VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

DIFUZNÍ EVOLUČNÍ ALGORITMUS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

ZBYNĚK ŽUNDÁLEK

BRNO 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

DIFUZNÍ EVOLUČNÍ ALGORITMUS

DIFFUSION EVOLUTIONARY ALGORITHM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — ZBYNĚK ŽUNDÁLEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. JIŘÍ JAROŠ, Ph.D.
SUPERVISOR

BRNO 2011

Abstrakt

Tato bakalářská práce se zabývá paralelizací difuzních evolučních algoritmů pomocí knihovny OpenMP. Náplní teoretické části práce je stručný úvod do problematiky evolučních a genetických algoritmů následovaný popisem paralelní verze těchto algoritmů na systémech se sdílenou pamětí. Teoretická část je zakončena rozborem klíčových vlastností knihovny OpenMP. Praktická část podrobně popisuje dvě možné varianty implementace difuzního evolučního algoritmu – synchronní a asynchronní. V experimentální části je na problému N dam provedeno srovnání těchto dvou variant s důrazem na maximální dosažené zrychlení. Kvalita nalezeného řešení je dále zkoumána s ohledem na použitý typ okolí, topologie a operátoru nahrazení.

Abstract

This bachelor thesis deals with a parallelization of cellular evolutionary algorithms using OpenMP. The theoretical part of the thesis contains an introduction to evolutionary and genetic algorithms followed by the description of their parallel implementation on shared memory systems. This part is completed with the OpenMP key features analysis. The practical part of this thesis describes two possible implementations of a diffusion evolutionary algorithm; synchronous and asynchronous. The comparison of achievable performance of these two methods carried out on the N-Queen problem is provided in the experimental part of the thesis. The quality of found solutions is further examined with respect to the neighborhood size, topology and the replacement operator of the diffusion evolutionary algorithm.

Klíčová slova

Evoluce, evoluční algoritmy, genetické algoritmy, difuzní, paralelizace, OpenMP.

Keywords

Evolution, evolutionary algorithms, genetic algorithms, diffusion, parallelization, OpenMP.

Citace

Zbyněk Žundálek: Difuzní evoluční algoritmus, bakalářská práce, Brno, FIT VUT v Brně, 2011

Difuzní evoluční algoritmus

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Jiřího Jaroše, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Zbyněk Žundálek
18. května 2011

Poděkování

Chtěl bych poděkovat Ing. Jiřímu Jarošovi, Ph.D, za odbornou pomoc a cenné rady. Rád bych poděkoval mým rodičům za trpělivost a podporu při studiu. Dále pak Doc. Ing. Lukáši Sekaninovi, Ph.D, který mi umožnil testování programu za účelem analýzy výsledků.

Obsah

Úvod

Rychlý vývoj v oblasti techniky a matematiky v minulém století, rapidní rozvoj počítačů, to všechno mělo velký vliv na rozmach umělé inteligence, která je v současnosti jedním z nejrychleji se rozvíjejících oborů zahrnující mnoho odvětví. Schopnost řešit netriviální problémy, zastoupit člověka v některých jeho činnostech, v těchto případech nám může pomoci právě umělá inteligence. Jedním z odvětví, které v poslední době zaznamenalo rozmach, jsou evoluční algoritmy, popsány v kapitole 1.

Kapitola 2 je věnována nejrozšířenějšímu typu evolučních algoritmů, konkrétně genetickým algoritmům. Je zde vysvětlen princip a základní pojmy, se kterými se můžeme často setkat.

Evoluční algoritmy (EA) jsou velmi často s úspěchem nasazované k řešení nejrůznějších problémů. Ať už se jedná o problémy návrhové, optimalizační, nebo jiné, jejich řešení je však většinou výpočetně náročné. Snad proto je v moderních počítačích stále více uplatňováno paralelní zpracování, které se ukázalo jako velmi výhodné z hlediska ceny i vysokého výkonu.

Tato bakalářská práce se zabývá difuzním evolučním algoritmem a jeho paralelizací pomocí OpenMP. U evolučních algoritmů je k nalezení řešení využita celá populace jedinců. Ohodnocení každého jedince lze provádět nezávisle a bývá většinou velmi časově náročné, proto je u evolučních algoritmů velmi výhodné využití paralelizace. Jednou z možností je použití OpenMP, což je soustava direktiv pro překladač/preprocesor, popisující paralelizaci ve zdrojovém kódu. OpenMP je vyčleněna samostatná kapitola, ve které jsou popsány základní architektury, direktivy a klauzule. Více v kapitole 4.

Difuzní evoluční algoritmus je forma evolučního algoritmu, kde jsou jedinci prostorově uspořádáni, a na rozdíl od panmiktické populace probíhá křížení mezi jedinci pouze v určitém okolí. Tato modifikace klasického schématu nám nejen výrazně zjednodušuje využití paralelizace, ale přináší i pozvolné šíření slibných řešení skrz populaci, a nabízí tak větší možnost prohledávání stavového prostoru. [1] Difuzní evoluční algoritmus je podrobněji popsán v kapitole 3.

V kapitole 5 je představen známý problém N dam, který bývá s oblibou používaný pro testování různých technik. Tento problém je inspirován hrou šachy, konkrétně šachovou figurkou dáma a jejím umístěním na šachovnici.

Kapitola 6 se zabývá návrhem a implementací. Jsou zde popsány a zdůvodněny použité techniky. Představeny jsou implementace synchronní i asynchronní verze, u kterých jsou uvedeny některé problémy, na které jsem během řešení narazil.

Experimentální výsledky jsou uvedeny v kapitole 7. Tato kapitola uvádí dosažená zrychlení, měřená při různém nastavení, u obou verzí (synchronní i asynchronní). Dále byl zkoumán vliv velikosti okolí na selekční tlak a čas potřebný k nalezení správného řešení. V závěru kapitoly jsou uvedeny grafy zachycující vliv obnovy na diverzitu populace.

Závěr bakalářské práce obsahuje shrnutí dosažených výsledků a návrhy na případná vylepšení.

Kapitola 1

Evoluční algoritmy

V této kapitole, rozdělené do čtyřech částí, jsou podrobněji popsány evoluční algoritmy. První část se zabývá jejich biologickým pozadím, v částech následujících jsou postupně vysvětleny základní pojmy, se kterými se můžeme v této oblasti setkat. Následně jsou popsány typické vlastnosti EA, jejich průběh a možnosti využití.

1.1 Darwinova evoluční teorie

Podle Darwinovy teorie založené na přírodním výběru spolu jedinci v rámci populace soupeří o zdroje potřebné k přežití a možnost rozmnožování. Silnější jedinci (jedinci s lepšími vlastnostmi) mají větší šanci přežít a rozmnožit se. Děje se tak na úkor slabších jedinců, kteří mají menší počet potomků, nebo zemřou bez potomstva. Potomci dědí část vlastností rodičů, které jsou však ovlivněny náhodnými mutacemi. Tento proces se opakuje po mnoho generací, čímž dochází k adaptaci populace na okolní prostředí.

U EA je řešení problému analogií vývoje druhu, kdy dochází k postupnému zlepšení výsledku (adaptaci jedince) v řešeném problému (prostředí). „Evoluční algoritmy jsou stochastické heuristické vyhledávací metody, založené na principech přírodní evoluce, nebo lépe na Darwinově evoluční teorii [2].“

1.2 Základní pojmy evolučních algoritmů

- **Populace:** Množina jedinců, kteří se zúčastňují evolučního procesu.
- **Jedinec:** Jedno z řešení daného problému. Zakódování jedince nazýváme *genotypem*, to co nám dané zakódování představuje nazýváme *fenotypem*. Např. genotyp 0011 může znamenat hodnotu funkce 3, nebo pořadí navštívených měst u problému obchodního cestujícího. Genotyp obsahuje jeden a více *chromozómů*, které obsahují *geny*. Konkrétní hodnotu genu nazýváme *alelou*.
- **Gen:** Parametr problému.
- **Generace:** Jednotlivé populace v čase nazýváme generacemi.
- **Fitness:** Hodnota fitness označuje kvalitu jedince. Na základě této hodnoty má jedinec větší, či menší šanci zúčastnit se reprodukčního procesu.
- **Selekce:** Výběr jedinců ke křížení na základě jejich fitness funkce.

- **Křížení:** Proces tvorby nového jedince na základě dvou, nebo i více jedinců.
- **Mutace:** Náhodná, malá změna nově vzniklého jedince.
- **Obnova populace:** Proces tvorby nové populace na základě potomků (nových jedinců) a rodičů (starých jedinců).

1.3 Průběh evolučních algoritmů

Obecné schéma je u všech EA stejné, jednotlivé varianty se však liší v typech použitých operátorů, selekci, nebo v obnově populace. Jedná se o algoritmus č. 1. [1], [2]

Algoritmus 1: *Průběh evolučního algoritmu*

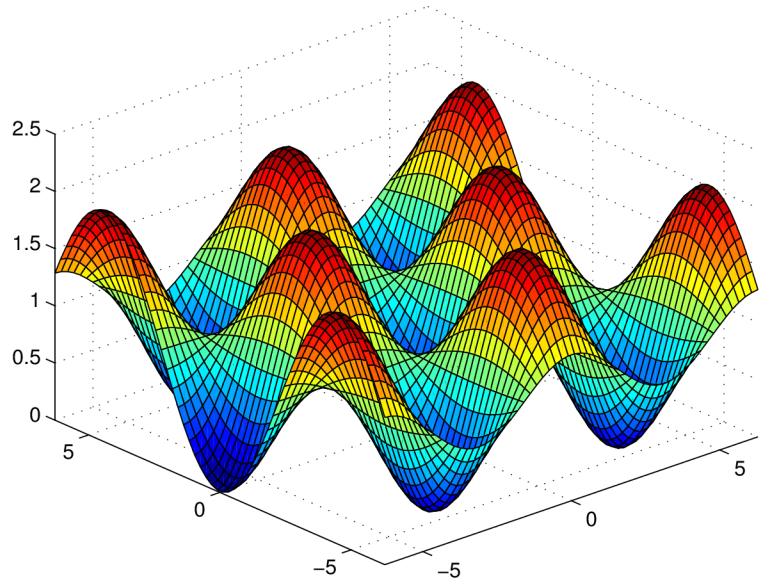
```
Initialization();
Generation = 0;
while !Termination_criterion do
    Evaluate_Finess();
    Selection();
    Reproduction();
    Replacement();
    Generation + +;
endw
```

- *Initialization* — vytvoření náhodné populace
- *Vynulování počítadla generací*
- *Dokud není splněno ukončovací kritérium*
- *Evaluate_Fitness* — ohodnocení každého jedince v populaci pomocí fitness funkce
- *Selection* — výběr jedinců k reprodukci
- *Reproduction* — tvorba nových jedinců pomocí operátorů (křížení, mutace, ...)
- *Replacement* — obnova populace
- *Zvýšení počítadla generací*

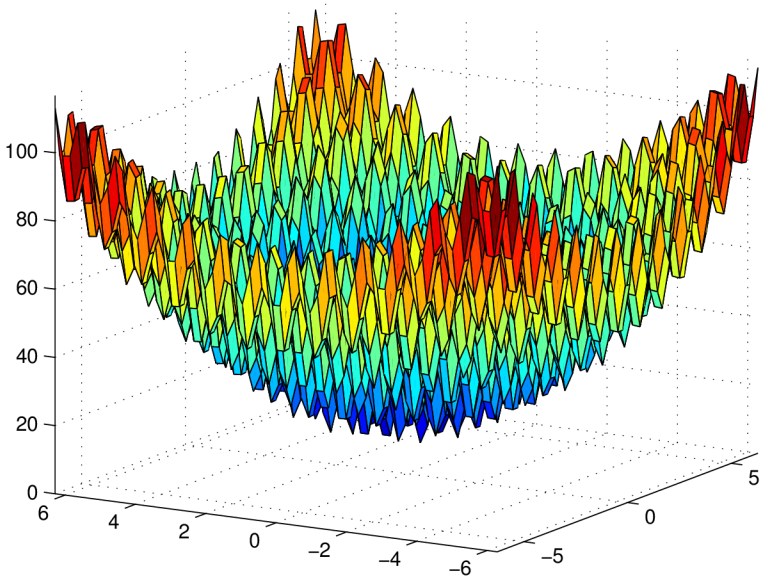
1.4 Využití

EA nalézají široké uplatnění v optimalizaci, plánování, strojovém učení, evolučním hardware, nebo v počítačových hrách. Jsou robustní a snadno modifikovatelné. Běžné je nasazení v problémech, kde je stavový prostor příliš rozsáhlý, nebo klasické metody selhávají. Typické je jejich použití v multikriteriálních a multimodálních úlohách. Na obr. 1.1 a 1.2 jsou uvedeny příklady multimodálních funkcí. EA mají však některé nevýhody — nezajišťují nalezení globálního optima.

Pokud uvážíme dostatečně velkou množinu problémů, potom všechny algoritmy prohledávání stavového prostoru mají v průměru stejnou výkonnost. Do EA můžeme však vložit co nejvíce znalostí o daném problému tak, abychom dosáhli vyšší účinnosti algoritmu než náhodné prohledávání. [3]



Obrázek 1.1: Griewangova funkce



Obrázek 1.2: Rastriginova funkce

Kapitola 2

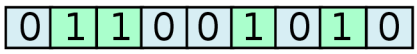
Genetické algoritmy

Evoluční algoritmy jsou obecný pojem, zahrnující především genetické algoritmy, evoluční strategie, genetické programování a evoluční programování [4]. Genetické algoritmy (GA) jsou z nich patrně nejznámější, za jejich otce je považován americký vědec John Holland. Základním stavebním kamenem GA je operátor křížení. [5] Bývá však také předmětem mnoha protichůdných názorů [6].

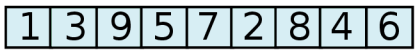
2.1 Reprezentace jedince

V GA je jedinec nejčastěji reprezentován pomocí binárního zakódování (obr. 2.1). Využívá se i zakódování permutační (obr. 2.2), nebo reálné, ta však mohou vyžadovat speciální typy operátorů. Na místo klasické binární reprezentace bývá někdy použit *Grayův kód*, a to kvůli menší *Hammingově vzdálenosti*. [6]

Pro zakódování jedince je většinou použit pouze jeden chromozóm, lze však využít i jiné varianty např. *diploidní GA*, kdy jsou pro zakódování jedince použity chromozómy dva. My však v této práci budeme uvažovat pouze variantu s jedním chromozómem tzv. *haploidní GA*. V klasických GA je pro zakódování použita pevná délka chromozómu, existují však i varianty, kde mají chromozómy proměnnou délku. [6]



Obrázek 2.1: Binární zakódování



Obrázek 2.2: Permutační zakódování

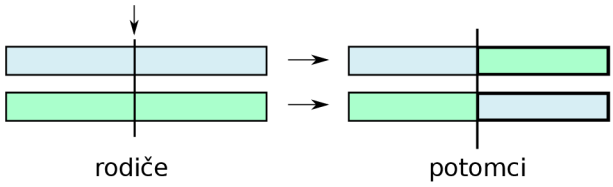
2.2 Křížení

Proces křížení hraje u GA klíčovou roli a je hnací silou evoluce. Existuje celá řada typů křížení, základní myšlenkou je vznik nového jedince, který získá „kvalitní“ genetický materiál od rodičů. Dochází tak zde k postupnému promíchání stavebních bloků (slibných sekvencí genů) a následné konvergenci k maximální fitness. Křížení má však za následek i rozbíjení těchto bloků. Proto je v některých implementacích použit pouze operátor mutace. [7]

Touto problematikou se zabývá teorie schémat, podrobnosti lze nalézt např. v [8], přesahují však rámec této práce.

2.2.1 Jednobodové křížení

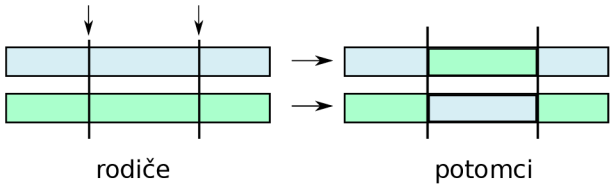
Nejjednodušší typ křížení je křížení jednobodové, jeho průběh je znázorněn na obr. 2.3. Je náhodně vygenerován bod křížení, a poté jsou prohozeny části chromozómů ležící za bodem křížení. Vznikají tak dva potomci.



Obrázek 2.3: Jednobodové křížení

2.2.2 Dvoubodové křížení

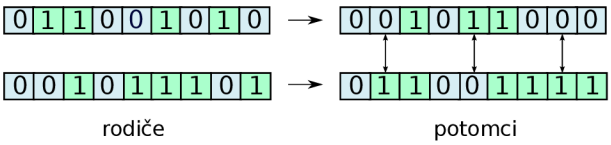
Dvoubodové křížení je velmi podobné křížení jednobodovému. Jak již název napovídá, jsou náhodně zvoleny dva body křížení a prohozeny jsou části chromozómů ležící mezi nimi. Také v tomto případě vznikají dva potomci. Průběh je znázorněn na obr. 2.4.



Obrázek 2.4: Dvoubodové křížení

2.2.3 Uniformní křížení

Uniformní křížení prochází celým chromozómem a s pravděpodobností p, provádí výměnu genů mezi rodiči. Vznikají tak dva potomci (obr. 2.5). Je jasné, že tento typ křížení s sebou přináší větší „rozbíjení“ stavebních bloků, avšak v některých složitých úlohách je právě tento přístup výhodný, neboť napomáhá řešení problému s uváznutím v lokálním extrému.

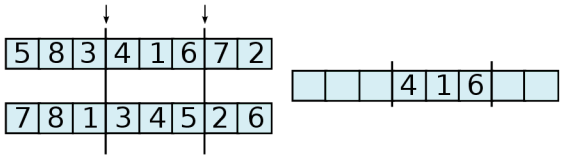


Obrázek 2.5: Uniformní křížení

2.2.4 Permutační křížení (PMX)

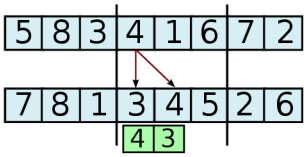
Problém nastává, pokud není zakódování binární, ale např. permutační. Pokud bychom pouze prohodili geny mezi body křížení, mohlo by se nám stát, že dostaneme chomozóm, který nebude validní. Mohl by obsahovat některá čísla vícekrát, nebo by nebylo dané číslo vůbec zastoupeno. Tyto problémy řeší speciální operátory křížení, a jedním z nich je právě PMX. Postup je následující:

1. Vygenerujeme náhodně dva body křížení a zkopírujeme geny prvního rodiče mezi body křížení na stejné místo u potomka.



Obrázek 2.6: PMX — krok č. 1

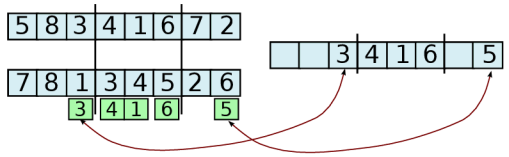
2. Procházíme geny u prvního rodiče zleva mezi body křížení a porovnáme ho s protilehlým genem u druhého rodiče. Pokud nejsou stejné, nalezneme pozici genu u druhého rodiče, který má stejnou hodnotu jako vybraný gen u prvního rodiče. Zkopírujeme je a prohodíme jejich pozice.



Obrázek 2.7: PMX — krok č. 2

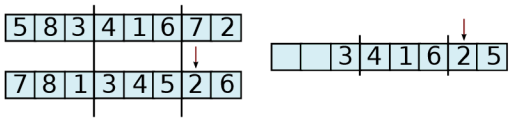
3. Postup v bodě 2 opakujeme pro všechny geny mezi body křížení u prvního rodiče.

4. Procházíme dříve zkopírované geny zleva (pouze ty, které neleží mezi body křížení) a umísťujeme je na stejné pozice do potomka.



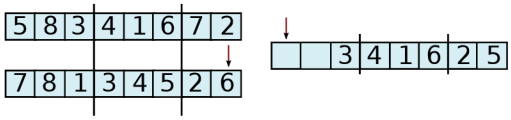
Obrázek 2.8: PMX — krok č. 4

5. Procházíme geny u druhého rodiče (zleva doprava), přičemž začínáme za druhým bodem křížení. Pokud hodnota genu není obsažena v potomkovi, je vložena na první volné místo za druhým bodem křížení potomka.



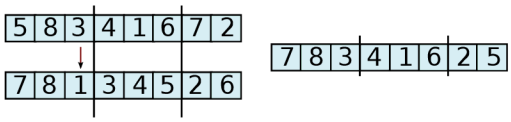
Obrázek 2.9: PMX — krok č. 5

6. Při dosažení posledního genu chromozómu pokračujeme prvním genem.



Obrázek 2.10: PMX — krok č. 6

7. Proces končí dosažením prvního bodu křížení u druhého rodiče.



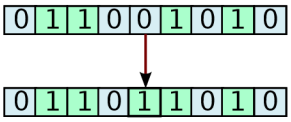
Obrázek 2.11: PMX — krok č. 7

Převzato z [9].

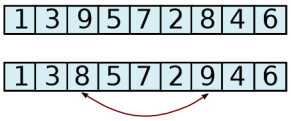
2.3 Mutace

Dalším důležitým operátorem používaným v GA je operátor mutace. Na rozdíl od křížení se jedná většinou o velmi jednoduchý operátor, který však hraje v průběhu evoluce důležitou roli. Brání příliš rychlému zjednotvárnění populace, její stagnaci a přináší do evoluce nové genetické informace.

U binárního zakódování je nejběžnější bitová negace, která se používá s pravděpodobností 0,0005 až 0,01. [6] Mutace je znázorněna na obr. 2.12. V případě permutačního zakódování je možné náhodně zaměnit hodnoty dvou genů (obr. 2.13).



Obrázek 2.12: Mutace — bitová negace



Obrázek 2.13: Mutace — permutační zakódování

2.4 Fitness funkce

Na základě fitness funkce jsou ohodnoceni všichni jedinci v populaci. Fitness funkce je specifická pro každý řešený problém, pro složité problémy je obvykle velice náročná na výpočet. Zvláštním případem je fitness funkce použita v *interaktivní evoluci*, kdy je do hodnocení jedinců zapojen uživatel. Tento nepříliš běžný přístup je využíván například v evolučním umění. Fitness funkce by měla od sebe dostatečne odlišovat kvalitu jednotlivých řešení.

Základním vyjádřením je *hrubá fitness*, která je vždy udávána v hodnotách přirozených problémové doméně. *Standartizovaná hodnota fitness* je vyjádřením hrubé hodnoty fitness do podoby, v které je stále menší numerická hodnota fintess žádanější. *Tento odstavec byl převzat z [7].*

2.5 Selekce

Selekce výrazně ovlivňuje průběh evoluce, jak již bylo zmíněno, větší šanci na reprodukci by měli mít jedinci s vyšší fitness.

Avšak výběr pouze nejlepších jedinců by měl za následek převládnutí silných jedinců v celé populaci. Tím by se výrazně zmenšila rozmanitost populace. Mohlo by tak dojít k rychlé konvergenci k suboptimálnímu řešení. Naopak zvolení kritéria, které by nedostatečně upřednostňovalo silné jedince, by mělo za následek velmi pomalou konvergenci.

S tímto problémem souvisí pojem *selekční tlak*, nebo také *selekční intenzita*. Je vyjádřena vztahem 2.1. [6] Kde $\overline{M}$ je průměrná fitness jedinců před selekcí, $\overline{M^*}$ je průměrná fitness po selekci a $\overline{\sigma}$ je rozptyl hodnot fitness před selekcí.

$$I = \frac{\overline{M^*} - \overline{M}}{\overline{\sigma}} \tag{2.1}$$

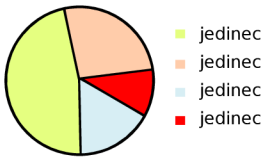
Pro vyjádření vlastností selekčních metod je využíván také pojem *takeover time*. Určuje čas, za který dojde k rozšíření nejlepšího jedince po celé populaci (nahradí všechny ostatní), bez použití rekombinačních a mutačních operátorů.

2.5.1 Ruleta

Prvním zástupcem selekce je mechanismus zvaný *ruleta*. Princip je velice jednoduchý, je zobrazen na obr. 2.14. Každému jedinci je přidělen výsek na této pomyslné ruletě. Velikost úseku přiděleného každému jedinci je proporcionální jeho fitness ohodnocení (jedinci jsou seřazeni sestupně od nejvyšší fitness hodnoty).

Následně je vygenerováno náhodné číslo, podle kterého se rozhodne o výběru jedince. Pravděpodobnost, že bude daný jedinec vybrán, je určena vzorcem 2.2, kde n je velikost populace, P_x je pravděpodobnost výběru jedince x a f představuje fitness ohodnocení.

Hlavní problém nastává, pokud se vyskytne jedinec s příliš vysokou fitness hodnotou vzhledem k ostatním. Jak je patrné ze vzorce 2.2 může dojít ke ztrátě diverzity v populaci, kdy bude při selekci příliš upřednostňován tento jedinec na úkor ostatních. Tento jev lze samozřejmě odstranit za použití škálování.



Obrázek 2.14: Ruletový mechanismus

$$P_x = \frac{1}{\sum_{i=0}^{n} f_i} f_x \tag{2.2}$$

2.5.2 Lineární a exponenciální uspořádání

Principiálně velmi podobné ruletě, pravděpodobnost však není vypočtena přímo na základě hodnoty fitness, ale podle pořadí. Jedinci tedy musí být setříděni, děje se tak na základě fitness. Pro samotný výpočet pravděpodobnosti existuje několik variant (lineární, exponenciální). Tyto techniky odstraňují nežádoucí efekt předchozí metody, nevýhodu je však právě nutnost seřazení populace.

2.5.3 Turnaj

Tento algoritmus dosahuje výsledků, které jsou velmi blízké exponenciální selekci. Jeho největším přínosem je absence požadavku na setřídění populace a jednoduchost vlastní

selekce. Z tohoto důvodu je turnajová selekce často používána. Spočívá v tom, že z populace je náhodně vybráno k jedinců a vítěz (jedinec s nejvyšší fitness) je vybrán k další reprodukci. [4],[6]

2.5.4 Pokročilé techniky selekce

U difuzních evolučních algoritmů je použita prostorově omezená selekce, což znamená, že ke křížení mohou být vybráni pouze jedinci v určitém okolí. Z tohoto okolí je následně vybrán jedinec pomocí standartních technik, jako např. turnajová selekce, nebo lineární uspořádání. [1]

2.6 Obnova populace

Po selekci, křížení a mutaci přichází na řadu obnova populace. Není však zaručeno, že i přes vysokou fitness bude jedinec vybrán k reprodukci. Pokud vybrán je, i zde je nebezpečí, že bude operátory křížení/mutace změněn. Což může vést k situaci, kdy je nejlepší jedinec, nebo i více slibných jedinců, ztraceno.

Proto je používán *elitismus*, kde vybraný počet nejlepších jedinců v populaci zůstane, čímž je zaručeno přežití nejlepšího řešení. Dalším přístupem, který jde ještě dále je *setrvalý stav* (*steady state*), který zanechá většinu populace v nezměněném stavu. Mění se pouze několik málo jedinců. Naprostým opakem je pak *generativní obnova*, při níž je celá populace rodičů nahrazena potomky. [4], [6]

Výběr typu obnovy velmi závisí na typu řešeného problému.

2.7 Ukončovací kritérium

Na ukončovacím kritériu závisí délka běhu evoluce. Jako ukončovací kritérium lze zvolit počet generací, což je sice jednoduché, ale někdy může být efektivnější použít sofistikovanější řešení. Běžná je varianta, při níž je uchovávána průměrná hodnota fitness celé populace a sledován její průběh. Ukončovací kritérium pak závisí právě na tomto průběhu. Hojně využívané je i spojení, kdy je dán maximální počet generací, ale pokud již nedochází k uspokojivému růstu průměrné fitness, je běh ukončen.

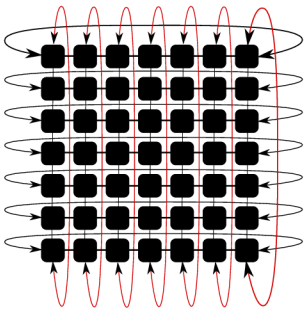
Kapitola 3

Difuzní EA

Difuzní algoritmy imitují situaci v přírodě, kdy je křížení jedinců ovlivněno geografickou izolací. Jak již bylo zmíněno, difuzní algoritmy mají díky tomuto omezení, v porovnání s panmiktickými algoritmy, pomalejší šíření slibných řešení. A právě kvůli těmto vlastnostem jsou vhodné pro řešení např. multimodálních úloh. [1]

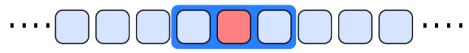
V této bakalářské práci byly implementovány dvě nejběžnější topologie, a to sice kruhová (3.1) a torus (3.2) topologie. Můžeme si všimnout, že jedinci jsou rozmístěni do pomyslné mřížky. Přičemž výběr topologie nám výrazně ovlivňuje vlastnosti evolučního algoritmu. [1]

Obrázek 3.1: Kruhová topologie



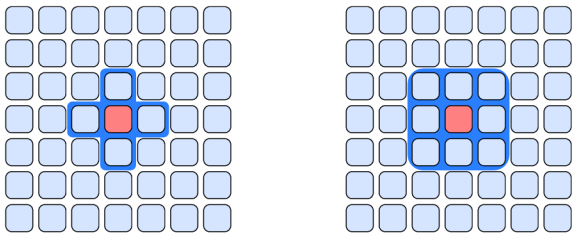
Obrázek 3.2: Torus topologie

Křížení je povoleno pouze mezi jedinci v *okolí*. U kruhové topologie je toto okolí tvořeno sousedními jedinci, tzn. může dojít ke křížení pouze u sousedních jedinců (buněk) v obou směrech, v různém rozsahu. Na obr. 3.3 je zobrazena varianta s velikostí okolí jedna.



Obrázek 3.3: Křížení — kruhová topologie

U torus topologie patří mezi nejpoužívanější typ okolí von Neumannovo 3.4(a) a Moorovo 3.4(b). Obě varianty jsou zobrazeny s velikostí okolí jedna.



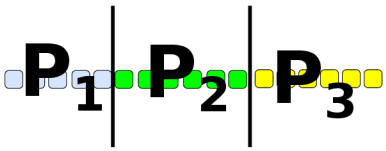
(a) von Neumannovo (b) Moorovo

Obrázek 3.4: Torus okolí 3.4(a), 3.4(b)

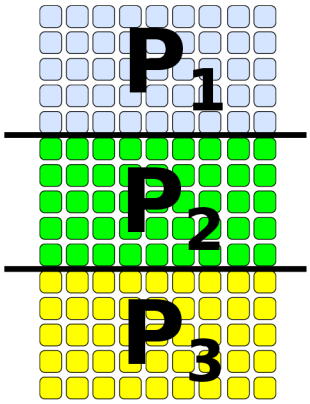
3.1 Synchronní difuzní EA

Pro víceprocesorový systém je základem rozdělení mřížky mezi jednotlivé procesory, tzn. na každém procesoru jsou zpracováváni pouze určití jedinci (obr. 3.5 a 3.6). Na začátku je náhodně vytvořena populace jedinců. Jedinci jsou rozděleni mezi jednotlivé procesory. Každý jedinec je ohodnocen fitness funkcí, je provedena selekce v daném okolí (lze využít různé typy selekce), je provedeno křížení a mutace. Následuje obnova populace (opět lze využít různé strategie). Lze například vybrat potomka, pokud je lepší než rodič, nahrazovat jedince vždy a to nejlepším potomkem, nebo využít pravděpodobnostní podobu obnovy.

Důležitým rozdílem oproti asynchronní verzi je, že evoluce probíhá po „ucelených krocích“. Tedy například: je zahájen krok, ve kterém má být ohodnocena populace pomocí fintess funkce. Teprve po dokončení ohodnocení všech jedinců (bez ohledu na to, kterému procesoru jsou přiděleni) je možno přistoupit k dalšímu kroku (selekci). Tzn. na konci každého kroku je „bariéra“. Každý procesor tak zpracovává svou část populace a jediným problémem jsou *hraniční jedinci* (jedinci, kteří mohou být použiti při křížení více procesory).



Obrázek 3.5: Kruhová topologie — procesory



Obrázek 3.6: Torus topologie — procesory

3.2 Asynchronní difuzní EA

Nicméně dokonalá synchronizace je pouze abstrakce: ve fyzikálních či biologických situacích je předpoklad synchronizace neudržitelný. V každém prostorově uspořádaném systému se signál nemůže šířit rychleji než světlo. Proto pro dané dimenze je nemožné, aby signál vyslaný společnými hodinami, zastihl kterékoli dva komponenty/agenty ve stejném čase. Samozřejmě tato anomálie u evolučních algoritmů nepředstavuje problém. Jde nám pouze o použití postupů, které dávají smysl výpočetně. Jak uvidíme dále, má asynchronní varianta některé vlastnosti, které jsou poměrně užitečné při řešení mnoha problémů. *"Tento odstavec byl převzat z [1]."*

Každý procesor tak dostane na starost svou *subpopulaci* (část jedinců), nad kterými probíhá „samostatný“ evoluční proces. Problémem je opět přeposílání hraničních jedinců.

U asynchronní verze jsou jedinci obnovováni v určitém sledu, liší se tak od synchronní verze, kde jsou obnoveni všichni „naráz“. Obecně je používáno několik typů těchto sledů obnovy:

- *Fixed line sweep* — obnova je prováděna stále ve stejném pořadí. Zleva doprava pro kruhovou topologii a zleva doprava, řádek po řádku pro torus topologii.
- *Fixed random sweep* — na začátku je vygenerováno pořadí obnovy jedinců, podle kterého je prováděna obnova po celý běh evoluce.
- *Fixed random sweep, uniform ...*

„Fixed line sweep varianta může mít spíše degenerativní účinek.“ [1]

Kapitola 4

OpenMP

Paralelní programovací jazyk musí poskytovat podporu pro tři základní aspekty paralelního programování: specifikaci paralelního provádění, komunikaci mezi vlákny a synchronizaci vláken. [10]

Existují různé typy těchto jazyků, které se liší v přístupu k poskytování těchto aspektů. Některé jazyky přidávají konstrukce v rámci jazyka, jiné poskytují direktivy, které mohou být vloženy do existujících sekvenčních programů v daném jazyce. Třetí typ umožňuje použití runtime knihoven. OpenMP používá druhý ze zmíněných přístupů, což je využití direktiv. [10]

Dle [11] je OpenMP definováno jako paralelní, multiplatformní programovací model pro multiprocesorové systémy. Jedná se o systémy se sdílenou pamětí a systémy s distribuovanou sdílenou pamětí.

OpenMP vzniklo roku 1997 a bylo vytvořeno firmami Kuck & Associates (KAI) a Silicon Graphics (SGI). Je dostupné v jazycích Fortran a C/C++, na všech architekturách a platformách Unix, Windows NT. Na vývoji se však dnes podílí spousta klíčových počítačových výrobců jako HP, Sun, IBM, Intel, nebo Compaq. V GNU GCC je OpenMP podporováno od verze 4.2 [12].

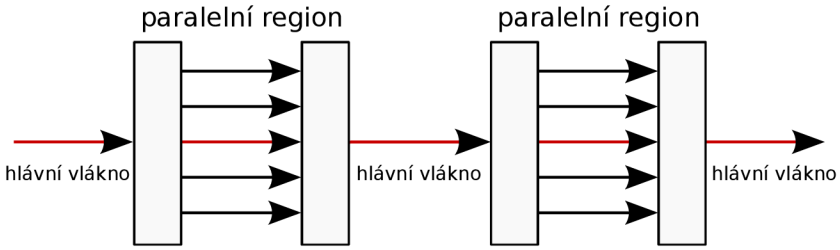
Použití OpenMP je relativně jednoduché, přenositelné a umožňuje paralelizovat pouze určité části kódu. Je zde i možnost přeložení překladačem, který OpenMP nepodporuje, jako sekvenční program. Mezi nevýhody patří, jak již bylo zmíněno, možnost využití pouze systémů se sdílenou pamětí a potřeba překladače/preprocesoru podporujícího OpenMP.

4.1 Vlákna

Základem OpenMP je hlavní vlákno (*master thread*), které běží od začátku programu a provádí kód sekvenčně. V OpenMP jsou k dispozici dvě základní konstrukce pro kontrolu paralelismu, jednou z nich je tzv. direktiva *omp parallel*. Pokud vlákno narazí na tuto direktivu, vytvoří skupinu vláken. Po skončení paralelního bloku jsou vlákna synchronizována a všechna kromě hlavního ukončena. Situaci ilustruje obr. 4.1.

Kód v těle této direktivy je prováděn všemi vlákny (pokud není explicitně určeno jinak). Na konci paralelního regionu je implicitní *bariéra*, existuje však možnost implicitní bariéru vypnout. V místě bariéry dochází k synchronizaci všech vláken tzn. „vlákna na sebe čekají“. Je tedy jasné, že provádění paralelního bloku s implicitní bariérou na konci bude tak rychlé, jako nejpomalejší vlákno.

V případě druhé konstrukce se jedná v podstatě o jakési „rozdělení práce“ mezi jed-

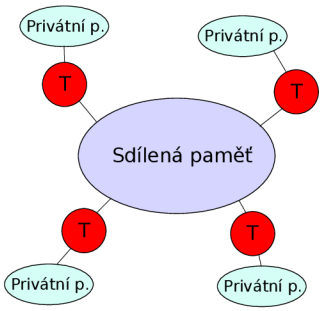


Obrázek 4.1: Paralelní blok

notlivá vlákna. Podrobnější vysvětlení této problematiky bude uvedeno v sekci věnované direktivě *omp parallel for*.

4.2 Paměť

Každé vlákno (*Thread*) má vlastní zásobník na který jsou ukládána příslušná data. Tento mechanismus je zobrazen na obr. 4.2.



Obrázek 4.2: Systém paměti u OpenMP

4.3 Klauzule

Rozsah platnosti každé proměnné může programátor specifikovat pomocí *klauzule*. Proměnná poté může nabýt třech základních podob: *shared*, *reduction*, nebo *private*. [10]

Private klauzule má za následek vytvoření privátní proměnné každým vláknem, ke které má přístup pouze vlákno „vlastnící“ tuto proměnnou. Její hodnota je však po vytvoření nedefinovaná.

Shared proměnná je, jak již její název napovídá, sdílená mezi jednotlivými vlákny. Tzn. pokud dojde ke změně hodnoty této proměnné, projeví se tato změna i u ostatních vláken. Použití sdílených proměnných však s sebou přináší řadu problémů, které budou vysvětleny dále.

4.4 Další použité direktivy

V této sekci budou podrobněji popsány některé běžně používané direktivy.

4.4.1 Direktivy single a master

Mezi často používané direktivy patří i direktiva *single*, kód v těle této direktivy je proveden pouze jedním vláknem. Na konci je implicitní bariéra, existuje však možnost implicitní bariéru vypnout. [13]

Velmi podobná předchozí je direktiva *master*. Kód v těle této direktivy je proveden pouze hlavním vláknem, přičemž na konci není implicitní bariéra. [13]

4.4.2 Direktiva critical

Občas potřebujeme, aby byla provedena část kódu pouze jedním vláknem ve stejném čase. K tomuto účelu slouží právě direktiva *critical*. Vymezuje místo v programu nazývané *kritická sekce*, do které může vstoupit pouze jedno vlákno ve stejném čase. Jedná se tak o určitý zámek, který znemožňuje přístup ostatním vláknům. [14]

4.4.3 Direktiva atomic

Obdobné chování má i direktiva *atomic*, jedná se však o „odlehčenou“ verzi direktivy critical. Zajišťuje nám, že specifikované místo v paměti bude měněno atomicky. [14]

4.4.4 Direktiva parallel for

Jde o jednu z nejpoužívanějších work-sharing direktiv. Pokud je znám předem počet iterací u smyčky a jednotlivé výpočty jsou mezi sebou nezávislé, můžeme použít tuto direktivu. Ta nám rozdělí smyčku tak, že každé vlákno dostane určitý počet iterací z původní smyčky. Tyto iterace jsou dále prováděny vlákny paralelně. Opravdovou výhodou je to, že pokud chceme paralelizovat určitou smyčku, můžeme toho docílit pouze nepatrnou úpravou původního kódu. Přičemž spoustu programů lze s úspěchem paralelizovat pouze s použitím této direktivy, kterou aplikujeme jen na některé význačné smyčky.

4.5 Přetahování o data

Při paralelním programování se sdílenou pamětí se však mohou objevit nové typy chyb, se kterými jsme se doposud u sekvenčního programování nesetkali. Tyto chyby jsou těžko odhalitelné, co je však zřejmě nejhorší, nemusí se projevit vždy. Nejčastěji se tyto chyby projevují při větším zatížení. Pak se stává, že například proměnná nabývá „podivných“ hodnot. K těmto chybám může docházet pokud dvě a více vláken přistupuje ke sdílené (shared) proměnné a alespoň jedno vlákno ji modifikuje. Tento typ chyby se nazývá *data-race*. [10]

K ošetření takových chyb nám slouží právě direktivy atomic/critical, kdy je vyloučen současný přístup vláken a tím pádem nemůže docházet k modifikaci jedním vláknem a zároveň přístupu ostatních.

4.6 Architektury

Paralelní počítače se sdílenou pamětí dělíme podle přístupu do paměti na *UMA* (*Uniform Memory Access*) a *NUMA* (*non UMA*). UMA se vyznačují tím, že každý procesor potřebuje pro přístup do libovolné části sdílené paměti stejný čas. [14]

Pro menší počítače se sdílenou pamětí je tato podmínka splněna, větší stanice jsou však většinou *cc-NUMA*. Což znamená, že přístupové doby do libovolné části sdílené paměti se mohou u jednotlivých procesorů lišit.

Podle hiearchického uspořádání pamětí platí, že čím je paměť více vzdálena od procesoru, tím má větší kapacitu. Přístupová doba však rapidně stoupá. Velmi rychlé provádění instrukcí procesorem by vyšlo v niveč, pokud bychom museli velmi dlouho čekat na načtení instrukcí/dat z paměti. Pokud však procesor přistupuje k určitému místu v paměti, tak s velkou pravděpodobností k němu bude přistupovat znovu (*datová lokalita*).

Byl tak zaveden systém *rychlých vyrovnávacích pamětí* (RVP), které v sobě uchovávají data/instrukce z pomalejší, ale větší paměti. U těchto dat/instrukcí je očekáváno další použití. Pokud se potřebná data/instrukce nacházejí v rychlé vyrovnávací paměti (*cache hit*), jsou použita. Situace, kdy v RVP data/instrukce nejsou se nazývá *cache miss*, a je načten celý blok (může obsahovat nepotřebná data). [15]

Vlákna přistupují k nezávislým datům, ležícím však na stejném řádku RVP. Pokud některé z vláken provede změnu dat, je tento řádek zneplatněn u všech vláken. Jedná se o *falešné sdílení* (*false sharing*). Pokud k této situaci dochází často, dojde ke snížení výkonu.[14]

4.7 Zrychlení

Při využití paralelizace je finální výkon ovlivněn několika aspekty. Je důležité paralelizovat velkou část kódu, ale pouze tohle nám nestačí. Paralelizujeme-li část kódu za použití mnoha procesorů, dostaneme se do situace, kdy část neparalelizovaného kódu může znatelně ovliňovat výsledný čas. Tento problém zachycuje Amdahalův zákon (4.1). [10]

$$S = \frac{1}{(1-F) + \frac{F}{S_p}} \tag{4.1}$$

F je část kódu, který je paralelizován, S_p je zrychlení dosaženo v paralelní sekci kódu a S potom definuje celkové dosažené zrychlení. Z výše uvedeného vzorce vyplývá, že pokud sekvenční část tvoří $\frac{1}{10}$ celkového kódu, výsledné zrychlení bude maximálně 10, i při použití nekonečného počtu procesorů.

Při střetnutí paralelního regionu je zapotřebí určité režie, jsou vytvořena vlákna, proměnné apod. Stejně tak na konci paralelního regionu/smyčky, zvláště pokud není vypnuta implicitní bariéra. Všechna tato režie nás něco „stojí“. Obecně platí, že nemá cenu paralelizovat část kódu, jejíž provádění netrvá dostatečně dlouho.

4.8 Efektivita

Pro výpočet efektivity je použit vzorec 4.2, kde E je výsledná efektivita v procentech, S je dosažené zrychlení a S_i je ideální možné zrychlení.

$$E = \frac{S}{S_i} 100\% \tag{4.2}$$

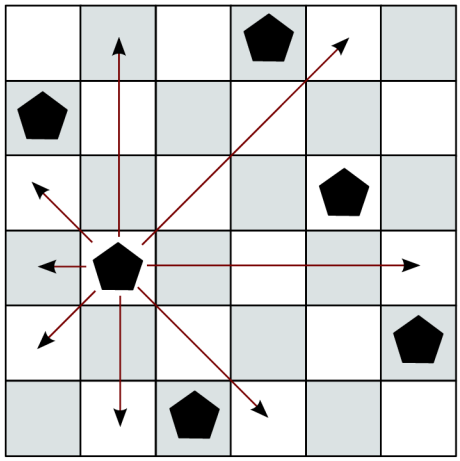
Kapitola 5

Problém N dam

Problém N dam je klasický kombinatorický problém, který je často studován různými metodami z oblasti umělé inteligence. Je to typický problém splňování podmínek (*constraint satisfaction problem*), který je jednoduše definovaný a zřejmě i proto je opakovaně používán jako vhodná testovací úloha. *Tento odstavec byl převzat z [4].*

Za úkol máme rozestavit na šachovnici dámy tak, aby se mezi sebou neohrožovaly. Dáma má možnosti pohybu stejné jako ve hře šachy, kterou je tento problém inspirován. Možné směry pohybu jsou v řádku, ve sloupci a v uhlopříčkách o libovolný počet políček. Šachovnice má rozměry $N \times N$ a počet dam umísťovaných na šachovnici je N.

Problém ilustruje obr. 5.1.



Obrázek 5.1: Problém N dam

Řešení tohoto problému však nalézá uplatnění ve spoustě praktických problémů. Podle [16] se jedná o aplikaci v řízení letového provozu, testování VLSI, plánování, v moderních komunikačních systémech a v mnoha dalších.

Kapitola 6

Návrh a implementace

Implementovány byly obě varianty difuzního EA: *synchronní* i *asynchronní*. Základním prvkem je knihovna *cEA_lib*, která obsahuje některé metody využívané v obou variantách a knihovna *cEA_generator_lib*, která obsahuje třídu generátoru. UML diagram tříd pro obě verze se nachází v příloze A.

Jedná se o jednoduchý kongruentní generátor pseudonáhodných čísel, jehož parametry byly převzaty z [17]. Generátor byl napsán jako samostatná třída, z důvodu jeho použití v paralelních verzích. Na začátku běhu programu jsou vytvořeny instance této třídy podle počtu vláken. Každé metodě pracující s náhodnými čísly je předáno *id* vlákna a poté podle tohoto *id* dojde k vybrání příslušného generátoru. Jako *seed* těchto generátorů je použit systémový čas, jenž je vynásobený *id* vlákna zvětšeného o jedničku. Je tak učiněno proto, že *id* hlavního vlákna je 0.

První metodou z knihovny cEA_lib je *create_chromosome*. Tato metoda vytvoří náhodný chromozóm v permutačním zakódování, podle požadované délky. Další metodou je *crossover_pmx*. Jak již název napovídá jde o pmx křížení, které je v této práci použito. Postup pmx křížení byl již popsán v kapitole 2, věnující se genetickým algoritmům. Operátor mutace nám poskytuje metoda *mutation_permutation*, která pomocí dvou náhodně vygenerovaných bodů zamění příslušné hodnoty genů v chromozómu. Velikost populace je uložena v *chromosome_length* a *column_length*.

Pro obnovu populace byly implementovány dvě varianty, označeny jako *generativní* a *steady state*. U klasických GA je pojmem *generativní* označen GA s „úplnou obnovou (vymírání rodičů), kdy je stará populace zcela nahrazena potomky“ [6]. V našem případě dochází k nahrazení rodiče potomkem vždy, i když má potomek menší fitness. Druhým typem je *steady state*, u klasických GA nazýván také jako částečná obnova. „Pouze jeden potomek nahradí nejslabšího jedince původní populace.“ [6] V naší implementaci dochází k nahrazení rodiče potomkem v případě, kdy má potomek vyšší fitness.

Jako ukončovací kritérium byl zvolen počet generací, lze však také běh předčasně ukončit při nalezení nejlepšího řešení.

6.1 Implementace synchronní verze

Pro načtení parametrů je použita funkce *get_parameters*. Následně dojde k vytvoření náhodné populace (metoda *init*), jejíž vlastnosti jsou nastaveny podle zadaných parametrů. Je volána metoda *make_fitness*, která projde celou populaci (v cyklu for) a provede ohodnocení všech jedinců. Tento for cyklus je paralelizován pomocí *#pragma omp parallel for*.

Základem evolučního běhu je smyčka *for*, jejíž počet iterací je roven zvolenému počtu generací. Z této smyčky jsou volány metoda *selection* a metoda *check_fitness*. Každá tato metoda obsahuje smyčku for, která prochází populací a vykonává příslušnou činnost nad aktuálním jedincem. Tyto smyčky jsou paralelizovány pomocí *#pragma omp parallel for*. Dochází tak k tomu, že je populace rozdělena mezi jednotlivá vlákna, které vykonávají příslušné, na sobě nezávislé operace.

6.1.1 Hraniční jedinci

Jak již bylo zmíněno dříve, problémem je křížení v „hraničních" oblastech. Tento problém je však řešen za pomoci dvou mřížek, kdy jsou jedinci postupující do další generace ukládáni do druhé mřížky. Protože jedinci použiti ke křížení (obsaženi v první mřížce), nejsou v průběhu křížení měněni, problém je vyřešen. Zbývá tak pouze po dokončení procesu křížení vyměnit mřížky, čímž dostáváme aktuální populaci.

6.1.2 Metoda selection

V této metodě dochází k výběru jedinců ke křížení. Následně je vytvořen potomek, který je ohodnocen a uložen do nové mřížky. Na konci této metody (cyklu for) je implicitní *bariéra*. Po ní následuje ono zmíněné prohození ukazatelů na mřížky. Tím dostáváme novou populaci.

6.1.3 Statistiky

Po obnově populace je volána metoda *check_fitness*, která ve for cyklu (paralelizovaného pomocí *#pragma parallel for*) zjišťuje nejlepší a nejhorší fitness pro každé vlákno. Děje se tak pomocí privátních proměnných *thread_best* a *thread_worst*. Do privátní proměnné *thread_sum* je ukládán součet fitness všech jedinců pro každé vlákno. Po skončení této operace jsou tyto hodnoty uloženy do sdíleného pole, je vypočítána celková průměrná fitness a vybrán nejhorší/nejlepší jedinec v rámci celé populace. Následně při volání metody *print_statistics()* jsou statistiky uloženy do souboru.

6.2 Implementace asynchronní verze

Asynchronní verze nepoužívá, na rozdíl od verze synchronní, dvě mřížky, ale pouze jednu. Jak již bylo řečeno, v každé subpopulaci probíhá evoluční proces samostatně, přičemž problémem jsou hraniční jedinci. Dalším problémem, který ovšem u synchronní verze nebyl, jsou statistiky. Klasický přístup ke statistikám, s využitím bariéry, selhává. U dvou libovolných subpopulací se mohou ve stejném čase lišit stupně generací, a proto nelze využít statistiky závislé na generacích.

Jednou z možností je využití speciálního vlákna, které vždy po určitém čase tyto statistiky ukládá do příslušného souboru.

6.2.1 Hraniční jedinci

Před samotným během evolučního procesu je vytvořena náhodná populace pomocí metody *init*. Dále jsou podle počtu vláken vypočteny hraniční body (souřadnice hraničních jedinců) pro jednotlivá vlákna. Ty jsou uloženy v *threads_borders*. Celá populace je ohodnocena pomocí fitness funkce, za pomoci metody *first_evaluation*, paralelizované pomocí

#pragma parallel for. Hraniční jedinci jsou uloženi do pole, přičemž každé vlákno kopíruje svoje hraniční jedince, ošetřeno pomocí direktivy #critical, dochází tak ke kopírování bloku dat najednou.

Je provedena selekce (vybrán vždy jedinec s nejvyšší fitness). Po skončení obnovy subpopulace je provedena obnova hraničních jedinců. Ostatní vlákna, pokud chtějí přistupovat k jiným, než ke svým jedincům, přistupují do toho pole (ošetřeno pomocí direktivy #critical).

6.2.2 Statistiky

Před samotnou smyčkou evolučního běhu jsou vytvořena dvě vlákna. Vlákno pokračující v běhu programu (evoluční běh) a vlákno speciální, které je využito pro tisk statistik do souboru.

Stejně jako v případě synchronní varianty, jsou po obnově populace ukládanány fitness pro každé vlákno zvlášť (*thread_sum*, *thread_best* a *thread_worst*). Ty jsou následně uloženy do souboru pomocí tohoto speciálního vlákna (vše je ošetřeno pomocí direktivy #critical). Do souboru jsou uloženy fitness pro nejhorší jedince a nejlepší jedince (a to jak pro každé vlákno, tak v rámci celé populace). Uložena je i průměrná fitness celé populace, kdy je součet všech fitness v populaci podělen počtem všech jedinců.

6.3 Benchmark

Na počátku jsem zvolil jednoduchý problém s velice triviální fitness funkcí. Tento problém je známý pod anglickým názvem *OneMax problem*. Použito je binární zakódování a hodnota fitness závisí na počtu jedniček v chromozómu. [18] Použit byl klasický operátor jednobodového křížení a mutace. Zrychlení po paralelizaci nebylo uspokojivé a obecně by se dalo říct, že velký vliv na zrychlení měla délka chromozómu.

Nakonec jsem se rozhodl jako benchmark zvolit problém *N dam* z řady důvodů uvedených výše. Ve výběru hrála roli i relativní znalost tohoto problému a použití genetických algoritmů.

Zakódování bylo zvoleno permutační. Binárního zakódování lze použít také, ale v tomto problému použití permutačního zakódování přináší veliké usnadnění v podobě vyřešení pohybu dam ve směru řádků. Což velmi příznivě ovlivňuje i počet platných hodnot chromozómů, kterých mohou nabývat. Podle [4] je počet možností $N!$. Délka chromozómu je zde rovna počtu dam. Pořadí genu nám určuje sloupec a pomocí hodnoty genu je určena poloha dámy v tomto sloupci. Vezmeme-li si obr. 5.1, jeho chromozóm by vypadal následovně: $(2, 4, 6, 1, 3, 5)$.

Velmi důležitou roli zde hraje fitness funkce. Jak je uvedeno v [19] pro rozhodnutí, zda se dvě dámy na šachovnici ohrožují, existuje analytické řešení. Vzhledem k větší složitosti fitness funkce však bylo zvoleno procházení po všech uhlopříčkách. Důvod je zřejmý. Mnohem složitější fitness funkce znamená větší část paralelizovatelného kódu v poměru k potřebné režii. Fitness funkce se pak spočítá dle vzorce 6.1. Kde f je výsledná hodnota fitness, l je počet dam (délka chromozómu) a c představuje počet kolizí mezi dámami. Ze vzorce plyne, že pro správné řešení tohoto problému (dámy se neohrožují) je hodnota fitness rovna $4l$.

$$f = 4l - c \tag{6.1}$$

Kapitola 7

Experimentální výsledky

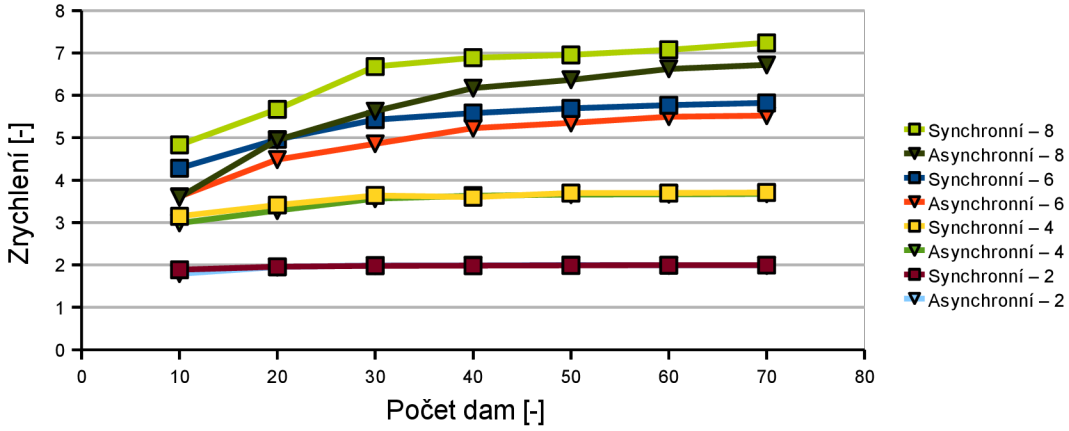
V této kapitole jsou uvedeny dosažené výsledky pro jednotlivé topologie a obě varianty (synchronní/asynchronní). Tyto testy byly provedeny (pokud není uvedeno jinak) na serveru Edesign2[1].

7.1 Zrychlení a efektivita

V této sekci jsou uvedeny dosažené zrychlení a efektivita, v závislosti na počtu dam. Jako ukončovací kritérium byl zvolen počet generací 100, typ použité obnovy populace steady state. Každý běh byl proveden 20krát.

7.1.1 Torus topologie

Výsledků bylo dosaženo při velikosti populace 30×30, přičemž bylo použito von Neumannovo okolí s velikostí jedna. U asynchronní verze byl použit fixed line sweep.



Obrázek 7.1: Zrychlení

Na obr. 7.1 je zobrazeno zrychlení při použití 2, 4, 6 a 8 jader. Můžeme vidět, že při použití více jader, má na výsledné zrychlení velký vliv počet dam (časová náročnost fitness funkce). Dále můžeme pozorovat menší dosažené zrychlení u asynchronní varianty,

[1]Supermicro X7DBR-I+, 2xQuad Core Intel Xeon 5355, 32 GB RAM, 150 GB HDD RAID-0

při použití více jader. Liší se tak od kruhové topologie, která je uvedena dále. Příčinou je počet hraničních jedinců, kterých je u torus topologie obecně větší počet, než u kruhové topologie.

Pokud budeme uvažovat pouze okolí uvedená v kapitole 4 a nebudeme uvažovat možnost překrytí hraničních jedinců, ať už z důvodu velkého okolí, nebo využití velikého počtu vláken, pro $t > 1$ je počet hraničních jedinců dán následujícími vzorci. Kde P je počet hraničních jedinců, o je velikost okolí, row je délka řádku a t značí počet použitých vláken. Velikostí okolí je myšlen počet sloupců ve směru nahoru/dolů u torus topologie a počet řádků doleva/doprava u kruhové topologie. Pro obr. 3.4(a), 3.4(b) a 3.3 je tedy velikost okolí jedna. Počet hraničních jedinců u kruhové, resp. torus topologie, je dán vzorcem 7.1, resp. 7.2.

$$P = 2 * o * (t + 1) \tag{7.1}$$

$$P = 2 * o * row * (t + 1) \tag{7.2}$$

Tabulka 7.1: Torus topologie — zrychlení v závislosti na počtu dam

	Počet dam	10	20	30	40	50	60	70
P. jader		Zrychlení [-]						
2	Synchronní	1,89	1,96	1,98	1,98	1,99	1,99	1,99
	Asynchronní	1,79	1,95	1,99	1,99	1,99	1,99	1,99
3	Synchronní	2,61	2,85	2,92	2,92	2,96	2,97	2,96
	Asynchronní	2,62	2,87	2,92	2,94	2,94	2,95	2,97
4	Synchronní	3,15	3,41	3,64	3,6	3,7	3,7	3,71
	Asynchronní	2,99	3,28	3,56	3,63	3,66	3,67	3,68
5	Synchronní	3,84	4,66	4,8	4,87	4,87	4,91	4,91
	Asynchronní	3,78	4,26	4,51	4,63	4,69	4,73	4,77
6	Synchronní	4,28	4,96	5,43	5,58	5,69	5,77	5,82
	Asynchronní	3,61	4,48	4,86	5,23	5,35	5,5	5,52
7	Synchronní	4,26	5,34	5,61	5,71	5,68	5,82	5,82
	Asynchronní	3,93	4,78	5,23	5,39	5,48	5,62	5,65
8	Synchronní	4,83	5,67	6,68	6,89	6,96	7,08	7,24
	Asynchronní	3,59	4,95	5,63	6,17	6,37	6,62	6,72

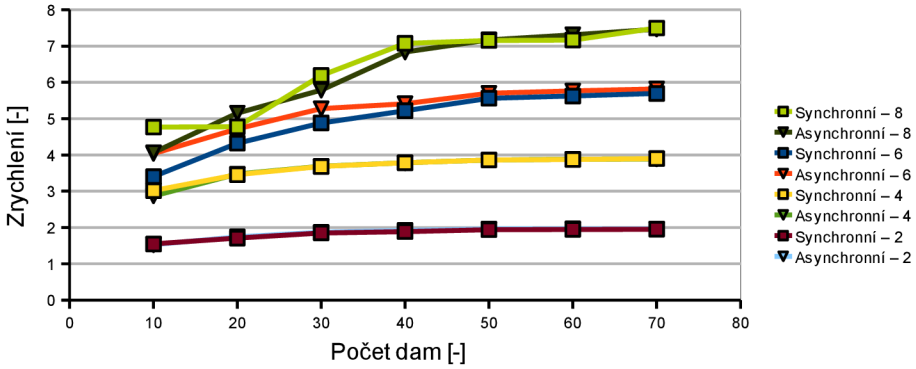
Tabulka 7.2: Torus topologie — efektivita v závislosti na počtu dam

	Počet dam	10	20	30	40	50	60	70
P. jader		Efektivita [%]						
2	Synchronní	94,50	98,00	99,00	99,00	99,50	99,50	99,50
	Asynchronní	89,50	97,50	99,50	99,50	99,50	99,50	99,50
3	Synchronní	87,00	95,00	97,33	97,33	98,67	99,00	98,67
	Asynchronní	87,33	95,67	97,33	98,00	98,00	98,33	99,00
4	Synchronní	78,75	85,25	91,00	90,00	92,50	92,50	92,75
	Asynchronní	74,75	82,00	89,00	90,75	91,50	91,75	92,00
5	Synchronní	76,80	93,20	96,00	97,40	97,40	98,20	98,20
	Asynchronní	75,60	85,20	90,20	92,60	93,80	94,60	95,40
6	Synchronní	71,33	82,67	90,50	93,00	94,83	96,17	97,00
	Asynchronní	60,17	74,67	81,00	87,17	89,17	91,67	92,00
7	Synchronní	60,86	76,29	80,14	81,57	81,14	83,14	83,14
	Asynchronní	56,14	68,29	74,71	77,00	78,29	80,29	80,71
8	Synchronní	60,38	70,88	83,50	86,13	87,00	88,50	90,50
	Asynchronní	44,88	61,88	70,38	77,13	79,63	82,75	84,00

V tabulce 7.1 jsou uvedeny všechny dosažené výsledky. V tabulce 7.2 jsou uvedeny výsledné efektivity pro daný počet dam. Efektivita je spočítána dle vzorce 4.2.

7.1.2 Kruhová topologie

Výsledků bylo dosaženo při velikosti populace 900 × 1, přičemž byla použita velikost okolí jedna. Na obr. 7.2 je zobrazeno zrychlení při použití 2, 4, 6 a 8 jader. Také zde má velký vliv na zrychlení počet dam.



Obrázek 7.2: Zrychlení

Tabulka 7.3: Kruhová topologie — zrychlení v závislosti na počtu dam

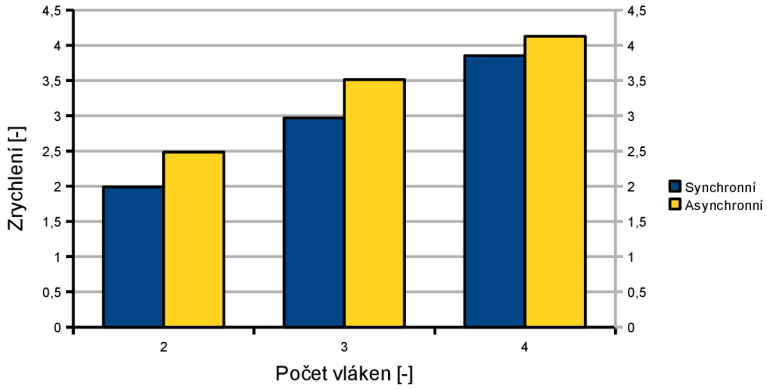
	Počet dam	10	20	30	40	50	60	70
P. jader	**Zrychlení [-]**							
2	Synchronní	1,55	1,71	1,85	1,89	1,94	1,95	1,95
	Asynchronní	1,53	1,74	1,87	1,92	1,96	1,96	1,96
3	Synchronní	2,08	2,5	2,73	2,81	2,88	2,89	2,91
	Asynchronní	2,12	2,57	2,8	2,81	2,94	2,91	2,96
4	Synchronní	3,02	3,46	3,68	3,78	3,86	3,88	3,9
	Asynchronní	2,87	3,47	3,69	3,78	3,86	3,88	3,89
5	Synchronní	3,18	4,14	4,48	4,68	4,78	4,81	4,83
	Asynchronní	3,24	4,11	4,4	4,65	4,79	4,82	4,87
6	Synchronní	3,4	4,32	4,88	5,22	5,56	5,62	5,69
	Asynchronní	4,04	4,72	5,27	5,41	5,7	5,76	5,82
7	Synchronní	3,63	4,87	5,6	6,18	6,45	6,42	6,68
	Asynchronní	3,74	4,61	5,78	6,31	6,3	6,39	6,8
8	Synchronní	4,77	4,78	6,19	7,08	7,15	7,17	7,5
	Asynchronní	4,06	5,15	5,78	6,84	7,17	7,31	7,47

Tabulka 7.4: Kruhová topologie — efektivita v závislosti na počtu dam

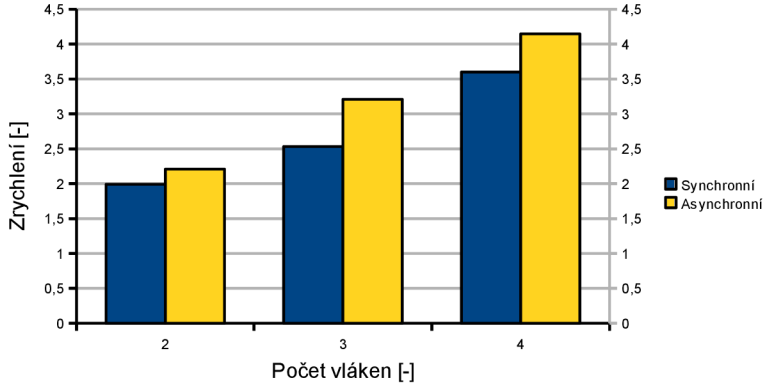
	Počet dam	10	20	30	40	50	60	70
P. jader	**Efektivita [%]**							
2	Synchronní	77,50	85,50	92,50	94,50	97,00	97,50	97,50
	Asynchronní	76,50	87,00	93,50	96,00	98,00	98,00	98,00
3	Synchronní	69,33	83,33	91,00	93,67	96,00	96,33	97,00
	Asynchronní	70,67	85,67	93,33	93,67	98,00	97,00	98,67
4	Synchronní	75,50	86,50	92,00	94,50	96,50	97,00	97,50
	Asynchronní	71,75	86,75	92,25	94,50	96,50	97,00	97,25
5	Synchronní	63,60	82,80	89,60	93,60	95,60	96,20	96,60
	Asynchronní	64,80	82,20	88,00	93,00	95,80	96,40	97,40
6	Synchronní	56,67	72,00	81,33	87,00	92,67	93,67	94,83
	Asynchronní	67,33	78,67	87,83	90,17	95,00	96,00	97,00
7	Synchronní	51,86	69,57	80,00	88,29	92,14	91,71	95,43
	Asynchronní	53,43	65,86	82,57	90,14	90,00	91,29	97,14
8	Synchronní	59,63	59,75	77,38	88,50	89,38	89,63	93,75
	Asynchronní	50,75	64,38	72,25	85,50	89,63	91,38	93,38

7.2 Nalezení požadovaného řešení

V této sekci jsou porovnány rychlosti při hledání požadovaného řešení. Bylo provedeno 50 běhů s populací 30×30 a 900×1 u obou verzí. Tyto testy byly provedeny na serveru Edesign1[2]. Počet dam byl zvolen 70. Na obr. 7.3 jsou zobrazeny dosažené výsledky u torus topologie. Zrychlení obou verzí je porovnáváno se synchronní verzí, u které byly dosaženy průměrně horší časy. Jak můžeme vidět, dostáváme superlineární zrychlení, což je způsobeno právě rychlejším nalezením hledaného řešení u asynchronní verze.



Obrázek 7.3: Zrychlení u torus topologie



Obrázek 7.4: Zrychlení u kruhové topologie

[2]Supermicro H8DME, 2xDual Core AMD Opteron 2220, 32 GB RAM, 150 GB HDD RAID-0

7.3 Vliv okolí na selekční tlak

Dalším provedeným testem je vliv typu okolí na seleční tlak. Tyto testy byly provedeny při počtu 50ti dam s využitím dvou vláken. Také zde byla použita steady state obnova a fixed line sweep u asynchronní verze.

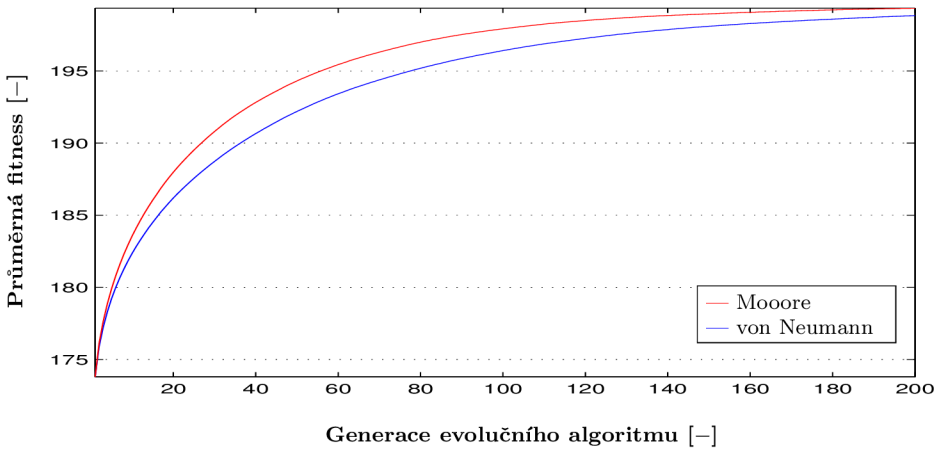
U synchronní verze je tisk statistik prováděn po každé generaci, zatímco u asynchronní verze, jak již bylo zmíněno, je řízen časem. Daný počet vláken byl zvolen kvůli malému počtu hraničních jedinců, aby se předešlo zbytečnému zpomalení, které v situaci, kdy tisk statistik závisí na čase, není žádoucí. Je však nutné uvést, že rozdíly v časech při výběru partnera ke křížení, způsobené odlišným typem okolí, byly zanedbány.

7.3.1 Torus topologie

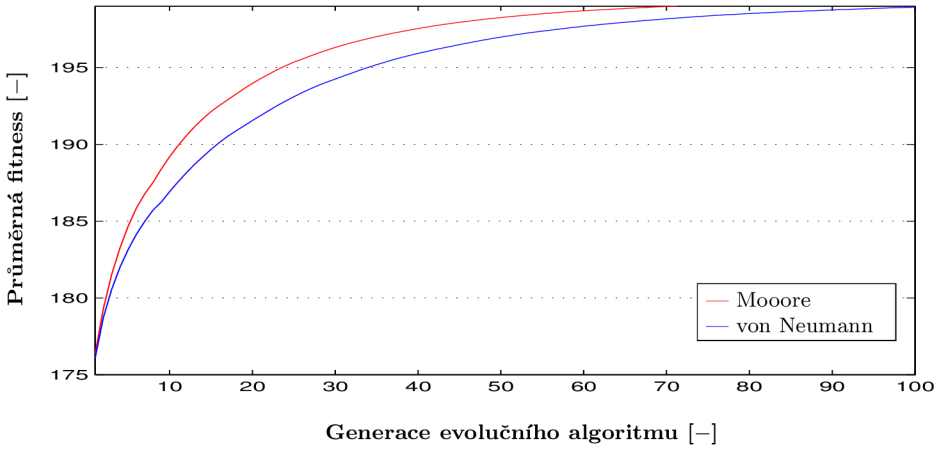
Zkoumány byly von Neumannovo a Moorovo okolí při různé velikosti.

Moorovo, von Neumannovo okolí

Tyto testy byly provedeny při velikosti populace 30×30. Na obrázcích 7.5 a 7.6 jsou zachyceny průměrné hodnoty průměrných fitness ze všech 100 běhů v průběhu generací. Můžeme zde pozorovat rychlejší konvergenci u Moorova okolí, jak v případě synchronním (obr. 7.5), tak i v asynchronním (obr. 7.6).



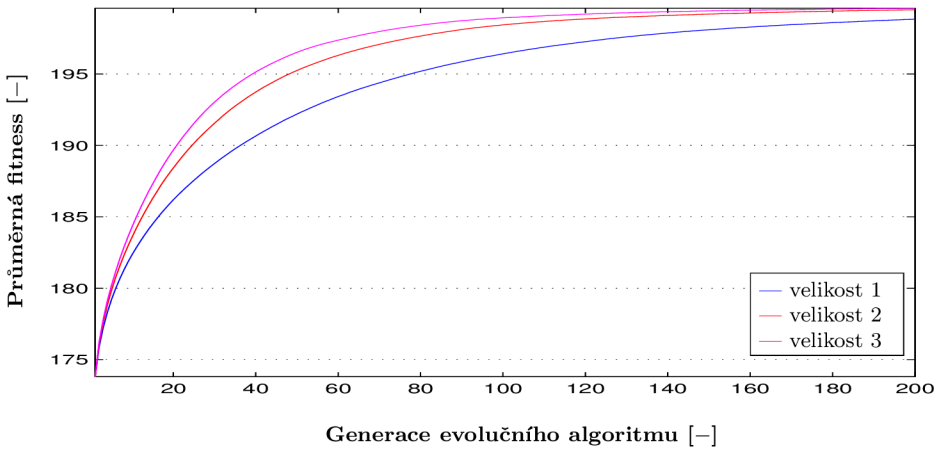
Obrázek 7.5: Moorovo, von Neumannovo okolí — synchronní verze



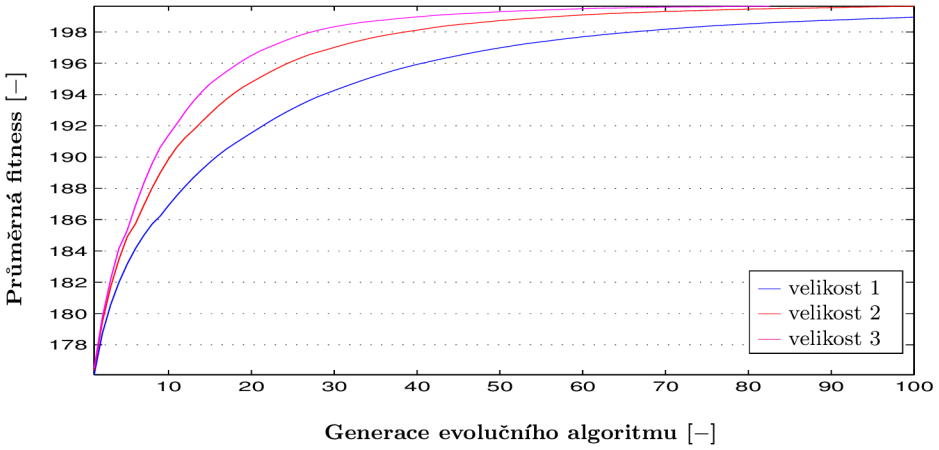
Obrázek 7.6: Moorovo, von Neumannovo okolí — asynchronní verze

Von Neumannovo okolí — různá velikost

Tyto testy byly provedeny při velikosti populace 30×30. Opět bylo provedeno 100 běhů a dosažené hodnoty byly zprůměrovány. Z obrázků 7.7 a 7.8 je vidět, že rychlost konvergence závisí na velikosti vybraného okolí.



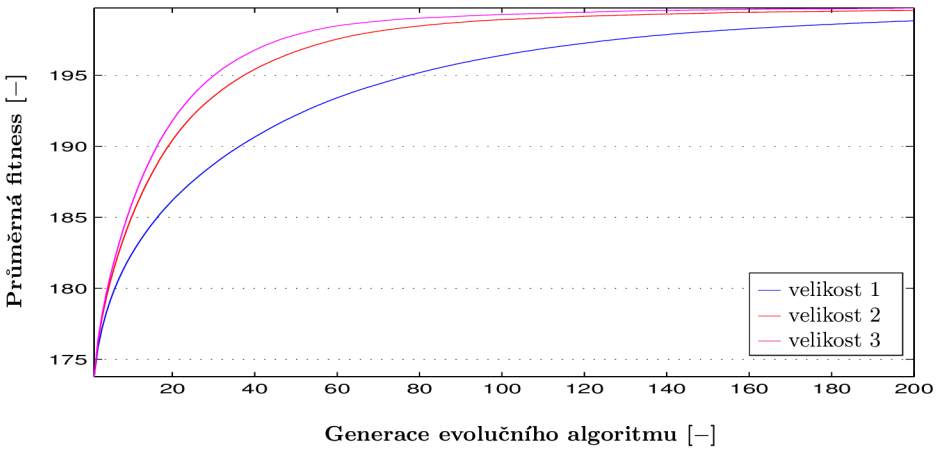
Obrázek 7.7: Různá velikost von Neumannova okolí — synchronní verze



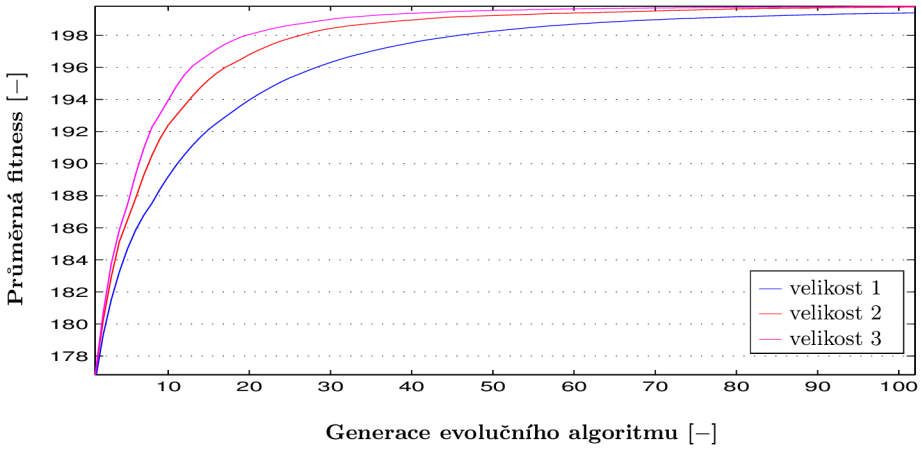
Obrázek 7.8: Různá velikost von Neumannova okolí — asynchronní verze

Moorovo okolí — různá velikost

Stejné nastavení, jako v předchozím případě. Z obrázků 7.9 a 7.10 je vidět, že rychlost konvergence závisí na velikosti vybraného okolí.



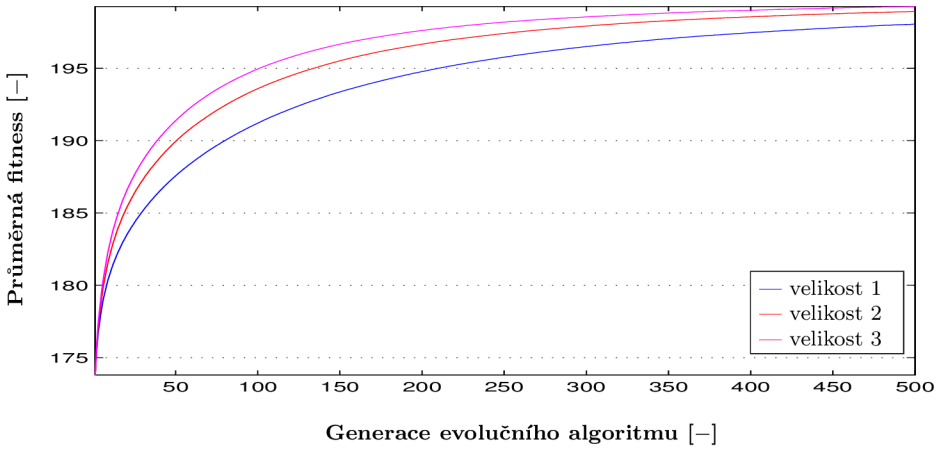
Obrázek 7.9: Různá velikost Moorova okolí — synchronní verze



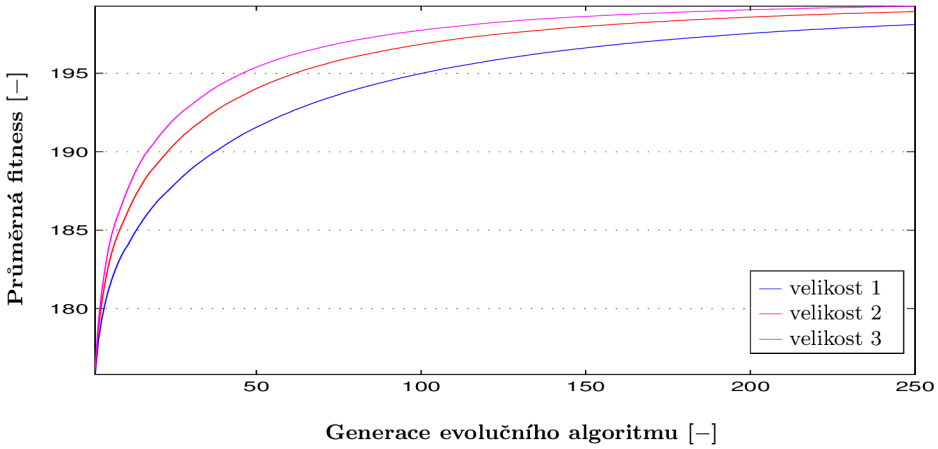
Obrázek 7.10: Různá velikost Moorova okolí — asynchronní verze

7.3.2 Kruhová topologie

Tyto testy byly provedeny při velikosti populace 900×1. Bylo provedeno 100 běhů, které byly zprůměrovány. Výsledky jsou uvedeny na obr. 7.11 a 7.12. Stejně jako u torus topologie, i zde při větší velikosti okolí, dochází k rychlejší konvergenci. Z obrázků 7.7 a 7.11 je však patrné, že při použití stejné velikosti okolí je u torus topologie výrazněji rychlejší konvergence, než u topologie kruhové.



Obrázek 7.11: Různá velikost okolí — synchronní verze



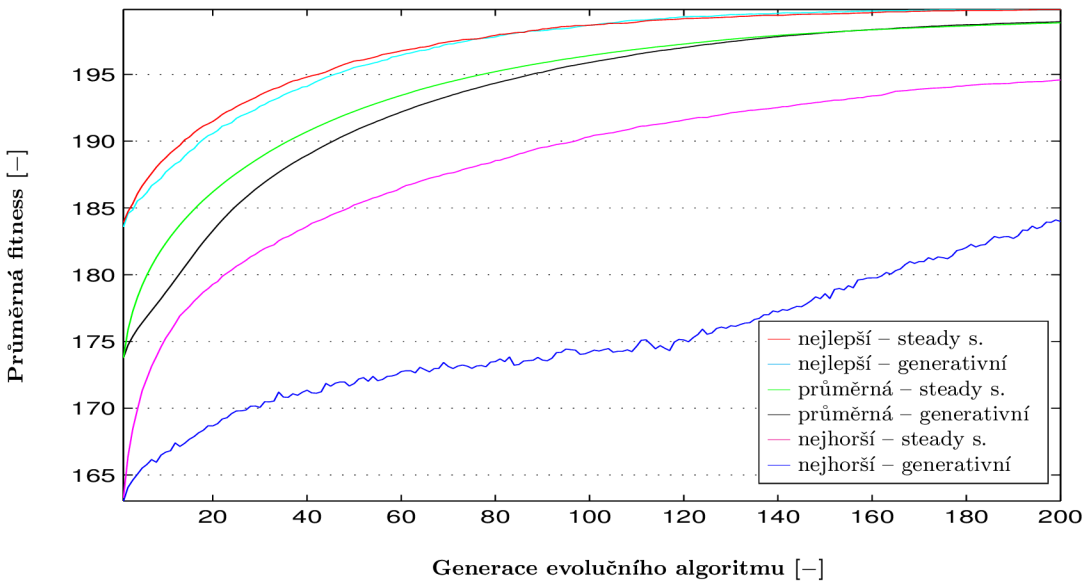
Obrázek 7.12: Různá velikost okolí — asynchronní verze

7.4 Vliv obnovy na diverzitu populace

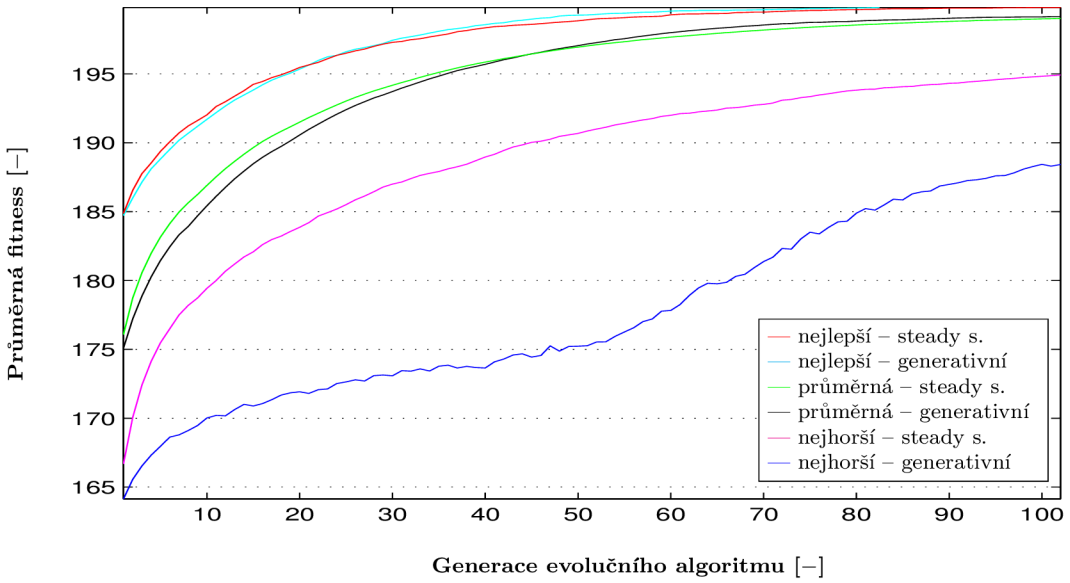
V této části byly otestovány obě varianty obnovy u synchronní i asynchronní verze s počtem vláken 2. Bylo provedeno 100 běhů při velikosti populace 30×30 a okolí typu von Neumann s velikostí 1. Počet dam byl zvolen 50.

- *Nejlepší* — fitness nejlepšího jedince (v rámci celé populace) průměrně na jeden běh.
- *Průměrná* — průměrná fitness (v rámci celé populace) průměrně na jeden běh.
- *Nejhorší* — finess nejhoršího jedince (v rámci celé populace) průměrně na jeden běh.

Jak můžeme vidět na obr. 7.13 a 7.14, rozdíl mezi obnovami je vidět především na nejhorší fitness, kdy generativní obnova dosahuje mnohem větší diverzity populace.



Obrázek 7.13: Diverzita populace — synchronní verze



Obrázek 7.14: Diverzita populace — asynchronní verze

Závěr

Cílem této práce byl návrh a implementace difuzního evolučního algoritmu, paralelizovaného pomocí OpenMP. Implementovány byly synchronní a asynchronní verze, umožňující různá nastavení.

Základní verze umožňuje nastavení typu topologie, velikosti populace a velikosti okolí. Výběrem okolí lze, jak bylo dokázáno v kapitole 7, úspěšně regulovat selekční tlak. Dalším užitečným rozšířením je možnost volby ze dvou typů obnovy ovlivňujících výrazně diverzitu populace. Nastavením těchto parametrů lze upravit vlastnosti algoritmu a umožnit tak jeho použití pro řešení různých typů úloh.

Při následném testování synchronní i asynchronní verze na problému N dam s důrazem na maximální zrychlení jsme dospěli k následujícím poznatkům: Největšího zrychlení bylo dosaženo při použití osmi jader. Zrychlení však neroste lineárně a pro osm jader je již výsledná efektivita menší, než efektivita dosažená pro dvě jádra. Z grafů uvedených v kapitole 7 je patrné, že při použití většího počtu jader efektivita výrazně závisí na složitosti fitness funkce. Při hledání optimálního řešení s ohledem na délku běhu evoluce, se jako efektivnější ukázala asynchronní verze, a to ve všech změřených případech.

Pokud z nějakého důvodu potřebujeme spustit evoluční běh vícekrát, příhodným řešením může být spuštění několika verzí současně, a využít tak pro každý běh pouze část z dostupných jader. Dosáhneme tak zrychlení i vysoké efektivity.

„Nejlepší EA jsou ty nejlépe vyladěné na konkrétní problém [3].“ Obecně tedy nelze říci, jaké nastavení je nejlepší. Vše závisí na řešeném problému, kterému musíme přizpůsobit nastavení parametrů. Máme-li například problémy s uváznutím v lokálním optimu, nebo dochází-li k předčasné konvergenci, může nám pomoci právě větší rozmanitost populace. V těchto případech je tedy vhodné využít generativní obnovu populace. Pokud se nám naopak zdá, že algoritmus konverguje příliš pomalu, můžeme zvětšit velikost okolí, a tím zvětšit i selekční tlak.

Jako rozšíření by bylo vhodné implementovat další operátory a topologie. Zajímavé by bylo nasazení této implementace při řešení praktického problému.

Literatura

[1] TOMASSINI, Marco. *Spatially structured evolutionary algorithms : artificial evolution in space and time.* Berlin : Springer, 2005. 192 s. ISBN 35-402-4193-0.

[2] JÜRGEN, Branke. *Evolutionary optimization in dynamic environments.* Vyd. 1. Boston : Kluwer Academic, 2002. 208 s. ISBN 07-923-7631-5.

[3] SEKANINA, Lukáš. *Biologií inspirované počítače : Evoluční design.* Brno, Přednáška 4. [s.l.] : [s.n.], 2011. 40 s.

[4] HYNEK, Josef. *Genetické algoritmy a genetické programování.* 1. vyd. Praha : Grada, 2008. 182 s. ISBN 978-802-4726-953.

[5] HOLLAND, John. *Adaptation in Natural and Artificial Systems.* Michigan : The University of Michigan Press, 1975. ISBN 0-262-58111-6.

[6] SCHWARZ, Josef; SEKANINA, Lukáš. *Aplikované evoluční algoritmy : EVO.* Brno, Studijní opora. [s.l.] : [s.n.], 2006. 101 s.

[7] SEKANINA, Lukáš, et al. *Evoluční hardware : od automatického generování patentovatelných invencí k sebemodifikujícím se strojům.* Vyd. 1. Praha : Academia, 2009. 321 s. ISBN 978-802-0017-291.

[8] MUNAKATA, Toshinori. *Fundamentals of the new artificial intelligence : neural, evolutionary, fuzzy and more.* 2nd ed. London : Springer, c2008. 255 s. ISBN 978-184-6288-388.

[9] SATOLA, Richard. *FIT VUT Brno: Genetické algoritmy*, 2004. 39 s.

[10] CHANDRA, Rohit. *Parallel programming in OpenMP.* San Francisco : Morgan Kaufmann, 2001. 230 s. ISBN 15-586-0671-8.

[11] *The OpenMP API specification for parallel programming* [online]. c2011, last updated on January 13, 2011 [cit. 2011-05-11]. About the OpenMP ARB and OpenMP.org. Dostupné z WWW: <http://openmp.org/wp/about-openmp/>.

[12] *GCC, the GNU Compiler Collection* [online]. 2787, last modified 2011-04-25 [cit. 2011-05-11]. Welcome to the home of GOMP. Dostupné z WWW: <http://gcc.gnu.org/projects/gomp/>.

[13] BLAISE, Barney. *Lawrence Livermore National Laboratory* [online]. last modified: 02/18/2011 [cit. 2011-05-11]. OpenMP. Dostupné z WWW: <https://computing.llnl.gov/tutorials/openMP/>.

[14] CHAPMAN, Barbara; JOST, Gabriele; VAN DER PAS, Ruud. *Using OpenMP : portable shared memory parallel programming.* Cambridge : MIT Press, c2008. 353 s. ISBN 978-026-2533-027.

[15] HWANG, Kai. *Advanced computer architecture.* Vyd. 1. USA : McGraw-Hill, 1993. 771 s. ISBN 00-703-1622-8.

[16] SOSIČ, Rok; GU, Jun. Efficient Local Search with Conflict Minimization : A Case Study of the n-Queens Problem. In *IEEE Trans. Knowl. Data Eng.* 1994. s. 661-668.

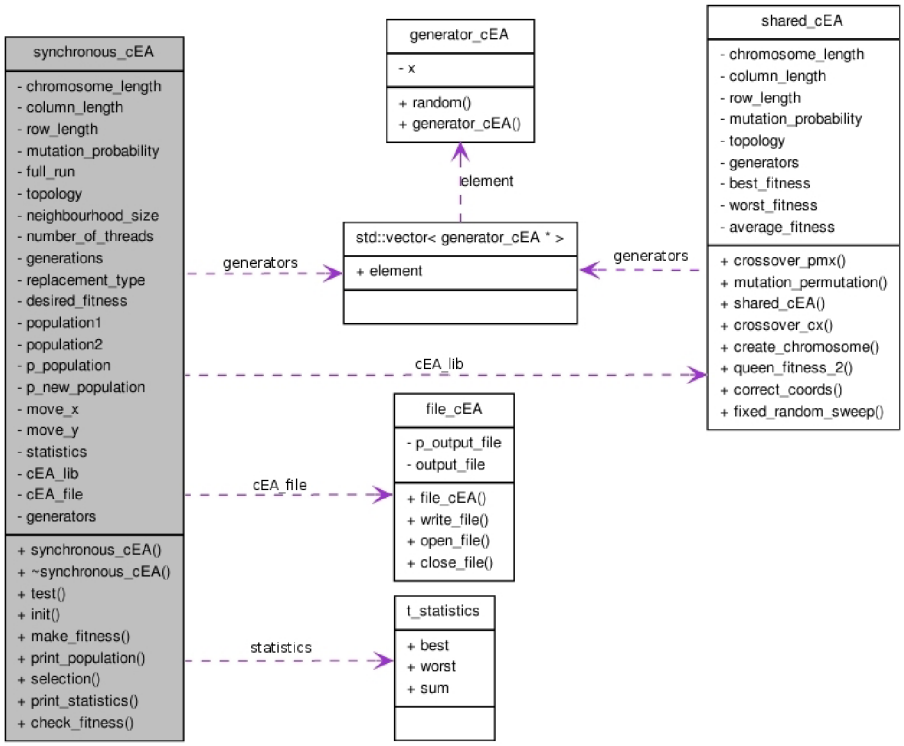
[17] PERINGER, Petr. *Modelování a simulace : IMS.* Brno, Studijní opora. [s.l.] : [s.n.], 2008. 85 s.

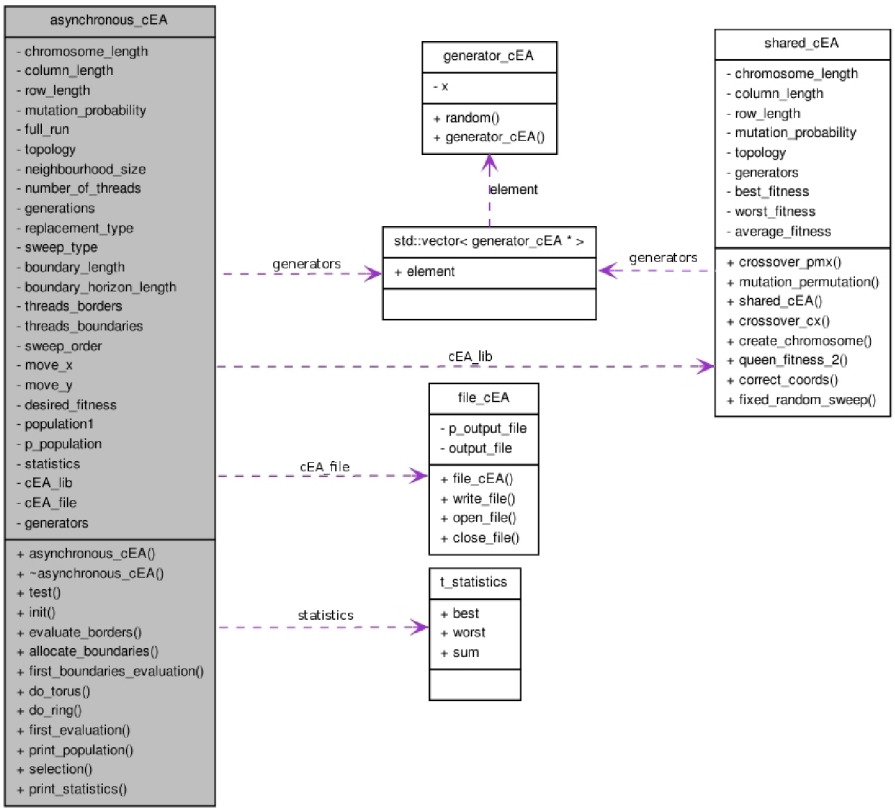
[18] The TRACER Project [online]. 2004, last updated 4/2/03 [cit. 2011-05-10]. The OneMax Problem. Dostupné z WWW: <http://tracer.lcc.uma.es/problems/onemax/onemax.html>.

[19] CRAWFORD, Kelly. Solving the n-Queens problem Using Genetic Algorithms. In *SAC '92 Proceedings of the 1992 ACM/SIGAPP symposium on Applied computing: technological challenges of the 1990's.* 1992. s. 1039-1047.

Příloha A

UML diagram




asynchronous_cEA
- chromosome_length
- column_length
- row_length
- mutation_probability
- full_run
- topology
- neighbourhood_size
- number_of_threads
- generations
- replacement_type
- sweep_type
- boundary_length
- boundary_horizon_length
- threads_borders
- threads_boundaries
- sweep_order
- move_x
- move_y
- desired_fitness
- population1
- p_population
- statistics
- cEA_lib
- cEA_file
- generators
+ asynchronous_cEA()
+ ~asynchronous_cEA()
+ test()
+ init()
+ evaluate_borders()
+ allocate_boundaries()
+ first_boundaries_evaluation()
+ do_torus()
+ do_ring()
+ first_evaluation()
+ print_population()
+ selection()
+ print_statistics()
generator_cEA
- x
+ random()
+ generator_cEA()
lelement
std::vector< generator_cEA * >
+ element
generators
generators
shared_cEA
- chromosome_length
- column_length
- row_length
- mutation_probability
- topology
- generators
- best_fitness
- worst_fitness
- average_fitness
+ crossover_pmx()
+ mutation_permutation()
+ shared_cEA()
+ crossover_cx()
+ create_chromosome()
+ queen_fitness_2()
+ correct_coords()
+ fixed_random_sweep()
cEA_lib
file_cEA
- p_output_file
- output_file
+ file_cEA()
+ write_file()
+ open_file()
+ close_file()
cEA_file
t_statistics
+ best
+ worst
+ sum
statistics

Příloha B

Obsah CD

- Zdrojové kódy obou verzí (C++)
- Dokumentace a návod k použití