ZÁPADOČESKÁ UNIVERZITA V PLZNI
FAKULTA STROJNÍ

BAKALÁŘSKÁ PRÁCE

2016/2017 Monika Knolová

ZÁPADOČESKÁ UNIVERZITA V PLZNI
FAKULTA STROJNÍ

Studijní program: B 2301 Strojní inženýrství
Studijní zaměření: Průmyslové inženýrství a management

BAKALÁŘSKÁ PRÁCE

Memetický algoritmus využitelný v rámci diskrétní simulační optimalizace

Autor: **Monika KNOLOVÁ**

Vedoucí práce: **Ing. Pavel RAŠKA, Ph.D.**

Akademický rok 2016/2017

Prohlášení o autorství

Překládám tímto k posouzení a obhajobě bakalářskou práci, zpracovanou na závěr studia na Fakultě strojní Západočeské univerzity v Plzni.
Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně, s použitím odborné literatury a pramenů, uvedených v seznamu, který je součástí této bakalářské práce.

V Plzni dne:……. …….....

Podpis autora

Poděkování

Ráda bych tímto poděkovala vedoucímu mé bakalářské práce Ing. Pavlu Raškovi, PhD. za odbornou pomoc a výborný přístup. Dále bych ráda poděkovala své rodině a příteli za jejich podporu.

ANOTAČNÍ LIST BAKALÁŘSKÉ PRÁCE

AUTOR	Knolová	Monika
STUDIJNÍ OBOR	Průmyslové inženýrství a management	
VEDOUCÍ PRÁCE	Ing. Raška, Ph.D.	Pavel
PRACOVIŠTĚ	ZČU - FST - KPV	

DRUH PRÁCE	~~**DIPLOMOVÁ**~~	**BAKALÁŘSKÁ**	**Nehodící se škrtněte**
NÁZEV PRÁCE	Memetický algoritmus využitelný v rámci diskrétní simulační optimalizace		

FAKULTA	strojní		**KATEDRA**	KPV	**ROK ODEVZD.**	2017

POČET STRAN (A4 a ekvivalentů A4)

CELKEM	41		**TEXTOVÁ ČÁST**	36	**GRAFICKÁ ČÁST**	5

STRUČNÝ POPIS (MAX 10 ŘÁDEK) **ZAMĚŘENÍ, TÉMA, CÍL POZNATKY A PŘÍNOSY**	Tato práce je zaměřena na memetické algoritmy. Memetické algoritmy jsou jednou z optimalizačních metod využitelných v rámci diskrétní simulační optimalizace. V práci jsou popsány jednotlivé části memetického algoritmu. V poslední části práce je ukázka navrženého memetického algoritmu pro řešení job shop scheduling problému.
KLÍČOVÁ SLOVA	Optimalizace, diskrétní simulace, memetický algoritmus, genetické algoritmy, fitness, pseudogradientní metody, Job Shop Scheduling problem

SUMMARY OF BACHELOR SHEET

AUTHOR	Knolová	Monika	
FIELD OF STUDY	Industrial Engineering and Management		
SUPERVISOR	Ing. Raška, Ph.D.	Pavel	
INSTITUTION	ZČU - FST - KKS		
TYPE OF WORK	~~**DIPLOMA**~~	**BACHELOR**	**Delete when not applicable**
TITLE OF THE WORK	Memetic algorithm Used for Discrete Event Simulation Optimization		

FACULTY	Mechanical Engineering		**DEPARTMENT**	KPV	**SUBMITTED IN**	2017

NUMBER OF PAGES (A4 and eq. A4)

TOTALLY	41		**TEXT PART**	36	**GRAPHICAL PART**	5

BRIEF DESCRIPTION **TOPIC, GOAL, RESULTS AND CONTRIBUTIONS**	This theses is focused on Memetic algorithms. Memetic algorithms are one of the optimization methods used for descrete event simulation optimization. In theses are described all parts of Memtic alorithms. At the end of the theses, there is showen designed Memetic algorithm for solving the job shop Scheduling problem.
KEY WORDS	Optimization, Descrete Event Simulation, Memtic algorithm, Genetic algorithms, fitness, pseudogradient methods, Job Shop Scheduling problem

Obsah

Seznam obrázků:

Seznam algoritmů:

Použité značení

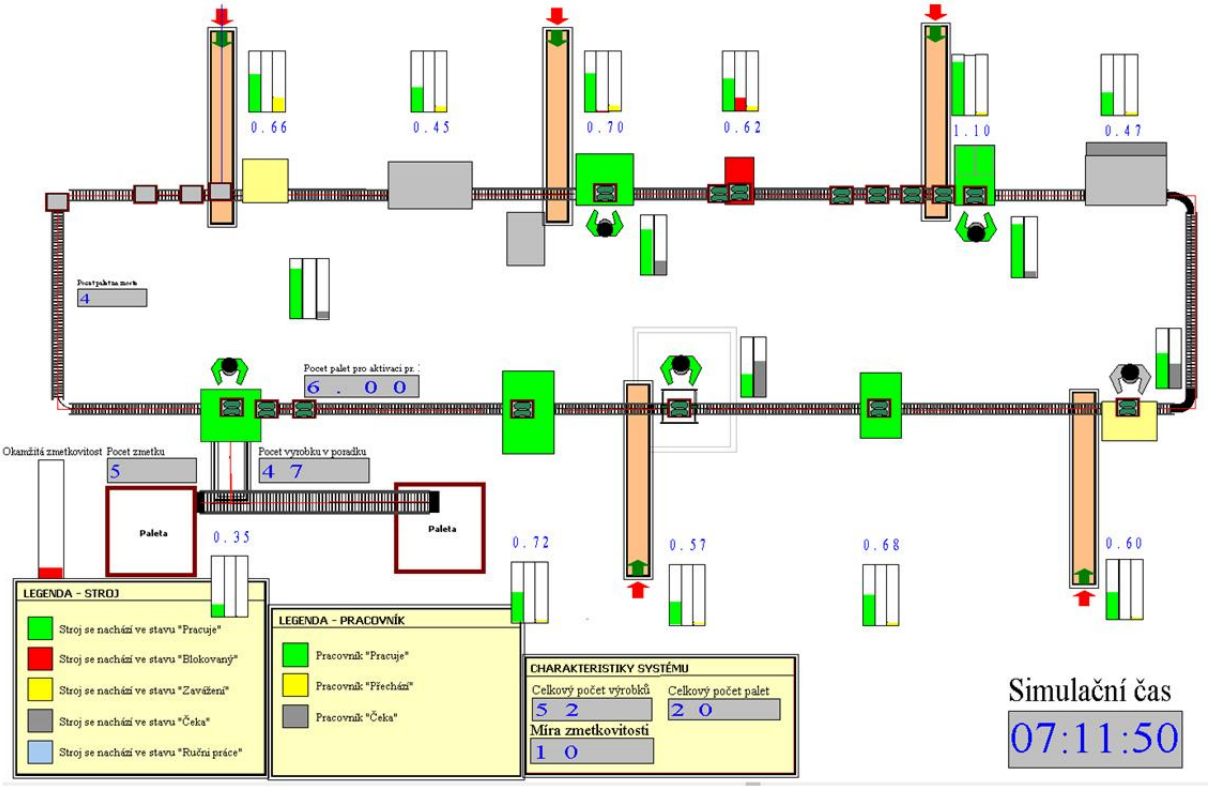
Označení	Popis
X	Jedinec ($P[i]$)
$X.sour$	Souřadnice jedince
$X.gen$	Gen jedince
$f(X.sour)$	Hodnota fitness jedince
X_{best}	Nejlepší jedinec z populace
P	Počáteční populace (seznam jedinců)
P_r	Populace určená k reprodukci
P_k	Populace vzniklá křížením
n	Počet jedinců v populaci P
n_r	Počet jedinců v populaci P_r
n_k	Počet jedinců v populaci P_k
dim	Počet dimenzí
t	Počet iterací
t_{max}	Maximální počet iterací
i	Značí pořadí jedince v populaci
j	Značí pořadí dimenze
k	Značí pořadí alely v genu

1 Úvod

Každý z nás se jistě už setkal s tím, že musel vyřešit nějaký problém. Pokud se jedná o problém, který má více než jedno řešení, jak ale poznat, které řešení je to nejlepší? Nejjednodušším způsobem je zkusit nastínit variantu a zjistit, co se stane. Představte si ale, že je řešen nějaký větší problém. Například rozmístění pracovišť (strojů) v podniku. Počet řešení je závislý na počtu strojů, a čím více strojů máme k dispozici, tím se samozřejmě nabízí více možností, jak je uspořádat. A bylo by asi hloupé, všechny možné varianty zkoušet, stěhovat stroje sem a tam a vždy čekat, abychom zjistili, zda je tato varianta rozmístění vhodná nebo nikoliv. A pokud bychom nalezli lepší řešení, než bylo to současné, měli bychom toto řešení ponechat, anebo zkoušet další varianty a doufat, že najdeme ještě lepší řešení? Nejlepší by bylo si zkusit danou variantu možného řešení nanečisto. A k tomu slouží právě simulace.

Určitá reálná situace by se přenesla do virtuální reality. Reálná situace může být popsána pomocí předpisu funkce. Nastaví se vstupní parametry a dále se jen zkoumá nejlepší kombinace těchto vstupních parametrů, za účelem získání optimálního řešení. Z předchozí věty, že se „jen“ zkoumá nejlepší kombinace, to zní jako hračka. Realita je ovšem jiná. Těchto možných kombinací je u většiny případů z praxe tolik, že není možné v nějakém rozumném čase všechny nalézt a porovnat mezi sebou.

Pro bližší představu je vložen obrázek, jak může například vypadat simulační model. Na obrázku je znázorněn simulační model výrobní linky.



Obrázek 1-1 Ukázka simulačního modelu výrobní linky [1]

2 Diskrétní simulace

Pomocí simulace můžeme skutečné objekty nahradit tzv. modely. Tyto modely by měli být co nejvíce podobné zkoumaným objektům. Měli by tedy mít všechny vlastnosti, které nás zajímají. Stejně jako objekty, můžeme modelovat i děje, ke kterým mezi objekty dochází. Na takovém modelu celé situace můžeme pak testovat různé varianty.

Simulace rozdělujeme podle jejich průběhu v čase. Pokud je čas zvyšován po malých, ale stejně velkých krocích, jedná se o *spojitou simulaci*. Opakem spojité simulace je *diskrétní simulace,* kdy sledujeme model jen v takových časech, kdy se něco děje. To, co se děje, označme jako *událost*. Skvělým příkladem je simulace přepravy vlakem, kdy přijedete na nádraží, koupíte si jízdenku, nastoupíte do vlaku, pak nějakou dobu jedete a nakonec vystoupíte. To všechno jsou události, mezi kterými jsou různě dlouhé časové intervaly. A jestli si dáte cigaretu předtím, než nastoupíte do vlaku a ve vlaku budete spát anebo si číst knížku, to jsou pro diskrétní simulaci nepodstatné věci.

Představme si, že čas uvnitř simulace ubíhá po časové přímce. Tento čas uvnitř simulace se nazývá *simulární čas*. Události, které zde probíhají, vyznačíme na přímce pomocí bodů. Všechny tyto události je třeba projít a zpracovat a to v takovém pořadí, v jaké jsou rozmístěny na časové přímce. Několik událostí, které se dějí na začátku, jsme naplánovali ještě před spuštěním simulace a další události vznikly v důsledku předešlých událostí. Postupným procházením a zpracováváním se o těchto událostech něco dozvídáme, a události, o kterých už víme, si můžeme udržovat ve frontě, seřazené podle času. Tuto frontu nazýváme *kalendář událostí.* A v poslední řadě objekty, které ovlivňují simulaci a které můžeme naplánovat, se nazývají *procesy*. Takovým procesem může být třeba stroj, člověk apod. [2]

Dalšími příklady pro využití diskrétní simulace mohou být třeba modely hromadné obsluhy (supermarkety, pošty, lékárny). Zkoumá se, jaký je optimální počet pokladních, aby se netvořily fronty. Dále třeba modely zásob, které zkoumají procesy, kde se mění velikost poptávky, kde intervaly mezi poptávkami jsou nestálé, kde zásoby jsou pořizovány od různých dodavatelů v různých intervalech apod. Cílem je většinou zjistit, kolik a jak často objednávat, aby byly náklady spojené se zásobováním co nejmenší. Dále třeba rozvrhování výroby, jak zpracovávat výrobky a v jakém pořadí. [3]

3 Optimalizační úlohy

Nyní, když už víme, co je to simulace, se pojďme zaměřit na téma optimalizace. Obecně, řešené praktické úlohy, kdy chceme dosáhnout co nejlepšího výsledku v rámci našich možností, nazýváme *optimalizační úlohy*. Základními prvky optimalizační úlohy jsou:

- Proměnné- vstupní parametry
- Omezující podmínky- popisují obor přípustných řešení úlohy
- Kriteriální (účelová) funkce- jednotlivým řešením přiřazuje jejich hodnotu

3.1 Prohledávaný prostor

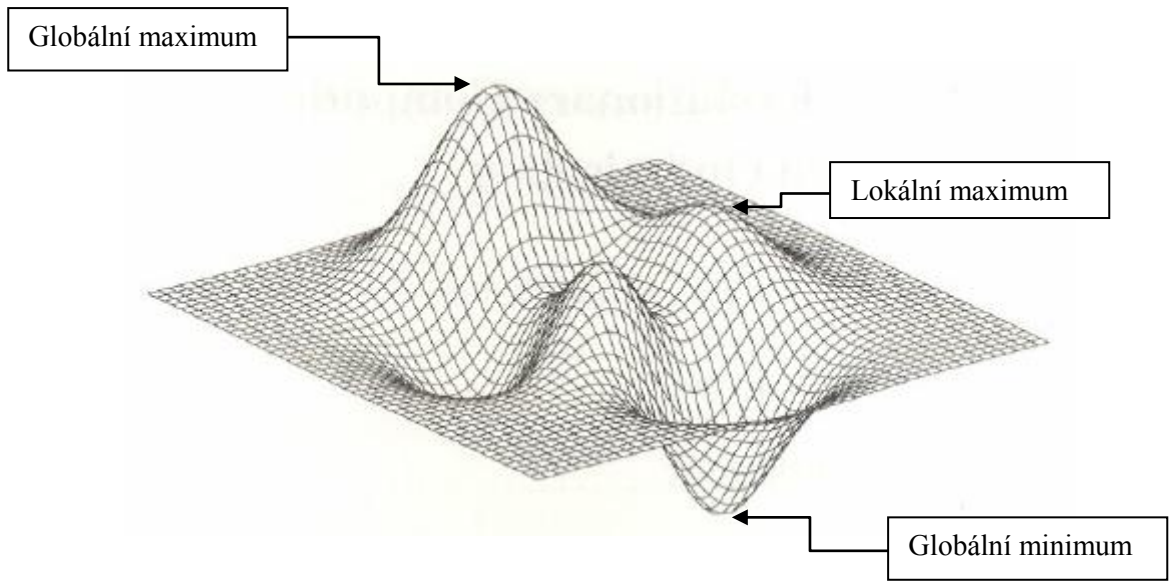
Pokud řešíme nějaký problém, tak obvykle hledáme řešení, které bude mezi ostatními nejlepší. Prostor všech možných řešení se nazývá *Problem Space,* který označíme jako X. Podprostor, ve kterém hledáme řešení pomocí nějaké optimalizační metody, se nazývá *Search Space* (v překladu prohledávaný prostor) a označíme ho jako $\tilde{X}$. Každý bod v tomto prostoru reprezentuje jedno možné řešení. Všechna tato řešení mohou být charakterizována jejich hodnotou nebo pomocí fitness u řešení optimalizačních problémů. [4]

Vzhledem k účelové funkci pak hledáme extrémy této funkce. Funkce $f(x)$ má v bodě lokální extrém, je-li funkční hodnota v tomto bodě vyšší (lokální maximum) či nižší (lokální minimum) než funkční hodnota v libovolném bodě nějakého okolí tohoto bodu.

- **Lokální maximum**: v bodě a je lokální maximum, pokud existuje nějaké okolí $U(a)$, pro které $x \in U(a) \wedge x \neq a \Rightarrow f(x) < f(a)$.
- **Lokální minimum**: v bodě a je lokální maximum, pokud existuje nějaké okolí $U(a)$, pro které $x \in U(a) \wedge x \neq a \Rightarrow f(x) > f(a)$.

V případě, že je funkční hodnota v nějakém bodě menší, resp. větší než funkční hodnota v libovolném jiném bodě celého definičního oboru, jedná se o **globální extrém.** [5]

V praxi se spíše využívá minimalizace účelové funkce a i v této práci budeme předpokládat, že hledáme minimum.



Obrázek 3-1 Extrémy funkce [6]

Existuje několik testovacích funkcí, na kterých se testují různé optimalizační metody. U těchto testovacích funkcí známe graf účelové funkce, a tudíž známe extrémy této funkce. Na základě toho pak lze posoudit, zda testovaný algoritmus nalezl globální minimum nebo maximum. Ovšem v praxi obvykle známe jen pár bodů tohoto prohledávaného prostoru, proto nelze o nalezeném řešení tvrdit, že se jedná o globální extrém. Nalezené řešení proto nazýváme *optimální řešení*. [4]

Obecně, existují dva způsoby, jak prohledávat prostor všech řešení.

- Průzkum (Exploration): Algoritmus vyšetří nové a neznámé oblasti v prohledávaném prostoru.
- Využití znalostí (Exploitation): Algoritmus použije poznatky, které získal z předchozího průzkumu a to mu pomůže najít lepší řešení.

Oba tyto způsoby jsou nezbytné, ale protichůdné v řešení optimalizačních problémů. Pro memetické algoritmy je charakteristické, že kombinují GA s nějakou jinou heuristickou metodou, nejčastěji se jedná o Local Search (lokální prohledávání). Genetické algoritmy se využijí pro průzkum prohledávaného prostoru a Local Search využije znalostí, které se získaly při předešlém průzkumu. Výsledkem je vysoká účinnost a lepší efekt. [7]

3.2 Problémy optimalizačních úloh

Jak již bylo řečeno na začátku práce, velkým problémem je, že pokud nalezneme nějaké řešení, které je třeba lepší než dosavadní, tak přesto nevíme, zda jsme nalezli globální nebo lokální extrém.

3.2.1 NP problémy

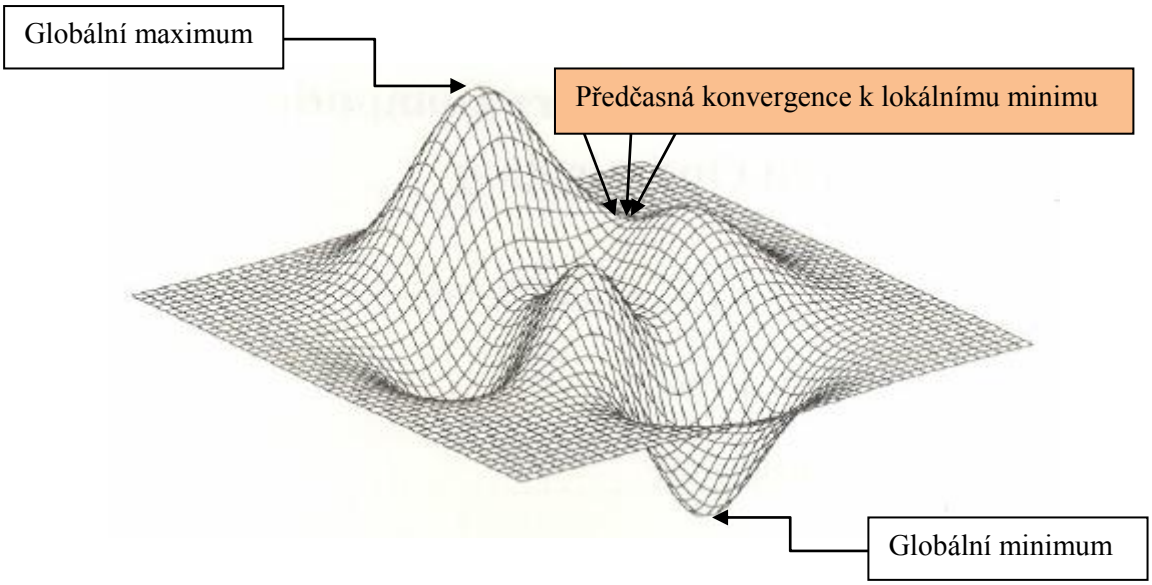
Nedeterministicky polynomiální (NP) problémy jsou skupina problému, které nelze řešit tradiční cestou. Jsou natolik složité, že nalezení všech možných řešení a porovnání těchto řešení není možné v nějakém uskutečnitelném čase. Pro řešení těchto problémů se stále vymýšlí různé alternativní metody a jednou z těchto metod mohou být i memetické algoritmy. Nejznámějším NP problémem je problém obchodního cestujícího. [4]

3.2.2 Předčasná konvergence

Předčasná konvergence znamená, že výpočet konverguje příliš rychle k nějakému neoptimálnímu řešení. Bod, který představuje jedno řešení, nalezne lokální extrém, tudíž při prohledávání okolí tohoto bodu jsou všechna řešení horší a algoritmus si myslí, že našel nejlepší řešení.

Předčasnou konvergenci může způsobovat:

- **Příliš velký selekční tlak**- přílišné upřednostňování několika výjimečných jedinců na úkor zbytku populace
- **Nedostatečná velikost populace**- optimální velikost populace roste exponenciálně s velikostí řešeného problému
- **Obsah počáteční populace**
- **Špatně navržené genetické operátory**- nutná rovnováha mezi diverzifikací (exploitation) a intenzifikací (exploration) [8]



Obrázek 3-2 Předčasná konvergence [6]

4 Seznam

Jelikož se jedná o evoluční programování, budou jednotlivá řešení zastupovat jedinci (prvky), které jsou umístěné v populaci. Tato populace představuje seznam prvků a proto je pro přehlednost v této kapitole krátce vysvětlena práce se seznamy.

- **Length** ($n = \mathrm{Length}(L)$) funkce, která navrací délku seznamu L, tj. počet prvků v seznamu.
- **CreateList** ($L.\mathrm{CreateList}(n, x)$) funkce, pro vytvoření nového seznamu L o délce n, naplněný prvkem x.
- **AddListItem** ($L.\mathrm{AddListItem}(x)$) funkce, která vloží prvek x na konec seznamu L.
- **Sort** $L.\mathrm{Sort}_{\mathrm{d}}(f(X))$ často je užitečné používat setříděný seznam, proto je zde definována funkce, která třídí seznam L podle pořadí sestupně na základě funkční hodnoty účelové funkce. Jestliže je seznam setříděný sestupně, znamená to, že na první pozici tj. $L[0]$ se nachází nejlepší prvek seznamu.
- **Clear** ($L.\mathrm{Clear}(\ )$) funkce, která vyčistí seznam L.
- **AppendList** $\left(M.\mathrm{AppendList}\,(L_1, L_2)\right)$ funkce pro vytvoření nového seznamu, který vznikne přidáním všech prvků ze seznamu L_2 do seznamu L_1.
- **DeleteListItem** ($M.\mathrm{DeleteListItem}(L, i)$) funkce pro vytvoření nového seznamu M pomocí odstranění prvků x na pozici i $(\forall i: 0 \leq 0 < \mathrm{Lenght}(L) - 1)$ ze seznamu L $(\mathrm{Lenght}(L) \geq i + 1)$.
- **LocalImprover** ($x \leftarrow \mathrm{LocalImprover}(L)$) funkce, která vylepší každý prvek x ze seznamu L využitím jedné z pseudogradientních metod, které jsou popsány v kapitole 5.3.
- **Selekce** ($M \leftarrow \mathrm{Selekce}(L)$) funkce pro vytvoření nového seznamu M využitím jedné ze selekčních metod které jsou podrobně popsány v kapitole 5.4.1.
- **Křížení** ($M \leftarrow \mathrm{Selekce}(L)$) funkce pro vytvoření nového seznamu M využitím funkce křížení, která je popsána v kapitole 5.4.2.
- **Mutace** ($M \leftarrow \mathrm{Selekce}(L)$) funkce pro vytvoření nového seznamu M využitím jedné z mutačních metod, které jsou popsány v kapitole 5.4.3.

5 Memetický algoritmus

Slovo *mem* zavedl Richard Dawkins v roce 1976, při popisu kulturní evoluce ve své knize Sobecký gen. Avšak první memetický algoritmus byl navržen později, a to v roce 1989 Pablem Moscatem. Z počátku se memetické algoritmy potýkaly s těžkými časy stejně tak jako například Evoluční algoritmy, ale nyní se stávají stále oblíbenějšími díky jejich úspěchu při řešení mnoha těžkých optimalizačních problémů. [7, 9]

Memetický algoritmus využívá koncept evoluce stejně jako Genetický algoritmus. Nicméně zatímco GA je založený na biologické evoluci, MA je založený na kulturní evoluci nebo také evoluci myšlenek (Ideas Evolution). V evoluci myšlenek nemusí být myšlenka zlepšena pouze křížením z ostatních myšlenek, ale také přizpůsobováním se. [10]

Memetické algoritmy patří mezi metaheuristické metody založené na populaci. To znamená, že algoritmus má k dispozici populaci řešení, tzv. seznam zahrnující několik řešení najedou. Každé z těchto řešení se nazývá *jedinec* (v MA se můžeme setkat i s pojmem *agent*). Tito jedinci jsou předmětem k soutěžení a vzájemné spolupráci.

Jako první je nezbytné představit základní populační událost a tou je *generace.* Každá generace se skládá z nové populace jedinců, vedoucí snad k lepšímu a lepšímu řešení. Jsou aplikovány tři hlavní kroky vedoucí k vytvoření nové generace. Selekce, reprodukce a nahrazení.

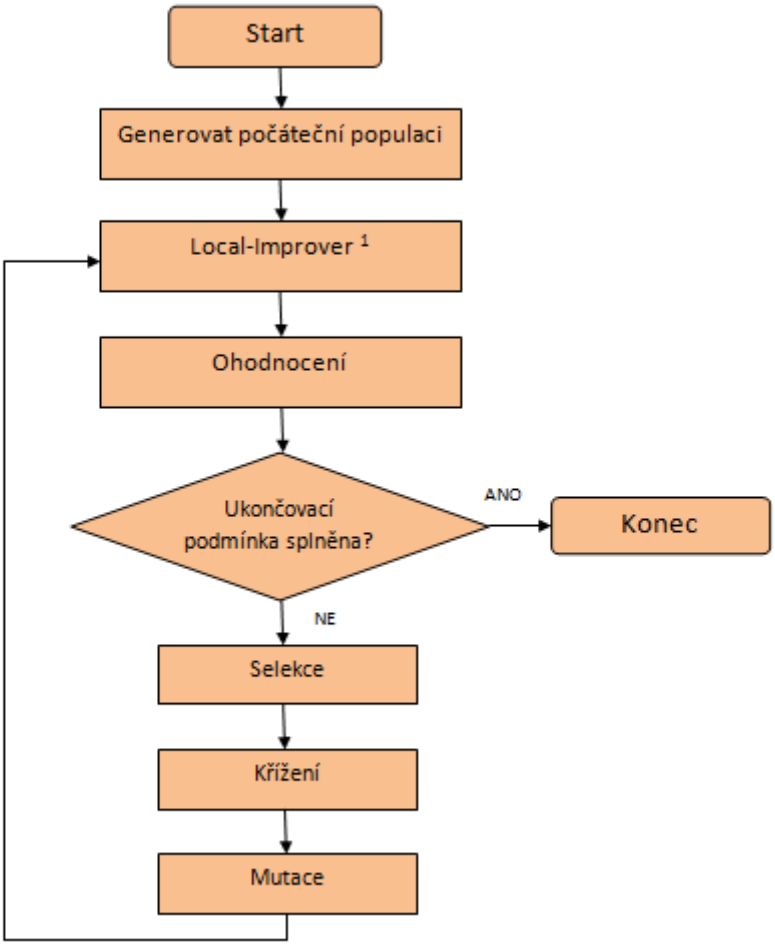
Selekční fáze (společně s fází nahrazení) je zodpovědná za soutěživou stránku jedinců. Použitím informací, které nám poskytuje řídící funkce (v MA se užívá spíše pojem fitness funkce), jsou jedinci v populaci ohodnoceni a následně je vzorek jedinců vybrán k reprodukci. Tento výběr může být uskutečněn několika způsoby. Nejčastěji se jedná o metody úměrné hodnotě fitness, metody na základě pořadí nebo turnajové metody. Jedinci mohou být však vybráni i zcela náhodně. Všechny tyto metody jsou popsány v následujících kapitolách.

Fáze nahrazení se stará o to, aby měla populace konstantní velikost. To dělá tak, že jedinci z původní populace se nahradí nově vytvořenými jedinci, získanými z reprodukční fáze.

Nejzajímavější je však reprodukční fáze. V této fázi musíme vytvořit nové jedince pomocí již existujících jedinců. Toho může být dosaženo pomocí mnoha reprodukčních operátorů. Nicméně, nejtypičtější situace zahrnuje použití pouze dvou operátorů a těmi jsou křížení (crossover) a mutace. Pomocí křížení je vytvořen nový jedinec (potomek), který používá informaci obsaženou v genu vybraných rodičů. Sekundárním operátorem je mutace, jejímž úkolem je celý proces „udržovat ve varu“ nepřetržitým vstřikováním nového materiálu do populace, ale pomalým tempem. V podstatě operátor mutace musí generovat nové jedince pomocí částečné modifikace existujících jedinců. Křížení a mutace budou více popsány v následujících kapitolách. [7]

Jedním z nejdůležitějších kroků, kterým se memetické algoritmy vyznačují je Local-Improver. V překladu se jedná o jakéhosi lokálního zlepšovatele. Local-Improver je proces, který začíná na určitém vrcholu a přesouvá se k sousedním vrcholům, pokud je sousední řešení lepší než aktuální řešení. Local-Improver se snaží najít „svah“ (tj. hodnotu funkce f) cesty v grafu, který byl popsán v první části práce. Oficiální název tohoto grafu je *fitness landscape*. Délka cesty nalezené pomocí Local-Improver je určena nějakou ukončovací podmínkou. Obvyklým příkladem je ukončení cesty, když není možný žádný další pohyb po svahu tj. když aktuální řešení je lokálním optimem. Nicméně nemusí to tak být nutně vždy. Například cesta může nabýt maximální povolené délky nebo může být cesta ukončena, jakmile zlepšená hodnota řídící funkce je považována za dostatečně dobrou. [7]

Fungování memetického algoritmu je založené na opakování těchto základních generačních kroků a je znázorněno na obrázku níže.



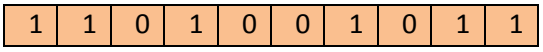
Obrázek 5-1Vývojový diagram pro MA

1. Krok: Vygeneruj počáteční populaci. Parametry pro MA jsou stejné jako u GA a to velikost populace, počet iterací, míra mutace a míra křížení. Poté zakóduj počáteční řešení do chromosomu. Opakuj tento krok, dokud nebude počet jedinců roven velikosti populace.
2. Krok: Aplikuj Local-Improver, aby se vylepšila kvalita každého jedince z populace. [1]
3. Krok: Dekóduj každého jedince z populace, abychom získali odpovídající hodnotu fitness pro každého jedince. A porovnej je, abychom získali nejlepší řešení.
4. Zkontroluj kritéria pro ukončení. Jestliže je jedno z kritérií splněno, pak zastav algoritmus a výstupem bude nejlepší řešení. Jinak přejdi na krok 5.
5. Krok: Vygeneruj novou populaci pro další generaci. Jsou aplikovány tři operace pro vytvoření potomků pro novou populaci. Těmito operacemi jsou: selekce, křížení a mutace. Po tomto se algoritmus vrátí k druhému kroku.

[1] Local-Improver může být použit v různých částech algoritmu. Například může být vložen hned na začátek, pro vylepšení počáteční populace, jako je to na obrázku. Nebo může být použit po využití jakéhokoliv jiného rekombinačního nebo mutačního operátoru. V neposlední řade, může být použit jen na konci reprodukční fáze.

5.1 Vlastnosti jedince

- $X.gen$ - Každý jedinec v populaci je charakterizován svým genem. Tento gen představuje seznam určitého počtu alel. V této práci budou hodnoty alel nabývat pouze hodnot 1 nebo 0. Hodnoty jsou do genu vloženy náhodně. Funkce, pro naplnění genu, je popsána pomocí pseudokódu, viz Algoritmus 5-2. V Algoritmu 3-1 je pouze popsáno generování náhodného čísla v určitém intervalu tak, aby bylo zajištěno, že do intervalu bude patřit dolní i horní hodnota. Jako příklad je ukázán gen (viz Obrázek 5-2), který obsahuje 10 alel (tj. $n_a = 10, k = 0, 1, 2, \ldots 9$).



Obrázek 5-2 Gen

$z \longleftarrow Random\ (a, b)$		
Funkce, jejímž výstupem je náhodné číslo z z intervalu $[a, b]$		
Vstup:	a:	Spodní hranice intervalu
Vstup:	b:	Horní hranice intervalu
Výstup:	z:	Náhodné číslo z intervalu $z \in [a, b]$

```
1   begin
2     z ← (b − a) ∗ random(  ) + a;
3     result← z;
4   end;
```

Algoritmus 5-1 Random

$X.gen \longleftarrow Naplňgen\ (n_a)$		
Funkce, jejímž výstupem je gen, jehož alely se náhodně zaplnily jedničkou nebo nulou.		
Vstup:	n_a:	Počet alel v jednom genu
Data:	k:	Proměnná – čítač
Výstup:	gen:	gen, jehož alely se naplnily nulou nebo jedničkou

```
5   Begin
6     X.gen ← (  );
7     for k ← 0 to n_a − 1 do
8         X.gen.AddListItem⌊Random(0,1)⌋;
9     end;
10    result← X.gen;
11  end;
```

Algoritmus 5-2 Naplň Gen

- $X.sour$ – Další vlastností jedince je jeho souřadnice. Souřadnice je vypočtena z hodnot obsažených v genu každého jedince. Tento výpočet je znázorněn na obrázku níže a popsán pomocí pseudokódu, viz Algoritmus 5-3. Tímto způsobem je vypočtena hodnota jedné souřadnice $X = [843]$. Počet souřadnic je dán počtem dimenzí neboli os. Pokud budeme mít dvě dimenze (tj. $dim = 2$, $j = 0{,}1$), bude každý jedinec charakterizován dvěma geny, tudíž bude mít dvě souřadnice $\left(X = \left[dim[0], dim[1]\right]\right)$.

1	1	0	1	0	0	1	0	1	1
2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

$$1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 + 0 \times 2^5 + 1 \times 2^6 + 0 \times 2^7 + 1 \times 2^8 + 1 \times 2^9 = 843$$

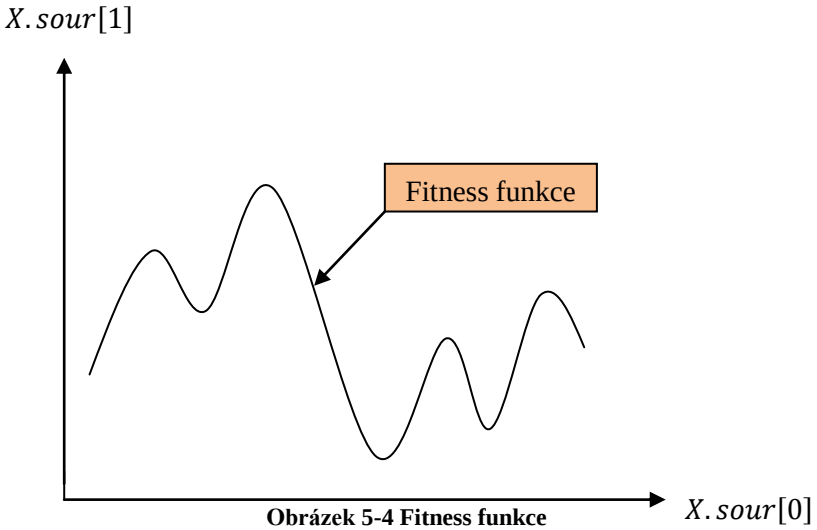
Obrázek 5-3 Výpočet souřadnice

$X.sour \leftarrow VypočtiSour\ (X.gen)$		
Funkce, jejímž výstupem je souřadnice $X.sour$ jedince.		
Vstup:	$X.gen$:	jeden gen
Data:	k:	Proměnná - čítač
Data:	y:	Proměnná - čítač
Data:	z:	Proměnná - čítač
Výstup:	$X.sour$:	Souřadnice

```
1   begin
2     for j ← 0 to dim − 1 do begin
3         k ← Length(X.gen) − 1 ;
4         y ← 0;
5         z ← 0;
6         while  k ≥ 0 do begin
7                 y ← y + (X.gen[k]) ∗ 2^z;
8                 k ← k − 1;
9                 z ← z + 1;
10        end;
11        X.sour ← y;
12    end;
13    result← X.sour;
14  end;
```

Algoritmus 5-3 VypočtiSour

- $X.fit$ – Poslední vlastností jedince je jeho hodnota fitness. Tato hodnota je dána jako funkční hodnota souřadnice $f(X.sour)$, tudíž se do předpisu funkce f dosadí hodnoty souřadnic. Na obrázku níže je znázorněn průběh funkce f ve 2D grafu v případě, že počet dimenzí bude roven dvěma (tj. $dim = 2, j = 0,1$). Na základě hodnoty fitness pak dochází k ohodnocení každého jedince. Nejprve se vypočítá hodnota fitness každého jedince dosazením souřadnic do předpisu funkce f a následně dojde k setřídění populace na základě této hodnoty. Populace bude setříděna sestupně, tudíž na první pozici tj. $P[0]$ bude jedinec s nejlepší hodnotou fitness. Funkce ohodnocení je popsána pomocí pseudokódu, viz Algoritmus 5-4.



Obrázek 5-4 Fitness funkce

$X_{best} \leftarrow Ohodnocení(P, X.gen, dim)$		
Funkce, která ohodnotí jedince v populaci na základě jejich hodnoty fitness. Výstupem této funkce je nejlepší jedinec v populaci X_{best}.		
Vstup:	P:	Populace jedinců
Vstup:	dim:	Počet dimenzí
Vstup:	$X.gen$:	Naplněný gen jedince
Data:	i:	Proměnná - čítač
Data:	j:	Proměnná - čítač
Výstup:	X_{best}:	Výsledná populace

```
1   begin
2     for i ← 0 to n − 1 do
3         for j ← 0 to dim − 1 do
4             X.sour ← VypočtiSour(X.gen);
5         end;
6             P.Sort_d(f(X.sour));
7             X_best ← P[0];
8     end;
9     result ← X_best;
10  end;
```

Algoritmus 5-4 Ohodnocení

5.2 Generace počáteční populace

Generování počáteční populace je prvním krokem memetického algoritmu. Jedná se o funkci, jejímž výstupem je počáteční populace P naplněná náhodně vybranými jedinci. Každý jedinec v populaci je reprezentován svým genem, který byl popsán v předchozí kapitole. Generování počáteční populace je popsáno pomocí pseudokódu, viz Algoritmus 5-5.

$P \leftarrow GenerujPočátečníPopulaci(n, dim, n_a)$		
Funkce, jejímž výstupem je počáteční populace naplněná náhodně vybranými jedinci.		
Vstup:	n:	Počet jedinců, kteří budou umístěni do výsledné populace
Vstup:	dim:	Počet dimenzí
Vstup:	n_a:	Počet alel v genu
Data:	i:	Proměnná - čítač
Data:	j:	Proměnná - čítač
Výstup:	P:	Výsledná populace

```
1   begin
2     P ← (  );
3     for i ← 0 to n − 1 do
4       for j ← 0 to dim − 1 do
5         X.gen ← NaplňGen(n_a);
6       end;
7       X ← X.gen;
8       P.AddListItem(X);
9     end;
10    result ← P;
11  end;
```

Algoritmus 5-5 GenerujPočátečníPopulaci

5.3 Local-Improvers (Pseudogradientní metody)

U pseudogradientních metod algoritmus postupuje při hledání řešení ve směru gradientu (ve směru největšího spádu). Předpokladem je, že funkce f je diferencovatelná. Za výchozí bod iteračního postupu zvolíme libovolný bod $X.sour$ z množiny P, tj. libovolné přípustné řešení, které označíme X.

Ve druhém kroku musíme určit směr a délku kroku. Řešení $(k+1)$ iteračního kroku vypočítáme pomocí směrového vektoru a délky kroku.

Při minimalizaci účelové funkce $\min\{f(X);\ X \in P\}$ bude směr: $d = -\Delta f(X)$.

Přitom musí platit: $X \in P$ a $f(X_{k+1}) = f(X_k + s_k) < f(X_k)$.

Pokud nelze nalézt směrový vektor nebo délku kroku tak, aby byly splněny tyto podmínky, je poslední nalezené řešení X_k optimálním řešením a výpočet končí. Jinak je určeno nové řešení X_{k+1} a přejde se ke kroku 2, tj. hledání nového směru a délky kroku.

Mezi nejznámější pseudogradientní metody patří metoda Local Search, Hill Climbing a nebo Tabu Search. Jednotlivé metody budou popsány v následujících kapitolách tak, jak fungují samostatně a poté budou v dalších kapitolách popsány změny, které je nutné provést, pokud je budeme chtít nakombinovat s jinou metodou.

5.3.1 Lokální prohledávání (Local Search)

Local Search patří do skupiny metaheuristických metod, tudíž nelze zaručit nalezení nejlepšího řešení. Princip této metody je založený na tom, že se náhodně vygeneruje prvek (jedinec) z prohledávaného prostoru (viz Algoritmus 5-6) a následně se aplikují lokální změny, jako jsou mutace (viz Algoritmus 5-7) a perturbace (viz Algoritmus 5-8). Pozměněný jedinec se porovná s původním, a jestliže je pozměněný jedinec lepší, nahradí původního jedince (původní se přesune na pozici pozměněného). Jak tento algoritmus funguje samostatně, je popsáno pomocí pseudokódu, viz Algoritmus 5-9.

$X.sour \longleftarrow Create\,(A,B)$		
Funkce, jejímž výstupem je náhodně vygenerovaný prvek na základě rovnoměrného rozdělení v mezích intervalu prohledávaného prostoru.		
Vstup:	A:	Seznam dolních mezí prohledávaných intervalů rozhodovacích proměnných
Vstup:	B:	Seznam horních mezí prohledávaných intervalů rozhodovacích proměnných
Data:	dim:	Dimenze prostoru všech rozhodovacích proměnných
Data:	j:	Čítač
Výstup:	$X.sour$:	Vygenerovaný prvek

```
1   begin
2     X.sour ⟵ ( );
3     dim ⟵ min(Length(A), Length(B));
      (*zjištění počtu rozhodovacích proměnných – délky seznamu A i B by měly být stejné*)
4     for j ⟵ 0 to dim − 1 do
5         X.sour.AddListItem( Random(A[j], B[j]));
      (*vygenerování hodnoty rozhodovací proměnné, která bude vložena do prvku*)
6     result ⟵ X.sour;
7   end;
```

Algoritmus 5-6 Create [11]

Jednou z lokálních změn, která je aplikována na vygenerovaného jedince, je mutace. Transformace jednotlivých složek vstupního prvku se provádí na základě vygenerování náhodného čísla generovaného podle rovnoměrného rozdělení v rozmezí délky intervalu dolní a horní meze definovaného pro každou jednotlivou rozhodovací proměnnou (geometricky pro každou osu).

$X.sour_{mut} \longleftarrow Mutace(X.sour, E)$		
Funkce, jejímž výstupem je zmutovaný prvek, který vznikl transformací rozhodovacích proměnných (v kontextu evolučních algoritmů mutací) vstupního prvku $X.sour$.		
Vstup:	$X.sour$:	Původní jedinec, jehož souřadnice budou transformovány
Vstup:	E:	Seznam obsahující jednotlivé délky intervalů (vzdálenost mezi dolní a horní mezí) pro jednotlivé rozhodovací proměnné užité pro generování náhodného čísla ($E[j] = \lvert b_j - a_j \rvert$)
Data:	j:	Proměnná - čítač
Výstup:	$X.sour_{mut}$:	Jedinec, jehož složky jsou zmutované - transformované

```
1   begin
2     X.sour_mut ⟵ ( );
3     for j ⟵ 0 to (Length(X.sour) − 1) do   //pro všechny rozhodovací proměnné
```

$X.sour_{mut} \leftarrow Mutace(X.sour, E)$		
Funkce, jejímž výstupem je zmutovaný prvek, který vznikl transformací rozhodovacích proměnných (v kontextu evolučních algoritmů mutací) vstupního prvku $X.sour$.		
Vstup:	$X.sour$:	Původní jedinec, jehož souřadnice budou transformovány
Vstup:	E:	Seznam obsahující jednotlivé délky intervalů (vzdálenost mezi dolní a horní mezí) pro jednotlivé rozhodovací proměnné užité pro generování náhodného čísla ($E[j] = \lvert b_j - a_j \rvert$)
Data:	j:	Proměnná - čítač
Výstup:	$X.sour_{mut}$:	Jedinec, jehož složky jsou zmutované - transformované

4 $X.sour_{mut}.AddListItem\left(Random\left(X.sour[j] - \frac{|E[j]|}{2}, X.sour[j] + \frac{|E[j]|}{2}\right)\right);$
(*přidání hodnoty souřadnice do $X.sour_{mut}$ pomocí funkce pro generování náhodného čísla v mezích intervalu podle rovnoměrného rozdělení*)
5 **result** $\leftarrow X.sour_{mut}$;
6 **end**;

Algoritmus 5-7 Mutace [11]

Druhou lokální změnou je perturbace neboli zrcadlení. Výstupem je jedinec vzniklý zrcadlením souřadnic rozhodovacích proměnných vstupního jedince, které ležely mimo rozsah intervalu jednotlivé rozhodovací proměnné. Princip zrcadlení hodnoty rozhodovací proměnné jedince kolem dolní meze, nebo kolem horní meze je uplatňován tak dlouho, dokud hodnota rozhodovací proměnné neleží uvnitř prohledávaného intervalu rozhodovací proměnné. [11]

$X.sour_{pert} \leftarrow Perturbace(X.sour, A, B)$		
Funkce, jejímž výstupem je prvek, který vznikl perturbací – zrcadlením - souřadnic rozhodovacích proměnných vstupního prvku, které ležely mimo rozsah intervalu jednotlivé rozhodovací proměnné.		
Vstup:	$X.sour$:	Jedinec, který obsahuje souřadnice rozhodovacích proměnných mimo prohledávaný interval rozhodovací proměnné
Vstup:	A:	Dolní meze prohledávaného prostoru X
Vstup:	B:	Horní meze prohledávaného prostoru X
Data:	j:	Čítač – vyjadřuje index rozhodovací proměnné
Výstup:	$X.sour_{pert}$:	Jedinec vzniklý perturbací původního jedince.

1 **begin**
2 $X.sour_{pert} \leftarrow X.sour$;
3 **for** $j \leftarrow 0$ **to** Length$(X.sour) - 1$ **do**
//indexování rozhodovacích proměnných - dimenzí
4 **while** $(X.sour[j] < A[j])$ **or** $(X.sour[j] > B[j])$ **do**
5 **if** $(X.sour[j] < A[j])$ **then**
6 $X.sour_{pert}[j] \leftarrow 2 \cdot A[j] - X.sour[j]$
7 **else if** $(X.sour[j] > B[j])$ **then**
8 $X.sour_{pert}[j] \leftarrow 2 \cdot B[j] - X.sour[j]$;
9 **result** $\leftarrow X.sour_{pert}[j]$;
10 **end**;

Algoritmus 5-8 Perturbace [11]

$X_{best} \leftarrow LocalSearch(E, A, B)$		
Funkce, jejímž výstupem je dosud nalezený nejlepší prvek. Generování prvku je prováděno v sousedství dosud nejlepšího nalezeného prvku (mutace).		
Vstup:	E:	Délky intervalů prohledávaného okolí jedince
Vstup:	A:	Dolní meze prohledávaného prostoru
Vstup:	B:	Horní meze prohledávaného prostoru
Data:	$X.sour$:	Nově vygenerovaný prvek pomocí mutace
Data:	$F(X.sour)$:	Hodnota účelové funkce prvku $X.sour$
Data:	i:	Proměnná - čítač
Výstup:	X_{best}:	Dosud nalezený nejlepší prvek

```
1   begin
2     X_best ← Create(A, B);
3     while not TerminationCriterion( ) do begin
4         X.sour ← Mutace(X_best, E);
5         X.sour ← Perturbace(X.sour, A, B);
6         if F(X.sour) < F(X_best) then
7             X_best ← X.sour ;
8     end;
9     result ← X_best;
10  end;
```

Algoritmus 5-9Local Search [11]

5.3.2 Horolezecký algoritmus (Hill Climbing)

Princip metody Hill Climbing (viz Algoritmus 5-10) je v podstatě stejný jako u metody Local Search s tím rozdílem, že tentokrát se lokální změny neprovádí na náhodně vygenerovaném jedinci, ale na aktuálně známém nejlepším jedinci. Samostatně tento algoritmus tedy funguje tak, že se vygeneruje počáteční populace, setřídí se podle hodnoty fitness a určí se nejlepší jedinec z populace. Na tohoto jedince budou aplikovány stejné změny jako u Local Search, tedy mutace a perturbace. Následně se opět porovná pozměněný jedinec s původním, a pokud je lepší, nahradí původního.

$X_{best} \leftarrow HillClimbing(n, E, A, B)$		
Funkce, jejímž výstupem je dosud nalezený nejlepší prvek. Generování prvků je prováděno v sousedství nejlepšího nalezeného prvku z předchozí populace. Generování je prováděno pomocí transformace (mutace) nejlepšího prvku z P na základě rovnoměrného rozdělení.		
Vstup:	n:	Velikost populace P
Vstup:	E:	Délky intervalů prohledávaného okolí jedince
Vstup:	A:	Dolní meze prohledávaného prostoru X
Vstup:	B:	Horní meze prohledávaného prostoru X
Data:	$X.sour$:	Nově vygenerovaný prvek pomocí mutace
Data:	P:	Seznam vygenerovaných prvků (populace)
Data:	$X.sour^*$:	Nejlepší jedinec z předchozí populace jedinců
Data:	$F(X.sour)$:	Hodnota fitness prvku $X.sour$
Data:	i:	Proměnná - čítač
Výstup:	X_{best}:	Dosud nalezený nejlepší prvek

```
1   begin
2     P ← GenerujPočátečníPopulaci(n, n_a, dim);
```

$X_{best} \leftarrow HillClimbing(n, E, A, B)$		
Funkce, jejímž výstupem je dosud nalezený nejlepší prvek. Generování prvků je prováděno v sousedství nejlepšího nalezeného prvku z předchozí populace. Generování je prováděno pomocí transformace (mutace) nejlepšího prvku z P na základě rovnoměrného rozdělení.		
Vstup:	n:	Velikost populace P
Vstup:	E:	Délky intervalů prohledávaného okolí jedince
Vstup:	A:	Dolní meze prohledávaného prostoru X
Vstup:	B:	Horní meze prohledávaného prostoru X
Data:	$X.sour$:	Nově vygenerovaný prvek pomocí mutace
Data:	P:	Seznam vygenerovaných prvků (populace)
Data:	$X.sour^*$:	Nejlepší jedinec z předchozí populace jedinců
Data:	$F(X.sour)$:	Hodnota fitness prvku $X.sour$
Data:	i:	Proměnná - čítač
Výstup:	X_{best}:	Dosud nalezený nejlepší prvek

```
3     P.Sort_d(F(X.sour));   //setřídění P podle hodnoty fitness
4     X.sour* ← P[0];   //nejlepší jedinec z P
5     X_best ← X.sour*;   //nejlepší jedinec z P je zároveň nejlepší dosud nalezený prvek
6     while not TerminationCriterion( ) do begin
7       P.Clear( );   //vyčištění P
8       for i ← 0 to n − 1 do   begin
9         X.sour ← Mutace(X.sour*, E);
      (*mutace nejlepšího jedince z P na základě rovnoměrného rozdělení*)
10        X.sour ← Perturbace(X.sour, A, B);     //perturbace jedince
11        P.AddListItem(X.sour);   //přidání vygenerovaného jedince do P
12      end;
13      P.Sort_d(F(X.sour));
14      X.sour* ← P[0];
15      if F(X.sour*) < F(X_best) then     //bylo nalezeno lepší řešení než aktuální
16        X_best ← X.sour* ;
17    end;
18    result ← X_best;
19  end;
```

Algoritmus 5-10 Hill Climbing [11]

5.3.3 Zakázané prohledávání (Tabu Search)

Odlišným způsobem rozšíření konceptu horolezeckého algoritmu je velmi oblíbená metoda nazvaná Tabu Search. Základní princip je stejný, jen se po vybrání nejlepšího jedince z aktuálního okolí nejprve zkontroluje, jestli již nebylo dříve navštíveno. Algoritmus udržuje seznam posledních n_{Tabu} navštívených bodů, tzv. Tabu List P_{Tabu} (seznam obsahující zakázané prvky, které nebudou akceptovány), jímž zabraňuje opakovanému navštívení stejných bodů. [12]

$X_{best} \leftarrow TabuSearch(n, n_{\mathrm{Tabu}}, E, A, B)$		
Funkce, jejímž výstupem je dosud nalezený nejlepší prvek. Během optimalizačního procesu je každý nově vygenerovaný prvek vložen do seznamu zakázaných prvků. Takový prvek nesmí být navštíven, pokud nesplňuje aspirační kritérium. Pokud je překročena povolená délka seznamu zakázaných prvků, je provedeno vyjmutí prvku z tohoto seznamu – metoda FIFO. Generování prvku v sousedství dosud nejlepšího nalezeného prvku z předchozí populace, se děje pomocí transformace (mutace) nejlepšího prvku z P na základě rovnoměrného rozdělení.		
Vstup:	n:	Velikost seznamu (populace) P tj. délka seznamu $m = \mathrm{Length(P)}$
Vstup:	n_{Tabu}:	Maximální délka seznamu obsahující prvky vytvořené v n_{Tabu} předchozích krocích
Vstup:	E:	Délky intervalů prohledávaného okolí jedince
Vstup:	A:	Dolní meze prohledávaného prostoru X
Vstup:	B:	Horní meze prohledávaného prostoru X
Data:	P:	Seznam vygenerovaných jedinců (populace)
Data:	$X.sour$:	Nově vygenerovaný jedinec pomocí mutace
Data:	$F(X.sour)$:	Hodnota účelové funkce jedince $X.sour$
Data:	$X.sour^*$:	Nejlepší jedinec z předchozí populace
Data:	P_{Tabu}:	Tabu List
Data:	i:	Proměnná - čítač
Výstup:	X_{best}:	Dosud nalezený nejlepší prvek

```
1   begin
2     P ⟵ GenerujPočátečníPopulaci(n, X.gen, n_a);
3     P.Sort_d(F(X.sour));//setřídění P podle hodnoty fitness
4     X.sour* ⟵ P[0];    //nejlepší prvek z P
5     X_best ← X.sour*;    //nejlepší prvek z P je zároveň nejlepší dosud nalezený prvek
6     P_Tabu ⟵ ( ); //vytvoření prázdného seznamu obsahujícího zakázané prvky
7     P_Tabu.AppendList(P_Tabu, P);
8     while not TerminationCriterion( ) do begin
9       P.Clear( );    //vyčištění P
10      for i ⟵ 0 to n − 1 do  begin
11        repeat
12          X.sour ⟵ Mutace(X.sour*, E);
      (*mutace nejlepšího prvku z P na základě rovnoměrného rozdělení*)
13          X.sour ⟵ Perturbace(X.sour, A, B);//perturbace prvku
14          P.AddListItem(X.sour);    //přidání vygenerovaného prvku do P
15        until (Search_u(X.sour, P_Tabu) < 0) or (F(X.sour) < F(X_best));
      (*prvek se nenachází v P_Tabu nebo splňuje aspirační kritérium*)
16        if Length(P_Tabu) ≥ n_Tabu then
17          P_Tabu.DeleteListItem(0);
18          P_Tabu.AddListItem(X.sour);
19          P.AddListItem(X.sour);    //přidání vygenerovaného prvku do P
20      end;
21      P.Sort_d(F(X.sour));
22      X.sour* ⟵ P[0];
23      if F(X.sour*) < F(X_best) then     //bylo nalezeno lepší řešení než aktuální
24        X_best ⟵ X.sour*;
25    end;
26    result ⟵ X_best;
27  end;
```

Algoritmus 5-11 TabuSearch [11]

5.4 Genetické algoritmy

V této kapitole budou popsány genetické algoritmy, které jsou nedílnou součástí memetických algoritmů. Stejně jako jednotlivé pseudogradientní metody budou nejprve genetické algoritmy popsány tak, jak fungují samostatně.

5.4.1 Selekce

Operace selekce nebo jinak výběr slouží k tomu, aby vybrala z již existující populace jedince, kteří se zúčastní následujícího křížení a mutace. Existuje několik možností, jak mohou být tito jedinci vybráni.

- Náhodná selekce (Random selection)
- Selekce úměrná hodnotě fitness (Fitness proportinate selection)
- Turnajová selekce (Tournament selection)
- Selekce na základě pořadí (Rank-based selection) [10]

U všech těchto druhů selekce je možné použít buď selekci se substitucí, nebo selekci bez substituce. Jestliže se jedná o selekci se substitucí, znamená to, že stejný jedinec se může ve výsledné populaci vyskytnout několikrát. U selekce bez substituce platí, že jedinec se výsledné populaci může vyskytnout pouze jednou. [11]

5.4.1.1 Náhodná selekce

Jedinci jsou vybíráni náhodně bez ohledu na jejich fitness. Každý jedinec (dobrý i špatný) má stejnou šanci na to být vybrán. [10]

Na ukázku byla vybrána náhodná selekce bez substituce, která je znázorněna na algoritmu níže (viz. Algoritmus 5-12Algoritmus 5-12 RandomSelection).

$P_r \leftarrow RandomSelection(P, n_r)$		
Funkce, která na základě rovnoměrného rozdělení vybírá m_{MP} jedinců ze seznamu P. Výstupem funkce je seznam jedinců určených pro reprodukci. Selekce bez substituce – jedinec se může vyskytnout ve výstupním seznamu maximálně jedenkrát.		
Vstup:	P:	Seznam, ze kterého bude proveden výběr do X_{MP}
Vstup:	n_r:	Počet jedinců, kteří budou umístěni do výsledného seznamu určeného k páření, $n_r \leq \mathrm{Length(P)}$
Data:	i:	Proměnná - čítač
Data:	j:	Index vybraného jedince
Výstup:	P_r:	Výsledná populace určená k páření

```
1   begin
2     P_r ⟵ ( );
3     for i ⟵ 0 to n_r − 1 do begin
4         j ⟵ ⌊Random_u(Length(P))⌋;
5         P_r.AddListItem( P[j]);
6         P.DeleteListItem(P_r, j);
7     end;
8     Result ⟵ P_r;
9   end;
```

Algoritmus 5-12 RandomSelection [11]

5.4.1.2 Selekce úměrná hodnotě fitness

Nejčastější metodou této selekce je selekce založená na mechanismu rulety (Roulette Wheel Selection). Šance jedinců, že budou vybráni, je přímo úměrná hodnotě jejich fitness. Efekt této metody je závislý na rozsahu hodnot fitness v aktuální populaci.

Nejprve se vypočítá hodnota fitness $f(X_i)$ pro každého jedince. V dalším kroku se spočítá celková fitness dle následujícího vztahu: [10]

$$F = \sum_{i=1}^{popsize} f(X.sour[i]) \quad (1)$$

A v posledním kroku můžeme spočítat pravděpodobnost P_i, s jakou bude jedinec vybrán. Tu spočítáme dle následujícího vztahu: [10]

$$P_i = \frac{f(X.sour[i])}{F}, \forall i \in [0, Lenght(P) - 1] \quad (2)$$

kde:

- $f(X.sour[i])$ … hodnota fitness pro každého jedince
- F … celková fitness
- P_i … pravděpodobnost, s jakou bude jedinec vybrán

$P_r \leftarrow RouletteWheelSelect(P, n_r)$		
Funkce, která na základě ruletového mechanismu vybírá n_r jedinců ze seznamu P. Výstupem funkce je seznam jedinců určených pro reprodukci. Selekce bez substituce – jedinec se může vyskytnout ve výstupním seznamu maximálně jedenkrát.		
Vstup:	P:	Seznam, ze kterého bude proveden výběr do X_{MP}
Vstup:	n_r:	Počet jedinců, kteří budou umístěni do výsledného seznamu určeného k páření
Data:	f:	Funkce, která poskytuje jako výstup hodnotu fitness jedince
Data:	i:	Proměnná - čítač
Data:	a, b:	Dočasné proměnné pro uložení numerických hodnot
Data:	A:	Pole hodnot fitness
Data:	$minf, maxf, sum$:	Hodnoty minima, maxima a součtu hodnot fitness
Výstup:	P_r:	Výsledná populace určená k páření

```
1   begin
2     A ← CreateList(Length(P), 0);
3     minf ← ∞;
4     maxf ← −∞;
5     for i ← 0 to Length(P) − 1 do begin
6       a ← f(P[i]);
7       A[i] ← a
8       if a < minf then minf ← a;
9       if a > maxf then maxf ← a;
10    end;
11    if maxf = minf then begin
12      maxf ← maxf + 1;
13      minf ← minf − 1;
```

$P_r \leftarrow RouletteWheelSelect(P, n_r)$		
Funkce, která na základě ruletového mechanismu vybírá n_r jedinců ze seznamu P. Výstupem funkce je seznam jedinců určených pro reprodukci. Selekce bez substituce – jedinec se může vyskytnout ve výstupním seznamu maximálně jedenkrát.		
Vstup:	P:	Seznam, ze kterého bude proveden výběr do X_{MP}
Vstup:	n_r:	Počet jedinců, kteří budou umístěni do výsledného seznamu určeného k páření
Data:	f:	Funkce, která poskytuje jako výstup hodnotu fitness jedince
Data:	i:	Proměnná - čítač
Data:	a, b:	Dočasné proměnné pro uložení numerických hodnot
Data:	A:	Pole hodnot fitness
Data:	minf , maxf , sum:	Hodnoty minima, maxima a součtu hodnot fitness
Výstup:	P_r:	Výsledná populace určená k páření

```
14    end;
15    sumNormf ⟵ 0;
16    for i ⟵ 0 to Length(P) − 1 do begin
17        Normf ⟵ (maxf − A[i]) / (maxf − minf);
18        A[i] ⟵ Normf;
19        sumNormf ⟵ sumNormf + Normf;
20    end;
21    sum ⟵ 0;
22    for i ⟵ 0 to Length(P) − 1 do begin
23        P_i ⟵ A[i] / sumNormf;
      //pravděpodobnost výběru jedince
24        sum ⟵ sum + P_i;
      //kumulativní ohodnocení
25        A[i] ⟵ sum;
      //vektor kumulativních ohodnocení
26    end;
27    for j ⟵ 0 to min{n_r, Length(P)} − 1 do begin
      //min je v tomto případě funkce, která navrací nejmenší hodnotu z množiny
28        a ← Random(0, sum);
29        for i ⟵ 0 to Length(P) − 1 do
30            if (A[i] > a) then break;
      //nalezení segmentu v ruletě tj. i-tého indexu jedince v populaci
31        if i = 0 then b ⟵ 0
32        else b ⟵ A[i − 1];
33        b ⟵ A[i] − b;
34        for k ⟵ i + 1 to Length(A) − 1 do
35            A[k] ⟵ A[k] − b;
36        sum ⟵ sum − b;
37        P_r.AddListItem(P[i]);
38        P.DeleteListItem(i);
39        A.DeleteListItem(i);
40    end;
41    Result ⟵ P_r;
42  end;
```

Algoritmus 5-13 RouletteWheelSelection [11]

5.4.1.3 Turnajová selekce

Jak vyplívá z názvu, turnajová selekce je založena na jakýchsi turnajích. Tyto turnaje se odehrávají mezi dvěma jedinci, kteří jsou náhodně vybráni z populace P. Turnaj určí, který jedinec je lepší a ten bude vybrán k reprodukci (bude zařazen do populace určené k reprodukci P_r). Tento turnaj ovšem není turnajem v pravém slova smyslu. Jedná se pouze o generování náhodných hodnot mezi nulou a jedničkou a porovnávání těchto hodnot, abychom předurčili, jaká je pravděpodobnost selekce. Jestliže náhodná hodnota je menší nebo rovna pravděpodobnosti selekce, je vybrán zdatnější kandidát, jinak je zvolen slabší kandidát. Parametr pravděpodobnosti poskytuje vhodný mechanismus k nastavování selekčního tlaku. Ten je v praxi vždy nastaven na více než 0,5 za účelem upřednostňování zdatnějších kandidátů. Tato selekční strategie funguje velmi dobře pro mnoho typů problémů a je široce využívána v evolučních algoritmech. [13]

Na ukázku byla vybrána selekce bez substituce a je znázorněna na algoritmu níže, viz Algoritmus 5-14.

$P_r \leftarrow TournamentSelection(P, n_r, k)$		
Funkce, která na základě k zápasů mezi náhodně vybranými jedinci vybírá n_r vítězných jedinců ze seznamu P do výsledného seznamu jedinců P_r určených k další reprodukci – tento seznam je výstupem funkce. Selekce bez substituce – jedinec se může vyskytnout ve výstupním seznamu maximálně jedenkrát.		
Vstup:	P:	Seznam, ze kterého bude proveden výběr do X_{MP}
Vstup:	n_r:	Počet jedinců, kteří budou umístěni do výsledného seznamu určeného k páření
Vstup:	k:	Počet jedinců, mezi nimiž se uspořádá turnaj
Data:	$X.fit$:	Hodnota fitness jedince
Data:	a:	Index jedince, který vyhrál zápas
Data:	i:	Proměnné - čítače
Data:	j:	Počet uskutečněných zápasů vybraného jedince proti soupeřům
Výstup:	P_r:	Výsledná populace určená k páření

```
1   begin
2     P_r ⟵ ( );
3     P.Sort_a(X.fit);
4     for i ⟵ 0 to min{Length(P), n_r} − 1 do begin
5         a ⟵ ⌊Random(0,Length(P))⌋;
6         for j ⟵ 1 to min{Length(P), k} − 1 do
7             a ⟵ min{a, ⌊Random(0,Length(P))⌋};
8         P_r.AddListItem(P[a]);
9          P.DeleteListItem(P, a);
10    end;
11    Result ⟵ P_r;
12  end;
```

Algoritmus 5-14 TurnamentSelection [11]

5.4.1.4 Selekce na základě pořadí

Tato selekční strategie je velmi podobná selekci úměrné hodnotě fitness až na to, že pravděpodobnost selekce je závislá spíše na relativní hodnotě fitness než na absolutní. Jinými slovy je jedno, zda nejzdatnější kandidát je desetkrát zdatnější než jiný anebo jen 0,001 krát zdatnější. V obou případech může být selekční pravděpodobnost stejná. Jediné na čem záleží, je pozice jedince po roztřídění celé populace.

Tato strategie má tendenci zabraňovat předčasné konvergenci díky tlumení selekčního tlaku pro velké rozdíly fitness, které se vyskytují v prvních generacích. Naopak, zesilováním malých rozdílů fitness v pozdějších generacích, je selekční tlak zvýšený ve srovnání s jinou selekční strategií. [13]

Na ukázku byla vybrána selekce bez substituce a je znázorněna na algoritmu níže, viz Algoritmus 5-15)

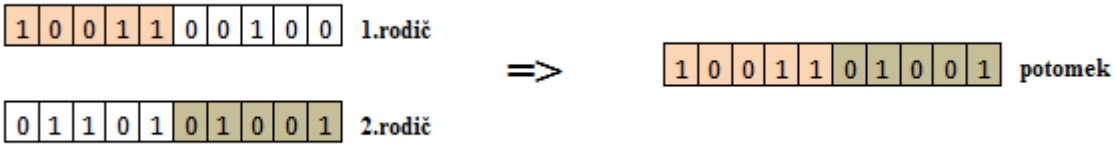
$P_r \longleftarrow RankBasedSelection(P, n_r, k)$		
Funkce, která na základě pravděpodobnosti výběru jedince, jenž je úměrná jeho pozici v setříděném seznamu jedinců populace vybírá n_r vítězných jedinců ze seznamu P do výsledného seznamu jedinců P_r určených k další reprodukci – tento seznam je výstupem funkce. Selekce bez substituce – jedinec se může vyskytnout ve výstupním seznamu maximálně jedenkrát.		
Vstup:	P:	Seznam, ze kterého bude proveden výběr do X_{MP}
Vstup:	n_r:	Počet jedinců, kteří budou umístěni do výsledného seznamu určeného k páření
Vstup:	k:	Selekční tlak - počet očekávaných potomků nejlepšího jedince, $k \neq m_{MP}$
Data:	$X.fit$:	Funkce, která poskytuje jako výstup hodnotu fitness jedince
Data:	q:	Mocnina indexu náhodně vygenerovaného jedince podle rovnoměrného rozdělení užitá pro seřazení
Data:	i:	Proměnná - čítač
Výstup:	P_r:	Výsledná populace určená k páření

```
1   begin
2     q ⟵ 1 / (1 − log k / log n_r);
3     P_r ⟵ ( );
4     P.Sort_a(X.fit);
5     for i ⟵ 0 to min{Length(P), n_r} − 1 do begin
6       j ⟵ ⌊Random_u( )^q * Length(P)⌋;
7       P_r.AddListItem(P[j]);
8       P.DeleteListItem(P, j);
9     end;
10    Result ⟵ P_r;
11  end;
```

Algoritmus 5-15 RankBasedSelection [11]

5.4.2 Křížení (Crossover)

Nyní, když pomocí selekce vybereme vhodné jedince, můžeme přejít k reprodukci. Reprodukcí se rozumí vznik nových jedinců tzv. potomků (offsprings) pomocí křížení a mutace. Proces křížení je znázorněný níže, viz Algoritmus 5-16 A pro lepší představu ještě obrázek, jak takové křížení může vypadat, viz Obrázek 5-5



Obrázek 5-5 Křížení

$P_k \leftarrow Crossover(P_r, n_k)$	
Jedná se o proceduru, kdy náhodně vybereme z populace P_r, která byla vytvořena pomocí selekce, dva jedince tzv. rodiče X_a a X_b. Zkřížením těchto rodičů získáme nového jedince $X.Gen$. Nakonec nového jedince přidáme do populace P_k.	
Vstup:	P_r: Populace, která byla pomocí selekce vybrána k reprodukci
Vstup:	n_k: Počet jedinců, kteří budou vloženi do výsledné populace n_k
Data:	y, z: Náhodně zvolené indexy rodičů
Data:	X_a, X_b: Náhodně vybraní jedinci (rodiče) z populace P_r
Data:	k: Náhodné číslo, $w \in \langle 0,1 \rangle$
Výstup:	P_k: Nově vzniklá populace jedinců (potomků), kteří vznikly zkřížením dvou jedinců z původní populace P_r

```
1   begin
2     P_k ← ( );
3     for i ← 0 to n_k − 1 do begin
4         y, z ← ⌊Random(0, Length(P_r))⌋;
5         X_a ← P_r[y];
6         X_b ← P_r[z];
7         for j ⟵ 0 to (Length (X. Gen) − 1) do begin
8             k ← ⌊Random (0,2)⌋; // vrátí 0 nebo 1
9             if k = 1 then
10            X.Gen[j] ← X_a.Gen[j];
11            else
12            X.Gen[j] ← X_b.Gen[j];
13    P_k.AddListItem (X);
14        end;
15    end;
16  end;
```

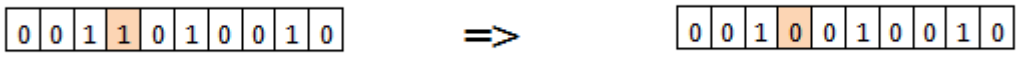
Algoritmus 5-16 Crossover

5.4.3 Mutace

Mutace je náhodná změna v genu jedince. Pro tuto změnu je charakteristická tzv. míra mutace (mutation rate). Mutace nastane přidáním náhodné hodnoty do alely. Tato změna je obvykle funkcí hodnoty fitness každého jedince. Jedinci s dobrou hodnotou fitness budou zmutováni méně, zatímco jedinci s horší hodnotou fitness budou zmutováni více. Způsobů, jak provést tuto změnu je spoustu a na ukázku jsem vybrala dva z nich.

5.4.3.1 Bit flip mutace

Tato mutace spočívá v tom, že vybereme jednu pozici a přiřadíme jí opačnou hodnotu jako je to vidět na Obrázek 5-6. Tento způsob se používá v binárním kódování GA.



Obrázek 5-6 Bit flip mutace

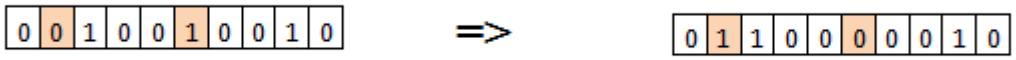
$P_k \leftarrow Bitflipmutace\ (P_k)$	
Jedná se o funkci, kdy všechny jedince z populace P_k zmutujeme. A to tak, že u každého jedince se náhodně vybere pozice alely v genu a hodnotě alely na této pozici přiřadíme opačnou hodnotu.	
Vstup:	P_k: Populace vzniklá křížením
Data:	$X.Gen$: Gen, který obsahuje určité množství alel
Data:	i: Čítač
Data:	k: Náhodně vybraná pozice alely v genu
Data:	$X.sour$: Jedinec z populace P_k
Výstup:	P_k: Populace obsahující zmutované jedince

```
1   begin
2     for i ← 0 to (Length (Pk) − 1) do begin
3         X.sour ← Pk[i];
4         k ← ⌊Random(0, Length(X.Gen))⌋ // vybere nějakou alelu z genu
5         if X.Gen[k] = 1 then
6         X.Gen[k] ← 0
7         else
8         X.Gen[k] ← 1;
9         if F(X.sour) < F(Pk[i]) then
10        Pk[i] ← X.sour;
11    end;
12  end;
```

Algoritmus 5-17 Bit flip mutace

5.4.3.2 Výměnná mutace (Swap mutation)

U této mutace vybereme v chromozomu náhodně dvě pozice a vyměníme hodnoty na těchto pozicích, jak můžeme vidět na Obrázek 5-7. Tento způsob je běžný pro permutační kódování. [14]



Obrázek 5-7 Swap mutace [13]

$P_k \leftarrow Swapmutace\ (P_k)$	
Jedná se o funkci, kdy všechny jedince z populace P_k zmutujeme. A to tak, že u každého jedince náhodně vybereme dvě pozice alel v genu a hodnoty alel na těchto pozicích vyměníme. Poté porovnáme nového jedince X s původním jedincem $P_k[i]$ a jestliže je nový jedinec lepší, tak jím nahradíme původního jedince v populaci P_k.	
Vstup:	P_k: Populace vzniklá křížním
Data:	$X.Gen$: Gen, který obsahuje určité množství alel
Data:	$Y.Gen$: Pomocný gen
Data:	i: Čítač
Data:	k: Náhodně vybraná pozice alely v genu
Data:	l: Náhodně vybraná pozice alely v genu
Data:	$X.sour$: Jedinec z populace P_k
Výstup:	P_k: Populace obsahující zmutované jedince

```
1   begin
2     for i ← 0 to (Length (Pk) − 1) do begin
```

$P_k \leftarrow Swapmutace\ (P_k)$	
Jedná se o funkci, kdy všechny jedince z populace P_k zmutujeme. A to tak, že u každého jedince náhodně vybereme dvě pozice alel v genu a hodnoty alel na těchto pozicích vyměníme. Poté porovnáme nového jedince X s původním jedincem $P_k[i]$ a jestliže je nový jedinec lepší, tak jím nahradíme původního jedince v populaci P_k.	
Vstup:	P_k: Populace vzniklá křížním
Data:	$X.Gen$: Gen, který obsahuje určité množství alel
Data:	$Y.Gen$: Pomocný gen
Data:	i: Čítač
Data:	k: Náhodně vybraná pozice alely v genu
Data:	l: Náhodně vybraná pozice alely v genu
Data:	$X.sour$: Jedinec z populace P_k
Výstup:	P_k: Populace obsahující zmutované jedince

```
3          X.sour ← P_k[i];
4          repeat
5             k ← ⌊Random(0, Length(X.Gen) − 1)⌋;
6             l ← ⌊Random(0, Length(X.Gen) − 1)⌋;
7          until k ≠ l;
8             X.Gen[k] ← Y.Gen[k];
9             X.Gen[k] ← X.Gen[l];
10            X.Gen[l] ← Y.Gen[k];
11         if F(X.sour) < F(P_k[i]) then
12            P_k[i] ← X.sour;
13      end;
14   end;
```

Algoritmus 5-18 Swap mutace

6 Využití v praxi

Memetické algoritmy lze využít pro řešení mnoha optimalizačních úloh. Jako příklad bude uveden Job Shop Scheduling problem. Název je uváděn v angličtině, ale jedná se o problém plánování nebo rozvržení pracovišť.

6.1 Job Shop Scheduling problem

Job Shop Scheduling problem je dobře známý jako jeden z nesložitějších optimalizačních problémů a hraje velmi důležitou roli v moderním výrobním průmyslu. Velmi složitý je kvůli jeho velmi rozsáhlému prohledávanému prostoru a mnoha omezením mezi stroji a prácemi. Patří do skupiny NP problémů, které byly popsány v první části práce.

Tento problém byl již řešen pomocí různých exaktních metod a dynamického programování. Nicméně většinou nebylo možné nalézt řešení v nějakém rozumném výpočetním čase. V nedávných letech je více pozornosti zaměřeno na přibližné metody jako heuristické a meta-heuristické metody. Přibližné metody mohou nalézt dobré kvalitní plány (blízké optimálnímu řešení) v rozumném výpočetním čase. Vznikne tedy dobrý kompromis mezi kvalitou řešení a výpočetním časem.

U memetických algoritmů hraje Local Search významnou roli. Genetické algoritmy jsou využity k průzkumu (exploration) a Local Search využije znalostí z předchozího průzkumu (exploitation), tím je dosaženo rovnováhy mezi intenzifikací a diverzifikací. [15]

6.2 Popis problému

Klasický JSP může být popsán následovně:

Máme n různých prací $\{Job_1, Job_2 \dots Job_n\}$ a m různých strojů $\{M_1, M_2 \dots M_m\}$. Každá práce se skládá z množiny operací a každá operace vyžaduje jiný stroj. Všechny operace každé práce jsou zpracovávány v pevném pořadí. A každá operace má daný procesní čas. Cílem je minimalizace makespanu, to je maximální čas pro zpracování všech prací.

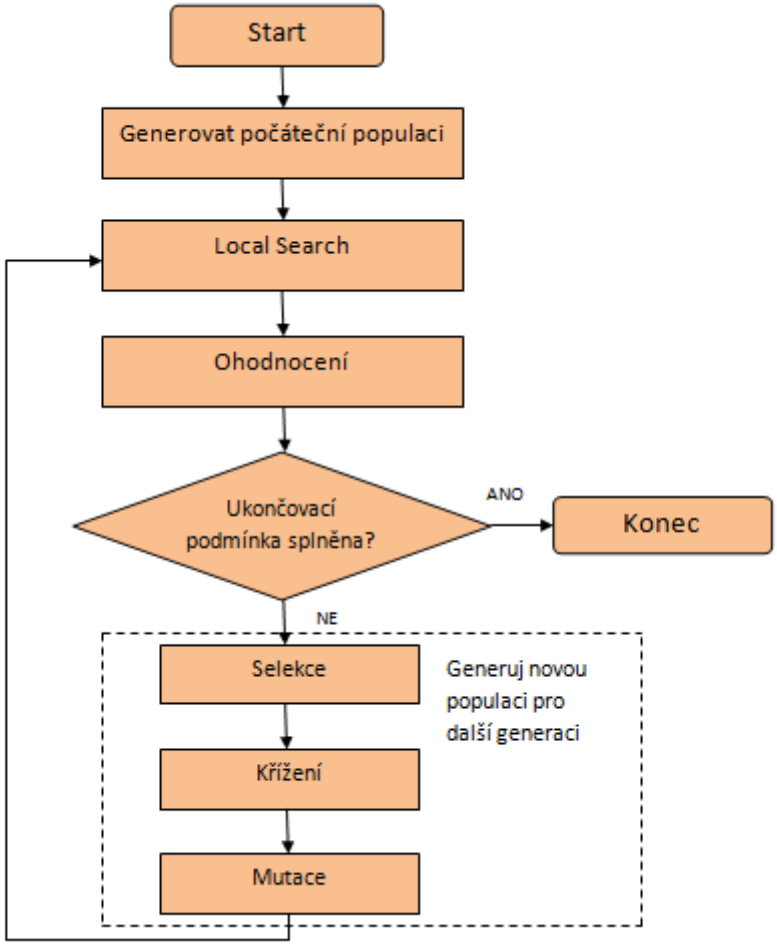
Podmínky:

- Každý stroj může zpracovávat nejvíce jednu práci najednou. A pouze pokud potřebný stroj je nevyužitý, může být operace naplánována.
- Každá práce je zpracována pouze jedním strojem. A každá operace může být naplánována, až když jsou všechny předchozí potřebné operace naplánované.
- Pořadí strojů, na kterých je vykonávána práce je pevně dané.
- Všechny práce musí být zpracovány pomocí každého stroje pouze jednou a je zde nejvíce m operací pro práci.
- Nejsou zde žádné přednostní omezení mezi operacemi a různými prácemi.
- Stroje jsou vždy dostupné a nikdy se nerozbijí.
- Procesní čas všech operací je známí. [15]

6.3 Navržený memetický algoritmus

Jak již bylo řečeno dříve, memetický algoritmus může dobře balancovat diverzifikaci a intenzifikaci k nalezení vysoce kvalitního řešení optimalizačního problému. Diverzifikace je prohledávání různých oblastní prohledávaného prostoru k nalezení nejslibnější oblasti.

Intenzifikace je prohledávání sousedství jedinců k produkování lepšího řešení. V navrženém memetickém algoritmu je Local Search aplikováno na každého jedince k nalezení lepšího řešení. Na obrázku níže je znázorněn vývojový diagram navrženého memetického algoritmu. [15]



6-1 Vývojový diagram navrženého MA [15]

6.4 Geny

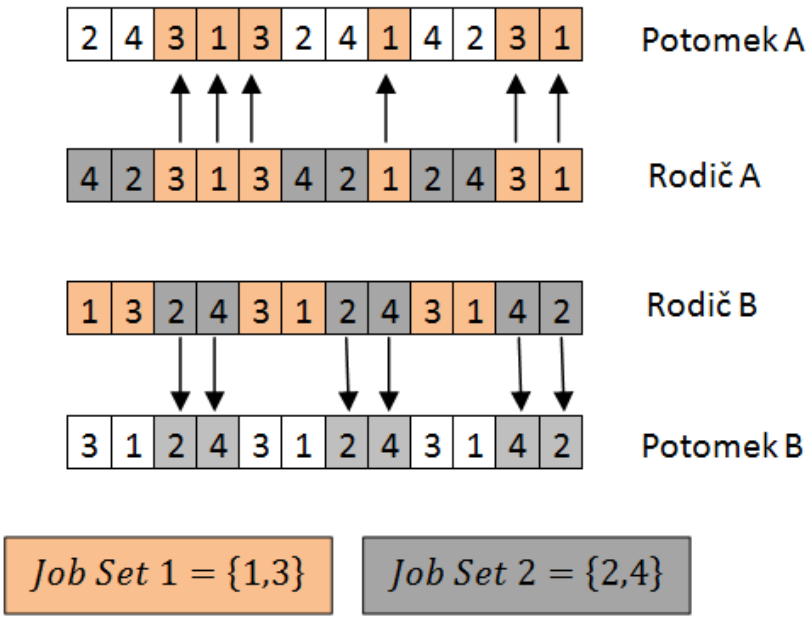
Každý gen má $n * m$ alel a každá alela je celé číslo. Počet různých celých čísel je roven počtu prací n. Práce reprezentuje množinu operací, které musí být naplánované na m strojích a každá práce je reprezentována m časy v rámci genu. Projetím genu zleva doprava, i-tý výskyt práce odkazuje na i-tou operaci, pak gen může být dekódovaný do plánu nepřímo. Vezmeme-li v úvahu 3 práce, každá z nich má 3 operace. Dostaneme gen [2 3 1 2 2 1 3 1 3], kde {1 2 3} označují korespondující práci $\{Job_1\ Job_2\ Job_3\}$. Jsou zde 3 různá celá čísla, každé je zopakováno třikrát. Zleva doprava, první alela 2 reprezentuje první operaci druhé práce, která má být zpracována první na příslušném stroji. Poté, druhá alela 3 představuje první operaci třetí práce. Nakonec, gen [2 3 1 2 2 1 3 1 3] je zaznamenán jako $[O_{21}\ O_{31}\ O_{11}\ O_{22}\ O_{23}\ O_{12}\ O_{32}\ O_{13}\ O_{33}]$, kde O_{ij} označuje i-tou operaci j-té práce. [15]

6.5 Genetické algoritmy

Selekce, křížení a mutace jsou nejdůležitějšími operátory pro reprodukci nové populace. Mají významný vliv na kvalitu řešení.

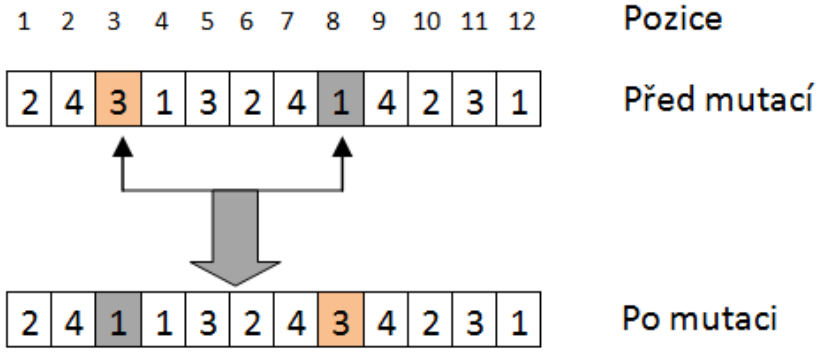
a) Selekce- V tomto případě byla zvolena turnajová selekce. Turnajovým přístupem je možné získat dobré jedince, kteří mají větší šanci na přežití, vyhýbající se vlivu předčasné konvergence.

b) Křížení- Operátor křížení je použit k získání lepšího chromozomu výměnou informace aktuálního řešení. V chromozomu jsou všechny práce náhodně rozděleny na dvě množiny, tj. $Job\ Set\ 1$ a $Job\ Set\ 2$. Jakýkoliv gen, který patří do množiny $Job\ Set\ 1$ v rodičovi A bude zachován na stejné pozici u potomka A. Stejným způsobem, jakýkoliv gen patřící do $Job\ Set\ 2$ v rodičovi 2 bude zachován na stejné pozici u potomka B. Poté, zbytek prázdných pozic u potomka A bude naplněn geny, které patří do množiny $Job\ Set\ 2$ v rodičovi B v jejich pořadí prezentovaném u rodiče B. Stejná metoda je použita k naplnění potomka B.



6-2 Příklad křížení [15]

c) Mutace- Procedura mutace je demonstrována na obrázku 6-3. Pozice 3 a 8 jsou náhodně vybrány. Hodnoty genů na těchto pozicích jsou vyměněny. Jedná se o Swap mutaci, která byla popsána v kapitole 5.4.3.2. [15]



6-3 Příklad mutace [15]

6.6 Local Search

Local Search je nezbytnou součástí memetických algoritmů, aby vylepšilo kvalitu řešení. Tato procedura může efektivně využít informace v sousedství. V našem případě jsou sousední řešení generovány hlavně na základně kritické cesty. Je zde požita nová strategie Local Search, která je založena na výměně a vkládání. Výměna znamená, že náhodně zvolíme dvě alely a poté je vyměníme, vkládání znamená, že vybereme náhodně alelu a vložený bod a poté vložíme vybranou alelu před vybraný bod. [15]

```
procedure: the proposed local search
input: one individual s from population
begin
    while ( terminated condition not met ) do
        k = 1
        s' = Exchange(s)
        while ( k≤2 ) do
            if ( k=1 ) then
                s'' = Insert (s')
            if ( k=2 ) then
                s'' = Exchange (s')
            if ( Cm(s'')<Cm(s') ) then
                s' = s''
            else
                k = k+1
        end while
        if ( Cm(s'')<Cm(s) ) then
            s = s''
    end while
end
```

Obrázek 6-4 pseudokód navrženého local search [15]

7 Závěr

Tématem této práce byl memetický algoritmus využitelný v rámci diskrétní simulační optimalizace. V první části práce jsou vysvětleny pojmy diskrétní simulace a optimalizační úlohy. V další části je popsán obecný memetický algoritmus a dále je většina práce věnována vysvětlení a popsání jednotlivých částí memetického algoritmu. Tyto jednotlivé části algoritmu jsou popsány i pomocí pseudokódů, podle kterých by mělo být možné programovat. V práci jsou jednotlivé algoritmy popsány tak, jak fungují samostatně, tudíž pro sestavení konkrétního memetického algoritmu je potřeba provést malé změny v jednotlivých algoritmech, aby se daly nakombinovat.

Nakonec je uveden příklad využití v praxi. Byl vybrán jeden z nejznámějších NP problémů a to Job Shop Scheduling problem. Nejdříve byl popsán problém jako takový a dále jsou popsány jednotlivé navržené části jako je selekce, křížení, mutace a Local Search.

Jelikož neovládám žádný programovací jazyk, byla tato práce pouze teoretická. Dalším krokem by ale jinak bylo otestovat různé varianty memetického algoritmu na několika testovacích funkcích. Výsledkem by bylo porovnání různých variant. Posledním krokem by pak bylo nejlepší variantu aplikovat na nějaký konkrétní příklad z praxe.

Citovaná literatura

[1] P. RAŠKA a Z. ULRYCH, *„Simulace výrobních linek,“* 2014. [Online]. Dostupné z: https://www.systemonline.cz/rizeni-vyroby/simulace-vyrobnich-linek.htm. [Přístup získán 29 Květen 2017].

[2] *„Diskrétní simulace,“* [Online]. Dostupné z: http://kam.mff.cuni.cz/~kuba/vyuka/programovani/xaver/diskretni_simulace.html. [Přístup získán 8 Listopad 2016].

[3] L. FIŘTOVÁ, *„Principy diskrétní simulace,“* 2014. [Online]. Dostupné z: http://www.jakplavejak.cz/sites/default/files/prilohy/4A_SIMUL_fin.pdf. [Přístup získán 29 Květen 2017].

[4] M. OBITKO, *„Introduction of Genetic Algorithms,“* 1998. [Online].Dostupné z: http://www.obitko.com/tutorials/genetic-algorithms/search-space.php. [Přístup získán 3 Březen 2017].

[5] *„Lokální extrémy tabulkovou metodou,“* Univerzita Karlova v Praze, 2011. [Online].Dostupné z: http://kdm.karlin.mff.cuni.cz/diplomky/karel.trnka/derivace/?page=34lokextremtm. [Přístup získán 10 Březen 2017].

[6] K. CLEGG, *„An Introduction to Evolution for Computer Scientists,“* 22 Leden 2008. [Online]. Dostupné z: https://www.researchgate.net/figure/240928090_fig13_Figure-13-Schematic-adaptive-or-fitness-landscape. [Přístup získán 1 Červen 2017].

[7] *„Memetic Algorithms,“* [Online]. Dostupné z: http://www.lcc.uma.es/~ccottap/papers/IntroMAs.pdf. [Přístup získán 10 Říjen 2016].

[8] L. STAŇKOVÁ, *„GA a předčasná konvergence,“* [Online]. Dostupné z: http://slideplayer.cz/slide/2751753/. [Přístup získán 6 Březen 2017].

[9] *„Memetic Algorithms,“* [Online]. Dostupné z: http://www.powershow.com/view/2524ae-MzZlM/Paper_Review_for_ENGG6140_Memetic_Algorithms_powerpoint_ppt_presentation. [Přístup získán 10 Říjen 2016].

[10] *„Genetic Algorithm to Constraint Satisfaction Problems,“* [Online]. Dostupné z: http://slideplayer.com/slide/7285101/. [Přístup získán 3 Říjen 2016].

[11] P. RAŠKA, *„Optimalizační metody pro diskrétní simulaci výrobních systémů a výrobních procesů ve strojírenství*“. Plzeň 2012 Disertační práce. Západočeská univerzita v Plzni. Vedoucí práce Doc.Ing. Václav Votava, CSc.

[12] Č. ŠANDERA, *„Hybridní model metaheuristických algoritmů,“* 2013. [Online]. Dostupné z: https://dspace.vutbr.cz/xmlui/bitstream/handle/11012/25357/Sandera_disertace.pdf?sequence=1&isAllowed=y. [Přístup získán 11 Duben 2017].

[13] D. W.DYER, *„Selection Strategies,“* [Online]. Dostupné z: http://watchmaker.uncommons.org/manual/ch03.html. [Přístup získán 9 Listopad 2016].

[14] *„Genetic Algorithms Mutation,“* [Online]. Dostupné z: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_mutation.htm. [Přístup získán 8 Listopad 2016].

[15] G. ZHANG, *„http://ieeexplore.ieee.org,“* 3 Prosinec 2010. [Online]. Dostupné z: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5657180. [Přístup získán 7 Listopad 2016].