Mendelova univerzita v Brně
Provozně ekonomická fakulta

Ant Colony Optimization: Implementace a testování biologicky inspirované optimalizační metody

Diplomová práce

Vedoucí práce:

doc. Ing. Jan Žižka, CSc.

Bc. Michal Havlík

Brno 2015

LIST ZADANÍ

Děkuji vedoucímu této diplomové práce doc. Ing. Janu Žižkovi, CSc. za cenné rady a čas věnovaný konzultacím.

Čestné prohlášení

Prohlašuji, že jsem tuto práci: **Ant Colony Optimalization : Implementace a testování biologicky inspirované optimalizační metody** vypracoval samostatně a veškeré použité prameny a informace jsou uvedeny v seznamu použité literatury. Souhlasím, aby moje práce byla zveřejněna v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách ve znění pozdějších předpisů, a v souladu s platnou *Směrnicí o zveřejňování vysokoškolských závěrečných prací.*

Jsem si vědom/a, že se na moji práci vztahuje zákon č. 121/2000 Sb., autorský zákon, a že Mendelova univerzita v Brně má právo na uzavření licenční smlouvy a užití této práce jako školního díla podle § 60 odst. 1 Autorského zákona.

Dále se zavazuji, že před sepsáním licenční smlouvy o využití díla jinou osobou (subjektem) si vyžádám písemné stanovisko univerzity o tom, že předmětná licenční smlouva není v rozporu s oprávněnými zájmy univerzity, a zavazuji se uhradit případný příspěvek na úhradu nákladů spojených se vznikem díla, a to až do jejich skutečné výše.

V Brně dne 20. května 2015 _____

Abstract

Havlík, M. Ant Colony Optimization: Implementation and testing of bio-inspired optimization method. Diploma thesis. Brno, 2015.

This thesis deals with the implementation and testing of algorithm Ant Colony Optimization as a representative of the family of bio-inspired optimization methods. A given algorithm is described, analyzed and subsequently put into context with the problems which can be solved. Based on the collected information is designed implementation that solves the Traveling salesman problem. Implementation contains graphical user interface to track the algorithm. Implementation is further optimized using parallel programming and other methods. Finally the implementation compared and summarized results.

Keywords

Ant Colony Optimization, Traveling salesman problem, Algorithm complexity, C#, Parallel programming

Abstrakt

Havlík, M. Ant Colony Optimization: Implementace a testování biologicky inspirované optimalizační metody. Diplomová práce. Brno, 2015.

Tato diplomová práce se zabývá implementací a testováním algoritmu Ant Colony Optimalization, jakožto algoritmu z rodiny optimalizačních metod inspirovaných přírodními procesy. Daný algoritmus je popsán, zanalyzován a následně dán do kontextu s problémy, jež může řešit. Na základě zjištěných poznatků je navržena implementace, která řeší problém obchodního cestujícího. Implementace je obohacena o grafické uživatelské rozhraní, umožňující sledovat průběh algoritmu. Běh implementace je dále optimalizován pomocí paralelního programování a dalších metod. Na závěr je implementace porovnána a shrnuty dosažené výsledky.

Klíčová slova

Ant Colony Optimization, Problém obchodního cestujícího, Složitost algoritmu, C#, paralelní programování

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod a cíl práce

1.1 Úvod

Optimalizace je něco s čím se každý z nás setkává v každodenním životě a často ji i sami podvědomě provádíme. Málokdy si to však uvědomujeme. Každé ráno se například vydáváme na cestu do školy či práce. Jestli zvolíme jako dopravní prostředek automobil, městskou hromadnou dopravu, či půjdeme po svých, záleží na mnoha okolnostech. Jakmile však tuto volbu provedeme, přiřadíme k dopravnímu prostředku adekvátní, pravděpodobně nejrychlejší, cestu a tu následně vykonáme. Ti nejúspěšnější z nás budou v očekávaný čas na daném místě, ti méně šikovní dorazí se zpožděním, i oni však budou pravděpodobně úspěšnější, než ti, kteří nechávají volbu cestu, včetně času na který si nastaví budík, náhodě.

V reálném světě existuje samozřejmě nepřeberné množství problémů, které lze optimalizovat a podstatná část z nich bude podstatně složitější než výše zmíněný problém. Mnoho z těchto úloh, jsou lidé schopni úspěšně řešit, jiné z nich jsou pro běžné lidi zcela neřešitelné. Počítačová automatizace, nám však každopádně umožňuje různé úlohy řešit opakovaně a ve velmi krátkém čase.

Algoritmus Ant Colony Optimalization, jak jeho název napovídá, patří mezi algoritmy inspirované přírodou, díky tomu je jeho základní princip jednoduše vysvětlitelný i osobám, které se nepohybují v informatickém prostředí. V našem případě, však pochopitelně půjdeme mnohem více do hloubky a nespokojíme se pouze se zevrubným pochopením principů algoritmu, ale spojíme jej i s vlastním návrhem a následnou implementací.

1.2 Cíl práce

Cílem diplomové práce je navrhnout efektivní implementaci zadaného algoritmu. Vytvořená implementace by dále měla poskytovat uživatelům možnost pohodlného ovládání a dát jim možnost daný algoritmus lépe poznat a pochopit. Tento požadavek vychází z očekávaného využití vzniklé aplikace, vedoucím této diplomové práce, při výuce na naší univerzitě.

K naplnění tohoto cíle nevede krátká cesta, a proto k jejímu úspěšnému absolvování je třeba splnit několik dílčích cílů. Prvním je vlastní seznámení se s algoritmem a jeho možnostmi. Druhým je na základě získaných poznatků navrhnout vyhovující implementaci. Ve chvíli, kde bude návrh realizován, je třeba prioritně ověřit jeho funkčnost a bude-li splněna, můžeme se pustit do zhodnocení algoritmu a jeho případných optimalizacím.

Klíčovou součástí úspěšného řešení musí být také návrh vhodné testovací metodiky, která bude zohledňovat v daném případě skutečně důležité parametry algoritmu. S volbou testovací metodiky, také souvisí volba vhodných vstupních testovacích dat.

2 Teoretická část

2.1 Ant Colony Optimalization

Ant Colony Optimalization (dále ACO), do češtiny většinou překládáno jako optimalizace mravenčí kolonií. Jedná se multiagentní systém, v něm je chování jednotlivých agentů, inspirováno chováním skutečných mravenců. Myšlenka napodobit chování mravenců, při hledání vhodných řešeních kombinatorické optimalizace, bylo iniciováno Marco Dorigem. Princip této metody je založen na způsobu, jakým mravenci vyhledávají cestu k potravě a zpět do jejich mraveniště. Během cest mravenců je zanechávána chemická feromonová cesta. Úlohou feromonu, je řídit další mravence směrem k cílovému bodu. Cesta jednotlivých mravenců je tedy vybírána v závislosti na množství feromonu. (Toksari, 2005)

2.1.1 Od skutečných mravenců k virtuálním

Mravenčí kolonie a další socializované hmyzí kolonie, například včely, vosy, či termiti (Krömer, 2012), lze považovat za distribuované systémy, které nehledě na jednoduchost jejich jedinců, představují vysoce strukturovanou sociální organizaci. V důsledku této organizace, mohou mravenčí kolonie dosahovat úspěšných řešení problémů, jejichž složitost vysoce přesahuje schopnosti jednoho mravence.

Oblast tzv. mravenčích algoritmů studuje modely chování vypozorované ze sledování reálných mravenců a používá tyto modely jako zdroj inspirace pro navrhování nových algoritmů pro řešení optimalizačních problémů.

Hlavní myšlenkou je samoorganizační princip, celé kolonie. Dalšími inspirativními hledisky, která můžeme sledovat u mravenců, je dělba práce či spolupráce při transportu. Ve všech těchto případech mravenci koordinují své aktivity pomocí stigmergie, formy nepřímé komunikace prováděné za pomoci modifikace životního prostředí. Biologové ukázali, že většina chování pozorovaného uvnitř hmyzích kolonií, se bez principu stigmergie neobejde. U některých druhů mravenců, bylo zjištěno, že jsou ve skutečnosti zcela slepí a komunikace pomocí feromonů, tak představuje jediný možný způsob komunikace. To je zásadní rozdíl od lidí, či jiných vyšších živočišných druhů, kde hlavní množství informací je přijímáno zrakově, či sluchově. Myšlenka ACO tedy poté převádí princip komunikace pomocí prostředí do virtuálního světa. (Dorigo, 2004)

2.1.2 Experiment s dvojitým mostem

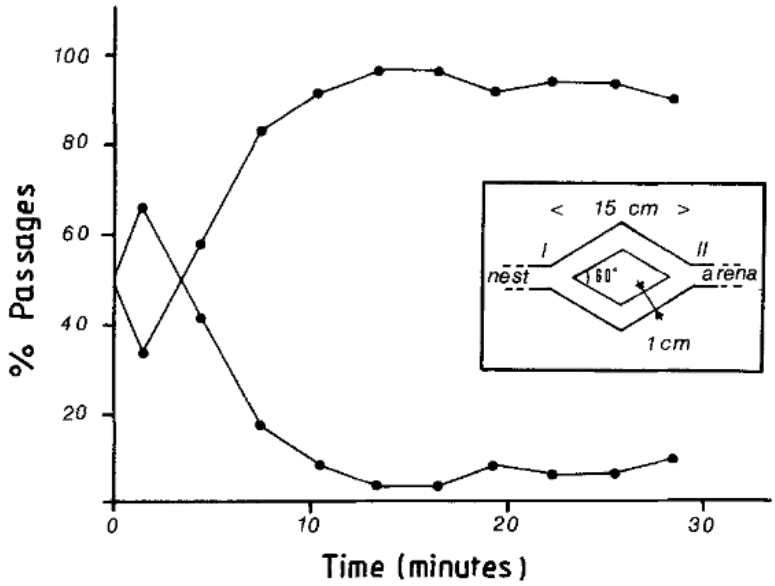
Mravenci tedy při hledání cesty k potravě mají tendenci, vybírat s větší pravděpodobností cestu s vyšším množstvím feromonu. Tvoření feromonové stezky a s ním spojené chování bylo zkoumáno v kontrolovaných pokusech několika výzkumných pracovišť. Jeden obzvláště významný experiment byl navržen a proveden pod vedením L. Deneubourga. (Dorigo, 2004)

Experiment probíhal následovně. Laboratorní kolonii mravenců druhu Linepithema Humilis byl dán přístup do prostoru o rozměrech 0,8 m × 0,8 m. Tento

prostor byl pokryt bílým pískem. Přístup z hnízda do prostoru byl realizován umístěním mostu. Prostor byl v průběhu experimentu v minutovém intervalu fotografován. K případné reinicializaci a opakování pokusu, postačilo vyměnit písek v prostoru.(Deneubourg, 1990)

Most byl rozdělen do dvou větví. Průzkumníci byli stavěni před binární volbu levé nebo pravé větve. Úhel mezi větvemi byl 60°. Většina mravenců při opuštění hnízda pokračovala na most, kde prošla jednou ze dvou větví a pokračovala dále, jen malé množství mravenců po průchodu větví zvolilo ihned návrat větví druhou. Vzdálenost mezi počátkem a koncem větví byla 15 cm a provoz na jednotlivých větvích byl počítán v 3 minutových intervalech.

Na počátku pokusu byla každá z větví vybírána stejně často, avšak průchod každého jednoho mravence, aktualizoval pravděpodobnost, s jakou následující mravenec vybere tu či onu cestu. (Deneubourg, 1990)



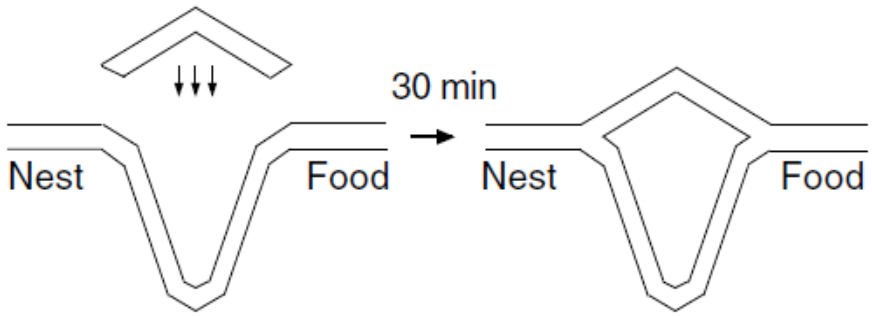
Obr. 1 Experiment s dvojitým mostem (Deneubourg, 1990)

Na obrázku 1 vidíme, jak se množství mravenců na jednotlivých větvích mostu měnilo v čase. Po uplynutí několika minut, byla jedna větev preferována u více jak 80% mravenců a tento poměr se již do konce experimentu nezměnil. To lze vysvětlit tak, že na začátku je hodnota feromonu nulová a nic tak neovlivňuje pravděpodobnost vybrání některé z větví. Nicméně vlivem náhodných fluktuací, si některou z větví vybere více mravenců, což ovlivní výběr těch následujících a tak dále. Rozdíl se tak zvětšuje, až mravenci konvergují k některé z větví.

Ve druhém experimentu, byla délka jedna z větví nastavena na dvojnásobek oproti druhé. Každý z mravenců při svém průchodu zanechává stejné množství feromonu, ale jelikož mravenci pohybující se po kratší cestě, cestu vykonají rychleji, mohou feromonovou cestu častěji obnovovat a bránit tak úbytku feromonu

na větvi. Na konci takto postaveného experimentu, drtivá většina průchodů byla vykonána po kratší větvi. (Dorigo, 2004) Vlivem menšího počtu mravenců na delší cestě, také nedocházelo k obnovování feromonu na ní a ten se postupně vypařil.

Velmi zajímavá je třetí verze experimentu, kdy ve výchozím stavu byla mravencům dostupná pouze delší cesta a cesta kratší byla zpřístupněna až po 30 minutách běhu experimentu (obr. 2).



Obr. 2 Experiment s dvojitým mostem a následným přidáním kratší větve (Dorigo, 2004)

V takovémto případě byla kratší větev vybírána jen velmi sporadicky a kolonie zůstala ve využívání delší cesty. (Dorigo, 2004) Feromonová stopa značící původní nejkratší cestu zkrátka byla příliš silná na to, aby se po nové cestě vydalo větší množství mravenců. Mravenčí kolonie tedy uvázla v lokálním extrému řešeného problému.

Další experimenty ukázali, že různé druhy mravenců používají různé metody pro volbu trasy. U mravenců druhu Lasius Niger se vyvinul postup, který jim umožní objevit i dodatečně přidanou, výhodnější cestu (Krömer, 2012)

Deneubourg s kolegy, na základě těchto experimentů vytvořil základní stochastický model, který adekvátně popisuje dynamiku mravenčí kolonie, tak jak ji vypozorovali při experimentu s dvojitým mostem.

Rovnice pro vypočítání pravděpodobnosti, se kterou bude vybrána první větev, jsou následující.

$$prob_A = \frac{(\mathrm{k} + A_i)^n}{(\mathrm{k} + A_i)^n + (\mathrm{k} + B_i)^n}$$

$$prob_A + prob_B = 1$$

$$A_{i+1} = A_i + \delta$$

$$B_{i+1} = B_i + (1 - \delta)$$

$$A_{i+1} + B_{i+1} = i$$

Jednotlivé proměnné v tomto případě znamenají následující. δ je stochastická proměnná, která nabývá hodnot 1 nebo 0 s pravděpodobností odpovídající $prob_A$ a $prob_B$. A_i a B_i vyjadřují množství feromonu.

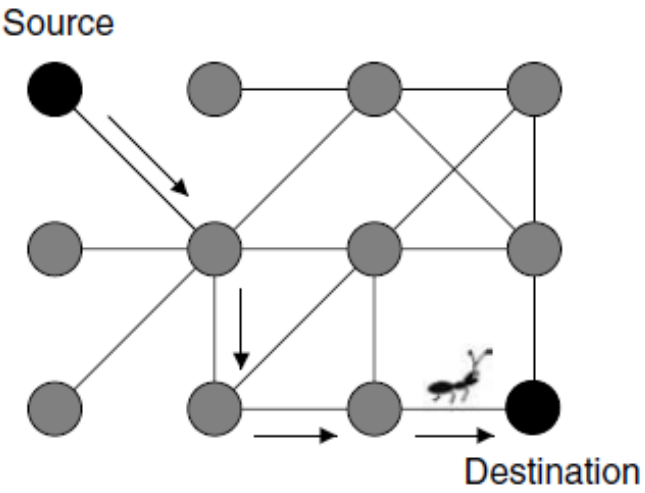
Parametr n určuje stupeň nelinearity výběru. Vyšší hodnota parametru n znamená, že pokud jedna větev má byť jen o trochu vyšší hodnotu feromonu než druhá, další mravenec bude mít podstatně vyšší pravděpodobnost, že ji vybere. Parametr k odpovídá stupni přitažlivost neoznačené větve, jinými slovy, čím vyšší parametr k, tím větší hodnota feromonu je potřebná, aby se volba stala významně nenáhodná. (Deneubourg, 1990)

Experiment s dvojitým mostem jasně ukazuje, že mravenčí kolonie má zabudovány optimalizační schopnosti. Za pomoci daných pravděpodobnostních pravidel, dokáže najít nejkratší vzdálenost mezi dvěma body v daném prostředí. Díky tomuto je možné navrhnout umělé mravence, kteří dokáží pohyb živých mravenců, při pohybu za potravou, napodobit. (Dorigo, 2004)

2.1.3 Virtuální mravenci a hledání minimální cesty

Naším cílem samozřejmě je definovat algoritmy, které mohou být použity k řešení problému nalezení minimální cesty v podstatně složitějších grafech, než je graf o dvou uzlech a dvou hranách, představující experiment s dvojitým mostem.

Představme si tedy libovolný spojitý graf s neorientovanými hranami, v něm chceme najít cestu s minimální cenou, za cenu se ve většině případů považuje délka hrany, ale může být představována i jinými atributy. Uzel, ve kterém cesta začíná, nazýváme počátečním. Uzel, ve kterém cesta končí koncovým. Ve spojení s mravenci můžeme ekvivalentně, pro počáteční a cílový uzel, používat označení mraveniště a potrava. Na obrázku 3, převzatém z anglického originálu je naopak užito anglické názvosloví.



Obr. 3 Nalezení cesty s minimálními náklady

Při hledání minimální cesty pomocí umělých mravenců, na grafové struktuře, prostým přenesením chování mravenců skutečných, narážíme na určité problémy. Jedním z nich je, že mravenci mohou při vytváření řešení generovat smyčky. V důsledku aktualizace feromonu, se stávají takovéto smyčky stále více a více atraktivní a mravenci v nich často uvíznou. Ale i pokud z nich nakonec některý z nich vyvázne, feromonová stopa na cestách je již natolik ovlivněna, že nejkratší cesty již většinou nejsou voleny a celý mechanizmus přestává fungovat.(Dorigo, 2004)

Proto musíme daný mechanizmus chování umělých mravenců rozšířit o principy, které při zachování výhod metody, umožní efektivně řešit hledání minimální cesty v obecných grafech. Tyto inovace se zaměřují v prvé řadě na metody aktualizace a vypařování feromonu.

Vypařování feromonu lze inovovat následujícím způsobem běžný mravenec za sebou zanechává feromonovou stopu neustále. U umělého mravence je výhodnější postup, kdy po dosažení cílového uzlu, před zpáteční cestou, zanalyzuje svoji dosavadní cestu. Z té se eliminují smyčky. Eliminace může být prováděna postupem, kdy procházíme seznam navštívených uzlů, pokud narazíme na některý uzel podruhé, vymažeme část cesty zaznamenané mezi těmito dvěma výskyty. K aktualizaci feromonové stopy dochází, až při zpáteční cestě zpět do počátečního uzlu, po cestě osvobozené od smyček. Zásadním aspektem je při aktualizaci také množství feromonu, o které bude nalezená cesta obohacena. V základním případě, pouze rozdílná délka cest ovlivňuje tuto hladinu. Jinými slovy, hodnota, o kterou se feromon zvyšuje je u všech mravenců vždy stejná, ale mravenci, kteří nalezli kratší cestu, tuto cestu aktualizují dříve a častěji, tudíž na ni roste feromon rychleji. Další možnou metodou je možnost, kdy je množství o které se feromon zvyšovat, určeno pomocí funkce, jejímž vstupním parametrem je délka dosažené cesty. Čím kratší bude cesta, tím bude hodnota přidávaného feromonu větší.

Odpařování feromonu, lze chápat jako mechanismus, který pomáhá zabránit rychlé konvergenci všech mravenců směrem k suboptimálnímu řešení. Odpařování tímto napomáhá prozkoumávání různých cest v průběhu celého vyhledávacího procesu. Odpařování feromonu hraje u umělých mravenců ještě větší roly, než je tomu u těch skutečných. Je to dáno pravděpodobně tím, že umělý mravenci většinou řeší optimalizace podstatně složitější. Mechanismus jako je odpařování způsobuje, něco co, bychom mohli nazvat zapomínáním chyb, či špatných voleb, provedených v minulosti. Dále odpařování napomáhá omezovat maximální množství feromonu nacházející se na jednotlivých částech cesty, jelikož čím více feromonu se na stezce nachází, tím více se jej odpaří.(Dorigo, 2004)

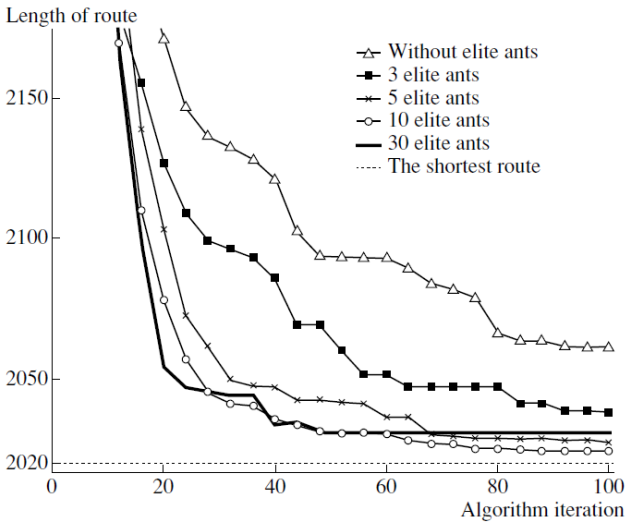
Toto lepší pochopení mechanismu a úskalí, které provázejí uměle vytvořené mravence, nám dále umožňuje navrhovat různá vylepšení a modifikace ACO.

2.1.4 Modifikace ACO

V průběhu let vzniklo několik aplikovaných řešení z rodiny ACO.

Elitist ant system

Jednalo se první modifikaci vzniklou ze základního algoritmu. V rámci této metody dochází k posílení doposud nejlepších řešení. (Krömer, 2012) K aktualizaci feromonu na hranách, které patří do nejlepšího nalezené cesty, dochází při každé iteraci. Potvrdilo se, že tato modifikace, v případě vhodně zvolených parametrů, vede k lepším výsledkům a způsobuje rychlejší konvergenci algoritmu.(Krömer, 2012) Příkladem správné implementace a zlepšení získaného použitím elitářského mravenčího systému mohou být výsledky dosažené S. D. Shtovbou z Vinnitské státní univerzity. Výsledky jeho pokusů vidíme na obrázku 4. Je zde také vidět, že přílišný počet elitních mravenců může řešení naopak zhoršovat.



Obr. 4 Zlepšení řešení za použití Elitist ant system (Shtovba, 2005)

Rank-based ant system
Dalším algoritmem modifikujícím výchozí ACO, je metoda, při které jsou mravenci seřazení podle kvality dosažené cesty a množství o jaké budou konkrétní mravenci zvyšovat množství feromonu na hranách je odvozeno od jejich pořadí. Počet nejlepších mravenců, kteří budou aktualizovat cestu, lze dále omezit parametrem. (Krömer, 2012)

Ant colony system
Systém mravenčí kolonie obohacuje algoritmus o elitismus a další modifikace, týkající se výběru přechodové hrany. Pro řízení pohybu mravenců se používá pseudonáhodné proporční pravidlo. Pouze nejlepší mravenec v iteraci v této metodě může ukládat feromon, vypařováním se ovšem upravují pouze feromony na nejlepší nalezené cestě. (Krömer, 2012)

Max-Min system
Jak již název naznačuje, dochází zde k omezování minimální a maximální hodnoty, a to hodnoty množství feromonu na hranách. Tím se zamezuje uváznutí v lokálním extrému. Opět pouze nejlepší mravenec z iterace může aktualizovat feromonovou stopu. (Krömer, 2012)

Algoritmus ant-Q
U tohoto algoritmu došlo k inspiraci Q-učením a feromonové hodnoty jsou nahrazeny AQ-hodnotam. Při každém přesunu mravence dochází k okamžité úpravě množství uložených AQ-hodnot. (Krömer, 2012)

Antabu
Algoritmus aplikuje tabu vyhledávání. Každý navštívený uzel je uložen do tabu seznamu a dále jsou navštěvovány pouze uzly, které v tomto seznamu nejsou. (Krömer, 2012)

Algoritmus ANTS
Název ANTS je zkratkou od anglických slov Approximated Non-deterministic Tree Search. Od výchozího algoritmu se odlišuje způsobem výpočtu přírůstku feromonu. Tato změna je motivována snížením rizika uvíznutí v lokálním optimu. (Krömer, 2012)

Fast Ant Systém
Rychlý mravenčí algoritmus vyvinutý pro řešení jedné konkrétní úlohy, a to kvadratického přiřazovacího problému (angl. Quadratic Assignment Problem). (Krömer, 2012)

Každá z těchto modifikací má své výhody i nevýhody a nelze ji slepě aplikovat. Vždy je tak třeba danou implementaci vztahovat, ke konkrétnímu problému.

Například M. Dorigo a T. Stützle, prováděli experimenty, z jejich výsledků vyplývá, že při řešení problému obchodního cestujícího si nejlépe vedla metoda Min-Max, která však v počátečních fázích řešení, dosahuje horších průběžných výsledků než jiné metody. (Krömer, 2012)

Z tohoto vyplývá, že nelze nikdy navrhnout zcela univerzální implementaci. Implementace, která bude jedním z výstupů této práce, tak také musí výt vytvářena s výhledem na použití u určitého typu optimalizačního problému.

2.1.5 Možnosti užití

ACO a další jemu podobné algoritmy se převážně používají pro řešení NP-těžkých úloh. K jejich podrobnějšímu popisu a celkové problematice složitosti řešených problémů si řekneme více v následující kapitole.

Většinu těchto úloh lze zařadit do některé z následujících oblastí:

- Směrování – jedná se o problémy, ve kterých jeden nebo více agentů, musí navštívit předdefinovanou sadu míst a jejichž kvalita řešení závisí na pořadí, ve kterém je navštíví.
- Přiřazování – úkolem je přiřadit sadu položek (objekty, aktivity atd.) na určitý počet zdrojů (lokace, osoby atd.), podle určitých omezení.
- Plánování – plánování se týká přidělování omezených zdrojů na úkoly v čase. Plánovací problémy jsou klíčové pro výrobu a zpracovatelský průmysl.
- Množinové problémy – uvažované řešení těchto problémů je reprezentováno jako podmnožina množiny dostupných položek, podléhající specifickým omezením. (Dorigo, 2004)

Jako důvody, proč si jako problém, jež budeme řešit pomocí ACO, vybrat právě problém obchodního cestujícího, uvádí například S. D. Shtovba, následující:

1. Problém může být vhodně dán do souvislosti s chováním mravenců v prostředí. Intuitivně pohyby obchodního cestujícího jsou podobné pohybům mravenců.
2. Jedná se o NP-úlohu.
3. Jedná se o tradiční problém, který se používá jako benchmark, pro optimalizační algoritmy.
4. Jde o vhodný didaktický problém, u něhož proces vyhledávání řešení je možné vysvětlit, i bez popisování technických podrobností algoritmu.
5. Jedná se o první problém, řešený pomocí ACO. (Shtovba, 2005)

2.2 Složitost

Pojem složitosti, kterým se v této kapitole budeme zabývat, je blízký jeho významu v běžném jazyce. V běžném životě se s tímto pojmem setkáváme často. Říkáme, že politická situace je složitá, neúspěchy ospravedlňujeme složitými podmínkami atd. Naopak jednoduchost je považována za pozitivní aspekt. Zde říkáme, že krása tkví v jednoduchosti, geniální myšlenky jsou vždy jednoduché apod. (Pudlák, 1988)

Postihnout, či kvantifikovat, takovéto situace z reálného života není vždy možné, avšak u problémů řešených počítačem jsou naše vyhlídky lepší. A právě zkoumání složitosti výpočtů vykonávaných na počítačích byl jeden z hlavních aspektů ovlivňující vývoj teorie složitosti.

Otázky týkající se spotřebovávaných prostředků (strojový čas, velikost paměti atd.) nutných k řešení úlohy, jsou převažující motivací pro zkoumání složitosti.

Při hodnocení složitosti úlohy, nemůžeme vycházet z toho, jak rychle jsou řešeny současnými počítači, jelikož technologie se rychle mění. Místo toho se nabízí jiný přístup. Většina řešených problémů má atributy, které můžeme označit jako vstupní data.(Pudlák, 1988) Velikost vstupních dat je počet bitů, které je potřeba k zápisu vstupu do počítače, většinou však velikost vstupních dat počítáme hruběji, v jednotkách odpovídajících konkrétnímu zadání. (Černý, 2013) U grafových úloh se bude jednat například o počet uzlů či hran v grafu. Složitost takovéto úlohy poté nelze vyjádřit jedním číslem, ale bude se jednat o posloupnost přirozených čísel v závislosti na vstupních parametrech. Otázku klasifikace jednoduchých a složitých úloh tak převádíme na otázku klasifikace funkcí. Jednoduché odpovídají pomalu rostoucím funkcím, složité rychle rostoucím funkcím. (Pudlák, 1988)

2.2.1 Časová složitost algoritmu

Časová složitost algoritmu je počet kroků, které budou při vykonávání algoritmu provedeny. Jde tedy o funkci T : N→N, kde T(n) je maximální počet kroků, které budou vykonány na datech o velikost n.

Krokem algoritmu, v tomto případě je jedna operace, daného stroje, zjednodušeně za operaci lze považovat libovolnou operaci provedenou v konstantním čase. (Černý, 2013)

V tabulce 1, vidíme jaký počet operací, bude třeba vykonat u algoritmů s různou časovou složitostí. Parametr n určuje velikost vstupních dat.

	n=10	**n=100**	**n=100**	**n=1000000**
$\log n$	3,3	6,7	10	20
$\sqrt{n}$	3,2	10	31,6	1000
n	10	100	1000	1000000
$n \log n$	33	664	9966	20×10^{6}
n^2	100	10^{4}	10^{6}	10^{12}
n^3	1000	10^{6}	10^{9}	10^{18}
2^n	1024	13×10^{30}	11×10^{302}	≈∞
$n!$	36×10^{5}	93×10^{157}	40×10^{2567}	≈∞

Tab. 1 Počet operací v závislosti na velikosti vstupu a časové složitosti (Černý, 2013)

Běžný dnešní počítač zvládne spočítat 10^{9} operací za sekundu. (Černý, 2013) Tabulka níže ukazuje, jaký časový úsek by trval výpočet algoritmů s danou časovou složitostí na běžném počítači.

	n=10	**n=100**	**n=100**	**n=1000000**
$\log n$	3,3 ns	6,7 ns	10 ns	20 ns
$\sqrt{n}$	3,2 ns	10 ns	31,6 ns	1000μs
n	10 ns	100 ns	1 μs	1 ms
$n \log n$	33 ns	664 ns	9966	20 ms
n^2	100 ns	10 μs	1 ms	16,5 min
n^3	1 μs	1 ms	1 s	31 let
2^n	1 μs	3×10^{14} let	3×10^{286} let	≈∞
$n!$	3 ms	3×10^{142} let	≈∞	≈∞

Tab. 2 Doba výpočtu v závislosti na velikosti vstupu a časové složitosti (Černý, 2013)

Z tabulky můžeme jasně vidět rozdíl mezi úlohami s polynomiální časovou složitostí a složitostí exponenciální či faktoriální. Dostáváme se do situace, kdy rozdíl nespočívá pouze v tom, jestli výsledek dostaneme v okamžiku, nebo na něj budeme muset chvíli čekat. Rozdíl může spočívat i v tom, jestli se výsledku vůbec dožijeme.

Při určování časové složitosti algoritmu, se zavádí také pojem asymptotická časová složitost. Například u funkcí n^2, $5n^2$ a $50n^2$ bude křivka růstu v závislosti na velikosti vstupních dat mít obdobný tvar, který je jasně oddělí od funkcí lineárních či kubických. Po zavedení asymptotické složitosti, můžeme konstanty považovat za nepodstatné a všechny funkce se stejnou složitostí zařadit do jedné třídy. Při reálném použití je však následně pochopitelně rozdílné, pokud na výsledek budeme čekat 5 nebo 50 minut.(Černý, 2013)

Obdobným způsobem lze zkoumat i prostorovou složitost, kdy se zabýváme použitou pamětí. Obě kritéria (čas, prostor) jsou obvykle proti sobě, proto si programátor musí zvolit, čemu dá přednost. V dnešní době se obvykle upřednostňuje zkrácení času. (Rybička, 2014)

2.2.2 Složitostní třída P a NP

Odbornou terminologii, tedy dělíme složitosti do dvou tříd, a to podle toho, jestli jsou řešitelné v polynomiálním čase deterministickým, či nedeterministickým Turingovým strojem:

- Složitostní třída P - jedná se o třídu problémů, které jsou řešitelné deterministickým Turingovým strojem v polynomiálním čase.
- Třídu složitosti NP - (nedeterministicky polynomiální) definujeme jako množinu problémů, které lze řešit v polynomiálně omezeném čase na nedeterministickém Turingově stroji. Nejtěžší z problémů ve třídě NP mají společnou vlastnost, že je umíme řešit, ale není znám algoritmus s polynomiální časovou složitostí (avšak není vyloučeno, že existuje). (Foltýnek, 2015)

2.3 Problém obchodního cestujícího

V této kapitole si podrobněji popíšeme problém obchodního cestujícího, v angličtině nazývaný Traveling salesman problem. Právě z anglického názvu vzniklou a hojně používanou zkratku TSP, budeme dále používat. Jak bylo řečeno v minulých kapitolách, jedná se o NP-těžký problém, na který se dá úspěšně aplikovat a testovat ACO, proto se jím budeme více zabývat.

TSP je známý kombinatorický problém. Jeho historie sahá až do roku 1759, kdy se úlohou tohoto typu zabýval švýcarský matematik Leonhard Euler. Zájem o tento problém ještě vzrostl s příchodem lineárního a celočíselného programování v druhé polovině 20. století. Nejedná se však pouze o teoretický problém, má totiž veliké využití v množství praktických aplikací v logistice, plánování, výrobě VLSI obvodů, krystalografii a mnoha dalších oborech. (Hynek, 2008)

Jelikož se tedy jedná o NP-těžký problém, je nepravděpodobné, že by se objevil deterministický algoritmus, jenž by umožňoval nalézt optimální řešení v polynomiálně omezeném čase. To vedlo v uplynulých desetiletích, k návrhu mnoha aproximačních algoritmů.

2.3.1 Definice

Při řešení TSP se snažíme najít optimální trasu pomyslného obchodníka, který při svých cestách potřebuje navštívit několik měst. Každé z těchto měst, s výjimkou počátečního chce navštívit právě jednou. Cena jeho cesty by měla být minimální, tuto cestu typicky vyjadřujeme vzdáleností mezi jednotlivými městy.

Pohledem teorie grafů se jedná o snahu najít hamiltonovskou kružnici minimální délky. V případě symetrického grafu platí, že se vzdálenost z města A do měst B, rovná vzdálenosti z B do A. U asymetrické verze toto není podmínkou. (Krömer, 2012)

2.3.2 Tradiční algoritmy pro řešení

Úplný algoritmus, který zaručuje nalezení skutečně nejlepšího řešení, by pro n měst musel vyzkoušet všechny permutace množiny měst 1 až n. (Hynek, 2008) Jak můžeme vidět v tabulce, i u relativně malého množství měst jde o obrovské množství kombinací. Časová náročnost takovéhoto algoritmu jej tedy činí nepoužitelným.

Počet měst	Počet různých cest
2	1
3	1
4	3
5	12
6	60
7	360
8	2 520
9	20 160
10	181 440
20	6×10^{16}
30	4×10^{30}
40	1×10^{46}
50	3×10^{62}

Tab. 3 TSP počet různých cest v závislosti na počtu měst

V případě 2 měst existuje pouze jedna možnost cesty řešení TSP, obdobně pro 3 města. Pro vícero měst můžeme počet kombinací vypočítat pomocí následujícího vzorce.

$$počet\ různých\ cest = \frac{(počet\ měst - 1)!}{2}$$

Hladový algoritmus

Mezi nejjednodušší, ale velmi rychle algoritmy patří hladový algoritmus, který je založen na myšlence nalézání nejbližšího souseda k právě navštívenému městu. Algoritmus začne s náhodně vybraným městem a najde k němu nejbližší, dosud nenavštívené město. Stejným způsobem pokračuje dále, dokud nejsou navštívena všechna města. V tu chvíli je třeba nalézt cestu zpět do startovacího města.

Takto nalezené řešení se ovšem velmi často bude lišit od optima. Poměrně snadno lze nalézt případy, kdy volba krátkých úseků na začátku běhu algoritmu, je vykoupena nutností volby nevhodných cest na konci běhu. (Hynek, 2008)

2-opt algoritmus

Velmi často je používána jednoduchá heuristika, která je v literatuře označována jako 2-opt algoritmus. V tomto případě začínáme s náhodně vytvořenou permutací cest T a snažíme se ji dále vylepšovat. Ta tímto účelem se definuje okolí cesty T tak, že do této množiny patří všechny cesty, které se od T liší vzájemnou výměnou dvou přímo nenavazujících hran z cesty T.

Danou heuristiku lze dále rozšířit na k-opt, čímž je algoritmus schopen sofistikovanějších modifikací cesty, Na druhé straně rostoucí parametr k exponenciálně zvětšuje mohutnost okolí původní cesty a tím roste i čas potřebný ke zpracování této množiny. Experimenty bylo zjištěno, že tento algoritmus dosahuje velmi solidních výsledků z hlediska času výpočtu i nalezeného řešení. (Hynek, 2008)

Existují i další možnosti řešení TSP, dále jmenujme například genetické algoritmy či Christofidesův algoritmus. V dalších kapitolách práce se však již budeme zabývat konkrétním návrhem a implementací ACO pro řešení TSP.

3 Metodika práce

Ze zadání práce vyplývá, že výstupem má být efektivní implementace algoritmu ACO, doplněná o vhodné uživatelské rozhraní. Toto rozhraní by uživateli mělo umožnit, srozumitelným způsobem nastavit parametry běhu algoritmu, zobrazit finální výstup, ale také uživatele průběžně informovat o průběhu výpočtu a pomoci mu pochopit jednotlivé kroky algoritmu.

Požadavek maximální efektivnosti na jedné straně, a jisté míry edukativnosti a přehlednosti na druhé straně, však jdou proti sobě. Z tohoto důvodu budou na vytvořené kmenové implementaci ACO, vystavěny 2 nezávislé aplikace, které lze stručně charakterizovat následovně:

1. Aplikace s grafickým rozhraním – aplikace s grafickým uživatelským rozhraním. Důraz na přehlednost, poutavost, možnost manuální zadání vstupů. Grafické zobrazení výstupu.
2. Konzolová aplikace – možnost dávkového zpracování dat, konfigurace vně programu. Maximální efektivita běhu programu. Výstup v textové podobě do textového souboru.

Jako vhodný programovací nástroj, poskytující platformu pro efektivní vývoj grafické aplikace i aplikace s důrazem na výkon byla vybrána platforma .NET, konkrétně objektově orientovaný programovací jazyk C#. Za hlavní nevýhodu dané platformy lze považovat omezení pro běh pouze na systémech platformy společnosti Microsoft.

S volbou programovací platformy dále souvisí volba vývojového prostředí. Bylo tedy vybráno vývojové prostředí Microsoft Visual Studio 2013, v nejvyšší edici Ultimate. Dané prostředí poskytuje nástroje nejen pro psaní a kompilaci daného programového kódu, ale také nástroje pro tvorbu potřebných diagramů jazyka UML. Zejména podpora generování programového kódu na základě diagramu tříd je na vysoké úrovni, a proto jí bude využito. Pro tvorbu ostatních diagramů však bude použit nástroj Visual Paradigm 12.0, určený primárně pro tvorbu UML diagramů, a tudíž jehož možnosti jsou v mnoha směrech na vyšší úrovni.

Pořadí a obsah kroků, které budou provedeny pří vývoji aplikace je následující:

1. Sumarizace uživatelských požadavků na aplikaci – souhrn vstupů, očekávaných výstupů, parametrů běh a dalších záležitostí, které implementaci umožní plnit uživatelem očekávané cíle.
2. Návrh implementace a její provedení – návrh funkčních modulů aplikace, objektových tříd a dalších programových elementů. Jejich implementace pomocí zvolené platformy.

3. Ověření funkčnosti řešení – výsledky, které implementace bude poskytovat, musejí být ověřeny z hlediska jejich správnosti, pohybujeme-li se však v oblasti NP problémů, budeme volit především syntetická data (rozmístění uzlů do pravidelné mřížky, do dvojité kružnice, atd.)
4. Optimalizace implementace pro maximální výkon – budu-li dostatečně prokázáno, že dosahované výsledky splňují očekávání, oddělí se vývojová větev dávkové aplikace a výchozí algoritmus bude optimalizován pro maximální rychlost poskytování očekávaných výsledků.
5. Sumarizace a porovnání dosažných výsledků – v případech, kde je toto možné, budou nároky na čas a prostor výsledné implementace porovnány s dostupnými implementace ACO.

Při všech krocích uvedených výše budou uplatňovány poznatky získané při zpracování předchozích kapitol.

4 Vlastní práce

4.1 Analýza a specifikace požadavků

Z poznatků uvedených v předchozí kapitole, vyplývá, že řešením je vytvoření 2 aplikací, které se budou lišit svými vlastnostmi, zejména co se ovládání uživatelem týče, ve výsledku však budou řešit stejný problém, jejich vstupy a výstupy budou stejné, lišit se budou pouze uživatelským rozhraním, efektivitou nalezení řešení a způsobem předání výsledku.

Jádro aplikace se tedy dá vyjádřit následovně. Aplikace bude řešit pomocí algoritmu ACO problém nalezení co nejkratší možné hamiltonovské kružnice v úplném ohodnoceném grafu.

Z poznatků, které byly získány v teoretické části, se jeví jako nejvhodnější implementace ACO kombinující MIN-MAX Ant systém, spolu s Ranked Ant Systemem.

Uživatel bude mít na výběr, zda uzly grafu definuje sám, anebo je nechá vygenerovat aplikaci. Zadávání uzlů i řešení celého problému bude prováděno v dvourozměrném prostoru, kdy každý uzel bude mít nastaveny souřadnice x a y. V případě generování uzlů aplikací půjde o náhodné rozmístění určeného počtu uzlů v ohraničeném prostoru, případně o umístění uzlů do předem definovaného obrazce, který uživateli umožňuje jednoduše ověřit správnost řešení, tedy například rozmístění uzlů do kruhu, mřížky apod. Po vytvoření uzlů již aplikace sama doplní do grafu hrany, tak aby se jednalo o úplný graf. Ohodnocení jednotlivých hran bude vyjádřeno jako vzdálenost mezi uzly, jež hrana spojuje. Pro možnost srovnávání dosažených výsledků v jednotlivých běžích programu, je potřeba, aby uživatel měl možnost importovat a exportovat, vytvořené zadání.

Uživatel bude mít možnost algoritmus parametrizovat, aby mohl dosahovat, co nejuspokojivějších výsledku. Jelikož jde o implementaci algoritmu ACO, parametry, které bude možné editovat, jsou následující:

- Alfa.
- Beta.
- Rychlost vypařování feromonu.
- Minimum feromonu na hraně.
- Způsob zvyšování feromonu.
- Maximum feromonu na hraně.
- Počet elitních mravenců.

Jak již bylo zmíněno v předchozích kapitolách, cílem by mělo být nalézt optimum, avšak při složitějších zadáních neexistuje způsob, který by nám umožňoval jednoduše zjistit, zda jsme ho již dosáhli, nebo další běh aplikace dokáže vytvořit lepší řešení. Z tohoto důvodu je třeba určit ukončovací podmínky běhu aplikace. Jako možnosti, které by nebyli zcela vhodné, se jeví například ukončení po uplynutí určitého časového limitu, což by však znamenalo, že počet iterací, které algoritmus vykoná, by byl přímo závislý na výkonu stroje a výsledky by byli vzájemně

neporovnatelné, anebo ukončení po dosáhnutí řešení, které je lepší nebo rovna jako pevně daná hodnota, což by však mohlo vyústit v nekonečnou smyčku. Jako vyhovující ukončující podmínky se jeví následovné:

- Po pevně daném počtu iterací.
- Pokud určitý počet iterací nedojde ke zlepšení výsledku.

Výsledek, který bude výstupem aplikace, uživatele bude informovat o délce nejlepší dosažené cesty, a ve které iteraci řešení byla tato cesta dosažena. Výsledná hamiltonovská kružnice bude vyjádřena jako posloupnost postupně navštívených uzlů.

V implementaci, která bude obsahovat i grafické uživatelské rozhraní, bude kladen důraz na uživatelsky přívětivé rozhraní, které umožní uživateli jednoduše editovat zadávání grafu a parametry běhu programu. Graf v této aplikaci bude graficky zobrazen a v průběhu řešení úlohy se bude vykreslovat nejlepší dosažené řešení. Aby měl uživatel možnost jednotlivé kroky algoritmu lépe pochopit a aplikace byla vhodná i k edukaci, musí mít uživatel možnost běh algoritmu pozastavit a mít možnost zobrazit aktuální stav hodnot v grafu, tedy zejména, hodnotu feromonů na hranách a s tím související pravděpodobnost vybrání hrany, informace o jednotlivých uzlech a postup konstrukce řešení jednotlivými mravenci. Možná je i ukázka aplikace ACO pro hledání nejkratší cesty v grafu, která lépe ukazuje prapůvodní myšlenku algoritmu a inspiraci u reálných mravenců.

4.1.1 Funkční požadavky

Funkční požadavky lze shrnout následovně:

1. Nalezení nejkratší možné hamiltonovské kružnice.
2. Import a export grafu.
3. Grafické rozhraní umožňující parametrizaci algoritmu a zobrazení výsledků.

4.1.2 Nefunkční požadavky

Nefunkční požadavky jsou následující:

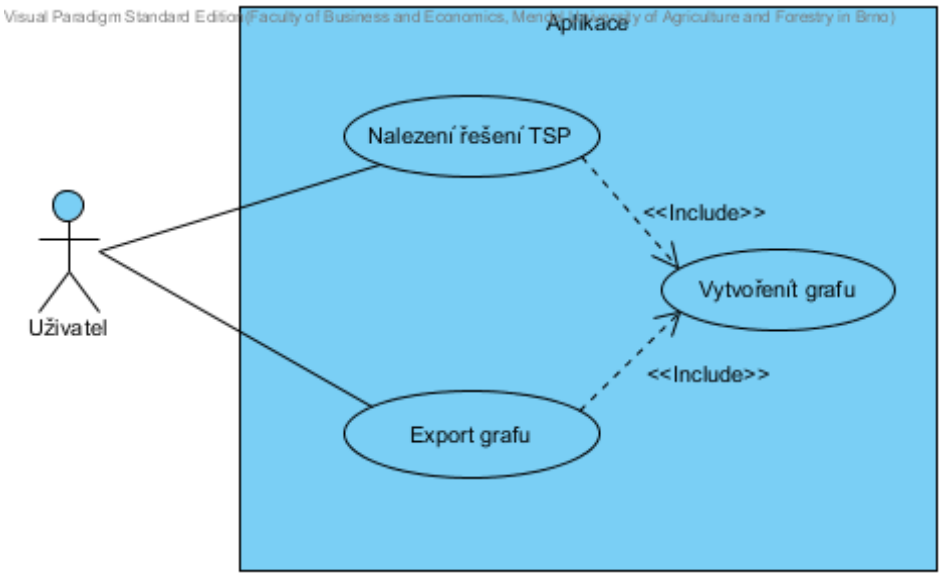
1. Běh aplikace na běžném PC s operačním systémem Windows, v rozumném čase.
2. Optimalizace pro více jádrové procesory.
3. Přiměřené hardwarové nároky.

4.1.3 Use case diagramy

Use case diagramy popisují systémové prostředí z pohledu uživatele. Pro vývojáře se jedná o hodnotný nástroj. Jde o vhodnou a vyzkoušenou techniku určenou ke sběru požadavků z pohledu uživatele. Získání těchto informací je nedílným krokem v cestě k softwaru, který budou skuteční uživatelé úspěšně využívat. (Schmuller, 2004)

Dále budou popsány základní tři scénáře, které bude uživatel nejčastěji vykonávat. Diagram vidíme na obrázku 5, v diagramu je použita relace <<Include>>, která není tak často využívána, proto bude v následujícím odstavci vysvětlena.

Relace <<Include>> nastavená mezi případy užití, umožňuje zahrnout chování dodavatelského případu užití do scénáře klientského případu užití. V této souvislosti říkáme, že klientský případ užití je zahrnujícím případem užití, zatímco zahrnovaný případ užití je označován jako dodavatelský případ užití. Důvodem je fakt, že zahrnovaný případ užití dodává chování klientskému případu užití. (Arlow, 2007)



Obr. 5 Use case diagram

Use case Vytvoření grafu
Scénář užití popisuje vytvoření vstupního grafu pro běh algoritmu.

Název	Vytvoření grafu.
Aktéři	Uživatel, aplikace.
Podmínky zahájení	Aplikace se nachází ve výchozím stavu pro zadání úlohy TSP.
Základní tok	**1.** Uživatel zadá nový graf. **2.** Aplikace převede graf do vlastní reprezentace.
Alternativní tok	**1.1**. Uživatel zadá zdroj, ze kterého se graf načte.
Výstupní podmínka	Aplikace obsahuje reprezentaci grafu.

Tab. 4 Scénář Use case - Vytvoření grafu

Use case Nalezení řešení TSP.
Tento případ užití popisuje postup nalezení nejkratší hamiltonovské kružnice v grafu.

Název	Nalezení řešení TSP.
Aktéři	Uživatel, aplikace.
Podmínky zahájení	V aplikaci je vytvořen graf.
Základní tok	**1.** Zahrnout (Vytvoření grafu) **2.** Uživatel nastaví parametry běhu algoritmu. **3.** Uživatel spustí algoritmus. **4.** Aplikace nalezne optimální řešení. **5.** Aplikace předá nalezené řešení.
Výstupní podmínka	Nalezena vhodná hamiltonovská kružnice.

Tab. 5 Scénář Use case - Nalezení řešení TSP

Use case Export grafu
Scénář užití popisuje export vytvořeného grafu pro opakované užití.

Název	Export grafu.
Aktéři	Uživatel, aplikace.
Podmínky zahájení	V aplikaci je vytvořen graf.
Základní tok	**1.** Zahrnout (Vytvoření grafu) **2.** Uživatel zadá umístění pro export grafu. **3.** Aplikace převede graf do definované znovupoužitelné formy. **4.** Aplikace uloží graf.
Alternativní tok	**4.1.** Pokud v lokaci pro uložení již graf existuje, bude původní reprezentace smazána.
Výstupní podmínka	Existuje trvalá reprezentace grafu, která je opakovaně použitelná.

Tab. 6 Scénář Use case - Export grafu

4.1.4 Vstupy

Vstupem aplikace je seznam uzlů grafu, tyto uzly jsou zadávaný uživatelem buď ručně pomocí grafického rozhraní, nebo jsou generovány samotnou aplikací, anebo jsou importovány z textového souboru, kde každý řádek značí jeden uzel grafu. V posledním případě je formát řádku v souboru následovný:

```
N souřadniceX souřadniceY
```

Označení písmenem N, v tomto případě značí první písmeno z anglického slova node, v překladu znamenající uzel. Po mezeře následují souřadnice uzlu také odděleny mezerou.

Další možností vstupu je zobrazení řešení vytvořeného předchozím během aplikace, v takovém případě bude řešení graficky zobrazeno. Formát řádku takovéhoto textového souboru s řešením je zobrazen níže. Počáteční písmena NR jsou v tomto případě inspirovaná anglickými slovy node result. Pořadí v jakém řádky následují, určuji pořadí, v jakém budou uzly propojeny do hamiltonovské kružnice.

```
NR souřadniceX souřadniceY
```

4.1.5 Výstupy

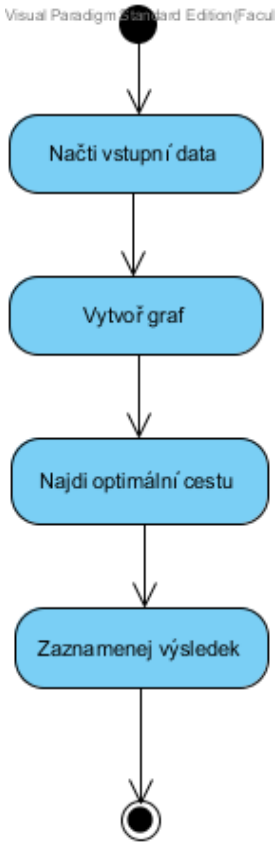
Výstupem je nejkratší zjištěná hamiltonoská kružnice dosažená v zadaném grafu. Ta je graficky zobrazena, případně vyexportována do textového souboru, ve kterém posloupnost řádku určuje postupně navštívené uzly. Řádek začíná písmeny NR a následují souřadnice uzlu. Více je tento zápis popsán v předchozím odstavci.

4.2 Diagramy aktivit

Diagramy aktivit jsou vývojové diagramy, díky nimž lze procesy modelovat jako aktivitu, která se skládá s kolekce uzlů spojených hranami. Diagramy aktivit lze připojit k libovolnému modelovanému prvku, umožňuje nám to modelovat jeho chování. S velkými výhodami, lze tyto diagramy využit rovněž k modelování obchodních procesů a pracovních postupů.(Arlow, 2007)

Níže jsou uvedeny jednotlivé diagramy aktivit, popisující návaznost aktivit vykonaných implementovaným algoritmem.

4.2.1 Základní diagram



Obr. 6 Diagram aktivit - Základní diagram

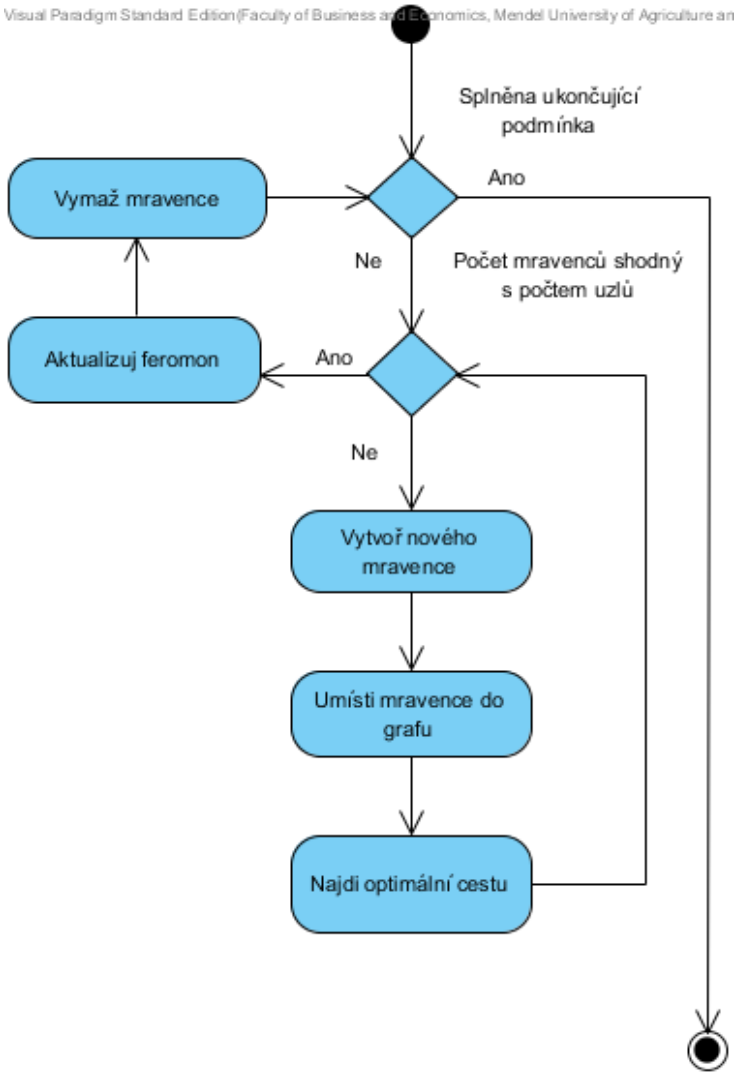
Základní diagram představuje základní posloupnost kroků, které budou prováděny. Výchozím stavem je situace, kdy má uživatel spuštěnou aplikaci. Jako první musí aplikace získat vstupní data, která jsou předána dále, a je z nich vytvořen graf, se kterým se bude dále pracovat.

V grafu poté algoritmus najde řešení daného problému a v poslední aktivitě je dosažený výsledek zaznamenám a předán uživateli.

Ukončujícím stavem je situace, kdy je algoritmus ukončen a připraven, k případnému dalšímu zpracování úlohy.

Klíčová aktivita nalezení optimální cesty, je dále dekomponována.

4.2.2 Najdi optimální cestu v grafu - aplikace

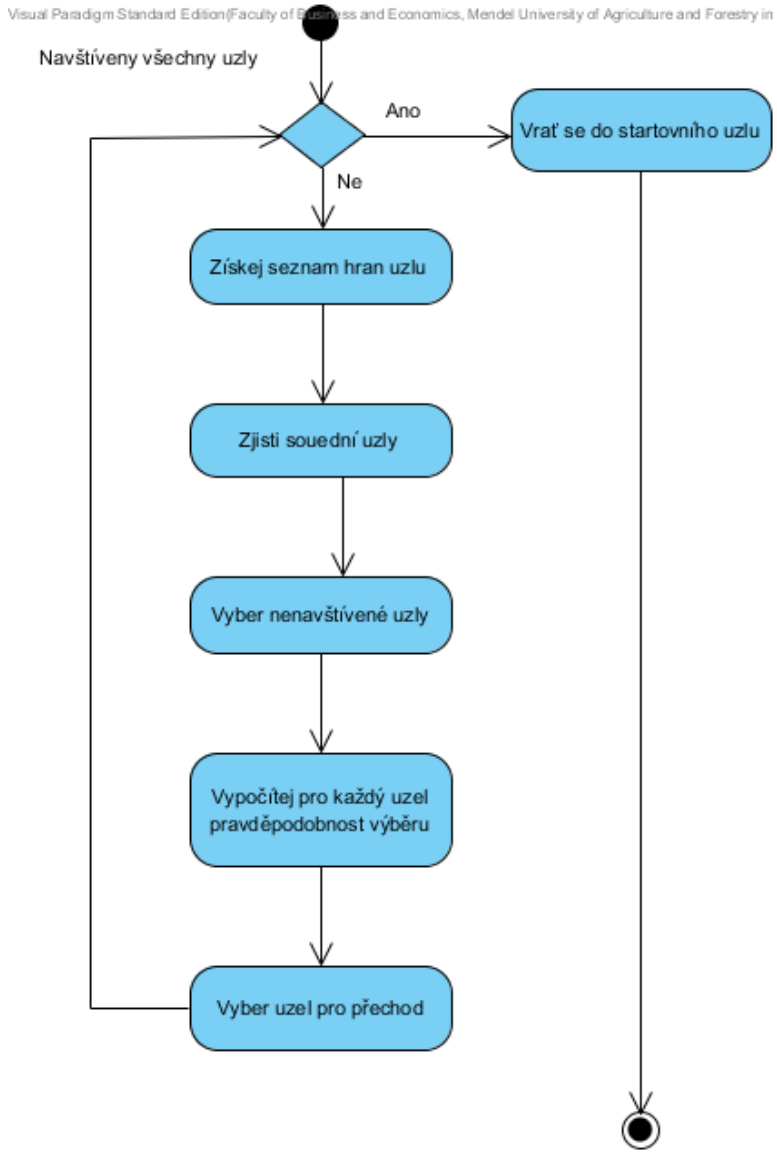


Obr. 7 Diagram aktivit – Najdi optimální cestu v grafu (aplikace)

Startovním stavem je situace, kdy je vytvořen graf a uživatel spustil algoritmus. Pokud není splněna podmínka pro ukončení algoritmu, pokračuje se k vytvoření mravenců, kteří budou hledat optimální cestu. Na začátku je jejich počet nulový. Vytvoří se mravenec, který se umístí do uzlu grafu, ze kterého doposud žádný z mravenců nestartoval. Mravenec poté hledá optimální cestu (dekomponováno v diagramu obr. 8). Následně se ověří, zda počet mravenců, odpovídá počtu uzlů, čili jestli z každého jednoho uzlu grafu startoval jeden mravenec. Pokud je podmínka splněna, je aktualizován feromon (dekomponováno v diagramu obr. 9). Mravenci jsou poté smazáni, a pokud není stále splněna ukončující podmínka,

probíhá daný proces znovu. V opačném případě se přechází do ukončovacího stavu. V tom máme graf s aktualizovanými feromonovými stopami a nejlepším dosaženým výsledkem.

4.2.3 Najdi optimální cestu - mravenec



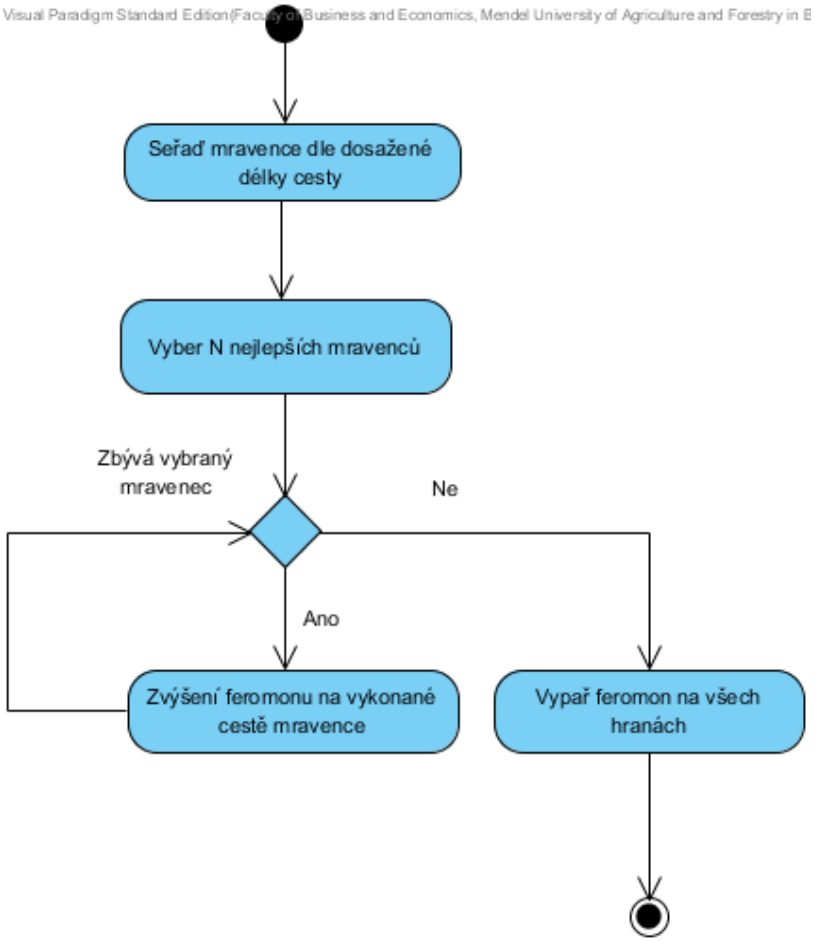
Obr. 8 Diagram aktivit – Najdi optimální cestu v grafu (mravenec)

Hledání optimální cesty mravencem probíhá následovně. Ve výchozím stavu je mravenec umístěn do startovacího uzlu. Pokud ještě nebyly navštíveny všechny uzly, získá seznam hran, které vedou z uzlu, ve kterém se nachází. Následně za pomoci těchto hran zjistí, jaké sousední uzly jsou dostupné a z nich vybere ty, které ještě nebyly navštíveny. Na základě vzdálenosti uzly a feromonové stopy na

hraně je pro každý nenavštívený uzel vypočítána funkce, určující s jakou pravděpodobností má být vybrán. Na základě takto vypočítaných hodnot je s určitou pravděpodobností vybrán některý z daných uzlů a do něj mravenec přechází.

Pokud stále nejsou navštíveny všechny uzly, postup se opakuje. V chvíli, kdy již mravenec navštívil všechny uzly, najde z uzlu, ve kterém se nachází cestu zpět do startovacího uzly a přechází se do ukončovacího stavu. V ukončovacím stavu tedy máme mravence, který navštívil všechny uzly a my známe délku jeho cesty a pořadí, ve kterém tak učinil.

4.2.4 Aktualizuj feromon



Obr. 9 Diagram aktivit – Aktualizuj feromon

Aktualizace feromonu je aktivita, ke které dochází ve chvíli, kdy máme stejný počet mravenců, kteří vyhledávali optimální cestu, jako je počet uzlů v grafu. Mravenci jsou následně seřazeni dle délky jejich cesty, od nejlepšího (nejkratší cesta) k nejhoršímu (nejdelší cesta). Dle hodnoty, kterou uživatel zadal při parametrizaci algoritmu, je následně vybrán daný počet nejlepších mravenců. U každého

z těchto mravenců jsou projity hrany, které navštívili, a je na nich daným způsobem zvýšena hodnota feromonu. Ve chvíli, kdy toto bylo provedeno u všech určených mravenců, je následně provedeno odpaření feromonu na všech hranách a aktualizace feromonu je tímto ukončena.

4.3 Diagramy tříd

Základním stavebním kamenem diagramu tříd jsou třídy a vztahy mezi nimi. Třídy představují uzly grafu a vztahy hrany mezi nimi. (Vrána, 2005)

Skupinu objektů, které vykazují podobné charakteristiky, můžeme vyjádřit jako třídu. Pro každou instanci dané třídy je možno specifikovat společné atributy a operace.

Mezi objekty mohou existovat vztahy, nejčastěji se setkáváme, se vztahem asociace, agregace a kompozice. Dalším hojně využívaným typem vztahu je vztah generalizace, jedná se o typ vztahu, kdy jedna třída je zobecněním vlastností jiné třídy, jedná se tedy o typ vztahu typu nadtřída/podtřída, respektive generalizace/specializace. (Vrána, 2005)

V následujícím obsahu kapitoly, budou ukázány vybrané diagramy tříd, představující návrh klíčových objektů navrhované aplikace.

4.3.1 Rozhraní

Rozhraní nám specifikuje, jak je objekt definovaný při pohledu zvenčí. Třída, která implementuje určité rozhraní, tedy musí mít veřejnou implementaci metod specifikovaných daným rozhraním. Vnitřní implementace třídy, však může být libovolná, ostatní metody, ať už veřejné, či neveřejné, anebo atributy třídy, nejsou v tomto případě jakkoli omezeny.

V praxi rozhraní programátorovi umožňuji v daném bodě programového kódu definovat odkaz na rozhraní a poté pracovat s jím definovanými metodami a to bez ohledu na jeho konkrétní implementaci, rozuměj konkrétní instanci třídy implementující rozhraní. Často udávaným příkladem rozhraní bývá situace, kdy máme skupinu tříd reprezentující jednotlivé geometrické útvary a u každého z nich chceme vypočítat jejich obsah. Každý geometrický útvar má rozdílné vlastnosti a pro výpočet jeho obsahu je potřeba použít jiné jeho vlastnosti, avšak výstupem je vždy číslo udávající obsah. V takovémto případě vytvoříme rozhraní, které bude definovat metodu vracející obsah objektu a jednotlivé třídy představující jednotlivé geometrické objekty jej budou implementovat.

Rozhraní také v rané fázi vývoje umožňuje týmu vývojářů rychleji definovat jednotlivé interakce mezi jednotlivými objekty, bez soustředění na konkrétní implementaci.

IAnt

Rozhraní IAnt, jehož diagram, spolu s třídami toto rozhraní implementujícími, můžeme vidět na obrázku dále, představuje vlastnosti, které očekáváme u každého mravence, ať už jeho konkrétní specializace bude libovolná. Každý mravenec, tedy musí být schopen, pokud se nenachází v cílovém uzlu, přesunout se do

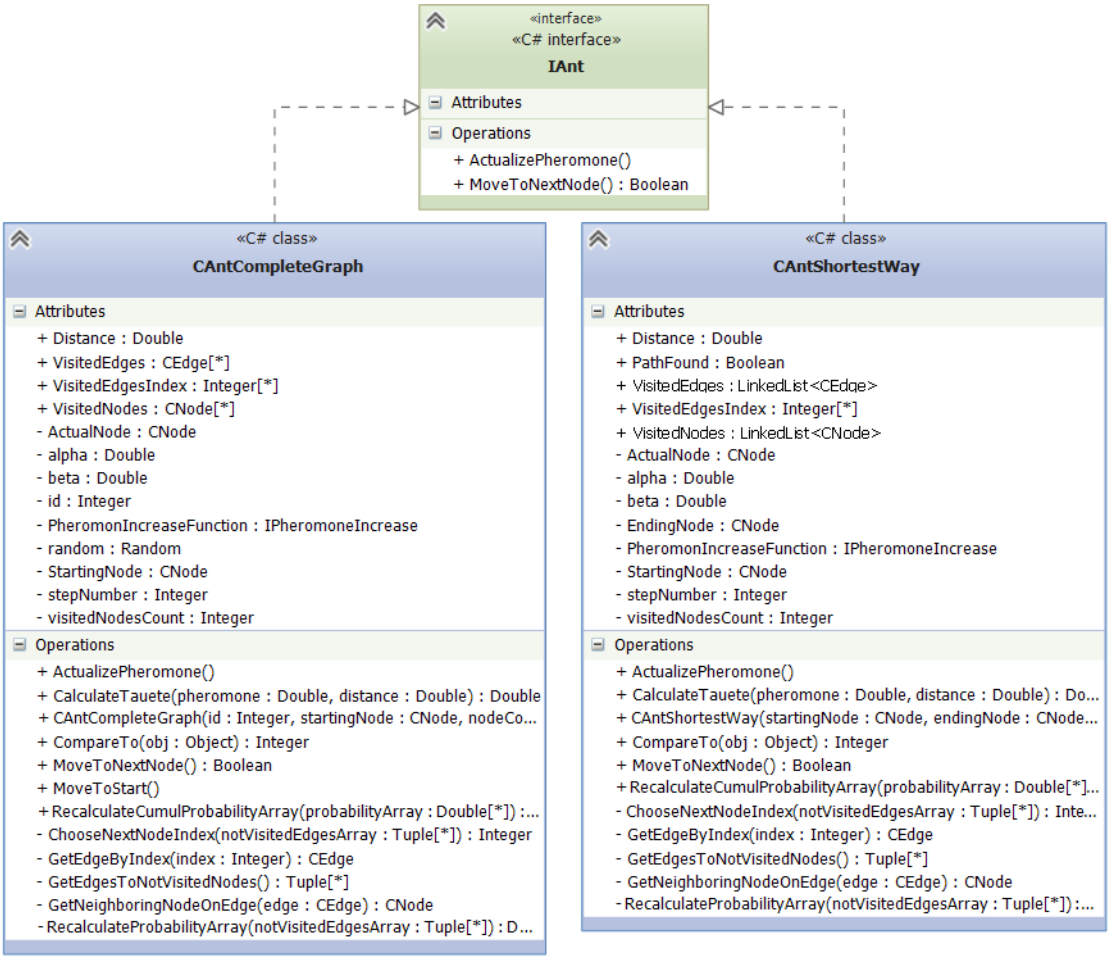
dalšího nenavštíveného uzlu, a dále v případě nalezení vhodné cesty aktualizovat feromon na jím vykonané cestě.

V našem případě budou vytvořeny 2 typy mravenců. První s názvem CAntCompleteGraph bude určen pro hledání hamiltonovské kružnice v úplném grafu. Druhý s názvem CAntShortestWay bude specializován na hledání nejkratší cesty v grafu, který nemusí být úplný. Tyto dvě třídy mají mnoho vlastností společný a ty budou implementovány obdobně, v jiných se však značně liší, což vyplývá z rozdílného řešeného problému, rozdíly jsou vyjádřeny v tabulce.

	CAntCompleteGraph	**CAntShortestWay**
Vždy nalezne řešení	Ano	Ne
Známe počet uzlů, které budou navštíveny při cestě	Ano	Ne
Začátek a konec ve stejném uzlu	Ano	Ne

Tab. 7 Rozdily mezi třídami implementujícími rozhrani IAnt

Vycházím z toho, že v grafu vždy vede cesta k cíli, avšak mravenec se může dostat do situace, kdy z uzlu nevede cesta do jiného nenavštíveného uzlu. V takovémto případě je cesta považována za zdegenerovanou, a práce mravence je ukončena. Toto řešení bylo zvoleno jako vhodnější oproti zpětnému očišťování cesty od smyček.



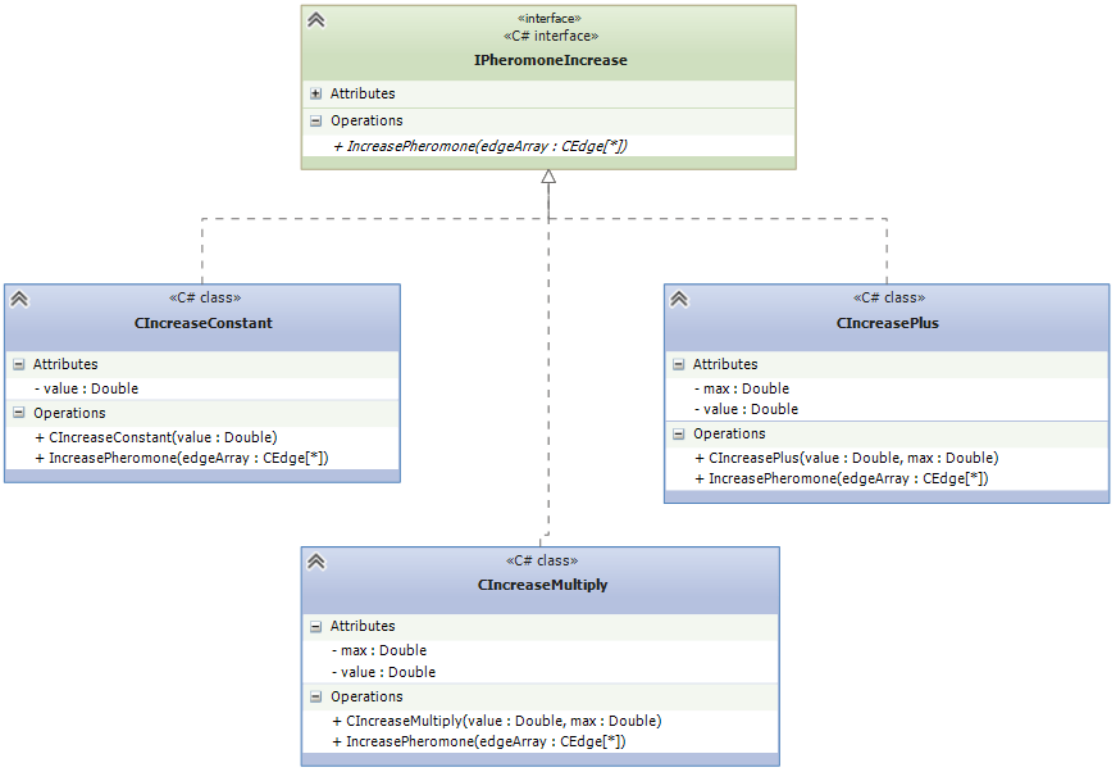
Obr. 10 Diagram tříd – rozhraní IAnt

IPheromoneIncrease

Rozhraní IPheromoneIncrease definuje způsob, jakým bude zvyšována úroveň feromonu na hranách. Rozhraní bude implementováno 3 třídami, které reprezentují jednotlivé způsoby zvýšení feromonu. Třída CIncreaseContant nastaví hranám konkrétní hodnotu, třída CIncreasePlus přičte hodnotě feromonu na hraně danou hodnotu a třída CIncreaseMultiply vynásobí hodnotu daným koeficientem.

Rozhraní bude využívat mravenec, kterému bude konkrétní implementace rozhraní předána při vytvoření v konstruktoru. Ten následně při splnění podmínek bude zvyšovat hodnotu feromonu na hranách jím vykonané cesty.

Diagram rozhraní IPheromoneIncrease je uveden na obrázku 11.



Obr. 11 Diagram tříd – rozhraní IPheromoneIncrease

IPheromoneDecrease

Interface IPheromoneDecrease je implementováno obdobným způsobem jako rozhraní předchozí. Opět jsou na výběr 3 implementace daného rozhraní a ty snižují hodnotu feromonu na hranách buď nastavením hodnoty, vynásobením hodnotou, anebo odečtením hodnoty.

Rozdíl však spočívá v tom, že v tomto případě budeme snižovat hodnotu feromonu v jeden čas na všech hranách grafu a rozhraní tedy nebude využívat instance mravence, ale nadřízená třída spravující běh algoritmu.

4.3.2 Enumerace

Enumerace je uživatelsky definovaný integerový typ. Deklarujeme-li enumeraci, specifikujeme akceptované hodnoty, které může instance dané enumerace obsahovat, a to navíc v uživatelsky přívětivém pojmenování. Pokud se kdekoliv v programovém kódu pokusíme přiřadit proměnné, která je datového typu určeného enumerací, hodnotu, která se v enumeraci nenacházím, nebude kód zkompilovatelný.

Hlavní výhody enumerací tedy jsou:

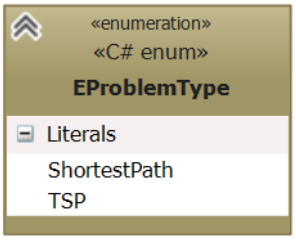
- Činní kód lépe udržovatelným, pomáhají programátorovi naplňovat proměnné pouze legitimními hodnotami.
- Integerové hodnoty jsou pojmenovány pomocí popisných názvů, vyhýbáme se vytváření nejasných číselníků.

- Pokud programátor napíše název enumerace, moderní vývojová prostředí mu usnadní práci automatickou nápovědou nabízející hodnoty enumerace, případně si i zapamatují poslední zvolenou hodnotu pro danou enumeraci a tu při nápovědě preferují. (Profesional c# 2008)

EProblemType

Tato enumerace určuje, jaký typ úlohy se bude řešit, jde o základní parametr vyplývající z požadavků na aplikaci, a ovlivňující jak samotný algoritmus výpočtu, tak také parametry, které bude třeba nastavovat v grafickém rozhraní.

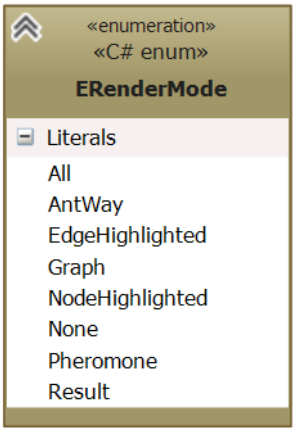
Enumerace může tedy nabývat dvou hodnot, TSP a ShortestPath (nejkratší cesta)



Obr. 12 Diagram tříd – Enumerace EProblemType

ERenderMode

Enumerace ERenderMode určuje, co vše má být při překreslení grafického panelu vykresleno. Každý z módů v sobě může skrývat vykreslení několika grafických prvků. Přehledně je vše popsáno v tabulce níže. V určitých případech je důležité i pořadí, ve kterém se jednotlivé prvky vykreslí, např. chceme napřed vykreslit všechny hrany a teprve poté jednotlivé uzly, v opačném případě by při větším počtu hran uzly vůbec nemuseli být vidět.



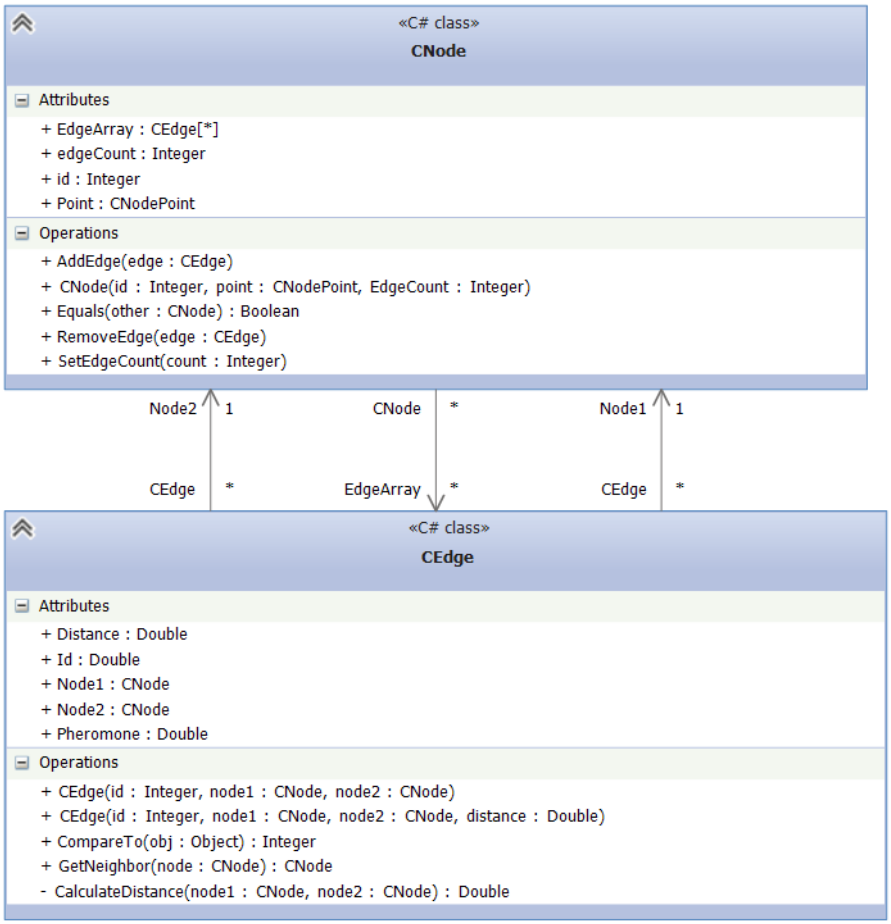
Obr. 13 Diagram tříd – Enumerace ERenderMode

4.3.3 Reprezentace grafu

Abychom mohli hledat řešení daných grafových úloh, musíme pochopitelně vytvořit třídy, které budou představovat graf. Graf je definován jako množina uzlů a hran, a proto vytvoříme třídy reprezentující tyto objekty.

Třída CNode představuje uzel grafu. Uzel musí obsahovat především souřadnice, k čemuž je využita třída Point, dále každý uzel má svůj unikátní číselný identifikátor a kolekci obsahující hrany spojené s uzlem.

Hrana grafu je reprezentována třídou CEdge. Hrana má také svůj unikátní číselný identifikátor, dále informaci o délce a hodnotu feromonu na hraně. Každá musí být napojena na dva uzly, to je vyjádřeno vlastností Node1 a Node2.



Obr. 14 Diagram tříd – třídy představující graf

Na diagramu v obrázku č. 6, vidíme u vztahů mezi třídami také násobnosti těchto vztahů. Třída CEdge tedy obsahuje právě dva odkazy na instance třídy CNode (po jednom odkazu ve dvou vlastnostech) a třída CNode obsahuje libovolný počet odkazů na instance třídy CEdge.

4.4 Návrh uživatelského rozhraní

Návrh uživatelského prostředí je klíčovým bodem k úspěšné aplikaci. Uživatelské prostředí tvoří most mezi uživatelem a samotnou funkční vrstvou aplikace, která vykonává podstatu aplikace.

V této kapitole bude rozebrána logika a návrh grafického uživatelského prostředí, které bude využito v aplikaci. U konzolové verze aplikace bude veškerá komunikace mezi uživatelem a aplikací realizována pomocí konfiguračního souboru, respektive vstupního a výstupního textového souboru.

4.4.1 Charakteristika uživatelů

Při návrhu jakéhokoli uživatelského rozhraní si musíme nejprve určit cílovou skupinu, pro kterou jej vyvíjíme. V tomto případě se jedná především o studenty, či jiné osoby pohybující se ve školním prostředí. Tuto množinu osob, bychom si mohli ještě dále rozdělit na skupinu osob, které aplikaci použijí osobně, pro své experimenty a poznávání algoritmu ACO a na skupinu zejména pedagogických pracovníků, kteří budou moci aplikaci použít pro demonstraci při výuce.

U těchto uživatelů, můžeme předpokládat alespoň základní znalost problematiky optimalizačních a grafových algoritmů, hledání nejkratší cesty a řešení problému TSP. Uživatel tedy bude znát, jaký problém chce řešit, či jaké výsledky předpokládá, od aplikace tedy bude očekávat především ukázku budování řešení a vliv jednotlivých parametrů na nalezení odpovídajícího výsledku.

4.4.2 Logická struktura

Logickou strukturu grafického rozhraní můžeme považovat za obdobu stavového diagramu. Jednotlivé stavy zde vyjadřují jednotlivá rozložení daných grafických prvků, které v určitou chvíli uživatel uvidí. V praxi se tak na jedné straně můžeme setkat s aplikací, jejíž grafické rozhraní má jen jeden stav. To při správném rozložení prvků může zlepšovat přehlednost aplikace, jelikož uživatel má všechny potřebné prvky stále na očích a v kontextu. To je však možné spíše při menším počtu prvků a nastavení. Na druhé straně jsou aplikace, kde se jednotlivé rozložení vždy mění tak, aby byly zobrazeny pouze prvky vztahující se k určitému právě řešenému kroku. Příkladem mohou být například průvodci instalací, se kterými se jistě každý uživatel setkal.

V tomto případě je vhodnější použít grafické rozhraní s více stavy. Rozhodující pro toto rozhodnutí je především množství parametrů, které bude potřeba nastavovat a zobrazovat. Navíc se potřebné grafické prvky budou lišit na základě řešeného problému a je vhodnější, pokud uživatel na první pohled pozná, ve kterém stavu se rozhraní nachází, než pokud bychom toto řešili, neustálím skrývám, či deaktivací určitých prvků.

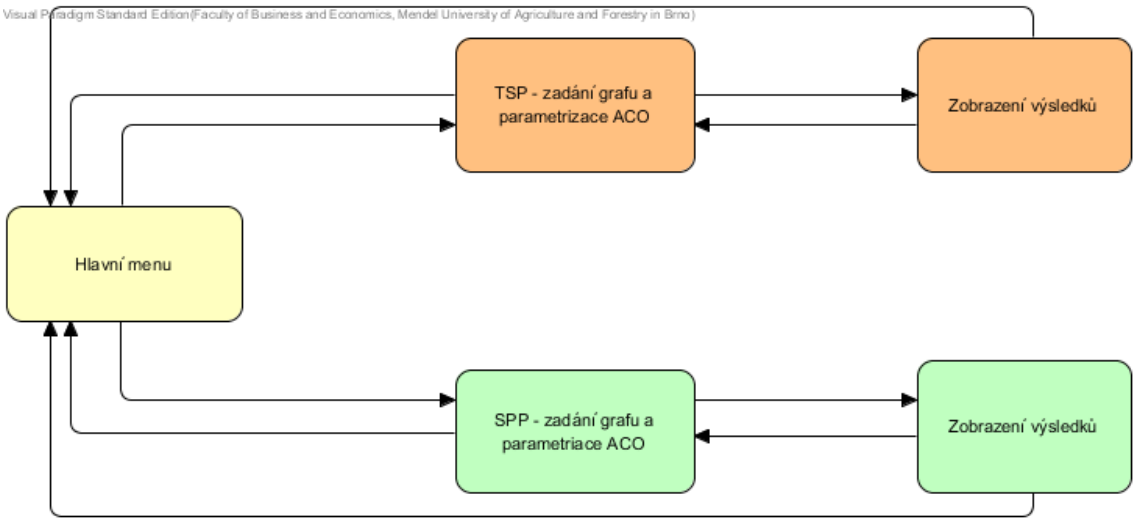
Výchozím bodem aplikace bude hlavní menu, v němž si uživatel vybere, který problém chce řešit, zda TSP či SPP. Poté se logika aplikace dělí do dvou větví, podle vybraného problému.

V dalším kroku uživatel musí zadat vstupy, se kterými se má pracovat a nastavit parametry algoritmu ACO. Tyto kroky jsou na sobě v zásadě nezávislé a mohli by být odděleny do dvou vlastních obrazovek, na druhé straně mnoho přechodových situací může uživatel mást, a jelikož zadání vstupních dat bude relativně stručně, bylo rozhodnuto, že zadávání vstupních dat i nastavování parametrů ACO bude řešeno v rámci jednoho stavu.

Ve chvíli kdy máme připraven vstupní graf i nastaven algoritmus, lze přejít do dalšího stavu, kdy již probíhá samotný výpočet a zobrazení výsledků.

Logickou strukturu grafického uživatelského rozhraní můžeme vidět na obrázku 15. Šipky, které můžeme v diagramu také vidět, nám říkají, jaké přechody mezi stavy jsou možné. Z hlavního menu se tedy dostaneme k zadání problému a

poté k výsledkům jeho řešení. Od výsledků se můžeme vrátit k zadání, které lze změnit a spustit algoritmus opakovaně. Z libovolného místa se lze vrátit do hlavního menu nebo celou aplikaci ukončit.



Obr. 15 Logická struktura GUI

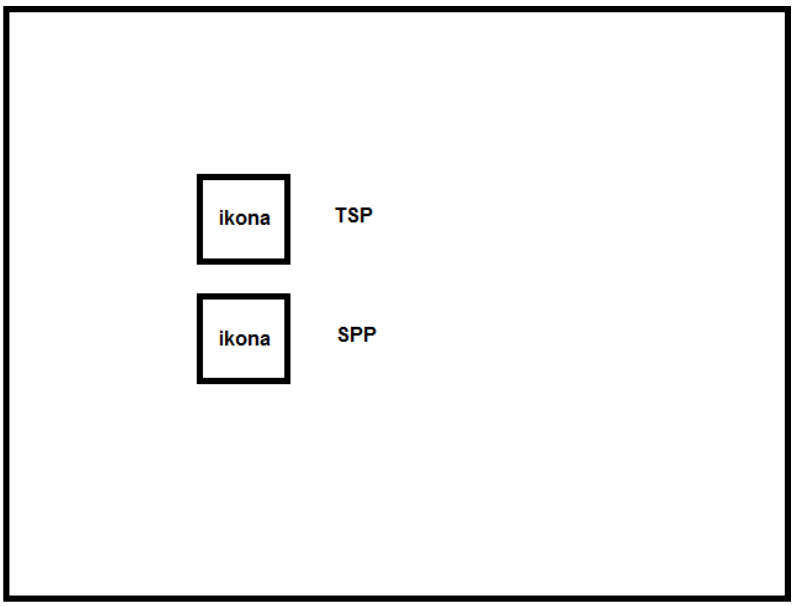
4.4.3 Drátěný model

Drátěný model se užívá k návrhu rozložení ovládacích prvků pro jednotlivé stavy uživatelského rozhraní. Tento model může mít různorodou míru podrobností, avšak nejedná se o podrobný grafický návrh. Model je většinou pouze černobílý. Známe-li již konkrétní platformu, ve které budeme návrh implementovat, můžeme pří návrh využít komponenty, které daná platforma poskytuje. Vhodnější je však se od konkrétní platformy oprostit a prvotní návrh vytvořit, tak aby co nejvíce vyhovoval našim představám o funkčním rozhraní pro daného uživatele.

Pro každý stav uživatelského rozhraní vytváříme jeden model představující rozložení prvků v daném stavu.

Hlavní menu

Hlavní menu je vstupní branou do celé aplikace, z tohoto důvodu by mělo být přehledné, ale také určitým způsobem poutavé. Hlavním účelem tohoto stavu, bude umožnit uživateli, vybrat si jaký problém chce řešit, zda TSP nebo SPP. V mnoha případech lidé mnohem lépe vnímají a reagují na obrázky než na psaný text. Z tohoto důvodu bude položka menu doplněna o zjednodušený piktogram, který uživateli umožní na první pohled si vybavit, o jaký úkol se jedná. Drátěný model obrazovky hlavního menu vidíme na obrázku 16.



Obr. 16 Drátěný model – hlavní menu

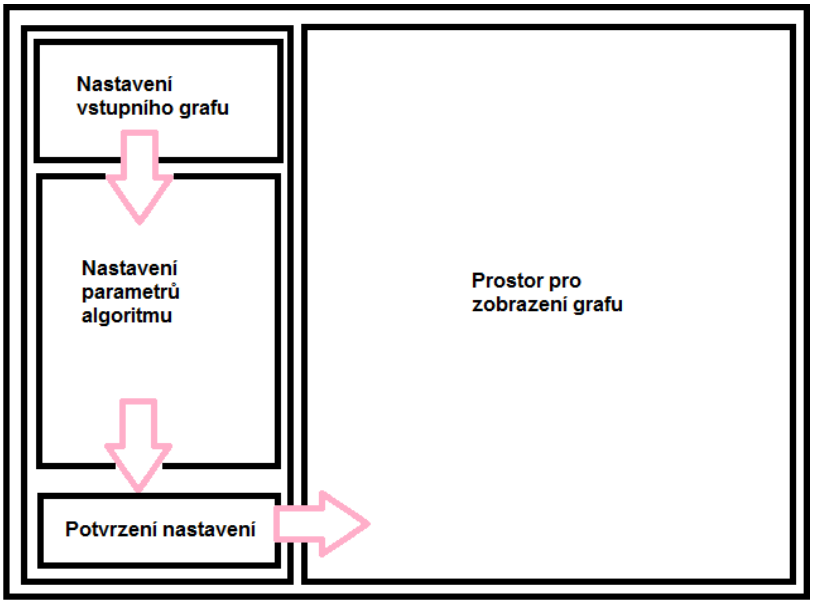
Nastavení parametrů

Z hlavního menu se uživatel dostane k dalšímu kroku, kdy bude vybírat vstupní data a nastavovat parametry algoritmu. Nastavované parametry se budou lišit, dle vybraného typu problému, avšak v této fázi konkrétní vstupní grafické prvky ještě neřešíme a jde o obecnější návrh, z tohoto důvodu nám pro reprezentaci rozložení u obou typů problému bude postačovat jeden model.

Při návrhu uživatelského rozhraní je vhodné vycházet z toho, co uživatel již běžně zná a používá, a to často nejen v počítačovém světě. Při prvním pohledu na stránku knihy, nám pohled spočine v levém horním rohu a zrak poté pokračuje směrem dolů, obdobně to bývá u počítačové obrazovky. Tomuto toku zraku je vhodné přizpůsobit rozložení. Prvky, které spolu souvisí, nebo je bude uživatel obvykle užívat v určité posloupnosti, by onu posloupnost měli následovat i v grafickém rozložení.

Na obrázku 17 vidíme návrh modelu. Rozložení je svisle rozděleno do dvou panelů. Menší část obrazovky zabírá ovládací panel, umístěný v levé části. Na tomto panelu uživatel bude zadávat potřebné parametry. Po jejich potvrzení, se ve větší části obrazovky zobrazí graf, který na základě parametrů vznikl.

Šipky v daném modelu zobrazují posloupnost, jakou bude uživatel rozhraním postupovat. V levém horním rohu je tedy umístěn první krok, nastavení vstupního grafu, poté se přechází k nastavení parametru algoritmu. Ve chvíli kdy je vše nastaveno a potvrzeno vzniká v hlavní části graf, se kterým se bude dále pracovat.



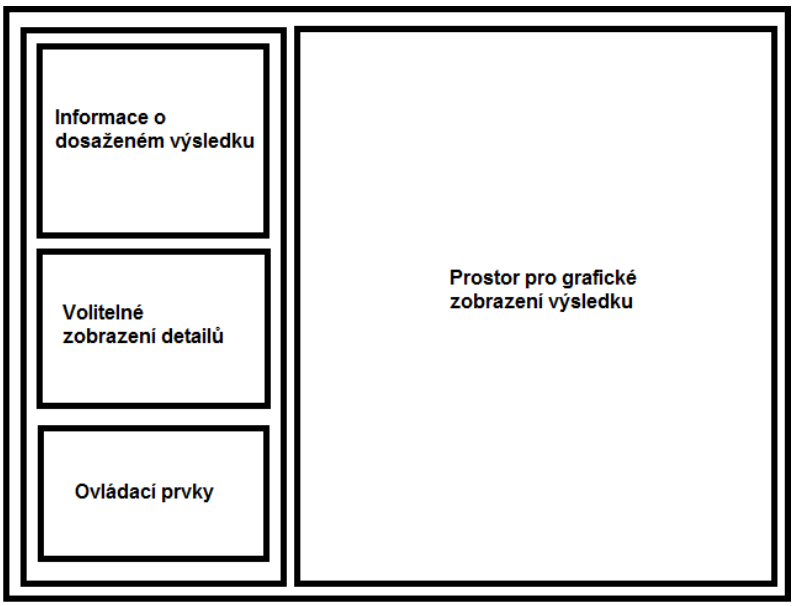
Obr. 17 Drátěný model – nastavení parametrů

Zobrazení výsledků

Pro zachování kontextu, zůstává základní rozložení pro zobrazení výsledků stejné jako ve stavu zadávání parametrů. Graf, který byl v předchozím kroku vytvořen, uživatel tedy dále uvidí v hlavní části obrazovky a v této části také bude průběžně v grafu zobrazováno vytvářené řešení.

Panel umístěný v levé části, zůstává i nadále rozdělen do tří částí. Ve spodní části se nacházejí ovládací prvky, kterými uživatel bude moci ukončit chod algoritmu, pozastavit ho apod. V levé horní části se nachází prioritní informace, tedy informace o nejlepším dosaženém výsledku, jeho délce, času, ve kterém byl dosažen atd. Ve střední části levého panelu je prostor pro zobrazení případných dalších doplňujících informaci, v tomto případě půjde zejména o doplňující informace o uzlech a hranách, hodnotách feromonu a pravděpodobnosti jejich vybrání.

Drátěný model tohoto rozložení vidíme na obrázku 18.



Obr. 18 Drátěný model – zobrazení výsledků

4.5 Implementace

Implementace byla provedena, jak bylo podrobněji uvedeno v kapitole popisující metodiku práce, ve vývojovém prostředí Visual Studio v jazyce C#. Grafické prostředí bylo vytvořeno pomocí komponent z knihovny Windows.Forms. Práce s nástroji z této knihovny je podporována přímo ve vývojovém prostředí a umožňuje nám pohodlně vytvářet aplikace s bohatým uživatelským prostředím, vycházejícím z konceptů, které každý uživatel OS Windows dobře zná.

4.5.1 Náhodná čísla

První věci, kterou bylo třeba vyřešit, je generování vstupních dat. Při maximalizaci okna se přizpůsobí i rozložení aplikace. Šířka postranního panelu zůstane stejná, ale podstatně se rozšíří, hlavní panel, tedy prostor pro zobrazení grafu. V době generování uzlů, se tedy vezme jako hranice aktuální rozměry hlavního panelu. V případě mřížky, či generování do kruhu, se použijí geometrické výpočty. Větším problémem se ukázalo generování náhodných dat. Generování náhodných, respektive pseudonáhodných čísel, je často oříšek, a to zejména voláme-li generátor náhodných čísel v iteracích ihned po sobě. V praxi docházelo k tomu, že bylo například 1000krát vygenerováno stejné číslo a až v další iteraci jiné, a tak dále. Generátor uzlů, si s touto situací poradil, jelikož bylo zajištěno, aby nebyl vytvořen nový uzel se stejnými souřadnicemi, jako má jiný již existující. Opakované generování nevhodných uzlů a čekání na rozdílné hodnoty však neúměrně navyšovalo dobu generování. Obdobný problém se navíc objevil u mravenců, kteří při řešení také využívají náhodné čísla, a zde opakované generování stejné hodnoty zcela znemožňovalo hledání optimálního řešení. Úspěšné řešení generující

pokaždé novou náhodnou hodnotu, bylo nakonec nalezeno pomocí systémového časovače a lze jej shlédnout ve třídě CRandomProvider.

4.5.2 Parametry algoritmu

V grafickém prostředí uživatel může nastavovat veškeré potřebné parametry. To mu umožňuje lépe pochopit, chod algoritmu. Ve výchozím nastavení jsou veškeré parametry nastaveny na hodnoty, které se osvědčily nejlépe. K těmto hodnotám, bylo dojito, jak na základě doporučení jiných autorů, tak především na základě vlastních pokusů a testování algoritmu. Jako nejklíčovější, se ukazuje především nastavení samotných mravenců, tedy nastavení parametrů alfa a beta, které ovlivňují funkci vyhodnocující pravděpodobnost výběru hran. Samotná metoda implementující výpočet kvality hrany je implementována následovně.

```
public double CalculateTauete(double pheromone, double distance)
    {
        double tauete = System.Math.Pow(pheromone, this.alpha) *
                        System.Math.Pow((1 / distance), this.beta);
        return tauete;
     }
```

Účelem nebylo algoritmus vyladit, na jeden konkrétní typ dat. Přesto však byly vypozorovány rozdíly mezi vhodnými nastaveními pro jednotlivá vstupní data. Zatímco pro náhodné rozložení je vhodné nastavit parametr alfa na hodnotu 2, u řešení rozložení do mřížky je vhodnější nastavit jej na 3,5. Ještě vyšší nastavení způsobuje velký rozptyl výsledků. Je tedy buď velmi rychle nalezeno optimální řešení, anebo algoritmus uvázne ve velmi nevhodném uspořádání hran. Tento stav je typický, v určité fázi se dostaneme do situace, kdy algoritmus již je odladěn, a další úpravou parametrů již upravujeme jednu vlastnost na úkor druhé. Volba je pak především na uživateli, co je pro něj vhodnější. Zda je výhodnější nastavení, které vždy dosáhne alespoň 90% optimálního řešení, anebo nastavení, které v 6 případech z 10 dosáhne 95% optima, ale ve zbytku případů zcela propadne.

Kompletní výchozí nastavení parametrů je zaneseno v tabulce 8.

Parametr	Hodnota	Poznámka
Alfa	2	pro mřížku 3,5
Beta	5	
Zvyšování feromonu	5	nastavením hodnoty
Snižování feromonu	0,9	násobením
Maximum feromonu	5	
Minumum feromonu	0,10	
Aktualizuje nejlepších	10	
Pokud je řešení lepší než	nejlepší × 1,03	
Výchozí hladina feromonu	0,5	neměnné

Tab. 8 Výchozí nastavení parametrů

Uživatel může nastavovat i počet nejlepších mravenců, kteří budou inkrementovat feromon a dále u těchto mravenců nastavit dodatečnou podmínku, říkající, jak dobré musí být jejich řešení, aby jim to bylo skutečně dovoleno. Kombinace těchto podmínek, algoritmus spíše zbytečně zesložiťuje, ale je vhodná pro případné experimenty. Nastavením počtu nejlepších mravenců na stejný počet jako je počet měst, můžeme simulovat situaci, kdy budou moci feromon pokládat všichni mravenci, jejichž řešení je lepší než nejlepší řešení krát daná podmínka. Nebo můžeme tuto podmínku naopak zcela deaktivovat a aktualizaci bude provádět vždy n nejlepších mravenců.

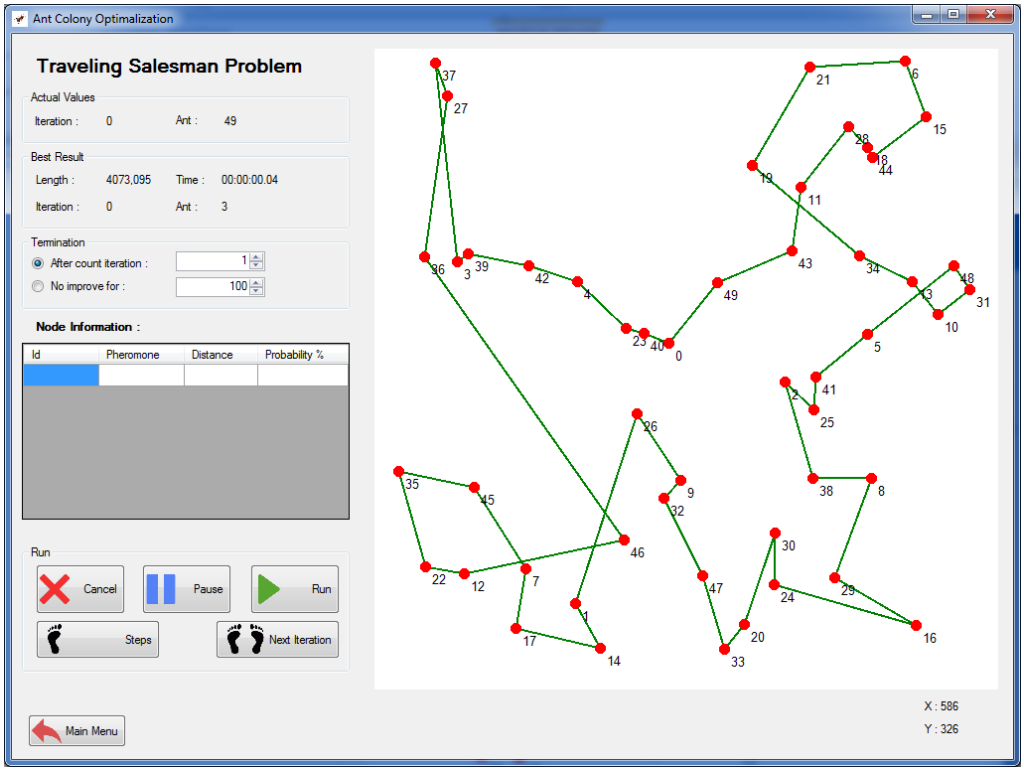
4.5.3 Zobrazení grafu

Zobrazování grafu, je prováděno na hlavním panelu pomocí postupného vykreslování patřičných objektů. Vhodné vykreslování postupně vytvářeného řešení je hlavní podmínka pro úspěšné edukativní použití aplikace. Na základě hodnoty z enumerace ERenderMode, je vždy rozhodnuto, které objekt jsou v dané situaci vykreslovány. Po vytvoření grafu jsou standardně vykreslovány i hrany vzniklého grafu. Z důvodů zdlouhavého vykreslování hran je však toto činěno pouze do počtu uzlů menšímu nebo rovno 100.

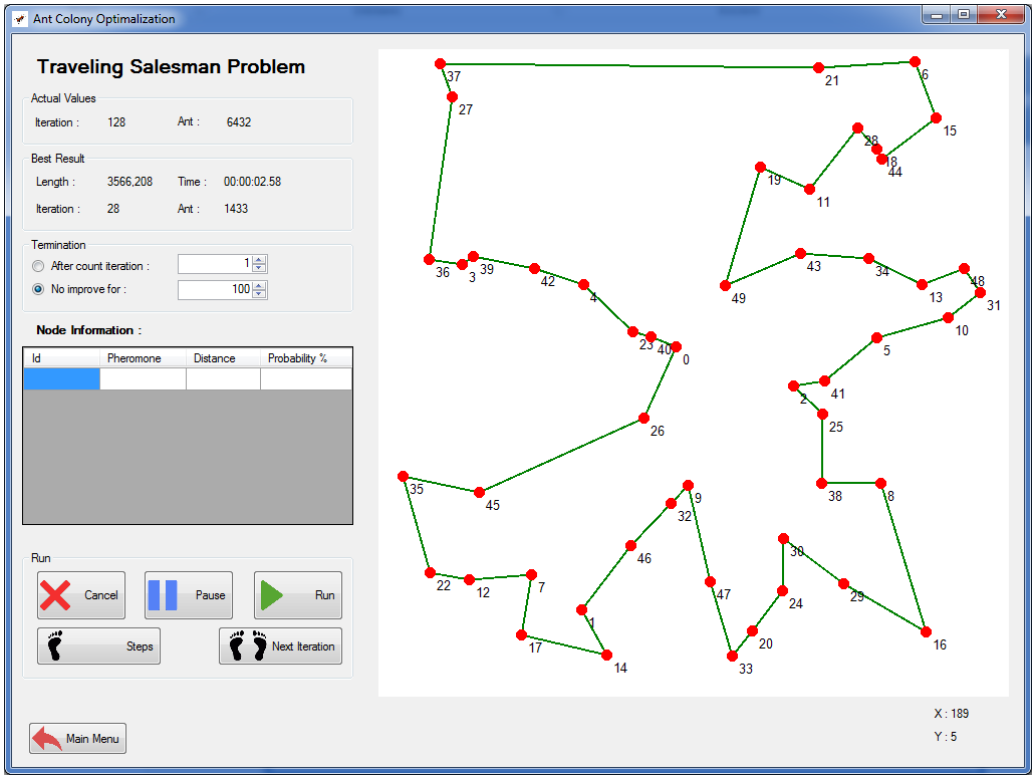
Uživatel vždy před spuštěním algoritmu rozhodne, zda se má výpočet ukončit po pevně daném počtu iterací, nebo pokud nedojde po určitý počet iterací ke změně. Výchozí hodnota je ukončení pokud nedojde po 100 iterací ke změně.

Na obrázku 19 a 20, vidíme způsob, jakým je vykreslováno vytvářené řešení. Na obrázku 19 vidíme stav, který vznikl po první iteraci algoritmu. V této iteraci byla na všech hranách výchozí hladiny feromonu. Na obrázku 20 je již vidět finální stav, k nalezení této cesty došlo v iteraci 28, v čase 2,58 sekundy.

Aby bylo možné překreslovat graf a zároveň aktualizovat informace a ovládat celou aplikaci, musí GUI a samotný algoritmus běžet na více vláknech. Více o této tématice je popsáno v kapitole 4.8.1.

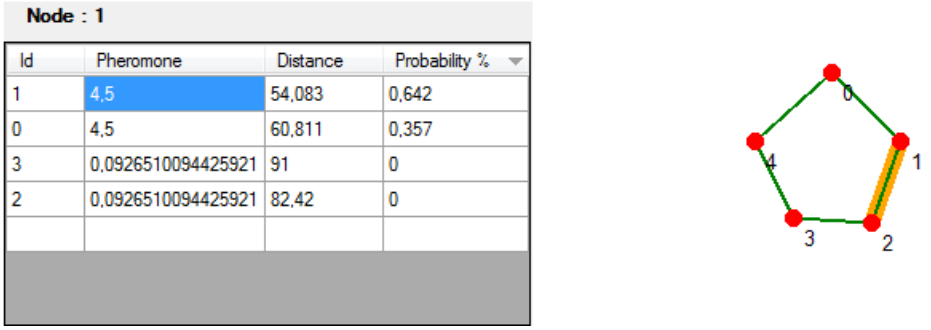


Obr. 19 Ukázka řešení po první iteraci



Obr. 20 Ukázka vyřešeného TSP

Uživatel si dále může v libovolnou dobu, kliknutím na uzel grafu zobrazit informace o něm. V takovém případě se v tabulce, v levém panelu zobrazí informace o hranách vycházejících z daného uzlu. Zobrazenými informacemi jsou délka hrany, hladina feromonu a z nich počítaná pravděpodobnost výběru. Po kliknutí na řádek tabulky, se navíc zobrazí daná hrana v grafu oranžově zvýrazněná. Opět jde o jeden z bodů, umožňující uživateli lépe pochopit algoritmus.



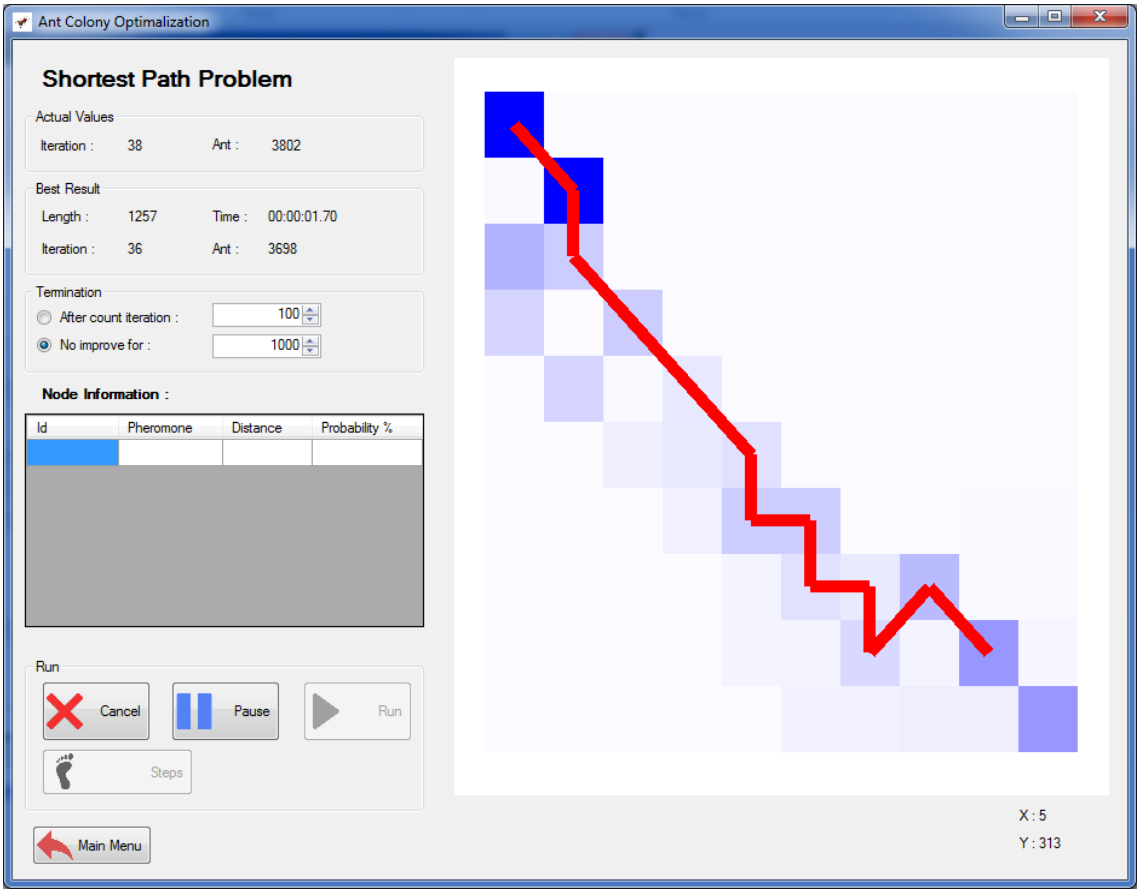
Obr. 21 Ukázka zobrazení podrobností o uzlu

4.5.4 Hledání nejkratší cesty

Hledání nejkratší cesty, bylo do programu přidáno zejména jako dobrá ukázka kořenů algoritmu, na kterém jsou velmi dobře vidět aktuální hodnoty feromonu a jejich zvyšující se a snižující se hodnoty. Hodnoty feromonu jsou samozřejmě standardně ukládány na hranách, pro přehledné vykreslování je zde ale hodnota feromonu počítána jako průměr hodnot feromonu na hranách z uzlu vycházejících. Uživateli v případě potřeby přesnějších informací stále zůstává možnost po kliknutí na uzel, si zobrazit více informací.

V případě hledání nejkratší cesty, se nejedná o kompletní graf. Uzly jsou v tomto případě poskládány do mřížky a každý uzel je spojen do hvězdice s 8 nejbližšími uzly. U krajních uzlů jde o nejbližších 5 sousedů, u rohových nejbližší 3.

Ukázku z průběhu řešení tohoto problému vidíme na obrázku 22. Optimalizace ACO na tento typ problému, by mohl obsáhnout samostatnou práci, ale i původní mravenci navržení pro TSP, jsou po úpravách schopni úspěšně problém řešit. U prostoru s většími rozměry, ve chvíli kdy jsou feromonové stopy na výchozí hodnotě, poměrně dlouhou dobu trvá první nalezení cíle, poté už aktualizace řešení probíhají podstatně dynamičtěji.



Obr. 22 Ukázka průběhu aplikace při hledání nejkratší cesty

4.6 Import, export zadání

Aplikace umožňuje, aby si uživatel vytvořený graf exportoval a importoval, a tím je umožněno opakované testování na stejných datech. Formát takto vytvořených souborů je popsán v kapitole návrhu aplikace. Této funkcionality lze dle využít, ve chvíli kdy řešíme TSP pomocí druhé vytvořené konzolové aplikace, ta standardně generuje soubor s výsledným sledem uzlů, který lze importovat do aplikace s GUI a nalezené řešení si zde tedy zobrazit.

4.7 Ověření správnosti implementace

Klíčovým faktorem ověření správnosti implementace, je schopnost dosahovat požadovaného optima. Jak již bylo řečeno, v oblasti NP úloh, je, pokud nechceme prohledávat celý stavový prostor, velmi obtížné dokázat, že se jedná skutečně o nejlepší možný dosažitelný výsledek.

Z tohoto důvodu je potřeba se spokojit, s jistými ukazateli, které nám indikují správnost řešení, či implementaci testovat na příkladech se známým řešením. Dále je třeba si uvědomit, že se jedná o metaheuristickou metodu, ve které hraje

určitou roli náhoda, a my se snažíme přiblížit optimu. Z tohoto důvodu situace, kdy při opakovaném spuštění algoritmu na stejných, složitějších datech získáme odlišné výsledky, není chybou. Ale například i rozptyl těchto výsledků, by případně mohl být jeden ze zkoumaných aspektů algoritmu.

Prvotním nástrojem, který dokáže zjistit hodnotu nalezeného výsledku je samotný uživatel. Pokud například jsou hrany překřížené, je jisté, že výsledek není optimální. Uživatel, však v tu chvíli je schopen odhalit, výsledek, který je velmi špatný, či u dobrého výsledku odhalit jisté možné suboptimalizace. Pokud však v grafu existuje řešení lepší, jehož konstrukce, je od začátku zcela odlišná, těžko toto bude lidským zrakem odhalitelné.

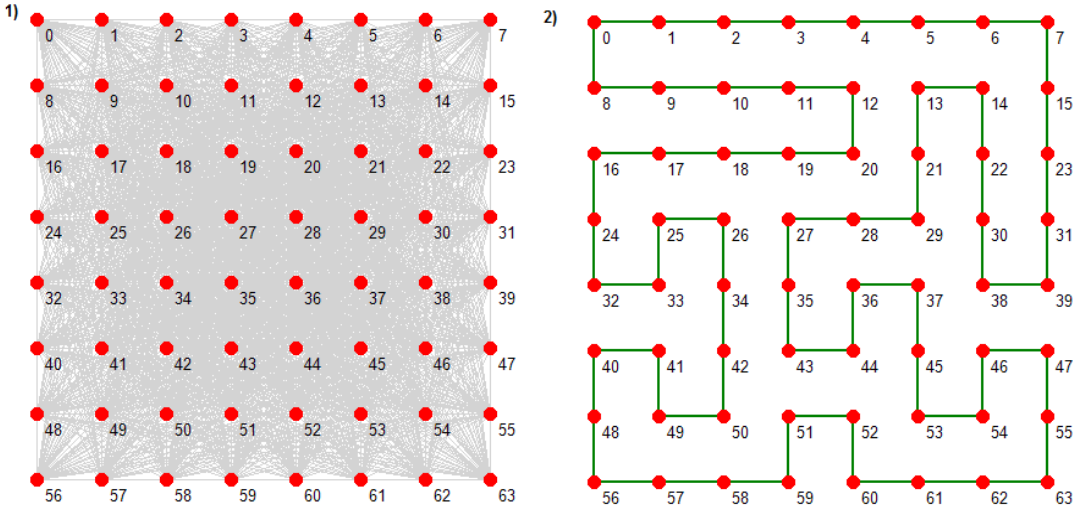
Další použitou metodou ověření správnosti řešení je tedy zkouška na příkladech, u kterých známe řešení.

4.7.1 Příklady se známým řešením

Mřížka

Umístíme-li uzly grafu do pravidelné mřížky, ve které jsou vertikální i horizontální rozestupy stejné, vznikne nám struktura, ve které není problém určit správné řešení. Pokud bychom znali i přesnou hodnotu rozestupů řádků a sloupců, moli bychom si i konkrétní hodnotu řešení i vypočítat.

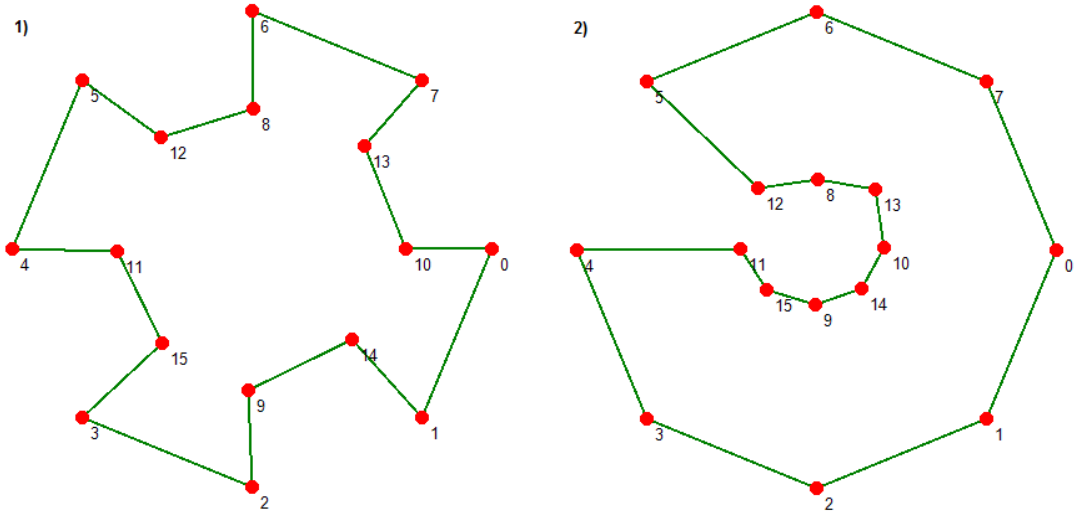
Zcela ekvivalentních řešení v této struktuře je vícero, avšak pokud má být řešení optimální, musí obsahovat pouze propojení uzlů pomocí vertikálních a horizontálních hran. Každý uzel, je tak propojen právě se 2 ze svých 4 nejbližších sousedů.



Obr. 23 Graf s uzly seřazenými do mřížky a zobrazenými hranami (1), graf s nalezeným optimálním řešením (2)

U uzlů rozmístěných do mřížky, či rozmístění mřížce se velmi blížícímu, se jeví vhodným zvýšit hodnotu parametru alfa do rozmezí 3,2- 3,8.

Dvojitá kružnice
Toto rozmístění uzlů, předpokládá jejich rozmístění do dvou soustředných kružnic. Na vzdálenosti těchto kružnic následně záleží, zda je optimální výsled podobající se ozubenému kolu, či výsledek připomínající písmeno c.



Obr. 24 Rozdílné optimální řešení v závislosti na poloměru 2 soustředných kružnic

4.7.2 Srovnání s hladovým algoritmem

Hladový algoritmus, nebo také algoritmus nejbližšího souseda pracuje tak, že z dané uzlu, nalezneme jeho nejbližšího souseda, a tyto dva sousední uzly spojíme hranou. Poté se přejde do tohoto nově připojeného souseda a v něm je provedeno totéž a tak dále, až dokud nejsou propojeny všechny uzly. Nakonec je propojen poslední připojený uzel se startovacím uzlem.

Tento algoritmus získá výsledek velice rychle a i jeho implementace byla velmi rychlá. Stačilo pouze mravenci upravit metodu výběru následující hrany, aby nebrala v potaz feromon a vybrala nejkratší hranu vedoucí z uzlu (tuto možnost uživatel nemá k dispozici, byla implementována pouze pro testování).

Výsledky tohoto algoritmu jsou lokálně velmi dobré, avšak v určité fázi se algoritmus většinou dostává do situace, kdy mu nezbývá jiná možnost, než vybrat některou z velmi nevýhodných hran a jeho výsledky jsou tak nakonec znatelně horší než u ACO.

Počet uzlů	Průměrný výsledek vlastní ACO	Výsledek hladový algoritmus	Rozdíl
25	2242	2482	+10,70%
50	3082	3360	+9,02%
75	4560	5325	+16,78%
100	4862	5416	+11,39%

Tab. 9 Srovnání výsledků s algoritmem nejbližšího souseda

4.7.3 Srovnání s grafickou aplikací využívajícími genetické algoritmy

Implementovaná aplikace byla srovnána s obdobnou aplikací, která také obsahuje grafické uživatelské rozhraní (značně zjednodušené), a která řeší problém TSP pomocí genetických algoritmů. Jedná se o aplikaci autora Michaela LaLena a je dostupná z jeho stránek http://www.lalena.com/AI/Tsp/. Jako ukončující podmínka byl zvolen limit 5 minut běhu aplikace.

Počet uzlů	Průměrný výsledek vlastní ACO	Výsledek algoritmu LaLena	Rozdíl
25	2242	2269	+1,20%
50	3082	3166	+2,72%
75	4560	4731	+3,75%
100	4862	5123	+5,36%

Tab. 10 Srovnání výsledků s genetickým algoritmem LaLena

Rychlost nalezení řešení je u vlastní implementace vyšší než u srovnávané aplikace, avšak v této fázi prozatím není klíčová. Klíčová pro zhodnocení algoritmu je schopnost dosáhnout co nejlepšího řešení, bez uváznutí v lokálním optimu. Toto porovnání je tedy spíše přibližné, především ověřující hodnotu výsledků. V pozdějších kapitolách bude implementace dále optimalizována a srovnána exaktnější metodou.

4.8 Optimalizace implementace

V tuto chvíli byla ověřena funkčnost vytvořené implementace. Je známo, které nastavení parametrů je pro dané problémy nejlepší, a že výstupy aplikace jsou relevantní vzhledem k hledanému optimu. Je tedy možné přistoupit k optimalizaci implementace, která zrychlí hledání řešení.

Dosažení řešení v co nejkratším čase, je v tuto chvíli naším prioritním cílem. Na stejně zadaných vstupních datech a za stejného nastavení parametrů výstupy

musejí odpovídat výstupům u předchozí implementace avšak pohodlnost ovládání aplikace, její edukativnost, či informování uživatele o průběhu řešení se stávají druhořadými požadavky.

Z výše uvedeného vyplývá, že prvním bodem optimalizace, je odstranění grafického uživatelského rozhraní. Grafické uživatelské rozhraní využívalo vlastní vlákno, které při běhu algoritmu neustále muselo zajišťovat vykreslování grafu, vykreslování nalezené nejkratší cesty a některé další textové informace o probíhající iteraci. Tento krok také sníží počet využívaných knihoven, a zmenší tak samotnou velikost vzniklých binárních souborů a jeho nároky na operační paměť.

Ve vývojovém prostředí Visual Studio tedy byl vytvořen nový projekt a to typu Console application. V takto vytvořeném projektu, máme kromě dalších souborů ihned vytvořený soubor s názvem App.config. Jedná se o konfigurační soubor aplikace. Tento soubor je distribuován spolu s aplikací, a ta z něj při spouštění čte nastavené atributy. Souboru může v rámci daných zásad editovat jak programátor ve vývojovém prostředí tak samotný uživatel aplikace, před jejím spuštěním. Této vlastnosti tedy využijeme a pomocí tohoto souboru, budou uživatelé specifikovat vstupní textový soubor obsahující informace o grafu, se kterým se bude pracovat a další parametry algoritmu. Po sestavení aplikace a její distribuci tento soubor již standardně nemá název App.config, ale běžný uživatel jej nalezne pod názvem shodným jako je název spustitelného souboru, obohacený o koncovku .config.

Do této chvíle jsme u algoritmu, řešili především hodnotu dosaženého výsledku, parametry algoritmu, či stav řešeni po určité iteraci, tyto hodnoty jsou zcela nezávislé na výkonu počítače, na kterém aplikace bude spuštěna. V tuto chvíli, však začneme operovat s časem, po který určitý počet iterací bude trvat. Tento údaj může být na různých počítačových sestavách různý a z tohoto důvodu lze v tabulce 11 vidět složení počítačové sestavy, na které byla implementace testována.

Základní deska	GIGABYTE GA-870A-UD3
Procesor	AMD Phenom II X4 955 Black Edition
Operační paměť	8 Gb, DDR3 670 MHz, Dual-Channel, Kingston
Primární pevný disk	Kingston SSD V300, 120 Gb
Grafická karta	Sapphire Radeon HD 5750
Operační systém	Microsoft Windows 7 Professional (x64)

Tab. 11 Testovací PC sestava

4.8.1 Paralelizace algoritmu

Operační systém používá procesy, k oddělení různých aplikací, které jsou vykonávány. Vlákno je základní jednotkou, které operační systém přiděluje čas procesoru a více vláken může vykonávat kód uvnitř procesu.

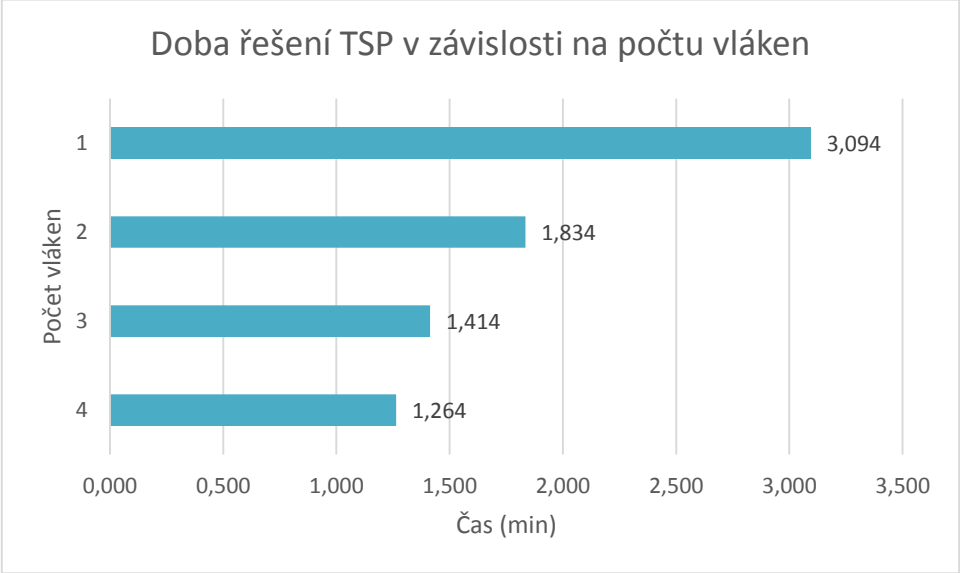
Operační systém, který podporuje preemptivní multitasking, vytváří efekt souběžného výkonu více vláken z procesu, případně více procesů. To provádí tak, že čas procesoru dělí mezi vlákna. Vlákno dostane přidělený určitý časové okno, jakmile tento úsek uplyne, je vykonávání prvního vlákna pozastaveno a je vykonáváno jiné vlákno.(Threads and Threading, 2015)

V případě systému s více procesory, nebo systému osazeném procesorem s více jádry, je umožněno reálné vykonávání více vláken skutečně paralelně.

Těchto vlastností vláken bylo využito při vytváření aplikace s grafickým rozhraním, kdy nám toto umožnilo informovat uživatele a měnit vykreslení grafu i ve chvíli, kdy byl spuštěn výpočet algoritmu. Další možností jak vláken využít, je paralelizace samotného výpočtu. ACO se k takovéto paralelizaci jeví jako velmi vhodný, jelikož jednotlivý mravenci hledají cestu na sobě nezávisle a pro své hledání potřebují pouze číst informace z grafu. K případné editaci informací, rozuměj aktualizaci feromonu, při které by bylo třeba zajišťovat exkluzivní přístup, dochází až na samotném konci iterace, kdy již všichni mravenci mají svoji cestu nalezenou.

Jazyk C# poskytuje bohaté možnosti, jak implementovat více vláknovou aplikaci. Jedním z nich, je metoda ForEach třídy Parallel, ze jmenného prostoru Threading. Funguje na principu více vláknového zpracování objektů v kolekci. Dané metodě dále můžeme předat objekt s nastavením, které určí, kolik vláken má být maximálně použito. Nastavením jednoho vlákna, tak lze simulovat situaci, kdyby žádná paralelizace zpracování nebyla provedena.

V našem případě půjde o kolekci, ve které budou vytvořeny instance mravenců, umístěných ve startovacích uzlech. Jednotlivý mravenci budou paralelně zpracování, čili bude u nich vykonána metoda MoveToNextNode,



Obr. 25 Doba řešení TSP v závislosti na počtu vláken

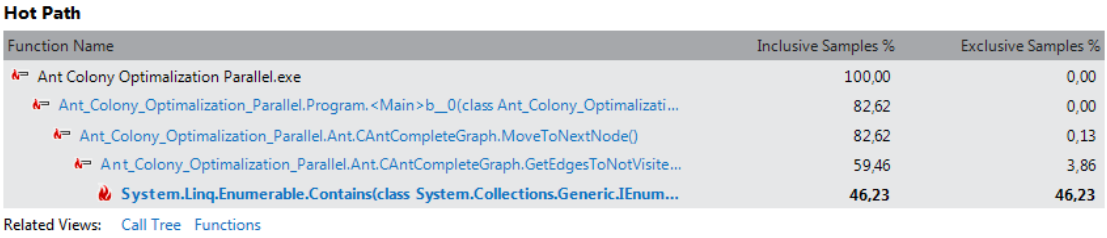
Jak vidíme v grafu (obr. 25) nárůst výkonu při paralelizaci je skutečně značný. Při stejném zadání je rozdíl mezi jedním vláknem a 4 vlákny téměř trojnásobný. Z čistě teoretického hlediska, bychom při použití dvou vláken, ve srovnání s jedním mohli dosáhnout dvojnásobného zrychlení. V tomto případě však nastavují úkony, které nelze efektivně paralelizovat. Jedná se seřazení mravenců a následnou aktualizaci feromonu. Dále samotná režie vláken, může efektivitu mírně snižovat.

Největší skok je mezi zpracováním pomocí1 vlákna a pomocí 2 vláken, kdy se blížíme poměru přibližně 1,7. Poté poměr klesá. Při využití 4 vláken se již dostáváme na maximální počet vláken, která dokáže daný procesor efektivně souběžně zpracovávat. Procesor však souběžně musí vytěžovat také operační systém, či další aplikace běžící na pozadí. Z tohoto důvodu, přestože aplikace využívá 4 vlákna, tyto vlákna jsou souběžně zpracovávána jen v minimální míře, a tedy nárůst výkonu ve srovnání s použitím 3 jader je minimální.

4.8.2 Ostatní optimalizace

Visual Studio 2013 Ultimate, obsahuje nástroje nazývané Profilling Tools, které programátorovi umožňují měřit, hodnotit a zaměřovat výkonové problémy v jejich kódu. Tyto nástroje jsou plně integrovány do vývojového prostředí.(Analyzing Application Performance, 2015) Pomocí těchto nástrojů můžeme objevit operace, které spotřebovávají nejvíce procesorového času při běhu aplikace.

Výstupem takovéto analýzy je tzv. Hot Path, tedy výpis zanoření od hlavní metody celé aplikace, až po inkriminovanou metodu, u které se spotřebovává nejvíce času.



Obr. 26 Ukázka analýzy Hot Path

Ukázku, jak může vypadat výstup pro analýzu Hot Path můžeme vidět na obr. 18. V daném případě nám ono zanoření říká, že z hlavní metody je volána metoda MoveToNextNode(), třídy CAntCompleteGraph. V ní je dále volána metoda GetEdgesToNotVisitedNodes() a v ní je ona nejnáročnější operace, a to volání metody Contains(CNode node), nad kolekcí, obsahující již navštívené uzly.

Principiálně ve všech případech, které byly analyzovány jako časově náročné operace, se jedná o operace, které jsou volány opakovaně, mnohokrát při jedné iteraci algoritmu. Jde tedy o operace, které využívají mravenci při svém hledání optimální cesty.

Vyhledání uzlu v navštívených uzlech

První operace, která byla analyzována jako náročná a byla optimalizována, je již zmíněná situace, vyhledání uzlu v navštívených uzlech.

Mravenec se nachází v uzlu a musí se rozhodnout, do jakého uzlu přejde. K tomu potřebuje vědět, které uzly jsou ze současného uzlu dostupné. Zajímají ho však pouze hrany vedoucí do uzlů, ve kterých ještě nebyl. Proto postupně musí projít všechny sousední uzly a ověřit si, zda se nacházejí v seznamu již navštívených uzlů.

Tento seznam byl původně implementován pomocí kolekce LinkedList, která umožňuje postupné přidávání objektů na konec seznamu. Pokud však v této kolekci chceme vyhledávat, v nejhorším případě bude nutné postupně projít všechny objekty v kolekci, náročnost této operace tedy roste lineárně s délkou seznamu.

Optimalizace byla provedena tak, že kolekce LinkedList byla nahrazena kolekcí HashSet, u níž je doba vyhledávání konstantní nehledě na počet prvků. Nevýhodou je, že prvky v ní nemáme nijak seřazené a chceme-li následně zrekonstruovat cestu mravence, musíme si tyto informace uchovávat jiným způsobem. Tato optimalizace přinesla značné zrychlení chodu programu, avšak poté byla opět vyhodnocena jako náročná a bylo potřeba provést další optimalizaci.

Ta byla provedena tak, že kolekce uchovávající navštívené uzly, byly nahrazeny polem. Kolik uzlů mravenec navštíví, víme, proto není problém pole iniciovat do požadované velikosti. Pole obsahuje pouze hodnoty pravda/nepravda, určující zda byl uzel navštíven. O který uzel se jedná, nám říká index prvku pole.

V praxi to tedy vypadá následovně. Mravenec získá pomocí hrany odkaz na sousední uzel, informace, která ho zajímá je jeho id (uzly jsou označeny číselně počínaje 0), toto id následně použije pro přístup ke konkrétnímu prvku pole a

bool hodnota nacházející se v tomto prvku mu oznámí, zda byl již uzel navštíven. Poté co mravenec do některého prvku přejde, se adekvátně do daného pole zaznamená nově navštívený prvek. Díky této optimalizaci, již nemusíme v kolekci vyhledávat, ale přistupujeme na konkrétní index pole, a to je velmi velice rychlá operace s konstantní časovou náročností nehledě na to, o který prvek se jedná.

Tato optimalizace přinesla největší časový přínos, ale ještě následovaly další.

Výpočet pravděpodobnostní funkce.
Ve chvíli, kdy má mravenec k dispozici seznam uzlů do kterých může z uzlu, ve kterém se nachází, přejí. Je vykonávána operace, které pro každou hranu, která do takovéhoto uzlu ve, je vypočítána hodnota funkce, která určí, na základě délky hrany a hodnoty feromonu, s jakou pravděpodobností bude daná hrana vybraná pro přechod.

Tato operace byla také vyhodnocena jako časově náročná a proto byla optimalizována. Při optimalizaci se vycházelo z toho, že pokud se nezmění hodnota feromonu na hraně, tak je hodnota pravděpodobnostní funkce nezměněná. U každé hrany byl tedy vytvořen atribut umožňující zjistit, zda došlo ke změně feromonu. Pokud ke změně nedošlo, je vrácena poslední známá hodnota pravděpodobnostní funkce, která je nově uložena na hraně. Pokud ke změně došlo, mravenec vypočítá nově pravděpodobnostní funkci a uloží její hodnotu na hranu.

Dotaz na délku kolekce v cyklech
Každý programátor se při své práci setkal s cykly. Mravenci při hledání optimální cesty vykonají cyklus prohledání hran vždy v každém uzlu. Ukončující podmínka tedy může znít: prováděj, dokud index prvku pole je menší než délka pole. Při takto formulované podmínce ukončení, musí být při jejím vyhodnocení vždy zjištěna i délka pole. Pokud si délku pole nejprve uložíme do lokální proměnné a do podmínky použijeme tuto lokální proměnou, rychlost vykonávání cyklu se zvýší.

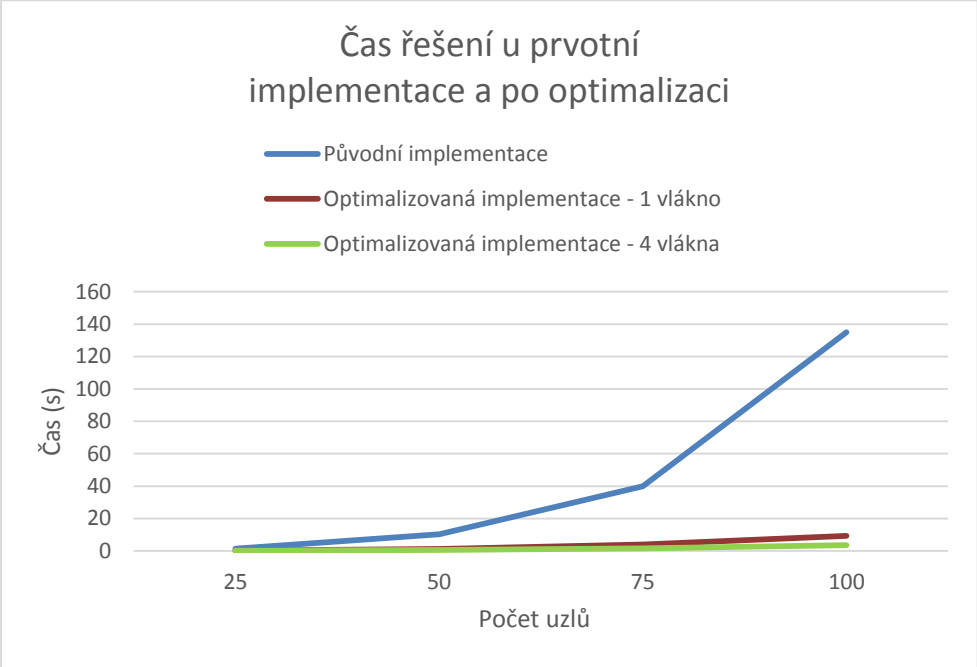
4.8.3 Shrnutí výsledků po optimalizaci

V tabulce 12 vidíme srovnání doby běhu jednotlivých implementací. Jedná se o dobu provedení 100 iterací algoritmu. Jak je vidět, provedené optimalizace nejsou pouze kosmetické, ale mají velmi zásadní vliv na dobu běhu algoritmu. Jestli se řešení daného problému bude hledat déle než dvě minuty, anebo budeme mít výsledek za 4 sekundy, to už je zcela zásadní rozdíl.

Uzly	Původní aplikace s GUI	Optimalizovaná aplikace - 1 vlákno	Optimalizovaná aplikace - 4 vlákna
25	1,279 s	0,2365 s	0,177 s
50	10,2435 s	1,256 s	0,5375 s
75	39,8225 s	3,8395 s	1,548 s
100	135 s	9,178 s	3,596 s

Tab. 12 Srovnání doby běhu v závislosti na implementaci

Ještě lépe je daný vliv vidět na grafu (obr. 27). V závislosti na počtu uzlů roste i časová náročnost algoritmu. Nehledě na implementaci, ve srovnání s polovičním počtem uzlů v přibližně stejných násobcích, avšak, to co při 50 uzlech dělá rozdíl v řádu jednotek sekund, při 100 uzlech už je v řádu minut. Při vyšším počtu uzlů by se tedy absolutní rozdíl dále zvyšoval a stával by se klíčovým faktorem použitelnosti uzlu.



Obr. 27 Srovnání doby běhu v závislosti na implementaci

4.8.4 Paměťové nároky

Faktor, určující vhodnost implementace, který tu nebyl doposud zmíněn, je paměťová náročnost implementace. Důvodem je fakt, že paměť v současné době při vývoji dané aplikace nebyla nikterak omezující faktor.

V praxi to tedy znamená, následující vlastnosti. Z důvodu kompatibility, je aplikace standardně kompilována jako 32-bitová aplikace, v takovém případě dokáže aplikace v reálném užití alokovat necelé 2 Gb operační paměti, pokud ji

z důvodu velkého počtu uzlů, respektive velikého počtu hran existujících mezi těmito uzly potřebuje alokovat více, aplikace spadne s chybou oznamující vyčerpání dostupné paměti. I s tímto omezením však lze vytvořit graf s 1000 uzlů. V tomto případě však 1 iterace algoritmu, při využití 4 vláken trvá 21 minut. V případě, že by uživatel chtěl jako vstup použít graf s ještě více uzly, lze vytvořit 64-bitovou aplikaci, která teoreticky dokáže alokovat paměť až v řádu Tb. Aplikaci lze jako 64-bitovou zkompilovat velice lehce, za pomocí zdrojových kódů, které jsou přiloženy jako elektronická příloha této práce, pouhou změnou jednoho parametru v nastavení projektu.

Z výše uvedeného vyplývá, že v praxi by pravděpodobně reálné využití aplikace neomezovali nesplnitelné paměťové nároky, ale spíše neúměrně dlouhá doba potřebná k nalezení žádaného řešení. Konkrétní hodnoty využívané paměti, lze shlédnout v kapitole srovnávající vlastní implementaci s jinou existující implementací ACO.

4.9 Porovnání řešení

Aby mohla být implementace definitivně zhodnocena, je pochopitelně vhodné ji srovnat s jinými implementacemi optimalizačních algoritmů. Tento úkol se však ukázal poněkud složitější než by se z počátku mohlo zdát.

Na Internetu a z dalších zdrojů lze najít mnoho odborných prací řešících TSP, avšak získat funkční aplikaci, nebo zdrojové kódy, které by šli bez problémů zprovoznit je velmi složitý úkol.

Mnoho volně dostupných aplikací má v sobě zabudovaný vlastní generátor, který jim vytváří vstupní data, v lepším případě má uživatel možnost definovaný vstupní soubor, ze kterého pokud splňuje jimi požadovaný formát, jsou schopny načíst vstupní graf, ale i pak je často problém dostat výstupy, jako je délka výsledné cesty ve srovnatelných jednotkách.

Dalším úskalím může být možnost parametrizace. Osobně jsem se ve své implementaci snažil vždy volit vhodné výchozí hodnoty, které budou na náhodných datech, dávat co nejlepší výsledky, z tohoto předpokladu jsem vycházel také u porovnávaných implementací, nelze však vyloučit, že v případě úpravy jejich nastavení, by bylo možné dosahovat lepších výsledků.

Výsledky, které jsou níže uvedeny, jsou vždy zprůměrované výsledky, 20 běhů aplikace. Byla zkoumána také závislost dosažených výsledků v závislosti na poštu uzlů v grafu. Uzly v grafu byly náhodně rozmístěny, jako vstup pro oba algoritmy však byly použity stejně rozmístěné uzly.

4.9.1 M. Tim Jones Ant algorithm

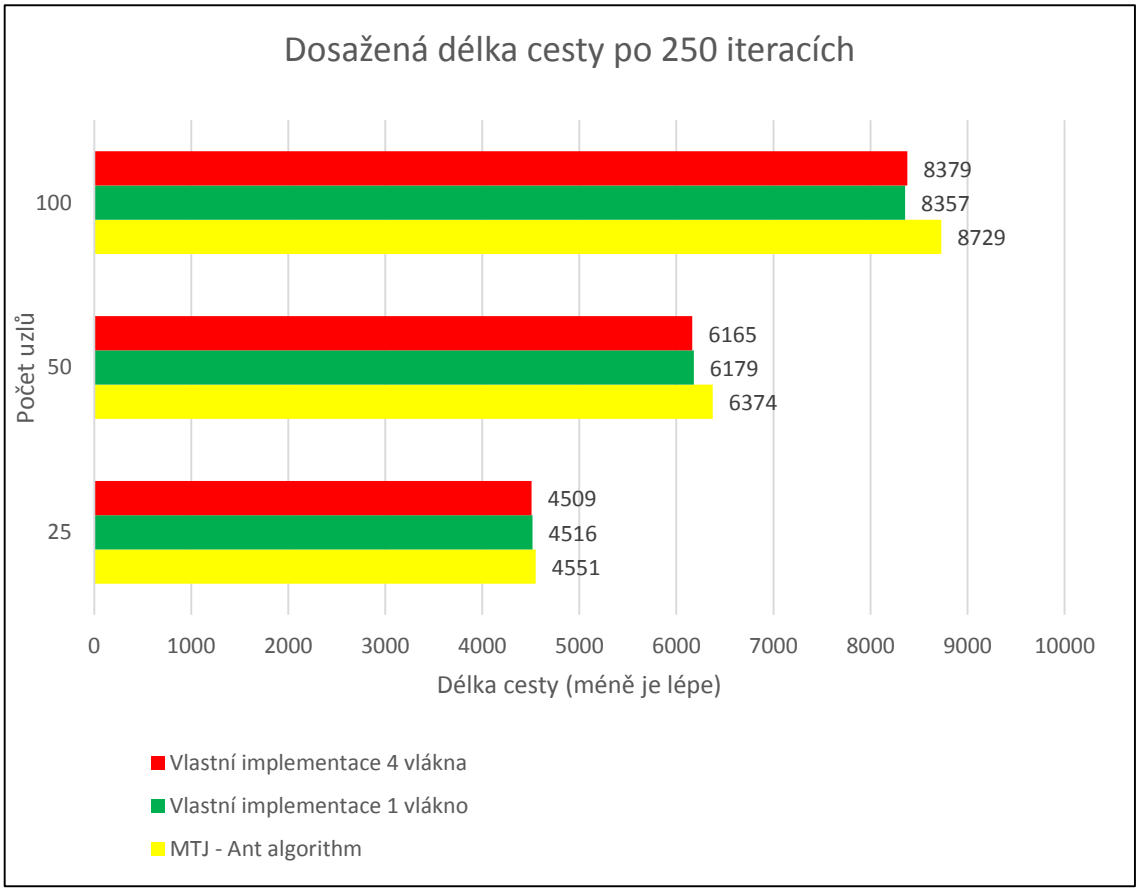
K porovnání byl nakonec použit algoritmus autor M. Tima Jonese (dále MTJ Ant algorithm), autora knihy AI Application Programming. Zdrojové kódy jsou napsány v jazyce C a byly mi poskytnuty vedoucím mé diplomové práce.

Aby mohla být implementace porovnána s moji vlastní, bylo potřeba daný zdrojový kód obohatit o možnost zadávání vlastních vstupů. Standardně bylo při každém spuštění generováno nové náhodné rozložení uzlů.

V této implementaci obdobně jako v implementaci mnou vytvořené je během jedné iterace vytvořeno tolik mravenců, kolik je hran a následně je hledána optimální cesta. Díky tomuto je umožněno srovnávání výsledků po určeném počtu iterací, což nám umožňuje přesněji získávat data, než pokud bychom chtěli srovnávat stav v určitém čase běhu programu.

Srovnání délky nejlepšího výsledku

Klíčovým ukazatelem je délka nejlepší dosažené délky hamiltonovské kružnice řešící TSP. Algoritmus, který je velmi rychlí, ale jeho výsledek je dalo od optima, může mít v určitém případě také smysl, ale rozhodně jej nelze srovnávat s algoritmem, jehož výsledek je podstatně lepší, byť v delším čase.



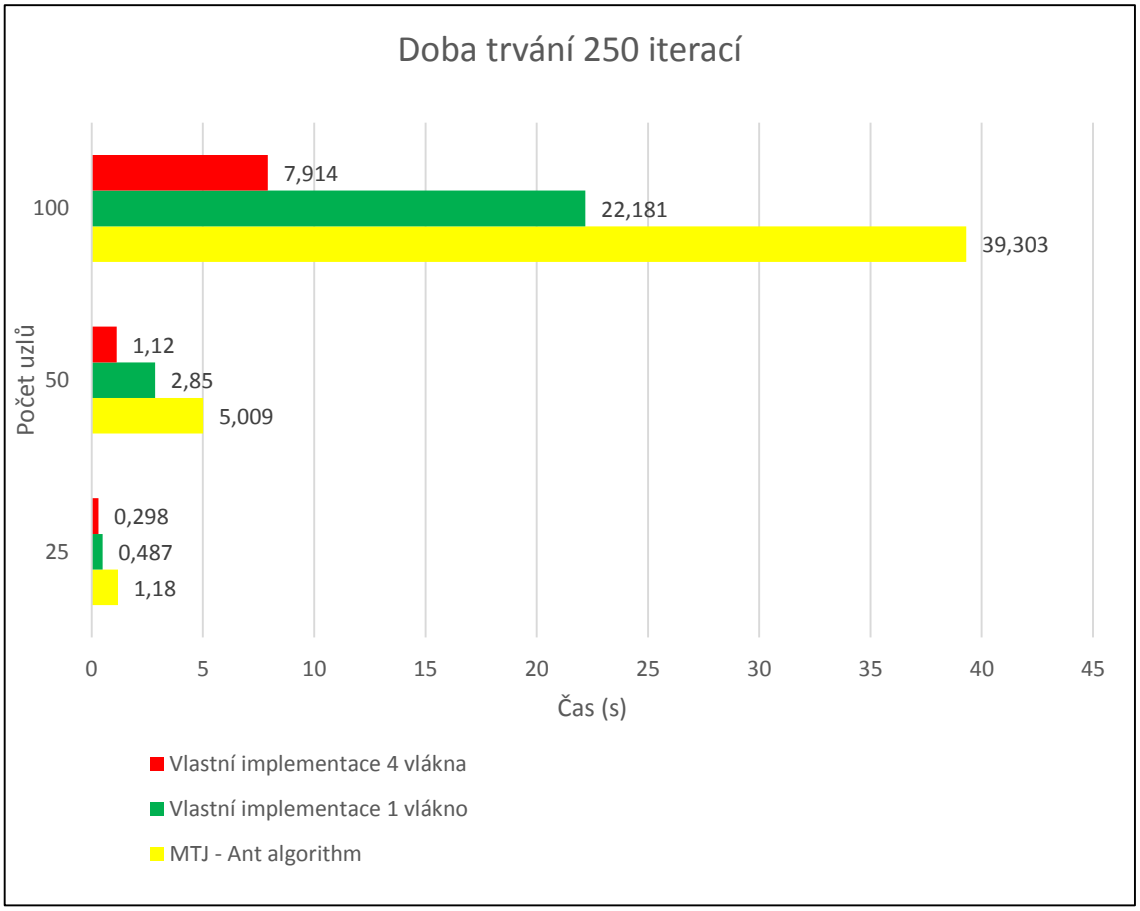
Obr. 28 Dosažená délka cesty srovnávaných implementací

Jak vidíme v grafu na obr. 28, délka nejlepší nalezené cesty se u aplikace běžící na jednom či více vláknech liší zcela zanedbatelně. To je logické, jelikož implementace algoritmu je u nich stejná, lišit se bude jen doba běhu algoritmu. Ve

všech případech však byla implementace MTJ Ant algorithm horší než mnou navržená implementace.

Doba trvání 250 iterací

Již tedy víme, že navržená implementace dosahuje lepších výsledů, než srovnávaná implementace, další otázkou je, jak rychle jich dosahuje. Pro srovnání byl proveden opakovaný běh aplikací na různém počtu uzlů. Výsledky jsou vidět v grafu na obr.29.



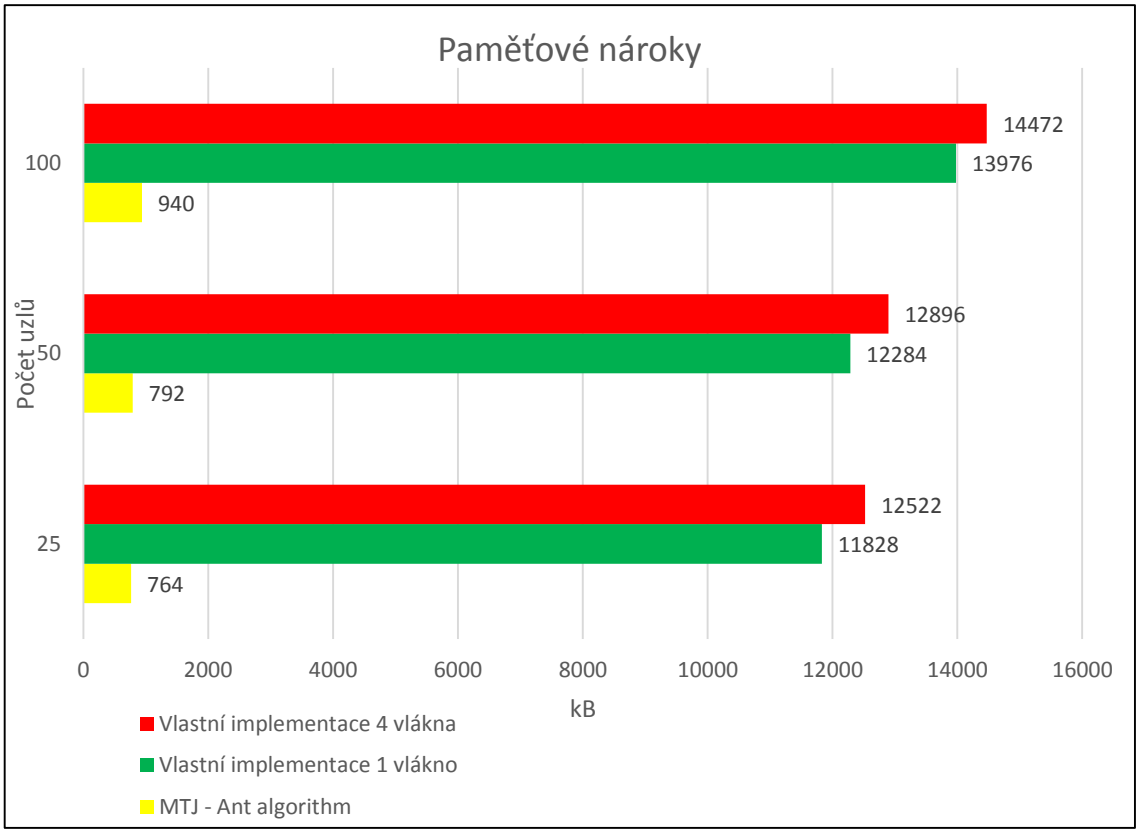
Obr. 29 Doba trvání vykonání 250 iterací srovnávaných algoritmů

Z výsledků testu vidíme, že úspora času je značná. Vlastní implementace je ve srovnání s MTJ Ant lgorithm téměř o polovinu rychlejší, a to při libovolném počtu uzlů, při 250 iteracích na 100 uzlech to tedy dělá již znatelný rozdíl 17 sekund.

Právě srovnání s použitím, jednoho vlákna, lépe vyjadřuje efektivitu samotného návrhu algoritmu. V praxi by však nebyl důvod proč nevyužít paralelizace, a při srovnání s délkou běhu aplikace běžící na 4 vláknech vidíme další podstatnou úsporu času. V nejnáročnějším uvedeném případě jde o rozdíl 31 sekund. Oproti běhu na jednom vlákně klesá čas běhu opět zhruba na třetinu. Více o postupech a výsledcích paralelizace algoritmu, se lze dočíst v kapitole 4.8.1

Paměťová nároky

V kapitole 3.7.7. se došlo k závěru, že paměťové nároky nejsou omezujícím faktorem implementace. Nicméně nároky na operační paměť jistě nelze zcela zanedbat. Paměťové nároky nejsou nijak závislé na počtu provedených iterací. Ve chvíli, kdy je vytvořen graf a algoritmus je spuštěn zůstávají rozdíly v podstatě zanedbatelné.



Obr. 30 Paměťové nároky srovnávaných implementací

Na grafu vidíme, že paměťové nároky vlastní implementace jsou znatelně vyšší. Je však dobré si povšimnout, nároky nerostou lineárně. Jejich převážnou většinu tvoří určitá konstantní část, která je přítomna i při běhu s nejmenším počtem uzlů.

Toto je pravděpodobně dáno použitou vývojářskou platformou a množstvím vyžadovaných knihoven, například pro práci s kolekcemi objektů, nebo pro matematické funkce. Oproti tomu MTJ Ant algorithm, který je implementován v jazyce C, je z tohoto hlediska podstatně skromnější.

Znovu je však třeba zopakovat, že cílem byla co nejefektivnější implementace, která bude dávat výsledky v co nejrychlejším čase, paměťové nároky zde jsou sekundárním hlediskem, i přesto však rozhodně nejsou omezujícím faktorem aplikace.

5 Závěr

Tato diplomová práce se zabývala problematikou algoritmu Ant Colony Optimalization. Tento algoritmus pocházející z rodiny algoritmů inspirovaných přírodou a lze jej aplikovat, na mnoho problémů z dnešního světa. Cílem této diplomové práce bylo navrhnout efektivní implementaci tohoto algoritmu. K tomu vedlo několik kroků, které zde shrneme.

Prvním krokem byla analýza současného stavu a nastudování informací o samotném algoritmu. V této fázi byla velmi poučná i část řešící historii algoritmu a ukazující reálné procesy, které probíhají v říši hmyzu. Málokdo by očekával, že počátky vývoje počítačového algoritmu mohou sahat do biologické laboratoře. V průběhu let vzniklo několik modifikací původního systému, ale všechny si zachovali jeho klíčové vlastnosti. Těmi je multiagentní mechanismus a komunikace těchto agentů prostřednictvím prostředí.

U každého systému je samozřejmě klíčová jeho efektivity a v tomto případě tomu není jinak. Efektivitu je ale třeba vždy poměřovat k jasně zadanému úkol. Různé problémy mají různou složitost a právě složitostí problémů se dnes zabývá samostatné vědecké odvětví. I v dnešním světě existuje spousta problémů, které jsou natolik složité, že pokud je chceme skutečně exaktně a nevyvratitelně vyřešit, narazíme na přílišnou časovou náročnost. A velmi často jsou to přitom problémy, jasně definované a lehce představitelné. Jako problém, který je nakonec implementovaným algoritmem řešen, byl vybrán problém obchodního cestujícího. A právě u něj jsem se při jeho vysvětlování osobám mimo informatické prostředí setkal s nepochopením jeho náročnosti. Lidé běžně posuzují problémy svýma očima. Pokud nakreslíme člověku na papír 50 teček a řekneme mu, aby je optimálně spojil, pravděpodobně se mu to optimálně nepovede, ale téměř jistě dojde k velmi solidnímu výsledku. Tak proč by to počítač nemohl, zvládnou rychleji a lépe?

Počítače to nakonec dokáží, ale musí jím k tomu pomoci vývojáři chytrým přístupem a neomezovat se na hrubou sílu. V tomto duchu byl následně navržen a implementován mnou vytvořený algoritmus vycházející z Ant Colony Optimalization.

Jedním z požadavků na výslednou aplikaci, byla možnost jejího využití při výuce optimalizačních algoritmů. Z tohoto důvodu bylo důsledně navrženo grafické uživatelské rozhraní. Toto rozhraní uživatelů umožňuje pohodlně zadávat parametry a zkoušet si tak jakým způsobem ony jsou schopny ovlivnit výsledky optimalizace. Dále je důležité, že takto vytvořený systém nefunguje jako černá skříňka, ale s uživatelem v průběhu řešení problému komunikuje a ukazuje mu postup řešení. Věřím, že tato aplikace bude užitečná, a některému ze studentů umožní seznámení se s tímto algoritmem a třeba jej i naláká k dalšímu studiu v této oblasti.

Poté co byla ověřena funkčnost algoritmu, čili jeho schopnost v rozumném čase dosahovat kvalitního řešení. Přikročilo se k jeho optimalizaci, s cílem co nejvíce zrychlit jeho chod. Toto se podařilo s výraznými výsledky, kdy rychlost řešení konkrétního zadání klesla z 3 minut na 9 sekund.

Vrcholem testování každého algoritmu řešícího nějaký problém, by měl být jeho souboj s některými jinými implementace, řešícími stejný problém a ideálně i jeho vítězné tažení. Aplikací řešící problém obchodního cestujícího je nepřeberné množství, a to ať už z rodiny Ant Colony Optimalization, anebo používající jinou metodu. Jejich exaktní a jednoznačné srovnání se však ukázalo jako větší problém, než se na první pohled jevilo. Je totiž třeba splnit několik podmínek, jako je možnost zadání stejných dat, práce ve stejné měrné jednotce a mnoho dalšího, aby se dala lepší efektivita jednoznačně pozorovat. Tyto podmínky nakonec byly splněny u jiné implementace Ant Colony Optimalization. Bylo tedy provedeno srovnání a z něj jako efektivnější vyšla vlastní navržená implementace. Bylo by však určitě zajímavé implementaci specializovat a naladit a určitý konkretizovanější problém a porovnávat ji s další takto odladěnou aplikací.

Co implementace jednoznačně potvrzuje je vhodnost algoritmu pro vícevláknové zpracování, kterým lze využít 100% potenciálu moderních procesorů a podstatně tak urychlit řešení problému.

Zásadní část času stráveného nad tvorbou této práce, byl čas strávený nad návrhem implementace a její realizací. Věřím, že takto strávený čas byl plodný a výstupem je funkční aplikace splňující očekávané požadavky. Cíle práce tedy lze považovat za splněné.

6 Literatura

Analyzing Application Performance by Using Profiling Tools. MICROSOFT, *MSDN* [online]. 2015 [cit. 2015-04-03]. Dostupné z: https://msdn.microsoft.com/en-us/library/z9z62c29.aspx

ARLOW, Jim a Ila NEUSTADT. *UML 2 a unifikovaný proces vývoje aplikací: objektově orientovaná analýza a návrh prakticky*. Vyd. 1. Brno: Computer Press, 2007, 567 s. ISBN 978-80-251-1503-9. [cit. 2015-01-22]

ČERNÝ, Jakub. Asymptotická časová složitost. *Algoritmy.eu* [online]. 2013 [cit. 2015-03-21]. Dostupné z: http://algoritmy.eu/zga/casova-slozitost/

DENEUBOURG, J. L., S. ARON, S GOSS a J. M. PASTEELS. The Self-Organizing Exploratory Pattern of the Argentine Ant. *Journal of lnsect Behavior*. 1990, 3(2). [cit. 2015-01-03].

DORIGO, Marco. *Ant colony optimization*. Cambridge: MIT Press, 2004, xiv, 305 s. ISBN 02-620-4219-3. [cit. 2014-010-10]

FOLTÝNEK, Tomáš, Milan ŠORM, Jana ANDRÝSKOVÁ, Michal HANZELKA, Pavel HALUZA, Miroslav PRACHAŘ a Jaromír ŠMÍD. *Teoretická informatika: eLearningová opora* [online]. [cit. 2015-05-21]. Dostupné z: https://is.mendelu.cz/auth/eknihovna/opory/index.pl?opora=623

HYNEK, Josef. *Genetické algoritmy a genetické programování*. 1. vyd. Praha: Grada, 2008, 182 s., [8] s. barev. obr. příl. Průvodce (Grada). ISBN 9788024726953. [cit. 2015-03-10]

KRÖMER, Pavel. *Optimalizace pomocí mravenčích kolonií: Inspirace, definice, aplikace a perspektivy* [online]. 2010 [cit. 2015-02-25]. Dostupné z: https://homel.vsb.cz/~kro080/mravenci.pdf

NAGEL, Christian. *Professional C#2008*. Indianapolis, IN: Wiley Pub., 2008, lvi, 1782 p. Wrox professional guides. ISBN 04-701-9137-6. [cit. 2015-02-25].

PUDLÁK, Pavel. O složitosti. *Pokroky matematiky, fyziky a astronomie*. Praha: Jednota českých matematiků a fyziků, 1988 : 20-34. ISSN 0032-2423. [cit. 2015-02-21]Dostupné také z: http://dml.cz/dmlcz/139598

RYBIČKA, Jiří a Petra ČAČKOVÁ. *Programovací techniky*. 1. vyd. V Brně: Mendelova univerzita, 2014, 258 s. ISBN 978-80-7509-136-9. [cit. 2015-02-25]

SHTOVBA Serhiy a ROTSHTEIN, Alexander. Ant Algorithms: Theory and Applications. *Programming and Computer Software* [online]. 2005, **31**(4): 287-300 [cit. 2015-12-1]. DOI: 10.1007/978-3-540-37368-1_9.

SCHMULLER, Joseph. *Sams teach yourself UML in 24 hours*. 3rd ed. Indianapolis, Ind.: Sams, 2004, xxi, 479 p. ISBN 06-723-2640-X. [cit. 2015-04-03]

Threads and Threading. MICROSOFT, *MSDN* [online] [online]. 2015 [cit. 2015-04-03]. Dostupné z: https://msdn.microsoft.com/en-s/library/aa720724%28v=vs.71%29.aspx

TOKSARI, M. D. Ant colony optimization for finding the global minimum. *Applied Mathematics and Computation* [online]. 2006, **176**(1): 308-316 [cit. 2015-01-11]. DOI: 10.1016/j.amc.2005.09.043.

VRÁNA, Ivan a Karel RICHTA. *Zásady a postupy zavádění podnikových informačních systémů: praktická příručka pro podnikové manažery*. 1. vyd. Praha: Grada, 2005, 187 s. Management v informační společnosti. ISBN 80-247-1103-6. [cit. 2015-01-03]

Přílohy

A Konfigurační soubor

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
      <sectionGroup name="applicationSettings" type= "System.Configuration.Ap-
           plicationSettingsGroup, System, Version=4.0.0.0, Culture=neutral, Pu-
           blicKeyToken=b77a5c561934e089" >
        <section name= "Ant_Colony_Optimalization_Parallel.Properties.Settings"
          type="System.Configuration.ClientSettingsSection, System,
          Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
          requirePermission="false" />
      </sectionGroup>
    </configSections>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
    </startup>
    <applicationSettings>
        <Ant_Colony_Optimalization_Parallel.Properties.Settings>
            <setting name="BestAntsCount" serializeAs="String">
                <value>10</value>
            </setting>
            <setting name="PheromoneMin" serializeAs="String">
                <value>0.1</value>
            </setting>
            <setting name="PheromoneMax" serializeAs="String">
                <value>5</value>
            </setting>
            <setting name="PheromoneDecrease" serializeAs="String">
                <value>0.85</value>
            </setting>
            <setting name="PheromoneIncrase" serializeAs="String">
                <value>5</value>
            </setting>
            <setting name="Alfa" serializeAs="String">
                <value>2</value>
            </setting>
            <setting name="Beta" serializeAs="String">
                <value>5</value>
            </setting>
            <setting name="OutputFile" serializeAs="String">
                <value>input.txt</value>
            </setting>
            <setting name="MaxThreadCount" serializeAs="String">
                <value>4</value>
            </setting>
            <setting name="IterationCount" serializeAs="String">
                <value>100</value>
            </setting>
            <setting name="StopIfNoChange" serializeAs="String">
                <value>1000</value>
            </setting>
            <setting name="InputFile" serializeAs="String">
                <value>output.txt</value>
            </setting>
```

```
        </Ant_Colony_Optimalization_Parallel.Properties.Settings>
    </applicationSettings>
    </configuration>
```

B Elektronické přílohy

1. Obě verze aplikace ve spustitelné podobě.
2. Vzorové vstupní soubory.
3. Zdrojové kódy implementace.