UNIVERZA V MARIBORU

FAKULTETA ZA ELEKTROTEHNIKO,

RAČUNALNIŠTVO IN INFORMATIKO

Miha Vuk

ALGORITEM ZA ISKANJE BINARNIH SEKVENC Z NIZKO AVTOKORELACIJO

Diplomsko delo

Maribor, september 2021

UNIVERZA V MARIBORU

FAKULTETA ZA ELEKTROTEHNIKO,

RAČUNALNIŠTVO IN INFORMATIKO

Miha Vuk

ALGORITEM ZA ISKANJE BINARNIH SEKVENC Z NIZKO AVTOKORELACIJO

Diplomsko delo

Maribor, september 2021

ALGORITEM ZA ISKANJE BINARNIH SEKVENC Z NIZKO AVTOKORELACIJO

Diplomsko delo

Študent:	Miha Vuk
Študijski program:	visokošolski strokovni program
Smer:	Računalništvo in informacijske tehnologije
Mentor:	prof. dr. Janez Brest
Somentor:	doc. dr. Borko Bošković
Lektorica:	dr. Aleksandra Gačić, univ. dipl. prof. zgo. in slov.

Zahvala

Zahvaljujem se prof. dr. Janezu Brestu za mentorstvo in možnost raziskovanja zanimivega področja. Rad bi se zahvalil tudi za izjemen odnos in predanost do dela ter sodelovanja s študenti. Dobre volje ni manjkalo.

Zahvaljujem se somentorju doc. dr. Borku Boškoviću ter vsem profesorjem in asistentom, ki so mi pomagali med študijem. Zahvaljujem se za sodelovanje in prenašanje znanja ter čas, ki smo ga skupaj preživeli.

Zahvaljujem se tudi celotni družini za vso podporo vsa leta študija. Hvala posebej očetu Borisu in materi Diani za vložen čas in voljo.
Hvala tudi za vso radost in ljubezen.

Algoritem za iskanje binarnih sekvenc z nizko avtokorelacijo

Ključne besede: algoritem, binarna sekvenca, avtokorelacija

UDK: 004:421(043.2)

Povzetek

V diplomskem delu predstavljamo algoritem za iskanje binarnih sekvenc z nizko avtokorelacijo. S tem algoritmom najdemo binarne sekvence nizkih avtokorelacijskih nivojev. Omenjeni algoritem se ponuja z učinkovitim delovanjem in enostavno implementacijo. Skozi diplomsko delo opišemo idejo, kakšna je strategija pristopa in zakaj, predpogoje ter delovanje. Predstavimo lastna opažanja in ugotovitve, do katerih smo prišli med razvojem in raziskovanjem ter prikažemo dobljene rezultate.

Algorithm for finding low autocorrelation binary sequences

Keywords: algorithm, binary sequence, autocorrelation

UDC: 004:421(043.2)

Abstract

In this thesis, we present an algorithm for finding low autocorrelation binary sequences. This algorithm is able to find sequences with low autocorrelation levels. The mentioned algorithm promises efficiency and simple implementation. Through this thesis, we present the idea behind the design, the approach strategy and why it is used, the prerequisites, and how the algorithm functions. We present newly acquired knowledge and findings reached through the development and research with the results.

KAZALO VSEBINE

KAZALO SLIK

KAZALO TABEL

SEZNAM UPORABLJENIH KRATIC IN SIMBOLOV

PSL – (angl. Peak sidelobe level); Maksimalni nivo stranskega režnja

AACF – (angl. aperiodic autocorrelation function); Aperiodična avtokorelacijska funkcija

MF – (angl. merit factor); Faktor ugodnosti

SHC – (angl. Shotgun Hill Climbing); Vzpenjanje na hrib z naključno ponastavitvijo

SSD – (angl. solid state drive);

IDE – (angl. Integrated development environment); Integrirano razvojno okolje

API – (angl. application programming interface); Vmesnik za namensko programiranje

VS – (angl. Visual Studio); Microsoftov Visual Studio (programska oprema)

OS – (angl. Operating system); Operacijski sistem

Int – (angl. Integer); Celo število

1. UVOD

Potreba po novih odkritjih na področju računalništva, informatike in komunikologije se pojavi zaradi ideje o povečanju natančnosti informacij in manjšanju odstopanj ali izgub. Tako so bili cilji višanje učinkovitosti in izboljšanje tehnologije, a seveda so se posledično izboljšali tudi komunikacijski kanali. Pri pogostosti uporabe le-teh je procesiranje signalov in podatkov že od nekdaj pomembno ter je zato zanimivo področje za iskanje novosti ali izvajanje raziskav. Tako je področje iskanja sekvenc z nizko avtokorelacijo pomembno za evolucijo računalniških, komunikacijskih in informacijskih tehnologij. Zaradi potrebe po uporabi na področjih procesiranja signala in informacij, ter tudi sinhronizacije, se je ustvarila potreba po iskanju takih sekvenc z novimi rešitvami, ki so boljše reševale zadane probleme kot do tedaj poznane. Znan primer rabe takšnih sekvenc so kompresije na pulzno-kodni osnovi za radarje in sonarje. Od odkritja smo skozi razvoj našli vedno boljše (tj. nam uporabnejše) sekvence, najdene na osnovi na novo predlaganih in predstavljenih načinov iskanja. Skozi razvoj strojne in programske opreme se je zaradi doseženega napredka ponudilo vedno več možnih pristopov in strategij. To je pripomoglo k zmanjšanju omejitev in tako omogočilo še hitrejši napredek ter natančnejši izračun rezultatov. Izboljšave v hitrostih procesiranja podatkov in povečanju kapacitet pomnilnikov so na področjih računalniških, komunikacijskih in informacijskih tehnologij prinesle nove možnosti za uporabo virov in razpolaganje z njimi. Ker je bilo na voljo vedno več procesorske moči in ker lahko hranimo vedno več podatkov, so danes računalniki zmožni izvajanja kompleksnih, časovno in prostorsko zahtevnih operacij, ki jih starejši niso zmogli zaradi strojnih in programskih omejitev. Takšne spremembe so zato vedno znova prinašale novosti in do takrat nepoznane rešitve, ki so bile rezultat napredka v razvoju. Za generacijo in iskanje omenjenih sekvenc je skozi zgodovino bilo predlaganih veliko rešitev, nekatere od teh so do danes bile deležne predelav in izboljšav, s čimer smo aplikacije manj učinkovitih, zastarelih ali več neuporabnih rešitev zamenjali z novejšimi, ki prinašajo izboljšave. Sekvence z nizko avtokorelacijo so od odkritja njihove uporabne vrednosti atraktivno področje. Posebej zanimive so binarne sekvence z nizko avtokorelacijo.

V drugem poglavju podamo pregled področja in opišemo binarne sekvence. V tretjem poglavju bomo predstavili potek izdelave in kodiranja predstavljenega algoritma za iskanje binarnih sekvenc z nizkimi avtokorelacijami, ter tudi uporabljeno programsko in strojno opremo, na kateri izvajamo iskanje. V četrtem poglavju predstavimo parametre algoritma in rezultate izvedenih eksperimentov. Peto poglavje zaključuje diplomsko nalogo.

2. PREDSTAVITEV PODROČJA

2.1 Binarne sekvence z nizko avtokorelacijo

V tem diplomskem delu smo se posvetili binarnim sekvencam z nizko avtokorelacijo, ki imajo aplikacije v različnih vejah inženirstva, a predvsem nas zanimajo področja računalništva, informacij in komunikacij, kjer imajo omenjene sekvence veliko različnih aplikacij. Med najbolj poznanimi so Barkerjeve kode [2], M-sekvence [7], Legendrove sekvence [18], Goldove [6] in Kasamijeve [8] kode ter Weilove sekvence [19]. Druge neomenjene so podane v [14]. Od navedenih imajo Barkerjeve sekvence najboljše avtokorelacijske lastnosti, medtem ko so Goldove in Kasamijeve sekvence znane po idealnih periodičnih avtokorelacijskih funkcijah. Iskanje ali gradnjo izvajamo torej po točno določenih postopkih, katerih koraki so natančno izbrani in preračunani za učinkovito in natančno delovanje. Pomembni sta seveda tako časovna zahtevnost reševanja danega problema kot tudi prostorska zahtevnost. Pri krajših dolžinah sekvenc je opravilo časovno in prostorsko nezahteven postopek, saj je število vrednosti, ki jih je treba ovrednotiti, majhno. Iskanje omenjenih sekvenc je lahko pri daljših dolžinah zelo časovno zahtevna naloga, medtem ko prostorska zahtevnost z današnjimi zmožnostmi računalnikov ni problematična, razen v primeru ekstremnih dolžin. Posebnost bi lahko bila morda boljša rešitev, ki uporablja drugačen pristop iskanja in zato posledično dosti boljše rešuje dan problem kot trenutno poznana najboljša. Zanimajo nas predvsem sekvence z nizkimi ali minimalnimi energijskimi nivoji (ang. PSL - Peak sidelobe level).

2.2 Algoritem za iskanje binarnih sekvenc

V članku [4] so predstavljeni algoritem, funkcija vrednotenja, predpogoji in izbira parametrov za učinkovito iskanje binarnih sekvenc z nizkimi PSL.

V omenjeni raziskavi je predstavljen algoritem SHC (angl. Shotgun Hill Climbing), ki je različica algoritma vzpenjanja na hrib. Strategija vzpenjanja na hrib je matematična optimizacijska tehnika družine lokalnega iskanja. To je iterativni algoritem, ki začne z določeno rešitvijo in vanjo uvaja spremembe, dokler je nova rešitev boljša. Tak pristop najde optimalne rešitve za probleme s konveksno hevristiko, medtem ko za drugačne vrste problemov najde le lokalne optimume in tem ni možno uiti z iskanjem sosedstva, razen z uvedbo spremembe pri SHC. Ta algoritem uvaja spremembo, ki nudi ponastavljanje začetne točke algoritma in na ta način uidemo lokalnim optimumom. Tudi ta algoritem izvaja iterativno vzpenjanje na hrib, vedno znova začne z naključno rešitvijo in si skozi iteracije hrani najboljšo najdeno.

Primera drugih predlaganih strategij sta algoritem simuliranega ohlajanja (angl. simulated annealing) in algoritem Tabu preiskovanja (angl. Tabu search). Algoritem simuliranega ohlajanja je verjetnostna tehnika za aproksimacijo globalnega optimuma dane funkcije. To je meta-hevristika za aproksimacijo globalne optimizacije v velikem prostoru preiskovanja danega optimizacijskega problema. Njeno delovanje je določeno s spremenljivko temperature T, ki je mera tolerance za spremembe. Manjša, kot je, manjše spremembe so možne. Pogosto se uporablja za diskretne prostore preiskovanja, vendar je uporabna za iskanje približka globalnega optimuma, in ne natančnega lokalnega optimuma v danem času. Algoritem tabu preiskovanja je meta-hevristična preiskovalna metoda, ki uporablja poznane metode lokalnega iskanja za matematično optimizacijo. Deluje torej na osnovi lokalnega preiskovanja oziroma preiskovanja sosedstva. Tudi tabu preiskovanje se začne iz neke začetne rešitve, kjer preverja sosedstvo za potencialne izboljšave. Ta algoritem ima zanimiv pristop k reševanju iz lokalnega optimuma. V primeru, da ni najdenih izboljšav (angl. improving move), so dopuščeni pomiki v slabše rešitve (angl. worsening moves), kar skozi iteracije vodi k rešitvi algoritma iz lokalnega optimuma. Algoritem tabu preiskovanja uporablja spominske strukture (angl. memory structures), ki se razlikujejo po načinu uporabe. Kratkotrajne (angl. short-term) se uporabljajo za razpoznavanje že prej najdenih rešitev, strukture srednjega trajanja (angl. intermediate-term) se uporabljajo za usmerjanje preiskovanja k boljšim področjem preiskovalnega prostora in dolgoročne (ang. long-term),

ki se uporabljajo za uvedbo raznolikosti in privedejo preiskovanje v nove regije, to je tudi rešitev za izstop iz najdenega lokalnega optimuma.

V članku [4] sta tudi opisani pravilna nastavitev in raba parametrov, ki vplivajo na učinkovitost delovanja dokumentiranega algoritma. V nadaljevanju na kratko predstavimo pomembnost vhodnih parametrov algoritma in njihovih vrednosti.

Predlagani algoritem kot vhod prejme dolžino binarne sekvence n, funkcijo vrednotenja F, mejno vrednost t, spodnjo mejo $hmin$ in zgornjo mejo $hmax$ ter ciljno vrednost G. Dolžina binarne sekvence n bo uporabljena za generacijo naključne začetne rešitve. Z večanjem dolžine sekvenc, preiskovanje prostora rešitev postane za računalnik časovno zahtevna operacija.

Naj bo B binarna sekvenca dolžine n, podana z enačbo (1.1), kjer je n večji od 1 in naj bo b_i posamezen element, kjer je i večji ali enak 0 ter manjši od dolžine n (glej enačbo(1.2)).

$$B = (b_0, b_1, b_2, \dots, b_{n-1}), \text{kjer } n > 1 \tag{1.1}$$

Tako b_i zavzema vrednosti 1 ali -1, kar nam da sekvenco B, dolžine n.

$$b_i \in \{-1,1\}, \quad 0 \leq i \leq n-1 \tag{1.2}$$

Za opisane sekvence nas zanimajo nizke vrednosti PSL, ki jih izračunamo s pomočjo matematičnih funkcij. Pri izračunu PSL uporabimo aperiodično avtokorelacijsko funkcijo (angl. AACF - aperiodic autocorrelation function), ki je podana z naslednjo enačbo (1.3).

$$C_u(B) = \sum_{j=0}^{n-u-1} b_j \, b_{j+u}, \text{za } u \in \{0,1,2, \dots, n-1\} \tag{1.3}$$

Povzemimo, da je AACF v osnovi opredeljena na intervalu I in podana z enačbo (1.4) in velja enačba (1.5):

$$I = \{-n+1, -n+2, \dots, -2, -1, 0, 1, 2, \dots, n-1\} \tag{1.4}$$

$$C_u(B) = -C_u(B) \tag{1.5}$$

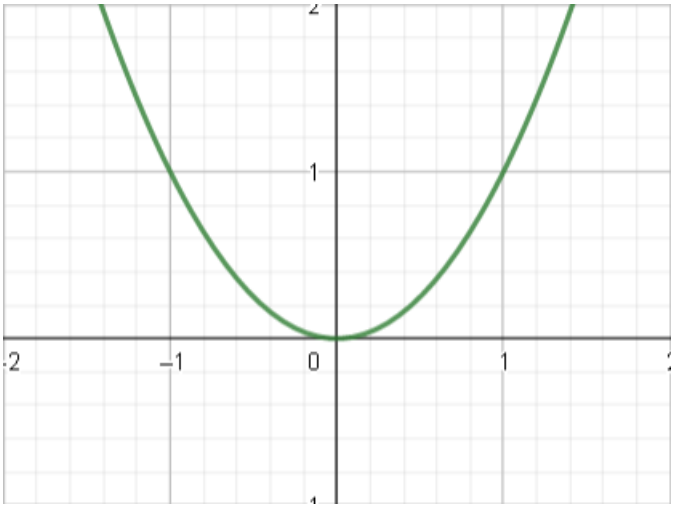
Z intervala in enačbe je razvidno, da je AACF soda funkcija, kar pomeni, da je graf funkcije simetričen glede na ordinatno os (slika 2.1). Preprost primer takšne funkcije podamo z enačbo (1.6):

$$f(x) = x^2 \tag{1.6}$$

Slika 2.1 prikazuje graf sode funkcije $f(x)$, od koder je razvidna lastnost sodih funkcij. Iz grafa je jasno razvidno, da so vrednosti sodih funkcij zrcalne po ordinatni osi, kar omogoča upoštevanje le polovice vseh vrednosti.

Zaradi tega opažanja bomo pri izračunih PSL upoštevali le interval J (1.7), ki vsebuje pozitiven del intervala I, vključno z 0.

$$J = \{0,1,2, \dots, n-1\} \tag{1.7}$$



Slika 2.1: Primer sode funkcije.

Aperiodična avtokorelacijska funkcija je sestavljena iz glavnega (angl. mainlobe) in vsaj enega stranskega režnja (angl. sidelobe), kjer $C_0(B)$ predstavlja glavni reženj in drugi $C_u(B)$, za $u \in \{1,2,\dots,n-1\}$ predstavljajo stranske režnje.

PSL binarne sekvence B izračunamo po enačbi (1.8) (glej članek [20]), ki torej izračuna maksimalno absolutno vrednost $C_u(B)$ za $u \in \{1,2,\dots,n-1\}$.

$$PSL(B) = \max_{0<u<n} |C_u(B)| \tag{1.8}$$

Izračunano vrednost PSL predstavimo tudi v decibelih, kot prikazujeta enačbi (1.9) in (1.10).

$$PSL_{db}(B) = 20 \log\left(\frac{PSL(B)}{n}\right) \tag{1.9}$$

$$PSL_{db}(B) = 20 \log\left(\frac{\max_{0<u<n} |C_u(B)|}{n}\right) \tag{1.10}$$

Pri izračunu aperiodičnih avtokorelacijskih funkcij je pomembna mera tudi faktor ugodnosti (angl. MF - merit factor), ki nam podaja razmerje med energijskim nivojem glavnega in stranskih režnjev. Izračunamo ga lahko po enačbi (1.11).

$$MF(B) = \frac{C_0(B)}{2\sum_{u=1}^{n-1}|C_u(B)|^2} \tag{1.11}$$

V delu [4] je opisana izbira funkcije vrednotenja (angl. fitness function). Za binarno sekvenco B je predlagana funkcija vrednotenja, prikazana z enačbama (1.12) in (1.13). Če želimo najti nižje vrednosti $PSL(B)$, ki je maksimalna vrednost $C_u(B)$ stranskih režnjev, opazimo, da je cilj prikazane funkcije istočasno znižati vrednosti vseh $C_u(B)$ za $u \in \{1,2,\dots,n-1\}$.

$$F(B) = \sum_{u=1}^{n-1}|C_u(B)|^P \tag{1.12}$$

$$F(B) = \sum_{u=1}^{n-1}\left(\left|\sum_{j=0}^{n-u-1} b_j b_{j+u}\right|\right)^P \tag{1.13}$$

Ker torej iščemo sekvence z nizkimi vrednostmi PSL, je logični pristop nižanje vseh $C_u(B)$, zato je takšna izbira smiselna, saj istočasno upošteva vrednosti vseh $C_u(B)$, razen $C_0(B)$. Kritična je izbira vrednosti spremenljivke P, s pomočjo katere izračunavamo potence absolutnih vrednosti $C_u(B)$. Z izbiro vrednosti P v predstavljeni funkciji vrednotenja vplivamo na njeno toleranco. V raziskavi [4] se izkaže, da zniževanje vrednosti P prepušča sekvence z relativno visokimi PSL, medtem ko v obratnem primeru zviševanja vrednosti P iskanje prepogosto zaide v lokalne minimume. Cilj iskanja je približevanje globalnemu minimumu, zato je vrednost P določena kot $P \in [3,5]$. Z upoštevanjem te ugotovitve P zavzame vrednost 4.

Izbira mejne vrednosti t je pomembna za ponastavitev algoritma na začetno stanje, to je začeti iskanje znova z novo začetno rešitvijo. Algoritem, predstavljen v [4], v primeru, da ne najde boljše rešitve s preiskovanjem sosedstva, izvede tako imenovani potres (angl. quake). Potresi se izvajajo, dokler se ne doseže meja t, ki opredeli število le-teh preden algoritem spet začne preiskovanje od začetka. To pomeni, da nizke vrednosti parametra t lahko algoritem ponastavijo, čeprav smo preiskovali dobro rešitev, ki bi morda privedla do še boljših ali celo optimalne rešitve. Visoke vrednosti t algoritem silijo v izvajanje novih potresov, čeprav bi bilo morda že bolje začeti znova. V omenjenem članku je t postavljen na 10^3.

Spodnjo in zgornjo mejo števila preobratov določata parametra $hmin$ in $hmax$. Omejujeta število h, ki določa število naključno izbranih elementov b_i sekvence B, ki jim bomo med izvajanjem potresa spreminjali vrednosti po vnaprej določenem pravilu preobrata (angl. flip)(enačba(1.14)), da se rešimo iz lokalnega minimuma.

$$flip(b_i) = -b_i \tag{1.14}$$

Z upoštevanjem ideje, da je možno z uvedbo sprememb najti boljše rezultate, izberemo vrednost za $hmin$ dovolj majhno, da podpira najmanjše možne spremembe, to je preobrat enega samega elementa v sekvenci. Zato $hmin$ zavzema vrednost 1.

Izbira vrednosti za $hmax$ je tudi iskanje najboljšega obnašanja na meji med natančnostjo in prilagodljivostjo delovanja potresov. V primeru napačne izbire vrednosti zmanjšujemo možnosti, da se rešimo iz lokalnega minimuma, to so premajhne vrednosti $hmax$, medtem ko pri prevelikih vrednostih znatno zmanjšamo uspešnost strategije vzpenjanja na hrib. Spremenljivka $hmax$ zato zavzame vrednost $hmax = \lceil \sqrt{n} \rceil$, kjer je n vhodni podatek algoritma in predstavlja dolžino binarne sekvence.

Zadnji vhodni parameter algoritma je G, ki predstavlja ciljno vrednost algoritma in je tudi končni pogoj ob številu ponastavitev. Parameter G služi za izhod iz algoritma v primeru, da

smo pri preiskovanju prostora uspešni in najdemo ustrezno rešitev. Na temelju vrednosti G lahko iščemo rešitve z določenim PSL.

Algoritem ima poleg ciljne vrednosti G, za izstop iz preiskovanja prostora, tudi maksimalno število iteracij. Naj bo E maksimalno število iteracij algoritma, v primeru [4] ima spremenljivka vrednost 10^5.

Do danes je iskanje binarnih sekvenc z nizko avtokorelacijo atraktivno področje, zato je skozi zgodovino veliko avtorjev, ki jim je uspelo vpeljati izboljšave na omenjenem področju. Z uporabo izčrpnega iskanja, kjer od začetka torej niso bile znane nobene rešitve za zadane probleme, avtorji skozi zgodovino predstavijo svoje najdbe.

2.3 Zgodovinski pregled

Skozi razvoj področja, ki je opisan v [16], so predstavljene rešitve in rezultati različnih avtorjev, medtem ko so v objavljenih raziskavah dokumentirane vrednosti PSL za velikosti sekvenc $n \leq 40$ [15], $n \leq 48$ [1], $n = 64$ [3], $n \leq 68$ [9], $n \leq 74$ [10], $n \leq 80$ [11], $n \leq 82$ [12], $n \leq 84$ [13] in $85 \leq n \leq 105$ [17]. Za sekvence dolžin $n \geq 106$ najdemo poznane vrednosti v literaturi v prispevkih [5] in [4].

Ker je procesiranje informacij in signalov tako pomembno, je potreba po raziskovanju obstoječih in iskanju novih rešitev velika. Namen je raziskati in implementirati algoritem na osnovi strategije SHC za iskanje binarnih sekvenc z malimi ali minimalnimi vrednostmi PSL.

Izhajamo iz obstoječih raziskav, kjer iščemo možnosti za izboljšave. S procesom eliminacije napačnih in neuspešnih postopkov se približujemo boljšim ter učinkovitejšim rešitvam, ki nam omogočajo tudi nadaljnje raziskave.

V članku [4] so dokumentirani delovanje algoritma in njegove nastavitve, ki krmilijo obnašanje, predstavljeni so tudi dobljeni rezultati, to so vrednosti PSL za sekvence dolžin $106 \leq n \leq 300$.

3. IMPLEMENTACIJA ALGORITMA

3.1 Namen

Implementirali smo algoritem, s katerim izvajamo poskuse in predstavimo dobljene rezultate. Spoznavanje in razumevanje omenjene strategije, raziskovanje obnašanja različnih velikosti sekvenc, razumevanje izbire funkcije vrednotenja in analiza drugih mejnih parametrov nam izboljša razumevanje trenutno atraktivnih pristopov na tem področju in zakaj so uporabnejši kot drugi.

Raziskava dobljenih rezultatov, dokumentiranje uporabljenih pristopov in sprotna diskusija dajo nov ali drugačen pogled na že znane informacije. S tem diplomskim delom želimo prikazati dobre in slabe prakse rabe algoritma [4] ter tudi prikazati obnašanje atraktivne rešitve ob različnih krmilnih parametrih. Pri časih izvajanja je pomembna velikost sekvenc, kjer pri dolgih sekvencah iskanje traja ogromno časa.

3.2 Strojna oprema

Za začetek je potrebna vzpostavitev okolja, v katerem lahko algoritem implementiramo in nato izvajamo eksperimente. Implementacijo smo naredili na osebnem prenosnem računalniku (tabela 3.1). Računalnikov hranilni medij (tabela 3.2) je tipa SSD (angl. solid state drive).

Tabela 3.1: Specifikacije sistema.

OS Name	Microsoft Windows 10 Home
Version	10.0.18362 Build 18362
Other OS Description	Not Available
OS Manufacturer	Microsoft Corporation
System Name	LAPTOP-E6FH4R0A
System Manufacturer	LENOVO
System Model	81FV
System Type	x64-based PC
System SKU	LENOVO_MT_81FV_BU_idea_FM_Legion Y530-15ICH
Processor	Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz, 2304 Mhz, 4 Core(s), 8 Logical Processor(s)
BIOS Version/Date	LENOVO 8JCN52WW, 6/14/2019
SMBIOS Version	3.0
Embedded Controller Version	1.52
BIOS Mode	UEFI
BaseBoard Manufacturer	LENOVO
BaseBoard Product	LNVNB161216
BaseBoard Version	SDK0J40709 WIN
Installed Physical Memory (RAM)	8.00 GB
Total Physical Memory	7.88 GB
Available Physical Memory	2.79 GB
Total Virtual Memory	12.6 GB
Available Virtual Memory	3.72 GB
Page File Space	4.75 GB

Kot je razvidno iz podanih podatkov, so specifikacije sistema dokaj povprečne ali podpovprečne, če upoštevamo razvoj tehnologije. To pomeni, da z rešitvijo [4] lahko izvajamo iskanje na osebnih računalnikih. Možnost uporabe šibkejših osebnih računalnikov, v primerjavi s superračunalniki, je za izvajanje iskanja binarnih sekvenc velika prednost. Uporaba osebnih računalnikov v te namene pripomore k hitrejšemu razvoju in iskanju rešitev na globalni ravni.

Tabela 3.2: Specifikacije sistema 2

Model	WDC PC SN720 SDAPNTW-512G-1101
Bytes/Sector	512
Media Loaded	Yes
Media Type	Fixed hard disk
Partitions	3
SCSI Bus	0
SCSI Logical Unit	0
SCSI Port	0
SCSI Target ID	0
Sectors/Track	63
Size	476.94 GB (512,105,932,800 bytes)
Total Cylinders	62,260
Total Sectors	1,000,206,900
Total Tracks	15,876,300
Tracks/Cylinder	255
Partition	Disk #0, Partition #0
Partition Size	260.00 MB (272,629,760 bytes)
Partition Starting Offset	1,048,576 bytes
Partition	Disk #0, Partition #1
Partition Size	475.69 GB (510,770,806,784 bytes)
Partition Starting Offset	290,455,552 bytes
Partition	Disk #0, Partition #2
Partition Size	1,000.00 MB (1,048,576,000 bytes)
Partition Starting Offset	511,061,262,336 bytes

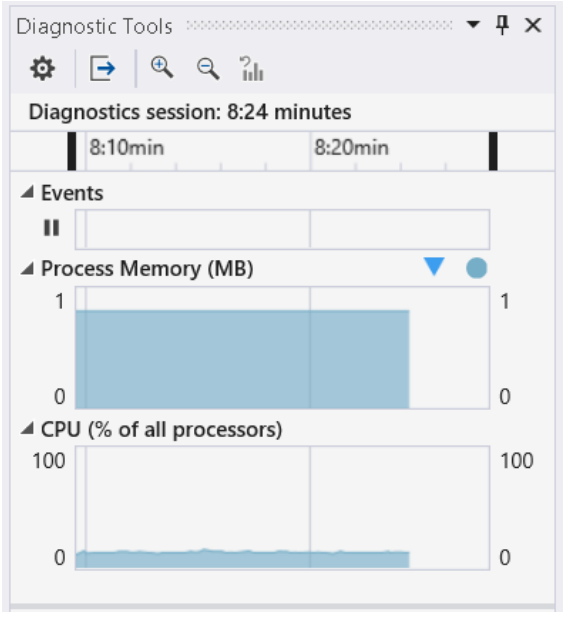
3.3 Programska oprema

Za razvijalno okolje smo izbrali Visual Studio (VS) podjetja Microsoft. VS je integrirano razvijalno okolje (angl. IDE – integrated development environment) , namenjeno razvoju programov za Windows OS, izdelavi aplikacij, osnovanih na ogrodju .NET ter izdelavi spletnih aplikacij, strani in storitev. Uporablja Microsoftove platforme za razvoj programske opreme, kot so Windows API (angl. application programming interface), Windows Forms, Windows Presentation Foundation in druge. Zmožno je producirati strojno in upravljano kodo. Uporabljeno je tudi za razvoj v interpretiranih jezikih.

Vsebuje urejevalnik kode, ki podpira tehnologijo IntelliSense, ki je komponenta za dopolnjevanje kode (slika 3.1) in komponenta za refaktoriranje (izboljševanje oblike, strukture in/ali implementacije brez spreminjanja delovanja).

Slika 3.1: Primer rabe IntelliSense.

Integriran je razhroščevalnik, ki deluje na nivoju razhroščevanja izvorne kode, in tudi na nižjih nivojih razhroščevanja strojnega jezika. Vključuje tudi uporabno orodje za profiliranje kode, ki meri čas izvajanja in zasedenost pomnilnika ter podaja številne druge podatke, ki služijo kot povratna informacija razvijalcem (slika 3.2).



Slika 3.2: Orodje za diagnostiko.

Možna je tudi uporaba razširitev, kot na primer sistem za verzioniranje kode ali dodajanje novih orodij (angl. Toolset). Ti so urejevalniki kode, grafični vmesniki za domensko-specifične jezike in orodja za pomoč pri razvoju programske opreme skozi svoj življenjski cikel.

VS podpira številne različne programske jezike ter dovoljuje urejevalniku kode in razhroščevalniku, da do določene mere dela s skoraj katerim koli jezikom. Vgrajeni jeziki so C, C++, C#, C++/CLI, Visual Basic .NET, F#, Javascript, Typescript, XML, XSLT, HTML in CSS, medtem ko je podpora za druge jezike, kot na primer Python, Ruby, Node.js in M, dosežena s pomočjo namestitve razširitev. V preteklosti sta bila podprta tudi Java in J#.

Prednost uporabe VS je v programski opremi Visual Studio Community Edition, ki je brezplačna različica z vsemi opisanimi funkcionalnostmi na operacijskih sistemih Windows za osebno in akademsko uporabo in rabo v manjših skupinah. Alternativa je izdaja Visual Studio Express, ki ne podpira razširljivosti prek vtičnikov. Trenutno podprta je izdaja Visual Studio 2019.

Pri implementaciji smo se odločili za uporabo programskega jezika C++. C++ je splošno-namenski programski jezik, razvit iz programskega jezika C in imenovan tudi C z razredi. Bjarne Stroustrup je v Bell Labs razvijal razširitev jezika C, ker je želel učinkovit in prilagodljiv jezik, ki podpira tudi visoko-nivojske funkcionalnosti za organizacijo. Jezik se je čez čas zelo razširil, in moderna različica jezika C++ podpira objektno-orientirano, generično in funkcionalno programiranje ter možnost nizko-nivojske manipulacije virov (npr. upravljanje hranjenja podatkov). Skoraj vedno je implementiran kot prevajan jezik in obstaja veliko različnih prevajalnikov (angl. compilers). Zato je na voljo na različnih platformah.

C++ je bil zasnovan za programsko opremo z omejenimi viri, a predvsem z namenom dobrega delovanja, učinkovitosti in prilagodljivosti. Njegova uporabna vrednost je vidna pri izdelavi namiznih aplikacij, video iger, strežnikov in predvsem za aplikacije, katerih učinkovitost je kritičnega pomena.

Ima dve glavni komponenti, prva je komponenta neposrednega mapiranja funkcionalnosti strojne opreme, kar je v osnovi lastnost jezika C. Avtor Stroustrup je opisal C++ kot programski jezik lahke kategorije oblikovan za gradnjo in uporabo učinkovitih in elegantnih abstrakcij, ponuja dostop do strojne opreme, kjer je cilj učinkovitost v primerjavi z drugimi jeziki. Podedoval je tudi večino sintakse programskega jezika C.

Standardizirala ga je mednarodna organizacija za standardizacijo (angl. ISO – International Organization for Standardization) z zadnjo različico objavljeno decembra 2017 kot ISO/IEC 14882:2017. Ta je neuradno poznan tudi kot C++17. V osnovi je bil standardiziran s strani ISO v letu 1998 kot ISO/IEC 14882:1998, ki so ga kasneje nadgradili še C++3, C++11 in C++14 vse do zdajšnjega C++17. Trenutni C++17 seveda zamenja in nasledi prejšnje z dodatnimi funkcionalnostmi in povečano standardno knjižnico.

Čez leta je jezik doživel tudi dosti kritike, pri čemer je bila kompleksnost jezika poudarjena kot njegova težava.

3.4 Implementacija

Implementirali smo algoritem za iskanje binarnih sekvenc z nizko avtokorelacijo v programskem jeziku C++ s pomočjo okolja Visual Studio. Algoritem je predstavljen v članku [4]. Okolje VS je pomagalo predvsem pri razhroščevanju in spremljanju porabe virov.

Idejo algoritma predstavimo s potekom skozi njegov življenjski cikel (slika 3.3).

Algoritem 1

```
1. PROCEDURA
2.      generiraj naključno začetno binarno sekvenco B
3.      ovrednoti generirano začetno binarno sekvenco
4.      najboljša poznana generirana sekvenca je trenutna
5.      naredimo kopijo trenutne sekvence
6.      PONOVI
7.          Generiraj vse sosede trenutne binarne sekvence
8.          Če je PSL soseda enak iskanemu, ga izpišemo in
            zaključimo
9.          Ovrednotimo soseda
10.         ČE je boljši kot sekvenca, iz katere izvirajo sosedi,
                postane nova osnovna sekvenca in se vrnemo
                na generiranje sosedov iz trenutne sekvence
                (vrstica 7)
11.         SICER Preverimo ali je trenutna sekvenca boljša kot
                najboljša poznana
12.             ČE je, shranimo novo najboljšo poznano in se
                vrnemo na generacijo sosedov (vrstica 7)
13.             SICER pa izvedemo potres kjer naključno
                spremenimo določena mesta sekvence, kar
                lahko ponovimo omejeno število krat
14.     DOKLER ne dosežemo željenega števila ponavljanj
15. KONEC PROCEDURE
```

Slika 3.3: Besedni opis poteka algoritma [4].

Najprej generiramo in ovrednotimo binarno sekvenco, ki je začetna in tudi najboljša poznana. Vedno hranimo kopijo najboljše najdene sekvence, zato shranimo tudi kopijo začetne sekvence. Hranimo tudi vrednost, ki pove število izvedenih potresov, katerih število seveda omejimo pred začetkom, to je v fazi kreiranja začetne sekvence.

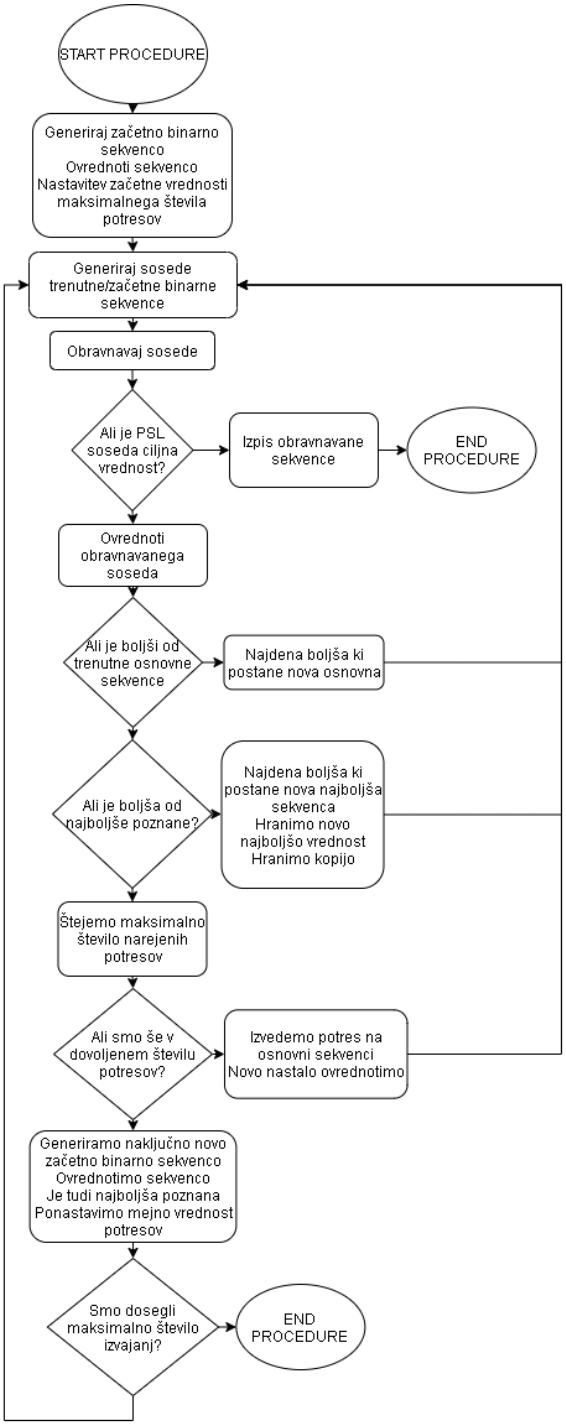
Nato v glavni zanki stopimo v fazo generacije sosedov začetne binarne sekvence. Zanima nas, ali je kateri od sosedov »popoln« oziroma je njegova vrednost PSL enaka naši iskani, kjer trenutno obravnavanega soseda izpišemo in zaključimo iskanje.

Če obravnavani sosed ne ustreza ciljnemu pogoju našega iskanja, ga primerjamo s trenutno najboljšo oziroma začetno binarno sekvenco. V primeru, da je obravnavani sosed boljši, postane ta naša nova osnovna sekvenca. V tem primeru se vrnemo nazaj h generaciji sosedov in preverjanju le-teh. Če na neki točki več ne najdemo boljših sosedov, postopek prekinemo z namenom ali hranjenjem najboljšega najdenega ali izvajanja potresov.

Kadar ni najdenih uspešnejših sosedov, preverimo najprej, ali je naša trenutno obravnavana binarna sekvenca boljša od najboljše poznane. Če je, potem shranimo novo najboljšo najdeno sekvenco in njeno vrednost ter ponastavimo vrednost maksimalnega števila dovoljenih potresov. Od tod se vrnemo nazaj h generaciji sosedov iz nove najboljše sekvence.

V drugem primeru, kjer trenutno obravnavana sekvenca ni boljša od najboljše poznane, izvajamo potres, ampak le če nam to mejna vrednost dovoljuje. Dokler smo znotraj omejenega števila, potrese izvajamo, pri čemer obravnavani sekvenci spremenimo naključno število vrednosti, kjer prav tako naključno izbiramo pozicije sprememb.

Tako obdelana sekvenca se nato ovrednoti in postavi standard za nadaljnje primerjave, a če smo prekoračili maksimalno število potresov, začnemo iskanje znova in naredimo popolno ponastavitev (angl. restart) vseh vrednosti. To pomeni, da začnemo z generacijo povsem nove začetne binarne sekvence, ki jo seveda ponovno ovrednotimo in bo tudi trenutno najboljša poznana, naredimo kopijo le-te in ponastavimo mejno vrednost, ki služi kot omejitev za maksimalno dovoljeno število potresov. Idejno algoritem predstavimo tudi z diagramom poteka (slika 3.4).



Slika 3.4: Diagram poteka implementiranega algoritma.

Razvidno je, da torej postopek iskanja konča izvajanje, kadar je najdena sekvenca z iskano vrednostjo PSL ali dosežemo maksimalno število ponovitev in tako prekinemo iskanje z vnaprej določenim mejnim parametrom.

3.5 Kodiranje algoritma

Binarna sekvenca vsebuje dve vrednosti: -1 in 1. Generiranje naključnih sekvenc prikazuje slika 3.5 (primer naključno generirane sekvence dolžine 10 na sliki 3.6).

```
//generate a RANDOM binary sequence
int* binarySeq = new int[n];
for (int i = 0; i < n; i++) {
    int temp = rand() % 2 + 0;
    if (temp == 0) temp = -1;
    binarySeq[i] = temp;
    cout << binarySeq[i] << " ";
}
```

Slika 3.5: Kodiranje začetne sekvence.

Slika 3.6: Primer naključno generirane binarne sekvence.

Funkcijo vrednotenja prikazujemo na sliki 3.7. Ob upoštevanju enačbe (1.13) smo funkcijo vrednotenja implementirali z uporabo tabele, v katero se shranjujejo posamezne vrednosti $C_u(B)$ za $u \in \{1,2,\dots,n-1\}$. Nato absolutne vrednosti posameznih elementov te tabele, pod pogojem, da je za dani element $u > 0$, potenciramo z vrednostjo P, ki je v našem primeru kar $P = 4$.

```
int evaluate1D(int sequence[], int n) {
    int fitness = 0;
    int* CuB = new int[n];
    int P = 4;
    for (int u = 0; u < n; u++) {
        CuB[u] = 0;
    }
    //calculation of Cu(B) where for every u we calculate CuB[u] from sum for j from 0 to n-u-1.
    for (int u = 0; u < n; u++) {
        for (int j = 0; j <= n - u - 1; j++) {
            CuB[u] += sequence[j] * sequence[j + u];
        }
        if (u > 0) { //we calculate fitness for sidelobe
            fitness += pow(abs(CuB[u]), P);
        }
    }
    delete[] CuB;
    return fitness;
}
```

Slika 3.7: Implementacija funkcije vrednotenja.

Po nastavitvi začetnih vrednosti algoritem izvaja iskanje po vnaprej določenem številu ponovitev, to je v našem primeru 10^5. Na začetku kreiramo sosede trenutne binarne sekvence $binarySeq$ (slika 3.8).

```
//define initiate neighbours
int** binarySeqNeighbours = new int*[n];
for (int i = 0; i < n; i++)
    binarySeqNeighbours[i] = new int[n];
//generate neighbours
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (i == j)
            binarySeqNeighbours[i][j] = binarySeq[j] * -1;
        else
            binarySeqNeighbours[i][j] = binarySeq[j];
    }
}
```

Slika 3.8: Tvorba sosedov.

Slika 3.9 prikazuje del kode za izračun vrednosti PSL.

```
for (int i = 0; i < n; i++) {
    //generate
    int PSL = 0;
    //calculate PSL with  AACF Cu(X) where X is neighbour and find PSL
    for (int u = 0; u < n; u++) {
        for (int j = 0; j <= n - u - 1; j++) {
            AACF[i][u] += binarySeqNeighbours[i][j] * binarySeqNeighbours[i][j + u];
        }
        //after calculation check PSL
        if (u > 0) { //for 0<u<n
            if (abs(AACF[i][u]) > PSL) {
                PSL = abs(AACF[i][u]);
            }
        }
    }
```

Slika 3.9: Izračun vrednosti PSL.

V naslednjem koraku (slika 3.10) algoritem preveri, ali je izračunana vrednost PSL enaka ciljni vrednosti G in če velja $PSL = G$, prekinemo izvajanje iskanja in izpišemo najdeno sekvenco na standardni izhod.

```
//Compare with desired PSL
if (PSL == G) {
    cout << "Perfect PSL reached with: " << PSL << endl;
    prepareHex(binarySeqNeighbours[i], n);
    for (int k = 0; k < n; k++) {
        cout << binarySeqNeighbours[i][k] << " ";
    }
    goalReached = true;
}
if (goalReached) break;
```

Slika 3.10: Primerjava izračunane vrednosti PSL.

V primeru, kadar ni najdena ciljna vrednost PSL, preverimo, ali smo našli boljšo sekvenco, kot je trenutna najboljša. Kadar je obravnavana sosedna sekvenca boljša od $binarySeq$, jo shranimo. V primeru da je boljša kot najboljša globalna, posodobimo tudi to.

Kadar v sosednih sekvencah ni več izboljšav, algoritem preverja, ali je še v dovoljenem številu potresov. Ko je pogoj izpolnjen, izvaja na najboljši najdeni sekvenci potres. To je

postopek, pri katerem naključno izbrane elemente sekvence algoritem spremeni, medtem ko preostale neizbrane preslika iz nespremenjene sekvence. Na ta način se uvede mutacija najboljše sekvence v upanju, da iskanje najde še boljše.

Potres prikazuje slika 3.11. Izvajamo ga na h naključno izbranih elementih. Vrednost spremenljivke h smo določili kot celo število na intervalu $[1, \lceil\sqrt{n}\rceil]$. Algoritem izbere h naključnih indeksov i, kjer $i \in \{0,1,2,\dots,n-1\}$ in spremeni vrednost elementov na izbranih indeksih, medtem ko drugi elementi ostanejo nespremenjeni.

```
    //flip h random bits in binSeq
    bool* flipPosPool = new bool[n];
    for (int i = 0; i < n; i++)
        flipPosPool[i] = false;
    int weGoodCounter = 0;
    while (weGoodCounter<h) {
        int flipPos = rand() % n;
        if (flipPosPool[flipPos] == false) {
            flipPosPool[flipPos] = true;
            weGoodCounter++;
        }
    }
    for (int i = 0; i < n; i++) {
        if (flipPosPool[i] == true) {
            binarySeq[i] = binarySeq[i] * -1;
        }
    }
    //calculate new bestFit
    bestFit = evaluate1D(binarySeq, n);
    //test output
    cout << "new binarySeq with h:"<< h << " flipped and new bestfit: " << bestFit << endl;
    //deallocate
    delete[] flipPosPool;
}
```

Slika 3.11: Drugi del pogojnega stavka thresholdLeft>0.

Implementirali smo tudi funkcijo $prepareHex()$ (slika 3.12), s pomočjo katere binarno sekvenco predstavimo v šestnajstiški obliki. Funkcija kot vhodna parametra prejme sekvenco $sequence$ in njeno dolžino n.

```
void prepareHex(int sequence[], int n) {
    int* binaryForm = new int[n];
    stringstream binaryStream;
    for (int i = 0; i < n; i++) {
        if (sequence[i] == -1) binaryForm[i] = 0;
        else binaryForm[i] = 1;

        binaryStream << binaryForm[i];
    }
    string SBF = binaryStream.str();
    int diff = SBF.length() % 8;
    int mod = 8 - diff;
    int padd = SBF.length() + mod;
    stringstream paddedStream;
    if (diff != 0) paddedStream << setw(padd) << setfill('0') << SBF;
    string padded = paddedStream.str();
    stringstream ssss;
    int base = 2;
    for (int i = 0; i < padded.length(); i += 8) {
        string eightBit = padded.substr(i, 8);
        char* pointer = NULL;
        const char* temp = eightBit.c_str();
        ssss << hex << strtol(temp, &pointer, base);
    }
    cout << ssss.str() << endl;
    delete[] binaryForm;
}
```

Slika 3.12: Funkcija prepareHex za predstavitev sekvence v hexadecimalni obliki.

Koda implementiranega algoritma je v prilogi A1.

3.6 Generatorji naključnih števil

Ker algoritem uporablja naključne vrednosti, je pomembno predstaviti pomembnost generatorjev naključnih števil. Kot osnovno možnost nudi jezik C++ metodo $rand()$, ki za delovanje potrebuje seme $srand(time(NULL))$ in je na voljo z uporabo knjižnic $< stdlib.h >$ in $< time.h >$. Alternativa je uporaba drugih, predstavljenih v kasnejših

različicah (C++11 in kasneje). Pojavi se možnost uporabe knjižnice $< random >$, ki vsebuje število različnih generatorjev za različne namene. Izbira ustreznega generatorja naključnih števil je lahko pomembna in obstaja veliko možnosti.

Za primerjavo smo uporabili naključni generator števil Marsenne Twister, to je $mt19937$ iz knjižnice $< random >$. Implementacija (slika 3.13) omenjenega generatorja naključnih števil je idejno podobna implementaciji, pri kateri smo uporabili funkcijo $rand()$. Tudi Marsenne Twister potrebuje seme, ki ga predstavlja spremenljivka rd podatkovnega tipa $random\_device$.

```
random_device rd;  //Will be used to obtain a seed for the random number engine
mt19937 gen(rd()); //Standard mersenne_twister_engine seeded with rd()
uniform_int_distribution<> distrib(0, 1);
uniform_int_distribution<> distrib2(hmin, hmax);
uniform_int_distribution<> distrib3(0, n-1);
```

Slika 3.13: Naključni generator števil Marsenne Twister.

Generator opredelimo s spremenljivko gen, ki je tipa $mt19937$ in pri inicializaciji v konstruktorju sprejme prej opredeljeno spremenljivko rd kot $mt19937\ gen(rd())$, ki v tem primeru služi kot seme. Na tej točki je omenjen generator naključnih števil zasejan. Zadnji korak priprav je opredelitev intervalov, znotraj katerih bo naš generator vračal vrednosti. To storimo tako, da opredelimo spremenljivke $distrib$, $distrib2$ in $distrib3$ podatkovnega tipa $uniform\_int\_distribution <>$, le-tem ob generaciji v konstruktorju podamo spodnjo in zgornjo mejo intervala kot $uniform\_int\_distribution <>$ $distrib(0, 1)$, znotraj katerega iščemo celoštevilsko vrednost. S tem je priprava generatorja Marsenne Twister končana in je pripravljen za uporabo. S klicem $distrib$ s podanim gonilnikom (gen tipa $mt19937$), kot $distrib(gen)$ (slika 3.14) generiramo naključno število. Klic $distrib(gen)$ v našem primeru vrača celoštevilsko vrednost iz intervala [0, 1]. Dobljeno vrednost zaradi načina in potrebe uporabe hranimo v začasno spremenljivko $temp$ podatkovnega tipa int.

```
int temp = distrib(gen);
```

Slika 3.14: Generacija naključnega števila.

Skozi algoritem se na več mestih uporablja naključni generator števil: pri tvorbi naključno generirane binarne sekvence in pri določitvi pozicije izvajanja operacije $flip(b_i)$, pri čemer izberemo h naključnih elementov, nad katerimi izvedemo operacijo $b = -b$ oziroma $flip(b_i)$. V teh primerih smo v času testiranja priredili implementacijo algoritma, in zamenjali predhodno uporabljen generator z Marsenne Twister (koda je v prilogi A2).

4. PREDSTAVITEV REZULTATOV

Z implementiranim algoritmom smo izvedli poskuse in dobljene rezultate predstavimo v tem poglavju. Rezultate bomo opremili tudi z razlago, izpostavili bomo posebnosti in govorili o novih ugotovitvah.

Pričakovali smo, da bodo rezultati raziskave, ki so bili predstavljeni v članku [4], ponovljivi in tudi so. S pomočjo podanega članka je možno izdelati implementacijo algoritma za iskanje binarnih sekvenc. Rešitev je torej uspešna in ima uporabno vrednost. Zaradi dolgih časov iskanja je testiranje časovno zahtevno in naporno v primeru pojavitve napak. Implementacija algoritma je tako potekala v več sprintih, skozi katere se je izdelovalo program.

Po fazi implementacije je sledila faza testiranja, tak cikel se je ponavljal, dokler ni bilo doseženo željeno obnašanje. Ker algoritem najde izboljšave, opisane v članku [4], dosti redkeje kot najboljše poznane vrednosti, je bil korak testiranja časovno zelo zahteven.

4.1 Nastavitev parametrov pri algoritmu

Obnašanje algoritma krmilijo vhodni parametri n, $hmax$, $hmin$ in t. Dolžino n binarne sekvence postavimo na celoštevilsko vrednost, kjer je n na intervalu $106 \leq n \leq 123$, mejno vrednost t definiramo kot $t = 10^3$ na podlagi predhodnega znanja in spoznanj. Enako se orientiramo pri postavitvi vrednosti za spremenljivki $hmin$ in $hmax$. Spremenljivki $hmin$ priredimo celoštevilčno vrednost 1, spremenljivki $hmax$ pa celoštevilčno vrednost $\lceil \sqrt{n} \rceil$. V tem primeru iščemo binarno sekvenco, katere vrednost PSL bo enaka iskani vrednosti G. Na primer, za $n = 123$ spremenljivki G priredimo vrednost 7, kar predstavlja ciljno vrednost PSL za binarno sekvenco dolžine 123 (glej [4]).

Parametru $hmin$ je nesmiselno spreminjati vrednost iz $hmin = 1$, saj mora dopuščati najmanjšo možno število sprememb, to je $h = 1$, torej $flip(b_i)$ za en naključno izbran element b_i. Ena sprememba lahko vodi do sekvence z iskano vrednostjo PSL. Najmanjše število možnih sprememb tako ostaja $hmin = 1$.

Parameter $hmax$ postavlja zgornjo mejo za število možnih prireditev h, elementom b_i kot $flip(b_i)$. Vrednost spremenljivke $hmax$ smo primerno pripravili pred uporabo v samem algoritmu, torej preden jo funkcija $procedureSHC()$ prejme na vhodu. V članku [4] je uporabljeno zaokroževanje navzgor, za to uporabimo funkcijo $ceil()$ standardne knjižnice $< math.h >$. Uporaba funkcij, kot so $floor()$, $trunc()$ in $round()$, se ne zdi napačna, ampak skozi testiranje ni pokazala nobenih izboljšav v delovanju oziroma je iskanje rezultatov bilo manj uspešno. Pričakovali bi, da se bo algoritem z uporabo katerekoli od teh funkcij za pripravo vrednosti $hmax$ obnašal podobno, ampak se izkaže, da je pomembno izvajanje operacije $flip(b_i)$ nad naključno izbranimi elementi sekvence največ $\lceil \sqrt{n} \rceil$ krat. Z manjšanjem vrednosti $hmax$ se veča možnost obstoja v lokalnem minimumu, takrat algoritem ni zmožen ustrezno spremeniti sekvence in kljub spremembam ne uide lokalnim minimumom. Višanje vrednosti $hmax$ prinese večjo mutacijo binarne sekvence in tako rešuje sekvenco iz lokalnega minimuma z uvedbo dovolj velike spremembe. Potrebna je postavitev zgornje meje saj posledično prevelike spremembe zelo nižajo učinkovitost procedure vzpenjanja. Enako velja za parameter t, s katerim nadziramo število potresov. V implementaciji algoritma parameter t nastavimo kot $t = 10^3$ na podlagi rezultatov raziskave [4] in s tem omogočamo ponastavitev algoritma na začetno stanje. Velike vrednosti t onemogočajo algoritmu učinkovito vzpenjanje zaradi ujetosti v lokalnem optimumu, istočasno pa majhne vrednosti prekinjajo vzpenjanje, ki lahko vodi do željenega rezultata.

4.2 Rezultati

Izvedli smo poskus, v katerem smo algoritem pognali 30 krat za vsako binarno sekvenco dolžine n, kjer $106 \leq n \leq 123$. Zanimala nas je najboljša, povprečna in najslabša najdena vrednost PSL.

Dobljene rezultate predstavimo v tabeli 4.1. V članku [4] najdemo binarne sekvence z ciljnimi vrednostmi PSL in jih prikazujemo v zadnjem stolpcu ($PSL[4]$).

V primeru izvajanja algoritma 30 krat za dolžine $106 \leq n \leq 113$ namreč nismo našli sekvence z vrednostjo PSL boljšo kot 7. Algoritem za dolžine $106 \leq n \leq 113$ vedno najde sekvence z vrednostjo PSL enako 7, je pa zmožen najti tudi sekvence, katerih PSL je enak 6. Pri sekvencah dolžine 114 je najslabši najden PSL enak 8, večina pa enaka 7. Za dolžine $115 \leq n \leq 123$ vedno najde sekvence z vrednostjo PSL enako 8, najde pa tudi sekvence s PSL enakim 7. V [4] so predstavljene sekvence s $PSL = 6$ za dolžine $106 \leq n \leq 114$ in $PSL = 7$ za $115 \leq n \leq 123$.

Izvajanje iskanja s predlaganim algoritmom na opisanem sistemu (glej tabelo 3.1) traja od nekaj deset minut, do več ur. Med izvajanjem poskusa smo ugotovili, da je za iskanje sekvenc, katere vrednost PSL je enaka podani v [4], potrebno izvajanje velikega števila zagonov algoritma, torej 30 zagonov algoritma je bilo premalo, da bi vedno našli sekvenco s PSL iz članka [4]. Zato smo se odločili, da bomo algoritem pognali še večkrat in dobljene rezultate predstavili v nadaljevanju.

Tabela 4.1: Poskus z izvajanjem 30 zagonov za posamezno dolžino binarne sekvence.

n	$najboljši\ PSL$	$najslabši\ PSL$	$povp.PSL$	$PSL[4]$
106	7	7	7	6
107	7	7	7	6
108	7	7	7	6
109	7	7	7	6
110	7	7	7	6
111	7	7	7	6
112	7	7	7	6
113	7	7	7	6
114	7	8	$7.0\overline{3}$	6
115	7	7	7	7
116	7	7	7	7
117	7	8	7.1	7
118	7	8	$7.2\overline{3}$	7
119	7	8	$7.1\overline{6}$	7
120	7	8	$7.3\overline{6}$	7
121	7	8	$7.6\overline{3}$	7
122	7	8	$7.7\overline{6}$	7
123	7	8	7.7	7

V tabeli 4.2 predstavimo najboljše najdene binarne sekvence, ko smo algoritem dodatno zaganjali (več zagonov kot 30, za določene sekvence je bilo potrebnih približno 100 zagonov). Podana je dolžina sekvence, dobljen PSL, sekvenca v šestnajstiški obliki in čas izvajanja algoritma. Za dolžine $106 \leq n \leq 112$ opazimo, da je algoritem uspel najti binarne sekvence, katerih vrednosti PSL so enake 6, za dolžine $123 \leq n \leq 126$ pa 7. PSL vrednosti sekvenc v tabeli 4.2 so enake najboljšim podanim v članku [4]. Čas iskanja je bil merjen od

začetka do konca izvajanja algoritma, kjer je bila najdena binarna sekvenca z ustreznim PSL. Dobljeni časi iskanja so presenetljivo kratki, v primerjavi s časom, ko se je algoritem ustavil po 10^5 ponastavitev (reinicializacija začetne binarne sekvence).

Tabela 4.2: Najboljše najdene binarne sekvence.

n	PSL	$binarySeq$ (Hexadecimal)	čas (s)
106	6	024d13e9ddff033d7ad299c575c	135
107	6	75b575f2b13f8902766bfc87186	336
108	6	8e95434f2d085beef32fe7b322	191
109	6	1fe1f78c3d98c8d92216955492f	537
110	6	1b36ac1851d65a84508cc07d381f	507
111	6	1d53c81f5c95dbf2e648061219c	447
112	6	a2794388c944d7d0defd3a444bdd	93
123	7	179cf2ba8a5a24e267e106c9fc326	28
124	7	7e6a0541a5befc9ecd9c7771956f46	231
125	7	0d7f18fbfa7a255cdc6d076c55a1a44	433
126	7	fefa151c6f5228689b0f6b2c333023	545

Povprečen čas iskanja binarnih sekvenc za dolžine $106 \leq n \leq 112$, pri katerih je algoritmu uspelo najti željeno vrednost PSL (torej pri izračunu uporabimo vrednosti iz zadnjega stolpca v tabeli 4.2) je enak 8 minut in 22 sekund ali 502 sekundi. Kot primer izpostavimo najdeno sekvenco za dolžino $n = 123$ (več rezultatov v prilogi B1), za katero je iskanje trajalo le 28 sekund. Primer hitro najdene ustrezne binarne sekvence, ki je bila najdena v 28 sekundah: $binarySeq_{(hex)} = 179\text{cf}2\text{ba}8\text{a}5\text{a}24\text{e}267\text{e}106\text{c}9\text{fc}326$.

Preizkusili smo delovanje dveh generatorjev naključnih števil. Izkazalo se je, da uporaba različnih generatorjev ni imela velikega vpliva na rezultate algoritma.

5. SKLEP

V diplomskem delu smo pripravili implementacijo algoritma, predstavljenega v članku [4], za iskanje binarnih sekvenc z nizko avtokorelacijo. Skozi delo predstavimo izvedbo izdelave, kjer govorimo o delovanju posameznih gradnikov implementacije in na koncu predstavimo eksperimentalne rezultate implementiranega algoritma.

Poskuse smo izvedli na dolžinah od 106 do 123, nad katerimi smo zagnali algoritem 30 krat. Izkazalo se je, da algoritem ne najde binarne sekvence z optimalno vrednostjo PSL v vsakem zagonu. Kljub izvajanju potresov nad najboljšo najdeno sekvenco in ponovnem preiskovanju sosedov lahko zaide v lokalni optimum. Tudi ponastavitev v začetno stanje ob omejenem številu potresov ni vedno uspešna. Skozi izdelavo algoritma od prve do zadnje verzije je bilo veliko sprememb, kot tudi poskusov z drugačnimi pristopi, kot je bilo na začetku načrtovano. Algoritem vsebuje računske operacije potenciranja in seštevanja v primeru ovrednotenja binarne sekvence ter množenja s celoštevilsko vrednostjo -1 za pridobitev nasprotne vrednosti posameznega elementa sekvence in primerjave vrednosti.

Z implementiranim algoritmom smo uspeli najti 224 binarnih sekvenc, ki imajo enako vrednost PSL kot v članku [4].

SEZNAM VIROV IN LITERATURE

[1] Baden J., Cohen M., "Optimal peak sidelobe filters for biphase pulse compression", IEEE Int. Conf. Radar, 1990, str. 249-252.

[2] Barker R. H. and Jackson W., “Group synchronization of binary digital systems,” v Communication Theory, New York, NY, ZDA: Academic, 1953, str. 273–287.

[3] Coxson G., Russo J., "Efficient exhaustive search for optimal-peak-sidelobe binary codes", IEEE Transactions on Aerospace and Electronic Systems, vol. 41, st. 1, Jan. 2005, str. 302-308.

[4] Dimitrov M., Baitcheva T., Nikolov N., 2020. Efficient generation of low auto-correlation binary sequences. IEEE Signal Processing Letters, 27, str. 341-345.

[5] Du K. L., Wu W. H., Mow W. H., "Determination of long binary sequences having low autocorrelation functions", U.S. Patent 8,493,245, Jul. 2013.

[6] Gold R., “Optimal binary sequences for spread spectrum multiplexing (Corresp.),” IEEE Transactions on Information Theory, vol. IT-13, st. 4, Okt. 1967, str. 619–621.

[7] Golomb S. W. et al., Shift Register Sequences, Laguna Hills, CA, ZDA: Aegean Park, 1967.

[8] Kasami T., “Weight distribution formula for some class of cyclic codes,” Coordinated Science Laboratory, Urbana, IL, ZDA, Por. st. R-285, 1966.

[9] Leukhin A., Potekhin E., "Binary sequences with minimum peak sidelobe level up to length 68", 2012.

[10] Leukhin A. N., Potekhin E. N., "Optimal peak sidelobe level sequences up to length 74", Proc. Eur. Radar Conf., 2013, str. 495-498.

[11] Leukhin A., Potekhin E. N., "Exhaustive search for optimal minimum peak sidelobe binary sequences up to length 80", Proc. Int. Conf. Sequences Appl., 2014, str. 157-169.

[12] Leukhin A., Potekhin E., "A Bernasconi model for constructing ground-state spin systems and optimal binary sequences", Journal of Physics: Conference series, vol. 613, st. 1, 2015.

[13] Leukhin A., Parsaev N., Bezrodnyi V., Kokovihina N., "The exhaustive search for optimum minimum peak sidelobe binary sequences", Bull. Russian Acad. Sci.: Phys., vol. 81, st. 5, 2017, str. 575-578.

[14] Levanon N., Mozeson E., Radar Signals, Hoboken, NJ, USA:Wiley, 2004.

[15] Lindner J., "Binary sequences up to length 40 with best possible autocorrelation function", Electronic Letters, vol. 11, st. 21, 1975, str. 507-507.

[16] Nasrabadi M. A., Bastani M. H., "A survey on the design of binary pulse compression codes with low autocorrelation" in Trends in Technologies, London, U.K.: IntechOpen, 2010.
[17] Nunn C. J., Coxson G. E., "Best-known autocorrelation peak sidelobe levels for binary codes of length 71 to 105", IEEE Transactions on Aerospace and Electronic Systems, vol. 44, st. 1, Jan. 2008, str. 392-395.
[18] Pott A., Finite Geometry and Character Theory, Berlin, Germany: Springer, 2006.
[19] Rushanan J. J., "Weil sequences: A family of binary sequences with good correlation properties", IEEE Int. Symp. Inf. Theory - Proc, 2006, str. 1648-1652.
[20] Turyn R. et al., "Sequences with small correlation" in Error Correcting Codes, Hoboken, NJ, USA: Wiley, 1968, str. 195-228.

PRILOGA A

A1 Implementacija algoritma z uporabo funkcije rand()

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <cmath>
#include <sstream>
#include <iomanip>
#include <chrono>

using namespace std;

int evaluate1D(int sequence[], int n) {
	int fitness = 0;
	int* CuB = new int[n];
	int P = 4;
	for (int u = 0; u < n; u++) {
		CuB[u] = 0;
	}
	for (int u = 0; u < n; u++) {
		for (int j = 0; j <= n - u - 1; j++) {
			CuB[u] += sequence[j] * sequence[j + u];
		}
		if (u > 0) {
			fitness += pow(abs(CuB[u]), P);
		}
	}
	delete[] CuB;
	return fitness;
}

void procedureSHC(int n, int F, int t, int hmin, int hmax, int G) {

	bool goalReached = false;

	cout << "Generate rand seq:" << endl;
	//generate a RANDOM binary sequence
	int* binarySeq = new int[n];
	for (int i = 0; i < n; i++) {
		int temp = rand() % 2 + 0;
		if (temp == 0) temp = -1;
		binarySeq[i] = temp;
		cout << binarySeq[i] << " ";
	}
	//set threshold
	int thresholdLeft = t;
	//define current best fit ,and globFit
	int bestFit = evaluate1D(binarySeq, n);
	//define globFit as bestFit
	int globFit = bestFit;
	//make binarySeqCopy
	int* binarySeqCopy = new int[n];
	for (int i = 0; i < n; i++) {
```

```
		binarySeqCopy[i] = binarySeq[i];
	}
	cout << "bestfit: " << bestFit << endl;
	cout << endl << "~~~~~~~~~~~~~~~~~~~" << endl << "Start of procedure:" <<
	endl << "~~~~~~~~~~~~~~~~~~~" << endl;
	int bestPSL = 999;
	int maxflips = 0;
	int biggestFlipPos = 0;

	//time
	chrono::steady_clock::time_point begin = chrono::steady_clock::now();

	//we repeat for 10^5 restarts
	int restarts = 0;
	int maxRestarts = pow(10, 5);
	do {
		//define initiate neighbours
		int** binarySeqNeighbours = new int*[n];
		for (int i = 0; i < n; i++)
			binarySeqNeighbours[i] = new int[n];
		//generate neighbours
		for (int i = 0; i < n; i++) {
			for (int j = 0; j < n; j++) {
				if (i == j)
					binarySeqNeighbours[i][j] = binarySeq[j] * -1;
				else
					binarySeqNeighbours[i][j] = binarySeq[j];
			}
		}
		//set flag
		bool flag = true;
		//define initiate CuB
		int** AACF = new int*[n];
		for (int i = 0; i < n; i++)
			AACF[i] = new int[n];
		//generate starting
		for (int i = 0; i < n; i++) {
			for (int u = 0; u < n; u++) {
				AACF[i][u] = 0;
			}
		}
		for (int i = 0; i < n; i++) {
			//generate
			int PSL = 0;
			for (int u = 0; u < n; u++) {
				for (int j = 0; j <= n - u - 1; j++) {
					AACF[i][u] += binarySeqNeighbours[i][j] *
					binarySeqNeighbours[i][j + u];
				}
				//after calculation check PSL
				if (u > 0) { //for 0<u<n
					if (abs(AACF[i][u]) > PSL) {
						PSL = abs(AACF[i][u]);
					}
				}
			}
			if (PSL < bestPSL)bestPSL = PSL;//testetste
			//Compare with desired PSL
			if (PSL == G) {
				cout << "Perfect PSL reached with: " << PSL << endl;
```

```
                        prepareHex(binarySeqNeighbours[i], n);
                        for (int k = 0; k < n; k++) {
                                cout << binarySeqNeighbours[i][k] << " ";
                        }
                        goalReached = true;
                }
                if (goalReached) break;
                int newEval = evaluate1D(binarySeqNeighbours[i], n);
                if (newEval < bestFit) {
                        //bestFit<-F(X)
                        bestFit = newEval;
                        //binSeq<-X
                        for (int j = 0; j < n; j++) {
                                binarySeq[j] = binarySeqNeighbours[i][j];
                        }
                        //flag<-false
                        flag = false;
                }
        }
        if (goalReached) break;
        if (flag) {
                if (bestFit < globFit) {
                        //set globFit
                        globFit = bestFit;
                        //make copy of binarySeq
                        for (int i = 0; i < n; i++) {
                                binarySeqCopy[i] = binarySeq[i];
                        }
                        //set threshold
                        thresholdLeft = t;
                }
                else {
                        thresholdLeft--;
                        if (thresholdLeft > 0) {
                                //fill binSeq from copy
                                for (int i = 0; i < n; i++) {
                                        binarySeq[i] = binarySeqCopy[i];
                                }
                                //choose random h from h within [hmin,hmax]
                                int h = rand() % hmax + hmin
                                if (h > maxflips) maxflips = h;//testestest
                                //flip h random bits in binSeq
                                bool* flipPosPool = new bool[n];
                                for (int i = 0; i < n; i++)
                                        flipPosPool[i] = false;
                                int weGoodCounter = 0;
                                while (weGoodCounter<h) {
                                        int flipPos = rand() % n;
                                        if (flipPosPool[flipPos] == false) {
                                                flipPosPool[flipPos] = true;
                                                weGoodCounter++;
                                        }
                                        if (flipPos > biggestFlipPos)
                                                biggestFlipPos = flipPos;
                                }
                                for (int i = 0; i < n; i++) {
                                        if (flipPosPool[i] == true) {
                                                binarySeq[i] = binarySeq[i] * -1;
                                        }
                                }
```

```
                                        //calculate new bestFit
                                        bestFit = evaluate1D(binarySeq, n);
                                        //test output
                                        //deallocate
                                        delete[] flipPosPool;
                                }
                                else {//threshold reached
                                        //generate new binarySeq
                                        for (int i = 0; i < n; i++) {
                                                int temp = rand() % 2 + 0;
                                                if (temp == 0) temp = -1;
                                                binarySeq[i] = temp;
                                        }
                                        //set threshold
                                        thresholdLeft = t;
                                        //evaluate new bestFit and assign bestGlobFit
                                        bestFit = evaluate1D(binarySeq, n);
                                        //globFit
                                        globFit = bestFit;
                                        //make new binarySeqCopy
                                        for (int i = 0; i < n; i++) {
                                                binarySeqCopy[i] = binarySeq[i];
                                        }
                                }
                        }
                }
                //deallocate
                for (int i = 0; i < n; i++) {
                        delete[] AACF[i];
                }
                delete[] AACF;
                for (int i = 0; i < n; i++) {
                        delete[] binarySeqNeighbours[i];
                }
                delete[] binarySeqNeighbours;

                //keep count of restarts
                restarts++;
        } while (restarts < maxRestarts);
        if(!goalReached)
                prepareHex(binarySeqCopy, n);
        //deallocate
        delete[] binarySeqCopy;
        delete[] binarySeq;
        //max restarts reched
        cout << endl;
        cout << "Max restarts reached: " << restarts << "/" << maxRestarts << endl;
        cout << "bestPSL: " << bestPSL << endl;
        cout << "maxFlips: " << maxflips << endl;
        cout << "biggestFlipPos: " << biggestFlipPos << endl;
        //time
        chrono::steady_clock::time_point end = chrono::steady_clock::now();
        cout << "Elapsed time: " << chrono::duration_cast<chrono::seconds>(end-
        begin).count() << endl;
}


void prepareHex(int sequence[], int n) {
        int* binaryForm = new int[n];
        stringstream binaryStream;
```

```
	for (int i = 0; i < n; i++) {
		if (sequence[i] == -1) binaryForm[i] = 0;
		else binaryForm[i] = 1;

		binaryStream << binaryForm[i];
	}
	string SBF = binaryStream.str();
	int diff = SBF.length() % 8;
	int mod = 8 - diff;
	int padd = SBF.length() + mod;
	stringstream paddedStream;
	if (diff != 0) paddedStream << setw(padd) << setfill('0') << SBF;
	string padded = paddedStream.str();
	stringstream ssss;
	int base = 2;
	for (int i = 0; i < padded.length(); i += 8) {
		string eightBit = padded.substr(i, 8);
		char* pointer = NULL;
		const char* temp = eightBit.c_str();
		ssss << hex << strtol(temp, &pointer, base);
	}
	cout << ssss.str() << endl;
	delete[] binaryForm;
}
int main() {
	//rand seed
	srand(time(NULL));

	//variables
	int n = 123; // size(we put 123)
	int t = pow(10, 3); // threshold(we fix it on 10 ^ 3)
	int Fchoice = 0;
	int hmin = 1; // hmin(we fix it on 1)
	int hmax = (int)ceil(sqrt(n)); // hmax(best as sqrt(n))
	int G = 7; // G(desired PSL we put 7, old=8)

	//procedure call:
	procedureSHC(n, Fchoice, t, hmin, hmax, G);

	return 0;
}
```

A2 Funkcija procedureSHC z uporabo generatorja naključnih števil Marsenne Twister

```
void procedureSHC(int n, int F, int t, int hmin, int hmax, int G) {

	bool goalReached = false;

	random_device rd;  //Will be used to obtain a seed for the random number
	engine
	mt19937 gen(rd()); //Standard mersenne_twister_engine seeded with rd()
	uniform_int_distribution<> distrib(0, 1);
	uniform_int_distribution<> distrib2(hmin, hmax);
	uniform_int_distribution<> distrib3(0, n-1);

	cout << "Generate rand seq:" << endl;
	//generate a RANDOM binary sequence
	int* binarySeq = new int[n];
	for (int i = 0; i < n; i++) {
		int temp = distrib(gen);
		if (temp == 0) temp = -1;
		binarySeq[i] = temp;
		cout << binarySeq[i] << " ";
	}
	//set threshold
	int thresholdLeft = t;
	//define current best fit ,and globFit
	int bestFit = evaluate1D(binarySeq, n);
	//define globFit as bestFit
	int globFit = bestFit;
	//make binarySeqCopy
	int* binarySeqCopy = new int[n];
	for (int i = 0; i < n; i++) {
		binarySeqCopy[i] = binarySeq[i];
	}
	cout << "bestfit: " << bestFit << endl;
	cout << endl << "~~~~~~~~~~~~~~~~~~~~" << endl << "Start of procedure:" <<
	endl << "~~~~~~~~~~~~~~~~~~~~" << endl;
	int bestPSL = 999;
	int maxflips = 0;
	int biggestFlipPos = 0;

	//time
	chrono::steady_clock::time_point begin = chrono::steady_clock::now();

	//we repeat for 10^5 restarts
	int restarts = 0;
	int maxRestarts = pow(10, 5);
	do {
		//define initiate neighbours
		int** binarySeqNeighbours = new int* [n];
		for (int i = 0; i < n; i++)
			binarySeqNeighbours[i] = new int[n];
		//generate neighbours
		for (int i = 0; i < n; i++) {
			for (int j = 0; j < n; j++) {
				if (i == j)
```

```
                        binarySeqNeighbours[i][j] = binarySeq[j] * -1;
                else
                        binarySeqNeighbours[i][j] = binarySeq[j];
        }
}
//set flag
bool flag = true;
//define initiate CuB
int** AACF = new int* [n];
for (int i = 0; i < n; i++)
        AACF[i] = new int[n];
//generate starting
for (int i = 0; i < n; i++) {
        for (int u = 0; u < n; u++) {
                AACF[i][u] = 0;
        }
}
for (int i = 0; i < n; i++) {
        //generate
        int PSL = 0;
        for (int u = 1; u < n; u++) {
                for (int j = 0; j <= n - u - 1; j++) {
                        AACF[i][u] += binarySeqNeighbours[i][j] *
                        binarySeqNeighbours[i][j + u];
                }
                //after calculation check PSL
                if (abs(AACF[i][u]) > PSL) {
                        PSL = abs(AACF[i][u]);
                }
        }
        if (PSL < bestPSL)bestPSL = PSL;
        //Compare with desired PSL
        if (PSL == G) {
                cout << "Perfect PSL reached with: " << PSL << endl;
                prepareHex(binarySeqNeighbours[i], n);
                getMeritFactor(AACF[i], n);
                for (int k = 0; k < n; k++) {
                        cout << binarySeqNeighbours[i][k] << " ";
                }
                goalReached = true;
        }
        if (goalReached) break;

        int newEval = evaluate1D(binarySeqNeighbours[i], n);
        if (newEval < bestFit) {
                //bestFit<-F(X)
                bestFit = newEval;
                //binSeq<-X
                for (int j = 0; j < n; j++) {
                        binarySeq[j] = binarySeqNeighbours[i][j];
                }
                //flag<-false
                flag = false;
        }
}
if (goalReached) break;
//after for
if (flag) {
        if (bestFit < globFit) {
                //set globFit
```

```
                globFit = bestFit;
                //make copy of binarySeq
                for (int i = 0; i < n; i++) {
                        binarySeqCopy[i] = binarySeq[i];
                }
                //set threshold
                thresholdLeft = t;
        }
        else {
                thresholdLeft--;
                if (thresholdLeft > 0) {
                        //fill binSeq from copy
                        for (int i = 0; i < n; i++) {
                                binarySeq[i] = binarySeqCopy[i];
                        }
                        //choose random h from h within [hmin,hmax]
                        int h = distrib2(gen);
                        if (h > maxflips) maxflips = h;//testestest
                        //flip h random bits in binSeq
                        bool* flipPosPool = new bool[n];
                        for (int i = 0; i < n; i++)
                                flipPosPool[i] = false;
                        int weGoodCounter = 0;
                        while (weGoodCounter < h) {
                                //mt19937 gen(rd());
                                int flipPos = distrib3(gen);
                                if (flipPosPool[flipPos] == false) {
                                        flipPosPool[flipPos] = true;
                                        binarySeq[flipPos] =
                                        binarySeq[flipPos] * -1;
                                        weGoodCounter++;
                                }
                                if (flipPos > biggestFlipPos)
                                        biggestFlipPos = flipPos;
                        }
                        //calculate new bestFit
                        bestFit = evaluate1D(binarySeq, n);
                        //deallocate
                        delete[] flipPosPool;
                }
                else {//threshold reached
                        //generate new binarySeq
                        for (int i = 0; i < n; i++) {
                                int temp = distrib(gen);
                                if (temp == 0) temp = -1;
                                binarySeq[i] = temp;
                        }
                        //set threshold
                        thresholdLeft = t;
                        //evaluate new bestFit and assign bestGlobFit
                        bestFit = evaluate1D(binarySeq, n);
                        //globFit
                        globFit = bestFit;
                        //make new binarySeqCopy
                        for (int i = 0; i < n; i++) {
                                binarySeqCopy[i] = binarySeq[i];
                        }
                }
        }
}
```

```
			//deallocate
			for (int i = 0; i < n; i++) {
				delete[] AACF[i];
			}
			delete[] AACF;
			for (int i = 0; i < n; i++) {
				delete[] binarySeqNeighbours[i];
			}
			delete[] binarySeqNeighbours;

			//keep count of restarts
			restarts++;
		} while (restarts < maxRestarts);//we fix it to 10^5, for testing purpose
		only 10
		if (!goalReached)
			prepareHex(binarySeqCopy, n);
		//deallocate
		delete[] binarySeqCopy;
		delete[] binarySeq;
		//max restarts reched
		cout << endl;
		cout << "Max restarts reached: " << restarts << "/" << maxRestarts << endl;
		cout << "bestPSL: " << bestPSL << endl;
		cout << "maxFlips: " << maxflips << endl;
		cout << "biggestFlipPos: " << biggestFlipPos << endl;
		//time
		chrono::steady_clock::time_point end = chrono::steady_clock::now();
		cout << "Elapsed time: " << chrono::duration_cast<chrono::seconds>(end -
		begin).count() << endl;
}
```

PRILOGA B

B1 Rezultati iskanja binarnih sekvenc za dolžino 123 v 50 zagonih

n	PSL	$binarySeq$ (Hexadecimal)	čas (s)
123	7	4931fbd4c15463432caf53f8b3f417b	654
123	7	7728695e40c82f3349a82d9ab978c5	356
123	8	4bdb81094146a89ca66f83693e8658e	669
123	8	34bfdbd8149d349f06bc6275d98a754	545
123	7	05ad06d2ea77d633049b7e266a88e	46
123	8	6069ec9ea18ad593d1e68194c8867	669
123	7	7736ca62d92e7eeb23af584a1c5bc0	201
123	7	3b24df48675631416a16058dd79382	89
123	8	74df788613af5736be5a33e88c2d236	669
123	8	5ea59bec95ec6d2c3eb8e84199baa7	669
123	8	6ef15789c25987ae322b606f482123	669
123	8	2532d87a7bfef528a3ee2c8dec45f32	669
123	8	15d8eecf08fdd408132c76b86a92968	670
123	8	5f117893e3d2633c37d688d95245	668
123	7	3e3c816178d3eefa9d733416f457621	469
123	8	64ab5146e61bc1c7fb68382884ac4c	668
123	8	161e9aa8840383799b533f846dd29c7	672
123	8	23fc510eb6d1a1460d278daee6a6e	671
123	7	7417a459c6affb18648e85ac456d	453
123	8	15ad6f8a85fc39e5c590cc14493532	668
123	7	011899fcb2795c2570a0bcb12e6adf	343
123	8	2f2ee591d39835bd986d72b960b87	668
123	8	02cbf6e8540962ce313e62b69ce8ee	668
123	7	7e2838323fc5526cc66957b4f6b7ad1	615

123	8	6f625f226e619d7cba54721e1febea	668
123	8	3236832687fd92835d4ebdea7e73958	669
123	8	2bc502823b380bb6cb5ad641d13e63e	671
123	8	52624862146abeb8922d79c2c7f1b2	668
123	8	73b88b8be7c9b583252847cb87d52	669
123	7	7ad416664425b75dd60c319ea12e5	181
123	8	38eabeba57d2b7bf6d9b93672358	671
123	8	2df3cb16f59c1504ec84ee8391c962a	668
123	8	5dceb9d87fa2f82af3e93d6c48b36b	669
123	7	294696dfdd528a3ee6813e098cc3e1c	650
123	7	427f0c33559612cb2152b9f3f6c685	475
123	8	2d2cf86df1dcc46147ebc028d51262	669
123	8	0756a34f38262b2cb3120c866f4bd	669
123	8	50f521d655c8dbeaf67692247deec6	669
123	8	6728db25d3a729fd8fef488d15fdd	670
123	7	613db247242c6395a2b47d197c4807	669
123	7	03b83abb8d333448ea75fb692b5dbc0	1
123	7	3ff1afe3d6aa2a5e465239132f7c924	62
123	8	555863614fbc97a43d0f9fed89deed1	669
123	8	3959bcf1a19adc4ab4371da22eca	670
123	8	5f77ed48558f81ade423d159a2498e1	669
123	8	3c813dcd96c4f73f8fcf2a754e8ca9	668
123	8	4d0d336b8bf537633e53fdaa97fa	669
123	8	3dfc685acdee18aed34779bb056c055	669
123	8	53735aeb01171c7ed5483792c0b46c	674
123	8	371fefbcb90e1abd53459b94c67f47	670