Univerzita Pardubice
Fakulta informatiky

Implementace evoluční strategie v prostředí Apache Spark

Bakalářská práce

2025 Daniel Martinek

Univerzita Pardubice
Fakulta elektrotechniky a informatiky Akademický
rok: 2024/2025

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Daniel Martinek**
Osobní číslo:	**I22115**
Studijní program:	**B0688A140009 Informační technologie**
Téma práce**:**	**Implementace evoluční strategie v prostředí Apache Spark**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování

Sepiště rešerši informačních zdrojů o prostředí Apache Spark, jeho programování a evolučních algorit- mech.
V některém z podporovaných jazyků v prostředí Apache Spark (Python, Java, Scala) vytvořte efektivní paralelní implementaci optimalizačního algoritmu evoluční strategie.
Demonstrujte jeho činnost na řešení optimalizačních úloh, které jsou popsány daty uloženými v databázi dotazované prostřednictvím SparkSQL.

Rozsah pracovní zprávy: **cca 45 stran**
Rozsah grafických prací: **dle pokynů vedoucího**
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

ZELINKA, Ivan. *Umělá inteligence v problémech globální optimalizace*. Praha: BEN – technická literatura, 2002. ISBN 80-7300-069-5.
HOLUBOVÁ, Irena; KOSEK, Jiří; MINAŘÍK, Karel a NOVÁK, David. *Big Data a NoSQL databáze*. Profes!onal. Praha: Grada, 2015. ISBN 978-80-247-5466-6.

Vedoucí bakalářské práce: **doc. Ing. Tomáš Brandejský, Dr.**
Katedra softwarových technologií

Datum zadání bakalářské práce: **15. prosince 2024**
Termín odevzdání bakalářské práce: **16. května 2025**

L.S

prof. Ing. Petr Doležel, Ph.D. v.r.
děkan

Ing. Jan Panuš, Ph.D. v.r.
vedoucí kated

V Pardubicích dne 28. února 2025

Prohlašuji: Práci s názvem Implementace evoluční strategie v prostředí Apache Spark. jsem vypracoval(a) samostatně. Veškeré literární prameny a informace, které jsem v práci využil(a), jsou uvedeny v seznamu použité literatury. Byl(a) jsem seznámen(a) s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše. Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 7/2019 Pravidla pro odevzdávání, zveřejňování a formální úpravu závěrečných prací, ve znění pozdějších dodatků, bude práce zveřejněna prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne

Daniel Martinek v. r.

ANOTACE

Tato práce má za úkol seznámit s některými algoritmy patřícími do rodiny evolučních algoritmů na úloze optimalizace hodnoty funkce a jejich implementaci v prostředí Apache spark. Těmito algoritmy jsou genetický algoritmus, diferenciální evoluce a simulovaným žíháním.

KLÍČOVÁ SLOVA

optimalizace, evoluční algoritmy, Apache Spark,

ANNOTATION

The goal of this work is to show selected optimalisation algorithms from a family of evolution algorithms and their implementation in Apache Spark environment. The shown algorithms are genetic algorithm, differential evolution and simulated annealing.

KEYWORDS

Optimalisation, evolution algorithms, Apache Spark

Obsah

SEZNAM ILUSTRACÍ A TABULEK

SEZNAM ZKRATEK A ZNAČEK

1. **RDD** – Resilient Distributed Datasets
2. **DAG** – Directed Acyclic Graph
3. **JVM** – Java Virtual Machine
4. **MLlib** – Machine Learning Library
5. **GA** – Genetický algoritmus (Genetic Algorithm)
6. **ES** – Evoluční strategie (Evolution Strategy)
7. **EP** – Evoluční programování (Evolutionary Programming)
8. **GP** – Genetické programování (Genetic Programming)
9. **DE** – Diferenciální evoluce (Differential Evolution)
10. **YARN** – Yet Another Resource Negotiator
11. **SQL** – Structured Query Language
12. **S-GA** – Scalable Genetic Algorithm

TERMINOLOGIE

1. **Apache Spark:** *Open-source distribuovaný framework pro rychlé zpracování velkých dat. Využívá in-memory caching a optimalizované dotazování pro analytické úlohy.*
2. **In-memory caching:** *Technika ukládání dat do operační paměti (RAM) místo na disk, což výrazně urychluje přístup k datům a výpočty.*
3. **Resilient Distributed Datasets (RDD):** *Základní datová struktura ve Sparku, která reprezentuje neměnnou distribuovanou kolekci dat s automatickou fault tolerance.*
4. **Directed Acyclic Graph (DAG):** *Grafová reprezentace posloupnosti operací ve Sparku, umožňující optimalizaci plánování úloh a paralelní zpracování.*
5. **Spark Core:** *Základní modul Sparku poskytující funkcionalitu pro distribuované zpracování dat, správu paměti a práci s RDD.*
6. **Spark SQL:** *Modul pro práci se strukturovanými daty pomocí SQL dotazů nebo DataFrame API.*
7. **Spark Streaming:** *Nástroj pro zpracování datových proudů v reálném čase rozdělených do mikrodávek.*
8. **MLlib:** *Knihovna Sparku pro distribuované strojové učení s předpřipravenými algoritmy a nástroji.*
9. **GraphX:** *Modul pro analýzu grafů a distribuované výpočty nad grafovými strukturami.*
10. **Spark Driver:** *Hlavní proces, který koordinuje úlohy v Spark aplikaci, vytváří SparkContext a rozděluje práci mezi exekutory.*

11. **Spark Executors:** *Procesy na worker uzlech, které vykonávají přidělené úkoly a ukládají data do paměti.*
12. **Cluster Manager:** *Nástroj pro správu zdrojů v clusteru (např. Kubernetes, YARN), který přiděluje výpočetní kapacitu Spark aplikacím.*
13. **Worker Nodes:** *Jednotlivé servery v clusteru, kde běží exekutoři a zpracovávají data.*
14. **Big Data:** *Rozsáhlé a komplexní datové sady, charakterizované 4V: objemem (Volume), rychlostí (Velocity), různorodostí (Variety) a důvěryhodností (Veracity).*
15. **Evoluční algoritmy:** *Optimalizační metody inspirované biologickou evolucí, které kombinují selekci, mutaci a křížení k hledání řešení.*
16. **Genetický algoritmus (GA):** *Typ evolučního algoritmu pracující s binárními řetězci, který simuluje přirozený výběr pomocí operací křížení a mutace.*
17. **Evoluční strategie (ES):** *Algoritmus pro spojité optimalizační problémy, který využívá Gaussovskou mutaci a parametry strategie.*
18. **Evoluční programování (EP):** *Metoda zaměřená na učení chování pomocí mutací bez křížení, často používaná pro optimalizaci stavových automatů.*
19. **Genetické programování (GP):** *Evoluční přístup pro automatické generování programů reprezentovaných stromovými strukturami.*
20. **Diferenciální evoluce (DE):** *Optimalizační algoritmus pro spojité problémy, který kombinuje rozdíly mezi řešeními k vytváření nových kandidátů.*

21. **Simulované žíhání:** *Heuristická metoda inspirovaná žíháním kovů, která přijímá horší řešení s klesající pravděpodobností pro únik z lokálních minim.*
22. **Metropolisův algoritmus:** *Základ simulovaného žíhání, který definuje pravidla pro přijímání nových řešení na základě energetické změny.*
23. **Hadoop MapReduce:** Starší model distribuovaného zpracování dat založený na dávkovém zpracování s častým zápisem na disk.
24. **In-memory computing:** *Zpracování dat přímo v paměti (bez ukládání na disk), klíčové pro rychlost Apache Spark.*
25. **Kubernetes:** *Open-source systém pro automatizaci správy kontejnerových aplikací v clusterech, používaný jako orchestrátor pro Spark.*
26. **Mesos:** *Nástroj pro správu clusterových zdrojů, který umožňuje efektivní běh distribuovaných aplikací včetně Sparku.*
27. **YARN:** *Modul Hadoopu pro správu zdrojů a plánování úloh v distribuovaném prostředí.*
28. **Scalable Genetic Algorithm (S-GA):** *Implementace genetického algoritmu optimalizovaná pro běh v distribuovaném prostředí Apache Spark.*
29. **Batch processing:** *Zpracování velkých objemů dat v dávkách, typické pro tradiční systémy jako Hadoop MapReduce.*
30. **Fault tolerance:** *Schopnost systému pokračovat v práci i při selhání části komponent, zajištěná ve Sparku pomocí RDD.*
31. **Populace: S**oubor jedinců čítající zpravidla více než jednoho jedince

32. **Jedinec**: Označení pro prvek účastnící se algoritmu (také označován jako chromozom) Většinou je reprezentován řetězcem, polem hodnot anebo stromem

33. **Gen:** Podle implementace a typu algoritmu se může jednat o znak v řetězci, skupinu znaků nebo číslo. Gen je nejmenší jednotka informace, která má v algoritmu smysl.

34. **Generace:** **P**opulace v určitý časový okamžik po dokončení kroků algoritmu. N-tá generace znamená populace po proběhnutí N opakování algoritmu.

35. **Rodič**: Jeden z jedinců, kteří jsou součástí křížení. Křížení zpravidla používá dva rodiče.

36. **Potomek**: Výsledek křížení a mutace.

ÚVOD

Tato práce je z větší části zaměřena na optimalizační problém a evoluční algoritmy. Zejména tedy na genetický a diferenciální algoritmus. Práce bude obsahovat i související oblasti jako jsou big data a použití technologie Apache Spark.

1. Použité technologie

1.1. Apache Spark

1.1.1. Použití a historie

Apache Spark je open-source, distribuovaný výpočetní systém / framework určený pro zpracování velkých objemů dat s mírou škálovatelnosti a s vysokou výkonností. (1). Využívá in-memory caching a optimalizované provádění dotazů pro rychlé analytické dotazy na data jakékoliv velikosti.

Vznikl jako reakce na potřebu rychlejšího a efektivnějšího zpracování velkých dat. Původní systémy, jako je Hadoop MapReduce, byly pomalé kvůli nutnosti neustálého zapisování a čtení dat z disku. Spark byl navržen tak, aby využíval právě in-memory caching, což výrazně zrychluje zpracování dat (2).

Začátky Apache Spark sahají do roku 2009, kdy byl vyvinut v rámci projektu AMPLab na University of California, Berkeley. Jeho kód byl otevřen v roce 2010 a na oficiálním GitHubu byl zveřejněn 30. 3. 2010 uživatelem mateiz (vlastním jménem Matei Zaharia) (3), který se věnoval této aplikaci až do 20. 6. 2016. Spark se stal open-source projektem pod Apache Software Foundation v roce 2013 (4).

Je napsán převážně v programovacím jazyce Scala s některými částmi, které jsou napsány v Pythonu a Javě z důvodu kompatibility s těmito jazyky. Přidává datové typy/struktury pro jednoduchou a účinnou paralelizaci (RDD a Dataframe). Zdrojový kód je dostupný na platformě GitHub.

Jeho hlavním cílem bylo zlepšit výkon a jednoduchost použití pro různé typy analytických úloh, včetně batch processingu, interaktivních dotazů, real-time analytiky a strojového učení

Jeho hlavní předností je unifikovaný framework, který integruje batch processing, streamování, SQL dotazy, strojové učení a analýzu grafů.

2. Architektura

2.1. Principy architektury Apache Spark

Architektura je navržena tak, aby byla efektivní a škálovatelná. Zde je základní přehled jeho struktury (5) **:**

Spark Driver:

Řídí celý proces zpracování dat. Driver vytváří SparkContext, který je vstupním bodem pro Spark aplikace.

Driver také rozděluje úlohy na menší části, které jsou následně zpracovány na různých uzlech.

Spark Executor:

Je zodpovědný za vykonávání úloh. Každý executor běží na uzlu v clusteru a provádí úlohy přidělené driverem.

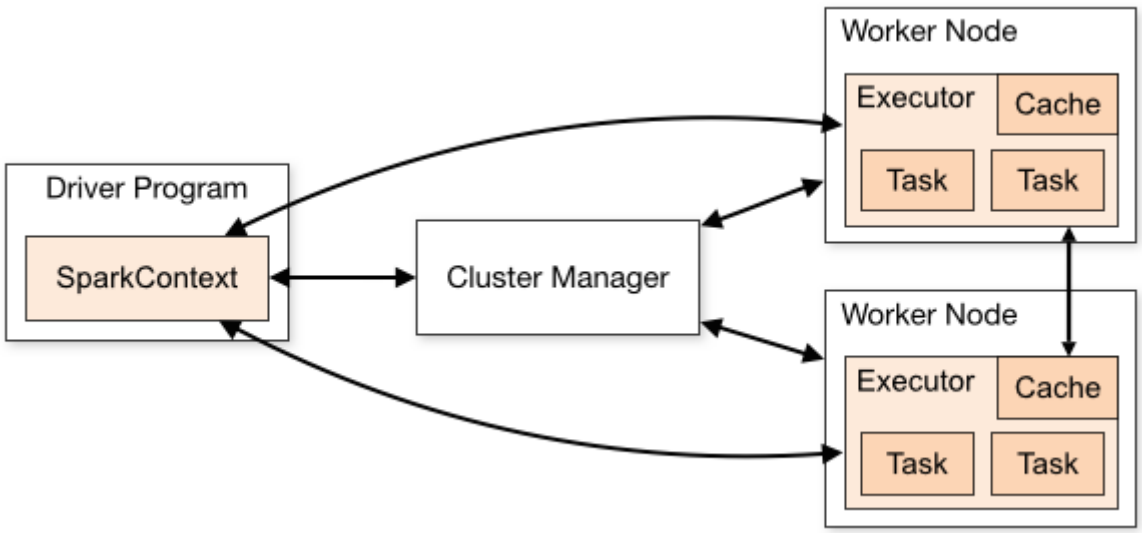
Ukládají také data do paměti pro rychlý přístup.

Cluster Manager:

Spravuje zdroje v clusteru. Může to být Standalone manager, Hadoop YARN, Kubernetes a jiní.

Worker Node:

Jedná se o uzel v clusteru, na kterém běží executory a zpracovávají se data.



Obrázek 1 - Diagram architektury Apache Spark (6)

Spark aplikace běží v rámci driver procesu (obsahujícího SparkContext), který komunikuje s cluster manažerem. Cluster manager přidělí pracovní uzly (executors), na kterých jsou pak spouštěny paralelní úlohy.

Spark tedy může být nainstalován v lokálním režimu, kdy celý běží na jednom počítači v jedné JVM. V tomto případě se v rámci této jedné JVM používají vlákna, aby simulovala Spark exekutory. Tento režim je vhodný zejména na vývoj a testování, protože nedisponuje možnostmi škálování a nemá mechanismy fault tolerance. Jako úložiště je v tomto případě používán především lokální souborový systém i když možnosti pro připojení vzdálených úložišť jsou k dispozici, např. HDFS.

Další způsob instalace je standalone režim. Jde o dedikovaný Spark cluster bez externích závislostí a určený pouze pro Spark masters/workers. Ty běží na samostatných JVM ať již na jednom serveru nebo distribuovaně. Tato konfigurace je statická a nelze ji automaticky nasadit. Tento režim se používá pro produkční nasazení a umožňuje obnovu worker node. Tento režim je optimalizován pro distribuovaná úložiště.

Poslední a nejrozšířenější možností nasazení je využití orchestrátoru. Externí orchestrátor se stará o alokování zdrojů, škálování i odolnost proti výpadkům a chybám ve výpočtu. Zdroje jsou alokované dynamicky. Stejný orchestrátor se může současně krom Sparku starat i o jiné typy nodů. Krom Hadoop YARN může být jako orchestrátor použit i Kubernetes (2). (Podpora Apache Mesos, byla ukončena ve Spark 4.0)

Každá Spark aplikace je plně oddělená od ostatních. To znamená, že dostane svůj spark driver. Tento ovladač vytvoří pro tuto aplikaci instanci objektu sparkContext. Aplikace jsou tímto od sebe kompletně odděleny a nemají možnost spolu komunikovat. Každá aplikace si drží svá data a sdílení je možné pouze přes externí úložiště. Nemožnost komunikace se může zdát jako nevýhoda, ale zvyšuje to bezpečnost aplikací a chrání to tyto aplikace před pády ostatních aplikací. (6).

2.2. Komponenty

Spark obsahuje tyto komponenty, které umožňují práci s daty v různých formátech a pro různé typy úloh:

- **Spark Core**: Základní modul, který poskytuje základní funkce pro distribuované zpracování dat, správu paměti a fault tolerance pomocí RDD (Resilient Distributed Datasets)
- **Spark SQL**: Modul pro práci se strukturovanými daty přes DataFramy a SQL dotazy
- **Spark Streaming**: Zpracování datových proudů v mikrodávkách. (Používáno pro výpočty v reálném čase jako je například zpracovávání logů běžícího systému)
- **MLlib**: Knihovna pro distribuované strojové učení.
- **GraphX**: Analýza grafů pomocí distribuovaných algoritmů.

Spark podporuje různé programovací jazyky, jako jsou Python, SQL, Scala, Java a R, a umožňuje opakované použití kódu napříč různými úlohami, včetně batch processingu, interaktivních dotazů, real-time analýzy, strojového učení a grafového zpracování

3. Prostředky procesu zpracování dat:

3.1. RDD

RDD – Resilient Distributed Datasets je jednou z možností, jak lze komunikovat s Apache Spark. Jedná se o základní datovou strukturu a vnitřní reprezentaci dat ve Sparku, která také zajišťuje odolnost vůči chybám a umožňuje paralelní zpracování dat. Je to neměnná (immutable) a rozdělená (distributed) kolekce objektů (partitions), které mohou být paralelně zpracovávány napříč uzly clusteru.

Data v RDD nelze měnit, tím se předejde některým nástrahám více vláknového zpracování jako je změna dat jedním vláknem, když jiné vlákno tyto data čte. Originální data, informace o transformacích a akcích a rozdělení na nodech je důležité uchovat, dokud se neprovedou všechny zamýšlené akce, aby šlo dopočítat chybějící data, pokud node selže.

Spark používá RDD pro zpracování dat a jejich distribuci v clusteru. RDD má dva druhy operací, a to transformace a akce. Transformace jsou úkony, které dělají z RDD jiné RDD. Akce jsou úkony, které nezachovávají RDD jako obalující typ a vrací výsledek operace ovladači. Jeho použití v Javě je podobné jako u datového proudu (stream). RDD používá odložené zpracování, což je jednou z vlastností Apache Spark: Operace jsou prováděny až, když jsou potřeba výsledky. Toho je dosaženo například voláním funkce reduce() nebo coalesce(). (7)

To může v určitých případech snížit výpočetní náročnost tím, že se transformace, které nejsou potřeba provést pro dokončení aktuální akce neprovedou. Které transformace se provedou se rozhoduje podle acyklického grafu (DAG - Directed Acyclic Graph), se směrovým označením hran a která akce se má provést (8). RDD si pamatuje V DAG posloupnost transformací, které vedly k jeho vytvoření. Pokud některý uzel v clusteru selže a ztratí data, Spark dokáže automaticky přepočítat ztracené části dat z původních dat pomocí tohoto grafu, aniž by se musela data načítat znovu od začátku.

3.2. DAG (Directed Acyclic Graph)

Spark vytváří DAG pro každou aplikaci, který reprezentuje posloupnost operací na RDD. DAG umožňuje optimalizaci a efektivní plánování úloh. Jedná se o reprezentace výpočtu. Uzel v DAG reprezentuje jedno RDD, Hrana představuje transformaci.

3.3. Dataframe

Dataframe je dvojrozměrná datová struktura složená ze sloupců. K RDD přidává schema. Každý sloupec má název a datový typ. Tímto se podobá tabulce v SQL databázi. Na rozdíl od tabulky tento objekt ale nelze změnit. Pouze lze vytvořit kopii obsahující změny. Dataframe umožňuje vysokoúrovňovou práci nad daty. Například načítání dat ze souboru nebo volání dotazů sparkSQL. Je to jeden ze způsobů, kterým spolu komunikuje aplikace se Spark driverem (9) (10).

3.4. Dataset

Dataset rozšiřuje DataFrame o typovou bezpečnost (pro Scala/Java). K datům se již nepřistupuje pod obecným typem Row ale data jsou mapovaná na konkrétní třídy. To umožňuje detekci chyb již v době kompilace.

3.5. Lambda funkce

Lambda funkce jsou součástí lambda kalkulu, což je součást matematiky zabývající se pouze funkcemi. Toto bylo nejspíše inspirací pro lambda funkce v programovacích jazycích.

Lambda funkce (také známé jako anonymní funkce) jsou nepojmenované funkce často menšího rozsahu, které se většinou používají jako argumenty pro jiné funkce. Lambda funkce v programovacích jazycích mají většinou tři části, které byly převzaty z lambda kalkulu. První část označuje, které proměnné se ve funkci použijí. U těchto proměnných mohou být omezení (Objekt implementuje rozhraní, jedná se o číslo...). Některé programovací jazyky jako například Java má omezení pro všechny lambda funkce. V Javě musí být tyto (zachycené)

proměnné konstanty nebo alespoň lokální konstanty. To znamená, že se v tomto bloku kódu nemění a jsou pouze v něm definovány. Druhou částí je oddělovač. Třetí část je již samotná funkce. U některých jazyků, jako je například c++, se oddělovač vypouští

Mechanismus, jak lambda funkce fungují jsou závislé na konkrétní implementaci v daném programovacím jazyce.

Například v Javě jsou lambda funkce instancemi funkčního interface. Toto interface má jednu abstraktní metodu. Tyto interface často používají generické typy. V Javě se často používají ve streamech a řazení (11).

Lambda funkce jsou často používané při práci s Apache Spark, zejména v kontextu programování Spark aplikací v jazycích jako Python (PySpark), Scala a Java. Často jsou používány při definování transformací. Lambda funkce umožňuje definovat funkci přímo na místě, bez nutnosti ji explicitně pojmenovávat a definovat jinde v kódu.

4. Postup výpočtu

Apache Spark používá několik kroků k rozdělení dat mezi jednotlivé workery a následnému shromáždění výsledků. Spark Driver (hlavní program, který spouští SparkContext) je zodpovědný za:

- Budování DAGu z RDD transformací.
- Rozdělení DAGu na fáze.
- Předložení těchto fází TaskScheduleru.

TaskScheduler pak ve spolupráci s Cluster Managerem (např. YARN, Mesos, Standalone) přiděluje úkoly jednotlivým workerům (executorům) v clusteru.

Každý úkol zpracovává jeden oddíl (partition) RDD. Takže v době běhu je konkrétní oddíl zpracováván konkrétním workerem. Pokud worker přestane pracovat správně, Spark Driver přepočítá ztracené oddíly na jiném workeru.

Zpracování dat obsahuje:

- **Transformace**: Operace jako map, filter, a join jsou aplikovány na jednotlivé části RDD. Tyto operace jsou "lazy", což znamená, že se neprovádějí okamžitě, ale jsou zaznamenány jako plán pro pozdější provedení.
- **Akce**: Operace jako count, collect, a save jsou "akcemi", které spouštějí provedení všech předchozích transformací. Akce způsobí, že Spark začne zpracovávat data a vracet výsledky

Shromáždění výsledků:

- Shuffle: Během operací jako join nebo groupByKey může být nutné přerozdělit data mezi nody. Tento proces se nazývá shuffle a zahrnuje přenos dat mezi nody, aby se zajistilo, že všechny relevantní části dat jsou na správných místech pro další zpracování.

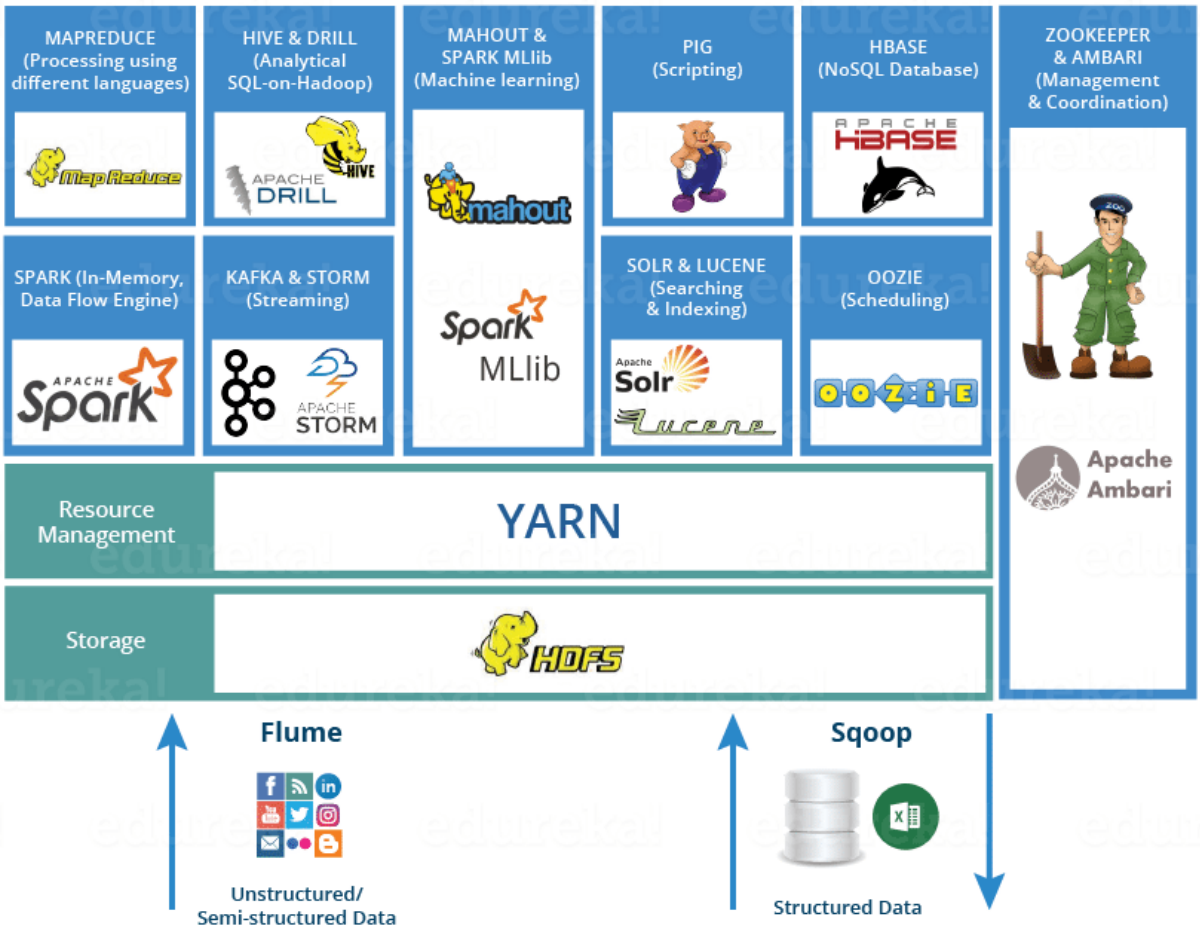
- Konečné shromáždění: Po dokončení všech transformací a akcí Spark shromáždí výsledky z jednotlivých nodů a vrátí je uživateli nebo uloží do určeného úložiště

(12), (13), (14)

5. Apache Hadoop ve vztahu k Spark

5.1. Princip a historie

Apache spark nevznikl samostatně, jak se může zdát z předchozí kapitoly. Apache spark staví ve vysoké míře na Apache hadoop. Apache spark lze bez problému nastavit, aby fungoval jako součást Apache Hadoop ekosystému, ve kterém nahrazuje Hadoop MapReduce.



Obrázek 2 - Ekosystém Apache Hadoop (15)

Apache Hadoop je distribuovaný nástroj pro ukládání a provádění výpočtů nad velkými daty (Big Data). Jeho výhodami od předešlých nástrojů jsou flexibilita, odolnost a škálovatelnost. Škálovatelnost je docílena schopností nástroje rozdělit velké úlohy na menší a ty paralelně provést na různých nodech. Odolnost proti výpadku je řešena použitím vlastního souborového systému, který dovoluje replikaci dat na různých nodech systému omezujících ztrátu dat. Hadoop je

flexibilní a umožňuje ukládat a zpracovávat jak nestrukturovaná, tak i polostrukturovaná a strukturovaná data (15) (16).

Jeho vývoj začal v roce 2003 s dvěma lidmi, a to Doug Cutting a Mike Cafarella poté, co se inspirovali projektem MapReduce od Googlu. Tehdy byl součástí projektu Nutch, nástroje na hromadné procházení webu (web crawler). Od toho se oddělil o několik let později. A v dubnu 2006 byl uvolněn release Hadoop 0.1.0 V roce 2007 byl napsán první design dokument o Hadoop Distributed File Systém (17). Je napsán převážně v Javě s částmi psanými v C.

Hadoop má také nevýhody. Mezi tyto nevýhody patří složitost nasazení, údržby a nároky na hardware. Z důvodu nutnosti vysokého počtu diskových IO operací potřebných pro výpočet je neefektivní a nehodí se pro operace v reálném čase. Pro škálování používá vlastní orchestrátor YARN (18).

Hadoop MapReduce byl překonán v některých případech Sparkem. Prvním z nich je rychlost. Hadoop ukládá mezivýsledky operací na souborový systém což zabraňuje jejich ztrátě při havárii softwaru nebo hardwaru za cenu vysokého počtu pomalých IO operací (Moderní SSD disky mají odezvu 1-10 milisekund). Spark mezitím udržuje mezivýsledky v RAM, (která zvládne miliardy dotazů za sekundu s odezvou v desítkách nanosekund). To může vést ke ztrátě mezivýsledků při výpadku nodu.

Apache Spark má i další výhody. V Apache Spark lze použít zrychlení výpočtů minimalizací počtu vstupně-výstupních operací použitím RAM cache. Její použití znamená udržování vstupních dat v paměti RAM po skončení úlohy, což umožňuje jejich znovupoužití v dalších výpočtech a tím pádem eliminuje nutnost čekání na úložiště (19)

5.2. Zabezpečení v Hadoop a Spark

Hadoop si stále zachovává silné schopnosti v oblasti zabezpečení, zejména pro distribuovaný souborový systém HDFS a komponenty jako YARN. Hadoop

podporuje uživatelské účty a kontrolu přístupu (19) (20). V základním nastavení může být zabezpečení vypnuté, ale pro produkční prostředí se obvykle používá Kerberos pro silnou autentizaci uživatelů a služeb. Kerberos se stará o ověření identity, zatímco autorizace (co může ověřený uživatel dělat) je řešena jinými systémy, jako jsou Apache Sentry nebo Ranger (pro kontrolu přístupu na úrovni HDFS, Hive, atd.). Hadoop File System (HDFS) podporuje šifrování dat v klidu (Data at Rest Encryption) a také šifrování dat v pohybu (Data in Transit Encryption) mezi uzly. Šifrování komunikace mezi uživatelem a Hadoop clusterem je možné, například při použití HTTPS pro webové rozhraní.

Apache Spark má vlastní mechanismy zabezpečení, které se primárně zaměřují na komunikaci v rámci clusteru a autentizaci aplikací. Spark ve svém samostatném režimu (Standalone mode) nepodporuje nativně robustní uživatelskou autentizaci a autorizaci pro přístup k aplikacím a datům jako v tradičních víceuživatelských systémech. Zabezpečení komunikace mezi uzly (driver a executory) je řešeno pomocí sdíleného secretu (předplacený klíč). Pro automatickou a bezpečnou distribuci tohoto secretu mezi uzly a pro podporu víceuživatelských systémů je obvykle nutné Spark provozovat na orchestrátoru. Některé komponenty Sparku, jako je REST API server, historicky neposkytovaly robustní vestavěné zabezpečení a vyžadují externí opatření (např. síťové segmentace, firewall) k omezení přístupu (21).

YARN, orchestrátor nativně používaný Hadoopem, umožňuje Sparku používat jeden secret pro jednu aplikaci a umožňuje jeho automatickou generaci. O tu se stará samotný Spark, zatímco YARN se stará o jeho distribuci, za pomoci mezi procesové komunikace přes síťové sockety.

Kubernetes pracuje se secret jinak. Secret je stejně jako při použití YARNu generovány Sparkem, ale distribuce do instancí aplikace probíhá přes proměnné prostředí, kde je nutné zabezpečit, aby ke clusteru neměl přístup někdo nepovolaný (21).

6. MapReduce

MapReduce je z hlediska programátora ze dvou hlavních upravitelných činností, které jsou Map a Reduce. Map má za úkol rozdělit úkol na menší části a namapovat je na výpočetní jednotky pro zpracování a Reduce tyto části spojí do celku a redukuje množství dat. MapReduce joby typicky čtou vstupní data z HDFS a zapisují výsledky zpět na HDFS. MapReduce vyvinuli Jeffery Dean a Sanjay Ghemawat v roce 2004, když pracovali pro google (22).

MapReduce pracuje s daty, které rozdělí mezi nody (pokud se jedná o dvojici, tak se použije první hodnota jako klíč, pokud ne, klíč se vytvoří) a výstup tohoto kroku obsahuje dvojici klíče a hodnoty. Tyto klíče nemusí být jedinečné a mohou být vytvořeny funkcí map. Poté jsou klíče dále zpracovávány za pomoci map.

Lépe to půjde ukázat na příkladu. Pokud například chceme vytvořit graf počtu událostí podle hodiny a na vstupu máme například CSV s časovým razítkem a názvem události, tak nejdříve systém rozdělí data na části, které se poté dají funkci Map. Nad konkrétním rozdělením nemá programátor větší kontrolu. Může pouze stanovit jaká jsou vstupní data a podle toho je systém rozdělí. V tomto případě jde o soubor CSV, kde je nutno dělit soubour na řádky.

Map je použita na vytvoření podmínek pro část Reduce. Vstupem do Map jsou jednotlivé řádky CSV souboru, každý řádek se zpracovává samostatně. Z tohoto řádku se stane přeměnou například dvojice s klíčem událost-hodina a hodnotou 1. Příkladem této přeměny je nejprve rozdělení CSV na 2 pole, z druhého (časové razítko) se získá hodina. To se spojí s prvním polem a tímto se z textového klíče stane klíč typu dvojice, kde první hodnota je text a druhá hodnota je celé číslo. Hodnota je nastavena na 1. Protože část map pracuje se záznamy izolovaně, lze na node využít další rozdělení systémem, aby byla použita všechna procesorová jádra.

Nad dalším krokem nemá programátor žádnou kontrolu. Tento krok má účel zrychlení části Reduce tím, že shromáždí stejné klíče na jednom uzlu. Tím se sníží nutnost části Reduce číst z různých uzlů při zpracování klíče, což může být pomalé.

Posledním krokem, nad kterým má programátor kontroulu je Reduce. Reduce je jediný krok, který pracuje nad více záznamy současně a je pod kontrolou programátora. Cílem tohoto kroku je spojení výsledků z map nebo použití funkce přes více klíčů. V tomto příkladě je Reduce použita pro spočítání počtu klíčů sečtením jejich hodnot a odstranění duplikací klíče.

Posledním krokem je spojení dat z jednotlivých operací Reduce do jednoho celku a uložení výsledku (23).

7. Big data

Big data je pojem, který není jasně ohraničený. Big data jsou datové sady, které jsou příliš velké nebo složité na to, aby je bylo možné zpracovat tradičními metodami zpracování dat. Vznikly díky rychlému nárůstu množství dat generovaných různými zdroji, jako jsou mobilní zařízení, senzory internetu věcí, kamery, mikrofony, RFID čtečky a bezdrátové senzorové sítě.

Big data se od běžných dat liší několika klíčovými charakteristikami, kterým se šasto říká 5V:

Rozsah (Volume): Velké množství často nestrukturovaných dat, často v řádu terabajtů nebo petabajtů.

- Rychlost (Velocity): Jak rychle data přicházejí a jak je na ně nutné reagovat, často v reálném čase.
- Různorodost (Variety): Data mohou být strukturovaná, nestrukturovaná nebo polostrukturovaná a mohou mít různé formy, jako jsou texty, obrázky, videa.
- Důvěryhodnost (Veracity): Kvalita a spolehlivost dat, která může být proměnlivá
- Cena (Value): Data mohou zajistit konkurenční výhodu a tím pádem mají cenu.

Přístup k big datům se liší od tradičního zpracování dat v několika ohledech. Big data často vyžadují použití distribuovaných systémů, jako je Apache Hadoop nebo Apache Spark, které umožňují paralelní zpracování dat na více uzlech. Vyžadují použití pokročilých analytických metod, jako je strojové učení, prediktivní analýza a analýza chování uživatelů. Big data přinášejí nové možnosti pro analýzu a využití dat, ale také výzvy v oblasti zpracování, ukládání a ochrany dat (24) (25).

8. Algoritmy

8.1. Evoluční algoritmy

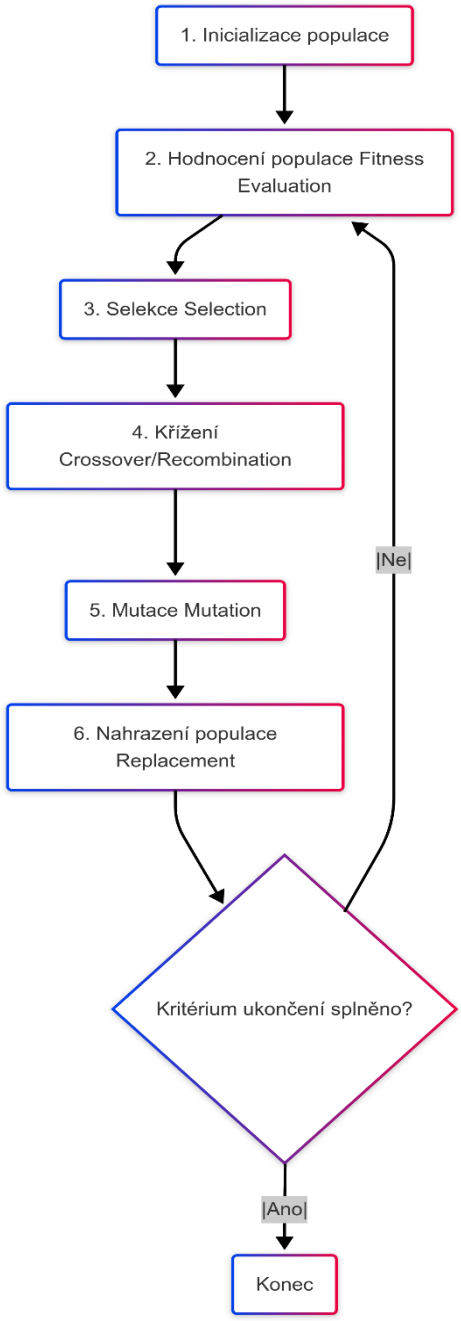
Evoluční algoritmy jsou heuristické optimalizační metody inspirované přirozeným výběrem živočišných druhů. (26) (27). Na počátku tedy byla Darwinova myšlenka zobecněná na hledání řešení úlohy – výběrem, mutací, křížením. Tyto algoritmy nepotřebují, aby funkce, kterou optimalizují měla derivaci. Stačí pouze, aby tato funkce měla hodnotu ve všech bodech, kam se mohou jedinci dostat. Evoluční algoritmy se začaly vyvíjet nezávisle ve třech liniích v 60. a 70. letech

- John_Holland (28) (29) Genetický algoritmus
- Ingo Rechenberg, Hans-Paul Schwefel (30) Evoluční strategie
- Lawrence J. Fogel (31) Evoluční programování

Tyto linie se v 90 letech sjednotily pod pojem evoluční algoritmy. (32) Evoluční algoritmy jsou často používány k řešení optimalizačních a vyhledávacích problémů, kde tradiční metody selhávají. Příklady aplikací zahrnují například optimalizaci rozhodovacích stromů, optimalizaci hyperparametrů (strojové učení) a kauzální inferenci (proces určování nezávislého, skutečného účinku určitého jevu, který je součástí většího systému)

Obecně lze tedy evoluční algoritmus popsat těmito kroky (33):

1. **Inicializace populace:** Vytvoření počáteční sady řešení (jedinců) pro daný problém. Tato populace může být generována náhodně nebo na základě nějaké heuristiky.
2. **Hodnocení populace (Fitness Evaluation):** Pro každého jedince v populaci se vypočítá jeho hodnota (fitness), která určuje, jak dobře dané řešení odpovídá cíli problému.
3. **Selekce (Selection):** Na základě jejich fitness hodnot se z aktuální populace vyberou jedinci, kteří budou použiti k vytvoření nové populace (potomků). Jedinci s vyšší fitness mají obvykle větší šanci být vybráni.
4. **Křížení (Crossover/Recombination):** Vybraní rodiče jsou zkombinováni, aby vytvořili nové potomky. Cílem je přenést a zkombinovat dobré vlastnosti rodičů do nových řešení.



Obrázek 3 - Diagram evolučního algoritmu

5. **Mutace (Mutation):** Na malé části nově vytvořených potomků dochází k náhodným změnám (mutacím). To pomáhá zavádět do populace novou genetickou informaci a zabraňuje uvíznutí algoritmu v lokálním optimu.
6. **Nahrazení populace (Replacement):** Nově vytvořená populace potomků nahradí původní populaci. Existují různé strategie nahrazování (např. nahradí se celá populace, nebo se nahradí jen nejhorší jedinci).

7. **Kritérium ukončení (Termination Condition):** Proces se opakuje (kroky 2-6) dokud není splněna nějaká podmínka pro ukončení algoritmu. Mezi typické podmínky patří dosažení určitého počtu generací, nalezení řešení s dostatečnou fitness, nebo dosažení stavu, kdy se populace již výrazně nezlepšuje.

Existuje více způsobů implementace evolučních algoritmů. Jde například o:

- Genetický algoritmus (GA)
- Evoluční strategie (ES)
- Evoluční programování (EP)
- Genetické programování (GP)
- Diferenciální evoluce (DE)

Tyto implementace se liší ve způsobu reprezentace optimalizované sady vlastností - „genetického kódu“, způsobem vyhodnocení úspěšnosti a výběrem do dalšího pokolení a samozřejmě též křížením a mutací. Tyto rozdíly v implementaci pak určují vhodnost daného evolučního algoritmu k řešení specifické sady úloh.

Tabulka 1 - ostatní algoritmy z rodiny evolučních algoritmů

Typ	Reprezentace	Křížení	Mutace	Hlavní použití
GA	Obvykle binární řetězce (např. 101010) nebo vylepšené varianty (reálné hodnoty, permutace).	Hlavní operátor – např. jednobodové, dvoubodové, uniformní crossover.	Náhodná změna jednotlivých bitů nebo hodnot.	Optimalizace problémů s diskrétními řešeními

Typ	Reprezentace	Křížení	Mutace	Hlavní použití
ES	Typicky vektory reálných čísel (např. [0.5, -2.3, 1.1]) + strategie parametry (např. mutační rozptyl).	Volitelný, méně důležitý než mutace.	Gaussovský šum– přidání náhodné hodnoty s určitým rozptylem.	Nepřetržité optimalizační problémy (např. strojové učení, inženýrské návrhy).
EP	Obvykle reálná čísla nebo stavové automaty, někdy i komplexní struktury.	Nepoužívá se – pouze mutace.	Náhodné změny s Gaussovským nebo jiným šumem.	Učení chování, např. učení agentů (řízení, rozhodování), optimalizace.
GP	Stromy – každý jedinec je program nebo výraz (např. matematický vzorec, IF struktura).	Výměna podstromů mezi rodiči.	Změna části stromu (např. uzlu, podvýrazu).	Automatická tvorba programů, predikce, symbolická regrese.
DE	Vektory reálných čísel (např. [1.2, 0.7]).	Binární/uniformní kombinace mutovaného a původního vektoru.	Diferenční mutace	Spojité vysoce-dimenzionální problémy: Fyzikální modelování,

Typ	Reprezentace	Křížení	Mutace	Hlavní použití
				optimalizace nákladů.

Ve všech případech jde o optimalizaci vlastností, hledání určitého extrému funkce. Jde o paralelní prohledávání prostoru řešení, protože algoritmy pracují s celou populací jedinců. Důležitou roli zde hraje kvalitní generátor náhodných čísel (s definovaným rozložením a případně vývojem), který ovlivňuje stochastické prvky algoritmu.

8.2. Genetický algoritmus

Genetický algoritmus je nejznámější a nejčastěji používaná forma evolučních algoritmů. Patří mezi první typy EA inspirované přírodní evolucí, a to hlavně díky práci Johna H. Hollanda v 60. a 70. letech, popsané v jeho knize Adaptation in Natural and Artificial Systems z roku 1975. Využívá evoluční mechanismy selekce, křížení (crossover) a mutace pro iterativní zlepšování populace možných řešení.

Tento algoritmus stejně jako ostatní evoluční algoritmy požaduje po účelové funkci pouze aby měla hodnotu, algoritmus nevyužívá spojitost ani diferencovatelnost této funkce. Tento algoritmus má vysokou odolnost vůči uváznutí v lokálních extrémech. (34)

Tento algoritmus se používá kroky totožné s evolučním algoritmem a to:

1. Generaci populace
2. Ohodnocení populace
3. Výběr nejlepších jedinců podle ohodnocení
4. Kombinace
5. Mutace

6. Kroky 2-5 se opakují, dokud se nedosáhne požadované přesnosti nebo počtu opakování
7. Ohodnocení populace

Jako výsledek se bere jedinec s nejlepším ohodnocením. V mém programu jsem mutaci jako samostatný krok nevyužil, jelikož výsledné ohodnocení nejlepšího jedince při populaci o velikosti 1024 se jejím odebráním jako samostatného kroku zlepšilo o tři řády. Tento krok jsem sloučil s krokem kombinace.

Pro tento algoritmus existuje úprava s názvem Scalable Genetic Algorithm (S-GA), která je speciálně navržena pro Apache spark. (35)

8.3. Diferenciální evoluce

Diferenciální evoluce je další z řady evolučních algoritmů. Tento algoritmus popsali Ken Price a Rainer Storm v roce 1995. Tento algoritmus vznikl z algoritmu genetického žíhání. Ten prošel několika změnami. První změnou byla změna typu genomu z binárního řetězce na vektorovou reprezentaci. Následovaly další změny jako vytváření potomků za pomoci šumového vektoru a způsob začlenění potomků do populace. Tento algoritmus poté prošel dalšími změnami, než se jeho podoba změnila do té dnešní. (36)

Průběh tohoto algoritmu je následující (všechny body kromě bodu 1 a bodu X jsou provedeny pro všechny jedince):

1. Generace počáteční populace
2. Ohodnocení jedince
3. Výběr tří různých náhodných rodičů a vytvoření šumového vektoru
4. Vytvoření potomka za pomoci čtvrtého rodiče a šumového vektoru
5. Ohodnocení potomka
6. Porovnání čtvrtého rodiče a potomka – lepší bude použit
7. Opakovat 3-6 dokud se nedosáhne ukončovací podmínky

8.4. Simulované žíhání

Simulované žíhání (anglicky simulated annealing) je heuristická optimalizační metoda inspirovaná procesem žíhání kovových slitin. (37) (38) (39)

Algoritmus simulovaného žíhání byl popsán v roce 1983 autory Scott Kirkpatrick, C. D. Gelatt a M. P. Vecchi. Inspirací pro ně byl Metropolisův algoritmus (1953) z oblasti statistické fyziky, který simuluje termodynamické procesy. Ten poskytl matematický rámec pro přijímání "horších" řešení s určitou pravděpodobností.

Tento algoritmus je navržen, aby napodoboval fyzikální proces, při kterém je materiál zahřát na vysokou teplotu a poté pomalu ochlazován, což umožňuje částicím dosáhnout stavu s minimální energií.

V kontextu optimalizace se simulované žíhání používá k nalezení globálního minima funkce v prostoru řešení, který může obsahovat mnoho lokálních minim.

Existují ale i druhá verze algoritmu, které řeší únik z lokálního minima jiným způsobem. V obou verzích začíná algoritmus s vysokou "teplotou", která v této verzi umožňuje velké změny v řešení. Postupně se teplota snižuje. Pro mojí práci jsem použil variantu se zkracujícími se krokem.

Simulované žíhání je užitečné pro řešení složitých optimalizačních úloh s mnoha lokálními minimy.

Kroky simulovaného žíhání jsou následující

1. Inicializace náhodným řešením, vysoká teplota
2. Generování sousedního řešení pomocí mutace
3. Vyhodnocení (porovnání výsledku mutace s rodičem)
4. Ochlazování (annealing):

- Teplota se postupně snižuje podle předem daného plánu (např. lineárně nebo exponenciálně).
- S klesající teplotou se zkracuje délka kroku

5. Ukončení:

   Algoritmus končí, když teplota klesne pod nastavenou mez nebo po dosažení předem daného počtu kroků.

8.5. Kombinace simulovaného žíhání a genetických algoritmů

Simulované žíhání lze kombinovat s genetickými algoritmy k vytvoření hybridních optimalizačních metod, které mohou využívat výhody obou přístupů, v závislosti na konkrétním problému a implementaci. Tento hybridní přístup může být velmi efektivní při řešení složitých optimalizačních úloh, které obsahují mnoho lokálních minim. V genetických algoritmech se simulované žíhání často používá jako metoda pro zlepšení kvality řešení během evolučního procesu. Po každé generaci genetického algoritmu může být aplikováno simulované žíhání na jednotlivé jedince, aby se zvýšila pravděpodobnost nalezení globálního minima.

Tento proces zahrnuje následující kroky:

1. **Inicializace**: Je vygenerována počáteční populace.
2. **Evoluce**: Je spuštěn genetický algoritmus.
3. **Simulované žíhání**: Na vybrané jedince se aplikuje simulované žíhání, které umožňuje uniknutí z lokálních minim při vysoké teplotě nebo doladění výsledku při nízké teplotě
4. **Iterace**: Proces se opakuje, dokud není dosaženo optimálního řešení nebo splněny zastavovací podmínky

Tento hybridní přístup může výrazně zlepšit výkon genetických algoritmů, zejména při řešení komplexních problémů, kde je důležité najít globální extrém.

9. Ukázková implementace

9.1. Úkol

V implementaci je úkolem pomocí použitých dat najít minimum zadané funkce. Pro ukázku jsem si vybral dvě funkce. Tou první je sombrero funkci, o které vím, že má maximum v 0 a má falešná maxima, která ve 2D tvoří prstence. Ve více rozměrech jsou tyto maxima kulová, což je vidět na grafu simulovaného žíhání. Jako druhou jsem si vybral funkci, kterou nazývám UF1. Její graf bude ukázán dále.

9.2. Zdroj dat populace

Data, ze kterých vytvářím počáteční počáteční populaci jsou uměle vytvořena jsem vytvořil jako náhodné body v pětirozměrném prostoru, kde se každý rozměr pohybuje v rozmezí <-5;5). Pro nalezení minima funkce jsem použil první čtyři rozměry. Pátý rozměr používám pouze jako místo pro uložení fitness skóre a při prvním kroku algoritmu je přepsán výstupem účelové funkce. Aplikaci jsem psal, aby dokázala přečíst jakýkoli počet vstupních rozměrů, ale tuto funkcionalitu jsem při srovnávání algoritmů v této práci nevyužil.

9.3. Použité technologie a jazyk

K vypracování jsem použil Apache Spark a pro samotnou implementaci jsem použil Javu, což je jeden z programovacích jazyků podporovaných Apache Spark. Dalšimi podporovanými jazyky jsou Scala, ve kterém je Apache Spark vytvořena a Python. Javu jsem si vybral z jediného důvodu, kterým je to, že s vývojem v Javě mám největší zkušenosti. Nejnovější podporovaná verze Javy je Java 11.

9.4. Použitý hardwarové a softwarové vybavení

Veškeré výpočty jsem prováděl v lokálním režimu. Verze Apache Spark je 3.5.0. Verze Javy je OpenJDK 11.0.27.

Použitý hardware je:

- CPU: 13th Gen Intel(R) Core(TM) i9-13950HX
- RAM: 2x 32GB DDR5 5.2GT (Crucial SO-DIMM 32GB DDR5 5600MHz CL46)
- SDD: NVMe WD PC SN560 SDDPNQE-1T00-1032

9.5. Postup výpočtu optimalizace

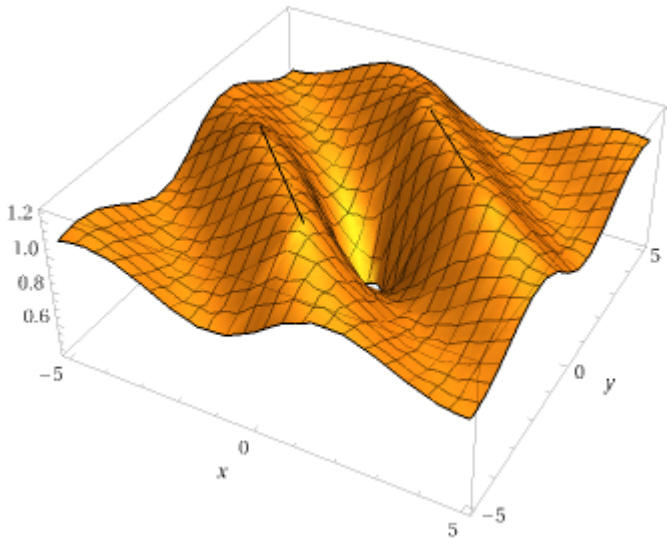
Program dokáže použít tři z dříve popsaných algoritmů, a to

- Genetický algoritmus
- Diferenciální algoritmus
- Paralelní simulované žíhání

Pro vyhodnocení těchto algoritmů jsem použil dvě rozdílné účelové funkce. První funkcí (UF1) je

$$1 - \frac{\cos(x + y + z + w)}{|x| + |y| + |z| + |w| + 1}$$

na které lze bez problému ukázat, jak si algoritmy poradí s lokálními extrémy. Zde jsou zobrazeny pouze osy X a Y s hodnotami rozměrů Z a W nastavených na 0.

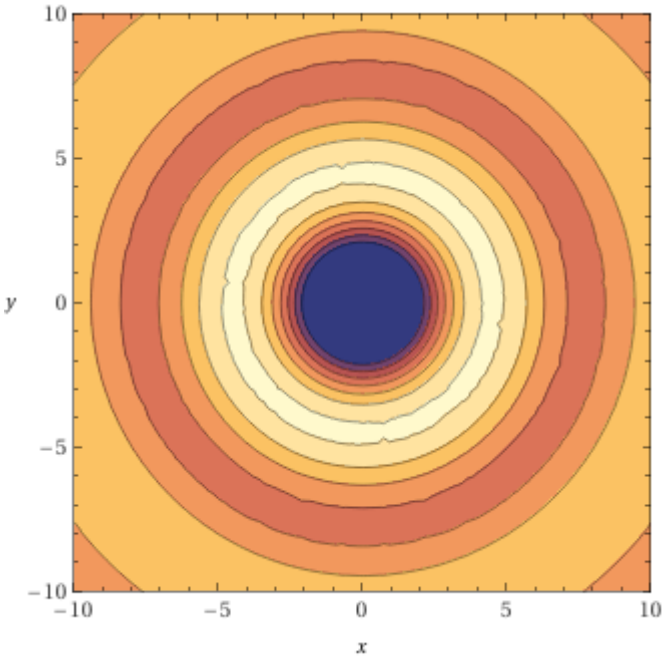


Obrázek 4 - funkce UF1 ve 3D

Tuto funkci si nelze jednoduše představit ve vícerozměrném prostoru. Proto jsem si vybral druhou funkci, jejíž minimum budu algoritmicky hledat. Touto funkcí je sombrero funkce, kterou jsem upravil, aby měla globální minimum místo globálního maxima v bodě (0;0;0;0) se vzorcem

$$1 - \frac{\sin\left(\sqrt{x^2 + y^2 + z^2 + w^2}\right)}{\sqrt{x^2 + y^2 + z^2 + w^2}}$$

Tato funkce se ve 2D skládá ze soustředných kružnic lokálních minim okolo globálního minima, které se po rozšíření do více rozměrů rozšíří na povrch koule. Pro zobrazení této funkce jsem použil konturový graf, jelikož je v tomto případě přehlednější než 3D graf.



Obrázek 5 - konturový graf sombrero funkce

Každý z těchto algoritmů má svoje výhody a slabiny. Uvedené časy jsou zaokrouhlené. Měření jsem začal před vytvořením SparkSession a toto měření jsem ukončil po získání výsledků z ovladače. Pro získání času výpočtu jsem od aktuálního počtu milisekund od 1.1.1970 odečetl počet milisekund, který byl, když jsem s měřením času začal.

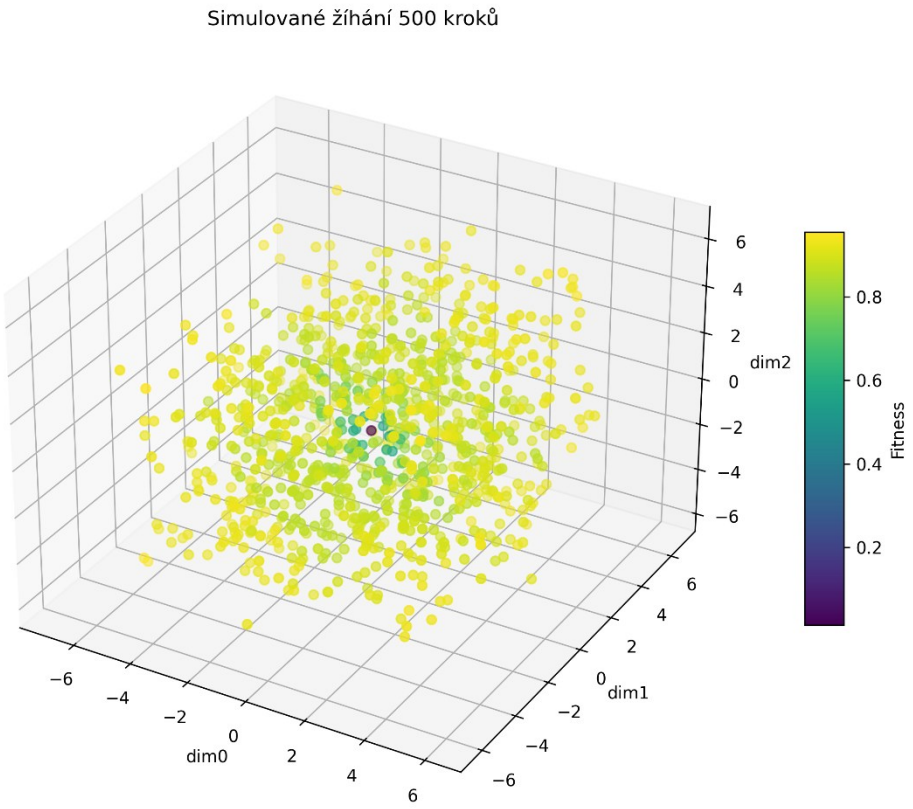
Genetický algoritmus je má tendenci velice rychle konvergovat k nejlepšímu dosaženému výsledku za cenu šance nalezení pravého minima. Je také velice výpočetně náročný z důvodu řazení a konkrétně transformace zipWithIndex. Trvání této transformace je přibližně 0.5 sekundy na iteraci. Data jsem získal z webové stránky, která je dostupná na ip adrese spark driveru a portu 4040. Tato transformace přiřazuje prvkům pořadové číslo, které se poté použije pro rozhodování, kteří jedinci dostanou možnost zůstat v populaci a mít potomka. Čas pro výpočet 100 kroků a vstupní populaci 1024 jedinců je 57 až 66 sekund.

Diferenciální algoritmus je rychlejší (15-17.5 s), ale jeho rychlost konvergence k výsledku je nižší. Má vyšší tendenci prozkoumávat i jiné než nejlepší nalezené řešení. Pro jeho správné nastavení je nutné nastavit sílu dvou ze tří složek šumového vektoru (F) a pravděpodobnost použití složky šumového vektoru jako gen jedince (NF). Tato pravděpodobnost nesmí být příliš velká. Vysoká pravděpodobnost vede k zvyšování náhodnosti výsledné pozice a může snižovat rychlost konvergence. Jelikož používám čtyři geny na jedince, tak snižování NF pod 25 % pouze zvýší pravděpodobnost, že jedinec nebude v daném kroku změněn. Tento algoritmus nepoužívá řazení, všichni jedinci existují po celou dobu běhu.

Nejrychlejším algoritmem na výpočet je paralelní simulované žíhání s časem 8.4 s pří 100 krocích, kde velkou většinu tohoto času (7.9 sekundy) tvoří příprava prostředí apache spark. Tento algoritmus je rozšíření algoritmu simulované žíhání, které používá jednoho jedince. Tento algoritmus je nejflexibilnější ze všech, a tak lze snadno udělat chybu. Použil jsem variantu se zkracujícím se krokem, kde nejvíce záleží na chladící funkci, což je kromě počtu kroků jediný parametr.

Funkce 1/x, která imituje reálné proudění tepla z horkého předmětu není vždy vhodná. Jako příklad mohu použít UF1. Po 500 krocích existuje 60% šance, že jeden z 1024 jedinců se dostane do okolí globálního minima, které jsem určil jako fitness < 0,1. Po 10 000 krocích trvajících 14.5 sekund je v globálním minimu

průměrně 3.8 jedinců. Zbylí jedinci jsou v nejbližším neglobálním minimu. Toto je vidět v grafu, kde jsou zaneseny pouze první 3 souřadnice.



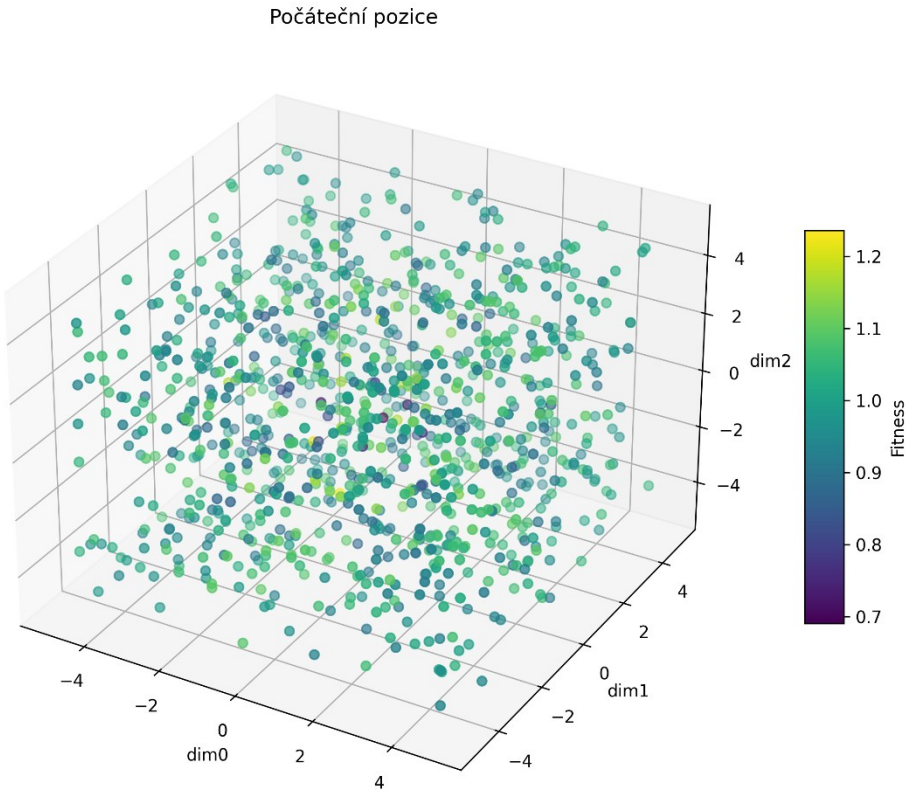
Obrázek 6 - 500 kroku algoritmu simulované žíhání funke UF1

Pro sombrero funkci se stejnou chladící funkcí je výsledek blíže globálnímu minimu. Po 500ti krocích je 18 % jedinců v blízkosti globálního minima. Ostatní jedinci jsou v nejbližším lokálním minimu.

Lepší variantou pro funkci chlazení je lineární funkce (1-i/c), kde i je aktuální krok a c je celkový počet kroků. Tato funkce dá jedincům více možností ocitnout se ve výhodnějším stavu. Již při padesáti krocích lze nalézt několik jedinců v blízkosti globálního minima. Nevýhodou lineární funkce je poměrně krátká doba ve stavu s nízkou teplotou, který by dovolil jedincům najít nejnižší bod jejich lokálního minima.

9.6. Ukázka vstupů a výstupů, vyhodnocení funkce algoritmu

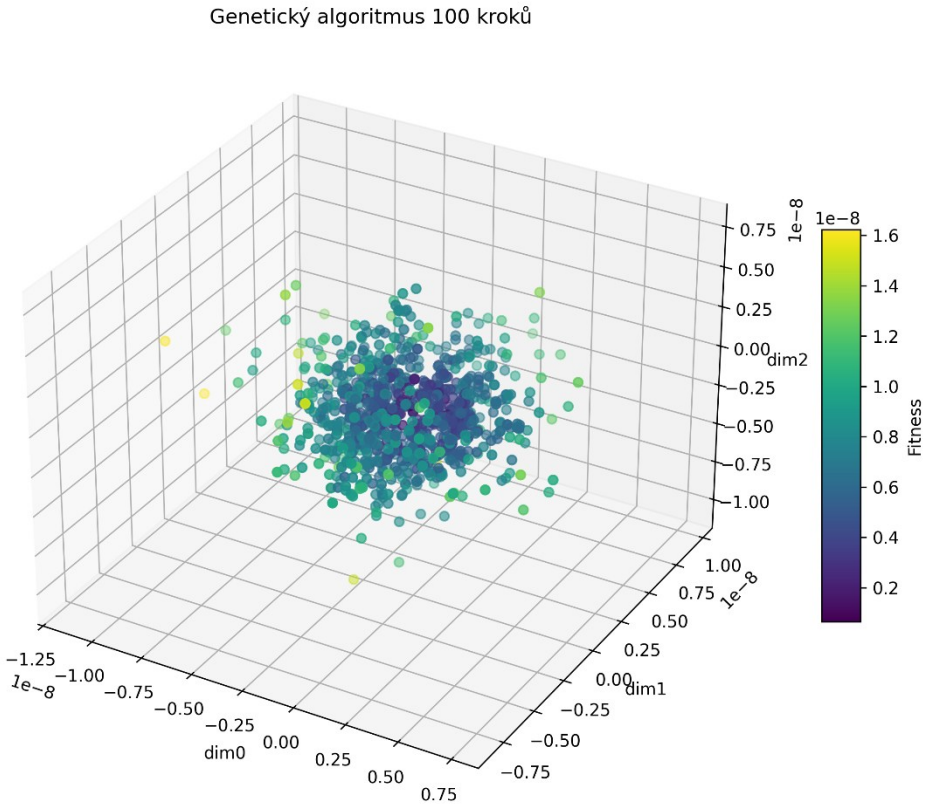
Vstupní data jsou rovnoměrně rozložena v prostoru, kde hledám minimum funkce. Data jsem vygeneroval pseudonáhodným generátorem a výsledky uložil do CSV souboru. Tento soubor načítám v ukázkové aplikaci do RDD, které poté převádím na list polí obsahujících desetinná čísla typu double. To poté převádím zpět do RDD. To proto, aby byla využita při výpočtu všechna jádra procesoru. Takto vypadají data s úpravou hodnoty fitness, aby tato hodnota odpovídala hodnotě fitness účelové funkce UF1. Při generaci těchto hodnot jsem hodnotu, která se stane hodnotou fitness naplnil čistě náhodnými hodnotami.



Obrázek 7 - graf počátečního rozložení dat

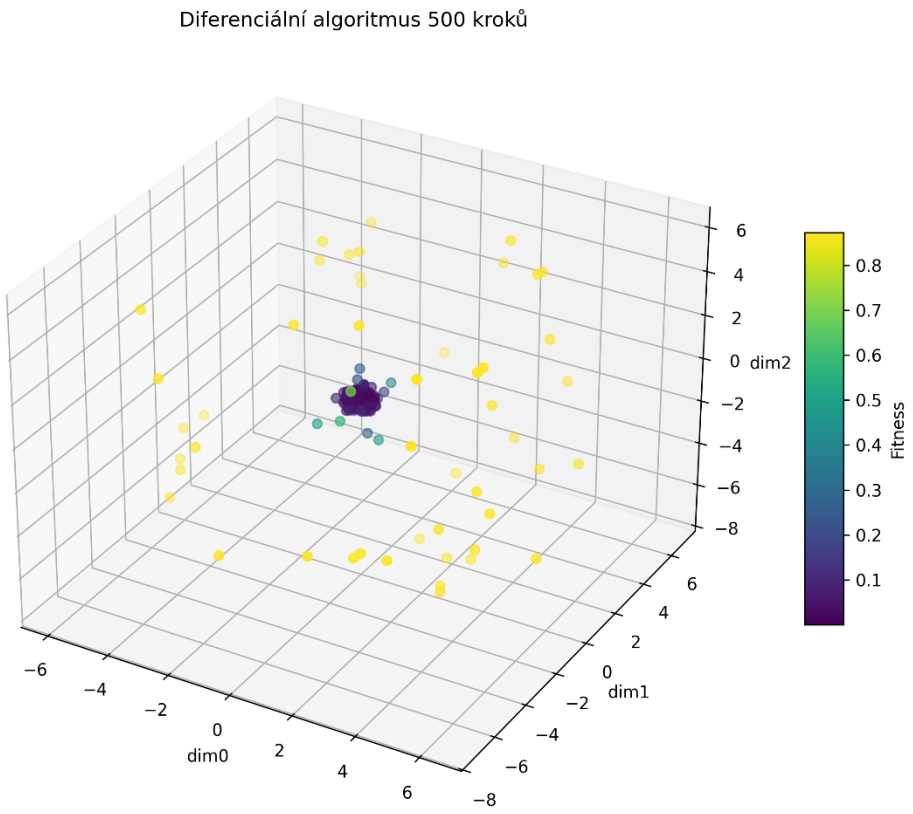
Nejlepších výsledků dosáhl genetický algoritmus, který po 100 krocích dosahuje nejlepšího fitness skóre při použití UF1 $2*10^{-9}$ nebo nižšího. Po 175 krocích je z důvodu konečné přesnosti typu double a implementace funkce cosinus hodnota fitness rovna nule a algoritmus nemůže dále pokračovat, dosáhl minima.

Pokud použijeme sombrero funkci pak se chování této metody nezmění. Jedinci, kteří se nedostanou ke globálnímu minimu jsou již po dvaceti generacích vyhubeni. Nejlepší jedinec má po těchto dvaceti generacích fitness skóre $2.5*10^{-4}$. Po sto krocích je minimální fitness skóre 0. Těchto jedinců se skóre 0 je polovina.



Obrázek 8 - graf výstupu genetického algoritmu (100 kroků UF1)

Diferenciální algoritmus dosáhl po sto krocích pouze fitness hodnoty $6*10^{-3}$. Pokud ale budeme brát v potaz čas výpočtu místo počtu kroků algoritmu tak za stejnou dobu jedné minuty při které genetický algoritmus dosáhne při 100 krocích hodnoty $2*10^{-9}$ dosáhne diferenciální algoritmus při 400 krocích fitness hodnoty menší než 10^{-14}. Jelikož je zvykem tohoto algoritmu nalézt i jiná než nejlepší ihned dosažitelná řešení bez toho, aby jedinci hromadně zůstávali v ne tak hlubokých lokálních minimech. To je vidět na grafu, kde se většina jedinců shromáždila v globálním minimu, ale někteří z jedinců zůstali v lokálních minimech.



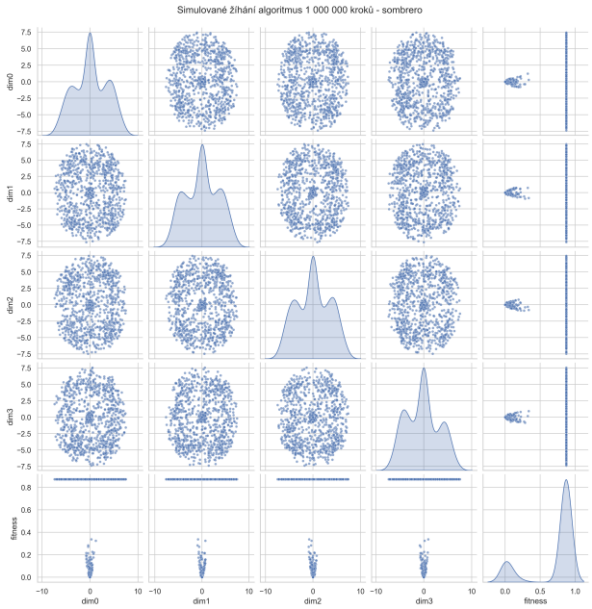
Obrázek 9 - graf diferenciálního algoritmu 500 kroků UF1

Posledním implementovaným algoritmem je simulované žíhání. Tento algoritmus nepoužívá populace nebo více jedinců. Simulované žíhání používá jednoho jedince. Toho využívá rozšíření paralelní simulované žíhání, které není nic jiného než spuštění více simulovaných žíhání paralelně s jiným počátečním stavem.

Tento algoritmus má pomalou konvergenci, ale během svého průběhu často prozkoumá velké oblasti. Výpočetní rychlost tohoto algoritmu mi umožňuje spuštění mnoha iterací. Při miliónu iterací a hyperbolické chladící funkci dokázali někteří jedinci překonat vzdálenost nejméně tří jednotek směrem od globálního maxima pří použití účelové funkce UF1 a dvě nebo více při použití sombrero funkce.

Při použití UF1 a miliónu iterací má lineární chladící funkce lepší výsledky než hyperbolická funkce. Naopak sombrero funkce vyžaduje hyperbolickou funkci teploty pro dosažení lepších výsledků. Při použití sombrero funkce jako účelové

funkce je při lineární chladící funkci dosaženo minimální hodnoty 10^{-13} a pří hyperbolické funkci je to méně než 10^{-16}.



Obrázek 10 - simulované žíhání - sombrero funkce - hyperbolická funkce - 1 000 000 kroků

9.7. Vyhodnocení výsledků

Všechny algoritmy dosáhly výsledků s dostatečnou přesností při méně než 200 iteracích.

Tyto algoritmy jsou vhodné, pokud jedinou garancí zkoumané funkce, jejíž minimum nebo maximum hledáme je, přítomnost hodnoty ve zkoumaném okolí.

Genetický algoritmus, diferenciální algoritmus a paralelní simulované žíhání nemohu srovnávat s jinými algoritmy, které například používají derivaci funkce k nalezení směru, kterým je extrém zajímavý pro algoritmus, což může výrazným způsobem snížit nutný počet kroků potřebných pro nalezení extrému funkce.

Mou testované algoritmy nejsou vhodné, pokud je počet dimenzí problému vysoký. To je jasně vidět například na genetickém algoritmu. Ten jsem spustil dvakrát. Poprvé s pěti geny a podruhé s deseti. Po padesáti krocích výsledky vypadaly takto: S pěti geny jsem získal fitness skóre 4.3256^{-8} a s deseti geny jsem

dosáhl pouhých 0.086. To potvrdilo mou domněnku závislosti počtu kroků algoritmu na počtu dimenzí/genů aby se algoritmus přiblížil ve stejné míře svému cíli. Algoritmy jako například hill climbing algorithm, požadují po účelové funkci další vlastnosti, ale jejich rychlost konvergence by neměla být ovlivněna počtem nezávyslích proměnných v problému nebo dimenzí funkce.

V příloze B lze nalézt tabulku s

10. Závěr

Účelem této práce bylo seznámení s některými evolučními algoritmy, jejich popis a implementace v prostředí apache spark za pomoci jednoho z podporovaných jazyků. Tyto algoritmy jsou poměrně jednoduché ale také mocné s mnoha uplatněními v reálném životě.a to nejen pro hledání extrému funkce ale i jiným aplikacím. (36)

Pro simulované žíhání budu pro srovnání používat lineární chladící funkci.

Reference

1. **The Apache Software Foundation.** spark.apache.org. [Online] 2018. https://spark.apache.org.

2. **Wikimedia Foundation, Inc.** Apache Spark. [Online] https://cs.wikipedia.org/wiki/Apache_Spark.

3. **Zaharia, Matei.** apache/spark. *Github.com*. [Online] 30. 3 2010. https://github.com/apache/spark/commit/df29d0ea4c8b7137fdd1844219c7d489e3b0d9c9.

4. **The Apache Software Foundation.** Spark history. *spark.apache.org*. [Online] https://spark.apache.org/history.html.

5. **InterviewBit.** Apache Spark Architecture. [Online] InterviewBit, 3. 6 2022. https://www.interviewbit.com/blog/apache-spark-architecture/.

6. **The Apache Software Foundation.** Cluster Mode Overview. [Online] https://spark.apache.org/docs/latest/cluster-overview.html.

7. —. RDD Programming Guide. *spark.apache.org*. [Online] https://spark.apache.org/docs/latest/rdd-programming-guide.html.

8. **IBM.** What is a Resilient Distributed Dataset. [Online] 2025. https://www.ibm.com/think/topics/resilient-distributed-dataset.

9. **The Apache Software Foundation.** Spark SQL, DataFrames and Datasets. [Online] https://spark.apache.org/docs/latest/sql-programming-guide.html.

10. **Microsoft.** Tutorial: Load and transform data using Apache Spark DataFrames. [Online] 2025. https://learn.microsoft.com/en-us/azure/databricks/getting-started/dataframes.

11. **GeeksforGeeks.** Java Lambda Expressions. [Online] 2025. https://www.geeksforgeeks.org/lambda-expressions-java-8/.

12. **The Apache Software Foundation.** Spark SQL, DataFrames and Datasets Guide. [Online] https://spark.apache.org/docs/3.5.4/sql-programming-guide.html.

13. **DataFlair .** Spark Tutorial – Learn Spark Programming. [Online] 2025. https://data-flair.training/blogs/spark-tutorial/.

14. **Databricks.** Tutorial: Load and transform data using Apache Spark DataFrames. [Online] 2025. https://docs.databricks.com/aws/en/getting-started/dataframes.

15. **Simpilearn.** Hadoop Ecosystem. [Online] 2025. https://www.simplilearn.com/tutorials/hadoop-tutorial/hadoop-ecosystem.

16. **TechVidvan.** Hadoop Ecosystem. [Online] DataFlair Web Services Pvt. Ltd. , 2025. https://techvidvan.com/tutorials/hadoop-ecosystem-tutorial/.

17. **Wikimedia Foundation, Inc.** Apache Hadoop. [Online] 2025. https://en.wikipedia.org/wiki/Apache_Hadoop.

18. **Databricks Inc.** What Is Hadoop. [Online] 2025. https://www.databricks.com/glossary/hadoop.

19. **Amazon Web Services, Inc.** What's the Difference Between Hadoop and Spark. [Online] 2025. https://aws.amazon.com/compare/the-difference-between-hadoop-vs-spark/.

20. **Apache Software Foundation.** Apache hadoop. [Online] 2024. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/SecureMode.html.

21. **The Apache Software Foundation.** Spark Security. [Online] https://spark.apache.org/docs/latest/security.html#authentication-and-authorization.

22. **Databricks Inc.** MapReduce. [Online] 2025. https://www.databricks.com/glossary/mapreduce.

23. **IBM.** What is MapReduce. [Online] 2024. https://www.ibm.com/think/topics/mapreduce.

24. **Wikimedia Foundation, Inc.** Big Data. *Wikipedia*. [Online] 5. 4 2025. https://en.wikipedia.org/wiki/Big_data.

25. **Oracle.** Co jsou big data. [Online] 2025. https://www.oracle.com/cz/big-data/what-is-big-data/).

26. **Eva Kalátová, Jaroslav Dobiáš.** Evoluční algoritmy. [Online] 2000. https://www.kiv.zcu.cz/studies/predmety/uir/gen_alg2/E_alg.htm.

27. **Volná, Eva.** EVOLUČNÍ ALGORITMY A NEURONOVÉ SÍTĚ. [Online] 2012. https://web.osu.cz/~Volna/Evolucni_algoritmy_a_neuronove_site.pdf.

28. **Wikimedia Foundation, Inc.** Genetic algorithm. *wikipedia.org*. [Online] 2025. https://en.wikipedia.org/wiki/Genetic_algorithm.

29. —. John Henry Holland. *wikipedia.org*. [Online] https://en.wikipedia.org/wiki/John_Henry_Holland.

30. **Wikimedia Foundation, Inc.** Evolution strategy. [Online] wikipedia.org. https://en.wikipedia.org/wiki/Evolution_strategy.

31. —. Evolutionary programming. [Online] wikipedia.org. https://en.wikipedia.org/wiki/Evolutionary_programming.

32. **Lepš, Matěj.** Evoluční algoritmy. [Online] 2007. https://mech.fsv.cvut.cz/~leps/teaching/mmo/prednasky/prednaska08_EAs.pdf.

33. **Wikimedia Foundation, Inc.** Genetický algoritmus. [Online] 2023. https://cs.wikipedia.org/wiki/Genetick%C3%BD_algoritmus.

34. **Luner, Petr.** Jemný úvod do genetických algoritmů. [Online] https://cgg.mff.cuni.cz/~pepca/prg022/luner.html.

35. **Fahad Maqbool, Saad Razzaq, Jens Lehmann, Hajira Jabeen.** Scalable Distributed Genetic Algorithm Using. [Online] 2020. https://project-lambda.org/sites/default/files/2020-02/UBO_SGA.pdf.

36. **Ivan, Zelenka.** *Umělá inteligence v problémech globální optimalizace.* Praha : BEN - Technická literatura, 2002. 80-7300-069-5.

37. **Wikimedia Foundation, Inc.** Simulated annealing. *wikipedia.org.* [Online] https://en.wikipedia.org/wiki/Simulated_annealing.

38. **Vitingerová, Zuzana.** TVORBA NEZÁVISLÝCH VEKTORŮ S UŽITÍM METODY SIMULOVANÉHO ŽÍHÁNÍ. [Online] 2005. https://mech.fsv.cvut.cz/~anicka/bazant/works/2004_vitingerova.pdf.

39. **Jacobson, Lee.** Simulované Žíhání pro začátečníky. [Online] 2013. https://kathryncoltrinbooks.com/cs/simulovan%C3%A9-%C5%BD%C3%ADh%C3%A1n%C3%AD-pro-za%C4%8D%C3%A1te%C4%8Dn%C3%ADky/.

11. Přílohy

A: Zdrojový kód programu

```
package cz.upce.martinekdaniel;

import org.apache.hadoop.util.Time;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
import org.joda.time.DateTime;



import java.io.IOException;
import java.io.PrintWriter;
import java.nio.charset.StandardCharsets;
import java.util.*;

enum Algorithm {NONE, GENETIC, DIFFERENTIAL, ANNEALING}

//TIP To <b>Run</b> code, press <shortcut actionId="Run"/> or
// click the <icon src="AllIcons.Actions.Execute"/> icon in the gutter.
public class Main {
    static Random random = new Random();
    static final int GENES = 4;
    static final int CPUS = Runtime.getRuntime().availableProcessors();
    static int poc = 0;
    static long compB;
```

```
    static long compE;

    public static void main(String[] args) throws IOException {

        Algorithm algorithm = Algorithm.NONE;
        Scanner s = new Scanner(System.in);

        do {
            System.out.print(
                    " 1: geneticky\n 2: diferencialni\n 3: simulovane
zihani\n\nAlgoritmus: "
            );
            int d = s.nextInt();
            switch (d) {
                case 1:
                    algorithm = Algorithm.GENETIC;
                    break;
                case 2:
                    algorithm = Algorithm.DIFFERENTIAL;
                    break;
                case 3:
                    algorithm = Algorithm.ANNEALING;
            }
        } while (algorithm == Algorithm.NONE);
        System.out.print("Pocet opakovani: ");
        int count = s.nextInt();
        PrintWriter writer = new PrintWriter("data_graph_best_" + algorithm + "_"
+ count + ".csv", StandardCharsets.UTF_8);
        PrintWriter writer1 = new PrintWriter("data_graph_all_" + algorithm + "_"
```

```
+ count + ".csv", StandardCharsets.UTF_8);
        writer.println("min;med;max");
        DateTime current;



        current = DateTime.now();
        SparkSession spark = SparkSession.builder().appName("Simple
Application").config("spark.master", "local[*]").config("spark.driver.host",
"localhost").getOrCreate();
        JavaRDD<double[]> rdd, rdd2;
        Dataset<Row> dataset=null;
        try {
            dataset = spark.read().csv("inpdata.csv");
        }catch(Exception e){ //org.apache.spark.sql.AnalysisException je
vyvolána, pokud se ale konkrétně odchytává tak nelze zkompilovat
            System.err.println("Nebyl nalezen soubor inpdata.csv. Data se
vygenerují pomocí java RNG.");
        }
        JavaRDD<Row> rdd1=null;
        if(dataset!= null)rdd1 = dataset.toJavaRDD();
        try (JavaSparkContext spc =
JavaSparkContext.fromSparkContext(spark.sparkContext())) {
            List<double[]> l = rdd1 == null ? new ArrayList<>() : rdd1.map(row ->
Arrays.stream(row.toString().replace("[", "").replace("]",
"").split(",")).mapToDouble(Double::parseDouble).toArray()).collect();
            //for (genes = 1;genes<100;genes++)
            {
                if (l.isEmpty() || GENES != l.get(0).length - 1) { //pokud se nenačetla
data nebo počet genů nesouhlasí s nastavením aplikace
```

```
                l = new ArrayList<>();
                for (int i = 0; i < 1024; i++) {
                    l.add(random.doubles(GENES + 1).map(a -> (a - 0.5) *
10).toArray());
                }
            }
            //  l =
l.stream().map(Main::setFitness).sorted(Comparator.comparingDouble(t ->
t[genes])).collect(Collectors.toList());
            rdd = spc.parallelize(l, CPUS);
            rdd = rdd.map(Main::setFitness);
            rdd2 = rdd; //kopie vstupních dat
            compB = Time.monotonicNowNanos(); //počátek výpočtu
            for (int i = 0; i < count; i++) {
                int finalI = i > 0 ? i : 1;
                if (algorithm == Algorithm.GENETIC) rdd = genetic(rdd, (int)
rdd.count(), writer); //Spuštění kroku genetického algoritmu
                else if (algorithm == Algorithm.DIFFERENTIAL) {
                    rdd = differential(rdd, 0.3, 0.3); //spuštění kroku diferenciálního
algoritmu s NF=0,3 a CR=0,3
                } else rdd = rdd.map(a -> { //krok paralelního simulovaného žíhání
                    int currentStep = finalI;
                    int countStep = count;
                    double[] b = Mutate(a, cooling(currentStep, countStep));
                    setFitness(b);
                    return a[GENES] < b[GENES] ? a : b;
                });
                  //writer.println((parents.get(0)[genes]
+";"+parents.get(parents.size()/2)[genes] + ";" + parents.get(parents.size()-
```

```
1)[genes]).replace('.',','));
                if (i % 200 == 0) {
                    rdd = spc.parallelize(rdd.collect(), CPUS); //spark nezvládá více než 230 opakování algoritmu bez zhroucení z důvodu hloubky DAC.
                }
            }
            List<double[]> lst = rdd.map(Main::setFitness).collect(); //přebírá výsledky z driveru
            compE = Time.monotonicNowNanos();
            rdd2.sortBy(a -> Math.abs(a[GENES]), true, CPUS).collect().forEach(e -> System.out.println(Arrays.toString(e)));
            lst.forEach(e -> writer1.println(Arrays.toString(e)
                    .replace("[", "")
                    .replace("]", "")));
            writer1.flush();

            //  rdd1.sortBy(a->Math.abs(Double.parseDouble(a.getString(genes))),true,cpus).collect().forEach(e -> System.out.println(e.toString()));
            System.out.println();

            System.out.println();
            System.out.println();
            System.out.println();
            lst.stream().sorted(Comparator.comparingDouble(a -> a[GENES])).forEach(e -> {
                System.out.println(Arrays.toString(e));
                if (e[GENES] < 0.1) poc++;
```

```
            });
            System.out.println((DateTime.now().getMillis() - current.getMillis())
* 1.0 / 1000 + " (Samotný výpočet trval " + (compE - compB) /
1_000_000_000.0 + "s)");

        }
    }
    writer.close();
    System.out.println(poc);
}


//
private static JavaRDD<double[]> differential(JavaRDD<double[]> rdd,
double nf, double cr) {
    List<double[]> parents = rdd.collect(); // nenašel jsem jinou možnost která
by neomezovala výběr. Operace ve Spark nelze vkládat do sebe(alespoň map a
take)

    rdd = rdd.map(a -> {

        double[] b;
        int x = random.nextInt(parents.size());
        int y = random.nextInt(parents.size());
        int z = random.nextInt(parents.size());//výběr prvních tří rodičů
        double[] sv = new double[GENES + 1];
        for (int j = 0; j < GENES + 1; j++) {
            sv[j] = nf * (parents.get(x)[j] - parents.get(y)[j]) + parents.get(z)[j];
        }// vytváření šumového vektoru
```

```
            b = Combine(a, sv, cr, true); // samotné křížení
            setFitness(a);
            setFitness(b);
            return a[GENES] > b[GENES] ? b : a;//výběr lepšího (nahrazení rodiče potomkem pokud je potomek lepší)
        });
        return rdd;
    }

    private static JavaRDD<double[]> genetic(JavaRDD<double[]> rdd, int k, PrintWriter writer) {
        rdd = rdd.map(Main::setFitness).sortBy(a -> a[GENES], true, CPUS).zipWithIndex().filter(a -> a._2() <= random.nextInt(k)).keys();
        List<double[]> parents = rdd.collect(); // nelze použít rdd.take nebo rdd.takeSample
        rdd = rdd.union(
                rdd.map(prnt -> Combine(prnt, parents.get(random.nextInt(parents.size() - 1))))
                        .map(s -> {/*Mutate(s,parents.get(0)[genes]*0.001);*/
                            return s;
                        }));
        return rdd;
    }

    private static double[] setFitness(double[] a) {
        a[GENES] = 0;
        //a[GENES]=Math.abs(a[0]+a[1]+a[2]+a[3]);
        //a[GENES]=1-Math.cos(Arrays.stream(a).sum())/(Arrays.stream(a).map(Math::abs).sum()+1);
```

```
        a[GENES] = 1 - Math.sin(Math.sqrt(Arrays.stream(a).map(b -> b * b).sum())) / Math.sqrt(Arrays.stream(a).map(b -> b * b).sum());
        return a;
    }

    private static double cooling(int currentStep, int countStep) {
        double coolFuncOld = 1.0 / currentStep;
        double coolFunc = 1.0 - (double) currentStep / countStep;
        return coolFunc;
    }

    static double[] Mutate(double[] value, double maxStep) {
        double[] valueMut = new double[value.length];
        for (int i = 0; i < value.length - 1; i++) {
            valueMut[i] = value[i] + (random.nextDouble() - 0.5) * 2 * maxStep;
        }
        return valueMut;
    }

    static double[] Combine(double[] parent1, double[] parent2) {
        return Combine(parent1, parent2, 0.5);
    }

    static double[] Combine(double[] parent1, double[] parent2, double probability, boolean isStrict) {
        if (parent1.length != parent2.length) throw new UnsupportedOperationException("Different species");
        double[] child = new double[parent1.length];
        double mixStr = random.nextDouble() * 2 - 0.5;
```

```
        for (int i = 0; i < parent1.length - 1; i++) {
            child[i] = random.nextDouble() <= probability ? parent1[i] : isStrict ?
parent2[i] : interpolate(parent1[i], parent2[i], mixStr);
        }
        return child;
    }

    static double[] Combine(double[] parent1, double[] parent2, double
probability) {
        return Combine(parent1, parent2, probability, false);
    }

    static double interpolate(double a, double b, double ratio) {
        return b * ratio + a * (1 - ratio);
    }
}
```

Příloha B: Výsledky běhu algoritmu

Tabulka 2 - Výsledky běhu algoritmů

ÚČELOVÁ FUNKCE	ALGORITMUS	POČET OPAKOVÁNÍ	ČAS S INICIALIZACÍ; ČISTÝ ČAS VÝPOČTU	VÝSLEDEK (MIN; MED; MAX)
UF1	Genetický algoritmus	50	31,8s; 25,3s	7,533938E-5 5,923884E-4 0,001386808

		100	56,0s; 50,2s	5,13052E-9 2,02428E-8 4,027610E-8
		200	87.8s; 82s	0.00000000; 0.00000000; 2.22044E-16
	Diferenciální algoritmus	50	10,2s; 3,8s	0,003997134 0,041150801 0,099403744
		100	15,8s; 9,5s	6,309593E-7 7,986841E-6 2,065072E-5
		200	40,0s; 34,0s	5,8619E-14 4,19442E-13 1,06847E-12
	Paralelní simulované žíhání	50	6,7s; 0,2s	0,061460429 0,869113758 0,949571103
		100	6,9s; 0,5s	0,018875058 0,861376349 0,949557495

		200	7,0s; 0,7s	0,007641872 0,861376078 0,949557112
SOMBRERO FUNKCE	Genetický algoritmus	50	32,6s; 26,8s	3,97232E-10 1,078238E-8 7,50140E-8
		100	52,4s; 46,5s	0,0 1,11022E-16 6,66133E-16
		200	65,2s; 59,4s	0,0 0,0 2,22044E-16
	Diferenciální algoritmus	50	9,9s; 3,8s	1,072723E-5 4,409288E-4 0,003173251
		100	15,6s; 9,6s	1,04405E-12 4,24801E-11 4,95205E-10
		200	39,0s; 32,8s	0.0 0.0 0.0

	Paralelní simulované žíhání	50	6,9s; 0,3s	2,586293E-4 0,871625657 0,871694371
		100	7,1s; 0,5s	1,153591E-5 0,871625487 0,871633385
		200	7,1s; 0,7s	2,662547E-6 0,871625452 0,871627029