VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

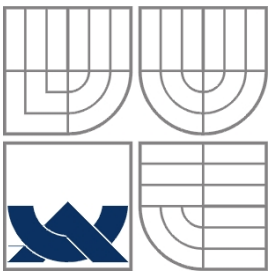
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

DIFÚZNE EVOLUČNÉ ALGORITMY

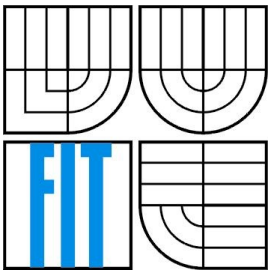
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE ISTVÁN MÉSZÁROS
AUTHOR

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

DIFŮZNÍ EVOLUČNÍ ALGORITMY

DIFFUSION EVOLUTIONARY ALGORITHMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

ISTVÁN MÉSZÁROS

VEDOUCÍ PRÁCE
SUPERVISOR

ING. JAROŠ JIŘÍ

BRNO 2008

Abstrakt

V dnešní době se objevují nové trendy v oblasti umělé inteligence. Metody známé jako evoluční algoritmy jsou jedny z nich. Tyto algoritmy nám umožňují optimalizovat a navrhovať systémy pomocí počítačů. Jedna z variant evolučních algoritmů je difůzní evoluční algoritmus. Tento typ algoritmu může probíhat paralelně a přináší přitom mnoho pozitivních vlastností. Otázkou je, při jakých podmínkách lze efektivně používat difůzní variantu evolučních algoritmů. Je možné jejich používání při plánování systémů nebo optimalizaci některých problémů? Proč jsou výhodnější než ostatní typy evolučních algoritmů?

Tato práce se snaží odpovědět na tyto otázky a podrobně vysvětlit fungování těchto algoritmů.

Abstract

There are new trends in artificial intelligence nowadays. Methods known as evolutionary algorithms are one of them. These algorithms allow us to design and optimize systems using computers. One of the variants of evolutionary algorithms is the diffusion evolutionary algorithm. This type of algorithms is able to run in parallel, and besides that it brings many positive features. The question is under what conditions the diffusion variant of evolutionary algorithms can effectively be used. Is it possible to use for planning systems and for problem optimization? Why are they more favorable than other types of evolutionary algorithms?

This work tries to answer these questions and explain the behavior of these algorithms.

Klíčová slova

Evoluční algoritmy, genetické algoritmy, genetické programování, difůzní evoluční algoritmy, paralelní programování, mostní konstrukce, návrch mostů, problém knapsack

Keywords

Evolutionary algorithms, genetic algorithms, genetic Programming, diffusional evolutionary algorithms, parallel programming, bridge constructions, bridge planning, knapsack problem

Citace

István Mészáros : Difúzne Evolučné Algoritmy, bakalářská práce, Brno, FIT VUT v Brně, 2010

Difúzne evolučné algoritmy

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením...
Další informace mi poskytli...
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
István Mészáros
19.5.2010

Poděkování

Především bych rád poděkoval Jiřímu Jarošovi, za jeho pomoc a užitečné rady při tvorbě bakalařské práce. Dále bych rád poděkoval a přátelům, kteří mě podporovali během vytvorení této práce.

Obsah

1 Úvod

V súčasnosti sa čoraz častejšie na riešenie zložitých (technických ale aj netechnických) problémov používajú tzv. "evolučné výpočtové techniky" alebo evolučné algoritmy (evolutionary algorithms). Sú to metódy resp. algoritmy, ktoré vo svojej podstate napodobňujú mechanizmy biologickej evolúcie. O evolučných výpočtoch sa prvýkrát zmienili v päťdesiatych rokoch minulého storočia. Od tej doby vzniklo množstvo publikácií týkajúce sa tejto témy.

Evolučné algoritmy patria do kategórie optimalizačných algoritmov, ich cieľom nie je nájsť presné riešenie na nejaký problém, ale skôr nájsť postačujúce riešení. Efektívne sa dajú využiť na aproximáciu riešenia NP (nondeterministic polynomlial time) problémov. Ďalšia využiteľnosť je tzv. evolučné plánovanie (Evolutionary design), čo sa využíva u vytvorenie komplexných systémov (viz. genetické programovanie)[1].

Existujú rôzne „podtechnológie“ evolučných výpočtov, ako napr. genetický algoritmus, genetické programovanie, evolučná stratégia, evolučné programovanie, hľadanie harmónie, gaussova adaptácia, optimalizácia mraveniska atď. Zaujímavú oblasť evolučných výpočtov tvoria difúzne evolučné algoritmy. Táto technológia je jedna z najmodernejších v rámci evolučných výpočtov. Tým, že populácia je rozdelená do priestoru, ich charakter sa podobá viac na skutočnú evolúciu. Ďalší nezanedbateľný príznak difúznych evolučných algoritmov je možnosť rozdelenia záťaže medzi procesmi. Riešenie problémov pomocou evolučných algoritmov je časovo náročné, z tohoto dôvodu je dôležité paralelizovanie týchto algoritmov. V dnešnej dobe sú dajú používať počítače, ktoré obsahujú viacero jadier, tým pádom je možné výrazne znížiť časovú náročnosť difúznych evolučných algoritmov[2].

Cieľom tejto práce bolo experimentálne nájdenie najvýhodnejšie parametre a dokázanie niektorých predpokladov difúznych evolučných algoritmov. Hlavnou témou je efektivita týchto algoritmov pri akých nastaveniach udávajú najlepšie výsledky. Porovnanie sekvenčnú s paralelnou variantou, pre aké problémy sú lepšie sekvenčné a pre aké problémy paralelné difúzne evolučné algoritmy. Ukázanie na konkrétnom príklade, že tieto typy algoritmov sú schopné vyriešiť aj pomerne zložité problémy, napr. NP-úplný problémy. Ďalšou časťou práce je preukázanie, že difúzne evolučné algoritmy sa dajú používať okrem optimalizačnej činnosti aj na navrhovanie a vytvorenie komplexných systémov ako napr. mostné konštrukcie (nad rámec vypracovaná práca).

Tento dokument v prvých kapitolách vysvetlí teóriu a základné myšlienky, ktoré sa skrývajú za evolučnými algoritmami a charakterizuje najpopulárnejšie metódy používané v dnešnej dobe. V štvrtej kapitole sú porovnané hlavné typy paralelných evolučných algoritmov, ďalej ich nastavenie tak, aby vykazovali najlepšie výsledky. V piatej kapitole je predstavená knihovňa OpenMP, ktorá je jedným z najlepších nástrojov na vytvorenie paralelných aplikácií. V šiestej kapitole je rozpísanie

metód na porovnanie paralelných a sekvenčných evolučných algoritmov. Ďalej je vysvetlený konkrétny cieľ práce a návrh riešenia. Siedma kapitola obsahuje vlastnú implementáciu daného problému a ako boli vyriešené hlavné problémy. V posledných kapitolách sú rozpísané výsledky testov, čo je možné z nich odvodiť a aké ďalšie vylepšenie by sa mohli uskutočniť. V prílohe sa nachádza stručný návod na použitie a obrázky výsledkov.

2 Evolučné algoritmy

V oblasti umelej inteligencie sú evolučné algoritmy podmnožinou evolučných výpočtov. Evolučné algoritmy sú metaheuristické optimalizačné algoritmy, ktorých základ tvorí model živej prírody. Slovo „metaheuristické“ je vytvorené z gréckej predpony „meta“ (mimo, vyšší stupeň) a zo slova „heuristika“ (spôsob riešenia). Toto slovné spojenie je možné pochopiť ako “optimalizácia na vyššej úrovni“. Evolučné algoritmy sú stochastické metódy na riešenie problémov, ktoré sú inšpirované prírodnou evolúciou. Používajú model základných mechanizmov evolúcie, ako reprodukcia, mutácia, úhyn a výber jedincov (selekcia) . V podstate sa jedná o simuláciu vývinu populácie v danom prostredí. V evolučných algoritmoch každý jedinec reprezentuje odlišné riešenie daného problému. Počas simulácie postupne vytvárame dokonalejších jedincov, čím sa uskutoční optimalizácia[1].

2.1 Biologické pozadie

V prírode je každá živá bytosť popísateľná s pomocou jeho génov. Tieto gény, čiže genotyp vytvorí každý viditeľný a neviditeľný charakter jedinca, známy ako fenotyp[1]. Rámci jedného druhu sa zdá, akoby všetky jedince boli rovnaké, ale v skutočnosti to tak nie je. Každý jedinec v rámci populácie sa mierne odlišuje od ostatných jedincov. Rozličnosť v druhu zabezpečuje prirodzenú flexibilitu voči neustálim zmenám prostredia. Niektoré jedince v rámci populácie sú vďaka rozmanitosti viac životaschopné ako ostatné. Tieto jedince majú väčšiu pravdepodobnosť na prežitie a reprodukciu, čo je základom prirodzeného výberu.

2.1.1 Mechanizmus dodržania flexibility jedného druhu

Ako už bolo spomenuté, každý jedinec sa mierne odlišuje od ostatných. Táto odlišnosť môže byť príčinou prežitia a reprodukcie. Reprodukcia je proces, v ktorom sa rodí nový jedinec, ktorý dedí vlastnosti od rodičov, tj. dedí ich genotyp. Je to spôsob na udržiavanie počtu organizmov v populácii. Vzhľadom na to, že sa množia len tie najlepšie jedince, do nasledujúcich generácií sa prenášajú len tie najlepšie vlastnosti. Tým pádom druh je v nekonečnej zmene, lebo musí odolať nekonečným výzvam stále sa meniaceho prostredia. Lenže reprodukčný proces nie je postačujúci faktor na prekonanie výzvy prostredia. Z faktu, že sa jedince len mierne odlišujú, nie je zabezpečená dostačujúca rozmanitosť. Tento nedostatok je kompenzovaný procesom mutácie. Priebeh mutácie náhodne mení genotyp. Aj keď zmeny spôsobené mutáciou sa málokedy končia pozitívnym výsledkom, mutácia je proces, ktorý je schopný spôsobiť radikálne zmeny na prekonanie problémov prostredia. Tieto tri pojmy: reprodukcia, mutácia a prirodzený výber zabezpečujú vývin druhu v prírode a inšpirujú evolučné algoritmy.

2.2 Základné pojmy evolučných algoritmov

2.2.1 Fitness, fitness funkcia, fitness aproximácia, povrch fitness

Pojem fitness je jeden z najzákladnejších pojmov evolučnej teórie, ktorý určuje úspech jedinca v populácii. Prvý krát ho použil Herbert Spencer v roku 1851 a neskôr ho Charles Darwin používal na vypracovanie svojej teórie[3].

Takmer vo všetkých typoch evolučných algoritmov sa vyskytuje pojem fitness hodnota, ktorá ukazuje vhodnosť jednotlivých chromozómov (jedincov). V podstate ide o numerické vyjadrenie kvality daného riešení. V evolučných algoritmoch iba jedince, ktoré majú vysokú fitness hodnotu, majú právo na reprodukciu. Hodnota fitnessu je zvyčajne určená s fitness funkciou.

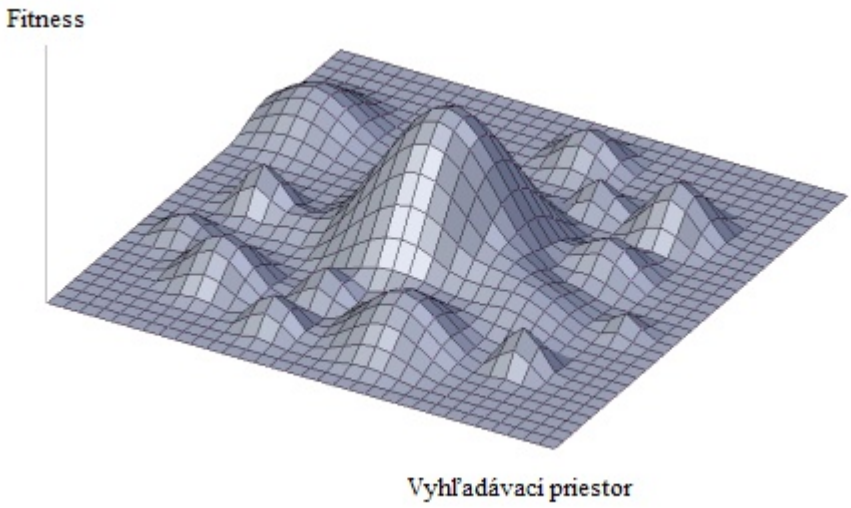
Fitness funkcia je jedna určitá cieľová funkcia, ktorá slúži na ohodnotenie optimalizácie. Implementácia tejto funkcie je väčšinou náročná úloha v oblasti evolučných algoritmov. Fitness funkcia musí striktne korelovať s cieľom algoritmu. Vzhľadom k tomu, že počas simulácie je táto funkcia najčastejšie vykonávaná, rýchlosť výpočtu funkcie je veľmi dôležitým aspektom. Hlavná nevýhoda fitness funkcie sa prejaví, keď chceme algoritmus aplikovať na netriviálny problém. V tomto prípade môže byť komplikovaná a ťažko vypočítateľná .

Na obídenie tohoto problému sa v praxi používa fitness aproximácia. Kvôli tomu aby sme dosiahli čo najefektívnejší algoritmus, je dôležité aby sme používali informácie získané počas doby simulácie. Z tohoto faktu vyplýva, že je nutné konštruovať model fitness funkcie. Poznáme viac spôsobov na vytvorenie modelu vyššie uvedenému pojmu. Vo všeobecnosti ide o interpoláciu fitness hodnôt, ale sú používané aj napr. umelé neuronové siete, regresívne metódy, atď. V praxi je fitness aproximácie aplikovaná paralelne s fitness funkciou. V súčasnej dobe je populárne riešenie Adaptívna Fuzzy Fitness Granulalizácia (AFFG) [4]. Využitie fitness aproximácie je prospešne v troch prípadoch:

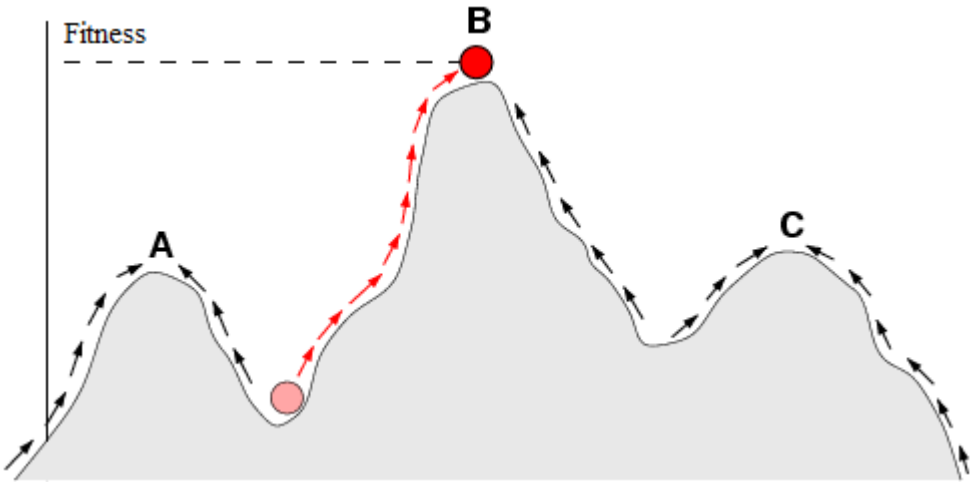
- Výpočet fitness funkcie jedného riešenia je časovo náročný.
- Chýba precízny model na výpočet fitness funkcie.
- Fitness funkcia je neistá alebo hlučná (zašumená).

Pojem “povrch fitness” (fitness surface/landscape) podstatne uľahčuje porozumenie princípu selekcie v priebehu evolúcie. Základná idea povrchu fitness spočíva v grafickej reprezentácií fitness hodnôt, podľa zloženie genotypu organizmov v danej populácií (viz. obr. 2.1). Populáciu potom môžeme znázorniť ako „oblak“ bodov na povrchu, kde každý bod je priradený k jedinci z populácie.

Priebeh mnohých generácií prirodzene spôsobuje pohyb populácii na povrchu do oblasti s väčšou hodnotou fitness (viz. obr. 2.2). Ak je dominantná selekcia, potom sa tento pohyb uskutočňuje v smere gradientu, v ktorom fitness najrýchlejšie rastie. V opačnom prípade, ak je v evolúcii dominantná náhodnosť (napr. mutácie), pohyb bodov na povrchu fitness má stochastický charakter[1].



Obraz 2.1: Povrch fitness - Výška ukazuje hodnotu fitness a na ostatných osách sú umiestnené gény (parametri riešení).



Obraz 2.2: Povrch fitness - Výška ukazuje hodnotu fitness, horizontálna osa zobrazuje jeden gén chromozómu a šípky pohyb jedincov k vrcholu povrchu fitnessu.

2.2.2 Selekcia

Selekcia je proces náhodného výberu jedincov na reprodukciu. V realite tie jedince, ktoré majú väčšiu fitness hodnotu, majú väčšiu šancu na výber do reprodukčného procesu. Na selekciu majú okrem

fitness hodnoty vplyv aj ďalšie udalosti ako napr. náhodné úmrtia atď, ktoré blokujú reprodukčnú činnosť jedinca[2].

2.2.3 Reprodukcia

Selekciu nasleduje reprodukcia. Proces reprodukcie slúži na vytvorenie potomkov a tým pádom aj na skúšanie nových riešení problémov. V tomto kroku sa genotyp dvoch rodičov zjednotí a vytvorí sa genotyp potomka. Proces spájania génov rodičov je stochastický[2].

2.2.4 Mutácia

Proces mutácie znamená náhodné malé zmeny v genotypu, ktoré sú spôsobené účinkami vonkajšieho prostredia. Vo väčšine prípadov mutácia spôsobí negatívne zmeny u jedinca, ale je schopný prinášať aj prospešné javy. Toto je proces, ktorý môže priniesť úplne nový smer vo vývine populácie. Keby do evolučných algoritmov nebola aplikovaná mutácia, chromozómy by konvergovali k jednému bodu riešenia, kde každý jedinec mal rovnaký genotyp [2].

2.3 Základná štruktúra evolučných algoritmov

Takmer všetky typy evolučných algoritmov sú rovnaké na úrovni základných krokov. Algoritmy sa dajú popísať nasledujúcim spôsobom:

```
Initialization();
Generations=0;
While ( termination_condition ) {
      Evaluate_fitness();
      Selection();
      Reproduction();
      Mutation();
      Replacement();
      ++Generations;
}
```

Algoritmus 2.1 : Štruktúra evolučného algoritmu[2]

1. **Initialization**: Vytvorí sa a inicializuje populácia.
2. **Evaluate_fitness**: Prevedie sa výpočet fitness hodnoty každého jedinca.
3. **Selection**: Uskutoční sa náhodný proces výberu jedincov z populácie.
4. **Reproduction**: Dôjde ku kombinácii genotypu oboch rodičov a k vzniku dvoch potomkov.

5. **Mutation**: Prevedie sa mutácia na novovytvorených jedincoch a náhodne sa zmenia ich genotypy.
6. **Replacement** : Potomky nahradzujú ich rodičov, tým pádom sú zahrnuté do populácie.

Mala by sa brať do úvahy aj podmienka ukončenia (Termination condition), ktorá je nenahraditeľná k ukončeniu aplikácie. Táto podmienka zvyčajne závisí na problematike, ale najbežnejšie príklady sú nasledovné:

1. Fitness hodnota najlepšieho jedinca je väčšia ako požadovaná fitness hodnota problematiky.
2. Počet generácií prekročil zadaný limit.
3. Hodnota fitness sa dostala do lokálneho alebo globálneho maxima, tj. ďalej sa už nezvyšuje. Toto maximum sa zistí tak, že fitness hodnota i napriek novo vytvoreným generáciám stagnuje po určitý počet generácií.
4. Vyčerpali sa prostriedky na beh programu.

Zvyčajne sa používajú kombinácie týchto podmienok[5].

3 Typy evolučných algoritmov

Existuje viac variant evolučných algoritmov. Každý typ slúži na riešenie inej triedy problémov. Hlavný rozdiel medzi variantami sa skrýva v reprezentácii genotypu jedincov. Rozdiel v reprezentácii genotypu prináša aj rozdiel ostatných charakterov evolučných algoritmov, líšia sa aj základné procesy ako selekcia, reprodukcia a mutácia. Štyri hlavné druhy evolučných algoritmov sú [5]:

- **Genetické algoritmy**
- **Genetické programovanie**
- **Evolučná stratégia**
- **Evolučné programovanie**

3.1 Genetické algoritmy

Genetické algoritmy sú jedny z najpopulárnejších evolučných algoritmov. V praxi sa tieto algoritmy najčastejšie používajú na nájdenie riešení na optimalizačné problémy.

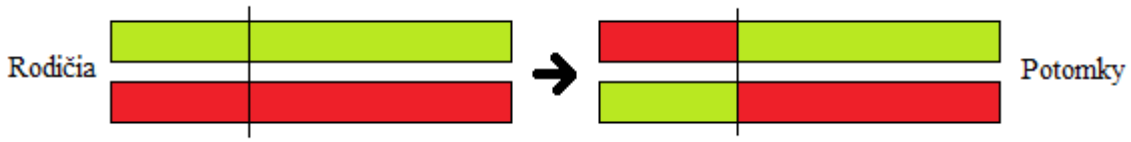
3.1.1 Reprezentácia genotypu

Chromozóm je najčastejšie reprezentovaný binárnym reťazcom (bitovým polom). Pozitívum tejto metódy je v jednoduchosti implementácie reprodukcie a mutácie jedinca, ale zároveň z dôvodu binárnej reprezentácie, výpočet fitness funkcie je komplikovanejší . Dajú sa použiť bitové reťazce konštantnej i premennej dĺžky. Bitové reťazce konštantnej dĺžky sa používajú kvôli jednoduchosti, zatiaľ čo reťazce premennej dĺžky poskytujú viac možností na riešenie problémov (napr. Messy genetické algoritmy)[1].

3.1.2 Kríženie chromozómov

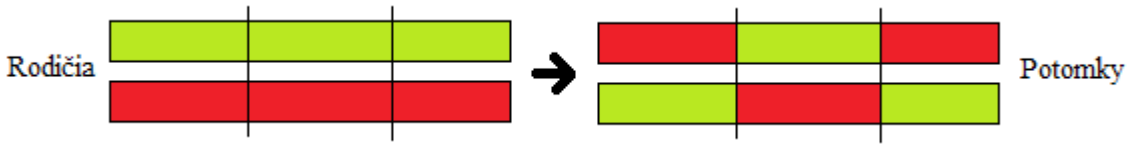
U genetických algoritmoch sa používajú viaceré typy kríženia chromozómov, ktoré sú:

- **Jednobodové kríženie:** Používa sa u chromozómov s konštantnou dľžkou. Náhodne je vygenerovaný bod kríženia, ktorý určí, kde majú byť chromozómy odseknuté. Bod kríženia je totožný u oboch rodičov. Nové jedince dostanú časť chromozómu každého z rodičov.



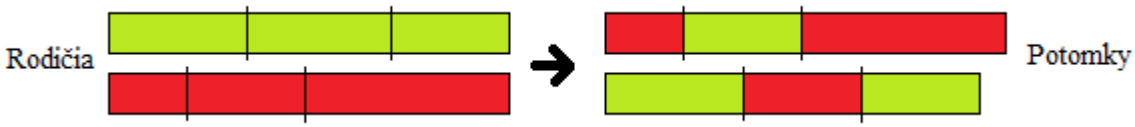
Obraz 3.1: Jedno bodové kríženie.

- **Dvojbodové kríženie:** Používa sa na konštantne veľké chromozómy. Náhodne sú vygenerované dva body kríženia. Tieto body určujú, kde dôjde ku kríženiu chromozómov. Body kríženia sú tiež totožné u oboch rodičov. Nové jedince dostanú jednu časť chromozómu od prvého a dve časti chromozómu od druhého rodiča.



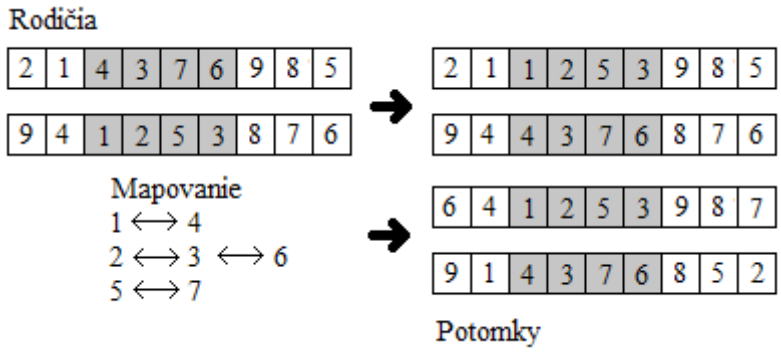
Obraz 3.2: Dvojbodové kríženie.

- "**Cut and splice" (seknúť a spojiť):** Dá sa použiť na kríženie chromozómov premennej dĺžky(tzv. Messy chromozomy) [1]. V tomto prípade sú body kríženia u oboch rodičov rôzne a tým pádom dĺžka chromozómov potomkov budú tiež rôzna.



Obraz 3.3: "Cut and splice" (seknúť a spojiť)[6]

- **Partially mapped crossover (čiastočne mapované kríženie):** Popis algoritmu :
  1. Výber rovnako veľkého podreťazca z oboch chromozómov.
  2. Výmena podreťazca a vytvorenie potomkov.
  3. Vyhľadanie mapovacích relácií podľa dvoch podreťazcov.
  4. Aplikovanie mapovacích relácií na ostatné časti chromozómov rodičov a prepísanie do chromozóm potomkov.



Obraz 3.4: Partially mapped crossover (čiastočne mapované kríženie)

Algoritmus je najpoužívanejší, keď hodnoty v reťazce (jednotlivé gény) sa vyskytujú v chromozóme len raz . Jedná sa o tzv. permutačné zakódovanie vhodné napr. na riešenie problému obchodného cestujúceho [7].

Je množné ešte používať mnoho ďalších metód na kríženie chromozómov ako sú uniformné kríženie, triedené kríženie (OX) a ďalšie[8].

3.1.3 Mutácia

Mutácia v genetických algoritmoch sa príliš ľahko implementuje. Je to vďaka tomu, že chromozóm je reprezentovaný binárnym reťazcom, pričom k zmene chromozómu dochádza jednoduchou zámenou hodnoty bytov. Táto metóda sa používa vo väčšine prípadov genetických algoritmov[1].

3.1.4 Selekcia

Najpopulárnejšia metóda na selekciu u genetických algoritmov je takzvaný „Roulette Wheel Selection“. Táto metóda náhodne vyberie jedincov z populácie na reprodukciu. Pravdepodobnosť, že jedinec vstúpi do procesu reprodukcie s iným jedincom, je priamo úmerná jeho fitness hodnote. Konfigurácia *Roulette Wheel Selection* je obtiažna, pretože ak príliš uprednostníme jedince s veľkou fitness hodnotou, tak populácia príliš rýchlo konverguje k jednému lokálnemu maximu čím môže dôjsť k ignorovaniu lepších riešení. V opačnom prípade, pokiaľ hodnota fitnessu nie je dôležitá, tak populácia konverguje pomaly. I keď algoritmus testuje veľa riešení, dopúšťa sa do viacerých zbytočných výpočtov[1].

3.2 Ostatné typy evolučných algoritmov

Ďalšie známe evolučné algoritmy sú genetické programovanie, evolučné programovanie a evolučná stratégia. Pre tieto algoritmy je charakteristické, že vo všeobecnosti sú schopné efektívne riešiť len špecifické druhy problémov.

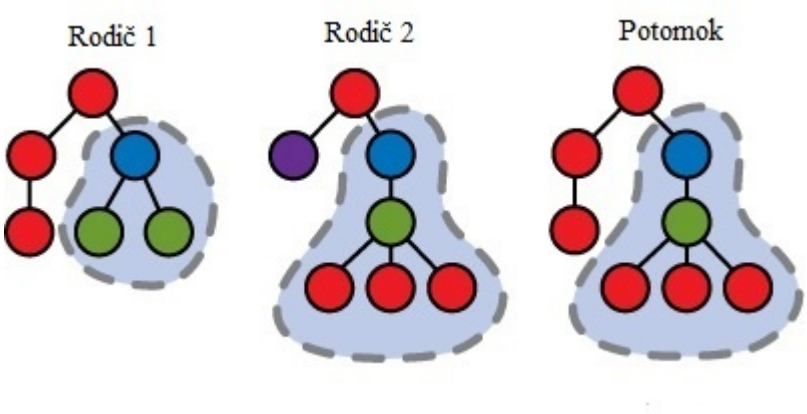
Veľmi významnou metódou je evolučná stratégia, ktorá na rozdiel od genetických algoritmov genotyp neukladá vo formáte bitového pola, ale vo formáte vektorov reálnych čísiel. Kým genetické algoritmy konvertujú svoje bitové pole na sémanticky „vyššiu“ hodnotu len v prípade nutnosti (napr. pri výpočtu fitness hodnoty), evolučná stratégia vždy pracuje s reálnymi hodnotami a vektormi.

Táto metóda vyžaduje celkom iný prístup z hľadiska reprodukčného a mutačného konania. Pokiaľ genetické algoritmy krížia bitové pole, algoritmus evolučnej stratégie počíta priemery génov oboch rodičov, pričom tieto hodnoty prináležia génom novovzniknutého potomka. Pre proces mutácie sa u evolučnej stratégie generuje náhodná veličina, ktorá je pripočítaná alebo odpočítaná od pôvodnej

hodnoty génu. Vo väčšine prípadov má náhodná veličina normálne (Gaussové) rozdelenie, ktoré sa na prírodnú mutáciu podobá najviac [5].

Veľmi zaujímavou oblasťou evolučných algoritmov je genetické programovanie, ktoré je schopné generovať počítačové programy, navrhovať matematické funkcie a riešiť problémy, ktoré potrebujú konštruktívne riešenie. V porovnaní s genetickými algoritmami je najväčší rozdiel opäť v reprezentácií chromozómu. v tejto metóde sú gény zabudované do komplexnej štruktúry. Ako príklad je možno uviesť stromovú štruktúru, ktorá je vhodná na reprezentáciu napr. bezkontextových jazykov. Iný typ dátovej štruktúry môže byť aj obecný graf, tento typ sa používa napr. na vytvorenie konečných automatov [9], [5].

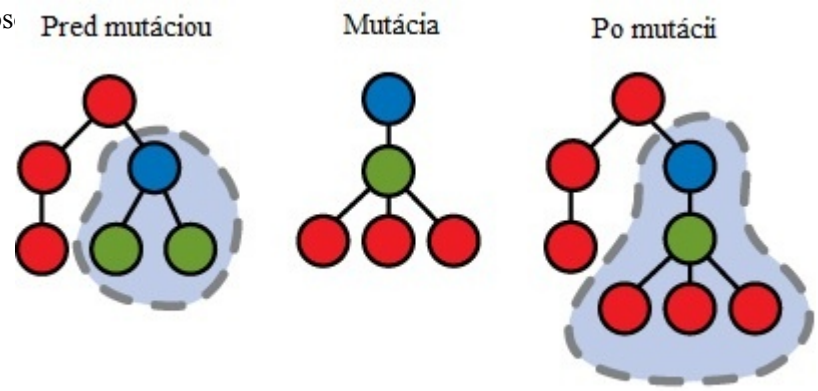
Dôsledkom komplexnej dátovej štruktúry je obtiažnosť reprodukcie a mutácie. U stromovej dátovej štruktúry funguje kríženie dvoch chromozómov nasledovne: Ako základ chromozómu potomka sa vezme graf jedného rodiča a od druhého rodiča sa vyberie časť grafu (podstrom), ktorá sa vloží na miesto jedného uzla grafu potomka. Tento proces zobrazuje obrázok 3.5 .



Obraz 3.5: Spôsob kríženia u genetických programovaní. Kríženie podľa stromovej dátovej štruktúry u genetického programovania

Mutácia funguje podobne ako kríženie. V prvom kroku sa vygeneruje náhodný podstrom, ktorý je v ďalšom kroku vložený na miesto jedného uzlu v grafu potomka.

Obraz 3.6: Spôs



4 Paralelné evolučné algoritmy

Evolučné algoritmy sú veľmi rýchlo sa rozvíjajúcim odborom informačných technológií. Výhody, okrem relatívne jednoduchej implementácie sú v riešení problémov, ktoré sú veľmi obtiažne pre ľudskú prácu. Napriek jeho dobrých charakterov existujú aj nevýhody, ako napr. časová náročnosť simulácie. Existujú programy, ktoré môžu behať niekoľko dní alebo týždňov, kým sú schopné nájsť prijateľné riešenie pre danú úlohu. Tento fakt bol základnou motiváciou pre vývin paralelných evolučných algoritmov.

Anglický pojem „multiprocessing“ alebo „parallel processing“ znamená používanie dvoch a viac procesorov v jednom počítačovom systéme. Existuje mnoho variant tejto technológie. Viacprocesorové systémy môžeme zaradiť do dvoch skupín. Tieto skupiny sú systémy so zdieľanou a systémy s distribuovanou pamäťou [10].

U viacprocesorových architektúrach so zdieľanou pamäťou má každý procesor rovnaké právo a rýchlosť prístupu k dátam. Tieto architektúry sa nazývajú UMA (Uniform Memory Access) architektúry. Napriek tomu viacprocesorové architektúry s distribuovanou pamäťou majú fyzicky rozdelené pamäťový priestor. Z tohoto vyplýva, že nemajú rovnaké právo a rýchlosť prístupu ku každej jednotke pamäti. Táto architektúra sa nazýva NUMA (Non – Uniform Memory Access) architektúra [11]. Tieto dva typy počítačových systémov sa využívajú u paralelných evolučných algoritmoch. Existujú dva hlavné typy paralelných algoritmov:

- Distribuované (ostrovský model)
- Celulárne (bunkový model)

4.1 Distribuované evolučné algoritmy

Distribuované evolučné algoritmy sú výhodné z dvoch dôvodov:

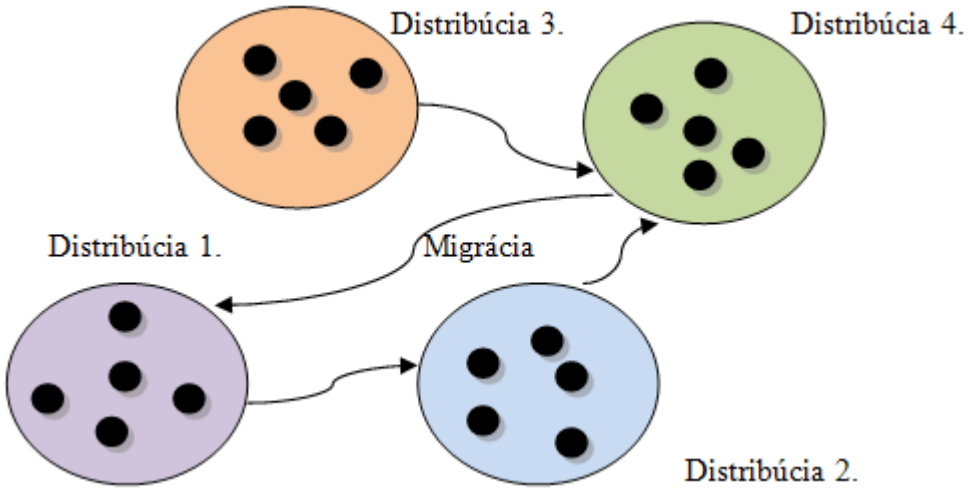
- Viac procesorov zvyšuje rýchlosť simulácie.
- Umožňujú simulovať dočasnú izoláciu populácií, tj. simulovať mikropopulácie tzv. *démy*.

Vo väčšine prípadov sú implementované na architektúre typu NUMA. Každý procesor simuluje jednu mikropopuláciu z celkovej populácie. Tento prístup má zmysel keď výmena genetickej informácie prebehne aj medzi jedincami rôznych mikropopulácií.

Tým, že samostatný procesor ako aj pamäť sú k dispozícií pre všetky mikropopulácie, ktoré spolu vzájomne komunikujú, je výmena genetickej informácie zdĺhavá a uskutočnená len podmienene. Komunikácia medzi mikropopuláciami je prevedená napr. po každej generácie alebo po danom perióde. Výmenu genetickej informácie je možné implementovať tak, že niektoré jedince sú vybrané do migrácie, tj. jedince z jednej mikropopulácie sú vymenené za jedince z druhej

mikropopulácie. Druhý spôsob výmeny genetickej informácie je vyberanie jedincov z dvoch distribúcii a potomkovia vznikajú ich reprodukciou. .

Distribučné evolučné algoritmy podľa štatistiky majú lepšie riešenia na danú problematiku. Každá distribúcia obsahuje jeden spôsob riešenia, tj. každá mikropopulácia konverguje k inému lokálnemu maximu a tým pádom je väčšia šanca pre nájdenie globálneho maxima. Distribuované evolučné algoritmy majú tzv. ostrovskú topológiu, ako to znázorňuje obrázok 4.1. [10].

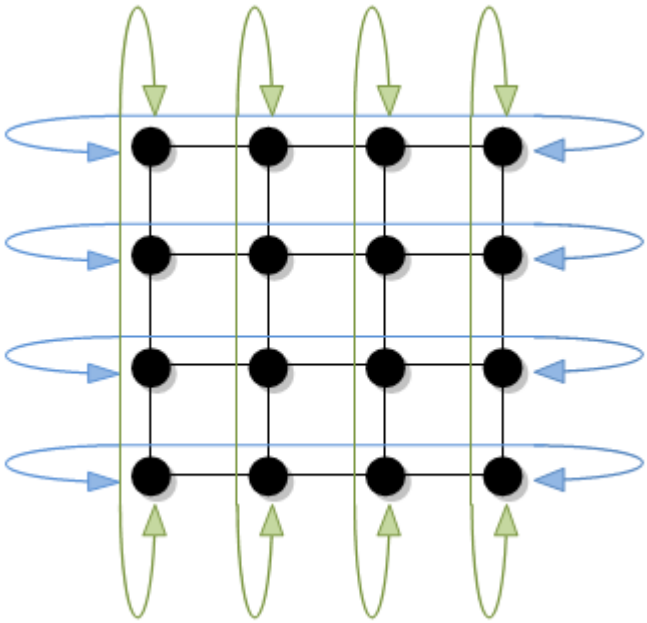


Obraz 4.1: Distribuované evolučné algoritmy (Island topology), obrázok ukazuje distribúcie a výmenu genetickej informácie medzi nimi.

S touto topológiou je možné charakterizovať situácia, keď jednotlivé distribúcie sú spojené pomocou internetu. Výmena genetickej informácie sa uskutočňuje pomocou komunikácie cez internet.

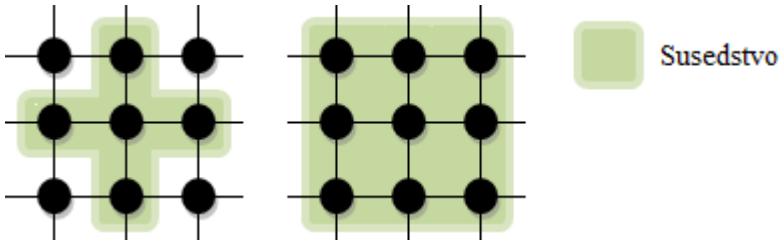
4.2 Celulárne evolučné algoritmy

Oproti distribučným algoritmom celulárne evolučné algoritmy nerozdeľujú populáciu do viacerých mikropopulácií. Charakter celulárnych evolučných algoritmov sa podobá skôr na charakter celulárnych automatov. Tieto evolučné algoritmy majú tzv. priestorové rozdelenie populácie (Spatial distribution). Najdôležitejším charakterom celulárnych evolučných algoritmov je priradená pozícia každému jedincovi. Táto pozícia u jedinca je uchovávaná ako vektor. Tento vektor môže byť dva alebo viac rozmeroví, ale najčastejšie sa jedná o dvoj alebo troj rozmerný priestor, v ktorom je rozložená populácia. Takto rozložená celková populácia sa podobá na šachovnicu s tým rozdielom, že dva krajné body ležiace v jednej rovine sú tiež navzájom susedné. (Túto vlastnosť vystihuje obrázok 4.2) Topológia tohoto typu sa nazýva „torus“ [5], [12].



Obraz 4.2: Celulárny evolučné algoritmus (difúzny) - Torus topológia.

V praxi sa používajú aj šesťuholníkové a iné topológie. Hlavným cieľom tejto štruktúry je, aby kríženie jedincov bolo umožnené len priestorovo sebe blízkym jedincom. Aj pri selekcie sú dané tzv. tvary, podľa ktorých sa určuje susedstvo jedincov(viz obr 4.3.). Najčastejšie sú používané tvary so štyrmi alebo ôsmymi vetvami. Podobné sa používajú aj v troj rozmerovom priestore. Kríženie medzi jedincami sa uskutoční v rámci tohoto tvaru. Vyberie sa najlepší jedinec z tvaru a ten sa kríži s centrálnym jedincom. Ak potomok po krížení má väčší fitness nahradí centrálneho jedinca.



Obraz 4.3: Formy selekcie v celulárnych evolučných algoritmov[10].

Existujú aj rozsiahlejšie tvary susedstva. Môžeme tvrdiť, že čím viac susedov má jeden jedinec, tým rýchlejšie konverguje populácia k jednému riešeniu. Tvary susedstva nám poskytujú možnosť nastavenia rýchlosti konvergencie.

Výmena potomka za rodiča je implementovatelná dvoma spôsobmi :

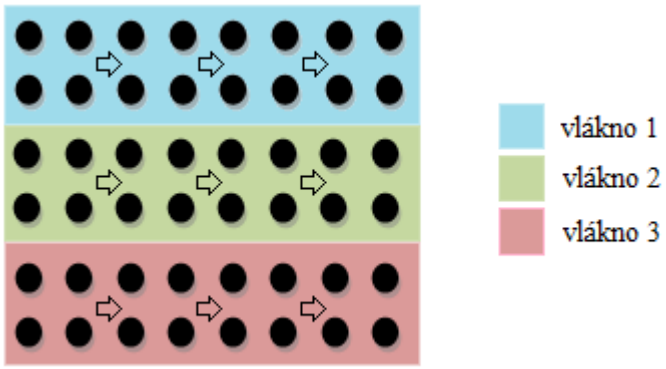
- Prvý prístup je, keď potomkovia nahradzujú rodičov až po výpočte a ukončení jednej generácie, to je tzv. „generačný“ prístup .

- Druhý prístup je „steady-state“, kde potomkovia nahradzujú rodičov okamžite po reprodukcii. V tejto metóde sa nedá striktne určiť kedy končí jedna generácia, celý proces prebehne stochasticky [10].

4.2.1 Rozdelenie záťaže medzi procesormi u celulárnych evolučných algoritmov

Najväčším problém rozdeleniu záťaže je, že celulárne topológie nám neposkytujú striktne ohraničené mikropopulácie, tak ako je tomu u distribuovaných algoritmoch. Môžeme tvrdiť, že kvôli tomu architektúra typu NUMA nie je najvýhodnejšia na takéto rozloženie populácie. Nedajú sa určiť hranice mikropopulácií. NUMA architektúra má distribuovanú pamäť, do ktorej ostatné procesory môžu pristupovať iba nepriamo. U celulárnych algoritmoch by bol neefektívny, s hladom na to, že kríženie sa uskutočňuje rovnakou rýchlosťou a princípom medzi všetkými jedincami. Z tohoto dôvodu sa používa architektúra typu UMA, u ktorej má každý procesor rovnakú rýchlosť prístupu k zdielanej pamäti. Počet procesorov u tejto architektúry je limitovaný, ale i tak sa dajú nájsť rýchle a kvalitné riešenia na optimalizačné problémy.

Existujú dva hlavné modely na rozdelenie záťaže výpočtov u celulárnych evolučných algoritmov. Prvý je model so statickým rozdelením záťaže, tj. každé vlákno vypočíta časť populácie. Tieto časti sa neprekrývajú a majú dotýkajúce sa hranice. Najjednoduchšia a možno aj najefektívnejšia varianta je, keď populáciu vertikálne rozdelíme na časti.



Obraz 4.4: Vertikálne rozdelenie záťaže u celulárnych evolučných algoritmov.

Vlákno postupne vyhodnocuje jednotlivé jedince , ktoré sa nachádzajú v jeho priestore. Po vyhodnotení každého jedinca z populácie sa uskutoční selekcia a reprodukcia. Tieto operácie sú tiež spracovávané zodpovedajúcim vláknom. Musíme si však uvedomiť, že kým nie je vyhodnotený každý jedinec v populácie, vlákna, ktoré skôr ukončili svoju úlohu nemôžu pristúpiť k ďalšiemu kroku algoritmu (k selekcie), tj. tieto vlákna sú daný čas nepoužívané. U nesynchronizovaného algoritmu by mohli nastať udalosti, pri ktorých by aplikácia skolabovala alebo by udávala nekorektné

výsledky. Predstavme si situáciu, kde jedinec s nevypočítanou fitness hodnotou by sa dostal do časti selekcie programu.

Tento statický model má najväčšiu nevýhodu v tom, že vlákna sa musia navzájom čakať. Synchronizácia celého algoritmu je nevyhnutná.

Druhá varianta je farmársky model, u ktorého je kľúčovým pojmom stav fitness funkcie jedincov. Populácia nie je rozdelená vertikálne na priestory, ale vlákna vyberajú postupne jedincov za sebou. Táto varianta dynamicky vyváži záťaž medzi vláknami. Jedince sú rozdelené do troch kategórií podľa vyhodnotenia fitness funkcie:

1. **Initial** - Nevyhodnotený fitness.
2. **Processing** - Práve prebieha vyhodnotenie fitness funkcie.
3. **Ready** - Vyhodnotená fitness funkcia.

Postup je jednoduchý, prvý proces vyberie jedného jedinca z populácie a zmení jeho stav z *Initial* na *Processing* a pričom vypočíta jeho fitness. Ostatné vlákna tohoto jedinca už nemôžu vybrať, a preto si vyberajú iných jedincov a zároveň tiež menia ich stavy. Po vypočítaní fitness funkcie vlákna zmenia stavy jedincov na *Ready,* ktorý indikuje, že jedinec už má vypočítaný fitness.

Tento model má nevýhodu v tom, že výber jedinca pre vyhodnotenie musí byť atomický. Predstavme si opačnú situáciu, že dva alebo viac vlákien by vybrali toho istého jedinca na vyhodnotenie v tom istom momente. Vtedy by mal jedinec v lepšom prípade nekorektnú fitness hodnotu, ale mohlo by dôjsť k skolabovaniu celého programu (napr. používanie deallokácie pamäti).

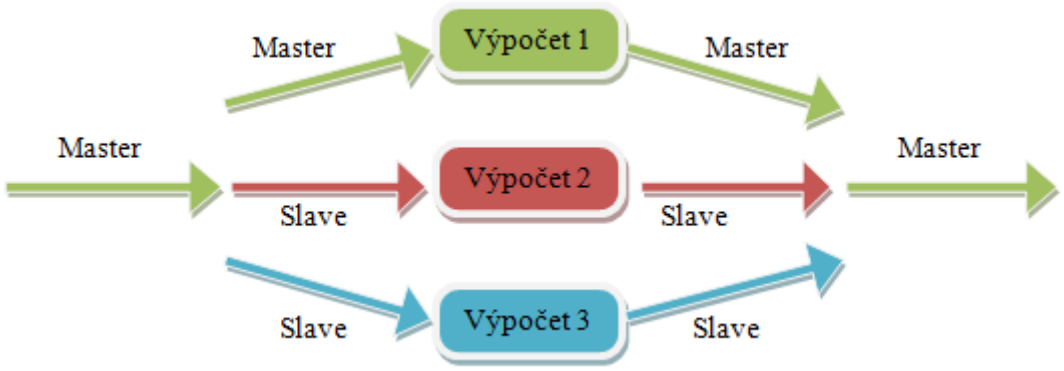
Farmársky model sa používa v prípade, že časová náročnosť vypočítania fitness hodnoty je rozličná u jedincoch. U modelu so statickým rozdelením, vysoká rozličnosť výpočtu fitness hodnoty spôsobuje, že niektoré priestory jedincov sú vypočítané rýchlejšie a hotové procesy musia zbytočne čakať. Inak povedané, výpočet jednej generácie trvá tak dlho, ako trvá výpočet najpomalejšieho vlákna. Farmársky model obíde tieto nedostatky statického modelu a čakanie na ostatné vlákna je zanedbateľné.

4.3 Kombinované paralelné evolučné algoritmy

Kombinované paralelné evolučné algoritmy sú také algoritmy, ktoré kombinujú viaceré topológie do algoritmu, napr. populácia je rozdelená do ostrovskej topológie, ale v rámci jednej mikropopulácie sú jedince v celulárnej topológie. Túto metódu je možné efektívne využiť pomocou internetu.

5 OPENMP (Open MultiProcessing)

OpenMP [13] je jeden z veľmi užitočných nástrojov na paralelné programovanie. Z hľadiska programátora sa jedná o jednoduché API na ovládanie a vytváranie viac vláknových aplikácií. Ide o viacplatformový API, ktorý sa skladá z niekoľkých direktív pre prekladač. Dá sa používať u programoch, ktoré sú napísané v jazyku Fortran a C/C++. Hlavný koncept viacvláknových aplikácií je, že na začiatku sa vytvorí *master* vlákno a pri programových častiach, kde je užitočné si rozdeliť záťaž, sa vytvoria *slave* vlákna, ktoré prevedú svoje časti výpočtov. Po viacvláknovej časti kódu vlákna opäť zanikajú a ďalej pobeží len *master* vlákno.



Obraz 5.1: Koncept viacvláknových aplikácií, pri používaniu OpenMP.

Ako som sa už zmienil OpenMP funguje na základe direktív, ktoré v jazyku C/C++ majú nasledujúci tvar:

#pragma omp <KEY_WORDS> [CLAUSE]

(5.1)

5.1 Hlavné direktívy

#pragma omp parallel [CLAUSE]

(5.2)

Táto direktíva označuje, že blok pod direktívou prebehne súbežne na všetkých vláknach. Počet vlákien je zadaný užívateľom pomocou funkcie *set_omp_threads*.

#pragma omp parallel for [CLAUSE]

(5.3)

Táto direktíva slúži na paralelizovania *for cyklu*. Vlákna rozdeľujú každú iteráciu *for* cyklu medzi sebou a to je podstata paralelizovania. Každé z vlákien vyberie jeden ešte neprevedený krok *for* cyklu a sú vypočítané súbežne. Tento proces sa opakuje dovtedy, kým sa nevypočítajú všetky iterácie. Na konci *for* cyklu je bariéra, ktorá slúži na to, aby každé vlákno, ktoré už má dokončenú prácu, počkalo na ostatné pracujúce vlákna.Master vlákno dostane riadenie naspäť iba vtedy, keď už všetky vlákna dokončili prácu.

#pragma omp critical

(5.4)

Do kritickej sekcie vždy vstúpi len jedno vlákno, ostatné čakajú pred ňou. Je to spôsob synchronizácie v OpenMP. Kritické sekcie niekedy spôsobujú spomalenie programu, je doporučené, aby sa používali čo najmenej. Táto sekcia je samozrejme definovatelná iba v rámci paralelnej sekcie.

#pragma omp atomic

(5.5)

Pragma *atomic* je ďalší spôsob synchronizácie. Dajú sa použiť len na jednoduché matematické operácie. Pôsobí len minimálne spomalenie programu.

#pragma omp barrier

(5.6)

Bariéra je jeden z najužitočnejších spôsobov na synchronizáciu programu. Princíp fungovania je, že každé vlákno je nútené čakať, kým ostatné vlákna dokončia prácu.

5.2 Clausule

OpenMP [13], [14] má jednoduché pravidlá na riešenie privátnej a verejnej pamäti. Prvé pravidlo je, že každá globálna premenná v rámci programu je verejná, tj. každá globálna premenná sa vyskytuje len raz v pamäti. Druhé pravidlo je, že premenné, ktoré boli vytvorené v rámci funkcie, kde sa nachádza paralelná sekcia, môžu byť privátne aj verejné na závislosti ich nastavenia užívateľom. Keď lokálne premenné nie sú nastavené, tak sa uskutoční tzv. základné nastavenie, ktoré je zmeniteľné s clausulou default. Clausula defualt môže previesť tri hodnoty, *shared*, *private* a *none*. Hodnota *shared* znamená, že každá nedefinovaná lokálna premenná bude verejná, naopak a *privát*, že budú privátne. Hodnota *none* sa používa na testovanie. Užívateľské nastavenie sa robia pomocou clausule *private* a *shared*. Posledné pravidlo že, každá premenná, ktorá bola vytvorená v rámci paralelnej sekcie je privátna.

Príklad clausule :

```
int x,z,q,w,f;
#pragma omp parallel for shared(x,z) private(q,w) default (none)
```

(5.7)

V tomto prípade premenné *x* a *y* budú verejné, *q* a *w* budú privátne. Ako už bolo zmienené predtým, hodnota *none* pri nastavení základnej hodnoty slúži na testovanie. V prípade existencie nejakých lokálnych premenných (ktoré sú používané v paralelnej sekcie), ktoré nie sú v zozname clausule *shared* a *private*, prekladač vypíše chybovú hlášku. Týmto spôsobom je zabezpečená kontrola nastavenia verejných a privátnych premenných.

6 Návrh difúzneho evolučného algoritmu

Ako už bolo zmienené, táto práca slúži na experimentálne zistenie najlepších parametrov a overenie viacerých predpokladov difúznych evolučných algoritmov. Na experimentálne zistenie parametrov je potrebná implementácia rôznych demonštračných problémov. Z tohoto faktu vyplýva, že jadro aplikácie musí byť prenosné a znova použiteľné. Túto funkciu by mohla splniť jedna obecná knihovňa, ktorá rieši hlavné procesy difúznych evolučných algoritmov, ako selekcia, reprodukcia, mutácia atď. Táto knihovňa musí poskytovať jednoduché rozhranie, čím je možné riadiť simulácia a umožňuje nastavenie parametrov evolučného algoritmu. Musí obsahovať zdrojový kód na prevedenie sekvenčného a zvlásť kód na prevedenie paralelného evolučného algoritmu. Do knihovne je nutné ešte zabudovať monitorovací systém, ktorým sa dá pozorovať stav, dočasné hodnoty a výsledky simulácie. Je nutné brať do úvahy, že monitorovací systém nesmie prekážať behu algoritmu, inak by sa dosiahlo neplatných výsledkov, hlavne čo sa týka sledovanie časovej náročnosti algoritmu. Z tohoto vyplýva, že monitorovací systém musí uložiť dáta na pevný disk, aby sa dosiahlo prípadnému zobrazeniu výsledkov po behu simulácie, čiže je potrebná implementácia takých samostatných aplikácií, s ktorými sa dajú zobraziť výsledky evolučných algoritmov a vývin populácie podľa uložených dát.

6.1 Demonštračné problémy

Ako demonštračný bolo vybrané dva príklady. Prvý príklad slúži na testovanie efektívnosti a sledovanie časovej náročnosti difúznych evolučných algoritmov aplikované na triviálnom problému. Druhý príklad požaduje dlhšie výpočty a slúži na overenie schopnosti plánovania difúznych evolučných algoritmov. Tento problém slúži na sledovanie časovej náročnosti, keď algoritmus vyžaduje robustné výpočty.

Ako triviálny bolo zvolené knapsack problém[15]. Druhý robustnejší problém je problém plánovania mostnej konštrukcie.

6.2 Knapsack problém

Prvý zvolený problém je tzv. knapsack problém. Je to NP-úplný problém. Problém sa dá popisovať nasledujúce :

„ Ide o situáciu zlodeja, ktorý chce vykradnúť objekty z izby. Je daný batoh zlodeja, ktorý má nijakú váhovú kapacitu a sú dané objekty v izbe nejakou váhou a nejakou cenou. Otázkou je, ktoré objekty by mal zlodej zobrať aby odniesol najväčšiu hodnotu, ale váha odnesených objektov nesmie prekročiť kapacitu batohu [15]. „

6.3 Návrh riešenia problému knapsack

Je to typický príklad, čo je možné riešiť (aproximovať riešenie) s evolučnými algoritmami. Jedince v populácií reprezentujú jednotlivé pokusy zlodeja. Objekty sú popísané s dvoma konštantami a s jednou premennou, ktoré sú :

- **váhová konštanta**
- **cenová konštanta**
- **logická premenná** (0/1) - či vec je v batohu zlodeja.

Algoritmus môže len zmeniť premennú „či objekt je v batohu zlodeja". Tento problém je ľahko implementovatelný s genetickým algoritmom. Z dôvodu, že je daná len jedna logická premenná, genóm (postupnosť bitov) popisuje, ktoré objekty sú v batohu. Mutácia a reprodukcia funguje obyčajne ako u všetkých genetických algoritmov, zmenia sa hodnoty jednotlivých bitov. Fitness funkciu tiež pomerne jednoducho navrhnutá. Jedince budú ohodnotené podľa ceny objektov, ktoré odnesie zlodej. Problémom je určenie fitness hodnoty tých chromozómov, ktoré majú prekročený limit váhy. Tieto jedince by sa dali ohodnotiť s nulovou fitness hodnotou, ale tento prístup by nebolo najvýhodnejšie riešenie. Stratili by sa riešenie, ktoré sa len mierne prekročia limit a sú blízko k lokálnemu maximu fitness povrchu. Je lepší spôsob zaviesť penalizáciu u týchto jedincov. Každému jedincovi, ktorý má prekročený limit, je odčítaná z jeho fitnessu hodnota, ktorá je v pomeru veľkosti prekročenia.

$$fitness = (\sum_{i=1}^{n} cena_i) - penalizácia \tag{6.10}$$

Penalizácia môže byť napr. nasledujúca [16] :

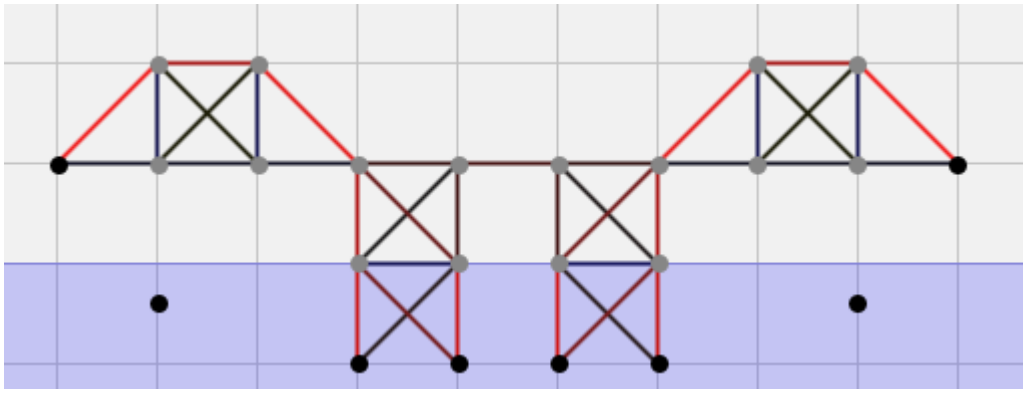
$$penalizácia = (\sum_{i=1}^{n} váha_i) - kapacita \tag{6.11}$$

Tým pádom aj jedince, ktoré prekročili kapacitu vrecka sa zúčastnia v selekcie, tj. môžu byť vybraný na reprodukciu.

6.4 Problém plánovania mostnej konštrukcie

Druhý príklad je simuláciou konštrukcie mostu. Cieľom algoritmu je vymodelovať mostnú konštrukciu tak, aby sa nezrútila. Z dôvodu že simulovanie mostu je časovo náročné, preto dátová reprezentácia a pravidlá modelu boli zjednodušené. Pre jednoduchosť nie sú rozlíšené materiály mostu, tj. všetky komponenty sú rovnakého homogénneho materiálu.

Model mostu je obecná grafová štruktúra. Z pohľadu implementácie je grafová štruktúra reprezentovaná ako množina uzlov a množina hrán. V modelu majú váhu a pozíciu len uzly, hrany(prúty) slúžia len na spájanie uzlov. Hrany sú schopné deformovať sa a prípadne sa zlomia pod veľkým záťažením. Hrany majú maximálnu dĺžku, ktorá je vždy predom definovaná. Aby sa model čím viac priblížil k reálnemu mostu, z dôvodu jednoduchosti váha uzlu je konštantná (jedna). Na simuláciu zemského povrchu existujú tzv. pevné uzly, ktoré majú nekonečnú váhu. Úlohou každého mostného modelu je stabilizácia cesty. V simulácii hlavnú rolu hrá gravitácia, ktorá pôsobí konštrukciu. V modelu je cesta definovaná, ako vodorovná postupnosť hrán a uzlov medzi dvoma pevnými uzlami. Cieľom celého algoritmu je stabilizovať túto cestu, aby sa pôsobením gravitačnej sily čo najmenej zdeformovala.



Obraz 6.1: Model mostu. čierne body na obrázku reprezentujú pevné uzly. Farba hrany reprezentuje záťaž aká na nich pôsobí. Modrá farba ukazuje, roztiahnutie hrany, červená farba stlačenie. Modrý obdĺžnik ukazuje hladinu vody.

6.4.1 Fitness hodnota mostu

V skutočnosti je posúdenie mostu zložité. Je nutné brať do úvahy napr. ekonomické, funkcionálne, estetické a ďalšie faktory. Fitness hodnota tohoto modelu mostu má tiež viacero aspektov, ale hlavným je vytrvalosť konštrukcie.

Otázkou je, ako je možné testovať vytrvalosť? Jeden z postupov je, že sa zväčšuje zaťaženie mostu až k jeho porušeniu prípadne zrúteniu. Veličina vytrvalosti mostu by sa potom rovnalo

zaťaženiu, ktoré bolo v poslednom momente na konštrukcii. Tento model kvôli jednoduchosti postupuje tak, že postupne zvyšuje gravitáciu ktorá pôsobí na celú konštrukciu. Ďalší parameter fitness funkcie je cena mostu, ktorá je určená podľa celkovej dĺžky spojov. Tento faktor okrem toho, že motivuje algoritmus vytvoriť lacnejšie mosty, zabráni aj zabudovaniu zbytočných komponentov do konštrukcie. U evolučného algoritmu, ktorý vytvorí konštrukciu mostu, je pridanie zbytočných komponentov vo väčšine prípadov škodný jav. Na druhej stranesa nedá posúdiť počas vývoja ktoré komponenty budú zbytočné a ktoré budú dôležité k stabilite modelu.

V tejto otázke je treba zaviesť kompromis. Na začiatku vývoja jedincov (mostov), cena ešte nie je započítaná do úvahy (je možné dokonca brať cenu ako kladný parameter - čím má most väčšiu cenu , tým bude mať väčší fitness, dôvodom tohoto prístupu je, že algoritmus na začiatku vývoja je motivovaný vytvoriť čím viac spojov, čím sa zrýchli rast fitness hodnoty). Pre stabilnejšie modely (mosty s vysokou fitness) sa cena berie opačne, čím má most väčšiu cenu, tým má menšiu fitness hodnotu. Toto je motiváciou na minimalizáciu a odstránenie zbytočných komponentov z konštrukcie.

Podľa skúsenosti na začiatku simulácie sú mosty ešte tak nestabilné, že ani najslabšiu gravitáciu nevydržia. Kvôli tomu je výhodnejšie počítať do fitness hodnotu deformácie celej cesty a gravitáciu nastaviť na malú konštantnú veličinu. Hodnota deformácie cesty pri malej gravitácii je vypočítaná tak, že každá pozícia uzlov cesty je porovnaná s jej počiatočnou pozíciou. Hodnota bude pomerná s veličinou rozdielu. To nám dáva oveľa podrobnejšie hodnotenie slabých mostov, hlavne na začiatku simulácie.

V zhrnutí fitness funkcia mostov má tri etapy:

1. **Konštrukcia** (test konštrukcie), v tejto etape fitness funkciu je možné navrhnúť nasledovne:

$$fitness = \frac{a}{b * deformácia} \tag{6.1}$$

   Gravitácia kvôli nepresnosti odhadu, nie je braná do funkcie a je nastavená na malú konštantu. Čím je viac deformovaná cesta tým má menšiu fitness, podľa testov cenu není treba brať kladne do úvahy.

2. **Optimalizácia** (test vytrvalosti), v tejto časti už je vyriešená počiatočná deformácia mostu, ktorá je spôsobená malou konštantnou gravitáciou. Gravitačná sila sa zvyšuje, pokiaľ deformácia cesty neprekročí dopredu definovaný limit. Funkcia fitnessu je :

$$fitness += c * gravitácia \tag{6.2}$$

Mosty musia najprv absolvovať test konštrukcie (1. etapa) a potom pokračujú do druhej etapy,preto je hodnota gravitácie pripočítaná a nie je prirovnaná k fitness hodnoty.

3. **Minimalizácia** (test ceny), v poslednej etape je treba odstrániť zbytočné komponenty mostu pomocou ceny. Čím má most väčšiu cenu tým má menší fitness:

$$fitness+=\frac{d}{e*cena} \tag{6.3}$$

Poznámka: a,*b* a *c* sú konštanty fitness funkcie. Na základe experimentov najlepšie hodnoty pre tieto konštanty sú :

a	*b*	*c*	*d*	*e*
100	1	2	100	1

Tabuľka 6.1: Nastavenie parametru fitness funkcie

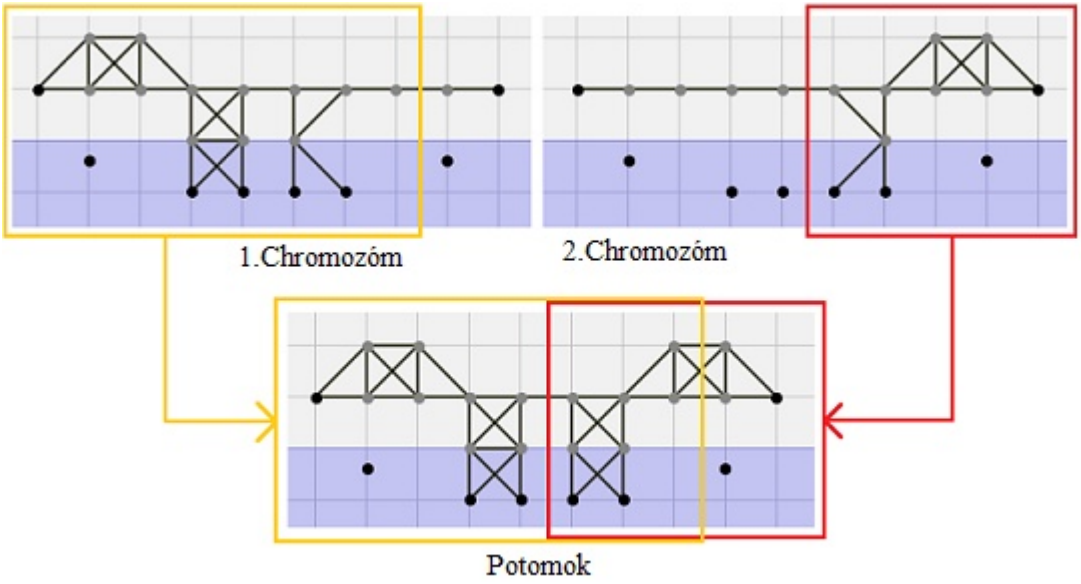
Je veľmi dôležitým faktom, že fitness hodnota v neskorších etapách je väčšia. To neumožňuje, aby došlo k podceneniu mostu v porovnaní (pri selekcii) s mostami z predošlých etáp fitness funkcie.

6.4.2 Inicializácia mostov

K inicializácii mostov v populácii je treba zmieniť, že je jedna určitá časť konštrukcie pre každý most totožná. Táto časť konštrukcie mostu je cesta, ktorá v procese reprodukcie a mutácie nie je zmenená. Každý potomok bez výnimky po reprodukcii bude obsahovať túto časť grafu.

6.4.3 Návrh reprodukcie mostov

Reprodukcia sa môže uskutočniť viacerými spôsobmi. Dátová reprezentácia konštrukcie je grafová štruktúra. Z tohoto faktu vyplýva, že algoritmus reprodukcie sa bude podobať na algoritmus reprodukcie u genetických programovaní. Problém spočíva v tom, že graf mostu je obecný graf a preto je ťažké si vybrať „podgraf“ na reprodukciu. Na prekonanie tohoto problému sa dajú používať napr. geometrické kríženia. Tým že majú uzly pozíciu, dajú sa rozdeliť genómy podľa osy kríženia.



Obraz 6.2: Geometrické kríženie. Existujú dve osy kríženia (ľavá a pravá), ktoré sú rozdelené do rodičovských modelov. Všetky body, ktoré sú na ľavej strane pravej osy, a všetky body, ktoré sú na pravej strane od ľavej osy modelu sú kopírované do genotypu potomka.

Druhá metóda kríženia spočíva v tom, že od oboch rodičov sú náhodne vybrané hrany s vhodnými uzlami, ktoré sú zabudované do konštrukcie potomka. Keď sa vyskytne situácia, že rovnaká hrana už existuje, tak nedôjde k prekrytiu tj. duplikácia hrán nie je povolená. Dajú sa používať variácie tejto metódy, ako napr. algoritmus s ocenenými hranami, kde dôležitejšie hrany sú pravdepodobnejšie vybrané. V prípade tejto metódy sa musí položiť otázka, ktoré hrany sú dôležité. Podľa skúsenosti, čím má hrana väčšiu deformáciu, tým je dôležitejšia. Príkladom tohoto tvrdenia sú stĺpy a oblúky. Ich komponenty sú pomerne viac deformované, pretože tieto časti prenášajú najväčšiu časťzaťaženia mostu a preto v tejto variante pravdepodobnosť na vybranie tých komponentov je väčšia.

6.4.4 Návrh mutácie mostu

Mutácia je pomerne jednoduchá operácia. Dá sa definovať 5 rozličných javov mutácie:

1. **Pridanie uzla** – v tomto prípade sa vytvorí nový uzol ktorý sa pridá do množiny uzlov. Vytvoria sa tri hrany, pričom jeden koniec týchto hrán je pripojený k novovytvorenému uzlu. Tak je zabezpečené aby prebytočné uzly neboli vytvárané.
2. **Pridanie hrany (spojenie uzlov) –** Spoja sa ešte dva nespojené uzly, ale len vtedy, keď vzdialenosť uzlov nie je väčšia ako maximálna dĺžka hrany.
3. **Odstránenie uzla –** Náhodne sa odstráni uzol, ktorý nie je súčasťou cesty.

4. **Odstránenie hrany –** Náhodne sa odstráni hrana, ktorá nie je súčasťou cesty.
5. **Zmeniť pozície uzla –** Náhodne sa zmení pozícia jedného vybraného uzla, ktorý nie je súčasťou cesty.

Pri procese mutácie vo väčšine prípadov sú hore definované operácie prevedené viackrát.

6.5 Váhovo-agregačná fyzika (Mass-aggregate physics)

Pre výpočet fitness hodnoty mostovej konštrukcie potrebujeme fyzikálny model. V súčasnej dobe sa prudko rozvíja tieto modely v oblasti počítačových hier, preto využijeme jeden, ktorý už existuje.

V dobe výkonných počítačov je požadované vyvinutie celkom novej sekcie informačných technológií. Je to technológia fyzických motorov, ktoré sú motivované s vývojom vysoko kvalitných rozhraní hier. Fyzické motory sú väčšinou zamerané na dynamiku a kinematiku. Tieto simulácie si vyžiadajú veľmi náročné výpočty. Fyzické motory (Physics engines) tvoria logicky oddelenú časť programu. Hry a simulácie si vyžiadajú ich fyzický motor, aby previedol svoje výpočty na základe aktuálnych stavov entít v simulácii a podľa odpovedí motora je obnovený stav entít. Väčšiny simulácií sú diskrétne .

Fyzické motory môžeme zariadiť do viacerých kategórií. Hlavné kategórie sú:

1. Časticové fyzické motory (Particle physics engines)
2. Váhovo-agregačné fyzické motory (Mass-aggregate physics engines)
3. Fyzický motor tuhých telies (Rigid body physics engines)

Tieto druhy sú základné, ktoré sú používané v počítačových hrách. Váhovo-agregačným fyzickým motorom je možné simulovať model mostu, ktorá je zmienená v predošlých kapitolách, preto bude ďalej rozpísaná [17].

Váhovo-agregačný fyzický motor je diskrétny model, čo sa snaží simulovať fyzické interakcie, pohyby a sily pomocou častíc a spojov medzi nimi. Častice sú vektory (pozícia), ktoré majú nejakú váhu. Spoje slúžia na spájanie častíc. Tieto spoje môžu nadobúdať rôzne charaktery ako sú tyče, laná alebo pružiny. Častice okrem váhy a pozície vlastnia ešte dva vektory: vektor akumulovanej sily, čo je suma všetkých síl, čo na jednu časticu pôsobí a vektor rýchlosti, čo zmení pozíciu častice. Pomocou týchto vektorov je možné vypočítať trajektória častice. Výpočet jedného kroku sa nazýva integrácia, kde vektor sily zmení rýchlosť a vektor rýchlosti zmení pozíciu častice. Sily sú v každom kroku vynulované a znova vypočítané podľa prostredia v ktorom sa častica nachádza. Základné sily sú gravitácia a sily spôsobené pružinami. Gravitácia je vektor pôsobiaci v smere „-*y*“.

Hlavné komponenty, ktoré sú nenahraditeľné k modelovaniu mostu sú tyče (Rods). Tyče sú také spoje, ktoré držia od seba dve častice v konštantnej vzdialenosti nezávisle od sily a rýchlosti, čo pôsobí na ne. Týmito tyčami sa dajú simulovať komponenty mostu. Reakcie tyče nepôsobia na vektor sily častice, ale priamo na vektor rýchlosti. Zmena rýchlosti spôsobená tyčami funguje na základe pojmu akcia-reakcia. Existujú tri základné premenné, podľa čoho sa určí nová rýchlosť častíc spojené tyčou :

- **Normála spoje (Contact normal)** – normalizovaný vektor z rozdielu pozície dvoch častíc (*A*,*B* sú častice).

$$normal = normalize(A_{pos} - B_{pos}) \tag{6.5}$$

  Normalizovaný vektor má dĺžku 1.

- **Rozdiel rýchlosti (Separating velocity)** – dá sa popísať rovnicou :

$$velocity_{separating} = (A_{velocity} - B_{velocity}).normal \tag{6.6}$$

  Bodka znamená skalárny súčin, tým pádom rozdiel rýchlosti nebude vektor, ale reálna hodnota.

- **Totálna inverzná váha častíc (Total inverse mass)** – znamená súčet váhy častíc, ktoré sú spájané tyčou. Kvôli jednoduchosti je recipročná hodnota váhy implementovaná vo fyzickom motoru, tj. nekonečná váha v motoru má hodnotu 0 atď. Tým pádom totálna inverzná váha je:

$$totalInverzMass = \frac{1}{A_{mass}} + \frac{1}{B_{mass}} \tag{6.7}$$

Nové rýchlosti častíc sa dajú vypočítať nasledujúcim spôsobom:

$$\begin{aligned} &If\ (velocity_{separating} > 0 \vee totalInverzMass = 0) return\ ; \\ &vector_{impulse} = \frac{-normal * velocity_{separating}}{totalInverzMass}; \\ &A_{velocity} = A_{velocity} + \frac{vector_{impulse}}{A_{mass}}; \\ &B_{velocity} = B_{velocity} - \frac{vector_{impulse}}{B_{mass}}; \end{aligned} \tag{6.8}$$

Ďalší jav čo je potreba simulovať v modelu je deformácia mostu. K tomu je však treba vysvetliť pojem preniknutie (Penetration). Preniknutie u tyčí znamená, že vzdialenosť častíc nie je rovná so pôvodnou vzdialenosťou. Na riešenie tohoto problému slúži systém, ktorá sa nazýva *anti-penetration systém.* Tento systém po zistení preniknutia opraví pozície častíc pomocou normál

vektoru tyče (pôsobí hneď na pozíciu častíc, nepôsobí však ani na rýchlosť ani na silu). Veľkosť preniknutia je implementovaná ako premenná tyče [17].

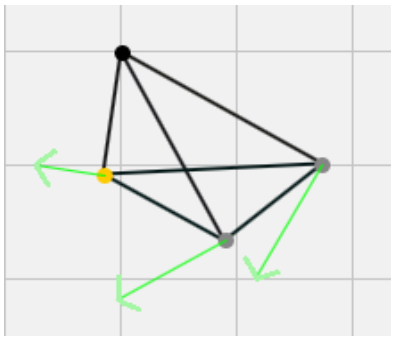
$$penetration = length_{default} - \sqrt{A_{pos}^2 + B_{pos}^2} \tag{6.9}$$

Hodnota deformácie tyče je v pomere so súčtom hodnôt preniknutia všetkých iterácií rezolúcie tyčí. (Rezolúcia znamená jeden krok na výpočet novej rýchlosti a opravu preniknutia u tyčí)

$$deformation \approx \sum_{i=0}^{n} penetration_i \tag{6.10}$$

Kde *n* je počet rezolúcií, ktoré sú prevedené na tyč v jednom kroku diskrétnej simulácie.

Rezolúcia sa neprevedie iba raz, ale niekoľko desať- až dokonca sto krát v každej iterácii pre všetky tyče v diskrétnej simulácii. Čím viac krát sa rezolúcia prevedie, tým viac sa bude model podobať skutočnej konštrukcii.



Obraz 6.3: Príklad na rezolúcie pohybu u váhovo-agregačnom fyzickom modelu.

Ako už bolo zmienené deformácia je základom fitness funkcie modelu mostu. Váhovo-agregačný fyzický model nám udáva pomerne presné hodnoty deformácie jednotlivých spojov v konštrukcie mostu, z tohoto dôvodu je používaná v simulácie.

7 Implementácia

Implementačnú časť tejto práce tvorí päť rôznych programov (modulov). Hlavným a centrálnym modulom je knihovňa difúzneho evolučného algoritmu, ktorá sa nazýva *Evolution*. Knihovňa *Evolution* poskytuje možnosť aplikovania difúzneho evolučného algoritmu všeobecne na rôzne typy optimalizačných problémov. Druhá časť implementácie je program na simuláciu fyziky mostov. Skladá sa z dvoch častí: fyzický motor a aplikovanie difúzneho evolučného algoritmu na plánovanie mostnej konštrukcie. Treťou časťou je aplikovanie difúzneho evolučného algoritmu na riešenie knapsack problému. Posledné dva programy sú *BridgeModeller* a *PopulationViewer,* ktoré slúžia na plánovanie a testovanie mostov. *PopulationViewer* je malý program, ktorým je možné analyzovať a graficky znázorniť vývin populácie po priebehu simulácie difúzneho evolučného algoritmu.

7.1 Knihovňa Evolution

Knihovňa *Evolution* je napísaná v jazyku C++. Skladá sa z dvoch modulov :

1. *Chromosome*
2. *Evolution*

Modul *Evolution* je implementácia difúzneho evolučného algoritmu pomocou OpenMP. Obsahuje jednu triedu. Modul *Chromosome* obsahuje tri triedy na všeobecné simulovanie jedincov a populácie. Triedy v modulu *Chromosome* sú : *Population*, *ChromosomeFactory a Chromosome*.

Chromosome je abstraktná trieda, slúži ako rozhranie (interface) jedincov. Obsahuje šesť dôležitých funkcií, ktoré sú :

1. ***virtual float fitness () = 0*** - je abstraktná funkcia pre jedinca. Implementácia tejto funkcie by mala vrátiť fitness hodnotu chromozómu. Je to privátna funkcia, a zavolá ju len verejná funkcia *getFitness().*
2. ***virtual void mutate (int ratio) = 0*** - v tejto abstraktnej funkcii by sa mal uskutočniť mutačný proces. Parameter *ratio* udáva, že aká veľká zmena by sa mala pomerne uskutočniť v genotypu jedinca.
3. ***virtual *Chromosome reproduceFrom (Chromosome * second) = 0*** - funkcia prevedie reprodukciu dvoch jedincov. Prvý jedinec je ten u ktorého je funkcia zavolaná, druhý jedinec je parameter funkcie. Mala by vrátiť novovytvoreného jedinca, ktorý bol vytvorený operátorom *new*.
4. ***float getFitness()*** - funkcia zavolá virtuálnu funkciu *fitness()*. Hodnotu vrátenú od nej uloží do premennej *cachedFitness*. Pri nasledujúcich volaniach je hodnota vrátená z tejto premennej.

5. ***virtual void serialize (ostream & stream)*** - vypíše jedinca na *stream* ako reťazec znakov. Uloží jedinca napr. do súboru, odkiaľ je možné znova načítať.
6. ***virtual void init()*** - táto funkcia slúži na inicializáciu jedinca. V implementácii tejto funkcie by sa malo uskutočniť nastavenie počiatočných hodnôt chromozómu.

Túto triedu nie je možné vytvoriť, je abstraktná. Všetky triedy na riešenie problémov (ako knapsack a plánovanie mostu), ktoré sú implementované, dedia od tejto triedy.

Druhá trieda je *Population*. Táto trieda obsahuje dvojrozmerné pole chromozómov. Veľkosť pola je uchovaná v premenných *width* a *height.* Populáciu je možné vytvoriť len pomocou *ChromosomeFactory*, preto treba pridať túto triedu pri vytvorení populácie ako parameter konštruktoru . *ChromsomeFactory* je trieda, ktorá funguje ako generátor jedincov, vytvorí dvojrozmerné pole chromozómov. Na vytvorenie pola sú k dispozícií dve funkcie : *createChromosomes()* a *createArray()*. *CreateChromosomes* vytvorí pole, ktoré bude obsahovať objekty jedincov a vráti ukazovateľ na toto pole. Napriek tomu *createArray* vytvorí len pole typu ukazovateľ na objekty *Chromosome,* ale nie sú v ňom allocované jedince. Je to dvojrozmerné pole ktoré obsahuje hodnoty NULL. Trieda *ChromsomeFactory* používa C++ šablónu (Template), podľa ktorej sú vytvorené jedince. (UML diagram tried je v prílohe tejto práce)

7.1.1 Trieda Evolution

Evolution je základná trieda na prevedenie difúzneho evolučného algoritmu. Táto trieda je navrhnutá tak, aby s ňou bolo možné testovať a pozorovať časový rozdiel medzi paralelnými a sekvenčnými algoritmami. Trieda má dve hlavné funkcie, ktoré sú: simulácia sekvenčného difúzneho algoritmu a simulácia paralelného difúzneho algoritmu. Obsahuje premenné na podmienky ukončenia simulácie (termination condition) a ďalšie premenné, ktoré sú nutné pre simuláciu. Sú implementované dve podmienky ukončenia :

1. Maximálna generácia dosiahnutá
2. Požadovaná fitness je dosiahnutá

Tieto podmienky sú uchované v premenných *generations a requestedFitness*. Ďalej je daná premenná, čí ide o kríženie podľa štyroch alebo ôsmych rozvetvených tvarov. Táto premenná sa nazýva *fourway*. Na nastavenie počtu vlákien u paralelnej simulácie slúži premenná *threads.* Premenná *population1* je hlavná populácia simulácie, do *population2* sú pridané nové jedince (potomky), ktoré sú vytvorené funkciou reprodukcie. V procesu rekombinácie, potomky, ktoré boli pridané do *population2* nahradzujú ich rodičov v *population1* v prípade, keď majú väčšiu fitness hodnotu ako ich rodičia. Každá premenná je nastaviteľná pomocou jeho *set* funkcií. Medzi dôležité funkcie patria ešte *init()*, *dirx(int q)* a *diry(int q).* Funkcia *init()* inicializuje jedincov v populácií

pomocou virtuálnej funkcie *init()* z triedy *Chromosome*. *Dirx(int q)* a *diry(int q)* sú špeciálne pomocné funkcie pre zistenie tvaru susedstva pri selekcií. Parameter *int q* môže nadobúdať hodnoty 0 až 3 pre štvor-rozvetvené okolie, 0 až 7 pre osem-rozvetvené okolie. Podľa parametru *q dirx* a *diry* sa vráti pozícia susedného jedinca z okolia. Tieto funkcie sú používané v funkcii *getBest()*, čo je hlavnou funkciou selekcie. Vráti najlepšieho jedinca z okolia.

7.1.2 Sekvenčná simulácia

Sekvenčnú simuláciu je možné spustiť pomocou funkcie *runSequential()*. Konštrukcia funkcie je relatívne jednoduchá. Najprv je vypočítaná fitness hodnota všetkých počiatočných jedincov. Výpočet všetkých hodnôt je implementovaný zloženým *for* cyklom (prvá stúpa vertikálne, druhá horizontálne). Virtuálna funkcia chromozómu *getFitness()* je zavolaná pri každej iterácie dvojitého cyklu. Táto funkcia okamžite uloží (cachuje) fitness hodnotu, takže pre ďalšie využitie nie je zbytočne počítaná. Algoritmus obsahuje cyklus typu *for*, čo počíta generácie. Tento cyklus zahrňuje všetky ďalšie časti simulácie. Ako prvé sa prevedie selekcia jedincov (každá fitness hodnota je už v tejto chvíli vypočítaná), v tejto časti je tiež používaný podobný zložený *for* cyklus. Pre každého jedinca sa vyberie jeden pár na reprodukciu. Tento proces sa uskutoční pomocou funkcie *getBest(int x, int y)*. Reprodukcia sa uskutoční pomocou virtuálnej funkcie *reproduceFrom(Chromosom * second)* z triedy *Chromosome*. Po reprodukcii je prevedená operácia mutácie: *mutate (int ratio)*. Novovytvorený jedinec je pridaný do populácie *population2* na totožnú pozíciu ako je pozícia jeho prvého rodiča (v *population1*). Po mutácii, potomky, ktoré majú lepšiu fitness, ako ich rodičia, nahradia ich rodičov v pole chromozómov *population1*. Ostatné jedince sú deallocované. Z tohto vyplýva že novovytvorené jedince majú už v tejto časti vypočítanú fitness hodnotu.

Posledná fáza funkcie je logovanie a serializácia. Logované sú fitness hodnoty jedincov v populácii pri každej generácii. Tieto dáta používa program *PopulationViewer*. Iba najlepší jedinec, ktorý priniesol vylepšenie, je serializovaný. Na konci každej generácie sú testované podmienky ukončenia. V prípade splnenia niektorej z podmienok program končí.

7.1.3 Paralelná simulácia

Paralelná simulácia je zložitejšia ako sekvenčná, ale základné kroky simulácie sú totožné. Je implementovan pomocou knihovne OpenMP. Na začiatku funkcie sa rozdelí populácia medzi vlákna. Funkcia *getDivide()* vráti pole z typu *integer* (statický model rozdelenia záťaže). Tieto hodnoty sú indexy jedincov a znamenajú hranice (začiatok a koniec) pre vlákna. Index jedinca je vypočítateľná pomocou vzorca :

$$index = x + y * width \tag{7.10}$$

Kde *x* a *y* sú parametre pozície jedinca a *width* je šírka populácie. Funkcia *getDivide()* vráti hodnotu, ktorá sa rovná počtom vlákien plus jedna. Dôvodom tohoto je, že okrem prvého a posledného vlákna, jeden index znamená začiatok pre jedno a koniec pre ďalšie vlákno. Napr. vlákno s číslom N bude počítať jedince s indexom od *getDivide()[N]* až do indexu *getDivide()[N+1]*. Po rozdelení populácie je nastavený počet vlákien pre OpenMP. Realizácia je uskutočnená pomocou funkcie *omp_set_num_threads(int t)*. Funkcia je zavolaná s parametrom *threads* (počet vlákien). Po nastavení vlákien nasleduje OpenMP direktíva:

```
#pragma omp parallel
```

(7.1)

Táto direktíva indikuje paralelnú sekciu simulácie. Od tohoto bodu program prebieha paralelne s viacerými vláknami. Každé vlákno počíta generácie tak, že hlavný cyklus na simuláciu generácií je pod paralelnou sekciou. Pred hlavným cyklom je vypočítaná fitness hodnota každého jedinca pomocou indexovaného *for* (pre vlákna) cyklu. Prvým krokom každej generácie je vynulovanie verejných premenných. Táto sekcia je vykonaná jedným vláknom, kvôli tomu je používaná direktíva :

```
#pragma omp single
```

(7.2)

Ostatné vlákna čakajú, kým vybrané vlákno vynuluje verejné premenné. Na synchronizáciu (čakanie) je používaná OpenMP bariéra. Po tejto sekcii sa nachádza selekcia, ktorá prebehne pomocou indexovaného for cyklu (pre každé vlákno iné indexy). Selekcia, reprodukcia a mutácia prebehnú podobne. Operácie rekombinácia a logovanie sú prevedené jedným vláknom, tj. sú pod OpenMP *single* sekciou. Z dôvodu synchronizácie medzi jednotlivými časťami *for* cyklu a na konci každej generácie sa nachádza bariéra. Hlavný rozdiel medzi sekvenčným a paralelným algoritmom je, že z dôvodu indexovania (používanie funkcie *getDivide()*) paralelný algoritmus nepoužíva dvojité *for* cykli. Podmienky ukončenia sú testované za poslednou bariérou v každej iterácii hlavného cyklu.

7.2 Implementácia riešenia problému mostu

Hlavným modulom tejto aplikácie je *World.h*. Modul *World* obsahuje jednu triedu, ktorá sa nazýva *World*. Táto trieda dedí od abstraktnej triedy *Chromosome*. Z tohoto vyplýva, že táto trieda reprezentuje jedinca v populácií. Modul obsahuje objekty fyzického motora. V tejto triede sú implementované častice a hrany fyzického motora pomocou štandardných vektorov jazyku C++ (*vector<T>*). Existuje jeden vektor na častice a jeden na hrany medzi nimi. V triede *World* sú ešte premenné na zúženie činnosti difúzneho evolučného algoritmu. Tieto premenné sú statické a sú nazývané *maxContacts a maxBodies*. Premenné *maxBodies* a *maxContacts* zabraňujú tomu, aby

algoritmus vytvoril veľké množstvo spojov a častíc. Každý model mostu (*World*) si naviac uchová dvojrozmerný geometrický vektor gravitácie. Objekt *resolver v* triede slúži na rezolúciu fyzických reakcií (viz. kapitola 6.2). Posledná a najdôležitejšia premenná v triede je premenná *temp*. Objekt *temp* je statický a obsahuje model cesty mostnej konštrukcie, tzv. šablónu mostu. Každý objekt vytvorený z triedy *World* je inicializovaný podľa tejto šablóny.

Trieda samozrejme implementuje virtuálne a čiste virtuálne (pure virtual) funkcie rodičovskej triedy *Chromosome*. Z tohoto vyplýva, že obsahuje funkciu *fitness()*, *reproduceFrom()* a *mutate()*. Fitness funkcia je viac etapová, ako už bolo zmienené v kapitole 6.2.1. Na začiatku sa nastaví gravitácia na malú hodnotu, prebehne 20 iterácií simulovania fyziky (funkcia *step()*). Po tejto časti je získaná kompletná deformácia mostu pomocou funkcie *getTotalDeformation()* (viz kapitola 6.3). Ak totálna deformácia neprekročila limit, tak fitness funkcia sa dostane do druhej etapy, pričom v jednom cykle sa postupne zvyšuje gravitácia. Pre každé zvýšenie gravitácie je vypočítaných päť krokov fyzickej simulácie. Tento cyklus beží, kým totálna deformácia neprekročí limit. Hodnota limitu sa rovná trojnásobkom počtov hrán v konštrukcii cesty. Ak zvislý parameter gravitácie je vyšší ako 1.5, tak funkcia postúpi do tretej etapy. V tejto etape je započítaná do úvahy celková dĺžka mostu (suma velikosti všetkých spojov).

Čo sa týka reprodukcie, je implementovaná metóda náhodného výberu hrán takou modifikáciou, kde hrany sú ocenené. Cenu hrany je možné určiť podľa veličiny preniknutia hrany. Preniknutie je v pomere záťaže, ktorú hrana prenáša. Dá sa povedať, čím má hrana väčšie preniknutie, tým je dôležitejšia. V prvom kroku u oboch rodičov vektor spojov je triedený podľa ceny spojov. Potom je vypočítaný koeficient náhodnosti, táto veličina je potrebná pri zabudovaní hrán do modelu potomka. Koeficient náhodnosti je možné vypočítať nasledovne:
(*fitness1* a *fitness2* sú fitness hodnoty rodičov)

$$N = 100 * \frac{fitness_1}{fitness_1 + fitness_2} \tag{7.2}$$

Do modelu potomka je kopírovaných toľko hrán, koľko je priemerný počet hrán rodičov. V cyklu *for* algoritmus prekopíruje spoje do potomka podľa jednej náhodnej hodnoty, ktorá je medzi 0 a 100. Ak táto hodnota je menšia ako koeficient náhodnosti (N), tak je prekopírovaná hrana z triedeného vektoru spojov prvého rodiča, ak nie tak z vektoru druhého rodiča. Výber hrany z vektoru je pomocou iterátoru vektora, po výbere iterátor preskočí na ďalšiu hranu, tzn. že hrany je možné kopírovať iba raz. Týmto krokom je dosiahnuté, aby sa najcennejšie spoje dostali do modelu potomka. To znamená, že existujú dominantné a recesívne hrany konštrukcie. Mutácia funguje jednoducho. Podľa parametru *ratio* sa uskutoční náhodná zmena v konštrukcii mostu. Toľko zmien je prevedených, aké hodnoty *ratio* nadobúda. Zmeny môžu byť nasledujúce:

1. Pridanie častice blízko k inej častici, a pritom nová častica bude spojená s tromi iným.
2. Pridanie spoja medzi dve častice.
3. Vymazanie častice, vymažú sa aj hrany, ktoré sú spojené s časticou.
4. Vymazanie spoja.
5. Zmena pozície.

Zmeny sa uskutočnia len medzi komponentami mostu, ktoré nie definované šablónou.

7.3 Implementácia riešenia knapsack problému

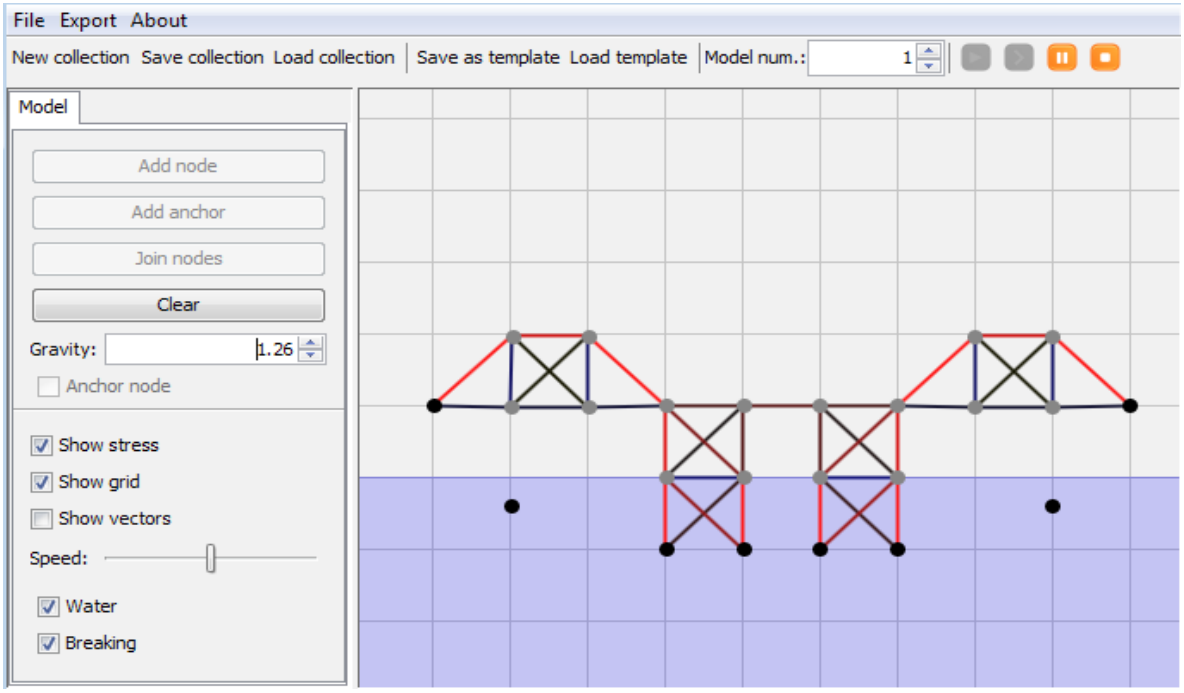
Aplikácia na riešenie knapsack problému je jednoduchá. Existujú tri triedy: *Thing*, *KTemplate*, knapsack. Trieda *Thing* reprezentuje objekty, ktoré sú v „izbe“. Trieda *Thing* má tri premenné: *wieght*, *value* a *inside.* Premenná *weight* je váha, *value* znamená cenu a *inside* je logická hodnota, či je v batohu. Trieda *KTemplate* slúži, ako šablóna pre problém. Podľa tejto šablóny sú inicializované jednotlivé jedince v populácie. Obsahuje vektor (C++ štandard vektor) objektov (*vector<Thing>*) a premennú na kapacitu batohu, ktorá je z typu *integer*. Tretia trieda je vlastná implementácia jedinca knapsack problému. Dedí od abstraktnej triedy *Chromosome,* takže implementuje funkciu *fitness()*, *mutate()*, *reproduceFrom()*, *serialize() a init()*. Obsahuje tiež vektor objektov a naviac statický objekt šablóny. Vo funkcii *init()*, ktorá je zavolaná na začiatku simulácie, vektor objektov sa vyplní podľa šablóny. Reprodukcia je implementovaná ako jednobodové kríženie (single point crossover). Mutácia je ovplyvnená parametrom *ratio*, toľko zmien je prevedených, aké hodnoty *ratio* nadobúda. Zmení sa len logická hodnota *inside*. Fitness hodnota sa vypočíta nasledovne :
V *for* cyklu sa vypočíta suma cien a suma váh, a v prípade prekročenia kapacity, je odčítaná zo sumy cien rozdielu kapacity a sumy váh.

7.4 Implementácia ostatných komponentov práce

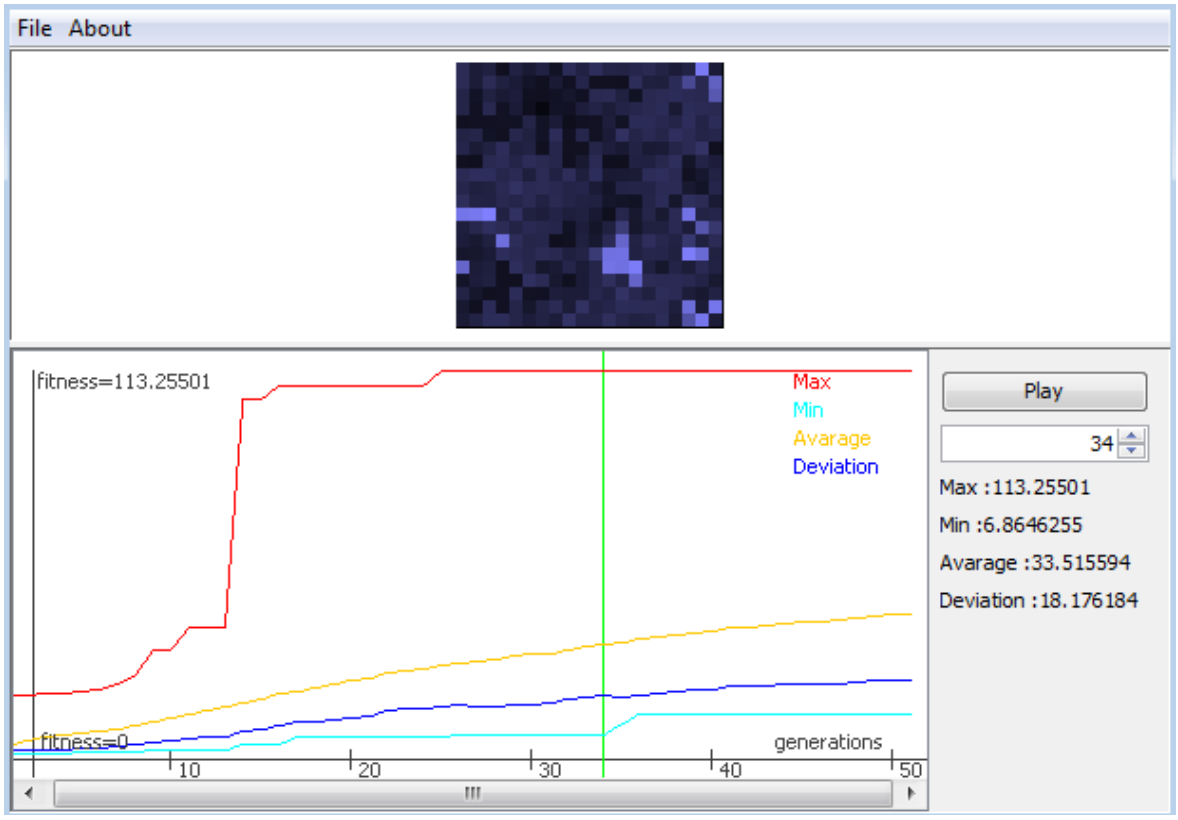
Ostatné komponenty práce sú *BridgeModeller* a *PopulationAnalyse*r. Tieto aplikácie sú používané na pozorovanie výsledkov simulácií. Sú napísané v jazyku Java.

BridgeModeller je program na sledovanie a testovanie modelov mostov. Tieto mosty sú načítané zo súboru, v ktorej je zakódovaná konštrukcia modelu s jednoduchým protokolom. V tomto programe je implementovaná príležitosť na vytvorenie a uloženie šablóny mostov, na ktoré je možné aplikovať program difúzneho evolučného algoritmu. *PopulationViewer* je program pomocou ktorého je možné sledovať vývin populácie. Tiež funguje na základe logovaných dát, takže sa dá používať po priebehu simulácie. V aplikácií je implementovaná grafika na zobrazenie populácie a grafu vývinu.

Graf vývinu obsahuje fitness hodnoty najlepšieho jedinca, najhoršieho, priemerný fitness, a smerodajnú odchýlku.



Obraz 7.1: BridgeModeller, aplikácia na modelovanie, testovanie a vytvorenie šablónou.



Obraz 7.1: PopulationViewer, program na analyzovanie vývinu populácie.

8 Výsledky testov

8.1 Výsledky testov problému knapsack

Tieto testy boli prevedené na počítače s parametrami : *Rýchlosť procesoru : **3.33 Ghz**, **4 jadra** s Intel® HT (Hyper Threading) technológiou, **8 MB** Intel® Smart Cache,**12GB** DDR3 **1066 MHz** pamäť* .

Prvý test porovnáva sekvenčnú a paralelnú variantu difúzneho evolučného algoritmu. Zvolený problém na testovanie je knapsack problém. Každý test obsahoval výpočet 50 generácií, na danom rozmeru populácie a daným počtom vlákien.

Pop.	**10 x 10**		**15 x 15**		**20 x 20**	
Vlákna	**Zrýchlenie**	**Efektivita**	**Zrýchlenie**	**Efektivita**	**Zrýchlenie**	**Efektivita**
1	**Čas : 0.0039sec**		**Čas : 0.014 sec**		**Čas :0.0593sec**	
2	0.4867x	11.74%	0.4698x	23.73%	0.4726x	23.08%
3	0.4645x	5.86%	0.4588x	13.22%	0.4012x	13.96%
4	0.0823x	2.84%	0.2171x	6.55%	0.2660x	7.23%
5	0.1278x	2.35%	0.2657x	3.22%	0.2173x	4.90%
6	0.0808x	1.74%	0.2429x	2.66%	0.2117x	2.63%
7	0.1999x	1.29%	0.0709x	1.25%	0.2233x	3.42%
8	0.0327x	0.68%	0.1131x	1.07%	0.1687x	1.53%
9	0.4255x	2.24%	0.6738x	3.75%	0.3025x	3.45%
10	0.4275x	2.22%	0.4751x	2.89%	0.2978x	3.16%

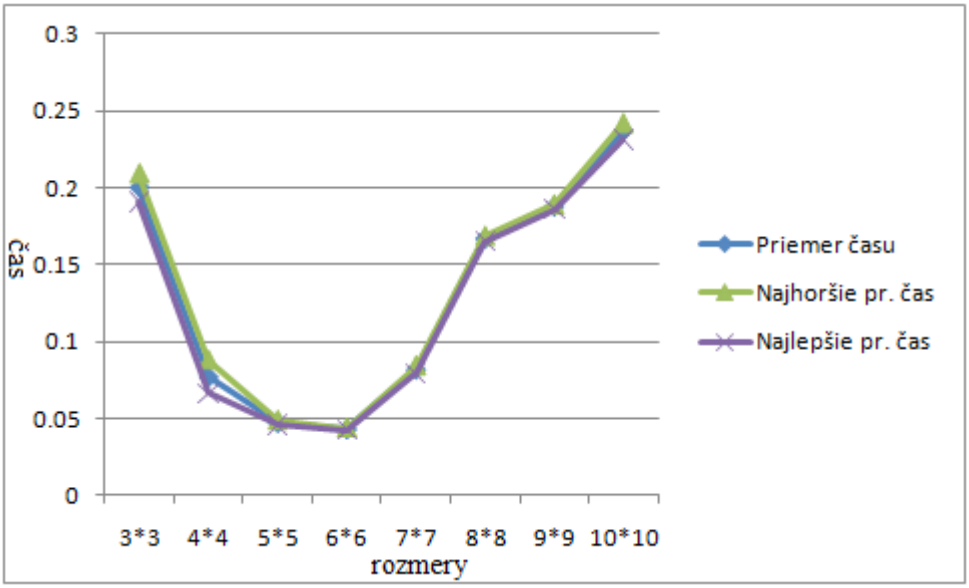
Tabuľka 8.1: Zrýchlenie a efektivita paralelného a sekvenčného algoritmu u problému knapsack na rozmery populácie 10x10, 15x15 a 20x20.

Podla výsledkov je možné konštatovať, že paralelné difúzne evolučné algoritmy nie sú príliš efektívne na výpočet knapsack problému. Dôvodom tohoto javu môže byť, že výpočet fitness funkcie je triviálny a pritom procesor neustále musí synchronizovať vlákna. Synchronizácia vlákna zoberie viac času ako výpočet fitness funkcie.

Druhý test sa snaží experimentálne nájsť najvýhodnejšie rozmery populácie, ktorým je možné najrýchlejšie dosiahnuť požadovanú fitness hodnotu. Požadovaná fitness bola 83 (priemerne veľká fitness na tento problém), metóda bola sekvenčná, a selekcia jedincov bola štyri-smerná.

Rozmery pop.	3x3	4x4	5x5	6x6	7x7	8x8	9x9	10x10
Priemer času	0.2002s	0.0770s	0.0472s	0.0431s	0.0818s	0.1669s	0.1877s	0.2368s
Interval spoľ.	0.0093	0.0108	0.0014	0.0006	0.0023	0.0017	0.0015	0.0054

Tabuľka 8.2 : Čas simulácie a interval spoľahlivosti času na základe rôznych rozmerov populácie pri štyri-smernej selekcie.

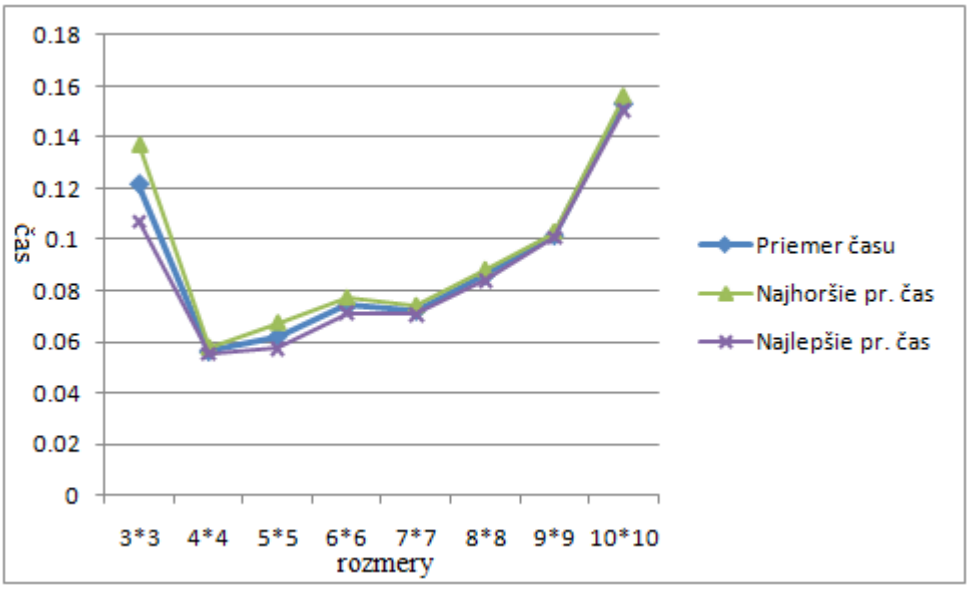


Obrázok 8.1 : Čas simulácie na základe rozmeru populácie s štyri-smernou selekciou. Horný a dolný limit sú hranice intervalu spoľahlivosti, ktorá ukazuje, že testované hodnoty sú 97.5% pravdepodobnosťou medzi tieto hranice.

Tretí test sa snaží experimentálně nájsť najvýhodnejšie rozmery populácie, ktorým je možné najrýchlejšie dosiahnúť požadovanú fitness hodnotu, tým rozdielom, že selekcia je osem-smerná. Požadovaná fitness bola opäť 83, metóda bola sekvenčná.

Rozmery pop.	3x3	4x4	5x5	6x6	7x7	8x8	9x9	10x10
Priemer času	0.1218s	0.0565s	0.0623s	0.0742s	0.0723s	0.0861s	0.1017s	0.1532s
Interval spoľ.	0.0149	0.0011	0.0049	0.0028	0.0016	0.0019	0.0010	0.0026

Tabuľka 8.3 : Čas simulácie a interval spoľahlivosti času na základe rôznych rozmerov populácie pri osem-smernej selekcie.



Obrázok 8.2 : Čas simulácie na základe rozmeru populácie, s osem-smernou selekciou. Horný a dolný limit sú hranice intervalu spoľahlivosti, ktorá ukazuje, že testované hodnoty sú 97.5% pravdepodobnosťou medzi tieto hranice.

Obidve testy boli prevedené na základe rovnakých vstupných hodnôt problému, tj. kapacita batohu bola vždy 100kg, počet objektov bol 50 a vlastnosti objektov (váha a cena) boli vždy rovnaké (nebola používaná funkcia *srand*). Aby boli dosiahnuté najpresnejšie výsledky, pre všetky rozmery boli prevedený 10 testov (celkom 200 testov). Smerodajná odchýlka ukazuje, že rozdiel medzi testami je zanedbateľný. Podľa testov si môžeme tvrdiť, že pre štyri-smernú selekciu najvýhodnejšie rozmery populácie sú 4x4, 5x5, 6x6. Ďalej je možné konštatovať, že efektivita osem-smernej selekcie sa len mierne líši od štyri-smernej, najväčší rozdiel je pri rozsiahlejších populáciách, tj. 9x9 a 10x10, kde osem-smerná selekcia je výhodnejšia.

Podrobný výsledok testu problému knapsack sa nachádza v prílohe B.

8.2 Výsledky testov problému mostnej konštrukcie.

Prvým testom na návrh mostnej konštrukcie bolo opäť meranie zrýchlenie výpočtov, porovnanie sekvenčnú a paralelnú variantu algoritmy. Každý test obsahoval výpočet 5 generácií, na danom rozmeru populácie a s daným počtom vlákien.

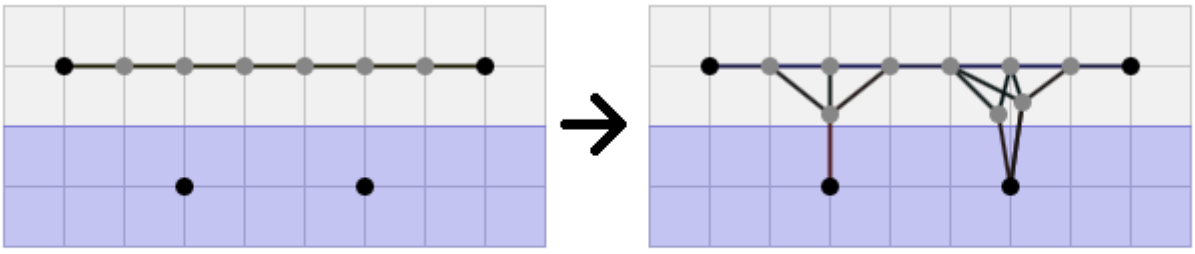
Pop.	10 x 10		15 x 15		20 x 20	
Vlákna	Zrýchlenie	Efektivita	Zrýchlenie	Efektivita	Zrýchlenie	Efektivita
1	Čas :3.257sec		Čas :7.495sec			Čas :14.24sec
2	1.4846x	74.23%	1.6700x	83.50%	1.6989x	84.95%
3	0.9294x	30.98%	1.9766x	65.89%	2.2610x	75.37%
4	2.3370x	58.42%	3.2652x	81.63%	2.6847x	67.12%
5	2.1346x	42.69%	2.5112x	50.22%	2.2149x	44.30%
6	3.0053x	50.09%	2.6130x	43.55%	3.7136x	61.89%
7	2.6594x	37.99%	3.4985x	49.98%	3.7704x	53.86%
8	3.0270x	37.84%	2.3764x	29.70%	3.1851x	39.81%
9	4.4837x	49.82%	4.1981x	46.65%	3.6546x	40.61%
10	2.9488x	29.49%	4.5394x	45.39%	4.1146x	41.15%

Tabuľka 8.4 : Zrýchlenie a efektivita paralelného a sekvenčného algoritmu u problému mostnej konštrukcie na rozmery populácie 10x10, 15x15 a 20x20.

Podla výsledkov je možné tvrdiť, že problémy so náročnejšou fitness funkciou ukázajú lepšiu efektivitu pri používaniu viacerých vlákien. Podla hodnoty zrýchlenia, pri riešeniu problému mostnej konštrukcie je najlepšie používať 8, 9 alebo 10 vlákna.

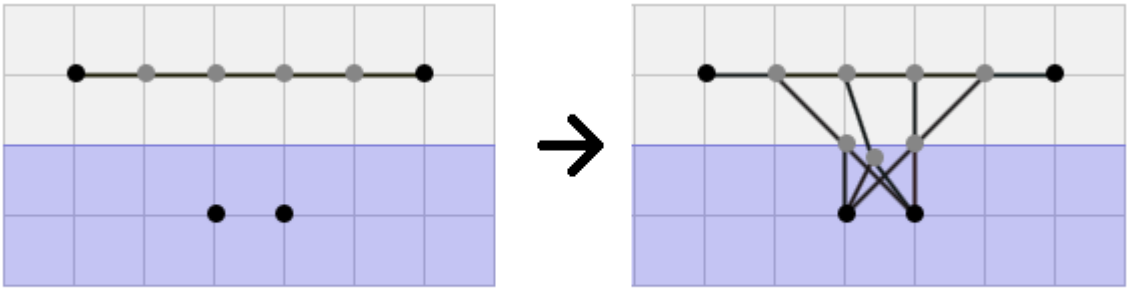
Druhým testom je test schopnosti plánovania difúznych evolučných algoritmov. Testy boli prevedené na jednoduché šablóny mostov.

Prvý návrh - pracovali ôsmi vlákna, rozmer populácie bolo 20x20, požadovaná fitness bola 70, dosiahnuť tento výsledok trvalo 128 sekúnd (70 generácií). Výsledná fitness hodnota je 97.09 . V prílohe B sa nachádza podrobný vývin tohoto návrhu.



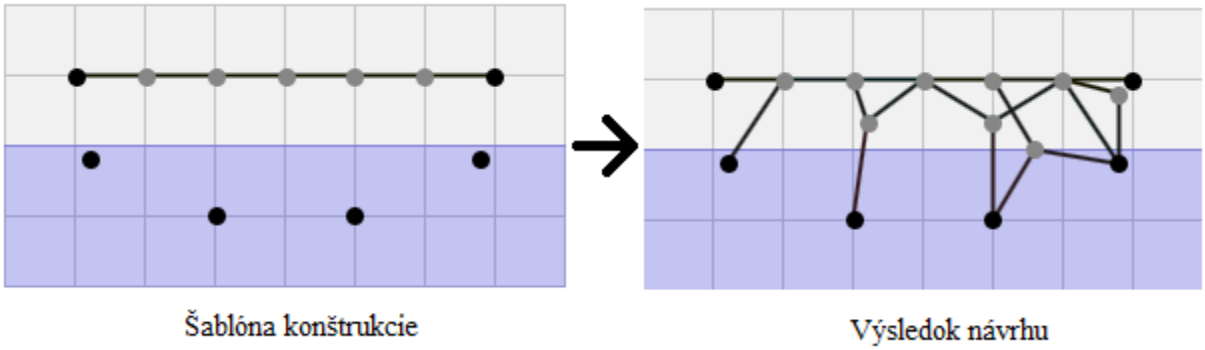
Obrázok 8.3 : Návrh mostnej konštrukcie difúznych evolučných algoritmov.

Druhý návrh - pracovali ôsmi vlákna, rozmer populácie bolo 20x20, dosiahnuť tento výsledok trvalo 153 sekúnd (50 generácií). Výsledná fitness hodnota je 118.458 . V prílohe B sa nachádza podrobný vývin tohoto návrhu.



Obrázok 8.4 : Návrh mostnej konštrukcie difúznych evolučných algoritmov.

Tretí návrh - pracovali ôsmi vlákna, rozmer populácie bolo 20x20, dosiahnuť tento výsledok trvalo 171 sekúnd (64 generácií). Výsledná fitness hodnota je 114.407 .



Obrázok 8.5 : Návrh mostnej konštrukcie difúznych evolučných algoritmov.

9 Záver

Ako už bolo zmienené, cieľom tejto práce bolo experimentálne nájsť najvýhodnejšie parametri difúznych evolučných algoritmov. Porovnanie časovej náročnosti paralelných a sekvenčných evolučných algoritmov. Ako nad rámec vypracovaná práca bola testovanie schopnosti návrhu mostnej konštrukcie difúznych evolučných algoritmov.

Čo sa týka porovnania paralelného a sekvenčného algoritmu pri riešeniu knapsack problému, výsledky sú prekvapujúce. Podľa testov trvali paralelné simulácie dlhšie ako sekvenčné. Vysvetlenie tohoto javu môže byť, že procesor potrebuje čas na synchronizovanie vlákien, tento čas je dlhší ako výpočet skupiny jedincov, ktorý sú priradený k jednému vláknu. Inak povedané, čas potrebný na synchronizáciu vlákien je dlhší, ako čas na výpočet fitness hodnôt jedincov. Z toho vyplýva, že paralelné evolučné algoritmy nie sú výhodné na riešenie problémov, ktorých fitness funkcia je triviálna, lepšie je používanie sekvenčných algoritmov.

Podľa testov môžeme povedať, že návrh mostných konštrukcií pomocou difúznych evolučných algoritmov je pomerne kvalitná. Kvalita konštrukcie záleží na zložitosti šablóny mostnej konštrukcie. Tieto simulácie boli časovo oveľa náročnejšie ako simulácie na riešenie knapsack problému. Paralelné evolučné algoritmy sú užitočné pre riešenie návrhových problémov. Výsledky potvrdzujú, že čím viac vlákien (do rozumnej miery) pracuje na výpočte jedincov, tým je celkový výpočet rýchlejší. Čas potrebný na synchronizáciu procesov je v tomto prípade zanedbateľný.

Čo sa týka ďalšieho vývoja aplikácií, v prvom rade by som zmienil, že k ďalším testom by bolo dobré implementovať farmársky model paralelných evolučných algoritmov. Dá sa predpokladať, že tento model by vykazoval lepšie výsledky ohľadom na čas simulácie. Ďalšie vylepšenie by sa mohli uskutočniť u fitness funkcie problému mostnej konštrukcie. Navrhnutie tejto funkcie bolo relatívne náročné. Jej prípadné vylepšenie by sa malo zamerať na vylúčenie zbytočných komponentov mostnej konštrukcie. Inak povedané, nedostatky sú u heuristickej časti tejto funkcie. Ako posledné dôležité vylepšenie by bolo pridanie novej funkcie do knihovne *Evolution,* ktorá by umožnila uloženie populácie. Táto funkcia by nám udávala možnosť na zastavenie a obnovenie behu programu, tým pádom by jedna simulácie nebola súvislá, ale dalo by sa previesť vo viacerých krokoch.

Na posledy by som zmienil, že táto práca obsahuje nad rámec vypracované časti. Tieto sú návrh mostnej konštrukcie pomocou difúznych evolučných algoritmov, program BridgeModeller, ktorým sa dá pozorovať výsledky mostných konštrukcií a program je PopulationViewer, ktorý slúži na pozorovanie vývinu a vyrobenie štatistiky o populácie.

Literatúra

[1] V. Kvasnička, J. Pospíchal, P. Tiňo: Evolučné Algoritmy, STU Bratislava, Bratislava, 2000. ISBN 80-227-1377-5

[2] I. Zelinka, Z. Oplatková, M. Šeda, P. Ošmera, F. Včelář: Evoluční výpočetní techniky - Princípy a aplikace, BEN technická literatúra, Praha, 2009. ISBN 978-80-7300-218-3

[3] WWW stránky. Fitness (Biology).
URL: <http://en.wikipedia.org/wiki/Fitness_%28biology%29>

[4] WWW stránky. Adaptive Fuzzy Fitness Granulation (AFFG).
URL: <http://www.davarynejad.com/Mohsen/index.php?n=Main.AdaptiveFuzzyFitnessGranulation>

[5] V. Mařík, O. Štěpániková, Jiří Lažanský: Umělá Inteligence (3), Academia, Praha, 2003. ISBN 80-200-0472-6

[6] WWW stránky. Cut and splice crossover.
URL: <http://en.wikipedia.org/wiki/Crossover_%28genetic_algorithm%29>

[7] WWW stránky. Partially mapped crossover.
URL: <http://students.uta.edu/bx/bxk7163/tsp/tsp.html>

[8] WWW stránky. Crossover - Genetic Algorithms.
URL: <http://en.wikipedia.org/wiki/Crossover_%28genetic_algorithm%29>

[9] J. R. Koza: Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT press, Cambridge, 1998. ISBN 0-262-11170-5

[10] Online dokument ‚Parallelism and Evolutionary Algorithms,
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.64.430&rep=rep1&type=pdf

[11] WWW stránky. Parallel computation.
URL: <http://en.wikipedia.org/wiki/Non-Uniform_Memory_Access>

[12] WWW stránky. Cellular Evolutionary Algorithms.
URL: <http://neo.lcc.uma.es/cEA-web/index.htm>

[13] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, J. McDonald : Parallel Programing In OpenMP, Morgan Kaufmann, San Diego, 2001. ISBN 978-1-55860-671-5

[14] WWW stránky. OpenMp manual. URL: <https://computing.llnl.gov/tutorials/openMP/>

[15] WWW stránky - Wikipedia. Knapsack Problem.
URL: <http://en.wikipedia.org/wiki/Knapsack_problem>

[16] J.Jíři: Online dokument. Fitness funkcia s penalizáciou. URL: <wis.fit.vutbr.cz>,
Predmet: Aplikované evolučné algoritmy, Cvičenie 2.

[17] I. Mailington: Game Physics Engine Development, Morgan Kaufmann, San Francisco, 2007. ISBN 978-0123694713

Zoznam príloh :

- Príloha A – Návod na použitie
- Príloha B – Obrázkové výsledky testov mostnej konštrukcie a ukážka vývinu populácie.
- Príloha C – UML diagram tried knihovni *Evolution*

Príloha A

Návod na použitie

Tento dokument obsahuje návody na použitie praktickej časti bakalárskej práce Difúzne Evolučné Algoritmy. Praktická časť práce sa skladá z piatich častí:

1. Knihovňa *Evolution*
2. Program na plánovanie mostnej konštrukcie
3. Program na riešenie *knapsack* problému
4. *BridgeModeller*
5. *PopulationViewer*

Sú k dispozícií dva testy, ktoré sa dajú previesť pomocou týchto aplikácií :

1. Evolučné plánovanie mostnej konštrukcie
2. Riešenie knapsack problému pomocou difúzneho evolučného algoritmu

Evolučné plánovanie mostnej konštrukcie

Plánovanie sa začína s vytvorením šablóny, tj. cesty, mostnej konštrukcie. Hlavnou úlohou tohto príkladu je stabilizovanie tejto cesty. Šablóny sa vytvoria pomocou programu *BridgeModeller.*

Program bol napísaný v jazyku Java pomocou vývojového prostredia *Netbeans IDE*. Súbor *Jar* sa nachádza v zložke *dist*. Aplikácia je spustiteľná pomocou príkazu:

```
java -jar dist/bridgemodeller.jar
```

(pozn. BridgeModeller a PopulationViewer sú ANT projekty, prekladať sa dajú pomocou príkazu *ant build*)

Existujú dva typy súborov :

1. Súbory s rozšírením „ .tmp“, ktoré obsahujú šablóny. Tieto súbory obsahujú iba jeden model mostnej konštrukcie.
2. Súbory s rozšírením „ .col“, ktoré obsahujú výsledky difúzneho evolučného algoritmu. Tieto súbory obsahujú viaceré modely mostnej konštrukcie.

Vytvorenie šablóny

Po spustení programu do priestoru fyzického modelu sa dajú pridať uzly a hrany. Na tento účel sú k dispozícií tlačidlá:

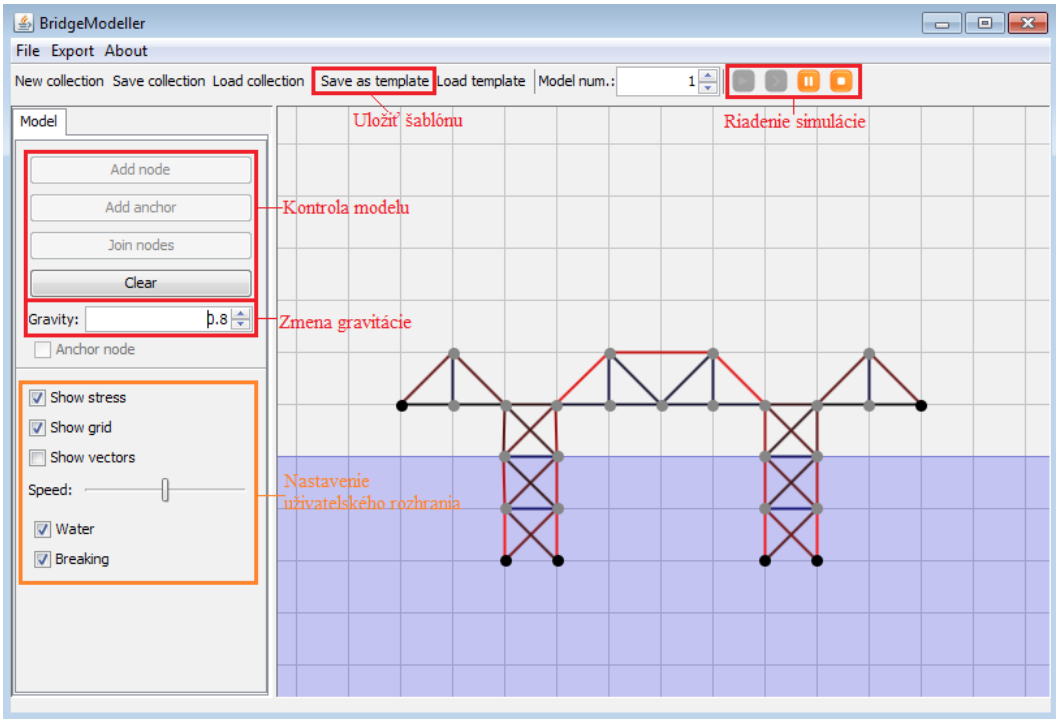
1. *Add Body* , ktoré slúži na pridanie uzla do fyzického priestoru.
2. *Add anchor* , ktoré slúži na pridanie pevného uzla.
3. *Connect* , ktoré slúži na spojenie dvoch uzlov.

Vyznačiť body je možné pomocou kliknutia na nich. Vyznačené body sa dajú premiestniť a vymazať. Vymazanie je možné pomocou pravého tlačítka myši. Premiestniť sa dajú pomocou „drag and drop“.

Uloženie šablóny pomocou tlačidla *Save as template.*

Simulácia pomocou BridgeModeller.

Simulácia je spustiteľná pomocou tlačítka *play*, ktoré je na hornej pravej strane okna aplikácie. Ostatné tlačítka v tejto priestore tiež slúžia na riadenie simulácie. Počas simulácie je možné zmeniť hodnota gravitácie pomocou komponentu *Gravity*. S potiahnutím uzlov vo fyzickom priestore sa interaktívne dá testovať konštrukcia.



Obraz 1: Obrázok ukazuje užívateľské rozhranie programu *BridgeModeller*.

Plánovanie mostnej konštrukcie

Pre plánovanie mostnej konštrukcie pomocou evolučného algoritmu je treba prekladať zdrojoví kód programu. Prekladať je možné pomocou príkazu :

make bridge

Tento príkaz prekladá zdrojoví kód do spustiteľného súboru : *bridge*

Spustenie simulácie

Spustiť simuláciu je možné nasledovne :

./bridge [arguments]

Argumenty simulácie :

1. ***-s rozmery* :** nastaví veľkosť populácie, hodnota *rozmery* znamená výšku aj šírku populácie. Napr. ak *rozmery* je 20 tak populácie bude mať veľkosť 20 x 20, tj. 400 jedincov.
2. ***-w :*** ak je nastavená, tak selekcia bude osem smerná.
3. ***-q :*** znamená sekvenčnú simuláciu, tj. pre nastavenie tohoto argumentu, simulácia prebehne pomocou jedného vlákna.
4. ***-t vlákna :*** nastavenie počtu vlákien. Tento argument má zmysel len vtedy, keď atribút ***-q*** nie je nastavený. Hodnota vlákna môže byť od 1 až do počtu vlákien, ktoré procesor podporuje.
5. ***-g generácie*** : nastavenie požadovaných počtu generácií.
6. ***-r fitness :*** nastavenie požadovanej fitness hodnoty.
7. ***-f log_súbor :*** nastavenie súboru na logovanie. Ak táto hodnota nie je nastavená vývinu populácie, nebude vypísaná a pozorovanie pomocou PopulationViewer nebude možné.

Príklad spustenia :

./bridge -s 20 -t 4 -r 70 -g 200 -w -f log.evo

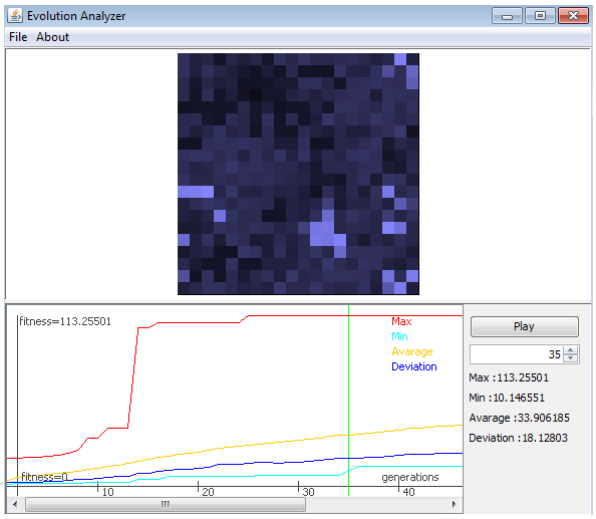
V tomto prípade populácia bude obsahovať 400 jedincov, štyri vlákna budú pracovať, požadovaná fitness hodnota je 70, počet generácií bude 200, selekcia bude osem smerná a vývin populácie bude logovaný do súboru *log.evo.*

Šablóna mostu bude permanentne načítaná zo súboru *bridge.tmp* . Tento súbor musí byť v tom istom adresári, v ktorom je spustiteľný súbor *bridge.*
Výsledky mostov budú permanentne ukladané do súboru *results.col.*

Pozorovanie výsledkov

Pomocou BridgeModeller (Open collection) je možné otvoriť súbor *results.col.* Program načíta všetky mosty, ktoré boli uložené do tohto súboru. Vyhľadávať mostové konštrukcie sa dajú pomocou komponentu *Model num,* ktorá je v hornej časti okna.

Vývin populácie je pozorovateľný pomocou programu PopulationViewer. Pomocou tlačidla *Open* sa načítavajú súbory, ktoré obsahujú dáta o vývinu. Tieto súbory musia byť ukončené príponou „ .evo“.
Po načítaní je možné pozorovať vývin populácie.



Obraz 2: Obrázok ukazuje užívateľské rozhranie programu *PopulationViewer*.

PopulationViewer sa dá spustiť s príkazom:

java -jar dist/PopulationViewer.jar

Riešenie knapsack problému

Pre problém *knapsack* neexistuje šablóna. Všetky parametre sú nastaviteľné pomocou argumentu aplikácie. Pred spustením je potrebné prekladať zdrojoví kód, toto sa dá uskutočniť pomocou nasledujúceho príkazu :

make knapsack

Príkaz *make* vytvorí spustiteľný súbor, ktorý má názov *knapsack*. Spustiť simuláciu je možné nasledovne:

./knapsack [arguments]

Argumenty sú takmer totožné, ako u simulácie modelu mostnej konštrukcie. Pre knapsack problém existujú ešte dva argumenty :

1. **-y kapacita :** nastaví kapacitu batohu zlodeja.
2. **-c počet_objektov :** nastaví počet objektov v izbe.

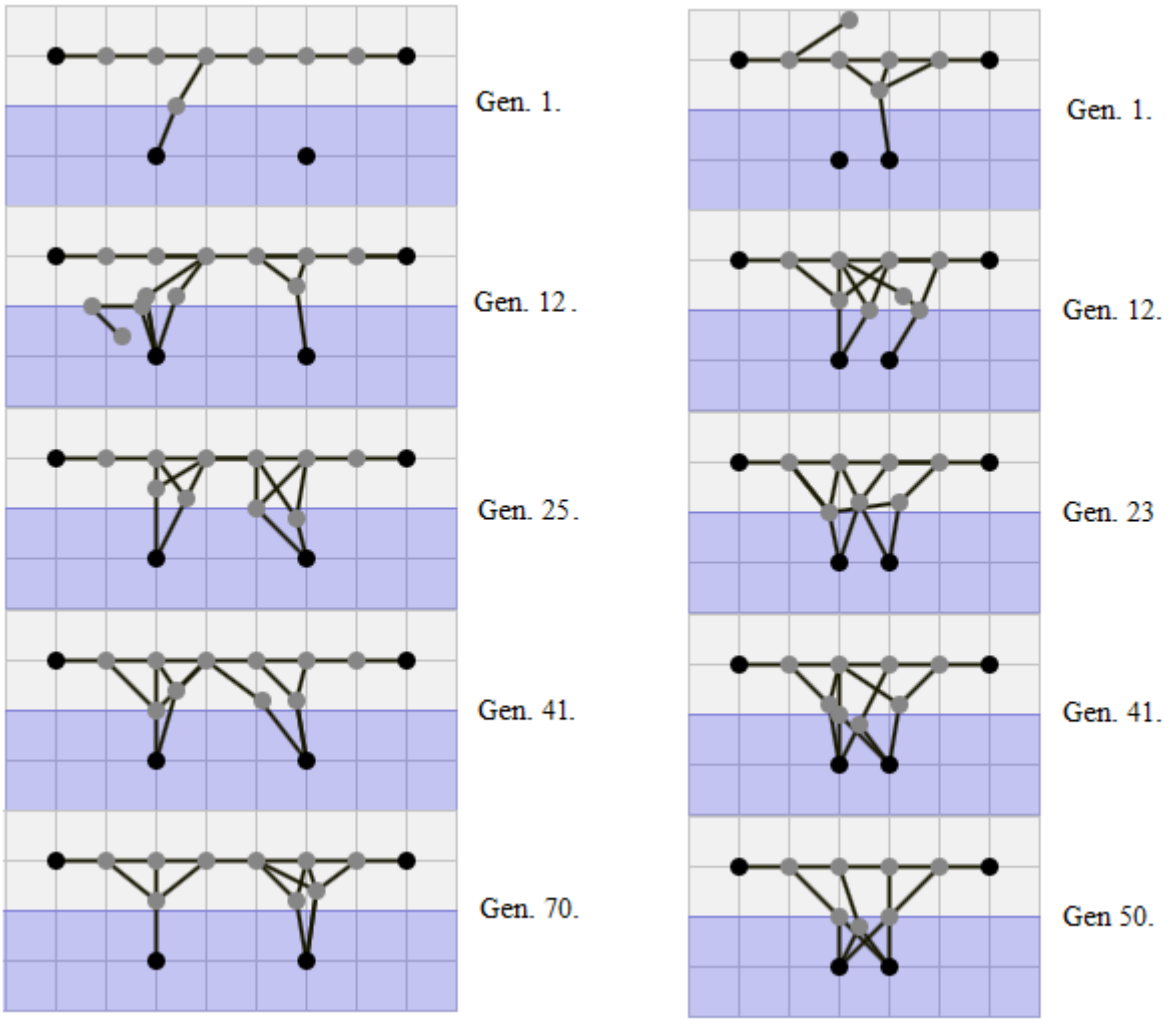
Podľa tých dvoch argumentov prevedie sa optimalizácia knapsack problému. Cena a váha všetkých objektov bude vygenerovaná náhodne. Kedže aplikácia používa rovnaké jadro ako u simulácie mostnej konštrukcie logovanie funguje podobne. Výsledky simulácie sú vždy uložené do súboru *results.txt*.

Príklad spustenia :

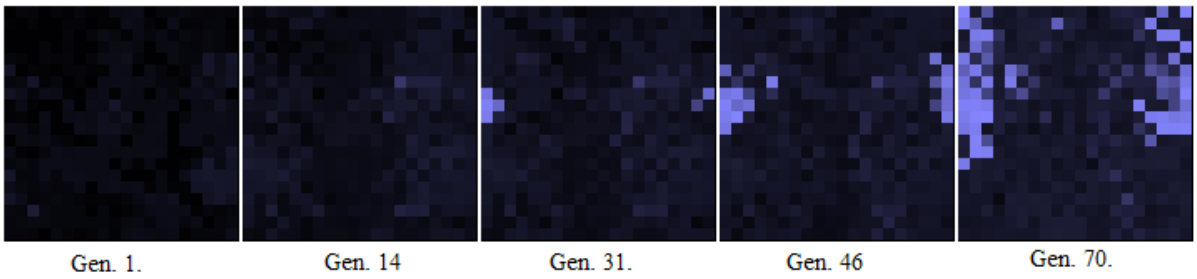
./knapsack -s 30 -g 300 -f log.evo -c 20 -y 60 -w

Príloha B

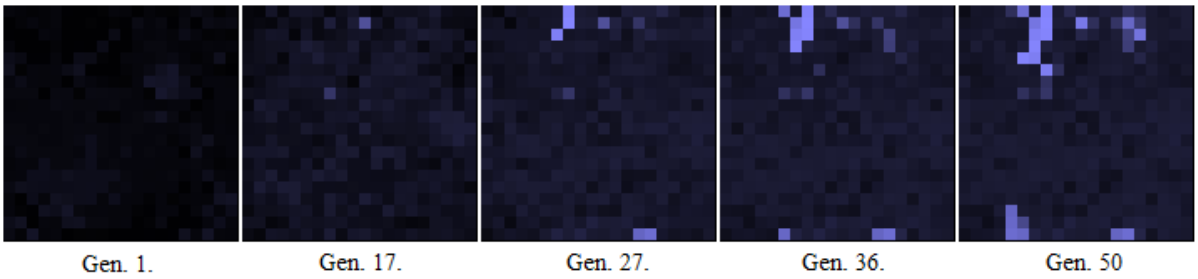
Obrázkové výsledky testov mostnej konštrukcie a ukážka vývinu populácie.



Obrázky ukázajú vývin mostných konštrukcií. Ľavý je vývin mostu prvého testu (viz. Kap. 8.2), Pravý je vývin druhého mostu (viz. Kap. 8.2).



Obrázok ukazuje vývin populácie prvého testu (viz. Kap. 8.2).



Obrázok ukazuje vývin populácie druhého testu (viz. Kap. 8.2).

Príklad riešeniu knapsack problému

Všetky objekty (50 kus, celkom 699kg, a 245$) :

(6$,13kg) (5$,27kg) (5$,23kg) (2$,16kg) (1$,9kg) (7$,2kg) (9$,20kg) (6$,23kg) (6$,0kg) (6$,22kg) (8$,11kg) (9$,27kg) (0$,2kg) (3$,2kg) (5$,7kg) (2$,29kg) (8$,12kg) (7$,29kg) (6$,13kg) (2$,1kg) (3$,9kg) (9$,21kg) (7$,14kg) (4$,8kg) (0$,5kg) (6$,23kg) (0$,1kg) (3$,26kg) (0$,2kg) (1$,16kg) (5$,15kg) (7$,14kg) (5$,26kg) (9$,16kg) (7$,13kg) (5$,24kg) (5$,12kg) (7$,14kg) (4$,14kg) (0$,3kg) (8$,7kg) (8$,6kg) (4$,8kg) (1$,3kg) (9$,14kg) (0$,12kg) (8$,26kg) (2$,19kg) (6$,16kg) (9$,24kg)

Výsledok :

využívaná váha :100/100, celková cena: 82

(7$,2kg) (6$,0kg) (8$,11kg) (0$,2kg) (3$,2kg) (5$,7kg) (8$,12kg) (2$,1kg) (4$,8kg) (0$,1kg) (7$,13kg) (7$,14kg) (8$,7kg) (8$,6kg) (9$,14kg)

Príloha C

UML diagram tried knihovni *Evolution*