VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

KNIHOVNA PRO OPTIMALIZAČNÍ ÚLOHY VYUŽÍVAJÍCÍ TECHNIKY PSO

PARTICLE SWARM OPTIMIZATION LIBRARY

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MILAN HRUBAN**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. MICHAL BIDLO, Ph.D.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2018/2019

Zadání bakalářské práce

Student:	**Hruban Milan**
Program:	Informační technologie
Název:	**Knihovna pro optimalizační úlohy využívající techniky PSO** **Particle Swarm Optimization Library**
Kategorie:	Umělá inteligence

Zadání:

1. Seznamte se s problematikou optimalizačních technik založených na konceptu Particle Swarm Optimization (PSO).
2. Proveďte rešerši dostupných nástrojů a knihoven pro PSO a zpracujte studii na toto téma.
3. Pro zvolené varianty PSO navrhněte knihovnu umožňující využití těchto technik pro řešení různých optimalizačních úloh. Tuto knihovnu implementujte.
4. Na vhodných úlohách demonstrujte použití knihovny a ověřte činnost implementovaných algoritmů PSO.
5. Zhodnoťte dosažené výsledky a diskutujte možnosti pokračování projektu.

Literatura:

- Podle pokynů vedoucího projektu.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce:	**Bidlo Michal, Ing., Ph.D.**
Vedoucí ústavu:	Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání:	1. listopadu 2018
Datum odevzdání:	15. května 2019
Datum schválení:	26. října 2018

Abstrakt

Cílem této práce je vývoj knihovny umožňující řešení optimalizačních úloh pomocí technik PSO. Práce představuje knihovnu implementovanou za tímto účelem v jazyce Kotlin, navrženou s ohledem na rozšiřitelnost a použitelnost. Součástí práce je nástroj implementovaný v Pythonu za pomocí technologie Jupyter Notebook, který umožňuje statistické zpracování experimentů provedených pomocí dané knihovny. V práci je tak vytvořen systém, který poskytuje vhodné prostředí pro pokusy se současnými a vývoj nových variací algoritmu PSO.

Abstract

The aim of this thesis is to develop a library that is able to solve optimization tasks using PSO. The library is implemented using Kotlin and is designed to achieve high extensibility and usability. Moreover, a tool for processing and statistical analysis of experiments performed using the library is implemented by means of the Jupyter Notebook environment. The utilization of these tools creates a setup suitable for experimenting with the current and developing new variations of the PSO algorithm.

Klíčová slova

optimalizace pomocí částic, inteligence hejna, umělá inteligence, kotlin, python, jupyter notebook, knihovna

Keywords

particle swarm optimization, swarm intelligence, artificial intelligence, kotlin, python, jupyter notebook, library

Citace

HRUBAN, Milan. *Knihovna pro optimalizační úlohy využívající techniky PSO.* Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Michal Bidlo, Ph.D.

Knihovna pro optimalizační úlohy využívající techniky PSO

Prohlášení

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

.....
Milan Hruban
14. května 2019

Poděkování

Chtěl bych poděkovat svému vedoucímu bakalářské práce Ing. Michalu Bidlovi, Ph.D. za odborné vedení, za pomoc a rady při zpracování této práce.

Obsah

Kapitola 1

Úvod

Optimalizace je snaha o nalezení vhodných hodnot množiny parametrů, která reprezentuje správné nebo nejlepší řešení daného problému. Jednou z možností, jak tyto problémy řešit, jsou algoritmy inspirované přírodou. Jedná se o třídu stochastických algoritmů, jejichž princip spočívá v napodobení nějakého přírodního jevu, např. evoluce (evoluční algoritmy) nebo chování zvířecího společenstva (mravenců, včel, ryb apod.).
Optimalizace pomocí hejna částic (z anglického Particle Swarm Optimization, dále jen PSO) je algoritmus inspirovaný chováním hejna ptáků. Mezi typické příklady využití patří plánování tras [25] [19], plánování časových rozvrhů [21] nebo návrh dimenzí podpěrných trámů [16]. Algoritmus lze aplikovat i na problém barvení grafů, pomocí kterého můžeme modelovat například alokaci CPU registrů, přiřazování rádiových frekvencí nebo problém barvení map. Další možnou aplikací je problém obchodního cestujícího, který lze uplatnit např. k optimalizaci dráhy nástroje při vrtání desek plošných spojů.
Cílem této práce je implementovat knihovnu umožňující využití PSO v různých variacích na různé optimalizační problémy. Knihovna umožňuje uživateli snadnou konfiguraci algoritmu i rozšíření o vlastní obměny. V následující kapitole jsou popsány současně dostupné nástroje zabývající se touto problematikou. Kapitola 3 vysvětluje princip fungování PSO a důležité komponenty algoritmu. Kapitoly 4 a 5 se zabývají návrhem knihovny a popisují rozhraní pro rozšíření knihovny a detaily implementace. V poslední kapitole jsou shrnuty provedené experimenty doplněné o statistické zpracování výsledků.

Kapitola 2

Přehled současných nástrojů

Tato kapitola poskytuje obraz o volně dostupných knihovnách, které se zabývají optimalizací pomocí hejna částic. Vzhledem k implementačnímu jazyku práce, popsaném v sekci 5.1, je tato sekce zaměřena především na knihovny dostupné v ekosystému Javy (Java[1], Kotlin[2], Scala[3]). Volně dostupných knihoven s řádnou dokumentací není mnoho. Do následujícího přehledu byly zahrnuty pouze ty, které obsahují alespoň stručný uživatelský manuál a jsou do jisté míry rozšiřitelné a použitelné na různé optimalizační problémy.

2.1 Ekosystém Javy

JSwarm-PSO

Tato knihovna [4] je navržena pouze pro práci s reálnými čísly. Z uživatelského hlediska poskytuje velmi limitované možnosti konfigurace. Lze nastavit základní parametry algoritmu a definovat vlastní optimalizační problém. Knihovna nabízí podporu pro N dimenzionální problémy. K dispozici je také základní zobrazení statistik, které ovšem nefunguje bez předchozí konfigurace a není integrováno napříč celým algoritmem. Bez dalšího rozšíření tedy není možné logy běhů algoritmu jednoduše statisticky zpracovávat a vyhodnocovat. Součástí knihovny je i jednoduché grafické uživatelské rozhraní, které nabízí možnost vizualizace částic v prohledávaném prostoru v průběhu algoritmu. Dokumentace ke knihovně je poměrně stručná a popisuje pouze základní příklad užití a význam jednotlivých konfigurovatelných parametrů.

Z pohledu architektonického návrhu je hierarchie tříd velice dobře navržena a představuje kvalitní základ připravený na rozšíření. Stejně jako pro uživatelskou část však chybí dokumentace a implementace variací algoritmu je tedy problematická. Poslední verze knihovny byla vydána v roce 2009 a od té doby už další vývoj neprobíhá. Implementačním jazykem je Java.

MOEA Framework

MOEA Framework [5] je komplexní knihovna umožňující experimenty s algoritmy multikriteriální optimalizace. Krom PSO nabízí celou řadu dalších evolučních algoritmů. Součástí

[1] https://www.java.com/en/
[2] https://kotlinlang.org/
[3] https://www.scala-lang.org/
[4] http://jswarm-pso.sourceforge.net/
[5] http://moeaframework.org/

knihovny je grafické uživatelské rozhraní, které slouží pro výběr a konfiguraci algoritmu a pro vizualizaci výsledků. Vzhledem k širokému zaměření knihovny zahrnuje pouze dvě variace PSO. Má však kvalitní dokumentaci a dobře navržené aplikační rozhraní a umožňuje tedy snadnou implementaci dalších algoritmů. Implementačním jazykem je Java.

2.2 Python

Pyswarm

Pyswarm [6] je jednoduchá knihovna, určená pro optimalizaci N dimenzionálních problémů ve stavovém prostoru s reálnými čísly. Její hlavní výhodou je jednoduchost použití. Knihovna nabízí pouze základní verzi algoritmu a spuštění experimentu tedy vyžaduje minimální konfiguraci. Uživatel však nemá možnost měnit chování algoritmu více než pomocí základních parametrů, z čehož vyplývá, že hlavní využití bude jako demonstrační nástroj. K tomuto účelu by bylo vhodné využití grafického uživatelského rozhraní, které knihovna postrádá. Pyswarm nenabízí možnost logování statistik ani zhodnocení výsledků běhů algoritmu.

Zajímavou funkcí je možnost paralelizace vyhodnocení optimalizované funkce a urychlení výpočtu. Knihovna není určena pro rozšíření a implementace nových funkcí a variací algoritmu by znamenala výrazné zásahy do současného kódu. Implementačním jazykem je Python.

Pyswarms

Knihovna Pyswarms [7] je sada nástrojů vytvořená pro snadné experimentování s různými variantami PSO. Umožňuje uživateli širokou škálu konfiguračních možností a množství variací algoritmu. Jako jedna z mála nabízí i možnost optimalizace diskrétních problémů, konkrétně pomocí implementace binárního PSO. Knihovna zahrnuje nástroj na statistické zpracování výsledků experimentů, dokáže z běhu algoritmu vygenerovat grafy a vizualizovat průběh optimalizace. Dalším z nástrojů je optimalizátor hyper parametrů, který dokáže pomocí zvolené strategie vybrat pro konkrétní optimalizační problém vhodné parametry algoritmu. Pyswarms má kvalitně navržené aplikační rozhraní, které umožňuje snadno přidávat a upravovat části algoritmu. Díky obsáhlé dokumentaci, která ukazuje několik příkladů využití a návod na rozšíření knihovny o vlastní implementaci, a výše zmíněným funkcím je knihovna Pyswarms vhodným nástrojem pro výzkum a experimentování s PSO. Implementačním jazykem je Python.

2.3 C/C++

V jazycích C a C++ existuje celá řada nástrojů, které implementují algoritmus PSO. Většina z těchto programů není navržena jako knihovna ale spíš jako demonstrační aplikace a soustředí se na několik málo variací algoritmu. Jednou ze zajímavých je například knihovna *pso* napsaná v jazyce C [8] fungující jako modul, který může být použitý pro optimalizaci spojitých funkcí. Tato knihovna však neposkytuje rozhraní pro organizované logování statistik a jejich vyhodnocení. Další podobnou knihovnou je PSOLib [9], napsaná v jazyce Objective-

[6] https://pythonhosted.org/pyswarm/
[7] https://pyswarms.readthedocs.io/en/latest/
[8] https://github.com/kkentzo/pso
[9] https://github.com/IvanRublev/PSOLib

C, zaměřená na platformy iOS a OSX. Stejně jako předchozí knihovny nenabízí nástroj na vyhodnocení statistik. Její předností je dobře navržené aplikační rozhraní a snadná možnost rozšíření.

Kapitola 3

Optimalizace pomocí hejna částic

Optimalizace pomocí hejna částic je algoritmus kopírující chování skupin, které nemají žádného vůdce. Tato společenstva nemají žádné celistvé povědomí o prostředí, ve kterém se nachází. Veškeré informace získávají pouze z pozorování omezené části svého okolí a z interakce s ostatními jedinci ve společenstvu. Díky tomu jsou pak schopna hromadně se pohybovat a dosahovat společných cílů - nejčastěji lov, přesun a hledání potravy.
PSO využívá těchto konceptů k nalezení globálního extrému (případně více různých, stejně dobrých řešení) optimalizované funkce. Motivací k využití PSO je přílišná časová náročnost jiných algoritmů, které jsou tudíž na složité problémy nepoužitelné. Původní forma algoritmu publikovaná v roce 1995 [12] je popsaná v následující kapitole.

3.1 Základní model PSO

Model se skládá z hejna částic, které nazýváme populací. Každá částice v rámci populace reprezentuje potenciální řešení pomocí svých N souřadnic v N-dimenzionálním prostoru. Částice se pohybují v prostoru na základě informace o své předchozí nejlepší pozici a informace o pozici aktuálně nejlepší částice ze svého okolí. Velikost okolí, z kterého získávají informace může zahrnovat celou populaci, nebo pouze její část, potom mluvíme o tzv. gbest PSO a lbest PSO (globální a lokální PSO). Kvalita aktuální pozice se hodnotí pomocí fitness funkce, jejímž vstupem je vektor souřadnic částice a výstupem reálné číslo - hodnota optimalizované funkce v daném bodě. Označíme-li $\vec{x}$ jako pozici částice, $\vec{v}$ jako její rychlost a t jako čas, který reprezentuje jednotlivé iterace algoritmu (jedná se tedy o diskrétní veličinu), potom pro každou iteraci platí následující:

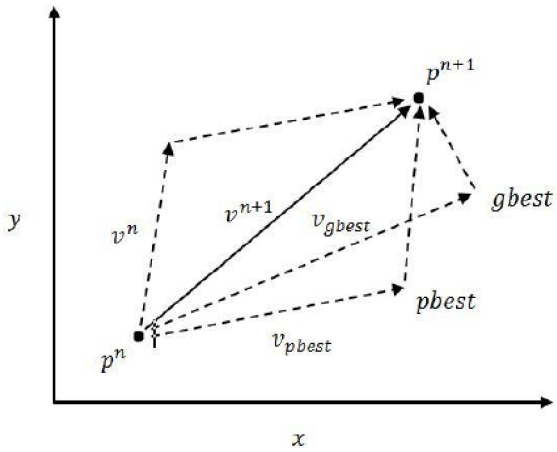
$$x_i(t+1) = x_i(t) + v_i(t+1) \quad pro \quad i \in 1..N \tag{3.1}$$

Rychlost částice $\vec{v}$ je, stejně jako její pozice, reprezentována pomocí N-dimenzionálního vektoru a je aktualizována každou iteraci dle rovnice 3.2.

$$v_i(t+1) = v_i(t)\cdot w + r_1\cdot c_1\cdot(gbest_i(t) - x_i(t)) + r_2\cdot c_2\cdot(pbest_i(t) - x_i(t)) \quad pro \quad i \in 1..N \tag{3.2}$$

Vektor *gbest* obsahuje souřadnice nejlepší částice v okolí, *pbest* představuje dosud nejlepší pozici dané částice, koeficienty r_1 a r_2 jsou náhodná čísla z rovnoměrného rozložení $(0, 1)$, generovaná pro každou souřadnici částice zvlášť, která do algoritmu přináší stochasticitu a zaručují jeho nedeterministický průběh. Koeficienty c_1 a c_2 jsou uživatelem nastavené parametry určené k ovládání kognitivního a sociálního komponentu algoritmu, tedy zda částice budou mít tendenci směrovat spíš k vlastní doposud nejlepší pozici, nebo spíš k nejlepší

pozici v rámci celého hejna. Vektor rychlosti z předchozího kroku iterace, který se k hodnotě v aktuálním kroku přičítá, násobíme konstantou w, jejíž hodnota je zpravidla menší než 1. Tím docílíme efektu setrvačnosti. Nový vektor rychlosti částice je tedy složen ze 3 vektorů - vektor směřující směrem ke globálně nejlepší částici, vektor směřující k nejlepší předchozí pozici dané částice a vektor zachovávající směr částice v předchozí iteraci. Ilustrace změny pozice částice je zobrazena na obrázku 3.1.



Obrázek 3.1: Výpočet vektoru rychlosti a aktualizace pozice částice [14] (Znázorněno jako příklad optimalizace ve dvourozměrném stavovém prostoru.)

3.2 Průběh algoritmu

Základní verze algoritmu [12] se řídí tímto postupem.

1. Inicializace uživatelem nastavených parametrů w, c_1, c_2, maximálního počtu iterací, velikosti populace a volba optimalizované funkce.

2. Inicializace populace a nastavení náhodných vektorů pozic a rychlostí pro každou částici.

3. Pro každou částici v populaci: Vypočítej hodnotu fitness funkce. Pokud je lepší než hodnota v *pbest*, aktualizuj souřadnice *pbest* na aktuální pozici. Pokud je fitness lepší než hodnota v *gbest*, nastav souřadnice *gbest* na aktuální pozici této částice.

4. Pokud je splněna ukončovací podmínka (podrobně popsána v kapitole 3.5), ukonči simulaci. Jinak pokračuj.

5. Pro každou částici v populaci: Vypočítej novou rychlost podle vztahu 3.2. Aktualizuj pozici částice podle vztahu 3.1

6. Pokud číslo aktuální iterace je menší než maximální počet iterací, vrať se na krok 3. Jinak ukonči simulaci.

Následující sekce popisují aspekty, které byly zkoumány a prokázaly vliv na chování algoritmu.

3.3 Inicializace populace

Na začátku běhu algoritmu je třeba vytvořit hejno částic a každé částici přiřadit nějakou pozici a rychlost. Rychlost se zpravidla generuje náhodně (s omezeními popsanými níže v sekci 3.4) nebo se nastaví na 0 - pro každou částici dojde prvně k výpočtu nové rychlosti podle rovnice 3.2 a teprve poté k aktualizace pozice podle rovnice 3.1, takže nehrozí, že by částice zůstaly na místě. Žádný z těchto přístupů nenabízí znatelnou výhodu. Experimenty prokázaly [4], že některé pokročilé strategie generování počáteční rychlosti v kombinaci s konkrétní topologií hejna dosahují na testovacích funkcích lepších výsledků než tradiční přístup náhodného generování. Volba počátečních pozic částic hraje výrazně větší roli v kvalitě algoritmu než výběr počátečních rychlostí. Hlavním cílem při tvorbě heuristiky pro volbu počátečních pozic je co nejrovnoměrněji pokrýt prohledávaný stavový prostor. Ačkoliv generování náhodných pozic poskytuje relativně kvalitní pokrytí, pomocí některých metod [4][18] lze dosáhnout startovní konfigurace, která zejména v problémech s velkým počtem dimenzí předčí náhodnou startovní konfiguraci. Zmíněné metody na druhou stranu také přináší zvýšenou výpočetní náročnost při generování populace.

3.4 Omezení rychlosti

Rychlost částice se v každé iteraci algoritmu standardně vypočítá podle rovnice 3.2. Kromě pozice částice, nejlepší pozice částice a pozice nejlepší částice v populaci na ni mají vliv i konfigurovatelné parametry c_1, c_2 a w. Pomocí těchto parametrů lze tedy regulovat rychlost do požadovaných mezí, ale ne vždy je to dostačující. Například může nastat situace, kdy částice má příliš vysokou rychlost, získanou kvůli její vysoké vzdálenosti od `gbest`, a `gbest` "přeletí", tím pádem se ocitne opět ve velké vzdálenosti. Tato interakce sice přispívá k prozkoumávání větší části stavového prostoru, ale pokud se v ní ocitne značná část populace, hrozí, že se všechny částice budou pohybovat vysokou rychlostí a ke konvergenci a nalezení optima nikdy nedojde. Abychom vyrovnali poměr průzkumu nových oblastí a detailního prohledávání okolí již nalezených dobrých řešení (`exploration/exploitation`), často zavádíme omezení na maximální rychlost částice, které může dosáhnout. Vhodná hranice je předmětem k diskuzi a v jednotlivých implementacích se liší. Označíme-li maximální rychlost jako v_{max}, rychlost v (zde v a v_{max} reprezentují velikost rychlosti jako skalární veličinu, ne vektor rychlosti) potom upravíme podle vztahu:

$$\begin{aligned} v &= v_{max} \quad pro \quad v \geq v_{max} \\ v &= -v_{max} \quad pro \quad v \leq -v_{max} \\ v &= v \quad jinak \end{aligned} \tag{3.3}$$

3.5 Ukončovací podmínky

Důležitou součástí algoritmu je volba vhodné ukončovací podmínky. Při jejím špatném nastavení může dojít k zbytečným iteracím algoritmu a vyhodnocování fitness funkce a nebo naopak k předčasnému ukončení simulace, což může způsobit zvýšenou výpočetní náročnost nebo získání nepřesného řešení. Ve většině implementací algoritmu byla použita jedna z následujících ukončovacích podmínek.

- **Ukončení na základě počtu provedených iterací**
  Tato podmínka se většinou používá jako doplněk k nějaké další ukončovací podmínce.

Hlavní motivací pro její zavedení je ukončení běhu algoritmu v případech, kdy se populaci nezdaří v rozumném počtu iterací naplnit hlavní ukončovací podmínku, např. konvergovat k nějakému bodu nebo najít dostatečně dobré řešení. Nastavením maximálního počtu iterací tak zabráníme teoreticky nekonečnému cyklu. Další použití je v případech, kdy chceme běh algoritmu časově limitovat a zajímá nás nejlepší nalezené řešení v omezeném časovém úseku.

- **Ukončení, pokud se populace přestane dostatečně zlepšovat**
  Tato podmínka může být implementována několika způsoby. Pokud algoritmus nemá nějakou mechaniku, kterou by podporoval pohyb částic a prozkoumávání prostoru i po nalezení lokálního optima, celá populace ve většině případů konverguje k nějakému řešení kdy všechny částice jsou téměř ve stejném bodě a už neexistuje šance, že bude nalezeno výrazně lepší řešení. Tento stav můžeme detekovat například měřením průměrné vzdálenosti mezi částicemi, měřením průměrné rychlosti populace nebo měřením vzdáleností mezi pozicemi částice v jednotlivých iteracích. Další možností je uchovávání si informace o kvalitě globálně nejlepšího řešení v jednotlivých iteracích a pokud se tato hodnota po určitou dobu nezmění, lze běh algoritmu ukončit. Tato metoda však nezaručuje, že došlo ke konvergenci populace. Pokud je parametr určující počet iterací, jaký musí proběhnout bez zlepšení, vhodně zvolen, lze jí detekovat běhy algoritmu, které budou pravděpodobně neúspěšné a včas je ukončit.

- **Ukončení při nalezení dostatečně dobrého řešení**
  U některých specifických problémů řešených pomocí PSO můžeme předem znát hodnotu optima (např. problém barvení grafů, více v kapitole 3.8.1), případně hranici hodnoty fitness funkce, kterou musíme dosáhnout. V takovém případě je pak vhodné simulaci ukončit hned, jakmile je nalezeno řešení vyhovující těmto požadavkům.

3.6 Stavový prostor

PSO lze aplikovat na problémy s ohraničeným i neohraničeným [23] [22] stavovým prostorem. Při optimalizaci v neohraničeném stavovém prostoru není potřeba zavádět žádné speciální podmínky. Pokud však máme nastaveny hranice, za kterými se nesmí nacházet nalezené řešení, je potřeba definovat chování částic tak, aby tato podmínka nebyla porušena. K tomu se používá několik metod podrobně popsaných v literatuře [5].

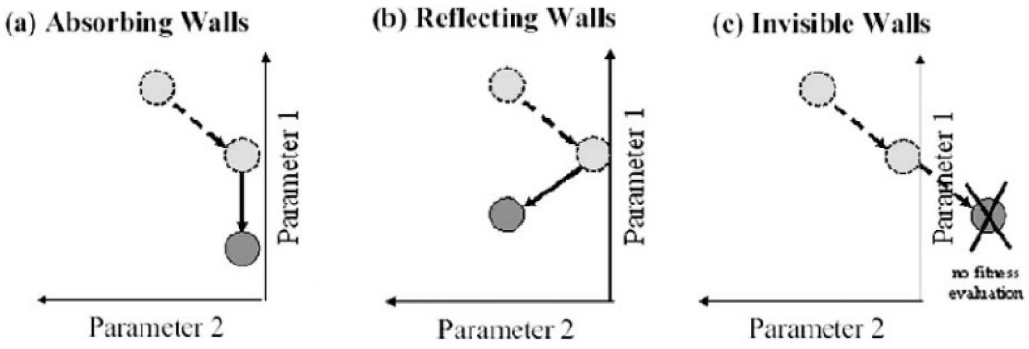
- **Odmítnutí částic mimo prostor**
  Nejjednodušším řešením je vyřazení částic, které se dostanou mimo stavový prostor, z populace a jejich nahrazení náhodně generovanými jedinci. Tato strategie je vhodná, zejména pokud se prohledávaný stavový prostor skládá z více vzájemně nepropojených úseků, protože dovolí částicím opustit oblast, v které se optimum pravděpodobně nenachází (jinak by se částice nedostala za hranici této oblasti, naopak by v ní zůstala) a nová částice může prozkoumávat jinou oblast.

- **Odmítnutí pozic mimo prostor jako `gbest` a `pbest`**
  Pokud se nějaká z částic dostane mimo stavový prostor, její paměť pro svou vlastní nejlepší pozici se nemění a zůstane nastavena na pozici, která se nachází v povoleném prostoru. Částice také nemůže být označena jako nejlepší v rámci populace. Pozice `gbest` a `pbest` výrazně ovlivňují směr, kterým se částice bude pohybovat, a tato strategie garantuje, že se dané pozice nachází v povoleném stavovém prostoru. Tím

zaručuje, že pokud se některá částice při prohledávání dostane mimo stavový prostor, bude postupně nasměrována zpět mezi zvolené hranice. Částice mimo stavový prostor také nemůže ovlivnit ostatní částice, protože nemůže být označena jako `gbest`.

- **Neumožnit částicím překročit hranice**
  Při změně pozice každé částice lze určit, zda se cílová pozice nachází mimo stavový prostor. V takovém případě pak můžeme vektor rychlosti částice upravit několika způsoby, které zaručí, že částice skončí ve stavovém prostoru. Hranici si můžeme představit jako stěnu, jak lze vidět na obrázku 3.2. Nejprostším způsobem je vynulování rychlosti částice ve směru, ve kterém narazí na stěnu stavového prostoru. Výsledkem bude pohyb rovnoběžný se stěnou. Další možnosti je simulovat odraz částice od pomyslné stěny stavového prostoru. Odraz může být deterministický, při kterém se částice odrazí s přesně stejnou velikostí vektoru rychlosti ale do opačného směru v dané dimenzi a nebo může být její nová rychlost generována náhodně.

- **Penalizace částic mimo stavový prostor**
  Pokud částice prohledávají více navzájem nepropojených regionů, je žádoucí, aby mezi nimi mohly cestovat. Zároveň však chceme částic odradit od setrvávání v prostoru, o kterém víme, že se v něm řešení nenachází. Jednou z možností je zavedení penalizační funkce, která je aplikována na dané částice a snižuje hodnotu jejich fitness, pokud se nachází mimo limity prohledávaného prostoru.



Obrázek 3.2: Příklad možného chování částice na hranicích prohledávaného stavového prostoru podle [20] při optimalizaci dvou parametrů. (a) Stěna absorbuje rychlost v dané dimenzi. (b) Částice se odrazí zpátky do stavového prostoru. (c) Částice cestuje mimo stavový prostor ale není dále zahrnována jakou součást populace, dokud se nevrátí.

3.7 Další rozšíření technik PSO

Kromě výše zmíněných možností, jak modifikovat algoritmus PSO, existují i další známé varianty, které upravují PSO. Tyto varianty jsou pokročilé, protože mění princip algoritmu do větší míry a zavadí do něj nové koncepty.

Struktura populace

Jako strukturu populace označujeme skupiny částic, které mezi sebou komunikují a navzájem se ovlivňují. Protože interakce a adaptace částic je hlavní myšlenkou algoritmu, zvolená struktura má výrazný vliv na průběh algoritmu a jeho schopnost konvergovat k dobrému řešení. Původní a nejjednodušší myšlenkou PSO byla jednolitá struktura všech částic, nazývaná jako `gbest PSO` nebo **`star`** (hvězda). V tomto uspořádání komunikuje každá částice s každou a populace hledá optimum jako jedno hejno. Hlavní výhodou této struktury je rychlá konvergence částice k jednomu řešení - všechny jsou ovlivňovány stejnou `gbest` částicí. Rychlá konvergence má však za následek nedostatečné prozkoumání stavového prostoru a časté uvěznění částic v lokálním optimu. Rozdílným přístupem je `lbest PSO`. Rozdělením populace do menších skupin zachováme myšlenku vzájemné komunikace a učení se od ostatních a zároveň docílíme důkladnějšího prohledávání stavového prostoru. Jednou z myšlenek je rozdělení populace do několika nezávislých hejn. Částice mezi sebou komunikují v rámci hejna a jednotlivá hejna si přenáší informace mezi sebou. Tento přístup byl úspěšně aplikován na řadu standardních testovacích optimalizačních problémů [24]. Výhodou několika hejn je možnost najít více nezávislých řešení a proto jsou jejich hlavní aplikací právě problémy, při kterých potřebujeme získat více než jedno řešení. Další možností je zavedení logicky tvořené sociální struktury. Částice komunikuje pouze s částí populace, která je zvolena na základě předem definovaných pravidel. Studie těchto tzv. topologií prokázala výrazný dopad na výkon algoritmu [11].

Predátor a kořist

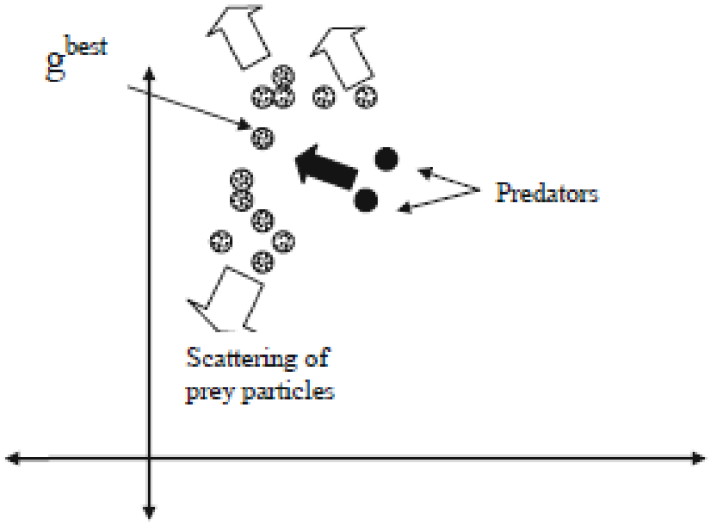
Modifikace predátor a kořist [1] rozdělí částice do dvou skupin. První skupina, označená jako kořist, stejně jako v původní verzi prozkoumává stavový prostor, avšak navíc je odpuzována od predátorů. Druhá skupina, predátoři, je přitahována k nejlepším jedincům ve skupině kořistí. Tyto nové síly mají za následek zavedení nové dynamiky. Hejno kořistí, konvergující kolem nejlepší částice v hejnu, je rozptýleno příchodem predátora a je nuceno prozkoumávat další části stavového prostoru.

3.8 Diskrétní PSO

Algoritmus PSO byl původně navržen pro práci s reálnými čísly. Motivací k vytvoření diskrétního PSO byla řada optimalizačních problémů, definovaných v diskrétním stavovém prostoru, jako například problém barvení grafů, problém obchodního cestujícího nebo problém n dam. Pomocí několika změn ve způsobu reprezentace pozice a rychlosti částic, definici operátorů v základních rovnicích PSO a dalších úprav můžeme dosáhnout adaptace původního PSO na zmíněné problémy. Konkrétních způsobů implementace diskrétního PSO existuje celá řada, protože každý ze zmiňovaných diskrétních problémů může vyžadovat jinou reprezentaci pozice částic a speciálně upravené rovnice pro změnu pozice. Speciální verzí diskrétního PSO je binární PSO [13], přičemž tato varianta byla vyvinuta pro optimalizaci v binárním stavovém prostoru.

Problém barvení grafů

Jako **graf** označujeme v teorii grafů objekt, který reprezentujeme pomocí množiny vrcholů a množiny hran spojujících některé z vrcholů. Pro problém barvení grafů se budeme za-



Obrázek 3.3: Skupina dvou predátorů působí na skupinu kořistí. [1].

bývat pouze obyčejným neorientovaným grafem s konečnou množinou vrcholů. Přesnější matematickou definicí je: [8]

> Graf G je uspořádaná dvojice (V, E), kde V je nějaká neprázdná množina a E je množina dvoubodových podmnožin množiny V. Prvky množiny V se jmenují vrcholy grafu G a prvky množiny E hrany grafu G.

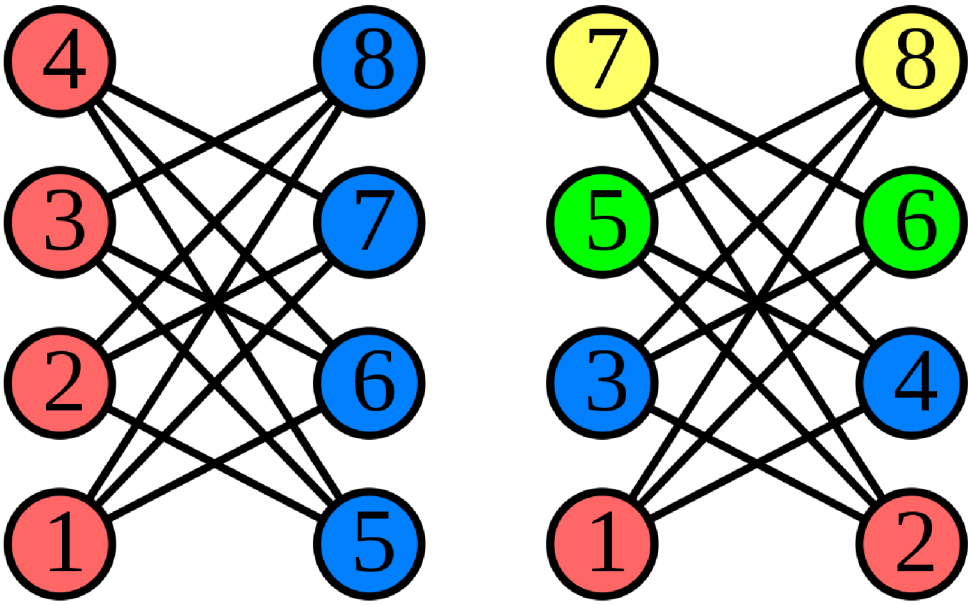
Grafem můžeme reprezentovat například města a cesty mezi jednotlivými městy nebo skupinu lidí a jejich vzájemné vazby (kteří se navzájem znají). Problém barvení grafů se zabývá přiřazením barvy každém z vrcholů tak, že žádné dva vrcholy spojené alespoň jednou hranou nemají stejnou barvu. Cílem je přitom graf vybarvit za použití co nejmenšího počtu barev. Jako barevnost grafu (nebo také `chromatické číslo`) označujeme nejmenší přirozené číslo $\chi(G)$, pro které existuje obarvení grafu G pomocí $\chi(G)$ barev. Obarvením grafu G pomocí k barev myslíme libovolné zobrazení [6]:

$$c : V\ (G) \rightarrow \{1, 2, ..., k\}$$

> takové, že každé dva vrcholy spojené hranou dostanou různé barvy, tj $c(u) \neq c(v)$ pro všechny $u, v \in E(G)$.

Problém barvení grafů se řadí do NP-úplných problémů [17]. Jedná se o třídu problémů, pro které lze v polynomiálním čase ověřit správnost řešení ale (zatím) nelze v polynomiálním čase správné řešení nalézt. Jednou z možností, jak tyto problémy řešit, jsou evoluční algoritmy. Přestože evoluční algoritmy nezaručují nalezení řešení i pokud existuje, protože jsou nedeterministické, stále jsou výrazně lepší možností než například prohledávání hrubou silou. Tímto přístupem bychom pro graf o n vrcholech za použití k barev museli provést (v nejhorším případě) k^n přiřazení barev a kontrol, zda se jedná o správné obarvení. Dalším zajímavým přístupem je `hladové barvení` (greedy coloring). Pomocí tohoto algoritmu

lze najít řešení pro každý graf, ne vždy je však řešení rozumně kvalitní. Hladové barvení prochází postupně všechny vrcholy grafu a každému přiřadí nejmenší možnou barvu (barvy jsou často reprezentovány jako celá čísla), které neporušuje podmínky obarvení grafu. Pořadí vrcholů, které si zvolíme, tedy výrazně ovlivní chromatické číslo grafu, které algoritmus vypočítá. Tato situace je ilustrována obrázkem 3.4. Hladové barvení je výpočetně nenáročné a dalo by se použít například jako součást některé metody inicializace populace (metody inicializace popsány v kapitole 3.3).



Obrázek 3.4: Ukázka dvou různých obarvení stejného grafu při změně pořadí vrcholů za použití hladového barvení. V prvním případě jsme získali chromatické číslo $\chi = 2$, v druhém $\chi = 4$. Příklad převzat z [7].

3.8.1 PSO pro barvení grafů

Jedna z možných modifikací klasického PSO na diskrétní byla navržena pro problém barvení grafů. Konkrétní implementační detaily, způsob reprezentace grafu, částic a úprava rovnic pro aktualizaci pozice částic, vycházející z literatury [10] [9], jsou popsány v následující kapitole.

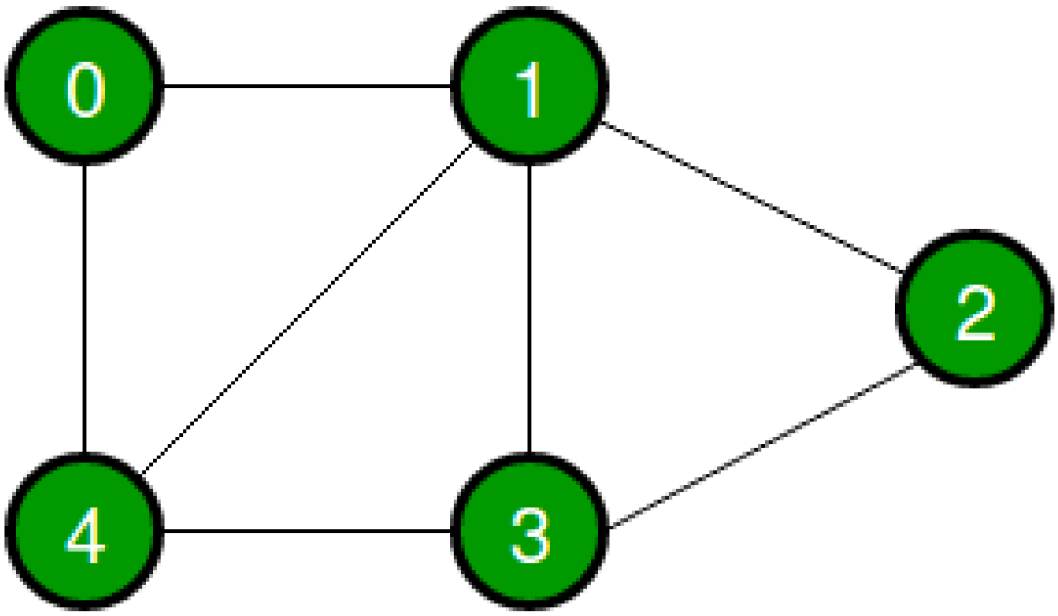
Graf

Z mnoha způsobů, jakými se dá graf reprezentovat, pro PSO postačí nejjednodušší reprezentace. Graf můžeme zapsat jako seznam všech hran. Jeden záznam v seznamu představuje jednu hranu a obsahuje informace o dvou vrcholech - z kterého hrana vychází a v kterém končí (jedná se o obyčejný neorientovaný graf, takže na pořadí vrcholů v zápisu nezáleží). Jednotlivé vrcholy jsou reprezentovány celými čísly, počínaje od jedničky, jde pouze o prosté číslování, aby šlo na vrcholy nějak odkazovat.
Graf na obrázku 3.5 tedy zapíšeme jako:

- hrana 1 2

- hrana 1 5
- hrana 2 3
- hrana 2 4
- hrana 2 5
- hrana 3 4
- hrana 4 5

Tento způsob zápisu nám umožní mimo jiné snadno kontrolovat, zda je graf správně obarvený. Stačí iterovat přes daný seznam hran, a pro každou ověřit, zda oba její vrcholy mají rozdílnou barvu.



Obrázek 3.5: Ukázka jednoduchého grafu. Příklad převzat z [2].

Stavový prostor

Stavový prostor představuje všechny možné pozice, na kterých se mohou jednotlivé částice v průběhu běhu algoritmu vyskytovat. Snažíme-li se obarvit graf z obrázku 3.5 pomocí tří barev, stavový prostor čítá 3^5 tedy 243 možných stavů. Jedná se tedy o konečnou množinu, na rozdíl od původního PSO s reálnými hodnotami. Obecně lze tedy počet kombinací vyjádřit jako k^n kde k je počet barev a n je počet vrcholů grafu. Reprezentaci jednotlivých částic ve stavovém prostoru se věnuje následující kapitola.

Pozice částic

Pozici částice zapisujeme jako vektor celých nezáporných čísel. Vektor má vždy délku právě n, kde n je počet vrcholů grafu. Čísla, která se mohou vyskytovat, začínají od 0 a maximálně mohou nabývat hodnoty $k - 1$, kde k je chromatické číslo, kterého se snažíme

barvením dosáhnout. Tato čísla reprezentují jednotlivé barvy a pořadí čísel souhlasí s pořadím očíslovaných vrcholů grafu, takže takto zapsaná pozice určuje konkrétní obarvení grafu (ve skutečnosti se nejedná o jediné konkrétní obarvení, pro pozici využívající k barev existuje $k!$ různých permutací z téhož zápisu, jde však pouze o změnu pořadí barev a to je z hlediska barvení grafů irelevantní).
Na obrázku 3.6 lze vidět graf s $n = 6$ a $k = 3$. Částici představující zobrazené obarvení bychom zapsali například jako $(0, 1, 2, 0, 2, 1)$.

Rychlosti částic

Rychlost částic je, podobně jako jejich pozice, zapisována pomocí vektoru celých čísel délky právě n, avšak může nabývat i záporných hodnot. Připomeňme, že rychlost určuje, o jakou vzdálenost (a kterým směrem) se v dané dimenzi částice v aktuálním kroku algoritmu posune. Protože se pohybujeme v diskrétních číslech, je potřeba rovnici výpočtu rychlosti upravit tak, aby jejím výsledkem byla diskrétní hodnota. Úprava rovnice je popsána níže.

Hodnocení částic

Částice hodnotíme pomocí fitness funkce. Hodnocením můžeme určit kvalitu řešení, která aktuální pozice reprezentuje. Ve většině problémů, na které lze PSO aplikovat, předem neznáme optimum, víme pouze, že se snažíme dosáhnout co nejnižšího (případně co nejvyššího, záleží, jak si zvolíme fitness funkci) ohodnocení. Pro barvení grafů definujeme výpočet fitness funkce tak, že hodnota 0 značí globální optimum, tedy korektně obarvený graf. Tento způsob představuje velkou výhodu zvlášť pro volbu koncové podmínky, jak je popsáno v kapitole 3.5. Označíme-li celkový počet hran, které mají oba vrcholy obarvené stejnou barvou jako $confictEdges$ a celkový počet vrcholů, které náleží alespoň k jedné hraně, která má oba vrcholy obarvené stejnou barvou jako $conflictVertices$, ohodnocení částice vypočítáme podle rovnice 3.4 [10].

$$f = a \times conflictVertices + confictEdges \tag{3.4}$$

Konstanta a je pozitivní koeficient, zpravidla nastavený na hodnotu větší než 1. U grafů s vysokou hustotou hran (vysokém poměru $hrany/vrcholy$) často dojde k tomu, že počet vrcholů s konfliktem je také výrazně větší než počet hran s konfliktem a nemá tím pádem no hodnotu fitness funkce velký vliv. Vynásobením konstantou a dosáhneme větší rovnováhy a rovnoměrného vlivu obou faktorů.

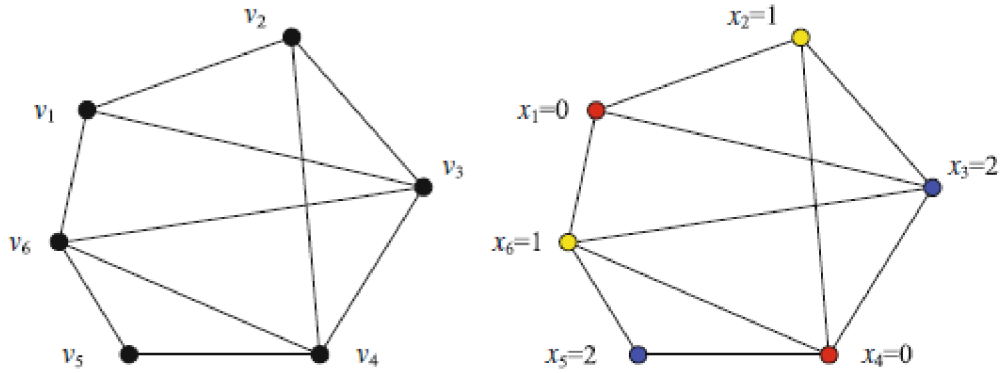
Upravené operace algoritmu

Jak již bylo řečeno, pro aplikaci PSO na barvení grafů musíme upravit některé rovnice algoritmu a upřesnit některé operace. Rovnice výpočtu rychlosti 3.2 musí produkovat vektor celých čísel, proto použijeme celočíselné zaokrouhlení. Výsledná rovnice potom bude vypadat takto:

$$v_i(t+1) = INT(v_i(t) \cdot w) + INT(r1 \cdot c1 \cdot (gbest_i(t) - x_i(t))) + INT(r2 \cdot c2 \cdot (pbest_i(t) - x_i(t))) \tag{3.5}$$
$$pro \quad i \in 1..N$$

Alternativní možností implementace výpočtu rychlosti je strategie založená na pravděpodobnosti. Vektory generované generované standardní rovnicí pro výpočet rychlosti se stanou vektory pravděpodobnosti, která určuje zda částice v dané dimenzi změní pozici.

Rovnice 3.1 popisující aktualizaci pozic částic pak může zůstat stejná. Jakou vstup dostane vždy celá čísla a provádí pouze sčítání, výstupem budou tedy vždy také celá čísla. Sčítání a odčítání rychlostí a pozic dodržuje pravidla práce s vektory, stejně jako násobení rychlosti konstantou.



Obrázek 3.6: Graf o 6 vrcholech barvený pomocí 3 barev. Převzato z [10].

Kapitola 4

Návrh knihovny kipso

Tato a následující kapitola popisují jádro práce – návrh a implementaci knihovny kipso. Součástí této kapitoly je popis požadavku na navrhovanou knihovnu. Požadavky vychází z potřeb cílové uživatelské skupiny a z nedostatků odhalených při rešerši současných nástrojů popsaných v kapitole 2. Na základě jejich analýzy je pak prezentován návrh dekompozice knihovny.

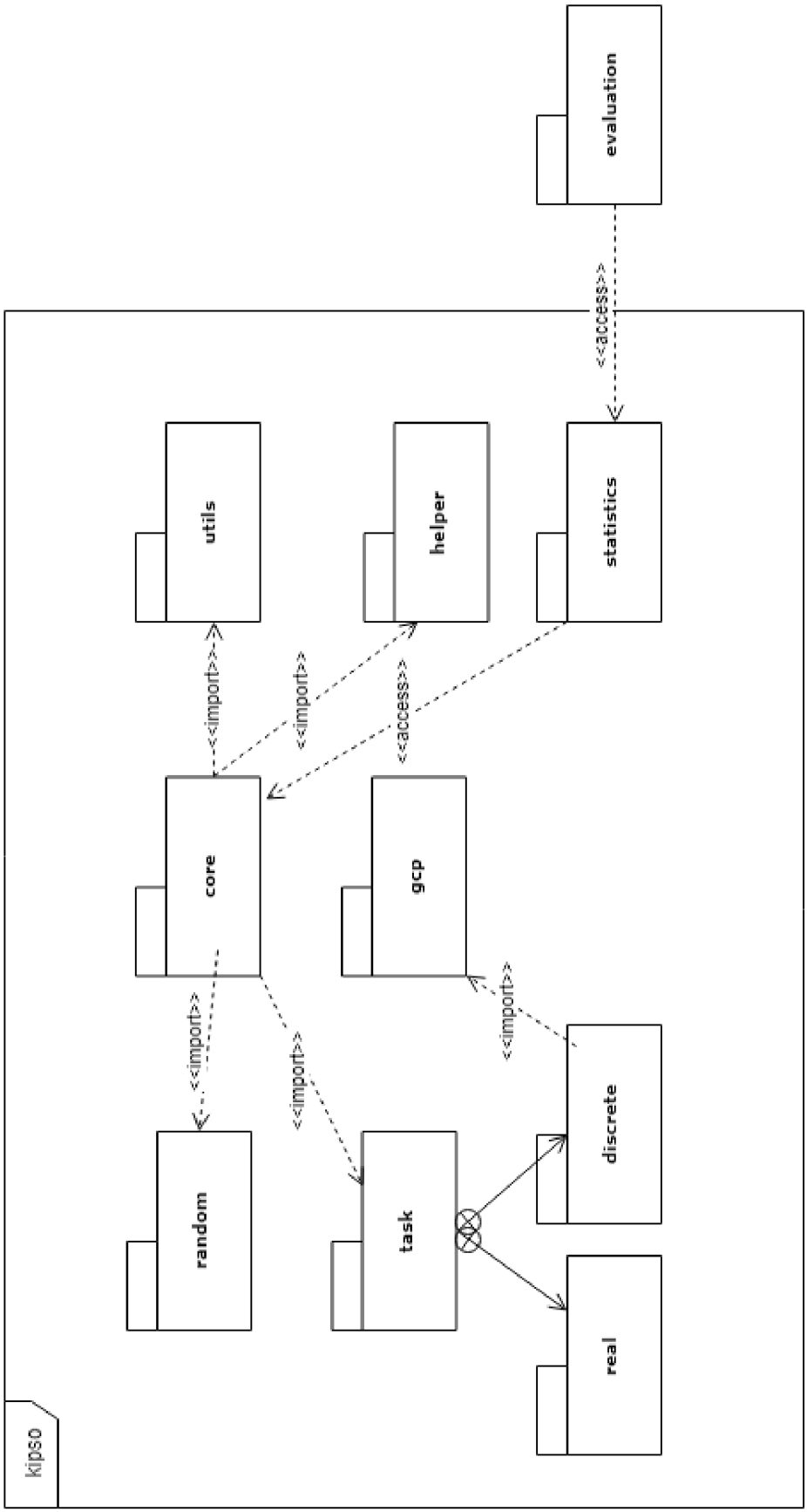
4.1 Specifikace požadavků

Hlavní cílové uživatelské skupiny pro vytvářenou aplikaci jsou studenti a výzkumníci v oblasti PSO algoritmů. Knihovna by měla sloužit primárně jako nástroj pro experimentování s PSO. Pro použití za tímto účelem musí poskytovat především následující funkčnost.

- Testování aplikace algoritmu na libovolné optimalizační úlohy.
- Implementace vlastních optimalizačních úloh.
- Možnost konfigurace algoritmu.
- Možnost implementace vlastních variací algoritmu a užití libovolných kombinací těchto variací.
- Sběr statistik z experimentů.
- Možnost vyhodnocení statistik a jejich interpretace a srovnání výkonosti jednotlivých variací algoritmu.
- Srozumitelné a kvalitně zdokumentované aplikační rozhraní.

4.2 Moduly

Návrh rozdělení do modulů byl zaměřen na použitelnost a rozšiřitelnost. Implementace algoritmu PSO musí být logicky oddělená od dalších částí knihovny, které poskytují pomocné funkce. Díky tomuto rozčlenění dosáhneme nízké provázanosti mezi jádrem a zbytkem knihovny. To je žádoucí, protože umožňuje úpravy vnitřní logiky jednotlivých částí bez nutnosti zásahu do implementace ostatních komponent. Další výhodou této struktury je možnost testovat jednotlivé části odděleně. Ve zbytku této sekce je popsán návrh rozdělení do modulů, znázorněný na obrázku 4.1, a vysvětlen význam a závislosti každého z modulů.



Obrázek 4.1: Diagram modulů knihovny kipso modelovaný v jazyce UML.

core

Cílem bylo vytvořit ústřední modul implementující algoritmus PSO, který je do největší možné míry izolovaný od zbytku knihovny. Tento modul zapouzdřuje veškerou funkcionalitu algoritmu. Balíček `core` se skládá z jedné hlavní komponenty, která je abstrakcí algoritmu PSO, a ostatních komponent, které reprezentují jednotlivé logické celky v rámci algoritmu. Ústřední komponenta je potom tvořena kompozicí ostatních komponent. Tento vztah rozdělí algoritmus do částí, které mohou být nezávisle na sobě nahrazovány různými implementacemi. Detaily implementace modulu jsou popsány v kapitole 5.3.

task

Tento balíček obsahuje konkrétní optimalizační úlohy. Ty jsou členěny do dvou podřazených balíčku `discrete` a `real` podle charakteru dané úlohy. Jeho jedinou závislostí je balíček `gcp`.

gcp

Balíček `gcp` je jednoduchá nezávislá jednotka, která poskytuje potřebné nástroje k práci s grafy. Je využíván balíčkem `task`, který s jeho pomocí formuje problém barvení grafů jako optimalizační úlohu. Nahrazením tohoto modulu implementací specifickou pro jiný optimalizační problém lze snadno vytvořit prostředky pro řešení daného problému.

random

Balíček `random` poskytuje funkcionalitu pro generování náhodných čísel. Tato komponenta je využita pro kompozici algoritmu PSO – každý vytvořený algoritmus má vlastní generátor náhodných čísel. Provedené experimenty jsou potom snadno reprodukovatelné. Komponenta není součástí balíčku `core`, protože se nejedná o konfigurovatelný prvek, který by byl přímou součástí algoritmu PSO, ale spíš o nástroj, který je modulem `core` využíván.

helper

Pro usnadnění práce s knihovnou existuje balíček `helper`. Obsahuje funkcionalitu, která je společná pro implementaci některých komponent v balíčku `core` a může tedy být znovu použitá při rozšíření knihovny. Dále také obsahuje pomocné nástroje, pomocí kterých lze snadno provádět experimenty.

utils

Balíček `utils` poskytuje obecné nástroje, které nesouvisí s PSO ale jsou v knihovně využívány například pro logování a nebo validaci.

statistics

Jediným výstupním bodem knihovny je balíček `statistics`, pod který patří dvě komponenty. První z nich přistupuje do `core` a z relevantních komponent extrahuje data o průběhu algoritmu. Je zodpovědná za zpracování těchto dat a jejich agregaci do podstatných metrik. Zpracovaná data předává druhé komponentě, která formuje výstup knihovny ve zvoleném formátu. Ten může být dále libovolně zpracován – to už je mimo kompetenci knihovny `kipso`.

evaluation

Na rozdíl od předchozích, `evaluation` není součástí knihovny `kipso`. Čte výstupy vytvořené modulem `statistics` a provádí jejich vyhodnocení, srovnání a vizualizaci. Je pouze jedním z příkladů, jak lze výstup knihovny využít.

Kapitola 5

Implementace

Tato kapitola popisuje prostředky zvolené pro implementaci knihovny. Dále rozebírá navržené aplikační rozhraní a funkcionalitu jejích klíčových komponent.

5.1 Implementační jazyk

V dané oblasti, jak vyplývá z popisu současných nástrojů v kapitole 2, standardně výrazně dominuje Python. Pro implementaci knihovny kipso byl zvolen Kotlin. Tento poněkud netradiční jazyk v oblasti umělé inteligence byl zvolen z několika důvodů. Kotlin je multiplatformní. Umožňuje kompilaci do javovského bajtkódu, javascriptu a do nativního strojového kódu, podporující všechny z hlavních platforem [1]. Knihovna kipso tak může potenciálně najít uplatnění v široké škále aplikací. Kotlin je kompatibilní s Javou a umožňuje používání jakékoliv javovské knihovny. K dispozici je tak obrovský ekosystém vybudovaný za dlouhou dobe existence Javy a celá řada nástrojů, která může být využita k rozšíření knihovny. V neposlední řadě knihovna implementovaná v Kotlinu také představuje alternativu pro uživatele, kteří nemají s Pythonem zkušenosti.

5.2 Uchovávání dat

Knihovna produkuje při běhu několik typů dat. Jedním z výstupů jsou logovací informace. Primárním účelem logovacích zpráv je ladění aplikace a podávání informací o aktuálním průběhu experimentu. Knihovna využívá logovací knihovnu Logback, která je plně konfigurovatelná a umožňuje několik typu ukládání dat [2]. Vzhledem k povaze knihovny jsou tyto výstupy důležité především při vývoji a proto je většinou není potřeba dlouhodobě uchovávat. Hlavním datovým výstupem jsou statistiky z průběhů experimentů. Způsob ukládání těchto dat je konfigurovatelný a to kvůli flexibilitě a možnosti vyhovět všem uživatelům. Předpokládá se však, že většina si vystačí s výchozí implementací která ukládá statistiky do souboru csv. Její konfigurace je popsaná v kapitole 5.3. Ta byla zvolena, protože data nevyžadují žádné šifrování, nedosahují extrémně velkého objemu a je žádoucí, aby formát byl čitelný v textovém editoru.

[1] https://kotlinlang.org/docs/reference/native-overview.html#target-platforms
[2] https://logback.qos.ch/manual/index.html

5.3 Rozhraní

Rozhraní a závislosti klíčových komponent knihovny je modelováno v diagramu tříd na obrázku 5.1. Diagram se zaměřuje především na třídy z balíčku `core`, protože tento balíček je hlavní komponentou, která se bude v průběhu životního cyklu knihovny vyvíjet a rozšiřovat. Zahrnuje však i další třídy a rozhraní, pokud jsou určena k rozšiřování a jsou tedy z pohledu uživatele knihovny důležitá.

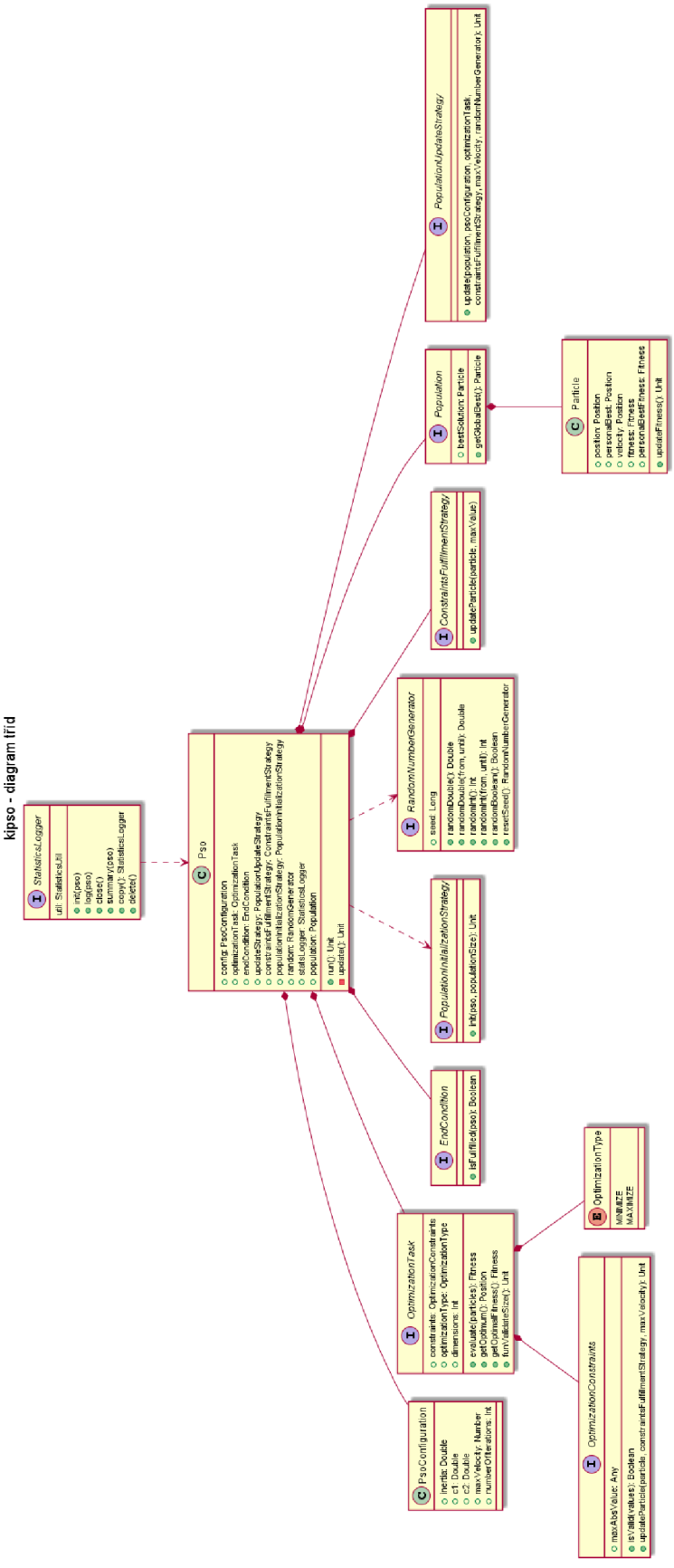
Kompozice algoritmu PSO

Algoritmus PSO je v rámci knihovny reprezentován třídou `Pso`. Tato komponenta implementuje průběh algoritmu ve zjednodušené podobě. Provede inicializaci a poté aktualizuje populaci, dokud není splněna ukončovací podmínka. Je zodpovědná za koordinaci a řízení celého algoritmu. Veškerou konfigurovatelnou logiku však deleguje. Vzhledem k vysoké obecnosti této třídy není pravděpodobné, že ji bude potřeba v rámci implementace některé z modifikací algoritmu měnit, přesto je však označena modifikátorem `open`[3] a poskytuje tak maximální flexibilitu. Pro instanciaci třídy je potřeba dodat implementaci jejich jednotlivých komponent a tím definovat chování algoritmu. Příklad vytvoření instance `Pso` je zapsaný v ukázce kódu 5.1. V příkladu uvedené komponenty jsou jednou z možných konkrétních implementací rozhraní znázorněných v diagramu tříd. Knihovna je navržena tak, aby mohla pracovat s různými datovými typy (`Double` pro PSO s reálnými čísly a `Integer` pro diskrétní PSO)[4]. K tomu využívá generické typové parametry[5]. Všechna rozhraní jsou definovaná jako jako nezávislá na datových typech a jednotlivé konkrétní implementace se mohou omezit pouze na zvolený datový typ. Následující sekce popisují funkcionalitu a použití jednotlivých rozhraní.

```
1  val pso = Pso(
2          PsoConfiguration(0.5, 2.0, 2.0, 0.6, 3000),
3          SphericalOptimizationTask(30),
4          CombinedEndCondition.any(
5                  FixedCountEndCondition(5000),
6                  FitnessValueEndCondition(RealFitness(.0))
7          ),
8          OriginalPopulationUpdateStrategy(),
9          SlidingConstraintsFulfillmentStrategy(),
10         RandomConstraintsConformingPopulationInitializationStrategy(),
11         DefaultRandomNumberGenerator(),
12         FileStatisticsLogger(
13                 LOG_DIRECORY,
14                 RealStatisticUtil(),
15                 StatisticsLoggingConfig(frequency = 10)
16         )
17     )
```

Výpis 5.1: Instanciace třídy Pso

[3] open je v kotlinu modifikátor přístupu, umožňující z dané třídy dědit
[4] Double v kotlinu reprezentuje 64 bitové číslo s pohyblivou řádovou čárkou, Integer je celé číslo
[5] https://kotlinlang.org/docs/reference/generics.html



Obrázek 5.1: Diagram tříd knihovny kipso modelovaný v jazyce UML.

Konfigurace parametrů

Jako první parametr očekává konstruktor třídy `Pso` objekt držící konfigurační parametry. V ukázce kódu jsou parametry vytvořeny na řádce 2. Třída `PsoConfiguration` je pouze `data class`[6]. V jejím konstruktoru jí předáváme parametry PSO, tedy koeficienty w, c_1, c_2, omezení rychlosti částic a počet částic.

Definice optimalizační úlohy

Řádek 3 ve výpisu 5.1 definuje optimalizační úlohu – zde reprezentovanou třídou `SphericalOptimizationTask` (protože se jedná o n dimenzionální funkci, je zde pomocí parametru specifikováno, že chceme vytvořit instanci s 30 dimenzemi). Každá třída reprezentující optimalizační úlohu musí implementovat rozhraní `OptimizationTask`. Toto rozhraní vyžaduje implementaci následujících instančních proměnných a metod.

- `optimizationType` – Určuje zda se pro danou úlohu snažíme minimalizovat či maximalizovat hodnotu *fitness* funkce. Nabývá hodnot z výčtového typu `OptimizationTask.OptimizationType`.
- `dimensions` – Specifikuje počet dimenzí dané úlohy.
- `optimizationConstraints` – Očekává instanci třídy implementující rozhraní `OptimizationConstraints`, která specifikuje omezení stavového prostoru pro danou optimalizační úlohu a provádí validaci jeho dodržení.
- `evaluate(particles)` – Tato metoda obsahuje samotnou logiku výpočtu hodnoty *fitness*.

Konfigurace ukončovací podmínky

Ukončovací podmínka se kontroluje jednou za každou iteraci v hlavní smyčce algoritmu. Tento kontrakt je pevně daný a při implementaci vlastní ukončovací podmínky s ním lze počítat. Třída reprezentující ukončovací podmínku musí implementovat rozhraní `EndCondition`. Toto rozhraní má jedinou metodu, `isFulfilled(pso)`, která jako parametr dostává aktuální stav běhu algoritmu a vrací informaci, zda je daná podmínka splněna a běh algoritmu má být ukončen. Je zodpovědností uživatele podmínku vhodně nakonfigurovat, neobsahuje žádný bezpečnostní mechanismus proti nekonečnému běhu. Speciálním případem ukončující podmínky je `CombinedEndCondition` umožňující vytvoření ukončovací podmínky složené z více různých podmínek. Ta může být pomocí svých továrních metod nakonfigurována tak, že pro její splnění stačí buď splnění pouze jedné z daných podmínek (`CombinedEndCondition.any(conditions)`) a nebo musí být splněny všechny specifikované (`CombinedEndCondition.all(conditions)`). V přiloženém výpisu kódu 5.1 je na řádku 4 až 7 ukázka definice ukončující podmínky, která je splněna, pokud algoritmus vykonal alespoň 5000 iterací a nebo pokud bylo nalezeno řešení s hodnotou *fitness* lepší nebo rovnou 0.

[6] Data class je v Kotlinu třída, jejíž jediným účelem je držet data, neposkytuje žádnou logiku. (Teoreticky je možné využít ji i jinak ale jedná se o porušení konvence.)

Aktualizace pozic částic

Způsob aktualizace pozice částic je ve výpisu 5.1 definován na řádku 8 a 9. Tato zodpovědnost je rozdělena mezi 2 komponenty. První z nich je implementace rozhraní `ConstraintsFulfillmentStrategy`. Ta dostává na vstupu aktuální pozici a rychlost částice, a jejím úkolem je pozici aktualizovat na základě dané rychlosti. Přitom však musí dbát na to, aby nedošlo k vybočení částice z omezeného stavového prostoru definovaného v `OptimizationTask.optimizationConstraints`. Pokud by k vybočení mělo dojít, její jediná metoda obsahuje logiku, jak aktualizovat rychlost a pozici částice, aby se opět nacházela v rámci stanoveného stavového prostoru. Za předpokladu, že daná optimalizační úloha nemá omezených stavový prostor, lze využít implementaci `IgnoreConstraintsFulfillmentStrategy`, která neprovádí žádnou kontrolu. Druhou z vytvářených komponent je implementace rozhraní `PopulationUpdateStrategy`. Její zodpovědností je vypočítat aktuální rychlost částic a provést aktualizaci jejich pozic (kterou deleguje na výše zmíněnou první komponentu). Výpočet nových rychlostí částic je samé jádro algoritmu. Zde je v přiloženém výpisu 5.1 na řádku 8 instanciována třída `OriginalPopulationUpdateStrategy`, která obsahuje implementaci původní verze PSO definované rovnicí 3.2.

Inicializace populace

Inicializace populace je řízena rozhraním `PopulationInicializationStrategy`. V příkladu 5.1 je implementace tohoto rozhraní instanciována na řádku 10. Rozhraní má jedinou metodu, `init`, která očekává instanci třídy `Pso` a počet částic k inicializaci.

Výpočet a logování statistik

Řádek 12 až 16 ve výpisu kódu 5.1 definuje způsob logování statistik. Třída zajišťující logování statistik musí implementovat rozhraní `StatisticsLogger` a musí povinně obsahovat `StatisticsLoggingConfig` (konfigurace definující frekvenci a úroveň detailu logovaných statistik) a pomocnou třídu implementující rozhraní `StatisticsUtil`. Ve výpisu kódu 5.1 je tato pomocná třída instanciována na řádku 14. Konkrétní zde zvolená implementace je třída `RealStatisticUtil`, protože optimalizovaná úloha pracuje s reálnými čísly . Implementace rozhraní `StatisticsUtil` musí poskytovat metody pro výpočet statistik z aktuálního stavu populace (např. získání nejlepší hodnoty *fitness*, výpočet průměrné rychlosti částic a další relevantní metriky). Zodpovědností výše zmíněného rozhraní `StatisticsLogger` je pak využití této pomocné třídy k výpočtu statistik a jejich logování. To se provádí pomocí následujících metod.

- `init(pso)` – Tato metoda je volána po inicializaci všech ostatních částí algoritmu. Měla by logovat počáteční konfiguraci algoritmu.
- `log(pso)` – Metoda log je volána každou iteraci algoritmu. Její zodpovědností je zaznamenat aktuální stav sledovaných metrik a umožnit tak následnou analýzu jejich vývoje.
- `summary(pso)` – Na konci algoritmu je pouze jednou volána metoda `summary`, která je určena k zaznamenání stavu algoritmu po ukončení běhu. Umožňuje logovat data, jejíchž vývoj není důležitý, ale zajímá nás pouze poslední dosažená hodnota.

Spouštění experimentů

Experiment definovaný ve výpisu kódu 5.1 lze spustit několika způsoby. Jednoduché jednorázové spuštění lze provést voláním metody `run` – `pso.run()`, případně zkráceně `pso()`[7]. Algoritmus zahájí běh a po jeho ukončení může být daná instance zahozena. Pro snadnější spouštění experimentů existuje třída `ExperimentRunner` nabízející následující funkce.

- `runExperiment()` – Slouží pro spuštění dané instance `Pso` vícekrát. Po každém jednotlivém běhu znovu inicializuje všechny potřebné komponenty. První řádek výpisu kódu 5.2 ukazuje použití této funkce pro spuštění experimentu stokrát.
- `gridSearch()` – Zbytek výpisu 5.2 demonstruje využití funkce `gridSearch`. Tato pomocná funkce je určena k optimalizaci parametrů PSO. Na základě dodané množiny parametrů vytváří a provádí experimenty s různými nastaveními. Ta je ve výpisu kódu 5.2 na řádku 4 a může být předána buď formou seznamu a nebo *range*[8]. Řádek 5 a 6 definuje optimalizovanou úlohu a implementaci rozhraní na logování statistik. Tyto dvě komponenty nepatří mezi optimalizované parametry a budou proto pro všechny běhy společné. Následující řádek definuje strategii pro volbu parametrů. K dispozici jsou `CartesianSearchStrategy` – strategie, která vyzkouší všechny možné kombinace parametrů právě jednou a `RandomSearchStrategy` – strategie, která při každém dalším experimentu vygeneruje parametry náhodně. Řádky 11 až 13 omezují maximální počet vyzkoušených kombinací na 1000, počet běhů pro jedno konkrétní nastavení na 100 a celkovou maximální dobu experimentu na 48 hodin. Další ukončovací podmínku je nalezení řešení s *fitness* alespoň 0,0001. Posledním předávaným parametrem je hodnota *seed*, která bude použita pro inicializaci generátoru náhodných čísel. Pomocí této hodnoty lze reprodukovat experimenty.

```
1  ExperimentRunner.runExperiment(times = 100) { pso }
2
3  ExperimentRunner.gridSearch(
4             hyperParams,
5             AckleyOptimizationTask(5),
6             FileStatisticsLogger(LOG_DIRECTORY,
7                     RealStatisticUtil(),
8                     StatisticsLoggingConfig(frequency = 10)
9             ),
10            HyperSpaceSearchStrategy.CartesianSearchStrategy(),
11            maxRuns = 1000,
12            runsPerConfig = 100,
13            maxRuntime = TimePeriod(48, TimeUnit.HOURS),
14            stoppingFitness = RealFitness(.0001),
15            seed = 123
16    )
```

Výpis 5.2: Ukázka použití `ExperimentRunner`

[7] Zde je využito přetížení operátoru volání. https://kotlinlang.org/docs/reference/operator-overloading.html#invoke

[8] *Range* je v Kotlinu výraz definující rozsah srovnatelných hodnot.

Interpretace výsledků

Požadavky na analýzu výsledků daly vzniknout nástroji, jehož účelem je načtení, zpracování, interpretace a vizualizace statistik z průběhu algoritmu. Není žádoucí, aby tento nástroj byl součástí samotné knihovny, ta poskytuje pouze logování surových dat a částečně agregovaných hodnot. Z tohoto důvodu byl nástroj implementován jako Jupyter notebook dokument [9] v jazyku Python. Tento dokument umožňuje bez dalších úprav načíst logy vyprodukované knihovnou a vygenerovat z nich zprávu o dosažených výsledcích. Jednou z motivací pro zvolenou technologii byla i vysoká míra interaktivity, kterou dané prostředí nabízí. Analýza dat často vyžaduje individuální přístup k jednotlivým experimentům a sledováních různých metrik. Python v kombinaci s Jupyter notebook umožňuje rychlé prototypování a úpravy. Aktuálně dostupné funkce pro analýzu dat jsou dostupné v Jupyter notebook dokumentu s názvem `model_multi_run_evaluation.ipynb`.

- `load_data(path)` – Načte data o průběhu experimentu ze složky určené proměnnou `path`.
- `plot_run(data)` – Vykreslí krabicové grafy vývoje všech sledovaných metrik během iterací.
- `plot_final_state(data)` – Vykreslí histogramy rozložení sledovaných metrik na konci algoritmu.
- `print_extreme_solutions(data)` – Vrátí nejlepší a nejhorší nalezené řešení a jejich hodnoty `fitness`.
- `get_configs_by_fitness(data)` – Vypočítá agregační statistiky pro všechny nalezené konfigurace. Vrací objekt který je vstupem pro následující dvě funkce.
- `best_configs_by_mean_fitness(configs)` – Vypíše všechny použité konfigurace seřazené podle průměrné dosažené hodnoty `fitness`.
- `best_configs_by_optimums_found(configs)` – Vypíše všechny použité konfigurace seřazené podle počtu nalezených optimálních řešení.
- `get_iterations_to_converge(data)` – Vypíše statistiky o výpočetní náročnosti daných experimentů.

[9] https://jupyter.org/

Kapitola 6

Experimentální výsledky

Tato kapitola se zabývá aplikací knihovny kipso na konkrétní vzorové optimalizační problémy a demonstruje funkčnost na známých benchmarkových úlohách. Dále ukazuje srovnání dosažených výsledků s teoretickým optimem pro jednotlivé úlohy. U některých jednodušších úloh je optimum známé, pro složitější úlohy však není dokázáno a počítáme tedy s nejlepším známým nalezeným řešením. Pro zajištění statistické relevance výsledků bylo pro každý experiment provedeno alespoň 100 nezávislých běhu a uvedené závěry vyhodnocují nejlepší a průměrné dosažené výsledky. V následujících sekcích jsou popsány podrobné výsledky aplikace algoritmu PSO v úlohách s reálnými čísly a varianta diskrétního PSO aplikovaná na problém barvení grafů.

6.1 Výsledky pro úlohy s reálnými čísly

Na základě předchozích experimentů byly zvoleny 4 různé konfigurace parametrů PSO pomocí kterých byly provedeny testovací běhy algoritmu. Parametry jsou zaneseny v tabulce 6.1. Použití těchto parametrů poskytovalo dobré výsledky na široké škále optimalizačních úloh.

Tabulka 6.1: Konfigurační parametry použité pro experimenty

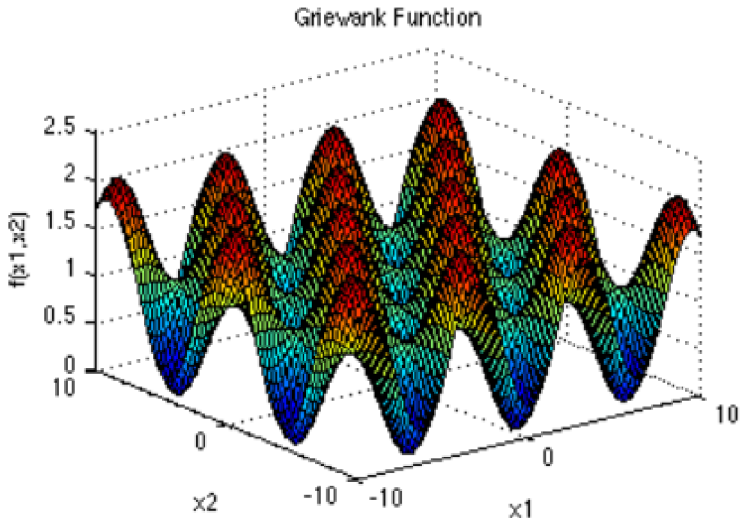
konfigurace	setrvačnost	c1	c2	limitace rychlosti	počet částic
1	0,3	2,0	2,0	0,6	3000
2	0,5	2,0	2,0	0,6	100
3	0,8	1,4	0,6	2,0	100
4	0,3	0,9	1,4	2,0	3000

Následující podsekce obsahují popis optimalizovaných problémů a podmínek experimentů. Ke každému problému je také uvedena interpretace výsledků a graf průběhu experimentu. Graf znázorňuje hodnotu `fitness` pro nejlepší částici z hejna v závislosti na počtu iterací. Každý graf obsahuje naměřené hodnoty ze 100 běhů, které byly omezeny na 10000 iterací a ukazuje nejvyšší a nejnižší hodnotu, medián a hodnoty prvního a třetího kvartilu. Tabulky s metrikami vychází ze stejných hodnot. Po provedení 10000 iterací se již nejlepší nalezené řešení za použití jakékoliv konfigurace nezlepšovalo, můžeme tuto hranici tedy prohlásit za dostačující. Naměřené hodnoty jsou zaokrouhleny na tři desetinná místa.

Griwankova funkce

Griewankova funkce je jedna z běžně používaných testovacích optimalizačních funkcí. Je definována pro n dimenzí, kdy $n \geq 1$. Následující test byl proveden pro $n = 10$.

$$f(\mathbf{x}) = \sum_{i=1}^{n} \frac{x_i^2}{4000} - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1 \tag{6.1}$$



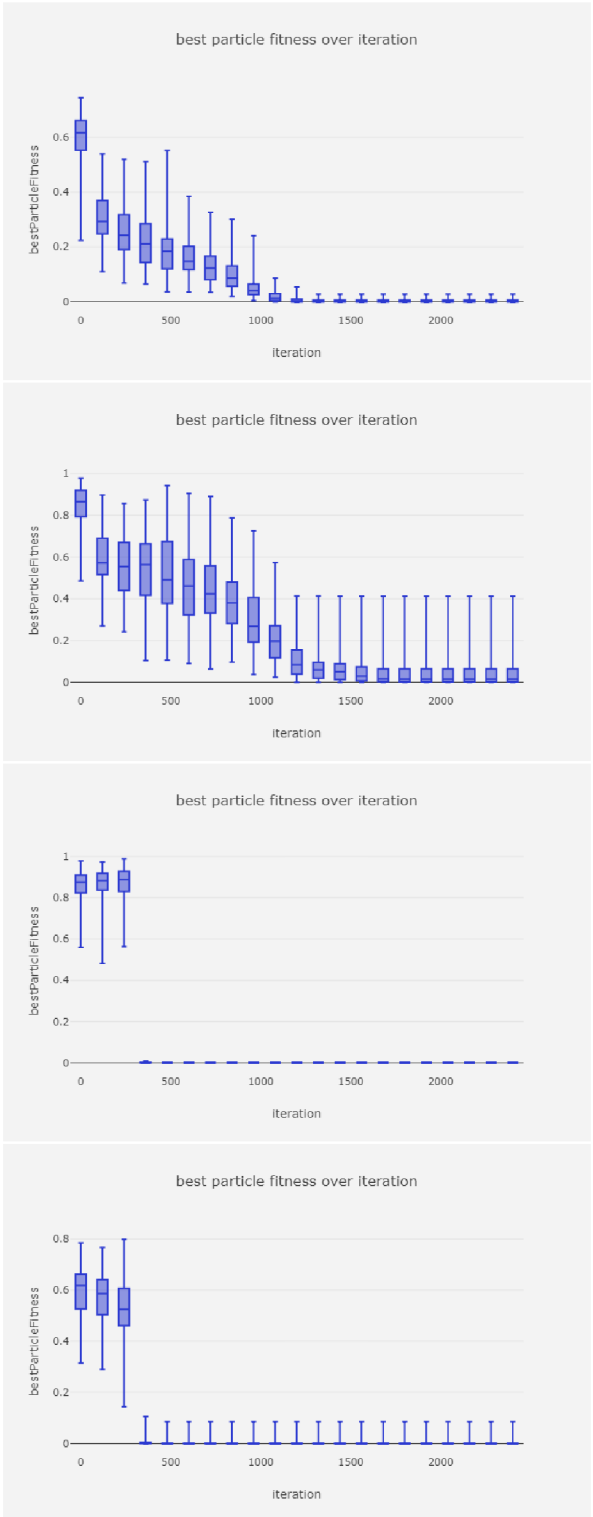
Obrázek 6.1: Griewankova funkce ve dvou dimenzích. Převzato z [15]. Upraveno.

Hodnota globálního minima je $f(x^*) = 0$. Jak lze vidět na obrázku 6.1, funkce má velké množství lokálních extrémů a představuje tak pro optimalizaci pomocí PSO náročný problém. Stavový prostor se omezuje na $x_i \in [-600, 600]$.

Tabulka 6.2: Naměřené hodnoty pro optimalizaci Griewankovy funkce.

metrika	konfigurace 1	konfigurace 2	konfigurace 3	konfigurace 4
optimum		0		
nejlepší řešení	0	0	0	0
podíl optimálních řešení	45 %	23 %	100 %	74%
medián	0,002	0,015	0	0
první kvartil	0	0,002	0	0
průměr iterací ke konvergenci	1410,4	2364,3	367,2	452,9

Pro optimalizaci Griewankovy funkce dosáhl algoritmus velice dobrých výsledků. Všechny konfigurace nalezly optimální řešení ve velké části běhů. Průměr iterací ke konvergenci je metrika, která ukazuje, za kolik iterací bylo nalezeno řešení, které již během stejného běhu nebylo překonáno. Lze tedy považovat za ukazatel výpočetní náročnosti. Je třeba si však



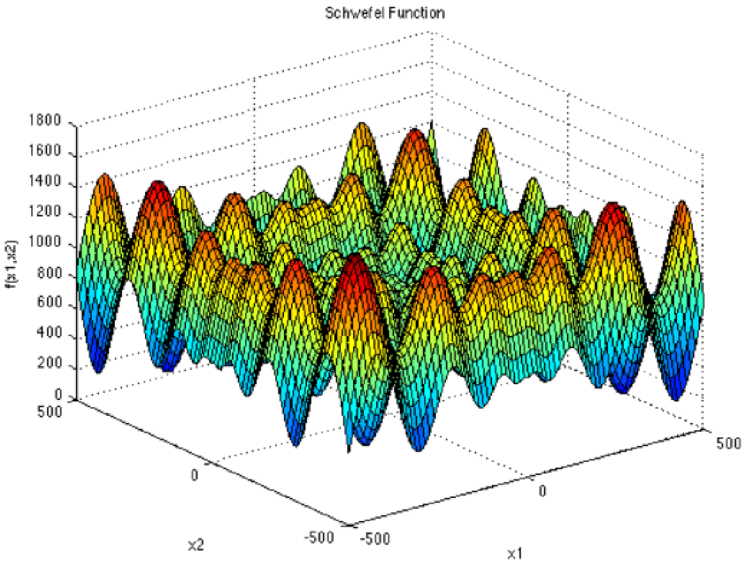
Obrázek 6.2: Optimalizace Griewankovy funkce pomocí specifikovaných konfigurací z tabulky 6.1.

uvědomit, že konfigurace s vyšším počtem částic bude provádět stejné množství iterací výrazně delší dobu a provede více vyhodnocení funkce *fitness*. Průměr iterací ke konvergenci lze tedy považovat pouze jako orientační hodnotu. Pokud srovnáme počet iterací, potřebný k nalezení řešení u Griwankovy funkce s ostatními příklady, zjistíme, že je znatelně vyšší. Důvodem je větší stavový prostor, který při této úloze algoritmus prohledával. Při pohledu na graf 6.2 si lze povšimnou, že konfigurace 3 a 4 nalezly řešení výrazně rychleji než konfigurace 1 a 2. Je to způsobeno nižšími koeficienty `c1` a `c2` v těchto konfiguracích. Částice mají menší celkovou rychlost a proto rychleji konvergují. Při ostatních experimentech, na grafech 6.4, 6.6 a 6.8 je tento jev také patrný.

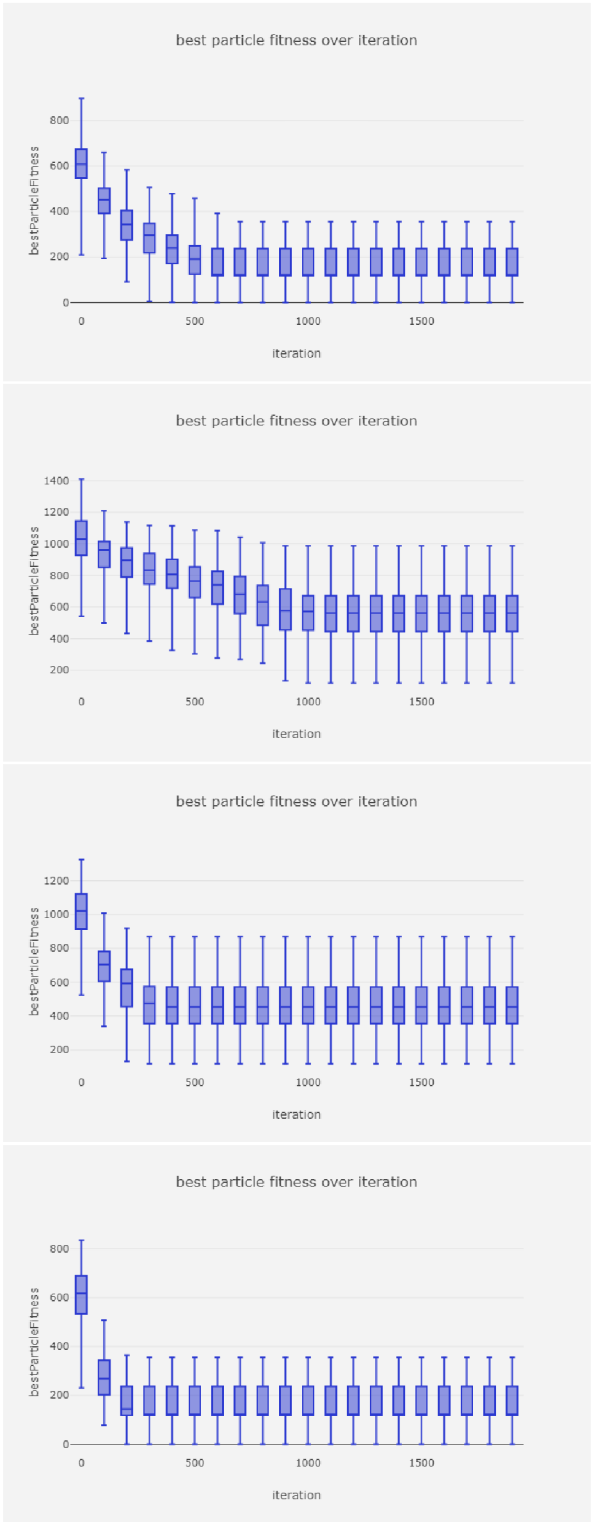
Schwefelova funkce

Schwefelova funkce je definovaná pro `n` dimenzí, kdy $n \geq 1$. Při vykreslení ve dvou dimenzích lze vidět, že obsahuje velké množství hustě rozmístěných lokálních extrémů. Stavový prostor se pro optimalizaci omezuje na $x_i \in [-512, 512]$. Hodnota globálního minima je $f(x^*) = 0$ pro $x^* = (420.968746, ..., 420.968746)$. Následující běhy algoritmu byly provedeny s $n = 5$.

$$f(\mathbf{x}) = 418.982887n - \sum_{i=1}^{n} x_i sin(\sqrt{|x_i|}) \tag{6.2}$$



Obrázek 6.3: Schwefelova funkce ve dvou dimenzích. Převzato z [3].



Obrázek 6.4: Optimalizace Schwefelovy funkce pomocí specifikovaných konfigurací z tabulky 6.1.

Tabulka 6.3: Naměřené hodnoty pro optimalizaci Schwefelovy funkce.

metrika	konfigurace 1	konfigurace 2	konfigurace 3	konfigurace 4
optimum		0		
nejlepší řešení	0	118,438	118,438	0
podíl optimálních řešení	9 %	0 %	0 %	11%
medián	118,483	562,586	454,016	118,438
první kvartil	118,483	444,148	355,315	118,438
průměr iterací ke konvergenci	581,1	996,3	326,6	225.5

Optimalizace Schwefelovy funkce je dle očekávání celkově méně úspěšná. Výsledky jsou zapsány v tabulce 6.3. Optimální řešení se podařilo najít pouze pomocí konfigurací 1 a 4, tedy těch s vysokým počtem částic v hejnu. Zde se ukazuje, jak je vysoký počet částic důležitý u optimalizací funkcí s které mají mnoho lokálních extrémů. Pokud není jedna z částic inicializována v blízkosti hledaného globálního extrému, šance, že globální extrém objeví, se výrazně zmenšuje.

Styblinski-Tang funkce

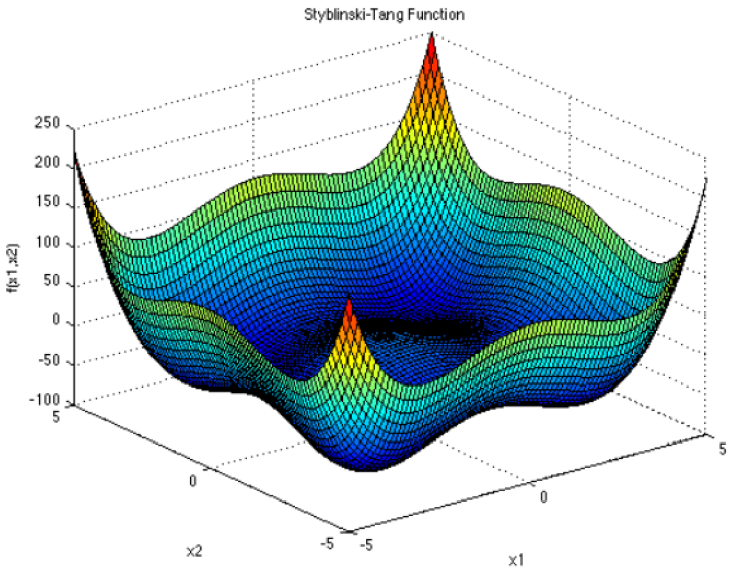
Styblinski-Tang je spojitá, nekonvexní funkce definovaná v `n` dimenzích, kdy $n \geq 1$. Jedná se o multimodální optimalizační problém. Stavový prostor se omezuje na $x_i \in [-5, 5]$. Globální minimum se nachází na $x^* = (-2.903534, ..., -2.903534)$ a jeho hodnota je $f(x^*) = -39.16599n$. Pro následující příklad byla optimalizována funkce s $n = 15$, takže hodnota globálního minima je -587.48985.

$$f(\mathbf{x}) = f(x_1, ..., x_n) = \frac{1}{2}\sum_{i=1}^{n}(x_i^4 - 16x_i^2 + 5x_i) \tag{6.3}$$

Tabulka 6.4: Naměřené hodnoty pro optimalizaci Styblinski-Tang funkce.

metrika	konfigurace 1	konfigurace 2	konfigurace 3	konfigurace 4
optimum		-587.48985		
nejlepší řešení	-587,4925	-573,356	-573,356	-573,356
podíl optimálních řešení	1 %	0 %	0 %	0%
medián	-559,219	-530,946	-516,809	-545,082
první kvartil	-559,219	-545.082	-530,946	-559,219
průměr iterací ke konvergenci	32,84	91,86	130,93	27,65

V tabulce 6.4 jsou zapsány výsledky experimentů na Styblinski-Tang funkci. Funkce vypadá na první pohled jako jednoduše optimalizovatelná – má tvar připomínající údolí. To má za důsledek velice rychlou konvergenci, pohybující se v desítkách iterací, na rozdíl od předchozích funkcí, kde byl počet iterací řádově vyšší. Přesto se však optimum podařilo nalézt pouze při jednom jediném běhu. Lze si všimnout, že nemalá část běhů má relativně dobré řešení, které však není přesným optimem. Zde opět hraje roli vysoký počet lokálních



Obrázek 6.5: Styblinski-Tang funkce ve dvou dimenzích. Převzato z [3].

extrému, které se od sebe navzájem liší jen o velice malé hodnoty funkce a algoritmus s danými konfiguracemi nedokáže najít ten nejlepší. Lepších výsledků by pravděpodobně bylo možno dosáhnout za pomocí některé z variací algoritmu podporující explorační fázi algoritmu.

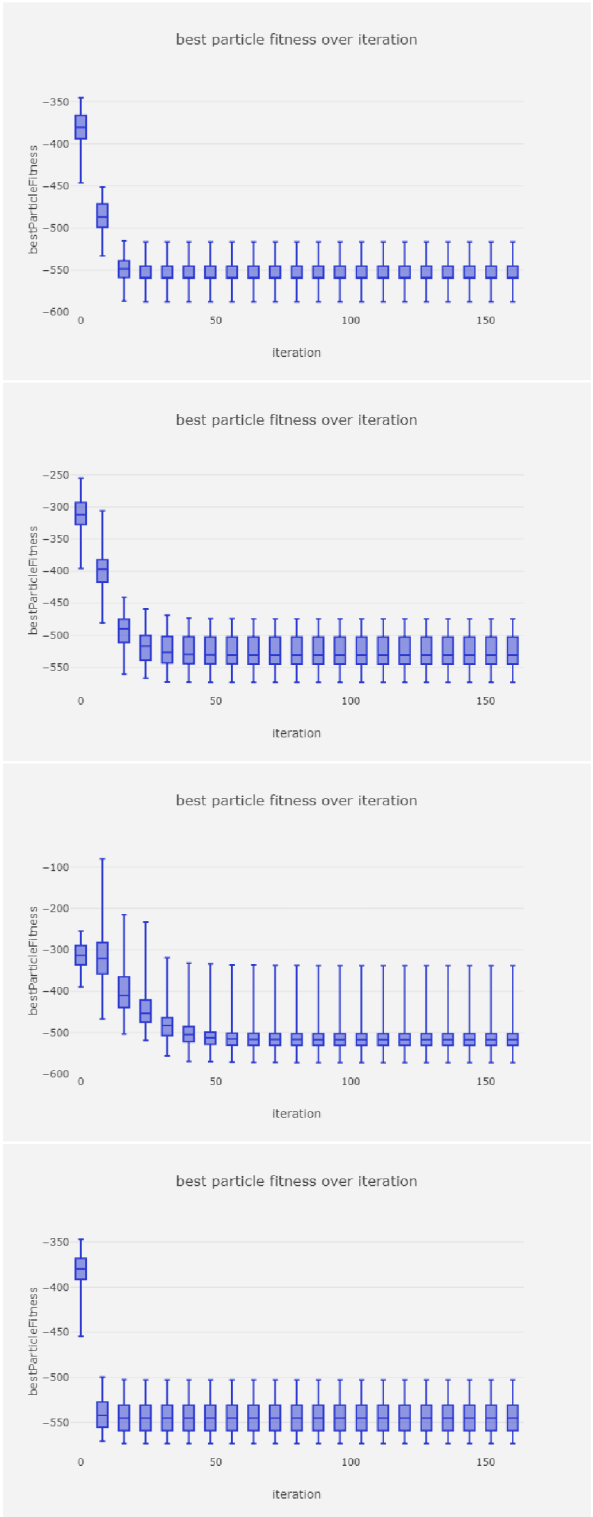
Ackleyho funkce

Ackleyho funkce představuje multimodální optimalizační problém definovaný v `n` dimenzích, kdy $n \geq 1$. Stavový prostor se pro optimalizaci omezuje na $x_i \in [-32,768, 32,768]$. Funkce je definovaná rovnicí 6.4. Koeficienty `a, b` a `c` byly pro tento příklad nastaveny na hodnoty 20, 0.2 a 2π. Globální minimum funkce se nachází na $x^* = (0, ..., 0)$ a jeho hodnota je $f(x^*) = 0$. Následující příklad počítá s funkcí ve 20 dimenzích.

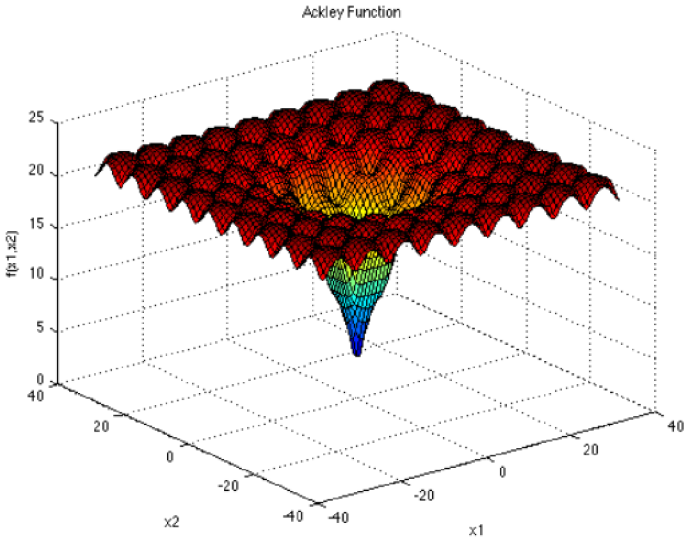
$$f(\mathbf{x}) = f(x_1, ..., x_n) = -a.exp(-b\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}) - exp(\frac{1}{n}\sum_{i=1}^{n} cos(cx_i)) + a + exp(1) \quad (6.4)$$

Tabulka 6.5: Naměřené hodnoty pro optimalizaci Ackleyho funkce.

metrika	konfigurace 1	konfigurace 2	konfigurace 3	konfigurace 4
optimum		0		
nejlepší řešení	0	0	0	0,010
podíl optimálních řešení	36 %	2 %	7 %	0%
medián	1,155	2,868	2,171	1,856
první kvartil	0	2,317	1,646	1,646
průměr iterací ke konvergenci	93,5	161,93	219,01	72,16



Obrázek 6.6: Optimalizace Styblinski-Tang funkce pomocí specifikovaných konfigurací z tabulky 6.1.



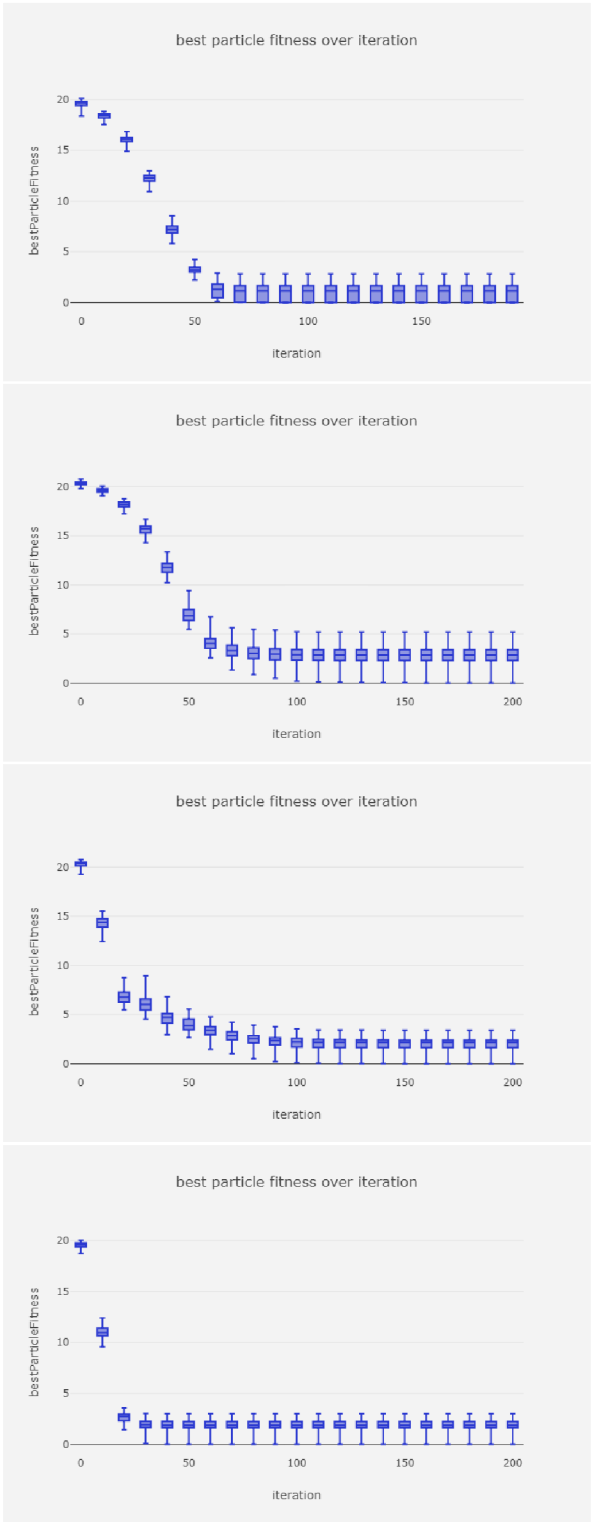
Obrázek 6.7: Ackleyho funkce ve dvou dimenzích. Převzato z [3].

Ackleyho funkci se podařilo úspěšně optimalizovat pomocí 1., 2. a 3. konfigurace. Obrázek 6.8 ukazuje průběh experimentu. Je zde vidět, že konfigurace 1 na dané úloze fungovala výrazně lépe než ostatní. Konfigurace 4, který měla na ostatních úlohách dobré výsledky, se na přesné optimum nedostala. Je zde vidět, že každý optimalizační problém vyžaduje pro nejlepší výsledky individuální konfiguraci.

6.2 Výsledky pro problém barvení grafů

Pro problém barvení grafů, popsaný v kapitole 3.8, byl zvolen lehce odlišný způsob experimentů. Bylo vybráno jedenáct instancí grafů[1], lišících se počtem vrcholů a hran. Tyto grafy i optimální počet barev pro jejich vybarvení jsou popsány v tabulce 6.6. Na každý z grafů byl aplikován algoritmus barvení pomocí různých konfigurací, zaznamenaných v tabulce 6.7. Algoritmus provedl pro každou instanci grafu a každou konfiguraci 100 běhů a byl nastaven tak, aby hledal řešení s optimálním počtem barev. Horší řešení než optimum je považováno za neúspěch.

[1]Zdroj a metoda použitá k vygenerování zvolených grafů jsou popsány na https://mat.tepper.cmu.edu/COLOR/instances.html#XXSGB



Obrázek 6.8: Optimalizace Ackleyho funkce pomocí specifikovaných konfigurací z tabulky 6.1.

Tabulka 6.6: Grafy barvené pomocí PSO

jméno	počet vrcholů	počet hran	optimální počet barev
anna.col	138	493	11
david.col	87	406	11
games120.col	120	638	9
huck.col	74	301	11
jean.col	80	254	10
miles250.col	128	387	8
myciel3.col	11	20	4
myciel4.col	23	71	5
myciel5.col	47	236	6
myciel6.col	95	755	7
queen5_5.col	25	160	5

Tabulka 6.7: Konfigurační parametry použité pro experimenty barvení grafů

konfigurace	setrvačnost	c1	c2	limitace rychlosti	počet částic	strategie
1	0,9	1,2	2,0	3	2000	bounce
2	0,9	1,6	0,8	3	2000	slide
3	0,9	1,6	2,0	5	2000	bounce
4	0,9	1,6	2,0	5	2000	slide

Konfigurační parametry které zde definujeme jsou stejné jako v předchozí kapitole. Navíc však přibyla strategie chování částic při kolizi s pomyslnou stěnou stavového prostoru. Význam této strategie je popsán v kapitole 5.3. Konfigurace 3 a 4 se liší pouze v této strategii a lze vidět, že při aplikaci algoritmu na některé konkrétní grafy se jejich výsledky výrazně liší, např. graf `games120`, u kterého konfigurace 3 našla optimum ve 14 %, zatímco konfigurace 4 ani v jednom běhu.

Tabulka 6.8: Výsledky barvení grafů získané pomocí konfigurace 1

graf	průměrné řešení	podíl optimálních řešení	nejlepší řešení	průměr iterací ke konvergenci
anna.col	10,45	2 %	0	154,09
david.col	7,57	4 %	0	134,74
games120.col	9,14	13 %	0	192,53
huck.col	2,77	44 %	0	118,43
jean.col	3,18	37 %	0	117,0
miles250.col	9,67	1 %	0	180,04
myciel3.col	0,0	100 %	0	1,43
myciel4.col	0,0	100 %	0	20,25
myciel5.col	0,96	76 %	0	69,23
myciel6.col	11,06	3 %	0	169,4
queen5_5.col	9,55	44 %	0	52,48

Tabulka 6.9: Výsledky barvení grafů získané pomocí konfigurace 2

graf	průměrné řešení	podíl optimálních řešení	nejlepší řešení	průměr iterací ke konvergenci
anna.col	9,26	1 %	0	213,78
david.col	7,79	0 %	4	207,47
games120.col	17,91	1 %	0	358,45
huck.col	3,87	31 %	0	216,65
jean.col	4,42	18 %	0	190,37
miles250.col	12,55	1 %	0	240,35
myciel3.col	0,0	100 %	0	1,47
myciel4.col	0,0	100 %	0	29,24
myciel5.col	0,99	76 %	0	94,7
myciel6.col	12,0	7 %	0	295,05
queen5_5.col	9,41	45 %	0	119,5

Tabulka 6.10: Výsledky barvení grafů získané pomocí konfigurace 3

graf	průměrné řešení	podíl optimálních řešení	nejlepší řešení	průměr iterací ke konvergenci
anna.col	8,92	0 %	4	267,14
david.col	6,91	1 %	0	224,72
games120.col	7,12	14 %	0	288,58
huck.col	1,65	65 %	0	196,7
jean.col	2,2	51 %	0	157,5
miles250.col	7,99	5 %	0	248,92
myciel3.col	0,0	100 %	0	1,18
myciel4.col	0,0	100 %	0	23,53
myciel5.col	0,48	88 %	0	90,28
myciel6.col	7,68	16 %	0	255,84
queen5_5.col	6,65	62 %	0	73,81

Tabulka 6.11: Výsledky barvení grafů získané pomocí konfigurace 4

graf	průměrné řešení	podíl optimálních řešení	nejlepší řešení	průměr iterací ke konvergenci
anna.col	10,06	0 %	4	289,68
david.col	7,91	2 %	0	240,5
games120.col	25,6	0 %	8	362,03
huck.col	3,09	38 %	0	199,99
jean.col	3,1	32 %	0	183,43
miles250.col	13,69	0 %	4	292,61
myciel3.col	0,0	100 %	0	1,17
myciel4.col	0,0	100 %	0	15,07
myciel5.col	0,63	85 %	0	74,75
myciel6.col	20,57	2 %	0	223,38
queen5_5.col	9,57	39 %	0	72,34

Hlavní metrika, která je u experimentů sledována, je podíl optimálních řešení. Metriky průměrné řešení a nejlepší řešení představují průměrné a nejlepší nalezené hodnoty `fitness`, nejedná se o počet barev použitý pro obarvení grafu. Pomocí nich můžeme získat informaci, jak blízko byl daný běh úspěšnému obarvení. Řešení s `fitness` = 0 představuje obarvení grafu optimálním počtem barev. Pro grafy `huck, jean, myciel3, myciel4, myciel5` a `queen5_5` dosahovaly všechny konfigurace velice dobrých výsledků. Jedná se o část sady grafů s nízkým počtem vrcholů (11 až 80). Pro složitější grafy se výkony jednotlivých konfigurací ve většině případů lišily pouze mírně. Nejlépe fungovala konfigurace 1, která dokázala optimalizovat všechny z uvedených grafů. Konfigurace, které byly použity při zde prezentovaných experimentech byly vybrány jako nejvíce výkonné ze všech testovaných konfigurací. Zajímavé je, že u problému barvení grafů ve třech ze čtyřech případů je složka `c2` větší než `c1`, zatímco na funkcích s reálnými čísly je tomu tak pouze u jedné ze čtyř

konfigurací. Sociální komponent rovnice 3.2 tedy nemá u diskrétního PSO v porovnání s originální verzí PSO tak významný vliv.

Kapitola 7

Závěr

Tato práce se zabývá vývojem knihovny využívající techniky PSO. Byl proveden průzkum aktuálního stavu knihoven v dané oblasti a na základě získaných poznatků navržena a implementována knihovna kipso. Knihovna řeší řadu problémů popsaných v kapitole 2. Díky flexibilnímu rozdělení do modulů umožňuje snadné rozšíření a využití na různé optimalizační úlohy. Podařilo se také vytvořit systém na výpočet, ukládání a interpretaci průběhu experimentů, který umožňuje snadné zhodnocení výsledků. Předností knihovny je navržené rozhraní, které je snadno srozumitelné a poskytuje pomocné nástroje pro nenáročné spouštění experimentů. Kipso je jednou z mála knihoven v javovském ekosystému s výše popsanou funkcionalitou.

Činnost implementované knihovny byla ověřena na několika netriviálních spojitých funkcích a na problému barvení grafů, využívajícím implementaci diskrétního PSO. Bylo demonstrováno využití různých konfigurací algoritmu na stejný problém a zvýrazněn vliv některých jeho složek. Pomocí knihovny bylo u všech vybraných úloh nalezeno teoretické optimum.

Projekt byl publikován jako otevřený software na portálu `github`. Plán pro jeho další vývoj spočívá primárně v implementaci dalších variací algoritmu a rozšíření. Dalším krokem může být nahrazení současného způsobu interpretace statistik plnohodnotnou aplikací s vlastním grafickým uživatelským rozhraním. Současný nástroj je vhodný pro menší projekty, ale pokud vznikne potřeba srovnávat velké množství experimentů provedených za delší úsek času, je žádoucí mít systém, který v nich umí vyhledávat a třídit. Posledním z navrhovaných rozšíření je prostá optimalizace knihovny. Původní návrh se soustředil na použitelnost na úkor rychlosti. Velká část operací může být prováděna paralelně a zvýšit rychlost experimentů. Jedná se však o netriviální zásah, který musí být dobře navržen a otestován.

Literatura

[1] Arlindo, S.; Ana, N.; Ernesto, C.: An Empirical Comparison of Particle Swarm and Predator Prey Optimisation. In *Artificial Intelligence and Cognitive Science. AICS 2002. Lecture Notes in Computer Science*, ročník 2464, editace O. M.; S. R.F.E.; R. C.; E. M.; G. N.J.L, Berlin, Heidelberg: Springer.

[2] Barnwal, A.: *Undirected graph with 5 vertices.* [Online; navštíveno 07.04.2018].
URL https://cdncontribute.geeksforgeeks.org/wp-content/uploads/undirectedgraph.png

[3] Bingham, D.: *Virtual Library of Simulation Experiments.* [Online; navštíveno 30.03.2019].
URL http://www.sfu.ca/~ssurjano/spheref.html

[4] Chen, S.; Montgomery, J.: Selection strategies for initial positions and initial velocities in multi-optima particle swarms. 07 2011.

[5] Engelbrecht, A.: *Fundamentals of Computational Swarm Intelligence.* 01 2005, ISBN 978-0-470-09191-3.

[6] Hliněný, P.: Teorie Grafů. Oct 2008.
URL https://www.fi.muni.cz/~hlineny/Vyuka/GT/Grafy-text07.pdf

[7] Husfeldt, T.: *Greedy colouring of a bipartite graph with 8 vertices.* [Online; navštíveno 01.04.2018].
URL https://en.wikipedia.org/wiki/Graph_coloring#/media/File:Greedy_colourings.svg

[8] Jiri, M.; Jaroslav, N.: *Kapitoly z diskretni matematiky.* Karolinum, 2009.

[9] Jitendra, A.; Shikha, A.: Acceleration Based Particle Swarm Optimization for Graph Coloring Problem. In *Acceleration Based Particle Swarm Optimization for Graph Coloring Problem*, 19th International Conference on Knowledge Based and Intelligent Information and Engineering Systems, Bhopal, India: Elsevier B.V., 2015.

[10] Kai, Z.; Wanying, Z.; Jun, L.; aj.: Discrete Particle Swarm Optimization Algorithm for Solving Graph Coloring Problem. In *Bio-inspired Computing: Theories and Applications*, 10th International Conference, BIC-TA 2015, Hefei, China: Springer, 2015.

[11] Kennedy, J.: Small worlds and mega-minds: effects of neighborhood topology on particle swarm performance. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, ročník 3, 1999, str. 1938 Vol. 3, doi:10.1109/CEC.1999.785509.

[12] Kennedy, J.; Eberhart, R.: Particle swarm Optimization. 12 1995, ISBN 0-7803-2768-3, s. 1942 – 1948 vol.4, doi:10.1109/ICNN.1995.488968.

[13] Kennedy, J.; Eberhart, R. C.: A discrete binary version of the particle swarm algorithm. In *1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation*, ročník 5, Oct 1997, ISSN 1062-922X, s. 4104–4108 vol.5, doi:10.1109/ICSMC.1997.637339.

[14] Khoucha, F.; Habib, M.; Benbouzid, M.; aj.: Rule-based energy management strategy optimized using PSO for fuel cell/battery electric vehicle. 01 2015.

[15] Kruger, F.; Ogier, M.; Jiménez Serrano, S.; aj.: *Generic Local Search (Memetic) Algorithm on a Single GPGPU Chip*. 01 2013, ISBN 978-3-642-37958-1.

[16] Li, L.; Huang, Z.; Liu, F.: A heuristic particle swarm optimization method for truss structures with discrete variables. *Computers & Structures*, ročník 87, č. 7, 2009: s. 435 – 443, ISSN 0045-7949, doi:https://doi.org/10.1016/j.compstruc.2009.01.004. URL http://www.sciencedirect.com/science/article/pii/S0045794909000261

[17] R. Garey, M.; Johnson, D.; J. Stockmeyer, L.: Some Simplified NP-Complete Problems. 01 1974, s. 47–63, doi:10.1145/800119.803884.

[18] Richards, M.; Ventura, D.: Choosing a starting configuration for particle swarm optimization. In *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No.04CH37541)*, ročník 3, July 2004, ISSN 1098-7576, s. 2309–2312 vol.3, doi:10.1109/IJCNN.2004.1380986.

[19] Roberge, V.; Tarbouchi, M.; Labonte, G.: Comparison of Parallel Genetic Algorithm and Particle Swarm Optimization for Real-Time UAV Path Planning. *IEEE Transactions on Industrial Informatics*, ročník 9, č. 1, Feb 2013: s. 132–141, ISSN 1551-3203, doi:10.1109/TII.2012.2198665.

[20] Robinson, J.; Rahmat-Samii, Y.: Particle swarm optimization in electromagnetics. Feb 2004, doi:10.1109/TAP.2004.823969.

[21] Shu-Chuan Chu; Yi-Tin Chen; Jiun-Huei Ho: Timetable Scheduling Using Particle Swarm Optimization. In *First International Conference on Innovative Computing, Information and Control - Volume I (ICICIC'06)*, ročník 3, Aug 2006, s. 324–327, doi:10.1109/ICICIC.2006.541.

[22] Snehal Kamalapur, S. S., Varsha Patil: Particle Swarm Optimization Performance for Unconstrained Optimization Problems. 2010.

[23] Wu, P.; Zhang, J.: Novel Particle Swarm Optimization for unconstrained problems. 2013, doi:10.1109/ccdc.2013.6560950.

[24] Yen, G. G.; Leong, W. F.: Dynamic Multiple Swarms in Multiobjective Particle Swarm Optimization. July 2009, doi:10.1109/TSMCA.2009.2013915.

[25] Zhang, Y.; Gong, D.-w.; Zhang, J.-h.: Robot path planning in uncertain environment using multi-objective particle swarm optimization. *Neurocomputing*, ročník 103, 03 2013: str. 172–185, doi:10.1016/j.neucom.2012.09.019.

Příloha A

Obsah CD

kipso/src – zdrojové kódy knihovny

kipso.jar – zkompilovaná knihovna

evaluation/model_multi_run_visualization – nástroj na interpretaci a vizualizaci výsledků

kipso/docs – dokumentace vygenerovaná ze zdrojového kódu

xhruba08_bp.pdf – vysázený text BP ve formátu PDF

data – data z experimentů prezentovaných v BP

Příloha B

Manuál

Cílem tohoto manuálu je ukázat uživateli jak lze knihovnu využít k nalezení extrémů libovolné funkce. V druhé části manuálu bude demonstrováno jak lze knihovnu rozšířit o jednoduchou funkcionalitu. Poslední část ukazuje jak lze výsledky interpretovat. Prerekvizitou této kapitoly je kapitola 5.

Pro kompilaci zdrojových kódů je používám systém *maven*. Pro sestavení knihovny stačí navigovat do adresáře se zdrojovými kódy a spustit příkaz `mvn clean install`. Pro úspěšnou kompilaci je vyžadován maven **verze 3.5.3 a vyšší** a java 8. Po dokončení kompilace bude vyprodukován artefakt v adresáři *kipso/target*s názvem kipso-*version*-jar-with-dependencies.jar. Tento artefakt lze přidat do existujícího projektu jako závislost a tím pádem v projektu využívat funkcí knihovny. Alternativní možností získání artefaktu je využití kopie na přiloženém CD.

Tento odstavec obsahuje návod k definici vlastní optimalizované úlohy. Vzorové vypracované řešení je ve zdrojových kódech knihovny v souboru *src/main/kotlin/com/ehmeed/kipso/demo/Demo01OptimizationTask.kt*. Předvedeme implementaci jednoduché funkce definované rovnicí $f(\mathbf{x}) = \cos^2(x_1) + \sin^2(x_2)$.

```
1 class Demo01Task : OptimizationTask<Double> {
2
3     override val optimizationType =
4             OptimizationTask.OptimizationType.MINIMIZE
5
6     override val dimensions = 2
7
8     override val constraints = SimpleRealOptimizationConstraints(5.0)
9
10    override fun evaluate(values: List<Double>): Fitness {
11        super.validateSize(values)
12
13        return RealFitness(
14                cos(values[0]).pow(2) + sin(values[1]).pow(2)
15        )
16    }
17 }
```

Výpis B.1: Definice optimalizované funkce

Prvním krokem je deklarace nové třídy pojmenované `Demo01Task`. Třída implementuje rozhraní `OptimizationTask` typu `Double`. Tím říkáme že vytváříme optimalizační úlohu pracující s reálnými čísly. Řádek 3 a 4 ve výpisu B.1 specifikuje typ optimalizace – cílem je minimalizovat hodnotu `fitness`. Počet dimenzí, tedy počet vstupních hodnot, je nastaven na 2. Omezení stavového prostoru je definováno na řádku 8 – zvolená implementace představuje stejné omezení ve všech dimenzích. Parametr 5.0 tedy znamená stavový prostor od [-5.0, 5.0]. Funkce `evaluate` má za úkol výpočet hodnoty `fitness`. Její parametr `values` obsahuje hodnoty řešení, které chceme ohodnotit. Nejprve tedy voláním funkce `validateSize` ověříme, že nedošlo k chybě a seznam `values` skutečně obsahuje očekávané 2 hodnoty. Následuje samotný výpočet funkce podle výše uvedené rovnice. Výsledek výpočtu je zabalen do objektu `RealFitness`, který je obálkou nad hodnotou `fitness` pro úlohy s reálnými čísly.

Tento odstavec obsahuje návod k definici vlastní ukončovací podmínky. Vzorové vypracované řešení je ve zdrojových kódech knihovny v souboru *src/main/kotlin/com/ehmeed/kipso/demo/Demo02EndCondition.kt*. Pro demonstrační účely vytvoříme ukončující podmínku, která je splněna právě tehdy, pokud všechny částice mají stejnou nebo lepší než specifikovanou hodnotu *fitness*. Ve výpisu kódu B.2 nejprve deklarujeme třídu implementující rozhraní `EndCondition` typu `Double`. Poté stačí implementovat jedinou metodu – `isFulfilled`. V její implementaci přistupujeme k populaci objektu `pso`, který byl předán jako parametr. Pro každou částici z populace ověříme, zda je její *fitness* menší než hodnota specifikovaná v konstruktoru třídy. Tím je implementace ukončující podmínky hotová.

```
1 class Demo02EndCondition(private val maxFitness: RealFitness) :
      EndCondition<Double> {
2     override fun isFulfilled(pso: Pso<Double>) =
3         pso.population.iterator().asSequence()
4                 .all { it.fitness <= maxFitness }
5 }
```

Výpis B.2: Definice ukončovací podmínky

Příklad využití právě vytvořených komponent v algoritmu je přiložen ve zdrojových kódech knihovny v souboru *src/main/kotlin/com/ehmeed/kipso/demo/Demo03RunExperiment.kt*. Tento příklad lze také spustit bez jakýkoliv úprav pomocí příkazu `java -jar kipso.jar pathToLogDirectory`. Kde proměnná `pathToLogDirectory` je adresář pro uložení dat. Výchozí nastavení je `/tmp/kipso_logs`.

Na analýzu výsledků je potřeba python verze 3.5, knihovny *pandas* a *plotly* a Jupyter notebook. Po otevření Jupyter notebooku příkazem `jupyter notebook` v kořenovém adresáři projektu navigujeme do adresáře `evaluation`. Otevřeme dokument `model_multi_run_visualization`. Následně stačí už jen přepsat parametr funkce `load_data`, který určuje zdroj načítaných dat na námi zvolený adresář (v případě ponechání původních hodnot v předchozím příkladu můžeme i zde ponechat původní hodnoty) a pomocí příkazu `Run All` spustíme analýzu dat.