

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA STROJNÍHO INŽENÝRSTVÍ
ÚSTAV AUTOMATIZACE A INFORMATIKY

FACULTY OF MECHANICAL ENGINEERING
INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

GENETICKÉ ALGORITMY – MULTI-CORE CPU IMPLEMENTACE

GENETIC ALGORITHMS - MULTI CORE CPU IMPLEMENTATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. VLADIMÍR STUDNIČKA
AUTHOR

VEDOUCÍ PRÁCE — Ing. RADOMIL MATOUŠEK, Ph.D.
SUPERVISOR

BRNO 2010

ZADÁNÍ ZÁVĚREČNÉ PRÁCE

(na místo tohoto listu vložte originál, nebo kopii zadání Vaší práce)

LICENČNÍ SMLOUVA

(na místo tohoto listu vložte vyplněný a podepsaný list formuláře licenčního ujednání)

ABSTRAKT

Cílem diplomové práce je vytvořit co možná nejuniverzálnější knihovnu pro genetické algoritmy v jazyce C++, s určitým počtem implementovaných univerzálních operátorů a následně vytvořenou knihovnu otestovat na příkladech. Musí být implementována podpora více-jádrových procesorů pomocí OpenMP. Knihovna bude testována celkově na třech příkladech. První dva příklady jsou matematické funkce, které se používají právě k testování genetických algoritmů. Dalším testovacím příkladem je problém rozložení n-dam na šachovnici, aby se vzájemně neohrožovali. Nakonec se pokusíme pomocí navrhnutých algoritmů zjistit řešení puzzle s názvem Eternity II, za jehož vyřešení je vypsána odměna 2 milióny dolarů.

ABSTRACT

This diploma thesis deals with creating the most universal library of genetic algorithms in C++, as much as possible, implemented with the certain number of universal operators, and then with testing created library on some examples. Library must support multi-core processors, implementation will be done over OpenMP. The library will be tested on three examples in all. The first two examples are mathematical functions, that are used just for genetic algorithms testing. Last problem for test is N-Queens problem. Finally we will use genetic algorithms to try find solution for Eternity II puzzle, there is declared a 2 million bounty for full solution.

KLÍČOVÁ SLOVA

Genetický algoritmus, hodnotící funkce, metody výběru, křížení, mutace, migrace, znovuvložení, populace, jedinec, chromozóm, minimum funkce, n-dam, Eternity II, OpenMP.

KEYWORDS

Genetic algorithm, fitness function, selection, crossover, mutation, migration, reinsertion, population, individual, chromosome, function minimum, N-Queens, Eternity II, OpenMP.

PODĚKOVÁNÍ

Tímto děkuji svému vedoucímu diplomové práce Ing. Radomilu Matouškovi, Ph.D. za jeho rady, připomínky, odbornou pomoc a věnovaný čas, který mi věnoval při vytváření diplomové práce.

Obsah

1 Teoretický úvod do genetických algoritmů 13

1.1 Základní pojmy genetických algoritmů 14

1.2 Etapy návrhu genetického algoritmu 15

1.3 Kódování 16

1.4 Hotnotící funkce (Fitness Function) 18

2 OpenMP 19

2.1 Programovací model OpenMP 19

2.2 Vrstvy OpenMP 19

2.3 OpenMP API 20

2.3.1 Formát direktiv 20

2.3.2 Direktiva parallel 20

2.3.3 Paralelizace for cyklů 21

2.3.4 Direktiva sections 21

2.3.5 Bloky pro jedno vlákno 22

2.3.6 Funkce knihovny a systémové proměnné 22

3 Implementace knihovny genetických algoritmů 23

3.1 Detailní popis základních objektů 24

3.2 Třída populace 30

3.2.1 Vytvoření nové generace EvolveNewGeneration() 31

3.2.2 Konfigurace a statistiky populace 31

3.2.3 Definování populace a jedince 32

3.3 Třída evoluce 33

3.3.1 Konfigurace GA 33

3.3.2 Definování evoluce a použití GA 34

3.4 Implementované metody, operátory a reprezentace 35

3.5 Využití OpenMP v knihovně GA 37

3.5.1 OpenMP Level 1 37

3.5.2 OpenMP Level 2 38

3.5.3 Paralelizované objekty 38

4 Popis implementovaných genetických operátorů 41

4.1 Metody výběru (Selection) 41

4.1.1 Ruletový výběr (Roulette Wheel Selection) 41
4.1.2 Stochastický univerzální výběr (Stochastic Universal Sampling) 43
4.1.3 Seříznutý výběr (Truncation Selection) 44
4.1.4 Turnajový výběr (Tournament Selection) 44
4.1.5 Elitní turnajový výběr (Elite Tournament Selection) 45
4.1.6 Elitismus (Elitism) 45
4.2 Operátory křížení 46
4.2.1 Binární operátory křížení (Binary Valued Crossover) 46
4.2.1.1 Jednobodové křížení (Singlepoint Crossover) 46
4.2.1.2 Dvoubodové a vícebodové křížení (Double-point & Multipoint Crossover) 46
4.2.1.3 Uniformní křížení (Uniform Crossover) 47
4.2.2 Křížení při použití permutačního kódování 47
4.2.2.1 Křížení s částečným přiřazením - PMX (Partially Mapped Crossover) 48
4.2.2.2 Křížení s rekombinací hran - ERX (Edge Recombination Crossover) 48
4.2.3 Křížení při využití kódování s reálnými hodnotami (Real Valued Crossover) 51
4.2.3.1 Střední křížení (Intermediate Recombination) 51
4.2.3.2 Liniové křížení (Line Recombination) 52
4.2.3.3 Rozšířené liniové křížení (Extended Line Recombination) 53
4.3 Operátory mutace 55
4.3.1 Binární operátory mutace 55
4.3.2 Reálné a celočíselné operátory mutace 56
4.3.3 Výměnná mutace (Swap Mutation) 56
4.3.4 Posuvná mutace (Shift mutation) 57
4.3.5 Inverzní mutace (Inverse Mutation) 57
4.4 Doplnění, znovuvložení (Reinsertion) 58
4.5 Paralelní genetické algoritmy 61
4.5.1 Vícepopulační migrační model (Multipopulation Migration Model) 61
4.5.2 Hardwérově paralelizovaný model (Hardwadre Parallelized Model) 63
4.5.3 Migrace při použití hardwérově paralelizovaného modelu 65
5 Testování vytvořené knihovny genetických algoritmů 69
5.1 Rosenbrockova funkce (Rosenbrock’s Function) 70

5.1.1 Vizualizace funkce 70
5.1.2 Analýza použití genetického algoritmu ke hledání globálního minima 72
5.1.3 Test výkonu OpenMP, při různém nastavení počtu populací a migrace 72
5.1.4 Test rychlosti nalezení řešení v závislosti na počtu dimenzí 74
5.1.5 Test rychlosti nalezení řešení v závislosti na velikosti populace 76
5.2 Rastriginova funkce více proměnných (Rastrigin's Function) 78
5.2.1 Vizualizace funkce 78
5.2.2 Analýza použití genetického algoritmu ke hledání globálního minima 80
5.2.3 Test výkonu OpenMP, při různém nastavení počtu populací a migrace 80
5.2.4 Test rychlosti nalezení řešení v závislosti na počtu dimenzí 82
5.2.5 Test rychlosti nalezení řešení v závislosti na velikosti populace 83
5.3 Problém N dam na šachovnici (N-Queens Problem) 84
5.3.1 Analýza použití genetického algoritmu k řešení problému N dam 85
5.3.2 Test výkonu OpenMP, při různém nastavení počtu populací a migrace 86
5.3.3 Test rychlosti nalezení řešení v závislosti na rozměru šachovnice 88
5.3.4 Test rychlosti nalezení řešení v závislosti na velikosti populace 89
6 Eternity II 91
6.1.1 Analýza použití genetického algoritmu k řešení Eternity II 92
6.1.2 Výsledky hledání řešení Eternity II 95
7 Závěr 97
8 Použitá literatura 99

1 Teoretický úvod do genetických algoritmů

Na počátku 60. let vznikl nový směr v rozvoji informatiky, a to genetické algoritmy. Genetické algoritmy vycházejí z Darwinovy teorie o vývoji druhu. V roce 1975 John Henry Holland popsal v knize *Adaptation in Natural and Artificial Systems* základní vlastnosti těchto algoritmů a obecně se považuje za jejich zakladatele.

Genetický algoritmus je heuristický postup, který se snaží aplikací principů evoluční biologie nalézt řešení složitých problémů, pro které neexistuje použitelný exaktní algoritmus. Genetické algoritmy, resp. všechny postupy patřící mezi tzv. evoluční algoritmy, používají techniky napodobující evoluční procesy známé z biologie jako je dědičnost, mutace, přirozený výběr, křížení a další pro zlepšení řešení zadané úlohy.

Genetické algoritmy jsou nejrozšířenějším typem evolučních algoritmů. Evoluční algoritmy globálně dělíme následovně na:

- Genetické algoritmy
- Genetické programování
- Evoluční strategie
- Evoluční programování

Princip práce genetického algoritmu je postupná tvorba generací různých řešení daného problému. Při řešení se uchovává tzv. populace, jejíž každý jedinec představuje jedno řešení daného problému. Jak populace probíhá evolucí, řešení se zlepšují. Tradičně je řešení reprezentováno binárními čísly, řetězci nul a jedniček, nicméně používají se i jiné reprezentace (celočíselná, reálná, permutační, strom, pole, matice, vektor, …). Typicky je na začátku simulace (v první generaci) populace složena z naprosto náhodných členů. V přechodu do nové generace je pro každého jedince spočtena tzv. fitness funkce, která vyjadřuje kvalitu řešení reprezentovaného tímto jedincem. Podle této kvality jsou stochasticky vybráni jedinci, kteří jsou modifikováni (pomocí mutací a křížení), čímž vznikne nová populace. Tento postup se iterativně opakuje, čímž se kvalita řešení v populaci postupně vylepšuje. Algoritmus se obvykle zastaví při dosažení postačující kvality řešení, případně po předem dané době, nebo pokud již nedochází k vylepšení ohodnocení pomocí fitness funkce.

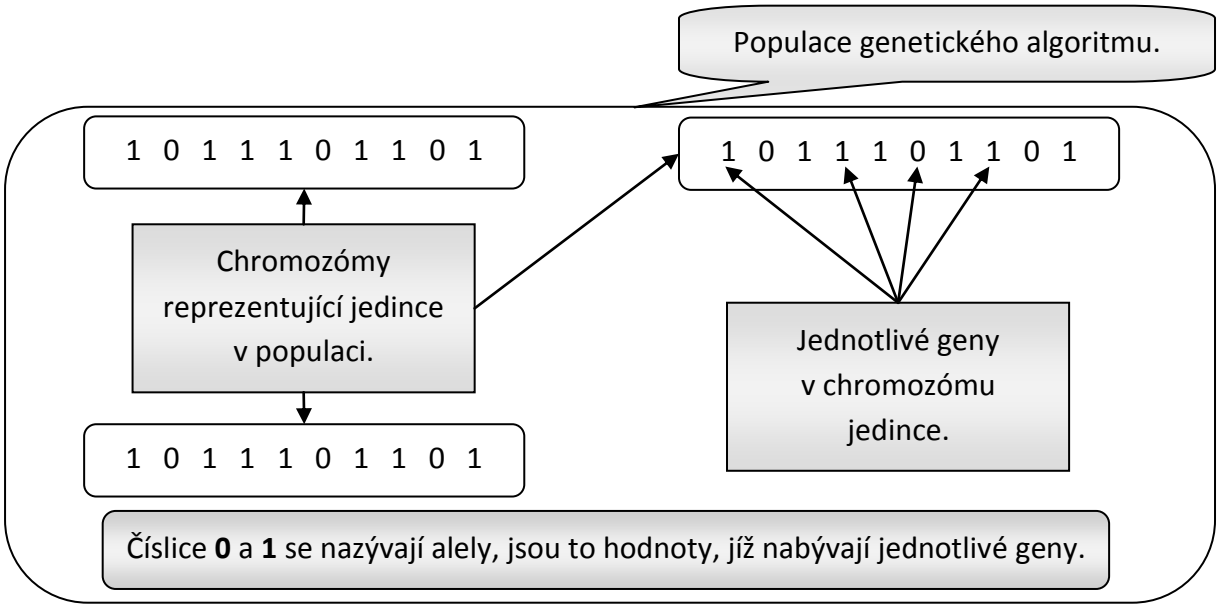
Převaha genetických algoritmů se projevuje u složitých úkolů. Obecně je možno říci, že se jedná o rozsáhlé nelineární problémy rozhodování za velké neurčitosti s mnoha faktory ovlivňujícími výsledek řešení. Tedy takové problémy, kde klasické metody neznají algoritmus řešení nebo jsou pro datovou náročnost nepoužitelné.

Jeden z důvodů síly genetických algoritmů je v samotné podstatě algoritmu hledání řešení, které spočívá v tom, že nehledáme nějaké určité řešení, ale skupinu přípustných řešení, z nichž vybíráme to nejlepší. Od klasických metod se genetické algoritmy odlišují způsobem, kterým se přibližujeme k řešení. Převaha genetických algoritmů se projevuje ve složitých problémech s mnoha parametry a NP-úplných úlohách. Podrobněji v [03].

1.1 Základní pojmy genetických algoritmů

Genotyp (Genotype)	:	obecná genetická informace
Fenotyp (Phenotype)	:	zobrazení genetické informace do konkrétního stavu
Jedinec (Individual)	:	nositel genetické informace
Chromozóm (Genome)	:	obecná genetická informace ve tvaru řetězce, je to sekvence symbolů z nějaké abecedy (mohou to být čísla, znaky nebo jejich kombinace)
Gen (Gene)	:	určité místo (pozice) v chromozómu
Alela (Allele)	:	konkrétní symbol v chromozómu, kterého nabývá gen
Rodič (Parent)	:	jedinec vstupující do rekombinace, vybraný metodou selekce
Potomek (Offspring)	:	jedinec, jenž je výsledkem rekombinace dvou nebo více rodičů
Ohodnocení (Fitness)	:	ohodnocení síly jedince v generaci, odvozuje jeho přežití
Křížení (Crossover)	:	konstrukce nových jedinců (potomků) dle vybraných jedinců generace (rodičů)
Mutace (Mutation)	:	změna alel(y) genu(ů) v chromozómu
Rekombinace (Recombination)	:	sled křížení a mutace
Selekce (Selection)	:	výběr jedinců, kteří přežívají v generaci nebo jsou určeny pro křížení
Generace (Generation)	:	skupina jedinců, na kterou je aplikována teorie o vývoji druhů
Populace (Population)	:	obecné vyjádření pro sled generací
Migrace (Migration)	:	předávání jedinců mezi různými populacemi
Doplnění, znovuvložení (Reinsertion)	:	přechod jedince ze starší generace do nové, bez procesu křížení a mutace
Schéma (Scheme)	:	prototyp chromozómu, mající určené některé geny

Tabulka 1.1: Základní pojmy genetických algoritmů.

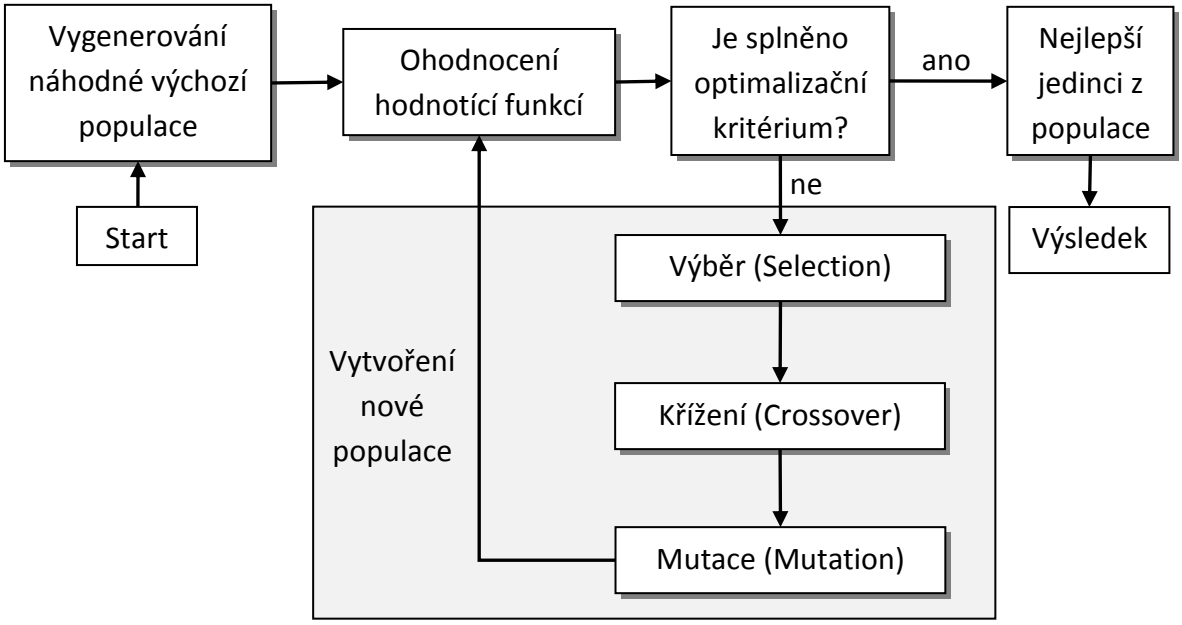


Obrázek 1.1: Zobrazení binární populace, chromozómu, genů a alel.

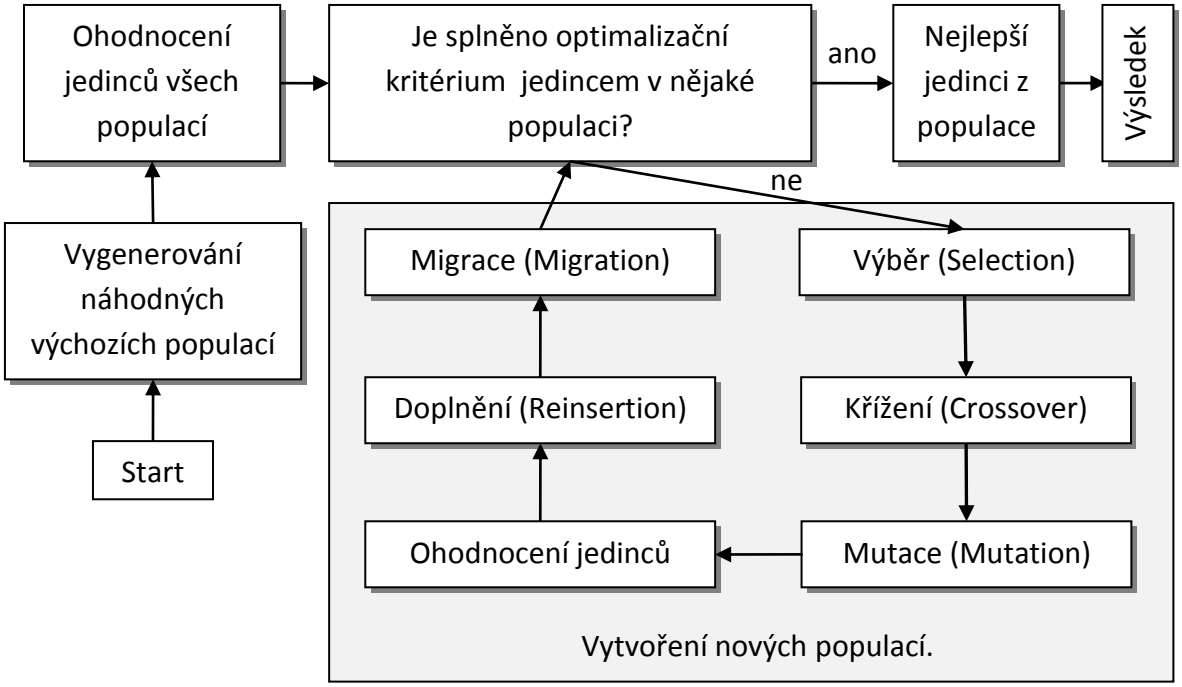
1.2 Etapy návrhu genetického algoritmu

Existuje mnoho různě formulovaných definic genetického algoritmu. Liší se zejména ve způsobu vytváření nové populace. Následuje obecné schéma, které je ve většině případů zcela stejné [03].

1) Reprezentace problému a vynulování hodnoty počítadla generací (*t = 0*).
2) Získání počáteční populace (většinou náhodným generováním).
3) Výpočet ohodnocení (*fitness*) každého individua v počáteční populaci *P(0)*.
4) Výběr dvojice individuí z populace *P(t)* a vytvoření jejich potomků *P'(t)*.
5) Ohodnocení nově vytvořených individuí v *P'(t)*.
6) Vytvoření nové populace *P(t+1)* z původní populace *P(t)* a množiny potomků *P'(t)*.
7) Zvětšení hodnoty počítadla generaci o jedničku (*t++*).
8) Výpočet ohodnocení (*fitness*) každého individua v populaci *P(t)*.
9) Pokud *t* je rovno maximálnímu počtu generací, nebo je splněno jiné ukončovací kritérium, tak se vrátí jako výsledek populace *P(t)*, jinak se pokračuje krokem číslo 4.



Obrázek 1.2: Diagram postupu jednoduchého genetického algoritmu.

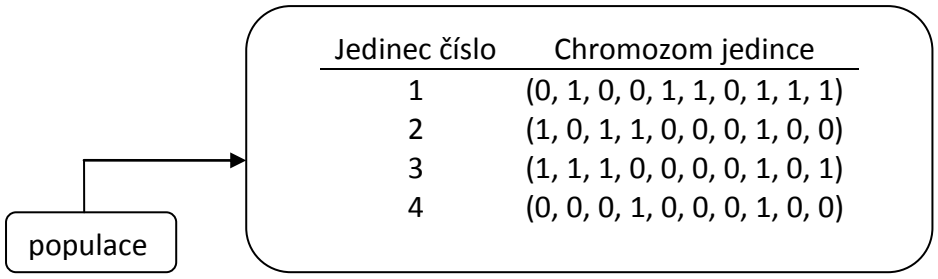


Obrázek 1.3: Diagram postupu vícepopulačního genetického algoritmu s migrací a znovuvložením.

U vícepopulačního genetického algoritmu máme navíc doplnění o migraci. Díky migraci se mohou jedinci mezi populacemi navzájem prohazovat. Doplnění neboli znovuvložení se využívá i u jednoduchého genetického algoritmu, ale bylo vynecháno pro zjednodušení.

1.3 Kódování

Vlastní způsob, jakým jsou jedinci geneticky popsáni, je velmi důležitý pro úspěch či neúspěch genetického algoritmu na konkrétní úloze. Způsob kódování bude pro názornost naznačen v binárním kódu. Binární kódování je historicky nejstarší a z celé řady důvodů je stále jedním z nejpoužívanějších. Při binárním kódování může každý gen v chromozómu nabývat hodnoty 0 nebo 1, přičemž tyto hodnoty jsou jednotlivé alely genu (stavy v níž se gen může nacházet). Pro názornost je vše zobrazeno níže, na obrázku Obrázek 1.4, pro 4 jedince s délkou chromozómů 10, kteří dohromady tvoří populaci [03].



Jedinec číslo	Chromozom jedince
1	(0, 1, 0, 0, 1, 1, 0, 1, 1, 1)
2	(1, 0, 1, 1, 0, 0, 0, 1, 0, 0)
3	(1, 1, 1, 0, 0, 0, 0, 1, 0, 1)
4	(0, 0, 0, 1, 0, 0, 0, 1, 0, 0)

Obrázek 1.4: Binární reprezentace chromozómů 4 jedinců jedné populace.

Binární kód si však s sebou nese jednu velice nepříjemnou skutečnost. Často se předpokládá, že malá změna vlastností jedince se projeví nepatrnou změnou v jeho

chromozómu a naopak, že nepatrná změna chromozómu (například mutací) se projeví jen malou změnou jeho fenotypu. V případě binárního kódování je tento předpoklad velmi silně narušen. Dojde-li, například díky mutaci, k inverzi bitu na vysoké pozici chromozómu určujícího vzdálenost objektu od počátku nějaké soustavy souřadnic, tak se oba jedinci liší třeba pouze v jednom genu, ale o numericky velkou vzdálenost [03].

Tuto nevýhodu standardního binárního kódu řešíme použitím Grayova kódu, jehož základní vlastností je, že dvě sousední hodnoty zakódovány binárními řetězci příslušné délky se liší právě v jednom bitu. Proto je Grayův kód často upřednostňován před kódem binárním. Srovnání těchto dvou kódů (4 bitových) je v tabulce Tabulka 1.2 zcela objasněno.

číslo	Binární kód	Grayův kód	číslo	Binární kód	Grayův kód
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

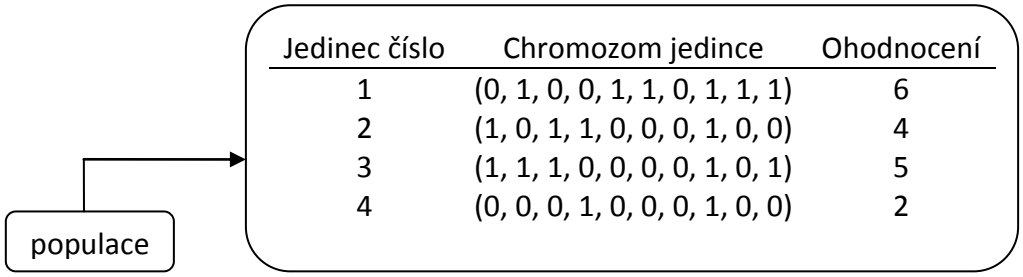
Tabulka 1.2: Porovnání binárního a Grayova kódu.

I přes jednoduchost binární abecedy je pro mnohé praktické aplikace výhodnější použít abecedu s více než dvěma znaky a někde přímo reálná čísla. Bylo možné očekávat, že genetické algoritmy budou účinnější při práci s binární abecedou, avšak v [33] byla prokázána výhoda používání víceznakových abeced a reálných čísel pomocí četných empirických porovnání. Reprezentace používající reálná čísla se jeví jako velmi výhodná, zejména při řešení úloh, kde je vyžadována velká přesnost a binární reprezentace by znamenala použít příliš dlouhé řetězce, což s sebou přináší vyšší nároky na paměť a čas zpracování [03].

Při řešení různých kombinatorických a plánovacích úloh se nejčastěji používá permutační kódování, kdy je chromozom jedince reprezentován permutací několika čísel či znaků. Permutace určuje pořadí jednotlivých objektů v řešení konkrétního problému a cílem je nalézt permutaci reprezentující optimální řešení.

1.4 Hotnotící funkce (Fitness Function)

Hodnotící funkce musí být vždy úzce svázána s konkrétním problémem, který chceme pomocí genetického algoritmu řešit. Pokud máme přímo zadanou nějakou účelovou funkci, na které chceme hledat minimum či maximum, pak můžeme jako ohodnocující funkci použít přímo tuto účelovou. Ohodnocující funkce přiřazuje chromozómům jednotlivých jedinců ohodnocení, podle kterého jsou následně vybíráni vhodní jedinci pro křížení. Pro jednoduchost uvedeme příklad ohodnocující funkce na populaci z obrázku Obrázek 1.4, kde každý chromozóm jedince získá právě takové ohodnocení, kolik je v něm jedniček. Vše je uvedeno na následujícím obrázku.



Jedinec číslo	Chromozom jedince	Ohodnocení
1	(0, 1, 0, 0, 1, 1, 0, 1, 1, 1)	6
2	(1, 0, 1, 1, 0, 0, 0, 1, 0, 0)	4
3	(1, 1, 1, 0, 0, 0, 0, 1, 0, 1)	5
4	(0, 0, 0, 1, 0, 0, 0, 1, 0, 0)	2

Obrázek 1.5: Ohodnocení chromozómů jedinců z obrázku Obrázek 1.4.

Ohodnocení může probíhat například podle účelové funkce, jak již bylo řečeno, ale i mnoha jinými způsoby. V některých případech, kde je velká populace, se může používat systém, kde se populace rozdělí na dvě podpopulace a vzájemně se porovnávají podpopulace mezi sebou, přičemž jedinci dostanou ohodnocení podle vzájemného porovnávání [03].

Jinou cestou je uspořádání turnaje mezi jedinci, přičemž vždy lepší jedinec z náhodně vybrané dvojice postupuje do dalšího kola, kde se utká s jiným, náhodně vybraným vítězem předchozího kola. Na konci turnaje máme vítězného jedince a ostatním jedincům se přiřadí hodnota odpovídající jejich působení v turnaji. V turnajovém případě stačí provést *N-1* porovnání (kde *N* je velikost populace). Tato metoda s sebou přináší riziko přílišného zjednodušení, kterým platíme za velmi malý počet provedených porovnání.

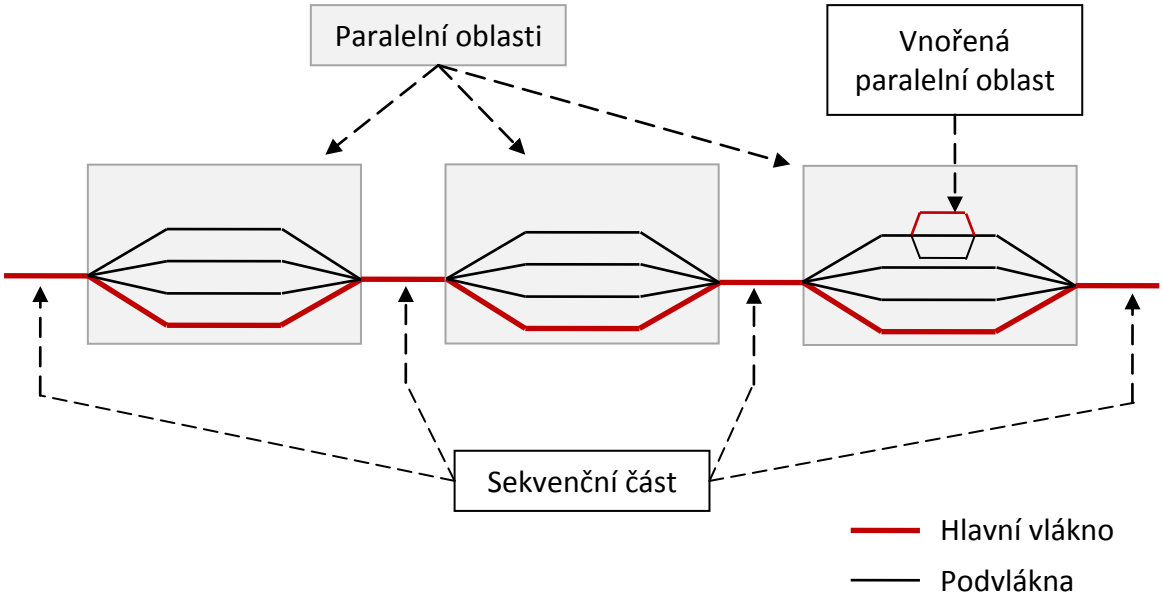
Často úspěšným bývá přístup zvaný „síň slávy“, kde ohodnocovaný jedinec podstupuje srovnání s jedinci z předchozích generací, kteří postupně vstupují do testovací množiny (síně slávy), a tím pádem představují náročné konkurenty [03].

2 OpenMP

OpenMP je soustava direktiv pro překladač a knihovních procedur pro paralelní programování. Jedná se o standard pro programování počítačů se sdílenou pamětí. OpenMP usnadňuje vytváření vícevláknových programů v programovacích jazycích Fortran, C a C++.

2.1 Programovací model OpenMP

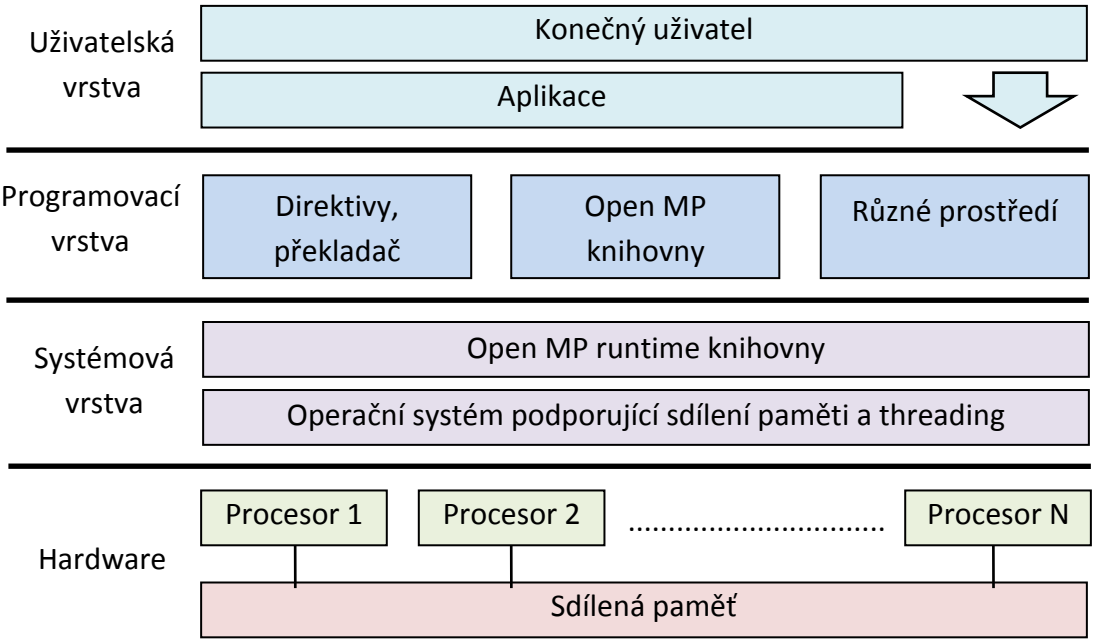
Hlavní vlákno (master thread) vytváří podle potřeby skupinu podvláken (subthread). Paralelizace programu se pak provádí postupně s ohledem na výkon aplikace, tj. sekvenční program je postupně (podle možností) pararelizován.



Obrázek 2.1: Hlavnívlákno a podvlákna OpenMP.

2.2 Vrstvy OpenMP

Uživatel přistupuje k OpenMP prostřednictvím aplikace, napsané pomocí programovací vrstvy. Ta při použití direktiv a API funkcí knihovny OpenMP ovládá run-time knihovnu, které je implementována na konkrétní druh operačního systému. Jinými slovy, programovací vrstva je platformě nezávislá, a týká se pouze programátora který OpenMP používá. Kdežto systémová vrstva je závislá na platformě a využívá konkrétní threading system. Samozřejmě nejnižší vrstva pod systémovou je hardware. Která nejvíce ovlivňuje rychlost paralelizace, tedy počtem jader a jejich taktem.



Obrázek 2.2: Schéma OpenMP.

2.3 OpenMP API

Aplikační programové rozhraní podporuje multiplatformní paralelní programování v C/C++ a Fortranu na všech architekturách pro počítače se sdílenou pamětí.

2.3.1 Formát direktiv

Open MP direktivy pro C a C++ jsou specifikovány direktivou preprocesoru pragma. Každá direktiva začíná jako #pragma omp. Direktivy jsou case-sensitive a týkají se následujícího strukturovaného bloku. Direktiva parallel způsobí, že následující blok instrukcí bude zpracováván více vlákny.

```
#pragma omp název_direktivy [ klauzule [[,] klauzule ]... ]
```

```
#pragma omp parallel [ klauzule ]
{
  // paralelní blok
}
```

Pokud chceme vkládat direktivu parallel do jiné direktivy parallel musí být systémová proměnná *OMP_NESTED* nastavena na hodnotu *TRUE*.

2.3.2 Direktiva parallel

Pomocí *[kaluzule]* lze zadat následující:

- podmínka paralelizace: *if (...)*, pouze jedna.
- počet vláken: *num_threads (int)*
- zacházení s daty:
  - *private* (seznam proměnných): určuje lokální proměnné, každé vlákno má svou vlastní kopii.

- *firstprivate* (seznam proměnných): stejně jako u private, ale u každé kopie se nastaví hodnota, kterou měla proměnná před forkováním běhu programu na vlákna.
- *shared* (seznam proměnných): tyto proměnné jsou sdíleny mezi vlákny.
- *reduction* (operátor: seznam proměnných): dané proměnné budou mít lokální kopie a nakonec se provede redukce pomocí asociativní operace +, *, &, |, &&, ||.
- default (shared, nic): výchozí hodnota

Pomocné funkce:

- *omp_get_num_threads()*: vrací počet vláken.
- *omp_get_thread_num()*: vrací celočíselný identifikátor vlákna

2.3.3 Paralelizace for cyklů

Po spuštění více vláken je nutné zadat, co mají jednotlivá vlákna provádět. Pokud všechna vlákna provádějí stejnou úlohu tak se dělí o *for* cyklus. V případě že každé vlákno provádí jinou úlohu tak zpracovávají sekce (*sections*) různého kódu. Paralelizace *for* cyklů:

```
#pragma omp for [ klauzule [[,] klauzule]... ]
```

Klauzule pro ošetření proměnných:

- *private* (seznam)
- *firstprivate* (seznam)
- *reduction* (operátor: seznam)
- *lastprivate* (seznam): hodnota proměnné je nastavena v posledním průběhu cyklu.
- ordered
- nowait: Standardně se nezačíná nový *for* cyklus, doku všechna vlákna neskončila práci na předchozím cyklu (bariéra mezi cykly). Pokud to není nutné lze použít klauzuli *nowait.*
- *schedule* (třída[, parametr]):
  - *static*: každé vlákno postupně dostává stejný počet iterací, které jsme zadali jako parametr. Není-li parametr uveden, jsou všechny iterace rozděleny na *n* stejných částí, kde *n* je počet vláken
  - *dynamic*: funguje podobně jako statické přidělování iterací. Nové iterace jsou ale přidány vláknu, které skončí svou práci jako první. Některá vlákna tak mohou provést více iterací než ostatní.
  - *guided*: s každým přidělením nových iterací exponenciálně zmenšuje parametr, ten udává dolní mez pro počet přidělených iterací.
  - runtime: podle systémové proměnné OMP_SCHEDULE se určí která z vyšších tří tříd se má použít.

2.3.4 Direktiva sections

Pomocí direktivy sections se provádí zpracování různých úloh každým vláknem.

```
#pragma omp parallel
{
  #pragma omp sections
  {
       #pragma omp section
       { Úloha1(); }
       #pragma omp section
       { Úloha2(); }
  }
}
```

Klauzule pro ošetření proměnných:

- *private* (seznam)
- *firstprivate* (seznam)
- *lastprivate* (seznam)
- *reduction* (operátor: seznam)
- *nowait*

2.3.5 Bloky pro jedno vlákno

Pokud chceme aby blok byl zpracováván jen jedním vláknem použijeme direktivu *single*. Pokud není uvedeno *nowait*, tak ostatní vlákna čekají na konci bloku.

```
#pragma omp single { ... }
```

Následující blok bude zpracováván je jen s vláknem ID = 0, ostatní vlákna nečekají.

```
#pragma omp master { ... }
```

Kritické bloky obsahují kód, který může současně provádět jen jedno vlákno. například částečné úlohy pro jednotlivá vlákna lze distribuovat pomocí centrální struktury (fronty). Přístup k ní pak může mít v daný okamžik jen jedno vlákno.

```
#pragma omp critical [(jméno)]
```

2.3.6 Funkce knihovny a systémové proměnné

Základní funkce:

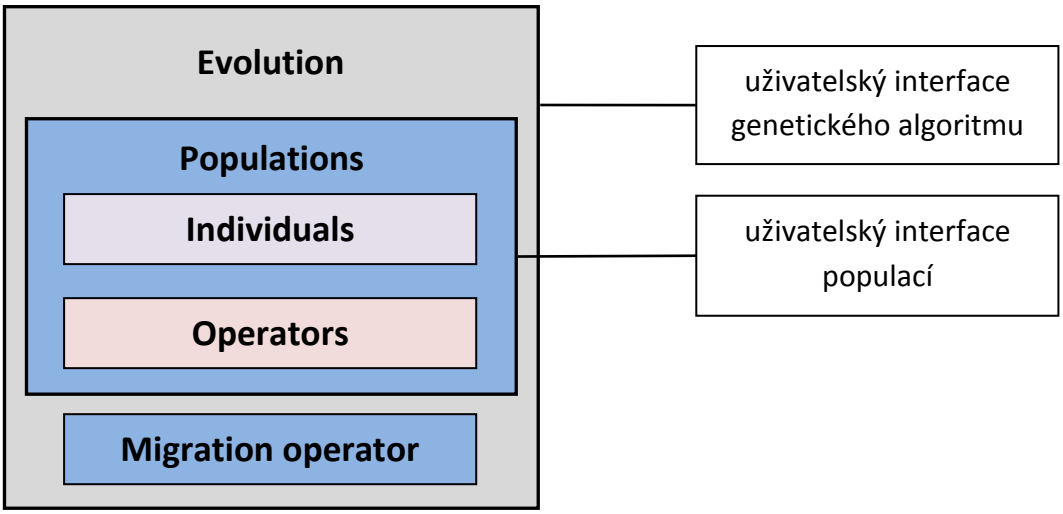
- omp_set_num_threads()
- omp_get_num_threads()
- omp_get_max_threads()
- omp_get_thread_num()
- omp_get_num_procs()
- omp_in_parallel()
- omp_set_dynamic()
- omp_get_dynamic()
- omp_set_nested()
- omp_get_nested()

Systémové proměnné:

- OMP_SCHEDULE
- OMP_NUM_THREADS
- OMP_DYNAMIC
- OMP_NESTED

3 Implementace knihovny genetických algoritmů

Cílem této práce bylo zejména vytvořit knihovnu pro jednoduché použití při aplikaci genetických algoritmů. Jelikož základní požadavek GA je rychlost, knihovna a její algoritmy jsou implementovány v jazyce C++, to zejména za pomoci C++ šablon tříd, které nám také zvětšují možnost rozšíření a použití algoritmů na jakékoli obecné problémy. Knihovna je rozdělena do několika bloků obsahujících základní prvky teorie genetických algoritmů. Všechny tyto bloky a veškerá funkcionalita této knihovny je zanořena do namespace GA (tedy například přístup na populaci lze realizovat GA::Population…).



Obrázek 3.1: Blokové schéma.

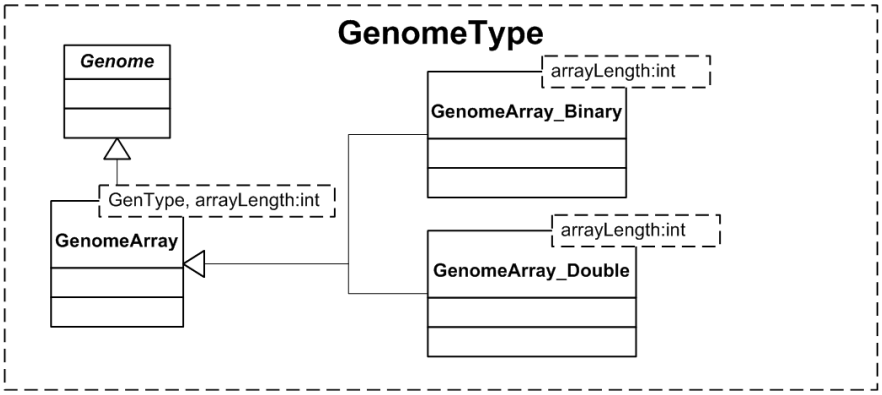
Hlavním objektem knihovny je *Evolution*, který provádí samotnou evoluci a obsahuje užitečné funkce pro práci uživatele s GA. *Evolution* v sobě udržuje populace definované uživatelem, které jsou tvořeny jedinci (Individuals) a jejich operátory. *Evolution* dále obsahuje operátor migrace provádějící migraci nad všemi populacemi. Jednotlivé bloky GA představují operátory a objekty GA, definujeme je v samostatných souborech. Tato knihovna je rozdělena celkem do 15-ti následujících souborů:

- **ga.h** – hlavní hlavičkový soubor vytvořené knihovny genetických algoritmů, deklarace třídy, která provádí evoluci populací
- **ga.inl** – implementace evoluce populací
- **gacommon.h** – deklarace základních objektů genetického algoritmu (vektory, random generátory)
- **gacommon.inl** – implementace šablonových objektů gacommon.h
- **gacommon.cpp** – implementace funkcí gacommon.h (random generátor)
- **gaindividual.h** – deklarace šablon genotypu a jedince
- **gaindividual.inl** – implementace šablon genotypu a jedince
- **gapopulation.h** – deklarace šablony populace a její rozhraní
- **gapopulation.inl** – implementace šablony populace
- **garandom.h** – deklarace rozhraní generátorů jedince a deklarace základních generátorů jedinců

- **gamutation.h** – deklarace rozhraní operátorů mutace jedince a deklarace základních typů operátorů
- **gaselection.h** – deklarace rozhraní operatorů výběru rodičů pro křížení a deklarace základních typů operátorů
- **gacrossover.h** – deklarace rozhraní operátorů křížení a deklarace základních typů operátorů
- **gareinsertion.h** – deklaracerozhraní operatoru doplnění jedinců populace a deklarace základních typů operátoru
- **gamigration.h** - deklarace rozhraní operatorů migrace jedinců mezi populacemi a deklarace základních typů operátorů

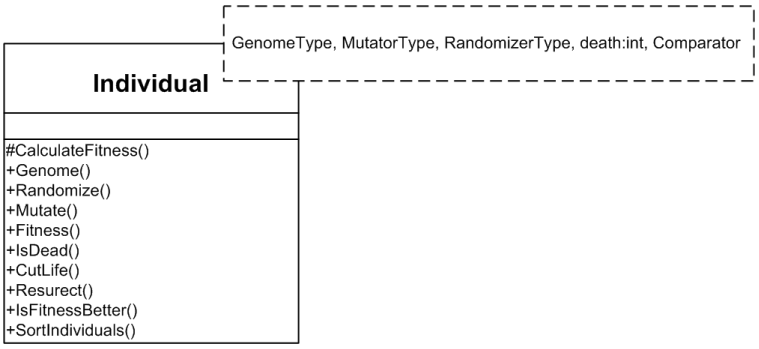
3.1 Detailní popis základních objektů

Nyní se soustředíme na detailnější popis objektů a rozhraní, které tvoří knihovnu GA.



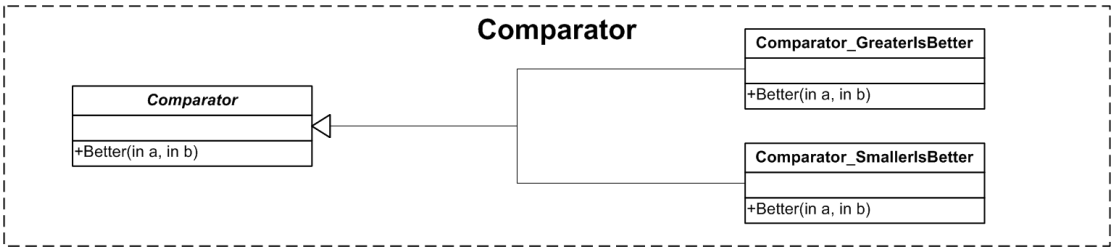
Obrázek 3.2: Typ genomu.

Nejzákladnějším objektem GA je chromozom (Genome). Nejběžnější druh chromozomu je pole genu, v GA definovaný jako *GenomeArray<GenType,arrayLength>*. Tato šablona definuje jednoduchý objekt statického pole. *GenType* vyjadřuje parametr typu genu a *arrayLength* počet genů chromozomu. Mějme například chromozom jako celočíselnou reprezentaci s 10-ti geny. Definujeme jej tedy takto *GenomeArray<int,10>*.



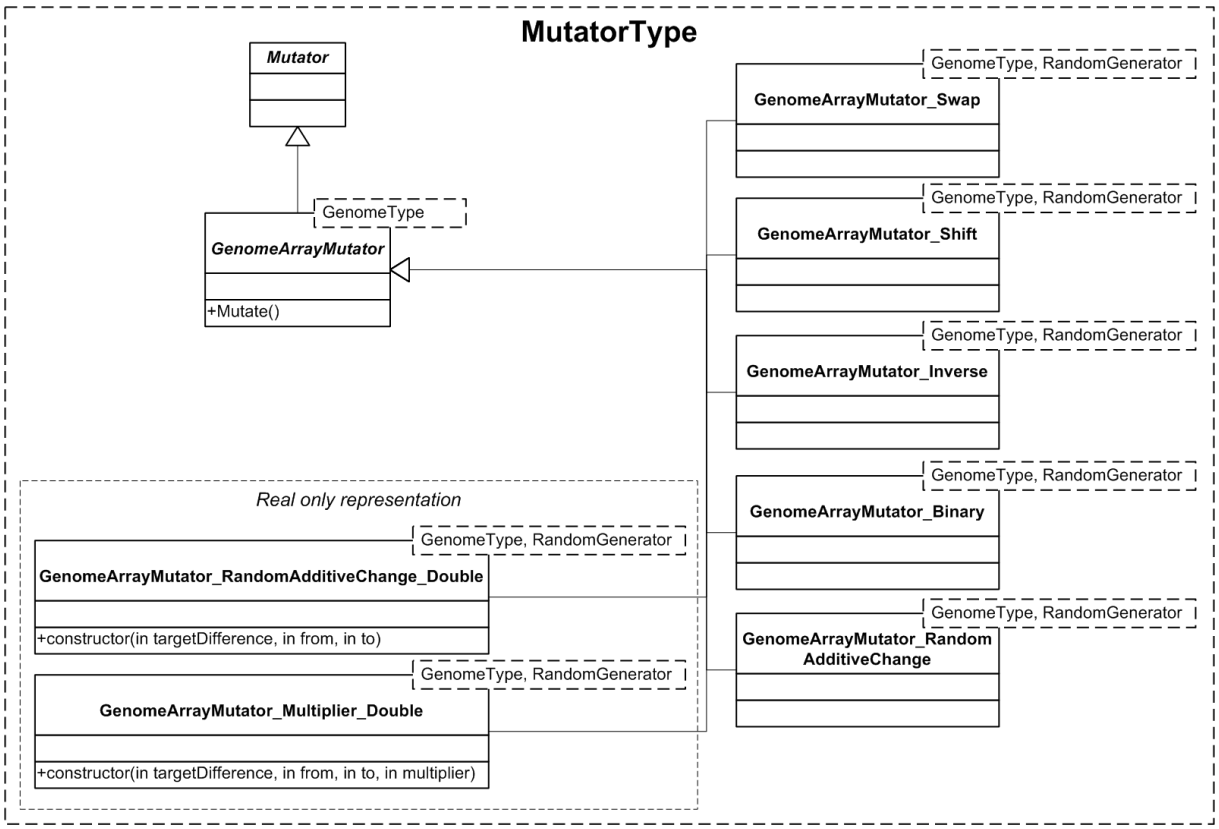
Obrázek 3.3: Jedinec a jeho základní metody.

Objekt Jedinec (Individual) je definován šablonou *Individual<GenomeType, MutatorType, Randomizer, death, Comparator>*. Kde *GenomeType* definuje typ chromozomu (viz. obr. Obrázek 3.2: Typ genomu.) jedince. *MutatorType* definuje operátor mutace jedince. *Randomizer* definuje objekt generující náhodné rozložení genů chromozomu jedince. Parametr *death* určuje životnost jedince.



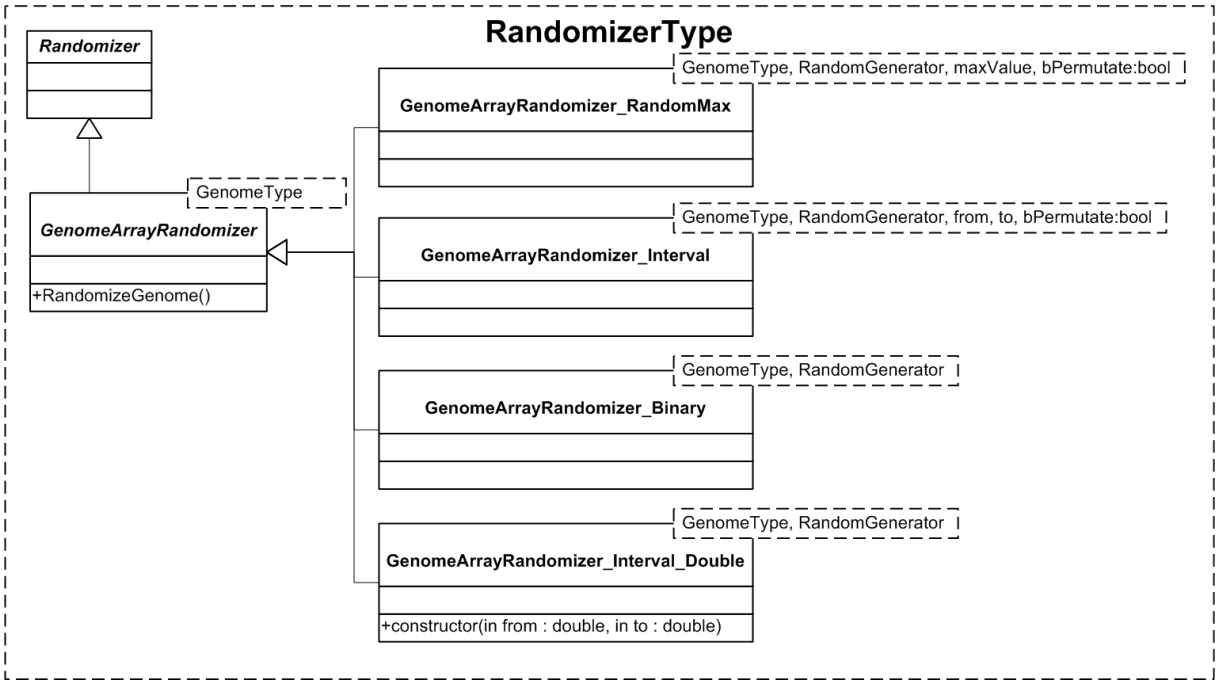
Obrázek 3.4: Rozhraní komparátoru a implementované třídy.

Poslední parametr, *Comparator*, definuje objekt porovnávající kvalitu dvou jedinců, pomocí tohoto objektu lze GA říct, například čím nižší fitness jedince, tím lépe, nebo naopak. Jsou definovány dva základní objekty komparátoru, *Comparator_GreaterIsBetter* a *Comparator_SmallerIsBetter*, ovšem uživatel si samozřejmě může definovat vlastní, speciální komparátor. Třída *Individual* obsahuje důležitou virtuální funkci *CalculateFitness()*, kterou si musí uživatel naimplementovat pro výpočet fitness funkce samostatného jedince. Detailní definice a funkce jedince jsou k nahlédnutí ve zdrojovém souboru gaindividual.h.



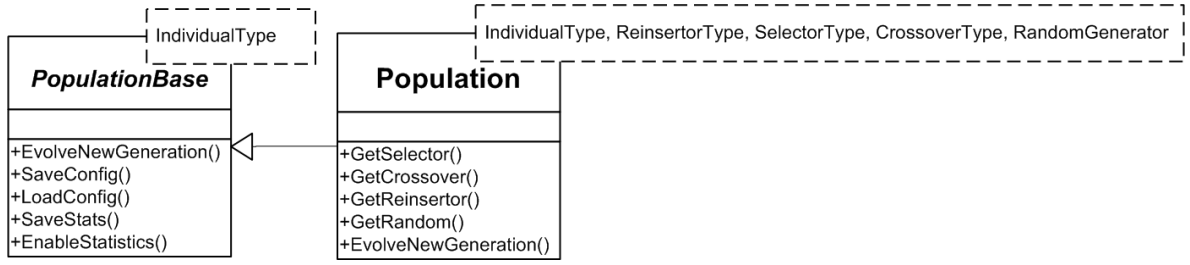
Obrázek 3.5: Rozhraní mutátoru a tříd implementovaných mutací.

Operátor mutace jedince (*Mutator*) je definován v souboru gamutation.h, kde základní objekt mutace chromozomu je definován v šabloně *GenomeArrayMutator<GenomeType>*, kde *GenomeType* vyjadřuje typ chromozomu jedince. *GenomeArrayMutator* definuje funkci *Mutate(GenomeType &genome, int gensMutationPropability)*, která je volána z jedince při požadavku na mutaci. Parametr genome je reference na genom jedince a *gensMutationPropability* procentuální pravděpodobnost mutace jednoho genu (tento parametr je definován v konfiguraci populace a nemusí být použit u všech mutací), některé typy mutací jej ignorují. Konkrétní operátory mutace jedince pouze dědí tuto třídu a implementují funkci *Mutate()*. Ke konkrétním operátorům mutace (i jiným) se dostaneme později.



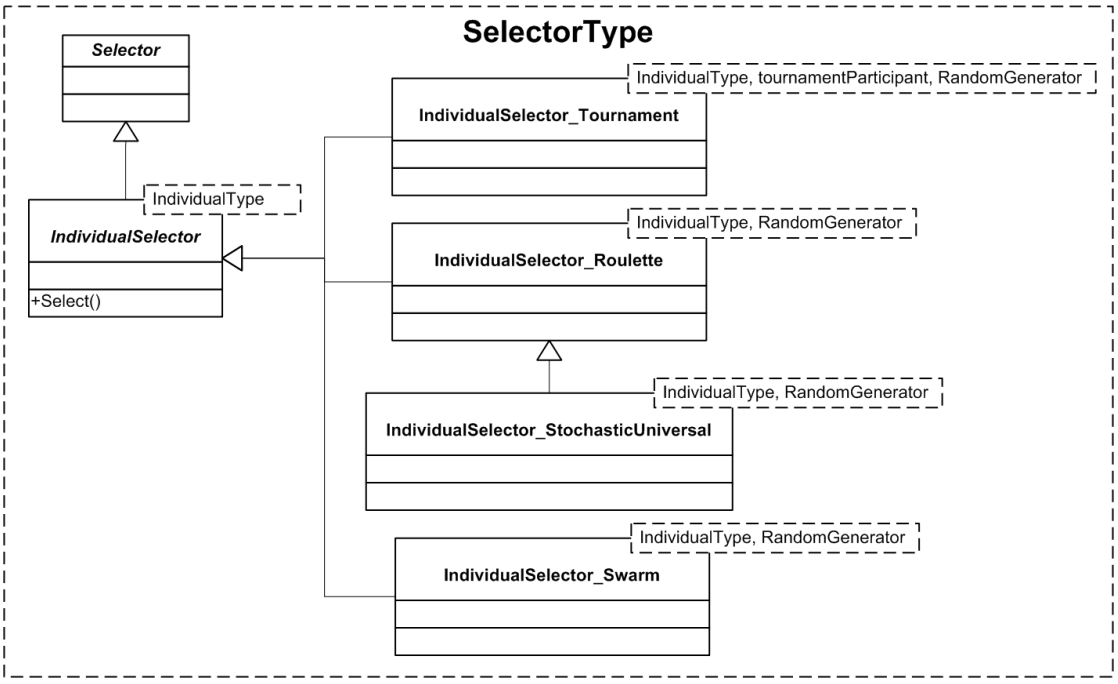
Obrázek 3.6: Rozhraní generátoru chromozómu a třídy implementovaných generátorů.

Objekt generující náhodné rozložení genů chromozomu jedince (*Randomizer*) je definován v souboru garandom.h, kde základní objekt náhodného generování je definován v šabloně *GenomeArrayRandomizer<GenomeType>*. *GenomeArrayRandomizer* definuje funkci *Randomize(GenomeType &genome)*, jejíž implementace v konkrétním poděděném objektu má vygenerovat náhodné rozložení hodnot genů v chromozomu. Také se zde musí vzít v potaz permutační rozložení genů chromozomu, proto konkrétní implementace generátorů náhodného chromozomu obsahují taky šablonový parametr, který určuje možnost generování náhodného permutačního chromozomu.



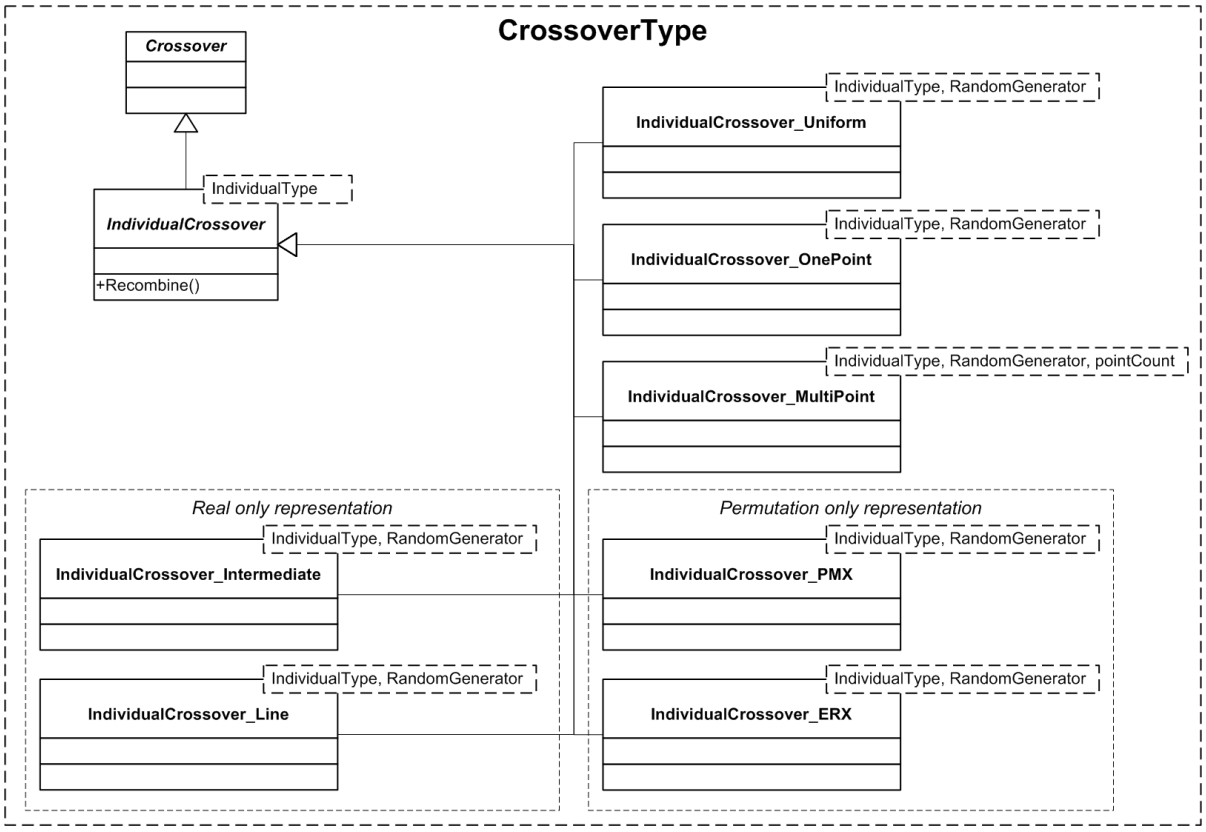
Obrázek 3.7: Hierarchie základního objektu populace a její metody.

Objekt populace (*Population*) je nejdůležitější objekt GA. Obsahuje jedince, operátory nad jedinci a evoluční algoritmus pro vytvoření nové generace. Šablona *Population<IndividualType,ReinsertorType,SelectorType,CrossoverType, RandomGenerator>* je definována v souboru gapopulation.h a je tvořena šablonovým parametrem *IndividualType*, který definuje typ jedince (viz. obr. Obrázek 3.3). Dále objektem *ReinsertorType* určující operátor doplnění, objektem *SelectorType* určující selekci jedinců pro křížení, objektem *CrossoverType* určující křížení selektovaných jedinců a *RandomGenerator* je objektem generování náhodných čísel (objekt je popsán v souboru gacommon.h). Základní třída *Population* dědí z *PopulationBase<IndividualType>*, která definuje společné rozhraní populace, kterého je využito při migraci (viz. obr.Obrázek 3.7). Populaci si totiž může uživatel nadefinovat a naimplementovat sám, ovšem nová populace musí dědit z *PopulationBase*, aby se zaručila kompatibilita s ostatními objekty (*Evolution* a operátory migrace). Populace obsahuje funkcionalitu výpočtu statistik, automatické ukládání po N generacích a jiné užitečné vlastnosti, které si detailněji popíšeme později.



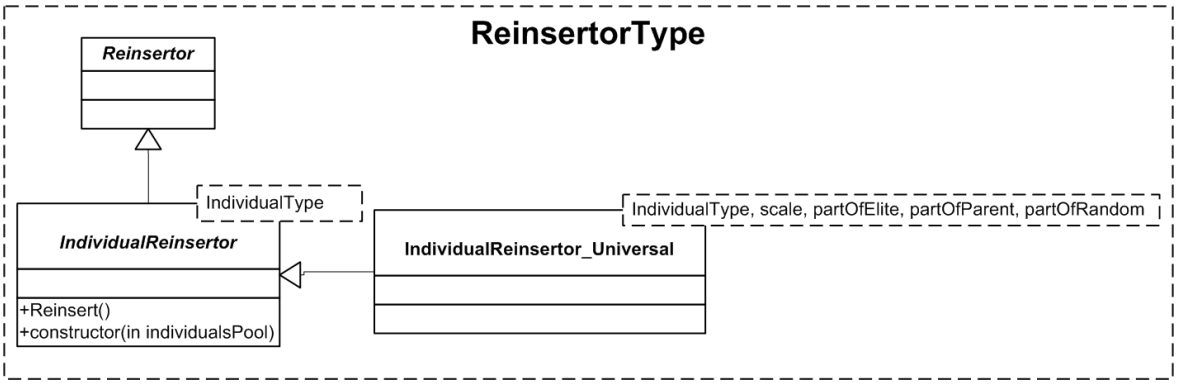
Obrázek 3.8: Rozhraní třídy metody výběru a tříd implementovaných druhů.

Operátor selekce jedinců pro rekombinaci (selector) je definován v souboru gaselection.h, kde rozhraní operátoru selekce je šablona *IndividualSelector <IndividualtType>*, *IndividualType* vyjadřuje typ jedince. *IndividualSelector* definuje funkci *Select(IndividualVec &source, IndividualVec &selected, int totalSelected)*, kde *source* je vektor jedinců jako zdroj pro selekci. Selektované jedince tato funkce vloží do vektoru *selected* a poslední parametr *totalSelected* vyjadřuje požadovaný počet selektovaných jedinců. Funkce *Select()* tohoto operátoru je volána z populace při generování nové generace.



Obrázek 3.9: Rozhraní třídy pro křížení a třídy jejich implementovaných druhů.

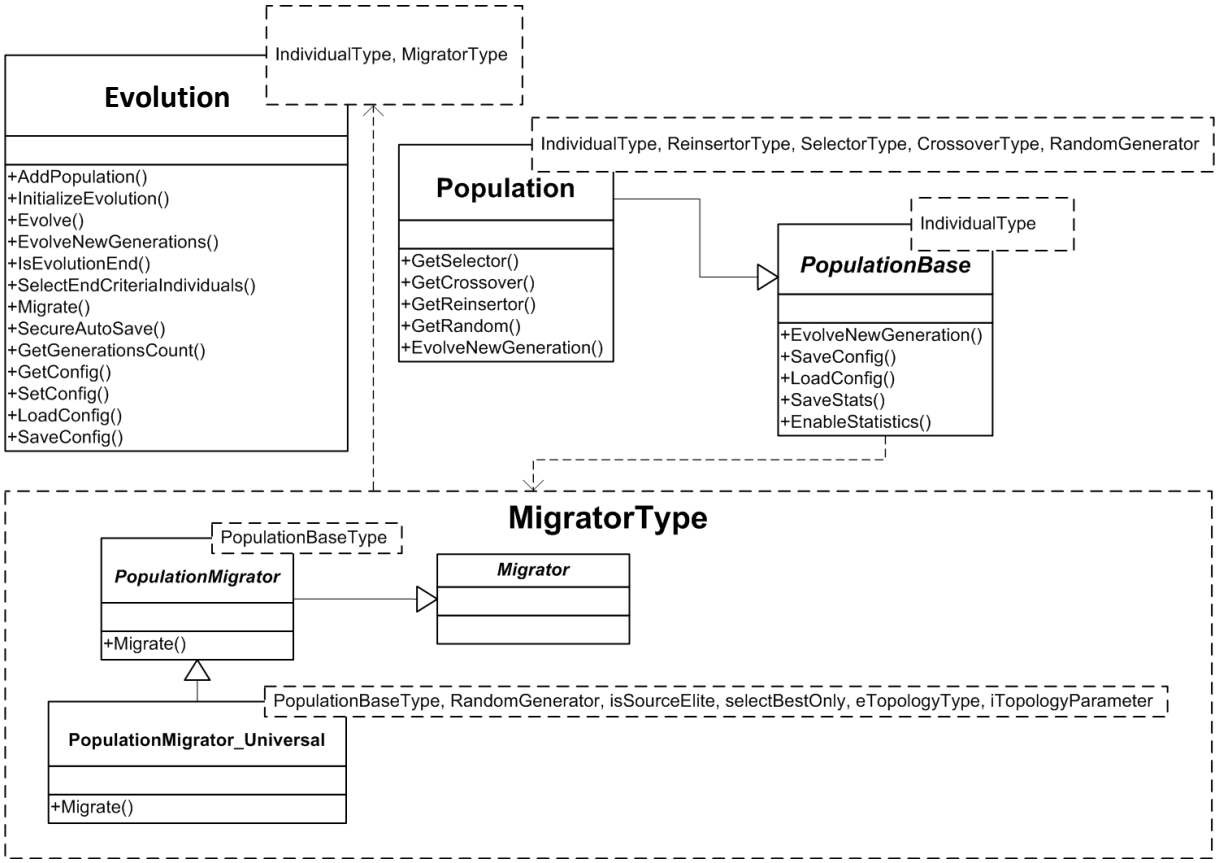
Operátor křížení selektovaných jedinců (*Crossover*) a vytvoření potomků (*Offspring*) je definován v souboru gacrossover.h, kde rozhraní operátoru křížení je šablona *IndividualCrossover<IndividualType>*, *IndividualType* vyjadřuje typ jedince. *IndividualCrossover* definuje funkci *Recombine(const IndividualType &parentA, const IndividualType &parentB, Individuals &offspring)*, kde funkce kříží dva rodiče *parentA* a *parentB* do několika potomků, které vloží do vektoru jedinců *offspring*. Funkce *Recombine()* tohoto operátoru je cyklicky volána z populace při generování nového potomstva generace (selektovaných jedinců selektorem).



Obrázek 3.10: Rozhraní třídy znovuvložení jedinců a její implementovaná třída.

Operátor doplnění nové generace (*Reinsertor*) je definován v souboru gareinsertor.h, kde je základní rozhraní operátoru doplnění šablona *IndividualReinsertor<IndividualType>*, *IndividualType* vyjadřuje typ jedince. *IndividualReinsertor* definuje funkci *Reinsert(IndividualsVec &offspring, const IndividualsVec &generation, const IndividualsVec &elite)*, kde je výsledkem doplnění úprava vektoru nově vygenerovaných jedinců *offspring*.

Zdrojem pro úpravu *offspring* jedinců jsou parametry *generation* (stávající generace jedinců) a *elite* (elitní množina ze všech generací). Je již pouze na volbě programátora reinsertoru, kterou z těchto množin použije při doplnění. Funkce *Reinsert()* tohoto operátoru je volána z populace při generování nové generace.



Obrázek 3.11: Obecná hierarchie hlavních tříd GA a migrace, včetně jejich metod.

Nejdůležitějším objektem z hlediska uživatele GA je *Evolution*, který je definován v souboru ga.h. *Evolution* je rozhraní mezi uživatelem a samotným genetickým algoritmem. Obsahuje sadu funkcí, které uživateli poskytnou nastavení a provedení evoluce definovaných populací (funkce *Evolve()*). Objekt *Evolution* je šablona *Evolution <IndividualType,MigrationType>*, kde *IndividualType* vyjadřuje typ jedince a *MigrationType* typ objektu migrace populací. *Evolution* také funguje jako kontejner populací, obsahuje tedy vektor *PopulationBase* instancí. Nad tímto kontejnerem provádí migraci dle nastavené konfigurace.

Operátor migrace jedinců populací (*Migrator*) je definován v soboru gamigration.h, kde je základní operátor migrace jedinců šablona *PopulationMigrator<PopulationBaseType>*. *PopulationBaseType* je typ bázové třídy populace (v našem případě se bude jednat pouze o *PopulationBase*). Tato třída definuje funkci *Migrate(Populations &populations, int migrationRateScale, int migrationRate)*, kde *populations* je vektor populací pro migraci, *migrationRateScale* měřítko pro poměr migrace *migrationRate* (například pokud chceme migrovat 5 jedinců z 1000, zadáme *migrationRateScale* = 1000, *migrationRate* = 5). Účelem této funkce je migrovat určitý počet jedinců do populací zadaných v parametru *populations*. Funkce *Migrate()* je volána na instanci migrátoru z *Evolution* instance při požadavku na migraci (ta je zadána dle konfigurace *Evolution*).

3.2 Třída populace

Třída *Population* je dána parametry šablony (operátory), které se v této třídě vytvoří jako instance (*m_selector*, *m_crossover*, *m_reinsertor*, *m_random*). To dává GA vlastnost variability, uživatel si může vytvořit vlastní operátor, který lze za chodu evoluce měnit.

Třída *PopulationBase* funguje jako kontejner jedinců generace (*m_generation*), konkrétní implementace *Population* pouze operuje s tímto kontejnerem při generování nové generace (virtuální funkce *EvolveNewGeneration()*). Z optimalizačních důvodů *PopulationBase* obsahuje takzvanou banku jedinců (*IndividualsPool*), která poskytuje recyklování již použitých jedinců (např. ze staré generace). Tímto dosáhne objekt *Population* při generování nové generace vyšší rychlosti, jelikož recyklací minimalizujeme alokace paměti na nulu. *PopulationBase* také obsahuje kontejner elitních jedinců (*m_generationElite*), který udržuje nejlepší jedince ze všech generací.

Třída *PopulationBase* pro optimalizaci řešení genetického algoritmu využívá řazení jedinců podle jejich fitness hodnoty. To znamená, že kontejnery jedinců generace a elitních jedinců jsou vždy v populaci seřazeny. Tato metodika nám poskytuje unikátní vlastnosti, které nám nejen práci s populací urychlí, ale i ulehčí. Jelikož můžeme při operátorech evoluce využít vlastnosti řazení, tj. nejlepší jedinec je vždy vlevo (na indexu 0) ve vektoru jedinců. Můžeme tedy rychleji selektovat jen ty nejlepší jedince například pro operátor křížení. Řazení jedinců nám obstarává funkce *PopulationBase::SortGeneration()* a *Comparator* definovaný na jedinci.

3.2.1 Vytvoření nové generace EvolveNewGeneration()

Třída *Population* implementuje děděnou funkci *EvolveNewGeneration()*, která využívá všech definovaných operátorů pro vytvoření nové generace, kterou nahradí stávající.

Algoritmus vytvoření nové generace:

1) *CutLife()* na všechny jedince aktuální generace (ubere jeden život jedinců)
2) *m_selector.Select()* – výběr jedinců pro křížení
3) *m_crossover.Recombine()* – křížení selektovaných jedinců, vytvoření offspring
4) *Mutate()* na všechny jedince z offspring, dle pravděpodobnosti mutace
5) Seřazení *offspring* (potomků) dle fitness
6) *m_reinsertor.Reinsert()* – doplnění offspring
7) Nahrazení rodičovské generace potomky (*offspring* -> *m_generation*)
8) Selekce elitních jedinců
9) Výpočet statistik

Tento algoritmus je volán z třídy *Evolution* po celou dobu evoluce.

3.2.2 Konfigurace a statistiky populace

Máme zde třídu *PopulationConfig*, která řídí určité vlastnosti vytvoření nové generace. Konfigurace lze načíst/uložit ze/do souboru pomocí *LoadConfig()*/*SaveConfig()* nebo lze konfiguraci číst či nastavit (*SetConfig()* či *GetConfig()*).

Parametry konfigurace populace:

- *PopulationConfig.Elite.TotalEliteIndividuals* – počet elitních jedinců které si udržuje populace
- *PopulationConfig.Crossover.SelectedIndividualsForRecombination* – počet jedinců vybraných pro křížení (posílá se jako parametr selektoru)
- *PopulationConfig.Mutation.Type* – typ mutace, definován v *EMutationType*.
- *PopulationConfig.Mutation.IndividualMutationPropability* – pravděpodobnost mutace jedince v procentech (používá se při výpočtu nové generace, určuje pravděpodobnost mutace *offspring* populace). Platí pouze, pokud typ mutace je zadán jako eMutationType_IndividualMutationPropability.
- *PopulationConfig.Mutation.IndividualMutationCount* – počet jedinců, u kterých se vynutí mutace. Platí pouze, pokud typ mutace je zadán jako eMutationType_IndividualMutationCount.
- *PopulationConfig.Mutation.GeneMutationPropability* – pravděpodobnost mutace genu jedince v procentech (posílá se jako parametr mutátoru)

Aktuální populaci lze také jednoduše uložit/načíst do/ze souboru funkcemi *SaveGeneraton()*/*LoadGeneration()*. Tato funkcionalita se používá spíše z objektu *Evolution*, kde podle konfigurace systém uloží aktuální generaci do souboru z důvodů bezpečnosti a kontinuální evoluce (pokračování přerušené evoluce).

Samotná evoluce vytváří statistiky, které lze s každou novou generací uložit do souboru. Formát ukládání lze na populaci definovat za pomoci virtuální funkce *SaveStats()*. Zapnout automatické ukládání statistik s každou novou generací lze pomocí funkce *EnableStatistics (filename)*, kde *filename* určuje název souboru, do kterého se statistiky budou připisovat (dle formátu zápisu *SaveStats()*). Tuto funkcionalitu lze také automaticky zapnout v konfiguraci třídy Evolution.

3.2.3 Definování populace a jedince

Pro použití populace si nejdříve musíme definovat jedince. Kupříkladu vytvoříme jedince se znakovou reprezentací (znaky abecedy) o délce chromozomu 20. Dále nadefinujeme Mutátor (*random additive change mutator*, pozmění gen v lokálním rozsahu) a *Randomizer* (interval z písmen od A-Z) chromozomu. A v poslední řadě musíme nadefinovat fitness funkci jedince (budeme hledat řešení chromozomu s textem “MRAVENECNIK”).

```
typedef GA::GenomeArray<char,11> MyGenome;
typedef GA::GenomeArrayMutator RandomAdditiveChange<MyGenome,'A','Z',10> MyMutator;
typedef GA::GenomeArrayRandomizer Interval<MyGenome,GA::Random,char,'A','Z'> MyRandomizer;

class MyIndividual : public
GA::Individual<MyGenome,MyMutator,MyRandomizer,5,GA::Comparator_GreaterIsBetter>
{
        virtual double CalculateFitness()
        {
                int ret = 0;
                static const char src[] = "MRAVENECNIK";
                GenomeType::GenType *p = GetGenome().GetGens();
                for( int i = 0; i<GenomeType::GetCount(); ++i )
                {
                        if( p[i] == src[i] )
                        {
                                ++ret;
                        }
                }
                return double(ret);
        }
};
```

Obrázek 3.12: Definice jedince a implementace fitness funkce.

Nyní nadefinujeme operátory populace a samotnou populaci. Selector (turnajová selekce), Crossover (křížení swarm), Reinsertor (univerzální).

```
typedef GA::IndividualSelector_Tournament<MyIndividual> MySelector;
typedef GA::IndividualCrossover_Uniform<MyIndividual> MyCrossover;
typedef GA::IndividualReinsertor_Universal<MyIndividual> MyReinsertor;
typedef GA::Population<MyIndividual,MyReinsertor,MySelector,MyCrossover> MyPopulation1;
typedef GA::Population<MyIndividual,MyReinsertor,MySelector,MyCrossover> MyPopulation2;
```

Obrázek 3.13: Populace a její operátory.

Třídu populace máme již nadefinovanou, nyní můžeme vytvořit instanci a inicializovat ji konfigurací.

```
PopulationConfig pop1Config;
pop1Config.Elite.TotalEliteIndividuals = 15;
```

```
pop1Config.Crossover.SelectedIndividualsForRecombination = 100;
pop1Config.Mutation.Type = eMutationType IndividualMutationPropability;
pop1Config.Mutation.IndividualMutationPropability = 100;
pop1Config.Mutation.GeneMutationPropability = 5;
MyPopulation1 population1(1/*ID*/,pop1Config,100/*počet jedinců v populaci*/);

MyPopulation2::Config pop2Config;
pop2Config.TotalEliteIndividuals = 15;
pop2Config.SelectedIndividualsForRecombination = 100;
pop2Config.IndividualMutationPropability = 100;
pop2Config.GeneMutationPropability = 5;
MyPopulation2 population2(2/*ID*/,pop2Config,100/*počet jedinců v populaci*/);
```

Obrázek 3.14: Vytvoření a konfigurace populace.

Nyní máme vytvořenou instanci populace, která je již připravena na evoluci.

3.3 Třída evoluce

Jak již bylo řečeno, třída *Evolution* obsluhuje samotnou evoluci. Můžeme zde nakonfigurovat, kdy má evoluce skončit. Evoluce končí podle třech kriterií, které si může uživatel nadefinovat.

Ukončující kritéria evoluce:

- Kritérium počtu generací – evoluce skončí po N generacích.
- Kritérium dosažení fitness hodnoty elitního jedince – evoluce končí, pokud alespoň jeden elitní jedinec, ze všech populací, má cílovou fitness (nebo lepší).
- Kritérium délky evoluce – evoluce končí po čase T.

Tyto kritéria ukončení evoluce se vztahují pouze na funkci *Evolution::Evolve(IndividualsVec &solution, EndType &endCondition)*, kde *solution* je vektor jedinců splňujících zadané kritéria (alespoň jedno kriterium) a *endCondition* indikuje, jakého kritéria bylo při evoluci dosaženo. Uživatel si samozřejmě může evoluci provádět „ručně" za pomoci funkcí *Evolution::InitializeEvolution()*,*Evolution::EvolveNewGeneration()* a *Evolution::IsEvolutionEnd()*.

3.3.1 Konfigurace GA

Stejně jako u populace, *Evolution* obsahuje také svoji konfiguraci, která má ovšem globálnější rozsah. Zejména to jsou kritéria konce evoluce a nastavení migrace populací. Konfiguraci lze načítat ze souboru či ukládat podobně jako u populace.

Konfigurace GA:

- *Config.IsolationTime* – určuje počet generací, v jejichž průběhu nebude prováděna migrace
- *Config.MigrationRateScale* – měřítko rozsahu migrace jedinců
- *Config.MigrationRate* – počet jedinců populace pro migraci (dle měřítka rozsahu migrace)
- *Config.EndCriteria* – enumerace kritéria ukončení evoluce
    - *eEnd_MaxGenerations* – ukončení po překročení generací

  - *eEnd_MaxTime* – ukončení při délce evoluce *EndBy_MaxTime* minut
  - *eEnd_EliteFintessValue* – ukončení při dosažení *EndBy_EliteFitnessValue*
- *Config.EndBy_MaxGenerations* – parametr ukončení dle počtu generací
- *Config.EndBy_MaxTime* – parametr ukončení dle času
- *Config.EndBy_EliteFintessValue* – parametr ukončení dle fitness
- *Config.IsStatsEnabled* – zapne či vypne automatické ukládání statistik každé generace populace
- *Config.SecureAutoSaveGenerations* – Bezpečnostní ochrana, pokud je hodnota nastavena, všechny populace uloží každou *N*-tou generaci svá data do souboru. Při opětovném spuštění evoluce si populace zkusí načíst data ze souborů, aby pokračovala tam, kde skončila. Data se ukládají do stejného souboru s názvem daným uživatelem a příponou „.gabin" (pro uložení dat generace a elitních jedinců), „.gacfg" (pro uložení konfigurace populace).

3.3.2 Definování evoluce a použití GA

Nadefinujeme instanci třídy *Evolution* a spustíme evoluci na populaci. Nejprve si ovšem nadefinujeme *Migrator* (migrace *CompleteNet* se selekcí pouze nejlepších jedinců elity). Využijeme zde nadefinovanou populaci *MyPopulation* .

```
typedef
GA::PopulationMigrator Universal<GA::PopulationBase<MyIndividual>,DefaultRandom,true,true,GA::
eTopology CompleteNet> MyMigrator;

typedef GA::Evolution<MyIndividual,MyMigrator> MyEvolution;

MyEvolution::Config evolutionConfig;
evolutionConfig.IsolationTime = 5;
evolutionConfig.MigrationRateScale = 100;
evolutionConfig.MigrationRate = 2;
evolutionConfig.EndCriteria = MyEvolution::eEnd_EliteFitnessValue;
//evoluce skončí až nalezne řešení "MRAVENECNIK"
evolutionConfig.EndBy EliteFitnessValue =   11;
evolutionConfig.IsStatsEnabled = true; // statistiky budou uloženy do souboru s příponou
".gastats"
evolutionConfig.SecureAutoSaveGenerations = 5; //uloží každou pátou generaci
//vytvoření instance MyEvolution;
MyEvolution ga(evolutionConfig);
```

Obrázek 3.15: Definice migrace, konfigurace a vytvoření GA (Evolution).

Nyní přidáme do GA vytvořené instance populace (abychom využili i migrace, původní instanci populace „*population*" nám nebude stačit, proto mějme vytvořené dvě populace s názvem instance „*population1*" a „*population2*"). Pak už nám jen zbývá spustit evoluci.

```
// přidáme populace do ga, druhým parametrem AddPopulation() zadáme název souboru
// pro bezpečnostní uložení generace populací, pokud není zadán, neukládá se.
ga.AddPopulation(&population1,"populatin1 filename");
//nebo ga.AddPopulation(&population1);
ga.AddPopulation(&population2,"populatin2_filename");

// spustíme evoluci
MyEvolution::PopulationType::IndividualsVec solution;
MyEvolution::EndType endBy;
ga.Evolve(solution, endBy);
```

```
// výpis výsledku
std::cout << "Evoluce nalezla řešení po " << ga.GetGenerationsCount() << " generacích.\n";
char ch[100];
::memcpy(ch,solution.front()->GetGenome().GetGens(),MyGenome::GetCount());
ch[MyGenome::GetCount()] = 0;
std::cout << "řešení: " << ch << std::endl;
```

Obrázek 3.16: Přidání populací do GA a spuštění evoluce.

Jak lze vidět, použití GA je pro uživatele velmi jednoduché. Stačí si pouze nadefinovat typy jedinců, operátorů a populace. Vytvořit instance populací, vložit je do GA a spustit evoluci. Výsledek dostanete v parametru evoluce. Samozřejmě za chodu si může kontrolovat statistiky, dle výstupního souboru statistik.

3.4 Implementované metody, operátory a reprezentace

Během vývoje knihovny byly implementovány některé ze základních operátorů genetických algoritmů, které nyní stručně popíši.

Definice reprezentace:

- GenomeArray<T,length> – kde T je typ genu, platí pro jakékoli celočíselné reprezentace (také i pro vlastní typy). Příklad: GenomeArray<int,length> - reprezentace, kde gen je integer a chromozóm je tvořen length integeru.
- GenomeArray_Binary<length> - binární reprezentace, detailně definován jako GenomeArray<BinaryType>, kde BinaryType je definován v gacommon.h jako char (v tomto souboru je také definována hodnota BINARY_TRUE, BINARY_FALSE).
- GenomeArray_Double<length> - reálna reprezentace, definovaná detailně jako GenomeArray_<double,length>.

Implementace randomizerů:

- GenomeArrayRandomizer_RandomMax - náhodně generuje celočíselné hodnoty genu v rozsahu 0 až maximum daným parametrem šablony.
- GenomeArrayRandomizer_Interval - náhodně generuje celočíselné hodnoty genu v rozsahu od - do podle parametrů šablony.
- GenomeArrayRandomizer_Interval_Double - náhodně generuje reálné hodnoty genu v rozsahu od - do podle parametrů.
- GenomeArrayRandomizer_Binary - náhodně generuje mezi GA_BINARY_TRUE / GA_BINARY_FALSE hodnoty genu, detailně je definován jako GenomeArrayRandomizer_Interval s parametry rozsahu od GA_BINARY_FALSE po GA_BINARY_TRUE.

Implementované mutátory:

- GenomeArrayMutator_Binary – pouze pro binární reprezentaci, implementace binárního operátoru mutace (viz. kapitola 4.3.1).
- GenomeArrayMutator_Swap – implementace Swap operátoru, použitelný pro všechny typy reprezentace (viz. kapitola 4.3.3).

- GenomeArrayMutator_Shift – implementace Shift operátoru, použitelný pro všechny typy reprezentace (viz. kapitola 4.3.4).
- GenomeArrayMutator_Inverse – implementace Inverse operátoru, použitelný pro všechny typy reprezentace (viz. kapitola 4.3.5).
- GenomeArrayMutator_RandomAdditiveChangele – mutace pomocí aditivní změny, použitelná pouze pro celočíselnou reprezentaci (viz. kapitola 4.3.2).
- GenomeArrayMutator_RandomAdditiveChange_Double – mutace pomocí aditivní změny, pouze pro reálnou reprezentaci (viz. kapitola 4.3.2).
- GenomeArrayMutator_Multiplier_Double – multiplikativní mutátor, pouze pro reálnou reprezentaci (viz. kapitola 4.3.2).

Implementované metody selekce:

- IndividualSelector_Tournament – implementace turnajového výběru s volitelným počtem účastníků turnaje (viz. kapitola 4.1.4).
- IndividualSelector_Roulette – implementace ruletového výběru (viz. kapitola 4.1.1)
- IndividualSelector_StochasticUniversal – implementace stochastického univerzálního výběru (viz. kapitola 4.1.2).
- IndividualSelector_Swarm – implementace výběru který vybírá vždy nejlepšího jedince v kombinaci se všemi ostatními.

Implementovaná křížení:

- IndividualCrossover_Uniform – implementace uniformního křížení, použitelné pro všechny typy reprezentací (viz. kapitola 4.2.1.3).
- IndividualCrossover_MultiPoint – implementace N bodového křížení, použitelné pro všechny typy reprezentací (viz. kapitola 4.2.1.1 a 4.2.1.2).
- IndividualCrossover_PMX – implementace PMX operátoru, použitelný pouze pro permutační reprezentaci kódování (viz. kapitola 4.2.2.1).
- IndividualCrossover_ERX – implementace ERX operátoru, použitelný pouze pro permutační reprezentaci kódování (viz. kapitola 4.2.2.2).
- IndividualCrossover_Intermediate – implementace středního křížení, použitelné pouze pro reálnou reprezentaci (viz. kapitola 4.2.3.1).
- IndividualCrossover_Line – implementace liniového křížení, použitelné pouze pro reálnou reprezentaci (viz. kapitola 4.2.3.2).

Implementované metody znovuvložení (Reinsertorů):

- IndividualReinsertor_Universal – byl implementován pouze jeden velice univerzální reinsertor který pomocí nastavení svých parametrů v kombinaci s nastavením konfigurace populace může splnit všechny typy znovuvložení z kapitoly 4.4 a ještě další.

Implementace migrátoru:

- PopulationMigrator_Universal – implementace univerzálního migrátoru podle kapitoly 4.5.1, umožněna je topologie kompletní sítě, kruhu a sousedství, migrovat mohou buď jedinci z populace, nebo z elitního vektoru populace, přičemž můžeme nastavit, zda mají migrovat náhodní jedinci, či jedinci s nejlepším ohodnocením.

3.5 Využití OpenMP v knihovně GA

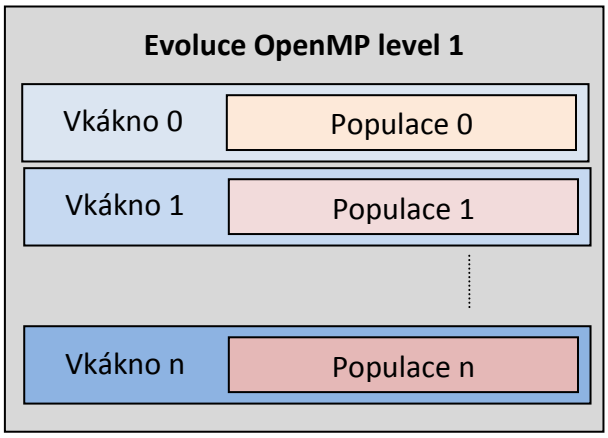
GA je paralelizováno na dvou úrovních, Open MP level 1 (paralelizace populací) a OpenMP level 2(paralelizace generace). Tento typ je definován konstanty *EOpenMP_Level*.

```
typedef enum EOpenMP_Level
{
  eOpenMP_Level_Automatic = -1, // default
  eOpenMP_Level_NoParallel = 0, // disable parallel
  eOpenMP_Level_PopulationOnly, // OpenMP Level 1
  eOpenMP_Level_Generation      // OpenMP Level 2
}EOpenMP_Level;
```

Kde *eOpenMP_Level_Automatic* určuje, že před spuštěním evoluce si GA detekuje hostitelský hardware a podle něj a počtu populací vybere optimální paralelní level. *Automatic* je zadán v GA jeko výchozí. *eOpenMP_Level_NoParallel* vynutí nepoužívání paralelizace. *eOpenMP_Level_PopulationOnly* definuje typ paralelizace OpenMP, tedy v rámci populací. *eOpenMP_Level_Generation*, neboli OpenMP Level 2 paralelizuje jak populace, tak samotnou evoluci. Typ paralelizace lze za chodu měnit, či číst pomocí funkcí *OpenMP_SetLevel(...)* a *OpenMP_GetLevel()*.

3.5.1 OpenMP Level 1

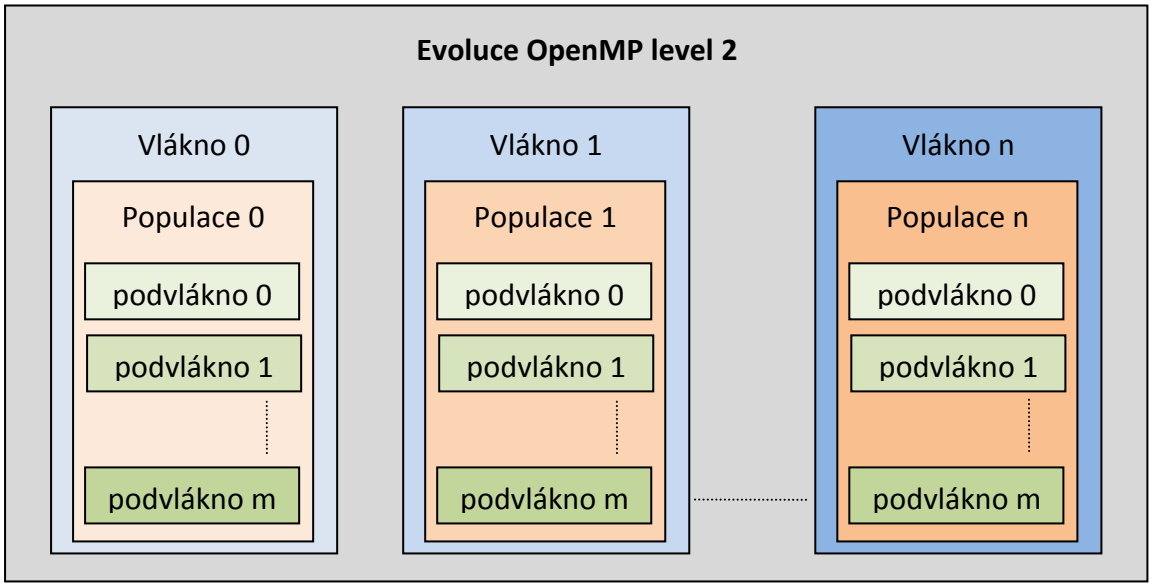
OpenMP je využíváno pouze v hlavním objektu *Evolution*. Při spuštění evoluce se vytvoří takový počet vláken jako je počet populací v *Evolution* objektu a každému se přiřadí patřičné vlákno. Tedy každá evoluce populace bude mít k dispozici vlastní vlákno. U tohoto typu paralelizace je dobré vytvořit tolik populací, kolik je na hostitelském počítači jader. Využije se tak maximální výkon hardwaru. OpenMP level 1 má ovšem stinnou stránku, a to že pokud budeme chtít provádět evoluci na menším počtu populací, nevyužijeme zcela potenciálu hardwaru. V takových to případech je lepší využít paralelizace typu OpenMP level 2.



Obrázek 3.17: Evoluce OpenMP level 1.

3.5.2 OpenMP Level 2

Typ paralelizace OpenMP level 2 je mnohem zanořenější než OpenMP level 1. Spustí samostatné vlákno na každou populaci (jak je tomu u OpenMP level 1), ale také spustí více podvláken při výpočtu nové generace. Paralelizovány jsou tedy konkrétní komponenty evoluce, jako selekce, křížení, mutace, výpočet fitness funkce a další komponenty, kde má paralelizace smysl. Ne vždy lze paralelizovat, například při sdílení stejných zdrojů by případné kritické sekce zbytečně zpomalili evoluci.



Obrázek 3.18: Evoluce OpenMP level 2.

Výhodou OpenMP level 2 je, že lze lépe využít hardwaru hostujícího počítače. Lze tedy spustit evoluci i menšího počtu populací, a přitom plně využít výkon hardwaru. OpenMP level 2 samozřejmě nemá smysl používat, pokud hostující počítač obsahuje pouze jedno výpočetní jádro. Takováto paralelizace by pouze evoluci zpomalila.

3.5.3 Paralelizované objekty

Během vývoje GA byla snaha paralelizovat veškeré objekty evoluce, což se z větší části povedlo. Ovšem existují části kódu, které paralelizovat nelze a je nutné aby byly počítány jen v jednom vlákně, nebo části, kde by paralelizace byla zbytečnou brzdou. Samotná evoluce, neboli výpočet nové generace je dána sadou funkcí v objektu *Population*. Ty jsou sice za sebou sekvenčně prováděny, ovšem samotný kód v těchto funkcích je paralelizován, tedy pokud je zapnut OpenMP Level 2, jinak se provádí sekvenčně. Nyní si průběh funkcí evoluce popíšeme jak jdou za sebou.

Evolution_CutLifes(): Paralelně se zavolá *CutLife()* na každém jedinci generace. Je zde využito příkazu sub-paralelní sekce "`#pragma omp parallel for`".

Evolution_Select(...): Selekce jedinců pro křížení. Samotná funkce paralelizována není, ovšem využívá selektoru definovaného uživatelem, který paralelizován je. Jsou to selektory typu *IndividualSelector_Tournament*, *IndividualSelector_Roulette* a *IndividualSelector_TournamentElite*.

Evolution_Recombine(...): Křížení selektovaných jedinců. Funkce paralelizovaná je, volá funkci *Recombine()* na objektu *crossover* na více vláknech. Tato funkce nejprve vytvoří vektor nových potomků (předalokuje všechny instance) a OpenMP dále rozloží práci nad křížení rodičů pomocí "`#pragma omp parallel for`". Díky předalokovanému vektoru potomků nedochází ke konfliktu vláken, při přístupu do společného objektu, tedy vektoru potomků.

Evolution_Mutate(...): Paralelní mutace vybraných jedinců podle typu mutace.

Evolution_CalculateFitness(): Tato funkce provádí paralelní výpočet fitness funkcí všech nových potomků. Je to jistá optimalizace, která se provádí pouze když je OpenMP v GA použito. V jiným případě tato funkce nic nedělá, protože by neměla smysl, fitness se na jedinci vypočítává naprosto automaticky, na vyžádání.

Evolution_Reinsert(...): Funkce doplní či znovu-vloží nově vytvořené potomky, aby nová generace měla stejný počet jedinců jako ta předchozí. Funkce paralelizovaná není, ovšem funkce nad objektem reinsertoru paralelizovaná je. Například *IndividualReinsertor_Universal*, kde je využito "`#pragma omp parallel sections`", zde jsou označené nezávislé sekce, které jsou spouštěny na více vláknech.

Evolution_SortIndividuals(): Paralelní seřazení nové generace podle fitness.

Evolution_SelectElite(...): Selekce nejlepších jedinců a přiřazení do elitního vektoru jedinců. Funkce paralelizovaná není, paralelizace by z důvodu kritické sekce byla pomalejší než sekvenční zpracování.

Všechny tyto funkce výpočtu evoluce jsou k nahlédnutí ve funkci *Population::EvolveNewGeneration()*. Díky univerzálnímu návrhu GA, jeho rozložení na operátory jako mutátor, selector, recombiner atd. je možné si při implementaci vlastního operátoru paralelizaci pomocí OpenMP naprogramovat sám. Je nutné zde ale dodržovat pravidla OpenMP Levelu, neboli při implementaci vlastního operátoru je nutné za chodu kontrolovat stav funkce *OpenMP_GetLevel()* a podle ní bud provádět kód sekvenčně, či paralelně. Viz. zdrojové kódy operátoru GA.

4 Popis implementovaných genetických operátorů

4.1 Metody výběru (Selection)

Selekční mechanizmus vybírá z populace jedince vhodné k další reprodukci. Měl by imitovat proces přirozeného výběru, kde má silnější jedinec větší šanci na to být vybrán k reprodukci. Doufáme, že potomstvo kvalitních jedinců se bude vyznačovat ještě kvalitnějšími vlastnostmi než rodiče a zároveň tedy lepším ohodnocením (*fitness*). Na druhou stranu musí být metoda selekce zvolena tak, aby v populaci vybraných jedinců byla zachována dostatečná rozmanitost. Pokud by nebyla zachována ona rozmanitost, tak by s největší pravděpodobností řešení konvergovalo k lokálnímu extrému (suboptimální řešení). Pokud by však selekční mechanizmus byl příliš volný, evoluční proces bude postupovat velmi pomalu a algoritmus by pracoval po mnoho generačních cyklů, aniž by byl zřetelný jakýkoliv pokrok. Následně bude popsáno několik základních metod selekce [03].

4.1.1 Ruletový výběr (Roulette Wheel Selection)

Tento způsob výběru se také v angličtině nazývá „Stochastic sampling with replacement“. Ruletový výběr je zřejmě nejčastěji využívaný způsob selekce. Postup je následovný. Sečteme ohodnocení (fitness) všech jedinců v populaci a v závislosti na velikosti jednotlivých ohodnocení je jedincům přiřazena velikost výseče na „ruletovém kole“. Pak je proveden náhodný výběr („hod“, random) a vybraný jedinec postupuje do další generace. Pravděpodobnost s jakou bude tedy *i*-tý jedinec vybrán je dána následovně [03].

$$p(i) = \frac{f(i)}{\sum_{j=1}^{N} f(j)} \tag{4.1}$$

Kde $i \in \{1 \dots N\}$, N je počet jedinců populace, $f(i)$ je ohodnocení i-tého jedince a suma $f(j)$ značí součet všech ohodnocení populace.

Pro prezentovaný ilustrativní příklad z kapitoly 1, obrázek Obrázek 1.5: Ohodnocení chromozómů jedinců z obrázku Obrázek 1.4., to konkrétně znamená rozdělit ruletové kolo na čtyři výseče odpovídající svou plochou ohodnocení příslušných jedinců tak, jak je níže vypočteno a následně uvedeno na obrázku Obrázek 4.1. Když ruletové kolo vytvoříme tímto způsobem, tak nám při každém výběru jedince z populace stačí vygenerovat náhodné číslo R v mezích $\langle 0, 1 \rangle$ a vybrat i-tého jedince právě když platí vztah [03]:

$$\sum_{j=1}^{i-1} p(j) < R < \sum_{j=1}^{i} p(j) \tag{4.2}$$

Není efektivní počítat výše naznačené sumy opakovaně pro výběr každého jedince, proto se zavádí souhrnné (kumulativní) ohodnocení definované jako [03]:

(4.3)

$$\overline{f(i)} = \sum_{j=1}^{i} p(j) = \sum_{j=1}^{i} \frac{f(j)}{\sum_{k=1}^{N} f(k)}$$

Následně je pak i-tý jedinec vybrán když platí:

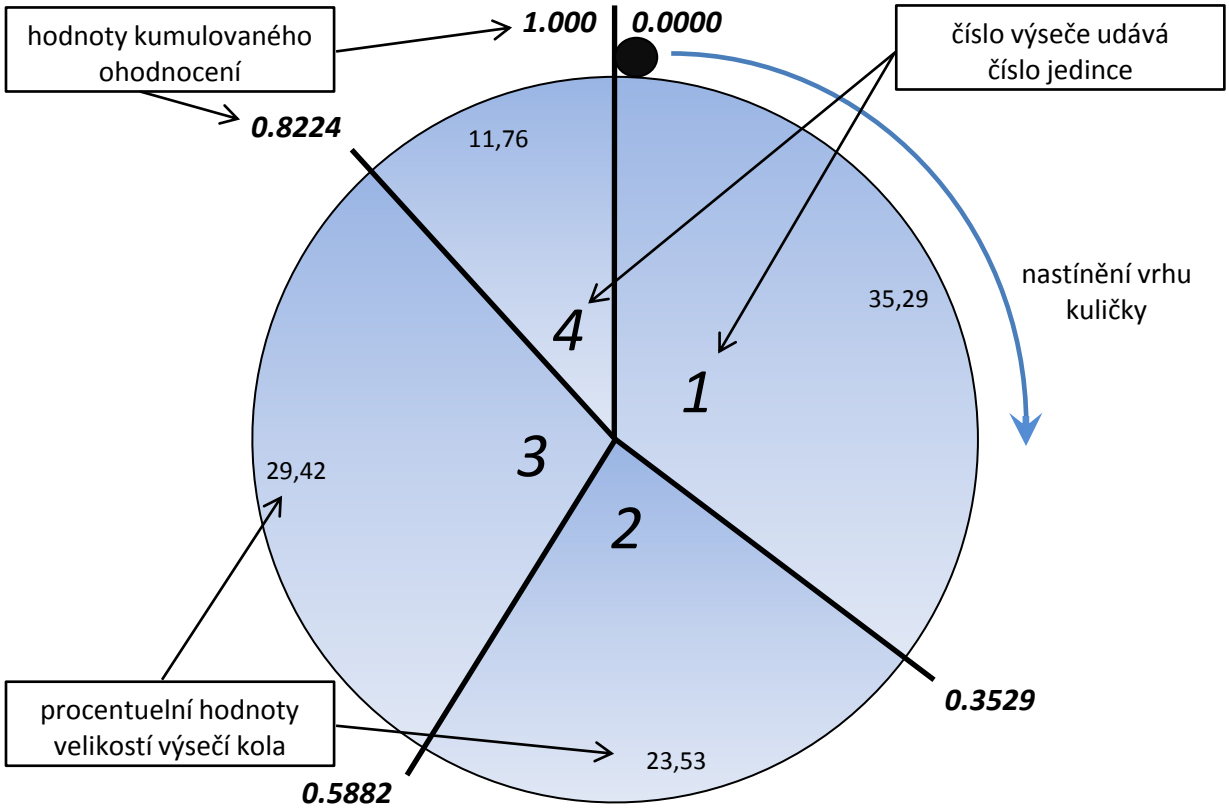
$$\overline{f(i-1)} < R < \overline{f(i)} \tag{4.4}$$

Kde $\overline{f(i)}$ reprezentuje kumulované ohodnocení.

Jedinec číslo	Chromozóm jedince	Ohodnocení *f(i)*	% z celkového ohodnocení *p(i)*	Kumulované Ohodnocení
1	(0, 1, 0, 0, 1, 1, 0, 1, 1, 1)	6	35.29 %	0.3529
2	(1, 0, 1, 1, 0, 0, 0, 1, 0, 0)	4	23.53 %	0.5882
3	(1, 1, 1, 0, 0, 0, 0, 1, 0, 1)	5	29.42 %	0.8824
4	(0, 0, 0, 1, 0, 0, 0, 1, 0, 0)	2	11.76 %	1.0000

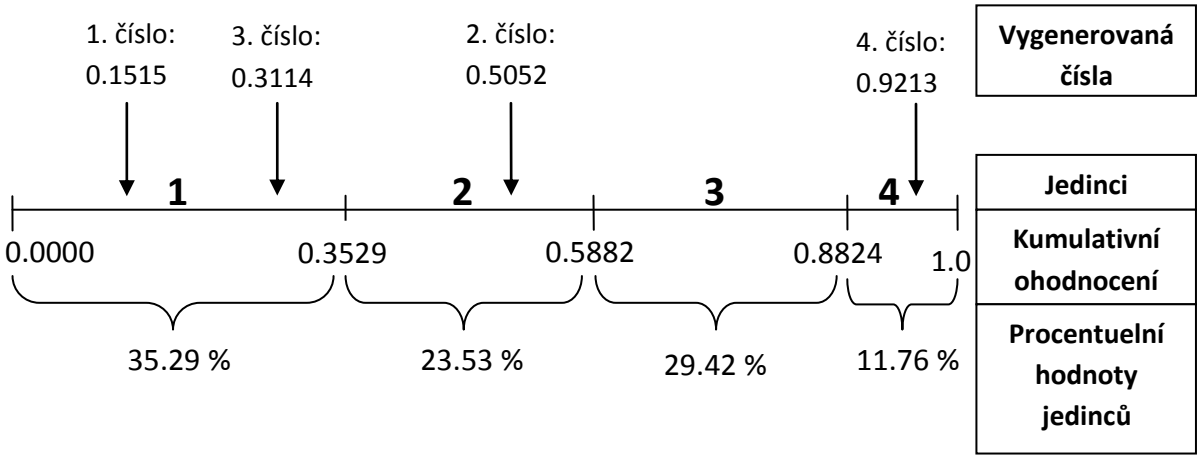
Obrázek 4.1: Procentuelní velikost výseče ruletového kola a kumulované hodnocení.

Grafické znázornění této tabulky na ruletové kolo vypadá následovně:



Obrázek 4.2: Grafické znázornění ruletového výběru na kole rulety.

Představme si čtyři náhodné spiny rulety, kulička se vždy zastaví v nějaké výseči, která reprezentuje určitého jedince (chromozóm). Pokud bychom generovali čtyři náhodná čísla v rozmezí *<0, 1>*, obdrželi bychom podle kumulovaného hodnocení také čtyři čísla. Pomocí tohoto mechanismu tedy vybereme čtyři jedince pro rekombinaci. Například pokud bychom náhodně vygenerovali postupně čísla 0.1515, 0.5052, 0.3114 a 0.9213, tak by nám do rekombinace vstupoval dvakrát chromozóm prvního jedince a po jednom chromozómy druhého a čtvrtého jedince. Všechny informace můžeme jednoduše také zobrazit na ose, jak je na obrázku Obrázek 4.3.



Obrázek 4.3: Selekce čtyř jedinců pomocí kumulativního ohodnocení.

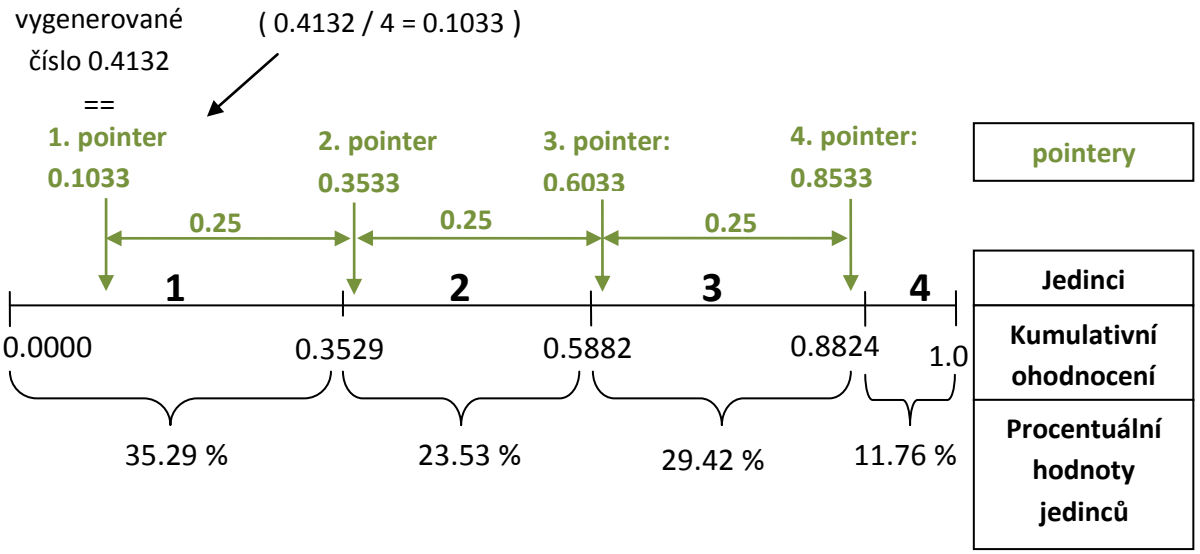
Z náhodného výběru jsme tedy získali čtyři jedince, kteří následně podstoupí proces rekombinace a vzniknou z nich nové chromozómy. Chromozómy vybrané pro rekombinaci se budou rekombinovat po párech, nejčastěji podle pořadí, tedy první s druhým a třetí se čtvrtým [03].

Ruletový selekční mechanizmus však příliš favorizuje nejsilnější jedince. Při použití tohoto výběru může velice rychle nastat převládání jedinců se silným ohodnocením v celé populaci. Dramaticky se sníží rozmanitost populace a výsledek nám může konvergovat k lokálnímu extrému. U ruletového mechanizmu zároveň není garantován minimální rozptyl.

4.1.2 Stochastický univerzální výběr (Stochastic Universal Sampling)

Tento mechanismus výběru je stejný jako předchozí, avšak s jedním rozdílem. Pokud bychom chtěli vybrat čtyři jedince pro rekombinaci, nemusíme vygenerovat čtyři náhodná čísla, ale stačí vygenerovat pouze jedno. Jedinci jsou opět namapováni do navazujících segmentů na ose jako na obrázku Obrázek 4.3. Procentuální jednotky těchto segmentů jsou také zachovány. Po vygenerování jednoho náhodného čísla mezi *<0, 1>* se toto číslo podělí počtem jedinců, z nichž máme vybírat. Výsledné číslo (po vydělení) nám určuje prvního vybraného jedince a na toto místo se přidá „pointer“. Na zbývající část osy se nanesou pointery, které nám určují zbylé jedince zvolené pro rekombinaci. Pointery mají mezi sebou

konstantní vzdálenost, která je rovna *1/N* (kde N je počet jedinců, který chceme vybrat). V případě, že chceme vybrat čtyři jedince, budou na ose právě čtyři pointery, jak naznačuje obrázek Obrázek 4.4:



Obrázek 4.4: Selekce čtyř jedinců pomocí stochastického univerzálního výběru.

K rekombinaci jsme tedy pomocí tohoto způsobu vybrali první dva jedince jednou a třetího dvakrát, i přes to, že jeho procentuální hodnota nebyla největší. Jak je vidět, tento způsob selekce nám umožňuje ponechat v populaci větší rozmanitost, ale zdaleka neřeší hlavní problém všech selekčních mechanizmů, kde je pravděpodobnost jedince přímo úměrná jeho ohodnocení (problém předčasné konvergence) [03].

4.1.3 Seříznutý výběr (Truncation Selection)

Ve srovnání s předchozími výběry metod modelování přirozeného výběru je seříznutý výběr umělá metoda. Je využívána pro populace s velikým počtem jedinců. Jedinci jsou seřazeni podle svého ohodnocení a pouze ti nejlepší jsou vybráni k rekombinaci. Parametrem pro seříznutý výběr je takzvaný trunction treshold (řezací práh). Řezací práh určuje, jaký podíl populace bude vybrán za rodiče, obvykle dosahuje výše 10% - 50%. Jedinci, kteří byli odříznuti neprodukují žádné potomky [20].

Tato metoda není tak sofistikovaná jako ostatní metody a ani se v praxi tak často nepoužívá, nebudeme ji zde tedy dále rozebírat.

4.1.4 Turnajový výběr (Tournament Selection)

Turnaj je v poslední době nejvíce používanou technikou v aplikacích genetických algoritmů. Hlavním důvodem je jednoduchost implementace při zachování kvality selekčního tlaku. Z populace jsou vybírání jednotlivci (nemusí jít obecně o dva) a ti se podrobují „souboji o přežití“, který spočívá v tom, že jedinec s největším ohodnocením (fitnessem) přežívá a postupuje do dalšího kola. Tím se splňuje základní předpoklad, aby přežívali jedinci s vyšší kvalitou, ale jedinci vstupující do souboje jsou vybíráni náhodně, což zaručuje diverzitu

populace. Každý jedinec může být vybrán do soubojů několikrát (jedná-li se o různé souboje). Vše je objasněno na nadcházejícím příkladu, který je na obrázku Obrázek 4.5 [03].

Jedinec číslo	Ohodnocení jedince	Jedinci vybraní do souboje	Výherce souboje	Poznámky
1	5	(1, 3)	3	
2	11	(1, 4)	4	
3	14	(6, 9)	6	
4	4	(5, 9)	5	
5	2	(1, 2)	2	
6	16	(2, 3)	3	
7	8	(7, 8)	7	
8	8	(8, 7)	8	náhodný výběr
9	1	(6, 5)	6	

Obrázek 4.5: Turnajový výběr.

4.1.5 Elitní turnajový výběr (Elite Tournament Selection)

Tento výběr je obdobou předchozího turnajového výběru s tím rozdílem, že první jedinec vybraný do turnaje je vždy určen předem, ostatní jsou vybíráni náhodně. Jako předurčeného jedince bereme vždy toho s nejlepším ohodnocením, který ještě nebyl nikdy vybrán na první pozici. Do prvního turnaje je tedy předurčen jedinec s nejlepším ohodnocením v populaci, k němuž je náhodně vybrán zbývající počet jedinců. Do druhého turnaje je předurčen jedinec s druhým nejlepším ohodnocením v populaci a k němu opět vybráni náhodně další jedinci (náhodně může být vybrán i jedinec s nejlepším ohodnocením).

Elitní turnajový výběr nám tedy zaručuje křížení jedince s nejlepším ohodnocením v populaci.

4.1.6 Elitismus (Elitism)

Předešlé techniky, krom elitního turnajového výběru, obecně nezaručují postup nejlepšího jedince do nové generace. Zvláště v malých populacích je tato ztráta vnímána velmi negativně. Experimenty prokázaly, že ztráta nejlepších jedinců může být opakována a znovuvytvoření jedince není automatické.

Elitismus je jednoduchá technika, kterou vybíráme určitý (velmi malý, například je vybrán pouze jeden) počet nejlepších jedinců a ti se nepodrobují selekčnímu tlaku, ale postupují do nové generace přímo.

4.2 Operátory křížení

Křížení je jeden z hlavních operátorů, jehož použitím získáme nové jedince v generaci, kteří nebyli součástí inicializace populace nebo předchozí generace. Tento operátor nám rozšiřuje prohledávaný prostor tím, že se snaží skládat nová řešení z částí již existujících řešení. Ve svém důsledku použitím této techniky snižujeme možnost rizika uváznutí v lokálních extrémech. Možnost vstupu jedinců do křížení by měla být úměrná jejich ohodnocení a jsou vybíráni pomocí objasněných metod výběru (selekce) z kapitoly Metody výběru (Selection) [03].

Dle typů řešených úloh bylo konstruováno několik křížících operátorů, které v určitých typech úloh mohou selhávat a v jiných dosahovat nejlepších řešení [03].

4.2.1 Binární operátory křížení (Binary Valued Crossover)

Při využití binární reprezentace jedinců (chromozómů) využíváme operátory binárního křížení. Nejčastěji používáme pět binárních operátorů křížení (rekombinace), které jsou popsány níže.

4.2.1.1 Jednobodové křížení (Singlepoint Crossover)

Jednobodové křížení je nejznámější operátor binární rekombinace. Náhodně vygenerujeme číslo mezi 0 a *K-1*, kde *K* reprezentuje délku chromozómů vstupujících do rekombinace (tyto chromozómy byly vybrány na základě selekčního mechanizmu). Právě na *K*-té pozici je hranice, která nám určuje, kde zaměnit geny obou chromozómů. Na následujícím obrázku je vysvětleno jednobodové křížení. Pro přehlednost jsou geny prvního chromozómu zvýrazněny tučně. Náhodně vygenerovaný bod křížení je *K = 4*.

Původní chromozómy (rodičovské)		Nové chromozómy (potomci)
(**1**, **1**, **0**, **0**, **1**, **1**, **1**, **1**, **1**, **1**, **0**, **0**)	→	(**1**, **1**, **0**, **0**, 1, 0, 1, 1, 0, 1, 0, 1)
(0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1)	→	(0, 1, 0, 0, **1**, **1**, **1**, **1**, **1**, **0**, **0**)

Obrázek 4.6: Jednobodové křížení, kde K = 4.

Bod křížení by měl být vybírán z rozsahu *0* až *K-1*, protože kdybychom vygenerovali číslo rovné *K*, tak by hranice křížení byla na poslední pozici a oba chromozómy by zůstaly nezměněné.

4.2.1.2 Dvoubodové a vícebodové křížení (Double-point & Multipoint Crossover)

Dvoubodové křížení je obdoba jednobodového křížení s tím rozdílem, že od druhého bodu se již nebudou geny obou chromozómů zaměňovat. První chromozómy prvního genu jsou opět zvýrazněny tučně a náhodně vygenerované body (hranice) křížení jsou 3 a 8.

Původní chromozómy (rodičovské)		Nové chromozómy (potomci)
(**1**, **1**, **0**, **0**, **1**, **1**, **1**, **1**, **1**, **1**, **0**, **0**)	→	(**1**, **1**, **0**, 0, 1, 0, 1, 1, **1**, **1**, **0**, **0**)
(0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1)	→	(0, 1, 0, **0**, **1**, **1**, **1**, **1**, 0, 1, 0, 1)

Obrázek 4.7: Dvoubodové křížení v bodech 3 a 8.

Stejným způsobem se provádí i vícebodové křížení, kde každý lichý bod určuje pozici, od které začneme geny chromozómů prohazovat. Každý sudý bod křížení vyznačuje pozici, od které zaměňování genů přerušíme. Situaci uveďme na příkladu se čtyřmi body křížení, a sice 3, 4, 6 a 9.

Původní chromozómy (rodičovské)		Nové chromozómy (potomci)
(**1**, **1**, **0**, **0**, **1**, **1**, **1**, **1**, **1**, **0**, **0**)	→	(**1**, **1**, **0**, 0, **1**, **1**, 1, 1, 0, **1**, **0**, **0**)
(0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1)	→	(0, 1, 0, **0**, 1, 0, **1**, **1**, **1**, 1, 0, 1)

Obrázek 4.8: Vícebodové křížení v bodech 3, 4, 6 a 9.

4.2.1.3 Uniformní křížení (Uniform Crossover)

Jednobodové a vícebodová křížení definují vždy body, na kterých se budou zaměňovat jednotlivé geny. Uniformní křížení zobecňuje toto schéma, aby každé místo bylo potencionálním bodem křížení. Využíváme takzvané křížící masky, ta má stejnou délku jako chromozómy, které vstupují do rekombinace. Na všech pozicích genů masky jsou náhodně vygenerovány alely s hodnotou *0* nebo *1.* Ty udávají, jaký rodičovský chromozóm předá konkrétní geny chromozómu potomka. Křížící maska druhého rodiče je inverzní ke křížící masce prvního rodiče. Každý gen, který rodič přispívá potomkovi, je náhodně vybrán s určitou pravděpodobností. První gen prvního potomka má hodnotu genu z prvního předka, pokud je v jeho křížící masce alela prvního genu 1, pokud je v křížící masce alela 0 tak gen potomka má první gen podle genu druhého předka [03].

Původní chromozómy (rodičovské)	Křížící maska	Nové chromozómy (potomci)
(**1**, **1**, **0**, **0**, **1**, **1**, **1**, **1**, **1**, **1**)	(1, 0, 1, 0, 1, 1, 1, 1, 1, 0)	(**1**, 0, **0**, 0, **1**, **1**, **1**, **1**, **1**, 1)
(0, 1, 0, 0, 1, 0, 1, 1, 0, 1)	(0, 1, 0, 1, 0, 0, 0, 0, 0, 1)	(0, **1**, 0, **0**, 1, 0, 1, 1, 0, **1**)

Obrázek 4.9: Uniformní křížení.

Jinými slovy se pro každý gen prvního rodiče náhodně určí do kterého (přičemž pozičně stejného genu z chromozómu) z potomků bude zděděn. Rodič, jehož gen nebyl přenesen do prvního potomka, dodává gen druhému potomkovi. Většinou první gen od prvního rodiče dědí první potomek s pravděpodobností *50* procent, ze zbylých *50*-ti procent dostane potomek první gen od druhého rodiče, tato pravděpodobnost nemusí být 50 na 50, ale můžeme si ji libovolně zvolit.

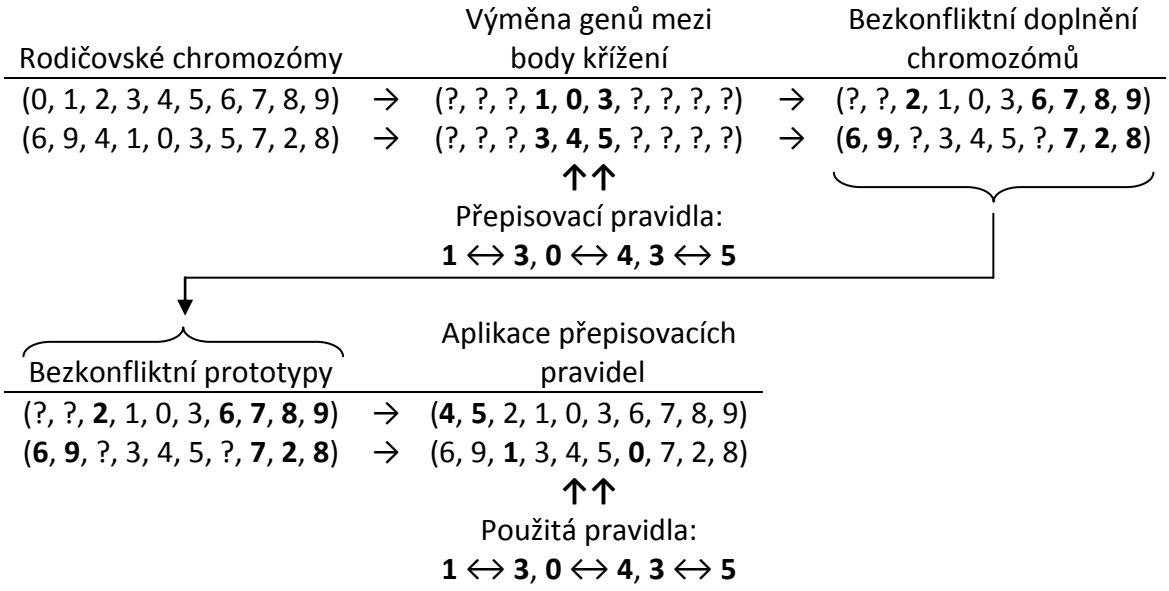
4.2.2 Křížení při použití permutačního kódování

Používání operátorů křížení s sebou u permutačního kódování nese určité problémy. V žádném chromozómu nesmí mít alely genů stejnou hodnotu, jinak by se již nejednalo o permutaci. Některé objekty by mohly v chromozómu chybět a jiné by mohly být naopak zdvojeny. Tomuto efektu se nedá obecně při výměně libovolných částí chromozomů zabránit, musí se následně aplikovat nějaká opravná procedura, která z nových jedinců udělá znovu permutace. Při permutačním kódování využíváme dva základní rekombinační operátory.

4.2.2.1 Křížení s částečným přiřazením - PMX (Partially Mapped Crossover)

Tento operátor byl vyvinut, aby využil uspořádané množiny permutace jednoho z rodičů a přitom zachoval pořadí a pozici co nejvíce objektů podle permutace odpovídající druhému z rodičovských chromozómů [32].

Nejprve se kříží (vyměňují) podřetězce definované pomocí bodů křížení a ostatní geny v nově vytvořených chromozómech jedinců zůstanou neobsazené. S touto výměnou je třeba si uložit jednotlivé geny, které se vzájemné výměny chromozómů rodičů zúčastnily, to nazýváme forma přepisovacích pravidel. Zbytek genů je prozatím neznámý a označen symbolem „?“. V dalším kroku se doplní do obou chromozómů z rodičů ty geny, které nepůsobí ve vznikání permutačních chromozómů potomků žádný konflikt. Ve finále se použijí všechna nadefinovaná přepisovací pravidla k obsazení zbylých volných míst. Postup je vysvětlen na obrázku Obrázek 4.10: Křížení s částečným přiřazením. pro body křížení 3 a 6.



Obrázek 4.10: Křížení s částečným přiřazením.

Jak je vidět ve výše uvedeném příkladu, pro chromozóm prvního potomka vybíráme čísla napravo od šipky v použitých přepisovacích pravidlech. Pro druhého potomka vybíráme číslo nalevo od šipek, přičemž číslo 3 je v obou případech ignorováno, protože ho obsahují oba rodičovské chromozómy. Detailněji popsáno v [03].

4.2.2.2 Křížení s rekombinací hran - ERX (Edge Recombination Crossover)

Tento operátor je nejúspěšnější operátor pro řešení úlohy obchodního cestujícího. Patří mezi složitější operátory, a proto ho popíši rovnou na příkladu. Máme dva rodičovské chromozomy, k jejichž jednotlivým genům si vypíšeme jejich sousedy (pro každý gen z obou chromozómů) [32].

Jedinec číslo	Chromozóm jedince
1	(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
2	(8, 2, 1, 9, 6, 7, 4, 5, 0, 3)

Gen	Sousedé genu z obou chromozómů		Odstranění duplicity sousedních genů
0	1, 9, 5, 3		1, 9, 5, 3
1	0, 2, **2**, 9		0, 2, 9
2	1, 3, 8, **1**		1, 3, 8
3	2, 4, 0, 8		2, 4, 0, 8
4	3, 5, 7, **5**	→	3, 5, 7
5	4, 6, 4, 0	→	4, 6, 0
6	5, 7, 9, **7**		5, 7, 9
7	6, 8, **6**, 4		6, 8, 4
8	7, 9, 2, 3		7, 9, 2, 3
9	8, 0, 1, 6		8, 0, 1, 6

Obrázek 4.11: Příklad operátoru ERX, první krok.

Nový jedinec je konstruován tak, že pro každý gen jsou náhodně vybíráni jeho sousedi z jeho seznamu sousedů, přičemž přednost má vždy ten soused, který má svůj seznam sousedů nejkratší. První gen nového jedince bude tedy také náhodně vybrán z genů, které mají nejméně sousedů. Náhodně vybereme tedy první gen 4 do první pozice v novém chromozómu jedince.

Nový chromozóm	Geny	Sousedé		Nový chromozóm	Geny	Sousedé
(**4**, _, _, _, _, _, _, _, _, _)	0	1, 9, 5, 3		(4, **5**, _, _, _, _, _, _, _, _)	0	1, 9, 3
	1	0, 2, 9			1	0, 2, 9
	2	1, 3, 8			2	1, 3, 8
	3	2, 4, 0, 8			3	2, 0, 8
	4	3, **5**, 7	→			
	5	4, 6, 0	→		**5**	**6**, 0
	6	5, 7, 9			6	7, 9
	7	6, 8, 4			7	6, 8
	8	7, 9, 2, 3			8	7, 9, 2, 3
	9	8, 0, 1, 6			9	8, 0, 1, 6

Obrázek 4.12: Příklad operátoru ERX, navazující krok.

Jak je vidět, tak vždy po vybrání genu do chromozómu odstraníme tento gen z tabulky genů (včetně seznamu jeho sousedů) a zároveň ze seznamu sousedů všech genů. Tímto způsobem se nám zkracuje tabulka genů k výběru a také seznam sousedů zbylých genů.

Nový chromozóm	Geny	Sousedé		Nový chromozóm	Geny	Sousedé
(4, 5, **6**, _, _, _, _, _, _, _)	0	1, 9, 3		(4, 5, 6, **7**, _, _, _, _, _, _)	0	1, 9, 3
	1	0, 2, 9			1	0, 2, 9
	2	1, 3, 8			2	1, 3, 8
	3	2, 0, 8	→		3	2, 0, 8
	6	**7**, 9	→			
	7	8			**7**	**8**
	8	7, 9, 2, 3			8	9, 2, 3
	9	8, 0, 1			9	8, 0, 1

Nový chromozóm	Geny	Sousedé		Nový chromozóm	Geny	Sousedé
(4, 5, 6, 7, **8**, _, _, _, _, _)	0	1, 9, 3		(4, 5, 6, 7, 8, **9**, _, _, _, _)	0	1, 3
	1	0, 2, 9			1	0, 2
	2	1, 3	→		2	1, 3
	3	2, 0	→		3	2, 0
	8	**9**, 2, 3				
	9	0, 1			**9**	**0**, 1

Nový chromozóm	Geny	Sousedé		Nový chromozóm	Geny	Sousedé
(4, 5, 6, 7, 8, 9, **0**, _, _, _)	**0**	1, **3**		(4, 5, 6, 7, 8, 9, 0, **3**, _, _)		
	1	2	→		1	2
	2	1, 3	→		2	1
	3	2			**3**	**2**

Nový chromozóm	Geny	Sousedé		Nový chromozóm	Geny	Sousedé
(4, 5, 6, 7, 8, 9, 0, 3, **2**, _)	1	-	→	(4, 5, 6, 7, 8, 9, 0, 3, 2, **1**)	**1**	-
	2	**1**	→			

Rodičovské chromozómy	Nový chromozóm potomka
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9) (8, 2, 1, 9, 6, 7, 4, 5, 0, 3)	(4, 5, 6, 7, 8, 9, 0, 3, 2, 1)

Obrázek 4.13: Příklad operátoru ERX, zobrazení zbývajících kroků a výsledek.

Jak je vidět, při použití operátoru ERX nám ze dvou rodičovských chromozómů vznikne pouze jeden potomek.

4.2.3 Křížení při využití kódování s reálnými hodnotami (Real Valued Crossover)

Tři následující operátory, uvedené v této sekci, mohou být použity pouze pro křížení chromozómů obsahujících reálné hodnoty. Celá kapitola je detailněji popsána v [20].

4.2.3.1 Střední křížení (Intermediate Recombination)

Hodnoty (alely) genů chromozómů potomka jsou vybírány z oblasti hodnot (alel) mezi geny chromozómů rodičů. Potomci se generují podle následujícího pravidla [20]:

$$\text{Potomek} = \text{Rodič2} + \alpha \cdot (\text{Rodič1} - \text{Rodič2}) \qquad (4.5)$$

Kde α je stupňující faktor, jednotně náhodně vygenerovaný na intervalu $[-d, 1 + d]$. Pro střední křížení je $d = 0$. Každý gen v chromozómu potomka je výsledek kombinování hodnot genů rodičů za použití vždy nově vygenerovaného α. Hodnota parametru d definuje velikost oblasti (prostoru) pro případné potomky. Hodnota $d = 0$ tedy definuje prostor pro potomky stejné velikosti, jaká byla oblast rodičů.

Tato metoda se tedy nazývá střední nebo standardní či průběžné křížení. Vzhledem k tomu, že většina genů potomka není generována na hranici možného prostoru, tak se prostor pro nové geny smršťuje v průběhu následných generací. K tomuto smršťování dochází pouze při použití průběžné rekombinace a můžeme mu lehce předejít nastavením vyšší hodnoty pro d. Hodnota $d = 0.25$ zajišťuje (statisticky), že variabilní prostor potomků je stejný jako variabilní prostor rodičů.



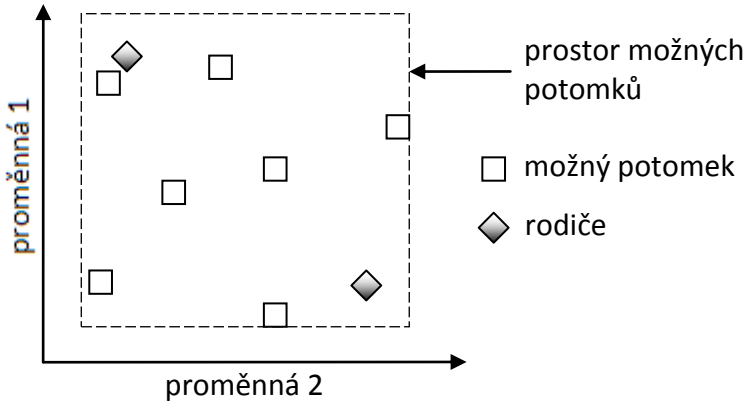
Obrázek 4.14: Zobrazení prostoru rodičů a potomků.

Následuje příklad, na kterém bude průběžné křížení prezentováno.

Rodičovské chromozómy	Náhodně vygenerované α		Chromozómy potomků
(24, 18, 7, 14, 128)	(-0.1, 0.5, 0,8, 0.1, 1.01)	→	(105.4, 37, 32, -24, 129.27)
(98, 56, 132, 24, 1)	(0.2, 0.4, -0.1, 0.8, 0.9)	→	(38.8, 33.2, -5.5, 22, 13.7)

Obrázek 4.15: Použití středního křížení.

Střední křížení je schopné produkovat potomky v prostoru nepatrně větším, než byl prostor rodičů. Tento prostor je zobrazen na následujícím obrázku.



Obrázek 4.16: Zobrazení dostupného prostoru pro potomky od rodičů.

4.2.3.2 Liniové křížení (Line Recombination)

Liniové křížení je velice podobné střednímu křížení pouze s tím rozdílem, že se používá shodný stupňující faktor pro všechny geny chromozómu. Používá se stejný vzorec pro generování potomků [20]:

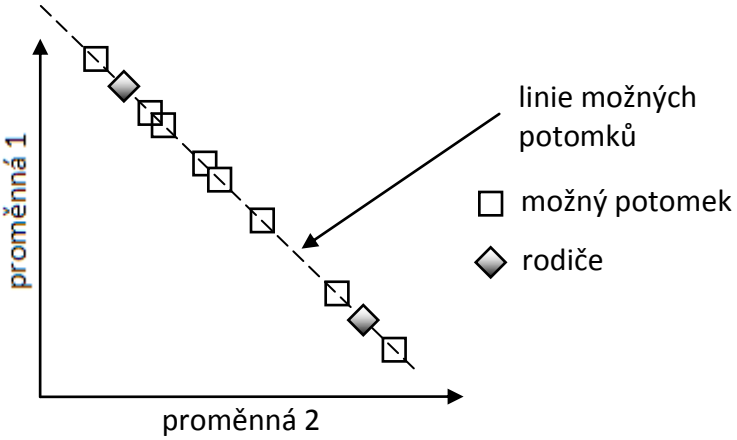
$$\text{Potomek} = \text{Rodič2} + \alpha \cdot (\text{Rodič1} - \text{Rodič2}) \qquad (4.6)$$

I u tohoto druhu křížení je uveden příklad na obrázku, kde $d = 0.25$.

Rodičovské chromozómy	Náhodně vygenerované α		Chromozómy potomků
(24, 18, 7, 14, 128)	0.3	→	(75.8, 44.6, 94.5, 21, 39.1)
(98, 56, 132, 24, 1)	0.8	→	(83.2, 48.4, 107, 22, 26.4)

Obrázek 4.17: Liniové křížení, při stejném α.

Liniové křížení může vygenerovat potomka kdekoli na lince, která je určena propojením obou rodičů, jak ukazuje obrázek Obrázek 4.18.



Obrázek 4.18: Liniové křížení, možnost vygenerování nových potomků.

4.2.3.3 Rozšířené liniové křížení (Extended Line Recombination)

Tento typ křížení, stejně jako výše uvedené, definuje nově vzniklé jedince na linii dané oběma rodiči. Nicméně se neomezuje pouze na danou linii mezi oběma rodiči a malý prostor mimo ně. Rodiče pouze definují onu linii, na které mohou noví jedinci vznikat. Velikost prostoru pro případné potomky je definována v oblasti proměnných.

Na linii udané rodiči však není rovnoměrně rozdělena pravděpodobnost vzniku potomka po celém prostoru. Máme vysokou pravděpodobnost vzniku potomka blízko u rodičů. Oproti tomu potomci vznikají s velmi malou pravděpodobností daleko od rodičů. Pokud je k dispozici ohodnocení obou rodičů, tak jsou častěji vytvářeni potomci ve směru od rodiče s horším ohodnocením k rodiči s větším ohodnocením (tzv. směrované rozšířené liniové křížení). Potomci se vytváří podle vzorce [20]:

$$\mathbf{Potomek1_i} = \mathbf{Rodič1_i} + \boldsymbol{\delta_i} \cdot \boldsymbol{\lambda_i} \cdot \boldsymbol{\tau_i} \cdot \frac{\mathbf{Rodič2_i} - \mathbf{Rodič1_i}}{\|\mathbf{Rodič1} - \mathbf{Rodič2}\|} \tag{4.7}$$

$$\mathbf{Potomek2_i} = \mathbf{Rodič2_i} + \boldsymbol{\delta_i} \cdot \boldsymbol{\lambda_i} \cdot \boldsymbol{\tau_i} \cdot \left(-\frac{\mathbf{Rodič2_i} - \mathbf{Rodič1_i}}{\|\mathbf{Rodič1} - \mathbf{Rodič2}\|}\right) \tag{4.8}$$

Kde parametry nabývají následujících hodnot:

$\boldsymbol{\delta} = \mathbf{2^{-k \cdot u}}$	*δ* definuje relativní velikost skoku, pro všechna *i* je identická
$\boldsymbol{k} \in \{\mathbf{4}, \mathbf{5}, \mathbf{6} \ldots \mathbf{20}\}$	*k* definuje preciznost rekombinačního kroku
$\boldsymbol{u} \in [\mathbf{0}, \mathbf{1}]$	*u* je náhodné, v rozmezí 0 až 1
$\boldsymbol{\delta_{max}} = \mathbf{1}$	maximální hodnota *δ*
$\boldsymbol{\delta_{min}} = \mathbf{2^{-20}} = \mathbf{9.5367^{-7}}$	minimální hodnota *δ*
$\boldsymbol{\lambda} = \mathbf{r} \cdot \mathbf{o}$	*λ* definuje maximální velikost skoku, *o* je oblast působnosti
$\boldsymbol{r} \in [\mathbf{0.00}, \mathbf{1}]$	r definuje rozmezí rekombinačního kroku, r je nabývá hodnot v rozmezí 0 až 100 procent
$\boldsymbol{\tau} \in \{-\mathbf{1}, \mathbf{1}\}$	*τ* definuje směr skoku
$\boldsymbol{\tau} = -\mathbf{1}$	s pravděpodobností menší než 0.5 pro nesměrované křížení
$\boldsymbol{\tau} = \mathbf{1}$	s pravděpodobností 0.5 a vyšší pro směrované křížení

Dvojité „|| ….. ||“ reprezentují eukleidovskou normu vektoru, která udává vzdálenost od počátku. Pro následující příklad je eukleidovská norma vektoru rovna 116.6319.

$$\begin{aligned} \text{Rodič1} &= (24, 18, 7, 14, 128) \\ \text{Rodič2} &= (98, 56, 132, 24, 1) \\ \|\text{Rodič1} - \text{Rodič2}\| &= \sum_{i=1}^{5} \sqrt{|\text{Rodič1}_i - \text{Rodič2}_i|} = 116.6319 \end{aligned} \tag{4.9}$$

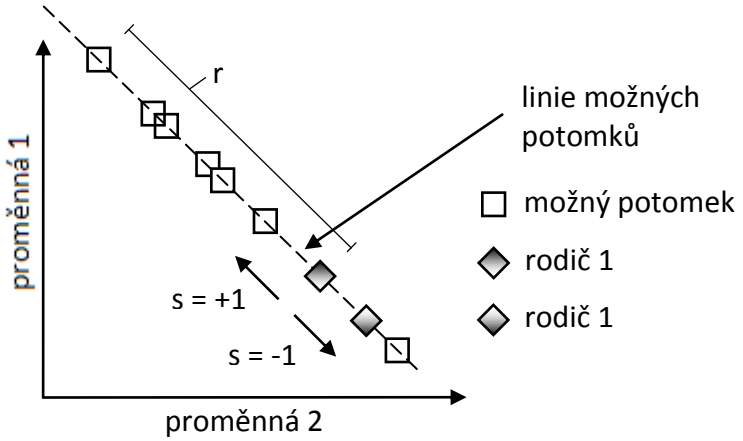
$$k = 8 \quad u = 0.16 \quad \delta = 2^{-8 \cdot 0.16} = 0.4117955086 \quad \tau = 1$$

$$o_i = 400 \quad r_i = 0.06 \quad \lambda_i = 0.06 \cdot 400 = 24 \quad i = 1..5$$

Rodičovské chromozómy		Chromozómy potomků
(24, 18, 7, 14, 128)	→	(30.27, 21.22, 17.59, 14.85, 117.24)
(98, 56, 132, 24, 1)	→	(104.27, 59.22, 142.59, 24.85, -9.76)

Obrázek 4.19: Rozšířené liniové křížení.

Rozšířené liniové křížení může tedy vygenerovat potomka kdekoli v oblasti působnosti, na přímce protínající oba rodiče v *i*-rozměrném prostoru.



Obrázek 4.20: Rozšířené liniové křížení, možnost vygenerování nových potomků.

4.3 Operátory mutace

Mutace rozšiřuje prohledávaný prostor o řešení, které není možno dosáhnout křížením, protože nebyla součástí inicializace populace. Extrémním případem takové situace je například stav, kdy se do první, náhodně vylosované generace dostane jeden velmi silný řetězec a zbytek řetězců bude slabý. V takovém případě může nastat situace, kdy se do další generace dostane nebo dostanou pouze exempláře jediného silného řetězce. Nastane-li takováto situace, můžeme dostat jiný řetězec pouze tak, že některý z exemplářů zmutujeme. Tato funkce mutace funguje i v neextrémních případech, kdy nám pomáhá udržet různorodost generace. Mutace se obecně aplikuje na právě vytvořené potomky s pravděpodobností nejčastěji okolo pěti procent [03].

4.3.1 Binární operátory mutace

Pro binárně reprezentované jedince znamená mutace přehození alely genů z hodnoty 1 na hodnotu 0 a naopak. Jelikož gen při binárním kódování vždy nabývá hodnoty 1 nebo 0, tak je každý mutační skok roven hodnotě jedna. V následném příkladu je zobrazen chromozóm, jehož každý gen projde mutací s pětiprocentní pravděpodobností, dejme tomu, že čtvrtý gen je zmutován [03].

Chromozóm před mutací: (0, 1, 0, **0**, 1, 0, 1, 1, 0, 1)
Chromozóm po mutaci: (0, 1, 0, **1**, 1, 0, 1, 1, 0, 1)

Obrázek 4.21: Binární mutace genu.

Rozsah mutací je možno zvětšovat u stagnujících generací, aby bylo opuštěno případné lokální maximum. Četnost mutací můžeme zvětšovat po určitém počtu stagnujících generací. Po zlepšení generace se koeficient mutace vrací zpět na původní hodnotu. Také lze zavádět takzvanou dynamickou mutaci, to znamená, že při inicializaci je koeficient nastavován na vyšší hranici, která s počtem generací klesá. To lze interpretovat, že z počátku dáváme přednost šířce prohledávaného prostoru, ale po určitém počtu generací se snažíme spíše prohledávat blízká okolí nejlepších jedinců.

Populace z obrázku Obrázek 4.22 nejsou schopny dosáhnou pomocí žádného operátoru křížení optimálního řešení, které má na čtvrté pozici jedničku. Toto řešení můžeme dosáhnout pouze pomocí mutace.

Jedinec č.	Chromozómy jedinců
1	(1, 0, 0, **0**, 1, 1, 0)
2	(1, 0, 1, **0**, 0, 1, 0)
3	(0, 1, 1, **0**, 1, 1, 1)
4	(1, 1, 0, **0**, 1, 1, 0)
5	(1, 0, 0, **0**, 1, 1, 1)
6	(1, 1, 1, **0**, 1, 1, 1)

Obrázek 4.22: Populace, jež není schopna dosáhnout optimálního řešení s jedničkou na čtvrté pozici pouze s použitím rekombinace, bez mutace.

4.3.2 Reálné a celočíselné operátory mutace

Při kódové reprezentaci pomocí reálných čísel se mohou operátory mutace snadno modifikovat. Máme celou škálu operátorů mutace pro práci s reálným kódováním. Mezi základní operátory mutace s reálnými čísly patří [03]:

- ✓ Náhodná výměna : Nahrazení původní alely náhodně vygenerovanou hodnotou
- ✓ Aditivní změna : Přičtení nebo odečtení náhodně vygenerované hodnoty k původní alele.
- ✓ Multiplikativní změna : Vynásobení nebo vydělení hodnoty náhodně vygenerovaným číslem.

Způsob mutace	Původní chromozóm		Zmutovaný chromozóm
Náhodná výměna **1.44**	(1.77, **3.14**, 2.15, 4.88, 5.56)	→	(1.77, **1.44**, 2.15, 4.88, 5.56)
Aditivní změna **+2.12**	(1.77, 3.14, 2.15, **4.88**, 5.56)	→	(1.77, 3.14, 2.15, **7.00**, 5.56)
Aditivní změna **-4.15**	(1.77, 3.14, **2.15**, 4.88, 5.56)	→	(1.77, 3.14, **-2.00**, 4.88, 5.56)
Multiplikativní změna ***2**	(**1.77**, 3.14, 2.15, 4.88, 5.56)	→	(**3.54**, 3.14, 2.15, 4.88, 5.56)
Multiplikativní změna **/2**	(1.77, 3.14, 2.15, 4.88, **5.56**)	→	(1.77, 3.14, 2.15, 4.88, **2.78**)

Obrázek 4.23: Tři základní způsoby mutace s reálnými čísli.

Ve sloupci způsobu mutace je vždy uvedena náhodně vygenerovaná hodnota jako přívlastek názvu. Z těchto operátorů můžeme použít náhodnou výměnu, aditivní změnu a multiplikativní změnu (pouze násobení) i u celočíselného kódování za předpokladu použití celočíselné mutační změny.

4.3.3 Výměnná mutace (Swap Mutation)

Při použití permutačního kódování nesmí být v chromozómu žádné dva geny, které by měly alely stejných hodnot. Je zřejmé, že náhodná změna jednoho čísla v řetězci by způsobila značné problémy, protože původní číslo by z permutace vypadlo a jiné by se v ní zřejmě vyskytovalo dvakrát (za předpokladu že nová hodnota mutace bude přípustná). Při permutačním kódování musí být po mutaci tedy zachován permutační typ rodiče u potomka. Řešení je jednoduché, zvolíme dva geny v chromozómu, jejichž alely se mutací vymění [03].

Původní chromozóm		Zmutovaný chromozóm
(0, 1, 2, **3**, 4, 5, 6, **7**, 8, 9)	→	(0, 1, 2, **7**, 4, 5, 6, **3**, 8, 9)
(0, 1, **2**, 3, 4, **5**, 6, 7, 8, 9)	→	(0, 1, **5**, 3, 4, **2**, 6, 7, 8, 9)
(A, B, **C**, D, E, F, **G**, H, I, J)	→	(A, B, **G**, D, E, F, **C**, H, I, J)
(**A**, B, C, D, E, F, G, H, I, **J**)	→	(**J**, B, C, D, E, F, G, H, I, **A**)

Obrázek 4.24: Výměnná mutace při použití permutačního kódování.

Jak je z příkladu jasné, tento druh mutace je možné použít na všechny typy kódování (binární, reálné, celočíselné i permutační).

4.3.4 Posuvná mutace (Shift mutation)

Tento druh mutace lze využít při všech výše zmíněných typech kódování jedinců (chromozómů). Náhodně zvolíme gen v chromozómu, který následně vyjmeme z jeho pozice a vložíme na náhodně vybranou (jinou, než původní) pozici. Tento způsob mutace nám tedy posune pozice genů od místa, kam vkládáme do místa, odkud bereme. Příklady pro uvedené druhy kódování jsou na obrázku Obrázek 4.25, kde jsou podtržením kurzívou vyznačeny posunuté geny a tučně přesouvaný gen [20].

Reprezentace kódování	Původní chromozóm		Zmutovaný chromozóm
Binární kódování	(0, 1, 1, 0, 0, 1, 1, **0**, 0, 0)	→	(0, 1, 1, 0, **0**, *0*, *1*, *1*, 0, 0)
Permutační kódování	(A, B, C, D, E, F, G, **H**, I, J)	→	(A, B, **H**, *C*, *D*, *E*, *F*, G, I, J)
Reálné kódování	(1.44, 2.18, **4.11**, 1.31)	→	(1.44, **4.11**, *2.18*, 1.31)

Obrázek 4.25: Posuvná mutace.

4.3.5 Inverzní mutace (Inverse Mutation)

Inverzní mutace lze také využít při všech výše zmíněných typech kódování jedinců (chromozómů). Náhodně zvolíme dva geny v chromozómu. Úsek mezi zvolenými geny, včetně těchto genů, vložíme invertovaný. Tento druh mutace je tedy použitelný i pro permutační kódování. Příklady jsou uvedeny na obrázku Obrázek 4.26. Invertovaná část je zvýrazněna tučně [20].

Reprezentace kódování	Chromozómy před mutací		Chromozómy po mutaci
Binární kódování	(1, 0, **1**, **1**, **0**, **1**, **0**, 0, 1)	→	(1, 0, **0**, **1**, **0**, **1**, **1**, 0, 1)
Celočíselné kódování	(4, 5, 1, 1, 8, 4, **9**, **8**, 6)	→	(4, 5, 1, 1, 8, 4, **8**, **9**, 6)
Permutační kódování	(**A**, **B**, **C**, **D**, E, F, G, H, I)	→	(**D**, **C**, **B**, **A**, E, F, G, H, I)
Reálné kódování	(2.14, 4.84, **9.45**, **3.11**)	→	(2.14, 4.84, **3.11**, **9.45**)

Obrázek 4.26: Inverzní mutace.

4.4 Doplnění, znovuvložení (Reinsertion)

Poté, co jsou vyprodukovány chromozómy nových jedinců pomocí selekce, křížení a mutace přichází na řadu doplnění neboli znovuvložení (reinsertion). Pokud je nově vyprodukovaných jedinců méně než v předchozí populaci, musí se do nové populace doplnit, aby měla stejný počet jedinců jako předcházející. Stejně tak, pokud je nových jedinců vyprodukováno více, musí se použít tento mechanizmus k výběru jedinců, co budou v nové populaci zahrnuti. Přebyteční, nově vygenerovaní jedinci, budou zapomenuti. Existují čtyři základní typy globálního doplnění [20].

Ryzí doplnění (Pure Reinsertion)

Je vyprodukováno přesně tolik nový jedinců, jako bylo v předchozí populaci. Je tedy vytvořena zcela nová populace. Ze staré populace není do nové vkládán žádný jedinec. Každý jedinec žije pouze jednu generaci.

Uniformní doplnění (Uniform Reinsertion)

Bylo vytvořeno méně jedinců, než obsahovala předchozí generace. Nová generace se doplní náhodně vybranými jedinci z předchozí populace.

Elitní doplnění (Elitist Reinsertion)

Stejně, jako v předchozím případě, bylo vytvořeno méně jedinců než obsahovala předešlá generace. Nová populace se doplní nejlépe ohodnocenými jedinci z předchozí generace.

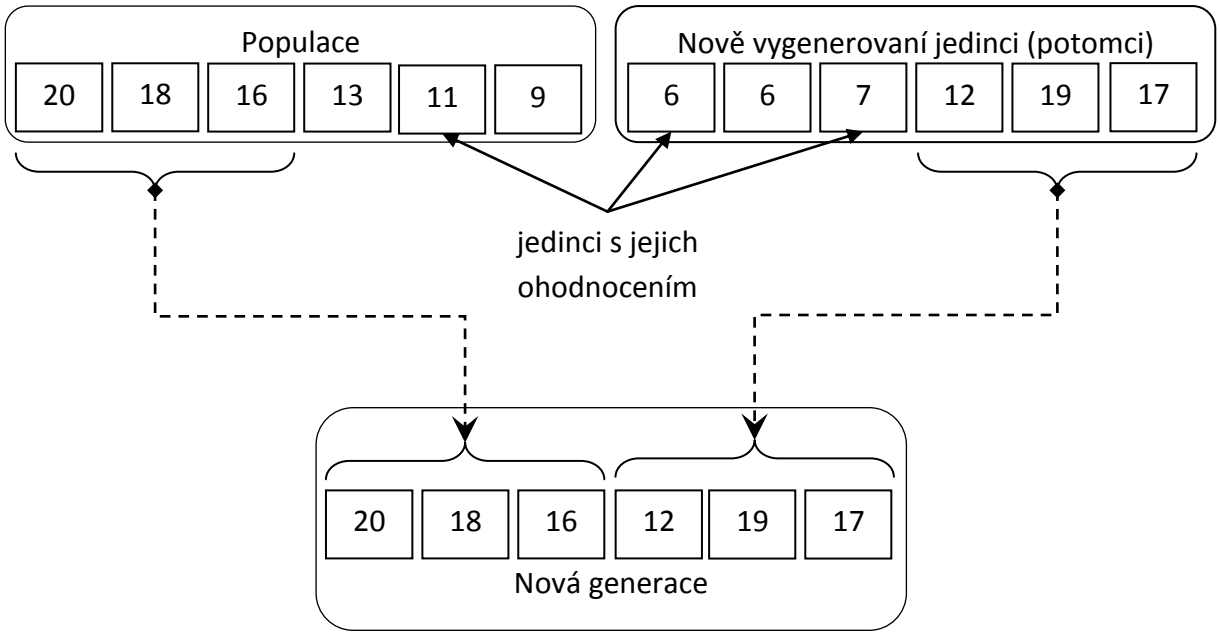
Doplnění závislé na ohodnocení (Fitness-based Reinsertion)

Vyprodukuje se více nových jedinců, než bylo v předchozí generaci. Do nové populace se zařadí na základě jejich ohodnocení. Nejhůře ohodnocení jedinci se nedostanou do nové populace.

Ryzí doplnění je to nejjednodušší, každý jedinec žije pouze jednu generaci. Využívá se především u jednoduchých genetických algoritmů. Avšak může nastat situace, kdy ztratíme jedince s velmi dobrým ohodnocením, aniž by vyprodukoval dobrého potomka, pak je pro nás důležitá informace obsažená v rodiči ztracena.

Prevenci ztráty této cenné informace poskytuje elitní doplnění v kombinaci s doplněním závislým na ohodnocení. Každá nová generace se skládá přibližně z poloviny nově vygenerovaných jedinců a z poloviny nejlepších jedinců předchozí generace. Schéma doplnění, závislém na ohodnocení aplikuje seříznutý výběr na nově vyprodukované jedince před tím, než je zařadí do populace. Každou novou generaci mohou být tedy jedinci ze starší generace nahrazeni nově vyprodukovanými jedinci nezávisle na jejich ohodnocení (nekontroluje se, zda nový jedinec s horším ohodnocením nahradil jedince z předchozí populace, který měl lepší ohodnocení). Díky elitismu a doplnění závislým na ohodnocení mohou jedinci s velmi dobrým ohodnocením přežívat mnoho generací, avšak mohlo by to být na škodu, pokud nevedou k řešení. Z tohoto důvodu je možné specifikovat životnost

každého jedince. Životnost jedince je dána operátorem „Smrt“, což je vlastně počítadlo, které udává, do kolika následujících generací může být zařazený tento jedinec bez obměny.



Obrázek 4.27: Elitní doplnění, závislé na ohodnocení, nové generace.

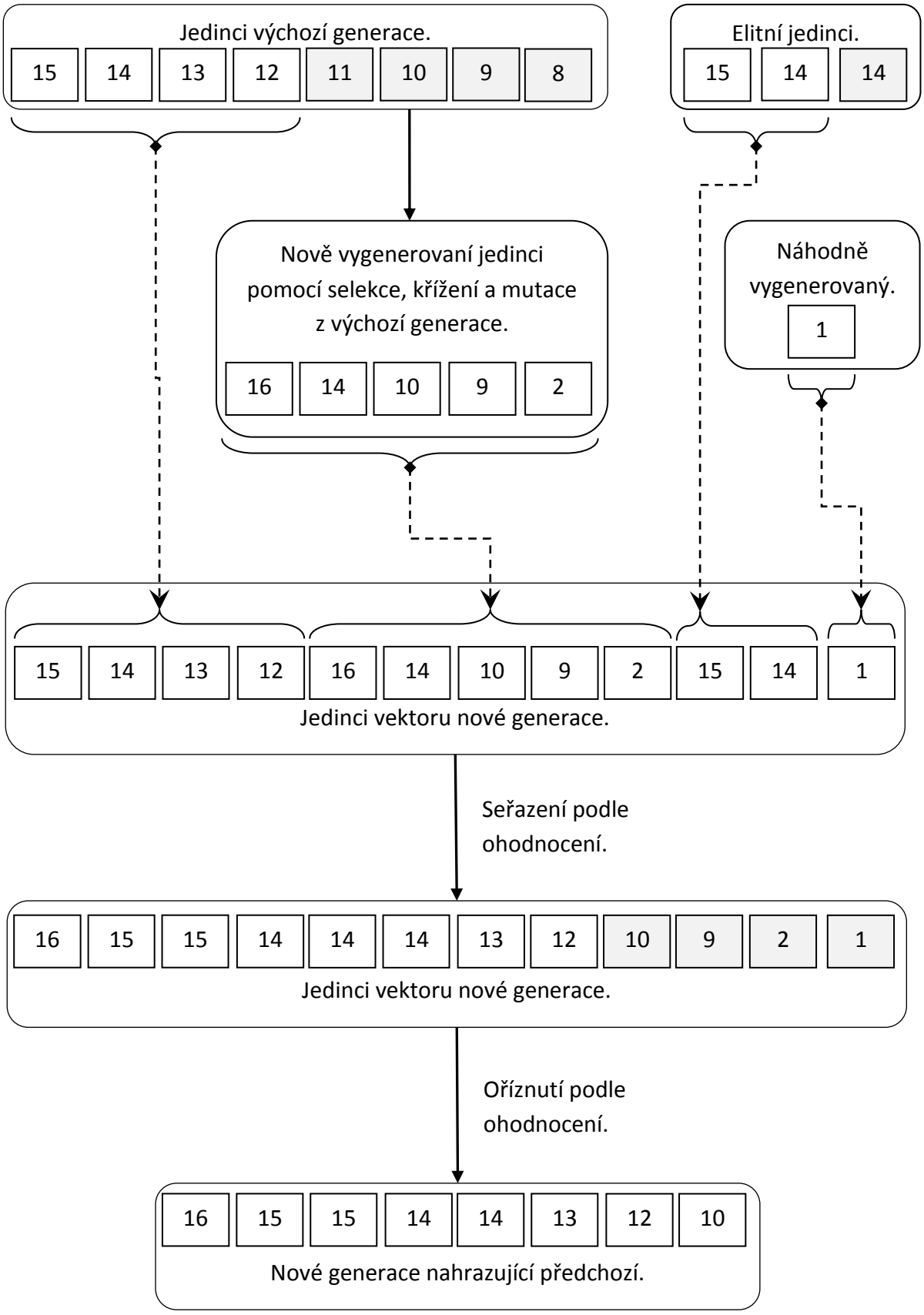
Naše knihovna má implementován univerzální typ znovuvložení. Můžeme zvolit tři základní parametry, podle kterých se vkládají jedinci do nově vytvářené generace:

- Počet jedinců z předchozí generace.
- Počet jedinců elitních jedinců nalezených po celou dobu evoluce odpovídající populace.
- Počet nově náhodně vygenerovaných jedinců.

U prvních dvou parametrů je volitelné zda použijeme výběr jedinců podle nejlepšího ohodnocení nebo je vybereme náhodně. Po vložení všech vybraných jedinců do vektoru reprezentujícího novou generaci dojde k jeho seřazení podle ohodnocení a následně se vektor seřízne na požadovanou velikost generace. Odstraněni budou tedy vždy jedinci s nejhorším ohodnocením. Pokud jsme do nové generace nechali vkládat nějaké nově náhodně vygenerované jedince je velice pravděpodobné, že odříznuti budou právě oni. Následuje příklad fungování našeho reinsertoru s danými parametry počtu jedinců:

- 4 z předchozí generace
- 2 elitní
- 1 nově, náhodně vygenerovaný

Přičemž pomocí selekce, křížení a mutace generujeme 5 nových jedinců a evoluce probíhá na populaci celkově s 8mi jedinci. Jedince vkládáme na základě ohodnocení jak z elitního vektoru, tak z předchozí generace. Elitní vektor obsahuje 3 jedince.



Obrázek 4.28: Postup implementovaného reinsertoru.

4.5 Paralelní genetické algoritmy

Paralelní genetické algoritmy byly vyvinuty s cílem urychlit výpočet výsledků. Existují dva základní populační modely paralelních genetických algoritmů.

4.5.1 Vícepopulační migrační model (Multipopulation Migration Model)

Vícepopulační model, též nazývaný jako migrační model spouští evoluci na více populacích zároveň. Tyto populace se vyvíjejí nezávisle na ostatních populacích po předem stanovený počet generací, tomuto počtu generací se říká izolační čas. Po vypršení izolačního času se určitý počet jedinců z každé populace přenese mezi ostatní populace, právě proto nazýváme tento model migrační, jelikož jsou jedinci migrují mezi populacemi [20].

Počet vyměněných jedinců, druh výběru (selekce) jedinců pro výměnu a druh populačního modelu určuje množství genetické rozmanitosti (která může nastat v populacích) a výměnu informací mezi populacemi.

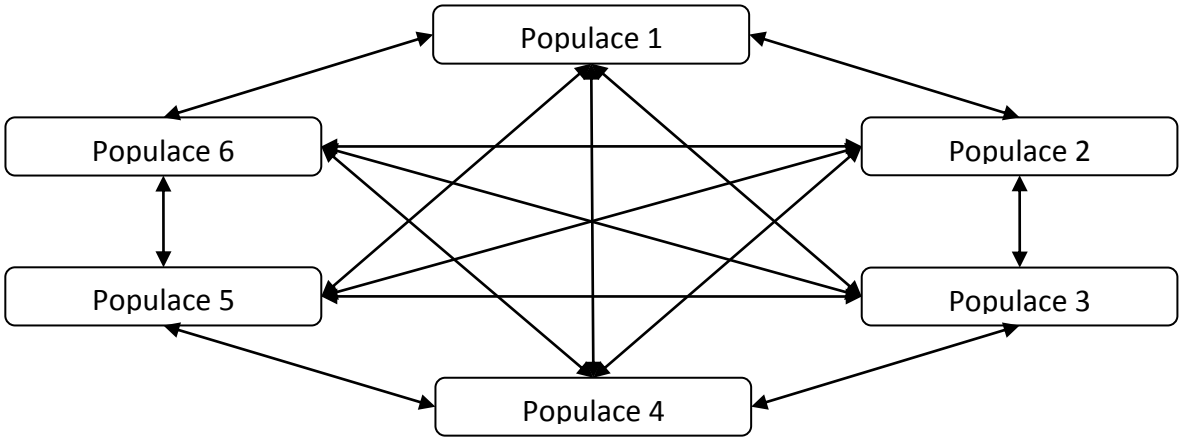
Paralelní implementace migračního modelu prokázala nejen zrychlení výpočtu při hledání optima, ale také nám stačí méně objektivní ohodnocující funkce ve srovnání s použitím pouze jedné populace. Jednoprocesorový počítač vykonávající paralelní algoritmus sériovým způsobem (pseudo-paralelní) přináší lepší výsledky s paralelizací (algoritmus najde globální optimum častěji, nebo rychleji) [20].

Výběr jedinců pro migraci může být pomocí následujících způsobů:

- Jednotně náhodně (výběr naprosto náhodných jedinců pro migraci).
- Založené na ohodnocení (migrují pouze ti nejlepší jedinci, obrázek Obrázek 4.30: Úplná migrace založená na ohodnocení.).

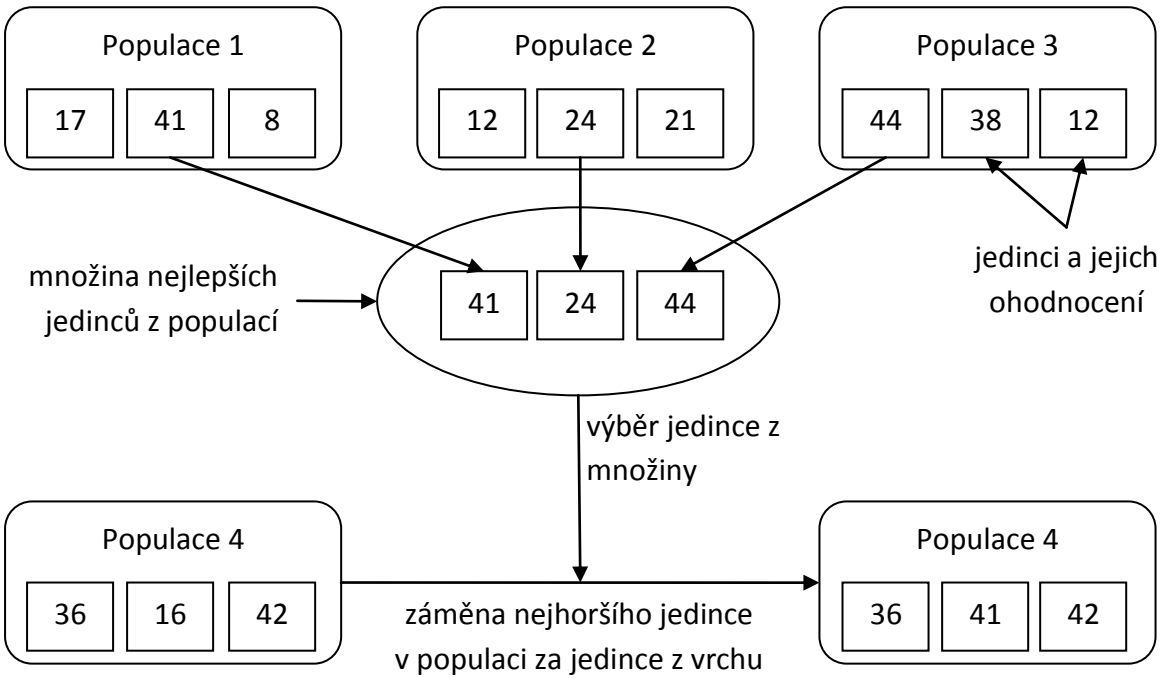
Migrace mohou probíhat následovně:

- Migrace mezi všechny populace (kompletní/úplná topologie sítě), dle obr. Obrázek 4.29: Úplná migrace s náhodným výběrem..
- Migrace do kruhu (topologie kruhu), dle obrázku Obrázek 4.31.
- Migrace do svého sousedství, dle obrázku Obrázek 4.32.



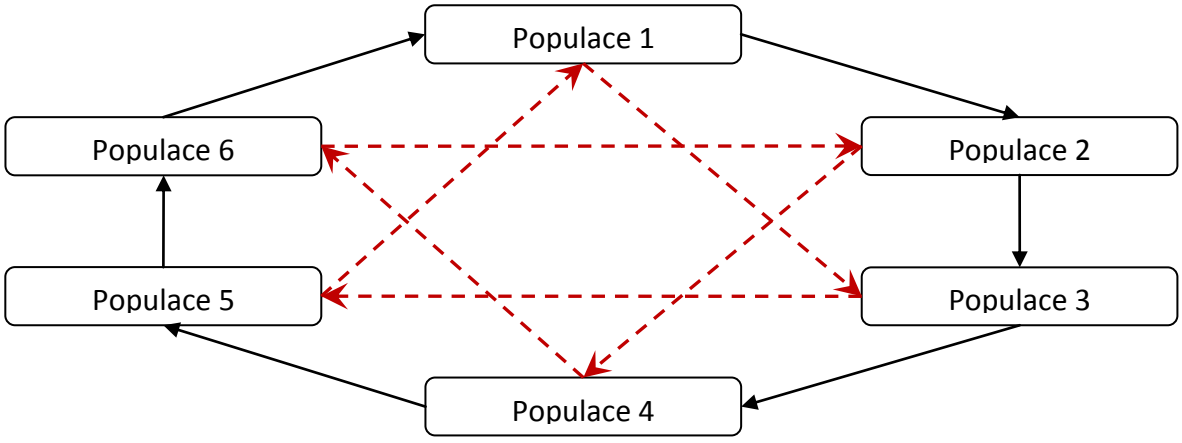
Obrázek 4.29: Úplná migrace s náhodným výběrem.

Obrázek Obrázek 4.29 znázorňuje úplnou migraci s náhodným výběrem, která patří k těm základním. Všichni jedinci každé populaci mohou migrovat do všech ostatních populací. Přičemž jedinci jsou vybíráni náhodně v každé populaci.



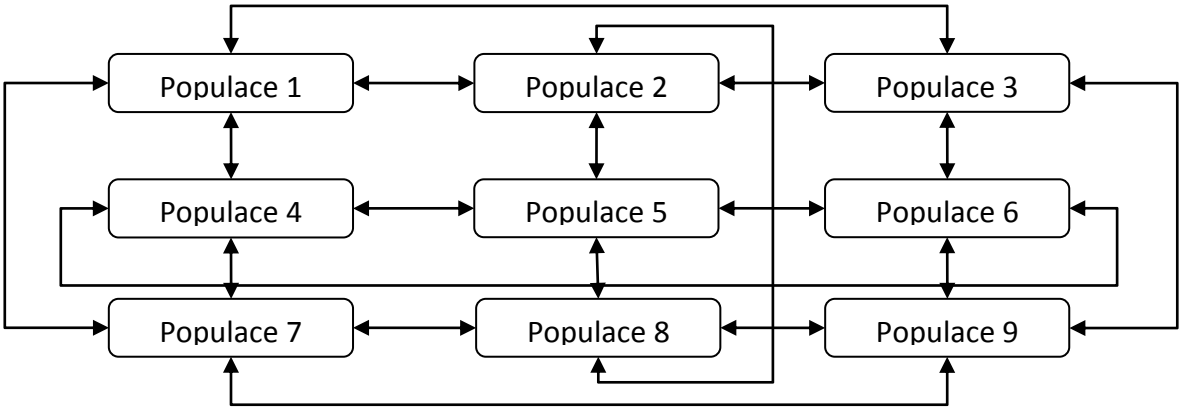
Obrázek 4.30: Úplná migrace založená na ohodnocení.

Na obrázku Obrázek 4.30 vidíme úplnou migraci založenou na ohodnocení. V dané populaci vždy vybereme jedince s nejhorším ohodnocením, který je nahrazen náhodně vybraným jedincem z množiny nejlepších jedinců ostatních populací. Tento cyklus se provádí pro každou populaci a tím je zajištěno, že žádná populace neobdrží při migraci svého jedince, kterého se snažila nahradit [20].



Obrázek 4.31: Migrace při využití kruhové topologie.

Obrázek Obrázek 4.32 popisuje využité nejzákladnější topologie migrace, a sice kruhové, neboli prstencové topologie. Z každé populace jsou jedinci přenášeni pouze do jedné, předem dané populace. Plné šipky naznačují přenos jedinců se vzdáleností jedna mezi populacemi. Šipky s přerušovanou čarou udávají přenos jedinců se vzdáleností dvě. Analogicky si odvodíme přenosy jedinců mezi vzdálenějšími populacemi.



Obrázek 4.32: Migrace při využití sousední topologie.

Sousední topologie je velice podobná kruhové topologii s tím rozdílem, že jedinci se mohou vyměňovat oboustranně mezi oběma sousedními populacemi. Výměna může nastat pouze mezi sousedními populacemi. Na obrázku je nakreslena 2D reprezentace devíti populací [20].

4.5.2 Hardwérově paralelizovaný model (Hardwadre Parallelized Model)

Tento model využívá základní paralelizace genetických algoritmů. Evoluce je počítána pomocí více vláken procesoru, díky tomu je možné zapojit do výpočtu více jader procesoru. Výkon jader nezůstává nevyužitý a z toho důvodu evoluce vypočítá více generací. Nutnost paralelní implementace často vyplývá z charakteru ohodnocující funkce, která zejména u reálných problémů bývá velmi náročná z hlediska potřeby výpočetního času. Při velkém množství jedinců, jež je nutné při běhu genetického algoritmu postupně ohodnotit, je zřejmé, že paralelizací této operace lze celkový čas potřebný pro běh algoritmu výrazně zkrátit. Dochází tedy k paralelizaci výpočtu na úrovni ohodnocující funkce, selekce, křížení a mutace.

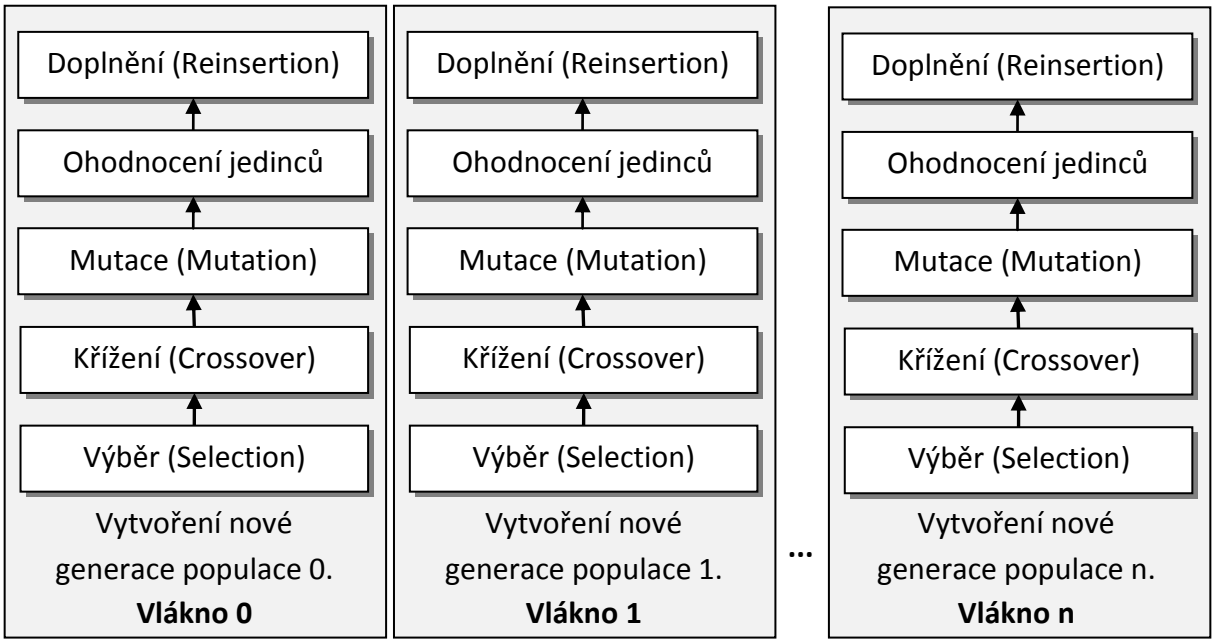
Naše knihovna využívá k paralelizaci genetického algoritmu již vytvořenou knihovnu OpenMP. Máme možnost zvolit ze tří vytvořených úrovní (nadále levelů) OpenMP, jak již bylo nastíněno v kapitole 3.5.

OpenMP Level 0

Podpora spuštění evoluce na více vláknech je při použití tohoto levelu vypnuta. Výpočet tedy provádí pouze jedno vlákno a vytíženo je pouze jedno jádro procesoru.

OpenMP Level 1

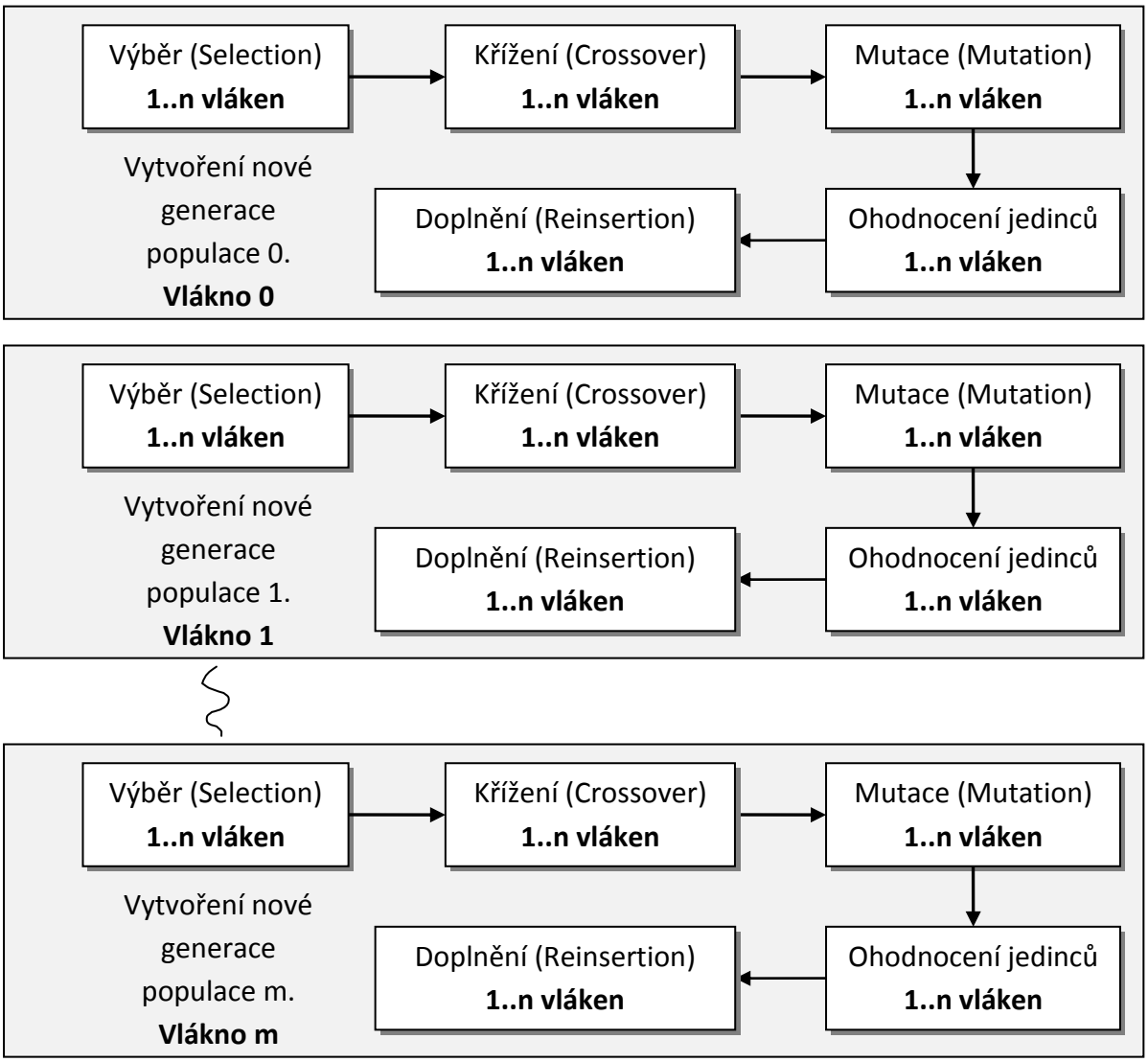
Tento level paralelizuje výpočty na úrovni populací. Pokud zapneme evoluci na OpenMP Levelu 1 spustí se pro výpočet evolucí tolik vláken kolik máme v genetickém algoritmu populací. V idealním případě se vyplatí provádět evoluci zároveň na tolika vláknech kolík máme počet jader na procesoru případně procesorech. Například pokud spustíme evoluci na dvou populacích a máme k dispozici quadcore procesor, stále budou dvě jádra nevyužitá. Spuštění evoluce na více populacích než máme jader procesoru je v pořádku avšak ztrácíme celkový výpočetní výkon se stoupajícím počtem populací, který je spotřebován na synchronizaci vláken mezi jádry procesoru. Evoluce populaci probíhá souběžně. Následuje vysvětlující obrázek.



Obrázek 4.33: Paralelní evoluce vice populací při použití OpenMP Level 1.

Open MP Level 2

Při použití OpenMP Level 2 je evoluce paralelizována na úrovni generace. Jako u předchozího OpenMP Levelu 1 se spustí stejný počet vláken jako je populací. Navíc jsou paralelizovány všechny operátory jako selekce, křížení, mutace a výpočet hodnotící funkce. OpenMP si samo určí kolik spustí subparalelních výpočetních vláken. Tento level je například výhodnější použít pokud bychom chtěli spustit evoluci dvou populací na quadcore procesoru. Díky subparalelizaci operátorů křížení a podobně budou využita všechny čtyři jádra procesoru. Při evoluci dvou populací s OpenMP Level 1 by byla využita pouze dvě jádra tohoto procesoru.



Obrázek 4.34: Paralelní evoluce více populací při použití OpenMP Level 2.

4.5.3 Migrace při použití hardwérově paralelizovaného modelu

Pokud chceme zároveň využívat více populací, které využívají k evoluci více jader procesoru vznikne nám problém s migrací. Populace můžou mít různý počet jedinců, mohou využívat jiné operátory například křížení či mutace. Je zřejmé, že některé populace budou mít evoluci generací rychlejší než ostatní, protože každý operátor spotřebovává jiný výpočetní výkon. Například pokud bychom spustili evoluci čtyř populací s izolačním časem sto generací, přičemž by u prvních dvou populací by trvala evoluce výrazně kratší dobu než u dalších dvou populací, tak by museli první dvě populace čekat než ostatní dvě dopočítají zbylý počet generací. Tento problém částečně řeší použití OpenMP Levelu 2, nikoli však celkově. Proto jsou implementovány tři následující nastavení migrace pro vícepopulační algoritmy u nichž je evoluce počítána více vlákny.

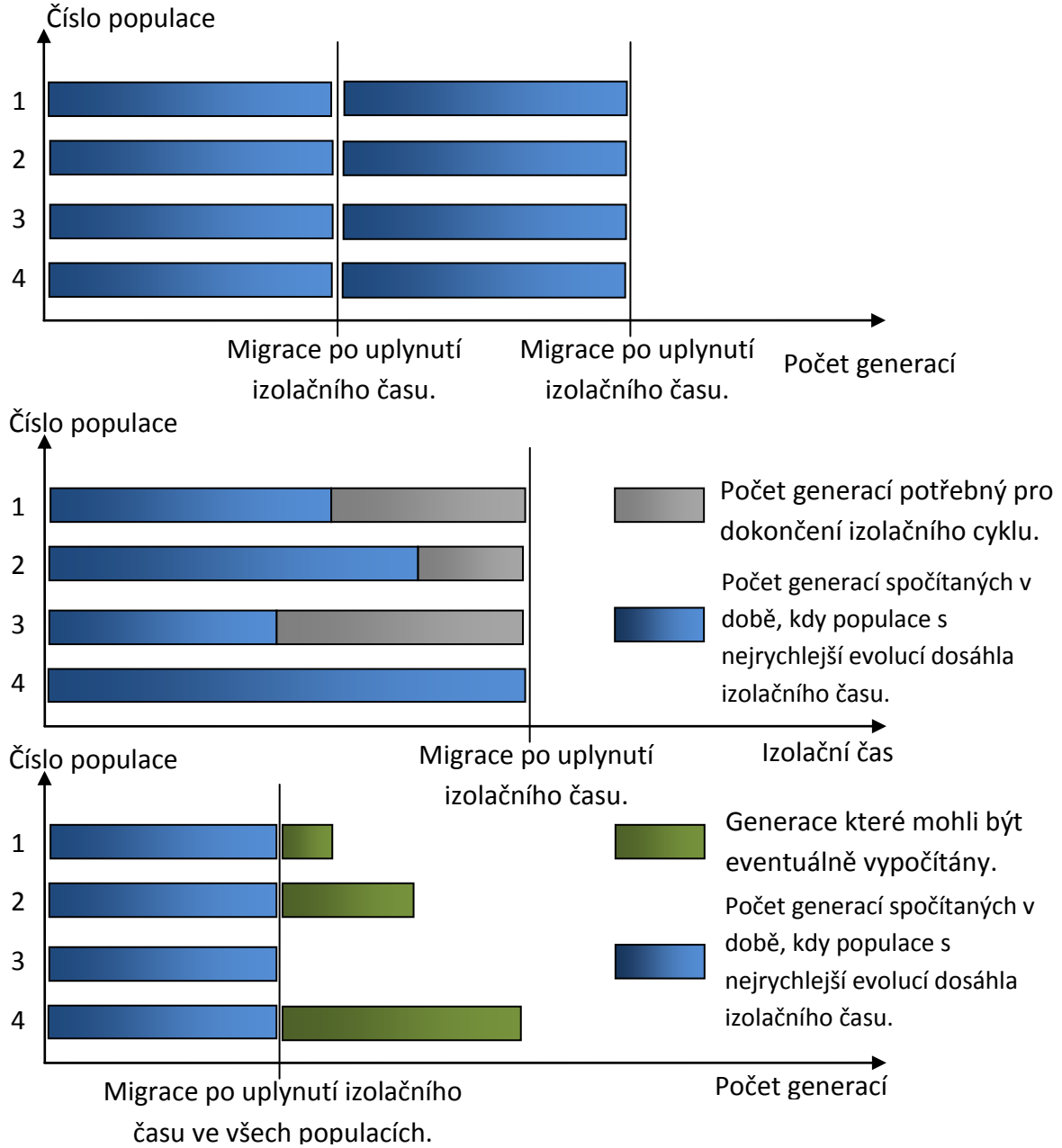
- Migrace je vypnuta.
- Migrace synchronní.
- Migrace asynchronní.

Vypnutá migrace

Při tomto nastavení neprobíhá žádná migrace jedinců mezi populacemi. Evoluce populací si jedou svým tempem a žádná na žádnou nečeká.

Synchronní migrace

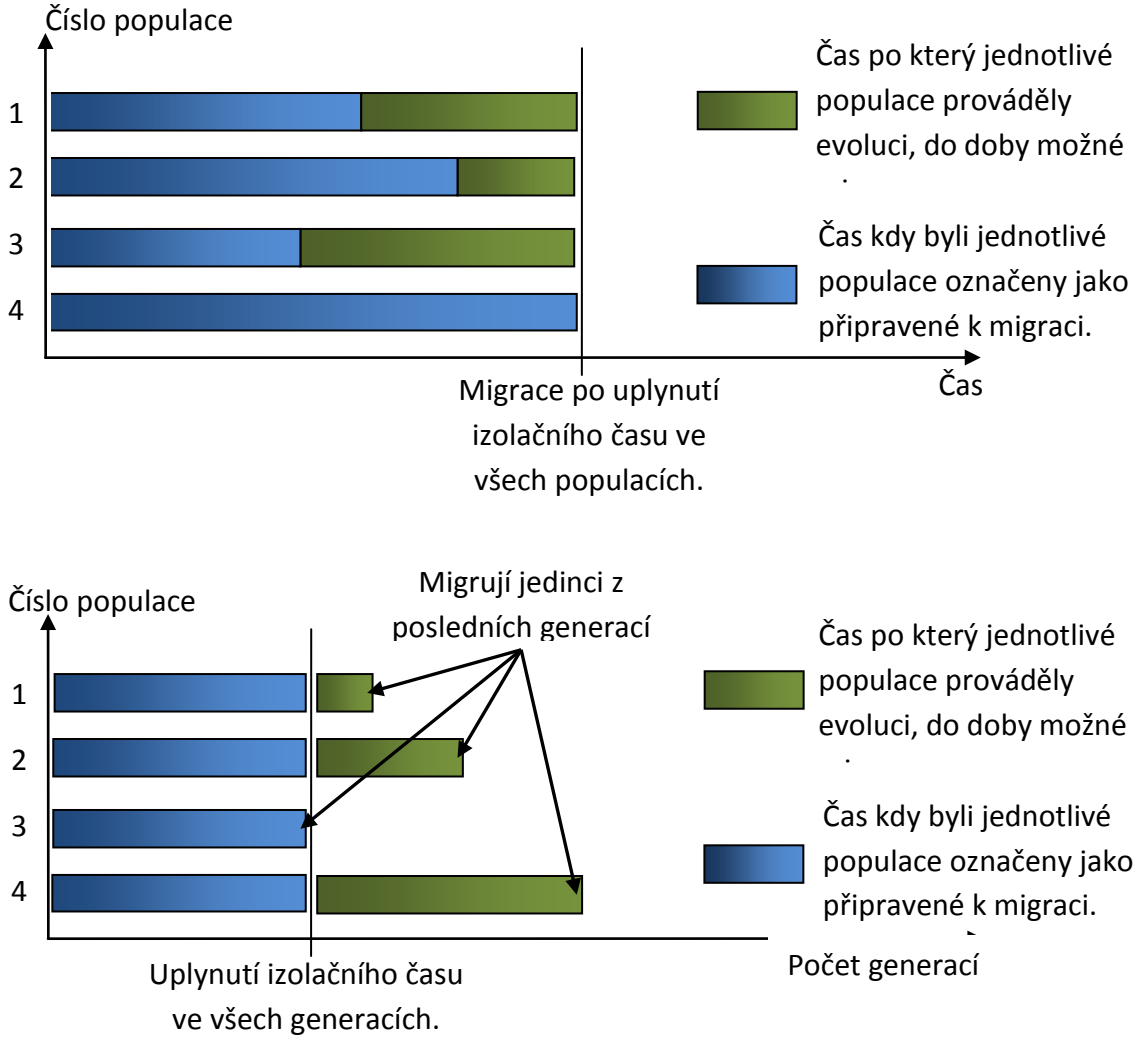
Při synchronní migraci se v každé populaci provede evoluce po dosažení izolačního času, který je dán určitým počtem generací. Po dosažení počtu generací se přesnou vybraní jedinci do migračního vektoru. Jedinci se vybírají podle nastavení buď náhodně nebo dle jejich ohodnocení. V době, kdy všechny populace dosáhly daného počtu generací, se provede migrace jedinců z migračního vektoru a začíná další cyklus evolucí pro generace, než všechny populace dosáhnou následné iterace izolačního času.



Obrázek 4.35 : Synchronní migrace při použití OpenMP Level 1.

Asynchronní migrace

U asynchronní migrace se evoluce jednotlivých populaci neohlíží na ostatní. Pokud jakákoli populace dosáhne izolačního času tak se označí jako připravená na migraci a její evoluce nadále pokračuje. V době, kdy i ta nejpomalejší populace dosáhne izolačního času, jsou již všechny ostatní označeny jako připravené na migraci a právě v tuto dobu se provede migrace.



Obrázek 4.36: Asynchronní migrace při použití OpenMP Level 1.

Asynchronní migrace se bude tedy nejvíce využívat v případě, kdy máme stejný počet populací (rozdílných) jako jader procesoru a chceme spustit OpenMP Level 1 paralelizaci. Na počet vytvořených generací bude vždy nejvýkonnější genetický algoritmus bez migrace. přičemž genetický algoritmus se synchronní migrací bude vždy méně výkonný než genetický algoritmus s asynchronní migrací.

Migraci provádí pouze jedno vlákno, nezáleží na OpenMP levelu, z důvodu paralelního přístupu k populacím (kritická sekce).

5 Testování vytvořené knihovny genetických algoritmů

Vytvořenou knihovnu budeme testovat na dvou matematických funkcích, které patří do množiny optimalizačních testovacích funkcí pro genetické algoritmy. Funkce jsou vizualizovány v prostředí MathWorks Matlab. Vizualizace je provedena vždy pro dvě osy, avšak řešení budeme hledat na větším počtu os, jde nám zejména o nastínění průběhu funkce.

U všech následujících matematických funkcí budeme hledat minimum na určitém intervalu. Hodnoty budou ukládány v Grayově kódu za použití binární reprezentace. Velikost chromozómu jedinců bude závislá na velikosti intervalu, na kterém hledáme minimum a na požadované přesnosti.

Dále budeme testovat knihovnu na problému N dam. Problém N dam bude využívat permutační reprezentace.

Testy budou prováděny na rozmanitém počtu populací, pro přehlednost se stejným nastavením počtu jedinců a se stejně zvolenými operátory selekce, křížení a mutace. U vícepopulačního genetického algoritmu budeme testovat asynchronní i synchronní migraci při všech levelech OpenMP.

Zjišťovat budeme jak rychlost konvergence tak výkon genetického algoritmu v závislosti na počtu jader výkonných procesorů.

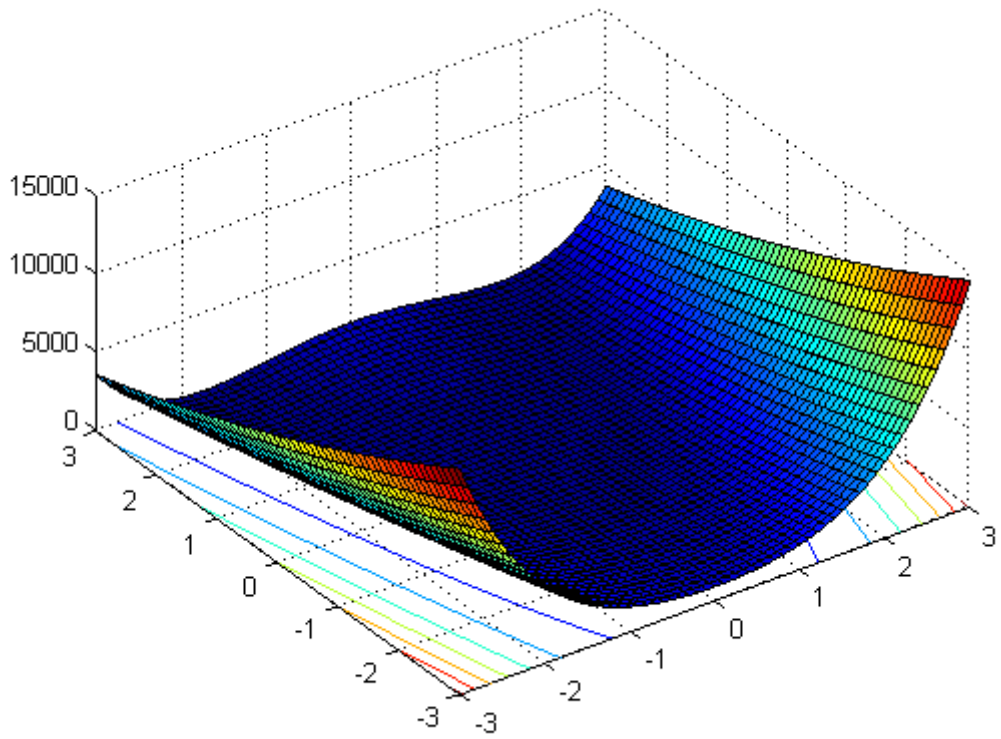
5.1 Rosenbrockova funkce (Rosenbrock's Function)

Jedná se o nekonvexní funkci též nazývanou jako Banánová funkce. Globální minimum se nachází uvnitř dlouhého, úzkého parabolického tvaru plochého údolí. Nalezení údolí je triviální avšak přiblížení ke globálnímu minimu je obtížnější. Vícedimenzionální definice funkce je následující.

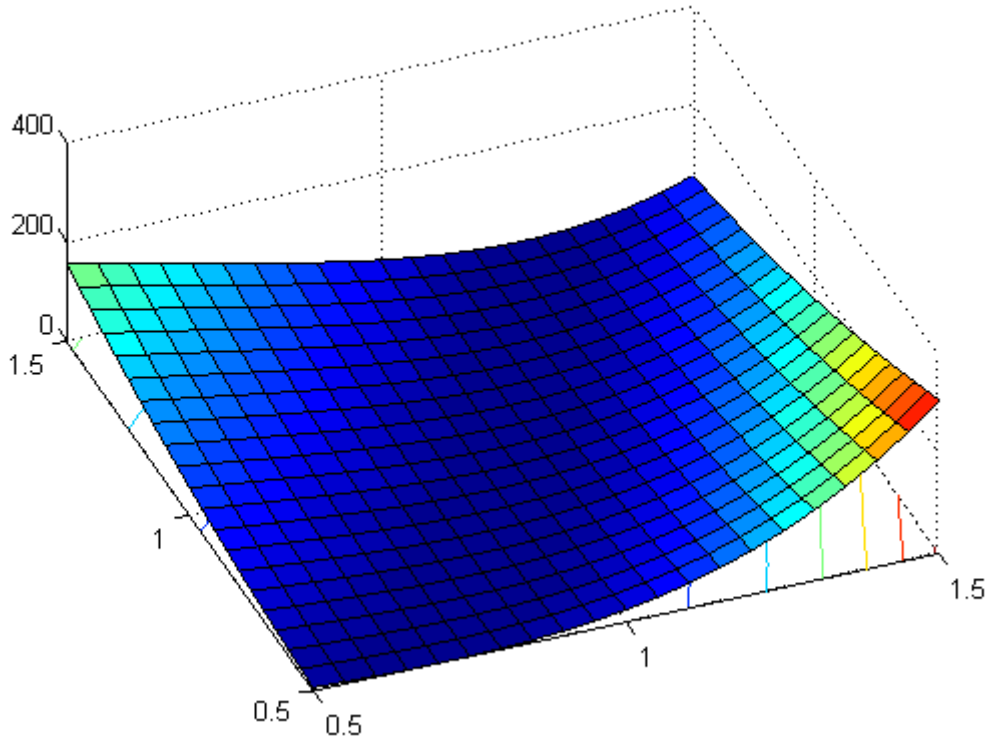
$$\boldsymbol{f(x) = \sum_{i=1}^{N-1} (1 - x_i)^2 + 100 \cdot \left(x_{i+1} - x_i^2\right)^2} \tag{5.1}$$

Kde *N* reprezentuje počet proměnných a x_i reprezentuje konkrétní hodnotu na *i-té* ose. Na každé ose nastává globální minimum v hodnotě 1, přičemž jeho funkční hodnota je na této ose rovna *0*. Globální minimum varianty této funkce je přesně tedy rovno 0.

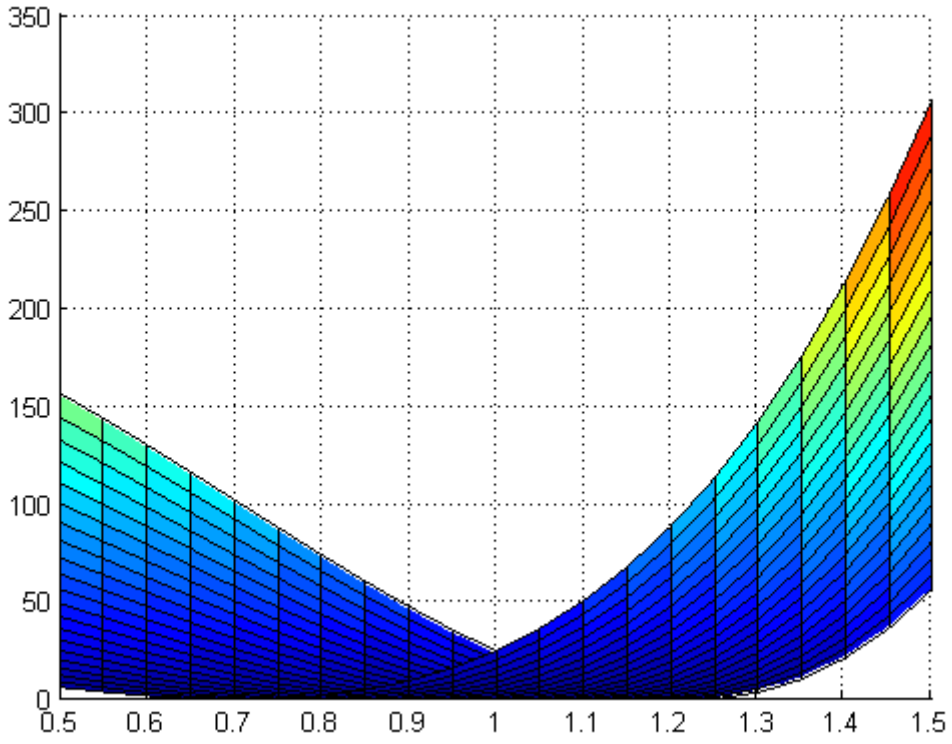
5.1.1 Vizualizace funkce



Obrázek 5.1 : Vykreslení Rosebrockovi funkce, dvou proměnných.



Obrázek 5.2: Detail minima v parabolickém údolí Rosebrockovi funkce.



Obrázek 5.3: Detail funkční hodnoty Rosenbrockovi funkce okolo minima.

5.1.2 Analýza použití genetického algoritmu ke hledání globálního minima

Globální minimum hledáme na intervalu <-8.388608, 8.388607> s přesností na šest desetinných míst na všech osách. Pro reprezentaci každé z os použijeme binární řetězec o délce 24 bitů, který reprezentuje v Grayově kódu dekadické číslo na ose.

5.1.3 Test výkonu OpenMP, při různém nastavení počtu populací a migrace

Nejdříve budeme testovat rychlost počítání počtu generací a porovnávat s jakým nastavením bude genetický algoritmus počítat jednotlivé generace co nejrychleji. Měnit budeme tedy počet populací, druh migrace (asynchronní, synchronní, bez migrace) a OpenMP Level.

Abychom mohli výpočetní rychlost objektivně porovnávat budou mít všechny populace vždy nastavené stejné parametry evoluce.

Celkově proběhne 92 druhů testů, z nichž se každý bude pětkrát opakovat a výsledky budou zprůměrované. Následuje tabulka globálního nastavení genetického algoritmu, nastavení je pro všechny testy stejné.

Doba trvání testu:	5 minut.
Velikost populací:	300 jedinců.
Reprezentace:	Binární reprezentace.
Mutace:	Binární, mutace proběhne u 15ti jedinců, přičemž jejich jednotlivé geny mutují s pravděpodobností 1 procento.
Selekce:	Elitní turnajová pro 2 soupeře, celkově se selektuje 300 jedinců pro křížení.
Křížení:	Uniformní křížení.
Doplnění:	Univerzální s parametry (300,1,0,0) vkládá se tedy do nové populace jeden elitní jedinec..
Počet populací:	Variabilní mezi 1 až 16ti. Uvedeno v následující tabulce podle ID testu.
OpenMP level:	Variabilní mezi 0 až 2. Uvedeno v následující tabulce podle ID testu.
Migrace:	Typ je vždy migrace do kompletní topologie sítě. Variabilně je nastavena na synchronní, asynchronní nebo bez migrace. Uvedeno v následující tabulce podle ID testu.

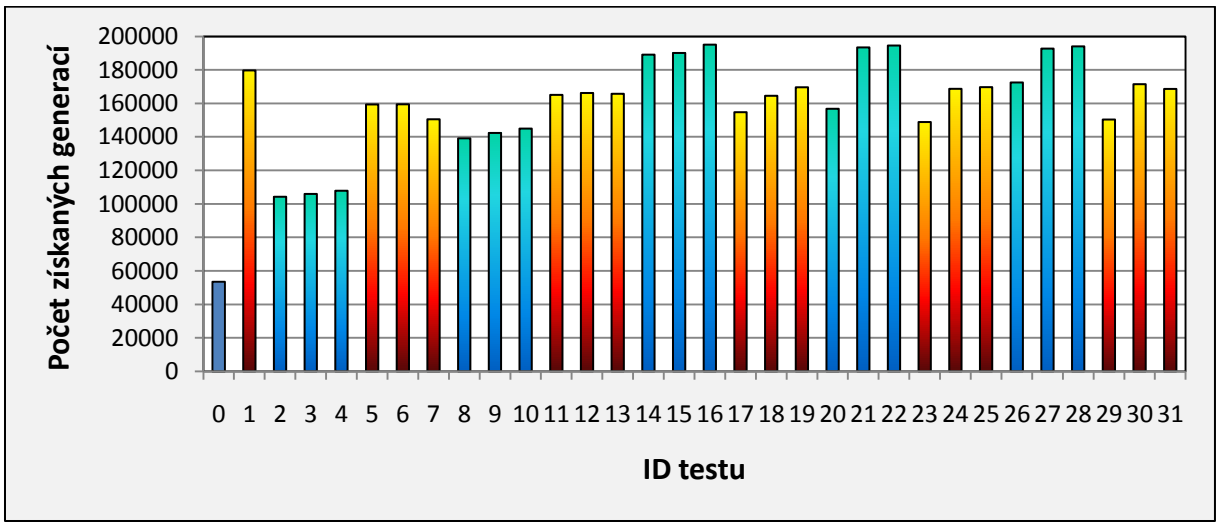
Tabulka 5.1: Globální nastavení genetického algoritmu pro všechny testy výkonu OpenMP.

V následující tabulce jsou uvedena jednotlivá nastavení testů v závislosti na jejich ID. Sloupec "OMP" určuje nastavení OpenMP levelu, sloupec "Pop" udává počet populací a sloupec "Mig" udává druh zvolené migrace. Přičemž 0 je test kde je migrace vypnuta, 1 je test se zvolenou synchronní migrací a 2 je test se zvolenou asynchronní migrací.

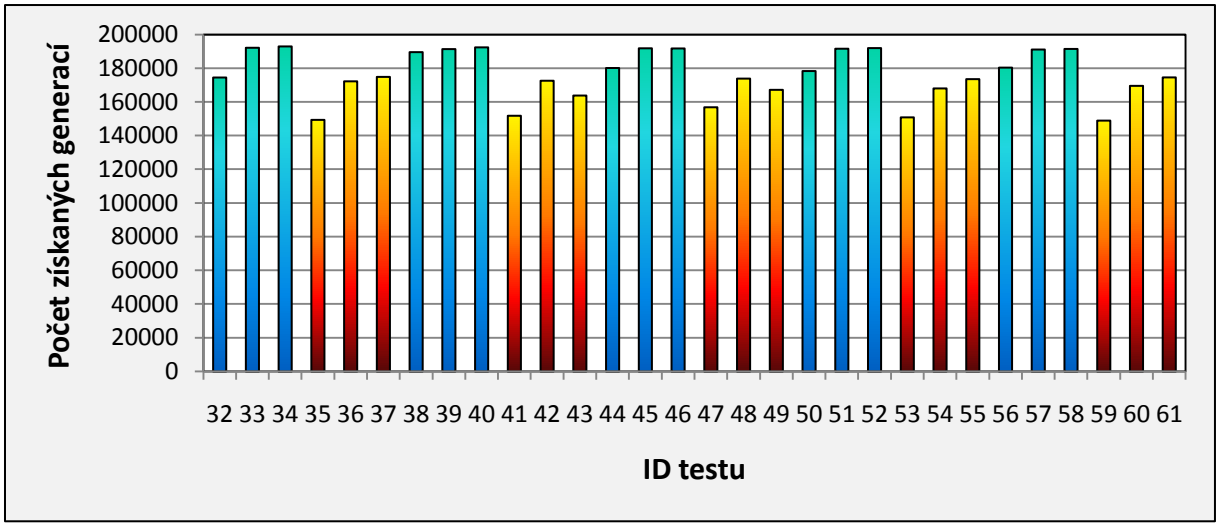
ID	Pop	OMP	Mig	ID	Pop	OMP	Mig	ID	Pop	OMP	Mig	ID	Pop	OMP	Mig
0	*1*	*0*	*0*	**23**	*5*	*2*	*1*	**46**	*9*	*1*	*0*	**69**	*13*	*1*	*2*
1	***1***	***2***	***0***	**24**	*5*	*2*	*2*	**47**	*9*	*2*	*1*	**70**	*13*	*1*	*0*
2	*2*	*1*	*1*	**25**	*5*	*2*	*0*	**48**	*9*	*2*	*2*	**71**	*13*	*2*	*1*
3	*2*	*1*	*2*	**26**	*6*	*1*	*1*	**49**	*9*	*2*	*0*	**72**	*13*	*2*	*2*
4	*2*	*1*	*0*	**27**	*6*	*1*	*2*	**50**	*10*	*1*	*1*	**73**	*13*	*2*	*0*
5	*2*	*2*	*1*	**28**	*6*	*1*	*0*	**51**	*10*	*1*	*2*	**74**	*14*	*1*	*1*
6	*2*	*2*	*2*	**29**	*6*	*2*	*1*	**52**	*10*	*1*	*0*	**75**	*14*	*1*	*2*
7	*2*	*2*	*0*	**30**	*6*	*2*	*2*	**53**	*10*	*2*	*1*	**76**	*14*	*1*	*0*
8	*3*	*1*	*1*	**31**	*6*	*2*	*0*	**54**	*10*	*2*	*2*	**77**	*14*	*2*	*1*
9	*3*	*1*	*2*	**32**	*7*	*1*	*1*	**55**	*10*	*2*	*0*	**78**	*14*	*2*	*2*
10	*3*	*1*	*0*	**33**	*7*	*1*	*2*	**56**	*11*	*1*	*1*	**79**	*14*	*2*	*0*
11	*3*	*2*	*1*	**34**	*7*	*1*	*0*	**57**	*11*	*1*	*2*	**80**	*15*	*1*	*1*
12	*3*	*2*	*2*	**35**	*7*	*2*	*1*	**58**	*11*	*1*	*0*	**81**	*15*	*1*	*2*
13	*3*	*2*	*0*	**36**	*7*	*2*	*2*	**59**	*11*	*2*	*1*	**82**	*15*	*1*	*0*
14	*4*	*1*	*1*	**37**	*7*	*2*	*0*	**60**	*11*	*2*	*2*	**83**	*15*	*2*	*1*
15	*4*	*1*	*2*	**38**	*8*	*1*	*1*	**61**	*11*	*2*	*0*	**84**	*15*	*2*	*2*
16	***4***	***1***	***0***	**39**	*8*	*1*	*2*	**62**	*12*	*1*	*1*	**85**	*15*	*2*	*0*
17	*4*	*2*	*1*	**40**	*8*	*1*	*0*	**63**	*12*	*1*	*2*	**86**	*16*	*1*	*1*
18	*4*	*2*	*2*	**41**	*8*	*2*	*1*	**64**	*12*	*1*	*0*	**87**	*16*	*1*	*2*
19	*4*	*2*	*0*	**42**	*8*	*2*	*2*	**65**	*12*	*2*	*1*	**88**	*16*	*1*	*0*
20	*5*	*1*	*1*	**43**	*8*	*2*	*0*	**66**	*12*	*2*	*2*	**89**	*16*	*2*	*1*
21	*5*	*1*	*2*	**44**	*9*	*1*	*1*	**67**	*12*	*2*	*0*	**90**	*16*	*2*	*2*
22	*5*	*1*	*0*	**45**	*9*	*1*	*2*	**68**	*13*	*1*	*1*	**91**	*16*	*2*	*0*

Tabulka 5.2: Nastavení OMP levelu, migrace a počtu populací pro různá ID testů.

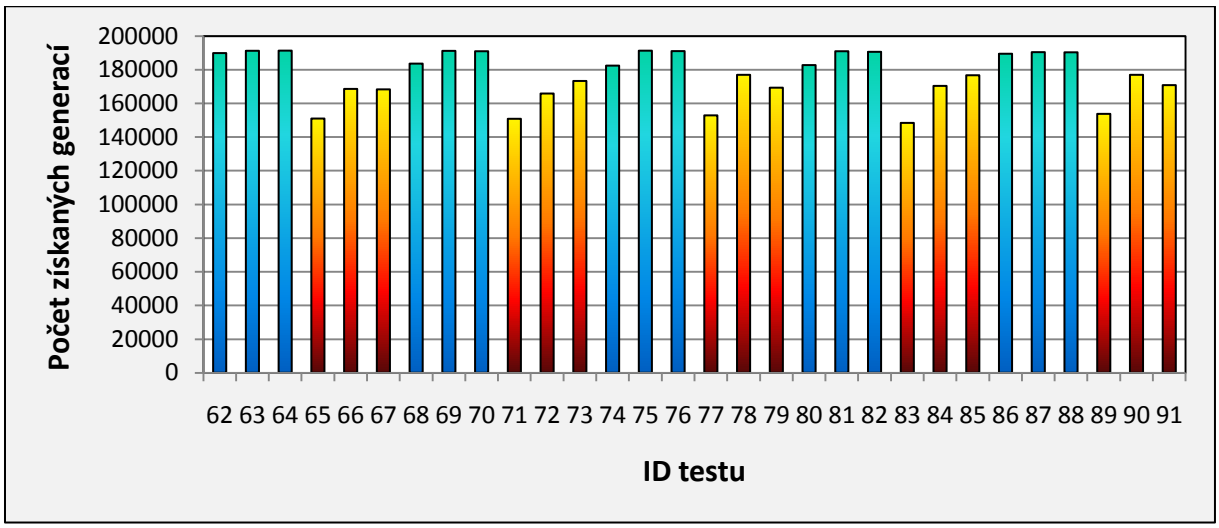
Následují grafy výsledků testů, pro názornost je přechodem do žluté vyznačen OpenMP level 2, přechodem do tyrkysové OpenMP level 1. V každé trojici přechodů je první test se synchronní migrací, druhý s asynchronní migrací a poslední bez migrace.



Obrázek 5.4: Výkonnostní výsledky testů s ID 0 až 31 Rosenbrockovi funkce.



Obrázek 5.5: Výkonnostní výsledky testů s ID 32 až 61 Rosenbrockovi funkce.



Obrázek 5.6: Výkonnostní výsledky testů s ID 61 až 91 Rosenbrockovi funkce.

Jak je vidět pro další testy nám bohatě postačí OpenMP level 2 na jedné populaci (test ID 1) s průměrným výsledkem 179615 generací spočítaných za 5 minut. Nejlepšího výsledku (194999,8) dosáhl test s ID 16, kde je vypnuta migrace a zapnut OpenMP level 1, paralelizace na úrovni populací. Testy byli prováděny na procesoru se 4mi jádry, nejlepší výkon testu s ID 16 je tedy zcela logický. Detailněji si lze testy prohlédnout v souboru přiloženém na DVD s názvem "garo-graphs-ompmig.xlsx".

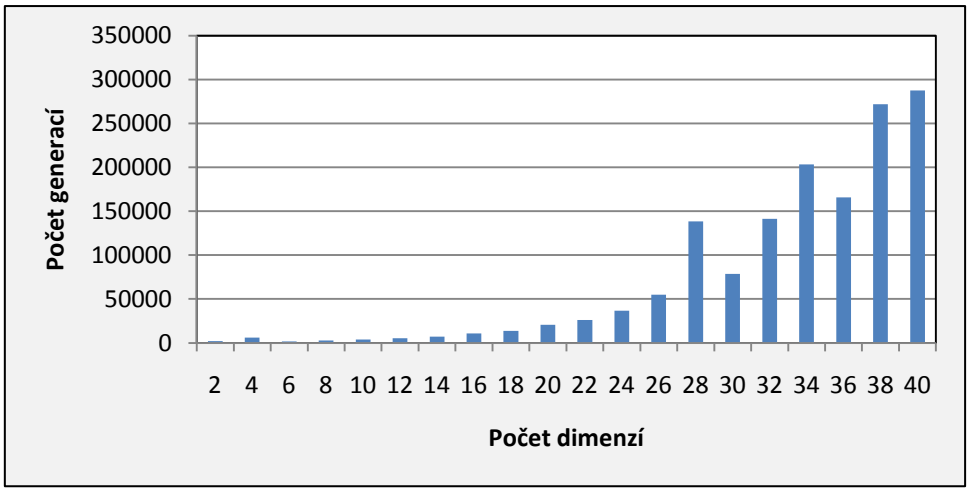
5.1.4 Test rychlosti nalezení řešení v závislosti na počtu dimenzí

Testy budou probíhat se stejným nastavením, měnit budeme pouze počet dimenzí Rosenbrockovi funkce, na kterých budeme hledat přesné minimum. Celkově proběhne 100 testů pro 2 až 40 dimenzí. Zobrazován je výsledný průměr.

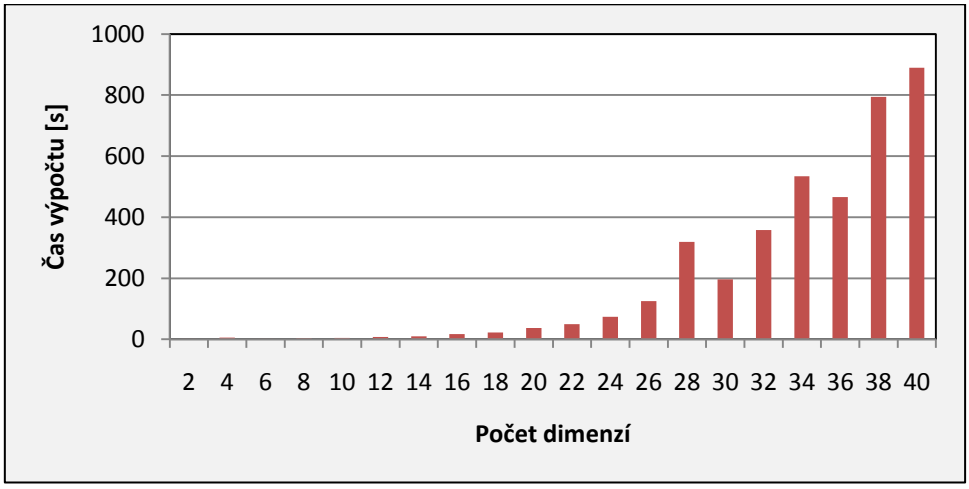
Reprezentace:	Binární.
Počet dimenzí:	Variabilní od 2 do 40ti s inkrementem 2.
Velikost genomu:	Variabilní od 48 do 960 s inkrementem 48.

Počet populací:	1
OpenMP úroveň:	2
Mutace:	Binární.
Typ mutace:	Pravděpodobnostní.
Pravděpodobnost mutace jedince:	40%
Pravděpodobnost mutace genu jedince:	1%
Selekce:	Elitní turnajová, turnaje se účastní 2 jedinci.
Křížení:	Jednobodové, kříží se vždy 80% populace.
Doplnění:	Univerzální (1000,0,200,0)
Velikost populace:	1000 jedinců.

Tabulka 5.3: Nastavení genetického algoritmu pro test rychlosti nalezení řešení v závislosti na počtu dimenzí.



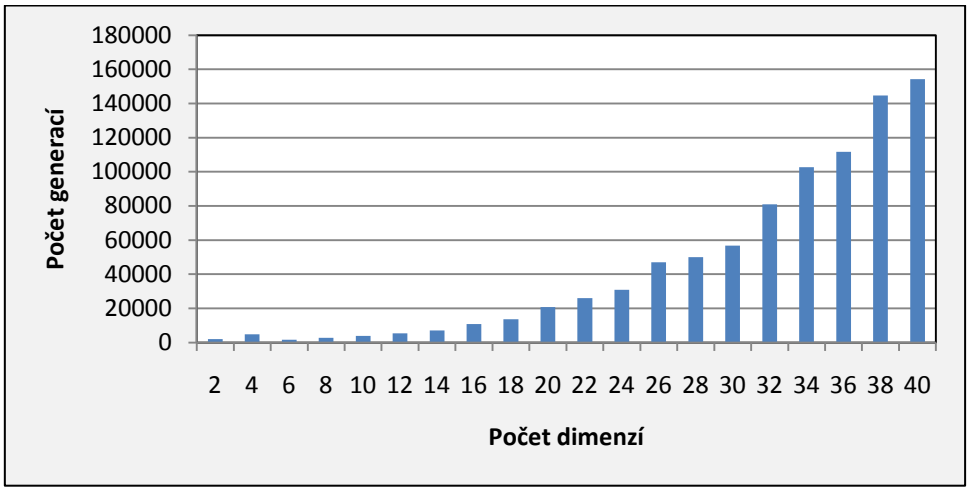
Obrázek 5.7: Počet generací potřebných k nalezení minima na různém počtu dimenzí.



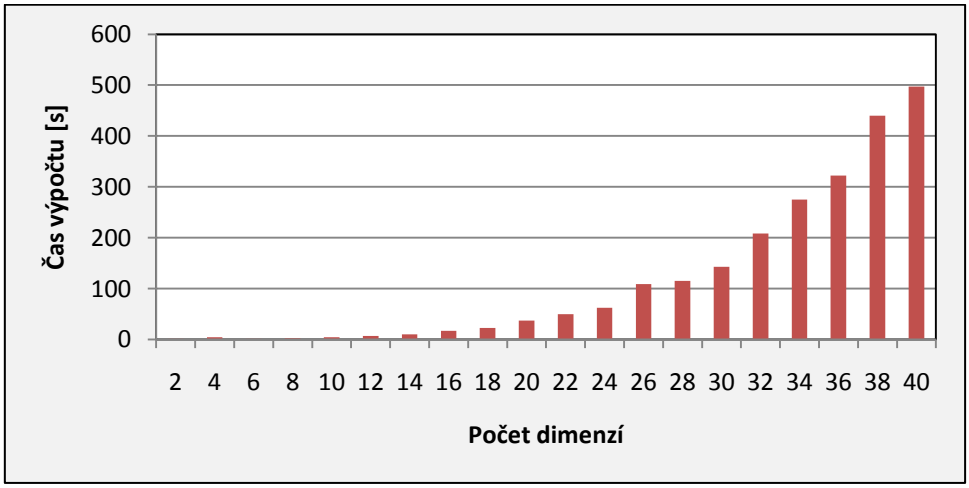
Obrázek 5.8: Potřebný čas k nalezení minima na různém počtu dimenzí.

Počet generací ani výpočetní čas se nezvyšuje ideálně při rostoucím počtu dimenzí. Příčina je v jednotlivých individuálních testech kde byl problém s nalezením minima, většinou stačilo k jeho nalezení změnit jeden bit v celém chromozómu. Pro názornost jsem tyto testy

vyjmul a udělal výsledek znovu. Celkově jsem odstranil 14 testů ze 400. Výsledky pak vypadají následovně a je vidět jak se výpočetní čas a počet potřebných generací přibližně exponenciálně zvětšuje.



Obrázek 5.9: Počet generací potřebných k nalezení minima na různém počtu dimenzí při vynechání testů s nadstandartní délkou.



Obrázek 5.10: Potřebný čas k nalezení minima na různém počtu dimenzí při vynechání testů s nadstandartní délkou.

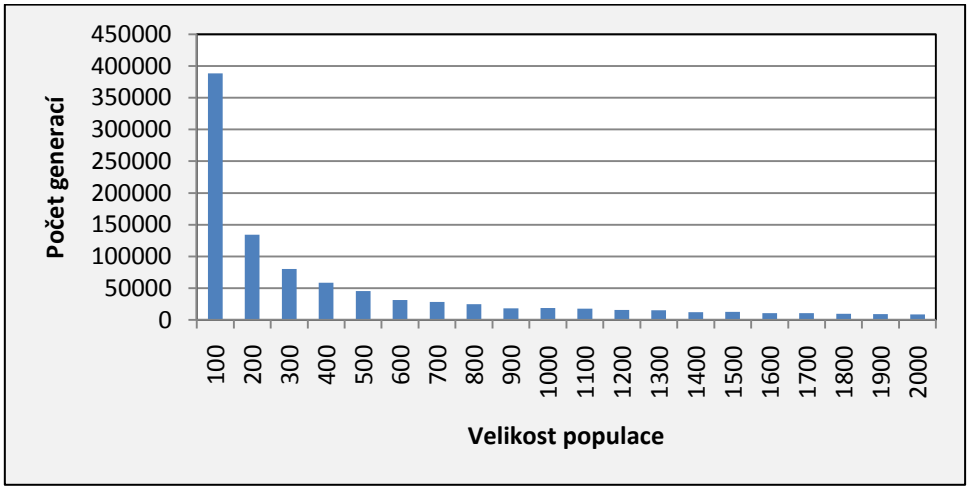
5.1.5 Test rychlosti nalezení řešení v závislosti na velikosti populace

Testy budou probíhat se stejným nastavením, měnit budeme pouze počet jedinců v populaci. Proběhne 100 testů pro všechny nastavení počtu jedinců v populaci. Zobrazován je výsledný průměr. V následující tabulce je zobrazeno nastavení genetického algoritmu.

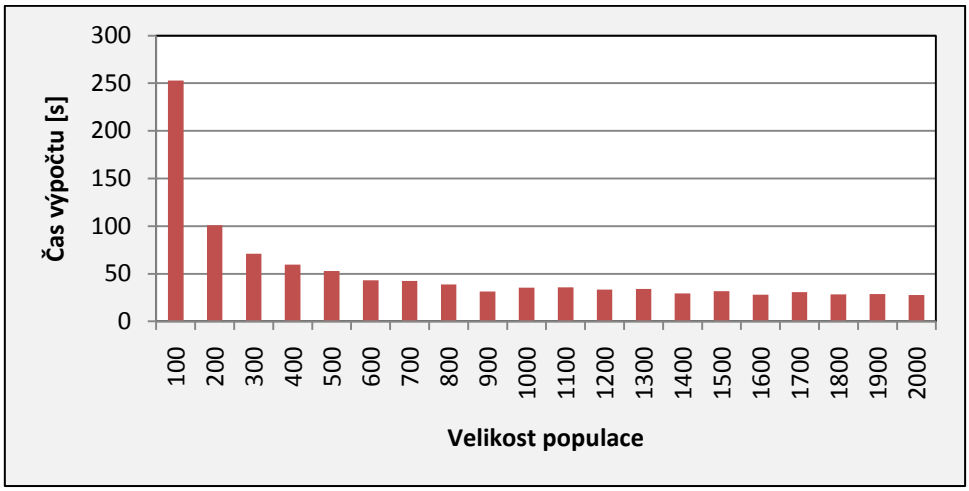
Reprezentace:	Binární.
Počet dimenzí:	20
Velikost genomu:	20 x 24 = 480 bitů
Počet populací:	1
OpenMP úroveň:	2
Mutace:	Binární.

Typ mutace:	Pravděpodobnostní.
Pravděpodobnost mutace jedince:	40%
Pravděpodobnost mutace genu jedince:	1%
Selekce:	Elitní turnajová, turnaje se účastní 2 jedinci.
Křížení:	Jednobodové, kříží se vždy 80% populace.
Doplnění:	Univerzální (100%,0,20%,0)
Velikost populace:	Variabilní, 100 až 2000 jedinců.

Tabulka 5.4: Nastavení genetického algoritmu pro test rychlosti nalezení řešení v závislosti na velikosti populace.



Obrázek 5.11: Výsledný počet generací potřebný k nalezení minima s různým počtem jedinců v populaci.



Obrázek 5.12: Výsledný čas potřebný k nalezení minima s různým počtem jedinců v populaci.

Testy nám potvrdili naše předpoklady a se zvětšujícím se počtem jedinců v populaci klesá počet generací které vedou k výsledku. V přibližně stejné závislosti klesá i potřebný čas na výpočet.

5.2 Rastriginova funkce více proměnných (Rastrigin's Function)

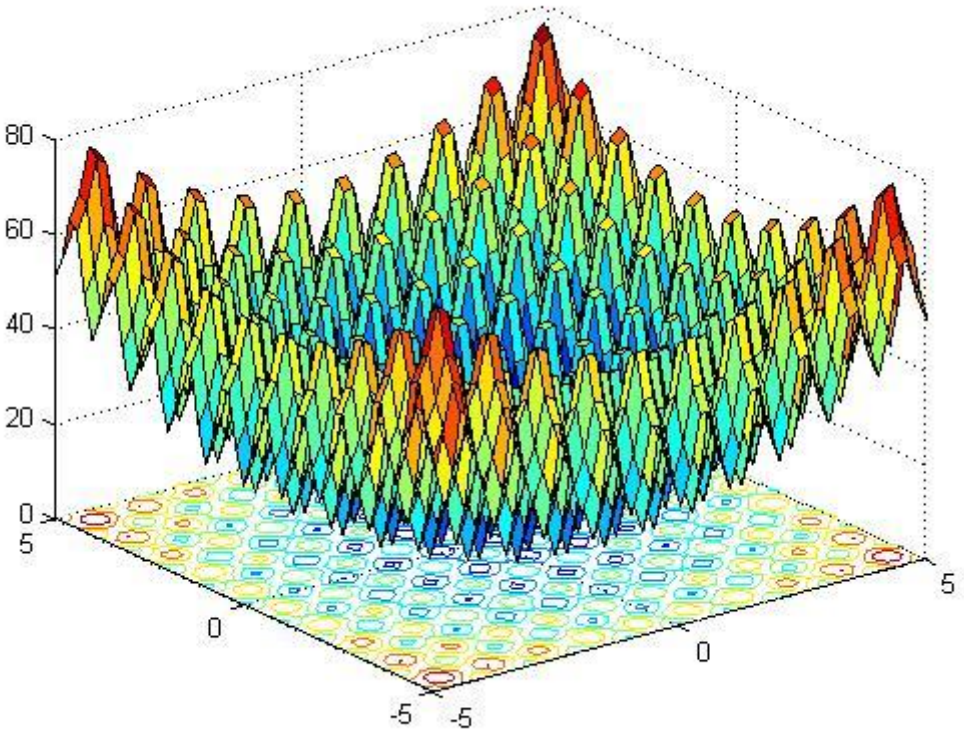
Rastriginova funkce založena na jedné exponenciální funkci druhé mocniny od které se odečítá kosinová modulace, aby vzniklo mnoho lokálních minim. Tyto lokální minima jsou pravidelně rozmístěna. Funkce má následující N-dimenzionální tvar [28].

Obecná definice funkce: $$\boldsymbol{f(x) = 10 \cdot N + \sum_{i=1}^{N}\left(x_i^2 - 10 \cdot \cos(2 \cdot \pi \cdot x_i)\right)}$$

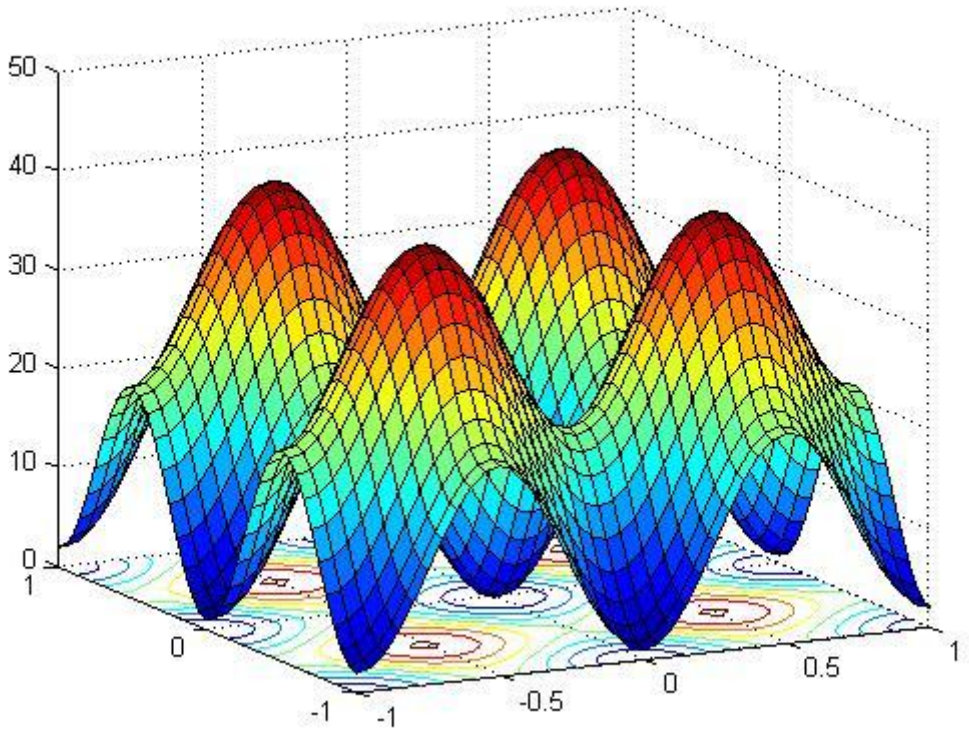
(5.2)

Kde *N* reprezentuje počet proměnných a x_i reprezentuje konkrétní hodnotu na *i-té* ose. Globální minimum budeme hledat na intervalu <-8.388608, 8.388607> na všech osách s přesností na šest desetinných míst. Globální minimum nastává na všech osách v *0* a jeho funkční hodnota je taktéž *0*.

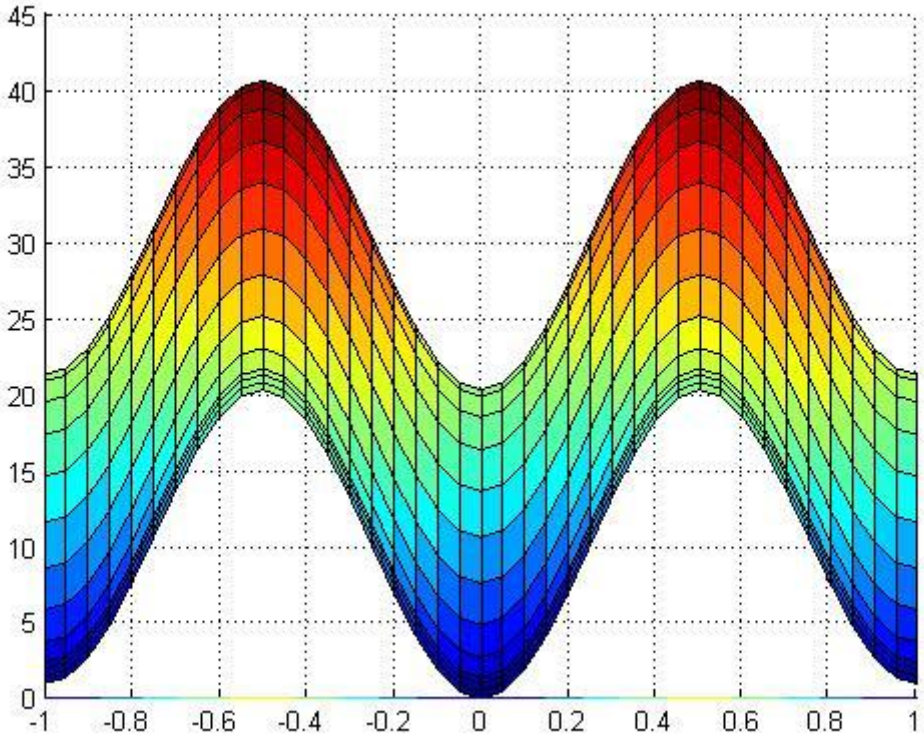
5.2.1 Vizualizace funkce



Obrázek 5.13: Vykreslení Rastriginovi funkce dvou proměnných.



Obrázek 5.14: Vykreslení Rastriginovi funkce, dvou proměnných, detail na globální minimum a nejbližší lokální minima.



Obrázek 5.15: Vykreslení detailu globálního minima Rastriginovi funkce dvou proměnných a nejbližších lokálních minim, horizontální pohled.

5.2.2 Analýza použití genetického algoritmu ke hledání globálního minima

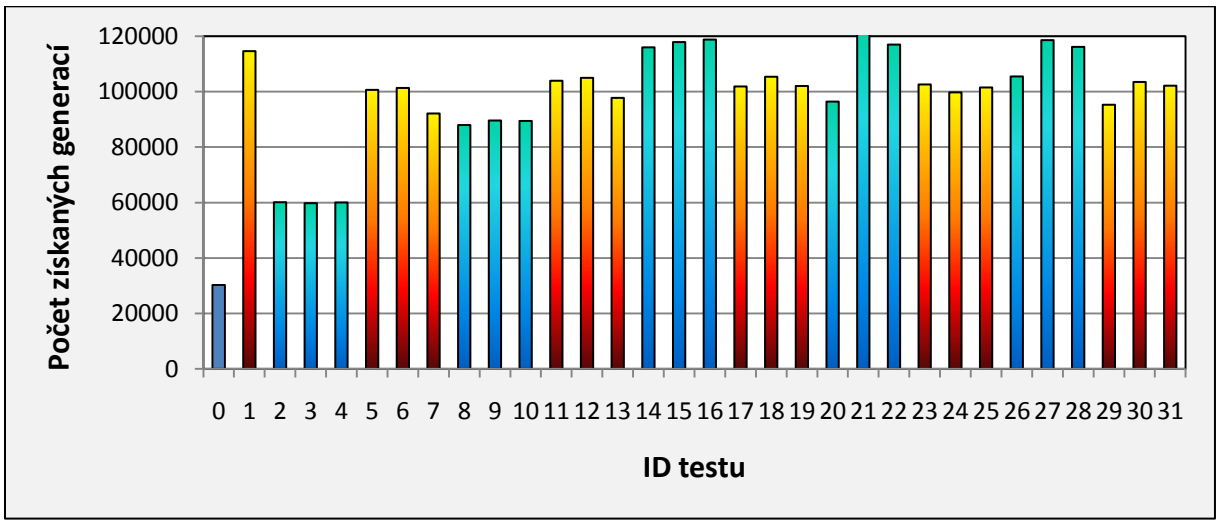
Globální minimum hledáme na intervalu <-8.388608, 8.388607> s přesností na šest desetinných míst na všech osách. Pro reprezentaci každé z os použijeme binární řetězec o délce 24 bitů, který reprezentuje v Grayově kódu dekadické číslo na ose.

Hledáme tedy řešení v Grayově kódu [110000000000000000000000], pro každou z os, čemuž odpovídá dekadická hodnota 0.0 na každé z os a je přesně uprostřed hledaného intervalu.

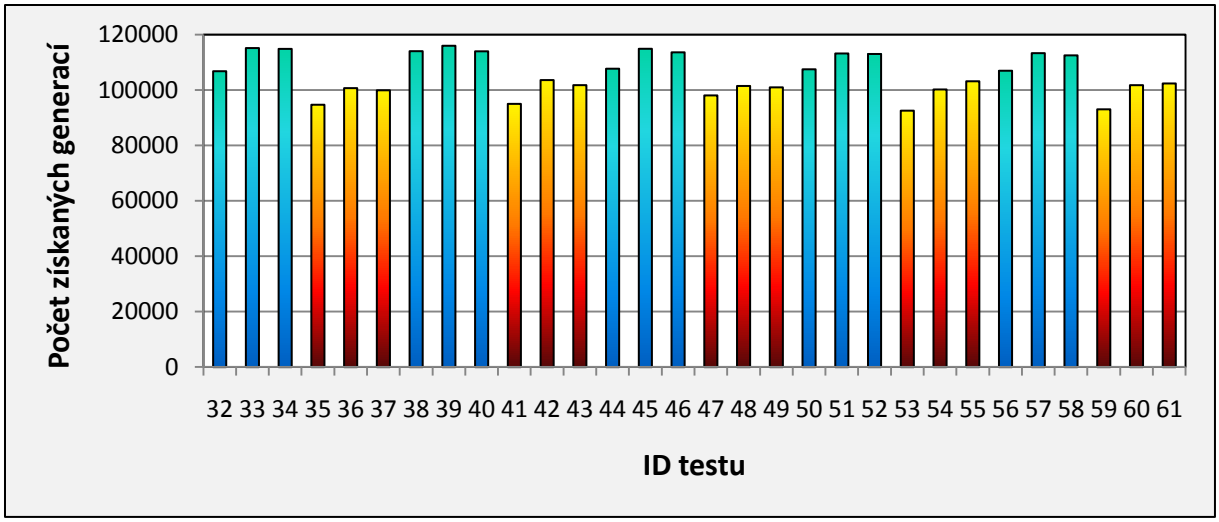
5.2.3 Test výkonu OpenMP, při různém nastavení počtu populací a migrace

Měnit budeme tedy počet populací, druh migrace (asynchronní, synchronní, bez migrace) a OpenMP Level. Jako v předchozím příkladu 5.1 budou mít všechny populace vždy nastavené stejné parametry evoluce a celkově proběhne také 92 druhů testů, z nichž se každý bude pětkrát opakovat a výsledky budou zprůměrované.

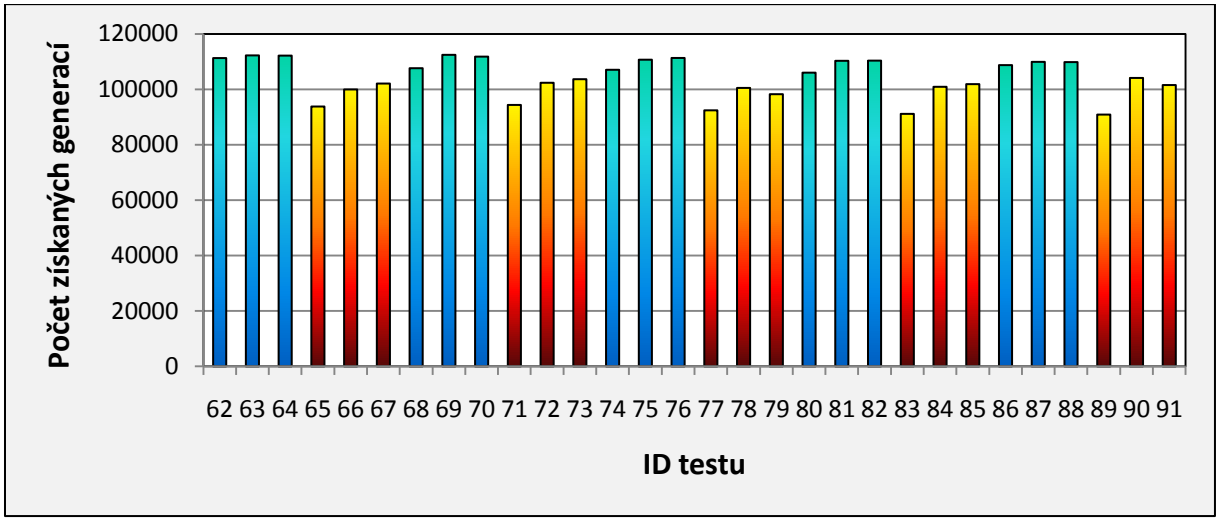
Nastavení genetického algoritmu je stejné, pouze s jedním rozdílem, jako v příkladu 5.1 a je uvedeno v tabulce Tabulka 5.1. Jediná změna je v křížení, u Rastriginovi funkce použijeme uniformní křížení. Všech 92 rychlostních testů bude mít stejné nastavení podle ID jako je uvedeno v tabulce Tabulka 5.2. Testy by měli dopadnout obdobně jako v kapitole 5.1.3.



Obrázek 5.16: Výkonnostní výsledky testů s ID 0 až 31Rastriginovi funkce.



Obrázek 5.17: Výkonnostní výsledky testů s ID 32 až 61Rastriginovi funkce.

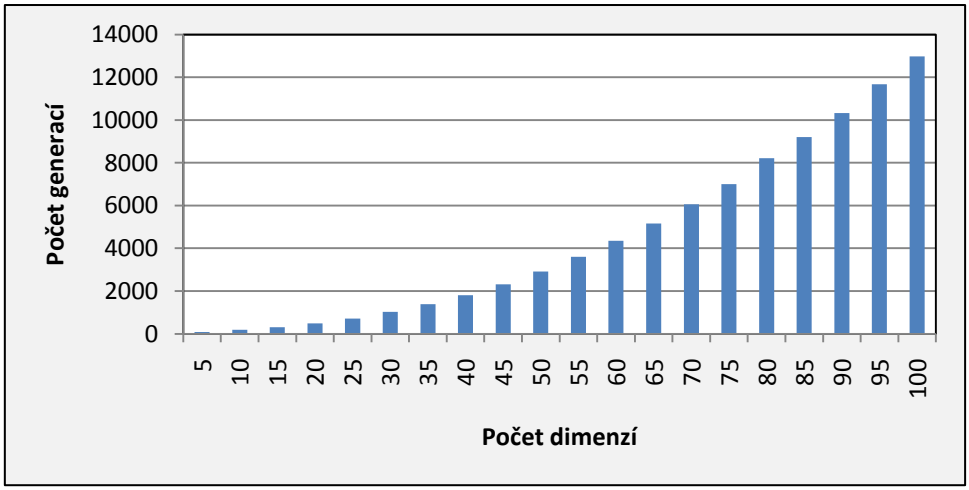


Obrázek 5.18: Výkonnostní výsledky testů s ID 62 až 91 Rastriginovi funkce.

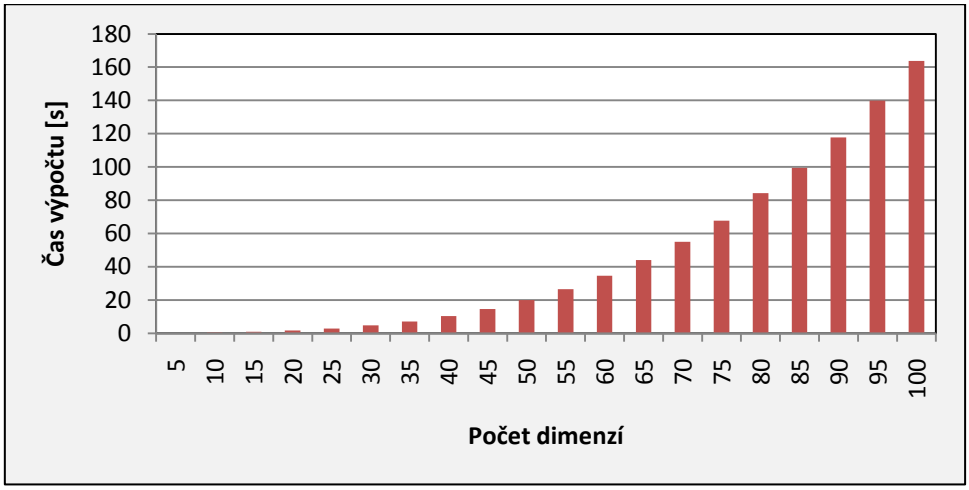
K provádění dalších testů nám bude tedy opět stačit genetický algoritmus s jednou populací a nastavením OpenMP úrovně 2.

5.2.4 Test rychlosti nalezení řešení v závislosti na počtu dimenzí

Testy budou opět probíhat se stejným nastavením, měnit budeme pouze počet dimenzí Rastriginovi funkce, na kterých budeme hledat přesné minimum. Nastavení těchto testů je tedy identické s nastavením jako v tabulce Tabulka 5.3: Nastavení genetického algoritmu pro test rychlosti nalezení řešení v závislosti na počtu dimenzí.. Počet dimenzí je variabilní od 5 do 100 s inkrementem 5 a velikost genomu je taktéž variabilní od 120 do 2400 s inkrementem 120. Použito je uniformní křížení. Celkově proběhne 100 testů pro 2 až 40 dimenzí. Zobrazován je výsledný průměr.



Obrázek 5.19: Počet generací potřebných k nalezení minima na různém počtu dimenzí.

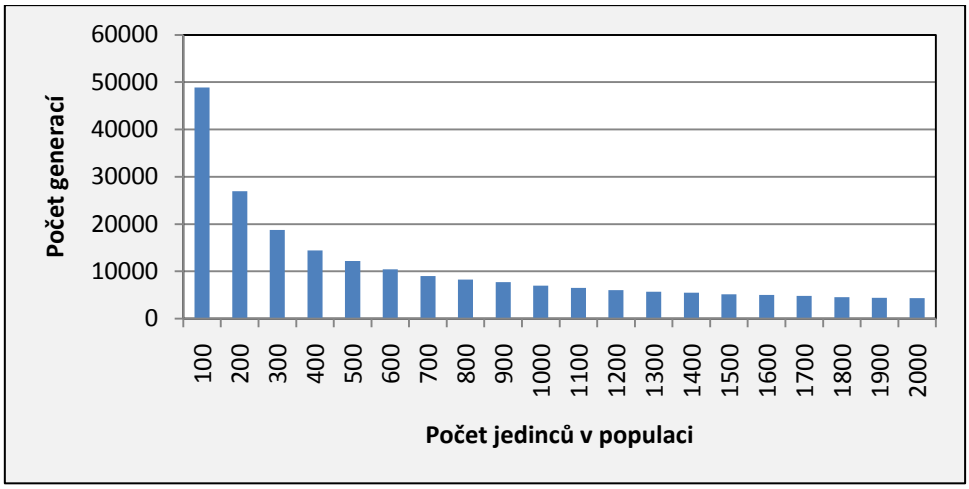


Obrázek 5.20: Výsledný čas potřebný k nalezení minima na různém počtu dimenzí.

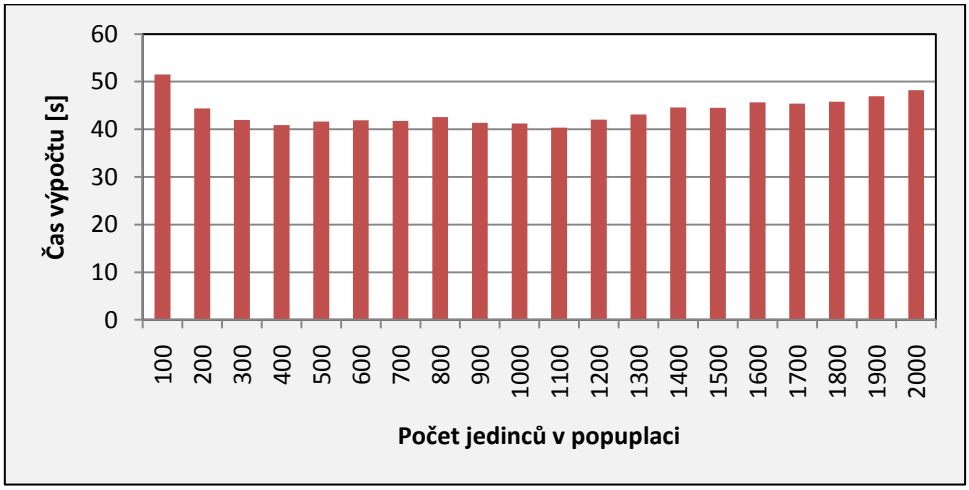
Dle výsledků je názorně vidět přibližně exponenciální zvyšování složitosti při zvětšování počtu dimenzí, na kterých funkci řešíme.

5.2.5 Test rychlosti nalezení řešení v závislosti na velikosti populace

Testy probíhají se stejným nastavením jako u Rosenbrockovi funkce, jejich nastavení je uvedeno v tabulce Tabulka 5.4: Nastavení genetického algoritmu pro test rychlosti nalezení řešení v závislosti na velikosti populace., pouze počet dimenzí je konstantní a výšen na 75. Velikost populace se mění od 100 do 2000 s inkrementem 100 jedinců na druh testu. Výsledky jsou zobrazeny v následujících grafech.



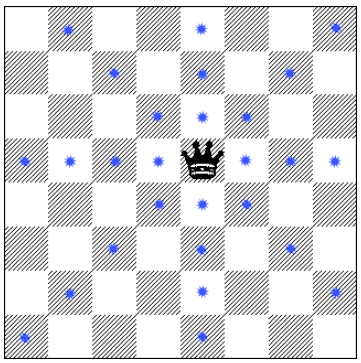
Obrázek 5.21: Výsledný počet generací potřebný k nalezení minima s různým počtem jedinců v populaci.



Obrázek 5.22: Výsledný čas potřebný k nalezení minima s různým počtem jedinců v populaci.

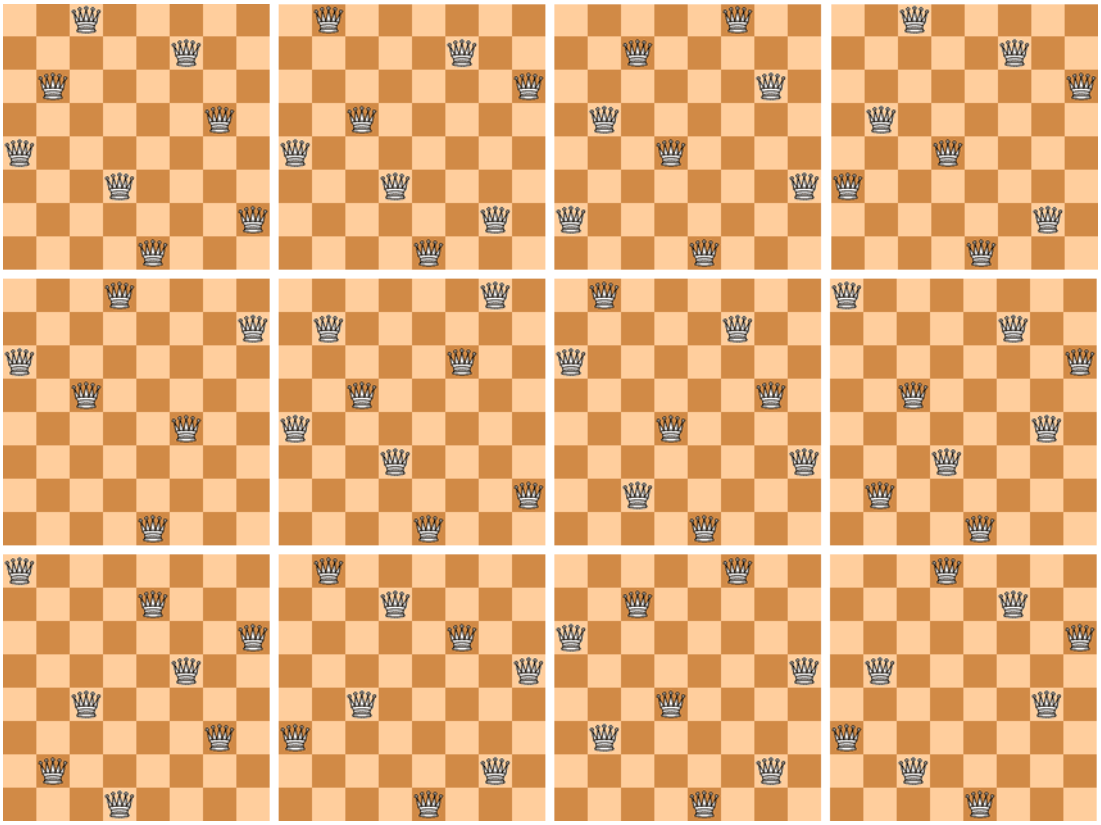
I u Rastriginovi funkce má velikost populace na výpočet velký vliv. Jak je vidět se zvětšující se populací nám klesá počet populací potřebných k dosažení výsledku. S ohledem na čas nám však genetický algoritmus našel, pro toto nastavení, řešení nejrychleji při velikosti populace 1100 jedinců. Testy s 1100 jedinci v populaci, ale potřebovali k nalezení řešení více generací než větší populace. Pro rychlý výpočet je tedy u problémů nutné vhodně zvolit velikost populace. Větší populace nám ne vždy zaručuje nalezení řešení za kratší časový úsek.

5.3 Problém N dam na šachovnici (N-Queens Problem)



Jedná se o klasický kombinatorický problém, který je často studován různými metodami z oblasti umělé inteligence. Problém *N* dam je jednoduše definovatelný. Vychází se z šachové hry, kde je dáma figurkou mající možnost se pohybovat v rámci sloupců, řádků a obou úhlopříček o jakýkoli počet políček přímým směrem.

Najít řešení problému *N* dam vyžaduje rozmístit *N* dam (kde *N > 0*) na šachovnici o rozměrech *N x N*, tak aby se dámy vzájemně neohrožovali. Při splnění tohoto řešení nesmí zbýt na šachovnici ani jediné políčko, na které by nemohla jedna z dam pomocí jednoho tahu přeskočit. Nejmenší rozměr šachovnice, na které je tento problém řešitelný je *N = 4*. Přičemž šachovnici o rozměru *1x1* s jednou dámou ignorujeme.



Obrázek 5.23: Zobrazení všech unikátních řešení problému N = 8 dam.

Problém *N* dam má celou řadu praktických aplikací v různých oborech. Například v oblastech řízení letového provozu, testování VLSI, komunikačních systémů, rozvrhování úloh a zdrojů, datové komprese a mnohých dalších [03].

Následující tabulka ukazuje počet všech možných řešení pro řád *4 < N < 26*. Některá tato řešení jsou shodná, použijeme-li na ně rotaci, překlopení či jinou operaci. Pokud tato shodná řešení nepočítáme, dostaneme počet unikátních řešení.

Řád N	Všech řešení	Unikátních řešení
4	2	1
5	10	2
6	4	1
7	40	6
8	92	12
9	352	46
10	724	92
11	2,680	341
12	14,200	1,787
13	73,712	9,233
14	365,596	45,752

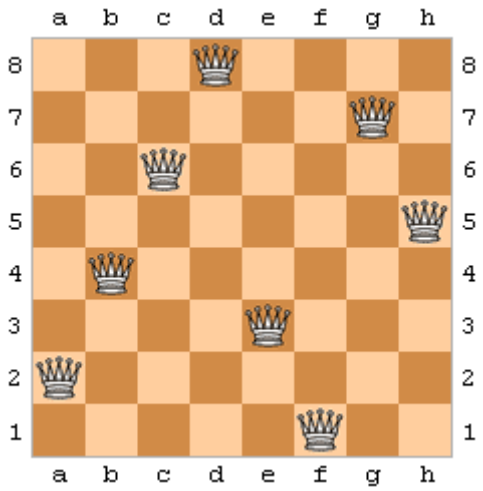
Řád N	Všech řešení	Unikátních řešení
15	2,279,184	285,053
16	14,772,512	1,846,955
17	95,815,104	11,977,939
18	666,090,624	83,263,591
19	4,968,057,848	621,012,754
20	39,029,188,884	4,878,666,808
21	314,666,222,712	39,333,324,973
22	2,691,008,701,644	336,376,244,042
23	24,233,937,684,440	3,029,242,658,210
24	227,514,171,973,736	?
25	2,207,893,435,808,352	?

Tabulka 5.5: Výpis počtu řešení problému N dam.

5.3.1 Analýza použití genetického algoritmu k řešení problému N dam

Skutečnosti, že dámy se nesmí vzájemně ohrožovat, využijeme. K reprezentaci jedinců použijeme permutační kódování, kde jsou jednotlivá potencionální řešení zakódována jako permutace množiny čísel *1* až *N*, přičemž *N* reprezentuje velikost problému. Číslo *i* na *j*-té pozici v této permutaci potom reprezentuje dámu, která je umístěna na *i*-tý řádek v *j*-tém sloupci (sloupce jsou na následujícím obrázku pro rozlišení označeny písmeny A-H namísto číslic 1-8). Chromozóm jedince, který reprezentuje řešení, bude tedy pro *N = 8* vypadat například takto:

$$(6,1,5,2,8,3,7,4) \approx (F,A,E,B,H,C,G,D)$$



Obrázek 5.24: Zobrazení chromozómu.

Všimněte si, že řešení z obrázku Obrázek 5.24: Zobrazení chromozómu. je stejné řešení jako rozložení číslo 10, na obrázku Obrázek 5.23: Zobrazení všech unikátních řešení problému N = 8 dam., pouze orotované o 90 stupňů.

Ohodnocující funkce bude přiřazovat jedincům ohodnocení podle jejich dam postavených v nekonfliktních pozicích, to znamená, podle počtu dam, jenž neútočí na žádnou jinou dámu. Chromozóm reprezentující řešení problému *N* dam, bude mít ohodnocení *N*. Chromozóm, který reprezentuje situaci, kde jsou umístěny všechny dámy a zároveň na sebe vzájemně diagonálně útočí pouze dvě z nich, bude mít ohodnocení *N - 2*. Pro úplnost chromozóm, který by reprezentoval postavení všech dam, například ve stejném sloupci, by měl ohodnocení *0*. Tato situace nemůže však nikdy, díky zvolenému způsobu kódování, nastat.

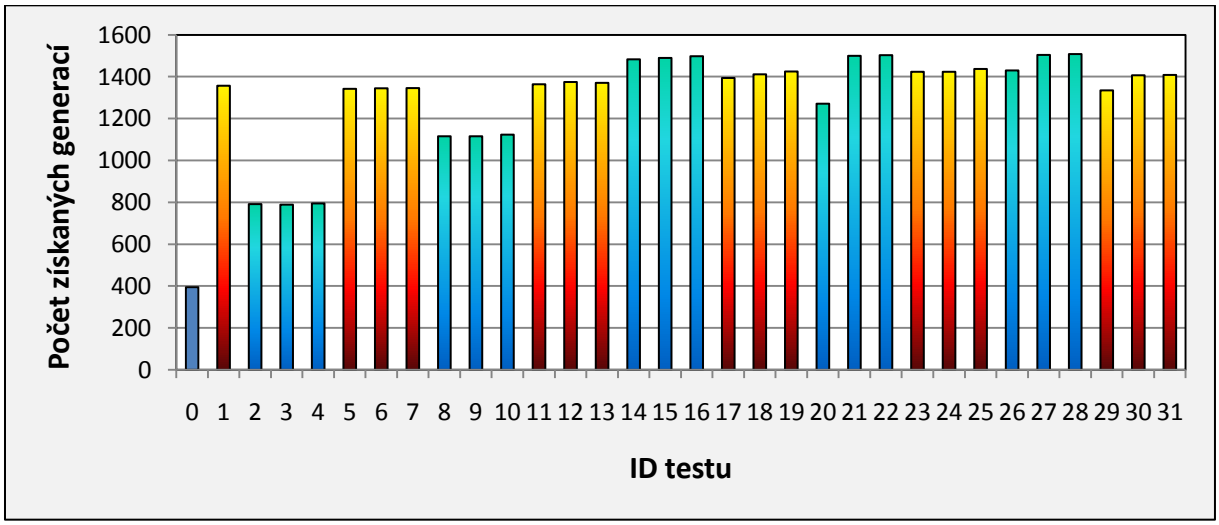
Jedinec č.	Chromozóm jedince	Ohodnocení (fitness)
1	(6, 1, 5, 2, 8, 3, 7, 4)	8
2	(1, 6, 5, 2, 8, 3, 7, 4)	4
3	(2, 4, 6, 1, 3, 5, 8, 7)	4
4	(7, 8, 4, 1, 5, 2, 6, 3)	4

Obrázek 5.25: Příklad ohodnocení jedinců, problém N dam.

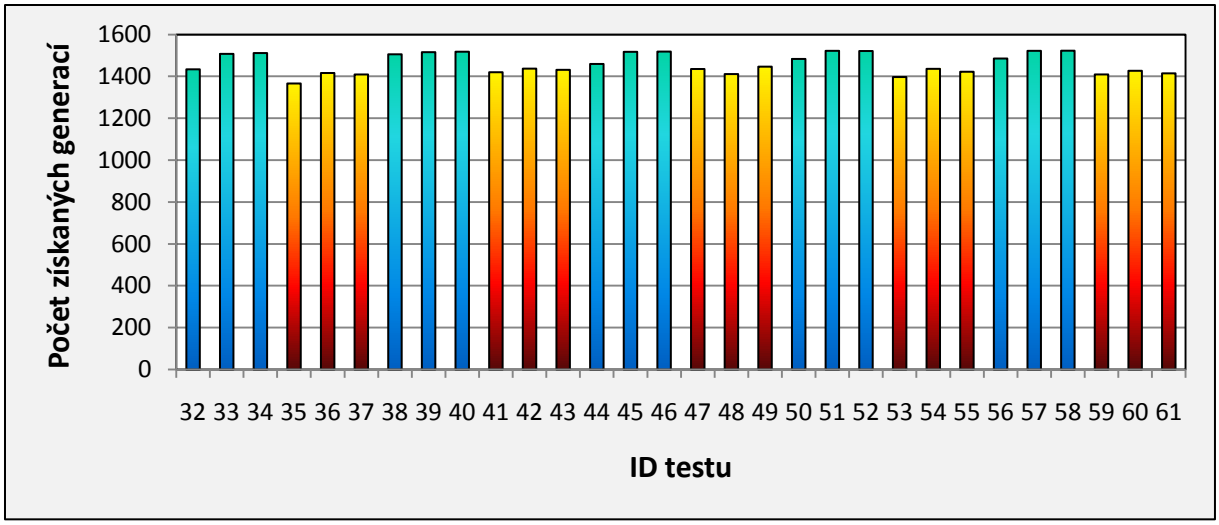
5.3.2 Test výkonu OpenMP, při různém nastavení počtu populací a migrace

Jako obvykle budeme měnit počet populací, druh migrace (asynchronní, synchronní, bez migrace) a OpenMP Level. Jako v příkladu 5.1 budou mít všechny populace vždy nastavené stejné parametry evoluce a celkově proběhne také 92 druhů testů, z nichž se každý bude pětkrát opakovat a výsledky budou zprůměrované.

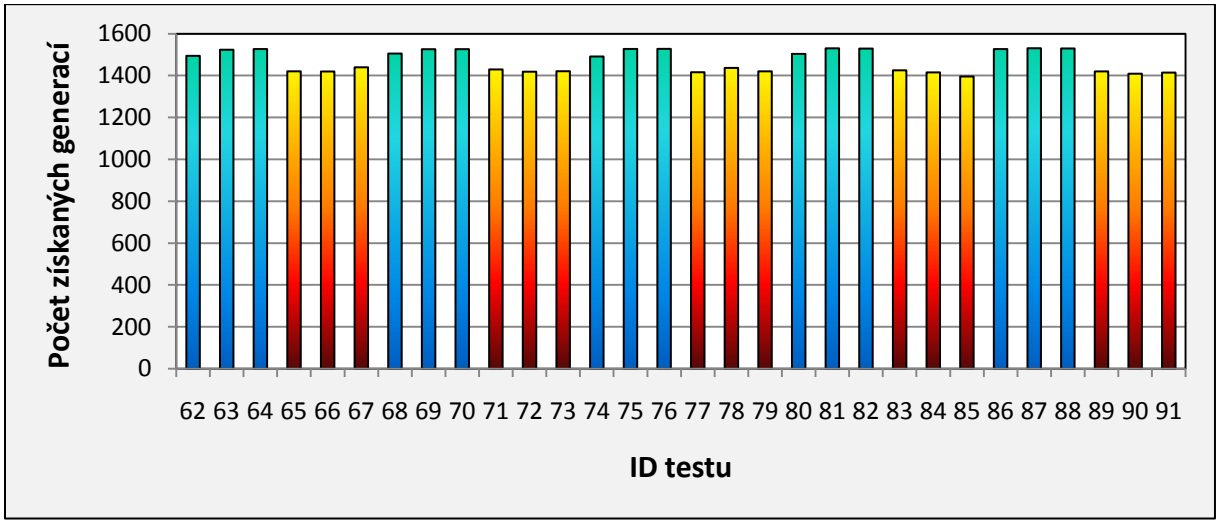
Použita je permutační reprezentace, výměnná mutace a PMX křížení.



Obrázek 5.26: Výkonnostní výsledky testů s ID 0 až 31problému N dam.



Obrázek 5.27: Výkonnostní výsledky testů s ID 32 až 61problému N dam.

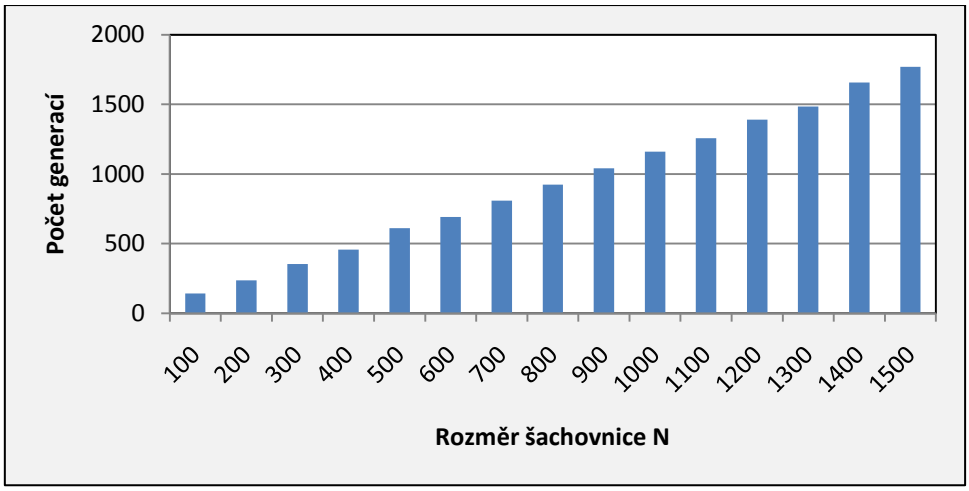


Obrázek 5.28: Výkonnostní výsledky testů s ID 62 až 91problému N dam.

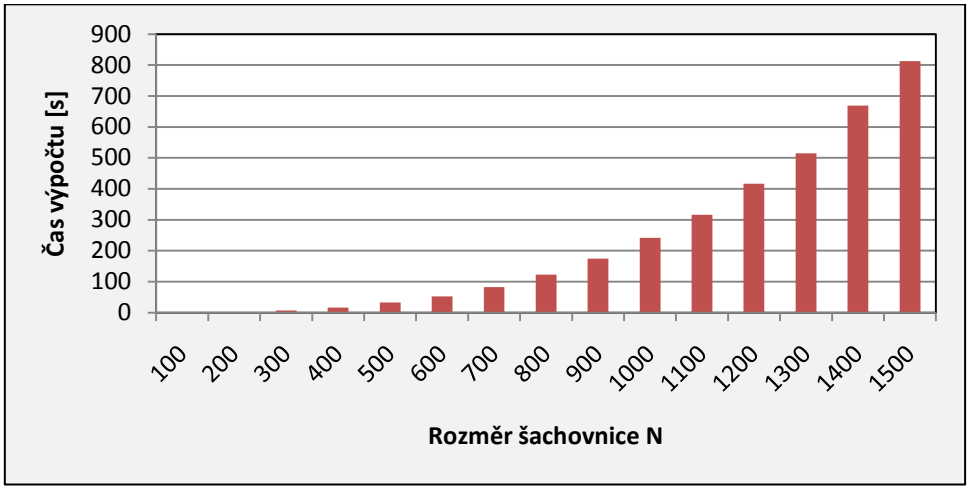
Podle grafů si opět vystačíme s využitím jedné populace a OpenMP úrovně 2.

5.3.3 Test rychlosti nalezení řešení v závislosti na rozměru šachovnice

V následujících testech zvětšujeme postupně šachovnici od rozměrů 100*100 do rozměru 1500*1500 vždy s inkrementem 100*100. Výsledky následných testů zobrazují lineární nárůst potřebných generací k vyřešení problému pokud složitost problému zvětšujeme tímto způsobem. N návaznosti roste časová složitost nalezení řešení exponenciálně. Testy probíhají s velikostí populace tisíce jedinců, křížní je opět PMX a mutace výměnná.



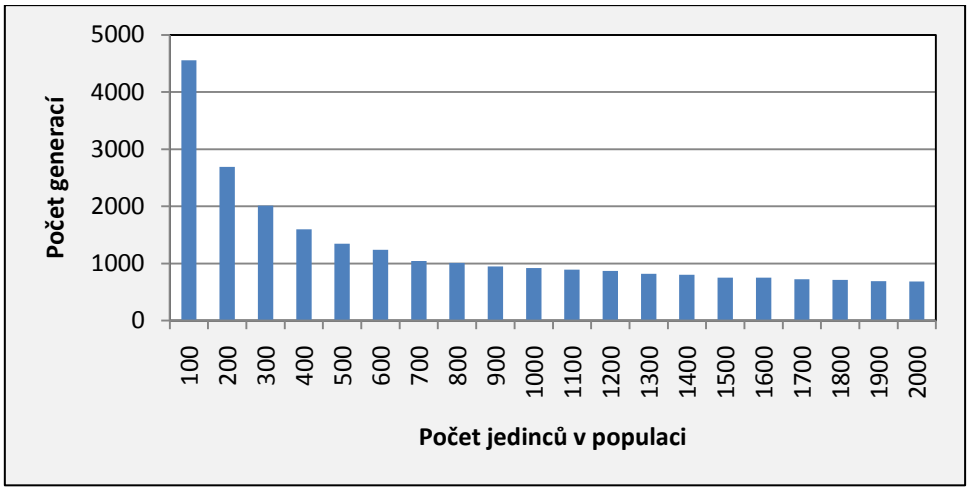
Obrázek 5.29: Počet generací potřebných k nalezení řešení problému N dam na různých rozměrech šachovnice..



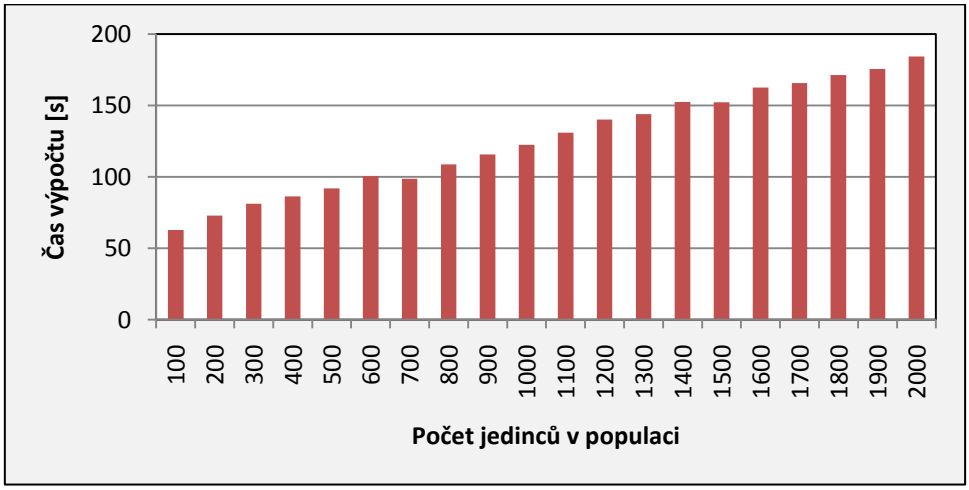
Obrázek 5.30: Výsledný čas potřebný k nalezení řešení problému N dam na různých rozměrech šachovnice.

5.3.4 Test rychlosti nalezení řešení v závislosti na velikosti populace

Následující testy budou probíhat na velikosti šachovnice 800*800, křížení bude opět PMX a použijeme výměnnou mutaci. Velikost populace budeme postupně zvětšovat ze 100 jedinců do 2000 jedinců s inkrementem 100 jedinců. Jak následující výsledky ukazují při těchto rozměrech šachovnice a zvětšujícím se počtu jedinců v populaci nám sice klesá počet potřebných generací ale čas potřebný k jejich vypočtení se zvětšuje.



Obrázek 5.31: Počet generací potřebný k nalezení řešení problému N dam v závislosti na velikosti populace.



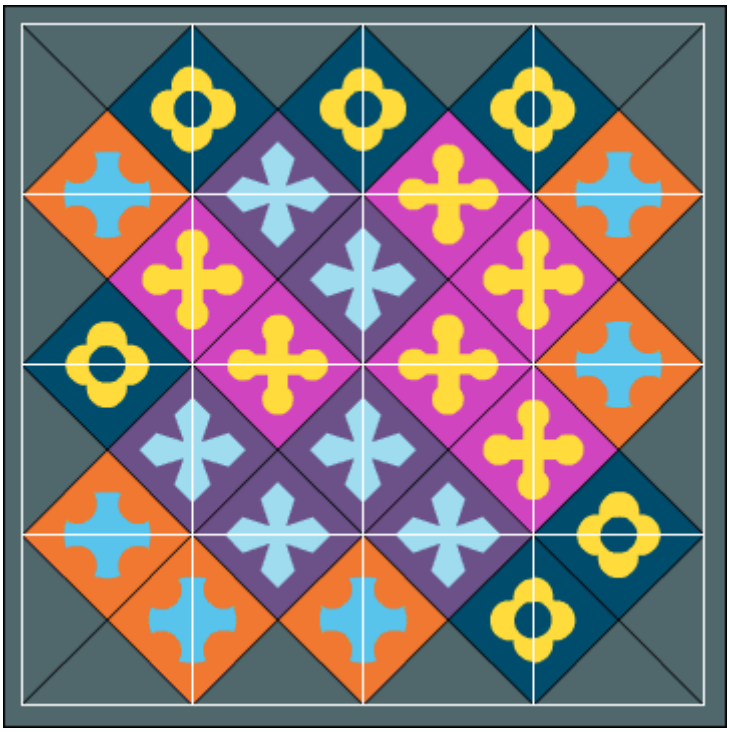
Obrázek 5.32: Výsledný čas potřebný k nalezení řešení problému N dam v závislosti na velikosti populace.

6 Eternity II



Puzzle Eternity II je pokračováním Eternity I, které před osmi lety, kdy bylo uvedeno na trh, fascinovalo tisíce lidí po celé Evropě. Šek na milión liber byl předán studentovi, který soutěžní puzzle úspěšně vyřešil za dobu pouhých osmnácti měsíců.

Eternity II je tedy soutěžní puzzle, vydané v roce 2007, jeho autorem je Christopher Monckton a vydavatel hry je společnost TOMY UK Limited. Oficiální stránka toho puzzle se nachází na http://uk.eternityii.com/. Aktuální odměna, k datu 29.9.2010, za vyřešení Eternity II je nyní 2 milióny dolarů. malou ukázku této hry si můžete zahrát na http://cz.eternityii.com/try-eternity2-online/.



Obrázek 6.1: Ukázka řešení zmenšeného Eternity II rozměru 4x4 při použití čtyř barevných vzorů.

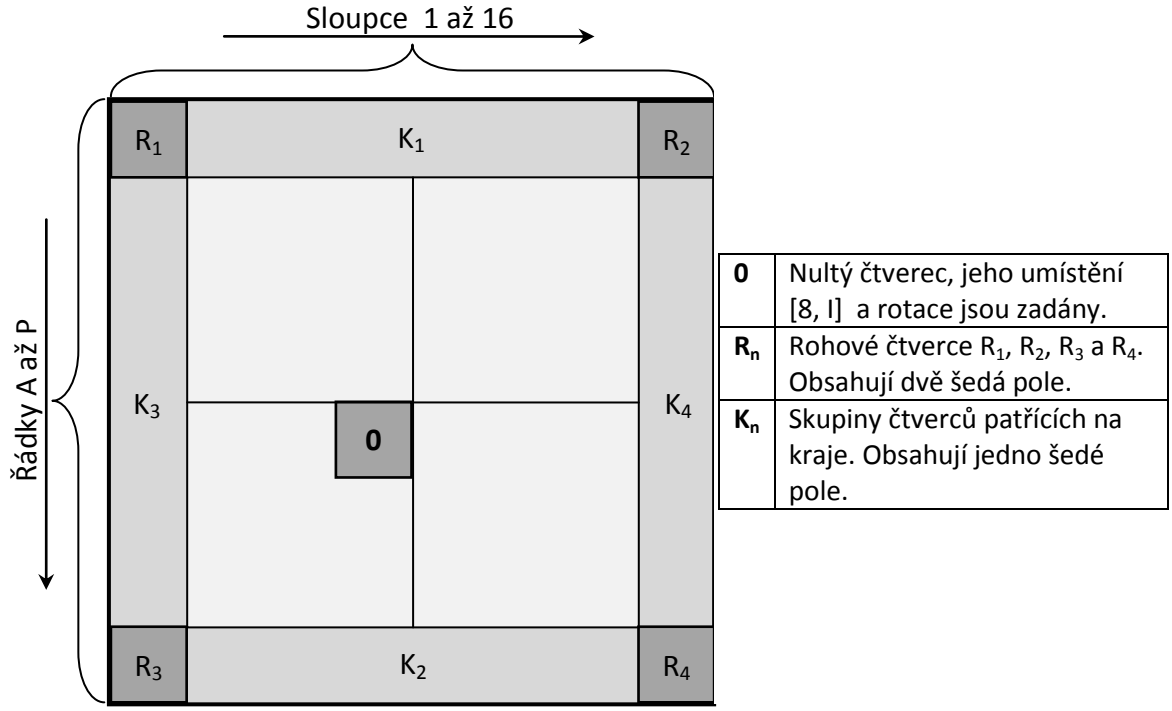
Zde je malá ukázka pole 4x4 čtverců. Každý čtverec má na svých stranách určitou barvu a tvar, které musí na každé hraně souhlasit s vedlejším čtvercem (celkem existuje 22 barevných a tvarem rozdílných variant). Na krajních stranách hry jsou čtverce, které na jedné či dvou stranách nemají vzor, ale jsou pouze jednobarevné. Takto se identifikují čtverce, které patří po okrajích.

Originální puzzle Eternity II má rozměry 16x16 a skládá se z 256 čtvercových dílků s 22 barevnými vzory. Řešením je propojení těchto vzorů přes celé puzzle na herním plánu.

Eternity II má celkově $256! \cdot 4^{256} \cong 1.15 \cdot 10^{661}$. Pokud vezmeme v úvahu, že známe pozici jednoho čtverce uprostřed a nadále víme které čtverce se vyskytují na krajních hranách, nám složitost problému klesne na $1 \cdot 4! \cdot 56! \cdot 195! \cdot 4^{195} \cong 1.115 \cdot 10^{557}$.

6.1.1 Analýza použití genetického algoritmu k řešení Eternity II

Je nutné zvolit vhodnou reprezentaci úlohy, křížení a mutaci. Celou plochu Eternity II musíme rozdělit do několika bloků jak je naznačeno na obrázku.



0	Nultý čtverec, jeho umístění [8, I] a rotace jsou zadány.
R_n	Rohové čtverce R_1, R_2, R_3 a R_4. Obsahují dvě šedá pole.
K_n	Skupiny čtverců patřících na kraje. Obsahují jedno šedé pole.

Obrázek 6.2: Rozdělení plochy Eternity II.

Kódování

Chromozóm je rozdělen celkem do šesti částí, jak je naznačeno na předchozím obrázku. První část obsahuje indexy čtverců, které se vyskytují na krajích mapy, z hlediska kódování se tedy jedná o permutaci. Druhá část obsahuje indexy čtverců které patří na kraj mapy, tato část chromozómu je také permutační. K rohovým a krajním indexům nepotřebujeme uchovávat jejich rotaci, ta je zřejmá díky příslušnosti krajního vzoru. Třetí část uchovává pozici neokrajových a nerohových prvků mapy, vyjma nultého čtverce, který je zadán uprostřed (a vyskytuje se ve čtvrtém bloku). Třetí část je tedy také permutační. K těmto vnitřním prvkům musíme znát jejich rotaci, která je uchována v páté části chromozómu, tato reprezentace není permutační. Rotace každého prvku dosahuje hodnot 0 až 3. Poslední část uchovává rotaci nultého prvku, který je zadán jak pozicí tak rotací.

R_1 ... R_4	K_1 ... K_4	Indexy neokrajových čtverců	0	Rotace neokrajových čtverců	0

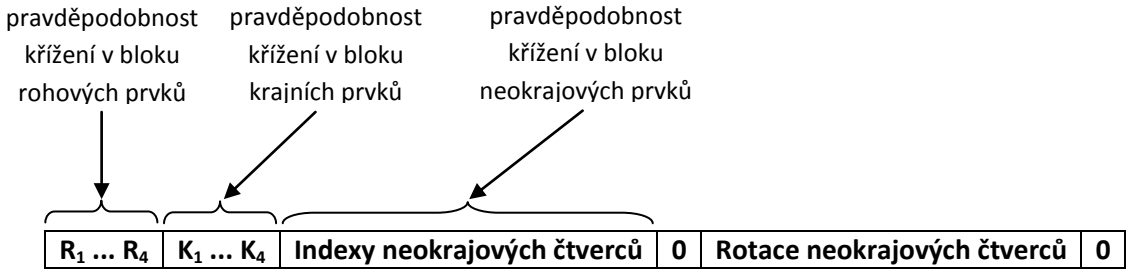
počet prvků v jednotlivých částech

4	56	195	1	195	1

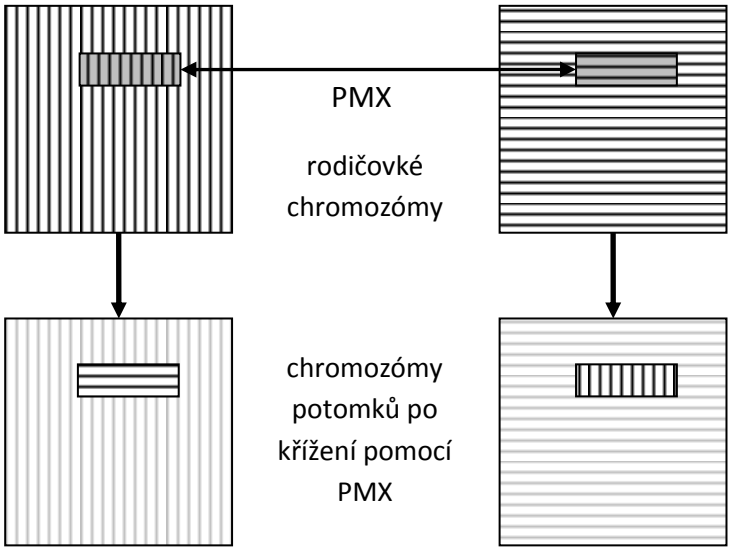
Obrázek 6.3: Chromozóm řešení Eternity II.

Křížení

Ke křížení použijeme operátor PMX popsaný v kapitole 4.2.2.1. Jelikož chromozóm obsahuje 6 bloků bude se křížení PMX spouštět s různou zadanou pravděpodobností na první tří bloky (rotace a zadaný prvek křížit nebudeme). Pokud PMX zkříží třetí blok a prohodí tedy pozice jednotlivých prvků tak se automaticky prohodí i jejich rotace. Dalším parametrem pro PMX křížení je maximální délka kříženého úseku v jednotlivých blocích.



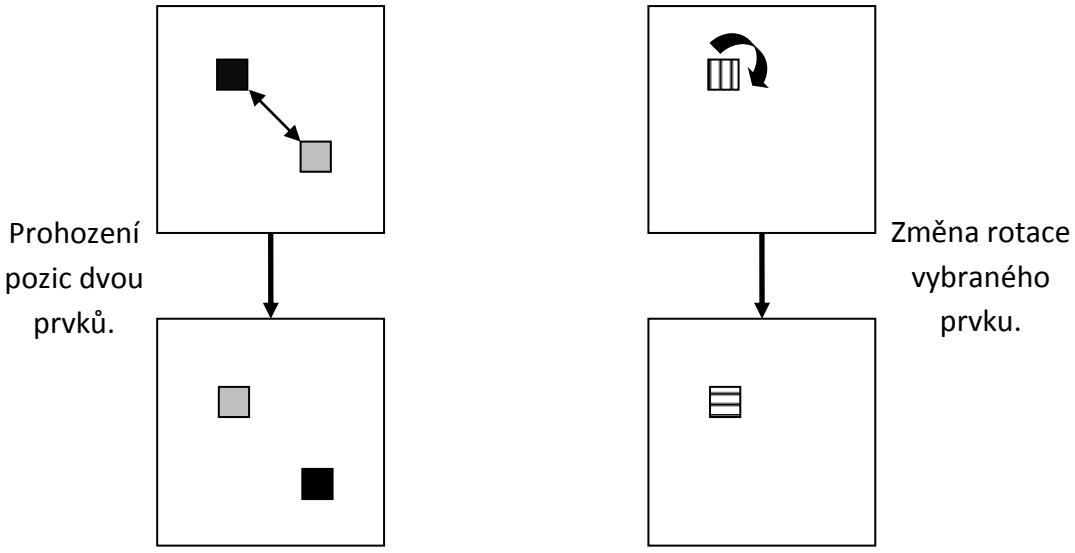
Obrázek 6.4: Křížení chromozómu reprezentující řešení Eternity II.



Obrázek 6.5: Křížení chromozómu Eternity II znázorněné graficky. Příklad křížení neokrajových prvků.

Mutace

Mutace náhodně volí jednu ze dvou variant. První varianta je prohození pozic dvou prvků, včetně jejich rotace, v chromozómu řešení. Druhá varianta náhodně vybere jeden prvek a změní mu rotaci. Druhá varianta tedy není aplikována na rohy a okrajové prvky, ty mají rotaci zadánu.



Obrázek 6.6: Mutace chromozómu Eternity II.

Hodnotící funkce

Hodnotící funkce přidává body za splnění jednotlivých kritérií. Celkově byla vytvořena tři kritéria. První kritérium přidá bod, pokud se vzory sousedních čtverců shodují na jejich sousedních stranách. Druhé kritérium zvětšuje ohodnocení na základě 4 prvků, které se shodují na všech vnitřních stranách. Za složený útvar je přidán 1 bod a za shodu čtyř vnitřních stran jsou přidány 4 body, celkově tedy 5 bodů. Třetí kritérium je obdoba druhého.



Kritérium 1
Strany: 1 bod
Celkově: 1 bod

Kritérium 2
Útvar: 1 bod
Strany: 4 body
(z kritéria 1)
Celkově: 5 bodů

Kritérium 3
Útvar: 1 bod
Strany: 4 body
(z kritéria 1)
Celkově: 5 bodů

Obrázek 6.7: Hodnotící kritéria pro fitness funkci Eternity II.

Řešení Eternity 2 má tedy při použití jednotlivých kritérií následující počet maximálních bodů (bodů při nalezení řešení).

Kritérium číslo	Maximum Eternity 2 mapa 16·16	Maximum Eternity 2 mapa N·N pro N≥3
1	480	$(N-1) \cdot N \cdot 2$
2	225	$(N-1)^2$
3	196	$(N-2)^2$
1+2	705	$(N-1) \cdot N \cdot 2 + (N-1)^2$
1+3	676	$(N-1) \cdot N \cdot 2 + (N-2)^2$
2+3	421	$(N-1)^2 + (N-2)^2$
1+2+3	901	$(N-1) \cdot N \cdot 2 + (N-1)^2 + (N-2)^2$

Obrázek 6.8: Maximální ohodnocení Eternity II.

6.1.2 Výsledky hledání řešení Eternity II

Pro hledání řešení jsme spustili na 4 na sobě nezávislé populace, každou s jiným nastavením. Všechny populace používali v hodnotící funkci kritérium 1+2+3 a elitní turnajový výběr, přičemž obsahují 10000 jedinců. Parametr úmrtí jedince je nastaven na 20 generací. Migrace byla vypnuta a OpenMP level nastaven na 1. Zbylé nastavení se liší dle následující tabullky. Použití křížení s popisem PMX(100,2,25,50,16) znamená že rohy/kraje/střed mapy se budou křížit pomocí PMX s pravděpodobností 2/25/50 procent, maximální velikost kříženého úseku bude přitom 16 prvků, podle posledního parametru.

Populace 1	
Křížení:	PMX(100,2,25,50,16)
Kříží se jedinců:	9500
Reinsertor:	Univerzální (10000,0,500,0)
Pst. mutace:	50

Populace 2	
Křížení:	PMX(100,0,26,52,16)
Kříží se jedinců:	10000
Reinsertor:	Univerzální (100,0,0,0)
Pst. mutace:	50

Populace 3	
Křížení:	PMX(100,0,52,52,16)
Kříží se jedinců:	10000
Reinsertor:	Univerzální (100,0,0,0)
Pst. mutace:	50

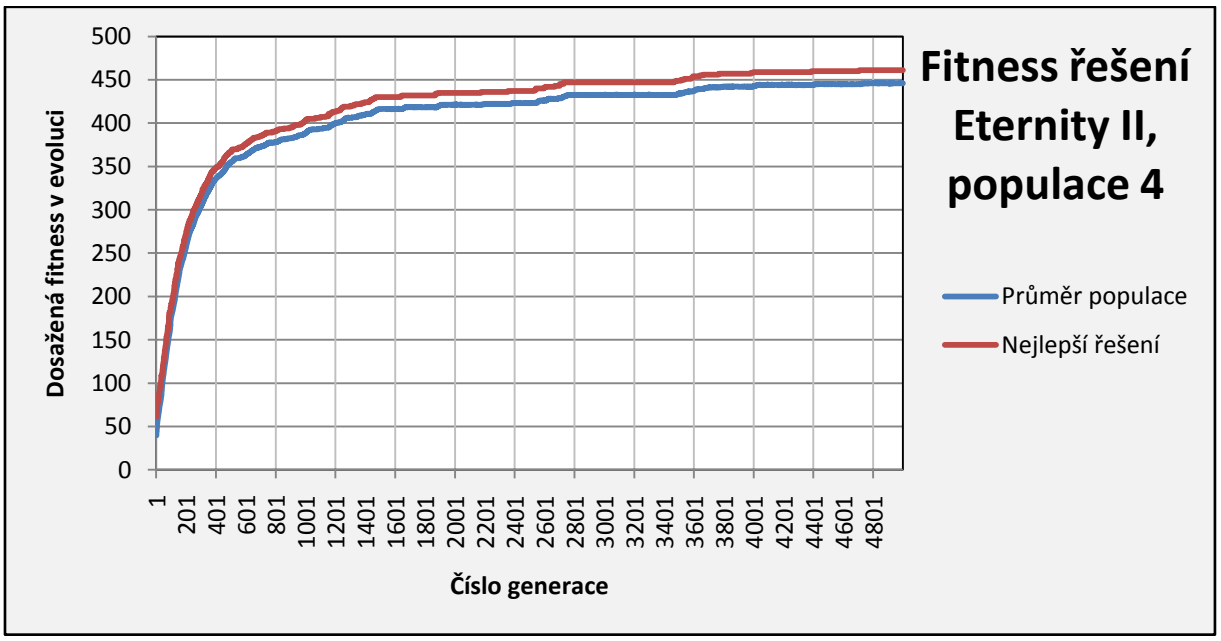
Populace 4	
Křížení:	PMX(100,1,20,70,16)
Kříží se jedinců:	9900
Reinsertor:	Univerzální (10000,0,100,0)
Pst. mutace:	70

Tabulka 6.1: Nastavení populací genetického algoritmu pro Eternity II.

Hledání řešení bylo spuštěno na dobu 180 minut. Nalezena byla následující partikulární řešení s jejich ohodnocením.

Populace	Počet generací	Maximální fitness
1	149442	512
2	141183	505
3	139386	529
4	**128454**	**538**

Tabulka 6.2: Zpracovaný počet generací a maximální nalezená fitness populací za 180 minut běhu programu.



Obrázek 6.9: Zobrazení fitness v prvních 5000 generacích.

Celkově bylo dosaženo maximální fitness v populaci číslo 4, její hodnota byla 536 z maximálního ohodnocení 901. Předchozí obrázek, znázorňuje průběh zlepšování partikulárního řešení v prvních 5000 generacích.

Zcela jistě by se dalo naleznout více ohodnocené řešení, ať již pomocí lepší volby parametrů genetického algoritmu tak delším časovým průběhem genetického algoritmu. Z důvodů autorských práv nemohu vizuálně prezentovat řešení dosažené řešení Eternity II. Definice prvků je obsažena v souboru *board16def.h* a spustitelný program, jsou pod heslem zabalené soubory. Pokud si přejeme otestovat funkčnost programu je nutné odkomentovat makro "//#define ETERNITY_TEST" ve zdrojovém kódu. Test bude pak probíhat na modifikovaném a zmenšeném Eternity 2 na 6x6.

7 Závěr

Hlavním cílem této diplomové práce bylo vytvořit knihovnu pro genetické algoritmy, která podporuje více-jádrové procesory. Následně knihovnu otestovat a poté ji použít k hledání nějakého praktického řešení.

Diplomová práce byla rozložena do několika logických celků. Nejprve je rozebrána obecná problematika genetických algoritmů. Druhá kapitola se věnuje OpenMP, které je využito pro podporu výpočtu na více jádrech procesoru. Třetí kapitola popisuje implementaci vytvořené knihovny genetických algoritmů. Ve čtvrté kapitole jsou popsány implementované genetické operátory, včetně paralelizace popocí OpenMP a vytvořených typů migrací. Pátá kapitola je věnována testování knihovny. Testujeme výkon při různých úrovních OpenMP, vliv velikosti populace na rychlost nalezení řešení a vliv zvětšování počtu dimenzí problémů na rychlost nalezení řešení. První dvě testovací funkce jsou typické funkce u nichž hledáme minimum a používají se k testování genetických algoritmů, jedná se o Rastriginovu a Rosenbrockovu funkci. Třetí problém použitý k testování je rozložení n dam na šachovnici aby se vzájemně neohrožovali. V šesté kapitole jsme vytvořili řešitel soutěžního puzzle Eternity II, které bylo vydáno v roce 2007 a za jeho vyřešení je vypsána odměna 2 miliony dolarů. K aktuálnímu datu puzzle ještě nikdo nevyřešil.

Výsledkem práce je podle mého názoru celkově univerzální knihovna zahrnující nejčastěji používané genetické operátory a reprezentace problémů. Všechny dosažené výsledky dopadli podle logického očekávání. Zdrojové kódy, včetně výsledů a průběhů evolucí problémů, jsou přiloženy na DVD. Obsah přiloženého DVD je následující:

- *CODE*\ : Vytvořená knihovna genetických algoritmů.
- *DOCUMENTATION*\ : Vytvořená online dokumentace.
- *ETERNITY_II*\ : Řešitel puzzle Eternity II.
- *TEST_RESULTS*\ : Výsledky testování knihovny na zvolených testovacích problémech.

V diplomové práci není popsán zdrojový kód knihovny ani kompletní zdrojový kód řešených problémů. Prostor v rozsahu diplomové práce mi bohužel neumožňuje popisovat kód takto rozlehlé knihovny. Z tohoto důvodu jsem se omezil pouze na popis jak knihovnu genetických algoritmů používat. Online dokumentace k vytvořené knihovně se nachází na http://diplomka.x-host.cz/GADOX/.

8 Použitá literatura

[01] Prata, Stephen: *Mistrovství v C++, 3. aktualizované vydání*. Computer Press, 10/2008, ISBN: 978-80-251-1749-1, EAN: 978-80-251-1749-1

[02] The C++ Resources Network, 2008, Dostupné z <http://www.cplusplus.com/>

[03] Hynek, Josef: *Genetické algoritmy a genetické programování*. Grada Publishing, 2008, 200 s. (ISBN: 978-80-247-2695-3).

[04] Hynek, Josef: Proč genetické algoritmy selhávají?. *Acta Electrotechnica et Informatica*. Vol. II, No. 4/2002, pp. 39-44 (ISSN 1335-8243).

[05] Hynek, Josef: Genetické algoritmy a problém N dam. *Acta Electrotechnica et Informatica*. Vol. II, No. 1/2002, pp. 27-32 (ISSN 1335-8243).

[06] Hynek, Josef: Genetické algoritmy. *Ekonomie a Management.* Vol. IV, No. 3/2001, pp. 33-38 (ISSN 1212-3609).

[07] Hynek, J.: *Genetic Algorithms for the N-Queens Problem*. In: Arabnia, H.R., Mun, Y. (Eds.): Proceedings of the 2008 International Conference on Genetic and Evolutionary Methods (GEM2008). Las Vegas, NV, July 14-17, 2008. CSREA Press, USA, pp. 64-68 (ISBN: 1-60132-069-8).

[08] Hynek, J.: *Genetic Algorithms for the N-Queens Problem*. In: Arabnia, H.R. (Ed.): Proceedings of the 2008 World Congress in Computer Science, Computer Engineering, and Applied Computing (WORLDCOMP'08). Las Vegas, NV, July 14-17, 2008. CSREA Press, USA, p. 5 (CD ROM; ISBN: 1-60132-090-6).

[09] Chu, P.C.; Beasley J.E.: *A Genetic Algorithm for the Multidimensional Knapsak Problem*. Springer Netherlands, Vol IV, Number 1, 1998, ISSN1381-1231 (Print) 1572-9397. Dostupné z: <http://www.springerlink.com/content/u7663408p38l03k2/fulltext.pdf>

[10] Raidl Günther R.; Kodydek, Gabriele: *Genetic Algorithms for the Multiple Container Packing Problem*, Lecture Notes in Computer Science, 1998, Springer Berlin / Heidelberg, Vol. 1498, ISBN: 978-3-540-65078-2.
Dostupné z: <http://www.springerlink.com/content/86m0347312347225/fulltext.pdf>

[11] Levenhagen , Jens; Bortfeldt , Andreas; Gehring, Hermann: *Path Tracing in Genetic Algorithms Applied to the Multiconstrained Knapsack Problem*, Lecture Notes in Computer Science, 2001, Springer Berlin / Heidelberg, Vol. 2037, ISBN: 978-3-540-41920-4.
Dostupné z: <http://www.springerlink.com/content/7wgl39twunh4e0g8/fulltext.pdf>

[12] Alonso, César L.; Caro, Fernando; Montaña, José Luis: *A Flipping Local Search Genetic Algorithm for the Multidimensional 0-1 Knapsack Problem*, Lecture Notes in Computer Science, 2006, Springer Berlin / Heidelberg, Vol. 4177, ISBN: 978-3-540-45914-9.
Dostupné z: < http://www.springerlink.com/content/rmt8g0220t322241/fulltext.pdf>

[13] Szeto, Kwok Yip; Zhang, Jian: Adaptive Genetic Algorithm and Quasi-parallel Genetic Algorithm - Application to Knapsack Problem, Lecture Notes in Computer Science, 2006, Springer Berlin / Heidelberg, Vol. 3743, ISBN: 978-3-540-31994-8.
Dostupné z: <http://www.springerlink.com/content/06517126q3366751/fulltext.pdf>

[14] Li, Hong; Jiao, Yong-Chang; Zhang, Li; Gu, Ze-Wei: *Genetic Algorithm Based on the Orthogonal Design for Multidimensional Knapsack Problems*, Lecture Notes in Computer Science, 2006, Springer Berlin / Heidelberg, Vol. 4221, ISBN: 978-3-540-45901-9.
Dostupné z: <http://www.springerlink.com/content/q10512l7344k6g75/fulltext.pdf>

[15] Akçay, Yalçın; Li, Haijun; Xu, Susan H.: *Greedy algorithm for the general multidimensional knapsack problem*, Annals of Operations Research, 2006, Springer Netherlands, Vol. 150, Number 1, ISSN: 0254-5330 (Print) 1572-9338.
Dostupné z: <http://www.springerlink.com/content/48x983ru4n5621l1/fulltext.pdf>

[16] Anagun, A.S.; Saraç, Tugba.: *Optimization of Performance of Genetic Algorithm for 0-1 Knapsack Problems Using Taguchi Method*, Lecture Notes in Computer Science, 2006, Springer Berlin / Heidelberg, Vol. 3982, ISBN: 978-3-540-34075-1.
Dostupné z: <http://www.springerlink.com/content/t8062w28r213376p/fulltext.pdf>

[17] Saraç, Tugba; Sipahioglu, Aydin: *A Genetic Algorithm for the Quadratic Multiple Knapsack Problem*, Lecture Notes in Computer Science, 2007, Springer Berlin / Heidelberg, Vol. 4729, ISBN: 978-3-540-75554-8.
Dostupné z: <http://www.springerlink.com/content/31205814036147w7/fulltext.pdf>

[18] Singh, Alok; Baghel, Anurag Singh: *A New Grouping Genetic Algorithm for the Quadratic Multiple Knapsack Problem*, Lecture Notes in Computer Science, 2007, Springer Berlin / Heidelberg, Vol. 4446, ISBN: 978-3-540-71614-3.
Dostupné z: <http://www.springerlink.com/content/b7x155k028067550/fulltext.pdf>

[19] Chen, Rung-Ching; Jian, Cheng-Huei: *Solving Unbounded Knapsack Problem Using an Adaptive Genetic Algorithm with Elitism Strategy*, Lecture Notes in Computer Science, 2007, Springer Berlin / Heidelberg, Vol. 4743, ISBN: 978-3-540-74766-6.
Dostupné z: <http://www.springerlink.com/content/k10576m344076233/fulltext.pdf>

[20] Pohlheim, Hartmut <support@geatbx.com>: GEATbx - The Genetic and Evolutionary Algorithm Toolbox for Matlab [Online Documentation].
Dostupné z: <http://www.geatbx.com/>

[21] The MathWorks <support@mathworks.com>: Genetic Algorithm and Direct Search Toolbox v 2.4.1 [Online Documentation].
Dostupné z: <http://www.mathworks.com/products/gads/>

[22] Leaders for Manufacturing, Dual-Degree Program - MBA & MS Engineering <http://lfm.mit.edu/>: GAlib - A C++ Library of Genetic Algorithm Components [Online Documentation].
Dostupné z: <http://lancet.mit.edu/ga/>

[23] Kaosar, Golam; Shorfuzzaman, Mohammad; Ahmed, Sayed: Solving the n-Queens Problem Using Genetic Algorithms, 2003.

[24] Nadel, B.A.: *Representation selection for constraint satisfaction: A case study using n-queens*. IEEE Expert, June, pp. 16-23, 1990.

[25] Zhong, Weicai; Liu, Jing; Jiao, Licheng: *Evolutionary Agents for n-Queen Problems*, Lecture Notes in Computer Science, 2007, Springer Berlin / Heidelberg, Vol. 3612, ISBN: 978-3-540-28320-1.
Dostupné z: <http://www.springerlink.com/content/8715a2cljq2qc6t2/fulltext.pdf>

[26] Eastridge, Richard; Schmidt,Cecil: *Solving N-Queens With a Genetic Algorithm and It's Usefulness in a Computational Intelligence*, Consortium for Computing Sciences in Colleges, 2008.

[27] Mitchell, Melanie: *An Introduction to Genetic Algorithms*, A Bradford Book The MIT Press, Cambridge, Massachusetts, 1999, First MIT Press paperback edition in 1998, ISBN 0−262−13316−4 (HB), 0−262−63185−7 (PB).
Dostupné z: <http://www.esnips.com/doc/e9ef63a3-28c1-40d9-898b-690b78341f2b/MIT-Press---An-Introduction-to-Genetic-Algorithms>

[28] Pohlheim, Hartmut <support@geatbx.com>: GEATbx – Examples of Objective Functions, version 3.8, 12/2006 [Online Documentation].
Dostupné z: <http://www.geatbx.com/download/GEATbx_ObjFunExpl_v38.pdf>

[29] Šeda, Miloš: Graph Theory, Brno University of Technology, FME, November 2006.
Dostupné z: < http://uai.fme.vutbr.cz/seda/teorie-grafu/TG06_eng.pdf >

[30] Wikipedie, otevřená encyklopedie: Extrém funkce, 18.2.2009.
Dostupné z < http://cs.wikipedia.org/wiki/Extrém_funkce >

[31] Zhou, Tao: *TSP Problem solution based on improved Genetic Algorithm*, paper in Natural Computation 2008 ICNC '08 Fourth International Conference on Oct. 2008, 2008-11-07, ISBN: 978-0-7695-3304-9. Dostupné z:
<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4666932&isnumber=4666792>

[32] Kajánek, Petr; Přikryl, Josef: *Seznámení se s genetickými algoritmy*.
Dostupné z: < http://mujweb.cz/www/prikrylj/Genetickealgoritmy.html#6.4) >

[33] Michalewicz, Z.: *Genetic Algorithms + Data Structures = Evolution Programs*. 3rd edition, Springer, Berlin, 1996.

[34] Jorge Muñoz, German Gutierrez, Araceli Sanchis: *Evolutionary genetic algorithms in a constraint satisfaction problem: Puzzle Eternity II*. 10th International Work-Conference on Artificial Neural Networks (Biological Inspired Systems. Computational and Ambient Intelligence) (IWANN 2009), strany 720-727, Salamanca, Španělsko, 10-12 2009.
Dostupné z:
<http://www.caos.inf.uc3m.es/~jorge/papers/2009_IWANN/JorgeMunoz2009b.pdf>