UNIVERZITA KOMENSKÉHO V BRATISLAVE

FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

HEURISTIKY A METAHEURISTIKY

Bakalárska práca

2015 **Martin Mašurik**

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

HEURISTIKY A METAHEURISTIKY

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	prof. RNDr. Pavol Ďuriš, CSc.

Bratislava, 2015 **Martin Mašurik**

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Martin Mašurik
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Heuristiky a metaheuristiky
Heuristics and metaheuristics

Cieľ: Vypracovať prehľad heuristík a metaheuristík používaných v rôznych aplikačných oblastiach,
určiť ich pozitívne a negatívne vlastnosti a porovnať tieto metódy navzájom.

Vedúci: prof. RNDr. Pavol Ďuriš, CSc.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 28.10.2013

Dátum schválenia: 28.10.2013

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

ČESTNÉ VYHLÁSENIE

Čestne vyhlasujem, že som bakalársku prácu „Heuristiky a metaheuristiky" vypracoval samostatne s použitím uvedenej literatúry a zdrojov dostupných na internete.

V Bratislave dňa 29.5.2015

…….....

Martin Mašurik

POĎAKOVANIE

Ďakujem môjmu školiteľovi prof. RNDr. Pavlovi Ďurišovi, CSc za ochotu, cenné rady a usmernenie pri písaní záverečnej práce.

ABSTRAKT

MAŠURIK, Martin: Heuristiky a metaheuristiky. [Bakalárska práca] – Univerzita Komenského v Bratislave. Fakulta matematiky fyziky a informatiky; Katedra informatiky. – Vedúci: prof. RNDr. Pavol Ďuriš, CSc. Bratislava: UK, 2015. 42 strán.

Cieľom tejto práce je vytvoriť prehľad najznámejších metaheuristík, pozrieť sa na ich charakteristické vlastnosti a porovnať ich s triedou algoritmov nazývaných heuristiky. Keďže v niektorých literatúrach sa tieto názvy zamieňajú, ďalším cieľom by mohlo byť vyjasnenie definícií a rozdielov medzi týmito dvoma kategóriami algoritmov. Hlavná časť tejto práce sa venuje dôkladnému popisu komplexných metaheuristických algoritmov z ktorého vyplýva, že sú oveľa flexibilnejšie a použiteľnejšie na riešenie problémov reálneho sveta než jednoduchšie heuristické metódy.

Kľúčové slová: optimalizácia, optimalizačný problém, heuristiky, metaheuristiky

ABSTRACT

MAŠURIK, Martin: Heuristics and metaheuristics. [Bachelor's thesis] – Comenius University in Bratislava. Faculty of mathematics, physics and informatics; Department of computer science. – Supervisor: prof. RNDr. Pavol Ďuriš, CSc. Bratislava: UK, 2015. 42 pages.

The aim of this work is to compile a list of the most well known metaheuristics, take a closer look at their features and compare them to a class of algorithms most well known as heuristics. Since in some literature the names of these two classes of algorithms are used interchangeably it is also a goal to make a clearer distinction between them. The main section of this work is devoted to the thorough description of the complex metaheuristic algorithms that provides reasons to believe that they are vastly more flexible and applicable to real life problems than the simpler heuristic methods.

Keywords: optimization, optimization problem, heuristics, metaheuristics

OBSAH

Úvod

Slovo heuristika, ako mnoho iných cudzích slov, má korene v grécku. Zdieľa spoločný základ so slovom „heuréka“, slávnym výkrikom vedcov a objaviteľov pri objavení niečoho geniálneho. V gréčtine sa to slovo používa v tvare „*heuriskó*“ a znamená to jednoducho „nájsť“ alebo „objaviť“. Heuristiky a metaheuristiky presne to robia. Nachádzajú alebo objavujú riešenia problémov a to často veľmi geniálnym spôsobom.

Slovo heuristika sa okrem informatiky vyskytuje aj v jednom ďalšom akademickom obore - v psychológii. Ľudský mozog počas miliónov rokov evolúcie tiež prišiel na ich úžasnú schopnosť nájsť akési skratky na urýchlenie riešenia problémov.

Poskladaním a zovšeobecnením viacerých úspešných heuristík vznikla všeobecnejšia trieda algoritmov – metaheuristiky. Kým heuristiky by sme mohli považovať za veľmi jednoduché procedúry na riešenie problémov, metaheuristiky sú niečo ako krok smerom k všeobecnej ľudskej inteligencii. Ktovie aká bude ďalšia fáza vývoja algoritmov riešiacich problémy?

V tejto práci sa však pozrieme bližšie na súčasný stav heuristík a metaheuristík, na akom princípe sú založené, a vymenujeme si zopár najznámejších.

1. Prehľad problematiky

1.1 Základné definície a špecifikácie

1.1.1 Optimalizačný problém

S optimalizačnými problémami sa často stretávame v mnohých oblastiach – či už vedeckého, technického, alebo ekonomického zamerania. Formálne sa optimalizačný problém dá zadefinovať ako dvojica (S, f), v ktorej S reprezentuje množinu všetkých potenciálnych *riešení* (stavov, konfigurácií) a f je zobrazenie $f : S \to \mathbf{R}$. Ďalej budeme f nazývať *objektívnou funkciou* (v literatúre sa tiež často nazýva *fitness* alebo *utility* funkciou)[2]. Objektívna funkcia teda priraďuje ku každému potenciálnemu riešeniu určité reálne číslo, čím je vhodnosť daného riešenia kvantifikovaná a je umožnené aby sa rôzne riešenia mohli podľa vhodnosti navzájom porovnávať.

1.1.2 Globálne optimum

Ďalej je potrebné zadefinovať pojem *globálneho optima*. Globálne optimum je také riešenie $s^* \in S$ pre ktoré platí: $\forall s \in S : f(s^*) \leq f(s)$, teda je to riešenie, ktoré je ohodnotené objektívnou funkciou ako najlepšie spomedzi všetkých riešení. Riešenie, ktoré nie je globálnym optimom nazývame *suboptimálnym* riešením. Hlavným cieľom optimalizačných problémov je teda nájsť globálne optimálne riešenie (v prípade, že je takých riešení viac – nájsť množinu všetkých globálnych optimálnych riešení).

1.1.3 Stavový priestor a search landscape

Každý algoritmus určený na nájdenie riešenia pre určitý typ problému by sme mohli charakterizovať ako spôsob prehľadávania množiny S. Na to, aby také prehľadávanie bolo umožnené, je potrebné splniť tri predpoklady[3]:

1. Reprezentácia stavu – každé riešenie musíme byť schopní zakódovať pomocou určitých symbolov alebo štruktúr do formátu v ktorom budú v množine S reprezentované.
2. Operátory - nástroje, ktoré dokážu pretransformovať jedno riešenie na druhé aby prehľadávanie mohlo prebiehať systematicky.

3. Systém pre efektívne plánovanie týchto transformácií, aby sme sa k výsledku dopracovali čo najrýchlejšie.

Množina všetkých stavov *S* spolu s operátormi (ktoré v podstate prepájajú jednotlivé stavy) vytvárajú konečný orientovaný graf nazývaný *stavový priestor* (tiež stavový graf, prehľadávací priestor).

Dvojicu (G, f), kde *G* je stavový priestor a *f* je objektívna funkcia, nazývame *fitness landscape* („krajina vhodnosti“)(alebo tiež *search landscape* – krajina prehľadávania). Preto je v názve krajina, lebo stavový priestor si môžeme predstaviť ako mapu, kde bod na každej súradnici predstavuje jeden konkrétny stav a objektívna funkcia určuje jeho nadmorskú výšku. Fitness landscape sa dá tým pádom opísať pojmami z geografie a môžeme hovoriť o dolinách, nížinách, kopcoch, plošinách a tak ďalej.

Jeden zo spôsobov akým sa nájdenie globálneho optima dá realizovať je prehľadávanie v stavovom priestore pri ktorom je skontrolované každé možné riešenie. Tento postup uplatňujú takzvané *exaktné* algoritmy. Pri mnohých optimalizačných problémoch (najmä tých, ktoré sú kombinatorického charakteru) je však stavový priestor enormne veľký. Exaktné algoritmy sú v prípade takýchto problémov neefektívne a síce sú schopné zaručiť, že riešenie, ktoré objavia bude skutočne globálnym optimom, takáto záruka nekompenzuje fakt, že sa daného riešenia s veľkou pravdepodobnosťou nedočkáme. Je zjavné, že potrebujeme iný prístup – taký, pri ktorom síce budeme nútený uspokojiť sa so suboptimálnym riešením, budeme ním disponovať za omnoho kratší čas.

1.2 Heuristické algoritmy

Heuristiky a metaheuristiky predstavujú jednu podskupinu kategórie algoritmov určených na riešenie optimalizačných problémov. Medzi ich silné stránky patria: vysoká rýchlosť a možnosť uplatnenia na širokú škálu problémov. Na druhej strane však nedokážu garantovať nájdenie globálneho optima. Vo väčšine prípadov sú založené na určitej pomerne jednoduchej myšlienke (stratégii) o ktorej sa predpokladá, že by nás mohla priviesť k prijateľnému riešeniu – k takému, ktoré sa od globálneho optima líši v akceptovateľnej miere. Ich efektívnosť nie je formálne dokázaná; tiež je nemožné ohraničiť odklon riešení od globálneho optima (ak by to možné bolo, jednalo by sa o tzv. *aproximačné* algoritmy). Rozdiel medzi heuristikami a metaheuristikami spočíva v tom, že kým heuristiky sú algoritmy akoby „ušité na mieru“ pre jeden špecifický problém, metaheuristiky reprezentujú všeobecnejší typ algoritmov – je ich možné aplikovať na riešenie takmer akéhokoľvek optimalizačného problému. Metaheuristiky teda môžeme vnímať ako všeobecné metódy riešenia problémov a vieme ich použiť aj ako akési vodidlo na zostrojenie konkrétnej heuristiky určenej na riešenie špecifických optimalizačných problémov. V tomto smere by sa dali prirovnať k návrhovým vzorom.

Pod heuristikou môžeme rozumieť akýkoľvek návrh spôsobu riešenia jedného konkrétneho problému, ktorý je založený iba na akomsi jednoduchom pravidle o ktorom sa odhaduje, že by nás mohlo priviesť k dobrým výsledkom. V anglickej literatúre sa často používa výraz *rule of thumb* – pravidlo „podľa palca na ruke“.

Heuristiky a metaheuristiky môžeme nazývať spoločným názvom *heuristické algoritmy* (toto pomenovanie zavádzame aby sa predišlo zbytočným konfúziám, keďže v staršej literatúre pojem heuristika zahŕňal aj metaheuristiky; v tejto práci sa pod heuristikou rozumie výhradne algoritmus navrhnutý pre špecifický problém – konkrétnu implementáciu metaheuristiky by sme teda tiež mohli nazvať heuristikou).

1.2.1 Lokálne optimum

Aj heuristické algoritmy sú založené na princípe prehľadávania stavového priestoru. Na rozdiel od exaktných algoritmov však neprechádzajú celým stavovým priestorom a preto sa kladie obzvlášť veľký dôraz na uprednostnenie lepších riešení pri prehľadávaní. Keďže sa nekontroluje každé riešenie, môže sa stať, že náš algoritmus uviazne v *lokálnom optime*. Na vysvetlenie pojmu lokálne optimum potrebujeme najprv zadefinovať, čo je to *okolie* jedného riešenia (stavu). Funkcia okolia N je zobrazenie $N : S \rightarrow 2^S$ ktoré ku každému

riešeniu *s* z *S* priradí množinu riešení $N(s) \subset S$. Riešenie s' z okolia *s* ($s' \in N(s)$) sa nazýva *susedom s*. Množina *N(s)* je určená operátormi, ktoré miernou úpravou jedného riešenia dokážu vygenerovať ďalšie. V stavovom priestore teda susedia riešenia *s* sú tie riešenia ktoré sú s *s* prepojené hranou, prípadne sa od *s* nachádzajú do určitej vopred určenej vzdialenosti ε. Formálne je to množina riešení $\{s' \mid d(s', s) \leq \varepsilon\}$ kde *d* je funkcia vracajúca vzdialenosť dvoch riešení v stavovom priestore.

Lokálne optimum je riešenie ktoré je objektívnou funkciou najlepšie ohodnotené zo svojho okolia, ale zároveň nemusí byť nutne globálnym optimom.

Formálne je to teda riešenie *s* pre ktoré platí $f(s) \leq f(s')$ pre všetky $s' \in N(s)$.

Jednoduché heuristické algoritmy majú tendenciu uviaznuť v lokálnych optimách a preto sa vyvinuli mnohé opatrenia ktoré sa snažia tomuto problému vyhnúť. Dobré metaheuristiky sú zväčša vyzbrojené aj viacerými mechanizmami, ktoré slúžia na to, aby sa im včas podarilo uniknúť z hrozby lokálnych optím.

2. Heuristiky

Keďže heuristiky sú navrhnuté pre špecifické problémy, ukážeme si príklady heuristík navrhnutých na riešenie konkrétneho optimalizačného problému.

Asi najznámejší zo všetkých optimalizačných problémov je *problém obchodného cestujúceho*. Obchodný cestujúci chce nájsť najkratšiu trasu medzi *n* mestami a vrátiť sa späť do mesta v ktorom začínal. Ide teda o nájdenie najkratšej hamiltonovskej kružnice v grafe. Existuje viacero algoritmov na nájdenie optimálnej trasy, pri väčších grafoch však zlyhávajú, lebo počet všetkých ciest narastá exponenciálne.

Heuristiky na riešenie problému obchodného cestujúceho môžeme rozdeliť do dvoch kategórií[6]:

- **Konštruktívne** – postupne skladajú riešenia a keď nejaké nájdu, ukončia sa.
- **Zlepšujúce** – vygenerujú nejaké riešenie a postupnými úpravami ho zlepšujú.

2.1 Konštruktívne heuristiky

2.1.1 Najbližší sused

Medzi najjednoduchšie konštruktívne heuristiky patrí heuristika *najbližšieho suseda* (*nearest neighbor*). Tá vytvára trasu tak, že za ďalšie mesto ktoré navštívi si vyberá to, ktoré je najbližšie k tomu v ktorom práve je.
Popis algoritmu:

1. Náhodne si zvolíme jedno z miest.
2. Nájdeme najbližšie nenavštívené mesto a pôjdeme tam.
3. Ak ešte ostávajú nejaké nenavštívené mestá, opakujeme krok 2.
4. Vrátime sa do prvého mesta.

2.1.2 Pažravá heuristika

Ďalšia heuristika je *pažravá heuristika* (*greedy heuristic*). Pažravá heuristika postupne vyberá najkratšie hrany medzi vrcholmi a tvorí z nich trasu. Samozrejme, do trasy nepridá dvakrát tú istú hranu, tiež nepridá hranu ak by sa pritom vytvorila kružnica s menším počtom hrán než počet všetkých miest *n*, alebo ak by to spôsobilo, že stupeň niektorého z vrcholov by bol väčší než 2.

Pažravá heuristika:

1. Usporiadame všetky hrany medzi mestami.
2. Vyberieme si najkratšiu z nich a ak neporušuje niektorú z horeuvedených podmienok, pridáme ju do trasy.
3. Ak ešte nemáme *n* vrcholov v našej trase, opakujeme krok 2.

2.2 Zlepšujúce heuristiky

2.2.1 Algoritmy 2-opt a 3-opt

Zlepšujúce heuristiky sú založené na postupnom vylepšovaní trasy. Menšími úpravami vedia z jedného riešenia vytvoriť iné. Pod menšou úpravou sa myslí napríklad odstránenie niektorých hrán z riešenia a ich nasledovné nahradenie inými hranami. Heuristika nazývaná *2-opt algoritmus* nahrádza takýmto spôsobom 2 hrany. Dve trasy, ktoré nám vzniknú vynechaním dvoch hrán sa dajú znovu prepojiť už len jedným spôsobom (aby výsledok bol stále hamiltonovská kružnica) a takéto prepojenie vykonáme ak bude lepšie než pôvodné riešenie. Tento proces opakujeme pokiaľ je možné nájsť takú dvojicu hrán, že výsledné riešenie bude lepšie než pôvodné. Keď algoritmus skončí, o výslednej trase hovoríme, že je *2-optimálna*.

Algoritmus 3-opt je založený na rovnakom princípe, akurat vynecháva 3 hrany a nie 2. Nemusíme sa zastaviť len pri čísle 3. Pokojne môžeme zostrojiť *k-opt algoritmus*, akurát operácia nahradenia strán sa pri väčších *k* stáva čoraz náročnejšou na čas.

2.2.2 Lin-Kernighanov algoritmus

Lin a Kernighan zostrojili variáciu na k-opt algoritmus (*Lin-Kernighanov algoritmus*) ktorý pri sa pri každej iterácii rozhoduje, aké *k* si má vybrať. Je síce pomalší než obyčajný 2-opt algoritmus, dosahuje však oveľa lepšie výsledky.

3. Metaheuristiky

Predpona meta- v názve je v prípade metaheuristík zavádzajúca. Nejedná sa totiž o heuristiky na výber vhodných heuristík (heuristiky na výber heuristík tvoria kategóriu algoritmov samú o sebe, nazvanú *hyperheuristiky*).
Spoločným znakom metaheuristík je, že predpokladajú veľmi málo o danom probléme a môžeme ich preto vnímať ako čierne skrinky do ktorých stačí vložiť zadanie problému a metaheuristika nám nájde akceptovateľné riešenie.

Využívajú rozmanité prístupy k problému a veľmi často je ich základná myšlienka inšpirovaná nejakým prírodným javom alebo procesom (napríklad genetické algoritmy sú inšpirované procesom evolúcie). Mnohé metaheuristiky prehľadávajú stavový priestor spôsobom, ktorý vo veľkej miere využíva prvok náhody. Často sa preto zaraďujú k *stochastickým* optimalizačným algoritmom.

3.1 Intenzifikácia a diverzifikácia

Pri všetkých metaheuristikách je prítomný konflikt dvoch protichodných princípov: princíp *intenzifikácie* a princíp *diverzifikácie*.
Princíp intenzifikácie znamená zamerať sa na jedno dobré riešenie a jemnými úpravami ho zlepšovať – prehľadávať stavový priestor v čo najbližšom okolí riešenia. Zameriava sa pri ňom pozornosť na také oblasti stavového priestoru, ktoré sa zdajú byť sľubné. Sľubné oblasti sa určujú na základe už nájdených dobrých riešení.

Na druhej strane princíp diverzifikácie hlása preskúmať čo najväčšie množstvo rôznych riešení. Teda v priestore prehľadávame zatiaľ nepreskúmané oblasti.

3.1.1 Náhodné prehľadávanie

Extrémny prípad algoritmu reprezentujúceho princíp diverzifikácie je *random search* – náhodné prehľadávanie. Mohli by sme ho zaradiť k heuristikám, aj keď je zrejmé, že sa jedná o pomerne neefektívny spôsob nájdenia riešenia. Pracuje nasledovným spôsobom:

1: Na začiatku si vygenerujeme náhodné riešenie *s* zo stavového priestoru.
2: Kým sa nenaplní určitá podmienka zastavenia (napr. prebehne určitý počet iterácií, alebo dosiahneme určitú úroveň vhodnosti) opakuje sa toto:
3: Vygenerujeme ďalšie náhodné riešenie s'.

4: Ak platí $f(s') \leq f(s)$, potom $s = s'$.

5: Na konci vrátime riešenie s.

3.1.2 Lokálne prehľadávanie

Filozofiu intenzifikácie na opačnom konci spektra zhmotňuje *local search* – lokálne prehľadávanie. Local search potrebuje už aj funkciu na vygenerovanie susedov jedného riešenia. O tom, že ktoré riešenie začne skúmať ďalej sa rozhoduje lokálne – iba na základe riešení nájdených v okolí kandidátneho riešenia. Ak je nakonfigurované tak, aby sa rozhodlo pre riešenie z okolia ktoré je lepšie ohodnotené objektívnou funkciou než súčasné najlepšie riešenie, jedná sa o jednu z najjednoduchších metaheuristík zvanú *hill climbing* (o ňom viac v ďalšej časti práce).

V každej metaheuristike sa dajú rozpoznať črty oboch týchto stratégií aj keď vo väčšine prípadov ich vplyv je vyvážený a nezachádza sa až do takýchto extrémov. Miera využitia princípu intenzifikácie (resp. diverzifikácie) rozdeľuje metaheuristiky do dvoch kategórií: metaheuristiky využívajúce **prehľadávanie založené na jednom riešení** a metaheuristiky využívajúce **prehľadávanie založené na populácii.**

Pri algoritmoch založených na jednom riešení sa naraz pracuje iba s jedným riešením a majú tým pádom sklon viac využívať princípy intenzifikácie.

Algoritmy založené na populácii pracujú naraz s celou populáciou riešení a využívajú najmä princípy diverzifikácie.

3.2 Hill Climbing

Hill climbing („lezenie na kopec“; nazýva sa aj horolezeckým algoritmom) by sa mohol považovať za najjednoduchšiu metaheuristiku. Nazýva sa hill climbing, lebo aj horolezec, ak má v pláne vyliezť na najvyšší vrchol kopca, vyberá si zväčša cestu, ktorá stúpa a nie klesá a teda ho posunie smerom do výšky – bližšie k svojmu cieľu. Je to v podstate greedy local search a pracuje nasledovným spôsobom[1]:

1: Nastavíme *s* (ktoré bude naším kandidátnym riešením) na nejaké riešenie z priestoru (ľubovoľné; môžeme vybrať napríklad náhodne).
2: Opakujeme, kým *s* nie je globálne optimum alebo nám vypršal čas:
3: Miernou úpravou *s* získame jeho suseda s'.
4: Ak platí $f(s') \leq f(s)$, potom $s = s'$.
5: Vrátime riešenie *s*.

Je zrejmé, že takýto algoritmus je vhodný na nájdenie lokálneho optima ale má tendenciu uviaznuť v ňom.

Všimnime si, že vždy generujeme len jedného suseda, na ktorého prejdeme ak je lepší než súčasné riešenie. Agresívnejšia verzia hill climbing-u vygeneruje naraz viacerých susedov a z nich si vyberá toho najlepšieho:

1: Určíme si *n* ako počet susedov na ktorých sa chceme pozrieť.
2: Nastavíme *s* na nejaké riešenie z priestoru.
3: Opakujeme, kým *s* nie je globálne optimum alebo nám vypršal čas:
4: Miernou úpravou *s* získame jeho suseda s'.
5: Opakujeme $n-1$ krát:
6: Miernou úpravou *s* získame jeho suseda *t*.
7: Ak platí $f(t) \leq f(s')$, potom $s' = t$.
8: Ak platí $f(s') \leq f(s)$, potom $s = s'$.
9: Vrátime riešenie *s*.

Ako sme aj v predchádzajúcej časti práce spomínali, hill climbing by sa dal považovať za extrémny prípad intenzifikácie, kým random search za extrémny prípad diverzifikácie.

3.2.1 Hill climbing s náhodnými reštartmi

Kombináciou týchto dvoch algoritmov nám vzniká *Hill climbing s náhodnými reštartmi*. Nejaký náhodne určený časový interval robíme hill climbing, keď čas vyprší, začneme odznova v novom náhodne zvolenom stave a pokračujeme v hill climbing-u, tentoraz iný náhodne určený časový interval.

1: *T* je množina možných časových intervalov.
2: *s* nastavíme na náhodné riešenie zo stavového priestoru.
3: *Best* bude riešenie, ktoré na konci vrátime a na začiatku ho inicializujeme na *s*.
4: Opakujeme, kým *Best* nie je globálne optimum, alebo nám vypršal všetok čas:
5: Premennú *čas* nastavíme na náhodný čas v blízkej budúcnosti, vybratý z *T*.
6: Opakujeme, kým *s* nie je globálne optimum, vypršal nám *čas*, alebo nám vypršal všetok čas:
7: Miernou úpravou *s* získame jeho suseda s'.
8: Ak platí $f(s') \leq f(s)$, potom $s = s'$.
9: Ak platí $f(Best) \leq f(s)$, potom $Best = s$.
10: *s* znovu nastavíme na náhodné riešenie zo stavového priestoru.
11: Vrátime riešenie *Best*.

Ak náhodne zvolené časové intervaly sú veľmi dlhé, algoritmus sa sústredí na hill climbing. Naopak, ak sú krátke, pre hill climbing neostane veľa času a výsledok sa čoraz viac podobá na náhodné prehľadávanie. Stredné hodnoty zapájajú oba algoritmy vo vyváženej miere. Aký je ideálny pomer hill climbing-u k náhodnému prehľadávaniu?

Na zodpovedanie tejto otázky potrebujeme vedieť o aký problém sa jedná, presnejšie, ako vyzerá fitness landscape problému. Ak fitness landscape má tvar jedného hladkého kopca, na ktorý treba vyliezť, bez žiadnych menších kopčekov – lokálnych optím, samotný hill climbing sa dá považovať za takmer optimálny algoritmus. Ak však fitness landscape je veľmi kostrbatý, s mnohými lokálnymi optimami, ak len nevieme niečo bližšie o funkcii, ktorá ho vytvorila, náhodné prehľadávanie je rovnako dobrý (ak nie lepší) prístup ako ľubovoľný iný.

3.3 Simulované žíhanie

Simulované žíhanie je ďalšia metaheuristika ktorá patrí do kategórie metaheuristík s prehľadávaním založeným na jednom riešení. Je to metaheuristika inšpirovaná reálnym procesom; jej názov aj hlavná myšlienka vychádza zo žíhania, čo je spôsob spracovania kovov pri ktorej zohriatím na určitú teplotu a pomalým ochladzovaním nadobúda kov stabilnú štruktúru. Ak sa kov vychladí príliš rýchlo a jeho atómy sa nestihnú uložiť do pevnej kryštálovej mriežky, výsledkom bude nadmerná krehkosť kovu. V simulovanom žíhaní vidíme metaforu práve na energiu týchto atómov ktorá sa – ako sa kov postupne ochladzuje – pomaly zmenšuje. V samotnom algoritme sa dajú rozoznať podobné myšlienky ako u hill climbing-u, rozdiel spočíva v tom, kedy sa nahrádza naše kandidátne riešenie *s* za jeho mierne modifikovanú verziu s'. Ak s' je objektívnou funkciou lepšie ohodnotené, nahradíme ním *s* vždy. Zaujímavejší prípad nastane ak s' bude objektívnou funkciou horšie ohodnotené než *s*. V takom prípade, s' nahradí *s* s pravdepodobnosťou *P*.

Tá je určená rozdielom v ohodnotení s' a *s* objektívnou funkciou a tiež parametrom *T* po ktorého uplynutí program skončí. Ohodnotenia $f(s)$ a $f(s')$ objektívnou funkciou reprezentujú energiu atómov pri žíhaní, parameter *T* je analógiou pre teplotu kovu. Vzorec na výpočet pravdepodobnosti je teda nasledovný: $P(f(s), f(s'), T) = e^{-\frac{f(s')-f(s)}{T}}$.

Parameter *T* (teplota) zohráva významnú úlohu pri úspešnosti simulovaného žíhania. Ak ju na začiatku nastavíme na dostatočne veľkú, algoritmus bude s väčšou pravdepodobnosťou skúšať horšie riešenia a prehľadávanie stavového priestoru bude viac pripomínať náhodné prehľadávanie. Pri teplote blížiacej sa k nule sa horšie riešenia menej často akceptujú a prehľadávanie pripomína hill climbing. Pri jednej zvolenej teplote majú väčšiu pravdepodobnosť byť akceptované tie horšie riešenia s' ktorých ohodnotenie objektívnou funkciou sa menej líši od $f(s)$ než ohodnotenia iných horších riešení.

Popis algoritmu:

1: Nastavíme počiatočnú teplotu *T* na dostatočne vysokú hodnotu.
2: Určíme si nejaké počiatočné riešenie *s* zo stavového priestoru.
3: *Best* nastavíme na *s*.
4: Opakujeme, kým *Best* nie je globálne optimum, alebo nám vypršal všetok čas, alebo $T \leq 0$:
5: Miernou úpravou *s* získame jeho suseda s'.

6: Ak platí $f(s') \leq f(s)$, tak $s = s'$ (akceptujeme s'), v opačnom prípade akceptujeme s' s pravdepodobnosťou $e^{-\frac{f(s')-f(s)}{T}}$.

7: Znížime teplotu *T*.

8: Ak $f(s) < f(Best)$, potom $Best = s$.

9: Vrátime riešenie *Best*.

Kvalita výsledných riešení do veľkej miery závisí od spôsobu znižovania teploty *T*. Ak ju budeme znižovať veľmi pomaly, získame lepšie riešenia za cenu dlhšieho času výpočtu. Existuje viacero spôsobom ako *T* znižovať, keďže tento aspekt simulovaného žíhania nie je preddefinovaný. Tu je zopár najznámejších funkcií ochladzovania[4]:

- **Lineárna** – od T pri každej iterácii odčítavame určitú konštantu c.
  $T = T - c$
- **Geometrická** – najpopulárnejšia metóda ochladzovania.
  Dá sa vyjadriť vzorcom $T = c \cdot T$, kde *c* je reálne číslo $0 < c < 1$. Praktické skúsenosti ukazujú, že jeho hodnota by mala byť medzi 0,5 a 0,99.
- **Logaritmická** – príliš pomalé aby malo praktické využitie.
  $T_i = \frac{T_0}{\log(i)}$
  T_i reprezentuje teplotu *T* v *i*-tej iterácii algoritmu (T_0 je teda počiatočná teplota).
- **Nemonotónna** – pri nemonotónnych funkciách ochladzovania teplota nie len klesá ale môže aj znova stúpnuť. Je to metóda prikláňajúca sa k princípu diverzifikácie, a je optimálna pre niektoré typy landscape-ov.
- **Adaptívna** – na zlepšenie riadenia znižovania teploty sa využíva nejaká informácia získaná priamo počas behu programu.

3.4 Tabu Search

Tabu search („zakázané prehľadávanie“) je jednou z najrozšírenejších metaheuristík. Je tiež založený na princípe lokálneho prehľadávania, ale tento algoritmus sa od predchádzajúcich líši v tom, že využíva pamäť, v ktorej má uložené údaje týkajúce sa prehľadávania. Práca s pamäťou je vlastne jeho charakteristickou črtou. Vytvára si totiž zoznam (*tabu list*) nedávno navštívených stavov v stavovom priestore. Riešenia v tomto zozname sú preňho tabu a má ich zakázané ešte raz navštíviť. Teda ak vo fitness landscape vojde do rokliny, nemôže v nej ostať a je z nej nútený vyliezť. To je jeho stratégia vyhýbaniu sa lokálnym optimám. Tým pádom v niektorých prípadoch musí akceptovať horšie riešenie než je jeho súčasné – v prípadoch, keď nenachádza lepšie riešenie, ktoré by už nebolo v zozname tabu. Udržiavať zoznam všetkých navštívených stavov začne byť po čase neefektívne (pri každom novom kandidátnom riešení, treba prezrieť zoznam, či sa v ňom už dané riešenie nenachádza) a preto zoznam tabu má zväčša konštantnú dĺžku. Pri zaplnení sa najstaršie riešenia začnú prepisovať novými.
Tu je základný algoritmus[7]:

1: Nastavíme *s* na nejaké riešenie zo stavového priestoru.
2: *Best* nastavíme na *s*.
3: Vytvoríme zoznam *TabuList*, ktorý na začiatku bude prázdny.
4: Opakujeme, kým *Best* nie je optimálne riešenie, alebo nám vypršal čas:
5: Nové *s* nastavíme na najlepšieho suseda *s*, ktorý nie je v *TabuList* ($N(s) - TabuList$).
6: Aktualizujeme *TabuList*.
7: Ak $f(s) \geq f(Best)$, potom $Best = s$.
8: Vrátime riešenie *Best*.

Narábať so zoznamom tabu by bolo neefektívne aj keby sa v ňom mali pamätať samotné kompletné riešenia. Okrem toho, ak je landscape príliš veľký, môže sa stať, že ani dlhý zoznam tabu nám neumožní vyliezť z jamy – lokálneho optima. Preto sa pamätajú iba určité *atribúty*, vlastnosti riešení – sú to napríklad nejaké komponenty riešení alebo rozdiely medzi riešeniami. Keďže môžeme brať do úvahy viac ako jeden atribút, potrebujeme zoznam tabu pre každý z nich. Množina atribútov s ich zodpovedajúcimi zoznamami sú *tabu podmienky*, pomocou ktorých filtrujeme okolie riešenia. Pamätať si atribúty riešení namiesto

samotných riešení je síce efektívnejšie, ale prichádzame o určitú časť informácie. To môže spôsobiť, že do zoznamu tabu sa dostane nepreskúmané dobré riešenie. Aby sme sa tomuto problému vyhli, boli vytvorené *ašpiračné kritériá*, ktoré umožňujú preskúmať aj zakázané riešenie. Najčastejšie je používané ašpiračné kritérium, ktoré akceptuje riešenia, ktoré sú lepšie ako najlepšie doteraz nájdené. Iným ašpiračným kritériom môže byť akceptovanie riešenia ktoré je lepšie ako zakázané riešenia ktoré majú spoločný nejaký atribút.

Pamätať si zoznam tabu je len jeden zo spôsobov využívania histórie prehľadávania. Pokročilejšie varianty tabu search-u umožňujú mať napríklad lepšiu kontrolu nad intenzifikáciou a diverzifikáciou algoritmu. Intenzifikáciu podporuje využívanie *strednodobej pamäte*, diverzifikáciu využívanie *dlhodobej pamäte*. Za *krátkodobú pamäť* sa považuje zoznam tabu.

Strednodobá pamäť si ukladá *elitné* (najlepšie) riešenia objavené počas prehľadávania. Atribúty týchto elitných riešení sú ďalej uprednostňované, a prehľadávanie častejšie uprednostňuje riešenia s rovnakými atribútmi.
Dlhodobá pamäť si tiež pamätá údaje o navštívených riešeniach. Využíva ich aby nasmerovala prehľadávanie k nepreskúmaným oblastiam stavového priestoru. Napríklad môže podporovať vyhľadávanie riešení, ktoré majú úplne iné atribúty než elitné riešenia.
Tabu search obohatený o ašpiračné kritériá, strednodobú a dlhodobú pamäť:

1: *s* bude počiatočné riešenie.
2: *Best* nastavíme na *s*.
3: Inicializujeme zoznam tabu, strednodobú a dlhodobú pamäť.
4: Opakujeme kým Best nie je optimálne riešenie, alebo nám vypršal čas:
5: Nájdeme najlepšieho suseda *s*, ktorý nie je v zozname tabu alebo ktorý spĺňa ašpiračné kritériá.
6: Ten sused bude naše nové *s*.
7: Aktualizujeme zoznam tabu, strednodobú a dlhodobú pamäť.
8: Ak platí *nejaké kritérium intenzifikácie*, potom *intenzifikujeme*.
9: Ak platí *nejaké kritérium diverzifikácie*, potom *diverzifikujeme*.
10: Ak $f(s) \geq f(Best)$, potom $Best = s$.
11: Vrátime riešenie *Best*.

V porovnaní so simulovaným žíhaním a hill climbing-om niektoré aspekty tabu search potrebujú byť navrhnuté pre špecifický problém, spôsob akým sa implementujú závisí od

konkrétneho typu optimalizačného problému. Určiť správny typ reprezentácie pre zoznam tabu, strednodobú a dlhodobú pamäť pre mnohé optimalizačné problémy nie je jednoduchá úloha. Tabu search môže byť tiež veľmi citlivý na nastavenie určitých parametrov, napríklad na dĺžku zoznamu tabu.

3.5 Iteračné lokálne prehľadávanie

Iteračné lokálne prehľadávanie je v podstate šikovnejšia verzia hill climbing-u s náhodnými reštartmi. Ak hill climbing-u dáme dostatočný čas, môžeme predpokladať že po každom reštarte sa ocitne v nejakom lokálnom optime (v ideálnom prípade bude nové optimum patriť medzi zatiaľ nepreskúmané). Tým pádom vlastne prehľadávame v priestore lokálnych optím a na konci vrátime to najlepšie z nich, dúfajúc, že sme objavili globálne optimum.

Iteračné lokálne prehľadávanie prechádza priestorom lokálnych optím inteligentnejším spôsobom, snaží sa akoby použiť hill climbing priamo na priestor lokálnych optím. Keď objaví lokálne optimum, snaží sa nájsť ďalšie v jeho „blízkosti“. Svoje počiatočné riešenie pri reštarte nevyberá náhodne ale opiera sa o pozíciu niektorého z doteraz nájdených lokálnych optím v priestore. To lokálne optimum môžeme považovať za akúsi „domovskú základňu“. Počiatočné riešenie pri ďalšom reštarte sa bude nachádzať niekde „v blízkosti“ základne, i keď nie v prílišnej. Využíva sa tu predpoklad, že lokálne optimá sa zväčša vyskytujú blízko seba. Ak by sme však začali príliš blízko, ukončíme prehľadávanie v tom istom optime. Ak začneme príliš ďaleko, nebude sa to líšiť od náhodného počiatočného riešenia. Dôležité je určiť na základe čoho sa vyberá nová domovská základňa. Ak si za domovskú základňu vyberáme vždy to nové lokálne optimum, je to vlastne určitá forma náhodného prechádzania stavovým priestorom. Ak si za domovskú základňu vyberáme vždy také lokálne optimum ktoré je lepšie ohodnotené než naša súčasná základňa, je to hill climbing na priestore lokálnych optím. Väčšinou sa nepoužíva jeden konkrétny z týchto dvoch prístupov ale niečo medzi. Problémom je ešte určiť, kedy sme nejaké lokálne optimum objavili. To sa dá vyriešiť obmedzením času pre jednu iteráciu algoritmu. Síce nám to nezaručí, že po uplynutí času budeme práve v lokálnom optime, pri nastavení dostatočného časového intervalu je pravdepodobné, že sa od nejakého nebudeme nachádzať ďaleko.

Čiže algoritmus na začiatku nejaký čas robí hill climbing, a získané lokálne optimum nastaví na domovskú základňu. Potom na domovskú základňu aplikuje operáciu, ktorú môžeme vnímať ako veľký náhodný skok od daného riešenia – vytvorenie nového riešenia

modifikáciou súčasného. Túto operáciu nazývame *perturbačnou funkciou*. Novonadobudnuté riešenie poslúži ako počiatočné riešenie pre nasledujúci hill climbing. Hill climbing nám objaví ďalšie lokálne optimum o ktorom sa následne môžeme rozhodovať, či nemá nahradiť domovskú základňu.
Algoritmus vyzerá takto:

1: *T* je množina časových intervalov.
2: *s* je nejaké počiatočné kandidátne riešenie.
3: *h* je súčasná domovská základňa, na začiatok nastavená na *s*.
4: *Best*, najlepšie riešenie, tiež najprv nastavené na *s*.
5: Opakujeme, kým *Best* nie je optimálne riešenie alebo nám vypršal všetok čas:
6: *čas* nastavíme na náhodne zvolený z množiny *T*.
7: Opakujeme kým *s* nie je optimálne riešenie, alebo nám vypršal *čas*, alebo nám vypršal všetok čas:
8: Za s' si zvolíme riešenie ktoré vznikne miernou úpravou *s*.
9: Ak platí $f(s') \leq f(s)$, potom $s = s'$.
10: Ak platí $f(Best) \leq f(s)$, potom $Best = s$.
11: Tu sa rozhodujeme, či máme *s* akceptovať za novú domovskú základňu *h*. $h = NewHomeBase(h, s)$
12: Zavoláme perturbačnú funkciu na domovskú základňu *h* a výsledné riešenie bude naše nové *s*. $s = Perturb(h)$
13: Vrátime riešenie *Best*.

To, že ako sú funkcie *NewHomeBase* a *Perturb* implementované, záleží od konkrétneho problému. Perturbačná funkcia musí modifikovať naše riešenie do dostatočnej miery aby sme sa dostali von zo súčasného lokálneho optima ale zároveň výsledok musí byť efektívnejší než náhodné prehľadávanie. Môže byť implementovaná iným spôsobom než operátor ktorý nám vracia suseda riešenia. Pri mnohých perturbačných funkciách je potrebné zohľadniť tieto kritériá:

- **Vzdialenosť od pôvodného lokálneho optima, po perturbácii:**

Môže byť *statická* – je určená vopred ešte pred začiatkom prehľadávania.
Dynamická – mení sa počas prehľadávania, ale nezávisle od informácii získaných pri prehľadávaní.

Adaptívna – mení sa počas prehľadávania, v závislosti od určitej informácie získanej počas prehľadávania.

- **Náhodná alebo semideterministická perturbácia:**

Pri *náhodnej* perturbácii sa nevyužíva pamäť a nové riešenie je zvolené náhodne z okolia. *Semideterministická* perturbácia závisí od pamäte prehľadávania. Môžu byť aplikované podobné štruktúry ako v prípade tabu search-u (strednodobá a dlhodobá pamäť).

Úloha *NewHomeBase* funkcie je určovať pomer medzi intenzifikáciou a diverzifikáciou prehľadávania, tým, že si vhodne vyberie novú domovskú základňu. Jeden extrémny spôsob (ktorý uprednostňuje diverzifikáciu) je akceptovať každé nové riešenie $NewHomeBase(h,s) = s$. Druhý extrémny spôsob (uprednostňujúci intenzifikáciu) je určiť za novú domovskú základňu riešenie ktoré je lepšie ohodnotené ako súčasná základňa $NewHomeBase(h,s) = \begin{cases} s, \text{ak } f(s) \le f(h) \\ h, \text{ak } f(s) > f(h) \end{cases}$. Väčšina implementácií iteračného lokálneho prehľadávania si volí strednú cestu medzi týmito dvoma extrémami. Napríklad chvíľu sa sústredí na intenzifikáciu a v prípade, že už dlho neobjavil zatiaľ nepreskúmané riešenie ktoré by bolo lepšie než súčasné, zmení stratégiu a začne sa na nejaký čas venovať diverzifikácii. Existujú však aj iné spôsoby, ktoré na určenie novej domovskej základne využívajú princípy iných metaheuristík, napríklad simulovaného žíhania alebo určitú formu tabu search-u.

3.6 Evolučné algoritmy

Evolučné algoritmy patria k metaheuristikám využívajúcim prehľadávanie založené na populácii (skrátene populačné metaheuristiky). Ako ich názov prezrádza, sú inšpirované procesom prírodného výberu. Za posledných štyridsať rokov sa vyvinulo viacero smerov evolučných algoritmov: genetické algoritmy (Holland), evolučné stratégie (Rechenberg, Schwefel), evolučné programovanie (Fogel) a genetické programovanie (Koza). Každý z týchto smerov reprezentuje iný prístup, ale v každom z nich sa dajú rozoznať tie isté princípy odpozorované z prírody, z evolúcie. Na začiatku sa náhodne vygeneruje celá *populácia* riešení, z ktorej sa vyselektujú tí *jedinci,* ktorí sú najlepšie prispôsobení na prežitie – tie riešenia ktoré sú najbližšie k optimálnemu. Jednotlivé riešenia sú reprezentované určitou formou *genetickej informácie*, ktorá sa ľahko dá pozmeniť (napr. náhodnou *mutáciou,* alebo *krížením* s ďalšími riešeniami), čím vzniknú úplne nové riešenia - *potomkovia*, ktoré sú potenciálne lepšie ako jeho *rodič* (pôvodné riešenie). Základný evolučný algoritmus najprv náhodne vygeneruje prvú generáciu riešení. Potom opakuje tieto tri procedúry: určí *fitness* (vhodnosť) každého jedinca objektívnou funkciou; túto informáciu určitým spôsobom využije na vytvorenie novej populácie potomkov („rozmnoží“ ich); napokon z potomkov a rodičov vytvorí ďalšiu generáciu.

Abstraktný evolučný algoritmus:

1: P na začiatku nastavíme na iniciálnu populáciu.
2: *Best* bude najlepšie riešenie, na začiatku však nemá hodnotu. $Best = \varnothing$
3: Opakujeme, kým *Best* nie je optimálne riešenie alebo nám vypršal čas:
4: $AssessFitness(P)$ - pri evolučných algoritmoch potrebujeme zvlášť funkciu ktorá ohodnotí celú populáciu pomocou objektívnej funkcie.
5: Pre každého jedinca $s \in P$:
6: Ak $Best = \varnothing$ alebo $f(s) \leq f(Best)$, potom $Best = s$.
(nájdenie najlepšieho jedinca z populácie)
7: $P = Join(P, Breed(P))$
Vyšľachtíme generáciu potomkov a v kombinácii s rodičovskou generáciou vytvoríme novú populáciu P.
8: Vrátime riešenie *Best*.

Evolučné algoritmy sa odlišujú v tom akým spôsobom implementujú operácie *Breed* a *Join*. Operácia *Breed* sa väčšinou skladá z dvoch častí: výber rodičov z pôvodnej populácie, a ich modifikácia (mutácie alebo vzájomné kríženie) na vytvorenie potomkov. Operácia Join buď úplne nahradí rodičov potomkami alebo vytvorí novú populáciu kombináciou potomkov a dobre ohodnotených rodičov. Teda paralela z biológie pre operáciu *Breed* je pohlavný výber (určuje ktorí jedinci môžu mať potomstvo) a pre operáciu *Join* je to prírodný výber (určuje ktorí jedinci prežijú).

Iniciálna populácia sa pri každom evolučnom algoritme generuje v podstate rovnako, vyberáme určitý počet náhodných jedincov zo stavového priestoru. Ak však máme nejakú informáciu o fitness landscap-e, a vieme kde sa zrejme budú nachádzať dobré oblasti, môžeme prvých jedincov vyberať z blízkosti tohto regiónu. Rôznorodosť prvotných riešení môže prísť vhod a preto je dôležité, aby sme zaručili, že medzi riešeniami sa nebudú vyskytovať duplikáty.

3.6.1 Evolučné stratégie

K najjednoduchším evolučným stratégiám patrí algoritmus (μ, λ). Začíname s populáciou λ náhodne vygenerovaných jedincov. Určíme ich *fitness* (objektívnou funkciou). Vyberieme μ najlepšie ohodnotených jedincov a zvyšok populácie odstránime. Každému z μ-ich jedincov vygenerujeme λ / μ potomkov. Operácia *Join* následne vytvorí ďalšiu populáciu nahradením rodičov potomkami. Celý proces sa potom opakuje s novou populáciou, až kým nebudú splnené kritériá zastavenia. μ v názve je teda číslo určujúce počet jedincov ktorí sa dožijú toho aby sa stali rodičmi a λ je počet ich potomkov. Konkrétne algoritmy (μ, λ) sa teda nazývajú podľa čísiel μ a λ (ak $\mu = 5$ a $\lambda = 20$ jedná sa o Evolučnú stratégiu (20, 5). Algoritmus vyzerá takto:

1: μ počet vyberaných rodičov.
2: λ počet potomkov generovaných rodičmi.
3: $P = \varnothing$ množina zahŕňajúca celú populáciu.
4: λ krát opakujeme:
5: $P = P \cup \{\text{nový náhodný jedinec}\}$
Vytvorenie iniciálnej populácie.
6: $Best = \varnothing$

7: Opakujeme kým *Best* nie je optimálne riešenie alebo nám vypršal čas:

8: $AssessFitness(P)$

9: Pre každého jedinca $s \in P$:

10: Ak $Best = \varnothing$ alebo $f(s) \leq f(Best)$, potom $Best = s$.

11: Q bude množina μ-ich najlepšie ohodnotených jedincov.

12: P je vyprázdnené, lebo operácia *Join* nahradí pôvodnú populáciu potomkami.
$P = \varnothing$

13: Pre každého jedinca $q \in Q$:

14: λ / μ-krát opakujeme:

15: $P = P \cup \{Mutate(q)\}$

Mutovaním vytvárame potomstvo a postupne ho vkladáme do P.

16: Vrátime riešenie *Best*.

Pri evolučných stratégiách sa nepoužíva operácia kríženia, operácia *Mutate* (mutačná funkcia) je však už aj tu zastúpená a jej úloha je miernou úpravou riešenia vytvoriť nové riešenie – suseda pôvodného riešenia.

(μ, λ) algoritmus nám dáva voľnú ruku pri určení týchto troch parametrov, pomocou ktorých sme schopný regulovať pomer intenzifikácie a diverzifikácie:

- **Veľkosť λ** – určuje sa ním vlastne veľkosť celej populácie v jednom momente; ak ju nastavíme na čím väčšiu prehľadávanie bude čoraz viac pripomínať náhodné prehľadávanie (diverzifikácia).
- **Veľkosť μ** – od počtu vybraných rodičov závisí *selektívnosť* algoritmu; vyššie hodnoty spôsobia, že proces bude menej „prísny“, umožní horším riešeniam mať potomstvo, naopak pri nízkych hodnotách prežijú len tí najlepší a prehľadávanie bude bližšie k princípu intenzifikácie.
- **Implementácia mutačnej funkcie** – ak nám *Mutate* generuje príliš vzdialených susedov pôvodného riešenia, výsledná populácia bude zložená z príliš náhodných riešení a to, že akí boli rodičia nebude zohrávať takmer žiadnu rolu.

Iná evolučná stratégia je algoritmus $(\mu + \lambda)$. Oproti algoritmu (μ, λ), *Join* vytvorí novú populáciu kombináciou rodičov a potomkov. Teda v ďalšej generácii spolu súperia nové riešenia so starými dobrými riešeniami, a veľkosť ďalších generácií je $\mu + \lambda$.

Popis algoritmu:

1: μ počet vyberaných rodičov.

2: λ počet potomkov generovaných rodičmi.

3: $P = \varnothing$

4: λ krát opakujeme:

5: $P = P \cup \{\text{nový náhodný jedinec}\}$

6: $Best = \varnothing$

7: Opakujeme kým *Best* nie je optimálne riešenie alebo nám vypršal čas:

8: $AssessFitness(P)$

9: Pre každého jedinca $s \in P$:

10: Ak $Best = \varnothing$ alebo $f(s) \leq f(Best)$, potom $Best = s$.

11: Q bude množina μ-ich najlepšie ohodnotených jedincov.

12: $P = Q$

Tu sa nachádza jediný rozdiel oproti algoritmu (μ, λ). V P ostanú riešenia z Q ktoré v ďalšej generácii budú konkurovať svojim potomkom.

13: Pre každého jedinca $q \in Q$:

14: λ / μ-krát opakujeme:

15: $P = P \cup \{Mutate(q)\}$

16: Vrátime riešenie *Best*.

Keďže dobre ohodnotený rodičia ostávajú v populácii, $(\mu + \lambda)$ je viac zameraný na intenzifikáciu než algoritmus (μ, λ). To však zo sebou prináša určité riziká. Veľmi zdatný jedinec môže opakovane víťaziť nad ostatnými členmi populácie, až kým celá populácia nebude zložená iba z jeho potomkov, a teda algoritmus uviazne v lokálnom optime.

3.6.2 Evolučné programovanie

Evolučné programovanie je kategória evolučných algoritmov sústrediacich sa na operáciu mutácie. Tiež nepoužívajú operáciu kríženia. Selekcia rodičov a obmedzenie veľkosti populácie sa podobá na algoritmus $(\mu + \mu)$, čiže ako $(\mu + \lambda)$ len $\lambda = \mu$. Rodičmi sa teda stane polovica populácie a zvyšok nepreží je.

Kvôli prílišnej podobnosti s evolučnými stratégiami, evolučné programovanie sa nevyužíva tak často ako ostatné typy evolučných algoritmov.

3.6.3 Genetické algoritmy

Genetické algoritmy sú veľmi populárnou triedou evolučných algoritmov. Podobajú sa na evolučné stratégie (μ, λ). Rozdiel medzi nimi spočíva v tom že kým algoritmus (μ, λ) naraz určí všetkých rodičov a následne vygeneruje ich potomstvo, genetické algoritmy si postupne vyberú zopár rodičov, vygenerujú im zopár detí, vyberú si ďalších rodičov, a tak ďalej, až kým sa nevygenerovalo dostatočne veľké potomstvo. Tiež sa tu využíva operácia kríženia. Algoritmus vyzerá takto:

1: n veľkosť populácie. V algoritme (μ, λ) je to λ.
2: $P = \varnothing$ množina zahŕňajúca celú populáciu.
3: n-krát opakujeme:
4: $P = P \cup \{\text{nový náhodný jedinec}\}$
5: $Best = \varnothing$
6: Opakujeme kým *Best* nie je optimálne riešenie alebo nám vypršal čas:
7: $AssessFitness(P)$
8: Pre každého jedinca $s \in P$:
9: Ak $Best = \varnothing$ alebo $f(s) \leq f(Best)$, potom $Best = s$.
10: $Q = \varnothing$
11: $n/2$ krát opakujeme:
12: Rodič $r_1 = Select(P)$
13: Rodič $r_2 = Select(P)$
14: Potomkovia $p_1, p_2 = Crossover(r_1, r_2)$
15: $Q = Q \cup \{Mutate(p_1), Mutate(p_2)\}$
16: $P = Q$
17: Vrátime riešenie *Best*.

Objavili nám tu dve nové funkcie: *Select* (vyberá ktorý rodičia budú mať spolu potomstvo) a *Crossover* (vytvára potomkov tým, že prekombinuje genetickú informáciu oboch rodičov).

Crossover vykonáva teda spomínanú operáciu kríženia, a je to funkcia charakteristická pre genetické algoritmy. Jej názov je odvodený z biologického procesu *prekríženia chromozómov*, pri ktorom si chromozómy medzi sebou vymieňajú genetický materiál.

Hlavnou črtou funkcie crossover je dedičnosť. Nové riešenie musí od svojich rodičov zdediť nejaký genetický materiál, musí sa v určitom zmysle podobať na svojich rodičov. Tá vlastnosť rodiča, ktorá spôsobila, že bol považovaný za dobré riešenie, by mala byť zachovaná aj v potomkovi.

Implementácia crossover funkcie závisí od spôsobu akým sú riešenia reprezentované. Predpokladajme, že v našej populácii sú jedinci reprezentovaný jednorozmerným poľom, vektorom dĺžky *n*. Potom existujú tri klasické spôsoby implementácie funkcie kríženia: *kríženie v jednom bode, dvojbodové kríženie* alebo *uniformné kríženie*. Pri krížení v jednom bode si určíme číslo c ($1 \le c \le n$) ktoré nám jeden vektor rozdelí na dve časti - prvých c zložiek vektora a zvyšok. Kríženie prebieha tak, že medzi dvoma vektormi vymeníme prvých $c-1$ zložiek. Algoritmus:

1: $\vec{v} = \langle v_1, v_2, ... v_n \rangle$ je prvý vektor na ktorý sa chystáme krížiť.

2: $\vec{w} = \langle w_1, w_2, ... w_n \rangle$ je druhý vektor.

3: Určíme si naše číslo c za náhodné prirodzené číslo také, že $1 \le c \le n$.

4: Ak $c \neq 1$, potom pre každé *i* od 1 do $c-1$ vymeníme hodnoty v_i a w_i.

5: Vrátime vektory $\vec{v}$ a $\vec{w}$.

Ak sa $c = 1$ nedôjde k žiadnemu kríženiu. Tento prípad nastane s pravdepodobnosťou $\frac{1}{n}$.

Samozrejme nie je zložité, zmeniť jeho pravdepodobnosť ak c budeme vyberať z intervalu od 2 do *n*. Problém kríženia s jedným bodom, je ten že je veľmi pravdepodobné, že v_1 a v_n budú oddelené od seba. Ak vysoká vhodnosť riešenia bola podmienená tým, že tieto dve zložky sa nachádzajú vo vektore súčasne, naše kríženie bude neustále rozbíjať dobré páry jedincov. Pravdepodobnosť, že napríklad v_1 a v_2 nebudú vedľa seba je oveľa menšia, c musí k tomu byť rovné presne 2.

Dvojbodové kríženie si nevyberá jedno číslo ale dve, c a d, čím rozdeľuje vektor na tri časti a kríženie vymení strednú časť medzi dvoma vektormi.
Popis algoritmu:

1: $\vec{v} = \langle v_1, v_2, ... v_n \rangle$ je prvý vektor.

2: $\vec{w} = \langle w_1, w_2, ... w_n \rangle$ je druhý vektor.

3: Náhodne si určíme číslo *c* za také, že $1 \leq c \leq n$.

4: Náhodne si určíme aj číslo *d* tiež také, že $1 \leq d \leq n$.

5: Ak $c > d$, vymeníme hodnoty *c* a *d* navzájom.

6: Ak $c \neq d$, potom pre každé *i* od *d* do $c-1$ vymeníme hodnoty v_i a w_i.

7: Vrátime vektory $\vec{v}$ a $\vec{w}$.

Ako vyriešilo dvojbodové kríženie neférové zaobchádzanie s v_1 a v_n pri jednobodovom krížení? Vektor si môžeme prestaviť ako prstenec (teda prvá a posledná zložka sú vedľa seba). Dvojbodové kríženie vymení jeden úsek z tohto prstenca. v_1 a v_n budú rozdelené len ak *c* alebo *d* rozdelí prstenec presne medzi nimi. Stále sa tu však vyskytuje problém, a to pre zložky v_1 a $v_{n/2}$, ktoré majú tiež zvýšenú pravdepodobnosť oddelenia. Férový spôsob rozdeľovania vektorov nám zaručuje uniformné kríženie. Pri ňom jednoducho prejdeme celým vektorom a pre každú jednotlivú zložku zhodnotíme, či ju zameníme alebo nie.
Algoritmus vyzerá takto:

1: *p* je pravdepodobnosť zámeny jednej zložky.

2: $\vec{v} = \langle v_1, v_2, ... v_n \rangle$ je prvý vektor.

3: $\vec{w} = \langle w_1, w_2, ... w_n \rangle$ je druhý vektor.

4: Pre každé *i* od 1 do *n* s pravdepodobnosťou *p* vymeníme hodnoty v_i a w_i.

5: Vrátime vektory $\vec{v}$ a $\vec{w}$.

Je viacero spôsobov ako implementovať funkciu *Select* ktorá vyberá dvojicu rodičov vhodných na párenie. Nemusíme sa obmedzovať len na tých najlepších jedincov ale čas od času si môžeme zvoliť aj horších na posilnenie princípu diverzifikácie. Jedno riešenie sa môže stať aj viacnásobným rodičom a ani celkový počet jeho detí nie je vopred zadefinovaný. Hlavný princíp ktorým sa selekcia riadi hovorí, že čím je jedinec lepšie ohodnotený, tým má väčšiu šancu splodiť potomstvo. Kvalitu jedincov môžeme zhodnocovať dvoma spôsobmi: podľa absolútnej hodnoty ich fitness alebo podľa ich relatívnej fitness v porovnaní s ostatnými jedincami v populácii (napríklad na ako mieste sa nachádzajú v zozname všetkých riešení populácie usporiadanom podľa ohodnotenia).

Existujú štyri stratégie na vyber rodičov: *ruletový výber*, *stochastické univerzálne vzorkovanie*, *turnajový výber* a *výber založený na poradí*.

Najčastejšie používaná stratégia je ruletový výber. Každému jedincovi priradí pravdepodobnosť, ktorá bude v pomere k jeho relatívnej fitness. Nech f_i je fitness *i*-teho jedinca z populácie *P*. Jeho pravdepodobnosť výberu (p_i) je určená týmto vzťahom $p_i = f_i \Big/ \left(\sum_{j=1}^{n} f_j\right)$. S ruletou táto stratégia selekcie súvisí tak, že si môžeme predstaviť koleso šťastia po ktorého obvode sú jedinci z populácie a každý z nich má pridelený taký výrez z kolesa, ktorý zodpovedá jeho fitness. Rodič sa vyberá roztočením kolesa. Lepší jedinci majú viac priestoru a tým je väčšia šanca, že budú vybraní. K nevýhodám tejto stratégie patrí to, že celá populácia sa po istom čase môže skladať výhradne z potomkov veľmi dobrých jedincov zo začiatku prehľadávania – konverguje k lokálnemu optimu a ostane v ňom uviaznuté. Ďalší problém sa môže vyskytnúť v prípade ak sa fitness-y jednotlivých riešení v populácii nelíšia príliš od seba a tým pádom nevznikne dostatočný tlak k výberu najlepších jedincov.

Ďalšia stratégia výberu je *stochastické univerzálne vzorkovanie*. Je to obmena ruletového výberu pri ktorom sa vyberá viacero jedincov naraz. Vizuálne si to tak môžeme predstaviť, že okolo kolesa je ešte vonkajší kruh, po obvode ktorého sa nachádza *n* ukazovadiel ktoré vyberajú riešenia (pričom *n* je počet rodičov ktorých vyberáme). Ukazovadlá sú po obvode rovnomerne rozmiestnené, vzdialenosť dvoch susedných ukazovadiel je vždy rovnaká.

Ruletový výber a stochastické univerzálne vzorkovanie nie sú vhodné pre prípady, v ktorých zhodnocujeme kvalitu riešení podľa ich relatívnej fitness. V takých prípadoch tá samotná hodnota fitness hovorí o riešení iba toľko, že je lepšie než riešenia s horšou fitness. Pre tieto prípady je vhodnejšia stratégia *turnajového výberu*. Jeho algoritmus je veľmi jednoduchý:

1: *P* je populácia.
2: *t* je veľkosť turnaja, $t \geq 1$.
3: *Best* je náhodný jedinec vybraný z P.
4: Pre *i* od 2 do *t* opakujeme:
5: *Next* ďalší náhodne vybraný jedinec z *P*.
6: Ak *Next* je lepšie ohodnotené riešenie než *Best*, *Next* sa stáva novým *Best*.
7: Vrátime riešenie *Best*.

Teda vrátime najlepšie riešenie z nejakej skupinky náhodne vybraných jedincov. Pri genetických algoritmoch sa táto stratégia stala najpoužívanejšou. Vhodným zvolením parametra veľkosti turnaja *t*, môžeme určovať vyberavosť algoritmu. Pri veľkých hodnotách uprednostňuje čo najlepšie riešenia, pri $t = 1$ ide o náhodné prehľadávanie.

Výber podľa poradia je ďalší známy spôsob výberu rodičov. Neopiera sa o hodnotu fitness ale o poradie jedinca v zozname všetkých jedincov populácie usporiadanom podľa fitness. Pravdepodobnosť výberu jedného konkrétneho jedinca *i* je vypočítaná nasledovným vzorcom: $P(i) = \frac{2-s}{\mu} + \frac{2 \cdot r(i)(s-1)}{\mu(\mu-1)}$, kde *s* je *selektívny tlak* $(1{,}0 < s \leq 2{,}0)$, μ je veľkosť populácie a $r(i)$ je poradie prislúchajúce jedincovi *i*. Čím väčší je selektívny tlak, tým sa kladie väčší dôraz na uprednostňovanie lepších jedincov.

3.6.4 Genetické programovanie

Genetické programovanie je pomerne mladá metaheuristika využívajúca evolučný prístup k riešeniu problémov. Od ostatných evolučných algoritmov sa líši tým, že vyvíjajúce sa riešenia ktoré prehľadáva sú programy. Aby optimalizácia bola možná, predpokladá sa, že pre daný problém existujú suboptimálne programy, a nie jednoducho programy ktoré problém vyriešia a programy ktoré ho nevyriešia. Čiže genetické programovanie sa zaujíma o problémy, ktorých stavové priestory obsahujú veľké množstvo (zväčša malých) programov pri ktorých nie je jasné, ktorý vyrieši danú úlohu lepšie. Keďže programy majú premenlivú dĺžku, musia byť reprezentované štruktúrami ktoré majú tiež premenlivú dĺžku – zväčša sú to stromy. Zaujímavý je spôsob akým sa určuje fitness riešení – jednoducho spustíme výsledný program a zhodnotíme jeho úspešnosť.

3.7 Inteligencia roja

Názvom swarm intelligence („inteligencia roja“) nazývame triedu metaheuristík, inšpirovaných kolektívnym správaním takých spoločenských druhov organizmov akými sú: mravce, včely, termity, ryby a vtáky. Ich hlavnou charakteristikou je výskyt „roja“ jednoduchých agentov, ktorý prechádzajú stavovým priestorom a nepriamo vzájomne komunikujú. Medzi najznámejšie algoritmy z tejto kategórie patria *optimalizácia kolóniou mravcov* (*ant colony optimization*) a *optimalizácia rojom častíc* (*particle swarm optimalization*).

3.7.1 Optimalizácia kolóniou mravcov

Základnou myšlienkou optimalizácie kolóniou mravcov je napodobniť kooperatívne správanie mravcov a využiť ho na riešenie optimalizačných problémov. Využíva populáciu agentov, pričom správanie jedného konkrétneho agenta je inšpirované správaním jedného mravca z kolónie. Hlavný dôvod prečo sa inšpirovať mravcami pri riešení problémov je schopnosť reálnych mravcov nájsť najkratšiu trasu k potrave. Mravce navzájom komunikujú veľmi jednoduchým spôsobom – zanechávaním pachovej stopy. Počas svojich výprav vylučujú chemické látky (feromóny), ktorým je zaznamenaná ich trasa. Ostatné mravce sú schopné zacítiť tieto feromóny a pomôžu im pri rozhodovaní o ďalšom postupe trasy. Dôležitá je intenzita pachovej stopy – ktorá zodpovedá počtu mravcov ktorí tadiaľ prešli. Viac mravcov znamená väčšiu pravdepodobnosť, že trasa vedie k potrave. Tiež je dôležitý fakt, že intenzita feromónu časom slabne, vyprchá.

Táto metaheuristika je tiež založená na populácii, podobne ako evolučné algoritmy. Pri optimalizácii kolóniou mravcov však rozoznávame dva typy populácií. Prvý typ je množina *komponentov* z ktorých sa skladajú riešenia (napríklad pri probléme obchodného cestujúceho by komponenty boli hrany medzi mestami). Množina komponentov sa počas prehľadávania nemení ale po čase upravujeme „vhodnosť“ (nazývame ju *feromónom*) jednotlivých komponentov. Druhú populáciu tvoria *mravčie trasy*. Tak sa nazývajú naše kandidátne riešenia. Mravec z kolónie je stochastická konštruktívna procedúra, ktorá postupne skladá jednotlivé riešenia (trasy) pridávaním komponentov[5]. Algoritmus vyzerá takto:

1: $C = \{C_1, \dots, C_n\}$ je množina komponentov.
2: *size* je počet ciest, ktoré budú súčasne vytvorené.
3: $\vec{p} = \langle p_1, \dots, p_n \rangle$ je vektor, ktorý určuje feromón pre jednotlivé komponenty.
4: $Best = \emptyset$

5: Opakujeme kým *Best* nie je optimálnym riešením alebo nám vypršal čas:

6: *P* bude množina ciest, ktoré mravce iteratívne zložia z komponentov na základe ich feromónov. Ich počet je určený *size*.

7: Pre každé $P_i \in P$:

8: V prípade potreby môžeme P_i vylepšiť pomocou hill climbing-u.

9: Ak $Best = \emptyset$ alebo $f(P_i) \leq f(Best)$, potom P_i je nové *Best*

10: Aktualizujeme feromóny komponentov na základe ohodnotenia ciest objektívnou funkciou.

11: Vrátime riešenie *Best*.

Dve hlavné časti algoritmu sú: vytváranie mravčích ciest a aktualizácia feromónov komponentov.

Pri iteratívnom vytváraní mravčích ciest mravce berú do úvahy:

- **Feromónové trasy** – vďaka feromónom sme schopný zapamätať si vlastnosti dobrých riešení a skladať pomocou nich nové dobré riešenia. Feromónové trasy sa dynamicky menia počas prehľadávania a tým odzrkadľujú novonadobudnuté poznatky.
- **Heuristickú informáciu špecifickú pre daný problém** – tiež pomáha mravcom pri rozhodovaní.

Vygenerované riešenia sú použité pri aktuálizácii feromónovej stopy komponentov.

Aktualizácia sa skladá z dvoch fáz: s *fázy vyparovania* a *fázy posilnenia*.

Vo fáze vyparovania, hodnota feromónu automaticky klesá. Vyparovanie slúži k tomu, aby sa podporila diverzifikácia prehľadávania a aby sme neuviazli v lokálnom optime.

Fáza posilnenia zvyšuje feromón komponentov. To znamená, že po tej trase buď prechádzalo veľké množstvo mravcov, alebo aspoň jeden, čím sa zvyšuje pravdepodobnosť, že sa jedná o veľmi dobré riešenie, ktoré budú mravce uprednostňovať aj pri ďalších iteráciách.

3.7.2 Optimalizácia rojom častíc

Táto metaheuristika je inšpirovaná hromadným správaním zvierat, ktoré sa pohybujú v rojoch alebo kŕdľoch. V optimalizácii rojom častíc, sa náš roj skladá z *n* častíc ktoré poletujú v niekoľko dimenzionálnom stavovom priestore. Každá častica je potenciálne riešenie a je reprezentovaná vektorom. Každá častica má tiež určenú svoju polohu, smer a rýchlosť. Úspešnosť niektorých častíc ovplyvňuje správanie ostatných. Častice sú schopné zmeniť si svoju polohu na základe najlepšej pozície ktorú navštívili, a na základe najlepšej

pozície navštívenej celým rojom. Častice neumierajú – nie je tam selekcia; iba sa riadeno mutáciou pohybujú v stavovom priestore.

Pozícia častice, určená vektorom $\vec{x} = \langle x_1, x_2, \dots \rangle$ reprezentuje samotné riešenie, podobne ako genetická informácia v prípade evolučných algoritmov. Rýchlosť a smer častice je tiež určená vektorom, $\vec{v} = \langle v_1, v_2, \dots \rangle$. Inými slovami ak $\vec{x}^{(t-1)}$ je pozícia častice v čase $t-1$ a $\vec{x}^{(t)}$ je pozícia častice v čase t, rýchlosť a smer častice v čase t je určený vzťahom $\vec{v} = \vec{x}^{(t)} - \vec{x}^{(t-1)}$.

Na začiatku prehľadávania každá častica začína na náhodnej pozíci s náhodnou rýchlosťou a smerom pohybu.

Tiež si musíme pamätať: najlepšiu pozíciu $\vec{x}^*$ ktorú častica $\vec{x}$ zatiaľ navštívila, najlepšiu pozíciu $\vec{x}^+$ ktorú navštívili *informátori* častice $\vec{x}$, najlepšiu pozíciu $\vec{x}^!$ navštívenú kýmkoľvek.

Algoritmus opakuje tieto tri kroky:

1. Zistí fitness každého riešenia a ak je to potrebné, aktualizuje doteraz najlepšie nájdené riešenia.
2. Rozhodne o tom ako aplikovať funkciu *Mutate* – ako sa zmení vektor $\vec{v}$ jednotlivých častíc. Berie pri tom do úvahy vektor smerujúci na pozíciu $\vec{x}^*$, vektor smerujúci na pozíciu $\vec{x}^+$ a vektor smerujúci na pozíciu $\vec{x}^!$. Tiež pridá mierny náhodný šum.
3. Aplikuje funkciu *Mutate* na jednotlivé častice, posunie ich podľa vektoru $\vec{v}$.

Presnejší popis algoritmu:

1: n je veľkosť roja.
2: α – určuje koľko zo súčasného vektora rýchlosti a smeru sa má zachovať.
3: β – určuje nakoľko má byť nový vektor ovplyvnený najlepšou nájdenou pozíciou $\vec{x}^*$ častice.
4: γ – určuje nakoľko má byť nový vektor ovplyvnený najlepšou nájdenou pozíciou $\vec{x}^+$ informátorov častice.
5: δ – určuje nakoľko má byť nový vektor ovplyvnený globálnym najlepšie nájdeným riešením $\vec{x}^!$.
6: ε – veľkosť skoku častice.
7: $P = \emptyset$ je množina všetkých častíc.
8: *size*-krát opakujeme: $P = P \cup \{$nová častica s náhodnou pozíciou $\vec{x}$ a náhodným počiatočným vektorom $\vec{v}\}$
9: $\overrightarrow{Best} = \emptyset$

10: Opakujeme, kým $\overrightarrow{Best}$ nie je optimálnym riešením alebo nám vypršal čas:

11: $AssessFitness(\vec{x})$

12: Pre každú časticu $\vec{x} \in P$ opakujeme:

13: Ak $\overrightarrow{Best} = \emptyset$ alebo $f(\vec{x}) \leq f(\overrightarrow{Best})$ potom $\overrightarrow{Best} = \vec{x}$

14: Pre každú časticu $\vec{x} \in P$ opakujeme: (rozhodnutie o tom, ako prebehne mutácia častíc)

15: $\vec{x}^* =$ najlepšia pozícia, ktorú častica $\vec{x}$ navštívila

16: $\vec{x}^+ =$ najlepšia pozícia, navštívená informátormi častice $\vec{x}$

17: $\vec{x}^! =$ najlepšia pozícia, ktorú navštívila ľubovoľná častica

18: Pre každú dimenziu *i* opakujeme:

19: $b =$ náhodné číslo z intervalu 0,0 až β

20: $c =$ náhodné číslo z intervalu 0,0 až γ

21: $d =$ náhodné číslo z intervalu 0,0 až δ

22: $v_i = \alpha v_i + b(x_i^* - x_i) + c(x_i^+ - x_i) + d(x_i^! - x_i)$

23: Pre každú časticu $\vec{x} \in P$ opakujeme: (samotná mutácia)

24: $\vec{x} = \vec{x} + \varepsilon\vec{v}$

25: Vrátime riešenie $\overrightarrow{Best}$.

Táto implementácia sa zakladá na piatich parametroch:

- α – koľko sa zachová z originálneho vektora $\vec{v}$
- β – nakoľko ovplyvní nový vektor najlepšie nájdená pozícia častice. Ak túto hodnotu nastavíme na vysokú, roj bude mať tendenciu rozdeliť sa na množtvo malých skupiniek hill climber-ov.
- γ – nakoľko je nový vektor ovplyvnený najlepšou pozíciou nájdenou informátormi. Záleží pritom aj na počte informátorov, čím viac informátorov, o to viac sa ich najlepšie nájdené riešenie bude podobať na globálne najlepšie riešenie. Čím je informátorov menej, ich najlepšia pozícia bude podobnej kvality ako najlepšia nájdená pozícia jedného riešenia.
- δ – nakoľko je do nového vektora zakomponované globálne najlepšie riešenie. Keď je táto hodnota veľká, celý roj sa správa ako jeden veľký hill climber a pohybuje sa smerom k najlepšie preskúmanej oblasti v stavovom priestore. Keďže algoritmus sa potom príliš podriaďuje princípu intenzifikácie, v moderných implementáciách táto hodnota často býva nastavená na 0.

- ε – určuje akou rýchlosťou sa častica pohybuje. Keď je to veľká hodnota, častice letia veľkou rýchlosťou smerom k dobrým pozíciám, lenže môže sa stať, že cez ne preletia. Na zjemňovanie kvality riešenia potrebujeme nastaviť menšie hodnoty. Často sa používa číslo 1.

Záver

Heuristiky sú veľmi užitočné nástroje na riešenie optimalizačných problémov. Ich užitočnosť a aplikovateľnosť je však obmedzená. Počet skontrolovaných riešení ani zďaleka nedosahuje takú úroveň ako v prípade metaheuristík. Ako sme mohli byť svedkami, za posledných pár desaťročí bolo objavených množstvo dômyselných metaheuristík. Ich stavebnými kameňmi sú jednoduchšie heuristiky, ktoré sú ale pospájané takým spôsobom, že samotný algoritmus sa stáva všeobecne uplatniteľným. Reálny svet je plný zložitých optimalizačných problémov na ktoré exaktné algoritmy nestačia. Metaheuristiky si s nimi hravo poradia. No čím je ľudstvo viac schopné riešiť zložitejšie a zložitejšie problémy, tým viac sa preňho hranica medzi tým, čo je možné a nemožné posúva ďalej.

Mnoho metaheuristík je inšpirovaných prírodou. Tento fakt je fascinujúci, ale niet sa čomu diviť, keďže evolúcia je sama o sebe optimalizačným procesom a mala k dispozícii milióny rokov pre zdokonaľovanie svojich jedincov.

V každom prípade, dopyt po metaheuristikách bude v najbližšej budúcnosti naďalej stúpať. Pri výbere vhodnej reprezentácie problému a správnom nastavení parametrov sú schopné objaviť prijateľné riešenia pre takmer akýkoľvek problém. Za túto vlastnosť vďačia čiastočne tomu, že málo o danom probléme predpokladajú a tiež vďaka tomu, že na rozdiel od heuristík majú v sebe zabudované mechanizmy, ktoré, v prípade ak zablúdia do okolia lokálnych optím, im umožnia dostať sa z nich von a vydať sa druhým, potenciálne nádejnejším smerom. Stále sú však odkázané na zásah človeka, a sme stále veľmi ďaleko od čias, v ktorých by sme algoritmy mohli považovať za univerzálnych riešiteľov problémov, podobných človeku. Metaheuristiky však predstavujú drobný krôčik smerom k takejto budúcnosti.

Zdroje a použitá literatúra

[1]Sean, Luke, 2013, *Essentials of Metaheuristics*, Lulu, 2. vyd., prístupné na http://cs.gmu.edu/⇠sean/book/metaheuristics/

[2]Talbi, El-Ghazali, 1965, *Metaheuristics : from design to implementation*

[3]Pearl, Judea, 1984, *Heuristics : intelligent search strategies for computer problem solving*

[4] Nouraniy,Yaghout a Bjarne Andresenz, 1998, *A comparison of simulated annealing cooling strategies*

[5] Dorigo, Marco, 2004, *Ant colony optimization*

[6] Nilsson, Christian, *Heuristics for the Traveling Salesman Problem*

[7] Blum, Christian a Andrea Roli, *Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison*