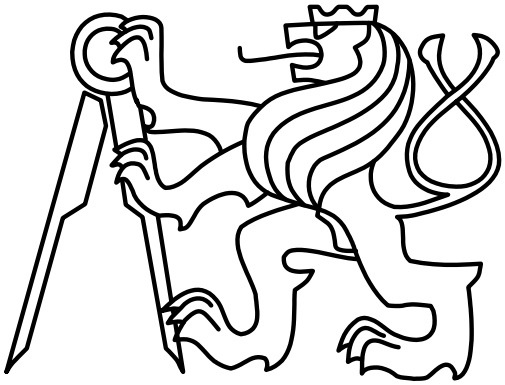
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA ELEKTROTECHNICKÁ

BAKALÁŘSKÁ PRÁCE



Martin Zaloga

Implementace optimalizačního algoritmu mravenčích kolonií (ACO)

Katedra kybernetiky
Vedoucí diplomové práce: **Ing. Richard Málek**

Praha, 2010

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Martin Z a l o g a

Studijní program: Elektrotechnika a informatika (bakalářský), strukturovaný

Obor: Kybernetika a měření

Název tématu: Implementace optimalizačního algoritmu mravenčích kolonií (ACO)

Pokyny pro vypracování:

1. Seznamte se s netradičními technikami řešení optimalizačních úloh pomocí přírodou inspirovaných algoritmů.
2. Implementujte algoritmus ACO (Ant Colony Optimization) a ověřte jeho funkčnost na problému splnitelnosti logických formulí (SAT).
3. Porovnejte výsledky algoritmu ACO s horolezeckým algoritmem (HC) (příp. GRASP).

Seznam odborné literatury: Dodá vedoucí práce.

Vedoucí bakalářské práce: Ing. Richard Málek

Platnost zadání: do konce zimního semestru 2010/2011

L.S.

prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry

doc. Ing. Boris Šimák, CSc.
děkan

V Praze dne 9. 12. 2009

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Praze dne 25.5.2010 .....

Podpis

Poděkování

Mé poděkování patří Ing. Richardu Málkovi, vedoucímu mé bakalářské práce, za poskytování rad, konzultací a pomoci, zejména při řešení problémů s implementací algoritmů.

Abstrakt

Tématem mé bakalářské práce je implementace algoritmu ACO inspirovaného chováním mravenčích kolonií, jeho aplikace na optimalizační problém splnitelnosti logických formulí (SAT) a následné porovnání např. s algoritmem GRASP.

Nejprve je v kapitole úvod popsán účel mé práce, proč se nechá inspirovat přírodou a několik příkladů možných inspirací.

V kapitole teoretický rozbor jsou nejprve uvedeny druhy optimalizačních problémů a podrobněji je zde popsán problém SAT. Následně jsou popsány přístupy k řešení optimalizačních algoritmů a několik konkrétních algoritmů spadajících do třídy informovaných metod prohledávání stavového prostoru.

V kapitole implementace je popsáno, jak jsem při implementací algoritmů postupoval a jak jsem naimplementoval algoritmus GRASP a ACO. U ACO algoritmu jsem zde rozebral několik důležitých tříd a jejich konkrétní způsob implementace.

Kapitola experimenty obsahuje testování, zpracování výsledků, porovnání algoritmů, a jejich zhodnocení.

Poslední kapitola závěr obsahuje celkové shrnutí a zhodnocení naimplementovaných a otestovaných algoritmů ACO a GRASP.

Abstract

The theme of my thesis is the implementation of the ACO algorithm inspired by the behavior of ant colonies and its application to the optimization problem of satisfiability of logical formulas (SAT) and comparison with e.g. GRASP algorithm.

The First chapter introduction describes the purpose of my work, why it is inspired by the nature and some examples of possible inspirations.

The chapter theoretical analysis specifies the types of optimization problems and describes more detail the SAT-problem in. Then it describes a way to solving optimization problems and some specific algorithms from class informed methods for state space search.

The chapter implementation describes how I have progressed in implementation of algorithms and how I have implemented GRASP and ACO algorithms. Several important classes of ACO algorithm and their specific way of implementation are described here.

The Chapter experiments contains description of experiments, results processing, comparison and evaluation algorithm.

Finally chapter conclusion includes a summary and evaluation of implemented and tested algorithms ACO and GRASP.

Obsah

Seznam obrázků

Seznam tabulek

1. Úvod

1.1 Účel mé práce

Účelem mé práce je řešit problém rozhodování splnitelnosti logických formulí (SAT). Řešení tohoto problému klasickými deterministickými metodami často není možné v přijatelném čase, neboť časová náročnost zde roste exponenciálně s velikostí problému. Z tohoto důvodu se často používají stochastické heuristické algoritmy, u kterých časová náročnost není tak vysoká, ale jejich použití je často za cenu méně přesného řešení (1).

Jedním z použitelných stochastických heuristických algoritmů je i algoritmus mravenčích kolonií (ACO), který je inspirován chováním skutečných mravenců. Právě tento algoritmus budu v mé práci implementovat na zmíněný problém SAT a poté ho porovnám s horolezeckým algoritmem (HC) případně s algoritmem GRASP, který též budu implementovat na problém SAT.

1.2 Co jsou stochastické heuristické optimalizační algoritmy?

Jak již z názvu vyplývá, jedná se o soubor algoritmů, jejichž úkolem je optimalizovat nějaký problém či proces. Jedním ze znaků těchto algoritmů je, že v nich hraje svou roli prvek náhody. Algoritmus zpravidla nepropočítává všechna možná řešení, ale nějakým způsobem náhodně vybere některé z možných, u kterého algoritmus předpokládá, že je výhodné. Pak může zjistit jeho skutečnou výhodnost vzhledem k celkovému řešení a porovnat ho s ostatními náhodně vybranými řešeními. Tyto algoritmy se s výhodou používají v případech, kdy je potřeba v rozumném čase nalézt nějaké přijatelné řešení, které nemusí být nutně to nejlepší (1).

1.3 Proč se nechat inspirovat přírodou?

Asi hlavním důvodem, proč se nechat inspirovat přírodou, je fakt, že algoritmy vyskytující se v přírodě jsou již dávno ověřené a máme jistotu, že fungují. Kdyby tomu tak nebylo, živí tvorové používající optimalizační algoritmy k procesům souvisejícím s přežitím, by s největší pravděpodobností dávno vyhynuli. Stejně tak přirozené fyzikální děje probíhající nějakým způsobem, by tímto způsobem určitě dlouho neprobíhaly, kdyby daný způsob nebyl optimální. Buď by takový děj přestal probíhat, nebo by začal probíhat jiným výhodnějším způsobem.

Další nespornou výhodou je zde množství možných inspirací, díky tomu, že v přírodě existuje nespočet přirozených procesů, kde je v podstatě implementován optimalizační algoritmus. Tyto procesy se dají nalézt třeba u všech živých tvorů či u mnoha fyzikálních procesů, jak již bylo odůvodněno výše.

1.4 Příklady procesů v přírodě, které implementují optimalizační algoritmus

Velice dobrým příkladem, může být boj o přežití, kdy kořist prchá před predátorem a náhodně volí svou cestu úniku. Na nějaké dlouhé přemýšlení o optimální cestě zde není čas, neboť by kořist byla sežrána predátorem. Mnohem lepší je pro kořist ihned zvolit některou z přijatelných cest, i když třeba ne tu nejlepší, a predátorovi se aspoň pokusit uprchnout.
Dalším dobrým příkladem mohou být procesy související s hledáním potravy a nalezení co možná nejkratší cesty k dopravě potravy na místo, kde bude bezpečně uschována před nepřáteli. Zde je potřeba podobně jako v předchozím případě nalézt přijatelnou cestu co nejrychleji, takže je zde také požadavek na co nejrychlejší nalezení přijatelné cesty. Tento příklad právě souvisí i s algoritmem mravenčích kolonií ACO.

Jiný příklad může být evoluční vývoj, kdy různé náhodné mutace a křížení živých tvorů mají za následek, že potomek má určité dispozice, které mu v daném prostředí umožňují či znemožňují přežít. Ti jedinci, kteří jsou nejvíce životaschopní, přežívají a ti ostatní vymírají. Zde je tedy optimalizována schopnost nějaké skupiny tvorů přežívat v daném prostředí.

Jako poslední příklad bych uvedl chování elektrického proudu. Elektrický proud má tu vlastnost, že se vždy snaží jít cestou nejmenšího odporu. Takže i zde je prováděna optimalizace, konkrétně elektrického odporu na cestě skrz elektrický obvod.

2 Teoretický rozbor

2.1 Druhy optimalizačních problémů

Základní rozdělení optimalizačních problémů může být na diskrétní a spojité. V mé bakalářské práci se řeší především diskrétní problém splnitelnosti logických formulí (SAT, Boolean Satisfiability Problem). Dále mezi diskrétní problémy patří například problém obchodního cestujícího (TSP, Travelling Salesman Problem), problém plánování (JSP, Job-shop Scheduling Problem), Problém obarvování grafu (GCP, Graph Coloring Problem) a další (2).

2.2 Problém splnitelnosti logických formulí SAT

Předmětem tohoto problému je nalézt logické ohodnocení logických proměnných zadané logické formule tak, aby po jejich dosazení vyšla formule jako splnitelná, či aspoň nalézt takové hodnoty logických proměnných, aby počet nesplněných klauzulí ve formuli byl minimální. Tento obecný problém je často transformován na tzv. 3SAT problém, kdy je formule v konjunktivní normální formě (CNF = konjunkce disjunkcí), kde každá klauzule obsahuje pouze tři logické proměnné či jejich negace (literály), jak je znázorněno v rovnici 2-1 (3). Pro tuto formu formulí jsou navrženy algoritmy v mé bakalářské práci.

$$f = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge (a_3 \vee b_3 \vee c_3) \wedge \dots$$

Rovnice 2-1 Tvar formule v 3SAT problému

2.3 Aplikace řešení problému SAT

Řešení problému SAT nachází uplatnění v mnoha oborech jako je například matematika, elektronika, plánování, konfigurace, obarvování grafů a další (3). Využití v matematice je zřejmé, při různých výpočtech, kde se pracuje s Booleovskými formulemi, se může hodit nalézt takové ohodnocení logických proměnných, aby formule byla splnitelná. V elektronice je možné na SAT problém převést například návrh desek plošných spojů, při známém požadovaném propojení jednotlivých součástek.

Velkou výhodou problému SAT je, že se na něj dá převést velké množství jiných problémů, patřících do skupiny tak zvaných NP problémů, neboť SAT problém spadá do třídy NP-úplných problémů (4).

2.4 Složitost problému SAT

Problém SAT se řadí mezi tak zvané NP-úplné problémy (NP = nedeterministicky polynomiální). O NP problémech víme, že je lze řešit v polynomiálně omezeném čase na nedeterministickém Turingově stroji. Což je fiktivní počítač, který umožňuje v každém kroku

rozvětvit výpočet na více větví, ve kterých se pak hledá řešení paralelně. Stejně tak by se mohlo uvažovat o počítači, který by v každém kroku výpočtu rozvinul správnou větev výpočtu. Srozumitelnější definice NP problémů zní, že je to množina problémů, u kterých není obecně známo, zda lze nalézt řešení v polynomiálním čase (tj. v čase úměrném některé mocnině velikosti problému).

NP-úplné problémy jsou pak všechny NP problémy na které jsou polynomiálně redukovatelné všechny ostatní NP problémy (5).

2.5 Prohledávání stavového prostoru

Prohledávání stavového prostoru je nástrojem k hledání řešení optimalizačních úloh. Při prohledávání se prochází stavy řešeného problému tak, aby se nalezl požadovaný stav, který splňuje zadaná kritéria. Požadovaným stavem může být v závislosti na řešeném problému například co nejkratší cesta, co nejnižší výrobní náklady, nalezení ohodnocení logických proměnných formule tak, aby měla co nejvíce splnitelných klauzulí a další.

Mnoho problémů má tak ohromné množství stavů, že jejich úplné procházení by i na nejvýkonnějších počítačích zabralo nepředstavitelnou dobu. Z tohoto důvodu se hojně používají optimalizační algoritmy, které neprocházejí všechny stavy, ale jen některé. Výběr stavů, které tyto optimalizační algoritmy projdou, může být, v závislosti na typu algoritmu, například na základě nějakého jednoduchého výpočtu, odhadu, náhody, kombinace předchozích a další.

Metody prohledávání stavového prostoru se mohou dělit na metody informované a neinformované.

Neinformované metody prohledávání stavového prostoru jsou ty, které nemají k dispozici žádné informace o prohledávaném stavovém prostoru, které by jim mohly nějakým způsobem pomoci nalézt cestu k cíli. Tyto metody jsou nuceny postupně procházet všechny stavy prostoru, dokud nenarazí na stav cílový. Jednotlivé algoritmy implementující neinformované prohledávání se od sebe liší pouze způsobem, kterým vybírají následující stavy.

Informované metody prohledávání stavového prostoru naopak mají nějaké informace o právě prohledávaném stavovém prostoru. Tyto informace jim umožňují odhadnout, jak daleko od aktuálního stavu se nachází stav cílový. Odhad zde reprezentuje heuristická funkce $h(n)$, na jejíž základě je algoritmus schopen odhadovat stavy, které by k cílovému stavu mohly vést rychleji (6).

2.6 Přístupy k řešení optimalizačních problémů

Jedno z možných dělení přístupů k řešení optimalizačních problémů je dělení na deterministické a heuristické algoritmy.

Deterministické algoritmy jsou ty, které vždy naleznou přesné a optimální řešení, které lze dokázat, a jsou vždy plně předvídatelné. Jejich velkou nevýhodou však je, že s lineárně rostoucím problémem, stoupá jejich časová náročnost exponenciálně.

Heuristické algoritmy využívají tzv. heuristické funkce, které určují volbu následujícího kroku, který bude učiněn při prohledávání stavového prostoru. Často se zde užívá různých odhadů výhodnosti daného kroku. Jedním z prvků heuristických algoritmů při výběru následujících kroků může být i stochastičnost, neboli náhoda. Takovéto algoritmy se pak nazývají stochastické (1).

2.7 Často používané heuristické algoritmy

Všechny níže popsané algoritmy se řadí do kategorie tzv. informovaných metod prohledávání. To znamená, že algoritmus využívá při svém chodu informaci o právě prohledávaném stavovém prostoru. Touto informací zpravidla bývá ohodnocení určující výhodnost následujícího možného kroku algoritmu. Co konkrétně znamená ohodnocení, záleží na typu problému, který řešíme. Například u problému nalezení nejkratší cesty má ohodnocení rozměr délky. U problému splnitelnosti logických formulí zas může mít rozměr počet nesplněných klauzulí.

2.7.1 Hladový algoritmus

Hladový algoritmus (GREEDY SEARCH) je jednou z nejjednodušších informovaných metod sloužících k rychlé konstrukci možného řešení. Algoritmus k rozhodování o následujícím kroku využívá pouze informací o výhodnosti následujících možných kroků. Ze všech následujících kroků pak vybere právě ten, který je za nejnižší cenu (lokální minimum). Nespornou výhodou tohoto algoritmu je, že dokáže zkonstruovat přijatelné řešení v rozumném čase, přičemž existuje možnost, že nalezené řešení bude i nejlepším možným řešením (globálním minimem). Nevýhodou je zde závislost výsledného řešení na volbě počátečního bodu (7).

2.7.2 Horolezecký algoritmus

Horolezecký algoritmus (HC, Hill-Climbing) – Jednoduchý algoritmus inspirovaný chováním horolezců při zdolávání hor. Tento algoritmus obvykle modifikuje nějaké existující možné řešení za účelem jeho zlepšení. Algoritmus jako následující krok volí vždy ten aktuálně nejvýhodnější, ale s podmínkou, že nesmí zhoršit aktuální výsledné řešení. Takovýto algoritmus je velice jednoduchý a snadno pochopitelný, ale bohužel může uváznout v lokálním minimu (lokálně nejlepším řešení), ze kterého se nedostane a nemusí tak nalézt globální minimum (nejlepší možné řešení).

Často se používá modifikace, kdy je tento algoritmus opakovaně restartován a spouštěn s náhodně vybraného počátečního bodu, přičemž se uchovává nejlepší dosažené řešení. Tímto je v podstatě zvyšována pravděpodobnost nalezení globálního minima (8).

2.7.3 GRASP algoritmus

GRASP algoritmus (Greedy Randomized Adaptive Search Procedure) je optimalizační algoritmus, který postupuje ve dvou fázích. V první fázi je vždy zkonstruováno počáteční řešení, kde je každý krok vybírán, podobně jako u hladového řešení, ten aktuálně nejvýhodnější. V druhé fázi je nalezené počáteční řešení dále optimalizováno některou z metod lokálního prohledávání, například algoritmem HC.

Pro zvýšení pravděpodobnosti nalezení globálního minima je zde, podobně jako u HC algoritmu, vhodné algoritmus opakovaně restartovat a spouštět z náhodně vybraného počátečního bodu (9).

2.7.4 Algoritmus mravenčích kolonií ACO

Algoritmus mravenčích kolonií (ACO, Ant Colony System) je inspirován chováním mravenců při hledání nejkratší cesty od hnízda k potravě. Skuteční mravenci mezi sebou komunikují pomocí feromonů, které zanechávají na cestách, kterými prošli. Množství feromonů na dané cestě je pak úměrné počtu mravenců, kteří přes ni prošli a nepřímo úměrné době, od kdy feromon na cestě leží. Každý mravenec umí tyto feromony rozpoznávat a s jejich pomocí se rozhodovat, zda danou cestu použije či nikoliv.

Nejprve je třeba dodat, že libovolný problém řešený tímto algoritmem je dobré nějakým způsobem transformovat do grafu složeného z uzlů a hran. Mravenci pak grafem procházejí a na hranách, kterými prošli, zanechávají feromon, obvykle úměrný ceně cesty, kterou od počátku k cíli urazili. Feromon pak ostatní mravenci detekují a podle jeho množství se dále rozhodují, kterou hranu použijí. Takto je využívána zkušenost, získávaná v průběhu výpočtu algoritmu. Důležitou vlastností algoritmu je, že v jeho průběhu je feromon odpařován z hran, aby se zamezilo předčasné konvergenci výsledného řešení k lokálnímu minimu. Dále v algoritmu hraje důležitou roli při rozhodování cena dostupné hrany.

Důležitost informace jak o ceně dostupné hrany, tak o množství feromonu na hraně, se určuje nastavením parametrů algoritmu (10).

3 Popis řešení

3.1 Obecný popis postupu při implementaci algoritmů

Při implementaci algoritmů jsem vycházel z myšlenky obecného a znovupoužitelného kódu algoritmu. Důvodem, proč jsem takto postupoval, bylo, že algoritmy je nyní možné implementovat na libovolný problém pouhým dodefinováním těch částí kódu, které jsou problémově závislé. Což jeden z hlavních požadavků projektu SEAGE, kterého je má práce součástí.

K tomu, abych splnil požadavek obecnosti jádra algoritmů, jsem využil předností objektového programování – dědičností a rozhraní.

Myšlenka znuvupoužitelnosti kódu je patrná ze zjednodušeného UML class diagramu, viz obr. 3-2, u popisu implementace algoritmu ACO, který je zároveň hlavním předmětem mé práce.

Ačkoliv bylo předmětem mé bakalářské práce především aplikovat optimalizační algoritmy na problém SAT, tak nejprve jsem vždy algoritmy aplikoval na problém TSP, který je lépe představitelný a graficky znázornitelný, a tudíž se na něm jednodušeji ověřuje funkčnost daného algoritmu. Teprve poté jsem aplikoval algoritmy i na problém SAT.

To že jsem algoritmy implementoval ve dvou různých problémech, mi nejen pomohlo tyto algoritmy odladit, ale zároveň jsem si mohl lépe uvědomit, které části algoritmů jsou skutečně obecnými a tím pádem mohou být v obecné, problémově nezávislé, části kódu.

3.2 Restartovaný HC algoritmus

V mé implementaci je použita stochastická verze tohoto algoritmu, kde množina následujících možných kroků, ze kterých může algoritmus vybírat, je generována náhodně.

Počáteční řešení, které tomuto algoritmu předám, se dá určit vstupními parametry a může to být buď zcela náhodné řešení, či hladové řešení. V případě hladového počátečního řešení, předaného dále tomuto HC algoritmu, se ve výsledku vlastně jedná o možnou variantu algoritmu GRASP. Hladové počáteční řešení se zde po každém restartu vytváří z náhodného počátečního bodu.

Dalšími vstupními parametry se u tohoto algoritmu ještě předává počet iterací a počet restartů.

Při implementaci tohoto algoritmu jsem využil rozhraní, které jsem měl zpočátku nadefinované jako požadované. To jsem dále naimplementoval v problémech TSP a poté SAT.

3.3 ACO algoritmus

Tento algoritmus je sám o sobě stochastický. Při průchodu mravence grafem volí mravenec následující hranu náhodně ale s tím, že „slibné“ hrany jsou zvýhodněny tak, aby pravděpodobnost jejich výběru byla vyšší oproti méně zajímavým. Slovo „slibné“ zde znamená kombinaci vlastností ceny hrany L a množství feromonu na hraně Q. Čím je hrana za nižší cenu L a čím více je na ní uloženo feromonu Q, tím více je pro mravence „slibná“.

V mé implementaci je zde několik vstupních parametrů, kterými lze ovlivňovat chování

mravenců. Mezi ně patří: Množství pokládaného feromonu q, rychlost odpařování e, počáteční množství feromonu na každé hraně d, parametry α a β, kterými se volí váha feromonu a ceny hrany při výběru následující hrany mravencem, počet mravenců a a počet iterací i.

Zde jsem při implementaci postupoval trochu odlišně od předchozího algoritmu. Nejprve jsem celý algoritmus naimplementoval v problémově závislé části na TSP problému, kde jsem ho také odladil. Teprve po té jsem oddělil obecné části kódu od problémově závislých a následně doimplementoval v problémově závislé části SAT.

Níže je zjednodušený UML class diagram na obr. 3-2, vystihující oddělení obecné a problémově závislé části algoritmu za použití dědičnosti. V jednotlivých třídách jsou ve zjednodušeném diagramu uvedeny pouze konstruktory a ty metody, které byly implementovány v dědících třídách.

3.3.1 Obecná část ACO algoritmu a její třídy

Níže je uveden stručný popis hlavních obecných tříd algoritmu, který definuje, co daná třída reprezentuje a co je jejím hlavním úkolem.

AntColony
- Reprezentuje hlavní část algoritmu.
- Zodpovídá za provádění a vyhodnocování jednotlivých iterací algoritmu.

Ant
- Reprezentuje jednotlivé mravence.
- Zodpovídá za průchod mravence grafem a zanechávání feromonu na hranách grafu.

AntBrain
- Reprezentuje mozek mravence.
- Zodpovídá hlavně za výběr jednotlivých hran mravencem při průchodu skrz graf.

Graph
- Reprezentuje problém zobecněný na graf složený z uzlů a hran.
- Zodpovídá za sestavení grafu.

Edge
- Reprezentuje hranu v grafu.
- Zodpovídá za spojení dvou uzlů a obsahuje důležitou informaci o ceně hrany.

Node
- Reprezentuje uzel grafu.
- Zodpovídá za uchování informace o spojení s ostatními uzly, pomocí hran.

3.3.2 Problémově závislá část ACO algoritmu a její třídy

Níže je stručně popsáno, co konkrétně a jakým způsobem je naimplementováno v problémově závislých třídách.

SatAnt

- Tato třída pouze přepisuje metodu updatePosition() pro aktualizaci pozice mravence. Tato metoda, jež je za aktualizaci pozice mravence zodpovědná, je již v rodičovské třídě obecně naimplementována. Přesto je v této třídě přepsána, protože díky zvolené konstrukci grafu, je možné, aby zde měla efektivnější tvar.

SatAntBrain

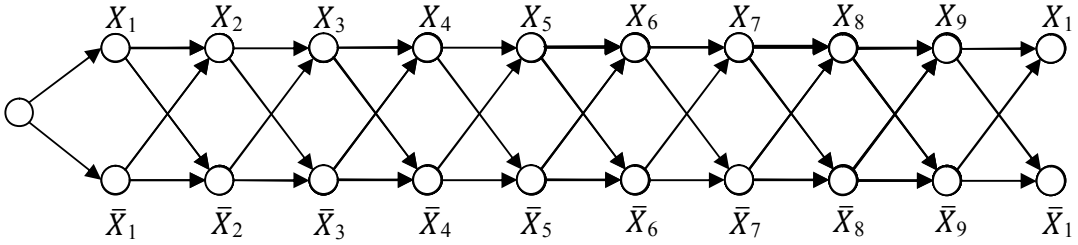
- Tato třída implementuje tři problémově závislé metody.
- Metoda getAvailableEdges() vrací všechny dostupné hrany aktuálního uzlu.
- Metoda pathPrice() vrací celkovou cenu cesty, reprezentovanou počtem nesplněných klauzulí, která se získá dosazením řešení do formule. Řešení je zde reprezentováno ohodnocením logických proměnných formule.
- Metoda selectNextEdge() vrací následující hranu vybranou mravencem, která je vybrána na základě pravděpodobnosti jejího výběru P a náhody. Pravděpodobnost výběru i-té hrany P_i je dána vzorcem 3.1, kde Q_i je množství feromonu na i-té hraně, L_i je cena i-té hrany a parametry α a β udávají váhy těchto informací. Tato metoda je také již obecně implementována v rodičovské obecné třídě, ale je zde přepsána na efektivnější tvar.

$$P_i = \frac{Q_i^{\alpha} \cdot \left(\frac{1}{L_i}\right)^{\beta}}{\Sigma_j \left(Q_j^{\alpha} \cdot \left(\frac{1}{L_j}\right)^{\beta}\right)}$$

Rovnice 3-1 Počítání pravděpodobnosti

SatGraph

- Tato třída implementuje metodu pro výpis feromonu na hranách printPheromone().
- Důležitým úkolem této třídy je převést kontrérní problém splnitelnosti logických formulí na obecný graf, kde každý uzel znamená zvolenou logickou hodnotu logické proměnné výsledného řešení. Cesta skrz graf pak tedy jednoznačně určuje výsledné řešení. Konstrukce grafu je znázorněna na obr. 3-1.

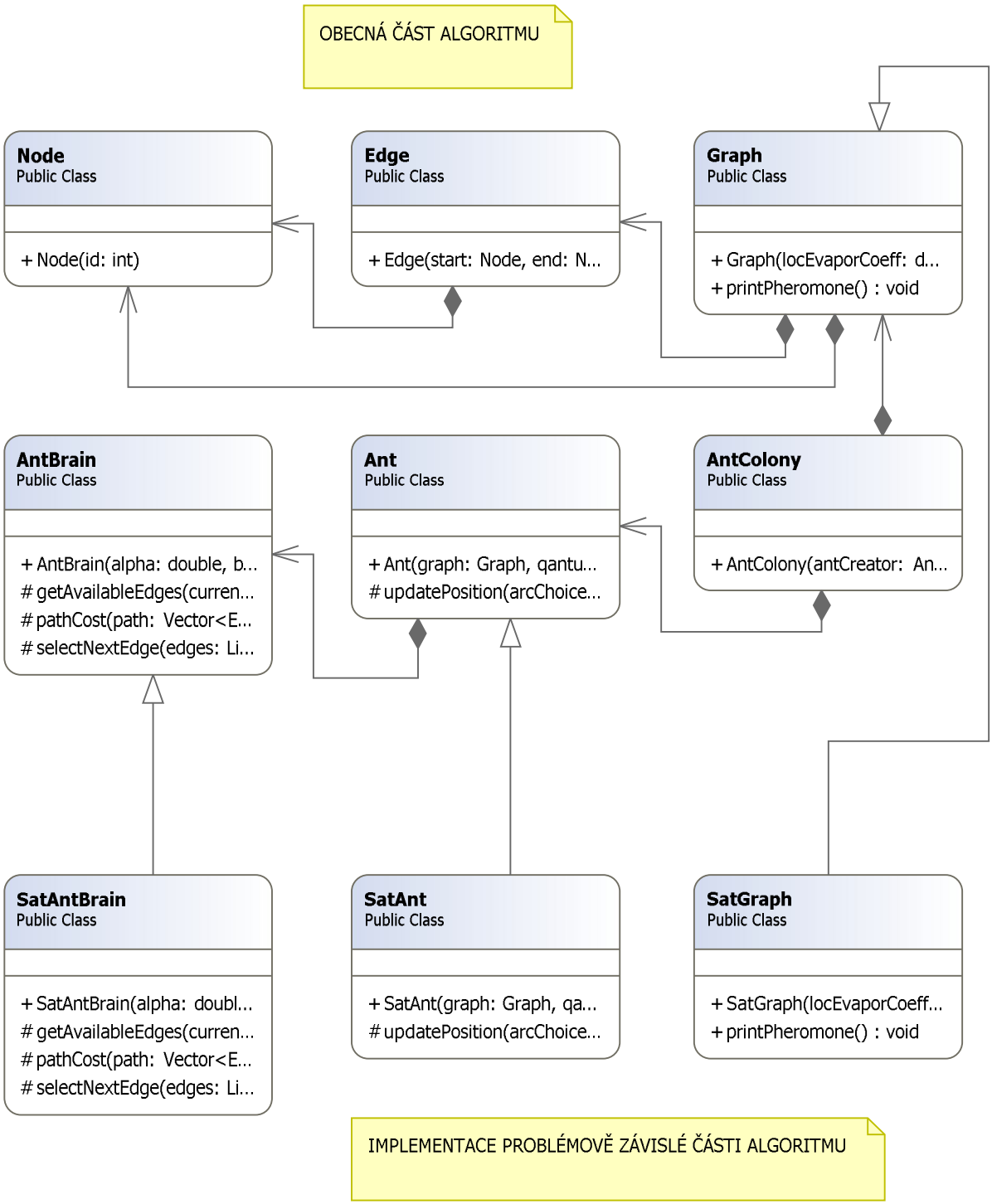


Obrázek 3-1 Konstrukce grafu, jak by vypadal pro formuli o deseti logických proměnných

Z grafu na obr. 3-1 je možné vidět, že uzly reprezentují jednotlivé logické proměnné formule X_n a jejich negace. Z každého uzlu, kromě posledního páru uzlů, vedou pouze dvě jednosměrné hrany, reprezentující volbu logické hodnoty následující logické proměnné X_{n+1}.

Velkou výhodou takovéto konstrukce grafu je, že při průchodu mravence grafem, není potřeba, aby mravenec při každém výběru hrany kontroloval, zda hrana nevede do již navštíveného uzlu. Díky tomu je zde jednodušší metoda selectNextEdge() z třídy SatAntBrain, než je obecná implementace v obecné části algoritmu. Kdybych použil úplný graf, kde je každý uzel s každým spojen hranou, tak by metoda selectNextEdge() z třídy SatAntBrain musela obsahovat kontrolu již navštívených uzlů a též by ještě musela obsahovat kontrolu, zda hrana

nevede do uzlu znamenající negaci aktuální logické proměnné, neboť by se tím zvolily 2 logické hodnoty pro jeden a tu samou logickou proměnnou formule.



Obrázek 3-2 Zjednodušený UML class diagram ACO algoritmu

4 Experimenty

4.1 Obecný postup při experimentování

U obou algoritmů jsem postupoval tak, že jsem si nejprve „ručně“ vyzkoušel, jak se algoritmus chová pro různá nastavení na daných datech. Dále jsem si určil, kdy se algoritmus chová rozumně, tj. kdy dává dobré výsledky. Na základě toho jsem pak určil různé možnosti nastavení vstupních parametrů, blízké tomu kdy se algoritmus choval rozumně a vytvořil testy, které otestovaly všechny požadované možnosti. Výstupem testů jsou tabulky, kde jsou pro každou konfiguraci vstupních parametrů zobrazeny výsledky a doby běhu algoritmů.

Po nastavení parametrů po optimální chování algoritmů jsem vytvořil testy pro srovnání restartovaného HC a ACO algoritmu z hlediska časů běhu t a dosažených výsledků f, v podobě počtu nesplněných klauzulí.

U restartovaného horolezeckého algoritmu bylo potřeba stanovit, pro jaké počáteční řešení dává i ve výsledku lepší řešení a určit jak se chová při změnách počtu nastavených iterací i a restartů r.

U algoritmu ACO to bylo o něco složitější, protože nastavitelných vstupních parametrů je v mé implementaci celkem sedm. Z čehož vyplývá, že i kdybych testoval pro každý parametr jen dvě jeho hodnoty a chtěl otestovat všechny možnosti nastavení, které takto vzniknou, tak řádků tabulky shrnující takovéto testování by muselo být 2^7 (to je 128). Jednak je to více řádků, než kolik se vejde na jednu stránku při rozumné velikosti písma a za druhé jsem některé parametry chtěl otestovat na více hodnot. Z tohoto důvodu bylo testování ACO potřeba rozdělit do dvou částí a tedy i tabulek.

V první části jsem vždy testoval nastavení parametrů α, β, množství počátečního feromonu d na hranách, množství pokládaného feromonu q a rychlost odpařování e. To vše při konstantním počtu mravenců a a konstantním počtu iterací i.

V druhé části jsem testoval chování algoritmu při změnách nastavení počtů iterací i a počtu mravenců a při konstantním optimálním nastavení parametrů zjištěných v první části.

4.2 Tabulky, grafy a výsledky experimentů

U některých testů jsem ve skutečnosti testoval více možných variant nastavení vstupních parametrů, ale tabulky by byly příliš velké. Z tohoto důvodu jsem sem umístil jen zajímavé části těchto tabulek.

Dále bych ještě uvedl, že jsem při měření časů potřebných pro výpočet algoritmů měřil pouze dobu mezi spuštěním a koncem výpočtu, nikoliv i dobu potřebnou pro vytváření instancí grafů a podobně.

Každé nastavení parametrů jsem testoval 10× a do výsledků jsem poté uvedl aritmetický průměr z těchto testů.

K testování byla použita data z webu http://www.satlib.org/, která jsou určená pro testování a porovnávání algoritmů na problému SAT.

4.2.1 Testování restartovaného HC algoritmu

U restartovaného HC algoritmu jsem otestoval každou variantu nastavení parametrů 10× a výsledek pak spočítal jako aritmetický průměr. Viz tabulka 4.1. Algoritmus jsem testoval na datech uf100/uf100-0100.cnf

Testování restartovaného HC algoritmu pro různá počáteční řešení, počty restartování *r* a počty iterací *i* na formuli o velikosti 100 literálů a 430 klauzulí				
Počáteční řešení	Počet restartů *r* [-]	Počet iterací *i* [-]	Průměrně dosahovaný výsledek *f* [nesplněné klauzule]	Průměrná doba běhu algoritmu *t* [s]
náhodné	5	10	**45,1**	0,0085
náhodné	5	100	**42,8**	0,0073
náhodné	5	1000	**44,5**	0,0072
náhodné	10	10	**42,0**	0,0136
náhodné	10	100	**42,8**	0,0135
náhodné	10	1000	**41,8**	0,0160
náhodné	50	10	**38,7**	0,0635
náhodné	50	100	**39,1**	0,0628
náhodné	50	1000	**37,2**	0,0663
náhodné	100	10	**36,9**	0,1230
náhodné	100	100	**36,6**	0,1234
náhodné	100	1000	**36,1**	0,1312
hladové	5	10	**11,1**	0,0092
hladové	5	100	**12,0**	0,0092
hladové	5	1000	**11,7**	0,0090
hladové	10	10	**11,6**	0,0171
hladové	10	100	**11,3**	0,0177
hladové	10	1000	**10,9**	0,0165
hladové	50	10	**9,8**	0,0747
hladové	50	100	**10,0**	0,0750
hladové	50	1000	**9,8**	0,0751
hladové	100	10	**9,3**	0,1488
hladové	100	100	**9,3**	0,1528
hladové	100	1000	**9,2**	0,1488

Tabulka 4-1 Testování restartovaného HC algoritmu na datech uf100/uf100-0100.cnf

V tabulce 4-1 je na první pohled patrné, že pro počáteční hladové řešení dává algoritmus lepší řešení i v konečném výsledku *f*. S Tímto budu již počítat i u srovnávání algoritmů a nebudu již toto znovu testovat.

Co se počtů iterací *i* týče, tak je zde z měřených dob *t* znát, že se algoritmus celkem rychle dostane do lokálního minima, kde se již dále výsledky *f* nezlepšuje a je restartován dříve než za nastavený maximální počet iterací *i*. Z toho vyplývá, že maximální počet iterací *i* by v algoritmu vlastně ani nemusel být, což jsem však při implementaci nemusel nepředpokládat. Na druhou stranu tento parametr ničemu nepřekáží a stačí ho nastavit dostatečně velký, aby algoritmus vždy došel do minima, ať už lokálního, či globálního.

U parametru počtu restartů *r* je celkem zřejmé, že čím je vyšší, tím lepší jsou i výsledky *f*. Celkově tedy z tabulky plyne, že optimální nastavení restartovaného HC algoritmu je pro hladové počáteční řešení, což je vlastně ve výsledku GRASP algoritmus. Dále že počet iterací *i* je dobré nastavit na dostatečně vysoké číslo (například 1000). Nakonec jediné co se zde vyplatí měnit a sledovat je počet restartů *r*.

Při srovnávacích testech algoritmů budu tedy využívat algoritmus GRASP, kde budu pouze měnit počet restartů *r*.

4.2.2 Testování na datech uf75/uf75-01.cnf

Data uf75/uf75-01.cnf jsou určena k testování algoritmů na formuli o velikosti 75 literálů a 325 klauzulí.

Testování ACO algoritmu - první fáze

V první fázi testování jsem zde nastavil konstantní počet mravenců $a = 100$ a konstantní počet iterací $i = 500$.

Tabulka testování ACO algoritmu pro různá nastavení parametrů α, β, d, q a e na formuli o velikosti 75 literálů a 325 klauzulí

Parametr α [-]	Parametr β [-]	Počáteční feromon na každé hraně d [-]	Množství pokládaného feromonu q [-]	Odpařování feromonu e [-]	Výsledek f [nesplněné klauzule]	Doba běhu algoritmu t [s]
1	1	0,1	0,2	0,2	**6,0**	4,136
1	1	0,1	0,2	0,5	**7,2**	4,170
1	1	0,1	0,5	0,2	**6,1**	4,217
1	1	0,1	0,5	0,5	**9,3**	4,140
1	1	0,1	1,0	0,2	**5,7**	4,157
1	1	0,1	1,0	0,5	**8,9**	4,148
1	1	1,0	0,2	0,2	**5,3**	4,141
1	1	1,0	0,2	0,5	**5,7**	4,167
1	1	1,0	0,5	0,2	**5,1**	4,211
1	1	1,0	0,5	0,5	**9,9**	4,151
1	1	1,0	1,0	0,2	**6,4**	4,126
1	1	1,0	1,0	0,5	**4,5**	4,146
1	2	0,1	0,2	0,2	**6,2**	4,191
1	2	0,1	0,2	0,5	**9,6**	4,201
1	2	0,1	0,5	0,2	**7,8**	4,186
1	2	0,1	0,5	0,5	**9,0**	4,280
1	2	0,1	1,0	0,2	**8,4**	4,214
1	2	0,1	1,0	0,5	**7,5**	4,182
1	2	1,0	0,2	0,2	**7,2**	4,201
1	2	1,0	0,2	0,5	**9,1**	4,131
1	2	1,0	0,5	0,2	**6,3**	4,169
1	2	1,0	0,5	0,5	**8,9**	4,216
1	2	1,0	1,0	0,2	**5,8**	4,191
1	2	1,0	1,0	0,5	**7,7**	4,144

Tabulka 4-2 První fáze testování ACO algoritmu na datech uf75/uf75-01.cnf

Lepší výsledky *f* podle tabulky 4-2 vycházejí, pro nastavení parametrů $\alpha = \beta = 1$. Lepší výsledky *f* také na první pohled vycházejí pro počáteční hodnotu feromonu $d = 1$. Za množství odpařovaného feromonu bych zvolil hodnotu $e = 0{,}2$, neboť pro tuto hodnotu je většinou výsledek *f* lepší. Celkový výsledek *f* vychází dle tabulky lépe pro hodnotu množství pokládaného feromonu $q = 0{,}2$.

Testování ACO algoritmu - druhá fáze

Zde použiji zjištěné optimální parametry z první fáze testování.

Tabulka testování ACO algoritmu pro různé počty mravenců *a* a různé počty iterací *i* na formuli o velikosti 75 literálů a 325 klauzulí			
Počet mravenců *a* [-]	Počet iterací *i* [-]	Výsledek *f* [nesplněné klauzule]	Doba běhu algoritmu *t* [s]
100	100	**14,4**	0,853
100	250	**6,9**	1,967
100	500	**5,0**	4,080
100	1000	**5,7**	8,199
250	100	**11,6**	2,395
250	250	**5,1**	5,270
250	500	**4,4**	10,532
250	1000	**4,0**	21,118
500	100	**9,6**	4,244
500	250	**4,3**	10,877
500	500	**5,1**	21,250
500	1000	**4,8**	42,596
1000	100	**10,3**	9,549
1000	250	**4,2**	22,743
1000	500	**4,8**	43,807
1000	1000	**3,5**	86,233

Tabulka 4-3 Druhá fáze testování ACO algoritmu na datech uf75/uf75-01.cnf

Z tabulky 4-3 lze vidět, že výsledné řešení *f* se nejprve při zvyšování počtu iterací *i* zlepší hodně, ale pak už se zlepšuje jen pozvolna.

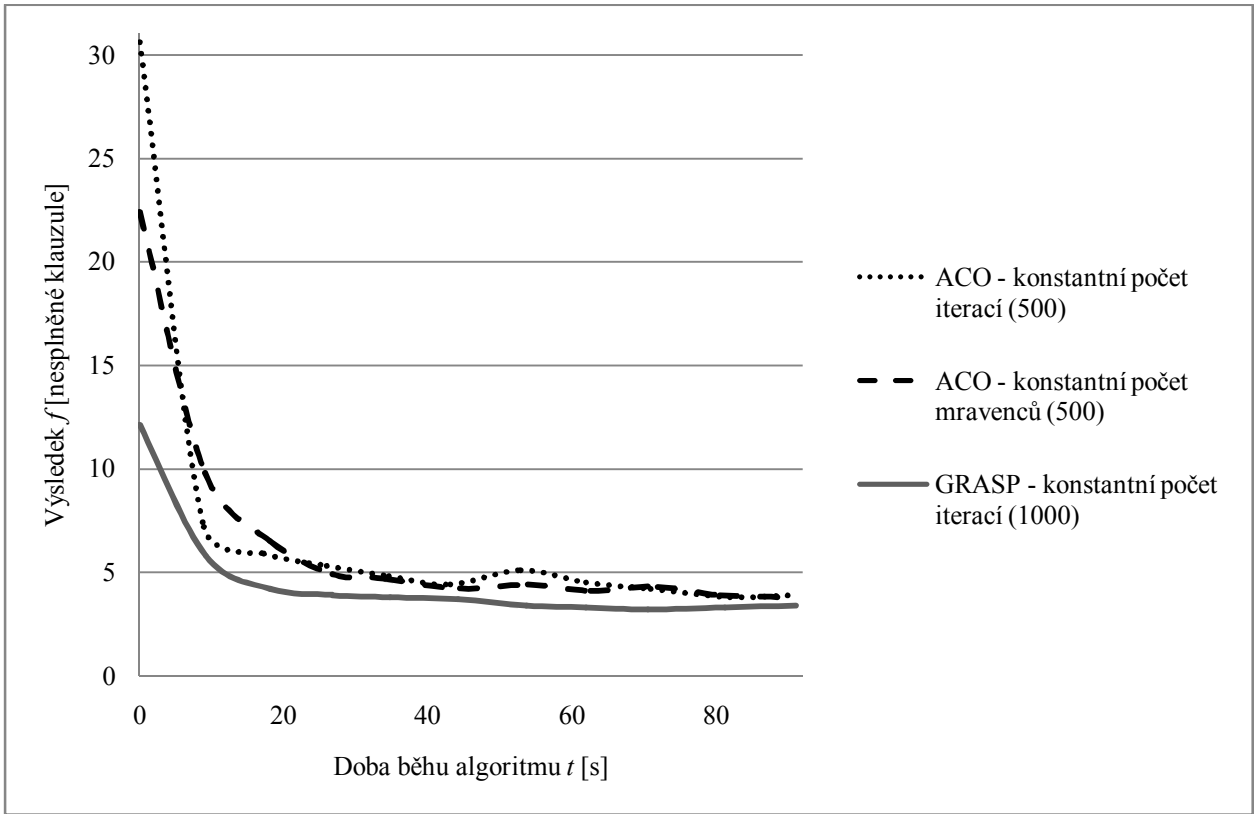
Protože však není příliš zřetelné, na kterém z těchto parametrů závisí výsledky *f* více, rozhodl jsem se, že při srovnávání ACO algoritmu s algoritmem GRASP budu testovat jak variantu s konstantním počtem mravenců $a = c$ a proměnným počtem iterací *i*, tak variantu s konstantním počtem iterací $i = c$ a proměnným počtem mravenců *a*. Za počet mravenců *a* i počet iterací *i* bych zde volil hodnoty $a = 500$ a $i = 500$.

Srovnání algoritmů ACO a GRASP

Při srovnávání algoritmů ACO a GRASP jsem u nich nejprve nastavil zjištěné parametry pro jejich optimální chování a dále jsem měnil jen parametry jako počet mravenců *a* a počet iterací *i* u ACO a počet restartů *r* u GRASP.

Srovnávací tabulka pro srovnání algoritmů ACO a GRASP na formuli velké 75 literálů a 325 klauzulí								
ACO - konstantní počet iterací (i = 500)			ACO - konstantní počet mravenců (a = 500)			GRASP - konstantní počet iterací (i = 1000)		
Počet mravenců a [-]	Výsledek f [nesplněné klauzule]	Doba běhu algoritmu t [s]	Počet iterací i [-]	Výsledek f [nesplněné klauzule]	Doba běhu algoritmu t [s]	Počet restartů r [-]	Výsledek f [nesplněné klauzule]	Doba běhu algoritmu t [s]
1	**30,6**	0,124	1	**22,4**	0,117	1	**12,1**	0,122
100	**7,2**	8,949	100	**9,8**	9,128	800	**5,8**	9,212
200	**5,9**	17,311	200	**6,5**	18,153	1600	**4,2**	18,196
300	**5,3**	26,036	300	**4,9**	26,82	2400	**3,9**	26,839
400	**4,8**	34,607	400	**4,7**	34,061	3200	**3,8**	34,776
500	**4,4**	43,154	500	**4,2**	44,488	4000	**3,7**	44,251
600	**5,1**	52,904	600	**4,4**	53,904	4800	**3,4**	53,72
700	**4,5**	61,928	700	**4,1**	62,541	5600	**3,3**	61,933
800	**4,2**	70,935	800	**4,3**	71,927	6400	**3,2**	70,595
900	**3,8**	81,893	900	**3,9**	80,51	7200	**3,3**	81,226
1000	**3,9**	90,704	1000	**3,8**	89,969	8000	**3,4**	91,085

Tabulka 4-4 Srovnání ACO a GRASP algoritmu na datech uf75/uf75-01.cnf



Obrázek 4-1 Graf srovnání ACO a GRASP algoritmu na datech uf75/uf75-01.cnf

Podle výsledků f z tabulky 4-4 neplyne, že by v tomto případě byl ACO algoritmus lepší než GRASP algoritmus. Důvodů proč toto testování takto dopadlo, může být několik. Je možné, že kdybych nechal algoritmy běžet déle, tak výsledky f ACO algoritmu by se po nějaké době ještě

zlepšily. Další možností je, že jsem svými testy neodhalil skutečně optimální nastavení vstupních parametrů a algoritmus ACO se tedy nechoval optimálně. Jinou další možností by mohlo také být to, že pro vstupní data, která nejsou příliš obsáhlá, se ACO algoritmus prostě nehodí tak jako algoritmus GRASP. Ať je jakkoliv, rozhodně ACO algoritmus nedává při dostatečně dlouhém běhu horší výsledky *f* než GRASP algoritmus.

Zde lze pozorovat, že u ACO algoritmu se výsledek *f* do určité míry zlepšuje se zvyšováním počtu mravenců *a* i počtu iterací *i*, kdežto GRASP algoritmus dospěje celkem rychle k dobrému výsledku *f*, ale pak už se tolik nezlepšuje.

Testů, které byly podobné tomuto, jsem na těchto datech vyzkoušel více, abych případně eliminoval možnost, že kvůli náhodě zde vychází GRASP algoritmus jako lepší než ACO algoritmus. Testy však dopadly velice podobně jako právě tento.

Z grafu na obr. 4-1 je možné vidět vyrovnanost ACO a GRASP algoritmu. Rychleji však celkem dobré řešení nalezne algoritmus GRASP a při ponechání delší doby běhu algoritmů jsou zde výsledky přibližně stejné.

4.2.3 Testování na datech uf100/uf100-0100.cnf

Data uf100/uf100-0100.cnf jsou určena k testování algoritmů na formuli o velikosti 100 literálů a 430 klauzulí.

Testování ACO algoritmu - první fáze

U ACO algoritmu jsem v první fázi testování vstupních parametrů nejprve nastavil počet mravenců na $a = 100$ a počet iterací na $i = 500$.

Tabulka testování ACO algoritmu pro různá nastavení parametrů α, β, d, q a e na formuli o velikosti 100 literálů a 430 klauzulí						
Parametr α [-]	Parametr β [-]	Počáteční feromon na každé hraně d [-]	Množství pokládaného feromonu q [-]	Odpařování feromonu e [-]	Výsledek f [nesplněné klauzule]	Doba běhu algoritmu t [s]
1	1	0,1	0,01	0,05	**20,2**	5,8
1	1	0,1	0,01	0,50	**12,0**	5,5
1	1	0,1	0,10	0,05	**19,6**	5,6
1	1	0,1	0,10	0,50	**11,2**	5,7
1	1	0,1	1,00	0,05	**17,5**	5,6
1	1	0,1	1,00	0,50	**10,8**	5,6
1	1	1,0	0,01	0,05	**19,9**	5,6
1	1	1,0	0,01	0,50	**11,2**	5,6
1	1	1,0	0,10	0,05	**18,5**	5,6
1	1	1,0	0,10	0,50	**11,7**	5,7
1	1	1,0	1,00	0,05	**18,2**	5,7
1	1	1,0	1,00	0,50	**11,5**	5,5
1	2	0,1	0,01	0,05	**17,8**	5,7
1	2	0,1	0,01	0,50	**13,3**	5,6
1	2	0,1	0,10	0,05	**17,2**	5,7
1	2	0,1	0,10	0,50	**10,8**	5,7
1	2	0,1	1,00	0,05	**17,4**	5,8
1	2	0,1	1,00	0,50	**14,8**	5,7
1	2	1,0	0,01	0,05	**17,2**	5,7
1	2	1,0	0,01	0,50	**9,9**	5,8
1	2	1,0	0,10	0,05	**18,4**	5,6
1	2	1,0	0,10	0,50	**14,2**	5,6
1	2	1,0	1,00	0,05	**17,7**	5,8
1	2	1,0	1,00	0,50	**15,0**	5,7

Tabulka 4-5 První fáze testování ACO algoritmu na datech uf100/uf100-0100.cnf

Z tabulky 4-5 celkem jasně plyne, že pro rychlost odpařování feromonu $e = 0,5$ jsou zde celkové výsledky f nejlepší. Dále jsou celkové průměrné výsledky pro nastavení parametrů $\alpha = \beta = 1$ i pro $\alpha = 1$ a $\beta = 2$ téměř stejné, zde zvolím například $\alpha = \beta = 1$. Množství pokládaného feromonu q by podle tabulky mohlo být v rozmezí $\langle 0,1; 1 \rangle$, například $q = 0,5$. Nakonec pro obě testované hodnoty množství počátečního feromonu d zde očividně vychází výsledky f přibližně stejně, takže zvolím například $d = 0,5$.

Testování ACO algoritmu - druhá fáze

V druhé fázi testování vstupních parametrů u ACO jsem opět použil vstupní parametry zjištěné z prvního testu.

Tabulka testování ACO algoritmu pro různé počty mravenců *a* a různé počty iterací *i* na formuli o velikosti 100 literálů a 430 klauzulí			
Počet mravenců *a* [-]	Počet iterací *i* [-]	Výsledek *f* [nesplněné klauzule]	Doba běhu algoritmu *t* [s]
50	5	**36,5**	0,028
50	10	**33,8**	0,051
50	50	**25,6**	0,262
50	100	**24,2**	0,531
50	500	**12,4**	2,657
50	1000	**9,2**	5,319
100	5	**34,5**	0,056
100	10	**26,9**	0,104
100	50	**23,8**	0,551
100	100	**18,0**	1,063
100	500	**9,8**	5,648
100	1000	**9,1**	10,893
500	5	**29,5**	0,313
500	10	**24,3**	0,629
500	50	**21,9**	3,117
500	100	**18,0**	6,181
500	500	**5,8**	29,211
500	1000	**5,7**	58,060
1000	5	**28,6**	0,625
1000	10	**24,4**	1,286
1000	50	**18,3**	6,316
1000	100	**17,2**	12,324
1000	500	**5,7**	60,424
1000	1000	**5,5**	119,138

Tabulka 4-6 Druhá fáze testování ACO algoritmu na datech uf100/uf100-0100.cnf

Tabulka 4-6 říká, že výsledky *f* závisí jak na počtu iterací *i*, tak na počtu mravenců *a*. I zde není příliš zřetelné, na kterém z těchto parametrů závisí výsledky *f* více. Opět jsem se rozhodl, že při srovnávání ACO algoritmu s algoritmem GRASP budu testovat jak variantu s konstantním počtem mravenců *a* = c a proměnným počtem iterací *i*, tak variantu s konstantním počtem iterací *i* = c a proměnným počtem mravenců *a*.

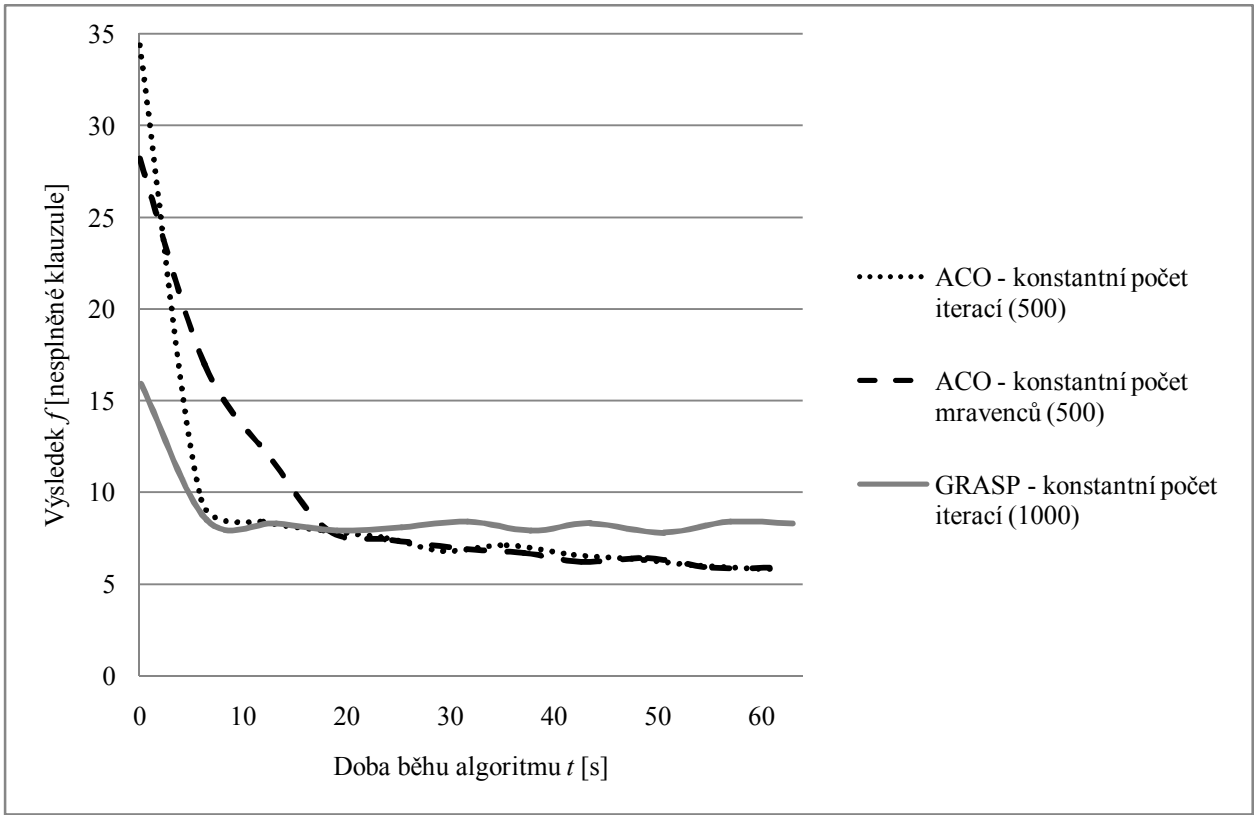
Jak lze z tabulky 4.3 vidět, rozumný počet mravenců by mohl být *a* = 500, a počet iterací *i* = 500.

Srovnání ACO a GRASP algoritmu

Ve srovnání opět použiji zjištěné optimální parametry.

Srovnávací tabulka pro srovnání algoritmů ACO a GRASP na formuli velké 100 literálů a 430 klauzulí								
ACO - konstantní počet iterací (i = 500)			ACO - konstantní počet mravenců (a = 500)			GRASP - konstantní počet iterací (i = 1000)		
Počet mravenců a [-]	Výsledek f [nesplněné klauzule]	Doba běhu algoritmu t [s]	Počet iterací i [-]	Výsledek f [nesplněné klauzule]	Doba běhu algoritmu t [s]	Počet restartů r [-]	Výsledek f [nesplněné klauzule]	Doba běhu algoritmu t [s]
1	**34,4**	0,090	1	**28,2**	0,089	1	**15,9**	0,204
100	**9,6**	5,978	100	**17,0**	6,317	400	**8,5**	6,507
200	**8,4**	11,976	200	**11,7**	12,911	800	**8,3**	13,215
300	**7,9**	17,769	300	**7,9**	18,277	1200	**7,9**	19,113
400	**7,5**	23,893	400	**7,4**	24,408	1600	**8,1**	25,300
500	**6,8**	29,351	500	**6,9**	31,611	2000	**8,4**	31,692
600	**7,1**	35,464	600	**6,7**	37,130	2400	**7,9**	37,808
700	**6,6**	41,948	700	**6,2**	42,740	2800	**8,3**	43,553
800	**6,3**	48,747	800	**6,4**	49,009	3200	**7,8**	50,637
900	**6,0**	54,175	900	**5,9**	55,273	3600	**8,4**	57,078
1000	**5,8**	61,526	1000	**5,9**	61,902	4000	**8,3**	63,015

Tabulka 4-7 Srovnání ACO a GRASP algoritmu na datech uf100/uf100-0100.cnf



Obrázek 4-2 Graf srovnání ACO a GRASP algoritmu na datech uf100/uf100-0100.cnf

Už i z pouhé tabulky 4-7 lze odvodit, že GRASP algoritmus dokáže nalézt celkem slušné řešení *f* už i při nízkém počtu restartů *r*, ale při jeho dalším zvyšování se výsledky *f* už tolik nezlepšují.

ACO algoritmus se zas z počátku nedostane na tak dobré řešení *f* jako GRASP algoritmus, ale zase má větší tendenci dospět k lepším výsledkům *f* při zvyšujícím se počtu iterací *i* či počtu mravenců *a*.

Z grafu na obr. 4-2 vyplývá schopnost ACO algoritmu dospět k lepšímu výsledku *f* za srovnatelnou dobu *t* oproti GRASP algoritmu, za předpokladu dostatečně dlouhého běhu algoritmu.

Algoritmus GRASP je zase schopen nalézt celkem dobré řešení *f* i za krátkou dobu *t*, oproti algoritmu ACO.

4.2.4 Testování na datech uf250/uf250-080.cnf

Data uf250/uf250-080.cnf jsou určena k testování algoritmů na formuli o velikosti 250 literálů a 1065 klauzulí.

Testování ACO algoritmu - první fáze

V první fázi testování jsem zde nastavil konstantní počet mravenců $a = 250$ a konstantní počet iterací $i = 500$.

Tabulka testování ACO algoritmu pro různá nastavení parametrů α, β, d, q a e na formuli o velikosti 250 literálů a 1065 klauzulí						
Parametr α [-]	Parametr β [-]	Počáteční feromon na každé hraně d [-]	Množství pokládaného feromonu q [-]	Odpařování feromonu e [-]	Výsledek f [nesplněné klauzule]	Doba běhu algoritmu t [s]
1	1	0,1	0,1	0,1	**64,0**	36,254
1	1	0,1	0,1	0,5	**27,9**	36,782
1	1	0,1	1,0	0,1	**65,1**	35,717
1	1	0,1	1,0	0,5	**23,6**	35,757
1	1	0,1	10,0	0,1	**61,0**	35,956
1	1	0,1	10,0	0,5	**25,2**	35,418
1	1	1,0	0,1	0,1	**62,4**	36,241
1	1	1,0	0,1	0,5	**28,8**	36,010
1	1	1,0	1,0	0,1	**66,0**	36,198
1	1	1,0	1,0	0,5	**20,2**	36,454
1	1	1,0	10,0	0,1	**56,1**	35,888
1	1	1,0	10,0	0,5	**26,0**	36,294
1	2	0,1	0,1	0,1	**60,5**	36,260
1	2	0,1	0,1	0,5	**40,2**	36,671
1	2	0,1	1,0	0,1	**59,6**	37,170
1	2	0,1	1,0	0,5	**26,2**	36,982
1	2	0,1	10,0	0,1	**61,6**	36,390
1	2	0,1	10,0	0,5	**41,8**	37,132
1	2	1,0	0,1	0,1	**56,5**	36,509
1	2	1,0	0,1	0,5	**30,2**	35,448
1	2	1,0	1,0	0,1	**57,6**	35,645
1	2	1,0	1,0	0,5	**34,8**	36,814
1	2	1,0	10,0	0,1	**54,2**	35,192
1	2	1,0	10,0	0,5	**37,0**	35,827

Tabulka 4-8 První fáze testování ACO algoritmu na datech uf250/uf250-080.cnf

Z tabulky 4-8 lze vidět, že pro hodnoty parametrů $\alpha = \beta = 1$, vychází výsledky f lépe. Jednoznačně lépe vychází výsledky f pro nastavení parametru odpařování feromonu na hodnotu $e = 0{,}5$. Vliv testovaných hodnot počátečního feromonu d je vzhledem k výsledkům f nerozhodný, takže zvolím třeba hodnotu $d = 0{,}5$. Nakonec množství pokládaného feromonu bych zde volil $q = 1$, neboť pro něj v testu vycházejí nejlepší výsledky f.

Testování ACO algoritmu - druhá fáze

V této fázi testování jsem opět použil vstupní parametry zjištěné z prvního testu, které byly po celé toto testování konstantní.

Tabulka testování ACO algoritmu pro různé počty mravenců *a* a různé počty iterací *i* na formuli o velikosti 250 literálů a 1065 klauzulí			
Počet mravenců *a* [-]	Počet iterací *i* [-]	Výsledek *f* [nesplněné klauzule]	Doba běhu algoritmu *t* [s]
50	10	**101,7**	0,142
50	50	**86,9**	0,755
50	100	**67,3**	1,529
50	500	**65,6**	8,189
50	1000	**64,2**	16,436
100	10	**88,5**	0,328
100	50	**88,2**	1,623
100	100	**70,4**	3,404
100	500	**43,0**	17,929
100	1000	**36,9**	38,104
500	10	**93,2**	2,030
500	50	**78,0**	10,307
500	100	**65,5**	19,841
500	500	**22,3**	100,209
500	1000	**20,9**	197,623
1000	10	**88,5**	3,997
1000	50	**75,2**	20,688
1000	100	**65,0**	39,883
1000	500	**21,6**	197,736
1000	1000	**16,1**	401,142

Tabulka 4-9 Druhá fáze testování ACO algoritmu na datech uf250/uf250-080.cnf

Z tabulky 4-9 lze vidět, že výsledky *f* se zlepšují, se zvyšováním počtu iterací *i*, i se zvyšováním počtu mravenců *a*. Při srovnávání ACO algoritmu s algoritmem GRASP budu testovat jak variantu s konstantním počtem mravenců *a* = c, tak variantu s konstantním počtem iterací *i* = c. Stejně jako při srovnávacích testech na předchozích datech.

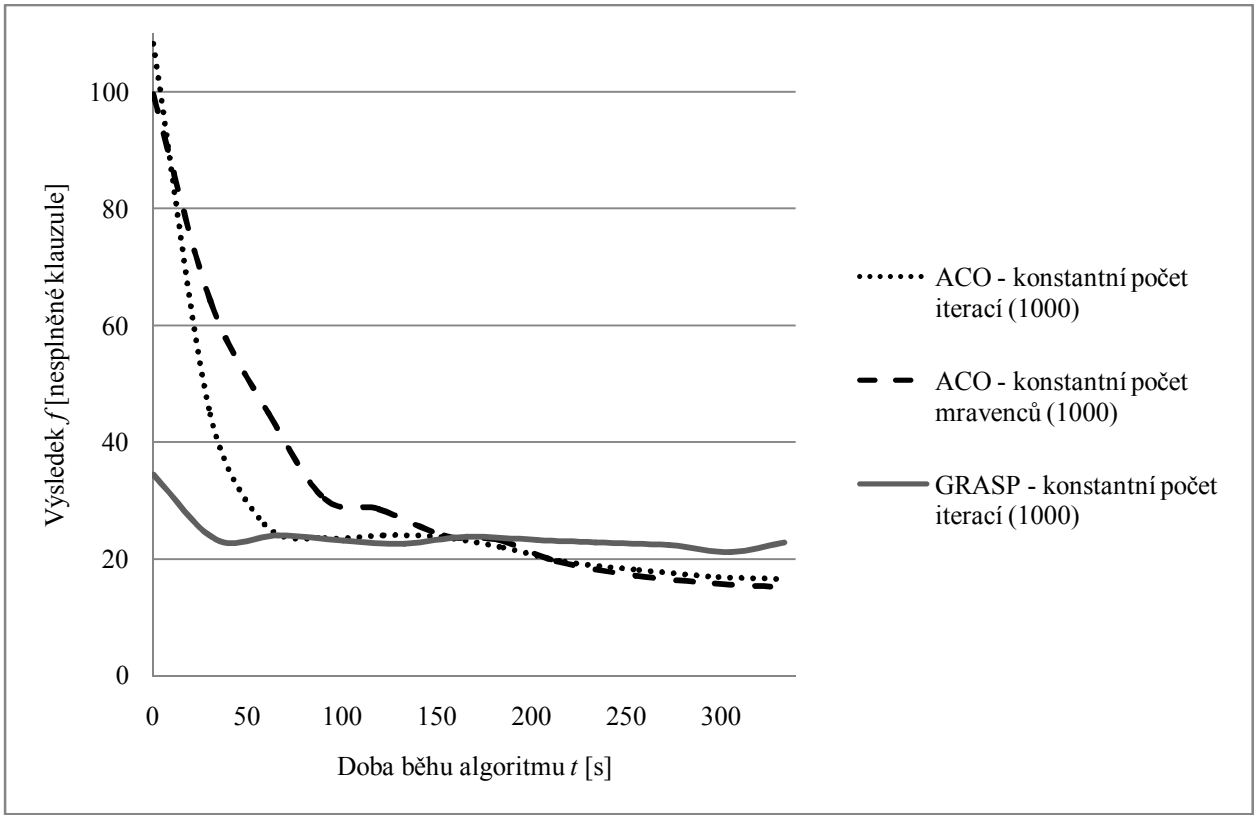
Rozumný počet mravenců by zde mohl být *a* = 1000, a rozumný počet iterací *i* = 1000. Neboť je zde patrné, že výsledky *f* jsou nejlepší právě pro tyto hodnoty parametrů. Kdybych testoval ještě vyšší hodnoty těchto parametrů, tak je dost možné, že by se výsledky *f* ještě dále zlepšovaly, ale testy by zas trvaly příliš dlouho a mně jde spíše o porovnávání ACO a GRASP algoritmu při srovnatelných časech, takže při srovnávání algoritmů použiji již zmíněné hodnoty.

Srovnání ACO a GRASP algoritmu

Zde jsem při srovnávání postupoval opět stejně jako v případě předchozích dat.

Srovnávací tabulka pro srovnání algoritmů ACO a GRASP na formuli o velikosti 250 literálů a 1065 klauzulí								
ACO - konstantní počet iterací (i = 1000)			ACO - konstantní počet mravenců (a = 1000)			GRASP - konstantní počet iterací (i = 1000)		
Počet mravenců a [-]	Výsledek f [nesplněné klauzule]	Doba běhu algoritmu t [s]	Počet iterací i [-]	Výsledek f [nesplněné klauzule]	Doba běhu algoritmu t [s]	Počet restartů r [-]	Výsledek f [nesplněné klauzule]	Doba běhu algoritmu t [s]
1	**108,3**	0,346	1	**99,6**	0,304	1	**34,4**	0,425
100	**44,7**	30,445	100	**64,3**	30,392	500	**23,3**	33,620
200	**25,9**	59,257	200	**45,0**	61,312	1000	**24,0**	66,420
300	**23,5**	91,153	300	**30,2**	91,111	1500	**23,1**	100,768
400	**24,0**	122,122	400	**28,5**	119,557	2000	**22,6**	133,126
500	**23,7**	153,229	500	**24,0**	153,905	2500	**23,8**	166,077
600	**21,9**	185,318	600	**23,3**	182,194	3000	**23,2**	202,981
700	**19,5**	217,949	700	**19,6**	213,013	3500	**22,8**	234,066
800	**18,1**	257,182	800	**17,1**	255,397	4000	**22,3**	272,922
900	**16,9**	299,000	900	**15,8**	297,085	4500	**21,1**	305,497
1000	**16,5**	331,827	1000	**15,2**	326,570	5000	**22,8**	333,744

Tabulka 4-10 Srovnání ACO a GRASP algoritmu na datech uf250/uf250-080.cnf



Obrázek 4-3 Graf srovnání ACO a GRASP algoritmu na datech uf250/uf250-080.cnf

Ze srovnávací tabulky 4-10 je opět možno odvodit, že výsledky *f* algoritmu ACO mají větší tendenci se zlepšovat oproti algoritmu GRASP.

Stejně tak je i zde možné pozorovat, že GRASP algoritmus nalezne rychleji lepší řešení *f*, než algoritmus ACO, které pak už ale nemá zlepšující se tendenci, když se algoritmus nechá déle běžet.

Graf na obr. 4-3 celkem názorně ukazuje, jak je algoritmus ACO schopen zlepšovat výsledné řešení *f* s rostoucí dobou *t*, po kterou se nechá běžet.

Algoritmus GRASP oproti algoritmu ACO zase rychle dospěje k lepšímu řešení *f*, ale pak už se moc, či vůbec, nezlepšuje s rostoucí dobou běhu *t*.

4.3 Celkové vyhodnocení experimentů

Experimenty dopadly podle očekávání. Algoritmus ACO zpočátku nedává tak rychle dobré výsledky jako algoritmus GRASP, ale s rostoucí dobou výpočtů má algoritmus ACO vetší tendenci, své výsledky zlepšovat. Po dostatečně dlouhé době běhu dá většinou ACO algoritmus lepší výsledek, než algoritmus GRASP.

Další věcí vypozorovanou z experimentů je, že algoritmus ACO nemusí vždy nutně dávat lepší výsledky než algoritmus GRASP, například když formule neobsahuje příliš mnoho literálů. Naopak u formulí o mnoha literálech dopadá algoritmus ACO při optimálně nastavených parametrech znatelně lépe než algoritmus GRASP.

Nevýhodou ACO algoritmu je, že optimální nastavení jeho parametrů je o dost složitější, neboť jich je více než u algoritmu GRASP.

5 Závěr

Jak algoritmus ACO, tak GRASP se ukázaly být dobrými nástroji pro řešení optimalizačního problému SAT v přijatelném čase.

Oba zmíněné algoritmy se podařilo úspěšně naimplementovat a odzkoušet na datech připravených k testování optimalizačních algoritmů na problému SAT.

Z výsledků testů je patrné, algoritmus GRASP se hodí především pro případy, kdy je potřeba rychle dosáhnout nějakého přijatelného řešení, avšak často za cenu jeho nižší kvality. Dále bylo experimentálně ověřeno, že v některých případech může algoritmus GRASP dávat i lepší řešení než sofistikovanější algoritmy. To zejména, když vstupními daty jsou formule, které neobsahují větší počty literálů a klauzulí. Pak má GRASP algoritmus velký náskok před algoritmem ACO už i jen proto, že v první fázi zkonstruuje hladové řešení, které pro menší objemy vstupních dat dává obvykle výsledky blízké nejlepšímu řešení.

U algoritmu ACO bylo vypozorováno, že nalezení dobrého řešení mu trvá déle, než algoritmu GRASP, ale po dostatečně dlouhém běhu dá nakonec většinou lepší nebo aspoň stejný výsledek jako algoritmus GRASP. ACO algoritmus má před GRASP algoritmem výhodu zejména když vstupními daty jsou formule o mnoha literálech a mnoha klauzulích. Nevýhodou je zde složitější nastavování optimálních vstupných parametrů algoritmu.

Nabízí se zde otázka, jak by asi vypadalo spojení algoritmů GRASP a ACO, kdy GRASP algoritmus by běžel v první části výpočtu a rychle by našel slušné řešení, které by poté v druhé části výpočtu předal algoritmu ACO, který by ho dále vylepšoval. Myslím, že toto spojení by za předpokladu optimálního nastavení vstupních parametrů mohlo dávat velice slušné výsledky za přijatelnou dobu běhu algoritmu.

Literatura

1. Heuristické algoritmy. *Wikipedie Otevřená Encyklopedie.* [Online] http://cs.wikipedia.org/wiki/Heuristické_algoritmy.

2. Search Agents for optimalization. *Seage.* [Online] http://seage.sourceforge.net/.

3. **Kutil, MICHAL.** Splnitelnost Booleovských formulí. [Online] http://www.tim.cz/publications/2005/sat05/sat05.pdf.

4. NP-úplnost. *Wikipedie Otevřená Encyklopedie.* [Online] http://cs.wikipedia.org/wiki/NP-úplnost.

5. NP (třída složitosti). *Wikipedie Otevřená Encyklopedie.* [Online] http://cs.wikipedia.org/wiki/Nedeterministicky_polynomiální_problém.

6. Prohledávání stavového prostoru. *Wikipedie Otevřená Encyklopedie.* [Online] http://cs.wikipedia.org/wiki/Prohledávání_stavového_prostoru.

7. Hladový algoritmus. *Wikipedie Otevřená Encyklopedie.* [Online] http://cs.wikipedia.org/wiki/Hladový_algoritmus.

8. Hill-climbing. *Wikipedie Otevřená Encyklopedie.* [Online] http://cs.wikipedia.org/wiki/Hill-climbing.

9. **Němec, MILOŠ.** Diplomová práce - Optimalizace pomocí mravenčích kolonií. [Online] http://www.milosnemec.cz/download/dp.pdf.

10. **Kubalík, JIŘÍ.** Optimalizační algoritmy inspirované chováním mravenců. [Online] http://labe.felk.cvut.cz/~kubalik/kopr/kopr_ants.pdf.

Příloha A - CD-ROM

Přílohou bakalářské práce je CD obsahující tuto práci v PDF formátu a projekt se zdrojovými kódy v následujících adresářích:

- PDF/ zde je bakalářská práce v PDF formátu
- PROJECT/ zde je projekt se zdrojovými kódy