Simulační nástroje a techniky
University Course Timetabling Problem

Nikola Valešová
xvales02@stud.fit.vutbr.cz

9. května 2018

1 Úvod

Tato práce se zabývá optimalizačním problémem *University Course Timetabling Problem* (dále označovaného jako *UCTP*). *UCTP* je definován jako množina místností, časů v rozvrhu, studentů a vlastností, jejichž splnění jednotlivé kurzy vyžadují. Řešením *UCTP* je potom přiřazení volné místnosti a místa v rozvrhu každému z daného počtu kurzů za splnění všech určených podmínek. Vzhledem k omezeným zdrojům (jakými jsou například kapacity místností nebo vlastnosti potřebné pro kurz) se tento problém může stát velmi komplikovaným a rozsáhlým. Jak je uvedeno například v článku [3], *UCTP* spadá do kategorie NP-těžkých problémů, avšak jelikož se jedná o problém, který je v praxi často řešený a analyzovaný, vzniklo již nespočet metod, které jsou používány při implementaci řešení. Kategorie, do kterých lze způsoby řešení optimalizačních problémů členit, jsou uvedeny ve zdroji [2]. V případě *UCTP* se často jedná o metodu simulovaného žíhání[1] nebo o aplikace genetických algoritmů a obecně algoritmů inspirovaných přírodou, jakými jsou například optimalizace mravenčí kolonií[2] a v neposlední řadě genetické algoritmy[3].

Cílem této práce je nastudování daného problému a implementace metody, která je popsána ve článku [1]. Navrhovaná metoda je pojmenována „algoritmus diferenciální evoluce“ (*Differential Evolution Algorithm*). Jedná se o algoritmus vycházející z principů genetických algoritmů, avšak na rozdíl od nich není stěžejní operací křížení, nýbrž mutace.

Poslední sekce této práce je věnována experimentálnímu srovnání uvedené metody s dalšími metodami a zobrazení vlivu hodnoty jednotlivých parametrů na kvalitu populace.

2 Popis problému

UCTP je optimalizační problém, který se zabývá procesem tvorby rozvrhů. Cílem je nalézt takové rozmístění kurzů do učeben a časů v rozvrhu, které není v rozporu s žádnou z neporušitelných omezení (tzv. *hard constraints*), a zároveň má co nejnižší skóre výskytu porušitelných omezení (tzv. *soft constraints*).

2.1 Formální definice problému

Instance *UCT* problému je definována

- množinou N kurzů – $E = \{e_1, \ldots, e_N\}$,
- 45 místy v rozvrhu,
- množinou R místností,
- množinou F znaků a
- množinou M studentů,

přičemž u každé z místností je dále uvedena její kapacita a znaky, které poskytuje, u každého studenta je vypsán seznam předmětů, jenž má zapsány, a pro každý kurz jsou uvedeny požadované znaky.

[1] Příklad metody řešení *UCTP* využívající simulovaného žíhání lze nalézt například v článku [4]
[2] Jedna z metod řešení *UCTP* založená na principu optimalizace mravenčí kolonií je uvedená v článku [5]
[3] Příklad aplikace genetických algoritmů je uveden v článku [6]

2.2 Porušitelná a neporušitelná omezení

Jakmile je nalezeno řešení dané instance *UCTP*, je nejprve posuzováno z hlediska splnění neporušitelných omezení. Pokud není v rozporu se žádným z potřebných omezení, je řešení označeno jako *feasible*, jedná se tedy o validní a použitelné řešení. K posouzení a srovnání kvality aktuálního výsledku potom slouží porušitelná omezení. Čím nižší je hodnota rozporu s těmito omezeními pro dané řešení, tím je řešení kvalitnější. U genetických algoritmů právě toto skóre představuje hodnotu *fitness* funkce daného řešení – chromozomu.

Mezi čtyři základní neporušitelná omezení patří:

- H_1 – žádný student nesmí mít v jeden čas naplánovaný více než jeden kurz,
- H_2 – místnost musí poskytovat všechny znaky požadované kurzem, který se v ní má odehrávat,
- H_3 – počet studentů kurzu nesmí přesáhnout kapacity místnosti, ve které se kurz koná,
- H_4 – v žádné místnosti nesmí být v jeden čas naplánovaný více než jeden kurz.

Třemi porušitelnými omezeními udávajícími kvalitu daného řešení jsou:

- S_1 – student by neměl mít kurz v posledním časovém okénku v daný den,
- S_2 – student by neměl mít více než dva kurzy ihned po sobě,
- S_3 – student by neměl mít v jeden den pouze jediný kurz.

Hodnota skóre řešení za porušení jednotlivých porušitelných omezení je počítána tak, že se přičítá jeden bod za každého studenta, který má

- zapsanou přednášku umístěnou do posledního časového okénka daný den,
- více než dvě přednášky v nějaký den a
- v nějaký den jen jedinou přednášku.

Kvalita celé populace je potom dána vztahem

$$\min \sum_{n=1}^{N} s_1 + s_2 + s_3,$$

kde s_i označuje skóre nesplnění i-tého porušitelného omezení a N představuje počet jedinců v populaci.

3 Analýza algoritmu

Implementovaný algoritmus lze logicky rozčlenit na dvě fáze – v první části je vytvořena počáteční populace, která představuje soubor validních rozvrhů, a následně je aplikován *differential evolution* algoritmus, jehož cílem je zvýšení kvality řešení při zachování neporušenosti všech neporušitelných omezení.

3.1 Tvorba počáteční populace

Prvním krokem algoritmu je vytvoření počáteční populace, tedy vyprodukování daného počtu chromozomů, z nichž každý představuje validní řešení zadané instance *UCTP*.

Proces tvorby jednoho rozvrhu spočívá v postupném přiřazování místností a časů v rozvrhu každému z kurzů. Pro zvýšení úspěšnosti této části algoritmu jsou nejprve kurzy seřazeny podle počtu místností, do kterých je lze přiřadit (což je určeno na základě kapacity místnosti a jejích znacích). Následně jsou seřazené kurzy do rozvrhu umisťovány postupně od těch, kterým lze přiřadit nejmenší množství místností. Při tomto procesu nejsou porušitelná omezení vůbec brána v potaz.

Pokud se tímto způsobem podaří vytvořit *feasible* řešení, je aktuální chromozom přidán do populace a algoritmus pokračuje tvorbou nového řešení, nebo přechodem do druhé části algoritmu. V opačném případě, kdy se algoritmus dostane do situace, že pro aktuálně umisťovaný kurz neexistuje již žádný volný čas a místnost, jsou na dosud umístěné kurzy aplikovány tzv. *neighbourhood moves*. Ty mají za cíl přeuspořádat již hotovou část rozvrhu tak, aby se uvolnilo místo pro aktuálně umisťovaný kurz. Nejprve je několikrát proveden *neighbourhood move* N_1, pokud se podařilo rozvrh úspěšně přeuspořádat, algoritmus pokračuje umístěním aktuálního kurzu. Jinak je opakovaně vykonán *neighbourhood move* N_2. Pokud ani nyní nelze najít místo pro aktuálně umisťovaný kurz, pokračuje se umisťováním následujících kurzů.

3.1.1 *Neighbourhood move* N_1

Při přeuspořádání rozvrhu s využitím *neighbourhood move* N_1 je náhodně vybrán jeden již umístěný kurz a následně umístěn na jiné místo v rozvrhu, ve kterém vznikne nejmenší penalizace aktuálního řešení.

3.1.2 *Neighbourhood move* N_2

Při aplikaci *neighbourhood move* N_2 je nejprve náhodně vybrána místnost, dále jsou náhodně vygenerovány čísla dvou již umístěných kurzů, které jsou v dosavadním rozvrhu naplánovány do dané místnosti, a jsou jim prohozeny časy konání.

Výše popsaný způsob by měl vést ke vzniknu jednoho validního řešení daného problému. V případě, že se nepodaří aktuální rozvrh úspěšně naplnit, je toto řešení odstraněno z populace.

3.2 Optimalizační algoritmus

V okamžiku, kdy je vygenerovaná kompletní počáteční populace, může započít druhá fáze celého algoritmu, kterou je optimalizace evolučním algoritmem. Implementovaný algoritmus diferenciální evoluce je inspirovaný generickými algoritmy a jeho hlavní důraz je kladen na provádění mutace. Jedna iterace popisovaného algoritmu sestává ze čtyř částí, kterými jsou mutace, křížení, evaluace a selekce. Tato sekvence kroků se aplikuje opakovaně, dokud není dosažena alespoň jedna z ukončujících podmínek. Těmi jsou dosažení pevně stanoveného počtu iterací (generací) a získání řešení, jehož hodnota *fitness* funkce je rovna 0. Základní schéma algoritmu je uvedeno v algoritmu 1.

Algoritmus 1: Schéma algoritmu diferenciální evoluce

```
while not ukoncujici_podminky_splneny do
    Mutace
    Křížení
    Evaluace
    Selekce
```

3.2.1 Mutace

Mutace je stěžejní částí implementovaného algoritmu diferenciální evoluce. Prvním krokem mutace je náhodný výběr dvou rodičovských chromozomů (tedy dvou úplných řešení) z populace. Poté se provede mutace, a to tak, že je vybrán jeden ze dvou mutačních operátorů a ten je následně aplikován na každého z rodičů zvlášť. Mutační operátory jsou stejné jako *neighbourhood moves* popsané v sekci 3.1. Tím vznikne upravený pár rodičovských chromozomů.

3.2.2 Křížení

Křížení je operace, jejímž cílem je vygenerování dvou nových potomků. Vstupem křížení jsou dva rodičovské chromozomy získané pomocí mutace. Pro každého z rodičů je náhodně vygenerován jeden čas v rozvrhu a v rodičovských chromozomech jsou vzájemně prohozeny kurzy naplánované na daný čas. Aby byla zachována validita řešení i u potomků, jsou na výměnu kurzů kladeny dvě podmínky:

1. kurz může být přesunut pouze, pokud je ve druhém rozvrhu na přesouvané pozici volno,
2. přesunutím kurzu na nové místo nedojde ke vzniku žádných kolizí mezi kurzy.

3.2.3 Evaluace a selekce

Evaluace slouží k ohodnocení nově vzniklých potomků. K ohodnocení každého z potomků je použita *fitness* funkce, která počítá penalizaci za nedodržení porušitelných omezení. Jestliže má lepší potomek nižší hodnotu *fitness* funkce nežli dosavadní nejlepší jedinec v populaci, je v populaci nejhůře ohodnocený chromozom nahrazen lepším z potomků.

4 Popis implementace

Jako implementační jazyk byl zvolen *Python* (verze 2.7.14) z důvodu snadné přenositelnosti výsledného programu. Při implementaci byly využity pouze standardní knihovny, například *numpy*, *time*, *random* a *copy*.

4.1 Implementační detaily

Implementovaný algoritmus odpovídá algoritmu popsanému v sekci 3. Za účelem uložení informací o vstupní instanci definice problému slouží třída `CTTProblem`. Jejími atributy jsou hodnoty počtu místností, studentů, kurzů a znaků, které jsou získány přímo při načítání vstupního souboru. Dále jsou to také atributy vypočtené z těchto informací, těmi jsou například místnosti vyhovující každému z kurzů, kolizní kurzy, které nemohou být naplánovány současně, atd. Tyto atributy jsou vypočteny ihned v rámci inicializace třídy. Tento přístup má o něco horší prostorovou složitost, jelikož je zde spousta informací i redundantně a bylo by je možné odvodit z jiných dat, avšak časová složitost je zde znatelně lepší, díky tomu, že není třeba opakovaně počítat to, co je neměnné. Především u populací o velikosti 100 000 a více jedinců je toto zrychlení výrazné.

Po dokončení inicializace třídy je zavolána metoda `isCorrect`, která zkontroluje, zda naplněná instance `CTTProblem` obsahuje validně zadaný problém. Ověřována je především správnost hodnot celočíselných atributů.

Dalším krokem je vytvoření počáteční populace. K tomu slouží metoda `createInitialPopulation`, která vytvoří populaci o požadované velikosti opakovaným voláním metody `_createFeasibleSolution`. Ta se výše popsaným způsobem snaží umístit postupně všechny kurzy a vytvořit tak kompletní rozvrh. Pokud nějaký kurz není kam umístit, aplikují se již zmíněné *neighbourhood moves*, konkrétně k tomuto účelu slouží metody `_applyN1` a `_applyN2`.

V další fázi výpočtu je zavolána metoda `removeUnfeasibleSolutions`, která z populace odstraní ta řešení, která nepředstavují kompletní a validní řešení. Jelikož je splnění neporušitelných omezení kontrolováno již při utváření rozvrhu, jsou zde odstraňovány pouze ty chromozomy, které nejsou úplně definované, tedy ani po aplikování *neighbourhood moves* nebylo pro nějaký kurz nalezeno vhodné umístění.

Následně je v metodě `evaluatePopulation` vypočteno ohodnocení všech chromozomů v populaci na základě toho, jak moc nesplňují porušitelná omezení. Pro vypočtení skóre penalizace za porušení omezení S_1, S_2 a S_3 jsou implementovány metody `_s1Score`, `_s2Score` a `_s3Score`. Kvalita každého jedince je pak dána jako součet těchto tří hodnot. Podle ohodnocení jednotlivých chromozomů je pak celá populace seřazena od nejkvalitnějšího řešení po to nejméně kvalitní v metodě `orderPopulation`.

Nyní může započít samotný algoritmus diferenciální evoluce. Dokud není přesažen stanovený počet iterací, nebo nejlepší řešení v populaci nemá nulovou hodnotu *fitness* funkce, je opakovaně volána metoda `executeDEAlgorithm`, která provádí jednu iteraci implementovaného algoritmu. Jsou tedy postupně vykonány metody `_mutate`, `_crossover`, dále je provedena evaluace a pokud představuje jeden z potomků lepší řešení než dosavadní nejlepší, nahradí nový potomek nejhorší řešení v populaci.

4.2 Modifikace algoritmu

Implementace původního algoritmu měla často problémy s nalezením *feasible* řešení. Článek navíc neobsahuje striktně daný detailní popis, jak má tvorba původní populace probíhat. Díky tomu byl tedy v této oblasti prostor pro změnu a pro vylepšení stávajícího algoritmu.

Umisťování kurzů stále probíhá postupně od těch nejvíce problematických. Pro výběr místnosti a času, do jakých bude kurz naplánován, je implementována metoda `_getBestPlacements`, která ze všech volných umístění vybere ty nejvhodnější. Nejprve jsou vybrány nejvhodnější časová okna, což jsou ta, ve kterých je již nejvíce místností obsazených. Důvodem je to, že takovéto časové sloty budou v budoucnu nejhůře obsaditelné, jelikož je vysoká pravděpodobnost, že zde bude již umístěn nějaký konfliktní kurz.

Všechna možná umístění kurzu jsou tedy omezena na ty, které se konají ve vybraných nejvhodnějších časových oknech. Následně jsou ze zbývajících umístění vybrána ta, která se konají v nejvhodnějších místnostech. To jsou místnosti, do kterých může být umístěn nejmenší počet ještě nenaplánovaných kurzů. Zde je důvod jasný, opět je cílem zabezpečit co nejvíce možností pro umístění dalších kurzů.

Z umístění, která splňují výše uvedené podmínky jak pro místnost, tak i pro časové okno, je potom náhodně vybráno výsledné umístění.

4.3 Spuštění aplikace

Pro spuštění na referenčním serveru `merlin` není třeba instalace žádných dodatečných knihoven.

Syntaxe příkazu pro spuštění aplikace je následující:

```
xvales02.py [-h] -i INPUT [-p POPULATION_SIZE] [-g GENERATIONS_COUNT]
            [-c CROSSOVER_RATE] [-m MUTATION_RATE].
```

Přehled významu všech parametrů aplikace je uveden v tabulce 1. Valná většina parametrů je volitelných. V případě, že uživatel některý z nich nezadá, použije se implicitní hodnota uvedená v téže tabulce. Výjimku tvoří parametr `input`, ten je nutné zadat vždy.

Parametr	Význam parametru	Implicitní hodnota
`-h|--help`	vypíše nápovědu a ukončí program	-
`-i|--input`	vstupní soubor obsahující popis problému	-
`-p|--population-size`	velikost populace	50
`-g|--generations-count`	počet generací	200000
`-c|--crossover-rate`	koeficient křížení	0,8
`-m|--mutation-rate`	koeficient mutace	0,5

Tabulka 1: Popis významu jednotlivých parametrů a jejich implicitních hodnot

5 Experimentální fáze

Odkaz na sadu testů, na kterých byla testována původní implementace, již není funkční a ani na jiných serverech ji nebylo možné dohledat. Pro otestování této aplikace byla tedy vytvořena nová sada testovacích vstupů. Stejně jako původní sada obsahuje ta nová celkem 11 testů, z nichž 5 odpovídá specifikaci malého testovacího vstupu, dalších 5 je středních a 1 vstupní problém spadá do kategorie velkých vstupů. Popis vlastností, které splňuje vstupní soubor dané velikosti, je uveden v tabulce 2.

Kategorie	Malé	Střední	Velké
Počet kurzů	100	300 – 400	400
Počet místností	5	10	10
Počet znaků	5	5	10
Počet studentů	80	200	200
Maximální počet kurzů na studenta	20	20	20
Maximální počet studentů na kurz	20	50	100
Průměrný počet znaků na místnost	3	3	5
Procentuální využití znaků	70	80	90

Tabulka 2: Popis vlastností jednotlivých kategorií testovacích vstupů

Ve snaze co nejvíce napodobit experimenty provedené ve článku [1] byly při provádění experimentů všechny argumenty nastaveny na stejné hodnoty jako při původních experimentech prováděných autory článku. Přehled použitých hodnot je uveden v tabulce 3. Tyto hodnoty platí pro všechny běhy všech experimentů, pokud není explicitně uvedeno jinak.

Parametr	Hodnota
Počet generací	200 000
Velikost populace	50
Koeficient křížení	0, 8
Koeficient mutace	0, 5

Tabulka 3: Nastavení parametrů aplikace při provádění testů

5.1 Srovnání výsledků implementace s dalšími algoritmy

Stejně jako u původních experimentů byl každý testovaný problém spuštěn celkem 11-krát a do této zprávy byla vybrána vždy nejnižší dosažená hodnota výsledku. Vzhledem k absenci původní sady byly výsledky implementace porovnány s implementacemi algoritmů vybraných kolegy. Konkrétně se jedná o algoritmy:

- Ondřej Valeš (xvales03) – *Using a randomised iterative improvement algorithm with composite neighbourhood structures for the University Course Timetabling Problem*,
- Matej Marušák (xmarus06) – *A hybrid evolutionary approach to the University Course Timetabling Problem*,
- Radek Vít (xvitra00) – *A honey-bee mating optimization algorithm for Educational Timetabling Problems*.

Srovnání algoritmů na jednotlivých testovacích vstupech je uvedeno v tabulce 5.1. Pro každou vstupní instanci problému je zde tučně vyznačena nejlepší z dosažených hodnot, tedy algoritmus, který na daném vstupu dosáhl nejkvalitnějšího řešení. Implementovaný algoritmus dosáhl celkem třikrát nejlepšího výsledku, jedná se tedy o konkurenceschopný algoritmus. Naopak ve čtyřech případech dopadl vůbec nejhůře. To může být způsobeno mimo jiné tím, že algoritmus byl publikován v roce 2012 a od té doby došlo v algoritmech tohoto typu k dalšímu vývoji.

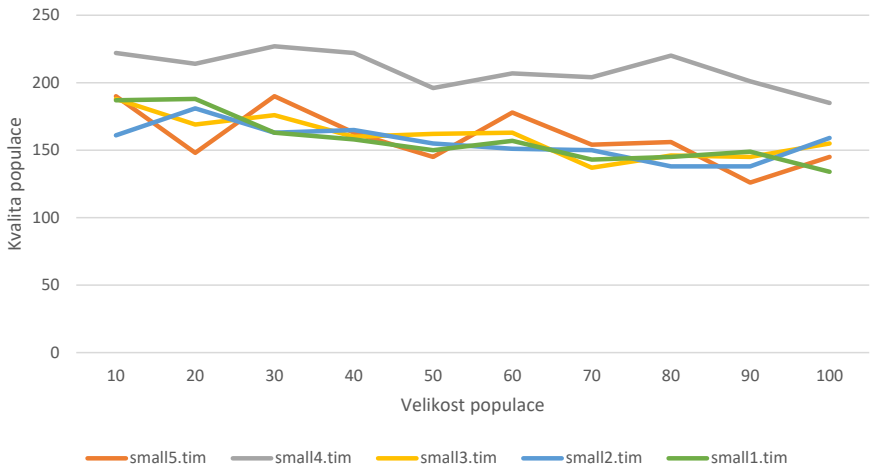
Pro srovnání s původní implementací jsou v tabulce 5.1 uvedeny také hodnoty získané autory původního článku. Toto srovnání však nemá příliš velkou vypovídající hodnotu, jelikož vstupní instance si sice odpovídají velikostí, avšak stále se jedná o různé problémy a hodnoty nejlepších možných řešení se mohou značně lišit. Reimplementace dosahuje lepších výsledků pouze pro vstupy `medium2.tim`, `medium4.tim` a `large.tim`.

Vstupní soubor	Článek [1]	xvales02	xvales03	xmarus06	xvitra00
small1.tim	0	47	31,2	33	**11**
small2.tim	0	**21**	34,5	32	**21**
small3.tim	0	38	28,0	26	**16**
small4.tim	0	53	**41,7**	42	45
small5.tim	0	37	32,6	30	**20**
medium1.tim	160	237	255,9	234	**27**
medium2.tim	81	**62**	126,2	132	154
medium3.tim	149	205	289,8	269	**117**
medium4.tim	113	66	173,1	170	**26**
medium5.tim	143	76	168,2	145	**27**
large.tim	735	**342**	408,4	468	369

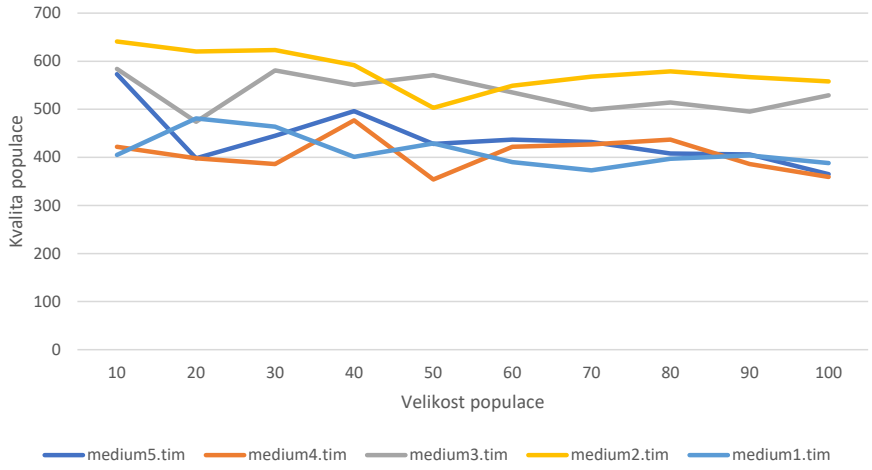
5.2 Vliv hodnoty parametru `population-size` na kvalitu řešení

Další část testování se zabývá zkoumáním vlivu hodnoty parametru `population-size` na kvalitu populace. Tento parametr má jistě vliv na hodnotu *fitness* funkce řešení, jelikož čím větší je původní populace, tím více rodičovských chromozomů může být vybráno pro křížení, a tedy tím rozmanitější synovské chromozomy můžeme získat.

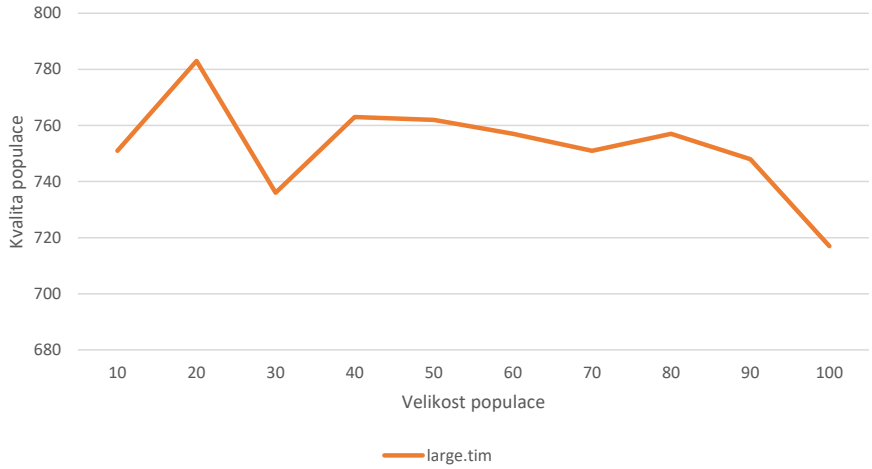
Pro každý vstupní soubor bylo experimentováno s velikostí populace v rozmezí od 10 do 100 jedinců s krokem 10. Počet generací byl vždy nastaven na hodnotu 100, aby se iterativní zvyšování kvality populace příliš nepromítalo do výsledků experimentů. Výsledná charakteristika pro každý z 11 testovacích vstupů je uvedena v grafu 1. Z uvedených grafů je patrné, že náhodnost má na kvalitu populace velký vliv, avšak při pohledu na celé křivky se potvrzuje očekávaná hypotéza, že čím větší počet jedinců se v populaci nachází, tím vyšší kvalitu populace má.



(a) Pro malé vstupní instance



(b) Pro střední vstupní instance



(c) Pro velkou vstupní instanci

Obrázek 1: Vliv velikosti populace na hodnotu *fitness* funkce

5.3 Vliv hodnoty parametru `generations-count` na kvalitu řešení

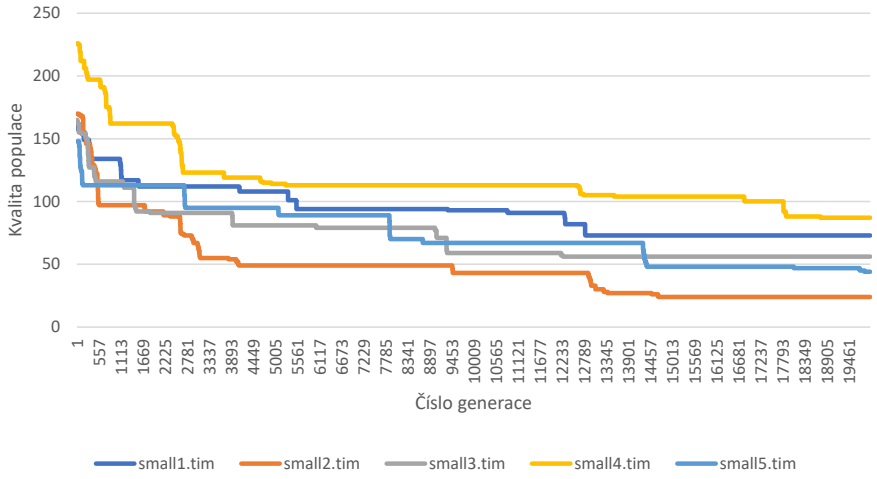
Poslední část testování je věnována prozkoumání vlivu hodnoty parametru `generations-count` na kvalitu výsledného řešení. Právě tento parametr výrazně ovlivňuje hodnotu *fitness* funkce výsledku,

jelikož v každé provedené generaci může dojít ke zkvalitnění populace. Jeho zkoumání může být užitečné také v případě, že bychom aplikaci chtěli využít pro kvalitní výpočty v co nejkratším čase, protože pro vyšší počet generací začne znatelně růst doba výpočtu.

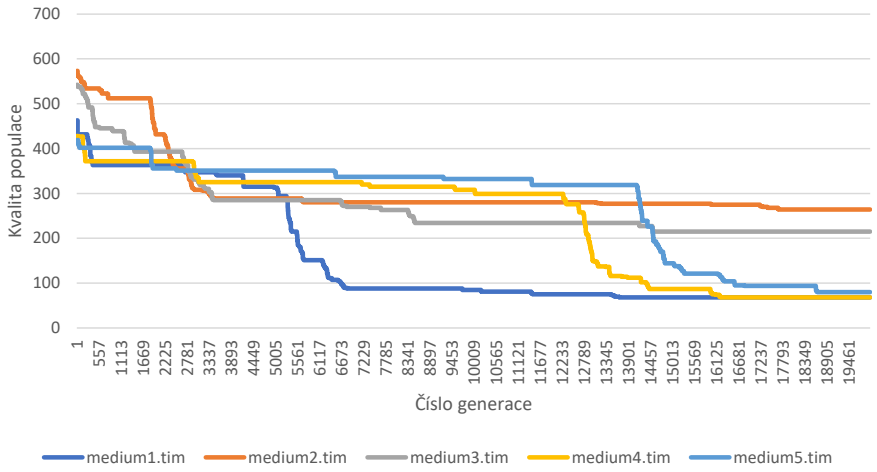
Pro každý vstupní soubor byl experiment proveden celkem 10-krát a do grafu byl poté zakreslen ten běh, který dosáhl největšího zlepšení v průběhu generací. Experimenty probíhaly s velikostí populace 50 jedinců a kvalita populace byla zaznamenávána v rozsahu od 0 do 20 000 generací s krokem 1. Hodnoty zakreslené v grafu byly získávány výpisem z aplikace po každé provedené iteraci.

Výsledná charakteristika pro každý z 11 testovacích vstupů je uvedena v grafu 2. Ve všech případech je jasně zřetelné zlepšení, zpočátku křivky klesají téměř logaritmicky, dále převážně lineárně. Největšího zlepšení bylo dosaženo u vstupu `medium1.tim`, a to o hodnotu 395.

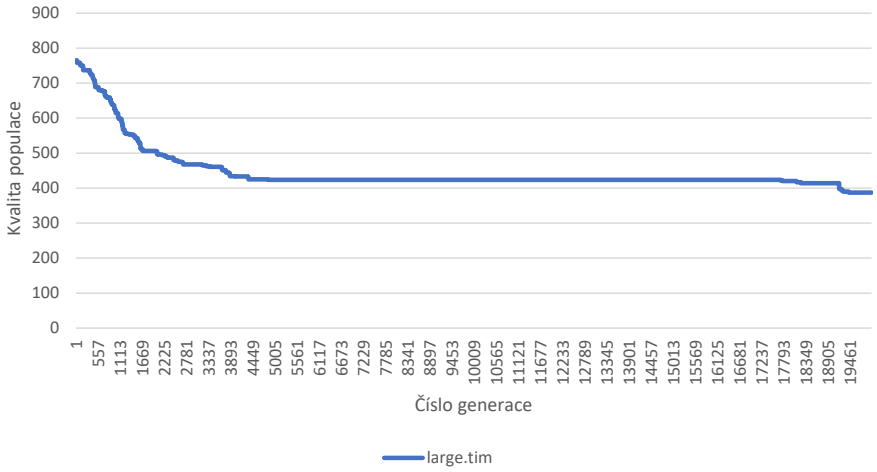
Tyto experimenty potvrdily jak správnost reimplementace algoritmu, tak také funkčnost a efektivnost algoritmu původního.



(a) Pro malé vstupní instance



(b) Pro střední vstupní instance



(c) Pro velkou vstupní instanci

Obrázek 2: Vliv počtu generací na hodnotu *fitness* funkce populace

6 Závěr

V rámci tohoto projektu byl nastudován a reimplementován algoritmus s názvem *Differential Evolution Algorithm*. Následně bylo s implementací provedeno několik experimentů, a to srovnání tohoto přístupu s dalšími algoritmy, zkoumání vlivu velikosti populace na kvalitu řešení a také zobrazení počtu generací na kvalitu celkové populace. Při srovnání implementovaného algoritmu s dalšími optimalizačními metodami si implementovaný algoritmus vedl vcelku obstojně, na některých vstupech poskytl nejlepší řešení, na jiných naopak nejhorší. Při zkoumání vlivu obou zmíněných parametrů byl patrný jejich vliv na hodnotu kvality populace, zejména u počtu generací. Těmito experimenty byla také ověřena funkčnost reimplementace.

Reference

[1] K. Shaker, S. Abdullah, A. Hatem. *A Differential Evolution Algorithm for the University course timetabling problem* [online], 2012 [cit. 2018-4-29]. Dostupné z: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6329805&isnumber=6329780.

[2] P. Peringer. *SNT – Simulační nástroje a techniky* [online], 2018, 123 – 130 [cit. 2018-4-30]. Dostupné z: https://www.fit.vutbr.cz/study/courses/SNT/public/Prednasky/SNT.pdf.

[3] K. Socha, M. Sampels, M. Manfrin. *Ant Algorithms for the University Course Timetabling Problem with Regard to the State-of-the-Art* [online], 2003, [cit. 2018-4-28]. Dostupné z: https://link.springer.com/chapter/10.1007/3-540-36605-9_31.

[4] R. Bai, E. K. Burke, G. Kendall, B. Mccollum. *A Simulated Annealing Hyper-heuristic for University Course Timetabling Problem* [online], 2006 [cit. 2018-4-29]. Dostupné z: https://www.researchgate.net/publication/228948408_A_Simulated_Annealing_Hyper-heuristic_for_University_Course_Timetabling_Problem.

[5] K. Socha, J. Knowles, M. Sample. *A MAX-MIN Ant System for the University Course Timetabling Problem* [online], 2002 [cit. 2018-4-29]. Dostupné z: https://www.researchgate.net/publication/2540563_A_MAX-MIN_Ant_System_for_the_University_Course_Timetabling_Problem.

[6] S. Abdullah, H. Turabieh. *Generating University Course Timetable Using Genetic Algorithms and Local Search* [online], 2008 [cit. 2018-4-29]. Dostupné z: https://www.researchgate.net/publication/232627738_Generating_University_Course_Timetable_Using_Genetic_Algorithms_and_Local_Search.