036	37.1	9.5	3.90
2566	52.6	13.5	3.89
3649	122.6	31.2	3.92
5087	181.7	46.2	3.93
10150	768.6	172.5	4.45

Tabela D.11: Mutacija SIM, križanje PMX

256 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	3.4	0.9	3.77
1083	88.3	22.3	3.95
2036	314.2	79.0	3.97
2566	536.0	117.4	4.56
3649	980.9	246.7	3.97
5087	2102.2	387.5	5.42
10150	9862.2	1673.7	5.89
128 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	1.7	0.4	4.25
1083	52.3	10.8	4.84
2036	148.3	37.7	3.93
2566	215.8	53.6	4.02
3649	689.0	118.2	5.82
5087	1016.9	187.2	5.43
10150	4290.4	772.6	5.55
64 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	0.9	0.2	4.50
1083	20.0	5.2	3.84
2036	68.6	17.5	3.92
2566	100.3	25.4	3.94
3649	220.1	55.3	3.98
5087	460.0	91.9	5.00
10150	1431.9	357.6	4.00

Tabela D.12: Mutacija DM, križanje OX

256 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	3.0	0.8	3.75
1083	52.8	13.5	3.91
2036	193.9	49.5	3.91
2566	261.1	67.2	3.88
3649	1048.8	181.6	5.77
5087	1201.0	216.9	5.53
10150	5564.7	953.9	5.83
128 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	1.5	0.4	3.75
1083	29.1	6.7	4.34
2036	97.4	25.5	3.81
2566	123.9	32.9	3.76
3649	523.8	90.1	5.81
5087	519.2	109.6	4.73
10150	2417.3	469.6	5.14
64 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	0.7	0.2	3.50
1083	13.0	3.4	3.82
2036	47.2	12.2	3.86
2566	60.5	16.2	3.73
3649	169.0	42.4	3.98
5087	214.1	53.3	4.01
10150	934.1	205.1	4.55

Tabela D.13: Mutacija DM, križanje PMX

D.2 Število blokov na mrežo je enako 32

256 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	9.8	1.7	5.76
1083	243.6	41.3	5.89
2036	859.5	146.6	5.86
2566	1298.3	220.5	5.88
3649	2712.6	455.2	5.95
5087	4181.5	715.1	5.84
10150	18677.1	3162.1	5.90
128 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	3.4	0.9	3.77
1083	117.3	19.9	5.89
2036	409.1	69.0	5.92
2566	598.2	101.1	5.91
3649	1271.9	217.0	5.86
5087	2042.6	341.1	5.98
10150	8669.1	1444.3	6.00
64 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	1.7	0.5	3.40
1083	37.5	9.7	3.86
2036	163.0	31.6	5.15
2566	187.1	47.3	3.95
3649	448.5	99.0	4.53
5087	666.8	167.4	3.98
10150	3975.3	671.5	5.92

Tabela D.14: Mutacija ISM, križanje OX

256 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	8.1	1.4	5.78
1083	135.6	23.2	5.84
2036	469.0	79.3	5.91
2566	651.5	111.7	5.83
3649	1859.1	315.6	5.89
5087	2134.8	359.2	5.94
10150	9238.6	1555.7	5.93
128 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	3.8	0.7	5.42
1083	66.9	11.5	5.81
2036	226.8	38.7	5.86
2566	317.1	54.7	5.79
3649	825.0	140.7	5.86
5087	1088.3	180.5	6.02
10150	4459.7	758.9	5.87
64 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	1.4	0.4	3.50
1083	22.5	5.8	3.87
2036	74.9	19.3	3.88
2566	106.7	27.2	3.92
3649	357.1	64.1	5.57
5087	360.3	93.1	3.87
10150	1593.8	370.4	4.30

Tabela D.15: Mutacija ISM, križanje PMX

256 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	9.3	1.7	5.47
1083	242.0	41.1	5.88
2036	862.6	145.8	5.91
2566	1300.0	219.2	5.93
3649	2714.0	454.0	5.97
5087	4204.3	701.3	5.99
10150	18463.9	3113.4	5.93
128 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	4.2	0.8	5.25
1083	111.4	19.6	5.68
2036	404.1	67.5	5.98
2566	590.4	99.8	5.91
3649	1283.5	216.6	5.92
5087	2022.5	344.1	5.87
10150	8345.1	1398.5	5.96
64 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	1.7	0.4	4.25
1083	44.3	9.5	4.66
2036	123.1	31.1	3.95
2566	182.7	46.0	3.97
3649	391.8	98.7	3.96
5087	895.0	167.2	5.35
10150	3285.9	655.8	5.01

Tabela D.16: Mutacija EM, križanje OX

256 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	8.2	1.4	5.85
1083	128.1	22.4	5.71
2036	378.9	64.4	5.88
2566	574.4	96.7	5.94
3649	1079.4	181.9	5.93
5087	2015.5	338.7	5.95
10150	7568.9	1274.6	5.93
128 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	3.8	0.7	5.42
1083	63.5	11.1	5.72
2036	186.2	32.1	5.80
2566	282.7	48.0	5.88
3649	536.5	90.7	5.91
5087	1000.8	169.2	5.91
10150	3771.7	634.8	5.94
64 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	1.4	0.4	3.50
1083	22.4	5.6	4.00
2036	90.5	16.2	5.58
2566	95.0	24.2	3.92
3649	253.0	45.5	5.56
5087	471.8	84.8	5.56
10150	1889.1	318.7	5.92

Tabela D.17: Mutacija EM, križanje PMX

256 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	8.9	1.7	5.23
1083	262.0	44.2	5.92
2036	945.1	158.5	5.96
2566	1387.3	235.0	5.90
3649	2939.2	493.8	5.95
5087	4633.9	762.2	6.07
10150	20219.3	3350.2	6.03
128 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	4.7	0.9	5.22
1083	123.0	21.5	5.72
2036	445.5	75.2	5.92
2566	643.4	109.1	5.89
3649	1405.7	238.0	5.90
5087	2202.0	378.6	5.81
10150	9107.8	1557.3	5.84
64 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	2.6	0.4	6.50
1083	57.7	10.2	5.65
2036	206.0	34.9	5.90
2566	298.8	51.0	5.85
3649	660.6	111.1	5.94
5087	1078.2	180.2	5.98
10150	4283.3	722.8	5.92

Tabela D.18: Mutacija IMV, križanje OX

256 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	8.6	1.5	5.73
1083	160.8	27.2	5.91
2036	565.8	97.9	5.77
2566	765.9	127.5	6.00
3649	2026.9	338.1	5.99
5087	2537.8	431.6	5.87
10150	10612.5	1847.7	5.74
128 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	3.8	0.8	4.75
1083	78.1	13.4	5.82
2036	281.5	47.8	5.88
2566	372.0	63.8	5.83
3649	1010.1	168.6	5.99
5087	1275.2	210.5	6.05
10150	5115.3	888.3	5.75
64 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	2.2	0.4	5.50
1083	37.8	6.8	5.55
2036	132.4	23.0	5.75
2566	181.1	30.9	5.86
3649	472.0	80.2	5.88
5087	620.2	104.6	5.92
10150	2458.7	416.9	5.89

Tabela D.19: Mutacija IMV, križanje PMX

256 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	9.8	1.7	5.76
1083	242.8	41.3	5.87
2036	869.3	146.7	5.92
2566	1291.3	216.8	5.95
3649	2708.4	455.9	5.94
5087	4187.1	700.1	5.98
10150	17799.2	3017.1	5.89
128 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	3.5	0.9	3.88
1083	116.9	19.8	5.90
2036	391.3	68.0	5.75
2566	574.5	96.1	5.97
3649	1262.5	211.7	5.96
5087	2024.7	341.4	5.93
10150	8000.3	1346.1	5.94
64 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	2.2	0.5	4.40
1083	56.4	9.6	5.87
2036	172.1	30.0	5.73
2566	267.7	45.4	5.89
3649	539.7	91.0	5.93
5087	995.1	168.0	5.92
10150	3791.8	634.7	5.97

Tabela D.20: Mutacija SM, križanje OX

256 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	6.1	1.5	4.06
1083	134.8	23.1	5.83
2036	385.0	65.9	5.84
2566	584.9	98.2	5.95
3649	1093.5	184.5	5.92
5087	2033.8	343.3	5.92
10150	7589.6	1284.2	5.90
128 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	2.9	0.7	4.14
1083	66.2	11.4	5.80
2036	193.5	32.8	5.89
2566	288.6	49.1	5.87
3649	544.1	92.1	5.90
5087	1015.6	171.1	5.93
10150	3781.4	640.2	5.90
64 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	2.2	0.4	5.50
1083	32.6	5.7	5.71
2036	96.5	16.6	5.81
2566	144.4	24.6	5.86
3649	271.6	46.2	5.87
5087	504.7	85.5	5.90
10150	1905.9	320.7	5.94

Tabela D.21: Mutacija SM, križanje PMX

256 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	7.9	1.7	4.64
1083	247.6	41.6	5.95
2036	878.3	149.1	5.89
2566	1315.8	220.2	5.97
3649	2754.2	460.8	5.97
5087	4246.9	697.3	6.09
10150	18746.0	3162.3	5.92
128 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	3.8	0.9	4.22
1083	117.6	20.1	5.85
2036	417.8	70.6	5.91
2566	599.2	100.2	5.98
3649	1310.4	220.9	5.93
5087	2025.3	347.5	5.82
10150	8326.9	1404.5	5.92
64 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	2.0	0.4	5.00
1083	55.5	9.8	5.66
2036	190.5	32.2	5.91
2566	273.8	46.7	5.86
3649	611.2	102.5	5.96
5087	998.0	169.7	5.88
10150	3875.7	667.4	5.80

Tabela D.22: Mutacija SIM, križanje OX

256 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	8.4	1.5	5.60
1083	141.8	24.6	5.76
2036	480.5	84.0	5.72
2566	664.9	113.1	5.87
3649	1626.4	277.6	5.85
5087	2158.0	365.6	5.90
10150	8893.8	1512.0	5.88
128 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	2.9	0.7	4.14
1083	69.6	11.9	5.84
2036	235.9	40.3	5.85
2566	322.7	55.6	5.80
3649	789.6	136.4	5.78
5087	1064.0	179.8	5.91
10150	4284.7	736.2	5.82
64 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	1.7	0.4	4.25
1083	33.7	5.9	5.71
2036	109.8	19.1	5.74
2566	158.4	26.2	6.04
3649	371.1	61.3	6.05
5087	549.0	90.6	6.05
10150	2083.7	357.7	5.82

Tabela D.23: Mutacija SIM, križanje PMX

256 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	8.8	1.7	5.17
1083	260.6	44.1	5.90
2036	939.4	158.2	5.93
2566	1396.9	232.4	6.01
3649	2930.4	491.9	5.95
5087	4537.6	762.7	5.94
10150	20166.0	3358.5	6.00
128 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	4.9	0.9	5.44
1083	125.4	21.4	5.85
2036	445.8	75.0	5.94
2566	651.3	108.2	6.01
3649	1408.2	237.4	5.93
5087	2180.4	378.0	5.76
10150	9128.7	1538.1	5.93
64 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	2.3	0.4	5.75
1083	59.8	10.2	5.86
2036	203.4	34.7	5.86
2566	297.4	50.4	5.90
3649	659.2	110.6	5.96
5087	1085.7	184.5	5.88
10150	4212.3	710.6	5.92

Tabela D.24: Mutacija DM, križanje OX

256 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	8.3	1.5	5.53
1083	155.8	26.7	5.83
2036	578.8	99.3	5.82
2566	778.5	129.2	6.02
3649	2107.0	356.9	5.90
5087	2493.1	430.1	5.79
10150	10750.7	1858.6	5.78
128 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	3.5	0.8	4.37
1083	74.4	13.2	5.63
2036	288.8	48.8	5.91
2566	371.6	64.6	5.75
3649	1076.6	177.7	6.05
5087	1261.3	213.4	5.91
10150	5352.9	878.8	6.09
64 niti na blok			
Št. mest	$t_{cpu\ 1\ nit}$	$t_{cpu\ 4\ niti}$	ψ
131	1.5	0.4	3.75
1083	38.1	6.7	5.68
2036	136.9	24.0	5.70
2566	184.8	32.0	5.77
3649	498.5	84.7	5.88
5087	647.9	106.5	6.08
10150	2538.1	432.0	5.87

Tabela D.25: Mutacija DM, križanje PMX

D.3 Interpretacija rezultatov

Ugotovitve:

- Meritve pohitritev niso bile povprečene na več izvedb algoritma. Ker naključna števila pogojujejo količino dela, ki se izvrši v genetskih operatorjih, lahko prihaja do časovnih odstopanj.
- S povečevanjem števila blokov na mrežo se pohitritve povečujejo.
- Vzporedni algoritem na CPE uspešno dosega pohitritve.
- S povečevanjem števila blokov in števila niti na blok se povečuje tudi čas izvajanja.

Slike

Tabele

Algoritmi

Kodni izseki

Literatura

[1] Uroš Breskvar in Blaž Rodič. *Optimiranje operatorjev genetskih algoritmov*. Založba Vega, Ljubljana, 2011.

[2] Su Chen, Spencer Davis, Hai Jiang in Andy Novobilski. CUDA-Based Genetic Algorithm on Travelling Salesman Problem. *Studies in Computational Intelligence*, Issue 364:241–252, Februar 2011. Dostopno na `http://dx.doi.org/10.1007/978-3-642-21378-6_19`.

[3] NVIDIA Corporation. NVIDIA CUDA C Programming Guide, version 5.5. Dostopno na `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.

[4] NVIDIA Corporation. NVIDIA CUDA Getting Started Guide for Linux, version 5.5. Dostopno na `http://docs.nvidia.com/cuda/pdf/CUDA_Getting_Started_Linux.pdf`.

[5] Andrej Dobnikar in Branko Šter. *Mehko računanje za modeliranje, razpoznavanje in regresijo*. Fakulteta za računalništvo in informatiko, Ljubljana, 2008.

[6] Marjan Mernik, Matej Črepinšek in Viljem Žumer. *Evolucijski algoritmi*. Fakulteta za elektrotehniko, računalništvo in informatiko, Inštitut za računalništvo, Maribor, 2003.

[7] Petr Pospichal, Jiri Jaros in Josef Schwarz. Parallel Genetic Algorithm on the CUDA Architecture. *Applications of Evolutionary Computation*, Issue 6024:442–451, Januar 2010. Dostopno na `http://dx.doi.org/10.1007/978-3-642-12239-2_46`.

[8] Borut Robič. *Aproksimacijski algoritmi*. Fakulteta za računalništvo in informatiko, Ljubljana, 2008.

[9] Andre Rohe. VLSI Data Sets. Dostopno na `http://www.math.uwaterloo.ca/tsp/vlsi/index.html`.

[10] Jason Sanders in Edward Kandrot. *CUDA by example : an introduction to general-purpose GPU programming.* Pearson Education, Inc., Boylston Street, Boston, USA, 2010.