MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁRSKA PRÁCA

Peter Kovács

Optimalizace mravenčí kolonií

Informatický ústav Univerzity Karlovy

Vedúci bakalárskej práce:	RNDr. Ondřej Pangrác, Ph.D.
Študijný program:	Informatika
Študijný odbor:	Obecní informatika

Praha 2018

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

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

V ..... dne ..... Podpis autora

Za pomoc pri písaní práce by som chcel poďakovať môjmu vedúcemu práce RNDr. Ondřejovi Pangrácovi, Ph.D.. Najmä mu ďakujem za pomoc pri hľadaní zdrojov a ochotnú pomoc pri riešení problémov, ktoré sa počas práce vyskytli.

Názov práce: Optimalizace mravenčí kolonií

Autor: Peter Kovács

Katedra: Informatický ústav Univerzity Karlovy

Vedúcí bakalárskej práce: RNDr. Ondřej Pangrác, Ph.D., Informatický ústav Univerzity Karlovy

Abstrakt: V práci sa venujem porovnávaniu metaheuristiky Ant Colony s inými metaheuristikami ako Simulated Annealing, Tabu Search alebo hladné algoritmy. Metaheuristiky som porovnával na probléme obchodného cestujúceho, pri ofarbovaní grafu a množinovom pokrytí. V práci sú podrobne rozobraté implementácie metaheuristík na jednotlivé problémy. Pri porovnávaní je braná do úvahy hlavne výsledná cenu riešenia, ale aj čas. V práci je snaha identifikovať, v ktorých prípadoch je vhodné použiť Ant Colony. Ant Colony v prípade množinového pokrytia a problému obchodného cestujúceho funguje spoľahlivo pri veľkých instanciách.

Kľúčové slová: optimalizácia, heuristiky, mravčia kolónia, problém obchodného cestujúceho, ofarbovanie grafu,množinové pokrytie

Title: Ant colony optimization

Author: Peter Kovács

Department: Computer Science Institute

Supervisor: RNDr. Ondřej Pangrác, Ph.D., Computer Science Institute

Abstract: The aim of this work was to compare metaheuristic Ant Colony with other metaheuristics like Simulated Annealing, Tabu Search or Greedy algorithms. Metaheuristics were compared on three different NP-hard problems. Specificaly, Traveling Salesman Problem, Graph Coloring Problem and Set Covering Problem. Detailed description of implementation of metaheuristics in particular problems can be found in this work. Metaheuristics were compared based on cost function and execution time. However, this work tries to identify instances, in which Ant Colony was the best choice. Ant Colony is reliable metaheuristic on large Traveling Salesman or Set Covering problems.

Keywords: optimization, heuristics, Ant Colony, Traveling Salesman Problem, Graph Coloring, Set Covering

Obsah

Úvod **3**

1 Metaheuristiky **4**
1.1 Riešenie NP-ťažkých problémov 4
1.2 Implementácia heuristík 4
1.3 Metaheuristiky použité v práci 5
1.3.1 Ant Colony 5
1.3.2 Simulated Annealing 6
1.3.3 Tabu Search 7
1.3.4 Hladné a iné jednoduché princípy 8

2 Problém obchodného cestujúceho **9**
2.1 Popis problému 9
2.2 Nasadenie Ant Colony 9
2.3 Nasadenie simulovaného žíhánia 10
2.4 Nasadenie tabu Search 11
2.5 Implementačné detaily 11
2.6 Výkon jednotlivých metaheuristík 11
2.7 Záver 15

3 Ofarbovanie grafu **16**
3.1 Popis problému 16
3.2 Nasadenie ant colony 16
3.3 Nasadenie hladných algoritmov 18
3.3.1 First Fit 18
3.3.2 Saturation Degree Ordering 18
3.4 Nasadenie simulovaného žíhánia 19
3.5 Implementačné detaily 20
3.6 Výkon jednotlivých metaheuristík 20
3.7 Záver 23

4 Množinové pokrytie **25**
4.1 Popis problému 25
4.2 Nasadenie Ant Colony 25
4.3 Nasadenie simulovaného žíhání 26
4.4 Nasadenie Tabu Search 27
4.5 Nasadenie hladného algoritmu 28
4.6 Implementačné detaily 29
4.7 Výkon jednotlivých metaheuristík 30
4.8 Záver 33

Záver **35**

Zoznam použitej literatúry **36**

Zoznam obrázkov **37**

Zoznam tabuliek **38**

A Prílohy **39**

A.1 Používateľská dokumentácia pre spustenie programov na testovanie metaheuristík 39

A.1.1 Nastavenie parametrov 39

A.1.2 Výstup 40

A.2 Príloha na CD 40

Úvod

Napriek veľkému zvýšeniu výpočtovej kapacity dnešných počítačov sú stále pre nás NP-ťažké problémy na veľkých vstupoch nespočítateľné v rozumnom čase. Mnohokrát nie je dôležité nájsť optimálne riešenie, ale nájsť riešenie, ktoré je blízko optimu, v čo najkratšom čase. Často totiž čas na nájdenie optimálneho riešenia je tak dlhý, že úspora prinesená optimalitou riešenia je oveľa menšia ako náklady na nájdenie tohto riešenia.

S potrebou vyriešiť problém v krátkom čase, ale stále pomerne blízko optimálneho riešenia nám môžu pomôcť heuristiky. Heuristiky zastupujú istú intuíciu algoritmu o tom, ako hľadať dobré riešenie. Mnoho z týchto heuristík vychádza zo zaujímavých inšpirácií v realite. Práve všeobecným heuristikám, ktoré popisujú všeobecný návod alebo kostru fungovania algoritmu sa budem venovať v tejto práci. Takéto heuristiky sa nazývajú metaheuristiky.

Jednou zo zaujímavých metaheuristík je optimalizácia mravčou kolóniou (anglicky Ant Colony Optimization). Táto metaheuristika dokazuje, že vždy sa máme od prírody čo naučiť. Metaheuristika Ant Colony sa inšpiruje správaním mravcov pri hľadaní potravy. Je zaujímavé, že tento postup je použiteľný nielen pre problémy hľadania cesty, ale dá sa implementovať na celú škálu problémov, ktoré zdanlivo s mravcami nič spoločné nemajú.

Cieľom mojej práce bolo porovnať výkon metaheuristiky Ant Colony s inými prístupmi na riešenie NP-ťažkých problémov. V práci sú použité tri rôzne NP-ťažké problémy a to konkrétne: Problém obchodného cestujúceho, Ofarbovanie grafu a Množinové pokrytie. Pri každom z týchto problémov boli okrem Ant colony zvolené aspoň dve ďalšie metaheuristiky, s ktorými boli výsledky porovnávané.

Okrem Ant Colony sú v práci použité metaheuristiky Simulované žíhanie (anglicky Simulated Annealing), Tabu Search ale aj hladné algoritmy. Metaheuristiky a ich implementácia pri jednotlivých problémoch sú vybraté na základe rôznych článkov.

Tieto výsledky sú následne porovnávané medzi sebou, pričom cieľom práce je určiť, na ktorý problém a za akých podmienok, sa ktorá metaheuristika hodí. Práca je zameraná hlavne na porovnanie Ant Colony, ako nosnej metaheuristiky.

1. Metaheuristiky

1.1 Riešenie NP-ťažkých problémov

Kategória NP-ťažkých problémov je v dnešnej dobe veľkou výzvou. Pre tieto problémy neexistuje riešenie v polynomiálnom čase, čo spôsobuje, že ak by sme chceli nájsť optimálne riešenie v takejto úlohe bude to trvať obrovské množstvo času. Z tohto dôvodu sa často uspokojíme s riešením, ktoré nájdeme rýchlo, aj keď nie je najlepšie možné.

Na postačujúce vyriešenie týchto problémov sa preto používajú dva postupy. Jedným z nich sú aproximačné algoritmy. Aproximačné algoritmy sa snažia nejakým druhom optimalizácie nájsť riešenie. Zároveň zaručujú maximálnu odchýlku od optimálneho riešenia. Takáto záruka sa môže hodiť ak vyžadujeme nutnú spoľahlivosť do nejakej úrovne. Často však táto maximálna odchýlka môže byť príliš veľká.

Jedným z prístupov, ktorým sa dá riešiť každý problém v práci je aj lineárne programovanie. Lineárne programovanie formuluje úlohu ako systém nerovníc a funkciu, ktorú treba optimalizovať. Tieto algoritmy zvládnu nájsť optimálne riešenie, ale jedine pokiaľ sa jedná o sústavu lineárnych nerovníc riešenú v reálnych číslach. Bohužiaľ pre väčšinu problémov potrebujeme sústavy vyriešiť v celých číslach. V tomto prípade musíme úlohu relaxovať na úlohu v reálnych číslach, vyriešiť a následne výsledok takéhoto riešenia zaokrúhliť. Finálne zaokrúhlenie môže spôsobiť, že získané riešenie nebude optimálne.

Druhou stratégiou sú heuristiky. Heuristiky nám poskytujú spôsob ako rozumne prehľadávať riešenia. Poskytujú nám nápovedu kadiaľ prehľadávať priestor riešení na základe ľahko spočítateľných dát. Táto metóda nezaručuje nájdenie optimálneho riešenia, no často sa dopracuje k postačujúcemu riešeniu. Zatiaľ čo heuristika je postup pri konkrétnom probléme, existujú aj postupy, ktoré sú všeobecným postupom pri riešení problémov. Tieto postupy sa nazývajú matehauristiky. Metaheuristika je teda všeobecný návod pre hľadanie optimálneho riešenia, ktorý sa už len prispôsobí na konkrétny problém.

1.2 Implementácia heuristík

V práci som si vybral tri problémy, ktoré sú známejšie [5] a dá sa na nich použiť metaheuristika Ant Colony. Pre každý problém som vytvoril program pre testovanie v C++, kde som implementoval jednotlivé metaheuristiky. C++ som si vybral z dôvodu rýchlosti, ktorá je pri riešení týchto problémov podstatná. Používanie jednotlivých programov je popísané v prílohe A.1 . Samotné zdrojové kódy programov, testované instancie a výstupné súbory sú uložené na CD v prílohe. Podrobnejší popis ako sa na CD orientovať je v prílohe A.2.

Pri výsledkoch testovaní jednotlivých implementácií metaheuristík je v práci uvedená tabuľka s cenou riešenia. Údaje o čase behu explicitne v práci nie sú uvedené. Napriek tomu čas behu hodnotím v záveroch. Tieto časy sa dajú nájsť v prílohe A.2 na CD. Programy boli testované na bežnom počítatči a preto tento čas nie je čistý beh programu. Do behu sa zarátava aj čas keď bol výpočet prerušený operačným systémom. Tieto čísla teda nie sú úplne presné a dá sa z nich

vyvodzovať iba záver vo veľmi hrubých odhadoch.

Pre všetky metaheuristiky existuje aj varianta, pri ktorej môže algoritmus bežať vo viacerých vláknach. Vo svojej práci som sa rozhodol všetky algoritmy programovať jednovláknovo, kvôli konzistentnejšiemu porovnaniu.

1.3 Metaheuristiky použité v práci

Cieľom práce je porovnať výkonnosť konkrétnej metaheuristiky (Ant Colony) na vybraných NP-ťažkých problémoch. Pre porovnanie som pri každom probléme vybral ďalšie metaheuristiky, ktoré sa na daný problém používajú. Vybral som si metaheuristiku Simulated Annealing, nakoľko sa dá naimplementovať na každý z problémov. Ďalej som vybral Tabu-Search, ktorá postupuje podobne pri vytváraní podobných riešení, no rozdiel je v spôsobe prehľadávania priestoru riešení. Ako vhodná metaheuristika sa ponúkajú taktiež genetické algoritmy. Túto metaheuristiku vo svojej práci neporovnávam, nakoľko je veľmi citlivá na nastavenie parametrov a samotné vyladzovanie parametrov by mohlo byť predmetom skúmania.

Pri niektorých problémoch dávajú dobré a veľmi rýchle výsledky aj hladné algoritmy. V práci som ich použil hlavne ako porovnanie toho, aký výsledok vieme dostať prakticky zadarmo.

1.3.1 Ant Colony

Metaheuristika Ant Colony je inšpirovaná chovaním mravcov v prírode pri hľadaní potravy. Mravce sa riadia feromónmy, ktoré vylučujú podľa toho, či sa im podarilo nájsť potravu.

Ant Colony postupne buduje riešenie úlohy tak, že vyšle nejaký počet mravcov, z ktorých každý postupne konštruuje riešenie. Pri každom rozšírení riešenia sa jednotlivý mravec rozhoduje náhodne, pričom pravdepodobnostnú distribúciu ovplyvňujú feromóny a heuristický parameter. Heuristický parameter zvyčajne zohľadňuje ako sa riešenie čiastkovo zlepší.

Pre mravca k, ktorý sa nachádza v stave x je pravdepodobnosť, že sa rozhodne pokračovať do stavu y následovná:

$$p_{xy}^k = \frac{\tau_{xy}^\alpha \cdot \eta_{xy}^\beta}{\sum_{z \in U_k} \tau_{xz}^\alpha \cdot \eta_{xz}^\beta}$$

kde η_{xy} je heuristický koeficient pre prejdenie zo stavu x do stavu y a τ_{xy} je hladina feromónu pre prejdenie zo stavu x do stavu y. Koeficienty α a β sú parametre metaheuristiky. Množina U_k je množina všetkých doposiaľ nenavštívených stavov mravcom k. Ďalej pre $y \notin U_k$ platí $p_{xy}^k = 0$.

Po tom ako každý z mravcov zkonštruuje riešenie, sa podľa výhodnosti riešenia aktualizuje feromónová stopa. Okrem pridávania feromónov v tomto kroku ešte prebieha vyparovanie starých feromónov a to tak, že vymizne vopred dané percento feromónov z každého prechodu medzi stavmi. Tento krok slúži na to, aby sa zabránilo uviaznutiu riešenia v lokálnom minime. Pre každý prechod zo stavu x a y sa feromónová stopa prepočíta takto:

$$\tau_{xy} = (1 - \rho)\tau_{xy} + \sum_k \Delta\tau_{xy}^k$$

kde $\rho \in [0,1]$ je koeficient vyparovania a $\Delta\tau_{xy}^{k} = max(0,Q/L_k)$, Q je konštanta a L_k je cena riešenia vyrobeného k-tým mravcom

```
procedure ACO
  {
    bestSolution <- null
    while(!stop_condition)
    {
       foreach (Ant a)
       {
        s <- generateSolution()
        if (s is best solution)
        {
            bestSolution <- s
        }
       }

       pheromoneUpdate()
    }
    return(bestSolution)
  }
```

Obrázok 1.1: Metaheuristika Ant Colony

1.3.2 Simulated Annealing

Metóda Simulated Annealing, v preklade simulované žíhanie, je inšpirovaná spracovávaním kovov. Kov je najskôr rozohriaty na vysokú teplotu a upravuje sa jeho tvar. Čím viac chladne, tým menej je tvarovateľný. Na začiatku teda vieme pretvárať kov do rôznych tvarov aj takých, ktoré ešte nevyzerajú ani zďaleka ako finálny produkt a na konci už len vylaďujeme detaily. Prácu skončíme, keď kov úplne vychladne.

Tento prístup potrebuje mať skonštruované nejaké iniciálne riešenie. Následne algoritmus iteruje, až po istú hranicu teploty (keď je kov príliš chladný, tak ho už ďalej nemôžeme tvarovať). V každej iterácií sa vytvorí takzvané *susedné riešenie*. Susedné riešenie je v nejakom zmysle blízke pôvodnému riešenie a je len čiastkovo zmenené. Zostrojenie tohto riešenia si môžeme predstaviť ako jeden úder kladivom do kovu.

Algoritmus následne rozhodne, či sa riešenie príjme, alebo zamietne náhodne. Pri náhodnom rozhodovaní je pravdepodobnostná distribúcia schválenia resp. odmietnutia daná teplotou a výhodnosťou riešenia, pričom pri vysokých teplotách je pravdepodobné, že sa akceptuje aj horšie riešenie ako aktuálne, zatiaľ čo pri nižších to je nízko pravdepodobné. Takto algoritmus zabraňuje uviaznutiu v lokálnom minime. Pravdepodobnosť akceptovania riešenia y, pri aktuálnom riešení x je:

$$p_{xy} = e^{-\frac{cost(y)-cost(x)}{t}}$$

kde t je aktuálna teplota a $cost(x)$ je účelová funkcia pre x.

Po akceptovaní alebo zamietnutí riešenia sa upraví teplota:

$$t = \alpha t$$

pričom α je koeficient chladnutia a je parametrom metaheuristiky. Tento výber opäť dáva zmysel v kontexte spracovania kovu. Na začiatku sme ochotný tvarovať kov aj na niečo čo menej pripomína žiadaný výsledok.

To čo sa líši pre implementácie jednotlivých problémov je účelová funkcia, metóda hľadania suseda a konštrukcia iniciálneho riešenia.

```
procedure SimulatedAnnealing
{
        temperature = LARGE_NUMBER
        s <- constructInitialSolution()
        bestSolution <- s
        while (temperature>MIN_TEMP)
        {
                x <- constructNeighbourSolution(s)
                if (accept(s,x,temperature))
                {
                        s <- x
                        if (cost(x)<cost(bestSolution))
                            bestSolution <- x
                }
                temperature <- temperature*coolingCoeficient
        }
}

function accept(oldSolution,newSolution,temperture)
{
    delta =  cost(newSoution) - cost(oldSolution)
    return (random(0,1)<exp(-delta/temperature))
}
```

Obrázok 1.2: Metaheuristika Simulated Annealing

Metaheuristika Simulated Annealing môže byť rozšírená o takzvaný plán ochladzovania (anglicky Cooling Schedule). Tento plán je zovšeobecnením spôsobu, ako sa upravuje teplota po každej iterácií. V práci je použitý iba jednoduchý model, kde sa ochladzovanie deje, tak ako je popísané vyššie. Dôvodom je, že voľba ochladzovacieho plánu by pridala ďalšie parametre algoritmu.

1.3.3 Tabu Search

Tabu Search [4] podobne ako Simulated Annealing prehľadáva priestor riešení pomocou hľadania susedných riešení. Rozdiel medzi metaheuristikami Simulated Annealing a Tabu Search je, že Tabu Search vždy vyberie najlepšieho kandidáta spomedzi susedov. To znamená, že výpočet je náročnejší, nakoľko je potrebné spočítať účelovú funkciu pre každého zo susedov. Taktiež vzniká problém, ako pri klasickom horolezeckom algoritme, že algoritmus sa zasekne v lokálnom minime.

Tabu Search však naviac prináša takzvaný Tabu-list, teda zoznam zakázaných riešení. Je to zoznam riešení, ktoré boli objavené v nedávnej dobe. Tabu Search teda spomedzi susedov môže vybrať iba tie riešenia, ktoré sa v Tabu-liste nenachádzajú. Takto sa zabráni uviaznutiu a je povolená len cesta k novým neobjaveným riešeniam. Veľkosť Tabu-listu je parameter algoritmu. Tabu list zvyčajne funguje ako fronta, teda systémom FIFO.

```
procedure TabuSearch
{
        TabuList <- emptyList()
        s <- constructInitialSolution()
        bestSolution <- s
        while (!stop_condition)
        {
                bestFromNeighbours <- constructNeighbourSolution(s)
                foreach(x in Neighbours(s))
                {

                        if (!isInTabuList(x) &&
                            cost(x)<cost(bestFromNeighbors))
                        {
                                bestFromNeighbours <- x
                            if (cost(x)<cost(bestSolution))
                                bestSolution <- x
                        }
                }
                s <- bestFromNeighbors
        }
}
```

Obrázok 1.3: Metaheuristika Tabu Search

1.3.4 Hladné a iné jednoduché princípy

V niektorých prípadoch dávajú dobré výsledky aj hladné algoritmy. Hladné algoritmi v každom kroku siahnu po riešení (resp. čiastkovom riešení), ktoré je lokálne optimálne, teda najmenej zväčší účelovú funkciu. Je preto veľmi pravdepodobné, že sa zaseknú v lokálnom minime. Tieto algoritmi sú zvyčajne veľmi rýchle, nakoľko často uplatňujú veľmi jednoduchú logiku rozhodovania.

Hladné algoritmy som použil v práci pri ofarbovaní grafu a množinovom pokrytí. Tieto algoritmy slúžia v prípade týchto problémov taktiež na nájdenie iniciálneho riešenia pre Simulated Annealing alebo Tabu Search, nakoľko je potrebné aby iniciálne riešenie bolo korektné.

2. Problém obchodného cestujúceho

Problém obchodného cestujúceho je asi najznámejší NP-ťažký problém. Problém ilustruje rozhodovanie obchodníka, ktorý chce precestovať isté miesta a nezáleží na tom, v akom poradí ich navštívi. Úlohou je nájsť najkratšiu cestu, ktorou obíde všetky miesta a vráti sa naspäť. Okrem tohto náhľadu sa problém často vyskytuje aj v oblastiach plánovania, logistických problémoch, pri výrobe mikročipov ale aj sekvenovaní DNA.

2.1 Popis problému

Majme úplný neorientovaný graf $G = (V,E)$. Cieľom je nájsť poradie vrcholov $v_{p(1)}, v_{p(2)}, \ldots, v_{p(|V|)}$, také, aby sme minimalizovali

$$\sum_{i=1}^{|V|} d(v_{p(i)}, v_{p((i+1) \bmod |V|)})$$

kde $d(i,j)$ je vzdialenosť vrcholov i a j

Hľadáme teda najkratšiu okružnú cestu vedúcu cez všetky vrcholy. Knižnica problémov TSPLIB [2] poskytuje viacero typov problémov. Niektoré sú zadané pomocou súradníc vrcholov, iné sú zadané maticou susednosti. Pre problémy zadané súradnicami vrcholov, je taktiež potrebná metrika, ktorá je súčasťou problému. Najviac problémov je zadaných súradnicami s euklidovskou metrikou. Vzhľadom na nízky počet iných problémov som sa rozhodol použiť v práci iba takto zadané problémy.

2.2 Nasadenie Ant Colony

Pri probléme obchodného cestujúceho je Ant Colony [3] použitá najintuitívnejšie. Môžeme si predstaviť, že mravce sa rozliezajú po grafe a hľadajú najkratšiu kružnicu obsahujúcu všetky vrcholy. Teda akoby každý mravec bol jeden obchodný cestujúci. V každom vrchole sa mravec rozhodne, do ktorého vrcholu pôjde ďalej, pričom nesmie navštíviť vrchol, v ktorom už bol. Ako heuristický koeficient η_{xy} sa najčastejšie používa prevrátená hodnota dĺžky hrany xy, teda $1/d(x,y)$. Feromóny sa následne nanášajú na jednotlivé hrany vybranej kružnice, pričom množstvo feromónu je nepriamo úmerné dĺžke nájdenej cesty.

Oproti obyčajnejnému hladnému algoritmu, ktorí by vybral najkrašiu hranu do nenavštíveného vrcholu, si Ant Colony vie pamätať, ktoré hrany boli súčasťou príliš dlhého okruhu.

```
function generateSolution()
{
  currentPath <- empty()
  NotVistedVertices <- allVertices
  currentVertex <- 0
  repeat(numberOfVertices)
  {
      foreach(Vertex v from NotVisitedVertices)
      {
        Distribution.Add(v, probability(v))
      }
      chosenVertex <- ChooseByDistribution(Distribution)
      currentPath.Add(chosenVertex)
      NotVisited.Remove(chosenVertex)
  }
  return currentPath
}
```

Obrázok 2.1: Generovania riešenia v Ant Colony pre obchodného cestujúceho

2.3 Nasadenie simulovaného žíhánia

Pre problém obchodného cestujúceho postačí použiť ako iniciálne riešenie náhodné zoradenie vrcholov. Prirodzená je aj účelová funkcia, kde cena za riešenie je jednoducho jeho dĺžka.

Pri konštrukcii susedného riešenia [10] budeme postupovať tak, že vezmeme náhodnú dvojicu vrcholov v ceste a celú cestu medzi nimi otočíme naopak. Teda ak pôvodná cesta vyzerala takto:

$$1,2,3,4,5,6,7,8$$

a vybrali sme náhodne vrcholy 3 a 6, tak nová cesta bude vyzerať takto:

$$1,2,6,5,4,3,7,8$$

```
function constructNeighbourSolution(s,i,j)
{
    new = s.reverseInterval(i,j)
    return new
}
```

Obrázok 2.2: Generovania susedného riešenia v Simulated Annealing pre obchodného cestujúceho

Účelová funkcia je rovnako ako pri Ant Colony dĺžka cesty.

2.4 Nasadenie tabu Search

Metóda na hľadanie susedného riešenia je rovanká ako pri Simulated Annealing. Rozdiel medzi prechodom na susedné riešenie je, že zatiaľ čo Simulated Annealing vygeneruje náhodné riešenie a s istou pravdepodobnosťou ho akceptuje, tak Tabu Search vyberie najlepšie zo susedných riešenie, ktoré nie je v Tabu liste. Tento postup spôsobuje, že Tabu Search je oveľa pomalší, hlavne pri problémoch s veľkým počtom vrcholov.

2.5 Implementačné detaily

Zadanie problému popisuje dolná trojuholníková matica susednosti. V tejto matici sú prepočítané jednotlivé vzdialenosti medzi vrcholmi, aby sme ušetrili čas prepočítavania. Rovnakú maticu používam aj na udržovanie hodnôt feromónov.

Vytvorené riešenie udržujem v triede *Solution*. Samotné riešenie je reprezentované vektorom vrcholov.

Ant Colony si naviac udržuje doposiaľ nenavštívené vrcholy, pre rýchlejšiu konštrukciu riešenia.

Pre rýchlejší beh heuristík Simulated Annealing a Tabu Search sa negenerujú vždy celé riešenia. Pri oboch heuristikách totiž stačí vypočítať zmenu ceny riešenia oproti pôvodnému riešeniu. Na vypočítanie tejto zmeny nie je nutné riešenie konštruovať. Keďže sa jedná iba o otočenie intervalu, tak v celom riešení sa zmenia iba 4 dĺžky na okrajoch intervalu. Riešenie sa skonštruuje až po výbere, respektíve akceptácií riešenia.

2.6 Výkon jednotlivých metaheuristík

Jednotlivé metaheuristiky som otestoval na 35 problémoch, ktoré sú verejné v knižnici TSPLIB [2]. Pre niektoré z problémov je známe optimálne riešenie, ktoré je spočítané prejdením všetkých možností. Treba upozorniť na to, že tieto riešenie boli počítané, tak, že každá hrana má celočíselnú hodnotu. Kvôli zaokrúhľovaniu vzdialeností teda mohli uznať ako optimálne riešenie také, ktoré je v skutočnosti je o niečo dlhšie. Vo výsledkoch testovania je teda možné nájsť prípady, kde sa podarilo nájsť kratšiu cestu, ako je optimálna. Pričom pri zaokrúhlenom počítaní by táto cesta bola buď rovanko dlhá alebo dlhšia.

Pri testovaní som použil nasledovné nastavenie parametrov:

Parameter	**Hodnota**
Ant Colony	
α	1
β	6
Evaporation	0,4
Number of Ants	30
Number Of Iterations	400
Simulate Annealing	
Počiatočná teplota	10 000 000
Koeficient ochladzovania	0,99999
Tabu Search	
Veľkosť tabulistu	30
Počet iterácií	1000

Tabuľka 2.1: Nastavenie parametrov v TSP

Parametre v Ant Colony som nastavoval podľa článku [7]. Simulated Annealing som nastavil na základe doby trvania. V týchto parametroch totiž už zvyčajne predĺženie behu nespôsobilo zlepšenie riešenia, iba zvýšilo dĺžku výpočtu.

V nasledujúcej tabuľke je porovnanie jednotlivých metaheuristík. Ak v stĺci *OPTIMAL* nie je vyplnená hodnota, znamená to, že optimum pre tento problém nebolo vypočítané v euklidovskej metrike a teda pre nás nie je dôležité. Jednotlivé čísla pre metaheuristiky sú priemerným výsledkom desiatich pokusov. V prípade Tabu Search som každý problém testoval iba raz, nakoľko sa jedná o deterministický postup.

Instancia	**|V|**	**OPT**	**ACO**	**SA**	**TS**
burma14	14		32,5929	30,8785	32,1586
dj38	38		6942,215	6659,43	6659,43
att48	48	33523,7	36757,61	33802,68	35579,4
eil51	51	429,983	466,3363	435,2504	462,71
berlin52	52	7544,37	8310,892	7641,483	8494,71
pr76	76	108159	123571	109565,2	121231
gr96	96	512,309	575,7119	537,4599	542,58
kroA100	100	21285,4	23790,54	21595,46	23294,4
eil101	101	642,31	720,3938	679,1048	679,919
bier127	127	118282	126002,6	122161	124205
ch130	130	6110,86	6785,313	6346,57	6521,73
gr137	137		837,8579	762,091	746,236
ch150	150	6532,28	6873,118	6879,539	7037,49
u159	159	42080	47058,28	43308,27	42769,4
rat195	195	2323	2527,987	2573,116	2406,59
d198	198	15780	17046,96	16420,72	16390,6
gr202	202		544,8534	676,9927	513,771
gil262	262	2378	2762,83	2701,213	2593,83
a280	280	2586,77	3030,244	2818,62	2749,78
pr299	299	48135	56782,45	54448,84	53187,8
linhp318	318	41345	48436,44	47045,84	45661,2
pcb442	442	50783,5	60828,63	61062,06	56019,1
d493	493	35002	40570,46	41748,24	37767,7
att532	532		101197,4	105468	93392,9
ali535	535		2422,636	3267,775	2180,09
u574	574	36905	43913,41	40219,4	39209
rat575	575	6773	8026,731	9478,663	7454
d657	657	48912	57690,4	65457,18	53956,2
gr666	666		3712,524	5097,85	3347,59
u724	724	41910	50229,5	60682,8	46094,5
rat783	783	8806	10742,36	14600,04	9953,47
dsj1000	1000	18659688	22912667	27932167	23174500
pr1002	1002	259067	320571,4	349438	280714
d1291	1291	50801	59808,19	125302,6	58228,6
u1817	1817	57201	70713,62	71899,8	63746,5

Tabuľka 2.2: Výkon metaheuristík pre obchodného cestujúceho

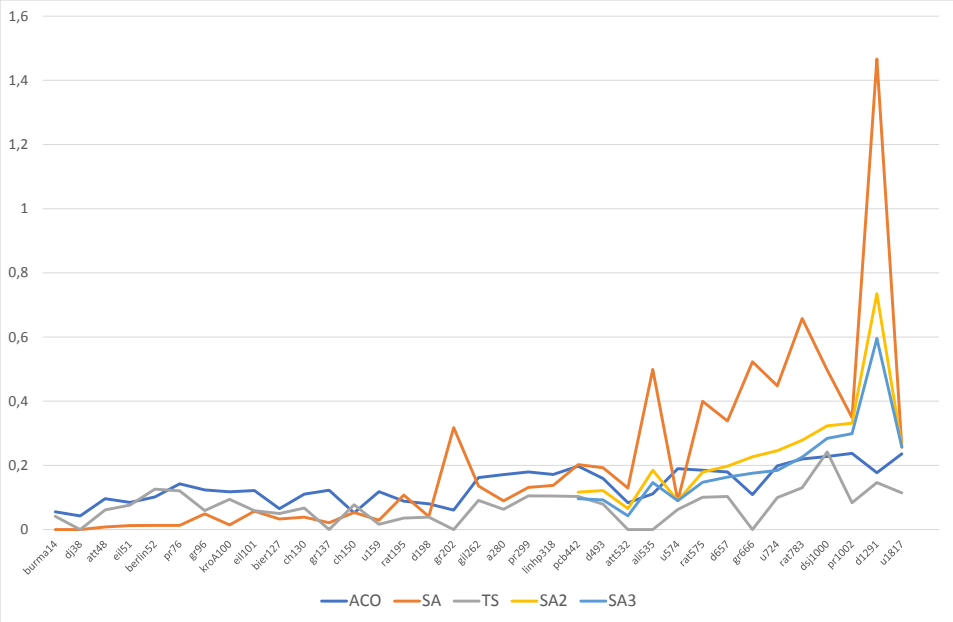
Ako vidieť z výsledkov, Simulated Annealing v prípadoch s vyšším počtom vrcholov výrazne zaostáva oproti zvyšným dvom metaheuristikám. Na týchto prípadoch síce beží veľmi krátko, no nepodarí sa mu v tak málo iteráciách prísť k dobrému riešeniu. Preto som Simulated Annealing na týchto väčších instanciách otestoval s rôznym ochladzovacím koeficientom. V nasledujúcej tabuľke sú uvedené výsledky, pričom $\alpha jeochladzovacíkoeficient$.

Je vidieť, že pri týchto instanciách sa podarilo zvýšením ochladzovacieho koeficientu zmenšiť dĺžku cesty. Pre $\alpha = 0{,}9999999$ bol už čas behu zhruba rovanko dlhý ako pri Ant Colony, preto som už ďalej parameter nezvyšoval.

Instancia	**|V|**	$\alpha = 0{,}99999$	$\alpha = 0{,}999999$	$\alpha = 0{,}9999999$
pcb442	442	61062,06	56717,98	55668,58
d493	493	41748,24	39255,12	38232
att532	532	105468	99494,66	97389,54
ali535	535	3267,775	2583,388	2498,882
u574	574	40219,4	40219,4	40219,4
rat575	575	9478,663	7986,208	7771,806
d657	657	65457,18	58582,86	56906,84
gr666	666	5097,85	4106,558	3934,846
u724	724	60682,8	52209,7	49635,18
rat783	783	14600,04	11259,54	10794,88
dsj1000	1000	27932167	24692220	23966220
pr1002	1002	349438	344924,8	336512,6
d1291	1291	125302,6	88145,58	81053,06
u1817	1817	71899,8	71899,8	71899,8

Tabuľka 2.3: Ladenie ochladzovacieho koeficientu, TSP

Výsledky sú v zhrnuté v nasledovnom grafe výkonnosti. Instancie v grafe sú zoradené podľa počtu vrcholov. V prípade, že pri instancii nepoznáme optimum, je ako optimum zvolený najlepší výsledok metaheuristík.



Obrázok 2.3: Graf výkonnosti metaheuristík pri obchodnom cestujúcom

V nasledujúcej tabuľke je zhrnutie priemerných odchýliek od optima. Opäť ak optimum nebolo známe, tak som ako optimum zvolil najlepší z výsledkov

testovaných heuristík. V prípade Simulated Annealing sú pri grafoch s vyšším počtom vrcholov použité výsledky s vyšším ochladzovacím koeficientom.

Heuristika	Priemerná odchýlka od optima
Tabu Search	7,29%
Simulated Annealing	11,72 %
Ant Colony	13,66 %

Tabuľka 2.4: Priemerné odchýlky heuristík pri probléme obchodného cestujúceho

2.7 Záver

Z výsledkov vidíme, že metaheuristika Tabu Search dopadla najlepšie. Na väčšine grafov poskytla táto heuristika najlepšie riešenie. Tomu zodpovedá aj najmenšia priemerná odchýlka.

Metaheuristika Simulated Annealing sa osvedčila hlavne na malých grafoch, kde v rýchlom čase našla najlepšie riešenie. S pribúdajúcimi vrcholmi však treba zvyšovať aj počet iterácií. Pri veľkých grafov sa jej ale nepodarilo dosiahnuť ani so zväčšenými parametrami tak dobrý výsledok, ako zvyšným dvom metaheuristikám. Je možné, že ďalšie ladenie parametrov by prinieslo lepší výsledok, no zároveň by predĺžilo beh na rádovo dlhší ako pri zvyšných metaheuristikách.

Ant Colony sa podarilo držať pomerne nízku chybovosť naprieč všetkými instanciami bez menenia parametrov. Beh tejto heuristiky bol dlhší ako Tabu Search a väčšinou aj jej výsledok bol o niečo horší. V porovnaní s heuristikou Simulated Annealing však vo veľkých grafoch dopadla dobre.

3. Ofarbovanie grafu

Ofarbovanie grafu je problém, kde každému vrcholu v grafe chceme priradiť farbu, tak aby žiadne dva susedné vrcholy nemali rovnakú farbu. Úlohou je minimalizovať počet použitých farieb. Problém sa dá použiť napríklad pri plánovaní, priradzovaní registrov procesoru alebo pri riešení sudoku.

Problém si môžeme prestaviť tak, že máme nejaké činnosti (vrcholy grafu) a tie potrebujeme rozvrhnúť na jednotlivé dni (farby). Lenže máme podmienky, ktoré hovoria, že niektoré dvojice činností (hrany) sa nemôžu vykonávať v rovnaký deň, napríklad preto, že máme iba jeden stroj, ktorý zvládne túto činnosť. Úlohou je všetky činnosti splniť za čo najmenej dní.

3.1 Popis problému

Majme graf $G = (V,E)$. Hľadáme ofarbenie vrcholov teda $c : V \to N$, kde V je množina vrcholov a prirodzené číslo reprezentuje farbu. Ofarbenie musí spĺňať:

$$\forall v_1, v_2 : (v_1,v_2) \in E \Rightarrow c(v_1) \neq c(v_2)$$

Cieľom je nájsť také c, aby počet použitých farieb bol čo najmenší.

Pre jednoduchšie vyjadrenie zaveďme pojem *konflikt*. Konfliktom nazveme dvojicu vrcholov, ktoré sú susedmi a majú rovnakú farbu. Je jasné, že správne ofarbenie nesmie obsahovať konflikty.

3.2 Nasadenie ant colony

Pre implementáciu Ant Colony na ofarbovanie grafu som zvolil postup z článku [9]. Pri ofarbovaní grafu nie je implementácia tak intuitívna, ako pri probléme obchodného cetujúceho.

Ant colony vždy konštruuje korektné riešenie. Postupuje teda tak, že postupne zafarbuje vrcholy jednou vybranou farbou, kým to ide. Začne na farbe číslo 0 a zafarbí ňou čo najviac vrcholov. Ak už nevieme nič zafarbiť prejde na ďalšiu farbu.

Mravce udržujú feromóny pre dvojicu farba a vrchol. Celkový feromón sa spočíta ako priemer týchto hodnôt, pre vrcholy, ktoré sme už touto farbou ofarbili. Vďaka vlastnostiam algoritmu, si môžeme číslo farby zároveň predstaviť ako kolo, v ktorom bude vrcholu priradená farba. Tým je zabezpečené, že mravce si pamätajú nielen poradie, v akom pre vrchol hľadajú farbu (vďaka dvojici vrchol farba), ale aj ktoré vrcholy spolu do jednej farby pasujú(vďaka počítaniu feromónu cez priemer už ofarbených). Miernou nevýhodou je fakt, že skupina vrcholov si nepamätá čistú informáciu, že bola spolu ofarbená, ale že bola spolu ofarbená v istom kole.

Pri každej novej farbe i si budeme udržovať množinu vrcholov, ktoré ešte môžeme touto farbou zafarbiť bez toho, aby vznikol konflikt. Nazvime túto množinu F_i. Na začiatku je F_i rovný množine všetkých nezafarbených vrcholov, pretože zatiaľ nemal ako vzniknúť možný konflikt. Vyberieme náhodný vrchol z množiny F_i a ofarbíme ho farbou i a z množiny F_i odstránime vrcholy, ktoré by po ofarbení farbou i boli v konflikte.

Postupne vyberáme ďalší vrchol podľa prechodovej funkcie. Ak práve ofarbujem farbou i tak vrchol $x \in F_i$ vyberiem s pravdepodobnosťou:

$$p_{ix} = \frac{\tau_{ix}^{\alpha} \cdot \eta_{ix}^{\beta}}{\sum_{z \in F_i} \tau_{iz}^{\alpha} \cdot \eta_{iz}^{\beta}}$$

Hladina feromónu τ_{ix} sa spočíta ako:

$$\tau_{ix} = \frac{\sum_{j \in V_i} t_{jx}}{|V_i|}$$

kde V_i je množina vrcholov, ktoré už sú ofarbené farbou i a t_{ix} je hladina feromónu prisúdená dvojici vrchol farba. Prirodzene ak $x \notin F_i$, tak $p_{ix} = 0$

Jediná rýchlo získateľná informácia o vrchole, ktorá nám môže poslúžiť v heuristickom koeficiente je stupeň vrcholu. V článku [3] sú zmienené tri možnosti voľby heuristického koeficientu:

- $\eta_{ix} = deg_W(i)$
- $\eta_{ix} = |F| - deg_F(i)$
- $\eta_{ix} = deg_{F \cup W}(i)$

kde $deg_X(i)$ je stupeň vrcholu i v grafe indukovanom množinou X a vrcholom i, F je množina neofarbených vrcholov, ktorá je stále vhodná pre farbu i a W je množina neofarbených vrcholov, ktoré už nie sú vhodné na ofarbenie farbou i.

Z týchto možností som si vybral $\eta_{ix} = |F| - deg_F(i)$, pretože vzhľadom na implemetáciu sa tento koeficient počítal najrýchlejšie.

Opäť po zafarbení vrcholov odstránim vrcholy, ktoré by mohli byť v konflikte a vrátim sa k vyberaniu ďalšieho vrcholu podľa prevdepodobnosti uvedenej vyššie. S ofarbovaním farbou i skončím, ak F_i je prázdna a prejdem na ofarbovanie farbou $i + 1$.

```
function generateSolution()
{
    solution.SetAllVerticesAsUncolored()
    uncoloredVertices<-vertices
    usedColors=0
    ColoredByColor[]<- empty
    while (!uncoloredVertices.isEmpty())
    {
        ++usedColors
        ColoredByColor[usedColors]<-empty()
        feasibleForColor <- uncoloredVertices
        chosenVertex <- chooseRandom(feasibleForColor)
        solution.SetColor(chosenVertex,usedColors)
        feasibleForColor.Remove(neighbours(chosenVertex))
        while (!feasibleForColor.isEmpty())
        {
            dist<-getDistribution(trailMatrix)
            vertex <-chooseWithDistribution(feasibleForColor,dist)
            solution.SetColor(vertex,usedColors)
            feasibleForColor.Remove(neighbours(vertex))
        }
    }
    return solution
}
```

Obrázok 3.1: Generovania riešenia v Ant Colony pre ofarbovanie grafu

3.3 Nasadenie hladných algoritmov

Na farbenie grafu sa hladné algoritmy celkom hodia. Všetky hladné algoritmy v každej iterácií vyberú jeden vrchol a ofarbia ho farbou s najnižším možným číslom, ktoré je ešte povolené. V čom sa prístupy líšia je, v akom poradí vrcholy na ofarbovanie vyberajú. V práci som použil nasledovné dva spôsoby zmienené v článku [6]

3.3.1 First Fit

Heuristika First Fit je veľmi rýchla a jednoduchá. Na začiatku máme všetky vrcholy bez farby. Následne postupne prechádzame vrcholy a priradíme im najnižšie možné číslo farby, teda také, že žiaden jeho sused nemá rovnaké.

3.3.2 Saturation Degree Ordering

V priradzovaní farby postupuje Saturation Degree Ordering rovnako ako First Fit. Teda keď vyberie vrchol, priradí mu najnižšiu možnú farbu. V čom sa líši od First Fit je poradie, v ktorom vrcholy vyberá. V každom kroku sa vyberie vrchol, ktorý má najviac rôzne ofarbených susedov a priradí mu farbu s najnižším

možným číslom. Vyberie teda vrchol, ktorý má okolo seba najviac použitých farieb a teda je pre neho komplikované nejakú vybrať.

Pre jednoduchosť budeme ďalej v texte túto heuristiku označovať SDO.

3.4 Nasadenie simulovaného žíhánia

Simulated Annealing je pri tomto probléme implementovaný klasicky. Jediný rozdiel je konštrukcia susedného a iniciálneho riešenia ako iniciálne riešnie sa použije riešenie skonštruované nejakým, z hladných algoritmov (popísané v predošlej podkapitole). V tejto práci som použil na skonštruovanie iniciálneho riešenia algoritmus Saturation Degree Ordering, pretože poskytoval lepšie výsledky ako First Fit.

Susedné riešenie [12] zavedieme jednoducho ako prefarbenie náhodného vrcholu na náhodnú farbu. Ak máme použité farby $1 \ldots n$, tak budeme vyberať z farieb $1 \ldots n+1$. Teda z rovnakou farbou vyberieme buď nejakú už použitú farbu alebo novú.

```
function constructNeighbourSolution(currentSolution)
{
    newSolution <- currentSolution
    if (HasConflict(currentSolution))
    {
        vertexToChange<-currentSolution.VerticesInConflict[random()]

    }
    else
    {
        vertexToChange <- random(1,currentSolution.NumberOfVertices)
            newColor <- random(1,currentSolution.NumberOfColorsUsed+1)
            newSolution.SetColor(vertexToChange,newColor)
    }

        return newSolution
}
```

Obrázok 3.2: Konštrukcia susedného riešenia v Simulated Annealing pre ofarbovanie grafu

Konštrukcia susedného riešenia sa pri ofarbovaní grafu mierne líši napríklad od problému obchodného cestujúceho. Skonštruované riešenie totiž nemusí byť korektné. To znamená, že môžu existovať dva susedné vrcholy, ktoré sú ofarbené rovnakou farbou a teda vznikol konflikt. Takéto riešenia nechceme úplne zakázať, pretože nám môžu pomôcť dostať sa z lokálneho minima a v konečnom dôsledku riešenie zlepšiť. Dôležité ale je takéto riešenie penalizovať a tak algoritmus usmerniť ku korektným riešeniam. Je nutné teda upraviť účelovú funkciu tak, aby udeľovala penalty za konflikty.

Zaveďme teda účelovú funkciu pre riešenie x takto:

$$c(x) = \chi_x + k(x) * |V|$$

kde χ_x je počet farieb použitých v riešení x a $k(x)$ je počet konfliktov,teda počet susedov, ktorí majú nejakú kolíziu. Z definície účelovej funkcie vidno, že akékoľvek riešenie s konfliktom, bude mať vyššiu hodnotu účelovej fukcie ako korektné riešenie. Korektné riešenie totiž môže mať prinajhoršom $|V|$ farieb, ak každý vrchol zafarbíme vlastnou farbou.

Keďže chceme zabrániť vznikaniu konfliktov, prioritne algoritmus odstraňuje konflikty. Teda ak sa v aktuálnom ofarbení nenachádza žiaden konflikt, prefarbíme náhodný vrchol podľa spomenutého postupu. Ak však v grafe je konflikt, tak budeme prefarbovať jeden z konfliktných vrcholov. Tým zabezpečíme, aby sme sa nevzdialili príliš od korektných riešení. Mohlo by sa totiž stať, že by po nejakom čase už neboli schopný vzniknuté konflikty napraviť.

3.5 Implementačné detaily

Pre uloženie zadania problému postačí matica susednosti. V mojej implemetácií som vytvoril triedu, ktorá reprezentuje dolnú trojuholníkovu maticu. Nakoľko sa jedná o neorientovaný graf nepotrebujem plnú maticu susednosti. Túto maticu používam ako pre uskladnenie hodnôt feromónov, tak pre poznamenanie susednosti.

Ďalšou triedou je trieda *Solution*, ktorá reprezentuje riešenie problému. Každá metaheristika má svoju vlastnú zdedenú triedu, na ktorej má dodefinované funkcie pre úpravu riešenia podľa danej metaheuristiky.

Ant Colony si pre *Solution* pamätá jednak farbu pre každý vrchol, ale zároveň má zoznam farieb a v ňom vrcholy ofarbené príslušnou farbou.

3.6 Výkon jednotlivých metaheuristík

Jednotlivé metaheuristiky som otestoval na 56 problémoch, ktoré sú verejné dostupné na webovej stránke [1] . Pre niektoré z problémov je známe optimálne riešenie, ktoré je spočítané prejdením všetkých možností.

Pri testovaní heuristík som použil nasledovné nastavenia parametrov:

Parameter	**Hodnota**
Ant colony	
α	2
β	5
Evaporation	0,4
Number of Ants	10
Number Of Iterations	500
Simulate annealing	
Počiatočná teplota	10 000
Koeficient ochladzovania	0,99

Tabuľka 3.1: Nastavenie parametrov pri ofarbovaní grafu

Parametre pri Ant colony som zvolil na základe testovania rôznych kombinácií parametrov a porovnávaní výsledkov, ktoré tieto parametre dávali. Počet iterácií

Ant Colony je nastavený tak, že viac iterácií už nedávalo vo väčšine prípadov lepšie výsledky ale spotrebuvávalo viac času. Parametre Simulated Annealing som zvolil na základe času behu. Pri testovaní rôznych parametrov sa riešenie nemenilo a parametre, ktoré spôsobovali rádovo dlhší beh ako Ant Colony, som neskúšal.

V nasledujúcej tabuľke je porovnanie jednotlivých metaheuristík. Metaheuristiku Simulated Annealing v tabuľke neuvádzam, nakoľko sa jej nepodarilo nikdy zlepšiť výkon SDO. Tejto metaheristike sa podarilo zlepšiť horšie iniciálne riešenia, no riešenie SDO bolo vždy natoľko dobré, ža sa jej ho nepodarilo opraviť. Ak v stĺci optimum nie je vyplnenené optimum znamená to, že optimum nie je známe. Jednotlivé čísla pre Ant Colony sú priemerným výsledkom desiatich pokusov. Zvyšné prístupy boli deterministické (okrem Simulated Annealing, ktorý neuvádzam).

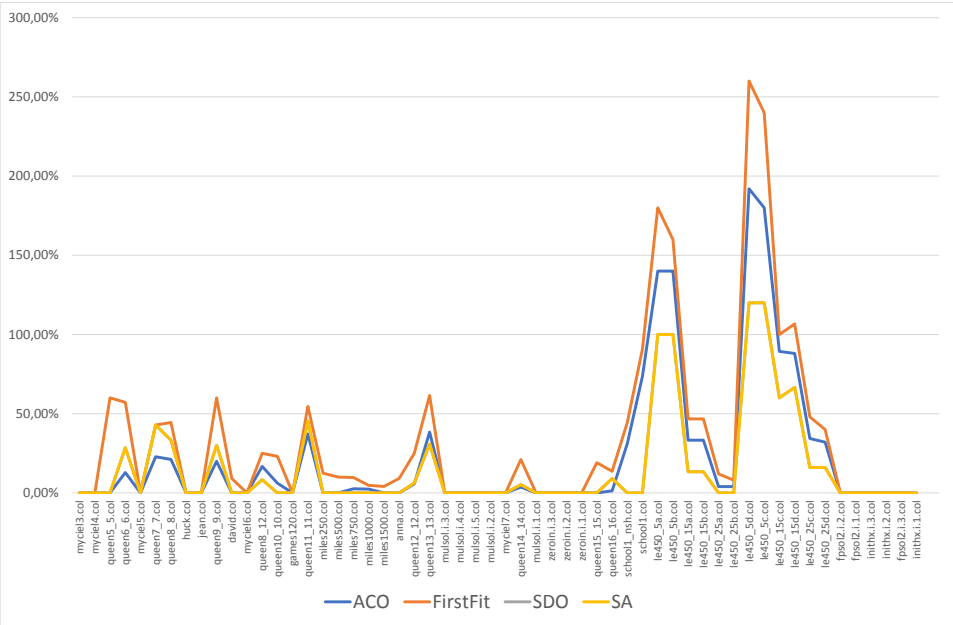
Instance	**Vertices**	**Edges**	**ACO**	**FirstFit**	**SDO**	**OPT**
myciel3.col	11	20	4,00	4,00	4,00	4,00
myciel4.col	23	71	5,00	5,00	5,00	5,00
queen5_5.col	25	160	5,00	8,00	5,00	5,00
queen6_6.col	36	290	7,90	11,00	9,00	7,00
myciel5.col	47	236	6,00	6,00	6,00	6,00
queen7_7.col	49	476	8,60	10,00	10,00	7,00
queen8_8.col	64	728	10,90	13,00	12,00	9,00
huck.col	74	301	11,00	11,00	11,00	11,00
jean.col	80	254	10,00	10,00	10,00	10,00
queen9_9.col	81	2112	12,00	16,00	13,00	10,00
david.col	87	254	11,00	12,00	11,00	11,00
myciel6.col	95	755	7,00	7,00	7,00	7,00
queen8_12.col	96	1368	14,00	15,00	13,00	12,00
queen10_10.col	100	2940	13,80	16,00	13,00	
games120.col	120	638	9,00	9,00	9,00	9,00
queen11_11.col	121	3960	15,10	17,00	16,00	11,00
miles1000.col	128	3216	43,00	44,00	42,00	42,00
miles1500.col	128	5198	73,00	76,00	73,00	73,00
miles250.col	128	387	8,00	9,00	8,00	8,00
miles500.col	128	1170	20,00	22,00	20,00	20,00
miles750.col	128	2113	31,80	34,00	31,00	31,00
anna.col	138	493	11,00	12,00	11,00	11,00
queen12_12.col	144	5192	16,90	20,00	17,00	
queen13_13.col	169	6656	18,00	21,00	17,00	13,00
mulsol.i.3.col	184	3916	31,00	31,00	31,00	31,00
mulsol.i.4.col	185	3946	31,00	31,00	31,00	31,00
mulsol.i.5.col	186	3973	31,00	31,00	31,00	31,00
mulsol.i.2.col	188	3885	31,00	31,00	31,00	31,00

Tabuľka 3.2: Výkon metaheuristík pre ofarbovanie grafu 1

Instance	**|V|**	**|E|**	**ACO**	**FirstFit**	**SDO**	**OPT**
myciel7.col	191	2360	8,00	8,00	8,00	8,00
queen14_14.col	196	8372	19,70	23,00	20,00	
mulsol.i.1.col	197	3925	49,00	49,00	49,00	49,00
zeroin.i.3.col	206	3540	30,00	30,00	30,00	30,00
zeroin.i.1.col	211	4100	49,00	49,00	49,00	49,00
zeroin.i.2.col	211	3541	30,00	30,00	30,00	30,00
queen15_15.col	225	10360	21,00	25,00	21,00	
queen16_16.col	256	12640	22,30	25,00	24,00	
school1_nsh.col	352	14612	35,40	39,00	27,00	
school1.col	385	19095	38,20	42,00	22,00	
le450_15a.col	450	8168	20,00	22,00	17,00	15,00
le450_15b.col	450	8169	20,00	22,00	17,00	15,00
le450_15c.col	450	16680	28,40	30,00	24,00	15,00
le450_15d.col	450	16750	28,20	31,00	25,00	15,00
le450_25a.col	450	8260	26,00	28,00	25,00	25,00
le450_25b.col	450	8263	26,00	27,00	25,00	25,00
le450_25c.col	450	17343	33,60	37,00	29,00	25,00
le450_25d.col	450	17425	33,00	35,00	29,00	25,00
le450_5a.col	450	5714	12,00	14,00	10,00	5,00
le450_5b.col	450	5734	12,00	13,00	10,00	5,00
le450_5c.col	450	9803	14,00	17,00	11,00	5,00
le450_5d.col	450	9757	14,60	18,00	11,00	5,00
fpsol2.i.2.col	451	8691	30,00	30,00	30,00	30,00
fpsol2.i.1.col	496	11654	65,00	65,00	65,00	65,00
inithx.i.3.col	621	13969	31,00	31,00	31,00	31,00
inithx.i.2.col	645	13979	31,00	31,00	31,00	31,00
fpsol2.i.3.col	864	8688	30,00	30,00	30,00	30,00
inithx.i.1.col	864	18707	54,00	54,00	54,00	54,00

Tabuľka 3.3: Výkon metaheuristík pre ofarbovanie grafu 2

V nasledujúcom grafe je znázornený výkon jednotlivých heuristík. Na x-ovej osi sú instancie problému usporiadané podľa počtu vrcholov v grafe a na y-novej osi odchýlka od optimáneho riešenia. Každá z čiar znázorňuje výkon jednej z heuristík. V niektorých prípadoch, nie je známe optimálne riešenie a v tomto grafe je potom optimálne riešenie stanovené ako minimum z výsledkov jednotlivých heuristík.



Obrázok 3.3: Graf výkonnosti metaheuristík pri ofarbovaní grafu

Z grafu vidíme, že najhoršie dopadla heuristika FirstFit, tá dosiahla vždy najhoršieho riešenia. Saturation degree ordering a Simulated Annealing mali v každom prípade rovnaký výkon. Táto dvojica bola väčšinou lepšia ako Ant Colony aj keď v pár prípadoch (hlavne instanciách z menším počtom vrcholov) sa podarilo Ant Colony dosiahnuť lepší výsledok.

V nasledujúcej tabuľke vidíme porovnanie priemerných odchýliek od optima (v prípadoch, kde nepoznáme optimum sme za optimum zvolili najnižší počet farieb, aký sa podarilo dosiahnuť).

Heuristika	Priemerná odchýlka od optima.
Saturation Degree Ordering	15,45%
Simulated Annealing	15,45%
Ant Colony	22,61%
First Fit	34,60%

Tabuľka 3.4: Priemerné odchýlky heuristík pri ofarbovaní grafu

3.7 Záver

Zo získaných dát je možné odpozorovať, že instancie problému sa dajú rozdeliť na dve skupiny. Pri jednej skupine dosiahli všetky heuristiky rovnaký a optimálny výsledok. Spomedzí testovaných instancií je táto skupina pomerne veľká. Druhou skupinou boli problémy, ktoré pri ktorých sa výsledky metaheuristík líšili.

Ako bolo očakávané najhoršie dopadla heuristika First Fit. Tá poskytuje síce extrémne rýchle riešenie, no v mnoho prípadoch použije aj značne väčší počet farieb.

Oproti nej heuristika Saturation Degree Ordering zohľadňujúca vlastnosti vrcholov pracuje oveľa lepšie. Často sa jej podarilo dosiahnuť veľmi dobrých výsledkov.

Metahauristika Simulated Annealing používala ako inicálne riešenie práve riešenia skonštruované pomocou SDO. Preto východzí bod tejto metaheuristiky bol často veľmi dobrý. Vzhľadom na fungovanie Simulated Annealingu, bolo pre metaheuristiku zrejme príliš ťažké preniesť sa z tohto lokálneho minima k lepším riešeniam. V žiadnom prípade sa metaheurstike Simulated Annealing nepodarilo nájsť lepšie riešenie ako SDO. Tento jav je pravdepodobne zapríčinený spôsobom výberu susedného riešenia. Simulated Annealing môže skonštruovať aj nekorektné riešenie. V momente ako tak spraví, sa ale primárne snaží ho opraviť. To ale spôsobuje, že sa riešenie príliš nezmení. Neznamená to ale, že Simulated Annealing pri ofarbovaní grafu nikdy nie je schopný výsledok zlepšiť. Ak som vložil heuristike ako východzie riešenie, riešenie skonštruované metódov First Fit, teda o niečo horšie, tak toto riešenie zvyčajne zlepšila. V každom prípade zlepšenie dosiahlo v najlepšom prípade rovnaký počet farieb ako riešenie vytvorené SDO.

Heuristika Ant colony si viedla vo väčšine prípadov viedla o niečo horšie ako SDO. Ant Colony pri rozhodovaní používa dve časti. Jednou sú feromóny a druhou heuristický koeficient. Feromóny v tomto prípade zohľadňujú v ktorom kole sa vrchol bude ofarbovať, pričom táto vedomosť nemusí byť vždy kľúčová. Zrejme aj preto Ant Colony nedosiahla takého výkonu ako SDO alebo Simulated Annealing. Jediné prípady, keď Ant Colony podala lepší výkon ako SDO boli menšie grafy, kde vyššie spomínaný faktor nemal zrejme taký vplyv.

Najlepšou heuristikou je teda Saturation Degree Ordering. Pri menších grafoch však dáva lepšie riešenia Ant Colony.

4. Množinové pokrytie

Problém množinového pokrytia (anglicky Set Covering) je problém, kde máme prvky a množiny, ktoré tieto prvky obsahujú. Úlohou je vybrať nejaké množiny, tak aby každý prvok bol obsiahnutý v aspoň jednej z vybraných množín. Tento problém sa často rieši pri plánovaní smien personálu, alebo pri pokrývaní klientov istou službou.

Problém si môžeme predstaviť na príklade rozmiestnenia pobočiek pošty. Pošta musí byť schopná zabezpečiť svoje služby po celej krajine a zároveň na to chce minúť čo najmenej peňazí. Majme nejaké miesta (prvky) a potom majme rôzne umiestnenia pobočiek pošty (množiny). Každé umiestnenie pobočky pokrýva službami niekoľko miest(prvky v danej množine). Samozrejme za každú pobočku treba platiť nájom (cena množiny). Hľadáme teda podmnožinu možných pobočiek, ktorá nás vyjde čo najmenej peňazí na prenájme, no zároveň táto sieť pobočiek pokryje službou všetky miesta.

4.1 Popis problému

Majme univerzum $\mathcal{U}$, v ktorom sa nachádza n prvkov. Ďalej majme množinu $\mathcal{S} = S_1, S_2, \ldots, S_m$, kde S_i je podmnožina univerza $\mathcal{U}$. Každá množina S_i má svoju cenu $cost(S_i)$. Cieľom je nájsť takú podmnožinu $\mathcal{C}$ množiny $\mathcal{S}$, že musí platiť:

$$\bigcup_{S \in \mathcal{C}} S = \mathcal{U}$$

Teda, že každý prvok je pokrytý (nachádza sa v) aspoň jednou množinou. Spomedzi vyhovujúich $\mathcal{C}$ sa snažíme nájsť také, ktoré minimalizuje účelovú funkciu:

$$cost(\mathcal{C}) = \sum_{S \in \mathcal{C}} cost(S)$$

4.2 Nasadenie Ant Colony

Ant colony v tomto prípade nevytvára žiadnu cestu, keďže poradie pridávania množín nie je podstatné. Pri množinovom pokrytí si môžeme predstaviť, že mravce sa snažia zapamätať si, ktoré množiny obsahovalo dobré pokrytie a tie zahŕňať pravdepodobnejšie.

Ako reprezentáciu problému použijeme maticu M, kde stĺpce reprezentujú množiny S_j a riadky reprezentujú jednotlivé prvky univerza. Prvok M_{ij} bude 1, v prípade, že prvok $i \in S_j$ a inak 0.

Algoritmus z článku [13] na začiatku nainicilaizuje zoznam nepoužitých množín a zoznam nepoužitých elementov univerza. Potom postupne do riešenia pridáva množiny tak, že vyberieme náhodne prvok univerza, ktorý ešte nebol pokrytý. Označme ho i. Potom vyberáme medzi množinami, ktoré pokrývajú i. Ani jedna z nich ešte nebola vybratá, lebo ak by bola, prvok by už bol pokrytý. V prípade, že podmienke vyhovuje viac množín, Ant Colony sa rozhodne náhodne. Pravdepodobnosť výberu podmnožiny j mravcom k je nasledovná:

$$p_{ij}^k = \frac{\tau_j^\alpha \cdot \eta_j^\beta}{\sum_{j \in N_i} \tau_j^\alpha \cdot \eta_j^\beta}$$

pričom predpokladáme, že ak množina j už bola vybratá mravcom k alebo neobsahuje i, tak $p_{ij}^k = 0$. Označnie η_i značí feromónovú stopu pre množinu S_j, η_j je heuristický koeficient pre množinu S_j a N_i je množina všetkých množín, ktoré pokrývajú prvok i. V mojej práci som ako η_j zvolil:

$$\eta_j = \frac{|U \cap S_j|}{cost(S_j)}$$

, kde U je množina nepokrytých prvkov.

Vo fázi aktualizovania feromónovej stopy sa prepočíta feromón takto

$$\tau_j = \rho\tau_j + \Delta\tau_j$$

kde ρ je vyparovací koeficient, ktorý je parametrom algorimtu a

$$\Delta\tau_j = \begin{cases} 1/\sum_{j \in Solution} cost(S_j), & \text{ak } j \in Solution \\ 0, & \text{ak } j \notin Solution \end{cases}$$

```
procedure generateSolution()
{

    solution <- empty()
    notUsedSets <- S
    notCoveredElements <- U

    while(!Empty(notCoveredElements))
    {
        x <- chooseRandomElement(notCoveredElements)
        probabilities <- empty
        foreach(s in notUsedSets)
        {
            if (M[x,s]==1) probablities[s]<-ACOformula(x,s)
        }
        solution += chooseBasedOnDistribution(probabilities)
    }
    return solution
}
```

Obrázok 4.1: Generovania riešenia v Ant Colony pre množinové pokrytie

4.3 Nasadenie simulovaného žíhání

Vzhľadom na nedostupnosť zdrojov, ktoré by popisovali tvorbu susedného riešenia Simulated Annealingu pri množinovom pokrytí, som zvolil priamočiaru a intuitívnu implementáciu. Ako spôsob tvorby sa ponúkali dve možnosti. Jedna by vedela odobrať alebo pridať množinu. Takáto metóda by však musela počítať s

tým, že pomerne často bude susedné riešenie nekorektné. S podobným postupom sme sa stretli pri ofarbovaní grafu. Druhým spôsobom je odstrániť jednu množinu. Po odstránení množiny nám môžu zostať nejaké nepokryté prvky. Vzápäti vyberieme nové množiny tak, aby tieto množiny pokryli nepokryté prvky. V prípade prvého spôsobu, ak by sme vytvorili nekorektné riešenie, bolo by potrebné aj tak v najbližších krokoch toto riešenie napraviť.

Implementoval som oba spôsoby, no ani v jednom prípade sa nepodarilo metaheuristike Simulated Annealing posunúť sa z iniciálneho riešenia. Túto metaheuristiku teda vo výsledkoch uvádzať nebudem.

4.4 Nasadenie Tabu Search

V dôsledku nedostatočných výsledkov metaheuristiky Simulated Annealing, bolo potrebné otestovať iný spôsob. Rozhodol som sa pre Tabu Search. Tabu Search bude mať podobnú konštrukciu susedného riešenia ako Simulated Annealing.

V každej iterácií vyberám najlepšie susedné riešnie, ktoré ešte nie je v Tabu liste. Opäť sa ponúkajú dva spôsoby hľadania susedného riešnia. Predstavme si, že by sme od suseda vyžadovali, aby to bolo korektné riešenie. Môžeme potom ako malú zmenu odstrániť jednu množinu a následne sa snažiť riešenie napraviť. To by vyžadovalo teda prejsť všetky množiny, ktoré potencinálne môžeme odstrániť a pre každú z týchto množín všetky možné opravy. Ako vidno už z tejto predstavy, toto riešenie bude pomalé. Tento spôsob som v práci testoval, ale čas spracovania bol rádovo vyšší ako Ant Colony.

Zvolil som preto druhú, jednoduchšiu implementáciu s penaltami. Susedné riešenie tu zadefinujeme tak, že sa líši v stave jednej množiny. Teda buď množina, ktorá bola v pôvodnom riešení tam už nie je, alebo množina ktorá pôvodne v pokrytí nebola v ňom teraz je. Je zjavné, že takto skonštruované riešenie nemusí byť korektné. V tom prípade stanovíme penaltu p takto:

$$p = M \cdot |U|$$

kde U je množina nepokrytých prvkov a M definujeme ako maximálnu možnú cenu riešenia, teda

$$M = \sum_{i \in \mathcal{S}} cost(S_i)$$

Túto penaltu následne pričítame k cene riešenia a dostávame tak účelovú funkciu. Vidíme, že ľubovoľné nekorektné riešenie je horšie, ako to korektné, čo nám pri výbere najlepšieho suseda zaručí, že smer riešenia bude smerovať ku korektnému pokrytiu.

```
function FindBestNeighbour(currentSolution){
    best<-empty()
    foreach(set in AllSets)
    {
        if (currentSolution.isSetInCover(set))
        {
            newSol = currentSolution.remove(set)
        }
        else
        {
            newSol = currentSolution.add(set)
        }
        if (!isInTabuList(newSol)
            && costWithPenalty(newSol)<costWithPenalty(best))
        {
            best = newSol
        }

    }
    return best
}
```

Obrázok 4.2: Konštrukcia susedného riešenia v Tabu Search pre množinové pokrytie

4.5 Nasadenie hladného algoritmu

Hladný algoritmus sa pokúša v každom kroku pridať čo najvýhodnejšiu množinu. Výhodnosť množiny vieme merať dvomi ukazovateľmi. Prvý ukazovateľ udáva koľko doposiaľ nepokrytých prvkov pridá množina do pokrytia a druhým prirodzeným ukazovateľom je cena množiny. Na základe popisu hladného algoritmuv článku [11] vieme vyrobiť účelovú fukciu $f(x)$, ktorá ohodnotí príspevok každej množiny x. V práci som ako účelovú funkciu použil:

$$f(x) = \frac{p_x}{c_x}$$

kde p_x je počet prvkov, ktoré množina pridá a c_x je cena množiny. Táto funkcia je rovnaká ako heuristický koeficient v metaheuristike Ant Colony.

V každom kroku teda vyberieme množinu s maximálnou hodnotou účelovej funkcie. Postup opakujeme dovtedy, kým niesú pokryté všetky prvky.

```
procedure Greedy
{
    Solution<-empty
    numberOfNotCovered=numberOfVertices
    while(numberOfNotCovered>0)
    {
        bestSet<-null
        foreach(s in Sets)
        {
            if(p(bestSet)/cost(bestSet)<p(s)/cost(s)) bestSet<-s
        }
        Solution.AddToSolution(s)
        numberOfNotCovered-=p(x)
    }
}
```

Obrázok 4.3: Hladný algoritmus pre množinové pokrytie

```
function p(Set s)
{
    sum=0
    foreach(element e in s)
    {
        if(!isElementCovered(e)) ++sum
    }
    return sum
}
```

Obrázok 4.4: Počet nepokrytých prvkov v Greedy algoritme

4.6 Implementačné detaily

Na popis danej instancie problému používam vo svojom kóde triedu *Problem*. Táto trieda v sebe obsahuje dve štruktúry. Obe sú implementované ako vektor vektorov. Prvá štruktúra je zoznam množín a pre každú množinu si pamätá zoznam prvkov, ktoré množina obsahuje. Druhá je zoznam prvkov, kde pre každý prvok máme zoznam množín, ktoré ho pokrývajú. Jednu informáciu teda držím duplicitne, ale vzhľadom na časovú náročnosť testovania, je vhodnejšie obetovať pamäť na dosiahnutie väčšej rýchlosti.

Ďalším dôležitým prvkom je trieda *Solution*. Táto trieda popisuje riešenie problému a je výstupom z jednotlivých metaheuristík. Pamätá si v boolean vektore, ktoré prvky už sú pokryté, pričom si drží číslo ešte nepokrytých prvkov zvlášť, pre rýchle zistenie, či je riešenie kompletné. Ďalej si ukladá zoznam vybratých množín, pre rýchle počítanie ceny aj boolean vektor množín, kde je popísané či množina v riešení je alebo nie. Tento vektor slúži na rýchle určenie, či sa množina nachádza v riešení.

Všetky metaheuristiky operujú na týchto dvoch triedach, pričom *Problem* sa neupravuje a *Solution* sa postupne modifikuje jej verejnými metódami.

4.7 Výkon jednotlivých metaheuristík

Pre testovanie som si vybral 75 problémov z knižnice problémov OR-Library [8]. Každý z týchto problémov som každou heuristikami Simulated Annealing a Ant Colony nechal vyriešiť 10-krát. Tabu Search a hladný algoritmus som na každom probléme pustil iba raz, keďže oba fungujú deterministicky.

Heuristiku Simulated Annealing vo výsledkoch ďalej neuvádzam na koľko ani v jednej z možných implementácií sa jej nepodarilo vylepšiť inciálne riešenie poskytnutné hladným algoritmom.

Predtým ako som spustil celkové testovanie, som na 10 vybraných problémoch odladil parametre jednotlivých metaheuristík. Najlepšie výsledky dávalo nasledovné nastavenie parametrov:

Parameter	**Hodnota**
Ant colony	
β	12
Evaporation	0,5
Number of Ants	30
Number Of Iterations	500
Tabu Search	
Počet iterácií	2000
Veľkosť tabu listu	100

Tabuľka 4.1: Nastavenie parametrov Ant Colony pri ofarbovaní grafu

V nasledujúcej tabuľke je priemer cien zo všetkých pokusov podľa jednotlivých heuristík. Ku každému problému je taktiež uvedený počet prvkov a počet množín. Stĺpec **Hustota** označuje pomer počtu množín k počtu prvkov. Podľa tohto stĺpca je tabuľka zoradená

Instancia	Množiny	Prvky	Hustota	ACO	TS	Greedy
scpclr13.txt	715	4095	0,1746	29,8	32	32
scpclr12.txt	495	2047	0,2418	28,7	31	32
scpclr11.txt	330	1023	0,3226	27,7	30	30
scpcyc11.txt	11264	28160	0,4	4931,2	4280	4304
scpclr10.txt	210	511	0,411	27,9	32	33
scpcyc10.txt	5120	11520	0,4444	2168	1916	1928
scpcyc09.txt	2304	4608	0,5	936	816	816
scpcyc08.txt	1024	1792	0,5714	394,7	352	364
scpcyc07.txt	448	672	0,6667	160,8	144	148
scpcyc06.txt	192	240	0,8	63,7	60	60
scp41.txt	1000	200	5	448,8	434	463
scp410.txt	1000	200	5	573,6	528	556
scp42.txt	1000	200	5	569,8	529	582
scp43.txt	1000	200	5	546,5	537	598
scp44.txt	1000	200	5	538,3	506	548
scp45.txt	1000	200	5	545,4	518	577
scp46.txt	1000	200	5	610	594	615
scp47.txt	1000	200	5	459,1	447	476
scp48.txt	1000	200	5	521,8	525	533
scp49.txt	1000	200	5	725,8	665	747
scp61.txt	1000	200	5	149,4	142	159
scp62.txt	1000	200	5	154,8	156	170
scp63.txt	1000	200	5	152,8	156	161
scp64.txt	1000	200	5	135,6	140	149
scp65.txt	1000	200	5	176,8	183	196
scpe5.txt	500	50	10	5	5	5
scp51.txt	2000	200	10	277,5	269	289
scp510.txt	2000	200	10	282,4	277	293
scp52.txt	2000	200	10	333,7	328	348
scp53.txt	2000	200	10	246	232	246
scp54.txt	2000	200	10	254,4	250	265
scp55.txt	2000	200	10	221,1	212	236
scp56.txt	2000	200	10	248,2	225	251
scp57.txt	2000	200	10	325,8	306	326
scp58.txt	2000	200	10	319,1	311	323
scp59.txt	2000	200	10	319,4	292	312
scpa1.txt	3000	300	10	267,7	261	288
scpa2.txt	3000	300	10	272,4	269	284
scpa3.txt	3000	300	10	246,9	245	270
scpa4.txt	3000	300	10	254,8	242	278
scpa5.txt	3000	300	10	250,3	247	271
scpb1.txt	3000	300	10	71,4	73	77
scpb2.txt	3000	300	10	80,4	78	86
scpb3.txt	3000	300	10	82,1	82	89
scpb4.txt	3000	300	10	82,6	83	89

Tabuľka 4.2: Výkon metaheuristík pri množinovom pokrytí

Instancia	Množiny	Prvky	Hustota	ACO	TS	Greedy
scpb5.txt	3000	300	10	73,6	75	78
scpc1.txt	4000	400	10	242,1	237	258
scpc2.txt	4000	400	10	240,4	224	258
scpc3.txt	4000	400	10	269,7	257	276
scpc4.txt	4000	400	10	243,1	237	257
scpc5.txt	4000	400	10	225,8	219	233
scpd1.txt	4000	400	10	63,6	67	74
scpd2.txt	4000	400	10	69,3	68	74
scpd3.txt	4000	400	10	76,3	79	83
scpd4.txt	4000	400	10	64,2	67	71
scpnre1.txt	5000	500	10	29	29	30
scpnre2.txt	5000	500	10	31,8	32	36
scpnre3.txt	5000	500	10	27,3	29	31
scpnre4.txt	5000	500	10	28,8	31	32
scpnre5.txt	5000	500	10	28	29	33
scpnrf1.txt	5000	500	10	14	16	16
scpnrf2.txt	5000	500	10	15	16	16
scpnrf3.txt	5000	500	10	14,6	17	17
scpnrf4.txt	5000	500	10	14	16	17
scpnrf5.txt	5000	500	10	13,8	15	16
scpnrg1.txt	10000	1000	10	195,8	189	203
scpnrg2.txt	10000	1000	10	167,5	161	182
scpnrg3.txt	10000	1000	10	182,2	181	192
scpnrg4.txt	10000	1000	10	186,3	180	191
scpnrg5.txt	10000	1000	10	185,8	181	194
scpnrh1.txt	10000	1000	10	68,8	69	76
scpnrh2.txt	10000	1000	10	68,4	73	74
scpnrh3.txt	10000	1000	10	64,4	63	65
scpnrh4.txt	10000	1000	10	62,6	66	69
scpnrh5.txt	10000	1000	10	58,2	61	63

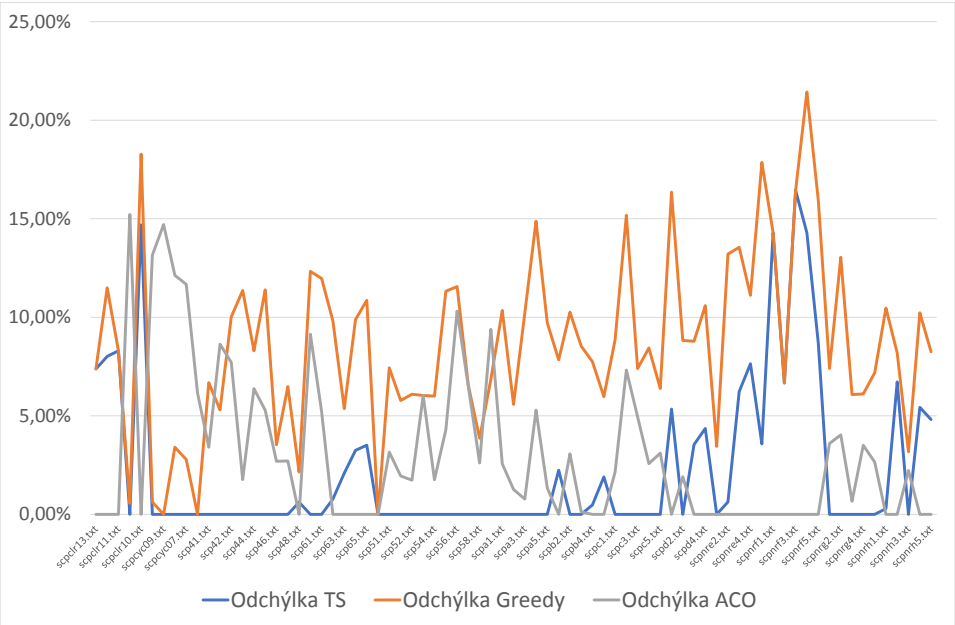
Tabuľka 4.3: Výkon metaheuristík pri množinovom pokrytí 2

Priemerná odchýlka jednotlivých heuristík ukazuje, že priemerne bol najlepší Tabu Search.

Heuristika	Priemerná odchýlka od optima.
Tabu search	2,16%
Ant colony	3,02%
Hladný algoritmus	8,61%

Tabuľka 4.4: Priemerné odchýlky heuristík pri množinovom pokrytí

V grafe je znázornená výkonnosť jednotlivých heuristík na problémoch. Na x-ovej osi sú instancie problému zoradené podľa hustoty, teda pomeru počtu množín k počtu prvkov. Na y-ovej osi je odchýlka od najlepšieho nájdeného riešenia, keďže optimálne riešeni v knižnici problémov uvedené neboli.



Obrázok 4.5: Graf výkonnosti metaheuristík pri množinovom pokrytí

Z grafu vidíme, že najhoršie dopadol hladný algoritmus. Tabu Search a Ant Colony sa striedajú. Aj keď v prvej časti grafu je úspešnejší Tabu Search, tak v druhej, s vyššou hustotou je úspešnejšia Ant Colony.

4.8 Záver

Hladný algoritmus podľa očakávaní dopadol najhoršie, čo sa týka kvality riešenia. Je vhodné ho použiť iba ak potrebujeme vo veľmi krátkom čase nájsť aj o dosť horšie riešenie. Ako je spomínané v predošlej podkapitole, Simulated Annealing nebol dostatočne dobrý na to, aby sa mu podarilo zlepšiť výsledky. Ako dve dobré metaheuristiky sa osvedčili Ant Colony a Tabu Search.

Ant Colony dosiahla vo väčšine instancií s vysokou hustotou najlepší výsledok. Silnú váhu v tejto heuristike mal heuristický koeficient, ktorý bol rovnaký ako pri hladnom algoritme. Ant Colony ale prisudzuje možnostiam pravdepodobnosť a teda preskúma väčší priestor, pričom si uchováva informáciu o tom, ktorá množina bola v pokrytí. Implementácia Ant Colony je v tomto prípade rovanko priamočiara ako pri probléme obchodného cestujúceho. Nositeľom feromónu totiž bola priamo časť riešenia.

Tabu Search bol najúspešnejší v instanciách z nižšou hustotou. Treba však dodať, že často spotreboval rádovo vyšší čas ako Ant Colony. Našlo sa však pár instancií, kde Tabu search, ale aj hladný algoritmus dostali lepšie riešenia. Jedná sa konkrétne o problémy s označením CYC. Tieto problémy sú konštruované tak, že prvky sú vrcholy n-dimenzionálnej kocky a množiny sú všetky 4-cykly v tejto

kocke, inak povedané dvojrozmerné steny. Pre tento druh problémov Ant Colony nie je vhodný.

Vo všeobecnosti dopadol o trochu lešie Tabu Search, aj keď jeho beh stojí viacej času.

Záver

V práci som porovnával 4 metaheuristiky na 3 problémoch. Nie vždy bolo možné jasne povedať, ktorá heuristika fungovala na danom probléme najlepšie. Pre lepšie pochopenie fungovania sa zvyčajne instancie problému dali rozdeliť podľa svojej charakteristiky do niekoľkých skupín. Na týchto skupinách sa dalo povedať, ktorá heuristika bola ako výkonná. Zvyčajne toto delenie spočívalo vo veľkosti instancie problému.

Metaheuristika Ant Colony sa neosvedčila pri ofarbovaní grafu, kde dosiahla nejlepší výsledok len v pár pokusoch. Naopak úspešná bola pri veľkých instanciách problémov obchodného cestujúceho a množinovom pokrytí. Jej veľkou výhodou bolo, že za pomerne dobrý čas vedela na týchto veľkých instatnciách dosiahnuť dobré riešenie.

Hladné algoritmy nám poskytujú veľmi rýchle riešenie, no pri väčšine problémov toto riešenie môže byť dosť vzdialené od optima. Takéto riešenie treba zvyčajne opracovať nejakou z metaheuristík Simulated Annealing, Tabu Search alebo aspoň klasickým horolezeckým algoritmom.

V problémoch, kde som použil Tabu Search zvyčajne dával dobré výsledky. Jeho veľká nevýhoda je prehľadávanie všetkých susedných riešení, čo pri veľkých instanciách môže spôsobovať dlhý čas výpočtu. To závisí od toho ako rýchlo vieme vypočítať zmenu hodnoty účelovej funckie pri prechode na susedné riešenie. Ak vieme hodnotu vypočítať jednoducho (ako pri probléme obchodného cestujúceho), tak je táto metaheuristika pomerne rýchla a spoľahlivá. Akonáhle je potrebné susedné riešenie vždy konštruovať alebo zložitejšie prepočítavať zmenu, stáva sa Tabu Search pomalým. Tabu Search sa hodí pre problémy, kde ako susedné riešenie môže skonštruovať aj nekorektné riešenie. Tieto riešenia budú totiž zvyčajne vylúčené pri výbere najlepšieho riešenia. K akceptácií nekorektného riešenia sa pristúpi iba v prípade, že už blízke okolie algoritmus prehľadal.

Simulated Annealing poskytuje druhé najrýchlejšie riešenia. Hneď po hladných algoritmoch. Tieto riešenia zvyčajne dosahujú dobrý výsledok. V práci som však zistil, že metaheuristike Simulated Annealing sa zvyčajne nedarí pracovať s riešeniami, ktoré nie sú korektné. To znamená, že nevie efektívne opravovať konflikty v riešení. Preto zvyčajne nedosiahla lepší výsledok ako iniciálne riešenie. Ako dobrá sa ukázala jedine pri probléme obchodného cestujúceho, kde pri menších instanciách dosahovala najlepšie a najrýchlejšie výsledky.

Zoznam použitej literatúry

[1] URL http://mat.gsia.cmu.edu/COLOR/instances.html#XXDSJ.

[2] URL http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/index.html.

[3] Anirudh Shekhawat, Pratik Poddar, D. B. (2009). Ant colony optimization algorithms :introduction and beyond. URL http://mat.uab.cat/~alseda/MasterOpt/ACO_Intro.pdf.

[4] Basu, S. (2012). Tabu search implementation on traveling salesman problem and its variations: A literature survey. **02**.

[5] Blum, C. (2005). Ant colony optimization: Introduction and recent trends. *Physics of Life Reviews*, **2**, 353–373. doi: https://doi.org/10.1016/j.plrev.2005.10.001.

[6] Dr. Hussein Al-Omari, K. E. S. (2006). New graph coloring algorithms. *American Journal of Mathematics and Statistics 2*, pages 739–741. ISSN 1549-3636.

[7] Gaertner, D. a Clark, K. (2005). On optimal parameters for ant colony optimization algorithms. **1**, 83–89.

[8] J.E.Beasley. OR-library. URL http://people.brunel.ac.uk/~mastjjb/jeb/orlib/scpinfo.html.

[9] Kathryn A.Dowsland, J. M. (2008). An improved ant colony optimisation heuristic for graph colouring. *Discrete Applied Mathematics*, **156**, 313–324. doi: https://doi.org/10.1016/j.dam.2007.03.025.

[10] Schneider, T. W. (2014). The traveling salesman with simulated annealing, r, and shiny. URL http://toddwschneider.com/posts/traveling-salesman-with-simulated-annealing-r-and-shiny/.

[11] Stern, T. URL http://math.mit.edu/~goemans/18434S06/setcover-tamara.pdf.

[12] Szymon Łukasik, Z. K. a Świętoń, G. *Parallel Simulated Annealing Algorithm for Graph Coloring Problem*. Springer.

[13] Zhi-Gang Ren, Zu-Ren Feng, L.-J. K. Z.-J. Z. (2010). New ideas for applying ant colony optimization to the set covering problem. *Computers Industrial Engineering*, **58**, 774–784. doi: https://doi.org/10.1016/j.cie.2010.02.011.

Zoznam obrázkov

1.1 Metaheuristika Ant Colony 6
1.2 Metaheuristika Simulated Annealing 7
1.3 Metaheuristika Tabu Search 8

2.1 Generovania riešenia v Ant Colony pre obchodného cestujúceho . 10
2.2 Generovania susedného riešenia v Simulated Annealing pre obchodného cestujúceho 10
2.3 Graf výkonnosti metaheuristík pri obchodnom cestujúcom 14

3.1 Generovania riešenia v Ant Colony pre ofarbovanie grafu 18
3.2 Konštrukcia susedného riešenia v Simulated Annealing pre ofarbovanie grafu 19
3.3 Graf výkonnosti metaheuristík pri ofarbovaní grafu 23

4.1 Generovania riešenia v Ant Colony pre množinové pokrytie 26
4.2 Konštrukcia susedného riešenia v Tabu Search pre množinové pokrytie 28
4.3 Hladný algoritmus pre množinové pokrytie 29
4.4 Počet nepokrytých prvkov v Greedy algoritme 29
4.5 Graf výkonnosti metaheuristík pri množinovom pokrytí 33

Zoznam tabuliek

2.1 Nastavenie parametrov v TSP 12
2.2 Výkon metaheuristík pre obchodného cestujúceho 13
2.3 Ladenie ochladzovacieho koeficientu, TSP 14
2.4 Priemerné odchýlky heuristík pri probléme obchodného cestujúceho 15

3.1 Nastavenie parametrov pri ofarbovaní grafu 20
3.2 Výkon metaheuristík pre ofarbovanie grafu 1 21
3.3 Výkon metaheuristík pre ofarbovanie grafu 2 22
3.4 Priemerné odchýlky heuristík pri ofarbovaní grafu 23

4.1 Nastavenie parametrov Ant Colony pri ofarbovaní grafu 30
4.2 Výkon metaheuristík pri množinovom pokrytí 31
4.3 Výkon metaheuristík pri množinovom pokrytí 2 32
4.4 Priemerné odchýlky heuristík pri množinovom pokrytí 32

A. Prílohy

A.1 Používateľská dokumentácia pre spustenie programov na testovanie metaheuristík

Pre všetky tri problémy som vyrobil testovacie programy, ktoré majú rovnaké uživateľské prostredie. Prostredie je jednoduché, keďže sa jedná o čisto výpočtové programy.

Spúšťa sa Source.cpp, ktorý vo vstupných súboroch nájde potrebné informácie.

Súbor *problems.txt* má na každom riadku názov súboru v ktorom je jedna instancia. V rovnakom adresári je vytvorený adresár "instances", kde sú uložené instancie v tvare podľa daného problému. Ak sa pri parsovaní problému vyskytne chyba, testovaný problém sa preskočí.

Špeciálne v prípade problému obchodného cestujúceho môžu pre každú instanciu problému existovať dva súbory. Jeden má príponu *.tsp* a je povinný. Z tohto súboru sa naparsuje samotný problém, teda vzdialenosti jednotlivých bodov. Ďalej voliteľne môže byť v rovnakom priečinku uložený aj súbor s rovnakým menom a príponov *.opt.tour*. Tento súbor popisuje optimálnu cestu. Pri probléme obchodného cestujúceho teda v súbore *problems.txt* uvádzame iba názov súboru bez prípon. Ak sa teda na riadku nachádza

```
subor1
```

tak program vyhľadá *subor1.tsp* a skúsi k nemu nájsť aj *subor1.opt.tour*.

A.1.1 Nastavenie parametrov

Niektoré z testovaných algoritmov majú parametre, tie sa dajú nastaviť v konfiguračnom súbore *params.conf* sú popísané v tvare názov

```
názov_parametru=hodnota
```

V prípade, že nie je dodržaný vzor písania parametrov, neexistuje záznam o parametri, alebo sa vyskytne chyba pri čítaní súboru, tak sa ako parameter volí východzia (prednastavená) hodnota. Parametre sú:

- Alpha,Beta - parametre Ant Colony
- Number_of_ants, Number_of_iterations - parametre Ant Colony, koľko iterácií sa má vykonávať a koľko mravcov má precovať.
- Cooling_coeficient, temperature - parametre Simulated Anneailng
- Number_of_tests - udáva koľko krát sa má každá instancia problému pretestovať. Tento paramater sa používa pri nedeterministických heuristikách Ant Colony a Simulated Annealing.
- Number_of_iterations_tabu, Tabu_size - parametre Tabu Search udávajúce počat iterácií a veľkosť Tabu listu

Príklad súboru *"params.conf"*

```
Alpha=2
Beta=5
Cooling_coeficient=0.99
Temperature=10000
Number_of_ants=10
Number_of_iterations=10
Number_of_tests=10
```

A.1.2 Výstup

Program ako výstup generuje *output.csv* súbor do ktorého vždy len pridáva záznamy, neprepisuje ho. Záznamy sú v tvare:

```
Názov instancie;Použitá metóda;Hodnota cieľovej funkcie;Čas behu
```

Programy generujú do *output.csv* aj bočné informácie. Konkrétne vždy pri spustení s novými parametrami, sa do súboru vygeneruje aktuálny stav parametrov. Teda pri každom novom spustení sa ako prvý záznam do *output.txt* vygeneruje zoznam parametrov z ich hodnotami a následne sa až pridávajú výsledky testov vo vyššie uvedenom tvare. Pri množinovom pokrytí sa taktiež generujú rozmery problému.

A.2 Príloha na CD

Na CD v prílohe je možné nájsť zdrojové kódy pre jednotlivé programy, testované instancie problémov a výstupy z testovaní. Súbory sú delené podľa problému ku ktorému patria, kde pri každom probléme sú tri priečinky:

- instances - instancie problému, na ktorých boli metaheuristiky testované.
- code - zdrojové kódy
- output - výstupy programov