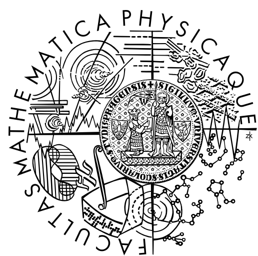


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

DIPLOMOVÁ PRÁCE

Bc. Petr Záboj

Efektivní paralelizace evolučních algoritmů

Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: Mgr. Martin Pilát, Ph.D.
Studijní program: Informatika
Studijní obor: IUI

Praha 2020

Prohlašuji, že jsem tuto diplomovou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V Praze dne 30.7.2020 Bc. Petr Záboj

Poděkování

V první řadě bych chtěl poděkovat svému vedoucímu Mgr. Martinu Pilátovi, Ph.D., za jeho podporu a vedení mé diplomové práce. Tato práce by nevznikla bez jeho podnětných vstupů a cenných rad. Také mu patří velké díky za trpělivost, kterou se mnou měl v celém průběhu práce na tomto projektu. Další poděkování patří organizaci MetaCentrum, jež poskytuje výpočetní prostředky v rámci projektu „e-Infrastruktura CZ“ (e-INFRA LM2018140) v rámci programu Projekty velkých výzkumných, vývojových a inovačních infrastruktur. Díky těmto výpočetním prostředkům jsem mohl provést sérii reálných výpočtů a také urychlit výpočet simulací. Také bych chtěl poděkovat všem, kteří se starají o laboratoře MFF UK na Malé Straně, protože v této laboratoři byla provedena velká část simulací. Mé poděkování patří i mým kolegům a přátelům, kteří mě podporovali a motivovali jak v rámci studia, tak při psaní této diplomové práce. Velké díky patří také mým vedoucím v zaměstnání Ing. Šárce Krkoškové a Ing. Adéle Ráčkové, Ph.D., které mi vyšly vstříc v rozvrhu mé práce a podporovaly mě na cestě k úspěšnému dokončení studia. Nakonec bych chtěl poděkovat své rodině, která mě v průběhu celého studia podporovala, motivovala a zabezpečovala. Bez nich bych se nikdy nedostal tak daleko.

Název práce: Efektivní paralelizace evolučních algoritmů

Autor: Bc. Petr Záboj

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: Mgr. Martin Pilát, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Evoluční algoritmy se často používají na těžké optimalizační problémy. Jejich řešení trvá netriviální čas a proto tu je tlak na efektivní paralelizaci těchto algoritmů. Bohužel klasické metody paralelizace nefungují moc dobře v případech, kdy jednotlivá ohodnocení problémů trvají výrazně různou dobu. V této práci se pokusíme rozšířit evoluční algoritmus s prokládáním generací, který nabízí lepší využití výpočetních zdrojů než klasické paralelní evoluční algoritmy, o spekulativní vyhodnocení. Spekulativním vyhodnocením myslíme odhad fitness funkce jedince a předpočítání následujících kroků, které v případě správného odhadu později využijeme. V sérii experimentů porovnáme algoritmus se spekulativním vyhodnocením s originální verzí a podíváme se na vliv přesnosti ve spekulativním kroku na výkon algoritmus.

Klíčová slova: evoluční algoritmy, paralelizace, vytížení CPU

Title: Effective Parallelization of Evolutionary Algorithms

Author: Bc. Petr Záboj

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Martin Pilát, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Evolutionary algorithms are often used for hard optimization problems. Solving time of this problems is long, so we want effective parallelization for this algorithms. Unfortunately, classical methods of parallelization do not work very well in cases where the individual evaluations of problems take significantly different times. In this project, we will try to extend the evolutionary algorithm with interleaving generations, which offers a better use of computational resources than classical parallel evolutionary algorithms, by speculative evaluation. Speculative evaluation means the estimation of an individual's fitness function and the prediction of the following steps, which we will use later in the case of a correct estimate. We compare the algorithm with speculative evaluation with the original version in a series of experiments and we look at the effect of accuracy in the speculative step on the performance of the algorithm.

Keywords: evolutionary algorithms, paralellization, CPU utilization

Obsah

Úvod **2**

1 Evoluční algoritmy **3**
1.1 Jednoduchý genetický algoritmus 5
1.2 Evoluční strategie 5
1.3 Paralelní evoluční algoritmy 6
1.3.1 Paralelizace globálního modelu 6
1.3.2 Ostrovní model 6
1.4 Evoluční algoritmus s prokládáním generací 6
1.4.1 (μ, μ)-IGEA 7
1.4.2 $(\mu + \mu)$-IGEA 8

2 Evoluční algoritmy s prokládáním generací a spekulativním vyhodnocením **10**
2.1 (μ, μ)-IGEA-SE 10
2.2 (μ, λ)-IGEA-SE a $(\mu + \lambda)$-IGEA-SE 11

3 Implementační podrobnosti **19**
3.1 Task 19
3.2 Paralelní prostředí 19
3.2.1 Simulátor 19
3.2.2 Reálný výpočet 19
3.3 Plánovač 21
3.4 Surrogate model 23
3.5 Transitions 24
3.6 Generace 24
3.7 IGEA-SE 25

4 Experimenty **29**
4.1 Prokládání generací 29
4.2 (100,100) clever tournament 31
4.3 (100,200) a $(100 + 200)$ 31
4.4 Vývoj fitness funkce 31
4.4.1 Rychlá konvergence a různé výsledky algoritmů 32
4.5 (50,100) a $(50 + 100)$ 32
4.6 (100,200) a $(100 + 200)$ s dvěma vítězi v turnaji 33
4.7 (100,100) bez chytrého turnaje a $(100 + 100)$ 33
4.8 $(100 + 200)$ reálný experiment 34

Závěr **43**

Seznam použité literatury **44**

Seznam obrázků **45**

A Obsah přílohy **47**

Úvod

Evoluční algoritmy se často používají na těžké optimalizační problémy. Jejich řešení trvá netriviální čas a proto tu je tlak na efektivní paralelizaci těchto algoritmů. Bohužel klasické metody paralelizace nefungují moc dobře v případech, kdy jednotlivá ohodnocení problémů trvají výrazně různou dobu. V takových případech se často musí dlouho čekat na vyhodnocení posledního řešení v generaci. Tímto čekáním samozřejmě snižujeme efektivitu využívání výpočetních zdrojů.

Tomu se snaží předcházet asynchronní paralelizace evolučních algoritmů, která ale již nevyužívá generace a jakmile existuje volný procesor, asynchronní evoluce vygeneruje nové řešení a nechá ho ohodnotit. V této verzi je tedy využití výpočetních prostředků ideální, ale tato verze algoritmu upřednostňuje řešení, která se dají rychle vyhodnotit, takže můžeme rychleji zkonvergovat k neoptimálnímu řešení.

V této práci si popíšeme možnosti paralelizace evolučních algoritmů a pokusíme se navrhnout nový paralelní algoritmus. Konkrétně se pokusíme rozšířit evoluční algoritmus s prokládáním generací, který nabízí lepší využití výpočetních zdrojů než klasické paralelní evoluční algoritmy, ale zároveň zachovává generace a je ekvivalentní sériovému evolučnímu algoritmu. Rozšíření spočívá v přidání spekulativního vyhodnocení v případě, kdy nejsou využity všechny výpočetní zdroje. Spekulativním vyhodnocením myslíme odhad fitness funkce jedince a předpočítání následujících kroků, které v případě správného odhadu později využijeme.

V sérii experimentů porovnáme algoritmus se spekulativním vyhodnocením s originální verzí a podíváme se na vliv přesnosti ve spekulativním kroku na výkon algoritmus.

Struktura textu

Tento text je členěn do čtyř kapitol. V kapitole 1 si vysvětlíme co je to evoluční algoritmus a jak funguje. Dále se v této kapitole podíváme na konkrétní typy evolučních algoritmů, na možnost jejich paralelizace a podrobněji se podíváme na evoluční algoritmus s prokládáním generací, z kterého budeme vycházet. V kapitole 2 si popíšeme základní princip fungování námi navrženého algoritmu a následně v kapitole 3 se na tento algoritmus podíváme více z implementační stránky, kde si popíšeme jednotlivé komponenty algoritmu. V poslední kapitole 4 se podíváme na výsledky experimentů ve kterých porovnáváme náš algoritmus s ostatními algoritmy z kapitoly 1.

1. Evoluční algoritmy

Evoluční algoritmy (EA) jsou optimalizační algoritmy, které se inspirují přirozenou evolucí, kterou jako první popsal Charles Darwin v knize On the Origin of Species [2] (O původu druhů [3]). Nesnaží se co nejpřesněji modelovat přirozenou evoluci, ale využívají pouze technik přirozené evoluce, jako je křížení, mutace a další. Zatímco Darwin mluví o přirozeném výběru z hlediska sebepozorování a reprodukce, evoluční algoritmy používají čistě umělá kritéria daná problémem, který optimalizují.

Evoluční algoritmus, jak je popsáno v knize Genetic Algorithms + Data Structures = Evolution Programs [5], prohledává množinu všech možných řešení optimalizačního problému. Jeden prvek této množiny je konkrétní řešení daného problému. Evoluční algoritmy většinou pracují s množinou řešení, kterou nazýváme populace, na rozdíl od ostatních optimalizačních algoritmů jako je například náhodné prohledávání, horolezecký algoritmus a další, které používají jen jedno řešení. Evoluční algoritmy často pracují v iteracích tzv. generacích. Nová generace vzniká z předchozí pomocí takzvaných genetických operátorů a selekčního tlaku. Nejčastějšími genetickými operátory jsou křížení a mutace, které si později popíšeme podrobněji. Selekčního tlaku dosahujeme pomocí fitness funkce, která ohodnotí úspěšnost daného řešení a do další generace vybereme jen ta nejlepší řešení. Pseudokód obecného EA najdeme v Algoritmu 1.

Algorithm 1 Obecný EA

Require: $P_0 \leftarrow$ počáteční populace
1: Ohodnocení jedinců v P_0
2: **while** není splněna podmínka ukončení **do**
3: Výběr rodičů
4: Rekombinace a mutace
5: Ohodnocení nových jedinců
6: Environmentální selekce vybere P_{t+1} na základě P_t a nových jedinců
7: **end while**
8: **return** P_{t+1}

Evoluční algoritmy mohou být velmi různorodé a mohou využívat různé techniky. Nyní si uvedeme ty nejčastěji používané.

Selekce

Selekce je operace, která se stará o selekční tlak, tedy o odstraňování neperspektivních jedinců. Můžeme jí rozdělit na environmentální selekci a selekci pro křížení. Environmentální selekce se stará o výběr jedinců, kteří postoupí do další generace. Selekce pro křížení vybírá jedince z populace, kteří projdou genetickými operacemi, tedy vybírá tzv. rodiče. Součástí selekce může být elitismus, který vybere jedince z populace, kteří postoupí do další generace beze změn. Tito jedinci se nadále účastní selekce pro křížení.

Selekci můžeme dělit také podle způsobu výběru:

- Uniformní selekce – náhodný výběr.
- Selekce nejlepších – vybereme n jedinců s nejlepší hodnotou fitness funkce.
- Ruletová selekce – každý jedinec má šanci na výběr podle toho, jakou má hodnotu fitness. Předpokládáme kladné hodnoty fitness funkce. Jedinec i s hodnotou fitness funkce f_i má pravděpodobnost výběru

  $$p_i = \frac{f_i}{\sum_j f_j}$$

  To si můžeme představit jako ruletu, kde každý jedinec má svou výseč, která je přímo úměrná hodnotě fitness funkce. Poté již stačí zatočit ruletou a vybrat jedince, v jehož výseči se zastaví kulička. Jedince do další generace tedy vybereme tak, že n-krát zatočíme ruletou.
- Stochasticky uniformní selekce (SUS) – je speciální případ ruletové selekce, kde zatočíme pouze jednou a dále posouváme kuličku po ruletě o jednu n-tinu a tím získáme zbývající jedince.
- Turnajová selekce – náhodně vybereme k-tici jedinců a z ní vybereme jedince s nejlepší hodnotou fitness funkce.

Křížení

Křížení, nebo-li rekombinace jedinců, je operace, která zkombinuje alespoň dva jedince s pravděpodobností p_c. Způsoby křížení se mohou opět lišit:

- Uniformní křížení – každá hodnota kódující nového jedince je hodnota z dané pozice náhodně vybrána z jedinců přistupujících ke křížení. Většinou je pravděpodobnost výběru uniformní. Tato verze uniformního křížení je diskrétní. Také existuje verze aritmetická, ve které provedeme nějakou operaci (například průměr) nad hodnotami z dané pozice z jedinců přistupujících ke křížení.
- n-bodové křížení – vybereme náhodně n pozic v jedinci, kde n je alespoň jedna. Na těchto pozicích dojde k řezu a výměně odpovídajících částí jedinců.

Mutace

Mutace je operace modifikace jedince, která probíhá s pravděpodobností p_m. Mutaci můžeme opět rozdělit podle typu:

- Zatížená mutace – vycházíme z původní hodnoty, kterou o kousek posuneme.
- Nezatížená mutace – novou hodnotu vybereme náhodně z celého spektra.

1.1 Jednoduchý genetický algoritmus

Jednoduchý genetický algoritmus [5] (Simple Genetic Algorithm) je výsledek původního návrhu genetických algoritmů, které vznikaly v 70. letech v USA. Tento model se snaží o co nejjednodušší zakódování a minimální sadu genetických operátorů, protože byl použit hlavně k teoretickému zkoumáni.

Jednoduchý genetický algoritmus kóduje jedince binárně. Používá ruletovou selekci, jednobodové křížení, které je hlavním genetickým operátorem, a zatíženou mutaci, která je v tomto případě přehození bitu a hraje spíše podřadnou roli. Původní model obsahoval i inverzi, otočení části řetězce se zachováním významu bitů, kterou se inspiroval v přírodě, ale tento genetický operátor se neukázal jako výhodný.

1.2 Evoluční strategie

Evoluční strategie [5] se zaměřují na optimalizaci funkcí, jejichž vstupy jsou reálná čísla. Využívá speciální přístup samo-adaptace, zvané také metaevoluce. Evoluční strategie kódují jedince reálnými čísly. Původně se používala pouze mutace, až později se začalo využívat i křížení. Nový jedinec je akceptován pouze, pokud je lepší než původní jedinec.

V evoluční strategii je využito uniformního křížení, které může být lokální (malý počet rodičů) nebo globální (použije všechny jedince v populaci, jako rodiče). Dále zde existují dvě verze křížení – diskrétní a aritmetické. Aritmetické vznikne použitím nějaké funkce na soubor rodičů.

Samo-adaptace se dosahuje pomocí rozdělení jedince na dvě části. První část kóduje instanci problému (parametry problému). Druhá část kóduje strategické parametry jedince ovlivňující evoluci, například rozptyl mutace tohoto jedince. Čím delší je druhá část jedince, tím specifičtěji můžeme mutaci definovat. Pokud druhá část obsahuje pouze jedno číslo, všechny parametry problému budou mít společnou odchylku definovanou právě tímto číslem v druhé části. Pokud má druhá část stejný počet parametrů jako první část, pak má každý parametr problému svou vlastní odchylku, takové mutace jsou nekorelované. Geometricky si to můžeme představit tak, že mutujeme po elipse rovnoběžné s osami. Pokud má druhá část dvojnásobek parametrů co první část, pak k mutacím po elipsách přidáváme rotace, tedy nemutujeme pouze podle os dimenzí. Tyto mutace jsou korelované a odpovídají mutaci z n-dimenzionálního normálního rozdělení, kde n je velikost první části.

První část jedince mutujeme přičtením náhodného čísla z normálního rozdělení s příslušnou odchylkou a případnou rotací. Druhou část jedince reprezentující odchylky můžeme upravovat podle úspěšnosti mutace. Pokud je méně než 20 % potomků lepších než rodiče, pak se odchylka zmenší, jinak se zvětší.

Evoluční strategie má dvě verze - plusová a čárkovaná. Čárkovaná verze, značíme (μ, λ), kde μ je velikost populace, λ je počet potomků, vybírá jedince do další generace pouze z λ potomků. Plusová verze, značíme $(\mu + \lambda)$, vybírá jedince do další generace jak z rodičů, tak z potomků, tedy z $\mu + \lambda$ jedinců. Jedná se vlastně o rozšíření čárkované verze o elitismus.

Toto značení verzí (plus a čárka) budeme využívat později i v našich algoritmech.

1.3 Paralelní evoluční algoritmy

Dosavadní algoritmy, které jsme zmínili, byly sekvenční, a s takovými algoritmy nejsme schopni plně využít potenciál dnešních procesorů, které dokáží počítat paralelně. Proto si nyní představíme několik algoritmů, které se snaží o paralelizaci genetických algoritmů. Některé z nich se chovají stejně jako sekvenční genetický algoritmus a některé odlišně. Rozdělení jsme převzali z knihy Efficient and Accurate Parallel Genetic Algorithms [6].

1.3.1 Paralelizace globálního modelu

Algoritmus provádí výpočty nad jednou populací stejně jako u obyčejného genetického algoritmu. Paralelizace je použita při ohodnocování jedinců v populaci. Ohodnocení jedinců v populaci zřejmě není na ničem závislé. S jistou opatrností můžeme paralelizovat i genetické operátory. Pokud se v každé generaci čeká na dokončení ohodnocení všech jedinců, pak se algoritmus chová stejně jako sekvenční genetický algoritmus a říkáme, že algoritmus je synchronní.

Naopak algoritmus asynchronní, popsaný také v knize Understanding simple asynchronous evolutionary algorithms [9], nečeká na dokončení vyhodnocení všech jedinců v populaci. Takovýto algoritmus se liší od sekvenčního genetického algoritmu, protože nepoužívá generace. Nový jedinec je vygenerován jakmile je nějaký jiný jedinec ohodnocen. Tato implementace má zásadní nedostatek, upřednostňuje jedince s kratším časem vyhodnocení. Tento jev popsali Scott a de Jong v článku Evaluation-Time Bias in Quasi-Generational and Steady-State Asynchronous Evolutionary Algorithms [10]. Na druhou stranu plně využije potenciál procesoru.

1.3.2 Ostrovní model

Tento model vznikl z ostrovního modelu popsaného v populační genetice, kde jsou populace relativně izolované. Odtud je převzata i terminologie. V algoritmu ostrovního modelu je populace rozdělena do několika subpopulací, tzv. kmenů, které se vyvíjí izolovaně. Jednou za čas proběhne tzv. migrace, kdy si kmeny mezi sebou vymění některé jedince.

Při migraci je potřeba určit kolik jedinců bude přesunuto a zdrojový a cílový kmen. Při přesunu bychom chtěli, aby každý kmen dostal stejné množství genetické informace. Tomuto plánu se říká topologie a existují různá propojení například do kruhu, do sítě, do hyperkrychle... Migrace je časově náročná operace (obecně chceme, aby každý kmen měl všechny jedince v populaci ohodnocené) a proto je potřeba najít správný poměr mezi diverzitou kmenů (migrace) a rychlostí běhu (paralelizace). Pokud by migrace probíhala po každé generaci, v podstatě by se jednalo o paralelizaci globálního modelu.

1.4 Evoluční algoritmus s prokládáním generací

V předchozí sekci jsme si ukázali možnosti paralelizace genetických algoritmů. Pouze synchronní verze paralelizace globálního modelu byla shodná se sekvenčním genetickým algoritmem. Oproti asynchronní verzi, která využije 100% CPU, se

u synchronní verze setkáme s kolísajícím využitím CPU. Při spuštění ohodnocení jedinců v nové generaci, využijeme CPU na 100%, ale vždy musíme počkat na dopočítání ohodnocení všech jedinců v generaci, tedy ke konci budeme využívat pouze jedno vlákno procesoru. Tento efekt je umocněn v případě, že délka výpočtu ohodnocení každého jedince trvá výrazně různou dobu. Autoři Pilát a Neruda si všimli, že někteří jedinci z nové generace mohou být generováni dříve, než je ohodnocena celá současná generace. Tuto myšlenku popisují ve svém článku Parallel evolutionary algorithm with interleaving generations [8].

Autoři popisují dva algoritmy (μ, μ) a $(\mu + \mu)$, kde značení je převzato z evoluční strategie. Tedy velikost populace i počet potomků je μ. U verze s čárkou (μ, μ) do další generace postupují pouze potomci. U verze s plusem $(\mu + \mu)$ do další generace vybíráme jedince z populace i z potomků.

Hlavní myšlenkou IGEA (evoluční algoritmus s prokládáním generací) je skutečnost, že každý jedinec závisí pouze na omezeném počtu jedinců z předchozí generace, a proto nemusíme čekat na dokončení ohodnocení všech jedinců z jedné generace, než budeme moci začít generovat jedince v další generaci a počítat jejich ohodnocení. Pilát s Nerudou popisují situaci pro binární turnajovou selekci a pouze unární a binární genetické operátory. V takovém případě jedinec závisí na maximálně čtyřech jedincích z předchozí generace.

Obě verze začínají ohodnocením jedinců z počáteční generace. Od tohoto okamžiku algoritmus reaguje pouze na nově ohodnocené jedince. Reakce začíná přiřazením fitness funkce ohodnocenému jedinci a tato informace se šíří do další generace. Propagační krok se u obou verzí trochu liší, a proto si jej popíšeme později.

Další společná část obou algoritmů je binární turnajová selekce. Všimněme si, že nemůžeme využít například ruletovou selekci nebo selekci nejlepších, protože bychom museli mít ohodnoceny všechny jedince. Turnajovou selekci rozdělíme do dvou kroků, abychom zachovali vlastnosti jako v sekvenčním genetickém algoritmu. Nejprve vygenerujeme tzv. turnajové aspiranty. Jsou to dvojice indexů jedinců, kteří budou porovnáváni v turnajové selekci. Tato část proběhne, jakmile vytvoříme novou generaci. Druhá část je samotný turnaj, který proběhne, jakmile máme ohodnocené oba turnajové aspiranty a pro křížení je vybrán lepší z aspirantů.

Genetické operátory se používají vždy u dvou po sobě jdoucích jedinců v seznamu, přičemž první jedinec je na sudé pozici. Genetické operátory se aplikují, jakmile jsou oba tito jedinci přidáni do seznamu a v případě potřeby jsou předloženi k ohodnocení.

Jedinci předložení k ohodnocení jsou zařazeni do prioritní fronty, která je tříděna podle generace, ve které byl jedinec vytvořen. Prvně jsou vyhodnoceni jedinci ze starších generací.

1.4.1 (μ, μ)-IGEA

Tato verze algoritmu je lehčí než verze plus nebo i obecná verze (μ, λ), protože všichni potomci v generaci postoupí do další generace - není zde environmentální selekce. Jinými slovy, nepotřebujeme ohodnotit potomky v generaci, abychom věděli, kteří se dostanou do další generace, ale stačí, když budeme mít ohodnocení až při provádění turnajové selekce. To má další zajímavý důsledek - některé je-

dince nemusíme vůbec ohodnocovat, protože se nikdy nedostanou do turnajové selekce. V dalších kapitolách budeme tento trik, kdy neohodnocujeme jedince, kteří se nebudou účastnit turnajové selekce, označovat jako chytrý turnaj (clever tournament).

Verze s čárkou začíná vytvořením náhodné počáteční populace. Tato generace také obsahuje seznam turnajových aspirantů. Poté jsou předloženi k ohodnocení všichni jedinci, kteří jsou na pozicích vyskytujících se v seznamu aspirantů. Poté algoritmus vstupuje do hlavní smyčky. V každé iteraci dostaneme ohodnocenou jednu fitness funkci, kterou přiřadíme příslušnému jedinci a informaci propagujeme dál.

Během propagace se nejprve vyberou všechny páry turnajových aspirantů, kteří obsahuji ohodnoceného jedince a zkontroluje se, jestli i druhý jedinec je ohodnocen, aby mohla proběhnout turnajová selekce. Vítěz turnajové selekce je umístěn do seznamu vítězů turnaje na stejný index jako byl index aspirujícího páru. Pokud jsou známy oba vítězové turnaje (rodiče), provedou se genetické operátory. Noví potomci jsou zařazeni do seznamu potomků na stejné indexy jako byli rodiče a jsou propagováni do populace následující generace. Nakonec, pokud se jedinci změnili a jsou na indexu vyskytujícím se v seznamu turnajových aspirantů, jsou předáni k ohodnocení, jinak je opět zavolána funkce propagace v nové generaci.

Zajímavostí algoritmu, jak uvádí autoři, je, že algoritmus je založený na programování řízeném událostmi, které běžné běžné implementace evolučních algoritmů nevyužívají.

1.4.2 $(\mu + \mu)$-IGEA

Verze plus je podobná verzi s čárkou, ale je nutné provést několik netriviálních změn a rozšíření. Potomci v této verzi musí být ohodnoceni před tím, než postoupí do další generace. Navíc kompletní populaci následující generace dostaneme až v moment, kdy ohodnotíme všechny potomky současné generace, tedy už není možné provést clever tournament.

Výběr jedinců do další generace je také složitější, tentokrát musíme vybrat nejlepších μ jedinců ze 2μ rodičů a potomků. Proto, abychom mohli využít prolévání generací, využijeme skutečnost, že pokud máme ohodnoceno $\kappa > \mu$ jedinců, pak víme, že nejlepších $\kappa - \mu$ jedinců z počáteční populace a potomků určitě postoupí do další generace, a proto je do další generace přidáme co nejdříve. Musíme pouze zajistit, abychom přidali pouze nové jedince do další generace (abychom nepřidali jednoho jedince vícekrát). Nový jedinec je přidán na první volnou pozici v populaci v další generaci.

Funkce propagace ve verzi plus začíná nalezením ohodnocených rodičů a potomků současné generace. Pokud je počet ohodnocených jedinců κ větší než μ, přidá se $\kappa - \mu$ nejlepších jedinců do další generace na první volné pozice v populaci, pokud ještě nebyli přidáni. Od tohoto okamžiku funkce propagace pokračuje velmi podobně jako ve verzi s čárkou. Zkontrolujeme všechny turnajové aspiranty, a pokud jsou oba uchazeči v páru již ohodnoceni, provedeme turnajovou selekci. Poté, pokud jsou oba rodiče k dispozici, jsou provedeny genetické operátory a vygenerováni noví potomci. V případě, že je potřeba ohodnotit nové potomky, jsou tito potomci přidáni do fronty k ohodnocení. Propagace je opětovně volána na

další generaci, kdykoli byl přidán jedinec do populace. A to i v případě, že je nutné vyhodnotit potomky, protože při každé změně v populaci je možné, že může být vybrán další jedinec do další generace.

2. Evoluční algoritmy s prokládáním generací a spekulativním vyhodnocením

V předchozí kapitole jsme si popsali základní modely evolučních algoritmů a techniky jejich paralelizace. Šlo o jednodušší paralelizaci globálního modelu v sekci 1.3.1 a pokročilejší paralelizaci s prokládáním generací v sekci 1.4, která navazuje na paralelizaci globálního modelu. Můžeme si všimnout, že došlo k významnému zlepšení využití výpočetních zdrojů. Ale i nadále se vyskytují situace, kdy nevyužijeme všechny výpočetní zdroje. Proto se tento nedostatek pokusíme vyřešit přidáním spekulativních výpočtů v případě, že nemáme co počítat a čekáme na dokončení nějakého delšího výpočtu fitness funkce. Tyto algoritmy se spekulativním vyhodnocením budeme označovat IGEA-SE.

Spekulativní vyhodnocení spočívá v tom, že se pokusíme odhadnout pořadí jedinců tak, jako by už měli všichni validní hodnotu fitness funkce, a poté s neohodnocenými jedinci budeme nakládat, jako bychom je již ohodnotili. S odhadem pořadí jedinců nám může pomoci náhoda a nebo surrogate model, který se snaží modelovat daný problém. Pokud pořadí odhadneme správně, předpočítané výsledky jsou správné a využijeme je. Pokud bude náš odhad špatný, počítali jsme zbytečně a výsledky musíme zahodit. V našem algoritmu budeme podporovat spekulativní vyhodnocení pouze u nespekulativních jedinců.

Kvůli zavedení spekulativního vyhodnocení musíme také zavést pojem užitečného využití procesoru. Pokud zahltíme procesor výpočtem úloh, které nevyužijeme, pak sice můžeme dosáhnout 100% využití procesoru, ale takovéto vytížení nám nepřináší žádnou hodnotu. Užitečné využití procesoru tedy bude počítáno jako vytížení procesoru nespekulativními výpočty a spekulativními výpočty, které budou využity.

2.1 (μ, μ)-IGEA-SE

Opět začneme s tou nejjednodušší verzí, tedy (μ, μ)-IGEA-SE, která navazuje na (μ, μ)-IGEA uvedené v sekci 1.4.1. Víme, že všichni potomci současné generace tvoří populaci příští generace a jejich ohodnocení potřebujeme až při provádění turnajové selekce. A právě v tomto okamžiku vstupuje do hry spekulativní vyhodnocení. Pokud máme nějaké volné procesory, zkusíme si tipnout vítěze turnaje, využijeme již zmíněnou náhodu nebo surrogate model. Pokud známe vítěze turnaje (rodiče) již pokračujeme jako doposud. Pokud známe všechny rodiče potřebné pro provedení genetických operátorů, provedeme je. Tím nám vzniknou noví spekulativní potomci, které můžeme předat k ohodnocení. Spekulativní vyhodnocení turnaje provádíme do té doby, dokud máme volné procesory, nebo dokud je to možné.

Až budeme mít ohodnoceny jedince vstupující do turnajové selekce, která byla provedena spekulativně, provedeme nespekulativní turnajovou selekci a zkontrolujeme se spekulativním výsledkem. Pokud je spekulativní turnaj validní, můžeme ho označit jako nespekulativní. Až označíme oba rodiče jako nespekulativní, pak

můžeme i potomky označit jako nespekulativní a případně tuto informaci propagovat do dalších generací. Pokud spekulativní turnaj validní není, pak odebereme potomky a vygenerujeme je znovu s vítězem nespekulativního turnaje.

2.2 (μ, λ)-IGEA-SE a $(\mu + \lambda)$-IGEA-SE

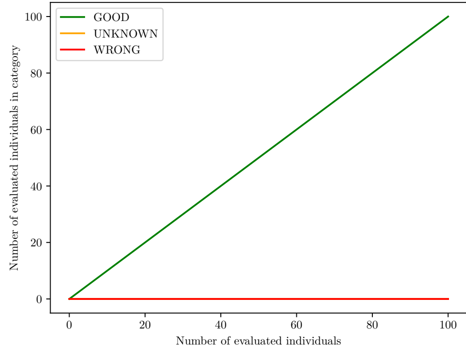
U těchto algoritmů ztratíme 2 vlastnosti z algoritmu (μ, μ)-IGEA-SE. Konkrétně to je vlastnost, že všichni potomci postupují do populace následující generace. A z toho vyplývá, že ztratíme i vlastnost chytrého turnaje, protože tentokrát dříve než potomci postoupí do další generace, musí být ohodnoceni, aby mohla proběhnout environmentální selekce. Jediný rozdíl mezi verzí plus a verzí s čárkou je, odkud budeme vybírat jedince do další generace. U verze plus je to z původní populace a potomků, u verze s čárkou jsou to pouze potomci. Tady se nám otevírá další možnost, kde využít spekulativní vyhodnocení, konkrétně budeme odhadovat, kteří potomci budou tvořit populaci následující generace. Všimněme si, že pro výběr jedinců do další populace nepotřebujeme znát přesné pořadí, ale stačí nám, abychom jedince, kteří postoupí do další generace, umístili na první místa.

Pojďme se ještě jednou podívat na skutečnost, která nám umožňuje propagovat jedince ze současné generace do další, aniž by byli všichni jedinci vyhodnoceni. V sekci 1.4.2 $(\mu + \mu)$ - IGEA říkáme, že pokud máme κ ohodnocených potomků a platí, že $\kappa > \mu$, pak víme, že $\kappa - \mu$ nejlepších jedinců postoupí do další generace. My tuto skutečnost zobecníme a rozšíříme ji o jedince, kteří určitě do další generace nepostoupí. Mějme tedy κ ohodnocených jedinců, velikost populace α (počet postupujících jedinců) a počet jedinců β, z nichž vybíráme do další generace. Ještě si označme množinu *good* jako nejlepší ohodnocení jedinci, u kterých víme, že určitě postoupí do další generace, *wrong* je množina nejhorších ohodnocených jedinců, u kterých víme, že určitě nepostoupí do další generace a konečně *unknown* je množina zbylých ohodnocených jedinců, o kterých nevíme, jestli postoupí nebo nepostoupí. Pak platí, že:

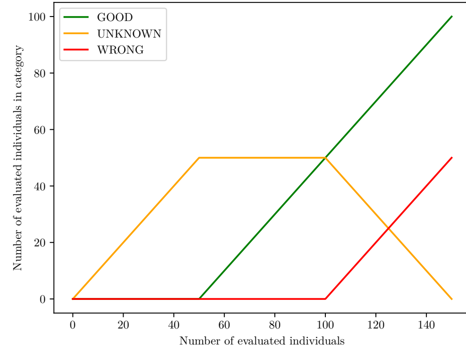
$$
\begin{aligned}
|good| &= max(\kappa - (\beta - \alpha), 0) \\
|wrong| &= max(\kappa - \alpha, 0) \\
|unknown| &= \kappa - |good| - |wrong| = min(\beta - \alpha, \alpha, \kappa, \beta - \kappa)
\end{aligned}
$$

Výpočet velikosti *good* není nic nového. Musíme mít ohodnoceno více než $(\beta - \alpha)$ jedinců, abychom dokázali s jistotou určit jedince, kteří postoupí do další generace. Pokud máme ohodnoceno $(\beta - \alpha)$ jedinců a zbylých α neohodnocených jedinců bude lepších než všichni ohodnocení jedinci, pak z ohodnocených nepostoupí žádný jedinec. Velikost množiny *wrong* neříká nic jiného, než že pokud postupuje α jedinců do další generace, tak všichni jedinci, kteří jsou na pozici mimo α nejlepších jedinců, už se mezi α nejlepších nedostanou, protože jejich umístění se může pouze zhoršit, tedy s jistotou víme, že už nepostoupí. Poslední vypočítaná hodnota je velikost množiny *unknown*, kde první část jasně vyplývá z předchozích dvou výpočtů. Pojďme si ještě rozebrat poslední část ekvivalence. První člen funkce minimum $\beta - \alpha$ dostaneme z výpočtu hodnoty $|good|$. Nemůžeme mít více jedinců v množině *unknown* než $\beta - \alpha$, protože pokud máme více než $\beta - \alpha$ ohodnocených jedinců, tak víme, že $\kappa - (\beta - \alpha)$ jich postoupí a tedy $\kappa - (\beta - \alpha) + (\beta - \alpha)$ je právě κ. Úplně stejně pokračujeme s druhým členem

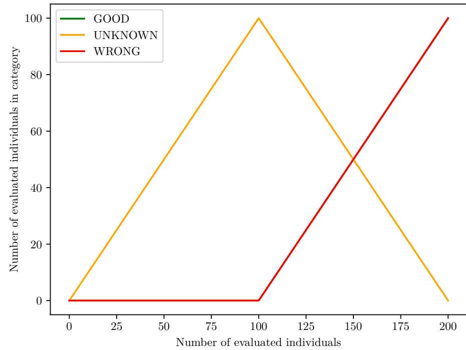
funkce minimum, jen použijeme rovnici na výpočet hodnoty $|wrong|$. Třetí člen funkce minimum je zřejmý, nemůžeme mít víc jedinců v množině *unknown*, než je počet ohodnocených jedinců κ. A konečně čtvrtý člen nám říká, že nemůže být víc $|unknown|$ jedinců než počet neohodnocených jedinců $\beta - \kappa$, protože každý neohodnocený jedinec může být lepší než všichni ostatní ohodnocení jedinci, to ale znamená, že pořadí se jedincům může zhoršit právě o $\beta - \kappa$ a tedy pouze u $\beta - \kappa$ nejhorších z nejlepších α ohodnocených si nejsme jisti, jestli zůstanou v množině *good* a nebo se přesunou do množiny *wrong*. Na obrázku 2.1 můžeme vidět vývoj jednotlivých skupin v závislosti na α, β a κ. Pro čárkovanou verzi



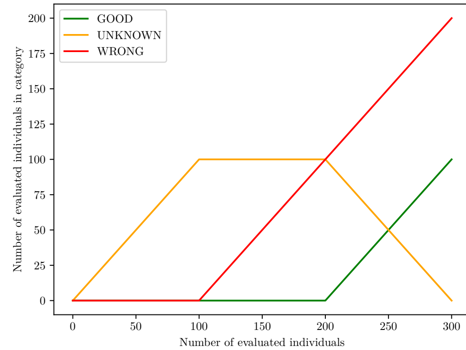
(a) $\alpha = 100, \beta = 100$



(b) $\alpha = 100, \beta = 150$



(c) $\alpha = 100, \beta = 200$



(d) $\alpha = 100, \beta = 300$

Obrázek 2.1: Na těchto grafech můžeme vidět vývoj počtu jedinců v jednotlivých kategoriích pro $\alpha = 100$ a různé nastavení β. Zeleně je označena velikost množiny *good*, žlutě *unknown* a červeně *wrong*. V grafu 2.1a je $\beta = 100$, pro toto nastavení každý jedinec postoupí, takže množiny *unknown* a *wrong* jsou vždy prázdné. V grafu 2.1b je $\beta = 150$. V grafu 2.1c je $\beta = 200$, tady máme vždy stejný počet v množině *good* a *wrong*. V posledním grafu 2.1d je $\beta = 300$.

algoritmu je $\alpha = \mu$ a $\beta = \lambda$, pro plusovou verzi je $\alpha = \mu$ a $\beta = \mu + \lambda$.

Nyní se již vraťme zpátky ke spekulativnímu vyhodnocování. Spekulativním jedincem v populaci se může stát neohodnocený potomek, nebo potomek z množiny *unknown*. Takového jedince umístíme na nějakou volnou pozici v populaci další generace. Poté už je situace stejná jako v algoritmu (μ, μ)-IGEA-SE, proběhne spekulativní turnaj a případně se vytvoří spekulativní potomci. Až bude možné zjistit, jestli spekulativní jedinec skutečně postoupí nebo ne, uděláme po-

třebné kroky. Pokud spekulativní jedinec skutečně postoupí, označíme ho jako nespekulativního, dále zrušíme označení spekulativní i u turnaje a potomků, pokud je to možné. Pokud spekulativní jedinec nepostoupí do další generace, musíme provedené výpočty zahodit.

Nyní, když už známe základní princip spekulativního vyhodnocení, a víme v kterých částech algoritmu ho můžeme využít, podíváme se podrobněji na celý algoritmus. Algoritmus pracuje v generacích a informace o každé generaci ukládá v jednoduchých strukturách jako jsou - list jedinců v populaci, list rodičů (vítězů turnaje), list potomků. Stejné struktury obsahuje i pro spekulativní část, tedy list spekulativních jedinců v populaci, list spekulativních rodičů a list spekulativních potomků. Do každé generace je také přidán list turnajových aspirantů, který je nastaven při vytváření generace a později se již nemění. List turnajových aspirantů je list náhodných dvojic indexů do listu jedinců (případně spekulativních jedinců) v populaci. Tím je zajištěno, že bez ohledu na pořadí jedinců v populaci bude výběr do turnaje nezávislý stejně jako v sekvenčním evolučním algoritmu.

Hlavní smyčka algoritmu je ukázána v Algoritmu 2. Na začátku algoritmu vytvoříme nultou generaci G_0, kde v populaci vytvoříme náhodné jedince a přiřadíme jim fiktivní velmi nízkou hodnotu fitness funkce. Do potomků přidáme počáteční populaci a předáme k ohodnocení všechny potomky, kteří nemají validní hodnotu fitness funkce. Tento krok děláme proto, abychom nemuseli řešit rozdílnost počáteční generace, která by začínala s vyhodnocením populace, s ostatními generacemi, kde jsou ohodnocováni potomci. Tím můžeme způsobit, že G_0 bude obsahovat více potomků než všechny ostatní generace, například u verze (100+1), bude mít G_0 sto potomků. Nyní již vstoupíme do hlavního cyklu. Pokud zakážeme spekulativní vyhodnocení, pak zbytek algoritmu probíhá stejně jako IGEA, tedy čekáme na vyhodnocení nějakého jedince, přiřadíme hodnotu fitness funkce danému jedinci a propagujeme výsledek. Pokud umožníme spekulativní vyhodnocení, po propagování výsledku ohodnoceného jedince, provedeme spekulativní propagaci a v případě, že máme nějaké volné procesory, zkusíme vytvořit nové spekulativní jedince v nejstarší generaci, v které je to možné. Po přidání každého jedince se opět provádí spekulativní propagace, která propaguje informace o novém spekulativním jedinci v populaci. Ačkoli logicky spekulativní část navazuje na tu nespekulativní, v algoritmu ji umístíme na začátek (s informacemi o vyhodnocení z minulé iterace cyklu), abychom nemuseli čekat na první vyhodnocení v G_0 než bude možné začít provádět spekulativní vyhodnocení.

Algoritmus 3 popisuje propagaci nově ohodnoceného nespekulativního jedince do populace další generace. Funguje obdobně jako propagace v algoritmu IGEA, akorát je potřeba ho rozšířit o úpravu spekulativní části algoritmu. Nejdříve odstraníme všechny spekulativní jedince, kteří jsou v množině *wrong*. Ta se mohla rozšířit, když jsme vyhodnotili dalšího jedince v generaci. Poté najdeme všechny jedince z množiny *good*, kteří ještě nepostoupili do další generace (pokud jsme vyhodnotili spekulativního jedince, tato množina bude prázdná a propagace končí). Pro každého takového jedince p najdeme pozici i v populaci následující generace. Pokud jde o algoritmus clever tournament, je pozice v populaci následující generace stejná jako v potomcích současné generace. V ostatních verzích algoritmu se nejdříve podíváme, jestli se jedinec nenachází na nějaké pozici ve spekulativní populaci. Pokud ano, index bude stejný jako ve spekulativní části. Pokud jedince ve spekulativní části nenajdeme, najdeme první volný index v nespekulativní

Algorithm 2 Hlavní smyčka IGEA-SE

1: $G_0 \leftarrow$ počáteční generace ▷ počáteční populace přiřazena jako potomci
2: Ohodnoť populaci G_0 malými hodnotami fitness funkce
3: Předej potřebné potomky v G_0 k ohodnocení
4: **while** není splněna podmínka ukončení **do**
5:     **if** chceme použít spekulativní vyhodnocení **then**
6:         $propagate_speculative(I.gen, G)$ ▷ Alg 4
7:         $success \leftarrow True$ ▷ Povedlo se vytvořit nového spekulativního jedince v minulé iteraci?
8:         **while** jsou volné procesory **and** *success* **do**
9:             $success \leftarrow create_speculative_parent(G)$ ▷ Alg 6
10:             $propagate_speculative(I.gen, G)$ ▷ Alg 4
11:         **end while**
12:     **end if**
13:     $I, f \leftarrow get_next_evaluated()$ ▷ Ohodnocený jedinec a jeho fitness
14:     Přiřaď fitness f jedinci I v $G_{I.gen}$
15:     $propagate(I.gen, G)$ ▷ Alg 3
16: **end while**

a zároveň i spekulativní populaci. Pokud takový index neexistuje, uvolníme jeden náhodný index ve spekulativní populaci. Poté co najdeme index i, tak do nespekulativní populace na tento index umístíme jedince p. Dále pak najdeme všechny aspirující páry (a,b), ve kterých se vyskytuje jedinec v populaci na pozici i. Pro každý takový aspirující pár zkontrolujeme, jestli máme oba jedince ohodnoceny, a pokud ano, provedeme turnajovou selekci. Následně zkontrolujeme, jestli máme stejného vítěze turnaje ve spekulativních rodičích. Pokud ano, potvrdíme ho jako nespekulativního, pokud se neshodují, odstraníme ho ze spekulativních rodičů. Nakonec zkontrolujeme, jestli máme i druhého rodiče pro provedení genetických operátorů. Genetické operátory provádíme na dvou rodičích jdoucích za sebou začínající na sudé pozici. Pokud máme oba rodičích, podíváme se do spekulativních potomků, a pokud pro dané rodiče již spekulativní potomky máme, prohlásíme je za nespekulativní. Pokud potomci ve spekulativní části nejsou, provedeme genetické operátory na rodiče a případně nové potomky předáme k ohodnocení.

Po propagaci informací v nespekulativní části se podíváme na propagaci ve spekulativní části. Tu jsme rozdělili na dvě části. První obsahuje pouze propagaci do další generace (Algoritmus 4), v druhé se provádí turnajová selekce a genetické operátory (Algoritmus 5). Algoritmus je rozdělen z toho důvodu, že spekulativního jedince do nové generace můžeme dostat dvěma způsoby. První je stejný jako v nespekulativní části, tedy propagací spekulativního potomka z předchozí generace. Druhý způsob je vytvoření spekulativního jedince z nespekulativního potomka předchozí generace, tento způsob si popíšeme později v Algoritmu 6. Poté co jedinec postoupí do další generace, už neřešíme jak vznikl, a k oběma případům se chováme stejně.

Spekulativní propagace je podobná té nespekulativní, pouze kombinuje informace z obou částí – nespekulativní i spekulativní. Nejprve najdeme všechny jedince z množiny *good*. Ve spekulativní části se tato množina vytváří z jedinců v nespekulativní i spekulativní populaci. Poté z množiny *good* odstraníme nespe-

Algorithm 3 *propagate*(*gen*,*G*)

```
1: g ← G_gen
2: n ← G_gen+1
3: odstraň všechny spekulativní jedince v množině wrong
4: S ← jedinci z množiny good, kteří ještě nepostoupili
5: for jedinec p in S do
6:     i ← select_index_in_population()    ▷ Pozice ve spekulativní části, pokud tam byl, jinak první volný index, případně uvolni náhodný index ve spekulativní části
7:     přidej jedince p do populace generace n na pozici i
8:     aps ← aspirující páry v generaci n, kde alespoň jeden z indexů je i
9:     for aspirující pár (a,b) na pozici j in aps do
10:        G_n.parents[j] ← tournament_selection(a,b)
11:        Odstraň špatné vítěze spekulativního turnaje (spekulativní rodiče)
12:        j1, j2 po sobě jdoucí indexy rodičů (obsahující j a j1 je sudý)
13:        if existují oba rodiče p1 a p2 na pozicích j1,j2 then
14:            if existuji spekulativní potomci těchto rodičů then
15:                prohlaš spekulativní potomky za nespekulativní
16:            else
17:                proveď genetické operátory
18:                pokud je to potřeba předej nové potomky k ohodnocení
19:            end if
20:        end if
21:    end for
22:    propagate(gen + 1,G)
23: end for
```

kulativní jedince a jedince, kteří již postoupili do spekulativní populace následující generace. Pro každého jedince p z této vyfiltrované množiny *good* se pokusíme najít volnou pozici ve spekulativní populaci (volná pozice musí být jak ve spekulativní populaci, tak v nespekulativní). Pokud najdeme volnou pozici, jedince na ni umístíme a zavoláme funkci, která provede turnajovou selekci a genetické operátory. Pokud volnou pozici nenajdeme, daný jedinec se do další generace nedostane. To může nastat například tehdy, když má více jedinců stejnou fitness funkci, nebo když jsme do následující generace přidali nového spekulativního jedince z nespekulativní části.

Nyní se podíváme na Algoritmus 5, který ukazuje provedení turnajové selekce a genetických operátorů. Nejprve najdeme aspirující páry, kde alespoň jeden z páru jedinců je spekulativní a ponecháme jen takové, mezi kterými neproběhl ještě turnaj, nebo tento turnaj je nevalidní. Turnaj se stane nevalidním, pokud odstraníme spekulativního jedince, který v turnaji nevyhrál. V takovém případě to vítěze turnaje (rodiče) neovlivní. Mohlo by se stát, že po nahrazení odstraněného jedince, vyhraje turnaj původní vítěz a my bychom znovu prováděli stejné výpočty, proto turnaj pouze označíme jako nevalidní a výsledky smažeme až v případě, že turnaj vyhraje jiný jedinec. Po výběru vhodných aspirujících párů proběhne pro každý tento pár turnajová selekce. V případě, kdy vítěz turnaje existoval, zkontrolujeme, zda je vítěz stále stejný. Pokud se vítěz změnil, odstra-

Algorithm 4 *propagate_speculative*(gen, G)

1: $g \leftarrow G_{gen}$
2: $n \leftarrow G_{gen+1}$
3: $S \leftarrow$ jedinci z množiny *good* (spojení spekulativní a nespekulativní populace), kteří jsou spekulativní a ještě nepostoupili do další generace
4: **for** jedinec p **in** S **do**
5:     **if** existuje volná pozice v populaci následující generace **then**
6:         $i \leftarrow$ první volná pozice v populaci (stejná pozice pokud jde o CT)
7:         přidej jedince p do spekulativní populace generace n na pozici i
8:         $propagate_speculative_cross_mutate(gen, G)$ ▷ Alg 5
9:     **end if**
10: **end for**

níme ho i jeho potomky. Poté se již nacházíme v situaci, jako by vítěz turnaje neexistoval, tedy ho pouze přidáme na danou pozici. Poté zkontrolujeme, jestli máme druhého rodiče (ať už spekulativního, nebo nespekulativního) a případně provedeme genetické operátory a nové jedince předáme k ohodnocení. Nakonec opět propagujeme informace o nově ohodnocených jedincích do další spekulativní generace.

Algorithm 5 *propagate_speculative_cross_mutate*(gen, G)

1: $n \leftarrow G_{gen+1}$
2: $aps \leftarrow$ spekulativní aspirující páry v generaci n, kde ještě neproběhl turnaj, nebo je nevalidní
3: **for** aspirující pár(a,b) na pozici j **in** aps **do**
4:     $G_n.speculative_parents[j] \leftarrow tournament_selection(a,b)$ ▷ Nový turnaj nebo aktualizace původního
5:     j_1, j_2 po sobě jdoucí indexy spekulativních nebo nespekulativních rodičů (obsahující j a j_1 je sudý)
6:     **if** existují oba rodiče (alespoň jeden spekulativní) p_1 a p_2 na pozicích j_1,j_2 **then**
7:         proveď genetické operátory
8:         pokud je to potřeba předej nové potomky k ohodnocení
9:     **end if**
10:     $speculative_propagate(gen + 1, G)$ ▷ Alg 4
11: **end for**

Nyní se podíváme ještě na funkci vytvoření nového spekulativního jedince, která je popsána v Algoritmu 6. Tuto část algoritmu využijeme v případě, že máme volné procesory a již nemáme žádného dalšího jedince, kterého bychom mohli ohodnotit. Nejprve najdeme všechny generace, které nejsou kompletně ohodnoceny a seřadíme je sestupně podle stáří. Nejprve chceme spekulativně ohodnotit jedince z nejstarších generací. V každé takové generaci najdeme neohodnocené potomky a potomky z množiny *unknown*. V případě, že se jedná o verzi clever tournament, tak odstraníme všechny jedince, kteří se nezúčastní turnajové selekce v následující generaci. Poté jedince předáme surrogate modelu, který nám vrátí jedince, kterého přidáme do spekulativní populace následující

generace. Poté zavoláme funkci pro provedení turnajové selekce a provedení genetických operátorů ve spekulativní části popsané v Algoritmu 5. Pokud jsme přidali nového spekulativního jedince, vrátíme hodnotu $True$, jinak vracíme $False$.

Algorithm 6 *create_speculative_parent*(G)

1: *G_not_completed* ← indexy generací seřazeny vzestupně, kde nejsou ohodnoceni všichni potomci a v následující generaci existuje volný index v populace a spekulativní populaci
2: **for** *gen* **in** *G_not_completed* **do**
3: $g \leftarrow G_{gen}$
4: $n \leftarrow G_{gen+1}$
5: *ps* ← neohodnocení potomci a jedinci ze skupiny *unknown*, kteří ještě nejsou spekulativně vyhodnocováni
6: **if** clever tournament **then**
7: odeber z *ps* jedince, kteří nejsou v aspirujících párech
8: **end if**
9: *sp* ← surrogate model nám vrátí nové speculative parents z *ps*
10: **if** clever tournament **then**
11: umísti *sp* na původní pozici v potomcích
12: **else**
13: umísti *sp* na první volný index
14: **end if**
15: *propagate_speculative_cross_mutate*(*gen*, G) ▷ Alg 5
16: **return** True
17: **end for**
18: **return** False

Nyní, když jsme podrobněji popsali celý algoritmus, je potřeba zmínit, že samotná implementace podporuje ve spekulativní části více jedinců na jedné pozici, což nezmění logiku algoritmu, ale zkomplikuje samotnou implementaci. Tedy například turnaj může vyhrát více než jeden jedinec. Podporu více jedinců na jedné pozici přidáváme, kvůli zvýšení pravděpodobnosti správného výběru ve spekulativní části.

Abychom byli schopni podporovat více jedinců na jedné pozici, musíme do generací přidat další struktury, které udržují přechody mezi populací, rodiči a potomky. Samotná implementace podporuje takovéto rozvětvení pouze v turnajové selekci a částečně při vytváření spekulativního jedince. Při vytváření spekulativního jedince je zde omezení, že vybíráme pouze z jedinců, kteří ještě nejsou spekulativně vyhodnocováni, tedy pokud bychom chtěli vybrat více než $\frac{\lambda}{\mu}$ jedinců ($\frac{\mu+\lambda}{\mu}$ pro verzi plus), zůstávala by nám volná místa ve spekulativní populaci následující generace, ale již bychom neměli žádného jedince, kterého bychom tam mohli přidat. Implementace také nepodporuje vytvoření nového spekulativního jedince na pozici, kde je spekulativní jedinec postupující z předchozí generace a opačně.

Algoritmus máme ve třech verzích, které se liší politikou ukončování běžících spekulativních vyhodnocení. Když předáme jedince k ohodnocení, je zařazen do fronty, z této fronty ho můžeme kdykoli odstranit, ale pokud tohoto jedince začneme vyhodnocovat vstupuje do hry právě zmíněná politika. V prvním případě

můžeme ohodnocování spekulativního jedince kdykoli ukončit. V druhém případě naopak vyhodnocení spekulativního jedince zastavit nemůžeme. Poslední případ povoluje ukončení ohodnocování pouze v případě, že jsme si již jistí, že vyhodnocovaný jedinec patří do množiny $wrong$.

3. Implementační podrobnosti

V této kapitole se podíváme na jednotlivé komponenty, ze kterých se skládá implementace algoritmu IGEA-SE.

3.1 Task

Pro lehčí práci a udržování jedinců předaných k ohodnocení vytvoříme novou strukturu $Task$. Tato struktura obsahuje nejen jedince předaného k ohodnocení, ale i informaci, zda se jedná o spekulativního jedince, v jaké generaci se nachází, čas začátku výpočtu, čas konce výpočtu a případně čas přerušení výpočtu. Dále také obsahuje informaci, na kterém procesoru proběhl výpočet a konečně i ohodnocení jedince.

Díky těmto informacím dokážeme určit prioritu takového tasku a víme v jakém pořadí je začít počítat. Priorita je n-tice atributů, kde nejnižší hodnota značí nejvyšší prioritu. Tato n-tice se skládá z informace o spekulativitě, pořadí generace a id tasku.

3.2 Paralelní prostředí

Pro otestování algoritmu si musíme připravit dvě jednoduchá paralelní prostředí. První prostředí bude pouhý simulátor, kde budeme moci vyzkoušet nějaké teoretické vlastnosti algoritmu. Druhé prostředí bude sloužit pro reálné výpočty. Chceme, aby obě tato prostředí měla stejný interface. Konkrétně chceme, aby každé paralelní prostředí umělo ohodnotit zadaný task, přerušit ohodnocování zadaného tasku, vrátit výsledek dalšího dopočítaného tasku, aktuální čas paralelního prostředí a provést úklid na konci algoritmu.

3.2.1 Simulátor

Simulátor implementuje simulaci paralelního prostředí, které při vytváření dostane zadaný počet procesorů, fitness funkci a časovou funkci, která ohodnocuje délku výpočtu fitness funkce. Při předložení tasku k ohodnocení nastavíme počáteční čas řešení na aktuální čas simulátoru, přidělíme tasku procesor, na kterém se bude provádět výpočet, ohodnotíme fitness funkci a pomocí časové funkce určíme konec výpočtu tasku. Takto připravený task přidáme do minimové haldy běžících procesů, která je řazena podle času ukončení tasků. Při přerušení výpočtu tasku najdeme tento task v běžících procesech, odtud ho odstraníme, opravíme haldu, uvolníme procesor, na kterém probíhal výpočet, nastavíme čas přerušení a důvod přerušení. Při dotazu na další dokončený task vybereme z haldy běžících tasků minimum, uvolníme procesor a nastavíme aktuální čas simulátoru na čas ukončení tasku.

3.2.2 Reálný výpočet

Druhé paralelní prostředí nám umožňuje provádět skutečné výpočty. Při vytváření tohoto prostředí nastavíme počet procesorů a fitness funkci. Při předložení

tasku k ohodnocení (Algoritmus 7) nastavíme počáteční čas řešení na aktuální čas paralelního prostředí, přidělíme tasku procesor a vytvoříme nový proces, který začne počítat danou fitness funkci pro daného jedince. Do listu aktuálně běžících tasků přidáme dvojici s procesem a taskem. Při přerušení tasku (Algoritmus 8) najdeme tento task v listu aktuálně běžících tasků, z tohoto listu ho odstraníme, přerušíme proces, procesor na kterém probíhal výpočet přidáme do volných procesorů, nastavíme čas a důvod přerušení. Nakonec ukončený task vrátíme. Při dotazu na další dokončený task (Algoritmus 9), nejprve zkontrolujeme, jestli alespoň jeden task je aktuálně počítán a jestli žádný výpočet fitness funkce neskončil chybou. Pokud tato podmínka není splněna, ukončíme celý běh algoritmu, protože ohodnocení jedince, který skončil chybou, již nikdy nedostaneme, a tedy algoritmus by se v tomto místě později zasekl, protože by se mohl dostat do situace, že již máme vše ohodnocené a pouze čekáme na dokončení tohoto tasku. Po této kontrole se podíváme do fronty s výsledky fitness funkce, která je sdílena všemi procesory a po skončení výpočtu do ní proces zapíše výsledek. Pokud tato fronta není prázdná, vybereme první ohodnocený task. Tento task odstraníme z listu běžících tasků, uvolníme procesor, na kterém probíhal výpočet, a task vrátíme. Pokud je fronta prázdná, čekáme dokud se v ní nějaký výsledek neobjeví. Při čekání na ohodnocení jedince musíme stále kontrolovat, jestli nějaký výpočet neskončil chybou, protože by se mohlo stát, že všechny výpočty skončí chybou a tedy se ve frontě s výsledky nikdy žádný výsledek neobjeví.

Algorithm 7 $pool.apply_async(task)$

1: $task.start_time \leftarrow$ aktuální čas poolu
2: $task.cpu \leftarrow$ nějaký volný procesor
3: $process \leftarrow$ proces počítající fitness funkci pro daný $task$
4: přidej dvojici $(process, task)$ do běžících tasků

Algorithm 8 $pool.terminate(task, terminate_type)$

1: **if** $task$ je v běžících tascích **then**
2: $\quad process, running_task \leftarrow$ položka z běžících tasků, kde $task = running_task$
3: $\quad$ odeber $(process, running_task)$ z běžících tasků
4: $\quad$ uvolni procesor, na kterém se počítal $running_task$
5: $\quad process.terinate()$
6: $\quad running_task.terminate_type \leftarrow terminate_type$
7: $\quad running_task.terminate_time \leftarrow$ aktuální čas poolu
8: $\quad$ **return** $running_task$
9: **else**
10: $\quad$ **return** $None$
11: **end if**

Tento paralelní simulátor slouží pouze k účelům testování algoritmu a není optimalizován. Vzhledem k tomu, že v CPythonu existuje GIL (global intepreter lock) [11], kvůli kterému nemůžeme používat vlákna, protože by výpočet stále běžel sekvenčně, musíme využít procesy. V tomto případě pro každý výpočet fitness funkce startujeme nový proces, což je velice náročná operace a minimálně

Algorithm 9 $pool.next_finished()$

1: **repeat**
2: **if** počet položek v běžících tascích je 0 **then**
3: skonči s chybou „Neběží žádný task!"
4: **end if**
5: **for** $(process, running_task)$ **in** běžící tasky **do**
6: **if** $process$ skončil **and** $process.exitcode \neq 0$ **then**
7: skonči s chybou„Výpočet fitness skončil chybou!"
8: **end if**
9: **end for**
10: **until** není vyhodnocen nějaký task
11: $task \leftarrow$ první task ve frontě vyhodnocených tasků
12: $process, running_task \leftarrow$ položka z běžících tasků, kde $task = running_task$
13: odeber $(process, running_task)$ z běžících tasků
14: uvolni proces na kterém běžel $running_task$
15: **return** $task$

tato část by mohla být optimalizována. Naším hlavním cílem ale nebylo napsat efektivní paralelní prostředí, ale prostředí, ve kterém budeme moci otestovat náš algoritmus.

3.3 Plánovač

Plánovač je část programu, která se stará o řízení ohodnocování jedinců. Když jsme při popisu algoritmu hovořili o předání jedince k ohodnocení, nebo o odstraňování ve spekulativní části, vždy jsme komunikovali s touto komponentou. Plánovač udržuje všechny tasky k ohodnocení, má vazbu na paralelní simulátor a informaci o tom, které tasky smějí být přerušeny. Plánovač také udržuje aktuálně zpracovávané tasky a všechny dokončené tasky, které jsou poskytovány jako trénovací množina pro surrogate model.

Plánovač umožňuje přidat task, odstranit task a zeptat se na další dokončený task. Ještě než si ukážeme konkrétněji, co se při jednotlivých akcích děje, popíšeme si, jak plánovač předává tasky paralelnímu prostředí (Algoritmus 10). Tato metoda na začátku zkontroluje, jestli mohou být ukončovány spekulativní tasky v případě, že v prioritní frontě se nachází nespekulativní task. Pokud je možné spekulativní tasky v tomto případě ukončit, jsou postupně ukončovány spekulativní tasky s nejnižší prioritou, dokud je nějaký spekulativní task vyhodnocován v paralelním prostředí, nebo dokud nemáme dostatečný počet volných procesorů pro nespekulativní tasky. Poté je předáno co nejvíce tasků s nejvyšší prioritou k ohodnocení v paralelním prostředí. Přidání tasku do plánovače (Algoritmus 11) je velice jednoduché, pouze se přidá nový task do prioritní fronty a poté se zavolá metoda pro předání tasků do paraleního prostředí. Odstranění tasku z plánovače popsané v Algoritmu 12 je o trochu složitější. Pokud se ještě task nezačal vyhodnocovat, pouze ho odstraníme z prioritní fronty. Pokud je task již vyhodnocován, musíme zkontrolovat nastavení algoritmu, jestli tento task můžeme ukončit, a v případě splnění podmínek pro ukončení, daný task ukončíme

Algorithm 10 *planner.start_tasks*()

1: **if** můžeme ukončovat spekulativní tasky, pokud je nespekulativní ve frontě **then**
2: $abortable_tasks \leftarrow$ běžící spekulativní tasky, které nejsou v množině *good*
3: **while** $|abortable_tasks| > 0$ **and** počet volných procesorů < počet nespekulativních tasků ve frontě **do**
4: $min_priority_task \leftarrow$ task z $abortable_tasks$ s nejnižší prioritou
5: $planner.remove(min_priority_task)$ ▷ Alg. 12
6: odeber $min_priority_task$ z $abortable_tasks$
7: vytvoř nový task podle $min_priority_task$ a přidej ho do fronty
8: **end while**
9: **while** máme volné procesory **and** máme tasky k ohodnocení **do**
10: $task \leftarrow$ task k ohodnocení s nejvyšší prioritou
11: přesuň *task* do běžících procesů
12: $pool.apply_async(task)$ ▷ Alg.
13: **end while**
14: **end if**

a odebereme z listu aktuálně zpracovávaných tasků. Poté opět zavoláme metodu na předání tasků k vyhodnocení do paralelního prostředí. Nakonec se dostá-

Algorithm 11 *planner.add*(*task*)

1: přidej *task* do fronty
2: *planner.start_tasks*() ▷ Alg. 10

Algorithm 12 $planner.remove(task, terminate_type)$

1: **if** *task* je spekulativní **and** není v množině *good* **then**
2: **if** *task* je ve frontě **then**
3: odeber *task* z fronty
4: **else if** *task* je aktuálně vyhodnocován **then**
5: **if** nastavení algoritmu dovoluje odebrat tento spekulativní task **then**
6: odeber *task* z běžících tasků
7: $pool.terminate(task, terminate_type)$ ▷ Alg.
8: **end if**
9: **end if**
10: **end if**
11: *planner.start_tasks*() ▷ Alg. 10

váme k získání ohodnoceného tasku (Algoritmus 13). V této metodě vybereme první task z fronty dokončených tasků (případně čekáme na první ohodnocený). Zkontrolujeme, jestli jsme tento task nepřerušili. Kontrolu provádíme proto, že může nastat situace, kdy task je již dokončen, ale ještě nedošlo k jeho zpracování, tudíž se pro plánovač jeví jako stále běžící. V tento okamžik přijde příkaz na přerušení tohoto tasku, ale protože výsledek je již ve frontě a proces skončil, tak přerušení na tomto procesu již nic nezmění. Pokud tedy tato situace nastane, task dále nezpracováváme a vybereme další task z fronty. U tasku aktualizujeme

informaci o spekulativitě, ta se mohla změnit v průběhu výpočtu z *unknown* na *good* nebo *wrong*. Tento task přesuneme z listu aktuálně zpracovávaných tasků do dokončených tasků a opět zavoláme metodu na předání tasků do paralelního prostředí.

Algorithm 13 $planner.next_finished()$

1: **repeat**
2: $task \leftarrow pool.next_finished()$
3: **until** *task* není v běžících tascích plánovače
4: *task.speculative* ← aktuální hodnota spekulativity tasku v plánovači
5: přesuň *task* z běžících tasků do dokončených tasků
6: $planner.start_tasks()$ ▷ Alg. 10
7: **return** *task*

3.4 Surrogate model

Tato komponenta nám pomáhá odhadnout, kteří jedinci vypadají perspektivně a má smysl je začít spekulativně vyhodnocovat. Protože surrogate model se může pro každý typ úlohy lišit, připravili jsme interface, který musí každý surrogate model implementovat. Tento interface obsahuje tři metody — odhad postupujícího do další generace z vybraného seznamu jedinců, odhad vítěze turnaje a natrénování modelu pomocí již ohodnocených jedinců.

My jsme implementovali několik surrogate modelů, které jsou velmi podobné, liší se pouze v metodě, která se stará o řazení jedinců a v trénování modelu. Odhad postupujících jedinců do další generace probíhá tak, že zavoláme metodu, která nám seřadí jedince a poté vrátíme zadaný počet nejlepších jedinců. V metodě, která odhaduje vítěze turnaje, nejprve zkontrolujeme, jestli je fitness funkce všech jedinců validní. Pokud všichni jedinci mají validní fitness funkci, seřadíme je podle jejich fitness funkcí. Pokud nějaký jedinec nemá validní fitness funkci, necháme jedince seřadit pomocí metody v surrogate modelu. Nakonec vrátíme zadaný počet vítězů turnaje.

Nyní se již pojďme podívat na konkrétní implementace surrogate modelů. Jako první si uvedeme surrogate model, který použijeme v kombinaci s paralelním simulátorem a na kterém budeme moci simulovat, jak bude algoritmus ovlivněn kvalitou surrogate modelu. Tomuto modelu nastavíme pravděpodobnost s jakou vybere lepšího ze dvou jedinců. Metoda, která nám seřadí jedince, pak probíhá následujícím způsobem. Spočítáme fitness funkci každého jedince a seřadíme od nejlepšího po nejhoršího, tento seřazený list nazvěme $sorted_ffs$. Poté začneme postupně vytvářet seznam seřazený podle surrogate modelu tak, že procházíme $sorted_ffs$ od nejlepšího po nejhoršího a vždy se zeptáme, jestli je tento konkrétní jedinec ten nejlepší. Pokud je náhodné číslo menší než pravděpodobnost úspěšného výběru, pak tohoto jedince zvolíme jako nejlepšího a přesuneme ho z $sorted_ffs$ do seznamu seřazeného podle surrogate modelu a začínáme opět na indexu 0 v $sorted_ffs$. Pokud je náhodné číslo větší, pokračujeme na dalšího jedince v seznamu. Trénování tohoto modelu nedává smysl, takže v tomto modelu tato metoda nic nemění. Všimněme si, že není důležité vybrat nejlepšího jedince do další generace, stačí nám vybrat jedince, který nakonec skončí v množině

good. Naopak v turnajové selekci, pokud vybíráme jednoho vítěze, potřebujeme mít tento odhad správný.

Další surrogate model již lze použít v obou paralelních prostředích. Tento surrogate model nevyužívá žádný statistický model, ale rozhoduje se podle informací, které má. Konkrétně jedince s validní fitness funkcí seřadí podle fitness funkce a jedince s nevalidní fitness funkcí zařadí nakonec. V tomto případě trénování modelu také nedává smysl a metoda nic nemění.

Poslední dva surrogate modely využívají statistické modely z balíčku scikit-learn [7]. První surrogate model využívá regresní modely a snaží se odhadnout přímo fitness funkci jedinců a podle ní pak jedince seřadit. Druhý surrogate model využívá klasifikační modely. V tomto případě se snažíme natrénovat komparátor. Tedy statistický model dostane na vstupu dva jedince a zařadí je do jedné ze tří kategorií – větší než, menší než, rovno. U těchto surrogate modelů již využíváme trénování modelu, které probíhá v zadaných intervalech podle velikosti trénovací množiny. V případě trénování klasifikačního modelu omezujeme velikost trénovací množiny, protože počet dvojic vytvořených z ohodnocených jedinců roste kvadraticky a trénování by trvalo dlouho.

3.5 Transitions

Vzhledem k tomu, že ve spekulativní části algoritmu můžeme mazat jedince, musíme být schopni zjistit, jak jedinec vznikl. V případě, že bychom měli jen jednoho jedince na jedné pozici, nebyla by situace až tak komplikovaná, ale protože umožňujeme ve spekulativní části algoritmu mít více jedinců na jedné pozici, připravili jsme si strukturu Transition, která bude uchovávat tyto informace. Ve struktuře transition najdeme zdroj jedince (spekulativní nebo nespekulativní, populace nebo potomek), pozice ve zdroji a pořadí na této pozici. Dále pak obsahuje cílovou pozici a pořadí na této pozici. Tyto informace ukládáme v tzv. přechodových tabulkách, což jsou listy nebo množiny s těmito strukturami.

V této struktuře jsme implementovali metodu pro vyhledávání přechodů v kolekcích, kde navíc můžeme zadat masku, podle které se má hledat. Například můžeme chtít hledat pouze podle zdroje a pozice ve zdroji, nebo naopak pouze podle cílového umístění. To se nám hodí při odstraňování jedinců v populaci, protože víme odkud pochází, ale neznáme jejich cíl. Naopak při odstraňování potomků potřebujeme zjistit všechny rodiče, ze kterých potomek vznikl, abychom mohli tyto přechody odstranit.

3.6 Generace

Generace je datová struktura, která uchovává všechny potřebné informace o generaci. Můžeme ji rozdělit na spekulativní a nespekulativní část. V nespekulativní části je to poměrně jednoduché, máme tu pole jedinců v populaci velikosti μ, pole vítězů turnaje (rodičů) a pole aspirujících párů velikosti $\lambda + (\lambda \bmod 2)$. Vzhledem k tomu, že dva potomky generujeme ze dvou rodičů, tak v případě lichého λ musíme vytvořit o jednoho rodiče více a následně z posledních dvou potomků využijeme pouze jednoho. Dále v generaci máme pole potomků velikosti λ. Pro clever tournament se nám bude hodit množina aspirantů, to je množina obsa-

hující indexy jedinců z populace, kteří se účastní turnaje. Dále generace obsahuje množinu *in_pop* s transition strukturami, které ukazují, který jedinec z předchozí generace postoupil do současné generace a na jakou pozici. Poslední množina v nespekulativní části je množina *waitings* obsahující indexy potomků u kterých se čeká na ohodnocení.

Spekulativní část obsahuje stejné struktury jako nespekulativní, akorát struktury aspirativních párů a aspirantů jsou společné a struktura *waitings* obsahuje trojice s pozicí potomka, pořadí na pozici a task. V ostatních strukturách, které jsou reprezentovány poli se nenachází jedinci, ale pole jedinců, abychom mohli mít více spekulativních jedinců na jedné pozici. Kvůli této vlastnosti a také kvůli tomu, že spekulativní jedinci nemusí být ohodnocení, musíme přidat několik dalších struktur – přechodových tabulek. Přechodová tabulka k turnaji, abychom věděli, kteří jedinci postoupili z turnaje a na které pozice. K turnaji také potřebujeme validační tabulku, abychom při odstranění jednoho jedince z turnaje nemuseli odstranit celý turnaj. Další je množina přechodů od rodičů k potomkům. Nakonec musíme ještě přidat množinu přechodů pro nové spekulativní jedince *speculative*.

3.7 IGEA-SE

V této části již nebudeme probírat celý algoritmus jako celek, ale podíváme se na některé části, které jsou v pseudokódech v kapitole 2.2 zmíněny jen slovně, ale jejich komplexnost je daleko větší než se na první pohled může zdát. Ještě zmíníme, že implementace je postavena na frameworku DEAP [4] a snažíme se o co největší kompatibilitu s funkcemi napsanými v tomto frameworku.

V algoritmu 3 $propagate(gen, G)$ na řádku 15 uvádíme, že prohlásíme spekulativní potomky za nespekulativní. Za touto větou je několik operací, které musíme provést. Nejdříve musíme z přechodové tabulky od rodičů k potomkům zjistit, jestli existují potomci pro tyto rodiče. Pokud ano, tak vezmeme všechny přechody od jednoho z rodičů do potomků. Všimněme si, že nám stačí jen jeden rodič, protože bude odkazovat na oba potomky a i naopak z jednoho potomka můžeme dostat oba rodiče. Každého potomka, jehož umístění jsme získali z přechodové tabulky, překopírujeme do nespekulativních potomků. Poté zkontrolujeme, jestli u tohoto potomka nečekáme na vyhodnocení. Pokud ano, přidáme index potomka do množiny *waitings* v nespekulativní části. Příslušný task k tomuto jedinci zařadíme do spekulativní skupiny *good*. Pokud na vyhodnocení potomka nečekáme, znamená to, že jeho ohodnocení již máme, tudíž už mohl postoupit do další generace. Musíme tedy zkontrolovat, jestli se přechod s indexem potomka nenachází v spekulativní množině *in_pop*. Pokud ho tam najdeme, odstraníme ho odtud a přidáme nový přechod do množiny *speculative*, který již nebude vycházet ze spekulativních potomků, ale z nespekulativních. Nakonec rekurzivně odstraníme všechny zbylé informace o přechodu mezi rodiči a potomky ve spekulativní části. Všimněme si, že odstraňování se zastaví u potomků, kvůli tomu, že jsme odstranili přechod do další generace ve spekulativní části.

V algoritmu 4 $propagate_speculative(gen, G)$ na řádku 3 vybíráme spekulativní množinu *good*. Výběr této množiny je přímočarý, pokud povolujeme pouze jednoho spekulativního jedince na jedné pozici. V takovém případě pouze spojíme spekulativní a nespekulativní potomky a dále postupujeme stejně jako když vy-

tváříme množinu *good* v nespekulativní části. Pokud ovšem máme více potomků na jedné pozici situace se změní. Musíme si položit otázku, kteří jedinci budou postupovat a kam zařadíme pozice, kde jsou namíchaní vyhodnocení a nevyhodnocení jedinci, jestli k vyhodnoceným, nebo nevyhodnoceným pozicím. Pokud se podíváme na otázku, kteří jedinci postoupí, máme několik možností. První může být, že z každé pozice vezmeme nejlepší fitness a klasicky vytvoříme množinu *good*. Do další generace pak postoupí všichni jedinci s fitness vyšší než je požadované minimum. To znamená, že děláme i prořezávání na úrovni jednotlivých pozic. V tomto případě se můžeme rozhodnout, kam zařadíme částečně vyhodnocené pozice. Tento přístup používáme v našem algoritmu s tím, že částečně ohodnocené pozice řadíme k nevyhodnoceným. Další možností může být, že vezmeme průměrnou fitness pozice a provedeme klasické vytvoření množiny *good*, poté postoupí všichni jedinci na pozicích s minimální průměrnou fitness. Další možnost může být, že místo průměru vezmeme minimum a takovýchto možností bychom našli jistě daleko více.

V algoritmu 5 $propagate_speculative_cross_mumate(gen, G)$ na řádku 4 provádíme nový turnaj nebo aktualizaci původního. Pokud jde o nový turnaj není na něm nic těžkého, vytvoříme vítěze turnaje podle surrogate modelu a vytvoříme přechody, abychom věděli odkud vítězové pochází. Pokud jde o aktualizaci, tak opět provedeme turnaj pomocí surrogate modelu, poté musíme zkontrolovat, jestli nějaký vítěz současného turnaje zvítězil i v předchozím turnaji, takového jedince umístíme na jeho původní pozici. Pokud v předchozím turnaji vyhráli jedinci, kteří v současném turnaji nevyhráli, tak je rekurzivně odstraníme a přečíslujeme pořadí v přechodech původních vítězů, kteří zůstávají. Poté již přidáme nové vítěze za ty, kteří zůstali z posledního turnaje a nastavíme jejich přechody.

Dále na řádku 7 provádíme genetické operátory. Zde je situace obdobná jako u turnaje, pokud na pozicích nejsou žádní potomci, provedeme klasické genetické operátory. Pokud máme alespoň na jedné pozici více rodičů, uděláme kartézský součin jedinců na jednotlivých pozicích a u nich provedeme genetické operátory. A také vytvoříme přechody od rodičů k potomkům. Nové potomky, kteří nejsou ohodnoceni, předáme k ohodnocení. Pokud jsme turnaj pouze aktualizovali a například přidali dalšího jedince k rodičům na jedné pozici, je to opět komplikovanější. Musíme vytvořit nové potomky, již neprovádíme kompletní kartézský součin, ale jen relevantní část. Tyto nové potomky zařadíme k původním na konec a vytvoříme k nim příslušné přechody.

Nakonec se podíváme na odstraňování spekulativních jedinců. Toto odstraňování probíhá rekurzivně od zadaného jedince přes další struktury v generaci až do dalších generací, tak abychom odstranili všechny jedince, kteří jsou na původním jedinci nějak závislí. My si zde uvedeme princip jen na dvou tabulkách, pro ostatní tabulky je princip podobný. V pseudokódu budeme používat funkci $in_collection$ pro vyhledávání přechodů popsanou v sekci 3.5 u které budeme používat parametry typu $Mask.TYPE$, kde $TYPE$ může být $SOURCE$, to znamená, že vyhledáváme přechody se stejným zdrojem jedince a stejnou zdrojovou pozicí, nebo to může být S_SOURCE to znamená, že vyhledáváme přechody se stejným zdrojem jedince, stejnou pozicí i stejným pořadím. Obdobně pro $DESTINATION$, kde vyhledáváme podle cílové pozice a $S_DESTINATION$, kde vyhledáváme podle cílové pozice a cílového pořadí. Začneme tím nejjednodušším, odstranění spekulativního jedince, který se přesunul do množiny *wrong* (Algoritmus 14).

Nejprve odstraníme přechod z nespekulativních potomků (případně i z populace u verze plus) do spekulativní populace následující generace. Poté zavoláme odstranění jedince z populace. Nakonec musíme vyhledat všechny přechody, které ukazují na pozici, ze které jsme odstranili spekulativního jedince a snížit o jedničku pořadí na cílové pozici u přechodů ukazujících za odebraného jedince.

Algorithm 14 $remove_speculative(gen, G, transition)$

1: $G_{gen}.speculative.remove(transition)$
2: $remove_population(gen, G, transition)$ ▷ Alg. 15
3: $T \leftarrow transition.in_collection(G_{gen}.speculative, Mask.DESTINATION)$ ▷ všechny přechody, které ukazují na stejnou cílovou pozici
4: **for** t **in** T **do**
5:     **if** $t.destination_list_index > transition.destination_list_index$ **then**
6:         $t.destination_list_index -= 1$ ▷ Snížíme pořadí u jedinců, kteří byly za odstraněným jedincem
7:     **end if**
8: **end for**

Nyní si popíšeme ještě odstranění z populace (Algoritmus 15). Jedince ve spekulativní populaci odstraníme pomocí přechodu, kde využijeme cílovou pozici a pořadí na této cílové pozici. Pro verzi plus mohlo nastat, že jedinec postoupil přímo do další generace, proto zkusíme najít v přechodové tabulce *in_pop* záznam o tomto přechodu. Pokud ho najdeme, jedinec postoupil, tak ho musíme odstranit. Poté musíme opět přečíslovat pořadí v přechodech v přechodové tabulce *in_pop*, které vedou ze stejné pozice s pořadím za odstraněným jedincem. Následně projdeme všechny aspirující páry (a,b). Pozice aspirujícího páru je i. V případě, že je odstraněný jedinec v aspirujícím páru, zkontrolujeme, jestli na pozici odstraněného jedince zůstal nějaký další jedinec, pokud ne, prohlásíme turnaj na pozici i za nevalidní. Dále zkontrolujeme, jestli odstraněný jedinec turnaj vyhrál. Pokud ano, musíme ho opět rekurzivně odstranit. Pokud nevyhrál, turnaj zůstává nezměněn, protože v příštím turnaji může vyhrát současný vítěz a my bychom opakovali stejný výpočet, proto nám prozatím stačí pouze označení nevalidní. V případě že do turnaje vstoupí jedinec, který současného vítěze porazí, pak dojde k odstranění původního vítěze. Poté musíme opět upravit pořadí v přechodech pro turnaj.

Podobně bychom postupovali i pro další spekulativní tabulky. Za zmínku stojí ještě odstranění přechodů od rodičů k potomkům, kde při odstranění jednoho rodiče musíme odstranit sudý počet potomků (neplatí pro poslední lichou pozici v potomcích) a tito potomci závisí na dvou rodičích, tedy musíme odstranit všechny přechody vedoucí do potomků odstraněného jedince. Proto se na první pohled může zdát, že odstraňujeme přechody, které s odstraněným jedincem nesouvisí.

Algorithm 15 *remove_population*$(gen, G, transition)$

```
1: di ← transition.destination_index
2: dli ← transition.destination_list_index
3: del G_gen.speculative_population[di][dli]  ▷ Odstraň jedince ze spekulativní populace podle cílové pozice dodaného přechodu
4: if Pozice G_gen.speculative_population[di] obsahuje prázdný seznam then
5:     valid_tournament = False
6: else
7:     valid_tournament = True
8: end if
9: transition ← Transition(S_PARENT, di, dli)  ▷ Aktualizace přechodu pro aktuálně odstraněného jedince
10: if transition.in_collection(G_gen.speculative_in_pop, Mask.S_SOURCE) then
11:     remove_in_pop(gen + 1, G, transition)  ▷ Verze plus - jedinec postoupil přímo do další generace, musíme ho odstranit
12: end if
13: T ← transition.in_collection(G_gen+1.speculative_in_pop, Mask.SOURCE)  ▷ všechny přechody, které ukazují na stejnou zdrojovou pozici
14: for t in T do
15:     if t.source_list_index > transition.source_list_index then
16:         t.source_list_index− = 1  ▷ Snížíme pořadí u jedinců, kteří byly za odstraněným jedincem
17:     end if
18: end for
19: for i, (a,b) in turnajový aspiranti do
20:     if a nebo b je transition.source_index then
21:         if valid_tournament == False then
22:             nastav validitu turnaje na pozici i na False
23:         end if
24:         T ← transition.in_collection(
25: G_gen.speculative_tournament[i], Mask.S_SOURCE)  ▷ Zjistíme jestli odstraněný jedinec vyhrál turnaj na pozici i
26:         if len(T) = 1 then
27:             remove_tournamnet(gen, G, T[0])  ▷ Odstraněný jedinec vyhrál turnaj - musíme ho odstranit z vítězů
28:         end if
29:     end if
30:     T ← transition.in_collection(
31: G_gen.speculative_tournament[i], Mask.SOURCE)  ▷ všechny přechody, které ukazují na stejnou zdrojovou pozici
32:     for t in T do
33:         if t.source_list_index > transition.source_list_index then
34:             t.source_list_index− = 1  ▷ Snížíme pořadí u jedinců, kteří byly za odstraněným jedincem
35:         end if
36:     end for
37: end for
```

4. Experimenty

Pro vyhodnocení efektu spekulativního vyhodnocování na užitečné využití procesoru jsme provedli sérii experimentů. V těchto experimentech budeme porovnávat asynchronní evoluci, obyčejný evoluční algoritmus, IGEA a IGEA-SE se všemi třemi politikami ukončování běžících tasků. V grafech budeme tyto verze značit IGEA-SE-A v případě, že můžeme ukončit libovolný spekulativní task, IGEA-SE-B v případě, kdy můžeme ukončit pouze tasky, o kterých už víme, že jsou špatné a nevyužijeme je a konečně IGEA-SE-N v případě, že spekulativní tasky ukončit nemůžeme. Pro výběr spekulativních jedinců budeme nejčastěji využívat surrogate model, který vybere lepšího z dvojice s 50%, 70% a 100% úspěšností. Ale vyzkoušíme i zbylé surrogate modely popsané v kapitole 3.4.

Pro výpočet fitness funkce budeme používat konstantní funkci a funkci rastrigin. Konstantní funkce vrací vždy stejné ohodnocení. V případě použití konstantní fitness funkce testujeme situace, kdy doba vyhodnocení není závislá na kvalitě jedince. Druhá zmíněná fitness funkce je funkce rastrigin, která je definována jako $f_R(x) = 10n + \sum_{i=1}^{n} x_i^2 - 10cos(2\pi x_i)$, kde n je počet parametrů v jedinci. V našich experimentech používáme jedince velikosti deset, tedy $n = 10$.

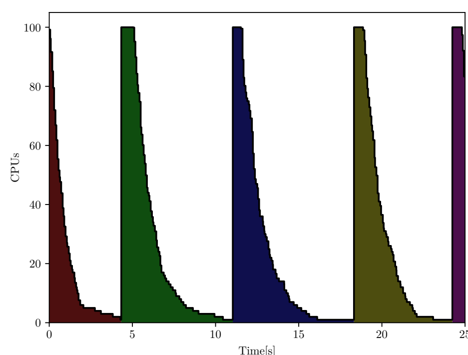
V simulacích potřebujeme i časovou funkci, která simuluje délku výpočtu fitness funkce. V našich experimentech používáme sedm časových funkcí. První z nich je konstantní časová funkce, která simuluje, že vyhodnocení každé fitness funkce trvá stejný čas. Další jsou uniformě náhodné časové funkce v rozmezí [1,10), [1,100) a [1,1000). Další časová funkce je také náhodná, ale s exponenciálním rozdělením a střední hodnotou 1. Všechny tyto časové funkce jsou nezávislé na fitness funkci. Dále jsme vyzkoušeli pozitivně a negativně korelovanou časovou funkci s funkcí fitness. Pozitivně korelovaná časová funkce je definována jako $T_p(x) = 1 + f(x)$, negativně korelovaná časová funkce je pak definována jako $T_n(x) = 1 + max(100 - f(x), 0)$. V negativně korelované časové funkci používáme funkci maximum pro ujištění, že čas nebude záporný. Funkce rastrigin často nepřesahuje hodnotu sto a naše konstantní fitness funkce vrací vždy jedničku, tedy limit sto je dostatečný pro simulaci negativní korelace mezi časovou a fitness funkcí.

Nastavení genetických operátorů je pro všechny experimenty stejné. Pravděpodobnost křížení je 0.8. Pro mutaci využijeme normální rozdělení $\mathcal{N}(0, 2.5)$ a pravděpodobnost mutace 0.1. Každý experiment zopakujeme 20 krát a ukážeme průměr a první a třetí kvartil (zašedlá oblast v grafech), avšak často jsou výsledky tak podobné, že oblast prvního a třetího kvartilu nebude vidět. Každý experiment ohodnotíme na 1, 10, 20, 30, ..., 100 procesorech.

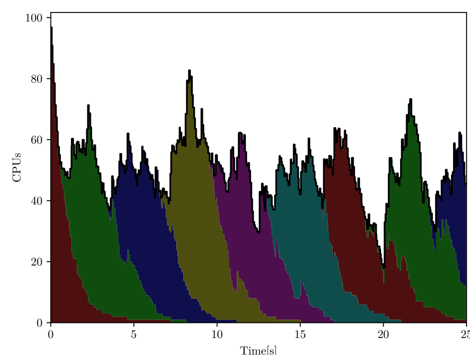
4.1 Prokládání generací

Ještě než si ukážeme výsledky experimentů, podíváme se na vizualizaci prokládání generací. Graf 4.1 ukazuje počet vyhodnocovaných jedinců z každé generace v čase. Generace jsou odlišeny různými barvami. Tmavší barva reprezentuje nespekulativní vyhodnocení, světlejší barvou jsou označeny spekulativní vyhodnocení (jedinec se začal počítat spekulativně). Černá linie označuje skutečný počet použitých procesorů. Vidíme, že IGEA má obrovské zlepšení oproti obyčejnému

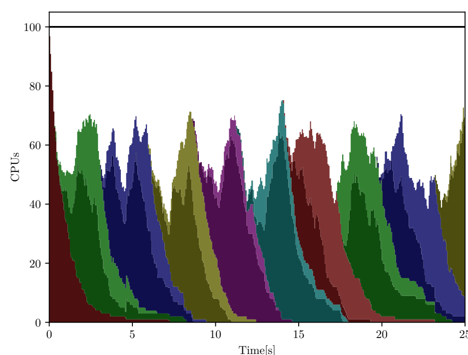
EA, ale stále je tam potenciál pro lepší vytížení. U IGEA-SE si můžeme všimnout, že spekulativní vyhodnocení je relativně úspěšné a zajistí nám ještě lepší užitečné vytížení procesoru. Občas se stane, že celkové vytížení procesoru není 100%, důvodem je, že nepovolujeme spekulativní jedince ze spekulativních jedinců a tudíž může dojít k situaci, že již nám ani spekulativní vyhodnocení na kompletní využití procesoru nestačí. Dále je potřeba říct, že 100% užitečného vytížení nedosáhneme ani s tím nejlepším modelem, protože spekulativní jedince volíme jen z jedinců, které máme v aktuální chvíli k dispozici a pokud ani jeden z jedinců z kterých vybíráme nepostoupí v nespekulativní části do další generace, pak ani výběr toho nejlepšího jedince z této množiny nám nezabrání v tom, že budeme muset tento výpočet později zahodit.



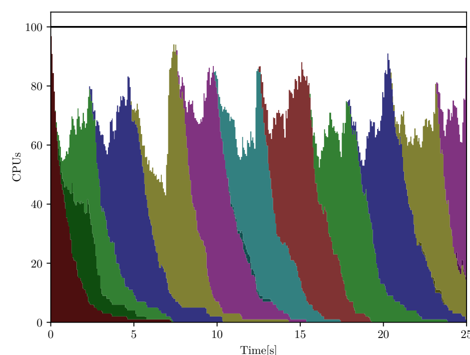
(a) (100,200)-EA



(b) (100,200)-IGEA



(c) (100,200)-IGEA-SE-N



(d) (100,200)-IGEA-SE-A

Obrázek 4.1: Na těchto grafech můžeme vidět vývoj vytížení procesoru se sto jádry v čase pro jednotlivé algoritmy. Všechny grafy používají exponenciální časovou funkci a optimalizují rastrigin fitness funkci. Různé barvy značí různé generace. Tmavší barva značí nespekulativní vyhodnocení, světlejší barva značí spekulativní vyhodnocení (task se začal vyhodnocovat jako spekulativní). Černá linie označuje skutečný počet využitých jader procesoru. Všechny algoritmy optimalizují funkci rastrigin s exponenciálním rozdělením v časové funkci.

4.2 (100,100) clever tournament

Nyní se již podíváme na provedené experimenty. Začneme s výsledky speciální verze chytrého turnaje (100,100). Na obrázku 4.2 můžeme vidět pár výsledků z tohoto experimentu. Všechny grafy které zde popisujeme, můžeme najít v příloze práce, jehož strukturu můžeme najít v příloze A. V této verzi probíhá spekulativní vyhodnocení pouze v turnajové selekci, tedy úspěšnost surrogate modelu je úspěšnost výběru správného spekulativního jedince. Vidíme, že IGEA-SE má téměř ve všech případech lepší užitečné vytížení procesoru než IGEA, která má lepší užitečné vytížení procesoru než klasický EA. IGEA-SE je horší než IGEA a pro vyšší počty procesorů dokonce i než obyčený EA pro negativně korelovanou a konstantní časovou funkci, pokud nemůžeme kdykoli ukončovat vyhodnocení spekulativních jedinců a surrogate model má úspěšnost 50 % nebo 70 %. V tomto případě spekulativní jedinci blokují nespekulativní jedince a tím zpomalují evoluci. V případě, že surrogate model má 100% úspěšnost, má IGEA-SE vždy lepší využití procesoru než IGEA. Konkrétně pro negativně korelovanou a konstantní časovou funkci je to vždy přes 95 %, pro exponenciální a pozitivně korelovanou časovou funkci a sto procesorů je to kolem 50 % a pro uniformě náhodné časové funkce a sto procesorů je to kolem 80 %. Můžeme si také všimnou, že rozdíl mezi surrogate modely s 50% a 70% úspěšností je minimální.

4.3 (100,200) a (100 + 200)

Další experiment, na který se podíváme, bude čárkovaná verze (100,200). Opět jsme vybrali pár příkladů, které můžeme vidět na obrázku 4.3. U tohoto nastavení dokáže IGEA i při využití šedesáti procesorů využívat téměř 100 % prostředků. Algoritmus IGEA-SE-A je ještě o kousek lepší a téměř 100% využití drží i při osmdesáti procesorech. Verze, kde není možné ukončovat spekulativní výpočty, se pohybuje mezi IGEA a IGEA-SE-A v závislosti na kvalitě surrogate modelu. U pozitivně korelované a exponenciální časové funkce dochází k poklesu užitečného vytížení procesoru dříve, už při využití 30, respektive 50 procesorů, ale stále má IGEA-SE toto vytížení lepší než IGEA. U těchto časových funkcí klesá užitečné vytížení na sto procesorech na hodnoty mezi 40 % až 45 % pro IGEA a mezi 50 % až 65 % u IGEA-SE.

Další z řady experimentů, které jsme provedli, je plusová verze (100 + 200), tato verze má téměř totožné užitečné vytížení procesoru jako verze čárkovaná. Pokud se podíváme na grafy v obrázku 2.1, které ukazují velikost množin *good*, *wrong* a *unknown* v závislosti na počtu ohodnocených jedinců, tak tento výsledek dává smysl, protože plusová verze začíná se sto ohodnocenými jedinci, ale musíme ohodnotit dalších sto jedinců, než budeme schopni určit, který jedinec postoupí do další generace. Stejný počet jedinců musíme ohodnotit i u čárkované verze, než budeme schopni rozhodnout, který jedinec určitě postoupí.

4.4 Vývoj fitness funkce

Pojďme si také ukázat vývoj fitness funkce v čase. Vždy porovnáváme stejné typy algoritmů a k nim přidáváme asynchronní evoluci. Ačkoli je těžké srovnávat

asynchronní evoluci s ostatními algoritmy, protože asynchronní evoluce se chová odlišně od ostatních algoritmů. Na obrázku 4.4 můžeme vidět negativně korelovanou časovou funkci (lepší jedinci se vyhodnocují rychleji), kde jsou rozdíly všech algoritmů minimální. U pozitivně korelované časové funkce už je situace jiná. Tam vidíme, že asynchronní evoluce upřednostňuje rychleji vyhodnocené jedince a proto na začátku rychleji klesá, ale poté co se dostane do lokálního minima, je pro ni těžké se odtud dostat, a proto nakonec ostatní algoritmy skončí s lepší fitness funkcí. Můžeme si také všimnout, že pro dvacet procesorů se algoritmy IGEA a IGEA-SE v rychlosti optimalizace neliší, protože oba algoritmy dokáží vytížit procesor na 100 %, ale při sto procesorech už rychleji konverguje IGEA-SE. V grafu vidíme, že všechny algoritmy se dostaly na podobnou úroveň fitness funkce. To je očekávaný výsledek, protože algoritmy jsou vytvořeny tak, aby byly ekvivalentní.

4.4.1 Rychlá konvergence a různé výsledky algoritmů

Občas se stane, že fitness funkce nejsou stejné, viz obrázek 4.5, nejvíce příkladů najdeme ve verzi clever tournament při použití sta procesorů. V grafu je konkrétně verze (100,100) s negativně korelovanou časovou funkcí. V těchto případech IGEA-SE často velmi rychle zkonverguje do neoptimálního řešení. Zrychlená konvergence by mohla naznačovat, že algoritmus má podobnou závislost na čase, jako asynchronní evoluce, ale zatím jsme nepřišli na to, kde bychom tuto závislost mohli vytvořit. Případně by se mohlo jedna o nějaký okrajový případ, kterého jsme si nevšimli.

Další příklad, který je v grafech je plusová verze $(100 + 200)$ s pozitivně korelovanou časovou funkcí. Tady už se fitness funkce o moc neliší, ale přesto by hodnoty fitness funkce mezi algoritmy měli mít menší rozdíl, protože algoritmy jsou navrženy tak, aby se chovaly ekvivalentně. Tyto menší rozdíly se občas vyskytují u každé verze algoritmu a bohužel se nám zatím nepodařilo najít příčinu těchto rozdílů.

Pro tyto případy je tedy potřeba do budoucna provést podrobný rozbor a zjistit, za jakých okolností tyto případy vznikají a následně se je pokusit odstranit.

4.5 $(50,100)$ a $(50 + 100)$

Vzhledem k tomu, že se objevilo několik situací, kdy algoritmus IGEA-SE dosáhl více než 90% užitečného vytížení procesoru pro $\mu = 100$ a $\lambda = 200$, podíváme se ještě na případy $(50,100)$ a $(50 + 100)$. Tyto simulace jsme vyzkoušeli pouze se simulaci surrogate modelu s úspěšností 50 %, 70 % a 100 %. Ostatní surrogate modely jsme nepoužili. Z výsledků na obrátku 4.6 vidíme, že pro negativně korelovanou časovou funkci dokážeme udržet užitečné vytížení procesoru nad 80 % v případě IGEA-SE-A. V případě IGEA-SE-B a IGEA-SE-N, kde nemůžeme ukončovat vyhodnocení spekulativních jedinců se opět dostáváme do situace, kdy spekulativní vyhodnocení zpomalí evoluci, protože zahltíme procesor výpočty ohodnocení, které nebudeme potřebovat. U uniformě náhodných časových funkcí klesáme k 50 % a u exponenciálně rozdělené časové funkce dokonce k 30 %.

4.6 (100,200) a (100 + 200) s dvěma vítězi v turnaji

Nyní se již dostáváme k experimentu, který využije více jedinců na jedné pozici. Konkrétně budeme vybírat dva vítěze turnaje. V této simulaci opět platí, že čárkovaná verze (100,200) a plusová verze (100 + 200) mají stejné užitečné využití procesoru. Rozdíl v užitečném využití procesoru se surrogate modelem s 50%, 70% a 100% úspěšností je minimální. Z grafů 4.7 můžeme vidět, že IGEA-SE-N a IGEA-SE-B má často horší užitečné vytížení procesoru než IGEA. U algoritmu IGEA-SE-A je toto vytížení lepší než u IGEA, ale je horší než ve verzi, kde turnaj vyhraje pouze jeden jedinec. Pokud z turnaje postupují dva jedinci, pak sice máme vyšší pravděpodobnost, že jeden z jedinců skutečně turnaj vyhraje. To je vidět i v minimálním rozdílu užitečného vytížení procesoru pro jednotlivé surrogate modely. Na druhou stranu máme jistotu, že minimálně jeden jedinec turnaj nevyhraje. To tedy znamená, že v podstatě degradujeme úspěšnost surrogate modelu na maximálně 50 %. Je otázkou, jak by postup více jedinců z turnaje ovlivnil turnaj, kterého by se účastnilo více jedinců. Pro horší surrogate model by tato technika mohla pomoci.

4.7 (100,100) bez chytrého turnaje a (100 + 100)

Experimentovali jsme i s nastavením, kde $\mu = \lambda$, konkrétně (100,100) bez chytrého turnaje a (100 + 100). Při těchto experimentech jsme u verze plus narazili na problém s dosahováním maximální hloubky rekurze. Tento problém byl způsoben faktem, že ve spekulativní části výpočtu jsme měli pro postup do další generace pouze kritérium hodnoty fitness funkce, na rozdíl od nespekulativní části, kde jsme používali ID jedince jako druhé kritérium a tím jsme upřednostňovali novější jedince. Pro konstantní fitness funkci jsme se dostali do situace, že jsme ohodnotili populaci a část potomků, tedy už jsme byli schopni určit minimální hodnotu fitness funkce pro postup. Ale protože všichni jedinci měli stejnou hodnotu fitness funkce, a protože jsme neměli druhé kritérium, postoupili všichni ohodnocení jedinci do populace následující generace. Tato populace se plně zaplnila a protože pravděpodobnost genetických operátorů není 100%, tak vždy vznikl potomek, který se nezměnil, tudíž u něj nebylo nutné počítat hodnotu fitness funkce. Díky těmto potomkům, kteří se nezměnili jsme měli opět dostatečný počet ohodnocených jedinců na to, abychom byli schopni určit minimální hodnotu fitness funkce na postup do další generace. A nyní se již celá situace opakovala. Při využití funkce rastrigin jako fitness funkce mohla tato situace nastat v případě, kdy populace zkonvergovala.

Všechny experimenty, které jsme doteď uvedli byly spočítány s touto nekonzistencí. Vzhledem k časové náročnosti provedení výpočtů nebylo možné experimenty přepočítat.

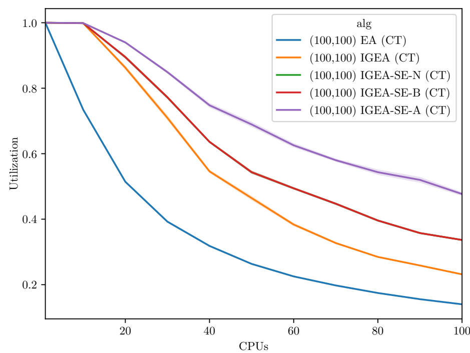
Na obrázku 4.8 můžeme vidět, u verze plus nedochází k výrazné změně při lepší predikci surrogate modelu. Zlepšují se pouze verze, které nemohou ukončovat spekulativní vyhodnocení. Naopak u verze s čárkou je zlepšení o poznání větší a u časové funkce s exponenciálním rozdělením se při sto procesorech dostaneme na téměř 60% vytíženost procesoru oproti 35% u algoritmu IGEA.

4.8 $(100 + 200)$ reálný experiment

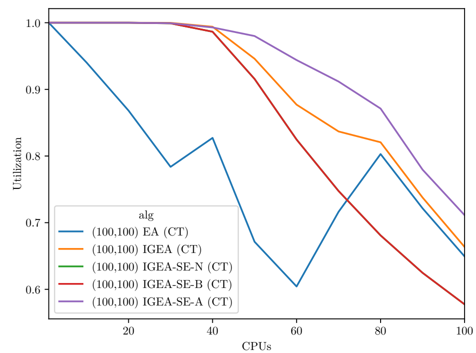
Nakonec se podíváme na reálný experiment, který jsme vypočítali v metacentru. V tomto experimentu jsme optimalizovali parametry mnohovrstvého perceptronu s jednou skrytou vrstvou z knihovny scikit-learn [7]. Perceptron jsme trénovali a testovali na datasetu Wine Quality [1]. Dataset je rozdělen na dvě části. Trénovací dataset obsahuje 4898 záznamů a testovací 1599. Každý záznam se skládá z dvanácti atributů, kde poslední určuje kvalitu vína, kterou se snažíme predikovat. Evoluční algoritmus bude optimalizovat šestnáct hyperparametrů modelu. Konkrétně to je pět kategorických, devět celočíselných na intervalu (0,1) a poslední dva jsou počet iterací učení algoritmu v intervalu [0,10 000] a velikost skryté vrstvy v intervalu [1,1000]. Hodnota fitness funkce bude přesnost natrénovaného modelu.

Vzhledem k časovým nárokům jsme vyzkoušeli pouze algoritmy EA, IGEA a IGEA-SE-A na 10,20 a 30 procesorech se třemi opakováními. Na obrázku 4.9 můžeme vidět užitečné vytížení procesoru a vývoj fitness funkcí. Užitečné vytížení procesoru je poměrně nízké u všech algoritmů, kromě IGEA a IGEA-SE na deseti procesorech. Když se podíváme na vývoj fitness funkce, můžeme si všimnout, že délka běhu algoritmu na deseti procesorech je nepřímo úměrná užitečnému vytížení, což je poměrně překvapivý výsledek. Proto se ještě podíváme na to, co se děje při výpočtu algoritmu.

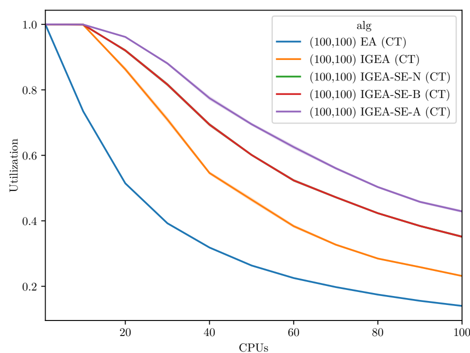
Na obrázku 4.10 můžeme vidět vytížení procesorů pro vybrané běhy algoritmu. Na tomto obrázku vidíme, že jsou obrovské rozdíly mezi časy vyhodnocení a čas vyhodnocení nezávisí úplně na hodnotě fitness funkce. U verze IGEA-SE-A vidíme, že ani spekulativní vyhodnocení nám nedokáže vytížit procesory, pokud čekáme na pár extrémně dlouhých ohodnocení jak spekulativních, tak nespekulativních, a všechny ostatní jedince dokážeme vyhodnotit rychle. Pak dopočítáme vše i ve spekulativní části, a protože nepovolujeme vytváření spekulativních jedinců ze spekulativních, už nemáme jak dál pokračovat. Také si můžeme všimnout, že úspěšnost spekulativních jedinců klesala s tím, jak jsme počítali více generací dopředu (na obrázku kolem 200 s) Z grafů také vidíme, že jednotlivé běhy se od sebe poměrně liší, a proto je i poměrně velký rozptyl utilizace. Proto bychom potřebovali provést více opakování, aby se výsledek ustálil. Věřím, že po více opakování by se projevil pozitivní účinek spekulativního vyhodnocení na užitečné vytížení procesoru.



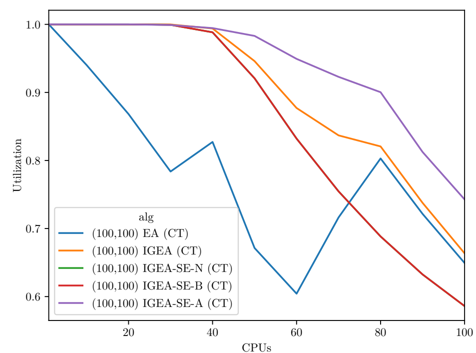
(a) pos tf, 50%



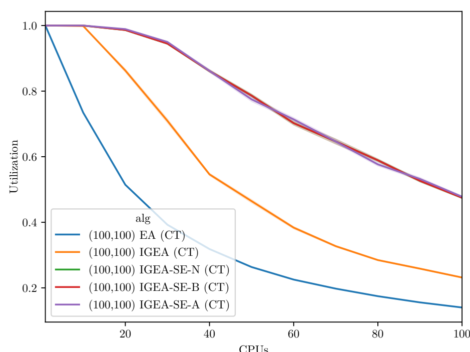
(b) neg tf, 50%



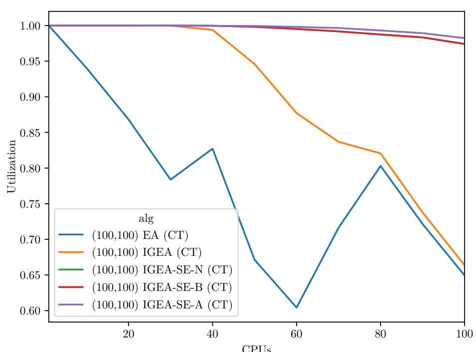
(c) pos tf, 70%



(d) neg tf, 70%

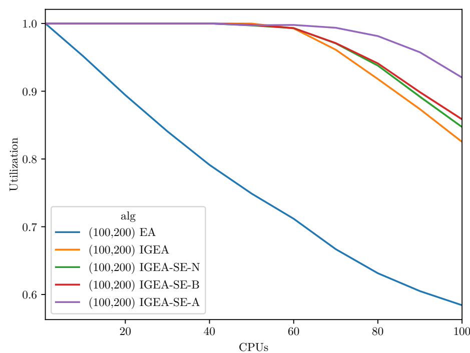


(e) pos tf, 100%

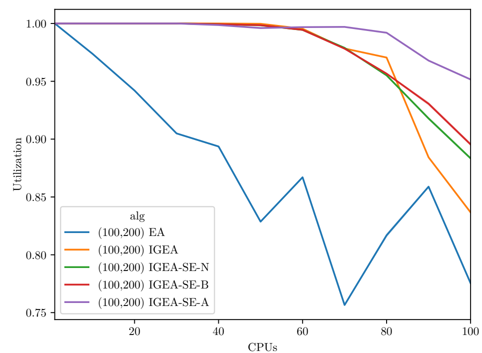


(f) neg tf, 100%

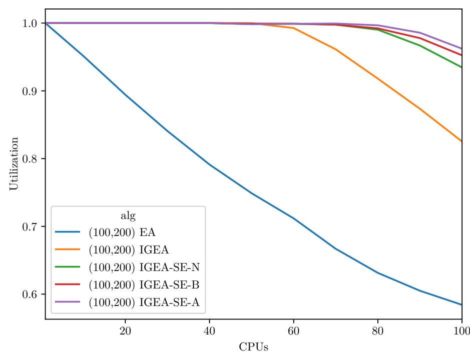
Obrázek 4.2: V těchto grafech vidíme užitečné využití procesoru pro speciální případ clever tournament (100,100). Všechny experimenty vyhodnocují rastrigin funkci. V levém sloupci je použita pozitivně korelovaná časová funkce a v pravém sloupci je negativně korelovaná časová funkce. Grafy v prvním řádku využily surrogate model s přesností 50 %, v druhém řádku s přesnotí 70 % a ve třetím řádku s přesností 100 %. IGEA-SE-B a IGEA-SE-N se překrývají.



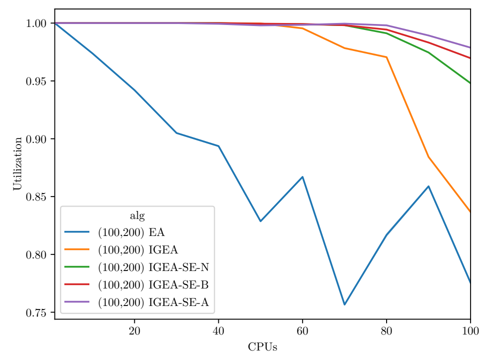
(a) r1000 tf, 70%



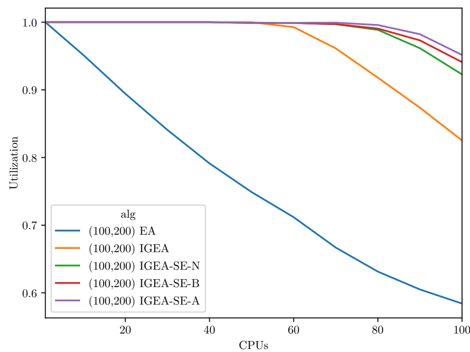
(b) neg tf, 70%



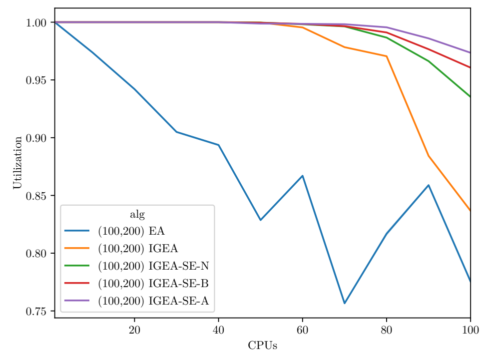
(c) r1000 tf, 100%



(d) neg tf, 100%

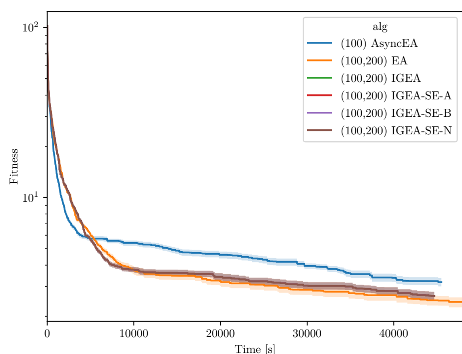


(e) r1000 tf, static

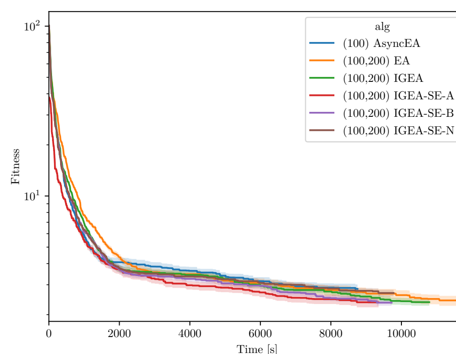


(f) neg tf, static

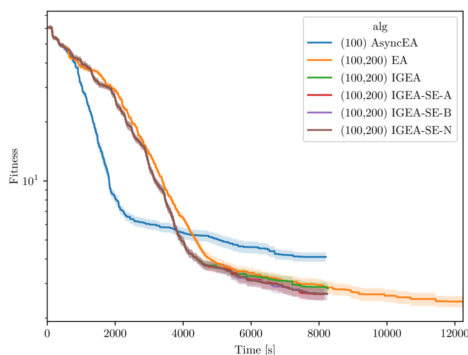
Obrázek 4.3: V těchto grafech vidíme užitečné využití procesoru pro čárkovanou verzi (100,200). Všechny experimenty vyhodnocují rastrigin funkci. V levém sloupci je použita uniformě náhodná časová funkce s rozmezím hodnot [1,1000) a v pravém sloupci je negativně korelovaná časová funkce. Grafy v prvním řádku využily surrogate model s přesností 70 %, v druhém řádku s přesnotí 100 % a ve třetím řádku jsme využily statický surrogate model, který se neučí, ale rozhoduje se nejlépe jak může z informací které má.



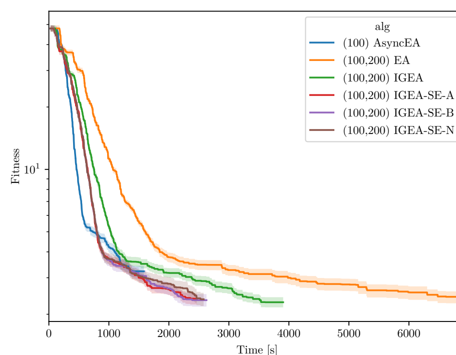
(a) neg tf, 20 CPU



(b) neg tf, 100 CPU

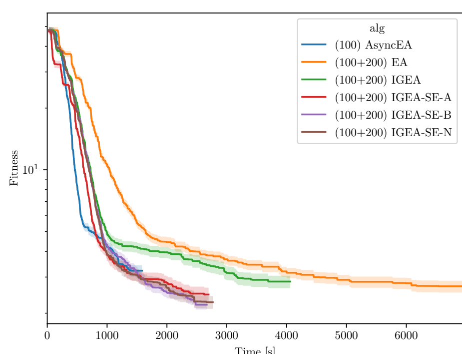


(c) pos tf, 20 CPU

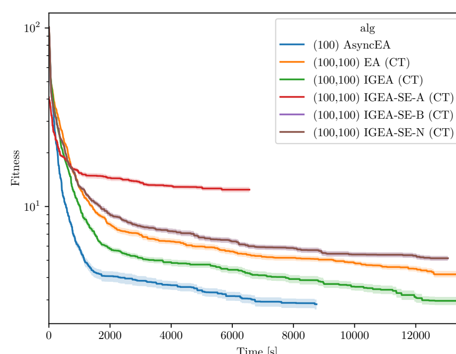


(d) pos tf, 100 CPU

Obrázek 4.4: V těchto grafech vidíme vývoj fitness funkce pro čárkovanou verzi (100,200). Všechny experimenty vyhodnocují rastrigin funkci a používají surrogate model se 70% úspěšností. V levém sloupci je použito 20 procesorů a v pravém sloupci je použito 100 procesorů. Grafy v prvním řádku byly počítány s negativně korelovanou časovou funkcí, v druhém řádku je použita pozitivně korelovaná časová funkce.

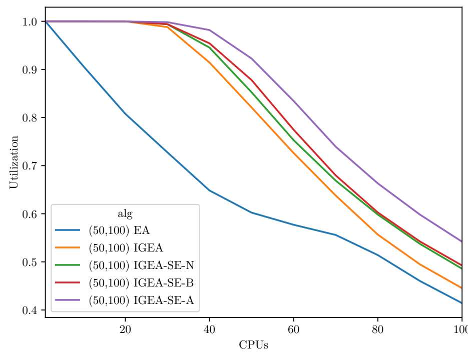


(a) pos tf, 100 CPU

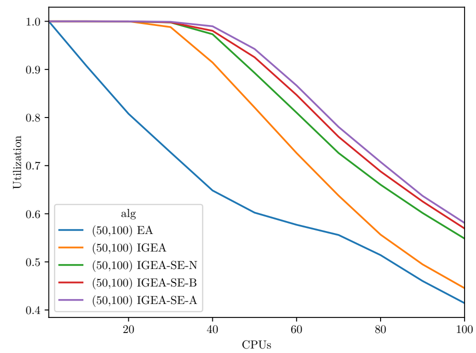


(b) neg tf, 100 CPU

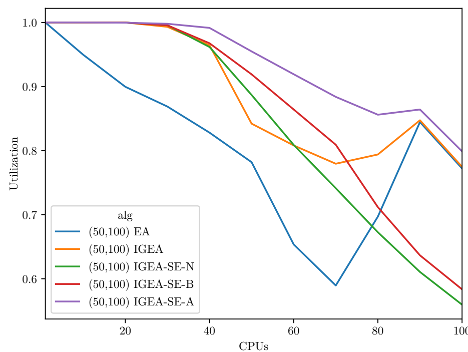
Obrázek 4.5: V těchto grafech vidíme vývoj fitness funkce. Oba grafy používají surrogate model s úspěšností 50 % a optimalizují rastrigin funkci na sto procesorech. Vlevo je verze plus s pozitivně korelovanou časovou funkcí. Vpravo je chytrý turnaj s negativně korelovanou časovou funkcí.



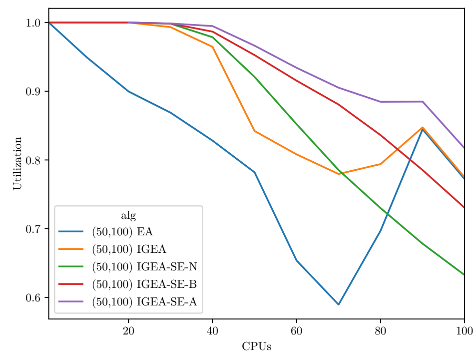
(a) r1000 tf, 70%



(b) r1000 tf, 100%

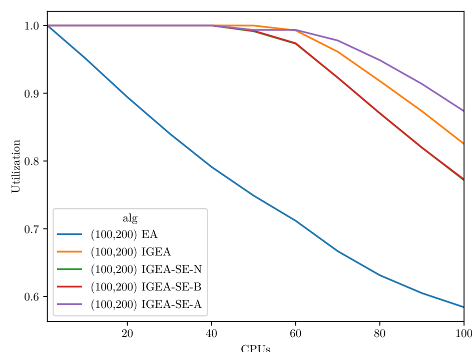


(c) neg tf, 70%

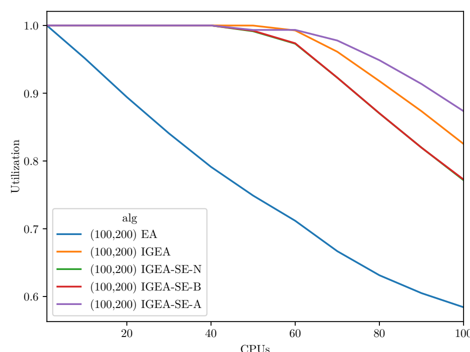


(d) neg tf, 100%

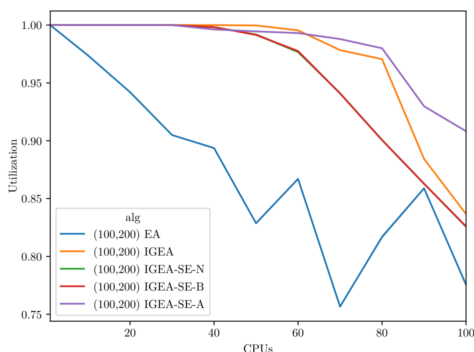
Obrázek 4.6: V těchto grafech vidíme užitečné využití procesoru pro čárkovanou verzi (50,100). Všechny experimenty vyhodnocují rastrigin funkci. V levém sloupci jsme využili surrogate model s přesností 70 %, v pravém s přesností 100 %. Grafy v prvním řádku zobrazují použití uniformě náhodné časové funkce s rozmezím hodnot [1,1000). Na druhém řádku je negativně korelovaná časová funkce.



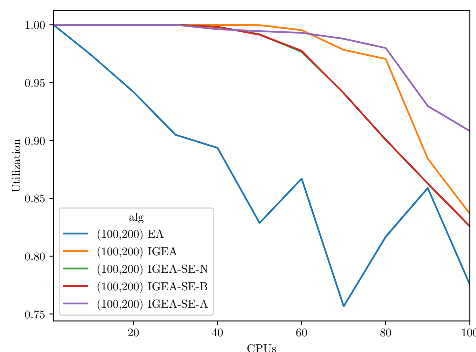
(a) r1000 tf, 70%



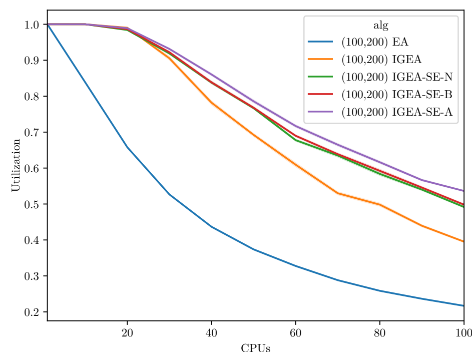
(b) r1000 tf, 100%



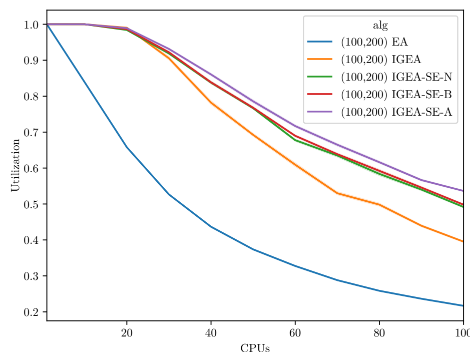
(c) neg tf, 70%



(d) neg tf, 100%

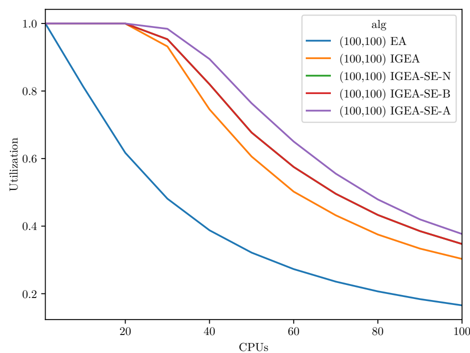


(e) pos tf, 70%

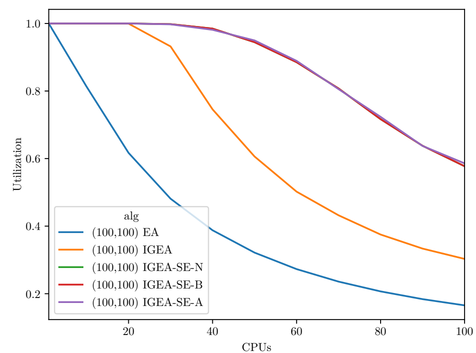


(f) pos tf, 100%

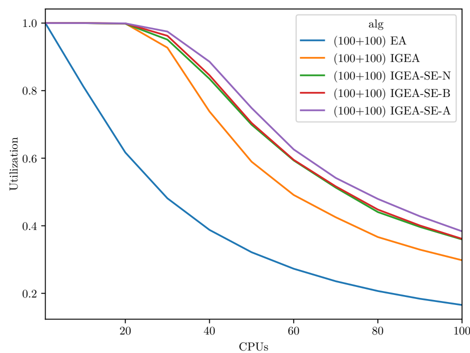
Obrázek 4.7: V těchto grafech vidíme užitečné využití procesoru pro čárkovanou verzi (100,200), kde v turnaji postupují dva jedinci. Všechny experimenty vyhodnocují rastrigin funkci. V levém sloupci jsme využili surrogate model s přesností 70 %, v pravém s přesností 100 %. Grafy v prvním řádku zobrazují použití uniformě náhodné časové funkce s rozmezím hodnot [1,1000). Na druhém řádku je negativně korelovaná časová funkce. Na třetím řádku je pozitivně korelovaná časová funkce.



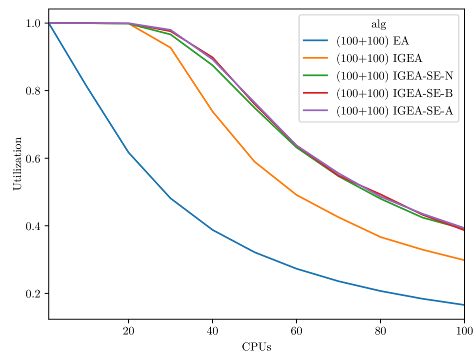
(a) exp tf, 70%



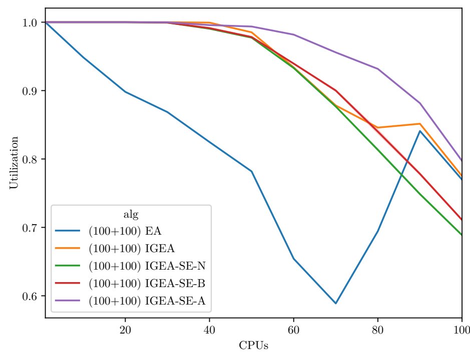
(b) exp tf, 100%



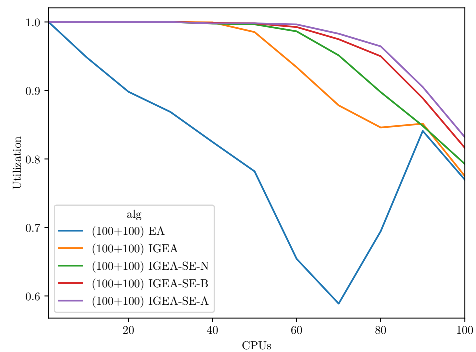
(c) exp tf, 70%



(d) exp tf, 100%

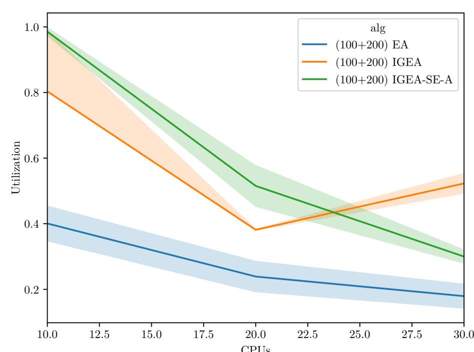


(e) neg tf, 70%

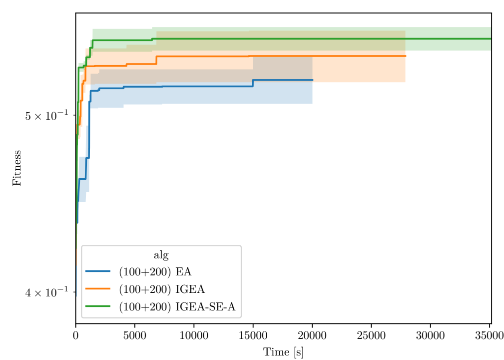


(f) neg tf, 100%

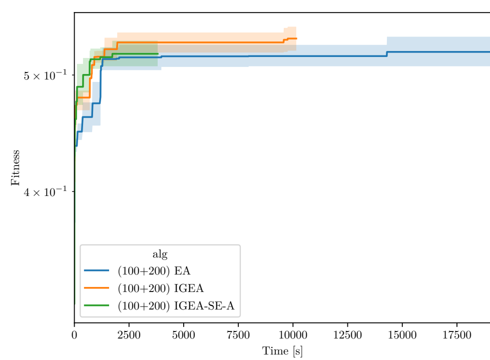
Obrázek 4.8: V těchto grafech vidíme užitečné využití procesoru pro čárkovanou verzi (100,100) v prvním řádku a pro plusovou verzi (100 + 100) ve druhém a třetím řádku. Všechny experimenty vyhodnocují rastrigin funkci. V levém sloupci jsme využili surrogate model s přesností 70 %, v pravém s přesností 100 %. Grafy v prvním řádku zobrazují použití časové funkce s exponenciálním rozdělením. Na druhém řádku je taktéž časová funkce s exponenciálním rozdělením. Na třetím řádku je negativně korelovaná časová funkce.



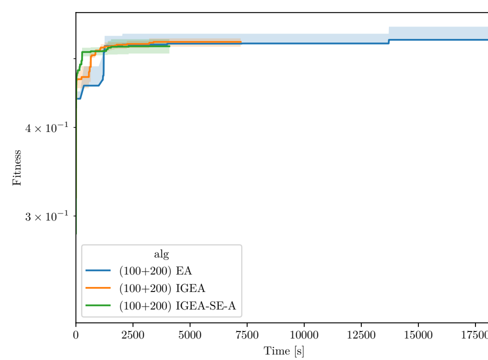
(a) užitečné využití procesoru



(b) fitness funkce, 10 CPU

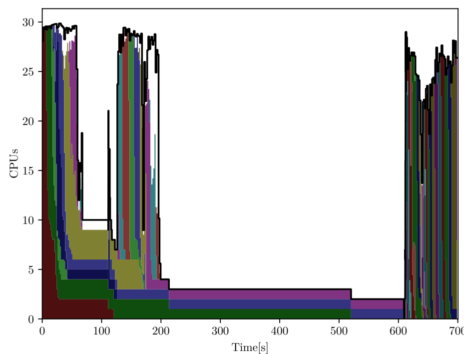


(c) fitness funkce, 20 CPU

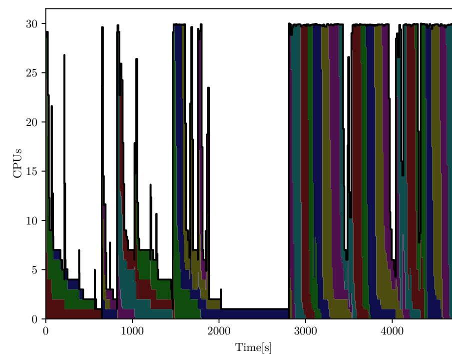


(d) fitness funkce, 30 CPU

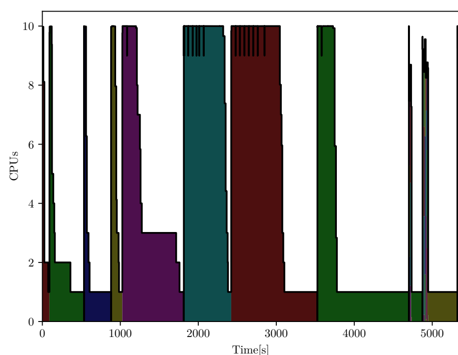
Obrázek 4.9: V prvním grafu 4.9a můžeme vidět užitečné vytížení procesoru při optimalizaci hyperparametrů modelu mnohovrstvého perceptronu. V dalších grafech můžeme vidět vývoj fitness funkce, která je v tomto případě přesnost natrénovaného modelu. Fitness funkce je ukázána pro 10, 20 a 30 procesorů.



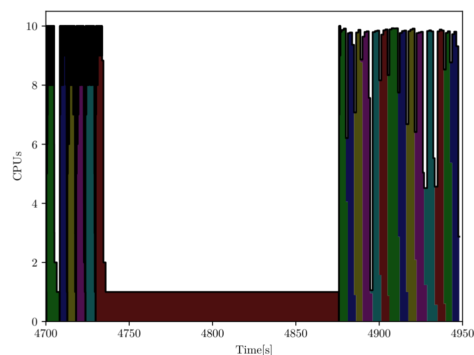
(a) IGEA-SE-A, 30 CPU



(b) IGEA, 30 CPU



(c) EA, 10 CPU



(d) EA, 10 CPU, zoom

Obrázek 4.10: V těchto grafech vidíme zatížení procesoru podle generací v čase. Všechny experimenty optimalizují hyperparametry mnohovrstvého perceptronu a využívají plusovou verzi algoritmu (100 + 200). V grafu 4.10a je algoritmus IGEA-SE-A, který může využít třicet procesorů. V grafu 4.10b je algoritmus IGEA a také může využít 30 procesorů. V grafu 4.10d je algoritmus EA a může využít 10 procesorů. V posledním grafu 4.10d je přiblížení grafu 4.10c na časy 4700s – 4950s kde dojde k extrémně rychlému ohodnocení několika generací.

Závěr

V práci jsme popsali a implementovali rozšíření evolučního algoritmu s prokládáním generací o spekulativní vyhodnocení. Hlavním cílem algoritmu je maximalizovat využití procesoru, ale oproti asynchronní evoluci se chceme vyvarovat upřednostňování rychle vyhodnotitelných řešení. V experimentech jsme ukázali, že algoritmus dokáže zvýšit využití procesoru oproti původní verzi bez spekulativního vyhodnocení. Tento efekt jsme si ukázali v závislosti na úspěšnosti surrogate modelu i v závislosti na rozložení doby vyhodnocování jedinců i v závislosti na politice ukončování spekulativních vyhodnocení.

Z experimentů můžeme pozorovat, že pokud spekulativní výpočty můžeme kdykoli ukončit, je tento algoritmus vždy lepší než původní verze. V tomto případě může být zlepšení dokonce i 20 %. V případech, kdy spekulativní výpočty ukončit nemůžeme, už tak jasná odpověď neexistuje a záleží na konkrétní úloze. V ojedinělých případech se může stát, že algoritmus se spekulativním vyhodnocením bude horší než obyčejná paralelní verze evolučních algoritmů a to v případech, kdy nemůžeme ukončovat spekulativní jedince a vyhodnocení těchto jedinců trvá dlouho. Toto ovšem platí až pro opravdu extrémní paralelizaci, kdy máme téměř jeden procesor na jednoho potomka v generaci. Při experimentování jsme také narazili na problém s rychlou konvergencí u verze clever tournament s vysokým počtem procesorů, a také se u pár experimentů objevil problém s rozdílnou hodnotou fitness funkce na konci algoritmu.

Budoucnost

Projekt bychom do budoucna měli více otestovat, protože v průběhu vyhodnocování jsme narazili na pár výsledků, které jsme neočekávali. Jde především o pár případů rozdílné hodnoty fitness funkce na konci algoritmu a rychlou konvergenci u verze clever tournament při použití sta procesorů. Dále by bylo dobré algoritmus otestovat více v reálném prostředí, abychom věděli, jak se v takovém prostředí algoritmus chová. Také by bylo jistě zajímavé vyzkoušet, jestli se někdy vyplatí mít více jedinců na stejné pozici, například pokud bychom měli turnajovou selekci více jedinců, nebo pokud bychom z více nespekulativních potomků přidali na jednu pozici do spekulativní populace následující generace.

Seznam použité literatury

[1] Cortez, P., Cerdeira, A., Almeida, F., Matos, T. a Reis, J. (2009). Modeling wine preferences by data mining from physicochemical properties. *Decision Support Systems*, **47**(4), 547 – 553. ISSN 0167-9236. doi: https://doi.org/10.1016/j.dss.2009.05.016. URL `http://www.sciencedirect.com/science/article/pii/S0167923609001377`. Smart Business Networks: Concepts and Empirical Evidence.

[2] Darwin, C. (1859). *On the Origin of Species: by Means of Natural Selection*. John Murray, Londýn.

[3] Darwin, C. (1923). *O původu druhů*. Nový názor. Dědictví Havlíčkovo, Brno.

[4] Fortin, F.-A., De Rainville, F.-M., Gardner, M.-A., Parizeau, M. a Gagné, C. (2012). DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, **13**, 2171–2175.

[5] Michalewicz, Z. (1996). *Genetic Algorithms + Data Structures = Evolution Programs*. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-662-03315-9. doi: 10.1007/978-3-662-03315-9.

[6] Paz, E. (2001). *Efficient and Accurate Parallel Genetic Algorithms*. Springer US, Boston, MA. ISBN 978-1-4615-4369-5.

[7] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M. a Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, **12**, 2825–2830.

[8] Pilát, M. a Neruda, R. (2017). Parallel evolutionary algorithm with interleaving generations. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 865–872. ACM. doi: 10.1145/3071178.3071309.

[9] Scott, E. O. a De Jong, K. A. (2015). Understanding simple asynchronous evolutionary algorithms. In *Proceedings of the 2015 ACM Conference on Foundations of Genetic Algorithms XIII*, pages 85–98.

[10] Scott, E. O. a Jong, K. A. D. (2016). Evaluation-time bias in quasi-generational and steady-state asynchronous evolutionary algorithms. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference - GECCO '16*, pages 845–852. ACM Press. doi: 10.1145/2908812.2908934.

[11] Wouters, T. (2017). Global interpreter lock. URL `https://wiki.python.org/moin/GlobalInterpreterLock`. [cit. 2020-07-14].

Seznam obrázků

2.1 Na těchto grafech můžeme vidět vývoj počtu jedinců v jednotlivých kategoriích pro $\alpha = 100$ a různé nastavení β. Zeleně je označena velikost množiny *good*, žlutě *unknown* a červeně *wrong*. V grafu 2.1a je $\beta = 100$, pro toto nastavení každý jedinec postoupí, takže množiny *unknown* a *wrong* jsou vždy prázdné. V grafu 2.1b je $\beta = 150$. V grafu 2.1c je $\beta = 200$, tady máme vždy stejný počet v množině *good* a *wrong*. V posledním grafu 2.1d je $\beta = 300$. . . . 12

4.1 Na těchto grafech můžeme vidět vývoj vytížení procesoru se sto jádry v čase pro jednotlivé algoritmy. Všechny grafy používají exponenciální časovou funkci a optimalizují rastrigin fitness funkci. Různé barvy značí různé generace. Tmavší barva značí nespekulativní vyhodnocení, světlejší barva značí spekulativní vyhodnocení (task se začal vyhodnocovat jako spekulativní). Černá linie označuje skutečný počet využitých jader procesoru. Všechny algoritmy optimalizují funkci rastrigin s exponenciálním rozdělením v časové funkci. 30

4.2 V těchto grafech vidíme užitečné využití procesoru pro speciální případ clever tournament (100,100). Všechny experimenty vyhodnocují rastrigin funkci. V levém sloupci je použita pozitivně korelovaná časová funkce a v pravém sloupci je negativně korelovaná časová funkce. Grafy v prvním řádku využily surrogate model s přesností 50 %, v druhém řádku s přesnotí 70 % a ve třetím řádku s přesností 100 %. IGEA-SE-B a IGEA-SE-N se překrývají. 35

4.3 V těchto grafech vidíme užitečné využití procesoru pro čárkovanou verzi (100,200). Všechny experimenty vyhodnocují rastrigin funkci. V levém sloupci je použita uniformě náhodná časová funkce s rozmezím hodnot [1,1000) a v pravém sloupci je negativně korelovaná časová funkce. Grafy v prvním řádku využily surrogate model s přesností 70 %, v druhém řádku s přesnotí 100 % a ve třetím řádku jsme využily statický surrogate model, který se neučí, ale rozhoduje se nejlépe jak může z informací které má. 36

4.4 V těchto grafech vidíme vývoj fitness funkce pro čárkovanou verzi (100,200). Všechny experimenty vyhodnocují rastrigin funkci a používají surrogate model se 70% úspěšností. V levém sloupci je použito 20 procesorů a v pravém sloupci je použito 100 procesorů. Grafy v prvním řádku byly počítány s negativně korelovanou časovou funkci, v druhém řádku je použita pozitivně korelovaná časová funkce. 37

4.5 V těchto grafech vidíme vývoj fitness funkce. Oba grafy používají surrogate model s úspěšností 50 % a optimalizují rastrigin funkci na sto procesorech. Vlevo je verze plus s pozitivně korelovanou časovou funkcí. Vpravo je chytrý turnaj s negativně korelovanou časovou funkcí. 37

4.6 V těchto grafech vidíme užitečné využití procesoru pro čárkovanou verzi (50,100). Všechny experimenty vyhodnocují rastrigin funkci. V levém sloupci jsme využili surrogate model s přesností 70 %, v pravém s přesností 100 %. Grafy v prvním řádku zobrazují použití uniformě náhodné časové funkce s rozmezím hodnot [1,1000). Na druhém řádku je negativně korelovaná časová funkce. 38
4.7 V těchto grafech vidíme užitečné využití procesoru pro čárkovanou verzi (100,200), kde v turnaji postupují dva jedinci. Všechny experimenty vyhodnocují rastrigin funkci. V levém sloupci jsme využili surrogate model s přesností 70 %, v pravém s přesností 100 %. Grafy v prvním řádku zobrazují použití uniformě náhodné časové funkce s rozmezím hodnot [1,1000). Na druhém řádku je negativně korelovaná časová funkce. Na třetím řádku je pozitivně korelovaná časová funkce. 39
4.8 V těchto grafech vidíme užitečné využití procesoru pro čárkovanou verzi (100,100) v prvním řádku a pro plusovou verzi (100 + 100) ve druhém a třetím řádku. Všechny experimenty vyhodnocují rastrigin funkci. V levém sloupci jsme využili surrogate model s přesností 70 %, v pravém s přesností 100 %. Grafy v prvním řádku zobrazují použití časové funkce s exponenciálním rozdělením. Na druhém řádku je taktéž časová funkce s exponenciálním rozdělením. Na třetím řádku je negativně korelovaná časová funkce. 40
4.9 V prvním grafu 4.9a můžeme vidět užitečné vytížení procesoru při optimalizaci hyperparametrů modelu mnohovrstvého perceptronu. V dalších grafech můžeme vidět vývoj fitness funkce, která je v tomto případě přesnost natrénovaného modelu. Fitness funkce je ukázána pro 10, 20 a 30 procesorů. 41
4.10 V těchto grafech vidíme zatížení procesoru podle generací v čase. Všechny experimenty optimalizují hyperparametry mnohovrstvého perceptronu a využívají plusovou verzi algoritmu (100+200). V grafu 4.10a je algoritmus IGEA-SE-A, který může využít třicet procesorů. V grafu 4.10b je algoritmus IGEA a také může využít 30 procesorů. V grafu 4.10d je algoritmus EA a může využít 10 procesorů. V posledním grafu 4.10d je přiblížení grafu 4.10c na časy 4700s – 4950s kde dojde k extrémně rychlému ohodnocení několika generací. 42

A. Obsah přílohy

- `AsyncEA` — obsahuje zdrojové kódy algoritmu
- `Experiments` — obsahuje výsledky provedených simulací rozdělených do složek.
- `README.txt` — popis obsahu přílohy (podobný této sekci)

AsyncEA

Ve složce AsyncEA najdeme zdrojové kódy algoritmu popsaného v této práci a zdrojový kód s příkladem použití algoritmu.

- `AsyncEA` — obsahuje implementaci hlavních částí algoritmu
  - `const.py` obsahuje konstanty a překladové tabulky
  - `fitness_functions.py` obsahuje implementaci všech použitých fitness funkcí
  - `Logging.py` se stará o logování algoritmu
  - `LogProcessing.py` vytváří sumarizaci logů vytvořených v Logging.py
  - `ParallelSimulator.py` obsahuje implementaci paralelních prostředí
  - `Planner.py` implementuje plánovač úloh
  - `PlottingGraph.py` vytváří grafy ze sumarizovaných logů
  - `SurrogateModels.py` obsahuje všechny použité surrogate modely
  - `Task.py` obsahuje strukturu uchovávající informace o jedinci předaném k vyhodnocení a o jeho vyhodnocení
  - `time_functions.py` obsahuje implementaci všech časových funkcí
  - `utils.py` obsahuje pár utilit využívaných v experimentech. Například generování jmen složek
- `simulation.py` — spustí experimenty nastavené v konfiguračním yaml souboru předaném jako parametr programu
- `settings.yml` — příklad konfiguračního souboru
- `environment.yml` — konfigurační soubor pro vytvoření virtuálního prostředí v Anaconda

Experiments

Ve složce `Experiments` najdeme experimenty rozdělené do složek. Složky mají specifickou struktur názvu, aby bylo vidět, které experimenty obsahují. Struktura je následující:

`logs_<mu>_<lambda>_<surrogate_model>_<winners>[_limit][_real]_unzip`

- `<mu>` je velikost populace
- `<lambda>` je počet potomků
- `<surrogate_model>` je typ použitého surrogate modelu:
  - `50` — simulace surrogate modelu s přesností 50
  - `70` — simulace surrogate modelu s přesností 70
  - `100` — simulace surrogate modelu s přesností 100
  - `static` — statický surrogate model — netrénuje se, využívá dostupných informací
  - `clasvm` — klasifikační surrogate model postavený na modelu SVM — model je komparátor dvou jedinců
  - `regsvm` — regresní surrogate model postavený na modelu SVM — model odhaduje fitness
- `<winners>` je počet vítězů v turnaji
- `_limit` se vyskytuje u výsledků experimentů, které byly spuštěny s limitem na počet rozpočítaných generací
- `_real` se vyskytuje u výsledků experimentů, které se reálně optimalizovali nějaký problém (v našem případě parametry mnohovrstvého perceptronu)

V jednotlivých složkách s experimenty najdeme PDF soubory s grafy popisující výsledek experimentů. Složky složky mají dvě struktury názvů, podle toho jestli obsahují vývoj fitness funkce nebo obsahují utilizaci. Struktury jsou následující:
`[log_]fitness_<cpu>_<ea_type>_<ff>_ff_<tf>_tf.pdf`
`util_<ea_type>_<ff>_ff_<tf>_tf.pdf`

- `log_` se vyskytuje pokud je osa y v logaritmickém měřítku
- `<cpu>` je počet použitých procesorů
- `<ea_type>` je typ evoluce:
  - `clever` — chytrý turnaj ($\mu = \lambda$) v případě, že jsme ve složce ke $\mu \neq \lambda$, tak je λ nastavena na hodnotu μ
  - `comma` — čárkovaná verze
  - `plus` — plusová verze
- `<ff>` je použitá fitness funkce
- `<tf>` je použitá časová funkce (`random<num>` označuje uniformě náhodnou časovou funkci na intervalu [1,`<num>`), ostatní názvy jsou zřejmé)