Slovenská technická univerzita v Bratislave

Fakulta informatiky a informačných technológií

Ilkovičova 2, 842 16 Bratislava

FIIT-5220-64422

Filip Pakan

Evolučné generovanie samoopravných kódov

Diplomová práca

Vedúci práce:	prof. RNDr. Jiří Pospíchal, DrSc.
Študijný program:	Softvérové inžinierstvo
Študijný odbor:	9.2.5. Softvérové inžinierstvo
Miesto vypracovania:	Ústav aplikovanej informatiky, FIIT STU, Bratislava
Dátum:	Máj 2014

Anotácia

Slovenská technická univerzita v Bratislave

FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Študijný program: Softvérové inžinierstvo

Autor: Filip Pakan

Diplomová práca: Evolučné generovanie samoopravných kódov

Vedúci práce: prof. RNDr. Jiří Pospíchal, DrSc.

Máj 2014

V dnešnom poprepájanom svete sa neustále stretávame s prenosom informácií po sieti. Vplyvom elektromagnetického rušenia a tepelného šumu nie je správnosť prenosu informácií garantovaná. Poškodenú informáciu dokážeme opraviť, ak ju pred odoslaním zakódujeme samoopravným kódom.

Rozsiahlejšie samoopravné kódy umožňujú zakódovať väčší počet zdrojových správ a zefektívniť tak komunikáciu. Nanešťastie generovanie väčších optimálnych samoopravných kódov je ekvivalentné NP-úplnému problému hľadania maximálnej kliky v grafe. Vyčerpávajúce hľadanie riešenia nie je možné pre exponenciálnu zložitosť problému. Jedným z možných riešení je použitie stochastických optimalizačných algoritmov, ktoré sú inšpirované biologickou evolúciou. Evolúcia je proces, kde sa po uplynutí mnohých generácií môže vyvinúť optimálne riešenie.

Práca analyzuje súčasne používané evolučné metódy a riešenia pre generovanie samoopravných kódov a hľadanie maximálnej kliky v grafe. Cieľom práce je navrhnúť vhodnú evolučnú heuristiku alebo vylepšiť existujúcu metódu na generovanie samoopravných kódov a porovnať navrhnuté riešenie s vybranými metódami.

Annotation

Slovak University of Technology in Bratislava

FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES

Degree Course: Software Engineering

Author: Filip Pakan

Master's Thesis: Evolutionary Generation of Error Correcting Codes

Supervisor: Prof. Dr. Jiří Pospíchal

May 2014

In today's interconnected world we experience a huge amount of data being transmitted over the network. However, sometimes errors may occur in transmitted data due to electromagnetic interference and thermal noise. Encoding the message in a redundant way by using an error-correcting code gives receivers the ability to recover corrupted data.

Larger error-correcting codes allow more messages to be encoded and hence improve communication efficiency. Unfortunately, the generation of error-correcting codes is equivalent to NP-complete problem of finding maximum clique in a graph. An exhaustive search for a solution is not feasible due to the exponential complexity of the problem. One possibility is to employ stochastic optimization algorithms inspired by biological evolution. Evolution is a process where optimal solution may emerge after many generations.

This work analyzes recent evolutionary approaches to the generation of optimal error-correcting codes and finding the maximum clique in a graph. The goal of this project is to design an evolutionary heuristic or to improve existing method for the generation of error-correcting codes and to compare designed solution to the selected methods.

Čestné prehlásenie

Čestne prehlasujem, že predloženú záverečnú prácu som vypracoval samostatne pod vedením prof. RNDr. Jiřího Pospíchala, DrSc. a uviedol v zozname literatúry všetky použité odborné zdroje.

.....

Filip Pakan

V Bratislave, Máj 2014

Poďakovanie

Na tomto mieste by som sa chcel poďakovať prof. RNDr. Jiřímu Pospíchalovi, DrSc. za ochotu, cenné rady a podnety, ktoré významnou mierou prispeli ku kvalite záverečnej práce.

Osobitne by som sa chcel poďakovať rodičom, ktorí mi vytvorili ideálne podmienky pre dôsledné vypracovanie záverečnej práce.

Obsah

Zoznam obrázkov

Zoznam tabuliek

KAPITOLA 1

Úvod

„Kto chce správne spoznávať, musí najprv správnym spôsobom pochybovať.“

~ Aristoteles (384 pred Kr. - 322 pred Kr.)

S prenosom informácií sa v dnešnom prepojenom svete stretávame neustále, a to najmä v počítačových a telekomunikačných sieťach. Informácie sa fyzicky prenášajú po komunikačných kanáloch ako napríklad medené drôty, optické vlákna alebo elektromagnetické vlny pri bezdrôtovom prenose. Prenos informácií je tvorený prúdom bitov, ktoré predstavujú základné jednotky informácie. Bity sú fyzicky reprezentované elektromagnetickým signálom ako elektrické napätie, rádiové vlny, mikrovlny alebo infračervený signál.

1.1 Motivácia

Všetci účastníci prenosu majú záujem na tom, aby prenesené informácie boli korektné a nepoškodené. Prenosové kanály nám však takúto službu neposkytujú. Na prenos informácie cez prenosové médium negatívne vplývajú mnohé faktory. Najčastejšie to je elektromagnetické rušenie a tepelný šum. Tieto faktory spôsobujú, že správnosť prenosu informácie nie je nikdy garantovaná. Pri prenose teda dochádza k chybám v jednotlivých bitoch prenášanej informácie.

Pri získaní poškodenej informácie by jednoduchým riešením bola žiadosť o opakovanie prenosu. Takýto prístup však nie je vždy možný. Okrem toho, že zaťažuje prenosové linky, nie je vždy aplikovateľný. Napríklad môže nastať situácia, že vysielač sa z komunikácie už odpojil a požiadať ho o opakovanie prenosu nie je viac možné. Eventuálne veľkosť prenášaných dát je taká veľká, že je nemysliteľné celý prenos opakovať znova.

Prípadne si predstavme služby ako prenos hlasu alebo šírenie televízneho vysielania cez IP protokol (VoIP a IPTV). Pri takýchto službách je nereálne vyžiadať opakovanie prenosu, pretože asi nikto by nechcel čakať na ďalší záber v televízii, kým sa opakovaná informácia opäť doručí. Tieto služby skrátka musia fungovať v reálnom čase a žiadne oneskorenie nie je prípustné. Zároveň však nechceme, aby obraz televízie vypadával alebo v telefonickom hovore chýbali fragmenty zvuku.

Jedným z možných riešení je prenášanú informáciu zakódovať samoopravným kódom. To spočíva v pridaní niekoľkých redundantných bitov k prenášanej správe. Samoopravné kódy umožňujú kontrolovať zmeny, ku ktorým došlo v priebehu prenosu informácie cez prenosový kanál. Zabezpečujú spoľahlivý prenos digitálnych dát cez nespoľahlivé komunikačné kanály. Ide o rovnomerné blokové kódy, ktoré sú schopné detegovať a dokonca aj opravovať chyby.

Vďaka detekcii chýb bude príjemca schopný zistiť, či v prijatom slove vznikli pri prenose chyby alebo nie. Opravenie chýb umožní príjemcovi zrekonštruovať najpravdepodobnejšiu vyslanú informáciu.

1.2 Pomenovanie problému

Rozsiahlejšie samoopravné kódy zväčšujú maximálnu veľkosť prenositeľných správ, čo zvyšuje efektívnosť komunikácie. Nanešťastie generovanie optimálnych samoopravných kódov je ekvivalentné NP-úplnému problému hľadania

maximálneho kompletného podgrafu v neorientovanom grafe, taktiež nazývaného aj klika grafu.

Jednoduché vyčerpávajúce hľadanie rozsiahlejších samoopravných kódov nie je možné kvôli exponenciálnej zložitosti problému. To znamená, že veľkosť prehľadávaného priestoru rastie exponenciálne vzhľadom na celkový počet kódových slov. Prístup pre generovanie optimálnych samoopravných kódov, ktorý sa skúma v posledných rokoch, je založený na evolučných algoritmoch. Problémom však ostáva nájdenie vhodnej evolučnej metódy pre generovanie optimálnych samoopravných kódov.

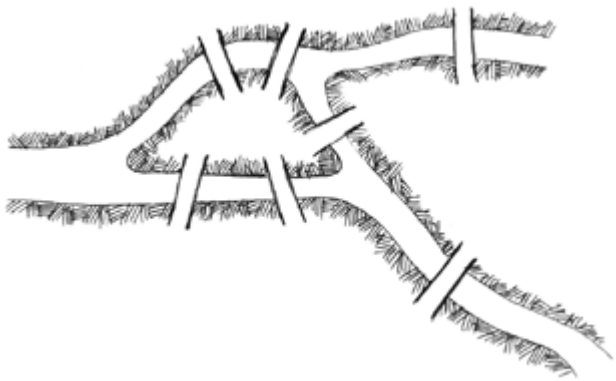
Kapitola 2

Teória grafov

„Ak ľudia neveria, že matematika je jednoduchá, potom si neuvedomujú, aký komplikovaný je život.“

~ John von Neumann (1903 - 1957)

Za zakladateľa teórie grafov možno považovať Leonharda Eulera, ktorý sa snažil vyriešiť problém mostov v Kráľovci [5]. Mesto malo vybudovaných 7 mostov, ktoré spájali brehy rieky a 2 ostrovy (Obr. 1). Úlohou bolo nájsť takú prechádzku mestom, pri ktorej sa každý most prejde práve raz a nakoniec sa vrátime na počiatočnú pozíciu. Euler v tej dobe dokázal, že pre danú úlohu je tento problém neriešiteľný.



Obr. 1. Mosty v Kráľovci [17]

2.1 Základné pojmy

Graf G v matematike pozostáva z množiny vrcholov V a z množiny hrán E spájajúcich tieto vrcholy. Formálne

$$G = (V, E)$$

V teórii grafov poznáme dva typy hrán:

- Orientované
- Neorientované

Ak vieme o hrane e povedať, v ktorom vrchole začína a v ktorom končí, nazýva sa orientovaná. Hranu e z množiny E nazývame neorientovaná, ak nie je orientovaná.

Podľa toho, či graf obsahuje orientované alebo neorientované hrany, môže byť

- orientovaný, ak obsahuje iba orientované hrany
- neorientovaný, ak obsahuje iba neorientované hrany
- zmiešaný, ak obsahuje obidva druhy hrán

Ďalej platí, že každá hrana v grafe spája práve dva vrcholy. Nech hrana e spája vrcholy u a v. Hovoríme, že hrana e je incidentná s vrcholmi u a v. O vrcholoch u a v hovoríme, že sú susedné. Formálne zapisujeme

- $e = \{u, v\}$ pre neorientovaný graf
- $e = (u, v)$ pre orientovaný graf

Počet hrán, s ktorými je vrchol v incidentný, nazývame stupeň vrcholu a označujeme $\deg(v)$. Vzhľadom na to, že každá hrana spája práve 2 vrcholy, platí medzi počtom hrán a stupňami vrcholov nasledovný vzťah:

$$\sum_{v \in V} \deg(v) = 2 \cdot |E|$$

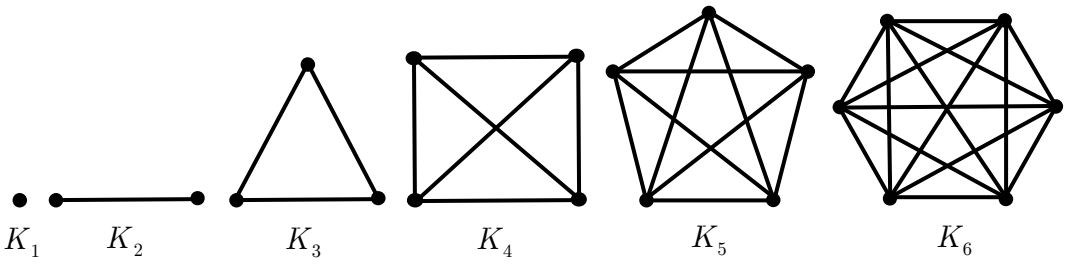
Vysvetlenie je priamočiare. Každá hrana pridáva jeden stupeň dvom vrcholom, ktoré spája. Preto suma stupňov všetkých vrcholov sa rovná dvojnásobku počtu hrán v grafe.

Súvislý graf je taký graf, v ktorom sa po hranách grafu vieme dostať z ľubovoľne zvoleného vrcholu do ľubovoľného iného vrcholu.

Komponent grafu je maximálna množina vrcholov s vlastnosťou, že každý jej vrchol je dosiahnuteľný z každého iného vrcholu v komponente. Inak vysvetlené, je to maximálny súvislý podgraf.

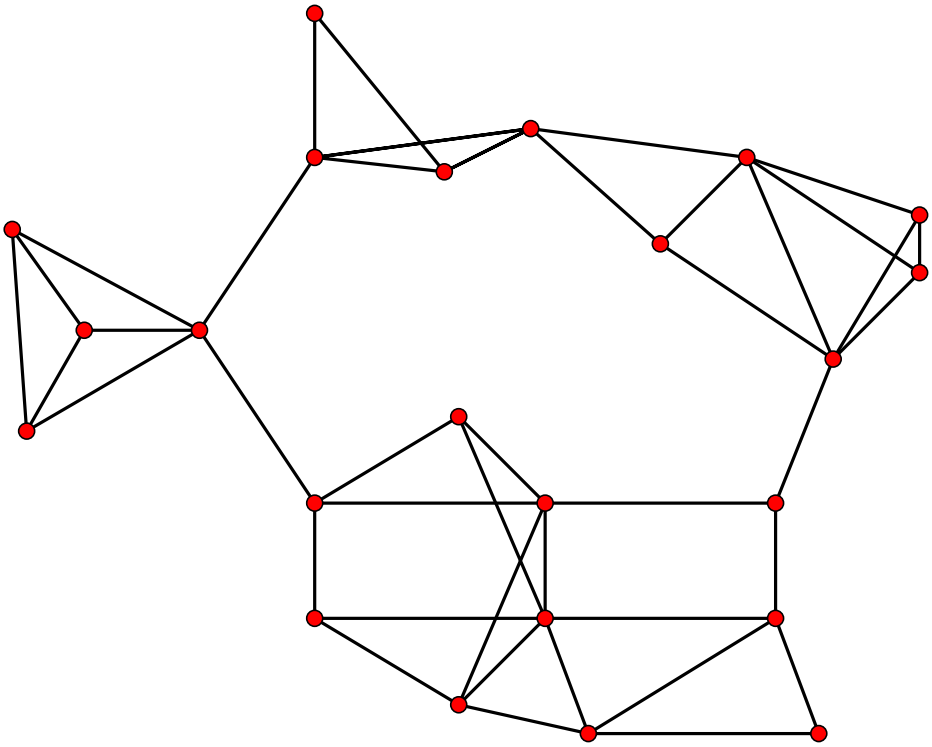
Neorientovaný graf, v ktorom je každý vrchol spojený hranou s každým iným vrcholom sa nazýva kompletný graf (Obr. 2). Označujeme ho K_n, kde n je počet vrcholov kompletného grafu. Počet hrán kompletného grafu je rovný počtu všetkých možností, ako vybrať 2 vrcholy z n, t.j.

$$\binom{n}{2} = \frac{n \cdot (n-1)}{2}$$



Obr. 2. Ukážky kompletných grafov

Klika v neorientovanom grafe G je taká podmnožina vrcholov grafu, v ktorej sú každé dva vrcholy spojené hranou. Ide teda o kompletný podgraf grafu G. Maximálna klika grafu je maximálny kompletný podgraf grafu G, ktorý je možné vytvoriť. Klikové číslo grafu udáva počet vrcholov v maximálnej klike grafu (Obr. 3).



Obr. 3. Graf s klikovým číslom 4

2.2 Reprezentácia grafov

Existuje viacero spôsobov reprezentácie grafov. Rôzne reprezentácie významným spôsobom ovplyvňujú pamäťovú a časovú zložitosť základných grafových operácií.

2.2.1 Matica incidencie

Uvažujme graf $G = (V, E)$ s n vrcholmi a m hranami. Maticou incidencie grafu G nazveme maticu $A = (a_{ij})$ typu $n \times m$, kde

$$a_{ij} = \begin{cases} 1 & \text{keď hrana } e_j \text{ je incidentná s vrcholom } v_i \\ 0 & \text{inak} \end{cases}$$

Výhodou matice incidencie je, že môže byť použitá na reprezentovanie viacnásobných hrán a slučiek. Matica incidencie v každom stĺpci obsahuje iba dve jednotky, zvyšok sú nuly.

2.2.2 Matica susednosti

Uvažujme opäť graf ako v predošlom prípade. Matica susednosti A grafu G je taká matica typu $n \times n$, ktorá obsahuje jednotky pre tie vrcholy, ktoré sú spojené hranou. Formálne $A = (a_{ij})$ kde

$$a_{ij} = \begin{cases} 1 & \text{keď existuje hrana, ktorá spája vrcholy } v_i \text{ a } v_j \\ 0 & \text{inak} \end{cases}$$

Matica susednosti pre neorientovaný graf je symetrická podľa hlavnej diagonály. Pre graf bez slučiek sú prvky na hlavnej diagonále rovné 0. Pre grafy s malým počtom hrán je matica susednosti riedka.

2.2.3 Zoznam vrcholov a zoznam hrán

V tomto prípade sa graf reprezentuje vymenovaním všetkých vrcholov a hrán. Každá hrana je reprezentovaná ako dvojica vrcholov, ktoré daná hrana spája. Takéto vrcholy sa nazývajú susedné.

Jedná sa o pamäťovo úspornú reprezentáciu, no základné grafové operácie majú vyššiu zložitosť.

2.2.4 Zoznam susedov

Teraz sa graf reprezentuje namiesto zoznamu hrán pomocou zoznamu vrcholov, pričom každý vrchol ďalej obsahuje zoznam všetkých svojich susedov.

V prípade neorientovaného grafu dochádza k redundancii údajov, pretože ak vrchol v_i má v zozname svojich susedov vrchol v_j, potom aj vrchol v_j má v zozname svojich susedov vrchol v_i.

2.2.5 Porovnanie reprezentácií grafov

Predpokladajme, že máme daný graf G s n vrcholmi a m hranami. Porovnanie zložitosti pri základných operáciách je uvedené v nasledovnej tabuľke (Tab. 1).

Tab. 1. Porovnanie výpočtovej zložitosti operácií nad grafmi

	Matica incidencie	**Matica susednosti**	**Zoznam vrcholov a zoznam hrán**	**Zoznam susedov**
Pamäťové nároky	$O(mn)$	$O(n^2)$	$O(m+n)$	$O(m+n)$
Sú dva vrcholy susedné?	$O(m)$	$O(1)$	$O(m)$	$O(n)$
Susedné vrcholy daného vrcholu	$O(mn)$	$O(n)$	$O(m)$	$O(n)$
Pridanie hrany	$O(mn)$	$O(1)$	$O(1)$	$O(1)$
Odstránenie hrany	$O(mn)$	$O(1)$	$O(m)$	$O(m)$

Pridanie vrcholu	$O(mn)$	$O(n^2)$	$O(1)$	$O(1)$
Odstránenie vrcholu	$O(mn)$	$O(n^2)$	$O(m)$	$O(m)$

2.3 Aplikácie

Grafy sú ideálne na modelovanie rôznych vzťahov a dynamických procesov vo fyzických, biologických, sociálnych a informačných systémoch. Množstvo praktických problémov môže byť reprezentovaných pomocou grafov. Uvediem niekoľko príkladov:

- Informačné technológie
    - Počítačové a komunikačné siete
        - Hľadanie najkratšej cesty pri smerovaní v sieťach
        - Maximálny tok v sieťach
        - Prideľovanie frekvencií v mobilných sieťach
    - Návrh plošných tlačených spojov
        - Návrh logických obvodov bez kríženia
    - Teória kódovania
        - Prefixové kódy
        - Huffmanovo kódovanie
    - Plánovanie udalostí
        - Paralelné spracovanie
    - Web
        - Prepojenie dokumentov na webe

        - Prepojenie ľudí na sociálnych sieťach
- Prírodné vedy
    - Reprezentácia molekúl v chémii
        - Odlíšenie izomérov
- Antropológia
    - Znázornenie príbuzenských vzťahov

Kapitola 3

Teória kódovania

„Informácia je rozlíšenie neistoty.“

~ Claude Shannon (1916 - 2001)

Teória kódovania študuje vlastnosti kódov a ich použitie pre konkrétne aplikácie. Pod kódom sa rozumie pravidlo na zmenu reprezentácie informácie. Kódovanie je zobrazenie abecedy zdroja na abecedu kódera. Kódy sa používajú na kompresiu dát, opravu chýb ako aj v kryptografii.

Štúdiom kódov sa zaoberajú viaceré vedné disciplíny, menovite teória informácie, elektrotechnika, informatika a matematika. Cieľom je navrhnúť efektívne a spoľahlivé metódy na prenos dát. Hlavné problémy, ktoré treba riešiť v teórii kódovania, je možné zhrnúť do troch bodov:

1. Efektívnosť kódovania
2. Detekcia chýb
3. Oprava chýb

Na základe vyššie uvedených bodov možno štúdium kódov rozdeliť na dve oblasti:

a) Nerovnomerné kódy

b) Rovnomerné (blokové) kódy

3.1 Nerovnomerné kódy

Nerovnomerné kódy tvoria podstatu štúdia efektívnosti kódovania. Kódové slová nerovnomerných kódov majú rôznu dĺžku. Ich cieľom je komprimovať zdrojové dáta za účelom efektívneho prenosu. Základná myšlienka spočíva v priradení najkratšieho kódového slova najčastejšie sa vyskytujúcemu znaku v správe.

Využívajú sa 2 typy kompresie dát:

a) Stratová

b) Bezstratová

3.1.1 Stratová kompresia

Pri stratovej kompresii dochádza k nenávratnej strate informácií. Stratová kompresia sa používa najmä vtedy, keď je strata informácií akceptovateľná výmenou za väčší kompresný pomer. Stratová kompresia sa používa najčastejšie pri kódovaní audio a video obsahu. Vtedy človek často ani nedokáže spozorovať stratu informácií (JPEG, MP3 kompresia).

3.1.2 Bezstratová kompresia

Bezstratové kompresné algoritmy využívajú informačnú redundanciu na kompaktnejšiu reprezentáciu dát bez straty informácií. Bezstratová kompresia umožňuje presne zrekonštruovať pôvodne zakódovanú informáciu.

Správne navrhnuté kódovanie musí byť jednoznačne dekódovateľné. Jedným zo spôsobov, ako to zaručiť, je použitie prefixového kódu. Kódovanie je prefixové, ak žiadne kódové slovo nie je prefixom iného od neho rôzneho slova.

Existuje viacero algoritmov na hľadanie najkratších binárnych kódov. Najznámejšie je Huffmanovo kódovanie [14], ktoré zaručí nájdenie najkratšieho

prefixového binárneho kódu. Ďalšie známa technika na konštrukciu prefixového kódu je Shannon-Fano kódovanie [6, 28]. Na rozdiel od Huffmanovho kódovania nezaručí nájdenie najkratšieho kódu a je teda suboptimálne.

3.2 Rovnomerné kódy

Cieľom nerovnomerných kódov bolo dosiahnuť úspornejšiu reprezentáciu dát pre efektívnejší prenos. Abstrahovali sme však od dôležitého detailu, s ktorým sa pri prenose dát cez sieť stretávame. Správnosť prenosu nie je garantovaná a pri prenose nastávajú chyby.

Ako bolo vysvetlené v úvodnej časti, nie vždy je možné kvôli poškodenej informácii opakovať prenos. V takom prípade je dobré použiť kódy, ktoré dokážu chyby detegovať a opraviť. Takéto kódy nazývame samoopravné a s ich myšlienkou prišiel americký matematik Richard Hamming [11].

3.2.1 Detekcia chýb

Hlavný princíp detekcie chýb spočíva v tom, že navrhnutý kód musí byť riedky, t.j. jeho kódové slová nesmú vyplniť celý priestor B^n, ktorý označuje množinu všetkých binárnych slov dĺžky n. Inak totiž, ak by nastala chyba a niektorý bit informácie by sa zmenil, nedokázali by sme to detegovať, pretože aj chybové prijaté slovo by bolo regulárne kódové slovo.

3.2.2 Oprava chýb

Hlavný princíp opravy chýb spočíva v tom, že chybové prijaté slovo, ktoré nie je kódové slovo, sa aproximuje najbližším kódovým slovom. Ako metrika na porovnávanie blízkosti kódových slov sa uvažuje *Hammingova vzdialenosť*.

Aby sme mohli splniť vyššie uvedené požiadavky, musia mať všetky kódové slová rovnakú dĺžku. Takéto kódy nazývame rovnomerné alebo blokové.

3.2.3 Lineárne kódy

Lineárne kódy sú také blokové kódy, ktorých kódové slová sú uzavreté na operáciu sčítania, t.j. suma ľubovoľných dvoch kódových slov je opäť kódové slovo. Lineárny kód obsahuje vždy nulové slovo a je charakterizovaný troma vlastnosťami:

1. Dĺžka kódu
2. Dimenzia kódu
3. Minimálna vzdialenosť kódu

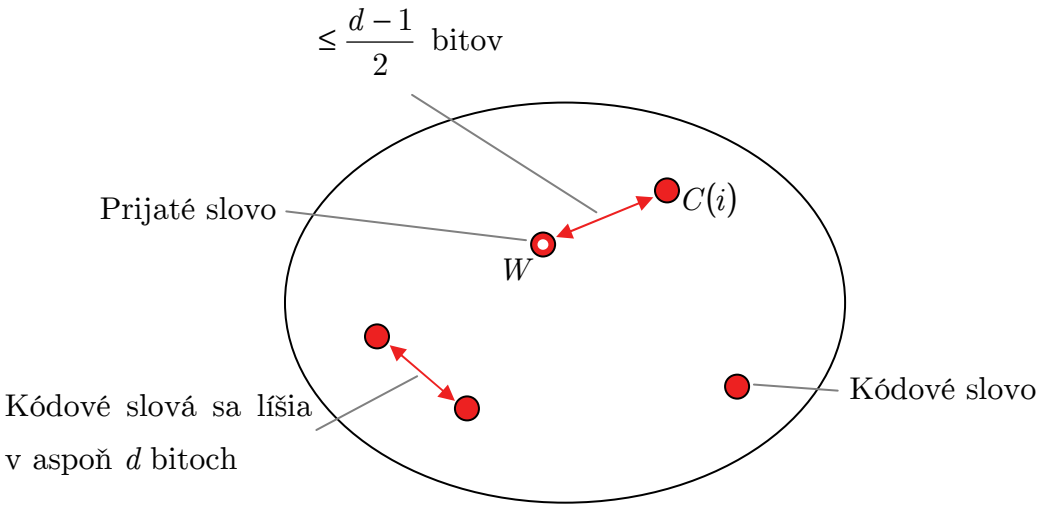
Lineárny kód s dĺžkou n, dimenziou k a minimálnou vzdialenosťou d sa označuje ako kód typu (n, k, d). Vyššie spomenuté vlastnosti sú definované nasledovne.

Minimálna vzdialenosť kódu

Nech $C \subset B^n$ je lineárny kód. Potom minimálna vzdialenosť d kódu C je najmenšia *Hammingova váha* nenulového kódového slova. Formálne

$$d = \min\{hwt(v) \mid v \in C, v \neq 0\}$$

Kód C s minimálnou vzdialenosťou d deteguje najviac $d-1$ a opravuje $\left\lfloor \frac{d-1}{2} \right\rfloor$ chýb (Obr. 4).



Obr. 4. Oprava prijatého chybového slova na najbližšie kódové slovo

Dimenzia kódu

Označme $S \subset B^n$ ako blokový kód dĺžky n (nemusí byť lineárny). Symbolom $\langle S \rangle$ označujeme množinu všetkých lineárnych kombinácií slov z množiny S a nazývame ju lineárny obal množiny S. Množina $\langle S \rangle$ je uzavretá na sčítanie a preto predstavuje lineárny kód.

Báza lineárneho kódu C je množina D pre ktorú platí, že je lineárne nezávislá a $\langle D \rangle = C$. Báza nie je jednoznačná, avšak všetky bázy majú ten istý počet slov. Počet slov v ľubovoľnej báze lineárneho kódu C nazývame dimenziou lineárneho kódu C a označujeme $\dim C$. Platí nasledovné

$$\dim C = |D| = k$$

Pre počet kódových slov platí $|C| = 2^k$. Počet redundantných bitov v kódových slovách je $n - k$.

Generujúca matica

Nech C je lineárny (n, k, d) kód. Generujúca matica je ľubovoľná matica typu $k \times n$, ktorá v riadkoch obsahuje slová bázy kódu C. Označujeme ju G_C.

Generujúca matica slúži na zakódovanie informačného slova. Kódovanie prebieha vynásobením informačného slova a generujúcej matice. Informačné slovo má dĺžku k, preto existuje 2^k správ, ktoré môžeme zakódovať pomocou 2^k kódových slov dĺžky n.

Výhodou lineárnych kódov je, že nám stačí uchovávať k riadkov generujúcej matice, namiesto 2^k kódových slov.

Perfektné kódy

Perfektný kód je taký kód, ktorý opravuje t chýb tak, že žiadne slovo sa nenachádza v rovnakej vzdialenosti od dvoch rôznych kódových slov, čo by viedlo k nejednoznačnosti pri dekódovaní.

Lineárny (n, k, d) kód C sa nazýva perfektný pre opravu t chýb, ak platí

$$|C| = \frac{2^n}{\sum_{i=0}^{t} \binom{n}{i}}$$

Triviálne perfektné kódy sú:

1. $t = 0 \Rightarrow |C| = 2^n \Rightarrow C = B^n$ je perfektný pre opravu 0 chýb
2. $t = n \Rightarrow |C| = 1 \Rightarrow C = \{o\}$ je perfektný pre opravu 1 chyby

Netriviálne perfektné kódy sú iba tieto lineárne kódy:

1. Hammingove $(2^r - 1,\ 2^r - r - 1,\ 3)$ kódy pre $r \geq 3$ perfektné pre opravu 1 chyby
2. Golayov $(23,\ 12,\ 7)$ kód perfektný pre opravu 3 chýb
3. Opakovacie kódy s nepárnou dĺžkou n, perfektné pre opravu $\frac{n-1}{2}$ chýb

3.2.4 Príklady samoopravných kódov

Medzi známe samoopravné kódy patria:

- Opakovacie kódy
- Hammingove kódy [11]
- Golayove kódy [9]
- Reed-Mullerove kódy [23, 24]
- Reed-Solomonove kódy [25]
- BCH kódy [2, 12]

3.3 Aplikácie

Po doručení chybnej informácie nie je vždy možné opakovať prenos. Najmä pri službách reálneho času ako šírenie televízie, videa a zvuku, prípadne pri prenose satelitných snímok Jupitera z kozmickej sondy Voyager. Vtedy je lepšie siahnuť po samoopravných kódoch.

Reed-Solomonove kódy našli široké uplatnenie v spotrebnej elektronike ako aj v medziplanetárnom komunikačnom priestore. Používajú sa najmä v elektronike ako sú CD, DVD, Blu-Ray disky, v technológiách na prenos dát ako DSL, WiMAX, v systémoch na vysielanie televízie DVB, ATSC, v diskových poliach ako napríklad RAID 6, v čiarových kódoch a QR kódoch.

V štandardoch šírenia digitálneho televízneho signálu DVB-T2, DVB-C2, DVB-S2 sa používa LDPC kódovanie (Low-Density Parity-Check) a BCH kódy na opravu chýb.

Hammingove kódy sa používajú na opravu NAND Flash pamäťových chýb. Poskytujú opravu jedného chybného bitu a detekciu dvoch chybných bitov. Preto sú vhodné na použitie pre spoľahlivejšie SLC NAND pamäte. Hustejšie

MLC NAND pamäťové moduly vyžadujú silnejší samoopravný kód, ako napríklad BCH alebo Reed-Solomonove kódy.

Teória kódovania nájde uplatnenie aj v kódovom multiplexe (CDMA), čo je v elektronike metóda prenosu viacerých digitálnych signálov prostredníctvom jediného zdieľaného média. Jednotlivé signály je možné rozlíšiť podľa toho, že každý z nich používa iné kódovanie.

Kapitola 4

Evolučné algoritmy

„Ak som videl ďalej, bolo to preto, že som stál na pleciach obrov.“

~ Isaac Newton (1643 - 1727)

Evolučné algoritmy patria medzi stochastické optimalizačné metódy, ktoré sú inšpirované biologickou evolúciou. Problematika evolučných algoritmov spadá do oblasti umelej inteligencie, presnejšie do oblasti výpočtovej umelej inteligencie.

Evolúcia je proces, kde v priebehu mnohých generácií môže dochádzať k optimálnym riešeniam. Neúspešné riešenia vymierajú, zatiaľ čo úspešné prežívajú a množia sa. Hybnou silou evolúcie je kríženie a mutácia. Evolučné algoritmy využívajú evolúciu, ako prostriedok na riešenie technických problémov.

Evolučné algoritmy sú obzvlášť vhodné na riešenie NP-úplných problémov, v ktorých naivné prehľadávanie stavového priestoru nie je možné aplikovať pre väčšie inštancie problémov, kvôli exponenciálnej zložitosti. Metrikou na porovnávanie evolučných algoritmov je počet vyhodnotení navrhnutých riešení v priebehu generácií potrebných na nájdenie optimálneho riešenia.

4.1 Základné stochastické optimalizačné algoritmy

Algoritmy uvedené v tejto časti patria medzi základné stochastické optimalizačné techniky, ktoré neobsahujú evolučné črty.

4.1.1 Slepý algoritmus

Slepý algoritmus (Blind Search) patrí medzi najtriviálnejšie stochastické optimalizačné metódy. Náhodne generuje riešenie z oblasti možných riešení a zapamätá si ho len vtedy, ak je lepšie ako najlepšie nájdené riešenie v predchádzajúcej histórii algoritmu. Každé riešenie je zostrojené úplne nezávisle od predchádzajúcich.

Tento triviálny algoritmus dokáže nájsť globálne optimum účelovej funkcie za predpokladu, že sa vykonáva nekonečne veľa krokov.

4.1.2 Horolezecký algoritmus

Horolezecký algoritmus (Hill Climbing) prehľadáva okolie určitého zvoleného riešenia, pričom z okolia si vyberie najlepšie riešenie, ktoré sa v ďalšom kroku použije ako stred novej oblasti, z ktorej sa vedie hľadanie. Algoritmus vráti najlepšie riešenie, ktoré sa vyskytlo v priebehu celej histórie algoritmu. Jedná sa o pomerne efektívny a robustný optimalizačný algoritmus, ktorý konverguje rýchlejšie ako slepé prehľadávanie.

Pri deterministickom generovaní okolia dôjde k uviaznutiu v prvom lokálnom optime. Preto okolie sa môže generovať aj stochasticky ako prevencia proti uviaznutiu. Iné riešenie je využitie multištartu, keď nastane uviaznutie. Ak sa kardinalita generovaného okolia zväčšuje do nekonečna, horolezecký algoritmus nájde globálne optimum.

Horolezecký algoritmus s učením predstavuje modifikáciu štandardného horolezeckého algoritmu. Okolie sa generuje na základe pravdepodobnostného vektora, ktorý sa učí podľa Hebbovho pravidla.

4.1.3 Zakázané prehľadávanie

Zakázané prehľadávanie (Tabu Search) [8] predstavuje zovšeobecnenie horolezeckého algoritmu, ktoré rieši problém uviaznutia. Využíva jednoduchú heuristiku, ktorá umožňuje pokračovať v hľadaní globálneho optima bez možnosti okamžitého návratu do lokálneho optima, ktoré sa už zaznamenalo v predchádzajúcej histórii algoritmu. Hlavná myšlienka je pamätať si niekoľko posledných krokov a hľadania smerom k nim zakázať.

Presnejšie vysvetlené, uchovávajú sa inverzné transformácie k tým transformáciám riešení, ktoré poskytovali lokálne optimálne riešenia. Tieto transformácie sú zakázané pri tvorbe nového okolia pre aktuálne riešenie. Nové okolie je vždy generované deterministicky pomocou prípustných transformácií.

4.1.4 Simulované žíhanie

Simulované žíhanie (Simulated Annealing) [4, 15] má svoj základ vo fyzike. Žíhanie označuje proces zahriatia telesa a postupného chladnutia, čím sa odstraňujú defekty telesa. Ak chladnutie prebieha dostatočne pomaly, materiál prijme konfiguráciu s najnižšou energiou. Proces žíhania sa využíva pri výrobe skla a tavení kovov v metalurgii.

Pri simulovanom žíhaní sme ochotní počas hľadania akceptovať aj horšie riešenie s nasledovnou pravdepodobnosťou

$$P = e^{-\frac{E}{T}}$$

kde E predstavuje zlepšenie riešenia a T teplotu. Na začiatku hľadania sa nastaví vysoká teplota, ktorá spôsobí takmer náhodné hľadanie v priestore.

Postupným znižovaním teploty sa znižuje aj pravdepodobnosť prijatia horšieho riešenia. Pri veľmi malej teplote na konci sa už takmer vždy vyberie iba najlepšie riešenie.

Simulované žíhanie dokáže na začiatku priestor hľadania poriadne preskúmať a postupným chladnutím sú preferované lepšie riešenia. Rýchlosť chladnutia je daná rozvrhom. Akceptovanie aj horších riešení nám umožňuje uniknúť z lokálneho optima.

4.2 Evolučné optimalizačné metódy

Existujú viaceré konkrétne typy evolučných algoritmov, ktoré sa líšia v implementačných detailoch a v charaktere problému, na ktorý sa aplikujú. V porovnaní so základnými stochastickými optimalizačnými algoritmami uvedenými v predošlej časti, evolučné algoritmy neuvažujú iba jedno riešenie ale populáciu riešení.

4.2.1 Genetický algoritmus

Genetický algoritmus [13] patrí medzi najčastejšie a najpoužívanejšie evolučné algoritmy. Predstavuje proces prirodzenej evolúcie populácie.

Základným princípom je vytvorenie prvej populácie náhodných jedincov. Následne sa ohodnotí kvalita každého jedinca pomocou fitness funkcie. Fitness funkcia F zobrazuje množinu jedincov P na kladné reálne čísla a interpretuje sa ako kvalita riešenia, ktoré reprezentuje daný jedinec.

$$F : P \to R_+$$

Najlepšie ohodnotené jedince sa použijú na reprodukciu. Pomocou kríženia a mutovania vytvoria potomkov, ktorí reprezentujú novú generáciu populácie. Celý proces sa opakuje, pokiaľ nenájdeme uspokojivé riešenie alebo neuplynie vyhradený výpočtový čas. Algoritmicky možno proces popísať nasledovne:

1. Vygenerovanie populácie náhodných jedincov
2. Opakuj:
    a) Vyhodnotenie fitness funkcie pre každého jedinca
    b) Výber najlepších jedincov na reprodukciu
    c) Vytvorenie potomkov pomocou kríženia a mutácie
    d) Nahradenie starej populácie novou
3. Ak nie je splnená podmienka zastavenia, pokračuj od bodu 2
4. Vrátenie jedinca s najvyšším ohodnotením

Mutovanie spôsobuje náhodné zmeny v jedincoch. Je dôležité pre zachovanie genetickej rôznorodosti a pôsobí ako prevencia proti uviaznutiu v lokálnom optime. Kríženie predstavuje binárnu reprodukciu. V začiatkoch hľadania silne urýchľuje optimalizáciu.

Aby sa zabezpečila evolúcia k lepším riešeniam, je dôležité vyberať na reprodukciu nadpriemerne dobré jedince. Výber rodičov sa realizuje proporcionálne na základe fitness funkcie (výber ruletou). Jedince s vyššou fitness majú vyššiu pravdepodobnosť výberu.

Pre urýchlenie optimalizácie možno použiť ďalšie metaheuristiky, ako napríklad elitárstvo. Vtedy n najlepších jedincov presunieme do novej populácie bez kríženia a mutovania. Tým zabezpečíme, že najlepších jedincov nikdy nestratíme.

4.2.2 Genetické programovanie

Genetické programovanie [16] je inšpirované biologickou evolúciou, za účelom nájsť počítačové programy, ktoré vykonávajú používateľom definovanú úlohu v prostredí. Počítačový program pozostáva z množiny jednoduchých inštrukcií a vyhodnocuje sa pomocou fitness funkcie. Vyhodnotenie fitness funkcie spočíva vo vykonaní programu jedinca a vyhodnotení úspešnosti objektu v prostredí

(napríklad počet nazbieraných pokladov). Tento prístup je vhodný na riešenie problémov adaptácie (učenia sa).

Iná interpretácia genetického programovania je symbolická regresia. Cieľom regresie je odhadnúť vzťah medzi veličinami na základe trénovacích príkladov. Pri parametrickej regresii sa optimalizujú iba koeficienty, ale tvar funkcie ostáva nemenný. Pri symbolickej regresii sa optimalizuje aj tvar samotnej funkcie.

Funkcie sa reprezentujú ako výrazy pomocou syntaktických stromov obsahujúcich premenné, konštanty a operácie. Úspešná implementácia genetického programovania vyžaduje efektívnu metódu kódovania stromových štruktúr. Jednou z možností je použiť Readov lineárny kód.

4.2.3 Evolučné programovanie

Evolučné programovanie [7] patrí medzi základné typy evolučných algoritmov. Uvažujme populáciu P, ktorá predstavuje množinu riešení. Z populácie P sa náhodne vyberie podpopulácia rodičov Q, ktorá je upravená pomocou operátora mutácie na podpopuláciu potomkov Q'. Z týchto dvoch podpopulácií sa vytvorí zjednotená podpopulácia R. Aplikovaním turnaja na túto podpopuláciu R dostaneme podpopuláciu nasledovníkov S. Podpopulácia nasledovníkov sa vracia do pôvodnej populácie P tak, že sa pôvodná rodičovská podpopulácia Q odstráni.

V evolučnom programovaní sa nevyužíva kríženie. Jediným zdrojom zmien sú mutácie. Ako metóda selekcie sa používa výhradne turnaj, ktorý sa však neaplikuje na výber rodičov, ale na výber nasledovníkov.

4.2.4 Evolučné stratégie

Evolučné stratégie [26, 29] patria medzi prvé úspešné stochastické optimalizačné algoritmy. Na rozdiel od väčšiny ostatných stochastických optimalizačných

metód nie sú evolučné stratégie založené na binárnej reprezentácii premenných, ale manipulujú priamo s reálnou reprezentáciou. Spravidla využívajú operátory mutácie a selekcie.

Evolučné stratégie dokážu optimalizovať viacrozmerné funkcie, kde premenné funkcie predstavuje niekoľko reálnych čísel zoradených do vektora. Mutácia reálnej premennej sa vykonáva pridaním náhodnej hodnoty z normálneho rozdelenia každému komponentu vektora. Sila mutácie je určená smerodajnou odchýlkou normálneho rozdelenia.

Nové riešenie je akceptované deterministicky, ak je funkčná hodnota potomka lepšia ako funkčná hodnota rodiča. Ak potomok predstavuje horšie riešenie ako rodič, tak sa vymaže a generuje sa nový potomok. Toto sa opakuje dovtedy, kým nie je potomok lepší a následne nahradí rodiča.

Evolučné stratégie boli pôvodne vytvorené na optimalizáciu vektorov reálnych čísel, no môžu byť použité aj ako heuristika pre NP-úplné kombinatorické problémy.

4.3 Inteligencia roja

Inteligencia roja (Swarm Intelligence) je kolektívne správanie sa decentralizovaných samoorganizujúcich sa systémov. Koncept inteligencie roja sa využíva najmä v umelej inteligencii. Inšpirácia často pochádza z prírody, najmä z biologických systémov.

Systém pozostáva z populácie jednoduchých agentov, ktorí interagujú lokálne medzi sebou a prostredím. Agenti sa riadia veľmi jednoduchými pravidlami. Napriek tomu, že neexistuje centrálny riadiaci element, interakcie agentov vedú k emergencii inteligentného skupinového správania, nepoznaného pre individuálnych agentov.

4.3.1 Optimalizácia mravčou kolóniou

Optimalizácia mravčou kolóniou je pravdepodobnostná technika na riešenie výpočtových problémov, ktoré môžu byť zredukované na hľadanie najkratšej cesty v grafe. Originálna myšlienka pochádza z pozorovania, ako mravce dokážu nájsť najkratšiu cestu medzi mraveniskom a potravou.

Prvý mravec nájde potravu náhodnou cestou a pri návrate do mraveniska vypúšťa na cestu feromóny. Ak ďalší mravec nájde túto cestu, tak ju bude nasledovať a posilní ju svojimi feromónmi. Feromóny sa časom vyparujú, čo znižuje atraktivitu danej cesty.

Čím dlhšie trvá prejsť danú cestu, tým viac času majú feromóny na vyparenie. Po krátkych cestách sa bude prechádzať častejšie, čím sa zvýši hustota feromónov na krátkych cestách. Postupom času bude mať najkratšia cesta najviac feromónov a feromóny na iných cestách sa vyparia. Na záver budú všetky mravce cestovať za potravou najkratšou cestou.

4.3.2 Optimalizácia včelím rojom

Optimalizácia včelím rojom sa zaraďuje do umelej inteligencie ako optimalizačná technika inšpirovaná správaním sa biologických včiel. Vychádza z nej množstvo konkrétnych algoritmov, ktoré využívajú inteligenciu roja. Najznámejší z nich je včelí algoritmus.

Roj včiel je rozdelený na dve skupiny: prieskumné a vyčkávajúce včely. Prieskumné včely sa náhodne rozmiestnia v priestore hľadania. Vyhodnotí sa kvalita ich riešenia a vyberú sa najlepšie včely. Následne sa vyčkávajúce včely pridružia k vybraným prieskumným včelám. Prieskumné včely s lepším ohodnotením získajú viacej vyčkávajúcich včiel. Pridružené včely vykonávajú lokálne prehľadávanie priestoru okolo pridelenej prieskumnej včely. Ak nájdu lepšie riešenie ako prieskumná včela, tak prieskumná včela si ho aktualizuje.

Z každej prehľadávanej oblasti sa vyberie iba najlepšia včela na vytvorenie novej populácie včiel. Zvyšné včely sa náhodne rozmiestnia v priestore hľadania. Na konci každej iterácie bude nová populácie pozostávať z dvoch častí, a to z najlepších včiel z jednotlivých oblastí hľadania a z náhodne rozmiestnených včiel.

4.4 Charakteristiky evolučných algoritmov

Pri evolučných algoritmoch hrajú dôležitú úlohu implementačné detaily a parametre, ktoré významným spôsobom ovplyvňujú úspešnosť algoritmu.

4.4.1 Reprezentácia jedincov

Cieľom reprezentácie jedincov je zakódovanie prehľadávaného priestoru riešení. Existujú viaceré spôsoby reprezentácie jedincov, pričom žiadna nie je ideálna pre všetky problémy. Rôzne reprezentácie sú vhodné na rôzne problémy.

Binárna reprezentácia

Binárna reprezentácia jedincov patrí medzi najpoužívanejšie reprezentácie v genetických algoritmoch. Je obzvlášť vhodná pre svoju efektivitu, jednoduchosť mutovania a kríženia.

Pri optimalizácii funkcií sa spravidla pracuje s reálnymi číslami. Pokiaľ chceme využiť výhody binárnej reprezentácie, musíme reálnu premennú reprezentovať binárne. Reálne číslo $x \in [a, b]$ sa prevedie na celé číslo z pomocou vzťahu

$$z = \left\lfloor \frac{x - a}{b - a} \left(2^k - 1\right) \right\rfloor$$

kde k je dĺžka bitového reťazca. Získané celé číslo z je následne možné priamočiaro reprezentovať binárnym spôsobom.

Štandardné kódovanie

Nevýhodou štandardného binárneho kódovania je, že susedné čísla sa môžu líšiť vo všetkých bitoch. Napríklad

$$3 = 011$$
$$4 = 100$$

Toto je obzvlášť nepríjemné pri hľadaní extrému funkcie, kde by sme sa mali pohybovať po malých krokoch pomocou mutácie. Avšak na zmenu z hodnoty $2^k - 1$ na hodnotu 2^k by sme potrebovali mutáciu, ktorá preklopí všetky bity. Takýto jav sa nazýva *Hammingova bariéra*.

Grayovo kódovanie

Jedným z riešení ako eliminovať Hammingovu bariéru je použitie Grayovho kódu. Grayov kód je taký binárny kód, v ktorom sa dve susedné čísla líšia v práve jednom bite. Pri optimalizácii funkcie nám Grayov kód umožňuje pohybovať sa po menších krokoch a dospieť skôr k riešeniu.

Nech α je binárny reťazec dĺžky k v štandardnom kódovaní a $\tilde{\alpha}$ je binárny reťazec v Grayovom kódovaní. Prevod jednotlivých bitov zo štandardného kódovania na Grayovo kódovanie je nasledovný

$$\begin{aligned}
\tilde{\alpha}_1 &= \alpha_1 \\
\tilde{\alpha}_2 &= \alpha_1 \oplus \alpha_2 \\
\tilde{\alpha}_3 &= \alpha_2 \oplus \alpha_3 \\
&\vdots \\
\tilde{\alpha}_k &= \alpha_{k-1} \oplus \alpha_k
\end{aligned}$$

pričom symbol $\oplus$ označuje logickú operáciu *XOR*.

Celočíselná reprezentácia

Celočíselná reprezentácia je prirodzená pre mnohé problémy, napríklad spracovanie obrazu. Taktiež je vhodná napríklad na výber kategórie farieb

z danej množiny. Špeciálnym prípadom sú jedince pozostávajúce z reťazcov znakov. Znaky sa reprezentujú pomocou celočíselných kódov z *ASCII* tabuľky.

Operátor mutácie môže aj pri celočíselnej reprezentácii vyjadrovať malú zmenu, napríklad pričítanie alebo odčítanie 1. Iný spôsob mutácie môže byť vygenerovanie úplne náhodného čísla z daného intervalu.

Reálna reprezentácia

Množstvo problémov je založených na reálnych číslach, ako napríklad optimalizácia spojitého parametra Jednou možnosťou je prevod reálnej premennej do binárneho tvaru. V takom prípade dĺžka bitového reťazca určuje maximálnu presnosť riešenia. Vysoká presnosť implikuje jedince s dlhým bitovým reťazcom, čo spomaľuje výpočty a evolúciu.

Iným riešením je pôvodnú reálnu reprezentáciu zachovať. Evolučné stratégie využívajú reálnu reprezentáciu na optimalizáciu viacrozmerných funkcií. Mutácia spočíva v pripočítaní náhodného celého čísla z Gaussovho rozdelenia $N(0, \sigma^2)$. Pomocou hodnoty rozptylu σ^2 sa kontroluje veľkosť zmeny mutácie.

Pri krížení bude hodnota potomka pochádzať od jedného zo svojich dvoch rodičov s rovnakou pravdepodobnosťou. Prípadne môže potomok nadobudnúť hodnotu medzi hodnotami svojich rodičov, napríklad priemer hodnôt rodičov.

4.4.2 Metódy selekcie

Výber kvalitných jedincov na reprodukciu patrí medzi základné črty genetického algoritmu. Výberom vhodných jedincov dokážeme výrazným spôsobom urýchliť optimalizáciu. Kvalita jedincov sa posudzuje podľa fitness funkcie.

Výberom iba najlepších jedincov by mohol genetický algoritmus veľmi rýchlo uviaznuť v lokálnom optime, preto aj horší jedinci majú nenulovú pravdepodobnosť výberu.

Selekcia ruletou

Pomocou rulety sa jedinci vyberajú proporcionálne na základe ich fitness. Jedincov s najvyšším ohodnotením vyberie s najväčšou pravdepodobnosťou a jedincov s najnižším ohodnotením s najmenšou pravdepodobnosťou.

Táto metóda selekcie sa nazýva podľa rulety, pretože na kolese rulety má každý jedinec vyhradený segment úmerný jeho fitness. Následne sa vygeneruje náhodné číslo, ktoré simuluje roztočenie kolesa rulety. Jedinec s lepšou fitness sa vyberie častejšie. Pravdepodobnosť výberu jedinca i z populácie P možno vyjadriť nasledovne:

$$P(i) = \frac{f(i)}{\sum_{j \in P} f(j)}$$

Ak fitness jedného jedinca priveľmi dominuje, výber ruletou môže príliš potlačiť šancu ostatných jedincov. Ak fitness jedincov je veľmi podobná, ruleta nedokáže dostatočne uprednostniť lepších jedincov.

Selekcia ohodnotením

Selekcia ohodnotením (Rank-based selection) odstraňuje vyššie uvedené problémy pri výbere ruletou. Po novom má na kolese rulety každý jedinec vyhradený segment úmerný jeho poradiu v usporiadaní podľa fitness. Najhorší jedinec dostane segment dĺžky 1, najlepší dĺžky N (počet jedincov v populácii).

Pri selekcii ohodnotením nehrá žiadnu rolu, koľkokrát je najlepší jedinec lepší od ostatných. Uvažuje sa iba poradie jedincov navzájom. Takýto spôsob môže spomaliť konvergenciu k riešeniu, pretože najlepší jedinci na kolese rulety už nebudú tak suverénne dominovať.

Selekcia turnajom

Pri selekcii turnajom sa uvažuje parameter veľkosti turnaja t. Z populácie sa náhodne zvolí t jedincov a z nich sa vyberie ten najlepší. Typicky $t = 2$.

V takom prípade sa z populácie náhodne vyberú dvaja jedinci a zoberie sa ten lepší z nich.

Čím väčšia veľkosť turnaja, tým rýchlejšia konvergencia. Dôvodom je, že najlepší jedinec z väčšieho počtu jedincov bude priemerne lepší ako najlepší jedinec z menšieho počtu.

Selekcia turnajom má niekoľko výhod. Je jednoduchá na implementáciu, efektívna a umožňuje pomocou veľkosti turnaja nastavovať silu selekcie a tým rýchlosť konvergencie.

Selekcia skrátením

Pri selekcii skrátením (Truncation selection) sa jedinci usporiadajú podľa fitness. Následne sa vyberie podiel p najlepších jedincov (napríklad $p = 1/3$) a reprodukuje sa $1/p$ krát.

Selekcia skrátením je menej sofistikovaná ako iné metódy selekcie, a preto sa príliš často nepoužíva.

4.4.3 Reprodukcia jedincov

Reprodukcia jedincov sa vykonáva za účelom vytvorenia novej generácie populácie. Reprodukcia prebieha aplikovaním dvoch nasledovných operátorov:

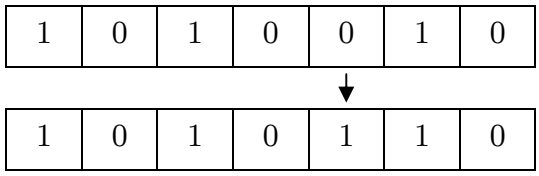
1. Mutácia
2. Kríženie

Mutácia

Mutácia predstavuje unárny reprodukčný operátor. Nový jedinec vzniká aplikovaním náhodných zmien na svoj pôvodný stav. Mutácie zabezpečujú zachovanie genetickej rôznorodosti jedincov. Bez použitia mutácie by populácia veľmi rýchlo skonvergovala do lokálneho optima a nedokázala by sa ďalej vyvíjať.

Hlavným parametrom mutácie je pravdepodobnosť mutovania. Tým, že mutácia prináša náhodné zmeny, veľká pravdepodobnosť mutovania spôsobí primitívne náhodné prehľadávanie. Preto sa väčšinou odporúča nastaviť malú pravdepodobnosť mutovania.

Konkrétna implementácia mutácie závisí od reprezentácie jedincov. V prípade bitového reťazca sa najčastejšie jedná o preklopenie náhodne vybraného bitu:



V prípade číselnej reprezentácie sa môže jednať o pripočítanie malej hodnoty náhodne vygenerovanej normálnym rozdelením.

Kríženie

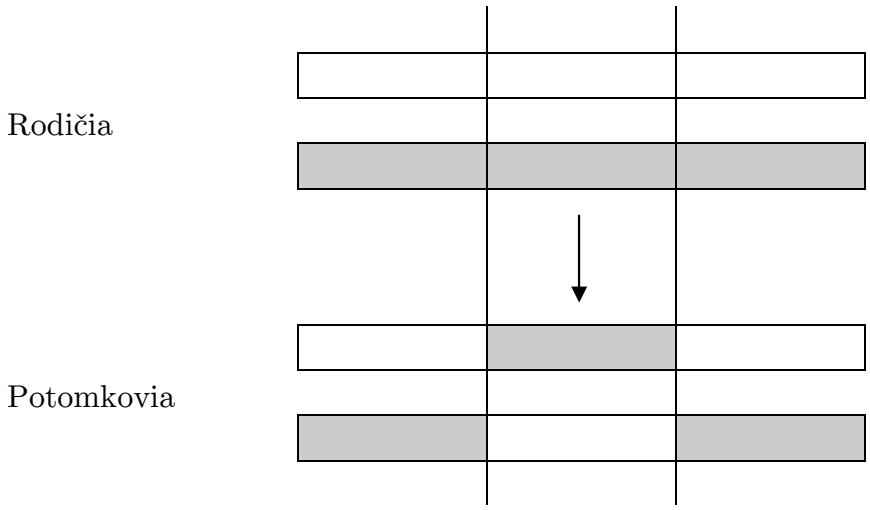
Kríženie predstavuje binárny spôsob reprodukcie. Na kríženie sa vyberú dvaja rodičia, ktorých rekombináciou vznikne nový jedinec. Kríženie silne urýchľuje hľadanie v začiatkoch evolúcie. Pri krížení očakávame, že noví jedinci zdedia dobré vlastnosti a budú lepší.

Existujú viaceré spôsoby kríženia:

a) Jednobodové

b) Viacbodové

Kríženie je podobne ako mutácia stochastický operátor. Pri aplikovaní kríženia na dvojicu jedincov sa bod kríženia určuje náhodne. Ukážka dvojbodového kríženia:



Podobne ako mutácia aj kríženie sa aplikuje s určitou pravdepodobnosťou. Ak je jednotková, všetci potomkovia vzniknú krížením. Odporúča sa časť populácie nechať prežiť do ďalšej generácie.

4.4.4 Populačné modely

Pri genetických algoritmoch existuje viacero populačných modelov, ktoré determinujú zloženie novej generácie populácie.

Generačný model

Generačný model je štandardne používaný v základnom genetickom algoritme. Založený je na fakte, že každý jedinec prežije iba jednu generáciu. Nová populácia bude pozostávať výhradne z potomkov súčasných jedincov.

Ustálený model

Ustálený model (Steady-state) je na opačnom konci spektra ako generačný model. Iba jeden nový potomok je vytvorený každú generáciu, ktorý nahradí iba jedného súčasného jedinca. Zvyšok populácie ostane zachovaný.

Proporcionálny model

Proporcionálny model predstavuje určitý kompromis medzi generačným a ustáleným modelom. Každú generáciu sa nahradí zvolený pomer populácie novými jedincami.

Kapitola 5

Generovanie samoopravných kódov

„Je lepšie riešiť správny problém nesprávnym spôsobom, ako nesprávny problém správnym spôsobom.“

~ Richard Hamming (1915 - 1998)

V tejto časti analyzujeme problematiku generovania samoopravných kódov a uvádzame príklad, ktorý odhaľuje zaujímavý vzťah medzi teóriou kódovania a teóriou grafov.

5.1 Analýza problematiky

Množstvo informácií, ktoré je možné daným kódom zakódovať, je závislé na dĺžke kódových slov a na abecede, z ktorej sú kódové slová vytvorené. Pre naše účely budeme uvažovať binárnu abecedu kódových slov. Dlhšie kódové slová majú za následok väčší počet možných kódových slov. Samoopravné kódy s väčším počtom kódových slov umožňujú zakódovať väčší počet zdrojových správ a zlepšujú tak efektívnosť komunikácie.

Nanešťastie ako rastie počet kódových slov, tak sa zväčšuje aj prehľadávaný priestor kódových slov, ktoré navzájom spĺňajú požiadavku na minimálnu

vzdialenosť. Každé kódové slovo sa musí líšiť od každého iného aspoň o toľko bitov, ako je požadovaná minimálna vzdialenosť.

Problém návrhu samoopravných kódov pozostáva z priradenia kódových slov prenášaným správam tak, aby sa minimalizovala veľkosť zakódovaných správ a zároveň, aby sa poskytla schopnosť opraviť dostatočný počet chýb.

Tieto dve podmienky sú v protiklade už z princípu. Minimalizovať veľkosť zakódovaných správ je možné tak, že im priradíme krátke kódové slová. Dosiahnuť schopnosť opraviť veľký počet chýb je možné pridaním väčšieho počtu redundantných bitov, aby sa zväčšila minimálna vzdialenosť medzi ľubovoľnými dvoma kódovými slovami.

Optimálny samoopravný kód pozostáva z M kódových slov dĺžky n takých, že minimálna Hammingova vzdialenosť d medzi všetkými pármi kódových slov je maximálna možná. Preto dobrý (n, M, d) kód je taký, ktorý má malé n, veľké M a veľké d. Hlavným problémom teórie kódovania je optimalizovať jeden z parametrov n, M, d pre zvolené hodnoty zvyšných dvoch. Maximálny možný počet kódových slov v kóde dĺžky n s minimálnou vzdialenosťou d a veľkosťou abecedy q sa označuje $A_q(n, d)$.

Pri rozširovaní počtu kódových slov v samoopravnom kóde je nevyhnutný exponenciálny počet porovnaní za účelom zachovania minimálnej vzdialenosti kódu a teda aj počtu chýb, ktoré je schopný opraviť. Už pri miernej dĺžke kódu je exponenciálna zložitosť generovania samoopravných kódov nepríjemná. Preto generovanie optimálnych samoopravných kódov je netriviálna záležitosť.

5.2 Redukcia problému

Problém generovania samoopravných kódov je možné zredukovať na problém hľadania maximálnej kliky grafu. Na redukciu inštancie problému z teórie kódovania na inštanciu problému z teórie grafov si predstavme vrcholy grafu ako všetky možné kódové slová danej dĺžky. Dva vrcholy prepojíme hranou

práve vtedy, ak dve kódové slová reprezentované danými vrcholmi spĺňajú požiadavku na minimálnu vzdialenosť. Následne maximálna klika takéhoto grafu je ekvivalentná optimálnemu samoopravnému kódu.

Predpokladajme, že by sme chceli nájsť maximálny počet kódových slov v binárnom kóde dĺžky 4 a minimálnou vzdialenosťou 2. Za účelom jednoduchosti a prehľadnosti predpokladajme, že nasledujúce kódové slová sú jediné možné:

$$\{0000, 0011, 1010, 1011, 1110, 1111\}$$

Pri každom vyhodnotení možného riešenia je potrebné vypočítať vzdialenosť medzi každým párom kódových slov za účelom zistenia minimálnej vzdialenosti. Preto je výhodné si tieto vzdialenosti predpočítať na začiatku.

Vytvoríme si maticu kompatibility A z potenciálnych kódových slov, kde

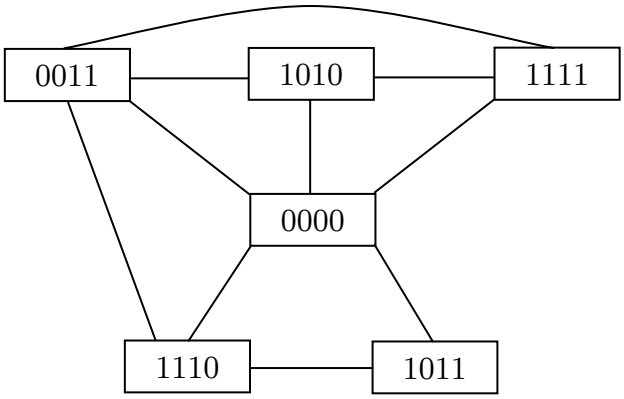
$$a_{ij} = \begin{cases} 1 & \text{keď kódové slová } i \text{ a } j \text{ spĺňajú minimálnu vzdialenosť} \\ 0 & \text{inak} \end{cases}$$

Tab. 2. Matica kompatibility kódových slov

	0000	**0011**	**1010**	**1011**	**1110**	**1111**
0000	0	1	1	1	1	1
0011	1	0	1	0	1	1
1010	1	1	0	0	0	1
1011	1	0	0	0	1	0
1110	1	1	0	1	0	0
1111	1	1	1	0	0	0

Matica kompatibility kódových slov (Tab. 2) vlastne predstavuje maticu susednosti grafu, v ktorom vrcholy sú kódové slová a hrany spájajú tie vrcholy,

ktorých kódové slová sú kompatibilné, t.j. spĺňajú požiadavku na minimálnu vzdialenosť (Obr. 5).



Obr. 5. Graf kódových slov

Veľkosť maximálnej kliky grafu uvedeného na predošlom obrázku je 4 a vrcholy tvoriace maximálnu kliku sú:

$$\{0000, 0011, 1010, 1111\}$$

Preto aj maximálny počet kódových slov je rovný 4. Kódové slová patriace do optimálneho kódu sú tie, ktoré tvoria maximálnu kliku.

5.3 Existujúce riešenia

V tejto časti analyzujem súčasné existujúce riešenia problému generovania optimálnych samoopravných kódov a hľadania maximálnej kliky v grafe.

5.3.1 Porovnanie evolučných algoritmov pre hľadanie optimálnych samoopravných kódov

V článku [10] autori porovnávajú rôzne evolučné a iné optimalizačné techniky na hľadanie optimálnych samoopravných kódov. Optimálny samoopravný kód je

taký, ktorý dosiahne maximálny možný počet kódových slov M pre zadanú dĺžku kódu n, minimálnu vzdialenosť d a abecedu kódera o veľkosti q.

Zatiaľ čo maximálny možný počet slov v kóde $(n, M, d)_q$ je pre menšie parametre známy, tak pri väčších inštanciách je známy iba interval hodnôt M. Cieľom článku je vyhodnotiť úspešnosť jednotlivých metód pre hľadanie optimálnych samoopravných kódov.

Návrh riešenia

Autori si uvedomujú nevyhnutnosť predpočítania matice kompatibility pre rýchlejšie zisťovanie, či ľubovoľné dve kódové slová sú navzájom kompatibilné, t.j. spĺňajú požiadavku na minimálnu vzdialenosť. Preto si vygenerujú všetky možné kódové slová dĺžky n vo vzdialenosti aspoň d od nulového slova a výsledok každej dvojice si uložia do matice kompatibility.

Následne autori redukujú problém generovania samoopravných kódov na problém hľadania maximálnej kliky v grafe a využívajú predpočítanú maticu kompatibility ako maticu susednosti grafu.

Porovnávané algoritmy

1. Horolezecký algoritmus
   - počet iterácií: 100
2. Hľadanie v lúči (Beam Search)
   - veľkosť lúča: 5
   - počet iterácií: 100
3. Simulované žíhanie
   - počiatočná teplota t_0: 2
   - počet prechodov pre každú teplotu t: 100
   - rýchlosť chladnutia: $t_{k+1} = 0.8\, t_k$

4. Greedy metódy
    - lexikografický kód
    - randomizovaný greedy
5. Genetické algoritmy
    - veľkosť populácie: 500
    - počet generácií: 100
    - počet behov: 10
    - metóda selekcie: turnaj veľkosti 3
    - pravdepodobnosť kríženia: 85%
    - pravdepodobnosť mutácie: 15%

Reprezentácia riešení

Autori v článku využívajú dva spôsoby reprezentácie možných riešení:

a) Priama reprezentácia

b) Nepriama reprezentácia

Priama reprezentácia

Jedince sú reprezentované binárnym vektorom, ktorý indikuje pre každé kódové slovo, či je súčasťou kódu alebo nie. Pri priamej reprezentácii sa aplikovaním štandardných genetických operátorov môže vytvoriť nepovolené riešenie, ktoré nie je klikou.

Pri tejto reprezentácii sa preto musia využiť rôzne techniky, ktoré zabezpečia korektnosť riešenia. Autori uvažujú nasledovnú techniku opravy pozostávajúcu z dvoch krokov:

1. Vyextrahovať kliku zo súčasného grafu opakovaným odstraňovaním vrcholov vybratých náhodne alebo s najmenším stupňom.

2. Rozšíriť súčasnú kliku na maximálnu kliku opakovaným pridávaním vrcholov vybratých náhodne alebo s najväčším stupňom.

Nepriama reprezentácia

Riešenia sa reprezentujú ako postupnosť celých čísel, kde prvé číslo reprezentuje prvé vybraté kódové slovo, druhé číslo reprezentuje druhé vybraté kódové slovo, a tak ďalej.

Za účelom zachovania čo najmenšieho prehľadávaného priestoru sa uvažujú iba tie riešenia, ktoré sú naozaj kliky. To sa dosiahne tým, že s celými číslami sa bude zaobchádzať ako s posunmi.

Napríklad uvažujme, že hľadáme binárny kód dĺžky 4 s minimálnou vzdialenosťou 2 a nasledovné vektory sú jediné možné kódové slová:

$$\{0000, 0011, 1010, 1011, 1110, 1111\}$$

Predpokladajme, že jedinec je $[3, 1, 1, 4, 5, 2]$. Najskôr sa vyberie do kódu nulový vektor [0000]. Všetky ostávajúce kódové slová sú kompatibilné:

$$\{0011, 1010, 1011, 1110, 1111\}$$

Podľa jedinca vyberieme najskôr tretie kódové slovo, takže dostaneme kód [0000, 1011]. Spomedzi možných kódových slov ostáva už len jediné, ktoré neporušuje požiadavku na minimálnu vzdialenosť:

$$\{1110\}$$

Podľa jedinca vyberieme teraz prvé kódové slovo, čím dostaneme kód [0000, 1011, 1110], ktorý je zároveň výsledný pre uvažovaného jedinca, pretože už žiadne ďalšie kompatibilné kódové slová neostali.

Výsledky

Pri hľadaní optimálnych binárnych kódov jednoznačne dominovali genetické algoritmy. Menovite najlepší bol genetický algoritmus s nepriamou reprezentáciou jedincov, ktorej veľkosť bola parametricky obmedzená. Navyše tento algoritmus na konci pridal do kódu všetky kompatibilné kódové slová, ktoré neboli vybraté najlepším jedincom. Na tento účel autori využili Conwayov greedy algoritmus na generovanie lexikografických kódov.

Tento algoritmus pridáva nové kódové slová v lexikografickom poradí tak, že musia spĺňať požiadavku na minimálnu Hammingovu vzdialenosť s doposiaľ pridanými kódovými slovami.

Pri optimalizovaní ternárnych kódov sa ukázal byť najvhodnejší opäť genetický algoritmus, ale teraz s priamou reprezentáciou jedincov a so špeciálnou technikou na opravu jedincov, aby reprezentovali kliky. Iné prístupy ako horolezecký algoritmus, hľadanie v lúči, simulované žíhanie a čisté greedy algoritmy sa nepresadili.

Napriek tomu, že sa autorom nepodarilo nájsť samoopravný kód s väčším počtom kódových slov, ako je doposiaľ známe, podarilo sa im úspešne ukázať, že genetické algoritmy sú vhodné na tento konkrétny problém.

5.3.2 Evolučné prístupy pre generovanie optimálnych samoopravných kódov

Článok [22] vychádza z článku [10] a ďalej rozvíja možnosti evolučných prístupov pre generovanie optimálnych samoopravných kódov. V tomto článku autori uvádzajú nový spôsob reprezentácie jedincov v genetickom algoritme a rovnako sa pokúšajú aplikovať myšlienky genetického programovania na riešený problém. Cieľom článku nie je objaviť nový optimálny samoopravný kód, ale porozumieť efektivite rôznych reprezentácií pri hľadaní nových kódov.

Predpokladá sa, že genetické programovanie môže byť vhodné na riešenie tohto problému, pretože by malo lepšie odolať efektu epistázy. Epistáza je forma génovej interakcie, kde nadradený gén potláča účinky podradeného génu. Obmedzenie tohto efektu spôsobí, že vzory kompatibilných kódových slov bude možné počas evolúcie efektívne dediť.

Autori porovnávajú svoje riešenia na nasledovných $A_2(n, d)$ kódoch:

a) $A_2(12, 6)$

b) $A_2(13, 6)$

c) $A_2(17, 6)$

Genetický algoritmus

Použitý genetický algoritmus vychádza z algoritmu navrhnutého v článku [10]. Okrem toho prináša nový spôsob reprezentácie riešení a porovnáva ho s nepriamou reprezentáciou z predošlého článku.

Autori taktiež využívajú kombináciu genetického algoritmu a greedy algoritmu na generovanie lexikografických kódov prezentovaného Conwayom. Genetický algoritmus sa použije na vygenerovanie počiatočných kódových slov. Tie sú na záver doplnené greedy algoritmom o ďalšie kódové slová, ktoré neporušujú požiadavku na minimálnu vzdialenosť kódových slov.

Reprezentácia

Reprezentácia jedincov prezentovaná v tomto článku využíva novú schému indexovania kódových slov. Jedná sa o priame indexovanie s využitím operácie modulo. Každému génu je priradená hodnota v intervale 0 až najväčšie celé číslo. Kódové slovo, ktoré prislúcha danému génu, sa získa ako hodnota génu modulo celkový počet kódových slov.

Ak sa dereferencuje nekompatibilné kódové slovo, spustí sa lineárne hľadanie od tohto kódového slova, až kým nie je nájdené nasledujúce kompatibilné kódové

slovo alebo kým nie sú všetky kompatibilné kódové slová vyčerpané. Autori sa domnievajú, že pomocou tejto reprezentácie sa dá obmedziť nežiaduci efekt epistázy.

Parametre

Základné parametre prezentovaného genetického algoritmu sú uvedené v (Tab. 3).

Tab. 3. Parametre genetického algoritmu

Parameter	**Hodnota**
Veľkosť populácie	500
Maximálny počet generácií	100
Metóda selekcie	Turnaj (veľkosť 3)
Elitárstvo	1 jedinec
Kríženie	Dvojbodové
Pravdepodobnosť kríženia	85%
Mutácia	Mutovanie náhodných génov
Pravdepodobnosť mutácie	15%

Genetické programovanie

V genetickom programovaní je kód reprezentovaný pomocou stromu. Každý strom vyjadruje kompletný kód. Na rozdiel od fixnej dĺžky jedinca v genetickom algoritme, stromy v genetickom programovaní majú premenlivú veľkosť. Premenlivá veľkosť stromu môže urýchliť konvergenciu tým, že obmedzí reprezentáciu neoptimálnych kódov v nadrozmerných génoch.

Počas vyhodnocovania stromu sa posúva medzi vrcholmi stavový objekt, ktorý obsahuje rozpracovaný zoznam kódov. Funkčné a terminálne vrcholy stromu pridávajú do tohto objektu nové kódové slová, ktoré sú kompatibilné s už pridanými. Týmto spôsobom sa počas vyhodnocovania stromu vytvoria iba validné kódy.

Reprezentácia

Kódové slovo je celočíselná hodnota indexu, ktorú je možné dereferencovať do binárneho kódového slova. Kód je zoskupenie jedného alebo viacerých kompatibilných kódových slov.

Autori využívajú dva základné genetické operátory:

1. Operátor zjednotenia, ktorý pozbiera kódové slová alebo kódy vytvorené jeho dcérskymi vrcholmi.
2. Conwayov operátor zjednotenia, ktorý poskytuje štandardnú operáciu zjednotenia, ale navyše je obohatený o Conwayov greedy algoritmus. V tomto prípade je možné parametricky určiť, koľko kódových slov by malo byť prítomných v rozpracovanom kóde predtým, ako povolíme Conwayovmu algoritmu skompletizovať kód greedy spôsobom.

Parametre

Základné parametre prezentovaného genetického programovania sú v (Tab. 4).

Tab. 4. Parametre genetického programovania

Parameter	**Hodnota**
Veľkosť populácie	2000
Maximálny počet generácií	100
Metóda selekcie	Turnaj (veľkosť 3)

Elitárstvo	1 jedinec
Pravdepodobnosť kríženia	60%
Pravdepodobnosť mutovania	10%

Výsledky

Najmenší testovaný kód $(12, 6)_2$ nerobil problém žiadnemu algoritmu. Genetický programovanie aj genetický algoritmus boli schopné nájsť optimálne riešenie. Zmena reprezentácie pri genetickom algoritme neovplyvnila výsledok.

Pri druhom testovanom kóde $(13, 6)_2$ žiadny algoritmus nebol schopný nájsť optimálne riešenie. Nová reprezentácia jedincov v genetickom algoritme nemala zásadný vplyv na výsledok pri použití Conwayovho greedy algoritmu na záver hľadania. Genetické programovanie sa dokázalo viac priblížiť k optimu ako genetický algoritmus.

V najťažšom testovanom kóde $(17, 6)_2$ sa ukázalo priame indexovanie s modulo operáciou v genetickom algoritme ako vhodnejší spôsob reprezentácie v prípade, že nebol použitý Conwayov greedy algoritmus. Pri použití Conwayovho algoritmu boli rozdiely zanedbateľné. Genetickému algoritmu aj genetickému programovaniu sa podarilo dosiahnuť cieľový výsledok v jednom pokuse, čo sa dá skôr pripísať náhode.

Autori potvrdili vhodnosť Conwayovho algoritmu na skompletizovanie kódu po ukončení evolúcie. Genetický algoritmus spolu s Conwayovým greedy algoritmom prekonali čistý genetický algoritmus v každom testovacom príklade.

Vo výsledku genetický algoritmus spolu s genetickým programovaním podávali podobné výsledky a nie je medzi nimi jasný víťaz. Nevýhodou genetického programovania je, že jeho beh trvá až 3-krát dlhšie ako beh genetického algoritmu.

5.3.3 Heuristický genetický algoritmus pre problém maximálnej kliky

V tomto článku [20] autorka navrhuje nový heuristický genetický algoritmus pre problém hľadania maximálnej kliky v grafe. V predošlých výskumoch bolo ukázané, že jednoduchý genetický algoritmus nie je najvhodnejší pre hľadanie maximálnej kliky. Bez dodatočnej heuristiky neposkytuje dostatočne kvalitné riešenia a aj s rôznymi vylepšeniami nedokáže prekonať algoritmy lokálneho hľadania, ako simulované žíhanie a zakázané hľadanie podporené sofistikovanými heuristikami.

Heuristický genetický algoritmus je kombináciou jednoduchého genetického algoritmu a naivnej heuristiky. Zaujímavosťou je, že napriek svojej jednoduchosti dokáže významným spôsobom prekonať všetky doposiaľ známe prístupy založené na genetických algoritmoch.

Heuristika pre problém maximálnej kliky

Navrhovaný jednoduchý heuristický algoritmus pracuje v 3 krokoch, ktoré sa sekvenčne aplikujú na podgraf. Najskôr sa pridá do podgrafu niekoľko náhodne vybratých vrcholov. Následne sa z podgrafu vyextrahuje klika použitím naivnej randomizovanej procedúry. Na záver sa táto klika rozšíri pomocou sekvenčnej greedy heuristiky.

Predpokladajme, že vrcholy grafu sú umiestnené v postupnosti $\langle n_1, \ldots, n_N \rangle$ pre N vrcholov tak, že prvok na i-tej pozícii v postupnosti predstavuje i-ty vrchol grafu. Heuristický algoritmus môžeme vyjadriť nasledovnými krokmi:

1. Rozšírenie podgrafu
    a) Pridaj do podgrafu zopár nových vrcholov vybraných náhodne z grafu.
2. Vyextrahovanie kliky

a) Vyber náhodne pozíciu idx takú, že $1 \leq idx \leq N$.

b) $\forall i \in [idx \ldots N]$: ak n_i patrí do podgrafu, tak

- buď odstráň n_i alebo
- $\forall j \in [i+1 \ldots N]$: odstráň n_j, ak patrí do podgrafu a n_j nie je spojené s n_i
- $\forall j \in [1 \ldots i-1]$: odstráň n_j, ak patrí do podgrafu a n_j nie je spojené s n_i

c) $\forall i \in [idx-1 \ldots 1]$: ak n_i patrí do podgrafu, tak

- buď odstráň n_i alebo
- $\forall j \in [i-1 \ldots 1]$: odstráň n_j, ak patrí do podgrafu a n_j nie je spojené s n_i

3. Rozšírenie kliky

a) Vyber náhodne pozíciu idx takú, že $1 \leq idx \leq N$

b) $\forall j \in [idx \ldots N]$: pridaj n_j, ak je spojené so všetkými vrcholmi podgrafu získanými doposiaľ

c) $\forall j \in [1 \ldots idx-1]$: pridaj n_j, ak je spojené so všetkými vrcholmi podgrafu získanými doposiaľ

Tento heuristický algoritmus nám pre ľubovoľný podgraf vráti kliku, ktorá nie je podmnožina inej kliky, t.j. už nemôže byť rozšírená o ďalšie vrcholy.

Heuristický genetický algoritmus

Genetický algoritmus je taktiež veľmi jednoduchý. Vychádza sa zo štandardného genetického algoritmu, kde populácia pozostáva z bitových reťazcov, ktoré reprezentujú podgraf daného grafu a funkcia fitness spočíta veľkosť podgrafu.

Heuristický algoritmus sa kombinuje s jednoduchým genetickým algoritmom tak, že sa heuristika aplikuje na každého jedinca populácie v každej iterácii pred výpočtom fitness. Celý heuristický genetický algoritmus potom vyzerá nasledovne:

1. Nastav $t = 0$
2. Inicializuj $P(t)$
3. Aplikuj heuristický algoritmus na $P(t)$
4. Ohodnoť $P(t)$
5. Kým nie je splnená podmienka zastavenia, opakuj:
    a) $t = t + 1$
    b) Vyber $P(t)$ z $P(t-1)$
    c) Aplikuj procedúru diverzifikácie na $P(t)$
    d) Pozmeň $P(t)$ pomocou mutácií a kríženia
    e) Aplikuj heuristický algoritmus na $P(t)$
    f) Ohodnoť $P(t)$

Ako je možné vidieť, autorka uplatnila s malou pravdepodobnosťou aj diverzifikačnú procedúru, ktorá nahradí jedinca vybratého na reprodukciu novým kvázináhodne vygenerovaným jedincom.

Reprezentácia

Jedinec je reprezentovaný bitovým reťazcom rovnakej dĺžky, ako je počet vrcholov grafu. Každý bit reprezentuje jeden vrchol uvažovaného grafu. Jedinec charakterizuje podgraf, ktorý pozostáva z tých vrcholov, ktorých bity v reťazci majú hodnotu 1.

Fitness

Fitness funkcia v navrhovanom algoritme vyjadruje iba jednu vlastnosť podgrafu na rozdiel od iných algoritmov, ktorých fitness kombinuje viaceré vlastnosti. Jediná vlastnosť podgrafu, ktorú funkcia fitness berie do úvahy je veľkosť podgrafu. Fitness je rovná počtu jednotiek v binárnom reťazci jedinca, ak reprezentuje kliku, inak sa rovná 0.

Typ genetického algoritmu

Autorka použila generačný genetický algoritmus s mechanizmom elitárstva, ktoré skopíruje dvoch najlepších jedincov do novej populácie.

Genetické operátory

V navrhovanom heuristickom genetickom algoritme sa využíva rovnomerné kríženie. Mutácia sa vykonáva výmenou náhodne vybratých pozícií.

Parametre

Použité parametre genetického algoritmu sú uvedené v (Tab. 5).

Tab. 5. Parametre heuristického genetického algoritmu

Parameter	**Hodnota**
Veľkosť populácie	50
Pravdepodobnosť mutácie	10%
Pravdepodobnosť kríženia	80%
Počet iterácií	100

Výsledky

Autorka porovnáva úspešnosť svojho algoritmu na testovacej sade DIMACS[1], ktorá obsahuje množstvo rozsiahlych grafov z rôznych aplikačných oblastí. Heuristický genetický algoritmus dokázal v 25 grafoch z 34 dosiahnuť najlepší známy výsledok z druhej výzvy DIMACS, ktorej sa zúčastnilo 15 heuristických algoritmov. Aj na zvyšných 9 grafoch dosiahol algoritmus uspokojivé výsledky.

Navrhnutý heuristický genetický algoritmus prekonal všetky doposiaľ známe prístupy založené na genetických algoritmoch a je zdatným súperom metód lokálneho prehľadávania s pokročilými heuristikami na riešenie tohto problému. Pokiaľ ide o rýchlosť behu, heuristický genetický algoritmus dokázal prekonať iné algoritmy na riešenie tohto problému o niekoľko rádov.

Vďaka jednoduchej fitness funkcii je hľadanie smerované výlučne k rozsiahlym grafom, zatiaľ čo aplikovanie heuristického algoritmu spoločne s genetickými operátormi je zodpovedné za kvalitu dosiahnutých riešení.

5.3.4 Evolučný algoritmus s riadenou mutáciou pre problém maximálnej kliky

V článku [30] autori prezentujú nový evolučný prístup pre problém hľadania maximálnej kliky. Hlavné črty nového evolučného prístupu sú nasledovné:

a) Riadená mutácia

b) Heuristická oprava riešení

c) Rozdelenie priestoru hľadania

Operátor riadenej mutácie

Algoritmy na odhad rozdelenia (Estimation of Distribution Algorithms) extrahujú globálne štatistické informácie z predchádzajúcich hľadaní a následne

[1] http://dimacs.rutgers.edu/Challenges

ich reprezentujú ako pravdepodobnostný model, ktorý charakterizuje rozdelenie sľubných riešení v priestore hľadania. Nové riešenia sa vytvárajú realizovaním štatistického výberu podľa tohto modelu.

Optimalizácia je potom vnímaná ako séria inkrementálnych aktualizácií pravdepodobnostného modelu počnúc od modelu, ktorý generuje rovnomerné rozdelenie riešení až po model, ktorý generuje optimálne riešenie.

Autormi navrhovaný operátor riadenej mutácie je kombináciou konvenčného mutačného operátora a pravdepodobnostného modelu. Mutácia je riadená pravdepodobnostným modelom a nové riešenia by mali spadnúť do sľubnej oblasti, ktorú charakterizuje pravdepodobnostný model.

Nech priestor hľadania je $\Omega = \{0, 1\}^n$. Pravdepodobnostný model reprezentovaný vektorom, ktorý charakterizuje rozdelenie sľubných riešení v priestore hľadania, označme $p = (p_1, \ldots, p_n) \in [0, 1]^n$, kde p_i je pravdepodobnosť, že hodnota na i-tej pozícii riešenia sa rovná 1. Operátor riadenej mutácie využíva pravdepodobnostný vektor p na mutovanie reťazca $x \in \{0, 1\}^n$ na reťazec $y \in \{0, 1\}^n$. Algoritmicky možno operátor riadenej mutácie zapísať nasledovne:

1. Hoď mincou tak, že s pravdepodobnosťou β padne hlava.
2. Ak padne hlava, tak s pravdepodobnosťou p_i nastav $y_i = 1$, inak $y_i = 0$.
3. Ak padne orol, tak nastav $y_i = x_i$.
4. Opakuj celý proces n-krát.

Podľa vyššie uvedeného operátora riadenej mutácie vidíme, že každý bit výsledného reťazca y_i sa buď priamo skopíruje od rodiča x_i alebo sa nastaví podľa pravdepodobnosti p_i na hodnotu 1 alebo 0. Čím je pravdepodobnosť β väčšia, tým viac elementov reťazca y sa vyberie podľa pravdepodobnostného vektora.

Reprezentácia a fitness

V navrhovanom evolučnom algoritme s riadenou mutáciou je každé potenciálne riešenie zakódované do binárneho reťazca. V každej generácii t sa udržuje populácia $Pop(t)$ pozostávajúca z N binárnych reťazcov a pravdepodobnostného vektora p.

Nech je zadaný graf $G = (V, E)$, kde $V = \{1, 2, \ldots, n\}$ je množina vrcholov a $E \subset V \times V$ je množina hrán. Množina vrcholov $U \subset V$ je zakódovaná ako reťazec $x = (x_1, x_2, \ldots, x_n) \in \{0, 1\}^n$, kde $x_i = 1$ práve vtedy, keď vrchol i je v U.

Ak reťazec x reprezentuje kliku, tak jeho fitness je definovaná ako kardinalita reťazca x. Keďže každé nové riešenie vygenerované operátorom riadenej mutácie bude opravované tak, aby reprezentovalo kliku, tak nie je nutné definovať hodnotu fitness pre neprípustné riešenia.

Rozdelenie priestoru hľadania

Priestor hľadania $\Omega = \{0, 1\}^n$ pre problém maximálnej kliky môže byť rozdelený do $n+1$ podpriestorov nasledovne:

$$\Omega = \Omega_0 \cup \Omega_1 \cup \cdots \cup \Omega_n$$

kde $\Omega_i = \left\{ x = (x_1, x_2, \ldots, x_n) : \sum_{j=1}^{n} x_j = i \right\}$. Očividne pre ľubovoľné Ω_i a Ω_j platí, že $\Omega_i \cap \Omega_j = \varnothing$. Každý reťazec v Ω_i reprezentuje množinu vrcholov s kardinalitou i.

Cieľom je nájsť maximálnu kliku v grafe. Ak sme našli maximálnu kliku s kardinalitou l, nie je viac potrebné prehľadávať podpriestory $\Omega_i, 1 \le i \le l$. Navyše, ak l bolo nájdené po dostatočne dlhom čase, nemá zmysel prehľadávať podpriestory Ω_i kde $i >> l$, pretože je nepravdepodobné, že maximálna klika bude oveľa väčšia ako l.

Navrhovaný algoritmus prehľadáva rôzne podpriestory v priestore hľadania počas rôznych fáz. Na začiatku sa vypočíta dolné ohraničenie *low* pre maximálnu kliku. Prvá fáza hľadania bude prebiehať v oblasti $\bigcup_{i=low+1}^{low+\Delta} \Omega_i$, kde Δ je zvolené neveľké celé číslo (napríklad 3). Ak sa nájde maximálna klika s väčšou kardinalitou s, tak súčasná fáza hľadania sa skončí a začne sa nová fáza hľadania v priestore $\bigcup_{i=s+1}^{s+\Delta} \Omega_i$.

Heuristická oprava riešení

Nové riešenie vygenerované operátorom riadenej mutácie nemusí byť klika. Na opravu riešenia autori aplikujú heuristiku z článku [20]. Uvažujme na vstupe množinu vrcholov $U \subset V$ a na výstupe kliku S v grafe G.

1. Vyextrahovanie kliky
    a) Priraď $W \leftarrow U$.
    b) Priraď $S \leftarrow U$.
    c) Ak $W = \emptyset$, choď na krok 2. Inak náhodne vyber vrchol k z množiny W a odstráň ho z W.
    d) Hoď mincou tak, že s pravdepodobnosťou α padne hlava.
    e) Ak padne hlava, odstráň vrchol k z množiny S. Inak odstráň z množín S a W všetky vrcholy v S, ktoré nie sú spojené s k.
    f) Choď na krok 1.c).
2. Rozšírenie kliky
    a) Priraď $W \leftarrow V \setminus S$.
    b) Ak $W = \emptyset$, vráť S. Inak náhodne vyber vrchol k z množiny W a odstráň ho z W.
    c) Ak vrchol k je spojený so všetkými vrcholmi v S, tak pridaj k do S.

d) Choď na krok 2.b).

Po vykonaní kroku 1, vo vyššie uvedenej oprave riešenia, množina S bude klika. Vo všeobecnosti by mala byť pravdepodobnosť α veľmi malá (napríklad 0,001), pretože inak by mohlo byť odstránených príliš veľa vrcholov z množiny S po 1. kroku algoritmu. Krok 2 jednoducho rozšíri množinu S na maximálnu kliku.

Pravdepodobnostný vektor

Pravdepodobnostný vektor p pre riadenú mutáciu sa musí v priebehu algoritmu inicializovať a aktualizovať.

Inicializácia

Predpokladajme, že súčasná populácia $Pop(t)$ obsahuje N binárnych reťazcov $x^j = (x_1^j, \ldots, x_n^j), \quad j = 1, 2, \ldots, N$. Pravdepodobnostný vektor $p = (p_1, \ldots, p_n)$ sa inicializuje nasledovne:

$$p_i = \frac{\sum_{j=1}^{N} x_i^j}{N}$$

Inicializácia prebieha na začiatku každej fázy hľadania alebo ak hľadanie musí byť reštartované. Prvok p_i pravdepodobnostného vektora p vyjadruje percentuálny podiel binárnych reťazcov s hodnotou 1 na i-tej pozícii v populácii $Pop(t)$.

Aktualizácia

V každej generácii t sa zo súčasnej populácie $Pop(t)$ vyberú niektoré binárne reťazce na vytvorenie množiny rodičov $Parent(t)$. Množina rodičov $Parent(t)$ sa následne použije na aktualizáciu pravdepodobnostného vektora p.

Nech množina $Parent(t)$ obsahuje M reťazcov $y^j = (y_1^j, \ldots, y_n^j), \quad j = 1, 2, \ldots, M$. Pravdepodobnostný vektor p sa aktualizuje podľa pravidla

$$p_i = (1-\lambda)p_i + \lambda \frac{\sum_{j=1}^{M} y_i^j}{M}$$

pre $i = 1, 2, \ldots, n$. Hodnota $\lambda \in (0, 1]$ sa nazýva rýchlosť učenia. Čím je rýchlosť učenia väčšia, tým viac reťazce v množine $Parent(t)$ prispievajú k aktualizovaným hodnotám pravdepodobnostného vektora. Ak $\lambda = 1$, globálne štatistické informácie zozbierané z minulosti nemajú žiadny vplyv na nový pravdepodobnostný vektor. Ako sa λ znižuje, príspevok štatistických informácií z minulosti sa zvyšuje.

Priebeh evolučného algoritmu

Navrhovaný evolučný algoritmus prebieha nasledovne:

1. Nastav nasledujúce parametre:
   a) Veľkosť populácie N
   b) Veľkosť priestoru hľadania Δ
   c) Rýchlosť učenia λ
   d) Pravdepodobnosť β pri riadenej mutácii
   e) Pravdepodobnosť α pri oprave riešenia
2. Nastav $t = 0$, náhodne vyber $x \in \Omega$ a aplikuj operátor opravy na x pre získanie maximálnej kliky U. Nastav $low = |U|$.
3. Náhodne vyber N reťazcov z priestoru $\bigcup_{i=low+1}^{low+\Delta} \Omega_i$ a aplikuj na každý z nich operátor opravy. N výsledných reťazcov vytvorí populáciu $Pop(t)$. Následne inicializuj pravdepodobnostný vektor p.
4. Vyber $N/2$ najlepších reťazcov z $Pop(t)$ na vytvorenie množiny rodičov $Parent(t)$ a aktualizuj pravdepodobnostný vektor p.

5. Aplikuj operátor riadenej mutácie $N/2$ krát na najlepší reťazec v $Parent(t)$ a následne aplikuj operátor opravy na získanie $N/2$ klík. Pridaj týchto $N/2$ klík do $Parent(t)$ za účelom vytvorenia $Pop(t+1)$. Ak sa splní podmienka zastavenia, vráť doposiaľ najväčšiu kliku.
6. Nastav $t = t+1$. Nech S je najväčšia klika v $Pop(t)$. Ak $|S| > low$, tak nastav $low = |S|$ a choď na krok 3.
7. Ak všetky reťazce v $Pop(t)$ sú identické, choď na krok 3, inak choď na krok 4.

Výsledky

Navrhnutý algoritmus autori overili na testovacej sade grafov DIMACS. Najskôr sa zamerali sa vplyv parametrov λ a β, taktiež uviedli prínos hlavných komponentov algoritmu a na záver porovnali svoj algoritmus s doposiaľ najlepším genetickým algoritmom na hľadanie maximálnej kliky, heuristickým genetickým algoritmom (HGA) [20].

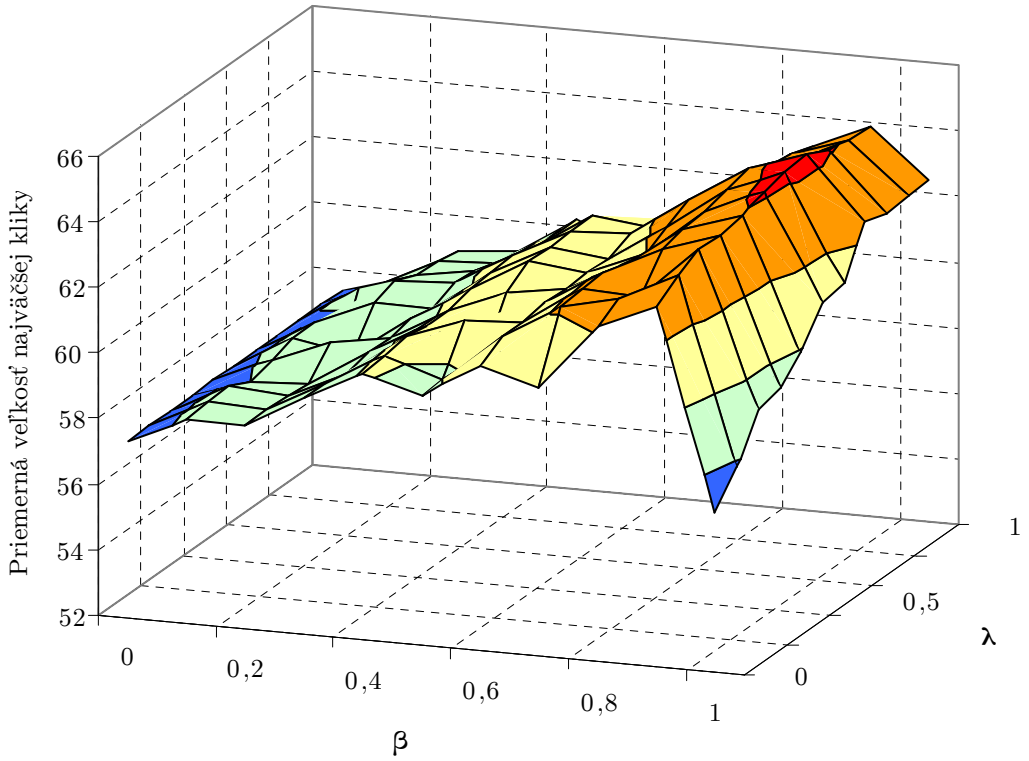
Vplyv vybraných parametrov

Autori porovnávajú vplyv rýchlosti učenia λ pri aktualizácii pravdepodobnostného vektora a vplyv parametra β, ktorý sa používa pri riadenej mutácii na určenie množstva elementov, ktoré sa vyberú podľa pravdepodobnostného vektora alebo sa skopírujú od rodiča.

Ak $\beta = 1$, tak sa všetky elementy reťazca určia podľa pravdepodobnostného vektora a žiadne sa priamo neskopírujú od rodiča. Ako môžeme vidieť na (Obr. 6), tak algoritmus s $\beta = 0{,}9$ dáva lepšie výsledky, čo dokazuje, že občasné priame kopírovanie elementov rodičov má pozitívny vplyv na výkonnosť algoritmu.

Rovnako sa podarilo ukázať, že uvažovanie predchádzajúcich hodnôt pravdepodobnostného vektora má zmysel pri jeho aktualizácii. Ak $\lambda = 1$, tak

informácie z minulosti nemajú vplyv na nové hodnoty pravdepodobnostného vektora. Avšak z grafu vidíme, že napríklad pre $\lambda = 0{,}7$ poskytoval algoritmus lepšie výsledky.



Obr. 6. Priemerné veľkosti najväčších klík počas 10 behov [30]

Prínos hlavných zložiek algoritmu

V tejto časti sa autori snažia zistiť, aký prínos majú hlavné zložky algoritmu, t.j. riadená mutácia a rozdelenie priestoru hľadania. Na porovnanie si autori vybrali HGA algoritmus, ktorý využíva rovnomerné kríženie a nevyužíva rozdelenie priestoru hľadania. Druhý porovnávaný algoritmus využíval rozdelenie priestoru hľadania, ale nevyužíval riadenú mutáciu.

Druhý algoritmus dokázal prekonať HGA algoritmus, čo potvrdzuje, že rozdeľovanie priestoru hľadania má zmysel. Zároveň bol druhý algoritmus porazený prezentovaným evolučným algoritmom, ktorý využíval rozdeľovanie

priestoru hľadania aj riadenú mutáciu. Z tohto výsledku vyplýva, že riadená mutácia poskytuje lepšie výsledky ako operátor rovnomerného kríženia. Tento výsledok nie je prekvapivý, nakoľko riadená mutácia využíva pravdepodobnostný vektor, ktorý sa v priebehu algoritmu aktualizuje.

Porovnanie s heuristickým genetickým algoritmom

Autori porovnávajú svoj algoritmus s doposiaľ najlepším evolučným algoritmom na hľadanie maximálnej kliky, ktorým je HGA. Porovnanie prebiehalo na 37 grafoch zo sady DIMACS.

Prezentovaný evolučný algoritmus dokázal prekonať HGA algoritmus na 31 z 37 grafov, pokiaľ ide o priemernú veľkosť nájdených klík. Na 13 grafoch našiel evolučný algoritmus s riadenou mutáciou väčšiu kliku ako HGA. V 30 prípadoch bol algoritmus schopný nájsť najlepšie doposiaľ známe riešenie.

Čas behu prezentovaného algoritmu je o 16% dlhší ako HGA, hlavne kvôli riadenej mutácii a rozdeľovaniu priestoru hľadania. Avšak dlhší čas behu je jasne vykompenzovaný kvalitou dosiahnutých riešení.

5.4 Zhodnotenie stavu poznania

Ako vidno z analyzovaných článkov, výskumníci sa niekoľko posledných rokov usilujú o generovanie optimálnych samoopravných kódov prostredníctvom evolučných algoritmov. Vzhľadom na to, že problém hľadania optimálnych samoopravných kódov je ekvivalentný problému hľadania maximálnej kliky v grafe, nezačína výskum v tejto oblasti nanovo, ale stavia na objavených, vyskúšaných a overených algoritmoch hľadania kliky v grafe.

Súčasný stav poznania je taký, že výskumníci sa snažia nájsť najlepšiu evolučnú heuristiku pre generovanie samoopravných kódov. Navrhujú, implementujú a testujú rôzne metódy prehľadávania stavového priestoru od lokálnych greedy optimalizácií až po genetické algoritmy a genetické programovanie.

Z existujúcich prístupov je zrejmé, že štandardný genetický algoritmus nie je vhodný na tento problém a musí byť doplnený o ďalšie heuristiky (Conwayov greedy algoritmus, riadená mutácia, rozdelenie priestoru hľadania), ktoré zlepšia jeho výkon pre tento konkrétny problém.

Žiadny z uvedených prístupov nedominuje jednoznačne nad ostatnými. Taktiež v žiadnej z analyzovaných prác sa nepodarilo nájsť nový samoopravný kód s väčším počtom kódových slov, ako je v súčasnosti známe. Napriek tomu sa v článkoch uvádzajú mnohé úspešné metódy, ktorých kombináciou by mohlo vzniknúť zaujímavé riešenie.

KAPITOLA 6

Návrh riešenia

„Výsledky vlastného premýšľania sú hodnotnejšie ako všetka získaná cudzia múdrosť.“

~ Carl Friedrich Gauss (1777 - 1855)

Táto časť obsahuje návrh vlastného riešenia, ktoré bude porovnávané s vybranými existujúcimi metódami. Na úvod sa popisujú dôležité vzťahy v teórii kódovania, ktoré nám umožnia problém lepšie uchopiť.

6.1 Prehľadávaný priestor

Pri konštrukcii samoopravných kódov môžeme vždy predpokladať existenciu nulového slova v kóde. Platí, že každý kód neobsahujúci nulové slovo je ekvivalentný kódu, ktorý ho obsahuje [10]. Preto namiesto uvažovania všetkých binárnych slov dĺžky n, môžeme medzi kandidátov na riešenie zaradiť iba tie kódové slová dĺžky n, ktorých vzdialenosť je aspoň d od nulového slova.

Vzdialenosť aspoň d od nulového slova znamená, že kandidátne kódové slová musia mať aspoň d jednotkových bitov. Počet slov dĺžky n, ktorých vzdialenosť od nulového slova je aspoň d možno vyčísliť ako

$$\sum_{i=d}^{n} \binom{n}{i}$$

Tým, že každé kandidátne kódové slovo môže do optimálneho samoopravného kódu buď patriť alebo nepatriť, zložitosť prehľadávania je exponenciálna v závislosti od počtu kandidátov.

6.2 Ekvivalencia kódov

Medzi parametrami samoopravných kódov platia určité vlastnosti, ktoré nám umožnia výrazne zredukovať priestor hľadania binárnych kódov.

6.2.1 Dôkaz

Uvažujme binárny $(n, M, 2r-1)$ kód s M kódovými slovami dĺžky n a s minimálnou vzdialenosťou $2r-1$. Pridaním jedného bitu na kontrolu parity vznikne $(n+1, M, 2r)$ kód, takže pre maximálny možný počet kódových slov platí $A_2(n, 2r-1) \leq A_2(n+1, 2r)$.

Poznamenajme, že kontrola parity pridá 1 bit na koniec každého kódového slova. Bit kontroly parity je 1, ak zvyšná časť kódového slova má nepárny počet jednotiek. Naopak bit kontroly parity je 0, ak zvyšná časť kódového slova má párny počet jednotiek. Dôsledok je, že každé kódové slovo má po novom párny počet jednotiek a minimálna vzdialenosť medzi každým párom kódových slov je párna.

Teraz uvažujme binárny $(n+1, M, 2r)$ kód. Odstránením jedného bitu dostaneme (n, M, d) kód, kde $d \geq 2r-1$. Preto získame nasledovný vzťah $A_2(n, 2r-1) \geq A_2(n+1, 2r)$.

Keďže $A_2(n, 2r-1) \leq A_2(n+1, 2r)$ a súčasne $A_2(n, 2r-1) \geq A_2(n+1, 2r)$ vo výsledku dostaneme, že $A_2(n, 2r-1) = A_2(n+1, 2r)$. Na záver vidíme, že kód

dĺžky n s minimálnou vzdialenosťou $2r-1$ má menej možných kódových slov ako kód dĺžky $n+1$ s minimálnou vzdialenosťou $2r$, takže priestor hľadania sa zredukuje, zatiaľ čo počet kompatibilných kódových slov zostane zachovaný [19].

6.2.2 Príklad

Uvažujme príklad, kde $A_2(10, 4) = A_2(9, 3) = 40$. Aplikovaním vzťahu o počte kandidátnych kódových slov dostaneme, že existuje 848 slov dĺžky 10, ktorých vzdialenosť je aspoň 4 od nulového slova. Tieto slová predstavujú kandidátov na riešenie v kóde $(10, 40, 4)$. V druhom prípade existuje iba 466 kódových slov dĺžky 9 s minimálnou vzdialenosťou 3 od nulového slova. V oboch prípadoch pri hľadaní optimálneho samoopravného kódu musíme nájsť 40 kompatibilných kódových slov. Je prirodzené, že jednoduchšie bude hľadať v menšej množine kandidátov.

Preto pre ľubovoľný $(n, 2r)$ kód môžeme hľadať riešenie vo výrazne menšom priestore prehľadávania kódu $(n-1, 2r-1)$.

6.3 Štruktúra grafov

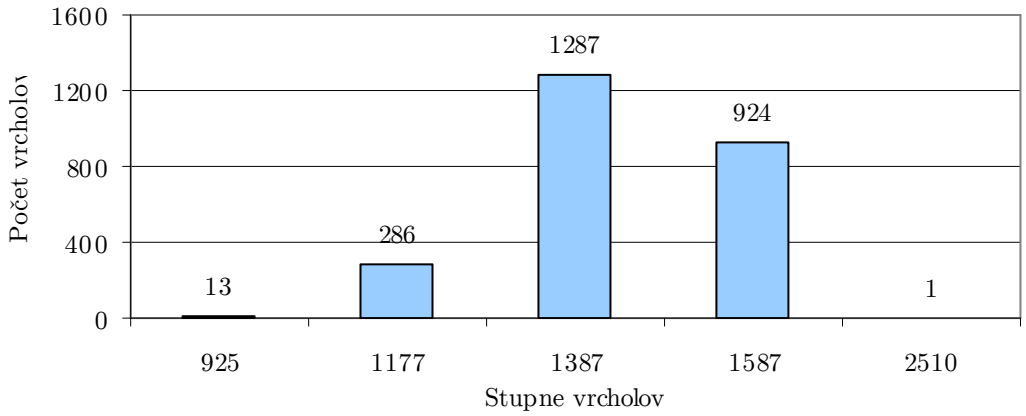
Pre lepšie uchopenie problému je dôležité zistiť, akú štruktúru grafov tvoria samoopravné kódy. Uvažujme parametre binárneho kódu $(12, 6)$. Kódové slová v tomto kóde majú dĺžku 12 bitov a každá dvojica slov má Hammingovu vzdialenosť aspoň 6. Počet kandidátnych kódových slov je nasledovný:

$$\sum_{i=6}^{12}\binom{12}{i} = \binom{12}{6}+\binom{12}{7}+\binom{12}{8}+\binom{12}{9}+\binom{12}{10}+\binom{12}{11}+\binom{12}{12} = 2510$$

Ak aj nulové slovo budeme považovať za kandidáta, výsledok bude ešte o 1 väčší. Počet kandidátov vlastne predstavuje počet vrcholov v grafe. Hrana bude medzi každými dvomi vrcholmi, ktorých kódové slová majú Hammingovu

vzdialenosť minimálne 6. V tomto kóde je 1 801 307 takých dvojíc, a to je teda aj počet hrán v grafe.

Okrem toho, že sa jedná o pomerne veľký a hustý graf, je tento graf špecifický aj stupňami vrcholov. Nasledujúci obrázok (Obr. 7) znázorňuje početnosť stupňov vrcholov. Vidíme, že je veľmi veľa vrcholov rovnakého stupňa a rôznorodosť stupňov vrcholov je pomerne malá.



Obr. 7. Početnosť stupňov vrcholov v grafe pre kód (12, 6)

Len pre ilustráciu, najväčší kód $(17, 4)$ má 130 239 kandidátov, no vďaka ekvivalencii kódov môžeme hľadať riešenie v kóde $(16, 3)$, ktorý má 65 400 kandidátov, čím sme priestor hľadania zredukovali zhruba na polovicu. Graf takéhoto kódu má 2 134 107 308 hrán a po uložení do súboru vo formáte DIMACS má tento súbor takmer 30 GB. Toto sú hodnoty, ktoré jasne dokazujú veľkosť a hustotu grafov, s ktorými máme dočinenia.

Počet kandidátnych kódových slov (počet vrcholov) a počet kompatibilných dvojíc slov (počet hrán) pre testovacie kódy je znázornený v (Tab. 6).

Tab. 6. Veľkosti grafov pre testovacie kódy

n	d	Vrcholov v (n, d)	Hrán v (n, d)	Vrcholov v $(n\text{-}1, d\text{-}1)$	Hrán v $(n\text{-}1, d\text{-}1)$
12	6	2 511	1 801 307	1 487	774 643
13	6	5 813	11 617 984	3 303	4 334 915
17	6	121 671	6 863 818 562	63 020	1 908 848 638
17	4	130 239	8 427 084 434	65 400	2 134 107 308

Vzhľadom na parametre a špecifické vlastnosti týchto grafov, nedá sa predpokladať, že štandardné riešenia na hľadanie maximálnej kliky v grafe by priamočiaro fungovali aj pre túto doménu.

6.4 Lexikografické kódy

Lexikografické kódy sú samoopravné kódy so zvolenou dĺžkou kódových slov a s minimálnou vzdialenosťou s mimoriadne dobrými vlastnosťami. Navrhol ich Levenshtein [18] a nezávisle Conway a Sloane [3]. Lexikografické kódy sú generované greedy algoritmom v lexikografickom (abecednom, resp. slovníkovom) poradí.

6.4.1 Algoritmus generovania

Algoritmus pre generovanie lexikografických kódov je nasledovný:

1. Inicializuj si prázdnu množinu kódových slov C
2. Usporiadaj kódové slová kandidátov lexikograficky
3. Pre každého kandidáta:

a) Ak je vzdialenosť kandidáta od všetkých kódových slov v C aspoň d, pridaj kandidáta do C

b) Ak je vzdialenosť kandidáta od ľubovoľného kódového slova v C menšia ako d, zamietni kandidáta

6.4.2 Ukážka generovania

Ukážka generovania lexikografického kódu dĺžky 3 s minimálnou vzdialenosťou 2 je znázornená v (Tab. 7).

Tab. 7. Generovanie lexikografického kódu

Kódové slovo kandidáta	Patrí do kódu
000	Áno
001	Nie
010	Nie
011	Áno
100	Nie
101	Áno
110	Áno
111	Nie

Výsledný lexikografický kód dĺžky 4 s minimálnou vzdialenosťou 3 medzi každým párom slov pozostáva zo 4 slov, $C = \{000, 011, 101, 110\}$.

Dôležité pozorovanie je, že všetky binárne lexikografické kódy sú lineárne. Pre mnohé parametre samoopravných kódov, sú práve lineárne kódy optimálne. Takže pomocou tohto algoritmu sa dá pre takéto kódy dosiahnuť dolné ohraničenie na maximálny počet kódových slov. Vzhľadom na to, že tento

algoritmus je deterministický, nie je možné pomocou neho dolné ohraničenie vylepšiť.

Mnohé známe samoopravné kódy patria tiež do skupiny lexikografických kódov. To znamená, že môžu byť vygenerované lexikografickým spôsobom. Medzi takéto kódy patria Hammingove kódy a binárne Golayove kódy.

6.5 Stochasticky generované kódy

Štandardný greedy algoritmus pre generovanie lexikografických kódov dokáže vygenerovať dobrú aproximáciu optimálnych kódov. Avšak tým, že je deterministický, nedokáže sa optimálnym kódom viac priblížiť. Taktiež nezáleží na tom, koľkokrát sa spustí, pretože pre dané parametre vráti vždy rovnaký výsledok.

6.5.1 Existujúce riešenie

V článku [10] autori uvažujú modifikovanú verziu pôvodného lexikografického algoritmu navrhnutého Conwayom. Na začiatku pridajú do kódu 3 kompatibilné kódové slová a následne spustia verziu lexikografického algoritmu, v ktorej sa ďalšie kódové slová pridávajú v náhodnom poradí. Autori si uvedomujú, že týmto postupom môžu skonštruovať nelineárny kód a zlepšiť tak doposiaľ známe dolné ohraničenie.

Podobný prístup bol aplikovaný aj v článku [1], kde autori použili stochastickú verziu lexikografického algoritmu na inicializovanie populácie v genetickom algoritme.

6.5.2 Vlastný návrh

V článku o porovnaní evolučných algoritmov pre hľadanie optimálnych samoopravných kódov [10] bol najúspešnejší genetický algoritmus so

záverečným doplnením kompatibilných kódových slov, ktoré neboli vybraté najlepším jedincom. Na toto záverečné doplnenie používali lexikografický algoritmus.

Motivujúca myšlienka

Pri pozornejšom skúmaní článku zistíme, že veľkosť jedinca pri nepriamej reprezentácii bola obmedzená parametrom *seed_size*. Autori zistili, že čím je hodnota tohto parametra menšia, tým lepšie výsledky táto metóda poskytovala. Aj pri najväčšom testovacom kóde $(17, 4)$ so 65 399 kandidátnymi kódovými slovami, kde doposiaľ najlepšie známe riešenie obsahuje 2 720 kódových slov, veľkosť jedinca bola obmedzená na 6 kódových slov. Inými slovami, iba 6 kódových slov bolo vybratých genetickým algoritmom a 2 232 zvyšných kódových slov v najlepšom dosiahnutom riešení bolo doplnených greedy lexikografickým algoritmom. Autori priznávajú, že väčšina kódových slov bola doplnená greedy algoritmom v záverečnej fáze a nie genetickým algoritmom.

Nápady

Tým, že genetický algoritmus v popisovanej metóde nie je príliš prospešný pre tvorbu optimálneho samoopravného kódu, logický podnet je pokúsiť sa genetický algoritmus nahradiť niečím iným, výpočtovo úspornejším.

Náhodná inicializácia

Efektívnejšie môže byť inicializovanie množiny vybratých kódových slov C niekoľkými náhodnými kompatibilnými kódovými slovami a spustenie greedy lexikografického algoritmu. Inými slovami, uvažovať budeme pôvodný algoritmus na generovanie lexikografických kódov a stochasticitu zabezpečíme tak, že mu pred spustením vnútime niekoľko náhodne vybratých kompatibilných slov do výslednej množiny C.

Multištart

Tým, že v tejto metóde sa snaží greedy algoritmus doplniť náhodne vnútené kódové slová do optimálneho kódu, nemusí vždy vrátiť rovnaký výsledok na výstupe. Výsledok totiž závisí od toho, aké náhodne vybraté slová mu boli vložené do množiny C. Preto má zmysel algoritmus spustiť viackrát a z tohto dôvodu zavádzame multištart.

Horolezecký algoritmus

Ďalšie vylepšenie spočíva v zavedení horolezeckého prístupu. Množinu vybraných kódových slov C inicializujeme náhodnými kompatibilnými kódovými slovami iba prvýkrát a ďalej sa budeme snažiť optimalizovať túto množinu. Cieľom je nájsť také kódové slová, ktoré greedy algoritmus dokáže doplniť do čo najväčšieho kódu. Namiesto optimalizovania výsledného kódu budeme optimalizovať vybrané inicializačné slová do množiny C. Týmto prístupom sa snažíme ešte viac pomôcť greedy algoritmu vyberať do kódu správne kódové slová.

Na začiatku inicializujeme množinu vybratých kódových slov C niekoľkými náhodnými kompatibilnými kódovými slovami. Táto množina bude prestavovať počiatok oblasti, z ktorej budeme ďalej viesť horolezecký algoritmus. Okolie budeme prehľadávať tak, že každé jedno kódové slovo v množine C nahradíme iným náhodne vybratým kódovým slovom, ktoré je kompatibilné so zvyšnými vybratými slovami, aby sme dodržali požiadavku na minimálnu vzdialenosť. Následne pre každé takéto okolie spustíme greedy algoritmus, ktorý sa pokúsi náhodne vybraté slová doplniť do čo najväčšieho kódu v lexikografickom poradí. Množina vybratých slov, ktoré po doplnení greedy algoritmom vygenerujú najväčší kód, sa zvolia ako stred hľadania v ďalšej iterácii algoritmu.

Tým, že okolie generujeme postupným nahradením každého slova iným náhodne vybratým kódovým slovom, jedná sa o stochastický horolezecký algoritmus. Na rozdiel od deterministicky generovaného okolia je takto menšia pravdepodobnosť

uviaznutia v lokálnom extréme. Kardinalita okolia sa rovná počtu náhodne inicializovaných slov v množine C.

Rozdiel oproti predošlým riešeniam

Rozdiel oproti existujúcim riešeniam popísanými vyššie je v tom, že v nich sa používa modifikácia lexikografického algoritmu, kde sa nové kódové slová uvažujú v náhodnom poradí, čím sa podľa nás vytratila pôvodná greedy myšlienka, ktorá poskytovala dobré výsledky. Naše riešenie stále využíva greedy myšlienku pridávania kódových slov v lexikografickom poradí, len s tým, že niektoré náhodne vybraté kódové slová sa najskôr vložia do výslednej množiny a úlohou greedy algoritmu je pokúsiť sa doplniť zvyšné kódové slová do optimálneho riešenia. Ďalším rozdielom je zavedenie multištartu, čím sa celý proces viackrát zopakuje a na výstup sa pošle najlepšie doposiaľ nájdené riešenie.

Skombinovanie horolezeckého algoritmu a greedy algoritmu pre generovanie lexikografických kódov možno považovať za inovatívny prístup, ktorý nebol doposiaľ vo vedeckých článkoch vyskúšaný. Vzhľadom na rozsah prehľadávaného priestoru sa dá predpokladať, že takéto riešenie by mohlo poskytnúť podobné výsledky ako kombinácia genetického algoritmu a greedy algoritmu v kratšom čase.

Očakávania

Predpokladáme, že kľúčovým parametrom bude stanoviť počet kódových slov, ktoré náhodne vyberieme a pridáme do výslednej množiny kompatibilných kódových slov.

Ak ich pridáme príliš málo, tak tým pôvodný deterministický greedy algoritmus na generovanie lexikografických kódov nemusíme príliš ovplyvniť. Je väčšia pravdepodobnosť, že pôvodný greedy algoritmus si vyberie do výsledného riešenia niektoré jedno kódové slovo, ako že si vyberie niektoré dve kódové slová.

Zvyšovaním počtu náhodne pridaných slov do riešenia, viacej ovplyvňujeme a obmedzujeme pôvodný greedy algoritmus. Deterministický greedy algoritmus už nemôže ísť podľa svojho vopred pripraveného scenára a musí sa vysporiadať s náhodnými kódovými slovami vo výslednej množine kompatibilných kódových slov. Pripomeňme, že pri lexikografickom generovaní samoopravných kódov sa nové kódové slovo pridá do riešenia iba vtedy, ak je Hammingova vzdialenosť kandidáta od všetkých doposiaľ uložených kódových slov v riešení aspoň minimálna požadovaná.

Ak náhodne vybratých kódových slov pridáme do výslednej množiny veľmi veľa, môžeme tým greedy algoritmus príliš obmedziť a pôvodnú greedy myšlienku zničiť. Greedy algoritmus potom nemusí nájsť riešenie blízke optimu, pretože je menšia pravdepodobnosť, že náhodne vyberie 5 kódových slov z prehľadávaného priestoru patriacich do optimálnej množiny, ako že vyberie 1 kódové slovo patriace do optimálnej množiny.

6.5.3 Algoritmy pre maximálnu kliku

Na problém riešený v tejto práci je možné nazerať aj z pohľadu teórie grafov. Preto by bolo zaujímavé vyskúšať aj generické algoritmy na hľadanie maximálnej kliky v grafe a výsledky porovnať. Opäť sa zameriame na greedy metódy, ktoré dokážu vrátiť výsledok v reálnom čase.

Vzhľadom na veľkosť grafov samoopravných kódov a ich štruktúru sa nedá očakávať, že algoritmy ktoré fungovali spoľahlivo na náhodne generovaných DIMACS grafoch, budú automaticky fungovať aj pre naše účely.

Jednoduchý greedy algoritmus

Jednoduchý greedy algoritmus je založený na myšlienke postupného pridávania susedných vrcholov s najvyšším stupňom. Algoritmus si na začiatku vyberie náhodný počiatočný vrchol a pridá ho do množiny vrcholov tvoriacich kliku. Následne si algoritmus vyrobí množinu potenciálnych vrcholov na pridanie. Pri

pridávaní do tejto množiny sa prechádzajú tie vrcholy, ktoré susedia s vrcholmi už pridanými do kliky, ale zaradia sa iba tie, ktoré susedia so všetkými pridanými, aby bolo splnené klikové kritérium. Z množiny potenciálnych vrcholov na pridanie sa napokon do kliky pridá ten s najvyšším stupňom. Tento proces sa opakuje dovtedy, dokedy je možné pridať nejaký vrchol.

Modifikovaný greedy algoritmus

Tento algoritmus vlastne predstavuje modifikáciu predchádzajúceho jednoduchého greedy algoritmu. Na rozdiel od neho ale obsahuje viaceré vylepšenia. Okrem množiny potenciálnych vrcholov na pridanie (susedné vrcholy už pridaných vrcholov do kliky, ktoré sú prepojené so všetkými vrcholmi v klike), si algoritmus vytvára aj množinu vrcholov, ktoré sú prepojené so všetkými okrem jedného vrcholu tvoriacich kliku. Táto množina sa využije pri odstraňovaní vrcholov z kliky [21].

V prípade, že už nebude možné rozšíriť kliku o ďalší vrchol, príde na rad odstránenie toho vrcholu z kliky, ktorého odstránením sa maximalizuje množina potenciálnych vrcholov na pridanie. Algoritmus takýmto spôsobom pridáva a odstraňuje vrcholy, pričom si sleduje, ako dlho nenastal žiadny pokrok, t.j. nenašiel väčšiu kliku. Ak táto hodnota prekročí stanovený limit, algoritmus sa sám reštartuje a začne opäť náhodným výberom počiatočného vrcholu. Limit na reštartovanie sme nastavili na dvojnásobok najväčšej doposiaľ nájdenej kliky.

Tabu algoritmus

Tabu algoritmus funguje na podobnom princípe ako modifikovaný greedy algoritmus popísaný vyššie s tým rozdielom, že si udržuje tabu zoznam nedávno pridaných, resp. odobraných vrcholov z kliky. Vrcholy v tomto zozname budú vylúčené z pridávania, resp. odoberania niekoľko najbližších krokov algoritmu.

Zaujímavosťou tohto algoritmu je dynamické prispôsobovanie veľkosti tabu zoznamu. Veľkosť tabu zoznamu sa dynamicky adaptuje podľa nájdených riešení. Algoritmus si udržiava množinu nájdených klík. Pokiaľ sa riešenia často

opakujú, veľkosť tabu zoznamu sa zväčší. Pokiaľ sa riešenia neopakujú, veľkosť tabu zoznamu sa zmenší, aby bolo možné vyberať z viacerých vrcholov [21].

KAPITOLA 7

Implementácia

„Testovaním programu môžete dokázať prítomnosť chýb, nie ich neprítomnosť.“

~ Edsger Dijkstra (1930 - 2002)

Vytvorený program sme implementovali podľa návrhu prezentovaného v predošlej kapitole. Okrem samotnej metódy pre stochastické generovanie optimálnych kódov založenej na generovaní lexikografických kódov a horolezeckom algoritme, sme implementovali aj niekoľko ďalších metód. Menovite genetický algoritmus s nepriamou reprezentáciou jedincov a lexikografickým doplnením kódových slov podľa článku [10], brute force riešenie použiteľné len pre malé inštancie kódov a knižnicu, ktorá obsahuje často používané funkcie, ktoré zdieľajú všetky algoritmy.

Riešenie je implementované v programovacom jazyku C pod operačným systémom Windows NT. Počas implementácie nás podporovalo integrované vývojové prostredie Visual Studio.

7.1 Generovanie lexikografických kódov

Uvedený pseudokód zobrazuje implementáciu deterministického generovania lexikografického kódu pre nasledovné parametre:

1. n – dĺžka kódových slov
2. d – minimálna vzdialenosť kódu
3. *code* – výsledný kód (množina kompatibilných kódových slov)

```
function GenerateLexicode(in n, in d, out code)
begin
    maxWord ← 2^n – 1
    for i ← 0 to maxWord do
        for j ← 0 to size(code) - 1 do
            if hammingDistance(i, code[j]) < d then
                break
            end
        end
        if j = size(code) then
            add(i, code)
        end
    end
end
```

7.2 Stochastické generovanie kódov

Nasleduje ukážka mnou navrhnutého stochastického generovania kódov založeného na algoritme generovania lexikografických kódov. Funkcia očakáva na vstupe nasledovné parametre:

1. n – dĺžka kódových slov

2. *d* – minimálna vzdialenosť kódu
3. *initRandomly* – počet náhodne nainicializovaných kódových slov
4. *code* – výsledný kód (množina kompatibilných kódových slov)

```
function GenerateStochastically(in n, in d, in initRandomly, out code)
begin
    maxNumber ← 2^n
    code ← {}
    i ← 0
    while i < initRandomly do
        candidate ← random() mod maxNumber
        for j ← 0 to size(code) - 1 do
            if hammingDistance(candidate, code[j]) < d then
                break
            end
        end
        if j = size(code) then
            add(candidate, code)
            i ← i + 1
        end
    end
    GenerateLexicode(n, d, code)
end
```

7.3 Použitie multištartu

Aplikovanie multištartu môžeme vidieť na nasledovnom pseudokóde. Funkcia požaduje na vstupe nasledovné parametre:

1. n – dĺžka kódových slov
2. d – minimálna vzdialenosť kódu
3. *initRandomly* – počet náhodne nainicializovaných kódových slov
4. *iterations* – počet iterácií multištartu
5. *code* - výsledný kód (množina kompatibilných kódových slov)

```
function GenerateIteratively(in n, in d, in initRandomly, in iterations, out code)
begin
    maxSize ← 0
    code ← {}
    for i ← 0 to iterations – 1 do
        codeNow ← {}
        GenerateStochastically(n, d, initRandomly, codeNow)
        if size(codeNow) > maxSize then
            code ← codeNow
            maxSize ← size(codeNow)
        end
    end
end
```

7.4 Horolezecký algoritmus

Na nasledovnom pseudokóde vidíme ukážku stochastického generovania okolia horolezeckým algoritmom a výber najlepšieho riešenia, ktoré sa použije ako stred oblasti hľadania v ďalšej iterácii algoritmu. Funkcia očakáva na vstupe nasledovné parametre:

1. n – dĺžka kódových slov
2. d – minimálna vzdialenosť kódu
3. *startSet* – počiatočná množina slov, z ktorej sa vedie hľadanie
4. *code* – výsledný kód (množina kompatibilných kódových slov)

```
function ScanSurrounding(in n, in d, inout startSet, out code)
begin
    maxNumber ← 2^n
    code ← {}
    bestSet ← {}
    maxSize ← 0
    for i ← 0 to size(startSet) – 1 do
        initSet ← startSet
        remove(startSet[i], initSet)
        candidateFound ← false
        while not candidateFound do
            candidate ← random() mod maxNumber
            for j ← 0 to size(initSet) - 1 do
                if hammingDistance(candidate, initSet[j]) < d then
```

```
                    break
                end
            end
            if j = size(initSet) then
                add(candidate, initSet)
                candidateFound ← true
            end
        end
        codeNow ← initSet
        GenerateLexicode(n, d, codeNow)
        if size(codeNow) > maxSize then
            maxSize ← size(codeNow)
            bestSet ← initSet
            code ← codeNow
        end
    end
    startSet ← bestSet
end
```

7.5 Použité technológie

Výsledné riešenie využíva tie najmodernejšie technológie s cieľom, čo najviac zrýchliť beh algoritmu. Nižšie spomenuté technologické vylepšenia síce nemajú za následok rýchlejšiu optimalizáciu riešenia, ani negenerujú väčší samoopravný kód, ale vďaka nim sa niekoľkonásobne skrátil čas behu jednotlivých

algoritmov, čo umožnilo vykonať viacero pokusov a v konečnom dôsledku dospieť k štatisticky presnejším výsledkom.

7.5.1 Generátor pseudonáhodných čísel

Základným stavebným kameňom každej stochastickej optimalizačnej metódy je kvalitný generátor pseudonáhodných čísel.

Mersenne Twister

Mersenne Twister je najpoužívanejší generátor pseudonáhodných čísiel. Navrhnutý bol v roku 1997 a v tej dobe to bol prvý generátor, ktorý poskytoval rýchle generovanie vysoko kvalitných pseudonáhodných čísiel.

Jeho názov je odvodený podľa dĺžky periódy, ktorá je Mersennovo prvočíslo. Mersennovo prvočíslo je také prvočíslo, ktoré je o 1 menšie ako celočíselná mocnina dvojky, takže sa dá zapísať v tvare

$$M_n = 2^n - 1$$

kde n je prirodzené číslo. Prvé štyri Mersennove prvočísla sú 3, 7, 31 a 127. Pomenované sú podľa francúzskeho mnícha Marina Mersenna, ktorý ich študoval v 17. storočí.

Najpoužívanejšia verzia Mersenne Twister generátora pseudonáhodných čísel je založená na Mersennovom prvočísle $2^{19937} - 1$. Označuje sa ako *MT19937* a produkuje postupnosť 32 bitových celých čísel.

V súčasnosti je známych 48 Mersennových prvočísel. Najväčšie známe Mersennovo prvočíslo $\left(2^{57\,885\,161} - 1\right)$ bolo objavené v januári 2013. Všetky nové Mersennove prvočísla od roku 1997 boli objavené vďaka distribuovanému projektu na Internete GIMPS (Great Internet Mersenne Prime Search).

Streaming SIMD Extensions 2

Streaming SIMD Extensions 2 (SSE2) predstavuje rozširujúcu inštrukčnú sadu procesorov obsahujúcu vektorové inštrukcie. Inštrukčná sada SSE2 bola uvedená spoločnosťou Intel v roku 2001 v procesoroch Pentium 4. Jej účelom je ďalej rozšíriť predchádzajúcu inštrukčnú sadu SSE a úplne nahradiť inštrukčnú sadu MMX.

Na základe Flynnovej taxonómie architektúry počítačov rozlišujeme 4 triedy architektúr podľa počtu súbežných inštrukčných a dátových tokov:

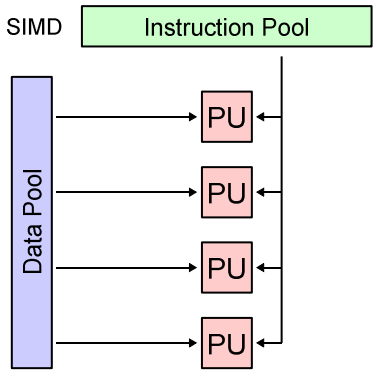
a) SISD – Single Instruction, Single Data stream

b) SIMD – Single Instruction, Multiple Data stream

c) MISD – Multiple Instruction, Single Data stream

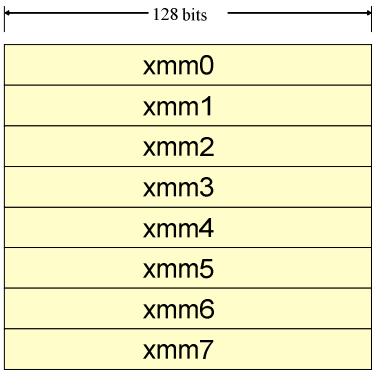
d) MIMD – Multiple Instruction, Multiple Data stream

SIMD architektúra predstavuje počítače, ktoré využívajú viacero dátových tokov na jednom toku inštrukcií, čo umožňuje vykonávať operácie, ktoré môžu byť prirodzene paralelizované (Obr. 8).



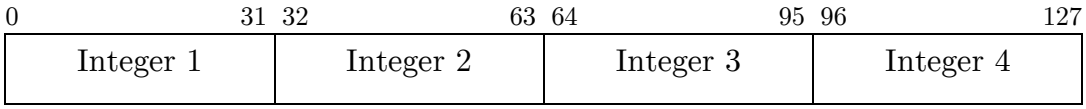
Obr. 8. SIMD architektúra

Inštrukčná sada SSE2 využíva 128 bitové registre procesora označované ako XMM0 – XMM7, ktoré boli do procesorov zakomponované s príchodom inštrukčnej sady SSE (Obr. 9).



Obr. 9. Osem 128 bitových registrov v procesore s podporou SSE

Väčšina z SSE2 inštrukcií implementuje vektorové operácie nad číslami s pohyblivou rádovou čiarkou. Ďalej SSE2 sada obsahuje aritmetické a bitové inštrukcie pre spracovanie viacerých celých čísel v niektorom z XMM registrov. Vďaka šírke XMM registrov je možné na 1 špeciálnu inštrukciu spracovať až 4 celé 32 bitové čísla naraz:



Podpora SSE2 inštrukčnej sady v procesore je indikovaná 26. bitom v registri EDX po vykonaní *CPUID* inštrukcie. Pred použitím špeciálnych vektorových SSE2 inštrukcií a XMM registrov je nevyhnutné podporu sady SSE2 v procesore skontrolovať, pretože inak by mohlo zavolanie SSE2 inštrukcie na nepodporovanom procesore spôsobiť nedefinované správanie.

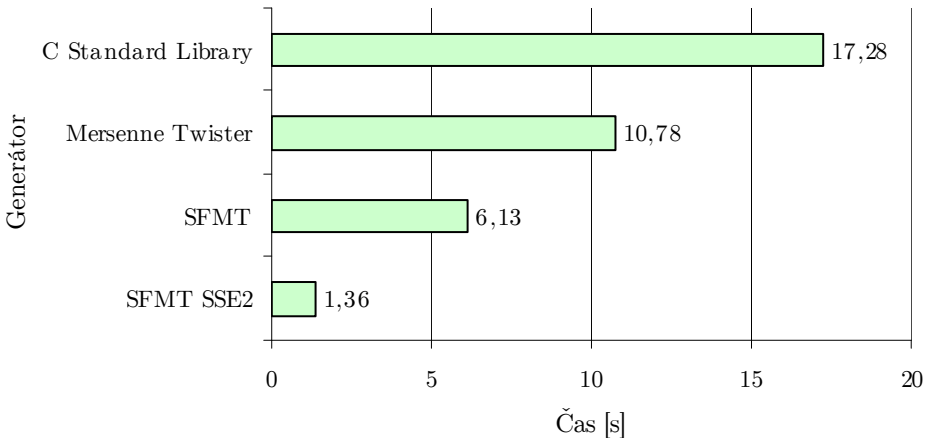
SIMD-oriented Fast Mersenne Twister

SIMD-oriented Fast Mersenne Twister (SFMT) je nový variant Mersenne Twister generátora pseudonáhodných čísel využívajúci SIMD inštrukcie procesora z inštrukčnej sady SSE2. Predstavený bol v roku 2006 v článku [27].

Cieľom tohto algoritmu je dosiahnuť vyšší výkon v porovnaní so štandardnou implementáciou Mersenne Twister generátora. Podľa autorov je SFMT variant

približne 2x rýchlejší ako Mersenne Twister. Navyše bol tento algoritmus od začiatku navrhovaný pre využitie vektorových SIMD inštrukcií moderných procesorov a viacúrovňového prúdového spracovania, čím sa dosiahlo ešte väčšie zrýchlenie. Okrem rýchlosti vyniká aj ďalšími pozitívnymi štatistickými vlastnosťami.

Na nasledujúcom grafe (Obr. 10) vidíme porovnanie rýchlosti jednotlivých generátorov pseudonáhodných čísel. Úlohou porovnávaných algoritmov bolo vygenerovať 1 miliardu pseudonáhodných čísel. Najpomalší bol lineárny kongruentný generátor reprezentovaný knižničnou funkciou *rand()* jazyka C. Približne dvojnásobné zrýchlenie algoritmu SFMT voči Mersenne Twister generátoru môžeme potvrdiť. Vyše štvornásobné zrýchlenie sa dá ďalej dosiahnuť využitím vektorových SIMD inštrukcií v moderných procesoroch.



Obr. 10. Porovnanie rýchlosti generátorov pseudonáhodných čísel

7.5.2 Výpočet Hammingovej váhy

Nami navrhované algoritmy vo veľkej miere využívajú greedy algoritmus pre generovanie lexikografických kódov. Suverénne najčastejšou operáciou v tomto algoritme je výpočet Hammingovej vzdialenosti medzi dvomi bitovými vektormi. Preto ak by sa nám podarilo výrazne urýchliť výpočet Hammingovej váhy,

pozorovali by sme aj výrazné urýchlenie všetkých algoritmov, ktoré využívajú túto greedy metódu.

HAKMEM algoritmus

V roku 1972 sa uskutočnila iniciatíva MIT AI Lab pod názvom *HAKMEM*[2], ktorej cieľom bolo spísať technický report užitočných techník a šikovných algoritmov pre matematické výpočty. Jedným z výsledkov tejto iniciatívy je aj nový algoritmus pre výpočet Hammingovej váhy založený na 12 aritmetických operáciách, ktorý je podľa mojich meraní 10x rýchlejší ako triviálna implementácia pomocou cyklu cez všetky bity.

Streaming SIMD Extensions 4.2

S uvedením inštrukčnej sady SSE4.2 sa dostala do procesorov podpora na úrovni inštrukcie pre výpočet Hammingovej váhy. Hammingova váha sa v prípade binárnych reťazcov nazýva aj *population count*, resp. *popcount*. Preto sa aj táto inštrukcia nazýva *POPCNT*.

Táto inštrukcia vráti počet bitov nastavených na 1. Vykonanie tejto inštrukcie zaberie procesoru 3 takty a jedná sa o jednoznačne najrýchlejší spôsob výpočtu Hammingovej váhy pre binárne reťazce. Podpora tejto inštrukcie v procesore je indikovaná 23. bitom v ECX registri po vykonaní inštrukcie *CPUID*.

Využitie inštrukcie *POPCNT* je viac ako 3x rýchlejšie v porovnaní s optimalizovaným *HAKMEM* algoritmom. To vo výsledku dáva 30 násobné zrýchlenie voči triviálnej implementácii pomocou cyklu, čo je citeľný rozdiel. Nevýhodou ostáva, že túto inštrukciu podporujú iba najnovšie modely procesorov uvedené od novembra 2008.

[2] http://home.pipeline.com/~hbaker1/hakmem/hakmem.html

Kapitola 8

Výsledky

„Hodnotenie človeka má vychádzať z toho, čo dáva, a nie z toho, čo je schopný získať.“

~ Albert Einstein (1879 – 1955)

Po implementácii boli algoritmy podrobené dôkladnému otestovaniu. Ukazuje sa, že navrhnuté riešenie vykazuje sľubné výsledky. Pre porovnanie sme vybrali nasledovné algoritmy:

a) Lexikografický algoritmus podľa [3]

b) Randomizovaný greedy podľa [10]

c) Stochastický lexikografický algoritmus

d) Horolezecký algoritmus v kombinácii s lexikografickým algoritmom

e) Genetický algoritmus s nepriamou reprezentáciou jedincov a lexikografickým doplnením kódových slov podľa [10]

f) Jednoduchý greedy algoritmus pre hľadanie maximálnej kliky

g) Modifikovaný greedy algoritmus pre hľadanie maximálnej kliky

h) Tabu algoritmus pre hľadanie maximálnej kliky

8.1 Spôsob vyhodnotenia

Porovnávané algoritmy sú založené na rôznych konceptoch. Aby bolo vyhodnotenie spravodlivé, je potrebné zadefinovať si porovnávacie kritérium.

Porovnávať rýchlosť optimalizácie algoritmov podľa počtu iterácií, resp. generácií by nebolo celkom objektívne. Genetický algoritmus v rámci jednej generácie vyskúša 500 riešení, z ktorých vyberá tie najlepšie a tie ďalej kríži a mutuje. Stochastický lexikografický algoritmus vyskúša v rámci jednej iterácie iba jedno riešenie a horolezecký algoritmus prehľadáva viacero riešení v okolí aktuálneho riešenia počas jednej iterácie.

Preto namiesto počtu iterácií, resp. generácií je spravodlivejšie uvažovať počet vyhodnotení funkcie. Najčastejšie vyhodnocovanou funkciou je práve doplnenie vybratých kódových slov pomocou greedy algoritmu na hľadanie lexikografických kódov. V prípade genetického algoritmu sa tento greedy algoritmus vykonáva pri výpočte fitness funkcie, pri stochastickom lexikografickom algoritme sa vykonáva každú iteráciu jedenkrát a pri horolezeckom algoritme sa vykonáva pre každé prehľadávané okolie v rámci jednej iterácie.

8.2 Úspešnosť algoritmov

V tejto časti sú zobrané výsledky jednotlivých algoritmov. Okrem najväčšieho vygenerovaného kódu počas viacerých behov uvádzame aj priemernú veľkosť kódu a hodnotu rozptylu. Zvýraznené hodnoty patria najlepšiemu algoritmu navrhnutému v tejto práci.

8.2.1 Algoritmy pre generovanie kódov

Každý z uvedených algoritmov v tejto časti využíva greedy algoritmus pre generovanie lexikografických kódov, resp. nejakú jeho modifikovanú verziu.

Základné porovnanie

V nasledujúcej tabuľke (Tab. 8) sú uvedené výsledky algoritmov po 50 000 vyhodnoteniach funkcie. Uvedené štatistické hodnoty boli namerané počas 20-tich behov.

Tab. 8. Výsledky algoritmov po 50 000 vyhodnoteniach funkcie

		$A_2(12, 6)$	$A_2(13, 6)$	$A_2(17, 6)$	$A_2(17, 4)$
Najlepšie známe optimum		24	32	256	2720
Lexikografický algoritmus		16	16	256	2048
Randomizovaný greedy	Maximum	13	21	127	1699
	Priemer	11,5	18,5	124,1	1681,8
	Rozptyl	0,89	1,63	5,15	92,06
Stochastický lexikografický algoritmus	Maximum	24	32	256	2207
	Priemer	24	29	256	2200,4
	Rozptyl	0	3,16	0	14,46
Horolezecký lexikografický algoritmus	Maximum	**24**	**32**	**256**	**2245**
	Priemer	24	29,6	256	2219,6
	Rozptyl	0	4,04	0	123,2
Genetický algoritmus	Maximum	24	28	256	2231
	Priemer	24	26,6	256	2210,3
	Rozptyl	0	0,88	0	62,09

Z tabuľky je zjavné, že nami navrhnutá stochastická verzia lexikografického algoritmu poskytuje lepšie výsledky ako pôvodný deterministický algoritmus na generovanie lexikografických kódov, aj ako jeho randomizovaná verzia podľa uvedeného článku.

Z výsledkov ďalej vyplýva, že nami navrhnutá kombinácia horolezeckého algoritmu a greedy algoritmu pre generovanie lexikografických kódov prekonala genetický algoritmus, ktorý v referenčnom článku dosahoval najlepšie výsledky. Náš algoritmus dokázal vygenerovať kód o 14 slov väčší v najťažšom testovacom prípade a o 3 slová väčší pre prípad $A_2(13, 6)$.

Rozšírené porovnanie

Tri najlepšie algoritmy zo základného porovnania sme sa rozhodli ďalej otestovať na väčší počet iterácií. Zaujímali nás výsledky algoritmov po 5 000 000 vyhodnoteniach funkcie. Štatistické výsledky boli vypočítané z 10-tich behov (Tab. 9).

Vzhľadom na to, že pre testovacie prípady $A_2(12, 6)$ a $A_2(17, 6)$ dokázali porovnávané algoritmy nájsť známe optimum v každom jednom behu v predošlom teste, nemá zmysel ich v tomto teste ďalej uvažovať.

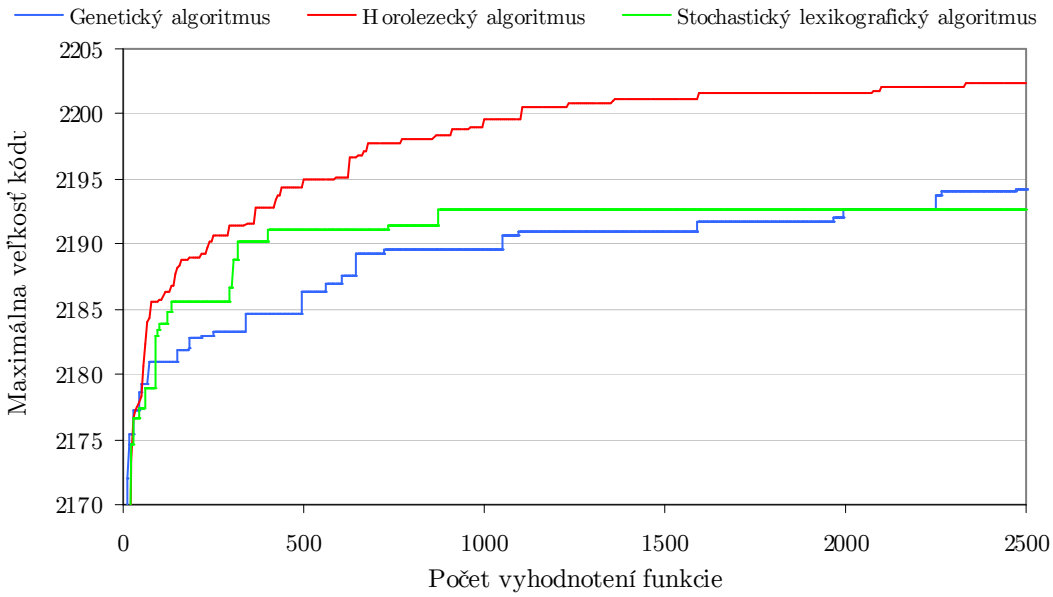
Tab. 9. Výsledky algoritmov po 5 000 000 vyhodnoteniach funkcie

		$A_2(13, 6)$	$A_2(17, 4)$
Najlepšie známe optimum		32	2720
Stochastický lexikografický algoritmus	Maximum	32	2218
	Priemer	32	2211,9
	Rozptyl	0	11,21
Horolezecký lexikografický algoritmus	Maximum	**32**	**2278**
	Priemer	32	2248,6
	Rozptyl	0	204,49
Genetický algoritmus	Maximum	32	2239
	Priemer	29,2	2218,3
	Rozptyl	3,73	66,01

Obidva nami navrhnuté algoritmy dokázali prekonať genetický algoritmus pri menšom testovacom prípade $A_2(13, 6)$. Pri väčšom prípade ho dokázal prekonať iba horolezecký algoritmus, ale za to s výrazným rozdielom 39 kódových slov. Za zmienku stojí, že genetický algoritmus nám v tomto porovnaní vygeneroval väčší kód ako jeho autorom v článku.

Rýchlosť optimalizácie

Zaujímavé je porovnať rýchlosť optimalizácie riešenia v závislosti od počtu ohodnotení funkcie. Nasledujúci graf (Obr. 11) zobrazuje priebeh optimalizácie pre najväčší testovací prípad $A_2(17, 4)$ počas prvých 2 500 ohodnotení funkcie.



Obr. 11. Rýchlosť optimalizácie algoritmov

Genetický algoritmus má pozvoľnejší začiatok, no po 2 000 vyhodnoteniach predbehne stochastický lexikografický algoritmus. Jasne najrýchlejšie však optimalizuje horolezecký algoritmus a ani s vyšším počtom vyhodnotení funkcie ho genetický algoritmus nepredbehne.

Veľkosť prehľadávaného okolia

Dôležitým parametrom stochastického lexikografického algoritmu je počet náhodne vybratých kompatibilných kódových slov, ktoré pridáme do výslednej množiny slov pred spustením algoritmu na generovanie lexikografických kódov.

V prípade horolezeckého algoritmu tento parameter zároveň určuje veľkosť prehľadávaného okolia, keďže okolie sa prehľadáva tak, že každé jedno náhodne vybraté slovo postupne nahradíme iným náhodne vybratým kódovým slovom. Optimálne hodnoty tohto parametra pre jednotlivé kódy sú zobrazené nižšie (Tab. 10).

Tab. 10. Optimálne hodnoty veľkosti prehľadávaného okolia

	$A_2(12, 6)$	$A_2(13, 6)$	$A_2(17, 6)$	$A_2(17, 4)$
Veľkosť okolia	1	3	1	6

Pôvodne sme si mysleli, že pre väčšie kódy bude rozumnejšie nainicializovať výslednú množinu viacerými náhodnými slovami, aby sme beh greedy algoritmu viacej ovplyvnili. Ukázalo sa však, že to závisí od konkrétneho prípadu, pretože niektoré testovacie kódy dokáže greedy algoritmus dobre optimalizovať automaticky.

8.2.2 Algoritmy pre maximálnu kliku

Nasleduje porovnanie generických algoritmov na hľadanie maximálnej kliky (Tab. 11). Tieto algoritmy nevyužívajú žiadnu vlastnosť z teórie kódovania a mali by nájsť dobré riešenie pre ľubovoľný graf.

Tab. 11. Výsledky algoritmov na hľadanie maximálnej kliky

	$A_2(12, 6)$	$A_2(13, 6)$	$A_2(17, 6)$	$A_2(17, 4)$
Najlepšie známe optimum	24	32	256	2720
Jednoduchý greedy	24	26	162	2157
Modifikovaný greedy	24	32	256	2181
Tabu algoritmus	24	32	256	2182

Porovnávané algoritmy na hľadanie maximálnej kliky možno zaradiť medzi menej úspešné. Jednoduchý greedy algoritmus zbehol relatívne rýchlo, no jeho výsledky nenaplnili očakávania.

Zvyšné algoritmy síce fungovali na menších kódoch dobre, ale najväčší kód im robil problém. Nielenže sa nedokázali priblížiť algoritmom z domény kódovania, ale aj ich výpočet trval niekoľko dní, čo v porovnaní s desiatkami minút ostatných algoritmov je výrazný rozdiel.

8.3 Časová zložitosť algoritmov

Výpočtovo najnáročnejšou časťou je dopĺňanie kódových slov pomocou greedy algoritmu na generovanie lexikografických kódov. V predošlom texte pod pojmom vyhodnotenie funkcie máme na mysli práve hodnotu, ktorú vráti tento algoritmus. Počet vyhodnotení funkcie je počet, koľkokrát sa tento algoritmus vykoná.

8.3.1 Asymptotická zložitosť

Označme si c ako počet kandidátov:

$$c = \sum_{i=d}^{n} \binom{n}{i}$$

Pre malé hodnoty d počet kandidátov c rastie exponenciálne v závislosti od n. Z Pascalovho trojuholníka vyplýva, že ak

$$d = 1 \Rightarrow c = 2^n - 1$$

Greedy algoritmus pre generovanie lexikografických kódov prechádza cez všetkých kandidátov lexikograficky a pre každého kandidáta testuje, či je vo vzdialenosti aspoň d od každého kódového slova. Ak áno, tak kandidáta pridá do kódu. Ak nie, tak pokračuje ďalším kandidátom.

V najhoršom prípade, t.j. ak $d = 1$, bude algoritmus pridávať všetkých kandidátov do kódu a pre každého ďalšieho kandidáta bude musieť otestovať vzdialenosť s o jedna väčším počtom kódových slov. Otestovanie vzdialenosti medzi dvomi slovami znamená výpočet ich Hammingovej váhy a porovnanie.

Na začiatku je množina kódových slov prázdna, preto prvý kandidát sa pridá automaticky do kódu. Druhý kandidát sa musí otestovať s jedným kódovým slovom a ak vyhovuje minimálnej vzdialenosti d, tak sa tiež pridá do kódu. Tretí kandidát sa musí otestovať už s dvomi kódovými slovami a c-ty kandidát s $c-1$ kódovými slovami. Preto počet otestovaní je asymptoticky rovný

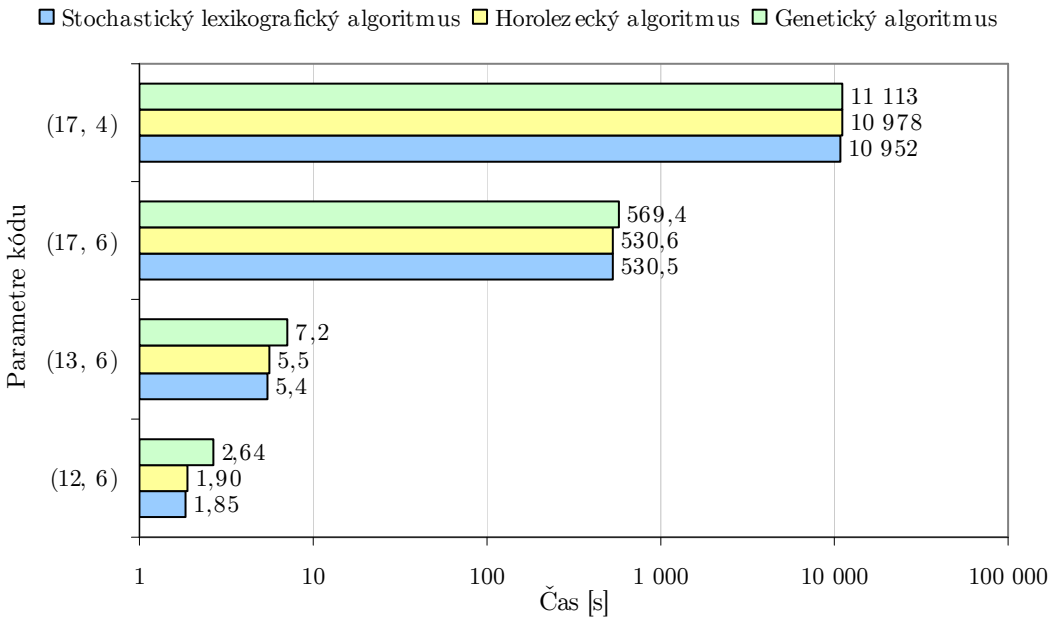
$$\sum_{i=1}^{c-1} i = 1 + 2 + 3 + \ldots + c - 1 = (1 + c - 1) \cdot \frac{c-1}{2} = \frac{c^2 - c}{2} \Rightarrow \Theta\left(\frac{c^2 - c}{2}\right) \Rightarrow \mathrm{O}(c^2)$$

Ak uvádzame výsledky algoritmov po f vyhodnoteniach funkcie, tak každý algoritmus musel vykonať f-krát vyššie popísaný greedy algoritmus na generovanie lexikografických kódov.

8.3.2 Čas behu algoritmov

Taktiež môže byť zaujímavé porovnať skutočný čas behu jednotlivých algoritmov. Pri rovnakom počte vyhodnotení funkcie by mali byť časy veľmi podobné, no vzhľadom na základné odlišnosti jednotlivých algoritmov, môže

dôjsť k určitým rozdielom. Porovnanie najúspešnejších algoritmov je zobrazené na nasledujúcom grafe (Obr. 12).



Obr. 12. Čas behu algoritmov pre 50 000 vyhodnotení funkcie

V grafe vidíme vyrovnané časy behov jednotlivých algoritmov pre 50 000 vyhodnotení funkcie. To je dôkaz toho, že uvažovať počet vyhodnotení funkcie ako metriku na porovnávanie algoritmov, bola dobrá voľba. Algoritmy vtedy vykonajú približne rovnaké množstvo výpočtov.

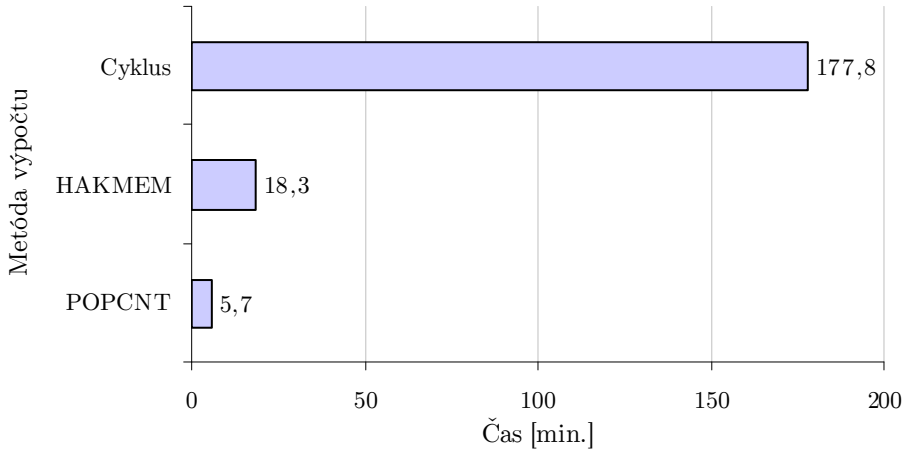
Genetický algoritmus je predsa len pomalší ako zvyšné dva. Dôvodom je väčšia réžia genetického algoritmu spojená s výberom jedincov, krížením, mutovaním, nahradením starej populácie novou a pod.

Vplyv technologických optimalizácií

Ďalej v tejto práci porovnávame vplyv technologických vylepšení na celkové zrýchlenie našich implementácií algoritmov. Uvedené časy boli získané po vykonaní 5 000 vyhodnoteniach funkcie.

Výpočet Hammingovej váhy

Najskôr sa pozrieme na efektivitu rôznych metód na výpočet Hammingovej váhy (Obr. 13).

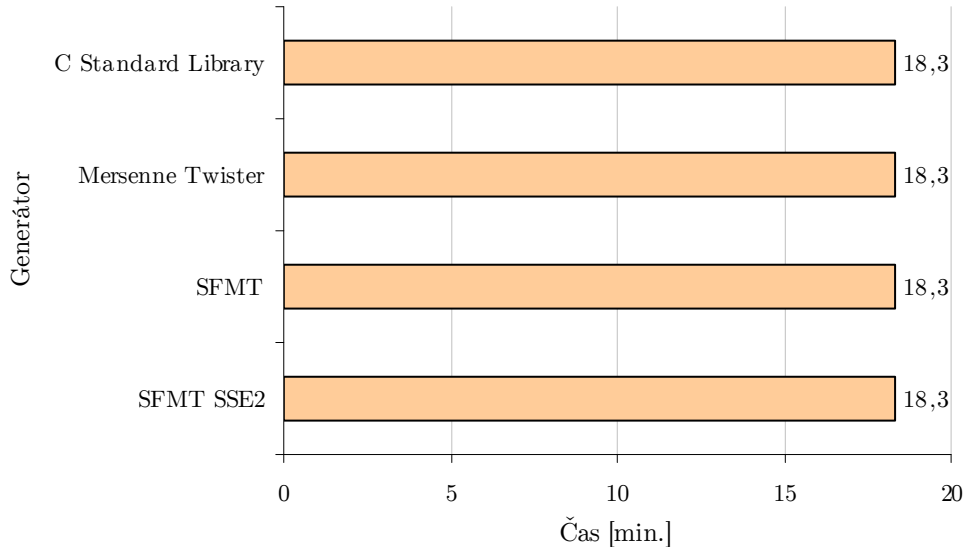


Obr. 13. Porovnanie metód na výpočet Hammingovej váhy

Výkon inštrukcie *POPCNT* sa pozitívne prejavil aj v našich algoritmoch. Jedná sa o zhruba 3x rýchlejší spôsob výpočtu Hammingovej váhy ako *HAKMEM* algoritmus. Oproti triviálnej implementácii pomocou cyklu cez všetky bity kódových slov sme takto dosiahli 30-násobné zrýchlenie.

Generovanie pseudonáhodných čísel

Ďalším technologickým prínosom v našej implementácii bolo použitie generátora pseudonáhodných čísel SFMT s podporou inštrukčnej sady SSE2 (Obr. 14).



Obr. 14. Porovnanie generátorov pseudonáhodných čísel

Napriek nepochybnej rýchlosti generátora SFMT SSE2 sa jeho pozitívny efekt v našich algoritmoch neprejavil. Dôvodom je fakt, že generovanie pseudonáhodných čísel nie je úzkym hrdlom porovnávaných algoritmov a ani to nie je najčastejšie sa vykonávaná operácia, v porovnaní napríklad s výpočtom Hammingovej váhy. Prínosom však ostáva generovanie vysoko kvalitných pseudonáhodných čísel.

8.4 Pamäťová zložitosť algoritmov

Genetický algoritmus z referenčného článku využíva maticu kompatibility na rýchle zisťovanie, či dve kódové slová sú vo vzdialenosti aspoň d. Najväčší testovací prípad $A_2(17, 4)$ má 130 239 kandidátov. Vďaka ekvivalencii kódov môžeme namiesto toho hľadať riešenie v prípade $A_2(16, 3)$, ktorý má 65 400 kandidátov.

Matica kompatibility sa predpočíta pre každú dvojicu kandidátov, t.j. bude sa jednať o štvorcovú maticu typu $c \times c$, kde c je počet kandidátov. Každý prvok tejto matice a_{ij} môže nadobúdať iba 2 možné stavy podľa toho, či i-ty kandidát

je kompatibilný s j-tym kandidátom alebo nie. Na zakódovanie dvoch stavov nám stačí 1 bit, takže celkové pamäťové nároky budú:

$$65\,400^2 \div 8 = 534\,645\,000 \text{ B} \cong 510 \text{ MB}$$

Tým, že sme optimalizovali výpočet Hammingovej váhy pomocou špeciálnej inštrukcie, tak sa nám osvedčilo ju zakaždým radšej vypočítať. Výkon inštrukcie *POPCNT* trvá procesoru 3 takty a je to rozhodne rýchlejšie ako pristupovať do pamäte a robiť bitové operácie nad bitovo reprezentovanou maticou. Vďaka odstráneniu matice kompatibility majú naše algoritmy menšie pamäťové nároky o nezanedbateľných 510 MB, čo predstavuje 99 % spotrebovanej pamäte v najväčšom testovacom prípade.

8.5 Zhodnotenie výsledkov

Z výsledkov vyplýva, že genetický algoritmus prezentovaný v referenčnom článku skutočne nie je dôvodom, prečo bol tento algoritmus najlepší. Dôvodom bola jeho kombinácia s greedy algoritmom pre generovanie lexikografických kódov. Je to práve tento greedy algoritmus, ktorý zabezpečí generovanie veľkých kódov.

Logickým podnetom preto bolo pokúsiť sa genetický algoritmus nahradiť niečím úspornejším. Ukázalo sa, že jednoduchou stochastickou úpravou pôvodného greedy algoritmu sa dajú dosiahnuť sľubné výsledky. Horolezecká optimalizácia náhodne vybratých kódových slov do výslednej množiny poskytla ešte lepšie výsledky a dokázala prekonať aj samotný genetický algoritmus. Nielenže náš horolezecký algoritmus ponúkol lepšie riešenie počas testovania (2245 proti 2231 kódovým slovám), ale výsledok dokázal vrátiť o niečo rýchlejšie a pri tom mal o 510 MB menšie pamäťové nároky.

Pri dlhšom testovaní na 5 000 000 vyhodnotení funkcie sme boli schopní nájsť samoopravný kód s 2 278 kódovými slovami pri dĺžke kódu 16 a minimálnej

vzdialenosti 3. Nie je to síce najviac na svete, ale je to výrazne viac, ako poskytol genetický algoritmus z referenčného článku.

Domnievame sa, že za slabými výsledkami generických algoritmov na hľadanie maximálnej kliky stojí najmä štruktúra grafov samoopravných kódov. Väčšina z týchto algoritmov je založená na fakte, že kliku rozširuje pridaním susedného vrcholu s najväčším stupňom, resp. kliku extrahuje odobratím tých vrcholov, ktorých odobratie maximalizuje množinu potenciálnych vrcholov na pridanie do kliky. Avšak pre grafy samoopravných kódov platí, že veľmi veľa vrcholov má rovnaký stupeň a rôznorodosť stupňov vrcholov v grafe je pomerne malá. To je podľa nás hlavný dôvod, prečo majú tieto algoritmy problém optimalizovať samoopravné kódy.

Do pozornosti dávame ešte jeden fakt týkajúci sa času behu jednotlivých algoritmov. Ak nám 5,7 minúty trval výpočet pri 5 000 vyhodnoteniach funkcie, tak základné porovnanie algoritmov, ktoré sa vykonávalo po 50 000 vyhodnoteniach, trvalo 57 minút t.j. približne 1 hodinu. Nebyť technologických optimalizácií, takýto výpočet by netrval 1 hodinu, ale 30 hodín. My sme však robili porovnania aj na 5 000 000 vyhodnoteniach funkcie. Takýto výpočet nám trval 4 dni, ale bez technologických opatrení by to boli 4 mesiace. Preto si myslíme, že technologické riešenia našli v tejto práci svoje uplatnenie.

8.6 Ďalšia práca

V tejto práci sme ukázali pomerne efektívny spôsob generovania samoopravných kódov pomocou kombinácie stochastických algoritmov a greedy algoritmu pre generovanie lexikografických kódov.

Najviac sa nám osvedčila kombinácia greedy algoritmu a horolezeckého algoritmu na optimalizáciu inicializačnej množiny náhodne vybraných kódových slov. Ale aj ostatné algoritmy skombinované s greedy algoritmom poskytovali slušný výsledok. Preto by stálo za námahu vyskúšať ďalšie kombinácie greedy

algoritmu a nejakej stochastickej optimalizačnej metódy, ako napríklad simulovaného žíhania, zakázaného prehľadávania, resp. genetického programovania.

Napriek tomu, že sme sa usilovne snažili čas behu algoritmov čo najviac skrátiť, stále výpočet trvá relatívne dlho. Dané je to najmä enormnými veľkosťami grafov. Preto ďalšie vylepšenie by mohlo spočívať v paralelizácii algoritmov.

Taktiež by stálo za vyskúšanie integrovať viacero heuristických informácií do algoritmu, ktoré by mohli zlepšiť kvalitu riešenia. Veľmi užitočné by bolo aj skúmanie ďalších greedy heuristík, ktoré by dokázali rýchlo vygenerovať veľké samoopravné kódy, ako to je v prípade lexikografického algoritmu.

Celý výskum v tejto problematike by mal vyústiť do nájdenia nových samoopravných kódov, väčších ako je doposiaľ známe[3]. Toto si však okrem rýchlo optimalizujúceho algoritmu vyžaduje aj mesiace procesorového času.

[3] http://www.win.tue.nl/~aeb/codes/binary-1.html

Kapitola 9

Záver

„Kto veľa hovorí, zriedka uskutočňuje svoje slová. Múdry človek sa vždy bojí, aby jeho slová nepredstihovali jeho skutky.“

~ Confucius (551 pred Kr. - 479 pred Kr.)

Počas uplynulých dvoch rokov sme sa podrobne oboznámili s problematikou riešenou v tejto práci. Detailne sme si naštudovali teóriu kódovania a súčasné algoritmy pre generovanie optimálnych samoopravných kódov založené na evolučných princípoch.

Cieľom tejto práce bolo navrhnúť vhodný evolučný algoritmus na generovanie optimálnych samoopravných kódov, ktorý by dokázal v niektorých prípadoch prekonať vybrané súčasné riešenia. Vzhľadom na veľkosť a hustotu grafov sme sa zamerali na stochastické modifikácie greedy algoritmov, ktoré dokážu v relatívne krátkom čase vygenerovať kód s dobrými vlastnosťami. Vďaka inovatívnej kombinácii horolezeckého algoritmu a greedy algoritmu pre generovanie lexikografických kódov sme boli schopní vygenerovať väčšie samoopravné kódy ako autori v referenčnom článku [10] pri rovnakých testovacích podmienkach.

Dosiahnutými výsledkami sa podarilo prekonať najlepšie algoritmy z referenčného článku, čo sa dá považovať za prínos. Myslíme si preto, že cieľ tejto práce bol úspešne naplnený.

KAPITOLA 10

Literatúra

„Od učenia ešte nikto nezomrel, ale načo riskovať?“

~ Albert Einstein (1879 – 1955)

[1] Ashlock, D., Guo, L., Qiu, F.: Greedy Closure Evolutionary Algorithms. In Proceedings of Congress on Evolutionary Computation, Vol. 2, s. 1296-1301, 2002.

[2] Bose, R.C., Ray-Chaudhuri, D.K.: On a Class of Error Correcting Binary Group Codes. Information and Control, Vol. 3, No. 1, s. 68-79, 1960.

[3] Conway, J.H., Sloane, N.J.A.: Lexicographic codes: Error-Correcting Codes from Game Theory. IEEE Transactions on Information Theory, IEEE Press, Vol. 32, s. 337-348, 1986.

[4] Černý, V.: Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm. Journal of Optimization Theory and Applications, s. 41-51, 1985.

[5] Euler, L.: The Solution of a Problem Relating to the Geometry of Position. The Imperial Academy of Sciences in Saint Petersburg, s. 128-140, 1736.

[6] Fano, R.M.: The Transmission of Information. Research Laboratory of Electronics, MIT Press, 1949.

[7] Fogel, L.J., Owens, A.J., Walsh, M.J.: Artificial Intelligence through Simulated Evolution. John Wiley and Sons, 1966.

[8] Glover, F., McMillan, C.: The General Employee Scheduling Problem: An Integration of MS and AI. Computers and Operations Research, s. 563-573, 1986.

[9] Golay, M.J.E.: Notes on Digital Coding. Proceedings of the Institute of Radio Engineers, 1949.

[10] Haas, W., Houghten, S.: A Comparison of Evolutionary Algorithms for Finding Optimal Error-Correcting Codes. In Proceedings of the 3rd IASTED International Conference on Computational Intelligence, ACTA Press, s. 64-70, 2007.

[11] Hamming, R.W.: Error Detecting and Error Correcting Codes. The Bell System Technical Journal, s. 147-160, 1950.

[12] Hocquenghem, A.: Codes correcteurs d'erreurs. Chiffres, s. 147-156, 1959.

[13] Holland, J.H.: Adaptation in Natural and Artificial Systems. University of Michigan Press, 1975.

[14] Huffman, D.A.: A Method for the Construction of Minimum-Redundancy Codes. In Proceedings of the IRE, s. 1098-1101, 1952.

[15] Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by Simulated Annealing. Science Vol. 220 (4598), s. 671-680, 1983.

[16] Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, 1992.

[17] Kvasnička, V., Pospíchal, J.: Algebra a diskrétna matematika. STU Press, s. 228, 2008.

[18] Levenshtein, V.I.: A class of systematic codes. Soviet Math, s. 368-371, 1960.

[19] MacWilliams, F.J., Sloane, N.J.A.: The Theory of Error-Correcting Codes. North Holland Publishing Company, 1977.

[20] Marchiori, E.: A Simple Heuristic Based Genetic Algorithm for the Maximum Clique Problem. In Proceedings of ACM Symposium on Applied Computing, ACM Press, s. 366-373, 1998.

[21] McCaffrey, J.: Greedy and Tabu Algorithms for Maximum Clique. MSDN Magazine, 2011.

[22] McCarney, E.D., Houghten, S., Ross, J.B.: Evolutionary Approaches to the Generation of Optimal Error Correcting Codes. In Proceedings of the 14th International Conference on Genetic and Evolutionary Computation, ACM Press, s. 1135-1142, 2012.

[23] Muller, D.E.: Application of Boolean algebra to switching circuit design and to error detection. Transactions of the IRE Professional Group on Electronic Computers, s. 6-12, 1954.

[24] Reed, I.S.: A class of multiple-error-correcting codes and the decoding scheme. Transactions of the IRE Professional Group on Information Theory, s. 38-49, 1954.

[25] Reed, I.S., Solomon, G.: Polynomial Codes over Certain Finite Fields. Journal of the Society for Industrial and Applied Mathematics, Vol. 8, No. 2, s. 300-304, 1960.

[26] Rechenberg, I.: Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. Fromman-Holzboog, 1973.

[27] Saito, M., Matsumoto, M.: SIMD-oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator. Monte Carlo and Quasi-Monte Carlo Methods, Springer Berlin Heidelberg, s. 607-622, 2008.

[28] Shannon, C.E.: A Mathematical Theory of Communication. The Bell System Technical Journal, s. 379-423, 1948.

[29] Schwefel, H.P.: Numerische Optimierung von Computer-Modellen mittels Evolutionsstrategie. Birkhäuser, 1977.

[30] Zhang, Q., Sun, J., Tsang, E.: An Evolutionary Algorithm With Guided Mutation for the Maximum Clique Problem. Transactions on Evolutionary Computation, IEEE Press, s. 192-200, 2005.

Príloha A

Technická dokumentácia

Systémové požiadavky

Minimálne systémové požiadavky:

- Operačný systém Windows NT 5.0 a vyšší

Odporúčané systémové požiadavky:

- Procesor s podporou inštrukčnej sady SSE4.2

Inštalačná príručka

Aplikácia nevyžaduje žiadne špeciálne inštalačné procedúry. Skompilovanú aplikáciu stačí spustiť na operačnom systéme Windows NT 5.0 a vyššom.

Pre skompilovanie programu je potrebné použiť Visual C++ kompilátor obsiahnutý vo vývojovom prostredí Visual Studio.

Používateľská príručka

Spustiteľné programy s jednotlivými algoritmami možno nájsť v adresári *CD-R\Programy\Diplomový projekt\Release*. Po spustení program vyzve používateľa na zadanie parametrov generovaného kódu. Používateľ zadá dĺžku kódových slov v bitoch n, minimálnu vzdialenosť medzi kódovými slovami d a počet náhodne vložených slov do kódu. Po dokončení výpočtu možno vygenerovaný samoopravný kód nájsť v podadresári *Codes*.

Príloha B

Samoopravný kód

Kód (16, 2278, 3) = {0, 7, 25, 30, 42, 45, 51, 52, 75, 76, 82, 85, 97, 102, 120, 127, 387, 396, 405, 410, 422, 425, 432, 447, 448, 463, 470, 473, 485, 490, 499, 508, 641, 646, 664, 671, 683, 684, 690, 693, 714, 717, 723, 724, 736, 743, 761, 766, 770, 773, 795, 796, 808, 815, 817, 822, 841, 846, 848, 855, 867, 868, 890, 893, 1154, 1157, 1179, 1180, 1192, 1199, 1201, 1206, 1225, 1230, 1232, 1239, 1251, 1252, 1274, 1277, 1281, 1286, 1304, 1311, 1323, 1324, 1330, 1333, 1354, 1357, 1363, 1364, 1376, 1383, 1401, 1406, 1539, 1548, 1557, 1562, 1574, 1577, 1584, 1599, 1600, 1615, 1622, 1625, 1637, 1642, 1651, 1660, 1920, 1927, 1945, 1950, 1962, 1965, 1971, 1972, 1995, 1996, 2002, 2005, 2017, 2022, 2040, 2047, 2180, 2185, 2194, 2211, 2222, 2232, 2247, 2257, 2268, 2285, 2294, 2299, 2312, 2319, 2321, 2326, 2338, 2341, 2363, 2364, 2371, 2372, 2394, 2397, 2409, 2414, 2416, 2423, 2570, 2573, 2576, 2583, 2593, 2622, 2651, 2658, 2668, 2677, 2955, 2964, 2976, 2983, 3001, 3009, 3014, 3032, 3039, 3058, 3083, 3092, 3104, 3111, 3129, 3137, 3160, 3167, 3186, 3466, 3469, 3472, 3479, 3489, 3518, 3547, 3554, 3564, 3573, 3720, 3727, 3729, 3734, 3746, 3749, 3771, 3772, 3779, 3780, 3802, 3805, 3817, 3822, 3824, 3831, 3844, 3849, 3858, 3875, 3886, 3896, 3911, 3921, 3932, 3949, 3958, 3963, 4232, 4239, 4241, 4246, 4258, 4261, 4283, 4284, 4291, 4292, 4314, 4317, 4329, 4334, 4336, 4343, 4356, 4361, 4370, 4387, 4398, 4408, 4423, 4433, 4444, 4461, 4470, 4475, 4619, 4628, 4640, 4647, 4665, 4673, 4678, 4696, 4703, 4722, 5002, 5005, 5008, 5015, 5025, 5054, 5083, 5090, 5100, 5109, 5130, 5133, 5136, 5143, 5153, 5182, 5211, 5218, 5228, 5237, 5515, 5524, 5536,

5543, 5561, 5569, 5574, 5592, 5599, 5618, 5764, 5769, 5778, 5795, 5806, 5816, 5831, 5841, 5852, 5869, 5878, 5883, 5896, 5903, 5905, 5910, 5922, 5925, 5947, 5948, 5955, 5956, 5978, 5981, 5993, 5998, 6000, 6007, 6145, 6150, 6168, 6175, 6187, 6188, 6194, 6197, 6218, 6221, 6227, 6228, 6240, 6247, 6265, 6270, 6528, 6535, 6553, 6558, 6570, 6573, 6579, 6580, 6603, 6604, 6610, 6613, 6625, 6630, 6648, 6655, 6786, 6789, 6811, 6812, 6824, 6831, 6833, 6838, 6857, 6862, 6864, 6871, 6883, 6884, 6906, 6909, 6915, 6924, 6933, 6938, 6950, 6953, 6960, 6975, 6976, 6991, 6998, 7001, 7013, 7018, 7027, 7036, 7299, 7308, 7317, 7322, 7334, 7337, 7344, 7359, 7360, 7375, 7382, 7385, 7397, 7402, 7411, 7420, 7426, 7429, 7451, 7452, 7464, 7471, 7473, 7478, 7497, 7502, 7504, 7511, 7523, 7524, 7546, 7549, 7680, 7687, 7705, 7710, 7722, 7725, 7731, 7732, 7755, 7756, 7762, 7765, 7777, 7782, 7800, 7807, 8065, 8070, 8088, 8095, 8107, 8108, 8114, 8117, 8138, 8141, 8147, 8148, 8160, 8167, 8185, 8190, 8330, 8333, 8336, 8343, 8353, 8382, 8411, 8418, 8428, 8437, 8459, 8468, 8480, 8487, 8505, 8513, 8536, 8543, 8562, 8708, 8713, 8722, 8739, 8750, 8760, 8775, 8785, 8796, 8813, 8822, 8827, 9096, 9103, 9105, 9110, 9122, 9125, 9147, 9148, 9155, 9156, 9178, 9181, 9193, 9198, 9200, 9207, 9224, 9231, 9233, 9238, 9250, 9253, 9275, 9276, 9283, 9284, 9306, 9309, 9321, 9326, 9328, 9335, 9604, 9609, 9618, 9635, 9646, 9656, 9671, 9681, 9692, 9709, 9718, 9723, 9867, 9876, 9888, 9895, 9913, 9921, 9926, 9944, 9951, 9970, 9994, 9997, 10000, 10007, 10017, 10046, 10075, 10082, 10092, 10101, 10243, 10252, 10261, 10266, 10281, 10288, 10303, 10310, 10329, 10341, 10346, 10355, 10364, 10432, 10625, 10630, 10648, 10655, 10667, 10668, 10674, 10677, 10698, 10701, 10707, 10708, 10727, 10745, 10750, 10824, 10887, 10905, 10910, 10916, 10922, 10931, 10955, 10962, 10965, 10977, 11000, 11007, 11008, 11022, 11027, 11037, 11060, 11066, 11077, 11110, 11115, 11121, 11406, 11411, 11421, 11444, 11450, 11461, 11494, 11496, 11505, 11527, 11545, 11550, 11556, 11562, 11571, 11584, 11595, 11605, 11640, 11647, 11777, 11782, 11800, 11807, 11819, 11820, 11826, 11829, 11853, 11859, 11860, 11872, 11879, 11897, 11902, 12162, 12165, 12187, 12188, 12200, 12207, 12209, 12214, 12233, 12238, 12240, 12247, 12259, 12260, 12282, 12285, 12290, 12293, 12315, 12316, 12328, 12335, 12337, 12342, 12361, 12366, 12368, 12375, 12387, 12388, 12410, 12413, 12686, 12691,

12701, 12708, 12730, 12738, 12741, 12776, 12783, 12785, 12931, 12940, 12949, 12954, 12966, 12969, 12976, 12991, 12992, 13007, 13014, 13017, 13029, 13034, 13043, 13052, 13057, 13062, 13080, 13087, 13098, 13101, 13107, 13131, 13132, 13138, 13141, 13152, 13159, 13177, 13182, 13440, 13447, 13465, 13470, 13482, 13485, 13491, 13515, 13516, 13522, 13525, 13537, 13560, 13567, 13571, 13580, 13589, 13594, 13606, 13609, 13616, 13631, 13647, 13654, 13657, 13669, 13674, 13683, 13692, 13838, 13843, 13853, 13860, 13882, 13890, 13893, 13928, 13935, 13937, 14263, 14280, 14302, 14315, 14324, 14475, 14484, 14496, 14503, 14521, 14552, 14559, 14578, 14602, 14605, 14608, 14615, 14625, 14654, 14683, 14690, 14700, 14709, 14863, 14865, 14870, 14882, 14885, 14907, 14908, 14915, 14916, 14938, 14941, 14953, 14958, 14960, 14967, 15236, 15241, 15250, 15267, 15278, 15288, 15303, 15313, 15324, 15341, 15350, 15355, 15364, 15369, 15378, 15395, 15406, 15416, 15431, 15441, 15452, 15469, 15478, 15483, 15752, 15759, 15761, 15766, 15778, 15781, 15803, 15804, 15811, 15812, 15834, 15837, 15849, 15854, 15856, 15863, 16010, 16013, 16016, 16023, 16033, 16062, 16091, 16098, 16108, 16117, 16139, 16148, 16160, 16167, 16185, 16193, 16198, 16216, 16223, 16242, 16523, 16532, 16544, 16551, 16569, 16577, 16582, 16600, 16607, 16626, 16650, 16653, 16656, 16663, 16673, 16702, 16731, 16738, 16748, 16757, 16904, 16911, 16913, 16918, 16930, 16933, 16955, 16956, 16963, 16964, 16986, 16989, 17001, 17006, 17008, 17015, 17284, 17289, 17298, 17315, 17326, 17336, 17351, 17361, 17372, 17389, 17398, 17403, 17412, 17417, 17426, 17443, 17454, 17464, 17479, 17489, 17500, 17517, 17526, 17531, 17800, 17807, 17809, 17814, 17826, 17829, 17851, 17852, 17859, 17860, 17882, 17885, 17897, 17902, 17904, 17911, 18058, 18061, 18064, 18071, 18081, 18110, 18139, 18146, 18156, 18165, 18187, 18196, 18208, 18215, 18233, 18241, 18246, 18264, 18271, 18290, 18434, 18437, 18459, 18460, 18472, 18479, 18481, 18486, 18505, 18510, 18512, 18519, 18531, 18532, 18554, 18557, 18830, 18835, 18845, 18852, 18874, 18882, 18885, 18920, 18927, 18929, 19075, 19084, 19093, 19098, 19110, 19113, 19120, 19135, 19136, 19151, 19158, 19161, 19173, 19178, 19187, 19196, 19201, 19206, 19224, 19231, 19242, 19245, 19251, 19275, 19276, 19282, 19285, 19296, 19303, 19321, 19326, 19584, 19591, 19609, 19614, 19626, 19629, 19635, 19659, 19660, 19666, 19669, 19681,

19704, 19711, 19715, 19724, 19733, 19738, 19750, 19753, 19760, 19775, 19791, 19798, 19801, 19813, 19818, 19827, 19836, 19982, 19987, 19997, 20004, 20026, 20034, 20037, 20072, 20079, 20081, 20407, 20424, 20446, 20459, 20468, 20483, 20492, 20501, 20506, 20518, 20521, 20528, 20543, 20544, 20559, 20566, 20569, 20581, 20586, 20595, 20604, 20865, 20870, 20888, 20895, 20907, 20908, 20914, 20917, 20938, 20941, 20947, 20948, 20960, 20967, 20985, 20990, 21127, 21145, 21150, 21156, 21162, 21171, 21186, 21196, 21205, 21217, 21231, 21240, 21248, 21262, 21267, 21277, 21300, 21306, 21317, 21350, 21352, 21361, 21375, 21646, 21651, 21661, 21684, 21690, 21701, 21704, 21734, 21739, 21745, 21767, 21785, 21790, 21796, 21802, 21811, 21826, 21836, 21845, 21857, 21871, 21880, 22017, 22022, 22040, 22047, 22059, 22060, 22066, 22069, 22090, 22093, 22099, 22100, 22112, 22119, 22137, 22142, 22402, 22405, 22427, 22428, 22440, 22447, 22449, 22454, 22473, 22478, 22480, 22487, 22499, 22500, 22522, 22525, 22666, 22669, 22672, 22679, 22689, 22718, 22747, 22754, 22764, 22773, 22795, 22804, 22816, 22823, 22841, 22849, 22854, 22872, 22879, 22898, 23044, 23049, 23058, 23075, 23086, 23096, 23111, 23121, 23132, 23149, 23158, 23163, 23432, 23439, 23441, 23446, 23458, 23461, 23483, 23484, 23491, 23492, 23514, 23517, 23529, 23534, 23536, 23543, 23560, 23567, 23569, 23574, 23586, 23589, 23611, 23612, 23619, 23620, 23642, 23645, 23657, 23662, 23664, 23671, 23940, 23945, 23954, 23971, 23982, 23992, 24007, 24017, 24028, 24045, 24054, 24059, 24203, 24212, 24224, 24231, 24249, 24257, 24262, 24280, 24287, 24306, 24330, 24333, 24336, 24343, 24353, 24382, 24411, 24418, 24428, 24437, 24577, 24582, 24600, 24607, 24619, 24620, 24626, 24629, 24650, 24653, 24659, 24660, 24672, 24679, 24697, 24702, 24960, 24967, 24985, 24990, 25002, 25005, 25011, 25012, 25035, 25036, 25042, 25045, 25057, 25062, 25080, 25087, 25218, 25221, 25243, 25244, 25256, 25263, 25265, 25270, 25289, 25294, 25296, 25303, 25315, 25316, 25338, 25341, 25347, 25356, 25365, 25370, 25382, 25385, 25392, 25407, 25408, 25423, 25430, 25433, 25445, 25450, 25459, 25468, 25731, 25740, 25749, 25754, 25766, 25769, 25776, 25791, 25792, 25807, 25814, 25817, 25829, 25834, 25843, 25852, 25858, 25861, 25883, 25884, 25896, 25903, 25905, 25910, 25929, 25934, 25936, 25943, 25955, 25956, 25978, 25981, 26112, 26119, 26137, 26142, 26154, 26157, 26163, 26164,

26187, 26188, 26194, 26197, 26209, 26214, 26232, 26239, 26497, 26502, 26520, 26527, 26539, 26540, 26546, 26549, 26570, 26573, 26579, 26580, 26592, 26599, 26617, 26622, 26760, 26767, 26769, 26774, 26786, 26789, 26811, 26812, 26819, 26842, 26845, 26857, 26862, 26864, 26871, 26884, 26889, 26898, 26915, 26926, 26936, 26951, 26961, 26972, 26989, 26998, 27003, 27147, 27156, 27168, 27175, 27193, 27201, 27230, 27250, 27530, 27533, 27536, 27543, 27553, 27582, 27611, 27618, 27628, 27637, 27658, 27661, 27664, 27671, 27681, 27710, 27739, 27746, 27756, 27765, 28043, 28052, 28064, 28071, 28089, 28097, 28102, 28120, 28127, 28146, 28292, 28297, 28306, 28323, 28334, 28344, 28359, 28369, 28380, 28397, 28406, 28411, 28424, 28431, 28433, 28438, 28450, 28453, 28475, 28476, 28483, 28484, 28506, 28509, 28521, 28526, 28528, 28535, 28804, 28809, 28818, 28835, 28846, 28856, 28871, 28881, 28892, 28909, 28918, 28923, 28936, 28943, 28945, 28950, 28965, 28987, 28988, 28995, 28996, 29018, 29021, 29033, 29038, 29040, 29047, 29194, 29197, 29200, 29207, 29217, 29246, 29275, 29282, 29292, 29301, 29579, 29588, 29600, 29607, 29625, 29633, 29638, 29656, 29663, 29682, 29707, 29716, 29728, 29735, 29753, 29761, 29766, 29784, 29791, 29810, 30090, 30093, 30096, 30103, 30113, 30142, 30171, 30178, 30188, 30197, 30344, 30351, 30353, 30358, 30370, 30373, 30395, 30396, 30403, 30404, 30426, 30429, 30441, 30446, 30448, 30455, 30468, 30473, 30482, 30499, 30510, 30520, 30535, 30545, 30556, 30573, 30582, 30587, 30727, 30745, 30750, 30756, 30762, 30771, 30786, 30796, 30805, 30817, 30831, 30840, 31106, 31109, 31131, 31132, 31144, 31151, 31153, 31158, 31177, 31182, 31184, 31191, 31203, 31204, 31226, 31229, 31360, 31374, 31379, 31389, 31403, 31412, 31429, 31462, 31464, 31473, 31532, 31538, 31541, 31560, 31571, 31572, 31873, 31878, 31896, 31903, 31916, 31922, 31925, 31946, 31949, 31955, 31956, 31968, 31975, 31993, 31998, 32000, 32014, 32019, 32029, 32043, 32052, 32069, 32102, 32104, 32113, 32258, 32261, 32283, 32284, 32296, 32303, 32305, 32310, 32329, 32334, 32336, 32343, 32355, 32356, 32378, 32381, 32643, 32652, 32661, 32666, 32678, 32681, 32688, 32703, 32704, 32719, 32726, 32729, 32741, 32746, 32755, 32764, 32910, 32915, 32925, 32932, 32954, 32962, 32965, 33000, 33007, 33009, 33079, 33094, 33096, 33131, 33140, 33341, 33374, 34237, 34270, 34487, 34504, 34539, 34548, 34574, 34579, 34589, 34596, 34618,

34626, 34629, 34664, 34671, 34673, 34854, 34880, 34895, 34902, 34997, 35034, 35202, 35205, 35227, 35228, 35240, 35247, 35254, 35273, 35278, 35280, 35287, 35299, 35300, 35325, 35331, 35332, 35353, 35368, 35375, 35378, 35532, 35558, 35614, 35637, 35658, 35661, 35667, 35668, 35681, 35704, 35711, 35729, 35770, 35842, 35845, 35870, 35884, 35891, 35945, 35956, 35992, 36011, 36038, 36045, 36051, 36064, 36094, 36172, 36178, 36198, 36344, 36430, 36432, 36439, 36451, 36474, 36477, 36739, 36748, 36774, 36777, 36784, 36799, 36800, 36815, 36822, 36825, 36837, 36842, 36851, 37130, 37141, 37215, 37216, 37297, 37370, 37378, 37381, 37420, 37427, 37481, 37492, 37586, 37631, 37657, 37675, 37763, 37764, 37800, 37810, 37821, 37833, 37838, 37863, 37891, 37892, 37913, 37928, 37935, 37938, 37966, 38047, 38069, 38100, 38119, 38137, 38196, 38219, 38274, 38277, 38316, 38355, 38428, 38538, 38560, 38908, 39105, 39147, 39156, 39187, 39204, 39226, 39229, 39234, 39237, 39279, 39281, 39438, 39496, 39863, 40334, 40349, 40392, 40595, 40634, 40637, 40642, 40645, 40670, 40680, 40687, 40689, 40811, 40820, 41127, 41145, 41161, 41172, 41218, 41221, 41246, 41260, 41299, 41341, 41440, 41482, 41493, 41504, 41645, 41652, 41778, 42113, 42118, 42156, 42162, 42186, 42337, 42383, 42389, 42523, 42543, 42545, 42550, 42569, 42596, 42653, 42666, 42707, 42750, 42836, 42906, 43313, 43368, 43548, 43575, 43586, 43615, 43636, 43648, 43783, 43785, 43965, 43976, 43998, 44015, 44093, 44104, 44143, 44197, 44240, 44255, 44259, 44289, 44308, 44471, 44482, 44532, 44845, 44924, 45208, 45227, 45254, 45267, 45310, 45389, 45396, 45414, 45432, 45440, 45447, 45485, 45494, 45515, 45598, 45860, 45979, 45980, 46032, 46039, 46049, 46135, 46144, 46187, 46196, 46306, 46542, 46564, 46589, 46593, 46626, 46668, 46678, 46717, 46855, 46888, 46901, 46922, 46931, 46985, 46998, 47011, 47022, 47024, 47045, 47098, 47104, 47133, 47156, 47179, 47186, 47231, 47237, 47246, 47283, 47336, 47366, 47403, 47454, 47514, 47536, 47551, 47577, 47589, 47612, 47775, 47788, 47797, 47818, 47821, 47828, 47847, 47865, 47926, 47971, 47994, 47997, 48138, 48177, 48283, 48284, 48303, 48310, 48329, 48378, 48408, 48415, 48428, 48434, 48480, 48487, 48505, 48661, 48681, 48703, 48719, 48729, 48741, 48746, 48755, 48772, 48898, 49099, 49100, 49106, 49126, 49151, 49288, 49325, 49334, 49379, 49404, 49411, 49412, 49433, 49448, 49455, 49490, 49500, 50058, 50069,

50080, 50111, 50186, 50197, 50208, 50239, 50496, 50819, 50820, 50841, 50856, 50863, 50866, 50894, 50952, 50989, 50998, 51043, 51068, 51329, 51334, 51359, 51412, 51449, 51506, 51710, 51764, 51782, 51800, 51819, 52103, 52139, 52140, 52327, 52388, 52657, 52682, 52736, 52743, 52777, 52827, 52854, 52892, 52917, 52937, 53017, 53092, 53138, 53205, 53378, 53381, 53403, 53404, 53454, 53456, 53463, 53476, 53538, 53577, 53629, 53807, 53814, 53859, 53882, 53929, 53936, 53963, 54023, 54049, 54082, 54092, 54107, 54182, 54230, 54250, 54259, 54321, 54354, 54409, 54422, 54435, 54509, 54526, 54529, 54630, 54651, 54682, 54704, 54711, 54735, 54745, 54760, 54933, 54976, 55103, 55134, 55141, 55341, 55351, 55390, 55398, 55400, 55474, 55517, 55560, 55598, 55651, 55668, 55683, 55701, 55721, 55744, 55829, 55835, 55840, 55960, 56019, 56062, 56092, 56144, 56151, 56488, 56582, 56629, 56653, 56659, 56702, 56801, 56844, 56870, 56956, 57059, 57060, 57091, 57128, 57138, 57209, 57216, 57246, 57261, 57268, 57566, 57658, 57713, 57875, 57925, 57928, 57967, 58306, 58344, 58356, 58434, 58472, 58771, 58788, 58821, 58824, 58859, 59121, 59406, 59460, 59669, 59675, 59680, 59714, 59788, 59814, 59855, 59862, 59882, 59891, 59917, 59930, 60012, 60017, 60082, 60128, 60207, 60238, 60313, 60353, 60419, 60441, 60454, 60502, 60528, 60622, 60669, 60826, 60829, 60908, 61002, 61071, 61354, 61363, 61432, 61501, 61583, 61600, 61623, 61640, 61956, 61977, 61992, 62030, 62034, 62081, 62138, 62263, 62315, 62344, 62354, 62413, 62462, 62469, 62472, 62483, 62494, 62574, 62765, 62854, 62876, 62905, 62915, 62966, 63024, 63259, 63296, 63471, 63745, 64003, 64064, 64150, 64266, 64287, 64297, 64325, 64345, 64358, 64420, 64570, 64747, 64842, 64848, 64981, 64990, 65177, 65207, 65224, 65341, 65377}

Príloha C

Článok na IIT.SRC 2014

Evolutionary Generation of Error-Correcting Codes

Filip Pakan*

Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies
Ilkovičova 2, 842 16 Bratislava, Slovakia
xpakanf@stuba.sk

Abstract. In today's interconnected world we experience a huge amount of data being transmitted over the network. However, sometimes errors may occur in transmitted data due to electromagnetic interference and thermal noise. Encoding the message in a redundant way by using an error-correcting code gives receivers the ability to recover corrupted data. Larger error-correcting codes allow more messages to be encoded and hence improve communication efficiency. Unfortunately, the generation of error-correcting codes is equivalent to NP-complete problem of finding maximum clique in a graph. An exhaustive search for a solution is not feasible due to the exponential complexity of the problem. One possibility is to employ stochastic optimization algorithms inspired by biological evolution. Evolution is a process where over many generations the optimal solution may be achieved. In this work we analyze recent evolutionary approaches to the generation of optimal error-correcting codes and finding the maximum clique in a graph. We propose a simple evolutionary heuristic for the generation of error-

* Master degree study programme in field: Software Engineering
Supervisor: Professor Jiří Pospíchal, Institute of Applied Informatics, Faculty of Informatics and Information Technologies STU in Bratislava

correcting codes. Our solution is based on greedy approach of generating lexicographic codes and the results after several experiments look very promising. We believe that our solution can outperform rather complicated genetic algorithms in terms of speed. Proposed algorithm is a very efficient approach to code discovery.

1 Introduction

When data is transmitted over the network or stored on data storage devices, errors may occur for a variety of reasons such as noise in the transmission or dirt on the storage media. For a binary data this has the effect of flipping 0 to 1 or vice-versa. It is not always possible to ask the sender for retransmission, because some services (namely VoIP, IPTV) have to operate in real time. These services require low latency, high throughput and huge amount of multimedia data to be transferred. Therefore these services mostly rely on UDP protocol which doesn't slow down the communication by retransmitting corrupted packets but on the other hand reliable data transfer is not guaranteed. This is a perfect scenario for employment of error-correcting codes.

Larger error-correcting codes increase the maximum sizes of transmittable messages and hence improve communication efficiency. Discovering optimal error-correcting codes for different code specifications is a hard problem that we are addressing in this work. We try to apply evolutionary algorithms to this problem and find some suitable evolutionary heuristic for generating error-correcting codes. It is not goal of this paper to find new larger error-correcting code for given parameters since this usually requires months of CPU time. Our goal is to find suitable technique for generating error-correcting codes.

2 Error-correcting codes

Error-correcting codes are mathematical constructs from the field of coding theory that offer communication error detection and correction. They were first pioneered by Richard Hamming in 1950 [2]. Prior to transmission, a message is encoded by adding redundant information to form a code word. The redundant information allows for a distance to be maintained between code words, which allows for error correction.

An error-correcting code is a set of code words. Every error-correcting code is specified by 3 parameters:

1. n – length of the code words (number of bits)
2. M – number of code words
3. d – minimum distance between each pair of code words

An error-correcting code is usually denoted as *(n, M, d)* code of M code words, each of length n and any pair of code words differs in at least d positions. The minimum distance of the code determines the number of errors that may be detected and corrected. In this paper, the distance measure used is Hamming distance – the number of positions at which corresponding bits are different in two binary strings of the same length.

If a corrupted code word that does not match a known code word is received, it is corrected by replacing it with the closest known code word (Figure 1).

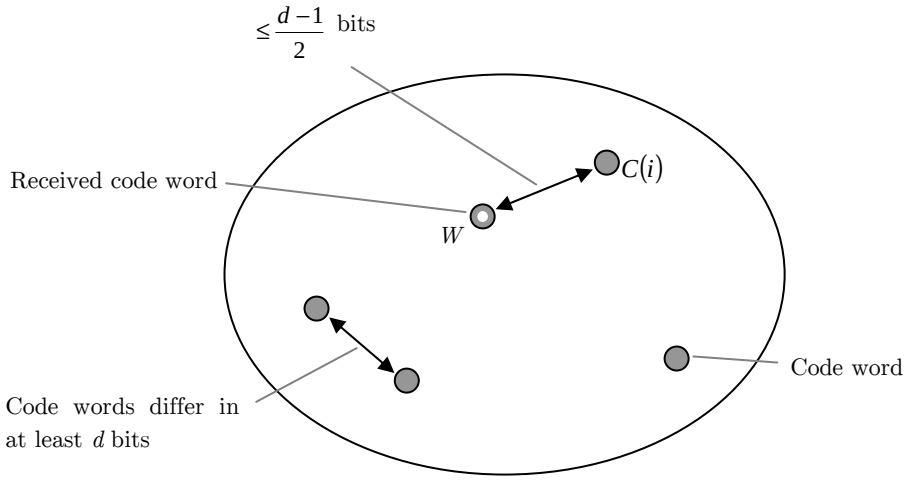


Figure 1. Correcting of corrupted code word by replacing it with the closest known code word

3 Generation of error-correcting codes

There are several desirable properties for a code. A good *(n, M, d)* code should have large M as this determines the number of different pieces of information that may be represented by the code. It should have small n, so that the code words are not too long which would affect communication efficiency. Then it should have large d, so that we can detect and correct as many errors as possible. As we can see, these properties are clearly conflicting. The main problem of coding theory is to optimize one of these parameters for given values of the other two. In this work we try to maximize number of code words M for given length of the code n and minimum distance d. The maximum possible number of code words in a code of length n and minimum distance d is denoted by $A_2(n, d)$.

3.1 Reduction to maximum clique problem

Every time the program looks at a possible solution, it needs to calculate distance between each pair of code words in order to meet minimum distance requirement. The distances between each pair of code words can be pre-computed and stored in compatibility matrix.

We can always assume presence of all-zero code word in the code since any code not containing the all-zero code word is equivalent to one that contains it. So we can generate all possible candidate code words of length n and distance at least d from all-zero code word. We store results in compatibility matrix A in which $a_{ij} = 1$ if code word i and code word j are compatible (i.e. meet minimum distance requirement), and 0 otherwise. It follows from the definition of compatibility matrix that this matrix is symmetric.

Suppose we wish to find maximum number of code words in a binary code of length 4, minimum distance 2 and the following words are the only possible candidate code words: {0000, 0011, 1010, 1011, 1110, 1111}. The compatibility matrix for this problem is shown in Table 1.

Table 1. Example compatibility matrix

	0000	**0011**	**1010**	**1011**	**1110**	**1111**
0000	0	1	1	1	1	1
0011	1	0	1	0	1	1
1010	1	1	0	0	0	1
1011	1	0	0	0	1	0
1110	1	1	0	1	0	0
1111	1	1	1	0	0	0

Turning compatibility matrix shown in Table 1 into adjacency matrix of a graph is straightforward. Vertices of a graph correspond to code words and two vertices are connected by an edge if two corresponding code words meet minimum distance requirement. A particular graph for a given adjacency matrix is depicted in Figure 2.

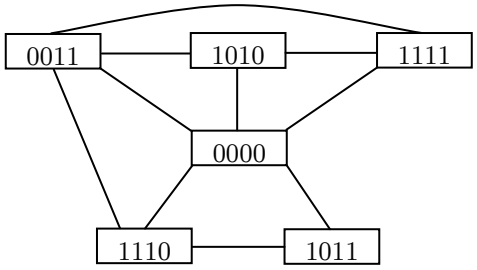


Figure 2. Graph for a given compatibility matrix [1]

Since the resulting code must contain only compatible code words, there must be an edge between each pair of corresponding vertices. So basically we have to find maximum complete sub-graph of a given graph also known as maximum clique. The set of vertices in maximum clique is equivalent to optimal error-correcting code. It is worth noting that maximum clique problem is NP-complete.

3.2 Search space

Let's take a closer look at the size of search space. By assuming the existence of all-zero code word in a code, we may generate all possible candidate code words of length n and distance at least d from the all-zero word. The minimum distance d from the all-zero code word means that every candidate code word must have at least d bits set to 1. Number of code words of length n and minimum distance d from all-zero code word equals to:

$$\sum_{i=d}^{n} \binom{n}{i} \tag{1}$$

Every candidate code word may belong to the code or not, therefore the complexity of this problem is exponential by the number of candidate code words. This exponential complexity prohibits naïve exhaustive search.

3.3 Relationship between parameters

A certain relationship exists between code parameters. By making use of it we may significantly reduce the search space.

Consider *(n, M, 2r-1)* code of *M* code words, each of length *n* and with minimum distance *2r-1*. By adding a parity bit we get *(n+1, M, 2r)* code, thus $A_2(n, 2r-1) \leq A_2(n+1, 2r)$. Let's note that a parity check adds a bit at the end of every code word. This bit is 1 if the remaining part of the code word has an odd number of 1's and 0 otherwise. As a consequence every code word has now even number of 1's.

Now consider *(n+1, M, 2r)* code. By removing one bit we get *(n, M, d)* code where $d \geq 2r-1$. Therefore we have $A_2(n, 2r-1) \geq A_2(n+1, 2r)$. Since $A_2(n, 2r-1) \leq A_2(n+1, 2r)$ and $A_2(n, 2r-1) \geq A_2(n+1, 2r)$, they must be equal. In conclusion a code of length *n* and minimum distance *2r-1* has fewer possible code words than a code of length *n+1* and minimum distance *2r*. This reduces the number of candidate code words by half [3].

4 Related work

Several recent papers address the problem of generating optimal error-correcting codes. Considering the fact that this problem is equivalent to a well known maximum clique problem, the research of this problem has a long history.

Haas and Houghten [1] made a comparison of stochastic optimization algorithms for generating of error-correcting codes. Namely they compared hill climbing, beam search, simulated annealing, greedy algorithm for generating lexicographic codes, randomized greedy and several variants of genetic algorithm with two different solution representations. Genetic algorithm with indirect chromosome representation combined with greedy lexicographic algorithm outperformed all other tested algorithms. However only 6 – 9 code words were chosen by genetic algorithm and thousands of code words in resulting code were actually selected by greedy algorithm.

McCarney, Houghten and Ross [5] examined genetic algorithms and genetic programming for generating optimal error-correcting code. The use of genetic programming is novel in this domain. They compared genetic programming to the same type of genetic algorithm as described in previous article which was considered as superior approach among other approaches at that time. In this more recent work they have shown that genetic programming is very competitive approach and there was no clear winner for all codes examined. Both GA and GP provided similar results but running GP took 3 times longer than GA.

Marchiory [4] proposed simple heuristic based genetic algorithm for maximum clique problem. In previous research it had been shown that pure genetic algorithm is not suitable for maximum clique problem. Therefore author introduced a simple heuristic to support optimization of genetic algorithm. The idea is to build a maximal clique by starting with a random subgraph of given graph. This graph is first enlarged by adding some randomly selected nodes, next it is reduced to a clique and finally it is extend using naïve sequential greedy heuristic. Despite its simplicity, this heuristic can outperform all other approaches based on genetic algorithm for finding maximum clique.

5 Proposed algorithm

Our proposed algorithm is based on greedy algorithm for generating lexicographic codes. Lexicographic codes are greedily generated binary error-correcting codes with remarkably good properties. They are generated in lexicographic (dictionary) order. A lexicographic code of minimum distance d and length n is generated by starting with the all-zero vector and iteratively adding the next vector in lexicographic order of minimum Hamming distance d from the vectors added so far. Since this algorithm is deterministic it always gives the same result.

Our algorithm is stochastic variant of lexicographic algorithm. Since the genetic algorithm described in [1] is not very beneficial in generating code words, we try to replace it with something faster. We randomly generate several compatible code words and afterwards we add them to the resulting code. Then we run greedy lexicographic algorithm which will try to optimize the code including those code words that had been inserted at the beginning. By doing this we will slightly modify the standard behavior of this greedy algorithm because now the greedy must cope with code words already present in result set. This stochastic version will provide various results. Therefore it is wise to introduce multi-start as an integral part of the algorithm and return the best solution found over multiple runs.

Randomized greedy in [1] was heavily modified by adding code words in arbitrary order what in our opinion destroyed the original greedy idea. We still use the main idea of greedy lexicographic algorithm that provides good results and we expect that our modifications can produce better results.

The only parameter of the algorithm that needs to be set is number of randomly selected and inserted code words at the beginning. The optimal value depends on code parameters and thus cannot be once optimized and hardcoded in a program. If the value is too small, we will not influence the run of original algorithm enough and resulting code will not be very different. If the value is too big we will limit greedy algorithm too much and the original greedy idea cannot optimize so well.

6 Implementation

Being able to efficiently compute Hamming weight of binary string is crucial for this application. Fortunately latest processors have implemented the new instruction set SSE4.2 which includes specific instruction for counting number of bits set to 1. This instruction is called POPCNT (population count) and requires 3 clock cycles for execution. The presence of this instruction in processor is indicated by 23^{rd} bit in ECX register after executing CPUID instruction.

According to our measurements, computing Hamming weight with POPCNT instruction is more than 3 times faster than a well known HAKMEM algorithm which is still 10 times faster than a trivial loop implementation. So we gain 30 times speedup comparing to loop implementation. The downside is that only the newest processor's architectures have support for this instruction. Nevertheless, this is by far the fastest way how to compute Hamming weight.

7 Results

Our stochastic lexicographic algorithm clearly outperformed randomized greedy algorithm from [1]. In each test case it produced code with larger number of code words. Moreover our algorithm provides comparable results as the best algorithm from mentioned article.

In addition it is significantly faster than complicated genetic algorithm with indirect chromosome representation combined with lexicographic algorithm. Results in more detail are shown in Table 2.

Table 2. Summary of results

	$A_2(12,6)$	$A_2(13,6)$	$A_2(17,6)$	$A_2(17,4)$
Best known	24	32	256	2720
Lexicographic algorithm	16	16	256	2048
Randomized greedy	13	20	129	1693
Stochastic lexicographic algorithm	24	32	256	2207
Genetic algorithm + Lexicographic finish	24	32	256	2238

8 Conclusion and further work

Finding the subset of code words that are compatible among hundreds of thousands candidates is not an easy task. Corresponding graph has almost 2 millions edges. Therefore we are convinced that greedy methods are the only feasible ones for such a big problem. We proposed an algorithm that outperforms some of the previously discovered methods.

The next challenge is improving our algorithm to achieve even better optimization. We also have to propose a way how to automatically optimize parameter of the algorithm – number of code words inserted to the result set.

We have been examining specific algorithms from the domain of coding theory so far. However, in the near future we would like to try out generic algorithms for finding maximum clique in a graph and compare provided results.

9 Acknowledgements

This contribution was supported by Grant Agency VEGA SR under the grants 1/0553/12 and 1/0458/13.

References

[1] Haas, W., Houghten, S.: A Comparison of Evolutionary Algorithms for Finding Optimal Error-Correcting Codes. *In Proceedings of the 3rd IASTED International Conference on Computational Intelligence*, ACTA Press, (2007), pp. 64-70.

[2] Hamming, R.W.: Error Detecting and Error Correcting Codes. *The Bell System Technical Journal*, (1950), pp. 147-160.

[3] MacWilliams, F.J., Sloane, N.J.A.: The Theory of Error-Correcting Codes. *North Holland Publishing Company*, (1977).

[4] Marchiori, E.: A Simple Heuristic Based Genetic Algorithm for the Maximum Clique Problem. *In Proceedings of ACM Symposium on Applied Computing*, ACM Press, (1998), pp. 366-373.

[5] McCarney, E.D., Houghten, S., Ross, J.B.: Evolutionary Approaches to the Generation of Optimal Error Correcting Codes. *In Proceedings of the 14th International Conference on Genetic and Evolutionary Computation*, ACM Press, (2012), pp. 1135-1142.

Príloha D

Obsah elektronického média

- CD-R médium
  - Dokument
    - Diplomova_praca.doc
    - Diplomova_praca.pdf
  - Programy
    - Diplomový projekt
      - Brute force
      - Genetic algorithm
      - Lexicographic code
      - Lexicographic construction
      - Library
      - Maximum clique
      - Maximum clique greedy
      - Maximum clique tabu
      - Simple greedy
      - Diplomovy_projekt.sln

- Výsledky
  - Algorithms.xls
  - Generators.xls
  - Optimization.xls
  - Timing.xls