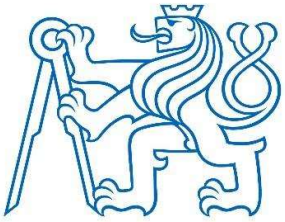


ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

FAKULTA DOPRAVNÍ

Bc. Patrik Baleka

Rozšíření knihoven implementující genetické algoritmy a umělé neuronové sítě

Diplomová práce

2024

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta dopravní
děkan
Konviktská 20, 110 00 Praha 1

K620..... Ústav dopravní telematiky

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení studenta (včetně titulů):

Bc. Patrik Baleka

Studijní program (obor/specializace) studenta:

navazující magisterský – ITS – Inteligentní dopravní systémy

Název tématu (česky): **Rozšíření knihoven implementující genetické algoritmy a umělé neuronové sítě**

Název tématu (anglicky): Extension of Libraries Implementing Genetic Algorithms and Artificial NN

Zásady pro vypracování

Při zpracování diplomové práce se řiďte následujícími pokyny:

- Seznamte se s vybranými bakalářskými pracemi zpracovávající tématiku knihovnen implementující genetické algoritmy a umělé neuronové sítě
- Prostudujte příslušné knihovny a jejich nedostatky
- Navrhněte rozšíření vybrané knihovny
- Navržené rozšíření implementujte

Rozsah grafických prací: dle pokynů vedoucího práce

Rozsah průvodní zprávy: minimálně 55 stran textu (včetně obrázků, grafů a tabulek, které jsou součástí průvodní zprávy)

Seznam odborné literatury:
Mařík a kol.: Umělá inteligence 3, Academia, 2001, ISBN 80-200-0472-6

Mařík a kol.: Umělá inteligence 4, Academia, 2003, ISBN 80-200-1044-0

Zelinka a kol.: Evoluční výpočetní techniky - Principy a aplikace, BEN, 2008, ISBN: 80-7300-218-3

Vedoucí diplomové práce: **doc. Ing. Tomáš Tichý, Ph.D.**
doc. Ing. Vít Fábera, Ph.D.

Datum zadání diplomové práce: **15. července 2023**
(datum prvního zadání této práce, které musí být nejpozději 10 měsíců před datem prvního předpokládaného odevzdání této práce vyplývajícího ze standardní doby studia)

Datum odevzdání diplomové práce: **15. května 2024**
a) datum prvního předpokládaného odevzdání práce vyplývající ze standardní doby studia a z doporučeného časového plánu studia
b) v případě odkladu odevzdání práce následující datum odevzdání práce vyplývající z doporučeného časového plánu studia

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE Fakulta dopravní

L. S.

Ing. Zuzana Bělinová, Ph.D.
vedoucí
Ústavu dopravní telematiky

prof. Ing. Ondřej Přibyl, Ph.D.
děkan fakulty

Potvrzuji převzetí zadání diplomové práce.

Bc. Patrik Baleka
jméno a podpis studenta

V Praze dne ..... 15. července 2023

Poděkování

Na tomto místě bych rád poděkoval všem, kteří mi poskytli podklady pro vypracování diplomové práce. Zvláště pak děkuji doc. Ing. Vítu Fáberovi, Ph.D. a doc. Ing. Tomáši Tichému, Ph.D za odborné vedení a konzultování diplomové práce. V neposlední řadě je mou milou povinností poděkovat své milované manželce, svým rodičům a blízkým za morální a materiální podporu.

Prohlášení

Předkládám tímto k posouzení a obhajobě svou diplomovou práci, zpracovanou na závěr studia na ČVUT v Praze Fakultě dopravní.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací a Rámcovými pravidly používání umělé inteligence na ČVUT pro studijní a pedagogické účely v Bc. a NM studiu.

Nemám závažný důvod proti užívání tohoto školního díla ve smyslu § 60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 15.05.2024 ……

Podpis

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta dopravní

Rozšíření knihoven implementující genetické algoritmy a umělé neuronové sítě

Diplomová práce

květen 2024

Patrik Baleka

Abstrakt

Předmětem diplomové práce „Rozšíření knihoven implementující genetické algoritmy a umělé neuronové sítě“ je rozšíření knihovny Galgo jazyka C++. Teoretická část práce se věnuje rozboru principu genetických algoritmů, neuronových sítí a různých knihoven na jejich generaci a práci s nimi. Praktická část následně poskytuje rozšíření vybrané knihovny v oblasti zastavovacího kritéria, metod křížení a metod mutace. Výsledná podoba knihovny po rozšíření byla ověřena na logickém problému.

Klíčová slova: Genetické algoritmy, umělé neuronové sítě, knihovny, C++

Abstract

The subject of the diploma thesis " Extension of libraries implementing genetic algorithms and artificial NN" is the extension of the Galgo library of the C++ language. The theoretical part of the work is devoted to the analysis of the principle of genetic algorithms, neural networks and various libraries for their generation and work with them. The practical part subsequently provides an extension of the selected library in the area of stopping criteria, crossover methods and mutation methods. The resulting form of the library after expansion was verified on a logic problem.

Key words: Genetic algorithms, artificial neural networks, libraries, C++

Obsah

Seznam použitých zkratek

GA	Genetické algoritmy
ET/EVT	Evoluční (výpočetní) techniky
NS	Neuronové sítě
BP/BPTT	Zpětné šíření (v čase) (backpropagation through time)
DE	Diferenciální evoluce
DS	deterministické vzorkování (deterministic sampling)
SRS	stochastické zbytkové vzorkování (stochastic residual sampling)
RNN	Rekurentní neuronové sítě (Recurrent neural network)
CNN	Konvoluční neuronové sítě (Convolutional neural networks)
LSTM	Typ neuronových sítí (Long short-term memory)
GRU	Typ neuronových sítí (Gated Recurrent Units)

Zkratky knihovny Galgo:

TNT	turnajová selekce
RWS	selekce ruletovým kolem (roulette wheel selection)
P1XO	jednobodové křížení
SPM	jednobodová mutace (single point mutation)

Úvod

V současné době jsou genetické algoritmy (GA) a neuronové sítě (NS) důležitými nástroji v oblasti umělé inteligence a optimalizace. Jejich kombinace a aplikace ve výpočetních systémech otevírá široké možnosti v různých oblastech, jako jsou strojové učení, optimalizace procesů nebo analýza dat. Pomocí těchto oblastí lze např. detekovat únavu řidičů.

Tato práce navazuje na předchozí bakalářské práce Radima Komendy [1] a Terezy Panské [2], které se zabývaly genetickými algoritmy a jejich aplikacemi. Cílem této diplomové práce je rozšířit stávající poznatky a implementace v oblasti GA a NS, především v kontextu programovacího jazyka C++.

První část práce poskytne přehled o genetických algoritmech a neuronových sítích, jejich základních principech a aplikacích. Dále budou představeny existující knihovny pro implementaci GA a NS v jazyce C++ a provedeno porovnání jejich vlastností a výkonnosti.

V druhé části práce bude provedeno rozšíření knihovny Galgo o nové metody a funkce, které budou diskutovány v této práci. Mezi tyto nové metody patří například uspořádané křížení (někdy též pořadové, anglicky ordered crossover), které umožňuje efektivnější křížení jedinců v genetickém algoritmu.

V závěrečné části budou prezentovány výsledky experimentů a testování nových metod a funkcí implementovaných v knihovně Galgo. Dále budou diskutovány možné budoucí směry vývoje v oblasti GA a NS a jejich aplikací v praxi.

Tato diplomová práce si klade za cíl přispět k rozvoji oblasti genetických algoritmů a neuronových sítí, zejména v kontextu programovacího jazyka C++, a poskytnout ucelený přehled o stávajících metodách a implementacích v této oblasti.

1 Evoluční výpočetní techniky

V této kapitole budou představeny principy a základní komponenty evolučních výpočetních technik (dále jen EVT), jako jsou populace, chromozomy, geny, fitness funkce, selekce, křížení a mutace. Dále bude obsahovat některé aplikace genetických algoritmů v různých oblastech, jako je strojové učení, plánování, umění nebo bioinformatika. V české literatuře najdeme EVT například v knihách série *Umělá inteligence I-V* (převážně v dílech III a IV) [3]. Dále se tímto tématem zabývá také prof. Ivan Zelinka, který uvádí v [4] EVT následovně: „*Evoluční techniky jsou numerické algoritmy, které vycházejí ze základních principů Darwinovy a Mendelovy teorie evoluce, jejíchž hlavní ideou je předávání rodičovského genomu novým potomkům a následné uvolnění životního prostoru potomkům.“* [4, s. 26]

I když stejná myšlenka, jakou měli Darwin a Mendel se vyskytuje už ve starověku [4], počátky samotných EVT spadají do 60. let 20. století do Berlína, kde se tři studenti tamní univerzity zabývali konstrukcí převodovek. Studenti náhodně kombinovali dvojice existujících konstrukcích a snažili se najít lepší s užitečnými vlastnostmi [3]. Dalším příkladem využití evoluce biologického druhu, tentokrát zamýšlené, je evoluční programování z roku 1966 Artificial Intelligence through Simulated Evolution [5] (kolektiv autorů vedených J. L. Fogelem). Zabývali se konečnými automaty a vzhledem k jejich schopnostem predikce prostředí, ve kterém se vyvíjely [3].

Základy genetických algoritmů položil americký teoretický biolog John Holland v roce 1975 ve své knize [6]. Genetické programování, které vychází z genetických algoritmů, pak vynalezl v devadesátých letech John Koza. [3]

1.1 Základní charakteristiky EVT

EVT mají nejčastější uplatnění při prohledávání určitého prostoru. Zaměřují se na větší prostory, protože malé lze prozkoumat úplným prohledáním, větší však heuristickými či znalostně expertními metodami. Mohou to být metody založené na bodové strategii (např. simulované žíhání, horolezecký algoritmus nebo zakázané prohledávání) nebo strategii populace, kam se řadí genetické algoritmy. [4]

Oproti klasickým optimalizačním úlohám však EVT nepracují s jediným kandidátem, nýbrž s množinami tzv. *populacemi* kandidátů (jedinců) $x_{t,i}$. Značí se pak $G(t) = \{ x_{t,1}, x_{t,2}, \ldots, x_{t,N} \}$. [3] Také umožňují zařazení horší varianty do dalšího kola (generace), čímž se snaží překonat problém lokálního optima. [4]

1.1.1 Kvalita jedinců

Kvalita, neboli fitness, jedince je základním pojmem v evolučních výpočetních technikách, které jsou inspirovány biologickou evolucí. Jedinec reprezentuje možné řešení daného optimalizačního problému a má přiřazenou hodnotu fitness, která vyjadřuje jeho kvalitu nebo přizpůsobivost. Od jeho kvality se odvozuje pravděpodobnost přežití do následující generace. Fitness jedince je možné vždy určit. Evoluční výpočetní techniky pracují s populací fitness jedinců, kterou postupně zlepšují pomocí operátorů selekce, křížení a mutace (viz následující podkapitola). [3]

1.1.2 Algoritmus EVT

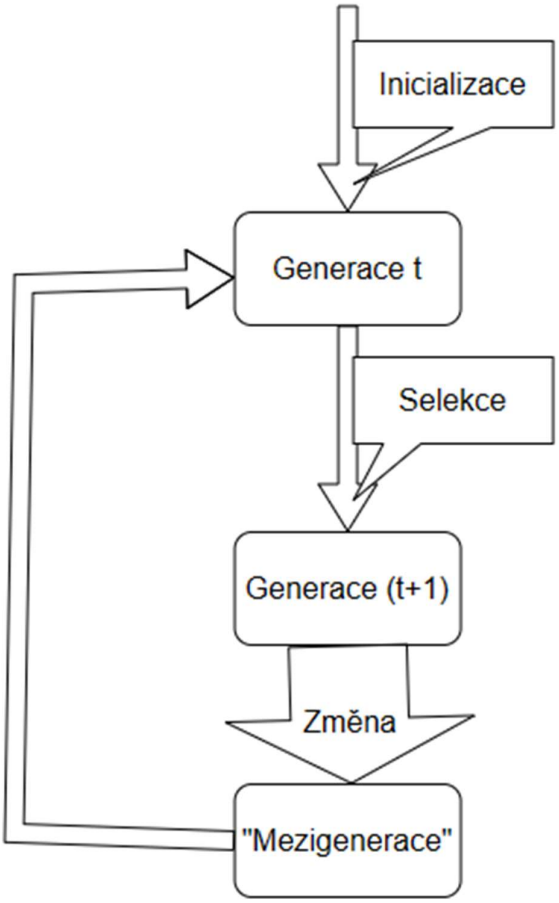
Obecný tvar algoritmu EVT lze pak zapsat následovně (viz obrázek 1).

```
begin
    t := 0;
    inicializace G(t);
    vyhodnocení G(t);
    while (not zastavovací_pravidlo) do
    begin
        t:= t + 1;
        selekce G(t) z G(t-1);
        změna G(t);
        vyhodnocení G(t);
    end
end
```

Obrázek 1 Základní tvar algoritmu EVT, zdroj: Lažanský [3]

Jedná se o algoritmus, který simuluje vývojový čas pomocí cyklu. Na začátku je nastaven vývojový čas na nulu, vhodně zvolena počáteční populace jedinců v kroku **iniciace** G(t). Většinou se využívá metody vytvoření množiny náhodných jedinců, avšak je také možné využít apriorní znalosti o úloze, pokud jsou k dispozici. [3]

Výpočet kvality všech jedinců (fitness funkce) se provádí v operaci **vyhodnocení** G(t). Obvykle se zde určí nejlepší jedinec a další statistické vlastnosti populace. Takto získaná kvalita je použita v následujícím cyklu (vývojový čas se zvýší o jedna) v kroku **selekce**. Zde se simuluje přirozený výběr a vybírá se nová populace na základě té předešlé většinou pravděpodobnostním mechanismem na základě kvality (proměnu generací je možné vidět na obrázku 2). Změna je pak inovativním prvkem celého procesu a provádí se pomocí tzv. *rekombinačních operátorů*: křížení a mutace. Změnových operátorů existuje více a vždy jsou těsně svázány s danou reprezentací jedinců. [3] Tyto rekombinační operátory spolu se selekcí budou dále popsány v rámci genetických algoritmů.



Obrázek 2 Vývojový diagram algoritmu EVT zdroj: vlastní podle Lažanský [3, s. 120]

Zastavovací pravidlo v algoritmu (viz Obrázek 1) slouží k ukončení celého algoritmu. Slouží k tomu, aby se populace příliš nehomogenizovala nebo pokud v daném případě neplatí, že čím déle algoritmus běží, tím lepší je výsledek. Při práci s GA je důležité si uvědomit, že jejich chování není při každém spuštění konzistentní. Výsledky se mohou lišit, což znamená, že ne

vždy dosáhneme požadovaného výsledku hned na první pokus. Z tohoto důvodu je vhodné toto pravidlo zavést a stanovit tak např. maximální počet pokusů, po kterém se proces ukončí, pokud není dosaženo uspokojivého výsledku. J. Lažanský definuje zastavovací pravidlo jako „*jsme již spokojeni s dosaženým výsledkem; např. při minimalizaci výrobních nákladů jsme dosáhli rozumně nízké hodnoty*.“ [3, s. 121] U některých problémů známe ohodnocení optimálního řešení, např. pokud je fitness měřeno v procentech, tak optimum může být 100% nebo 0%.

1.2 Genetické algoritmy

Genetické algoritmy jsou jednou z klasických metod umělé inteligence, které napodobují proces přírodního výběru a evoluce. Cílem genetických algoritmů je nalézt optimální řešení složitých optimalizačních problémů, které jsou obtížně řešitelné klasickými metodami. U genetických algoritmů jsou jedinci reprezentováni pomocí struktur S ve tvaru řetězců $(s_1,s_2,\ldots,s_L)$, konečné délky L. Tyto struktury se nazývají chromozomy. [3]

Genetické algoritmy mají tu výhodu, že nepotřebují znát přesný tvar cílové funkce ani její vlastnosti. Jsou vhodné pro problémy, kde je prostor možných řešení velmi velký a složitý a lze je použít na různé typy úloh. Na druhou stranu jejich hlavní nevýhodou je, že nemohou zaručit nalezení globálního optima, protože mohou uváznout v lokálních minimech.

1.2.1 Selekce

Jak již bylo zmíněno, selekce simuluje přirozený výběr nové populace. Hlavním úkolem selekce je zvýšení průměru kvality. V selekci obecně platí, že nadprůměrní jedinci se objeví v nové populaci častěji, průměrní zhruba ve stejném počtu a omezí se počet slabších jedinců, tím se celkový průměr kvality zvýší. Schopnost selekčního mechanismu potlačovat podprůměrné jedince a zvýrazňovat jedince nadprůměrné se nazývá *selekční tlak*. [3]

Náhradové strategie jsou důležitou součástí genetických algoritmů. Jejich úkolem je vybrat z rozšířené množiny jedinců novou populaci o rozsahu N. Je mnoho náhradových strategií, ale mezi ty nejdůležitější se řadí:

- Náhodný výběr (ruletové kolo)
- Pořadová selekce
- Turnajová selekce
- Elitismus
- Postupný vývoj (steady state evolution)

Náhodný výběr (také mechanismus **ruletového kola**) pracuje s principem, že u algoritmů SGA (mají základ reprezentace pomocí „*chromozomů fixní délky tvořených dvouhodnotovými symboly“* [3, s. 121]) platí, že pravděpodobnost každého jedince je přímo úměrná jeho kvalitě. Vztah $Pr_i(i) = k_t f(x_{t,i})$, i=1,…,N pak určuje, s jakou pravděpodobností je vybrán *i*-tý jedinec v *t*-é generaci.

U **pořadové selekce** je každý jedinec uspořádán podle své kvality a ohodnocen, nejhorší 1 (nebo 0), nejlepší *N* (resp. *N-1*). „*Tato transformovaná ohodnocení se pak přepočtou na pravděpodobnosti selekce vynásobením konstantou takovou, aby součet pravděpodobností byl roven jedné*“ [3, s. 136].

Místo náhodného výběru podle kvality jednotlivců se používá **turnajová selekce**, která vybírá jednotlivce podle výsledku jejich souboje. Z původní populace se náhodně utvoří skupiny jednotlivců, kteří spolu soutěží o místo v další generaci. Jednotlivec nebo jednotlivci s nejlepším ohodnocením, tedy vítězové, postupují do další generace. Tento proces se opakuje, dokud není naplněna další generace. Tato selekce se kombinuje s ukládáním nejlepšího dosaženého řešení a poskytuje nejlepší výsledky. Turnajová selekce má také výhodu jednoduché implementace bez ohledu na to, zda řešený problém minimalizujeme nebo maximalizujeme. [3]

Elitismus „*vybere mezi všemi jedinci jednoho nebo několik nejlepších a zbytek do N se vhodně doplní zpravidla upřednostněním potomků a mutantů*.“ [3] **Postupný vývoj** pak spočívá v tom, že místo celých populací podstupuje rekombinaci jen nepatrná část populace. Zabrání se tak ztrátě nejlepších jedinců.

Selekce lze dále modifikovat, např. abychom neeliminovali nejlepšího jedince. Je proto vhodné zavést paměťovou pozici nejlepšího řešení, které zatím bylo nalezeno (anglicky *best so far).* [3]

1.2.2 Křížení

Jak již bylo zmíněno, křížení a mutace jsou základními změnovými operátory GA. Těmito operacemi mohou, ale nemusí, projít všichni selektovaní jedinci.

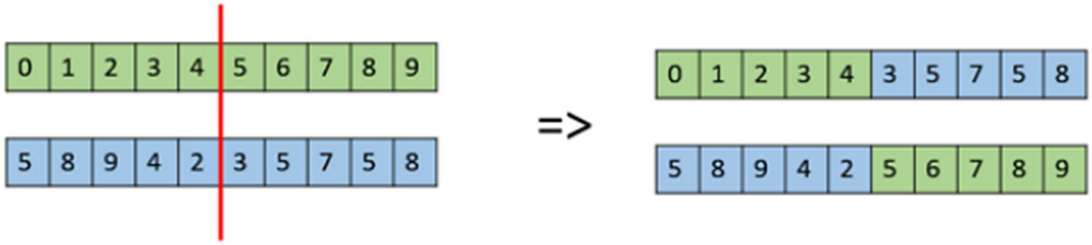
Křížení spočívá v tom, že se vyberou dva rodiče z populace a vymění se mezi nimi části jejich genotypů. Tím se vytvoří dva potomci, kteří mohou mít lepší přizpůsobivost než jejich rodiče. Křížení je založeno na biologické analogii, kdy se kombinují geny obou rodičů a vznikají nové variace. I když rodiče bývají většinou dva, na chování algoritmů EVT mají pozitivní vliv „orgie“. "Orgie" jsou speciální typ křížení, který používá více než dva rodiče k vytvoření potomka. Tento typ křížení může zvýšit diverzitu populace a zlepšit schopnost algoritmu přizpůsobit se složitým

problémům. Jedním z typů, kde se používá více než dvou rodičů, je diferenciální evoluce, při kterém se křížení účastní čtyři jedinci.

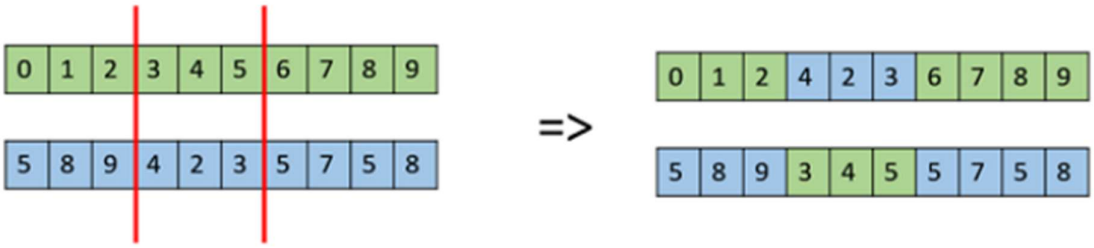
Mezi nejznámější typy křížení patří:

- **Jednobodové křížení** (One point crossover): Vybere se náhodně jeden bod křížení a potomci zdědí geny od jednoho rodiče před tímto bodem a od druhého rodiče za tímto bodem. Příklad jednobodového křížení je možné vidět na obrázku 3.
- **Dvoubodové křížení** (Two point crossover): Vyberou se náhodně dva body křížení a potomci zdědí geny od jednoho rodiče mezi těmito body a od druhého rodiče mimo tyto body. Příklad dvoubodového křížení je možné vidět na obrázku 4.
- **Uniformní křížení** (Uniform crossover): Pro každý gen se náhodně rozhodne, od kterého rodiče ho potomek zdědí. Tento typ křížení je nezávislý na počtu bodů křížení a může vytvářet různorodější potomky. Příklad uniformního křížení je možné vidět na obrázku 5.
- **Sudé – liché křížení** (Even-Odd crossover**):** Proces křížení zajišťuje výměnu hodnot atributů mezi vzorky, a tím vytváří nové vzorky jako staré vzorky. Je založen především na výměně hodnot atributů mezi vzorky na každém sudém nebo lichém místě (bitu). Navrhovaná metoda řeší problém nevyvážených dat generováním nových vzorků, díky nimž jsou data vyvážená. [7] Příklad tohoto křížení je v kapitole 5.4.1.
- **Křížení s částečnou shodou** (Partially matched crossover) funguje v počátečním přístupu jako dvoubodové křížení. Spočívá v generování potomků párováním dvojic hodnot mezi dvěma náhodnými indexy. Ale kromě toho musí mít potomstvo pouze jeden z každého očíslovaného genu. Proto se zmapují geny ve vnitřní části a následně se vymění geny ve vnější části podle vytvořené mapy. [8] Příklad tohoto křížení je možné vidět na obrázku 6.
- **Uspořádané křížení** (Ordered crossover) funguje zpočátku jako dvoubodové křížení. Je vybrán rozsah indexů genů, který se používá k vytvoření vnitřní části potomků. Vnější část je formována a začíná na pravé straně vnitřní části kopírováním genů v pořadí, pokud ještě nejsou přítomny. [9] Příklad tohoto křížení je možné vidět na obrázku 7.
- **Cyklické křížení** (Cycle crossover)**:** Náhodně se vyberou dva rodičovské chromozomy, a poté se vytvoří potomci tak, že se vytvoří cyklus, který zahrnuje určité alely z obou rodičů. Tento cyklus se vytváří postupným sledováním alel mezi rodiči na základě jejich umístění v chromozomu. Potomci jsou pak vytvořeni tak, že použijí alely obsažené v tomto cyklu z jednoho rodiče a doplní chybějící alely z druhého rodiče. Tímto způsobem se kombinují různé vlastnosti obou rodičů, což má za následek nové

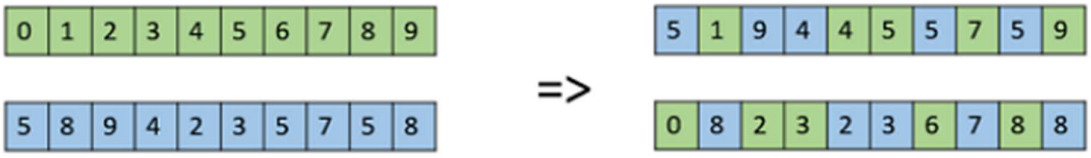
jedince, kteří mohou mít vylepšené vlastnosti oproti svým rodičům. [10] Příklad tohoto křížení je možné vidět na obrázku 8.



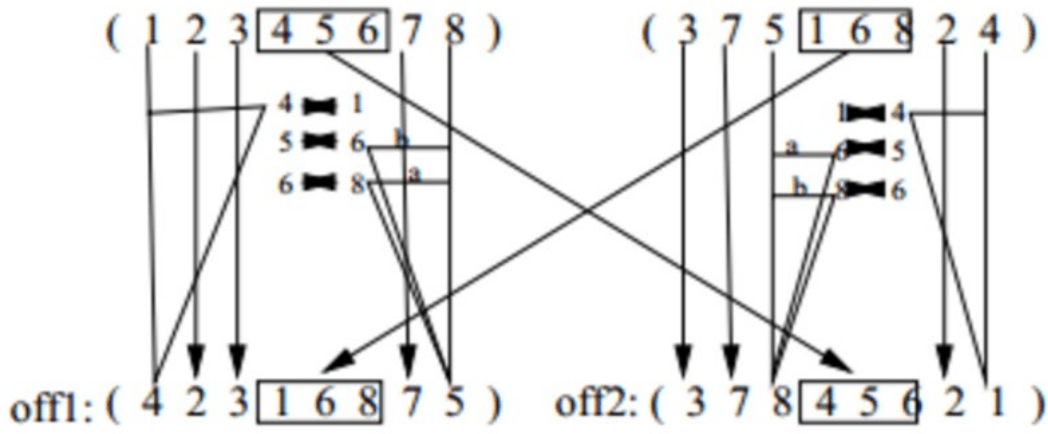
Obrázek 3 Jednobodové křížení zdroj převzato z [2]



Obrázek 4 Dvoubodové křížení převzato z [2]



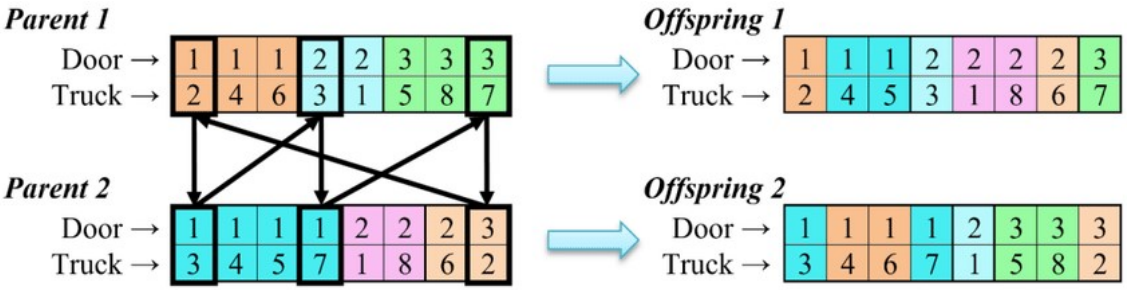
Obrázek 5 Uniformní křížení převzato z [2]



Obrázek 6 Křížení s částečnou shodou [8]

```
p1 = (8 4 7 3 6 2 5 1 9 0)    X    c1 = (0 4 7 3 6 2 5 1 8 9)
                 ---------                      ---------
p2 = (0 1 2 3 4 5 6 7 8 9)    X    c2 = (8 1 2 3 4 5 6 7 9 0)
```

Obrázek 7 Uspořádané křížení [9]



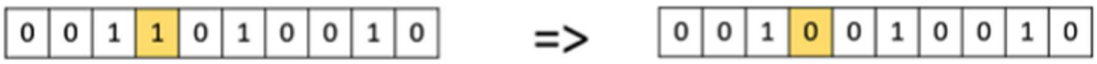
Obrázek 8 Cyklické křížení [10]

1.2.3 Mutace

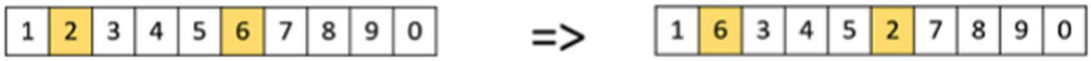
Mutace je druhým genetickým operátorem. Mutace náhodně mění některé bity v chromozomu jedince, a tím vytváří nové varianty. Mutace pomáhá překonávat lokální optima a udržovat populaci různorodou. J. Lažanský popisuje mutaci následovně: „Nad každým bitem v celé populaci (a těch je N. L) provedeme náhodný experiment s pravděpodobností úspěchu rovnou P_m. Vyjde-li příslušný výsledek, pak bit invertujeme“. [3]

Existuje několik druhů mutace, které se liší, podle toho, jakým způsobem a kolik chromozomů modifikují. Některé jsou např.:

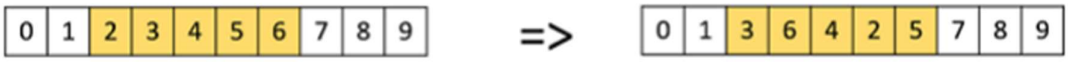
- Bodová mutace (viz Obrázek 9) – Jedná se o nejjednodušší typ mutace, kdy se náhodně změní jeden nebo více bitů v chromozomu. Tato mutace může mít malý nebo velký vliv na fitness chromozomu v závislosti na umístění bitu a typu problému. Při změně jednoho bitu se nazývá jednobodová, při změně n bitů pak *n*-bodová mutace. [4]
- Vložení – Náhodně se vloží nový bit do chromozomu na specifické pozici. Tato mutace může vést k prodloužení chromozomu a změně jeho struktury.
- Vymazání – Náhodně se vymaže jeden bit z chromozomu na specifické pozici. Tato mutace může vést ke zkrácení chromozomu a změně jeho struktury.
- Inverzní mutace (viz Obrázek 12) – Náhodně se vybere část chromozomu a pořadí bitů v této části se obrátí. Tato mutace může zachovat délku chromozomu, ale změní jeho strukturu.
- Translokace – Náhodně se vyberou dva úseky chromozomu a vymění se jejich pozice. Tato mutace může zachovat délku chromozomu, ale změní jeho strukturu. [11]
- Swap mutace (viz Obrázek 10) – prohazuje hodnoty dvou náhodně vybraných genů v chromozomu.
- Scramble mutace (viz Obrázek 11) – náhodně zamíchá vybranou podmnožinu genů. Tyto geny spolu mohou, ale také nemusí sousedit.



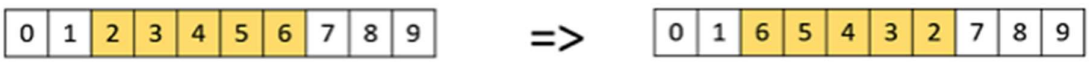
Obrázek 9 Jednobodová mutace [2]



Obrázek 10 Swap mutace [2]



Obrázek 11 Scramble mutace [2]



Obrázek 12 Inverzní mutace [2]

Další typy mutací mohou být:

- Zrcadlová mutace (Mirror mutation) a binární bit-flipping mutace jsou podobné v tom, že zrcadlový mutátor nahradí gen se svou zrcadlovou hodnotou ve středním bodě hraničního intervalu pro gen, zatímco v bit-stringové reprezentaci GA zůstává bit-flip mutace nezměněna. GA založené na pořadí a seskupení GA jsou instancemi GA mínus bit-flip mutace. [12]
- Náhodná (uniformní) mutace je běžný mutační operátor založený na Gaussově rozdělení, kdy uživatel specifikuje rozsah jednotné náhodné hodnoty, která nahradí hodnotu zvoleného genu. [12]
- Řízená mutace (Directed mutation) je založena na gradientu nebo extrapolaci. Řízená mutace deterministicky najde nový bod v populaci pomocí informací aplikovaných v předchozích generacích. Řízená mutace založená na hybnosti je standardní Gaussova mutace, která se používá k urychlení trénování gradientu sestupu neuronových sítí. Stávající hybnost funguje jako mutátor pro každou složku jedince. [12]
- Gaussovská mutace je mutační metoda, která zahrnuje přidání náhodné hodnoty získané z Gaussova rozdělení ke každému genu v chromozomu. Velikost mutace může být řízena parametrem rychlosti mutace.

2 Neuronové sítě

Neuronové sítě (NS), jejichž vznik sahá do roku 1958, jsou nejstarší částí umělé inteligence. Jsou podskupinou strojového učení (machine learning) a v jejich středu se nachází algoritmy hlubokého učení (deep learning). Neuronové sítě se používají v modelování, řízení, identifikaci, predikci. [4] NS stály u zrodu rozpoznání řeči (speech recognition) a v mnoha případech využití robotiky v medicíně. [13]

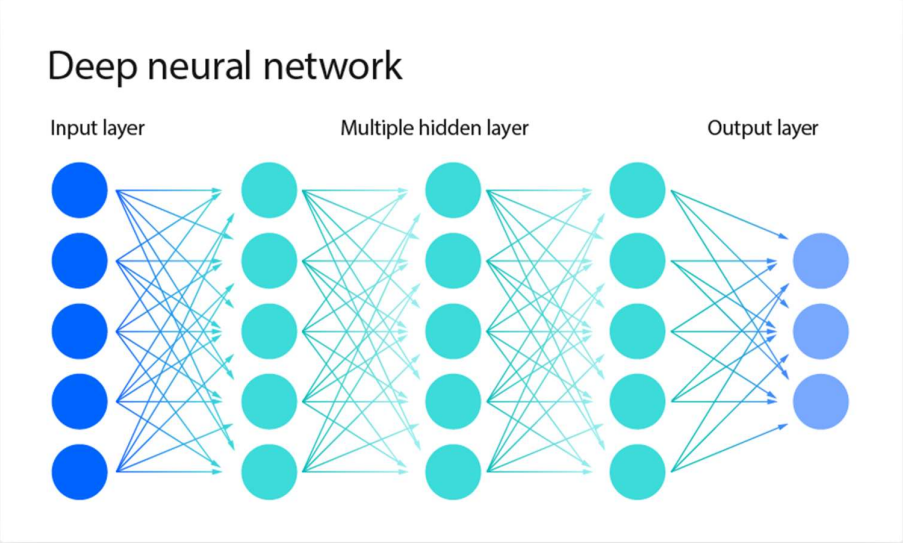
Neuronové sítě se snaží napodobit způsob, jakým se učí a zpracovává informace lidský mozek. Základními stavebními kameny neuronových sítí jsou umělé neurony, které jsou inspirovány biologickými neurony. Neuronové sítě dokážou rozpoznávat složité vzorce a řešit úkoly v oblasti umělé inteligence. Jsou vhodné pro data, která nejsou lineárně oddělitelná a mají velký počet vstupů, jako například obrázky. NS jsou schopné pracovat s různým počtem vstupů a provádět paralelní výpočty, dále také umožňují shlukovat data podle jejich podobnosti nebo je klasifikovat, pokud mají k dispozici trénovací množinu.

Algoritmus neuronových sítí se učí na základě mnoha příkladů (trénovací data), kdy je znám správný výstup pro každý příklad. Tento typ učení se nazývá učení s učitelem. Při tomto učení se měří přesnost modelu pomocí chybové funkce (cost function). Cílem je najít takové parametry modelu, které minimalizují tuto funkci, což znamená, že model správně klasifikuje vstupy. Při každém tréninkovém příkladu se parametry modelu upravují tak, aby se postupně blížily k optimálním hodnotám. Po dostatečném počtu iterací může neuronová síť zpracovávat nové vstupy a poskytovat přesné výstupy. Kvalita modelu závisí na velikosti a rozmanitosti trénovací množiny. Při učení bez učitele neexistuje žádné vnější kritérium pro hodnocení správnosti výsledku. Algoritmus se musí spoléhat pouze na informace z trénovacích dat. Tento typ učení se používá pro samoorganizaci dat. [14]

Základem neuronových sítí jsou vrstvy (layers) a uzly (notes). Ty se dělí na vstupní vrstvu (input layer), jednu nebo více skrytých vrstev (hidden layer) a výstupní vrstvu (output layer). Schéma neuronové sítě je na obrázku 13. Základní neuronové sítě obsahují dvě nebo tři vrstvy. Pokud NS obsahuje vrstev více, mluvíme o hlubokém učení.

Každý neuron je spojen se synaptickou vahou a prahovou hodnotou. Synaptické váhy slouží k určení důležitosti jednotlivých vstupů – čím vyšší váha, tím větší vliv má příslušný vstup na výstup neuronu v rámci dané úlohy, kterou neuronová síť zpracovává. Tyto váhy jsou přiděleny po definování vstupní vrstvy. Vstupy jsou následně váhovány a sečteny. Celkový součet je poté podroben přenosové funkci. Pokud překročí danou prahovou hodnotu, aktivuje se uzel a jeho výstup je předán další vrstvě neuronové sítě. Práh určuje minimální hodnotu součtu

vstupů, která je nutná k aktivaci uzlu. Výstup z jednoho neuronu tak slouží jako vstup pro další neuron v síti.



Obrázek 13 Schéma hluboké neuronové sítě [15]

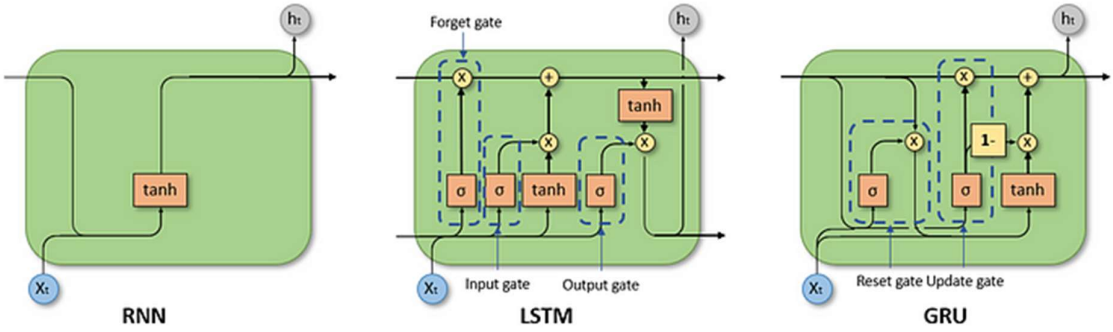
2.1 Typy neuronových sítí

Neuronové sítě lze rozdělit do různých typů, které se používají pro různé účely. Perceptron je nejjednodušší a nejstarší neuronová síť, který se skládá pouze z jednoho neuronu. Vytvořil ho Frank Rosenblatt v roce 1958.

Mezi typy neuronových sítí se řádí:

- Dopředná síť (Perceptron)
- Konvoluční neuronové sítě (CNN)
- Rekurentní neuronové sítě (RNN) jsou identifikovány jejich zpětnovazebními smyčkami. Tyto algoritmy učení se primárně využívají při používání dat časových řad k předpovědím budoucích výsledků, jako jsou předpovědi akciového trhu nebo předpovědi prodeje. [15]
    - Typem RNN s vylepšenou architekturou jsou Long short-term memory (LSTM). Zatímco standardní RNN mají problémy s učením se dlouhodobých závislostí v sekvenčních datech, LSTM sítě byly navrženy právě pro překonání této limitace.

- Dále sem lze zařadit i Gated Recurrent Unit (GRU), která na rozdíl od LSTM používá méně bran a nemá separátní vnitřní paměť. Pro srovnání klasických RNN, LSTM a GRU viz Obrázek 14

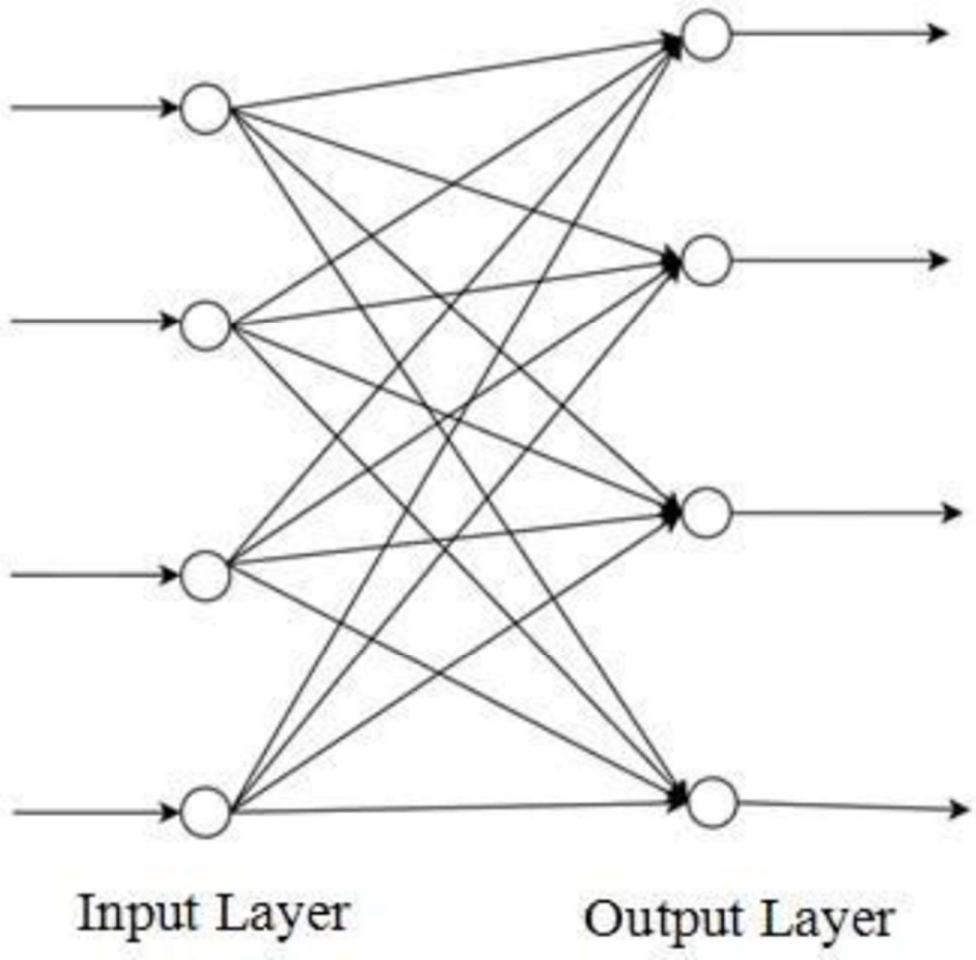


Obrázek 14 Srovnání RNN, LSTM a GRU

2.1.1 Dopředné neuronové sítě

Dopředné neuronové sítě netvoří cykly a jsou oproti ostatním NS jednoduché. Využívají nelineární přenosovou funkci a ke svému trénování využívají algoritmus zpětného šíření chyby (backpropagation). Tento algoritmus umožňuje vypočítat a přiřadit chybu spojenou s každým neuronem, což následně využívá k vhodnému přizpůsobení parametrů modelu.

Sítě jsou nazývány dopřednými, protože informace putuje pouze dopředným směrem sítí. Tyto sítě lze dělit na jednovrstvé a vícevrstvé dopředné NS. Do počtu se nezahrnuje vstupní vrstva (input layer), a tudíž jednovrstvá síť vypadá jako na obrázku 15. Vícevrstvá NS (Obrázek 13 a Obrázek 16) obsahuje jedno nebo více skrytých vrstev (hidden layer). [13]



Obrázek 15 Jednovrstvá dopředná neuronová síť

2.1.2 CNN

Konvoluční neuronové sítě jsou podobné dopředným sítím, ale obvykle se používají pro rozpoznávání obrazu, rozpoznávání vzorů nebo počítačové vidění. Tyto sítě využívají principy lineární algebry, zejména násobení matic, k identifikaci vzorů v obraze. Na rozdíl od dopředných sítí mají aspoň jednu smyčku zpětné vazby (feedback loop). Oproti dopředným sítím, které měly vstupní, skryté a výstupní vrstvy, mají CNN ještě zpožďovací vrstvu (Delay layer). Velkou výhodou CNN je to, že si pamatují minulé výstupy, avšak největší nevýhodou je náročnost jejich trénování. [13]

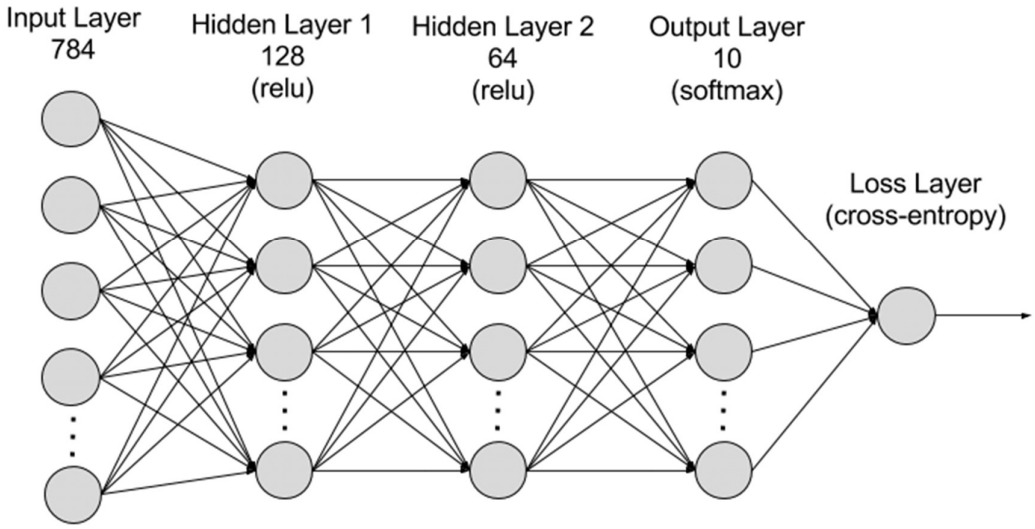
2.1.3 RNN

RNN využívají informace z předchozích vstupů pro ovlivnění současných výstupů. Tato "paměť" umožňuje RNN uchovávat informace o předchozím stavu a použít je pro lepší predikci následujícího výstupu.

RNN jsou široce využívány v aplikacích, jako je strojový překlad, zpracování přirozeného jazyka, rozpoznávání řeči a popisování obrázků. Díky své schopnosti zpracovávat sekvence mohou RNN efektivně modelovat jazykové struktury a předpovídat následující slovo nebo frázi v textu. Například v případě idiomu "cítit se pod psa", který se používá, když je někdo nemocný, musí RNN vzít v úvahu pořadí každého slova v idiomu, aby mohlo správně predikovat následující slovo v sekvenci.

Jedním z klíčových aspektů RNN je sdílení parametrů napříč vrstvami sítě. Zatímco dopředné sítě mají různé váhy pro každý uzel, RNN sdílí stejný váhový parametr uvnitř každé vrstvy sítě. Tyto váhy se přesto upravují pomocí procesů zpětné propagace a gradientního sestupu, což umožňuje posílení učení. RNN využívají algoritmus zpětné propagace v čase (BPTT) pro určení gradientů, což je mírně odlišné od tradiční zpětné propagace, protože je specifické pro sekvenci dat. Principy BPTT jsou stejné jako u tradiční zpětné propagace, kde model se učí vypočítáním chyb od výstupní vrstvy ke vstupní vrstvě. Tyto výpočty nám umožňují přizpůsobit parametry modelu tak, aby byly správné.

RNN se však mohou setkat s problémy, jako jsou explodující a mizející gradienty. Tyto problémy jsou definovány velikostí gradientu, který je sklonem ztrátové funkce podél křivky chyby. Když je gradient příliš malý, stává se stále menším, aktualizuje váhové parametry, dokud se nestanou nepatrnými, tedy nulovými. Aby se předešlo těmto problémům, byly vyvinuty různé varianty RNN, jako jsou LSTM a GRU, které jsou navrženy tak, aby lépe zachytávaly dlouhodobé závislosti a minimalizovaly problémy s gradienty. Tyto pokročilé typy RNN umožňují efektivnější a stabilnější učení pro složité sekvence dat. [16]



Obrázek 16 Schéma RNN [16]

2.2 Neuronové sítě a evoluční techniky

Způsob učení NS je klíčový pro jejich úspěšné použití a zaručení spolehlivosti. Jedním z nejznámějších a nejpoužívanějších algoritmů je zpětné šíření (backpropagation – BP), které postupně upravuje váhy sítě podle rozdílu mezi požadovaným a skutečným výstupem. Tento algoritmus však není bez nedostatků, například může uváznout v lokálním minimu a neoptimalizovat síť dostatečně. Proto se hledají alternativní metody učení, které by překonaly tyto problémy. Mezi ně patří i genetické algoritmy (samostatně, nebo jako doplnění BP) nebo třeba diferenciální evoluce (DE). [17] DE se snaží řešit problém s kódováním, na který naráží GA (diskrétní přístup ke spojitým problémům).

T. Panská shrnuje typy učení následovně: „*Nicméně jak genetický algoritmus, tak diferenciální evoluce se ukazují být lepší možnost oproti klasickému algoritmu zpětného šíření chyby, který se běžně používá k učení neuronových sítí*.“ [2, s. 35] Dodává také však, že záleží na daném souboru dat a např. GA nemusí být vždy vhodná volba, hlavně bez paralelismu, kdy vycházel horší výsledek v podobě větší chyby a délky běhu GA. [2]

V následující kapitole budou rozebrány jednotlivé knihovny obsahující NS a GA. Knihovny s neuronovými sítěmi jsou představeny později (viz kapitola 3.7).

3 Knihovny pro C++ zabývající se neuronovými sítěmi a genetickými algoritmy

V této části jsou uvedeny některé knihovny implementující evoluční algoritmy a neuronové sítě v různých jazycích se zaměřením na knihovny pro C++. Pro porovnání knihoven pro ET budeme používat tři hlavní kritéria: způsob kódování chromozomů, typ fitness funkce a kritérium ukončení. Tato kritéria ovlivňují výkon a efektivitu knihoven pro ET a jsou důležitá pro jejich správnou volbu a použití. Mimo knihovny zmíněných zde existují ještě knihovny pro jazyk Python (DEAP, Scikit-Learn, GeneAI nebo PYGAD) a Javu (JGAP, ECJ a Opt4j). [2] [1]

Pro jazyk C++ jsou nativní následovné knihovny:

- Z řad knihoven implementující genetické algoritmy
  - OpenGA
  - Galgo
  - Galib
  - EALib*
  - Paradiseo
- Z řad knihoven implementující neuronové sítě
  - Genann
  - Fann
  - OpenNN
  - EALib*

*EALib obsahuje jak komponenty evolučních algoritmů, tak umělé neuronové sítě. [18]

3.1 OpenGA

Knihovna OpenGA se prezentuje jako knihovna, která poskytuje volnost uživatelům při navrhování jejich datového modelu vlastního řešení bez omezení. Knihovna je schopna optimalizace v každém z jednotlivých režimů: cílových, vícecílových a interaktivních. Svojí volností však klade požadavky na programátora definovat genetické operátory mutace a křížení. [19]

V knihovně OpenGA je fitness hodnota odvozena od výsledku hodnotící funkce, která provádí složitý proces ohodnocování řešení a shromažďuje všechny potřebné údaje. Tyto údaje se nazývají střední náklady a mohou obsahovat více informací, než je nutné pro výpočet fitness. Některé informace mohou být výpočetně nákladné, takže není efektivní je opakovat, pokud už byly zjištěny hodnotící funkcí. Tato funkce také zajišťuje dodržování omezení a může nahradit

jakéhokoli jedince, který je poruší. Knihovna OpenGA dává programátorovi možnost umístit kontrolu omezení před nebo během ohodnocování. [2]

OpenGA pracuje se zastavovací funkcí v podobě:

- V režimu jednotlivých cílů je vhodnost nejlepšího řešení pozastavena na upravený počet generací.
- V režimu jednotlivých cílů se průměrná zdatnost celé populace zablokuje na upravený počet generací.
- Byl dosažen maximální počet generací.
- Uživatel požádal o zastavení. [19]

Knihovna nevyžaduje žádné další externí knihovny.
T. Panská ve své bakalářské práci [2] naráží na problém, že knihovna nemá detailní manuál a pro zjištění detailů je nutné prozkoumat zdrojový kód knihovny. Dále zmiňuje: „*Selekce vybírá jedince s menší hodnotou fitness funkce, GA tedy minimalizuje problém a nelze to změnit žádným nastavením*“. [2, s. 27] Zastavovací pravidlo je sice pevně zabudované [2], ale uživatel může zastavit algoritmus nastavením *user_request_stop* na hodnotu *true*. [19]

3.2 GALGO

Galgo je knihovna pro C++, která umožňuje řešit problémy s omezeními optimalizací cílové funkce podle zadaných omezujících podmínek. Knihovna Galgo nabízí několik běžných metod pro výběr, křížení a mutaci, ale také podporuje jednoduché začlenění jiných metod. Programátor má možnost využít již existující metody nebo přidat své vlastní. [2]

Evoluce je v knihovně Galgo implementovaná pomocí:

- Metody selekce
    - proporcionální výběr ruletového kola (RWS)
    - stochastické univerzální vzorkování (SUS)
    - klasický lineární výběr na základě pořadí (RNK)
    - lineární výběr založený na pořadí se selektivním tlakem (RSP)
    - výběr turnaje (TNT)
    - transformace výběru hodnocení (TRS)
- Metody křížení
    - jednobodové křížení (P1XO)
    - dvoubodové křížení (P2XO)
    - jednotné (uniform) křížení (UXO)
- Mutační metody

- mezní mutace (BDM)
- jednobodová mutace (SPM)
- uniformní mutace (UNM)

Jednobodové a dvoubodové křížení spočívá v náhodném výběru dvou rodičů a jednoho nebo dvou bodů křížení, podle kterých se prohazují geny mezi rodiči. Uniformní křížení spočívá v náhodném výběru dvou rodičů a následném výběru genu z jednoho z nich pro každou pozici v chromozomu potomka. Jednobodová mutace změní náhodně vybraný bit na opačnou hodnotu. Uniformní mutace změní náhodně vybraný gen na jinou hodnotu. Mezní mutace změní náhodně vybraný gen na jeho horní nebo dolní limit. [2]

3.3 GAlib

GAlib je knihovna, která umožňuje řešit optimalizační problémy pomocí genetických algoritmů. Nabízí různé varianty selekce, křížení a mutace, které lze upravit podle potřeb nebo nahradit vlastními implementacemi. Knihovna podporuje čtyři typy chromozomů: bitové pole, pole reálných čísel, seznam a strom. Uživatel musí definovat funkci pro výpočet fitness každého jedince. Mezi dostupnými metodami selekce jsou rank selekce, ruletové kolo, stochastické univerzální vzorkování, turnajová selekce, deterministické vzorkování (DS) a stochastické zbytkové vzorkování (SRS). DS spočívá v tom, že se nejprve určí očekávaný počet jedinců každého typu a dočasná populace se naplní těmi s nejvyššími hodnotami. Pokud zůstanou nějaká volná místa, vyplní se podle desetinných částí očekávaných počtů v sestupném pořadí. Nakonec se z dočasné populace náhodně vybírají jedinci. SRS funguje podobně, ale místo desetinných částí používá zlomkové hodnoty ke zvýšení pravděpodobnosti výběru jedinců s nižšími očekávanými počty. [2]

K dispozici jsou vestavěné metody pro křížení jako jednobodové, dvoubodové, uniformní, sudé – liché, cyklické, uspořádané (ordered), křížení s částečnou shodou (partial match) a jednobodové křížení stromu. Tyto metody jsou implementovány pro jednobodové, dvoubodové a uniformní křížení. Sudé – liché křížení střídá geny z obou rodičů tak, že sudý gen je z prvního rodiče a lichý gen je z druhého rodiče nebo naopak (nultý gen se považuje za sudý). Křížení s částečnou shodou kopíruje část genů z jednoho rodiče do druhého rodiče tak, aby se zachovalo jejich původní pořadí. Pokud se v novém chromozomu objeví duplicitní geny, jsou nahrazeny genem z prvního rodiče, který ještě v chromozomu není. Pořadové křížení (order crossover) přenáší sekvenci genů z jednoho rodiče do potomka a doplní zbytek genů z druhého rodiče tak, že odstraní geny, které už potomek má, a vloží je do volných pozic. Cyklické křížení naplňuje každou pozici potomka prvkem z jednoho z rodičů tak, že vytvoří cyklus stejných pozic. Křížení s částečnou shodou (partial match), pořadové a cyklické křížení

funguje jen pro chromozomy reprezentované polem a seznamem. Jednobodové křížení u stromu zamění uzel s podstromem z jednoho rodiče za uzel s podstromem z druhého rodiče. Tato metoda je použitelná jen pro jedince definované jako stromová struktura. [2]

GAlib nabízí i několik dalších typů mutací a křížení (názvy jsou zachovány v originálu), např.:

- Křížení
    - Tree Single Point Crossover,
    - List Single Point Crossover,
    - Fixed-Length Array Single Point Crossover,
    - List Order-Based Crossover,
    - Variable-Length Array Single Point Crossover,
    - Array Uniform Crossover
- Mutace:
    - Sub-Tree Swap Mutation,
    - Tree Node Swap Mutation,
    - Sub-Tree Destructive Mutation,
    - List Generative Mutation,
    - List Destructive Mutation,
    - List Swap Node Mutation,
    - List Swap Sequence Mutation [20]

3.4 EALib

EALib je série knihoven v jazyce C++ obsahující řadu generických komponent, které jsou užitečné pro vytváření evolučních algoritmů. EALib se zaměřuje na flexibilitu v době kompilace, na rozdíl od flexibility v době běhu. EALib se tak ideálně hodí pro vysoce výkonné a/nebo rozsáhlé evoluční algoritmy. Navíc mnoho komponent v EALib (např. selekční strategie) je generických a lze je snadno znovu použít jako stavební bloky pro propracovanější evoluční algoritmy. Obsahuje tři části:

- libea: Knihovna evolučních algoritmů. Obsahuje komponenty související s různými druhy evolučních algoritmů, stejně jako další funkce související s podporou, jako je kontrolní bod, parametry, rozhraní příkazového řádku a podobně.
- libmkv: síťová knihovna Markov. Obsahuje implementaci sítí Markov a komponenty kompatibilní s EALib, které umožňují jejich vývoj.
- libnn: Knihovna umělé neuronové sítě. Obsahuje implementaci umělých neuronových sítí: dopředná vazba, opakující se a opakující se kontinuálně (feedforward, recurrent, and continuous-time recurrent). Tato část však není dokončená. [18]

3.5 Paradiseo

Paradiseo je framework (aplikační rámec) navržený pro paralelní výpočty, který umožňuje efektivní implementaci genetických algoritmů na více procesorech. Paradiseo je robustní a flexibilní platforma pro evoluční výpočty, která je vhodná pro širokou škálu optimalizačních úloh. Platforma se neustále vyvíjí a nabízí inovativní funkce, jako je automatické navrhování metaheuristik. Současná podoba projektu Paradiseo vznikla roku 2012 sloučením frameworku „Evolving objects (EO, později EOlib) z roku 1999 a původního modulu Paradiseo z roku 2003. [21] Vzhledem k povaze frameworku se Paradiseo nebudeme více zabývat.

3.6 Shrnutí a porovnání knihoven EVT

V této kapitole jsme se zabývali převážně knihovnami obsahující genetické algoritmy. Jejich srovnání je v tabulce 1, která je převzata z práce R. Komendy [1], který porovnává jiné knihovny stejnými metrikami.

Tabulka 1 Srovnání knihoven a jejich funkcí, předloha [1]

Funkce	OpenGA	Galgo	Galib	EALib
Genetická reprezentace	Binární, celočíselná, reálná, řetězcová	Binární, reálná	Binární, celočíselná, reálná	Binární, celočíselná, reálná, komplexní
Registrace operátorů	Ano	Ano	Ano	Ano
Hodnotící funkce	Ne	Ano	Ano	Ne
Předdefinovaná zastavovací pravidla	Ano	Ne	Ne	Ano
Evoluční cyklus	Ano	Ano	Ano	Ano
Modulární architektura	Ano	Ano	Ano	Ano
Vizualizace vývoje	Ano	Ano	Ano	Ano
Multiobjektivní optimalizace	Ano	Ano	Ne	Ano
Evoluce symbolického vyjádření	Ne	Ano	Ano	Ano

Elitismus	Ano	Ano	Ano	Ano
Paralelní výpočet	Ano	Ano	Ne	Ano
Evoluce neuronových sítí	Ne	Ne	Ne	Ano

Genetická reprezentace: Knihovna poskytuje nástroje pro reprezentaci jedinců a řeší způsob zakódování jedinců. To zahrnuje generaci počáteční populace, definici struktury jedinců (například řetězce, stromové struktury) a operace spojené s genetickou operací, jako je křížení, mutace a ohodnocení.

Registrace operátorů: Uživatel má možnost registrovat vlastní genetické operátory, jako jsou operátory selekce, křížení a mutace. Tato funkce umožňuje definovat a používat vlastní operátory, které lépe odpovídají specifickým potřebám řešené úlohy.

Hodnotící (fitness, kriteriální) funkce: Knihovna umožňuje uživatelům definovat a používat vlastní hodnotící funkci.

Předdefinovaná zastavovací pravidla: Knihovna obsahuje zastavovací pravidla mimo maximální počet generací. Příkladem může být, že se fitness funkce nemění po několik generací.

Evoluční cyklus: Knihovna umožňuje provedení evolučního cyklu, který zahrnuje procesy selekce, křížení, mutace, vyhodnocení kvality a nahrazení části původní populace novými jedinci. Tato funkce umožňuje postupné zlepšování výsledků genetického algoritmu.

Modulární architektura: Knihovna je navržena tak, aby byla snadno upravitelná a rozšířitelná. To umožňuje uživatelům upravit knihovnu podle specifických požadavků jejich úlohy.

Vizualizace vývoje: Knihovna disponuje nástroji pro vizualizaci vývoje genetického algoritmu, což zahrnuje grafy zobrazující změny kvality jedinců v průběhu generací. Vizualizace pomáhá lépe pochopit výsledky algoritmu a umožňuje ladit a optimalizovat jeho parametry.

Multiobjektivní optimalizace: Knihovna dokáže řešit úlohy s více než jednou cílovou funkcí a optimalizuje tedy více než jedno kritérium.

Evoluce symbolického vyjádření: Knihovna umožňuje práci se symbolickými výrazy, jako jsou logické operace nebo matematické výrazy.

Elitismus: Knihovna podporuje elitismus jako strategii, která zaručuje, že nejlepší jedinec postoupí do další generace bez křížení či mutace (viz kapitola 1.2.1).

Paralelní výpočet: Knihovna podporuje paralelní výpočet, což znamená, že může využívat více výpočetních prostředků pro současné vypočítávání a zpracování dat, čímž urychlí proces.

Evoluce neuronových sítí: Knihovna poskytuje nástroje pro evoluci neuronových sítí, včetně optimalizace architektury sítí, váhových parametrů a struktury sítí (viz kapitola 2).

3.7 Knihovny pro umělé NS

3.7.1 Genann

Genann je knihovna v jazyce C pro dopředné neuronové sítě. Genann klade důraz na jednoduchost, rychlost a spolehlivost. Nabízí proto pouze základní funkce, ale je otevřená pro rozšíření a podporuje i jiné trénovací metody než klasické učení s učitelem, například genetické algoritmy.

Genann vytváří umělou neuronovou síť pomocí funkce, která specifikuje počet vstupů, výstupů, skrytých vrstev a neuronů v každé skryté vrstvě. Síť se potom učí pomocí algoritmu zpětného šíření chyby. Váhy neuronové sítě jsou uloženy jako souvislý blok paměti, což umožňuje snadné použití numerických optimalizačních algoritmů s přímým vyhledáváním.

U stromu se křížení provádí tak, že se vybere uzel s celým jeho podstromem od jednoho rodiče a vymění se za uzel od druhého rodiče (také s celým jeho podstromem). Tento typ křížení funguje pouze pro jedince definované jako stromová struktura.

3.7.2 FANN

Knihovna FANN (Fast Artificial Neural Network) je bezplatná otevřená (open-source) knihovna neuronových sítí, která implementuje vícevrstvé umělé neuronové sítě v jazyce C s podporou plně propojených i řídce propojených sítí. Je podporováno provádění napříč platformami v pevné i pohyblivé řádové čárce. Obsahuje rámec pro snadnou manipulaci s tréninkovými datovými sadami. Je snadno použitelná, všestranná, dobře zdokumentovaná a rychlá. K dispozici jsou vazby na více než 15 programovacích jazyků. Knihovnu doprovází snadno čitelný úvodní článek a referenční příručka s příklady a doporučeními, jak knihovnu používat. Pro knihovnu je také k dispozici několik grafických uživatelských rozhraní. FANN knihovna má také další funkce jako např. trénování pomocí zpětného šíření (backpropagation training) a lehké ukládání a načítání neuronových sítí. [22]

3.7.3 Dlib

Dlib je moderní sada nástrojů C++ obsahující algoritmy strojového učení a nástroje pro vytváření komplexního softwaru v C++ pro řešení problémů reálného světa. Používá se v průmyslu i akademické sféře v celé řadě oblastí včetně robotiky, vestavěných zařízení, mobilních telefonů a velkých vysoce výkonných výpočetních prostředí. V dlib se hluboká neuronová síť skládá ze tří částí: vstupní vrstva, vícero výpočetních vrstev (computational layers) a volitelně tzv. loss layer, které vypočte ztrátu sítě na daném úkolu, tj. jak dobře síť funguje. [23]

3.7.4 Knihovny nativní pro jiné jazyky

Knihovny Keras, TensorFlow a PyTorch budou uvedeny spolu, protože nejsou nativní pro C++, nýbrž pro Python, ale lze je použít i pro C++.

TensorFlow je knihovna vyvinutá týmem Google Brain pro strojové učení a umělou inteligenci. TensorFlow umožňuje vývojářům vytvářet grafy toku dat, což jsou struktury popisující pohyb dat v grafu nebo sérii uzlů. Každý uzel v grafu představuje matematickou operaci a spojení mezi uzly je realizováno pomocí vícedimenzionálních datových polí, zvaných tenzory. Všechny tenzory jsou neměnné, což znamená, že jejich obsah nelze upravovat, pouze vytvářet nové instance. Pro reprezentaci hodnot, které se mění v průběhu algoritmu (například synaptické váhy), lze použít proměnné typu TensorFlow. [2]

Vysokoúrovňové rozhraní pro knihovnu TensorFlow poskytuje knihovna **Keras**. Keras usnadňuje používání TensorFlow tím, že je uživatelsky přívětivější a snadno rozšiřitelný. Nicméně TensorFlow lze použít i samostatně, zejména pokud je potřeba rychlého učení sítě a vysokého výkonu. [2]

PyTorch je open-source framework pro hluboké učení a strojové učení vyvinutý společností Facebook (dnes Meta) a nyní součástí zastřešující Linux Foundation. Umožňuje vývojářům snadno konstruovat a trénovat neuronové sítě s flexibilní architekturou a širokou škálou funkcí. PyTorch je mocný nástroj pro tvorbu a trénování neuronových sítí. Jeho propojení s C++ umožňuje vývojářům těžit z flexibility PyTorch a zároveň využívat specifické vlastnosti C++. PyTorch je vhodný pro širokou škálu projektů v oblasti hlubokého učení, od prototypování až po produkční nasazení. [24]

4 Knihovna GAlib

V následující kapitole bude přiblížena knihovna Galib, ze které budou vycházet změny pro knihovnu Galgo. Bude vysvětleno zaměření práce na druhou zmíněnou knihovnu a důvod, proč rozsáhlé rozšiřování knihovny Galib není vhodné.

Knihovna Galib vznikla na konci minulého (20.) století. Na rozdíl od knihovny Galgo není tvořena pouze hlavičkovými soubory *.hpp*, nýbrž (jelikož se jedná o starší knihovnu) obsahuje hlavičkové soubory *.h* a zdrojové kódy *.C (*namísto *.hpp* a *.cpp)*. Důsledkem toho je více souborů v repositáři.

Zároveň je důležité umístění různých metod mutace a křížení. Zatímco u knihovny Galgo jsou tyto metody v rámci jednoho souboru (*Evolution.hpp* viz kapitola 5), zde jsou tyto metody vázány vždy s typem genomu. Knihovna obsahuje celkem šest typů genomu a ke každému jsou přiřazeny jiné metody. Ucelený přehled, které genomy obsahují, jaké rekombinační operátory, se nachází v tabulce 2 (jsou zachovány původní názvy z knihovny). [20]

Tabulka 2 Přehled funkcí knihovny GAlib

Genom	Křížení	Mutace
1D array genome	UniformCrossover, EvenOddCrossover, OnePointCrossover, TwoPointCrossover, PartialMatchCrossover, OrderCrossover, CycleCrossover	SwapMutator
1DBinaryString Genome	UniformCrossover, EvenOddCrossover, OnePointCrossover, TwoPointCrossover	FlipMutator
2D array genome	UniformCrossover, EvenOddCrossover, OnePointCrossover	SwapMutator

2DBinaryStringGenome	UniformCrossover, EvenOddCrossover, OnePointCrossover,	FlipMutator
3D array genome	UniformCrossover, EvenOddCrossover, OnePointCrossover	SwapMutator
3DBinaryStringGenome	UniformCrossover, EvenOddCrossover, OnePointCrossover,	FlipMutator

Jednotlivé typy genomů v tabulce 2 se liší hlavně počtem dimenzí (1D, 2D nebo 3D) a následně, zda se jedná o jednorozměrná pole, nebo jestli jsou kódovány binárně. Genom typu pole je vhodný pro jednoduché problémy s jednoduchými jedinci, kde stačí jednorozměrné uspořádání genetických informací. Binární genom se často používá pro binární kódování jedinců, což je užitečné pro problémy, kde jsou jednotlivé části jedince binární, nebo kde je vhodné použít binární reprezentaci pro optimalizaci.

4.1 Použití knihovny

Knihovna Galib obsahuje několik příkladů. Na rozdíl od knihovny Galgo se atributy jako velikost populace (*popsize*) zadává ve funkci *main* hlavního programu. V prvním příkladu z knihovny (*ex1.C*) je postup tvoření genetického algoritmu pomocí této knihovny následovný:

- Definování cílové (objective) funkce
- Definování hodnot *seed*, nebo vytvoření náhodných
- Definování atributů *width, height, popsize, ngen, pmut* a *pcross* pro genetický algoritmus
- Vytvoření algoritmu se správnými atributy
- Spuštění běhu genetického algoritmu a přiřazení dalších atributů
- Tisk výsledků na obrazovku nebo do souboru

Pro použití knihovny je potřeba kompilace celé knihovny nebo velké části a samotného souboru, který knihovnu používá. Protože je knihovna přes 25 let stará, není tak lehké ji kompilovat a použít jako nové knihovny. Tato knihovna bude sloužit primárně jako inspirace pro rozšíření knihovny Galgo.

5 Knihovna Galgo

Tato kapitola je zaměřena na knihovnu Galgo a možné rozšíření této knihovny. Možné implementace vychází z předchozího rozboru knihovny v kapitole 3.2. Rozšíření knihovny umožní uživatelům využít více metod křížení a mutace.

5.1 Seznámení s knihovnou

Základním prvkem knihovny je třída *GeneticAlgorithm*, který řídí celý algoritmus. Je v něm definována forma a parametry selekce, křížení a mutace, obsahuje velikost a počet populací. Následně jsou definovány ještě třídy *Population* a *Chromosome*, které podle názvu obsahují populaci a chromozomy a vše, co se k nim váže (např. fitness). Křížení a mutace jsou definovány v souboru *Evolution.hpp*, kterému se věnuje kapitola 5.4.

Pro přehlednost budou kódy zobrazovány fondem `CascadiaMono` (písmo TrueType s neměnnou mezerou), který je výchozí pro C++ kódy ve VisualStudio, ve kterém byl kód upravován a doplňován. Proměnné, metody, funkce apod. budou psány písmem *italic*.

K využití knihovny je potřebné zadat následující vstupy:

- Funkce pro optimalizaci (v příkladu z dokumentace je minimalizace f(x,y) = (1 - x)^2 + 100 * (y - x^2)^2 nastavena pomocí třídy *MyObjective*)
- Velikost populace (int)
- Počet generací (int)
- Nastavení, jestli se výstupy budou zobrazovat (True/False)
- Parametry (pro horní a spodní mez)

Součástí dokumentace a knihovny samotné je i příklad použití s výsledky. [25] Autor knihovny O. Mallet uvádí možné výsledky jako v tabulce 3 s omezeními: C1(x) = -0,00021, C2(x) = -0,00338.

Tabulka 3 Výsledky knihovny příkladu Galgo od autora knihovny

Generation	X1	X2	F(x)
0	0,22918	12,61854	-15791,07606
10	0,81093	12,39359	-13773,38448
20	0,81209	12,38605	-13751,28199
30	0,81201	12,33666	-13635,97220
40	0,81212	12,31742	-13590,66665

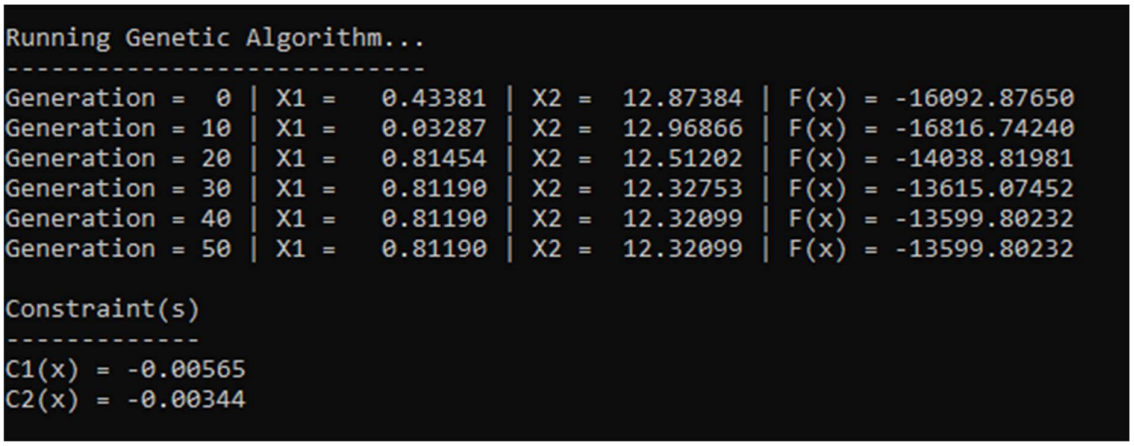
50	0,81224	12,31583	-13586,50453

Kód příkladu *Example.cpp je následovný* [25]*:*

```
//============================================================================
===============
//                     Copyright (C) 2017 Olivier Mallet - All Rights Reserved
//============================================================================
===============

#include "Galgo.hpp"

// objective class example
template <typename T>
class MyObjective
{
public:
   // objective function example : Rosenbrock function
   // minimizing f(x,y) = (1 - x)^2 + 100 * (y - x^2)^2
   static std::vector<T> Objective(const std::vector<T>& x)
   {
      T obj = -(pow(1-x[0],2)+100*pow(x[1]-x[0]*x[0],2));
      return {obj};
   }
   // NB: GALGO maximize by default so we will maximize -f(x,y)
};

// constraints example:
// 1) x * y + x - y + 1.5 <= 0
// 2) 10 - x * y <= 0
template <typename T>
std::vector<T> MyConstraint(const std::vector<T>& x)
{
   return {x[0]*x[1]+x[0]-x[1]+1.5,10-x[0]*x[1]};
}
// NB: a penalty will be applied if one of the constraints is > 0
// using the default adaptation to constraint(s) method

int main()
{
   // initializing parameters lower and upper bounds
   // an initial value can be added inside the initializer list after the upper
bound
   galgo::Parameter<double> par1({0.0,1.0});
   galgo::Parameter<double> par2({0.0,13.0});
   // here both parameter will be encoded using 16 bits the default value inside the
template declaration
   // this value can be modified but has to remain between 1 and 64

   // initiliazing genetic algorithm
   galgo::GeneticAlgorithm<double>
ga(MyObjective<double>::Objective,100,50,true,par1,par2);

   // setting constraints
   ga.Constraint = MyConstraint;

   // running genetic algorithm
   ga.run();
}
```

Při kompilaci se používá flag -std=c++11.

Genetické algoritmy nám poskytují výsledky, které jsou sice podobné, ale nikdy zcela identické. Tento jev je patrný při opakovaném spuštění, jak ukazuje Obrázek 17. Zde lze vidět, že i když se čísla mohou lišit, v pozdějších generacích algoritmu dochází k větší konzistenci výsledků. Srovnáním tabulky 3 s obrázkem 17 je možné vidět rozdíly několika procent (stovky) u prvních generací a pouze promile (desítky) u 50. generace.



Obrázek 17 Další výsledek příkladu genetického algoritmu, zdroj: vlastní

5.2 Zastavovací pravidlo

První změna bude v implementaci předdefinovaného zastavovacího pravidla. Knihovna Galgo obsahuje dvě formy zastavovacího pravidla:

- Maximální generace: při inicializaci genetického algoritmu určí uživatel počet generací genetického algoritmu *nbgen*. Algoritmus probíhá ve *for* cyklu *nbgen*-krát.
- Kontrola konvergence: uživatel může nastavit toleranci *tolerance* na nenulovou hodnotu. Algoritmus pak porovná aktuální výsledek s předchozím nejlepším výsledkem ‚*prevBestResult*', a pokud je rozdíl mezi nimi menší než hodnota tolerance, tak se cyklus ukončí pomocí *break*. [25]

Tato zastavovací pravidla budou změněna a doplněna následovně:

- Kontrola konvergence nahrazena stabilizací nejlepšího řešení: Algoritmus může být zastaven, pokud se nejlepší nalezené řešení po určitém počtu generací nezmění. Toto pravidlo rozšiřuje výše uvedenou kontrolu konvergence, kde se eliminuje náhodné vybrání stejného výsledku, přestože by v dalších generacích byl lepší.
- Doplnění kontroly kvality: Pokud je dosažena požadovaná hodnota fitness, algoritmus bude ukončen. K této podmínce bude přistupováno dvěma způsoby, které budou dále popsány v samostatné podkapitole (5.2.2).

Tyto změny se budou implementovat odděleně.

5.2.1 Stabilizace nejlepšího řešení

Implementace tohoto pravidla spočívá v nahrazení kódu, kterým původní knihovna ověřovala, jestli je nastavená tolerance (*tolerance)* a jestli nejlepší výsledek současný a ten předchozí generace nejsou už tak podobné, aby byly v toleranci nastavenou uživatelem:

```
      // checking convergence
      if (tolerance != 0.0) {
         if (fabs(bestResult - prevBestResult) < fabs(tolerance)) {
            break;
         }
         prevBestResult = bestResult;
      }
   }
```

Nejprve je nutné doplnit atribut *bestResultStreak* třídě *GeneticAlgorithm* (a to včetně konstruktoru) a následně doplnit do kódu podmínku *if*. Výsledná podoba této podmínky bude:

```
      // checking stability of best result
     if (bestResultStreakmax != 0.0) {
          if (fabs(bestResult - prevBestResult) <= fabs(tolerance)) {
              bestResultStreak ++;
                      if (bestResultStreak >= bestResultStreakmax) {
                            std::cout << "Genetic Algorithm reached maximum
streak of same best result. Terminating.\n";
                            break;
                      }

             } else {
                      bestResultStreak = 0; // Reset number of same results
                       }
        prevBestResult = bestResult;
      }
```

Konstruktor se upraví, aby uživatel mohl dávat vstup pro *bestResultStreakmax*. *bestResultStreakmax* se také zařadí mezi veřejné atributy třídy *GeneticAlgorithm* (`int bestResultStreakmax;  // terminal condition limit (inactive if equal to zero)`).

```
// constructor
template <typename T> template <int...N>
GeneticAlgorithm<T>::GeneticAlgorithm(Func<T> objective, int popsize, int
nbgen, bool output, int bestResultStreakmax, const Parameter<T,N>&...args)
{
   this->Objective = objective;
   // getting total number of bits per chromosome
   this->nbbit = sum(N...);
   this->nbgen = nbgen;
   // getting number of parameters in the pack
   this->nbparam = sizeof...(N);
   this->popsize = popsize;
   this->matsize = popsize;
   this->output = output;
   this->bestResultStreakmax = bestResultStreakmax;
   // unpacking parameter pack in tuple
   TUP<T,N...> tp(args...);
   // initializing parameter(s) data
   this->init(tp);
}
```

5.2.2 Kontrola kvality

Fitness funkce je vázána na daný chromozom v dané generaci. Je dána cílovou (*objective)* funkcí, tj. funkcí, kterou chce uživatel optimalizovat. Třída *GeneticAlgorithm* obsahuje ukazatel na tuto funkci. Knihovna obsahuje metodu *getSumFitness* třídy *Population*, která sčítá fitness všech chromozomů v generaci. [25] Nabízejí se dvě formy kontroly kvality, které je možné implementovat:

- Součet fitness všech chromozomů nabude určité hodnoty zadanou uživatelem.
- Fitness nejlepšího řešení nabude určité hodnoty zadanou uživatelem.

Stanovení požadované hodnoty fitness vyžaduje určitý náhled do problematiky funkce a odhad průběhu algoritmu. Z tohoto důvodu bude po implementaci této zastavovací funkce stanovena její výchozí hodnota na 0, a tím vypnuta (podobně jako původně Kontrola konvergence).

Nejprve se provede implementace fitness nejlepšího řešení. K tomu v třídě *GeneticAlgorithm* bude sloužit metoda *getBestFitness(),* která bude vypadat následovně:

```
//get best fitness function
template <typename T>
T GeneticAlgorithm<T>::getBestFitness() const {
    return pop(0)->fitness;
}
```

Následně přidáme atribut této třídy *FitnessBestTarget,* která bude mít stejné parametry a chování jako *tolerance.* Podobně implementujeme i podmínku, která bude navazovat na tu popsanou v kapitole 5.2.1.

```
      //checking fitness of best result
      if (FitnessBestTarget != 0.0) {
          if(getBestFitness() >= FitnessBestTarget){
              genstep = 1;
              print();
              std::cout << "Genetic Algorithm reached the target fitness.
Terminating. \n";
              break;
          }
      }
```

Ověření součtu fitness bude podobně implementováno, avšak pomocí metody *getSumFitness()*, která už je obsažena v původní knihovně v rámci objektu P*opulation*. Kód ověřování zastavovacího kritéria je následující:

```
      if (FitnessTarget != 0.0) {
          if (pop.getSumFitness() >= FitnessTarget) {
              genstep = 1;
              print();
              std::cout << "Genetic Algorithm reached the target fitness.
Terminating. \n";
              break;
          }
      }
  }
```

Ukázka použití je v kapitole 6.

5.3 Selekce minima

Olivier Mallet, autor knihovny Galgo, upozorňuje na skutečnost, že Galgo v základu optimalizuje funkci na maximum. [25] Zároveň uvádí, že řešením tohoto problému je obrátit znaménko funkce. Toto řešení také ukazuje v příkladu použití knihovny *example.cpp* (viz příloha 1). [25]

U turnajové selekce lze také minimum určit tak, že podmínku *if (fit > bestFit)* změníme na *if (fit < bestFit).* Jedná o změnu toho, že „souboj“ vyhraje jedinec s menší hodnotou fitness na rozdíl od toho s větší hodnotou.

5.4 Implementace dalších metod křížení

V odborné literatuře (viz kapitola 1.2.2 a 1.2.3) a ostatních knihovnách se můžeme inspirovat pro další metody křížení a mutací. Nejprve bude implementováno sudé – liché křížení (Even-Odd crossover), kdy jeden potomek zdědí sudé bity od matky a liché od otce a druhý naopak. Vzhledem k povaze knihovny (šablonová knihovna) budou všechna nová křížení a mutace implementovány pomocí šablonových funkcí.

U nových operátorů křížení a mutace je použita funkce *setGeneValue()*, která se používá k nastavení genu na indexu i na odpovídající hodnotu získanou z mapování. Vypadá následovně:

```
//sets Gene Value
template <typename T>
inline void Chromosome<T>::setGeneValue(int k, const T& value)
{
   #ifndef NDEBUG
   if (k < 0 || k >= ptr->nbparam) {
      throw std::invalid_argument("Error: in
galgo::Chromosome<T>::setGeneValue(int), argument cannot be outside interval
[0,nbparam-1], please amend.");
   }
   #endif

   // Encoding the specific value to string
   std::string s = ptr->param[k]->encode(value);
   // Adding or replacing gene in chromosome
   chr.replace(ptr->idx[k], s.size(), s, 0, s.size());
}
```

5.4.1 Sudé – liché křížení

Křížení Even-Odd bude implementováno jako funkce podobná stávajícím funkcím v rámci *Evolution.hpp*. Bude vypadat následovně:

```
// Even-Odd crossover of 2 chromosomes
template <typename T>
void EOX(const galgo::Population<T>& x, galgo::CHR<T>& chr1, galgo::CHR<T>&
chr2)
{
    // choosing randomly 2 chromosomes from mating population
    int idx1 = galgo::uniform<int>(0, x.matsize());
    int idx2 = galgo::uniform<int>(0, x.matsize());

    for (int j = 0; j < chr1->size(); ++j) {
        // choosing even bits from mother for daughter and from father for
son
        if (j % 2 == 0) {
            chr1->addBit(x[idx1]->getBit(j));
            chr2->addBit(x[idx2]->getBit(j));
        }
        else { // choosing odd bits from mother for son and from father for
daughter
            chr1->addBit(x[idx2]->getBit(j));
            chr2->addBit(x[idx1]->getBit(j));
        }
    }
}
```

5.4.2 Křížení s částečnou shodou

Další metodou, která bude do knihovny implementovaná je křížení s částečnou shodou (Partially matched crossover), která je popsaná v kapitole 1.2.2. Vypadá následovně:

```
// Partially matched crossover
template <typename T>
void PMX(const galgo::Population<T>& x, galgo::CHR<T>& chr1, galgo::CHR<T>&
chr2)
{
    // Randomly select two crossover points
    int point1 = galgo::uniform<int>(0, chr1->size() - 1);
    int point2 = galgo::uniform<int>(0, chr1->size() - 1);

    // Ensure point1 < point2
    if (point1 > point2) {
        std::swap(point1, point2);
    }

    // Create a mapping between corresponding genes in the parents
    std::unordered_map<T, T> mapping;
    for (int i = point1; i <= point2; ++i) {
        mapping[chr1->getParam()[i]] = chr2->getParam()[i];
        mapping[chr2->getParam()[i]] = chr1->getParam()[i];
    }
```

```
    // Swap the outer part according to the mapping
    for (int i = 0; i < chr1->size(); ++i) {
        if (i < point1 || i > point2) {
            T gene = chr1->getParam()[i];
            while (mapping.find(gene) != mapping.end()) {
                gene = mapping[gene];
            }
            chr1->setGeneValue(i, gene);
        }
        if (i < point1 || i > point2) {
            T gene = chr2->getParam()[i];
            while (mapping.find(gene) != mapping.end()) {
                gene = mapping[gene];
            }
            chr2->setGeneValue(i, gene);
        }
    }
}
```

1. Vytvoření nové funkce *PMX*: definuje se nová funkce speciálně pro křížení s částečnou shodou (Partially Matched Crossover) v knihovně Galgo.
2. Generování náhodných indexů: náhodným výběrem dvou indexů v chromozomu jsou definovány body křížení.
3. Zmapování vnitřní části: identifikování genů mezi dvěma body křížení a vytvoření mapování mezi odpovídajícími geny u rodičů.
4. Prohození vnější části: proměnění genů mimo body křížení mezi rodiči podle mapování vytvořeného v předchozím kroku.

Generování potomků: Vytvoření chromozomů potomků kombinací zmapované vnitřní části od jednoho rodiče a prohozené vnější části od druhého rodiče.

5.4.3 Uspořádané křížení (Ordered crossover)

Uspořádané křížení, jak je popsáno v kapitole 1.2.2 se implementuje do knihovny následujícími kroky (také viz kód níže):

1. Výběr bodů křížení: náhodným výběrem dvou bodů v chromozomu se definuje rozsah křížení.
2. Vytvoření vnitřní části: zkopírují se geny v rámci křížení od jednoho rodiče k odpovídajícímu potomkovi.
3. Vytvoření vnější části: začne se zprava od křížení a zkopírují se geny z druhého rodiče na potomka, pokud již nejsou přítomny.
4. Generování potomků: spojením vnitřní a vnější části se vytvoří potomstvo chromozomů.

```
// Ordered crossover
template <typename T>
void OrderedCrossover(const galgo::Population<T>& x, galgo::CHR<T>& chr1,
galgo::CHR<T>& chr2)
{
    // Randomly select two crossover points
    int point1 = galgo::uniform<int>(0, chr1->size() - 1);
    int point2 = galgo::uniform<int>(0, chr1->size() - 1);

    // Ensure point1 < point2
    if (point1 > point2) {
        std::swap(point1, point2);
    }

    // Create inner part by copying genes from one parent to the offspring
    for (int i = point1; i <= point2; ++i) {
        chr1->setGeneValue(i, chr2->getParam()[i]);
        chr2->setGeneValue(i, chr1->getParam()[i]);
    }

    // Create outer part by copying genes from the other parent if they are
not already present
    for (int i = point2 + 1; i < chr1->size(); ++i) {
        T geneValue = chr2->getParam()[i];
        bool found = false;
        for (int j = 0; j <= point2; ++j) {
            if (chr1->getParam()[j] == geneValue) {
                found = true;
                break;
            }
        }
        if (!found) {
            chr1->setGeneValue(i, geneValue);
        }

        geneValue = chr1->getParam()[i];
        found = false;
        for (int j = 0; j <= point2; ++j) {
            if (chr2->getParam()[j] == geneValue) {
                found = true;
                break;
            }
        }
        if (!found) {
            chr2->setGeneValue(i, geneValue);
        }
    }
}
```

5.5 Implementace dalších metod mutace

Mimo křížení lze rozšířit knihovnu o metody mutací. Implementované metody mutace pracují s *param* v chromozomu, a tudíž je nejdřív nutné přesunout jej z privátních do veřejných

atributů. Vzhledem k velkému počtu různých mutací (viz kapitola 1.2.3) se budeme zabývat třemi vybranými:

- Gaussovská mutace
- Swap mutace
- Inverzní mutace

5.5.1 Gaussovská mutace

Gaussovská mutace používá Gaussovo rozdělení, které není součástí knihovny, je třeba ho tedy definovat.

```
//define Gaussian destribution
template <typename T>
double gauss(double mean, double stddev) {
    static std::random_device rd;
    static std::mt19937 gen(rd());
    std::normal_distribution<double> dist(mean, stddev);
    return dist(gen);
}
```

Gaussovská mutace je specifická mutační metoda, která aplikuje náhodné změny na hodnoty genů v chromozomu podle Gaussova rozdělení. Nejdřív funkce přijímá ukazatel na chromozom (‘*galgo::CHR<T>& chr*‘), následně získáváme míru mutace (‘*mutrate*‘) z chromozomu. Následuje podmínka *if* (když), která zkoumá, jestli není *mutrate* nulová (pokud ano, funkce končí). Pro každý gen v chromozomu se generuje náhodné číslo z Gaussova rozdělení s parametry střední hodnoty 0 a standardní odchylky 1. Tato hodnota je násobena mírou mutace a přičtena k hodnotě genu, čímž vznikne nová hodnota genu. Tento proces je opakován pro všechny geny v chromozomu pomocí cyklu *while*.

```
//Gaussian mutation
template <typename T>
void GaussianMutation(galgo::CHR<T>& chr)
{
   T mutrate = chr->mutrate();

   if (mutrate == 0.0) return;

   for (int i = 0; i < chr->nbgene(); ++i) {
      // Apply mutation with probability mutrate
      if (galgo::proba(galgo::rng) <= mutrate) {
         // Generate a random number from a Gaussian distribution
         std::random_device rd;
         std::mt19937 gen(rd());
         std::normal_distribution<T> distribution(0.0, 1.0); // mean = 0,
standard deviation = 1
         T perturbation = distribution(gen);

         // Perturb the gene value
         T newValue = chr->getGene(i) + perturbation;
         chr->setGeneValue(i, newValue);
      }
   }
}
```

5.5.2 Swap Mutace

Swap mutace je implementována následujícím způsobem. Nejdřív funkce přijímá ukazatel na chromozom (‘*galgo::CHR<T>& chr*‘), následně získáváme míru mutace (‘*mutrate*‘) z chromozomu. Následuje podmínka *if* (když), která zkoumá, jestli není *mutrate* nulová (pokud ano, funkce končí). Pro každý gen v chromozomu se generuje náhodné číslo mezi 0 a 1, a to je porovnáno s *mutrate*, jestli má dojít k mutaci (pokud je toto číslo menší nebo rovno *mutrate*, pak nastane mutace). Má-li dojít k mutaci, jsou generovány náhodné indexy pro výměnu dvou genů. Poté jsou tyto geny vyměněny, čímž vznikne nová mutovaná verze chromozomu.

Funkce je (po dokončení cyklu) ukončena a nevrací žádný výsledek, protože se mutace provádí přímo na chromozomu.

```
// Swap Mutation
template <typename T>
void SwapMutation(galgo::CHR<T>& chr)
{
   T mutrate = chr->mutrate();

   if (mutrate == 0.0) return;

   for (int i = 0; i < chr->nbgene(); ++i) {
      // Apply mutation with probability mutrate
      if (galgo::proba(galgo::rng) <= mutrate) {
         // Generate random indices for swapping
         int index1 = galgo::uniform<int>(0, chr->nbgene() - 1);
         int index2 = galgo::uniform<int>(0, chr->nbgene() - 1);

         // Swap the genes at index1 and index2
         T gene1 = chr->getGene(index1);
         T gene2 = chr->getGene(index2);

         chr->setGeneValue(index1, gene2);
         chr->setGeneValue(index2, gene1);
      }
   }
}
```

5.5.3 Inverzní mutace

Inverzní mutace obsahuje nejdelší kód z implementovaných mutačních metod. Nejdřív funkce přijímá ukazatel na chromozom (‘*galgo::CHR<T>& chr*‘), následně získáváme míru mutace (‘*mutrate*‘) z chromozomu. Následuje podmínka *if* (když), která zkoumá, jestli není *mutrate* nulová (pokud ano, funkce končí). Dále následuje výběr bodů inverze, který spočívá v náhodném výběru dvou různých indexů chromozomu, které tyto body určují (indexy jsou vybrány od 0 do délky chromozomu - 1).

Samotné provedení inverze předchází kontrola, zda jsou vybrané indexy ve správném pořadí (*index1* < *index2*) podmínkou *if*. Jestliže jsou ve špatném pořadí, uvedou se do správného pomocí proměnné *temp*. Inverze je prováděna v cyklu *while:* Postupujeme tak, že zaměníme hodnoty genů mezi indexem *index1* a indexem *index2* a postupně se posouváme směrem k sobě, dokud se nezachytíme na stejném místě, nebo se nepotkáme.

Funkce je (po provedení inverze) ukončena a nevrací žádný výsledek, protože se mutace provádí přímo na chromozomu.

```
// Inversion Mutation
template <typename T>
void InversionMutation(galgo::CHR<T>& chr)
{
   T mutrate = chr->mutrate();

   if (mutrate == 0.0) return;

   for (int i = 0; i < chr->nbgene(); ++i) {
      // Apply mutation with probability mutrate
      if (galgo::proba(galgo::rng) <= mutrate) {
         // Generate two random indices within the range of the genes
         int index1 = galgo::uniform<int>(0, chr->nbgene() - 1);
         int index2 = galgo::uniform<int>(0, chr->nbgene() - 1);

         // Ensure index1 is less than index2
         if (index1 > index2) {
            std::swap(index1, index2);
         }

         // Invert the order of genes between index1 and index2
         while (index1 < index2) {
            T gene1 = chr->getGene(index1);
            T gene2 = chr->getGene(index2);

            chr->setGeneValue(index1, gene2);
            chr->setGeneValue(index2, gene1);

            index1++;
            index2--;
         }
      }
   }
}
```

5.6 Výsledné použití knihovny

Se změnami v knihovně také přichází potřeba upravit zdrojový soubor .cpp, který knihovnu používá. Je potřeba zadat nový parametr bestResultStreakmax. Iniciace genetického algoritmu se upraví např. následovně:

Z původního:

```
    // initiliazing genetic algorithm
   galgo::GeneticAlgorithm<double>
ga(MyObjective<double>::Objective,100,50,true,par1,par2);
```

na upravený:

```
    // initiliazing genetic algorithm
   galgo::GeneticAlgorithm<double>
ga(MyObjective<double>::Objective,100,50,true,10,par1,par2);
```

Kde argumenty jsou: cílová funkce, velikost populace (počet jedinců v populaci), počet generací, informace, zdali vypisovat výsledek, maximální počet stejných nejlepších výsledků a jednotlivé parametry.

Zastavovací kritérium pro cílovou fitness se začlení do zdrojového kódu daného programu a vypadá např. *ga.FitnessBestTarget = 14*; ukázka použití je v kapitolách 7.3 a 7.4.

Nové metody křížení lze použít pomocí příkazů v tabulce 4.

Tabulka 4 Zdrojový kód použití nově implementovaných metod

Typ křížení/mutace	Zdrojový kód
Sudé – liché (Even-Odd) křížení	`ga.CrossOver = EOX;`
Křížení s částečnou shodou (Partially matched)	`ga.CrossOver = PMX;`
Uspořádané (ordered) křížení	`ga.CrossOver = OX;`
Gaussovská mutace	`ga.Mutation = GaussianMutation;`
Swap mutace	`ga.Mutation = SwapMutation;`
Inverzní mutace	`ga.Mutation = InversionMutation;`

V následující kapitole je uveden příklad použití knihovny na logickém problému.

6 Použití knihovny Galgo na příkladu

Pro možnost porovnání výsledků je použití knihovny Galgo uvedeno na logickém příkladu stejném, jako používala T. Panská ve své práci [2].

6.1 Zadání logického problému

Logický problém je definován následovně:

Na pouti pět chlapců různého věku konzumuje různé druhy jídel a dává přednost různým atrakcím. Tvrzení jsou následující:

1. Ron jí zmrzlinu.
2. Joe nemá rád žvýkačky.
3. Samovi je 14 let.
4. Sam není na horské dráze.
5. Chlapci na zvonkové dráze je 15 let.
6. Len není ve strašidelném zámku.
7. Don je na kolotoči.
8. Chlapci, který nemá rád zmrzlinu, je 13 let.
9. Chlapec ve strašidelném zámku jí párek v rohlíku.
10. Joe je jedenáctiletý.
11. Joe jí hranolky.
12. Joe je na lochnesce.
13. Donovi je 12 let.
14. Don si pochutnává na cukrové vatě.

6.2 Limitace kódováním

Než bude představen samotný kód programu, je důležité představit omezení knihovny, kterými jsme při řešení problému limitováni. Genetický algoritmus přijímá na vstupu cílovou funkci pouze typu *float* nebo *double* (desetinné číslo), nikoliv *int* (celé číslo), který by se zde hodil více. Dekódovaný chromozom vrací hodnoty typu double (viz kód níže), proto je v kódu použito přetypování na typ int.

Chromozomy jsou v knihovně Galgo reprezentovány jako binární řetězce obsahující zakódované parametry. Počet bitů potřebných pro zakódování každého parametru je dán uživatelem a odpovídá počtu parametrů specifikovaných v konstruktoru genetického algoritmu (…, `const Parameter<T,N>&...args`). Tento počet bitů může být libovolný z intervalu [1;64]. Při inicializaci populace je pro každý parametr vygenerován náhodný 64bitový unsigned

integer v rozsahu [0, MAXVAL], kde MAXVAL představuje největší hodnotu pro daný počet bitů (N). Tento náhodně vygenerovaný unsigned integer je poté převeden na binární řetězec, který je následně zkrácen na požadovaný počet bitů a přidán do chromozomu. Po aplikaci selekce, křížení a mutace je binární řetězec opětovně převeden na unsigned integer, přičemž je zohledněna omezení parametrů.

Kódování a dekódování probíhá následujícím kódem (knihovna obsahuje metody pro kódování známého celého čísla i neznámého čísla):

```
   // encoding random unsigned integer
   std::string encode() const override {
      std::string str = GetBinary(galgo::Randomize<N>::generate());
      return str.substr(str.size() - N, N);
   }
   // encoding known unsigned integer
   std::string encode(T z) const override {
      uint64_t value = Randomize<N>::MAXVAL * (z - data[0]) / (data[1] -
data[0]);
      std::string str = GetBinary(value);
      return str.substr(str.size() - N, N);
   }
   // decoding string to real value
   T decode(const std::string& str) const override {
      return data[0] + (GetValue(str) /
static_cast<double>(Randomize<N>::MAXVAL)) * (data[1] - data[0]);
   }
```

Pro řešení problému jsou v kapitole 7.4 implementovány speciální metody mutace a křížení, které optimalizují řešení zadaného logického problému.

6.3 Implementace kódu

V prvním kroku je nutné si uvědomit, jaké jsou vstupní informace konstruktoru genetického algoritmu. V tuto chvíli jsou nejdůležitější cílová funkce (objective function), parametry a omezení (constrains). Dále je zde omezení na typ proměnné, který musí být *float* nebo *double.*

V kódu se definuje cílová funkce pomocí čtrnácti tvrzení, která musí být splněna. Při každém splnění podmínky se zvýší proměnná počtu splněných podmínek *conditions*. To nám zaručí, že fitness funkce se bude zvyšovat a nemusí se měnit implementace seřazování v populaci od nejlepšího po nejhorší. Tuto funkci by sice bylo možné implementovat, avšak se seřazením a fitness se operuje na vícero místech kódu a bylo by nutné pokrýt všechny výskyty. Přínos této změny by byl moc malý (většinou lze fitness obrátit velmi snadno) vzhledem k počtu úprav.

Funkce vrací vektor hodnot typu *double.* Ukazatel na funkci je pak vstupním parametrem pro konstruktor v *main()*.

```cpp
using namespace std;
template <typename T>
vector<T> objective(const vector<T>& x) {
    // The count of satisfied conditions
    int conditions = 0;
    int i;

    // 1. Ron jí zmrzlinu
    if (static_cast<int>(x[0]) == 1) conditions++;

    // 2. Joe nemá rád žvýkačky
    if (static_cast<int>(x[1]) != 0) conditions++;

    // 3. Samovi je 14 let
    if (static_cast<int>(x[12]) == 14) conditions++;

    // 4. Sam není na horské dráze
    if (static_cast<int>(x[7]) != 0) conditions++;

    // 5. Chlapci na zvonkové dráze je 15 let
    // hledám, kdo je na zvonkové dráze
    for (i=5;i<10;i++)
      if (static_cast<int>(x[i])==1) break;
    if (i!=10 && static_cast<int>(x[i+5])==15 ) conditions++;

    // 6. Len není ve strašidelném zámku
    if (static_cast<int>(x[8]) != 2) conditions++;

    // 7. Don je na kolotoči
    if (static_cast<int>(x[9]) == 3) conditions++;

    // 8. Chlapci, který nemá rád zmrzlinu, je 13 let
    // hledám, komu je 13
    for (i=10;i<15;i++)
      if (static_cast<int>(x[i])==13) break;
    if (i!=15 && static_cast<int>(x[i-10])!=1) conditions++;

    // 9. Chlapec ve strašidelném zámku jí párek v rohlíku
    // hledám, kdo je ve strašidelném zámku
    for (i=5;i<10;i++)
      if (static_cast<int>(x[i])==2) break;
    if (i!=10 && static_cast<int>(x[i-5])==2) conditions++;
```

```
    // 10. Joe je jedenáctiletý
    if (static_cast<int>(x[11]) == 11) conditions++;

    // 11. Joe jí hranolky
    if (static_cast<int>(x[1]) == 3) conditions++;

    // 12. Joe je na lochnesce
    if (static_cast<int>(x[6]) == 4) conditions++;

    // 13. Donovi je 12 let
    if (static_cast<int>(x[14]) == 12) conditions++;

    // 14. Don si pochutnává na cukrové vatě
    if (static_cast<int>(x[4]) == 4) conditions++;

    T objectiveValue = static_cast<T>(conditions);

    // Vytvoření vektoru s hodnotou objektivní funkce
    vector<T> result = { objectiveValue };

    return result;
}
```

Protože náhodné počáteční vygenerování chromozomů může vytvořit neplatného jedince (dva chlapci jí stejnou pochutinu, některá pochutina „vypadne“ atd.), je definováno omezení. Knihovna pracuje s omezující funkcí tím způsobem, že záporné hodnoty omezení jsou akceptovatelné, v případě kladné hodnoty omezení je tato odečítána od hodnoty kriteriální funkce a dochází tak k penalizaci neplatných jedinců.

```
vector<double> MyConstraint(const vector<double>& x)
{
   vector<double> cst;
   int index[3]={0,5,10};
   int val;
   int i;

   for(i=0;i<3;i++)
   {
     val=-1;
     if (static_cast<int>(x[index[i]+0])==static_cast<int>(x[index[i]+1]))
val=14;
     if (static_cast<int>(x[index[i]+0])==static_cast<int>(x[index[i]+2]))
val=14;
     if (static_cast<int>(x[index[i]+0])==static_cast<int>(x[index[i]+3]))
val=14;
     if (static_cast<int>(x[index[i]+0])==static_cast<int>(x[index[i]+4]))
val=14;
     if (static_cast<int>(x[index[i]+1])==static_cast<int>(x[index[i]+2]))
val=14;
     if (static_cast<int>(x[index[i]+1])==static_cast<int>(x[index[i]+3]))
val=14;
     if (static_cast<int>(x[index[i]+1])==static_cast<int>(x[index[i]+4]))
val=14;
     if (static_cast<int>(x[index[i]+2])==static_cast<int>(x[index[i]+3]))
val=14;
     if (static_cast<int>(x[index[i]+2])==static_cast<int>(x[index[i]+4]))
val=14;
     if (static_cast<int>(x[index[i]+3])==static_cast<int>(x[index[i]+4]))
val=14;
     cst.push_back(static_cast<double>(val));
   }
  return cst;
}
```

Tato funkce bere jako vstup vektor *double x* (hodnoty parametrů) a porovnává konkrétní prvky *x* na základě předdefinovaných indexů (vždy v rámci pětice). Pokud jsou některé prvky stejné, přiřadí *val* hodnotu 14 (maximální penalizace), jinak *val* zůstane -1. Funkce pak připojí výslednou hodnotu *val* (přetypovanou na *double*) k novému vektoru *cst* pro každou sadu porovnání (pětici). Nakonec vrátí vektor *cst* obsahující výsledky porovnání.

Nakonec bude vytvořena funkce *main(),* která je jádrem programu. Je v ní obsaženo definování parametrů, konstruktor GA, jeho vlastnosti (velikost a počet generací a míra mutace a křížení aj.), a příkaz *ga.run(),* který spustí samotný genetický algoritmus a vypisuje výsledky v čitelné formě. Funkce *main()* formálně vrací hodnotu 0 a v základu vypadá následovně:

```
int main() {
    // Initialize parameters lower and upper bounds
    // Define parameter bounds for food
    galgo::Parameter<double,3> RonFood({0, 4.9,1.2}); // 0: Žvýkačka, 1:
Zmrzlina, 2: Párek v rohlíku, 3: Hranolky, 4: Cukrová vata
    galgo::Parameter<double,3> JoeFood({0, 4.9,3.5}); // 0: Žvýkačka, 1:
Zmrzlina, 2: Párek v rohlíku, 3: Hranolky, 4: Cukrová vata
    galgo::Parameter<double,3> SamFood({0, 4.9,0.5}); // 0: Žvýkačka, 1:
Zmrzlina, 2: Párek v rohlíku, 3: Hranolky, 4: Cukrová vata
    galgo::Parameter<double,3> LenFood({0, 4.9,2.8}); // 0: Žvýkačka, 1:
Zmrzlina, 2: Párek v rohlíku, 3: Hranolky, 4: Cukrová vata
    galgo::Parameter<double,3> DonFood({0, 4.9,4.2}); // 0: Žvýkačka, 1:
Zmrzlina, 2: Párek v rohlíku, 3: Hranolky, 4: Cukrová vata
    // Define parameter bounds for attractions
    galgo::Parameter<double,3> RonAttraction({0, 4.9,0.3}); // 0: Horská
dráha, 1: Zvonková dráha, 2: Strašidelný zámek, 3: Kolotoč, 4: Lochneska
    galgo::Parameter<double,3> JoeAttraction({0, 4.9,4.3}); // 0: Horská
dráha, 1: Zvonková dráha, 2: Strašidelný zámek, 3: Kolotoč, 4: Lochneska
    galgo::Parameter<double,3> SamAttraction({0, 4.9,2.7}); // 0: Horská
dráha, 1: Zvonková dráha, 2: Strašidelný zámek, 3: Kolotoč, 4: Lochneska
    galgo::Parameter<double,3> LenAttraction({0, 4.9,1.6}); // 0: Horská
dráha, 1: Zvonková dráha, 2: Strašidelný zámek, 3: Kolotoč, 4: Lochneska
    galgo::Parameter<double,3> DonAttraction({0, 4.9,3.2}); // 0: Horská
dráha, 1: Zvonková dráha, 2: Strašidelný zámek, 3: Kolotoč, 4: Lochneska
    // Define parameter bounds for age
    galgo::Parameter<double,3> RonAge({11, 15.9,15.6}); // Věk od 11 do 15
let
    galgo::Parameter<double,3> JoeAge({11, 15.9,11.6}); // Věk od 11 do 15
let
    galgo::Parameter<double,3> SamAge({11, 15.9,14.7}); // Věk od 11 do 15
let
    galgo::Parameter<double,3> LenAge({11, 15.9,13.8}); // Věk od 11 do 15
let
    galgo::Parameter<double,3> DonAge({11, 15.9,12.8}); // Věk od 11 do 15
let

    // Initialize genetic algorithm with defined parameters
    galgo::GeneticAlgorithm<double> ga(objective,2000,5000,true,1000,Ron-
Food, JoeFood, SamFood, LenFood,DonFood,RonAttraction,JoeAttraction,SamAt-
traction,LenAttraction,DonAttraction,RonAge,JoeAge,SamAge,LenAge,DonAge);

    // set parameters of GA
    //enter GA parameters here

    // Run genetic algorithm
    ga.run();

      std::cout << "Nalezene reseni:\n";

      int i;
    for(i = 0; i < 5; i++)   std::cout << convert_name(i) << ": " << con-
vert_food(ga.pop(0)->getParam()[i]) << std::endl;
```

```
    for(i = 5; i < 10; i++)  std::cout << convert_name(i - 5) << ": " <<
convert_attraction(ga.pop(0)->getParam()[i]) << std::endl;;
    for(i = 10; i < 15; i++) std::cout << convert_name(i - 10) << ": " <<
convert_age(ga.pop(0)->getParam()[i]) << std::endl;;;

    return 0;
}
```

Jednotlivé varianty programu s výsledky jsou popsány v následující kapitole.

7 Výsledky a diskuse

Tato kapitola pojednává o tom, která kombinace parametrů (včetně volby typu selekce, mutace a křížení) vede k tíženému výsledku. Důležité je splnění všech podmínek, což nastane, když fitness funkce F(x) bude rovna 14. Níže v tabulce 5 jsou vypsány výsledky vybraných pokusů, které v průběhu práce byly zkoušeny. V jednotlivých kombinacích se vyskytují: výchozí náhodná selekce ruletového kola (RWS) a turnajová selekce (TNT), výchozí jednobodová mutace (SPM) a swap mutace včetně vlastní mutace pro tuto úlohu (vlastní), výchozí jednobodové křížení (P1XO) a křížení s částečnou shodou vytvořené pro tento logický problém (vlastní). Vlastní operátory v příkladu 3 a ve srovnání s knihovnou GA fungují stejně, avšak nejsou zcela totožné (liší se například použitím cyklu). Tabulka 5 také obsahuje časový údaj o tom, jak dlouho trvá logický problém vyřešit pomocí knihovny OpenGA [2].

Tabulka 5 Přehled výsledků genetických algoritmů

Číslo příkladu (case)	Velikost generace	Selekce	Mutace	Křížení	F(x)	Doba běhu programu [s]	Počet gen
0 – Basic	200	RWS	SPM	P1XO	3 (8)	0,735	500
1 – Swap	200	TNT	Swap	P1XO	13	0,452	280*
2 – Velké generace	800	TNT	Swap	P1XO	14	0,597	106**
3 – Speciální operátory	200	TNT	Vlastní	Vlastní	13	0,097	53**
Srovnání s OpenGA [2]	200	Vlastní	Vlastní	Vlastní	14	3,475	500

*zastavovací pravidlo *bestResultStreak* ukončilo algoritmus

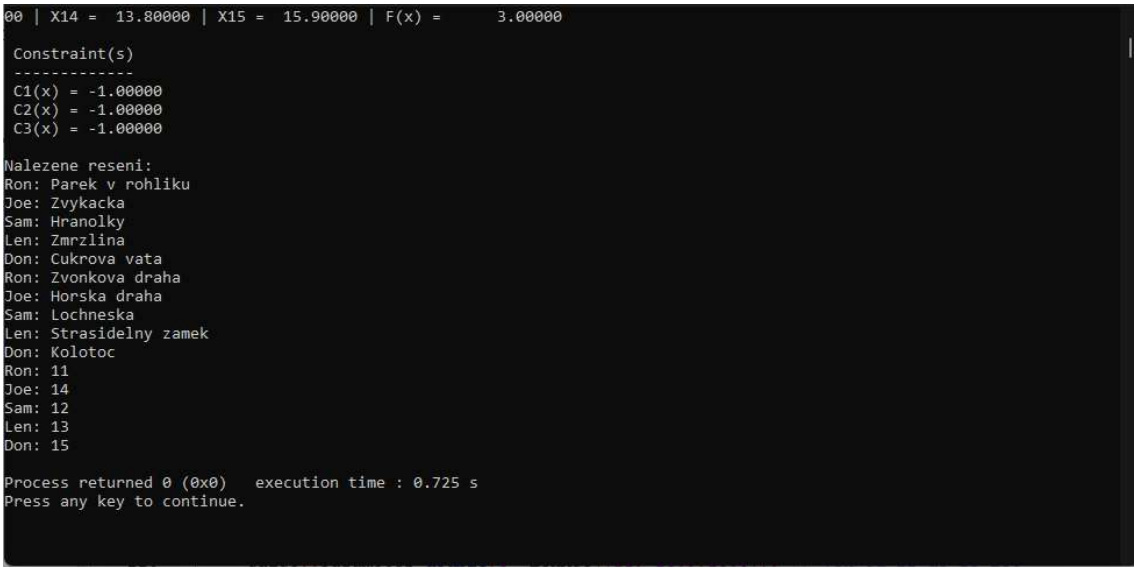
**Cílová fitness ukončila algoritmus

7.1 Case 0

První nastavení GA je provedeno pro srovnání s prací T. Panské [2], a to implementací řešení pomocí výchozích metod (RWS selekce, SPM mutace a P1XO křížení) a nastavení pouze velikosti populace (200), počtu generací (500), počtu elit (100) a koeficientu křížení (0.8) a mutace (0.4). Číslo 0 je vybráno, protože se výsledek vůbec neblíží požadovanému a je čistě pro ilustraci, že nastavení, které funguje v jedné knihovně, nemusí fungovat v jiné, hlavně pokud jsou jinak implementovány křížení a mutace. Tento algoritmus vypadá následovně:

```
int main() {
…
    // Initialize genetic algorithm with defined parameters
    galgo::GeneticAlgorithm<double> ga(objective,200,500,true,0,RonFood,
JoeFood, SamFood, LenFood,DonFood,RonAttraction,JoeAttraction,SamAttrac-
tion,LenAttraction,DonAttraction,RonAge,JoeAge,SamAge,LenAge,DonAge);

    // set parameters of GA
    ga.covrate = 0.8;
    ga.mutrate = 0.4;
    ga.elitpop=100;
    ga.Constraint=MyConstraint;
    // Run genetic algorithm
    ga.run();
    return 0;
}
```

Výsledky však nejsou dobré a fitness dosahuje nejvýše hodnoty 8, avšak ke konci algoritmu klesá na hodnotu 3 (pro výsledky viz Obrázek 18).



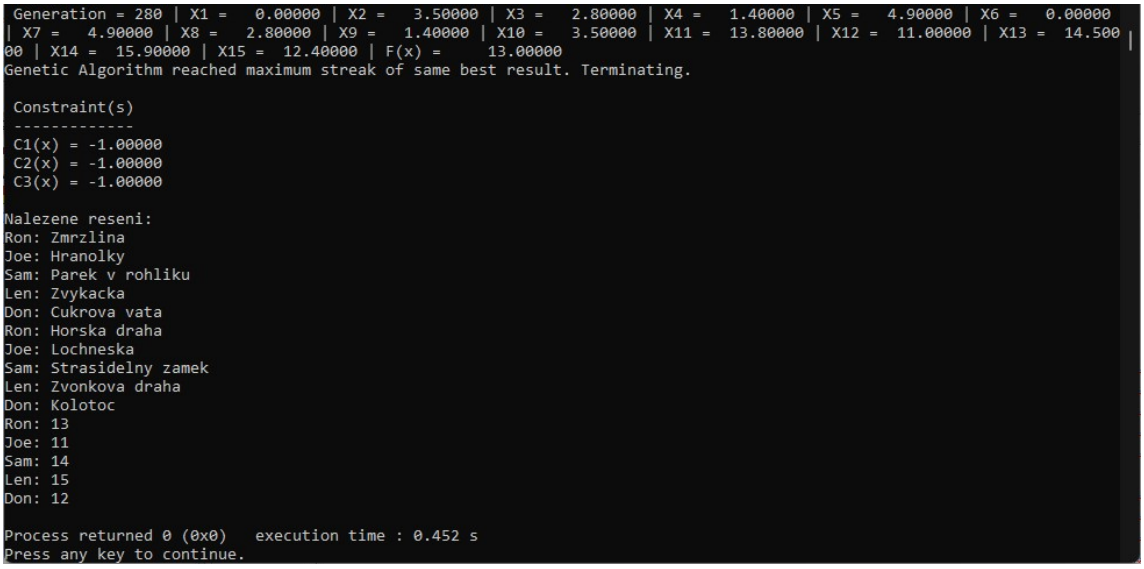
Obrázek 18 Výsledek case 0, zdroj: vlastní

7.2 Case 1

Lepší příklad je použití Swap mutace (viz kapitola 5.5.2) a selekce TNT. Zároveň je zde upraveno *mutrate*, které určuje četnost mutace. Zde je fitness funkce 13 a výsledek není správný ani po 500 generacích. Parametry je nutné dále upravovat, ale už nyní lze mluvit o posunu správným směrem. Kód tohoto příkladu je následující:

```
int main() {
…
    // Initialize genetic algorithm with defined parameters
    galgo::GeneticAlgorithm<double> ga(objective,200,500,true,0,RonFood,
JoeFood, SamFood, LenFood,DonFood,RonAttraction,JoeAttraction,SamAttrac-
tion,LenAttraction,DonAttraction,RonAge,JoeAge,SamAge,LenAge,DonAge);

    // set parameters of GA
    ga.Mutation=SwapMutation;
    ga.covrate = 0.8;
    ga.mutrate = 0.1;
    ga.elitpop=100;
    ga.Selection=TNT;
    ga.tntsize=2;
    ga.Constraint=MyConstraint;
    ga.FitnessBestTarget = 14.0;
    // Run genetic algorithm
    ga.run();
    return 0;
}
```

U tohoto příkladu je použito zastavovací pravidlo pro maximální počet generací se stejnou hodnotou fitness. Program se tedy zastaví, pokud se tato hodnota po 200 generací nezmění. V tomto případě ušetří toto zastavovací pravidlo 0,286 s, což odpovídá zhruba 63 % časové náročnosti (viz obrázek 19).



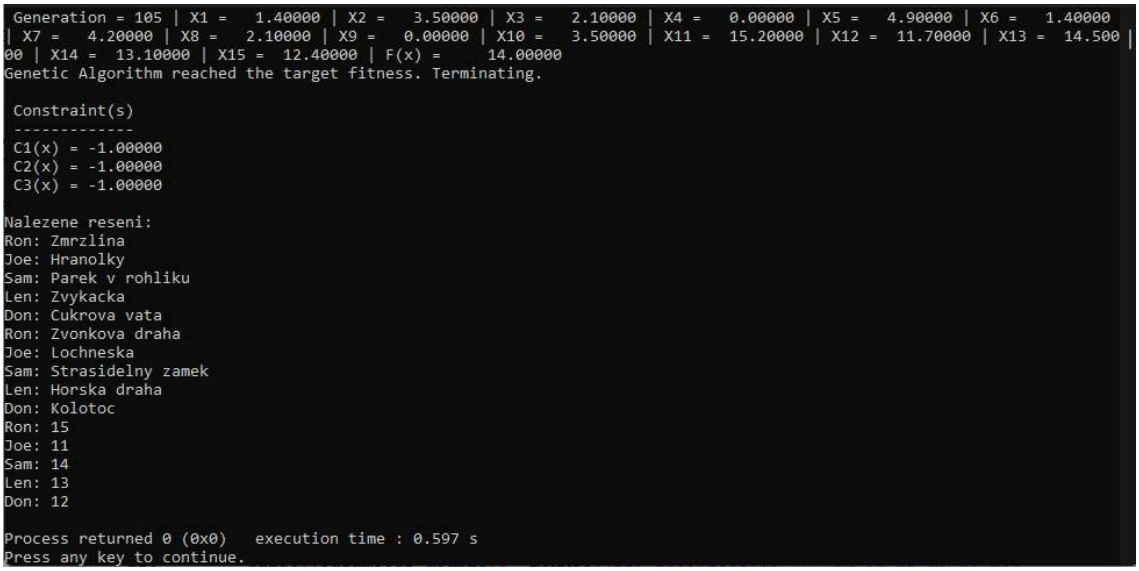
Obrázek 19 Výsledky Case 1, zdroj: vlastní

7.3 Case 2

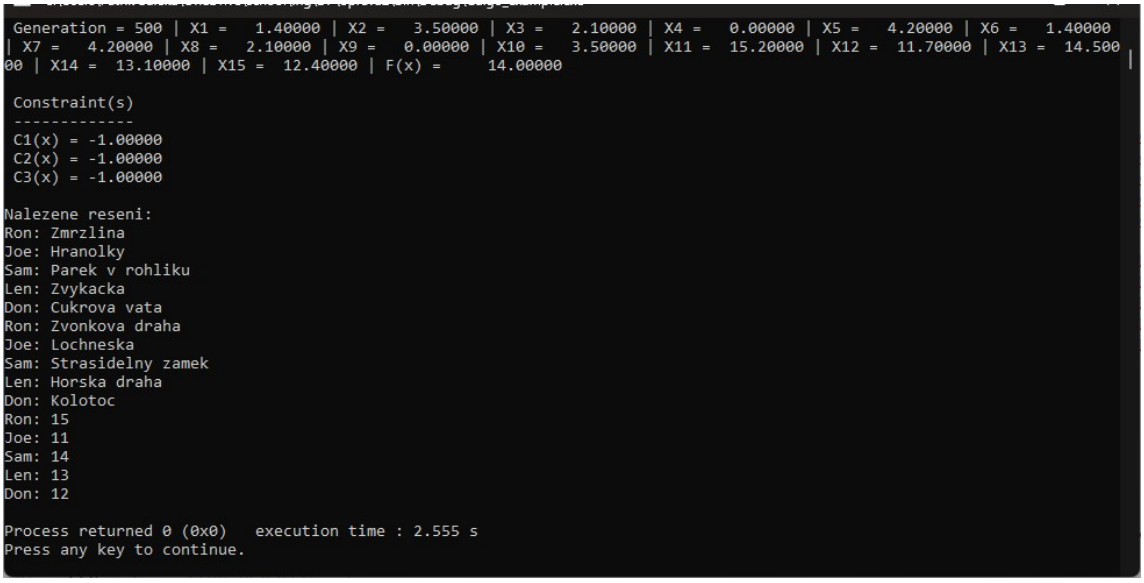
Algoritmus se od toho v předchozím příkladu liší velikostí populace, která je zde navýšena na 800 jedinců. Je zde uplatněno zastavovací pravidlo implementované v kapitole 5.2.2. Kód vypadá následovně:

```
int main() {
…
    // Initialize genetic algorithm with defined parameters
    galgo::GeneticAlgorithm<double> ga(objective,800,500,true,0,RonFood,
JoeFood, SamFood, LenFood,DonFood,RonAttraction,JoeAttraction,SamAttrac-
tion,LenAttraction,DonAttraction,RonAge,JoeAge,SamAge,LenAge,DonAge);

    // set parameters of GA
    ga.Mutation=SwapMutation;
    ga.covrate = 0.8;
    ga.mutrate = 0.1;
    ga.elitpop=100;
    ga.Selection=TNT;
    ga.tntsize=2;
    ga.Constraint=MyConstraint;
    ga.FitnessBestTarget = 14.0;
    // Run genetic algorithm
    ga.run();
    return 0;
}
```

Tento program také běží kratší dobu než bez něj (0,597 s oproti 2,555 s; viz obrázky 20 a 21), přičemž výsledky zůstávají stejné. Použité zastavovací pravidlo je tedy vhodné, pokud požadujeme co nejkratší dobu běhu programu. Na obrázku 20 lze vidět chování programu při dosažení požadované fitness.



Obrázek 20 Výsledky Case 2 se zastavovacím pravidlem, zdroj: vlastní



Obrázek 21 Výsledky Case 2 bez zastavovacího pravidla, zdroj: vlastní

7.4 Case 3

Pro poslední příklad optimalizace programu byla použita vlastní úprava křížení a mutace (v kódu označeny jako mutace a křížení (logického) problému *ProblemCrossover* a *ProblemMutation)*. Toto křížení a mutace, které avšak nejsou univerzální, nejsou tudíž zařazeny do knihovny v kapitole 5.4.

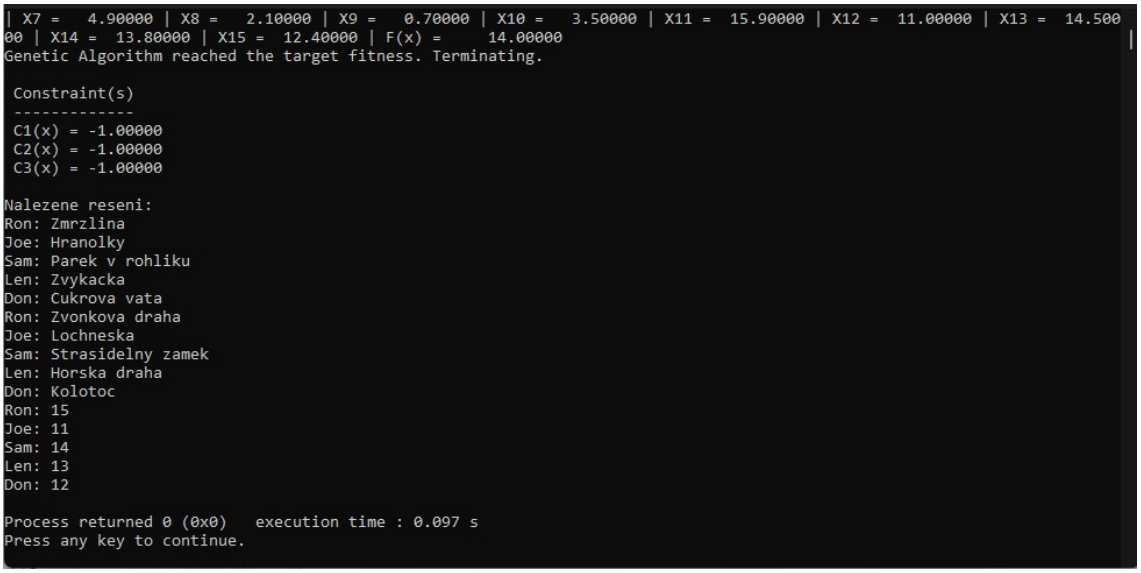
Princip křížení *ProblemCrossover* je ten, že potomek zdědí od jednoho rodiče přiřazení pochutin a atrakcí a od druhého věk všech chlapců a tím zabrání vzniku neplatného jedince (což hrozí u obecného křížení). Probíhá následovně: operátor nejprve náhodně vybere dva chromozomy z populace určené k páření. Poté náhodně vybere pozice pro křížení mezi dvěma hranicemi. Tato pozice je určena násobením náhodného čísla mezi 1 a 3 (včetně) a následným násobením 5 a 3. Tím se vytvoří hodnota, která určuje pozici křížení mezi dvěma hranicemi. Následně jsou přeneseny části genů do nových chromozomů. První část genů z prvního chromozomu je přenesena do prvního nového chromozomu, zatímco první část genů z druhého chromozomu je přenesena do druhého nového chromozomu. Poté jsou přeneseny zbylé části genů v opačném pořadí, což vytváří dva nové chromozomy s kombinací genů obou rodičů. Kód operátoru vypadá následovně:

```
template <typename T>
void ProblemCrossover(const galgo::Population<T>& x, galgo::CHR<T>& chr1,
galgo::CHR<T>& chr2)
{
   // choosing randomly 2 chromosomes from mating population
   int idx1 = galgo::uniform<int>(0, x.matsize());
   int idx2 = galgo::uniform<int>(0, x.matsize());
   // choosing randomly a position for cross-over between two boundaries
   int pos = galgo::uniform<int>(1, 3);
   pos = pos*5*3-1;
   // transmitting portion of bits to new chromosomes
   chr1->setPortion(*x[idx1], 0, pos);
   chr2->setPortion(*x[idx2], 0, pos);
   chr1->setPortion(*x[idx2], pos + 1);
   chr2->setPortion(*x[idx1], pos + 1);
}
```

Mutace opět zabraňuje vzniku neplatného jedince a to způsobem, že prohazuje přiřazení jídel, atrakcí a věků. Probíhá následovně: Nejdřív funkce přijímá ukazatel na chromozom (‘*galgo::CHR<T>& chr*‘), následně získáváme míru mutace (‘*mutrate*‘) z chromozomu. Následuje podmínka *if* (když), která zkoumá, jestli není *mutrate* nulová (pokud ano, funkce končí). Poté funkce postupně prochází všechny geny chromozomu. Pro každý gen se rozhoduje, zda na něj bude aplikována mutace s pravděpodobností určenou mírou mutace. Pokud je pravděpodobnost mutace splněna, generují se náhodné indexy pro výběr dvou genů v rámci skupiny po pěti. Tyto indexy slouží k výměně hodnot mezi těmito geny, což představuje operaci swap mutace. Nakonec jsou hodnoty obou genů vyměněny a chromozom je mutován. Kód mutace vypadá následovně:

```
template <typename T>
void ProblemMutation(galgo::CHR<T>& chr)
{
   T mutrate = chr->mutrate();

   if (mutrate == 0.0) return;

   // swap mutation only within group of 5 elements
   for (int i = 0; i < chr->nbgene(); ++i) {
      // Apply mutation with probability mutrate
      if (galgo::proba(galgo::rng) <= mutrate) {
         // Generate random indices for swapping
         int low = (i / 5)*5; // lower bound of swap boundaries
         int high = low + 5; // higher bound of swap boundaries
         int index1 = galgo::uniform<int>(low, high);
         int index2 = galgo::uniform<int>(low, high);
         //Swap the genes at index1 and index2
         double gen1=chr->getGene(index1);
         double gen2=chr->getGene(index2);
         chr->setGeneValue(index1,gen2);
         chr->setGeneValue(index2,gen1);
      }
   }
}
```

Řešení logického problému včetně doby běhu programu je možné vidět na obrázku 22.



Obrázek 22 Řešení problému Case 3, zdroj: vlastní

Jak již bylo zmíněno, navazuje tato práce na bakalářskou práci T. Panské [2], která tento logický problém řešila pomocí knihovny OpenGA. Rozdíly mezi knihovnami už byly rozebrány v kapitole 3, a tudíž zde budou srovnávány již jen výsledky. V základním nastavení nedosahuje knihovna Galgo výsledků jako OpenGA, kde však operátory mutace a křížení byly implementovány ve zdrojovém kódu, a nikoliv jako součást knihovny.

Porovnáme-li pak poslední příklad programu knihovny Galgo s tím z OpenGA, pak můžeme sledovat, že čas běhu programu je u knihovny Galgo podstatně menší, než u knihovny OpenGA (0,115 s oproti 3,475 s). Je však důležité si uvědomit, že program byl napsán tak, aby vypisoval výsledky pouze formou, jak je nastaveno v knihovně (x_1=…) a nikoliv slovně (Ron jí zmrzlinu).

Závěrem lze shrnout, že program představený v této práci je časově efektivnější, avšak poskytuje výsledky méně přehledně a srozumitelně. Rozšířená knihovna Galgo je tedy vhodnější pro další práci v rámci většího projektu, kde je důležitější optimalizace výpočetního času.

Závěr

Prvním cílem této práce bylo seznámení s principy genetických algoritmů a neuronových sítí a knihoven zabývající se touto tematikou. Byly popsány různé části genetických algoritmů (selekce, křížení a mutace) a uvedeny jejich nejčastější typy. Knihovny byly vybrány primárně nativní pro jazyk C++.

Hlavním cílem práce bylo rozšířit vybranou knihovnu. Po pečlivém posouzení všech dostupných knihoven byly pro detailní hodnocení vybrány dvě: Galgo a GAlib. S ohledem na zastaralost a omezené možnosti knihovny GAlib bylo rozhodnuto soustředit se na knihovnu Galgo. Zároveň se využily osvědčené metody křížení a mutace z knihovny GAlib a tyto metody se adaptovaly a začlenily do modernější a výkonnější knihovny Galgo. Proběhl test funkčnosti všech implementovaných metod a odstranění všech chybových hlášení při kompilaci.

Poslední částí práce je demonstrace vylepšené knihovny na konkrétním příkladu logického problému a porovnávání dosažených výsledků s knihovnou OpenGA, kterou využila T. Panská ve své bakalářské práci [2]. Nově implementované operátory, jako je například swap mutace, a zavedená zastavovací pravidla jsou aplikovány v této ukázce.

Zároveň je poukázáno na to, že i když obecné metody poskytované knihovnou často vedou k úspěšnému řešení, je lepší vytvořit si vlastní operátory přizpůsobené konkrétnímu problému, což může vést k efektivnějšímu a přesnějšímu výsledku. V neposlední řadě je zdůrazněna vhodnost použití různých zastavovacích kritérií pro různé nastavení genetického algoritmu.

Použité zdroje

[1] KOMENDA, Radim. Využití genetických algoritmů ve shlukové analýze. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta dopravní, 2023.

[2] PANSKÁ, Tereza. VYUŽITÍ EVOLUČNÍCH TECHNIK PŘI UČENÍ UMĚLÝCH NEURONOVÝCH SÍTÍ. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta dopravní, 2022.

[3] MAŘÍK, Vladimír; ŠTĚPÁNKOVÁ, Olga a LAŽANSKÝ, Jiří. Umělá inteligence. 1. Praha: Academia, 1993. ISBN 80-200-0472-6.

[4] ZELINKA, Ivan. Evoluční výpočetní techniky: principy a aplikace. Praha: BEN - technická literatura, 2009. ISBN 978-80-7300-218-3.

[5] FOGEL, Lawrence J.; OWENS, Alvin J. a WALSH, Michael John. Iskusstvennyj intellekt i evoljucionnoje modelirovanije. Moskva: Mir, 1969. ISBN 0471265160.

[6] HOLLAND, John H. Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence. Ann Arbor: University of Michigan Press, 1975. ISBN 04-720-8460-7.

[7] A. SHAMS, Somia; HEKAL OMAR, Asmaa; S. DESUKY, Abeer; T. ABOU-KREISHA, Mohammad a A. ELSHARAWY, Gaber. Even-odd crossover: a new crossover operator for improving the accuracy of students' performance prediction. online. Bulletin of Electrical Engineering and Informatics. 2022, roč. 11, č. 4, s. 2292-2302. ISSN 2302-9285. Dostupné z: https://doi.org/10.11591/eei.v11i4.3841. [cit. 2024-04-13].

[8] Alleles, Loci, and the Traveling Salesman Problem. In: Proceedings of the First International Conference on Genetic Algorithms and their Applications. 1985, s. 154-159.

[9] LUIZ CLAUDIO OLIVEIRA DE ANDRADE, . GENETIC ALGORITHMS APPLICATION IN LINE SIMPLIFICATION. online, Final assignment report for the Postgraduate Diploma, vedoucí Dr., R, Zurita-Milla. Hallenweg 8, 7522 NH Enschede, Nizozemsko: Faculty of Geo-Information Science and Earth Observation of the University of Twente, 2014. Dostupné z: https://www.researchgate.net/figure/Partially-matched-crossover-Goldberg-Lingle-1985_fig7_329358589. [cit. 2024-04-02].

[10] DULEBENETS, Maxim A. A Comprehensive Evaluation of Weak and Strong Mutation Mechanisms in Evolutionary Algorithms for Truck Scheduling at Cross-Docking Terminals. online. IEEE Access. 2018, roč. 6, s. 65635-65650. ISSN 2169-3536. Dostupné z: https://doi.org/10.1109/ACCESS.2018.2874439. [cit. 2024-04-02].

[11] HAUPT, Randy L. a HAUPT, S. E. Practical genetic algorithms. 2nd ed. Hoboken: John Wiley, 2004. ISBN 04-714-5565-2.

[12] LIM, Siew Mooi; SULTAN, Abu Bakar Md.; SULAIMAN, Md. Nasir; MUSTAPHA, Aida a LEONG, K. Y. Crossover and Mutation Operators of Genetic Algorithms. online. International Journal of Machine Learning and Computing. 2017, roč. 7, č. 1, s. 9-12. ISSN 20103700. Dostupné z: https://doi.org/10.18178/ijmlc.2017.7.1.611. [cit. 2024-04-13].

[13] SHARKAWY, Abdel-Nasser. Principle of Neural Network and Its Main Types: Review. online. Journal of Advances in Applied & Computational Mathematics. 2020, č. 7, s. 8-19. ISSN 2409-5761/20. Dostupné z: https://avantipublisher.com/index.php/jaacm/article/view/851/502?fbclid=IwAR3QKq8CpvAEQVhJC_YSZKco5YkbmpyFkFyuXZGdsfK6Qy_A5fZtpGfr9hU_aem_Afv0KxNAOWdl2jlFLKpzK_wYylntuU9VuKeDLiZqQPusNQFzLy7gEAyx5DhzJfBl8j47miedzuoDv7UkSQ4hHHIA. [cit. 2024-04-15].

[14] ŠNOREK, Miroslav. Neuronové sítě a neuropočítače. Praha: Vydavatelství ČVUT, 2002. ISBN 80-010-2549-7.

[15] Neural Networks. online. In: IBM Cloud Education. 2020. Dostupné z: https://www.ibm.com/topics/neural-networks. [cit. 2024-02-17].

[16] AMAZON WEB SERVICES, INC. Recurrent Neural Network. online. In: AMAZON WEB SERVICES, INC. AWS. Dostupné z: https://aws.amazon.com/what-is/recurrent-neural-network/. [cit. 2024-04-13].

[17] STTAR ISMAIL WDAA, Abdul. Differential evolution for neural networks learning enhancement. online. Journal of University of Anbar for Pure Science. 2011, roč. 5, č. 2, s. 79-84. ISSN 2706-6703. Dostupné z: https://doi.org/10.37652/juaps.2011.44119. [cit. 2024-03-05].

[18] KNOESTER, Dave. EALib. online. In: GITHUB, INC. Github. 2008. Dostupné z: https://github.com/dknoester/ealib. [cit. 2024-02-26].

[19] MOHAMMADI, Arash; ASADI, Houshyar; MOHAMED, Shady; NELSON, Kyle a NAHAVANDI, Saeid. OpenGA, a C Genetic Algorithm Library. online. 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC). 2017, s. 2051-2056. ISBN 978-1-5386-1645-1. Dostupné z: https://doi.org/10.1109/SMC.2017.8122921. [cit. 2024-01-24].

[20] MASSACHUSETTS INSTITUTE OF TECHNOLOGY. Galib. online. 1995, 1999. Dostupné z: http://lancet.mit.edu/ga/. [cit. 2024-04-03].

[21] DREO, Johann; LIEFOOGHE, Arnaud; VEREL, Sébastien; SCHOENAUER, Marc; MERELO, Juan J. et al. Paradiseo: from a modular framework for evolutionary computation to the automated design of metaheuristics. online. Proceedings of the Genetic and Evolutionary Computation Conference Companion. 2021, s. 1522-1530. ISBN 9781450383516. Dostupné z: https://doi.org/10.1145/3449726.3463276. [cit. 2024-03-01].

[22] NISSEN, Steffen. Implementation of a Fast Artificial Neural Network Library (fann). Graduate project report. Copenhagen: Department of Computer Science University of Copenhagen (DIKU), 2003. Dostupné také z: http://leenissen.dk/fann/wp/2015/11/fann-in-research/.

[23] KING, Davis E. Dlib-ml: A Machine Learning Toolkit. online. Journal of machine learning research. 2009, č. 10, s. 1755-1758. ISSN 1533-7928. Dostupné z: https://doi.org/1755-1758. [cit. 2024-03-02].

[24] THE LINUX FOUNDATION. PyTorch C++ API. online. In: THE LINUX FOUNDATION. PyTorch. Dostupné z: https://pytorch.org/cppdocs/. [cit. 2024-03-02].

[25] MALLET, Olivier. Galgo 2.0. online. In: GITHUB, INC. Github. 2008. Dostupné z: https://github.com/olmallet81/GALGO-2.0. [cit. 2024-02-23].

Seznam obrázků

Seznam tabulek

Seznam příloh

Elektronické přílohy ve formátu archivu .zip jsou:

- GALGO-2.0-master.zip – stažený balík knihovny Galgo (původní knihovna)
- Galgo_example.zip – aplikace demonstrující využití knihovny Galgo – řešení logického problému (projektový soubor, zdrojová kód, upravená knihovna Galgo, příklad výstupu, výsledky jednotlivých příkladů programu)