VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Bio-inspirované výpočty a shluková analýza

Cluster Analysis based on Bio-Inspired Algorithms

2013 Bc. Michal Rečka

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Michal Rečka**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Bio-inspirované výpočty a shluková analýza
Cluster Analysis based on Bio-Inspired Algorithms

Zásady pro vypracování:

Cílem diplomové práce je prostudovat a naimplementovat vybrané metody z oblasti Swarm Intelligence. Je to technika umělé inteligence založená na kolektivním chování decentralizovaných, samo-organizovaných systémů. Cílem práce je využití vybrané metody pro shlukování dat.

Jednotlivé body práce:
1. Prostudovaní metody Simulated Bee Colony (SBC).
2. Prostudovaní metody Particle Swarm Optimization (PSO).
3. Řešení problému obchodního cestujícího pomocí SBC a PSO, pro demonstraci funkčnosti metod.
4. Použití PSD pro shlukování dat.
5. Experimenty nad vybranou datovou kolekcí.
6. Zhodnocení experimentů.

Seznam doporučené odborné literatury:

[1] James McCaffrey: Use Bee Colony Algorithms to Solve Impossible Problems, MSDN Magazine, April 2011
[2] Ajith Abraham, Swagatam Das and Sandip Roy: Swarm Intelligence Algorithms for Data Clustering, Soft Computing for Knowledge Discovery and Data Mining, 2008, Part IV, 279-313

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Jan Martinovič, Ph.D.**

Datum zadání: 16.11.2012

Datum odevzdání: 07.05.2013

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.*

V Ostravě 26. dubna 2013 .....

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 26. dubna 2013 .....

Rád bych poděkoval všem lidem, kteří mi pomohli při vytváření této práce. Obzvláště bych chtěl poděkovat panu Ing. Janu Martinovičovi, Ph.D. za velmi zodpovědné vedení, četné a účelné konzultace a za lidský přístup ke mě samému.

Abstrakt

Shlukování dat je již delší dobu intenzivně zkoumaným tématem. V praxi můžeme nalézt shlukování dat na každém kroku – vyhledávání podobných obrázků, reklamní kampaně na cílové skupiny, napovídání stránek, které by vás mohly zajímat, či lidí, které byste mohli znát, atd. Tato práce se soustředí na shlukování dat pomocí algoritmu založeném na optimalizaci rojení částic, což je jedna z technik biologicky inspirovaných výpočtů.

Klíčová slova: shlukování dat, optimalizace rojení částic, swarm intelligence, simulace včelí kolonie, bio-inspirované výpočty, diplomová práce

Abstract

Data clustering has been intensively researched topic for quiet a long time. We can find the data clustering in a real world on every corner – searching for similar images, advertisement campaigns aimed to specific groups, suggestion of web pages you could be possibly interested in or of people you might know, etc. This work is focused on data clustering using Paricle Swarm Optimization based algorithm, which is one of biologicaly inspired computing techniques.

Keywords: data clustering, particle swarm optimization, swarm intelligence, simulated bee colony, bio-inspired computing, master thesis

Seznam použitých zkratek a symbolů

ACO	–	Ant Colony Optimization
ES	–	Evoluční strategie
EP	–	Evoluční programování
EVT	–	Evoluční výpočetní techniky
FCM	–	Fuzzy c-means
GP	–	Graph Partitioning
GUI	–	Grafical User Interface
ICS	–	Intra-Cluster Spread
MEPSO	–	Multi-elitist Particle Swarm Optimization
PSO	–	Particle Swarm Optimization
SBC	–	Simulated Bee Colony
SI	–	Swarm Intelligence
TSP	–	Travelling Salesman Problem
WPF	–	Windows Presentation Foundation

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Spousta začínajících (a bohužel i nejen začínajících) programátorů se domnívá, že pokud budou mít k dispozici super výkonný počítač, pak jsou schopni vyřešit úplně všechny problémy, a pokud ne dnes, pak za pár let, kdy se výkon hardware posune zase o něco dále. Toto přesvědčení je však mylné. Není žádným tajemstvím, že výkon počítačů je a vždy bude omezen určitými fyzikálními limity. Tyto limity činí spousty problémů v podstatě neřešitelnými. Tedy ne doslova, ale „neřešitelnými v rozumném čase“. Jsou to často problémy, které jsou v teoretické informace označovány jako NP-úplné [5]. Algoritmy, které tyto problémy jednoznačně řeší, buď mohou trvat příliš dlouho, nebo dokonce ani nemusí existovat.

Ale to, že člověk neví, jak něco udělat ještě neznamená, že to udělat nejde. Lidé jsou dnes a denně neustále fascinováni sofistikovaností výtvorů přírody. Příroda si sama našla způsob, jak řešit spoustu problémů, které by byly i pro dobře organizovaného člověka téměř nemožný úkol. Pokud něco neumíme řešit, je velmi rozumné nechat se inspirovat tam, kde to nějak funguje. Díky této myšlence vzniklo odvětví biologicky inspirovaných výpočtů. Myšlenky jako evoluční teorie, kolektivní spolupráce rojů či hejn, fyzikální zákony, apod. se lidé snaží pochopit a aplikovat na nové problémy.

V této práci se budeme zabývat některými z NP-těžkých problémů, konkrétně problémem obchodního cestujícího, problémem rozdělení grafu a shlukováním dat. Nastíníme problematiku evolučních výpočetních technik, představíme dva konkrétní algoritmy – simulaci včelího roje a optimalizaci rojení částic. Probereme stávající metody shlukování dat a navrhneme vlastní rozšířený algoritmus shlukování dat založený na optimalizaci rojení částic. Seznámíme se s použitým softwarem a nakonec shrneme výsledky experimentů.

1.1 Struktura práce

S problémem obchodního cestujícího se seznámíme v kapitole 2 a v kapitole 3 si představíme základy biologicky inspirovaných technik. Mezi ně patří i algoritmus simulace včelí kolonie (sekce 3.2) a algoritmus optimalizace rojení částic (sekce 3.3). Dalším probíraným problémem je problém rozdělení grafu, který je popsán v kapitole 4. Hlavní náplní této práce je shlukování dat, kterému se věnuje kapitola 5. V sekci 5.3 jsou zmíněny nejčastěji používané algoritmy shlukování dat a sekce 5.4 popisuje, jak se dají data shlukovat pomocí algoritmu optimalizace rojení částic. V kapitole 6 je navržen nový rozšířený algoritmus automatického shlukování dat, založený na optimalizaci rojení částic. Kapitoly 7 a 8 jsou pak již praktické a popisují použitý software a výsledky provedených experimentů.

2 Problém obchodního cestujícího

K vysvětlení tvrzení z úvodu, že některé problémy nejsou řešitelné v rozumném čase, použijeme problém obchodního cestujícího (anglicky Travelling Salesman Problem, zkráceně TSP) [6].

Definice 2.1 *V daném ohodnoceném úplném grafu najděte nejkratší hamiltonovskou kružnici.*

Nebo taky laicky řečeno, že existuje n měst a všechna jsou navzájem propojena silnicí. Každá silnice má svoji délku. Úkolem obchodníka je objet všechna města tak, aby každé navštívil právě jednou, aby urazil co nejmenší vzdálenost a na konci své cesty se vrátil do města, ze kterého vyrazil.

Příklad 2.1
Mějme města A, B, C a D. Vzdálenosti mezi nimi jsou $|AB| = 20$, $|AC| = 42$, $|AD| = 35$, $|AB| = 20$, $|DB| = 34$, $|DC| = 12$ a $|CB| = 30$. Pro tento případ nechť není rozdíl mezi vzdáleností z města X do města Y a naopak. Nejkratší okružní cesta obchodního cestujícího je $A \to D \to C \to B \to A$ přičemž obchodník urazí vzdálenost $35 + 12 + 30 + 20 = 97$. ■

Pokud je n dostatečně malé, pak lze tento problém vyřešit hrubou silou – tedy vyzkoušet každou možnou cestu a následně vybrat tu nejlepší. Možných kombinací existuje $n!$, tedy pokud bychom měli na vstupu 10 měst, pak bychom museli vyzkoušet $10! = 3628800$ různých cest. Dejme tomu, že máme k dispozici počítač, který je schopen vyhodnotit 10^9 cest za sekundu. Tabulka 1 ukazuje závislost časové složitosti na počtu měst n. Jak lze vidět, tak se zvyšujícím se n exponenciálně roste doba potřebná k ověření všech možných kombinací hrubou silou. Termínem „neřešitelný problém v rozumném čase“ tedy míníme to, že se výsledku člověk nemusí dožít (např. již pro $n = 20$).

n	Čas
15	21 minut 47 sekund
16	5 hodin 48 minut 42 sekund
17	5 dní 2 hodin 48 minut 7 sekund
18	2 měsíce 16 dní 2 hodin 26 minut 13 sekund
19	4 roky 10 měsíců 8 dní 22 hodin 18 minut 20 sekund
20	78 let 1 měsíc 4 dny 14 hodin 6 minut 48 sekund
21	1620 let 4 dny 8 hodin 22 minut 51 sekund

Tabulka 1: Závislost časové složitosti nalezení řešení problému obchodního cestujícího hrubou silou na velikosti vstupu při 10^9 vyhodnocení cest za sekundu

Experimenty spojené s TSP lze nalézt v sekci 8.1 a vizualizaci řešení TSP pomocí algoritmu SBC v sekci 7.3.

3 Biologicky inspirované techniky

Ne nadarmo se říká, že v jednoduchosti je síla. Mravenec sám o sobě je pro člověka hloupý tvor, jehož smyslem života je hledat potravu či stavět a bránit mraveniště. Nicméně pokud se podíváme na mravenčí kolonii, zjistíme, že jejich chování jako celku je velmi sofistikované. Jde o tzv. Swarm Intelligence (zkráceně SI), neboli kolektivní inteligenci [1]. S kolektivní inteligencí se můžeme setkat prakticky denně, podíváme-li se na ptačí hejna, včelí kolonie, mravenčí kolonie, hejna ryb, dokonce částečně i davy lidí. Obecně jde o to, že každý jedinec se řídí sadou jednoduchých pravidel. Pokud se však podobnými pravidly řídí všichni jedinci v daném společenství, postupem času začne společenství vykazovat určité známky organizovaného chování a to i přes to, že jedince nikdo doopravdy neřídí.

3.1 Evoluční algoritmy

Evoluční výpočetní techniky (zkráceně EVT) jsou numerické algoritmy inspirované teorií evoluce Darwina a Mendela. Hlavní myšlenkou je křížení jedinců, předání genů jejich potomkům, kteří podléhají mutacím, a následný zánik starých jedinců a jedinců, kteří nejsou dostatečně vhodní pro dané životní prostředí. Většinu EVT tvoří tzv. evoluční algoritmy. Kromě nich EVT ale zahrnuje i genetické programování, evoluční hardware aj. Zobecněný cyklus evolučních algoritmů je znázorněn na obrázku 1.

Postup evolučního algoritmu je přibližně následující:

1. Definice vstupních parametrů – řídící parametry, ukončovací parametry a funkce Fitness [11][1] (funkce, která vypočítává podle atributů jedince a jiných dalších informací číselnou hodnotu – vhodnost jedince).

2. Vygenerování počáteční populace jedinců – jedinci jsou vektory o dimenzi $D+1$, kde první položka slouží k uchování vhodnosti (výsledek funkce Fitness) a zbylých D položek jsou hodnoty D optimalizovaných parametrů funkce Fitness, dále atributy jedince. Každý jedinec tedy představuje jedno konkrétní řešení. Atributy jedinců počáteční populace jsou nastaveny náhodně v rámci povolených hodnot.

3. Výpočet vhodnosti pomocí funkce Fitness u všech jedinců.

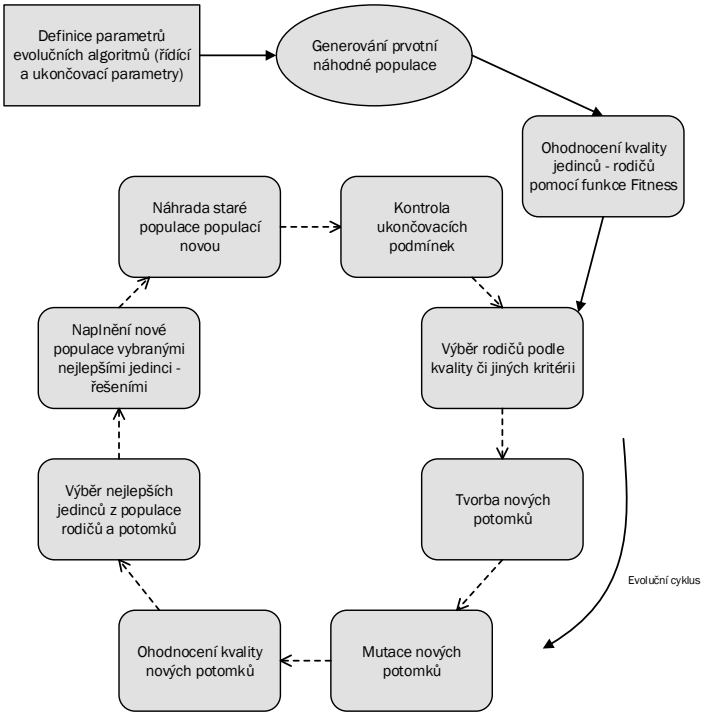
4. Podle vhodnosti (příp. i jiných kritérií) jsou vybráni rodiče.

5. Dochází ke skřížení rodičů a vznikají noví potomci.

6. Noví potomci podléhají mutacím, tzn. nějakým náhodným způsobem jsou drobně pozměněni.

7. Je vypočtena vhodnost každého potomka stejně jako v bodu 3.

[1] Přesněji bychom měli psát „účelová funkce“ (cost function). Fitness, neboli vhodnost, je pouze normalizovaná návratová hodnota účelové funkce. V této práci pracujeme pouze s termínem „funkce Fitness“ a myslíme tím účelovou funkci. Je to z toho důvodu, že nezáleží na tom, jestli porovnáváme čísla z intervalu $\langle 0; 1 \rangle$, nebo jakákoli reálná čísla, a protože proces normalizace zbytečně zpomaluje výpočty



Obrázek 1: Zobecněný cyklus evolučního algoritmu. Obrázek převzat z [11]

8. Jsou vybráni nejlepší jedinci.
9. Nejlepší jedinci „postupují“ do další generace.
10. Jedinci, kteří se nedostali do další generace vymírají (jsou smazáni a nahrazeni těmi lepšími) a pokračuje se opět krokem 4.

Kroky 4 - 10 tvoří jednu iteraci, která se opakuje tak dlouho, dokud nejsou splněny ukončovací podmínky definované v kroku 1. Výše zmíněný postup je velmi obecný, proto se bude u každého konkrétního algoritmu mírně lišit. Podobné algoritmy, které se však neřídí kroky 1 - 10 se neřadí jako Evoluční algoritmy, ale pouze jako algoritmy patřící k EVT. Dokonce podle některých evolucionistů vůbec k EVT nepatří – např. v případě algoritmu optimalizace mravenčí kolonie (Ant Colony Optimization, zkráceně ACO).

Často se evoluční algoritmy dají poměrně snadno paralelizovat, což je velká výhoda, která umožňuje efektivně využít možnosti moderního hardware.

Žádný algoritmus ale není univerzální a všemocný. I evoluční algoritmy mají své nevýhody. Jednou z hlavních nevýhod je to, že jsou velmi citlivé na vstupní parametry. Stačí minimálně změnit jeden z řídících parametrů a výsledky mohou začít vycházet úplně jinak [2]. Velmi kritická je definice funkce Fitness, která musí být navržena opravdu

[2] Výsledky evolučních algoritmů z pravidla bývají při každém spuštění jiné. Tady však myslíme výsledky, které se liší hodně. Např. může dojít k překonání lokálního extrému, který s předchozími parametry překonat nešlo.

sofistikovaně často i s různými penalizacemi pro neplatná řešení apod. Další nevýhodou je to, že si nikdy nemůžeme být jistí, že algoritmus našel opravdu nejlepší řešení (pokud jej však předem neznáme). Algoritmus může např. uvíznout v lokálním extrému funkce Fitness, nebo prostě skončit dříve, než se jedinci stihnou dopracovat k lepšímu řešení. Tento nedostatek se však zpravidla řeší opakovaným spouštěním algoritmu s různými vstupními parametry, což statisticky pravděpodobnost chybného výsledku minimalizuje. Z principu se evoluční algoritmy tedy hodí na řešení optimalizačních problémů, u kterých nepotřebujeme znát výsledek naprosto přesně, ale stačí nám alespoň přibližná hodnota [11].

3.2 Algoritmus simulace včelí kolonie

Jedním z konkrétních příkladů biologicky inspirovaných technik je algoritmus simulace včelí kolonie (Simulated bee colony algorithm, zkráceně SBC). Je inspirován chováním včelího roje při hledání zdrojů potravy. Jedinci včelí kolonie se v tomto případě dělí na 3 typy: aktivní včely, průzkumníci a neaktivní včely. *Aktivní včely* hledají optimální zdroj potravy a průběžně se poohlížejí kolem, jestli není lepší zdroj potravy o kousek dál. *Průzkumníci* také hledají optimální zdroj potravy, ale pátrají naprosto náhodně po okolí. *Neaktivní včely* vyčkávají v úle, pozorují ostatní včely a učí se od nich, aby později mohly některé jiné aktivní včely nahradit.

Každá včela si pamatuje informace o nejlepším zdroji jídla, který doposud našla. Zdrojem jídla se myslí konkrétní řešení daného problému, např. u TSP je to posloupnost měst. Nejlepší zdroj jídla je takový, který po dosazeni do funkce Fitness dává jako výsledek nejnižší/nejvyšší hodnotu – nejlepší vhodnost. Aby mělo použití SBC smysl, musí existovat „podobné“ zdroje jídla, které se liší pouze nějakým detailem (simulace hledání sousedských zdrojů jídla). Např. u TSP získáme podobný (sousedský) zdroj potravy tak, že prohodíme navzájem pořadí dvou náhodných měst. Pokud nějaká podobnost mezi řešeními neexistuje, pak celá myšlenka SBC selhává.

SBC je inicializován populací aktivních včel, průzkumníků a neaktivních včel. Poměr aktivních včel, průzkumníků a neaktivních včel bývá zhruba 75:10:15 [6]. Každá včela má na začátku v paměti náhodný zdroj jídla. Včelí úl drží informace o nejlepším zdroji potravy – $gBest$. Nejlepší zdroj potravy je aktualizován po každém návratu včel do úlu, pokud byl nalezen lepší.

Každý průzkumník prohledává zcela náhodně definované okolí úlu a pokud najde lepší zdroj potravy, pak si ho vždy zapamatuje místo původního. Průzkumníci se nikdy nemýlí. Pokud průzkumník našel lepší zdroj potravy, předvádí po návratu do úlu tzv. Waggle Dance. Waggle Dance je proces, kdy včela předává informace neaktivním včelám, čekajícím v úle. To znamená, že se snaží přesvědčit každou neaktivní včelu, která má v paměti horší zdroj jídla, aby převzala zdroj lepší. Pravděpodobnost přesvědčení u Waggle Dance je dána P_P. Pokud průzkumník nenašel lepší zdroj potravy, vrací se do úlu a jeho akce v tomto cyklu končí.

Aktivní včela sbírá potravu ze zdroje, který má v paměti. Je definováno N_{max}, což je maximální počet návštěv jednoho zdroje jídla, než se jeho obsah vyčerpá. Aktivní včela při každé návštěvě zdroje jídla prohledává sousedské okolí. Pokud nenajde lepší zdroj

jídla, pak u svého zdroje jídla přičte 1 k počítadlu návštěv tohoto zdroje. Jakmile hodnota počítadla návštěv zdroje přesáhne N_{max}, včela se vrací do úlu, stává se neaktivní a jedna z původně neaktivních včel ji vystřídá a stane se aktivní. Pokud však včela najde lepší zdroj jídla, přesune se na něj a dále sbírá potravu z něj, tedy aktualizuje svou paměť na nový zdroj jídla a začne počítat jeho návštěvy opět od 0. Důležitá vlastnost aktivních včel je to, že se mohou zmýlit s pravděpodobností P_O a přijmout horší, nebo odmítnout lepší zdroj potravy. Pokud aktivní včela našla lepší zdroj potravy, po návratu do úlu předvádí Waggle Dance a končí svou činnost v tomto cyklu.

Neaktivní včela pouze vyčkává v úle, pozoruje přilétající včely a učí se od nich. Jak již bylo zmíněno, pokud se jedna z aktivních včel vrátí s tím, že vyčerpala svůj zdroj jídla, je vystřídána jednou z neaktivních včel.

Pseudokód algoritmu simulace včelí kolonie je znázorněn v algoritmu 1.

```
1  Vstup:
2      včely: celkový počet včel,
3      X_A: počet aktivních včel,
4      X_N: počet neaktivních včel,
5      X_P: počet včel průzkumníků,
6      iterace: maximální počet cyklů (provedení akce všech včel),
7      N_max: maximální počet návštěv jednoho zdroje jídla (u aktivních včel),
8      P_P: pravděpodobnost přesvědčení neaktivních včel o přijetí lepšího zdroje jídla od jiných včel,
9      P_O: pravděpodobnost, že se aktivní včela zmýlí a příjme horší, nebo odmítne lepší, zdroj jídla.
10 Výstup:
11     gBest: Globální nejlepší paměť, obsahuje nejlepší doposud nalezené řešení.
12
13 gBest = GenerujNahodnyZdrojJidla()
14 for i < počet včel do:
15     if i < X_A then: bee_i.Status = Aktivní;
16     else if i < (X_A + X_P) then: bee_i.Status = Průzkumník;
17     else: bee_i.Status = Neaktivní;
18     bee_i.ZdrojJidla = GenerujNahodnyZdrojJidla()
19     if bee_i.ZdrojJidla.Vhodnost > gBest.Vhodnost then: gBest = bee_i.ZdrojJidla;
20 end for
21
22 for i < iterace do:
23     for j < počet včel do:
24         if bee_i.Status = Aktivní then:
25             zdroj = GenerujSousedskyZdrojJidla()
26             if zdroj.Vhodnost > bee_i.ZdrojJidla.Vhodnost then:
27                 if random > P_O then: // pokud se včela nesplete, přejde na lepší zdroj
28                     bee_i.ZdrojJidla = zdroj
29                     if gBest.Vhodnost > bee_i.ZdrojJidla.Vhodnost then: gBest = bee_i.ZdrojJidla;
30                     DoWaggleDance()
31                 else:
32                     počet návštěv ++
33                 end if
34             else:
35                 if random <= P_O then: // pokud se včela splete a přejde na horší zdroj
36                     bee_i.ZdrojJidla = zdroj
37                     DoWaggleDance()
38                 else:
39                     počet návštěv ++
```

```
40              end if
41            end if
42            if počet návštěv > N_max then: // pokud byl zdroj vyčerpán, vystřídání včely
43              bee = náhodná neaktivní včela
44              bee.Status = Aktivní
45              bee_i.Status = Neaktivní
46            end if
47          else if bee_i.Status = Průzkumník then:
48            zdroj = GenerujNahodnyZdrojJidla()
49            if zdroj.Vhodnost > bee_i.ZdrojJidla.Vhodnost then:
50              bee_i.ZdrojJidla = zdroj
51              // návrat do úlu
52              if gBest.Vhodnost > bee_i.ZdrojJidla.Vhodnost then: gBest = bee_i.ZdrojJidla;
53              DoWaggleDance()
54            end if
55          else: // v podstatě nedělá nic aktivního, pouze čeká
56          end if
57        end for
58    end for
59
60    DoWaggleDance:
61    vstup: zdrojJidla : zdroj jídla aktivní včely nebo průzkumníka
62
63    for each bee in neaktivní včely do:
64        if zdrojJidla.Vhodnost > bee.ZdrojJidla.Vhodnost and random <= P_P then: bee.ZdrojJidla =
              zdrojJidla;
65    end for
```

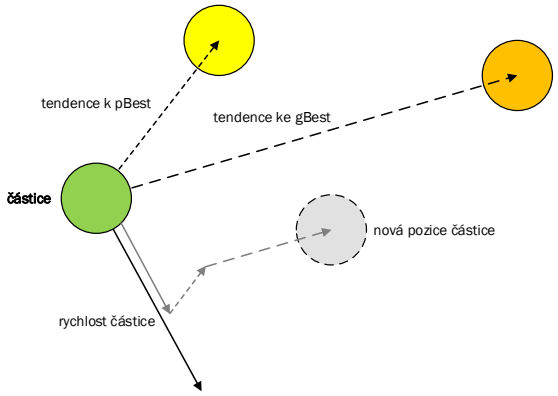
Algoritmus 1: Pseudokód SBC algoritmu

3.3 Algoritmus optimalizace rojení částic

Dalším příkladem evolučních technik je optimalizace rojení částic (Particle Swarm Optimization, zkráceně PSO). Algoritmus simuluje chování ptačího hejna/roje částic. Hlavní myšlenka spočívá v tom, že hejno ptáků prohledává oblast a hledá vrcholek hory. Žádný z ptáků neví, kde přesně vrcholek leží, ale každý ví, který z nich našel doposud nejvyšší místo, a v následujícím kroku tohoto ptáka následují.

Jedinci se v případě PSO nazývají částice. Atributy každé částice představují souřadnice v prohledávaném prostoru o D dimenzích. Dále má navíc každá částice D-rozměrný vektor rychlostí, ve kterém má zaznamenáno, jak rychle se v dané dimenzi pohybuje, a pozici nejlepšího řešení – pBest (podle vhodnosti dané funkcí Fitness), které doposud našla.

Počáteční populace je inicializována s náhodnými pozicemi v rámci prohledávaného prostoru a s náhodnými vektory rychlostí. Při každé změně pozice a rychlosti je přepočítána vhodnost částice pomocí funkce Fitness a pokud je nová pozice lepší, než stávající pBest, pak částice nahradí pBest novou pozicí. Roj má společnou paměť, ve které je udržována nejlepší pozice, kterou našly všechny částice dohromady – gBest. Každá částice tedy ví, kde globálně nejlepší doposud nalezené řešení leží. Jedinci kontrolují gBest a pokud zjistí, že je jejich pBest lepší, pak nahradí gBest svou pBest.



Obrázek 2: Skutečný pohyb částice ovlivněný jejími tendencemi

Pohyb částice je složen ze tří směrů:

- vlastní směr částice,
- k lokálně nejlepšímu řešení pBest,
- ke globálně nejlepšímu řešení gBest.

Poměr složení těchto směrů lze ovlivnit vstupními konstantami ω, c_1 resp. c_2. Princip skládání směrů pohybů lze vidět na obrázku 2. V dalším kroku částice upraví svou rychlost podle rovnice (1) a přesune se na novou pozici pomocí rovnice (2).

$$v_d(t+1) = \omega \cdot v_d(t) + c_1 \cdot rand \cdot (pBest_{i,d} - x_{i,d}(t)) + c_2 \cdot rand \cdot (gBest_d - x_{i,d}(t)), \quad (1)$$

$$x_{i,d}(t+1) = x_{i,d}(t) + v_d(t+1), \quad (2)$$

kde d – je pořadí dimenze,
$v_d(t)$ – je rychlost částice v t-tém kroku,
$x_{i,d}(t)$ – je pozice částice v t-tém kroku,
$pBest_{i,d}$ – je nejlepší doposud nalezená pozice částice,
$gBest_d$ – je nejlepší doposud nalezená pozice v celé populaci,
$rand$ – je náhodné číslo $0 < rand \leq 1$,
ω – je konstanta setrvačnosti částice ve svém vlastním směru,
c_1 – je konstanta ovlivňující směr k $pBest_{i,d}$,
c_2 – je konstanta ovlivňující směr ke $gBest_d$.

Jakmile se částice přesune na novou pozici, je přepočítána vhodnost a celý cyklus se opakuje. Rychlost částice však má tendenci prudce růst, takže částice brzy dosáhne hranice prohledávané oblasti. Proto je rozumné zavést maximální rychlost V_{max}. Pokud

rychlost částice překročí V_{max}, pak je jí buď vygenerována rychlost nová, nebo je prostě omezena na rychlost V_{max} [11].

Pseudokód PSO popisuje algoritmus 2.

```
1  Vstup:
2      iterace : maximální počet iterací
3      vstupní parametry : c1, c2, ω
4      omezení: hranice prohledávaného prostoru
5  Výstup:
6      gBest: nejlepší nalezené řešení v populaci
7
8  for j=< počet částic do:
9      inicializuj náhodnou pozici a rychlost částice particlej
10 end for
11 for i < iterace do:
12     for j =< počet částic do:
13         vypočítej rychlost částice podle (1) s danými vstupními parametry
14         vypočítej pozici částice podle (2)
15         uprav pozici částice podle daných omezení
16         vhodnost = F_cost(p_i)
17         if vhodnost > pBest.Vhodnost then:
18             pBest = pozice
19         end if
20         if pBest.Vhodnost > gBest.Vhodnost then:
21             gBest = pBest
22         end if
23     end for
24 end for
```

Algoritmus 2: Pseudokód PSO algoritmu

PSO má i své nedostatky [11]:

- pokud není nastavena V_{max}, pak se často částice velmi rychle vzdalují od nejlepšího doposud nalezeného řešení,
- pokud má funkce Fitness mnoho lokálních extrémů, má PSO tendenci k předčasné konvergenci,
- nastavení vstupních parametrů může být samo o sobě problémem.

4 Problém rozdělení grafu

Problém rozdělení grafu (Graph Partitioning, zkráceně GP) je definován následovně [7].

Definice 4.1 *Nechť $G(V, E)$ je neorientovaný graf, kde V je množina vrcholů, E je množina hran, $n = |V|$ je počet vrcholů a $m = |E|$ je počet hran. Disjunktní k-násobné vyvážené rozdělení grafu G je $D = (V_1, V_2, \ldots, V_k)$, kde $2 \leq k \leq \lfloor \frac{n}{2} \rfloor$ s omezením $|V_i| \approx |V_j|$ $\forall i, j$. Dále graf musí být spojitý, neboli nesmí obsahovat oddělené podgrafy. Z toho vyplývá, že každý vrchol musí mít alespoň 1 souseda. Chceme minimalizovat cenu rozdělení, což je počet hran, které mají vrcholy v různých oddílech. Nechť $c(e)$ je jednotná cena hrany: $(v_i, v_j) = 1$ $\forall i, j$ a nechť*

$$F_i = \{(v_a, v_b) \in E \mid v_a \in V_i, v_b \notin V_i\} \; \forall a, b \tag{3}$$

je množina externích vrcholů oddílů V_i, pak je cílem minimalizovat

$$c(D) = \frac{1}{2} \sum_{i=1}^{k} \sum_{F_i} c(e) \tag{4}$$

Počet $c(D)$ se nazývá počet řezů rozdělení.

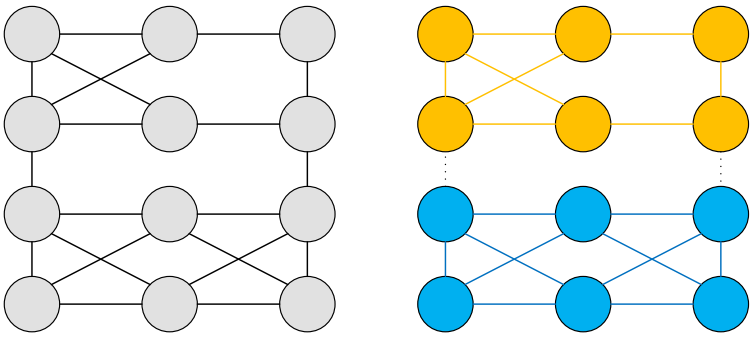
Obrázek 3 ukazuje příklad problému rozdělení grafu. Existuje několik cest, jak zajistit omezení vyváženosti, že velikost všech oddílů musí být přibližně stejná. Jedním z běžných přístupů je omezit velikost největšího oddílu v P na méně nebo rovno $\lceil \frac{n}{k} \rceil$. Dalším přísnějším přístupem je vyžadovat, že velikost prvních $k - 1$ oddílů bude rovna $\lceil \frac{n}{k} \rceil$ a proto velikost k-tého oddílu bude $|V| - \sum_{i=1}^{k-1} |V_i|$. Je známo, že rozdělení grafu je NP-úplný problém [9]. Obecně není řešení hrubou silou proveditelné. Pokud je počet vrcholů stejně dělitelný počtem oddílů, což znamená, že $\frac{n}{k} = \lfloor \frac{n}{k} \rfloor = \lceil \frac{n}{k} \rceil$, pak je výpočetní vzorec počtu možných rozdělení grafu $\varphi(n, k)$ dán rovnicí (5)

$$\varphi(n, k) = \frac{\left[\prod_{i=1}^{k-1} \binom{n-(|V_i| \cdot (i-1))}{|V_i|} \right]}{k!} \tag{5}$$

Funkce φ v rovnici (5) roste velmi rychle, jak se n zvyšuje. Např. je 126 způsobů, jak rozdělit graf o 10 vrcholech do 2 oddílů stejné velikosti, ale jednoduše vypadající problém rozdělení grafu o 40 vrcholech do 8 oddílů stejné velikosti má 4706245478917332058722773 76 možných řešení. I kdyby bylo možné vyhodnotit 10^9 možných řešení za sekundu, prozkoumání všech možností ke zjištění optimálního rozdělení by zabralo přibližně $14,9$ miliard let [7].

4.1 Řešení pomocí SBC

V článku [7] byla zmíněna možnost řešení problému rozdělení grafu pomocí algoritmu simulace včelí kolonie. Ve své práci McCaffrey uvedl, že na známých datových souborech



Obrázek 3: Problém rozdělení grafu: vlevo zadaný graf, vpravo nejlepší řešení, kde jsou přerušeny pouze 2 hrany

dosáhl pomocí SBC stejných nebo i lepších výsledků rozdělení grafů, než nejlepší doposud známé algoritmy. Zároveň zmínil časovou náročnost uvedené metody, která se pohybuje v minutách až hodinách. Algoritmus SBC se tedy hodí pouze pro případy, kdy nám mnohem více záleží na přesnosti výsledků, než na době exekuce programu.

Rozhodli jsme se rozdělení grafu pomocí SBC také vyzkoušet. Algoritmus SBC jsme si popsali již v sekci 3.2. Pro možnost aplikace na problém GP bylo zapotřebí následujících úprav:

1. Reprezentace dat jako neorientovaný graf.
2. Vygenerování náhodného rozdělení grafu.
3. Vygenerování „sousedského“ rozdělení grafu.
4. Upravení výpočtu funkce Fitness.

Výpočet funkce Fitness odpovídá rovnici (4). Postup bodu 2 – vygenerování náhodného rozdělení – může být následující:

1. Náhodně vybrat 1 vrchol a zařadit ho do prvního oddílu.
2. Náhodně vybrat sousedský vrchol dříve vybraného a zařadit jej do prvního oddílu.
3. Náhodně vybrat 1 vrchol z právě generovaného oddílu, který má alespoň 1 sousední vrchol, který ještě nebyl přiřazen do žádného oddílu.
4. Náhodně vybrat 1 sousední vrchol vybraného vrcholu, který ještě není přiřazen k žádnému oddílu, a zařadit jej do právě generovaného oddílu.
5. Opakovat kroky 3-4, dokud není generovaný oddíl naplněn dostatečným počtem vrcholů.
6. Opakovat kroky 3-5, dokud nejsou naplněny všechny požadované oddíly kromě posledního.

7. Poslední oddíl jsou všechny vrcholy, které zůstaly doposud nepřiřazeny.

Problém nastal ve chvíli, kdy jsme chtěli generovat sousedská řešení. V případě TSP byl tento úkol jednoduchý, protože se jednalo o úplný graf. Díky tomu jsme si mohli dovolit prohodit libovolné dva vrcholy, aniž bychom se museli starat, jestli jsme náhodou neporušili spojitost grafu. V případě neúplného grafu to ale takto dělat nelze. Generování sousedského řešení GP musí totiž splňovat ještě jednu podmínku navíc: **odebráním vrcholu z oddílu ani přidáním vrcholu k oddílu nesmíme porušit spojitost tohoto ani souvisejícího oddílu**. To znamená, že pokud změníme příslušný oddíl daného vrcholu, musíme vždy ověřit spojitost obou oddílů, kterých se změna týká (toho, ze kterého vrchol odebíráme + toho, do kterého vrchol přidáváme).

Ověření spojitosti podgrafu není triviální záležitost. V podstatě musíme zjistit, jestli existuje cesta přes všechny vrcholy. Toho lze docílit tak, že začneme v jednom bodě a rekurzivně budeme procházet přes všechny sousedy daného vrcholu a následně přes všechny sousedy těchto sousedů a tak dále a při tom hlídat, aby nedocházelo k zacyklení. Pokud alespoň jedna rekurzivní cesta prochází přes všechny vrcholy, můžeme říct, že je podgraf spojitý. Pokud však žádná rekurzivní cesta přes všechny vrcholy nalezena nebyla, znamená to, že podgraf spojitý není, z čehož vyplývá, že daný vzor takto vložit/odstranit nemůžeme, musíme zvolit jiný a celý proces kontroly spojitosti provést znovu. Pokud je vrcholů v grafu hodně, představuje už jen operace vygenerování sousedského řešení netriviální problém. A pokud si uvědomíme, že algoritmus SBC je stejně jaké ostatní evoluční techniky založen na kolektivní inteligenci, tedy na spolupráci mnoha jedinců, pak v případě, že bychom chtěli tento proces paralelizovat, by nám velmi snadno mohla jednoduše dojít paměť.

Výše zmíněná podmínka nám navíc nabourává i dříve navržený postup vygenerování náhodného řešení. Může totiž nastat situace, kdy vygenerujeme $k - 1$ spojitých oddílů, ale poslední oddíl spojitý nebude. Navržený algoritmus totiž poslední oddíl vůbec nekontroluje, pouze do něj přiřadí zbylé body, ať už mezi nimi hrana je, nebo není. Jsou tedy 2 možnosti. Buď navržený postup opakujeme, dokud není splněna podmínka spojitosti posledního oddílu, nebo musíme navrhnout algoritmus jiný.

Pokud bychom netrvali striktně na podmínce, aby všechny oddíly musely vždy být spojité, pak by se problém dal řešit poměrně jednoduše podobně jako v případě TSP. Nicméně za těchto okolností to může představovat tak velký výkonnostní problém, že jeho řešení pro nás přestalo být aktuálně zajímavé.

5 Shlukování dat

Shlukování (clustering) slouží k reprezentaci velkých datových souborů menším počtem prototypů či shluků (clusters) [1]. Přináší jednoduchost do modelování dat a proto hraje klíčovou roli v procesu objevování informací a data miningu. Úlohy data miningu dnes vyžadují rychlé a precizní dělení (partitioning) obrovských datových souborů, které může využívat různé atributy či vlastnosti. To pro změnu vyžaduje náročné výpočetní požadavky na relevantní techniky shlukování. Ukázalo se, že biologicky inspirované algoritmy, známé jako Swarm Intelligence, tyto požadavky splňují a úspěšně se využívají při řešení velkého počtu shlukovacích problémů z reálného světa [1].

5.1 Úvod do shlukování dat

Shlukování znamená proces rozdělení nepopsaného datového souboru do skupin podobných objektů. Každá skupina, nazývaná „shluk“, se skládá z objektů, které jsou navzájem podobné a odlišné od objektů v jiných skupinách. V minulých pár desetiletích hrála analýza shluků klíčovou roli v různých odvětvích od strojírenství (učení strojů, umělá inteligence, rozpoznávání znaků, mechanické inženýrství, elektro-inženýrství), výpočetních věd (web mining, analýza prostorové databáze, kolekce textových dokumentů, segmentace obrázků), biologické a medicínské vědy (genetika, biologie, mikrobiologie, paleontologie, psychiatrie, patologie), až po vědy o Zemi (geografie, geologie, dálkový průzkum), sociální vědy (sociologie, psychologie, archeologie, vzdělávání) a ekonomie (marketing, byznys).

Z pohledu učení strojů shluky odpovídají skrytým vzorům v datech. Hledání shluků je typ učení bez učitele a výsledný systém reprezentuje *datový koncept*. K problému shlukování dat je přistupováno z různých oblastí vědění jako statistika (multivariační analýza), teorie grafu, algoritmy maximalizace očekávání, umělé neuronové sítě, evoluční výpočetní techniky atd. Vědci po celém světě pravidelně přicházejí s novými algoritmy reagujícími na vzrůstající složitost rozsáhlých reálných dat.

Data mining je mocná nová technologie, jejímž cílem je extrakce skrytých prediktivních informací z velkých datových souborů. Nástroje data miningu předpovídají budoucí trendy a chování a umožňují činit aktivnější, znalostmi řízená, rozhodnutí v byznyse. Proces objevování znalostí z databází vyžaduje rychlé a automatické shlukování velmi rozsáhlých datových souborů s několika atributy různých typů. Pro klasické shlukovací techniky to představuje závažný problém. Nedávno přitáhly biologicky inspirované algoritmy pozornost několika výzkumníků z oblasti rozpoznávání znaků a shlukování. Shlukovací techniky založené na SI nástrojích údajně překonaly spoustu klasických metod pro rozdělení komplexních reálných dat [1].

5.2 Definice problému

Vzor (pattern) je fyzická nebo abstraktní struktura objektů. Navzájem se liší společnou sadou atributů nazývanou vlastnosti (features), které dohromady reprezentují vzor. Nechť $R = R_1, R_2, \ldots R_n$ je sada n vzorů nebo datových bodů, kde každý má d vlast-

ností. Tyto vzory mohou být také reprezentovány profilovou datovou maticí $X_{n\times d}$ o n d-rozměrných řádkových vektorech. i-tý řádkový vektor X_i charakterizuje i-tý objekt ze sady R a každý prvek $X_{i,j}$ v X_i odpovídá j-té reálné hodnotě vlastnosti ($j = 1, 2, \ldots, d$) i-tého vzoru ($i = 1, 2, \ldots, n$). Pomocí $X_{n\times d}$, se shlukovací algoritmus pokouší nalézt oddíl $C = C_1, C_2, \ldots, C_K$ o K třídách takových, že podobnost vzorů ve stejném shluku je maximální a vzory z různých shluků se liší, co nejvíce to lze. Oddíly by měly počítat s následujícími vlastnostmi:

- každý shluk by měl mít alespoň jeden přiřazený vzor, tedy $C_i \neq \varnothing, \forall i \in 1, 2, \ldots, K$,
- dva různé shluky by neměly mít žádný vzor společný. Tedy $C_i \cap C_j = \varnothing, \forall i \neq j$ a $i, j \in 1, 2, \ldots, K$. Tato vlastnost je vyžadována pro ostré (crisp) shlukování. Při neostrém (fuzzy) shlukování tato vlastnost neexistuje,
- každý vzor by měl rozhodně být přiřazen ke shluku, tedy $\bigcup_{i=1}^{K} C_i = R$.

Protože daný datový soubor lze rozdělit na oddíly mnoha způsoby, pro dodržování výše zmíněných vlastností musí být definována funkce Fitness (určitá míra vhodnosti rozdělení). Problém se následně mění na nalezení oddílu C^* s optimální nebo přibližně optimální vhodností v porovnání s ostatními možnými řešeními $\mathbf{C} = C^1, C^2, \ldots, C^{N(n,K)}$, kde

$$N(n, K) = \frac{1}{K!} \sum_{i=1}^{K} (-1)^i \binom{K}{i}^i (K - i)^i \tag{6}$$

je počet možných oddílů, což je stejné co

$$\underset{C}{\text{Optimize}} f(X_{n\times d}, C), \tag{7}$$

kde C je jeden oddíl ze sady $\mathbf{C}$ a f je statisticko-matematická funkce, která vypočítává vhodnost oddílu na základě míry podobnosti vzorů. Definice odpovídající míry podobnosti hraje základní roli ve shlukování. Nejpopulárnější způsob vyhodnocení podobnosti mezi dvěma vzory je měření vzdálenosti. Nejpoužívanějším způsobem měření vzdálenosti je Eukleidovská vzdálenost, která je mezi dvěma d-rozměrnými vzory X_i a X_j dána vzorcem

$$d(X_i, X_j) = \sqrt{\sum_{p=1}^{d} (X_{i,p} - X_{j,p})^2} = \|X_i - X_j\| \tag{8}$$

Bylo ukázáno v [3], že problém shlukování je NP-těžký jakmile počet shluků překročí 3.

5.3 Klasické shlukovací algoritmy

Shlukování dat vychází převážně ze dvou přístupů: *hierarchický* (hierarchical) a *oddílový* (partitional). U obou těchto typů existuje mnoho podtypů a různých algoritmů pro nalezení shluků. U hierarchického shlukování je výstupem strom znázorňující sekvenci shlukování, kde každý shluk je oddílem datového souboru. Hierarchické algoritmy mohou

být *aglomerační* (zdola-nahoru) nebo *dělící* (shora-dolů). Aglomerační algoritmy začínají s každým prvkem jako samostatným shlukem a postupně je spojují do větších shluků. Dělící algoritmy začínají s celým datovým souborem a postupně jej rozdělují na menší shluky. Hierarchické algoritmy mají dvě základní výhody. Zaprvé není potřeba určit počet tříd předem a zadruhé jsou nezávislé na počátečních podmínkách. Nicméně hlavním nedostatkem techniky hierarchického shlukování je, že prvky přiřazené do shluku nelze přesunout do jiného shluku. Navíc mohou selhat při oddělování překrývajících se shluků kvůli chybějícím informacím o celkovém tvaru či velikosti shluků. V této práci se hierarchickými algoritmy dále nezabýváme.

Algoritmy oddílového shlukování se naopak pokoušejí rozložit datový soubor přímo do sady disjunktních shluků. Snaží se optimalizovat určitá kritéria. Funkce kritérií může zdůraznit lokální strukturu dat ať už přiřazením shluků do vrcholů funkce hustoty pravděpodobnosti, nebo globální strukturou. Typicky globální kritéria zahrnují minimalizaci míry odlišnosti mezi vzory uvnitř každého shluku, ale zároveň i maximalizaci odlišnosti různých shluků. Výhody hierarchických algoritmů jsou nevýhody oddílových algoritmů a naopak. Rozsáhlý průzkum různých shlukovacích technik lze nalézt v [4].

Shlukování může být prováděno ve dvou různých režimech: *ostrý* (crisp) a *neostrý* (fuzzy). V ostrém shlukování jsou shluky disjunktní a v zásadě se nepřekrývají. Jakýkoli vzor může v tomto případě patřit pouze do právě jedné třídy. V případě neostrého shlukování může vzor patřit do všech tříd s určitým stupněm neostrého členství .

Nejčastěji používaný iterativní K-means algoritmus pro oddílové shlukování se zaměřuje na minimalizaci ICS (Intra-Cluster Spread), kterou lze pro K středů shluků definovat jako

$$ICS(C_1, C_2, \ldots, C_K) = \sum_{i=1}^{K} \sum_{X_i \in C_i} \|X_i - m_i\|^2 \tag{9}$$

Algoritmus K-means (nebo Hard c-means) začíná s K středy shluku (tyto středy jsou na počátku vybrány náhodně nebo jsou odvozeny z nějaké předem dané informace). Každý vzor v datovém souboru je poté přiřazen k nejbližšímu středu shluku. Středy jsou aktualizovány pomocí průměru asociovaných vzorů. Proces je pak opakován, dokud není splněna nějaká ukončovací podmínka.

Fuzzy c-means (zkráceně FCM) algoritmus se zdá být nejpopulárnějším algoritmem v oblasti neostrého shlukování. V klasickém FCM algoritmu je minimalizována funkce součtu uvnitř shluků J_m pro rozvoj vhodných středů shluků:

$$J_m = \sum_{j=1}^{n} \sum_{i=1}^{c} (u_{i,j})^m \|X_j - V_i\|^2, \tag{10}$$

kde V_i je i-tý střed shluku, X_j je j-tý d-rozměrný vektor dat a $\|.\|$ je skalárním součinem-indukovaná norma v d rozměrech. Pokud máme c tříd, můžeme určit jejich středy shluků

V_i pro $i = 1$ až c pomocí následující rovnice:

$$V_i = \frac{\sum_{j=1}^{n} (u_{i,j})^m X_j}{\sum_{j=1}^{n} (u_{i,j})^m} \tag{11}$$

Zde je m nějaké reálné číslo ovlivňující stupeň členství, kde $m > 1$. Nyní diferencujeme výkonnostní kritérium vzhledem k V_i ($u_{i,j}$ považujeme za konstantu) a vzhledem k $u_{i,j}$ (V_i považujeme za konstantu) a dosadíme za ně nulu, lze získat následující vztah [1]:

$$u_{i,j} = \left[\sum_{k=1}^{c} \left(\frac{\|X_j - V_i\|^2}{\|X_j - V_k\|^2} \right)^{\frac{1}{m-1}} \right]^{-1} \tag{12}$$

$$i = 1, \ldots, c$$

$$j = 1, \ldots, n$$

U je $c \times n$ matice, $U = [u_{i,j}]$, kde $u_{i,j}$ je reálné číslo z intervalu $\langle 0; 1 \rangle$, které udává stupeň členství j-tého vzoru v i-tém shluku. Čím vyšší hodnota $u_{i,j}$ je, tím více vzor patří do daného shluku.

5.4 Shlukování dat pomocí PSO

Úsilí v oblasti výzkumu umožnilo pohlížet na shlukování dat jako na optimalizační problém. Tento přístup nabízí možnost aplikovat PSO algoritmus pro vyvinutí sady kandidátských středů shluků a tím pádem určení přibližného optimálního rozdělení datového souboru. Významnou výhodou PSO je jeho schopnost vypořádat se s lokálními optimy udržováním, rekombinací a porovnáváním několika kandidátských řešení současně. Oproti tomu heuristiky lokálního hledání, jako algoritmus simulovaného žíhání, vylaďují pouze jedno kandidátské řešení a jsou notoricky slabé ohledně vypořádávání se s lokálními extrémy. Deterministické lokální hledání, které je používáno v algoritmech jako K-means vždy konverguje do nejbližšího lokálního extrému od startovní pozice hledání.

Algoritmus shlukování na základě PSO byl poprvé představen Omranem. Jeho výsledky ukázaly, že metoda založená na PSO překonala K-means, FCM a pár dalších dosavadních shlukovacích algoritmů. V jeho metodě používal pro posouzení výkonu shlukovacích algoritmů míru fitness na základě kvantizační chyby. Kvantizační chyba je definována jako:

$$J_e = \frac{\sum_{i=1}^{K} \sum_{\forall X_j \in C_i} \frac{d(X_j, V_i)}{n_i}}{K}, \tag{13}$$

kde C_i je i-tý střed shluku a n_i je počet datových bodů patřících do i-tého shluku. Každá částice v PSO algoritmu reprezentuje možnou množinu K středů shluků jako:

$$\vec{Z}_i = \boxed{\vec{V}_{i,1} \mid \vec{V}_{i,2} \mid \cdots \mid \vec{V}_{i,K}},$$

kde $\vec{V}_{i,p}$ odkazuje na p-tý vektor středu shluku i-té částice. Kvalita každé částice je měřena následující funkcí Fitness:

$$f(Z_i, M_i) = w_1 \bar{d}_{max}(M_i, X_i) + w_2(R_{max} - d_{min}(Z_i)) + w_3 J_e \tag{14}$$

R_{max} je maximum hodnoty vlastnosti v datovém souboru a M_i je matice reprezentující přiřazení vzorů do shluků i-té částice. Každý prvek $m_{i,k,p}$ značí, jestli vzor X_p patří do shluku C_k i-té částice. Uživatelem definované konstanty w_1, w_2 a w_3 jsou použity k vážení významu jednotlivých složek. K tomu

$$\bar{d}_{max} = \max_{k \in 1,2,\ldots,K} \left\{ \sum_{\forall X_p \in C_{i,K}} \frac{d(X_p, V_{i,k})}{n_{i,k}} \right\} \tag{15}$$

a

$$d_{min}(Z_i) = \min_{\forall p,q,p \neq q} \{d(V_{i,p}, V_{i,q})\} \tag{16}$$

je minimum Eukleidovské vzdálenosti mezi nějakými dvěma shluky. $n_{i,k}$ je počet vzorů, které patří do shluku $C_{i,k}$ částice i. Funkce Fitness je několikanásobný optimalizační problém, který minimalizuje vzdálenost uvnitř shluku, maximalizuje separaci mezi shluky a snižuje kvantizační chybu. Pseudokód PSO shlukování je shrnut v algoritmu 3.

```
1  inicializace každé částice s K náhodnými středy shluků
2  for t < maximum iterací do:
3      for all částice i do:
4          for all vzor X_p in datový soubor do:
5              vypočítat Eukleidovskou vzdálenost X_p od všech středů shluků
6              přiřadit X_p do shluku s nejbližším středem k X_p
7          end for
8          vypočítat funkci fitness f(Z_i, M_i)
9      end for
10     nalézt osobní nejlepší a globální nejlepší pozici každé částice
11     aktualizovat středy shluků podle vzorce změny rychlosti (1) a pozice (2) PSO
12 end for
```

Algoritmus 3: Pseudokód shlukování dat pomocí základního PSO algoritmu

Tento základní algoritmus pak prošel mnoha úpravami a hybridizacemi. Výsledky PSO vypadají velmi slibně, lepší shlukování než pomocí K-means ukázali např. Paterlini a Krink či Cui a spol. [1].

6 Algoritmus automatického shlukování založený na PSO

Za posledních pár let bylo snahou nalézt shluky v komplexních datových souborech pomocí evolučních výpočetních technik stráveno obrovské množství času. Už se ale tolik neřešilo, jak určit optimální počet shluků. Většina existujících shlukovacích technik založených na evolučních algoritmech přijímá počet tříd K jako vstup, místo aby jej samy určily za běhu. Nicméně ve spoustě praktických situací nemusí být vhodný počet skupin v novém datovém souboru znám, nebo může být nemožné jej určit i třeba jen přibližně. Například při shlukování množiny dokumentů plynoucí z dotazu na vyhledávač se počet tříd K mění pro každou množinu dokumentů, která z interakce s vyhledávačem vyplývá. Zároveň pokud je datový soubor popsán vysoko-dimenzionálním vektorem vlastností (což se stává velmi často), může být prakticky nemožné data vizualizovat kvůli zjištění jejich počtu shluků.

Nalezení optimálního počtu shluků v rozsáhlém datovém souboru bývá obvykle náročný úkol. Problém byl několikrát zkoumán, nicméně výsledek je stále neuspokojující. Lee a Antonsson použili metodu založenou na Evoluční strategii (zkráceně ES) k dynamickému shlukování datového souboru. Navrhli ES implementované jednotlivce proměnné délky k hledání středů a optimálního počtu shluků zároveň. Sarkat a spol. ukázal přístup k dynamické klasifikaci datového souboru pomocí Evolučního programování (zkráceně EP), kdy jsou optimalizovány dvě Fitness funkce zároveň: jedna dává optimální počet shluků, zatímco druhá vede ke správné identifikaci každého středu shluku. Bandyopadhyay a spol. navrhl genetický algoritmus proměnné textové délky k řešení problému dynamického shlukování pomocí jediné Fitness funkce. Omran a spol. přišel s automatickým těžkým shlukovacím schématem. Algoritmus začíná rozdělením datového souboru do relativně velkého počtu shluků kvůli snížení efektu inicializace. Optimální počet shluků je vybrán pomocí binárního PSO. Nakonec jsou středy vybraných shluků vyladěny pomocí K-means algoritmu. Autoři aplikovali algoritmus na segmentaci přírodních, syntetických a multi-spektrálních obrázků [1].

V této sekci probereme neostrý shlukovací algoritmus, který dokáže automaticky určit počet shluků v daném datovém souboru. Algoritmus je založen na algoritmu PSO s vylepšenými konvergenčními vlastnostmi.

6.1 Modifikace klasického PSO

Kanonické PSO bylo podrobeno empirickému a teoretickému zkoumání několika vědců. V mnoha případech je konvergence předčasná, zejména pokud používá roj malou hodnotu parametru setrvačnosti/hmotnosti ω či konstrikčního koeficientu. Protože globálně nejlepší řešení nalezené brzy ve vyhledávacím procesu může být lokální minimum, používáme multi-elitní strategii hledání globálně nejlepšího výsledku PSO nazvanou MEPSO navrženou Abrahamem a spol [1]. Definuje pro každou částici tempo růstu β. Jakmile je vhodnost částice t-té iterace vyšší než vhodnost částice $(t-1)$-té iterace, je β zvýšena o 1. Poté, co jsou určeny lokálně nejlepší ze všech částic v každé generaci, přesuneme lokální nejlepší částici, která má vhodnost vyšší než globálně nejlepší, do oblasti kandidátů. Následně je globálně nejlepší částice nahrazena lokálně nejlepší s nejvyšší hodnotou

tempa růstu β. Vhodnost nové globálně nejlepší částice je tak jako tak vždycky vyšší, než hodnota staré globálně nejlepší částice. Tento přístup snižuje pravděpodobnost rychlé konvergence do lokálního extrému. Pseudokód MEPSO je popsán v algoritmu 4. Algoritmus MEPSO jsme rozšířili navíc o různé typy částic (viz sekce 6.3). Výsledný rozšířený algoritmus budeme značit MEPSO*.

```
1  for t < t_max do:
2      for j < N do: // velikost roje je N
3          if částice_j.Vhodnost v t-tém kroku > částice_j.Vhodnost v (t − 1)-tém kroku then:
4              β_j = β_j + 1;
5          end if
6          aktualizovat pBest_j
7          if lokální pBest_j.Vhodnost > gBest.Vhodnost nyní then:
8              vlož pBest_j do kandidátské oblasti
9          end if
10     end for
11     vypočítat β pro každého kandidáta a poznačit kandidáta s β_max
12     nahradit gBest kandidátem s β_max
13 end for
14 nahradit gBest částicí s nejvyšší vhodností
```

Algoritmus 4: Pseudokód algoritmu MEPSO

6.2 Reprezentace částice

V navržené metodě pro n datových bodů, o d dimenzích a pro uživatelem specifikované maximum počtu shluků c_{max} je částice vektorem reálných čísel s dimenzí $c_{max} + c_{max} \times d$. Prvních c_{max} hodnot jsou kladná desetinná čísla v intervalu $\langle 0; 1\rangle$, kde každé řídí, jestli bude odpovídající shluk aktivován (tj. bude opravdu použit pro klasifikaci dat) nebo ne. Zbývající proměnné jsou rezervovány pro c_{max} středů shluků, každý o d dimenzích. Jednu částici lze tedy zobrazit jako:

$$\vec{Z}_i(t) = \begin{array}{|c|c|c|c|c|c|c|c|} \hline T_{i,1} & T_{i,2} & \cdots & T_{i,c_{max}} & \vec{V}_{i,1} & \vec{V}_{i,2} & \cdots & \vec{V}_{i,c_{max}} \\ \hline & & & & flag_{i,1} & flag_{i,2} & \cdots & flag_{i,c_{max}} \\ \hline \end{array}$$

Aktivační prahy — Středy shluků

Každý pravděpodobný střed shluku $m_{i,j}$ má d vlastností a asociovaný binární příznak $flag_{i,j}$. Střed shluku je aktivní (tj. vybrán pro klasifikaci), pokud $flag_{i,j} = True$, nebo neaktivní, pokud je $flag_{i,j} = False$. Každý příznak je nastaven v závislosti na hodnotě aktivačního prahu $T_{i,j}$. Poznamenejme, že tyto příznaky jsou skryté informace spojené se středy shluků a neúčastní se PSO mutace částice. Pravidlo pro výběr shluků určených jednou částicí je:

$$\textbf{if } T_{i,j} > 0,5 \textbf{ then } flag_{i,j} = True \textbf{ else } flag_{i,j} = False \tag{17}$$

Nutno dodat, že příznaky u potomka lze změnit pouze pomocí prahů $T_{i,j}$ (podle (17)). Jakmile částice přeskočí na novou pozici v průběhu optimalizace podle (1) a (2), jsou

nejprve získány hodnoty T, které jsou následně použity pro výběr hodnot m (nastavení příznaku u středů nových shluků). Pokud v průběhu mutace některý z prahů T u částice překročí hodnotu 1, je opraven na 1 případně na 0. Pokud je zjištěno, že žádný příznak v částici nebyl nastaven na $True$ (všechny aktivační prahy jsou menší než $0,5$), vybereme náhodně 2 prahy a přenastavíme je na náhodné hodnoty mezi $0,5$ a $1,0$. Proto je počet možných shluků vždy minimálně 2.

Abychom se lépe orientovali, budeme pro orientaci dále uvažovat dvou rozměrný prostor (rovinu), ve kterém budeme shlukovat body o souřadnicích X, Y. Pro hledání maximálně 5 shluků ($c_{max} = 5$) bodů v rovině se středy shluků $[4,5; 18,34]$, $[89,43; 116,93]$, $[783,44; 56,29]$, $[901,84; 72,66]$, $[39,14; 206,82]$ a aktivačních prazích $0,64$, $0,32$, $0,87$, $0,19$, $0,71$ by částice vypadala takto:

$0,64$	$0,32$	$0,87$	$0,19$	$0,71$	$4,5$	$18,34$	$89,43$	$116,93$	$783,44$	$56,29$	$901,84$	$72,66$	$39,14$	$206,82$

Na základě prvních pěti hodnot se pak určí příznaky $flag$ pro každý shluk. Tyto příznaky zde nejsou úmyslně uvedeny, aby bylo zřetelněji vidět, jaká čísla se doopravdy účastní evolučního cyklu.

Výpočet matice U musí být lehce upraven, aby nepočítal s neaktivními shluky:

$$u_{i,j} = \begin{cases} 1 & \text{pokud je } i\text{-tý shluk aktivní a } v_i = x_j, \\ \left[\sum_{k=1}^{c_a} \left(\frac{\|x_j - v_i\|}{\|x_j - a_k\|} \right)^{\frac{2}{m-1}} \right]^{-1} & \text{pokud je } i\text{-tý shluk aktivní,} \\ 0 & \text{jinak,} \end{cases} \tag{18}$$

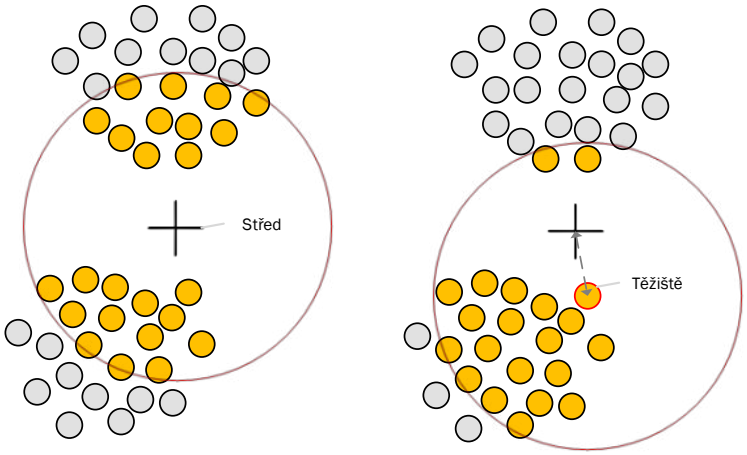
kde c_a je počet aktivních shluků, x_j je j-tý vzor, v_i je i-tý střed shluku, a_k je k-tý střed aktivního shluku a m je nějaký koeficient ovlivňující stupeň členství, v našem případě $1,5 \leq m \leq 3$.

6.3 Typy částic

Ukázalo se, že není špatný nápad rozšířit algoritmus i o myšlenky ze SBC. Konkrétně jde o různé typy částic.

Pro další výklad je zapotřebí nejprve definovat pojem těžiště shluku. Částice, která letí prohledávaným prostorem po iteracích posouvá své středy shluků. Jednotlivé shluky jsou definovány jejich středy. Částice se svými souřadnicemi mohou během letu dostat do takových pozic, že vzniklé souřadnice středů shluků leží úplně mimo rozložení vzorů, nebo mohou být v prostoru mezi shluky. Obzvláště, pokud prohledáváme velkou převážně prázdnou oblast. Pokud budeme vycházet z myšlenky, že čím blíže je střed k jakémukoli bodu A, tím větší je pravděpodobnost, že je blíže i k ostatním bodům shluku, ve kterém bod A leží, můžeme navrhnout systém shlukování podle těžišť. **Těžiště shluku** je tedy vzor, který leží nejblíže ke středu shluku. Rozdíl mezi shlukem definovaným středem a shlukem definovaným těžištěm lze vidět na obrázku 4.

Možná vás napadne otázka, proč tedy částice neobsahuje souřadnice těžišť místo souřadnic středů. Představme si dva shluky, které jsou od sebe velmi vzdálené. Dejme tomu, že částice podle globálně nejlepšího řešení ví, že by měla jeden střed posunout



Obrázek 4: Vlevo shluk podle středu, vpravo shluk podle těžiště. Červená kružnice značí orientační hranici příslušnosti k danému shluku. Vzory mimo tuto hranici mohou být součástí jiných shluků

směrem ke druhému shluku. Pokud by byla definována souřadnicemi těžiště, nemohla by se od něj nikdy vzdálit na větší vzdálenost, než je definovaná maximální rychlost částice. Tím pádem by částice stagnovala stále na přibližně stejné pozici a po celý zbytek běhu algoritmu by nám byla k ničemu. Proto je důležité migrace částic provádět na souřadnicích *středů* i přes to, že shluky podle těžišť většinou dávají lepší výsledky (naopak tomu je např. u shluků ve tvaru O).

Samotné shlukování podle středů shluků může velmi pomalu konvergovat. Nicméně samotné shlukování podle těžišť taky nemusí vždy přinést nejlepší výsledky – viz dříve zmíněné shluky tvaru U, O apod. Nabízí se tedy možnost vytvořit různé typy částic:

- *základní* – shlukuje podle souřadnic středů,
- *konzervativní* – shlukuje podle souřadnic těžišť,
- *náhodný* – shlukuje podle souřadnic těžišť, ale souřadnice středů jsou v každé iteraci vygenerovány zcela náhodně.

Důvod, proč jsme zavedli i náhodný typ částice je snaha o minimalizaci možnosti konvergence do lokálního extrému funkce Fitness. Pokud by např. hned na začátku běhu algoritmu všechny částice konvergovaly do lokálního extrému, pak by už neměly vůbec žádnou šanci se z něj dostat pryč. Přidáním náhodných částic se tato možnost alespoň trochu zlepší.

Poměr částic je volitelný, my jsme použili 15 % základních, 75 % konzervativních a 10 % náhodných.

6.4 Funkce Fitness

Kvalitu oddílu lze soudit pomocí odpovídajícího indexu validity shluku. Indexy validity shluku odpovídají statisticko-matematickým funkcím používaným k ohodnocení výsledků shlukovacích algoritmů. Obecně slouží index validity shluku ke dvěma účelům. Zaprvé může být použit k určení počtu shluků a zadruhé zjišťuje nejlepší odpovídající oddíl. Jeden tradiční přístup pro určení optimálního počtu tříd je spustit algoritmus opakovaně s různými počty tříd na vstupu a poté vybrat rozdělení dat s nejlepší mírou validity. Ideálně by měl index validity počítat s následujícími aspekty dělení:

- *soudružnost* – vzory v jednom shluku by si měly být navzájem podobné, jak nejvíce to lze. Rozptyl vhodností vzorů uvnitř shluku indikuje soudružnost a kompaktnost shluku,
- *oddělenost* – shluky by měly být dobře odděleny. Vzdálenost mezi středy shluků (např. jejich Eukleidovská vzdálenost) indikuje oddělenost shluku.

V této práci je funkce Fitness založena na validačním indexu, který byl navržen v [10]. Validační index je definován jako:

$$I = \frac{\sum_{i=1}^{c_{max}} \sum_{j=1}^{n} u_{i,j}^2 \|x_j - v_i\|^2 + \frac{1}{c_a(c_a-1)} \sum_{i=1}^{c_a} \sum_{k=1,k\neq i}^{c_a} \|a_i - a_k\|^2}{\min_{i\neq k} \|a_i - a_k\|^2}, \tag{19}$$

kde c_{max} je maximální počet shluků, c_a je počet aktivních shluků určených pro klasifikaci, $u_{i,j}$ je stupeň členství j-tého vzoru do i-tého shluku, x_j je j-tý vzor, v_x je střed/těžiště x-tého shluku a a_x je x-tý střed/těžiště aktivních shluků.

První část čitatele je součet vzdáleností vzorů od středů/těžišť shluků, do kterých přísluší podle stupně členství $u_{i,j}$. Druhá část čitatele je průměrná vzdálenost středů/těžišť aktivních shluků a funguje jako postihová funkce, která zlepšuje chování funkce pro počet aktivních shluků $c_a \to n$.

V [10] je uvedena ještě jedna postihová funkce – ve jmenovateli – a to z důvodu horšího chování funkce Fitness v případě, kdy $m \to \infty$ (m je koeficient ovlivňující výpočet stupně členství, viz rovnice (18)). Vzhledem k tomu, že funkce dává rozumné výsledky pro $1,5 \leq m \leq 3$ [2], nemá smysl se $m \to \infty$ moc zabývat.

Pomocí I lze získat optimální počet shluků minimalizací hodnoty validačního indexu. Funkce Fitness tedy může být napsána takto:

$$F_{cost} = \frac{1}{I + eps}, \tag{20}$$

kde eps je nějaká velmi malá konstanta, zde je použita hodnota $0,00002$. Maximalizace funkce Fitness tedy znamená minimalizaci validačního indexu I.

6.5 Ošetření nekorektních stavů

Částice se nesmí dostat mimo hranice prohledávaného prostoru. Při načítání dat je tedy potřeba specifikovat platné rozmezí $lowerBound$ a $upperBound$ a to pro každou dimenzi zvlášť. Je rozumné nastavit výchozí $lowerBound_x$ a $upperBound_x$ na nejnižší respektive nejvyšší hodnotu x-té souřadnice vzorů. Tím, že specifikujeme meze pro každou dimenzi zvlášť docílíme efektivnějšího shlukování, kdy se částice nebudou snažit hledat tam, kde žádná data nejsou. Jakmile se částice přesune na novou pozici, je pro každou dimenzi zkontrolována hodnota dané souřadnice a pokud leží mimo definované hranice, je vygenerována náhodně uvnitř těchto hranic (pouze v této dimenzi). Pokud bychom pozici pouze nastavili na $lowerBound$ respektive $upperBound$, pak by se nám mohlo stát, že všechny částice budou postupem času rozmístěny po okrajích prohledávané plochy, což je nám k ničemu.

Jak již bylo dříve zmíněno, částice mají tendenci prudce zvyšovat svou rychlost. Pokud je rychlost příliš velká, částice skáče od okraje k okraji prohledávané zóny a vzhledem k ošetření překročení hranic vysvětlené v předchozím odstavci se shlukování stává zcela náhodným generováním souřadnic středů shluků. Proto je zapotřebí tuto rychlost nějak korigovat. Jednou z možností je jedna globální konstanta v_{max} při jejímž překročení se na ni rychlost omezí. Tento přístup ale není vhodný pro nesymetrická data, kde jsou větší rozdíly mezi rozsahy jednotlivých dimenzí. Např. pokud máme rovinu o šířce 2000 a výšce 150, pak pokud nastavíme v_{max} na 200, což je pro velikost šířky plochy ještě stále akceptovatelná rychlost (částice musí uskutečnit alespoň 10 iterací, než se dostane z jednoho konce na druhý), pak se začne částice velmi rychle dostávat mimo hranice dimenze výšky, čímž se začne generovat náhodně. Další alternativou je definovat v_{max} pro každou dimenzi zvlášť. Tato možnost už bude fungovat, nicméně je velmi pracná. Pokud bychom např. chtěli shlukovat data o 60-ti dimenzích, pak bychom museli 60-krát upravit maximální rychlost, což obnáší i 60-krát prozkoumat platný rozsah dané dimenze. Abychom se tomuto vyhnuli, definujeme maximální rychlost $maxVelocityPercent$ jako poměr rozsahu dané dimenze v procentech. Tím pádem stačí zadefinovat pouze jednu hodnotu a pro každou dimenzi se už omezující rychlost vypočte za běhu. Poznamenejme, že rozsah hodnot aktivačních prahů je vždy 1 a vzhledem k tomu, že je vhodnost částice na tyto hodnoty velice citlivá, není dobré je zatěžovat stejně velkým omezením rychlosti, jako klasická data. V našem případě je maximální rychlost pohybu v dimenzích aktivačních prahů nastavitelná další procentuální konstantou.

Námi definovaná funkce Fitness je navržená tak, aby částice směřovaly středy shluků co nejdál od sebe. Nicméně pokud bychom chtěli použít funkci jinou, mohli bychom narazit na to, že nejlepší vhodnost částice bude taková, kdy všechny středy shluků budou v jednom bodě (všechny vzdálenosti najednou budou rovny 0 a výsledná vhodnost např. ∞). Ideálně by se měla samozřejmě změnit funkce Fitness. Pokud funkci Fitness z nějakého důvodu měnit nechceme, pak je nutností definovat minimální vzdálenosti $minDistance_i$ mezi jednotlivými souřadnicemi. Ošetření minimálních vzdáleností však nemusí být tak jednoduché, jak se může zdát. Posunutím jedné částice o kousek dál může nastat kolize s jinou částicí. Je třeba tedy posunout i všechny ostatní případné kolidující částice. Pak se zase může stát, že poslední částice překročila hranici prohledávaného prostoru, tím

pádem se musí všechny částice posunout zpátky. Aby vůbec bylo možné vždy částice takto umístit, musí platit vztah:

$$minDistance_i < \frac{upperBound_i - lowerBound_i}{N}, \tag{21}$$

kde N je počet částic a i je index dimenze.

Nakonec abychom minimalizovali shluky, ve kterých jsou méně než 2 vzory, ověříme po každém přesunutí částice na novou pozici, zda každý aktivní shluk obsahuje alespoň 2 vzory s největším stupněm členství $u_{i,j}$. Pokud ne, pak vypočítáme všechny souřadnice této částice jako průměr souřadnic ostatních částic. Provedeme dříve zmíněné korekce, přiřadíme stupně členství znovu a pokud ani tentokrát neobsahuje některý aktivní shluk alespoň 2 vzory, pak posuneme středy shluků do jejich těžišť nezávisle na typu částice.

6.6 Shrnutí algoritmu

V každé iteraci přesuneme částice na nové pozice podle rovnic (1) a (2), čímž získáme u každé z nich nové pozice středů shluků a jejich aktivační prahy. Na základě aktivačních prahů určíme pomocí rovnice (17), které středy shluků se budou účastnit klasifikace v této iteraci. Ke každému aktivnímu shluku najdeme jeho těžiště a podle rovnice (18) vypočítáme stupně členství (matice U) každého vzoru ke každému shluku. Jakmile máme k dispozici všechny tyto informace, vypočítáme u každé částice pomocí rovnic (19) a (20) vhodnost nalezeného řešení. Po vyhodnocení všech kvalit řešení aktualizujeme globálně nejlepší nalezené řešení (podle sekce 6.1). Jakmile je dokončen předem definovaný počet iterací, je globálně nejlepší řešení nahrazeno lokálním nejlepším řešením s nejvyšší vhodností (ze všech částic) a provede se u něj tzv. „defuzzyfikace“, neboli každý vzor se přiřadí pouze do toho shluku, u kterého má největší stupeň členství.

Mezi hlavní výhody tohoto algoritmu patří beze sporu fakt, že dokáže automaticky určit optimální počet shluků, dá se poměrně snadno paralelizovat (každá částice může běžet v rámci jednoho cyklu ve vlastním vlákně) a má poměrně dobré výsledky shlukování.

Jako každý algoritmus má i MEPSO* své nevýhody. Už z principu fungování PSO je nutné, aby částice měla rychlost a polohu ve všech dimenzích prohledávaného prostoru. Tím pádem se algoritmus stává nepoužitelným pro obrovská data. Každá částice zabírá paměťo velikosti c_{max} pro aktivační prahy $+d.c_{max}$ pro souřadnice středů shluků $+n.c_{max}$ prvků matice U, celkem tedy $c_{max}.(1 + d + n)$ reálných čísel.

7 Software

Součástí této práce je následující software:

- implementace algoritmů SBC, PSO a MEPSO*,
- hlavní program pro shlukování dat pomocí algoritmu MEPSO* a FCM,
- výpočet TSP pomocí SBC algoritmu,
- vizualizace řešení problému TSP a rozdělení grafu pomocí SBC algoritmu,
- výpočet TSP pomocí PSO algoritmu.

Veškeré kódy jsou napsány v jazyce C# frameworku Microsoft .NET 4.5. Uživatelská rozhraní slouží pouze k usnadnění interakce uživatele s algoritmy. Cílem této práce nebylo vytvoření konzistentního a stabilního systému, ale prozkoumat možnosti shlukování dat. Z tohoto důvodu nebyla věnována pozornost optimálnímu rozložení prvků GUI a u aplikací nejsou ošetřeny všechny možné vstupy. Přesto by alespoň hlavní aplikace neměla spadnout a veškeré chyby by měla přinejmenším zapisovat do souborů v adresáři Logs.

7.1 Shlukování dat pomocí algoritmů MEPSO* a FCM

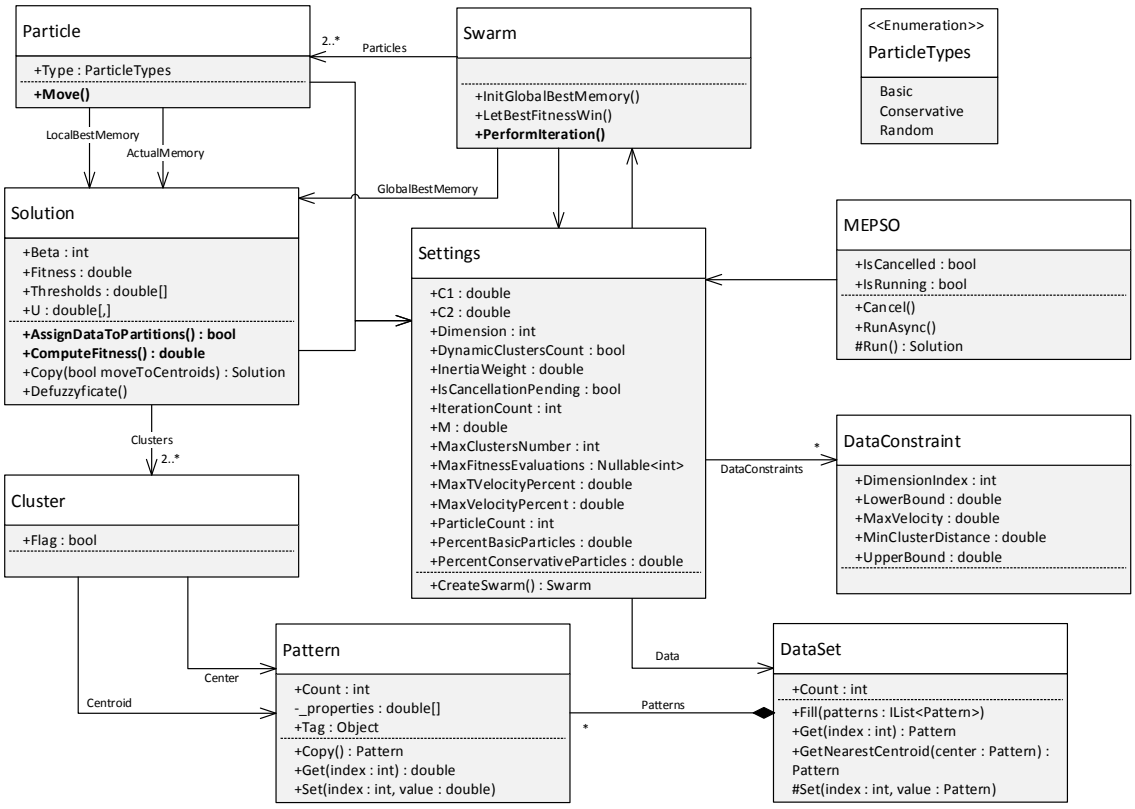
Hlavní aplikace této práce. Je napsána ve WPF a slouží k testování shlukování dat pomocí algoritmu MEPSO* či FCM, přičemž každý algoritmus má své vlastní okno. V horní části se vždy nachází pole pro nastavení vstupních dat a parametrů pro jednotlivé algoritmy. V levé části pak jsou zobrazeny presentery výsledků. Tyto presentery slouží k prezentaci vypočtených dat v přehlednější formě. Pro algoritmus MEPSO* jsou implementovány tyto presentery:

- *export dat* – slouží k uložení výsledků do binárního souboru, nebo zjednodušenému uložení výsledků do textového souboru,
- *2-D body* – slouží ke zobrazení shlukování bodů v rovině,
- *podobné obrázky* – zobrazí shlukování podobných obrázků, přičemž cesta k adresáři s obrázky musí být specifikována v konfiguračním souboru.

V dolní části pak uživatel může sledovat a spouštět či zastavit průběh algoritmů.

Výběr vstupních dat pro algoritmus MEPSO* je složen ze čtyř kroků:

1. Výběr textového souboru.
2. Výběr prvního řádku souboru a specifikace regulárního výrazu oddělujícího jednotlivé hodnoty sloupců.
3. Výběr sloupců, které se budou účastnit klasifikace (např. pokud data obsahují nějaké poznámky, štítky apod. lze je zde vyloučit a není třeba upravovat samotný soubor).



Obrázek 5: Třídní diagram algoritmu MEPSO*

4. Specifikace hranic prohledávané oblasti.

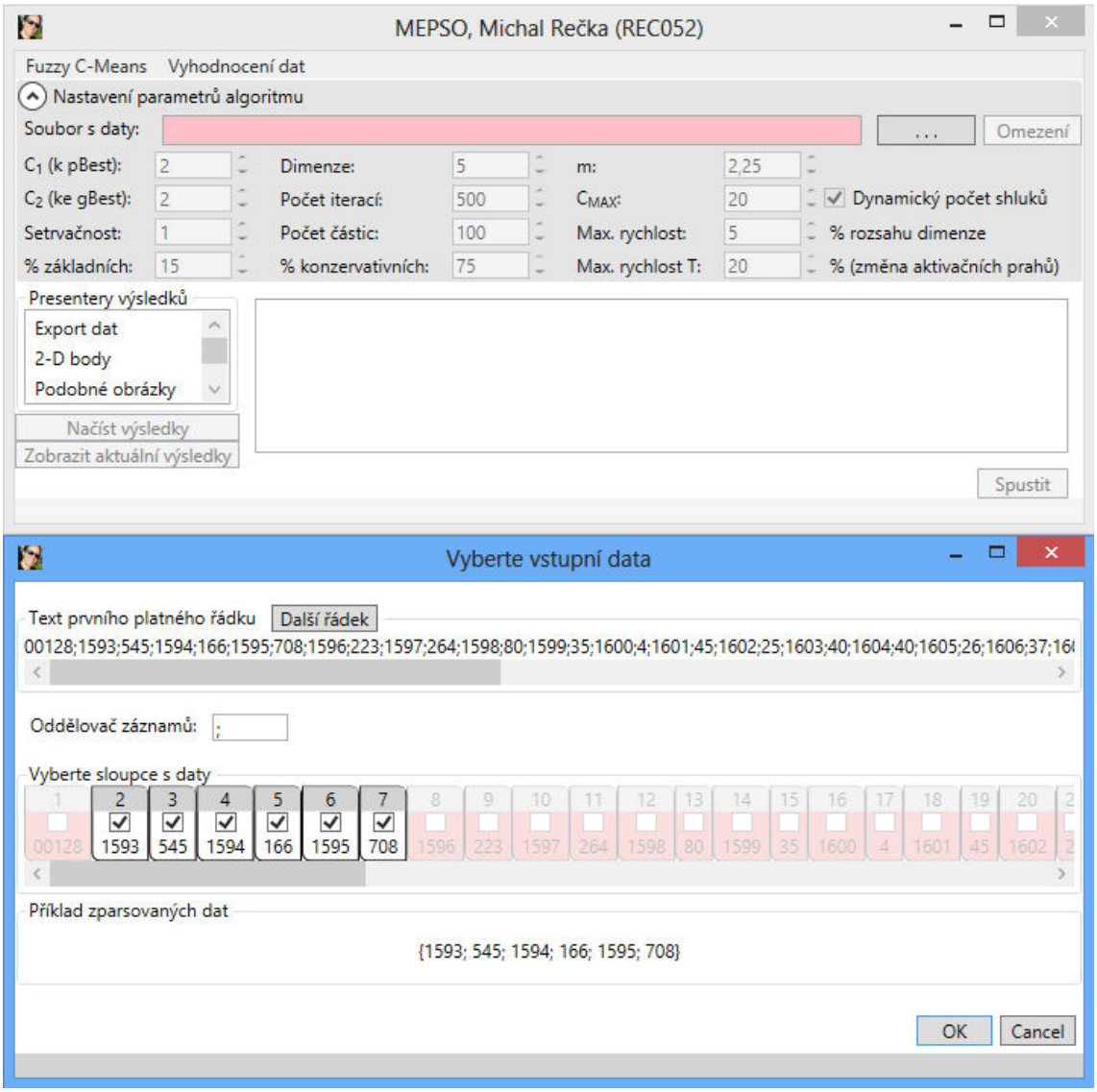
Výběr vstupních dat pro algoritmus FCM se skládá pouze z kroků 1, 2 a 3.

Výsledky algoritmů MEPSO* a FCM lze pak vyhodnotit nástrojem v horním menu. Ten vyžaduje jako vstup složku se soubory s příponou .bin a názvy souborů musí splňovat pravidlo, že výsledky algoritmu FCM musí začínat znakem _. Při dodržení této struktury pak lze získávat statistiky velmi rychle.

Snímek obrazovky hlavního programu je na obrázku 6 a zjednodušený třídní diagram algoritmu MEPSO* je na obrázku 5. Z důvodu lepší čitelnosti jsou v diagramu zachyceny pouze základní třídy algoritmu MEPSO* bez různých dědičností, rozhraní pro načítání dat a jiných vazeb, které by zde mohly být matoucí.

Struktura knihoven vypadá následovně:

- dll knihovny začínající *Rec052* jsou podpůrné knihovny se základní v .NET Frameworku často používanou funkcionalitou,
- *DataClustering.Core.dll* obsahuje základní třídy pro práci se shlukováním,
- *DataClustering.FCM.dll* je konkrétní implementace shlukování pomocí algoritmu Fuzzy c-means,



Obrázek 6: Snímek obrazovky programu pro shlukování dat pomocí algoritmu MEPSO* + dialog pro specifikaci dat

- *DataClustering.MEPSO.dll* je konkrétní implementace shlukování pomocí algoritmu MEPSO*,
- *DataClustering.MEPSO.GUI.WPF.exe* je knihovna s grafickými uživatelskými rozhraními,
- *DataClusteringLibrary.dll* obsahuje funkcionalitu exportu výsledků do textového souboru.

7.2 Výpočet TSP pomocí SBC a PSO

Jedná se o konzolové aplikace, které mají na vstupu (jako argument volání) název souboru s definicí úplného váženého orientovaného grafu a počet iterací pro jeden běh algoritmu. Následně je algoritmus spuštěn 100-krát za sebou a jsou vypsány výsledky. Nakonec umožní aplikace zapsat výsledky do textového souboru. Součástí je i generátor XML souborů s definicemi grafů. Na vstupu je povinný počet vrcholů a nepovinný název výstupního souboru.

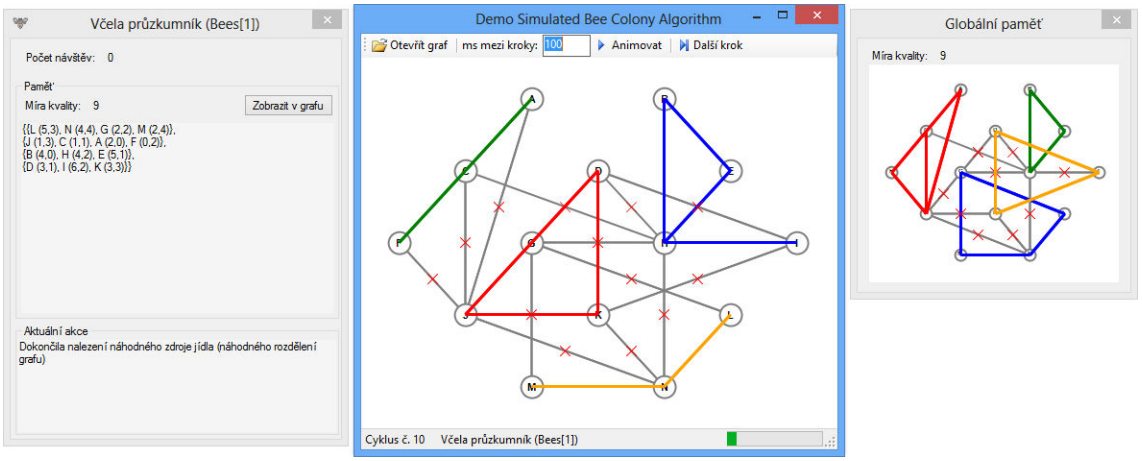
Knihovny této části jsou následující:

- *SBC.dll* obsahuje obecnou implementaci algoritmu simulace včelí kolonie,
- *SBC.Graph.dll* definuje data pro SBC jako úplný orientovaný vážený graf a obsahuje funkcionalitu s nimi spojenou,
- *SBC.Graph.Generator.exe* slouží k vygenerování validního XML souboru s náhodnými daty grafu,
- *SBC.TSP.dll* je konkrétní implementace algoritmu SBC pro účel řešení problému obchodního cestujícího,
- *SBC.TSP.Console.exe* slouží k provádění testů řešení TSP pomocí algoritmu SBC,
- *PSO.dll* obsahuje obecnou implementaci algoritmu PSO,
- *PSO.TSP.dll* je konkrétní implementace algoritmu PSO pro účel řešení problému obchodního cestujícího,
- *PSO.TSP.Console.exe* slouží k provádění testů řešení TSP pomocí algoritmu PSO.

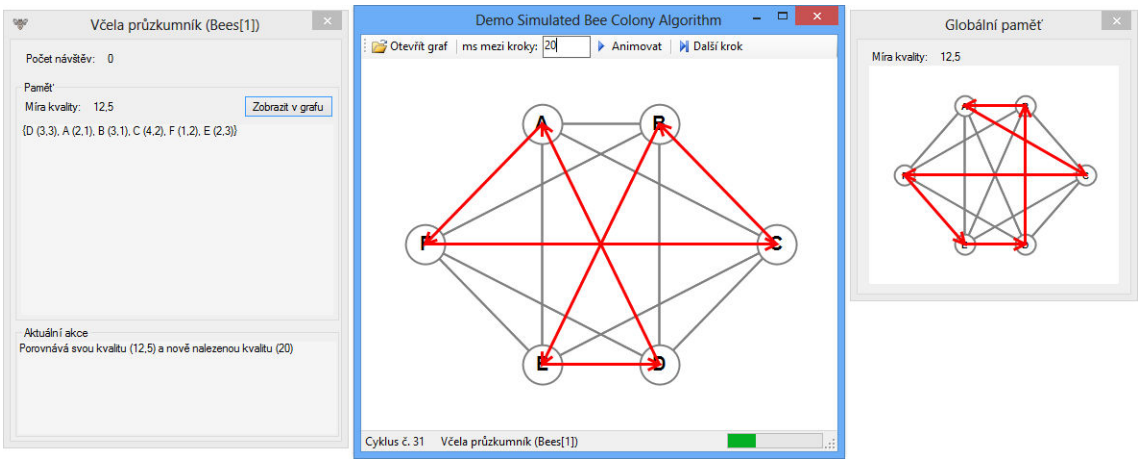
7.3 Vizualizace algoritmu SBC

Jedná se o podpůrnou aplikaci napsanou ve Windows Forms, která slouží k vizualizaci průběhu algoritmu simulace včelí kolonie. Po spuštění se zobrazí okno, ve kterém je třeba nejprve vybrat vstupní data. Jakmile jsou vybrána vstupní data, je zapotřebí vybrat typ problému. Na výběr jsou problémy GP[3] a TSP[4]. Po vyplnění parametrů se zobrazí dialog

[3]Problém GP není dokončen z důvodů popsaných v sekci 4.1. Sousedské řešení je generováno náhodně
[4]Pro problém obchodního cestujícího je nutno vybrat data s váženým grafem



Obrázek 7: Snímek obrazovky vizualizace problému rozdělení grafu pomocí SBC



Obrázek 8: Snímek obrazovky vizualizace problému obchodního cestujícího pomocí SBC

se znázorněním grafu, dialog zobrazující globálně nejlepší paměť a pokud je to aktuální, zobrazí se dialog reprezentující danou včelu, kde je vždy napsáno, co včela zrovna dělá a jaký má obsah paměti – viz obrázek 7 a 8.

Vizualizace SBC zahrnuje tyto knihovny:

- *SimulatedBeeColony.dll* obsahuje upravenou implementaci SBC pro řešení problémů TSP a GP,
- *SimulatedBeeVisualDemo.exe* zahrnuje upravenou implementaci úlu pro možnost vizualizace a obsahuje veškeré související GUI prvky.

XML soubor s definicí grafu musí mít strukturu popsanou ve výpise 5. V případě problému GP je možno vynechat atributy *Weight*.

8 Experimenty

Veškeré experimenty byly prováděny na školním serveru s procesorem s 8 jádry o frekvenci $2,7$ GHz a operační pamětí 12 GB. Toto testovací prostředí bylo navrženo především proto, abychom mohli otestovat efektivitu využití moderního hardware. S tím souvisely drobné úpravy testovaného algoritmu MEPSO*, aby využíval více vláken.

8.1 TSP

V sekci 2 jsme si popsali problém obchodního cestujícího. V této práci jsme vyzkoušeli k nalezení řešení TSP algoritmy SBC a PSO.

Algoritmus SBC byl naimplementován podle článku Johna McCaffreyho [6]. Na vstupu byl orientovaný ohodnocený úplný graf a řešením byla cesta (posloupnost vrcholů) např. A, F, B, E, C, D, A. Funkce Fitness byla obyčejný součet vah hran cesty a optimalizovali jsme minimum funkce Fitness (čím kratší celková vzdálenost, tím lepší řešení). Sousedská řešení byla generována pouze prohozením dvou náhodně vybraných vrcholů cesty.

U **algoritmu PSO** bylo zapotřebí specifikovat, jak bude reprezentována částice. Každé město má své číslo pořadí a částice je reprezentována polem reálných čísel $1 \leq x \leq n$, kde n je počet měst. V každé dimenzi může být libovolné číslo ve dříve zmíněném rozsahu. Při výpočtu funkce Fitness se pak tato čísla zaokrouhlí na celá čísla, čímž dostaneme pořadová čísla měst. Funkce Fitness pro SBC je tedy definována např. takto:

$$\begin{aligned} L &= \sum_{i=1}^{n} dist\left(m_i, m_{(i+1)\%n}\right) \\ F_{cost} &= \begin{cases} L & \text{pokud } d = 0, \\ (L+100)^{2+\frac{d}{100}} & \text{jinak} \end{cases} \end{aligned} \tag{22}$$

kde m_i je i-té město v posloupnosti měst, $\%n$ značí zbytek po celočíselném dělení počtem měst n, $dist(a, b)$ je délka cesty z města a do města b a d je počet duplicitních měst v posloupnosti. Je totiž nutné penalizovat řešení, kde se nevyskytují všechna města.

V tabulce 2 jsou znázorněny výsledky testů implementace algoritmů SBC a PSO na TSP. Počet iterací byl 1000, počet částic 100, maximální rychlost u PSO byla $20\,\%$ počtu měst, poměr aktivních, neaktivních a včel průzkumníků u SBC byl $75 : 15 : 10$. Každý algoritmus byl spuštěn desetkrát v dávkách po 100. Znázorněný čas tedy značí dobu, za jak dlouho byl daný algoritmus proveden 100-krát. N značí počet měst, F_{avg} je průměrná vhodnost, F_{best} je nejlepší nalezená vhodnost a F_{worst} je nejhorší nalezená vhodnost. Jak lze vidět, algoritmus SBC dává mnohem lepší výsledky. Pro větší N se algoritmus PSO v podstatě stává nepoužitelný. Je to dáno především principem algoritmu, kdy PSO funguje ideálně, pokud má funkce Fitness nějaký trend, což v případě TSP rozhodně nemá. SBC vyloženě trend funkce Fitness nepotřebuje. Zároveň to potvrzuje fakt, že PSO není univerzální algoritmus, který se hodí na řešení všech optimalizačních problémů a je třeba dobře zvážit, na které problémy jej použít.

N	Algoritmus	F_{best}	F_{avg}	F_{worst}	Čas [h:mm:ss]
6	SBC	197	**197,000**	**197**	0:00:13
	PSO	197	3266, 036	10000	0:00:46
15	SBC	**170**	**249,641**	**298**	0:00:39
	PSO	914	61924, 199	135424	0:01:57
30	SBC	**447**	**606,158**	**689**	0:02:10
	PSO	314721	602637, 844	853776	0:04:49

Tabulka 2: Porovnání výsledků problému obchodního cestujícího pomocí SBC a PSO. Oba algoritmy měly shodný počet ohodnocení funkce Fitness

8.2 Shlukování dat

Níže zmíněné testy byly prováděny algoritmem MEPSO* popsaným v sekci 6. Pokud není uvedeno jinak, platí následující konfigurace:

počet iterací:	500	základních částic:	15 %	c_1:	2
počet částic:	100	konzervativních částic:	75 %	c_2:	2
m:	2, 25	náhodných částic:	10 %	ω:	1
c_{max}:	20	max. rychlost středů:	5 %	max. rychlost prahů:	20 %

Testy byly prováděny s počtem ohodnocení funkce Fitness $F_{count} \leq 50000$. Jakmile počet ohodnocení funkce Fitness dosáhl F_{count} algoritmus dokončil iteraci a skončil, pokud však nebyly splněny ukončovací podmínky již dříve.

8.2.1 Body v rovině

Otestovali jsme účinnost dynamického shlukování námi navrženého algoritmu MEPSO* na bodech v rovině. Výsledky se velmi lišily v závislosti na nastavení vstupních parametrů – zejména poměr c_1, c_2 a ω, pak nastavení omezení rychlostí a stupně neostrosti m. Optimální nastavení vstupních parametrů velmi závisí na charakteru datového souboru a jeho určení by mohlo být dalším optimalizačním problémem. Výsledky shlukování byly srovnány s nejběžnějším algoritmem na shlukování dat – Fuzzy c-means – a ukazují, že pokud jsou dobře zvoleny vstupní parametry, může algoritmus MEPSO* shlukovat skutečně lépe. Tabulka 3 dané výsledky zachycuje. K porovnání výsledků byla použita funkce Fitness zmíněná v sekci 6.4. N_{exp} je očekávaný/skutečný počet shluků. Minimální, průměrný a maximální počet nalezených shluků je v tabulce značen N_{min}, N_{avg} resp. N_{max}. Testy byly prováděny cyklicky. Vstupní parametry algoritmu MEPSO* byly mírně měněny, zatímco u algoritmu FCM byly zadány stejné hodnoty počtu cyklů, stupně neostrosti m a byl dán skutečný počet očekávaných shluků. Z tabulky lze vyčíst, že MEPSO* podle výsledné vhodnosti řešení předčí algoritmus FCM.

Dalším testem bylo porovnání statického shlukování algoritmem MEPSO* a FCM. Cílem bylo otestovat, jak přesně jsou schopny algoritmy určit jednotlivé shluky, pokud znají skutečný počet shluků. I v tomto případě se ukázalo, že MEPSO* dává lepší výsledky.

N_{exp}	Alg.	N_{min}	N_{avg}	N_{max}	F_{best}	F_{avg}	F_{worst}
5	MEPSO*	3	$5,1875$	9	**0,02017**	**0,0375**	**0,0461**
	FCM	–	–	–	$1,9.10^{-6}$	$0,0327$	$0,0363$
6	MEPSO*	3	$4,7$	8	**0,0052**	**0,0107**	**0,0129**
	FCM	–	–	–	$2,7.10^{-6}$	$0,0056$	$0,0104$
7	MEPSO*	2	$5,1$	8	**0,0046**	**0,0071**	**0,0084**
	FCM	–	–	–	$1,3.10^{-7}$	$0,0029$	$0,0052$
15	MEPSO*	3	$6,1$	11	**0,0014**	**0,0024**	**0,0029**
	FCM	–	–	–	$2,6.10^{-6}$	$1,9.10^{-5}$	$0,0001$

Tabulka 3: Porovnání výsledků dynamického shlukování algoritmem MEPSO* s výsledky statického shlukování algoritmem FCM

N_{exp}	n	Alg.	D_{min}	D_{avg}	D_{max}	F_{best}	F_{avg}	F_{worst}
7	788	MEPSO*	162	$263,4$	430	**0,0111**	**0,0138**	**0,0178**
		FCM	174	$303,6$	497	$2,3.10^{-6}$	$0,0022$	$0,0060$
15	600	MEPSO*	15	$98,7$	195	**0,0086**	**0,0129**	**0,0173**
		FCM	158	$181,6$	241	$1,5.10^{-6}$	$9,0.10^{-6}$	$,4.10^{-5}$

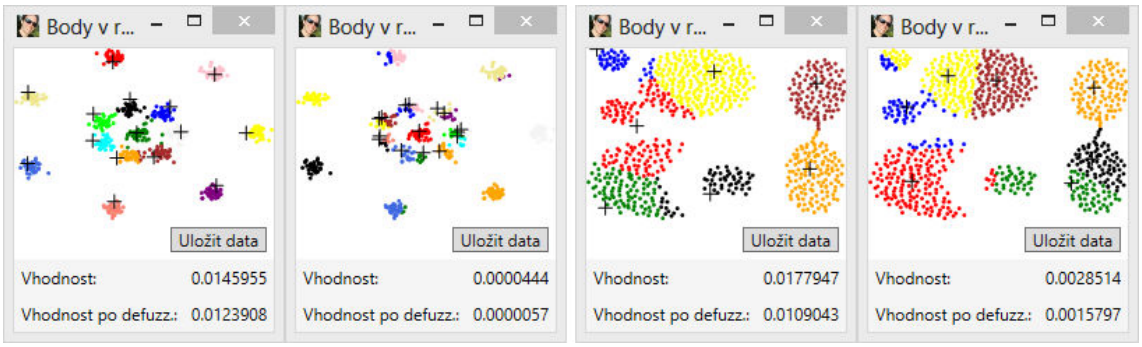
Tabulka 4: Porovnání výsledků statického shlukování datového souboru o N_{exp} definovaných shlucích algoritmem MEPSO* a FCM

Výsledky tohoto testu lze najít v tabulce 4. Díky tomu, že v tomto případě byly vzory opatřeny štítky, do kterých vzorů by měly patřit, bylo možné dopočítat hodnoty D_{min}, D_{avg} a D_{max}, které značí minimální, průměrný a maximální počet špatně klasifikovaných bodů. Příklad rozdílu statického shlukování MEPSO* a FCM je vidět na obrázku 9.

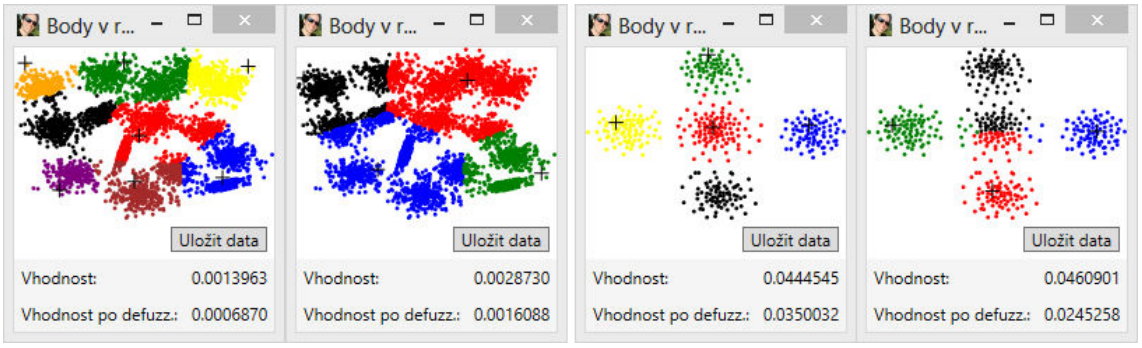
I když výsledky vypadají slibně je třeba uvést i na první pohled přehlédnutelné charakteristiky. MEPSO* má velký rozptyl a jeho výpočet trvá mnohonásobně déle. Zatímco FCM doběhl zpravidla do několika sekund, doba trvání algoritmu MEPSO* byla spíše otázkou minut až desítek minut. S tím souvisí i paměťová náročnost, která je přímo úměrná počtu částic. Pokud však porovnáváme výsledky na základě počtu ohodnocení funkce Fitness, dává MEPSO* skutečně lepší výsledky. Dále se ukázalo, že algoritmus má problémy s odhadem počtu shluků, pokud jsou velmi blízko u sebe. Často má tendenci shluky blízko sebe spojovat do jednoho většího shluku – viz obrázek 10. Nicméně toto chování je dáno nedokonalostí funkce Fitness a ne samotným algoritmem.

8.2.2 Podobné obrázky

Co se týče vícerozměrných shluků, provedli jsme test na shlukování obrázků. Data jsme čerpali z práce [8], kde byly vektory již přichystány. Ze všech obrázků jsme vybrali 8 skupin na první pohled podobných obrázků a byly na nich otestovány shlukovací algoritmy MEPSO* a FCM. Ani jednomu algoritmu se ani jednou nepodařilo nalézt očekávané rozdělení. Některé shluky byly po defuzzyfikaci nakonec prázdné, detailnější informace výsledků testů jsou zachyceny v tabulce 5. Opět se u algoritmu MEPSO* potvrdil velký



Obrázek 9: Porovnání statického shlukování pomocí MEPSO* (vždy vlevo) a FCM (vždy vpravo)



Obrázek 10: Rozptyl algoritmu MEPSO*. Na obrázku lze vidět (podle vhodnosti) tendence ke shlukování do větších celků

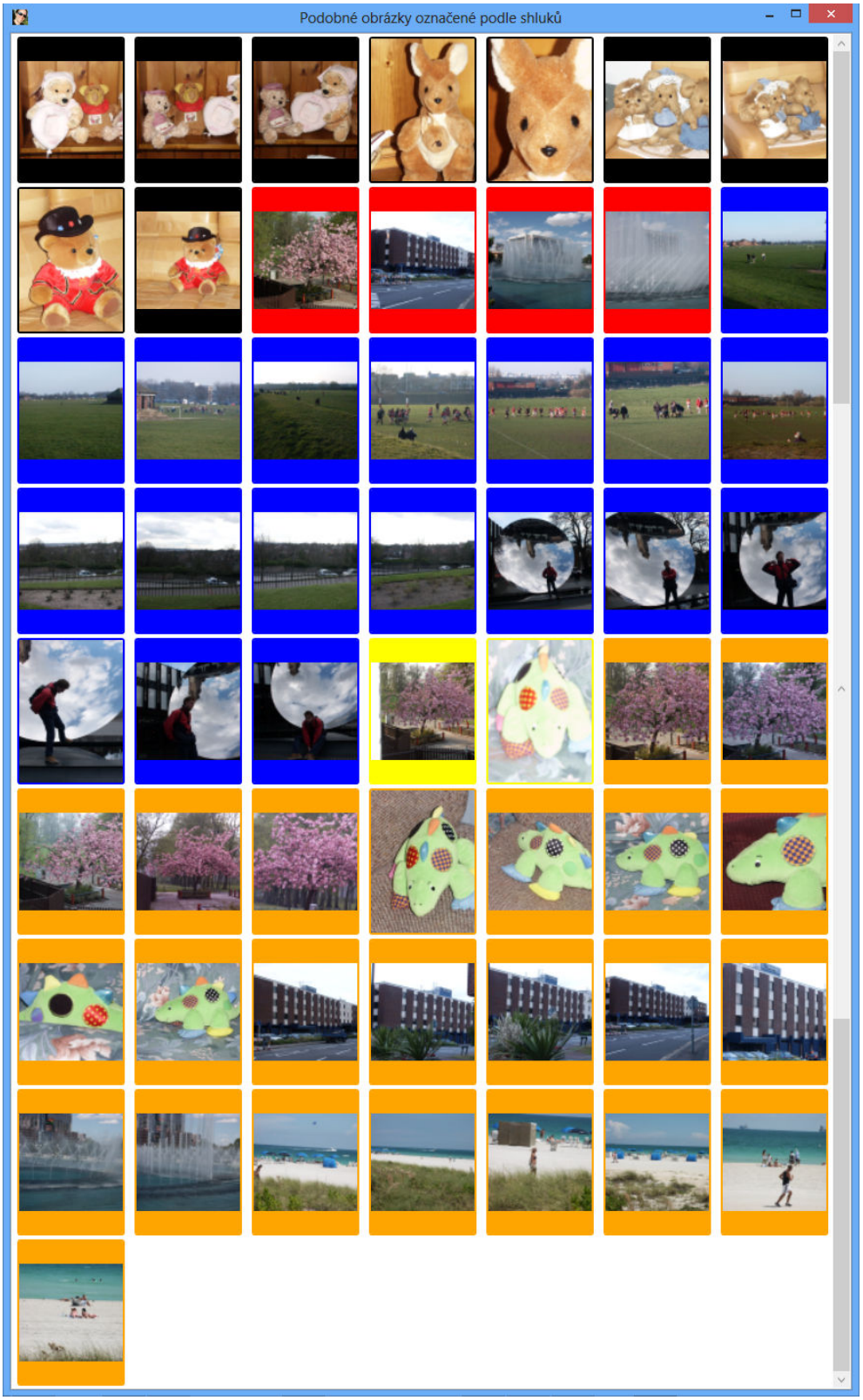
N_{exp}	Alg.	N_{min}	N_{avg}	N_{max}	F_{best}	F_{avg}	F_{worst}
8	MEPSO*	3	$5,7857$	10	**0,0827**	**0,107**	**0,1187**
	FCM	5	$7,3$	8	$2,3.10^{-6}$	$0,0098$	$0,0225$

Tabulka 5: Porovnání výsledků shlukování podobných obrázků pomocí algoritmu MEPSO* a FCM

rozptyl nalezených shluků a tendence spojovat menší shluky do větších. Když už obrázky byly v jednom shluku, byly si opravdu alespoň trochu podobné. Podle vhodností jednotlivých řešení jsou výsledky MEPSO* algoritmu opět lepší než u FCM. Příklad shlukování obrázků algoritmem MEPSO* je zachycen na obrázku 11.

8.3 Zhodnocení

Ukázali jsme si, že rozšířený algoritmus shlukování dat založený na PSO principu umí shlukovat lépe, než dnes asi nejpoužívanější shlukovací algoritmus – Fuzzy c-means – za předpokladu, že nalezneme optimální vstupní parametre. Algoritmus je na vstupních parametrech extrémně citlivý a protože je jich hodně, není vůbec jednoduché jejich optimální nastavení nalézt. Rovněž je PSO náročnější na paměť a především je mnohem náročnější časově. Už z principu PSO musí být částic přiměřeně hodně, aby se kolektivní inteligence měla možnost projevit. Algoritmus se dá sice paralelizovat, ale čím sofistikovanější je předávání informací mezi částicemi, tím větší nastává problém se synchronizací datových toků, je třeba používat zámky a pokud nebudete dost opatrní, nakonec to může vést až k dead lockům a úplnému znehodnocení algoritmu. Pokud je pro vás kvalita shlukování důležitější než doba exekuce, můžete se zaměřit na vyladění vstupních parametrů tohoto algoritmu tak, aby vám dával pro daný datový soubor co nejlepší výsledky. Pokud byste však chtěli shlukovací algoritmus, který by měl běžet, pokud možno, v reálném čase např. někde na webu, pak navrhovaný algoritmus není správná volba. Obzvláště pro velká data je nepoužitelný právě z důvodu většího počtu částic, kde každá musí držet informace ohledně dat a jejich shlukování. S využitím projekce do nižších dimenzí by se tento problém mohl kompenzovat. Definovaná funkce Fitness se ukázala být funkční, avšak nedokonalá, protože upřednostňuje menší počet prolínajících se shluků před větším počtem malých shluků.



Obrázek 11: Shlukování obrázků pomocí algoritmu MEPSO*

9 Závěr

V této práci jsme si představili základní principy evolučních výpočetních technik se zaměřením na simulaci včelího roje a optimalizaci rojení částic. Následně jsme si popsali, co je to shlukování dat a jaké základní techniky se používají. Navrhli jsme modifikaci neostrého shlukovacího algoritmu založeného na PSO a inspirovaného SBC. Tento algoritmus dokáže automaticky nalézt optimální počet shluků, který je však silně závislý na definici funkce Fitness. Z tohoto pohledu algoritmus minimalizuje nutnost analýzy dat uživatelem. Je ale zapotřebí nastavit vstupní parametry algoritmu a to už zase ve srovnání s jinými běžnými algoritmy vyžaduje práce mnohem více. Nicméně, pokud je vše správně nastaveno, umí algoritmus dávat dobré výsledky. Ve srovnání s algoritmem FCM dosahuje MEPSO* lepších výsledků shlukování dat, avšak za cenu delšího času a vyšší paměťové náročnosti.

Navzdory tomu, že je shlukování dat velmi starý problém, zůstává dodnes aktivně zkoumaným odvětvím. Není znám žádný jediný algoritmus, který dokáže seskupovat všechny datové soubory reálného světa efektivně a bez chyb. I samotné validační indexy jsou většinou tvořeny empíricky a není znám žádný univerzální index, který by fungoval stejně dobře pro všechny datové soubory. Evoluční výpočetní techniky obecně staví na vhodnosti jednotlivých řešení a proto je definice funkce Fitness naprosto klíčová. Proto by další výzkum měl směřovat i tímto směrem, aby např. nemělo shlukování tendenci spojovat menší shluky do větších celků.

Michal Rečka

10 Reference

[1] ABRAHAM, Ajith, Swagatam DAS a Sandip ROY. Swarm Intelligence Algorithms for Data Clustering. *Soft Computing for Knowledge Discovery and Data Mining* [online]. Boston, MA: Springer US, 2008, s. 279-313 [cit. 2013-04-25]. DOI: 10.1007/978-0-387-69935-6_12. Dostupné z: http://www.springerlink.com/index/10.1007/978-0-387-69935-6_12

[2] BEZDEK, James C., Robert EHRLICH a William FULL. FCM: The fuzzy c-means clustering algorithm. *Computers* [online]. 1984, roč. 10, 2-3, s. 191-203 [cit. 2013-04-25]. ISSN 00983004. DOI: 10.1016/0098-3004(84)90020-7. Dostupné z: http://linkinghub.elsevier.com/retrieve/pii/0098300484900207

[3] BRUCKER, P. On the Complexity of Clustering Problems. [online]. 1978, s. 45-54 [cit. 2013-04-25]. DOI: 10.1007/978-3-642-95322-4_5. Dostupné z: http://www.springerlink.com/index/10.1007/978-3-642-95322-4_5

[4] JAIN, A. K., M. N. MURTY a P. J. FLYNN. Data clustering: a review. *ACM Computing Surveys* [online]. 1999, roč. 31, č. 3, s. 264-323 [cit. 2013-04-25]. ISSN 03600300. DOI: 10.1145/331499.331504. Dostupné z: http://portal.acm.org/citation.cfm?doid=331499.331504

[5] JANČAR, Petr. *Teoretická informatika.* Ostrava: Vysoká škola báňská - Technická univerzita, 2007, 1 CD-R. ISBN 978-80-248-1487-2.

[6] MCCAFFREY, James. Use Bee Colony Algorithms to Solve Impossible Problems. MSDN Magazine [online]. 2011 [cit. 2013-04-04]. Dostupné z: http://msdn.microsoft.com/cs-cz/magazine/gg983491%28en-us%29.aspx

[7] MCCAFFREY, James D. Graph partitioning using a Simulated Bee Colony algorithm. *2011 IEEE International Conference on Information Reuse* [online]. IEEE, 2011, s. 400-405 [cit. 2013-04-28]. DOI: 10.1109/IRI.2011.6009581. Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6009581

[8] ŘÍMAN, Jakub. *Vyhledávání v kolekci obrázků pomocí komprese.* Ostrava, 2012. Diplomová práce. Vysoká škola báňská - Technická univerzita Ostrava, FEI. Vedoucí práce Ing. Jan Martinovič, Ph.D.

[9] SOPER, A. J., C. WALSHAW a M. CROSS. A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph-Partitioning. *Journal of Global Optimization* [online]. 2004, roč. 29, č. 2, s. 225-241 [cit. 2013-04-28]. ISSN 0925-5001. DOI: 10.1023/B:JOGO.0000042115.44455.f3. Dostupné z: http://link.springer.com/10.1023/B:JOGO.0000042115.44455.f3

[10] YUANGANG, Tang, Sun FUCHUN a Sun ZENGQI. Improved validation index for fuzzy clustering. *Proceedings of the 2005, American Control Conference, 2005* [online]. IEEE, 2005, s. 1120-1125 [cit. 2013-04-25]. DOI: 10.1109/ACC.2005.1470111. Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1470111

[11] ZELINKA, Ivan, Zuzana OPLATKOVÁ, Miloš ŠEDA, Pavel OŠMERA a František VČELAŘ. *Evoluční výpočetní techniky: principy a aplikace.* 1. české vyd. Praha: BEN, 2009, 534 s. ISBN 978-80-7300-218-3.

A Vzor vstupních dat

```
1  <?xml version="1.0" encoding="UTF−8"?>
2  <Vertices>
3    <Vertex Name="A" X="2" Y="1">
4      <Neighbor Name="B" Weight="1" />
5      <Neighbor Name="C" Weight="2" />
6      <Neighbor Name="D" Weight="3" />
7      <Neighbor Name="E" Weight="4" />
8      <Neighbor Name="F" Weight="5" />
9    </Vertex>
10   <Vertex Name="B" X="3" Y="1">
11     <Neighbor Name="A" Weight="1,5" />
12     <Neighbor Name="C" Weight="1" />
13     <Neighbor Name="D" Weight="2" />
14     <Neighbor Name="E" Weight="3" />
15     <Neighbor Name="F" Weight="4" />
16   </Vertex>
17   <Vertex Name="C" X="4" Y="2">
18     <Neighbor Name="A" Weight="3" />
19     <Neighbor Name="B" Weight="1,5" />
20     <Neighbor Name="D" Weight="1" />
21     <Neighbor Name="E" Weight="2" />
22     <Neighbor Name="F" Weight="3" />
23   </Vertex>
24   <Vertex Name="D" X="3" Y="3">
25     <Neighbor Name="A" Weight="4,5" />
26     <Neighbor Name="B" Weight="3" />
27     <Neighbor Name="C" Weight="1,5" />
28     <Neighbor Name="E" Weight="1" />
29     <Neighbor Name="F" Weight="2" />
30   </Vertex>
31   <Vertex Name="E" X="2" Y="3">
32     <Neighbor Name="A" Weight="6" />
33     <Neighbor Name="B" Weight="4,5" />
34     <Neighbor Name="C" Weight="3" />
35     <Neighbor Name="D" Weight="1,5" />
36     <Neighbor Name="F" Weight="1" />
37   </Vertex>
38   <Vertex Name="F" X="1" Y="2">
39     <Neighbor Name="A" Weight="7,5" />
40     <Neighbor Name="B" Weight="6" />
41     <Neighbor Name="C" Weight="4,5" />
42     <Neighbor Name="D" Weight="3" />
43     <Neighbor Name="E" Weight="1,5" />
44   </Vertex>
45 </Vertices>
```

Výpis 5: Struktura vstupních dat pro vizualizaci TSP pomocí SBC