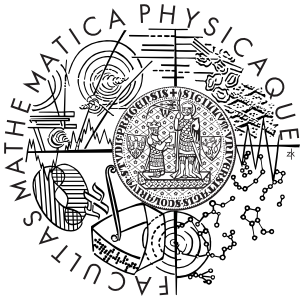
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Luboš Turek

Použití ACO algoritmu na řešení jednoduché substituční šifry

Katedra algebry

Vedoucí bakalářské práce: doc. RNDr. Jiří Tůma, DrSc.
Studijní program: Obecná Informatika

2012

Na tomto místě bych rád poděkoval vedoucímu této bakalářské práce Jiřímu Tůmovi za jeho ochotu, trpělivost a odborné rady. Velké poděkování patří Lubomíru Bulejovi za další podněty, nápady a hodnotné připomínky.

Prohlašuji, že jsem svou bakalářskou práci napsal(a) samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 23. května 2012 Luboš Turek

Obsah

Název práce: Použití ACO algoritmu na řešení jednoduché substituční šifry
Autor: Luboš Turek
Katedra (ústav): Katedra algebry
Vedoucí bakalářské práce: doc. RNDr. Jiří Tůma, DrSc.
e-mail vedoucího: jiri.tuma@mff.cuni.cz

Abstrakt: V předložené práci studujeme kombinatorickou metaheuristiku Ant Colony Optimization a zkoumáme možné způsoby jejího použití k prolomení jednoduché substituční šifry. Součástí práce je návrh a implementace programu. Tento program je srovnán s genetickým algoritmem.

Klíčová slova: jednoduchá substituční šifra, ant colony optimization, ACO, kryptologie, kryptografie

Title: Application of ACO to simple subsitution ciphers
Author: Luboš Turek
Department: Department of Algebra
Supervisor: doc. RNDr. Jiří Tůma, DrSc.
Supervisor's e-mail address: jiri.tuma@mff.cuni.cz

Abstract: In the present work we study combinatorial metaheuristic Ant Colony Optimization and we search for its application to the problem of cracking simple subtitution cipher. Functional implementation is a part of the thesis. The program is compared to genetic algorithm.
Keywords: simple substitution cipher, ant colony optimization, ACO, cryptology, cryptography

Kapitola 1

Úvod

Jednoduchá substituční šifra, jejímuž prolamování se v této práci věnujeme, byla popsána již v Kámasútře [24], nicméně používána byla o stovky let dříve. Frekvenční analýza, velmi spolehlivá metoda k prolomení této šifry, byla popsána v arabských textech již v devátém století našeho letopočtu [21]. Je zřejmé, že použití této šifry v dnešní době, pokud to s bezpečností myslíme alespoň trochu vážně, je naprosto nemyslitelné.

Přesto má výzkum nových metod prolamujících klasické šifry stále velký význam, jelikož moderní šifry staví na základech klasické kryptografie. Dnes velmi rozšířená šifra Data Encryption Standard (DES) v podstatě používá jen velmi jednoduché operace: substituce, permutace (transpozice) a bitový XOR [5]. Klasické šifry (kromě jednoduché substituce zejména transpoziční šifra a polyalfabetická šifra) jsou často používány jako první meta při zkoušení nových metod kryptoanalýzy.

Ant Colony Optimization (nebo také ACO), metaheuristika, kterou se v práci zabýváme, je poměrně nová technika inspirována chováním mravenců při hledání potravy. Na některé problémy, jako třeba SOP[1] nebo OSSP[2], je ACO dokonce tou nejlepší známou metodou [9]. Možnosti použití ACO v kryptologii zatím příliš zkoumány nebyly. Našel jsem pouze článek popisující útok na transpoziční šifru [20].

Téma této práce spadá do průniku obou oborů. V tomto textu se zabýváme využitím ACO k prolomení jednoduché substituční šifry. Následující dvě kapitoly proto věnujeme popisu těmto tématům. Ve čtvrté kapitole se zabýváme tím, jak „našroubovat“ problém prolomení jednoduché substitu-

[1] Sequential Ordering Problem

[2] Open Shop Scheduling Problem

ční šifry na metaheuristiku ACO. Součástí práce je i implementace takto navrženého systému. Technické detaily této implementace jsou popsány v páté kapitole, zatímco práce s programem (uživatelská dokumentace) je popsaná v příloze A. V šesté kapitole hledáme ideální parametry simulace. Dále použijeme implementovaný program a srovnáme ho s genetickým algoritmem, který je popsán v příloze B. V závěrečné sedmé kapitole výsledky vyhodnotíme a navrhneme vhodnou návaznost na bakalářskou práci.

1.1 Používané pojmy

Dále bych tento úvod rád využil k definování některých pojmů z kryptografie, které se v dalším textu objevují a jejich význam nemusí být zcela zřejmý.

Abeceda je množina písmen (znaků). Množinu velkých písmen anglické abecedy $\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z\}$ nazýváme **anglickou abecedou**[3].

Otevřený text je text zprávy před zašifrováním.

Šifrový text je otevřený text po zašifrování.

N-gram je N-tice sousedních písmen. Speciálně **bigram** a **trigram** jsou dvojice, resp. trojice sousedních písmen.

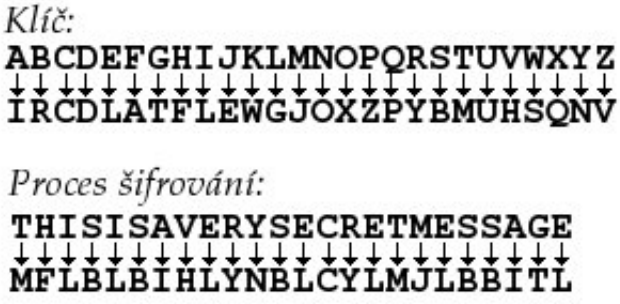
[3] V této práci se budeme zabývat pouze anglickou abecedou. Všechny použité principy jsou však obecně použitelné.

Kapitola 2

Jednoduchá substituční šifra

Substituční šifrou se obecně rozumí metoda šifrování textu taková, při níž se množina znaků otevřeného textu zaměňuje (substituuje) za jinou množinu znaků - může to být znak ze stejné abecedy, bigram, trigram, znak z naprosto odlišné abecedy nebo kombinace všech zmíněných postupů.

Speciálním případem je jednoduchá substituční šifra (někdy také nazývaná monoalfabetická substituční šifra), které se budeme věnovat v celém textu této práce. Taková šifra zaměňuje každé jedno písmeno otevřeného textu za jedno písmeno šifrové abecedy podle pevně dané substituční tabulky. i-té písmeno otevřené abecedy je vždy zobrazeno na i-té písmeno klíče. Šifruje se po písmenech. Zašifrování textu $THISISAVERYSECRETMESSAGE$ pomocí klíče $IRCDLATFLEWGJOXZPYBMUHSQNV$ je znázorněno na obrázku 2.1.



Obrázek 2.1: Proces zašifrování zprávy

2.1 Speciální případy

Speciálními případy jsou Ceasarova šifra nebo Atbaš. Ceasarova šifra zaměňuje každé písmeno za takové písmeno, které se nachází v abecedě o pevně daný počet pozic dále. Počet možných klíčů je tedy pouze 26, z čehož jeden klíč (identita) není zrovna účinný. V případě Atbaše je klíčem otevřená abeceda uspořádaná pozpátku. Tedy, v případě angické abecedy je situace následující:

$$K = (ZYXWVUTSRQPONMLKJIHGFEDCBA)$$

Zajimavá vlastnost šifry Atbaš je, že šifrovací klíč je zároveň dešifrovacím klíčem.

2.2 Prolamování jednoduché substituční šifry

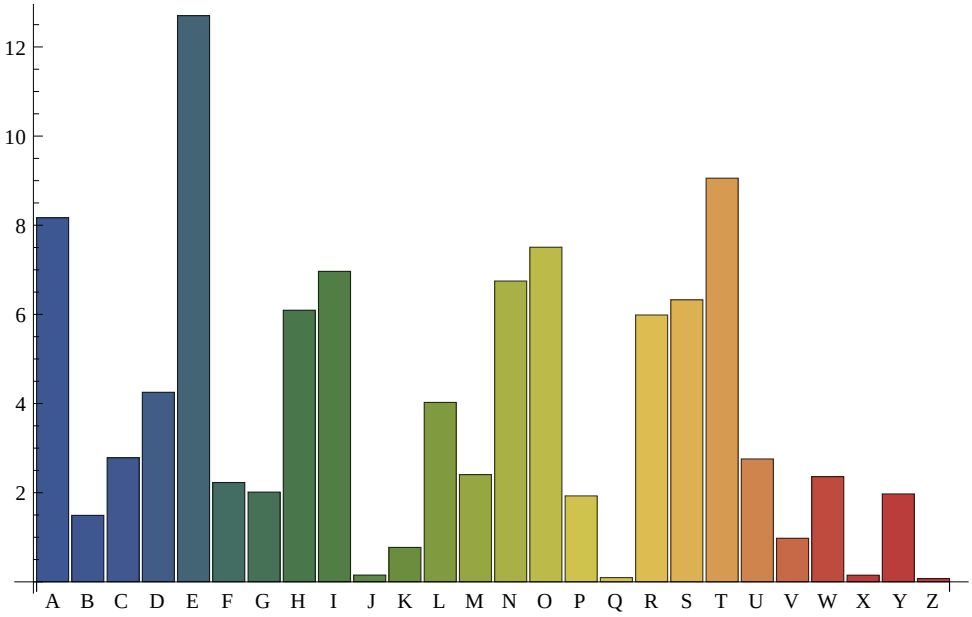
Pro anglickou abecedu čítající 26 znaků je počet všech možných klíčů $26! = 403291461126605635584000000 \approx 4 * 10^{26,6} \approx 2^{88,4}$. I se současnou technologií je nemožné projít všechny možné klíče. Dejme tomu, že by současná výpočetní technika dokázala ověřit správnost klíče za jednu milisekundu. I přes to by zkoušení všech klíčů trvalo přes 10^{16} let, což je 932813 krát víc než odhadované stáří vesmíru.

Při prolamování jednoduché substituční šifry se využívá následujícího [14]:

- Frekvence, jak písmen, tak n-gramů zůstávají zachovány. Tohoto využívá většina počítačových útoků.
- Samohlásky, a hlavně souhlásky, se vyskytují ve speciálních vzorcích.
- Většina anglických textů používá velmi omezený slovník a frekvence slov je nerovnoměrná. Nejčastěji používaná slova se v textech objevují stále.
- Nepravidelné je řetězení slov. Některé dvojice slov po sobě následují velmi často (např. „SHE IS“), jiné nikdy (např. „SHE ARE“)
- Jazyky mají složitou gramatiku povolující jen určité množství konstrukcí. Je však velmi těžké naučit počítač porozumět hlubším gramatickým konceptům. Toto je tedy doména lidských luštitelů.

2.3 Frekvenční analýza

Frekvenční analýza je algoritmus, který se snaží najít správný klíč a rozluštit šifrový text. Využívá toho, že substituční šifra zachovává rozložení frekvence znaků. Nejprve předpokládáme, že otevřený text je v nějakém určitém jazyce (např. v angličtině). Budeme potřebovat typickou frekvenci písmen v tomto jazyce. Tu buď získáme z nějakého zdroje anebo ji zjistíme sami počítáním jednotlivých znaků v dostatečně dlouhém textu. Dále spočítáme frekvenci znaků v šifrovém textu. Dojdeme k přesvědčení, že znak šifrového textu s i-tou nejvyšší frekvencí je obrazem znaku s i-tou nejvyšší frekvencí v předpokládaném jazyce.



Obrázek 2.2: Frekvence jednotlivých písmen v anglickém textu v procentech [17]

Na delších textech dosahuje frekvenční analýza velmi dobrých výsledků. Klíč je buď správný nebo se mu alespoň velmi podobá, a následně není velký problém správný klíč ručně dohledat.

Častá chyba frekvenční analýzy je, že zamění písmena, jejichž frekvence je podobná. V angličtině je to například I a N. Všimněme si, že jedno je souhláska a druhé samohláska, tudíž se dá předpokládat, že budou v otevřeném textu zpravidla stát vedle rozličných pismen. Toto je idea následujícího

algoritmu.

2.4 Jakobsenův algoritmus

Velmi spolehlivý algoritmus prolamující jednoduchou substituční šifru publikoval Thomas Jakobsen [15]. Algoritmus začíná s nějakým klíčem a ten postupně vylepšuje.

Původní klíč může být jakýkoliv, avšak čím lepší původní odhad je, tím dříve algoritmus konverguje k řešení. Proto je dobré začít s rozumným odhadem, který můžeme získat například frekvenční analýzou.

Jakmile máme nějaký klíč, algoritmus zkusí v klíči prohodit dvě písmena. Pokud je takto získaný klíč horší než původní, zkusí se prohodit jiná dvojice písmen. Pokud se již každá dvojice písmen zkusila prohodit, algoritmus končí, a aktuální klíč je výsledný klíč.

Abychom mohli porovnat kvalitu dvou klíčů, potřebujeme funkci, která nám pro daný text T řekne, jak moc se podobá textu v požadovaném jazyce. Jakobsen ve svém algoritmu používá funkci založenou na porovnávání frekvence bigramů popsanou v 4.1.2.

Jestliže si uvědomíme, že jde vlastně o lokální prohledávání (local search), je zřejmý i největší problém Jakobsenova algoritmu. Jako gradientní metoda se může „zaseknout“ v lokálním minimu.

2.5 Další metody

Ke strojovému prolomení substituční šifry se často používají různé kombinatorické metaheuristiky. Často se používají právě ty biologicky motivované. V literatuře je popsán i útok technikou *Scatter Search* [13] nebo *Tabu Search* [23]. Persi Diaconis použil k prolomení metodu Markov Chain Monte Carlo (MCMC) [7]. Úspěšně byly použity i genetické algoritmy [23].

Konkrétně mravenčí kolonie však byly použity k prolomení transpoziční šifry [20]. Jejich použití k prolomení substituční šifry, pokud je mi známo, zatím nikde popsáno není.

Kapitola 3

Algoritmy mravenčí kolonie

Techniku představil Marco Dorigo ve své dizertační práci [8]. Ant Colony Optimization (dále ACO) je metaheuristika, využívající množství agentů (mravenců) k řešení kombinatorických problémů. Tito agenti komunikují nepřímo skrz prostředí.

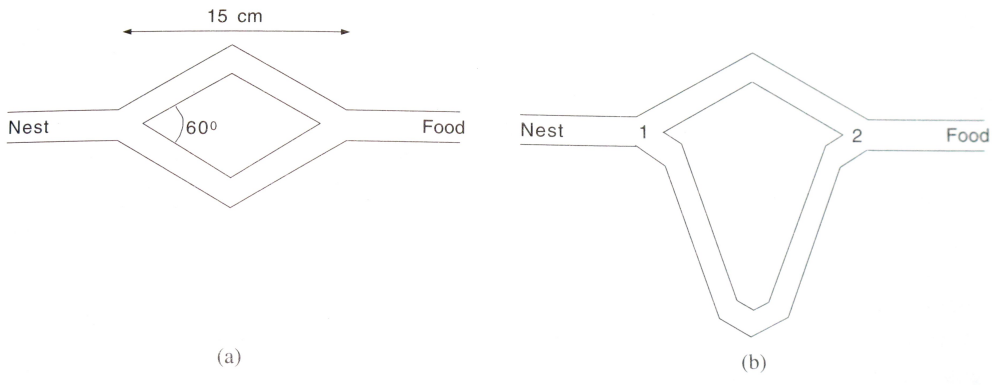
3.1 Biologická inspirace

Některé druhy mravenců mají velmi primitivní zrak, jiné druhy jsou slepé. Přesto mají velmi dobrou schopnost nalézt potravu a dopravit ji do mraveniště po efektivní cestě. Mravenci nekomunikují přímo, ale mění prostředí, a jejich vlastní chování záleží na stavu prostředí. Tento princip nepřímé komunikace se nazývá *stigmergy*.

Konkrétně mravenci k této komunikaci využívají feromony. Mravenec opouštějící mraveniště náhodně prohledává prostor a na své cestě nechává určité množství feromonu. Poté, co nalezne potravu, se stejnou (popřípadě kratší) cestou vrátí. Na této cestě je pak dvojité množství feromonu. Tento feromon stimuluje ostatní mravence k tomu, aby použili stejnou cestu.

Popsané chování umožňuje mravencům najít nejkratší cestu. Tento princip byl ukázán tzv. Double Bridge Experimentem [6] znázorněným na obrázku 3.1.

Mezi mraveništěm a potravou vedou dvě cesty. V prvním případě (a) jsou obě cesty stejně dlouhé. V takto nastaveném prostředí mravenci vyjdou z mraveniště hledajíce potravu. Po chvíli narazí na místo, kde se musí rozhodnout, po které cestě se vydají. Ani na jedné cestě není v tuto chvíli feromonová stopa, a tak si cestu zvolí náhodně. Jedna cesta je zvolena více



Obrázek 3.1: Double bridge experiment [12]. (a) Ramena mají stejnou délku. (b) Ramena mají různou délku.

mravenci než ta druhá a tak se na ní nachází více feromonu. Tím se zvyšuje pravděpodobnost, že tuto cestu si vyberou další mravenci, čímž se opět zvýší množství feromonů na této cestě. Tímto způsobem mravenci konvergují k jednomu řešení, kdy téměř všichni používají jednu cestu.

V druhém případě (b) mají cesty z mraveniště k potravě rozdílnou délku. Když se první mravenci dostanou na rozcestí, někteří si zvolí kratší cestu a někteří delší. Ti, kteří si zvolí kratší cestu, se k potravě dostanou dříve. Při zpáteční cestě pak na rozcestí budou feromony pouze na kratší cestě, a tuto cestu si mravenci s nejvyšší pravděpodobností zvolí. Po odevzdání potravy do mraveniště půjdou tito mravenci opět za potravou. Tentokrát však naleznou více feromonu na kratší cestě a tak si ji pravděpodobně zvolí. Mechanismem popsaným výše opět mravenci konvergují k jednomu řešení. Za zmínku stojí, že i když je dlouhá cesta dvakrát delší než krátká, někteří mravenci stále využívají delší cestu. Toto je chápáno jako hledání zkratek.

Z experimentu je zřejmé, že kolonie mravenců mají schopnost nacházet nejkratší cestu. Takovéto chování mravenců se dá lehce simulovat. Jako prostředí pro mravence můžeme použít libovolný graf.

3.2 Optimalizační problémy

Optimalizační problém reprezentuje trojice $(\mathcal{S}, f, \Omega)$, kde $\mathcal{S}$ je množina možných řešení, f je účelová funkce $f : \mathcal{S} \to \mathbb{R}$, a Ω je množina omezení. Naším cílem je najít přípustné řešení $s^* \in \mathcal{S}$ takové, že f je v s^* minimální.

Některé optimalizační úlohy jsou z podstaty maximalizační. Maximalizační úlohu $(\mathcal{S}, f, \Omega)$ můžeme jednoduše převést na minimalizační úlohu $(\mathcal{S}, g, \Omega)$, kde $g(x) = -f(x)$.

Pro dynamické problémy se používá reprezentace jiná. Účelová funkce f pak má i druhý parametr t, reprezentující čas. $\Omega(t)$ je pak množina omezení v čase t. Jelikož optimalizační úloha, kterou zkoumáme v této práci, je statická, nebudeme se použitím ACO na dynamické problémy zabývat. Za zmínku však stojí, že toto jsou často problémy, ve kterých ACO exceluje.

3.3 Reprezentace optimalizačního problému

Abychom mohli využít ACO, vyjádříme kombinatorickou úlohu $(\mathcal{S}, f, \Omega)$ následujícími entitami:

- Konečná množina komponent $C = \{c_1, c_2, ..., c_{N_c}\}$, kde N_c je počet komponent
- Stavy problému definované jako konečné posloupnosti prvků z C. Například $x =< c_i, c_j, ..., c_h, ... >$
- Počet prvků stavu označíme $|x|$
- Množina všech možných stavů značíme X
- Množina přípustých řešení S je tedy podmnožinou X
- Množina přípustných stavů $\tilde{X}$. Stav x je přípustný, pokud je možné do této posloupnosti přídávat prvky z C tak, že výsledná posloupnost je přípustné řešení

$$\tilde{X} = \{x \in X | \exists y \in X;\ < x_1, x_2, ..., x_n, y_1, y_2, ..., y_m > \in S\}$$

- Neprázdnou množinu optimálních řešení S^* takovou, že $S^* \subseteq \tilde{X}$ a $S^* \subseteq S$
- Funkci $g : S \to \mathbb{R}$. $g(s)$ nazveme *cenou* s. V naprosté většině případů $g(s) = f(s) \forall s \in \tilde{S}$, kde $\tilde{S} \in S$ je množina všech přípustných řešení (řešení splňující podmínky Ω). Smysl této funkce (oproti funkci f) je případná penalizace řešení nesplňující podmínky Ω

- V některých případech má smysl zkonstruovat funkci $J(x)$, která pro stav x zdola odhaduje cenu řešení, které lze získat ze stavu přidáváním komponent. Pokud x_i může být získáno z x_j přidáváním komponent, pak $J(x_j) \leq J(x_i)$. Pokud $x \in S$, pak $J(x) = g(x)$

- Konstrukční graf $G_C = (C, L)$, kde $L = C \times C$. Množině L říkáme *spoje*[1]. Schválně jsem nepoužil slovo spojení, abych mohl rozlišit pojmy spoje (množinu L) a spoj (prvek množiny L). Takto definovaný konstrukční graf je úplný graf

Množinu omezení Ω můžeme implementovat různými způsoby. Zaprvé, konstrukci nepřípustného řešení můžeme úplně zakázat. Druhá možnost je nepřípustná řešení povolit, ale takto zkonstruovaná řešení penalizovat podle míry nepřípustnosti. Kterou strategii je vhodné zvolit záleží na konkrétním problému.

V případě, že se rozhodneme nepřípustná řešení povolit se funkce g (cena) obvykle implementuje jako $g(s) = f(s) + n(s)$, kde $n(s)$ je míra nepřípustnosti řešení s.

3.4 Umělí mravenci

Kolonii mravenců můžeme použít k hledání optimálního řešení problému vyjádřeným způsobem popsaným v 3.3. Ty jej hledají pravděpodobnostně založenými procházkami na konstrukčním grafu.

Pod pojmem mravenec k myslím agenta s následujícími vlastnostmi:

- *Počáteční stav* x_s^k

- *Ukončující podmínky* e^k

- Paměť $\mathcal{M}^k$

Počáteční stav je obvykle prázdná posloupnost nebo posloupnost obsahující pouze jednu komponentu.

Paměť může být použita k různým účelům:

- Implementace omezení ω například tak, že je znemožněna konstrukce nepřípustného řešení,

[1] V originále *connections*

- výpočet hodnot heuristické funkce η,
- ohodnocení nalezeného řešení,
- uložení nalezeného řešení.

Mravenci se řídí poměrně jednoduchým algoritmem: Agent k ve stavu x_r nejprve zkontroluje, zda je splněna nějaká z ukončujících podmínek. Pokud ano, mravenec skončí s konstrukcí řešení. Pokud ne, přesune se do jednoho ze sousedních vrcholů konstrukčního grafu. Po přesunu do vrcholu může mravenec aktualizovat množství feromonu na použité hraně nebo vrcholu. Poté, co mravenec skončí s konstrukcí řešení, může projít cestu zpět a aktualizovat na ni množství feromonu.

3.5 ACO obecně

Jde o opakované provádění tří procedur.

```
procedure ACOMetaheuristic
  ScheduleActivities
    ConstructAntSolutions
    UpdatePheromones
    DeamonActions            %Optional
  end-ScheduleActivities
end-procedure
```

Obrázek 3.2: Algoritmy mravenčí kolonie popsané v pseudokódu

Takto jsou obecně popsány algoritmy mravenčí kolonie a konkrétní detaily implementace jsou ponechány na řešiteli. V dalším textu jsou popsané vybrané konkrétní strategie.

3.6 Ant System

První algoritmus spadající do rámce ACO algoritmů je *Ant System* (někdy zkracovaný jako AS). Na začátku se položí určité množství feromonu na každou hranu, resp. vrchol konstrukčního grafu. Množství takto položeného

feromonu je dobré určit o něco větší než množství feromonu, které mravenci položí na graf v první iteraci.

Mravenec k se v každém kroku pravděpodobnostně rozhoduje do jakého dalšího vrcholu půjde. Pravděpodobnost, že z vrcholu i půjde do vrcholu j, je

$$p_{i,j}^k = \frac{[\tau_{i,j}]^\alpha [\eta_{i,j}]^\beta}{\sum_{l \in \mathcal{N}_i^k} [\tau_{i,l}]^\alpha [\eta_{i,l}]^\beta},$$

pokud $j \in \mathcal{N}_i^k$, jinak $p_{i,j}^k = 0$. $\mathcal{N}_i^k$ je množina všech přípustných sousedů i pro mravence k. $\tau_{a,b}$ je množství feromonu na hraně (a, b), resp. na vrcholu b, pokud feromony přičleňujeme k vrcholům. $\eta_{a,b}$ je heuristická informace. α a β jsou parametry.

Na konci každé iterace je množství feromonu v grafu aktualizováno. Nejprve je implementováno odpařování feromonu tak, že na každé hraně/vrcholu je sníženo množství feromonu

$$\tau_{i,j} \leftarrow (1 - p)\tau_{i,j},$$

kde p je parametr nazvaný *faktor odpařování*.

Dále je položen feromon na hrany/vrcholy, které mravenci použili. Každý mravenec vypustí za jednu procházku stejné množství feromonu. Množství feromonu položené mravencem k na hranu (a, b) značíme $\Delta\tau_{a,b}^k$. Množství feromonu položené jedním mravencem na konkrétní hranu je nepřímo úměrné délce procházky, kterou tento mravenec vykonal. Formálně

$$\Delta\tau_{a,b}^k = \frac{1}{|T^k|},$$

pokud $(a, b) \in T^k$ a

$$\Delta\tau_{a,b}^k = 0,$$

pokud $(a, b) \notin T^k$. T^k označuje procházku, kterou mravenec k vykonal. $|T^k|$ označuje délku této procházky.

S tímto značením můžeme aktualizaci feromonu na hraně (a, b) zapsat následovně:

$$\tau_{a,b} \leftarrow \tau_{a,b} + \sum_{k \in K} \Delta\tau_{a,b}^k \ ,$$

kde K je množina všech mravenců.

3.7 Elitist Ant System

Elitist Ant System (EAS) je ACO algoritmus rozšiřující Ant Systém. Algoritmus pracuje stejně jako Ant Systém. Navíc se v paměti uchovává dosud nejlepší nalezené řešení. V rámci *DeamonActions* je na takovou cestu položen dodatečný feromon, jehož množství záleží na parametru.

Označme dosud nejlepší nalezené řešení T^{bs}. Na každou hranu $(a, b) \in T^{bs}$ je posléze položeno množství $\frac{e}{|T^{bs}|}$ dodatečného feromonu, kde e je parametr.

3.8 Rank-Based Ant System

Stejně jako EAS i Ranked-Based Ant System vychází z Ant Systému. Poprvé byl popsán v [2]. Někdy je v literatuře označován AS_{rank}. Myšlenkou AS_{rank} je, že množství feromonu, které agent položí, závisí na kvalitě jím zkonstruovaného řešení vzhledem ke kvalitě řešení zkonstruovaném ostatními mravenci.

Dále je v paměti držena dosud nejlepší nalezená cesta T^{bs}, podobně jako v případě Elitist Ant Systemu. Na tuto cestu je také položen dodatečný feromon. Všimněme si, že žádný mravenec z množiny všech mravenců v dané iteraci nemusí řešení T^{bs} najít. Vytvořme si tedy dalšího mravence bs (best-so-far), reprezentující řešení T^{bs}.

Dále je nutné mravence seřadit podle kvality řešení, které reprezentují. Označme si i-tého nejlepšího mravence k^i. V případě, že více mravenců má stejnou kvalitu řešení, můžeme je seřadit náhodně.

Vypařování je implementováno stejně jako v Ant Systému. Pokládání feromonu na graf se týká pouze $w-1$ nejlepších mravenců. Množství feromonu položeného na hranu (a, b) je:

$$\sum_{r=1}^{w-1}(w-r)\Delta\tau_{a,b}^{k^r} + w\Delta\tau_{a,b}^{bs}$$

Při experimentech popsaných v [2] dosahoval Ranked-Based Ant System daleko lepších výsledků než Ant System a nepatrně lepších výsledků než Elitist Ant System.

3.9 $\mathcal{MAX} - \mathcal{MIN}$ Ant System

$\mathcal{MAX} - \mathcal{MIN}$ Ant System (ve zkratce $\mathcal{MM}$AS) je také odvozen z Ant Systému. Představili ho Stützle a Hoos [22] v roce 1999.

Podobně jako Rank-Based Ant System si $\mathcal{MAX} - \mathcal{MIN}$ Ant System udržuje v paměti nejlepší zatím dosaženou cestu a mravence bs. Pouze mravenec s nejlepším řešením v dané iteraci a mravenec bs mohou pokládat feromon. Dále, $\mathcal{MM}$AS stanovuje horní a dolní mez pro množství feromonu na hraně. Při začátku algoritmu je množství feromonu na hranách inicializováno na horní mez. Pokud systém projeví stagnaci nebo pokud po určitý počet iterací není dosud nalezené nejlepší řešení ještě vylepšeno, celý algoritmus začne znovu.

3.10 Ant Colony System

Ant Colony System [10] se liší od Ant Systemu v několika bodech:

- Používá se jiné pravidlo pro výběr následujícího vrcholu. S pravděpodobností rovnou parametru q_0 přejde mravenec do uzlu na nejlepší známé cestě z aktuálního uzlu. S pravděpodobností $1 - q_0$ se využije pravidlo z Ant Systemu,
- pouze na hranách patřící do $T^b s$ se provádí pokládání a vypařování feromonu,
- množství feromonu na hraně se sníží poté, co ji mravenec použije.

Existuje paralelní rozšíření Ant Colony Systemu nazvané Parallel Ant Colony System [4].

3.11 Další

Do kategorie ACO spadají i některé další méně používané algoritmy. Za zmínku stojí ANTS [18] (Approximate Nondeterministic Tree Search) a Hyper-Cube Framework for ACO [1].

Kapitola 4

Použití ACO k prolomení jednoduché substituční šifry

Tato kapitola je věnována možnostem „našroubování“ Ant Colony Optimization na problém prolomení monoalfabetické substituční šifry. Aby bylo možné použít ACO, je potřeba najít funkci ohodnocující kvalitu řešení. Dále je potřeba sestrojit konstrukční graf.

4.1 Hledání ideální fitness funkce

K použití Ant Colony Optimization potřebujeme funkci hodnotící kvalitu klíče, resp. dešifrovaného textu. O jedné jsme se zmínili již v kapitole 2.4, ale možných funkcí je mnohem víc. Hledání, té pro naší aplikaci nejvhodnější, se věnuje tato kapitola.

4.1.1 Frekvence jednotlivých písmen

Člověk znající frekvenční analýzu pravděpodobně dostane přímočarý nápad založit účelovou funkci na frekvenci jednotlivých písmen, tedy minimalizovat

$$\sum_i |c_i - d_i| \ ,$$

kde c_i označuje frekvenci písmene i v dešifrovaném textu a d_i je frekvence i v korpusu. Korpus je rozsáhlá množina textů v daném jazyce.

Typické rozložení písmen v anglickém a českém textu je k vidění na obrázku 2.3 na straně 11.

4.1.2 Frekvence bigramů

O této účelové funkci jsme se zmínili již v kapitole o Jakobsenově algoritmu 2.4. Funguje podobně jako předchozí funkce, ale porovnává frekvence bigramů namísto frekvencí jednotlivých písmen. Tedy minimalizujeme

$$\sum_i \sum_j |C_{i,j} - D_{i,j}|,$$

kde $C_{i,j}$ je frekvence bigramu ij v dešifrovaném textu a $D_{i,j}$ je frekvence tohoto bigramu v korpusu. V tomto případě můžeme C a D zapsat jako matice indexované písmeny.

Použitím několika knih[1] jako korpusu jsme získali rozložení frekvencí bigramů vyobrazené v tabulce 4.1.

Tabulka 4.1: Řádky jsou označeny prvním písmenem bigramu. Sloupce druhým písmenem. Frekvence je v promile.

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	0.13	2.05	3.4	4.54	0.02	0.97	1.75	0.37	4.14	0.05	1.34	6.07	2.49
B	0.92	0.1	0	0.01	5.8	0	0	0.01	0.39	0.19	0	1.84	0.02
C	3.02	0.01	0.56	0.01	4.32	0.01	0	4.44	1.02	0	1.26	0.93	0.01
D	3.89	1.76	0.65	1.18	5.18	0.94	0.66	2.06	5.57	0.1	0.28	1.19	2.03
E	10.42	2.13	4.66	11.77	4.97	2.77	1.7	2.69	4.45	0.18	0.38	5.54	4.9
F	2.34	0.24	0.38	0.18	1.89	1.24	0.12	0.79	2.68	0.02	0.04	0.57	0.72
G	2.14	0.31	0.18	0.16	2.48	0.28	0.57	3.3	1.51	0.03	0.03	0.63	0.37
H	12.36	0.17	0.17	0.14	26.59	0.16	0.1	0.67	9.04	0.02	0.02	0.16	0.5
I	1.45	0.57	4.23	3.97	2.65	2.02	1.93	0.92	0.09	0.01	0.7	2.93	3.48
J	0.1	0	0	0	0.25	0	0	0	0.05	0	0	0	0
K	0.52	0.1	0.06	0.05	2.66	0.22	0.02	0.25	1.62	0.01	0.02	0.2	0.22
L	3.5	0.37	0.28	3.56	6.46	1.11	0.09	0.29	4.46	0.02	0.34	5.1	0.42
M	4.52	0.75	0.09	0.07	7.61	0.2	0.06	0.28	3.2	0.01	0.02	0.19	0.69
N	3.47	0.59	2.84	11.9	6.3	0.85	9.2	1.25	3.51	0.13	0.81	0.89	0.99
O	1.33	1.48	1.45	1.83	0.45	8.05	0.55	1.1	1.24	0.04	1.4	2.14	5.51
P	2.21	0.07	0.03	0.02	3.93	0.05	0.01	0.3	1.08	0	0.01	1.66	0.06
Q	0	0	0	0	0	0	0	0	0	0	0	0	0
R	5.23	0.76	1.31	1.95	13.65	1.2	0.6	1.18	5.09	0.09	0.91	0.95	1.89
S	6.79	1.1	1.53	0.6	7.57	0.91	0.44	5.03	5.72	0.08	0.57	0.94	1.69
T	5.82	1.02	0.86	0.68	7.84	0.75	0.22	29.68	9.28	0.06	0.25	1.99	1.38
U	0.87	0.58	1.5	0.56	0.65	0.32	1.4	0.15	0.98	0.01	0.11	3.29	0.95
V	0.63	0	0	0	7.14	0	0	0	1.2	0	0	0	0
W	5.38	0.46	0.12	0.13	3.38	0.08	0.02	4.97	4.08	0.01	0.01	0.22	0.2
X	0.16	0.01	0.23	0	0.15	0.01	0	0.03	0.17	0	0	0	0.01
Y	2.17	0.81	0.63	0.69	1.38	0.6	0.25	0.84	1.38	0.04	0.11	0.49	0.93
Z	0.02	0	0	0	0.13	0	0	0	0.03	0	0	0.01	0

[1] konkrétně Charles Babbage: On the Economy of Machinery and Manufactures, Charles Dickens: David Copperfield a Franz Kafka: The Trial

	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	15.48	0.08	1.93	0.04	7.07	8.92	10.82	1.02	2.65	1.55	0.06	2.41	0.06
B	0	1.71	0	0	0.78	0.32	0.15	1.87	0.02	0.01	0	1.29	0
C	0.01	5.15	0.01	0.08	0.81	0.03	2.36	0.95	0	0.02	0	0.13	0
D	1.16	4.21	0.63	0.05	1.09	2.83	4.7	1.11	0.27	1.67	0	0.76	0
E	11.26	3.4	3.14	0.47	18.45	11.68	7.93	0.55	2.42	4.42	1.33	1.86	0.02
F	0.1	4.02	0.33	0.01	1.69	0.52	3.3	0.86	0.05	0.4	0	0.27	0
G	0.48	2.27	0.18	0.02	1.27	0.78	1.57	0.56	0.05	0.45	0	0.11	0
H	0.11	4.88	0.14	0.01	0.59	0.51	2.87	0.56	0.02	0.47	0	0.33	0
I	19.22	3.35	0.55	0.03	2.81	8.61	9.93	0.09	1.35	0.87	0.13	0.01	0.09
J	0	0.24	0	0	0	0	0	0.43	0	0	0	0	0
K	0.97	0.26	0.05	0.01	0.04	0.56	0.37	0.05	0.01	0.26	0	0.12	0
L	0.16	3.49	0.34	0.02	0.23	1.07	1.46	0.57	0.2	0.5	0	3.88	0
M	0.17	2.79	1.36	0.01	1.49	0.93	0.83	1.06	0.03	0.34	0	2.74	0
N	0.76	5.74	0.37	0.13	0.26	3.41	11.19	0.76	0.33	1.14	0.04	1.3	0.02
O	10.91	3.59	1.87	0.02	8.62	2.95	5.66	10.78	1.14	4.31	0.08	0.65	0.02
P	0.02	2.47	1.47	0	2.64	0.45	0.72	0.76	0	0.12	0	0.2	0
Q	0	0	0	0	0	0	0	1.19	0	0	0	0	0
R	1.41	5.42	0.88	0.05	1.11	4.1	4.74	0.83	0.47	1.28	0	2.16	0
S	0.94	5.28	1.83	0.16	0.51	4.71	10.19	2.43	0.22	1.9	0	0.39	0
T	0.58	12	0.53	0.06	2.74	3.47	6.13	1.84	0.1	2.74	0	1.8	0.01
U	3.17	0.08	1.61	0.01	4.45	3.36	4.16	0.02	0.05	0.22	0.01	0.04	0.01
V	0	0.37	0	0	0	0	0	0	0	0	0	0.11	0
W	0.95	2.44	0.06	0	0.26	0.39	0.41	0.03	0.02	0.23	0	0.2	0
X	0	0.03	0.48	0	0	0.01	0.31	0.01	0	0.02	0	0.01	0
Y	0.32	3.89	0.54	0.02	0.36	1.94	2.05	0.17	0.07	1.11	0	0.16	0
Z	0	0.01	0	0	0	0	0	0	0	0	0	0.01	0.02

4.1.3 Frekvence trigramů

Některé aplikace používají ke konstrukci účelové funkce frekvenci jednotlivých trigramů, tedy:

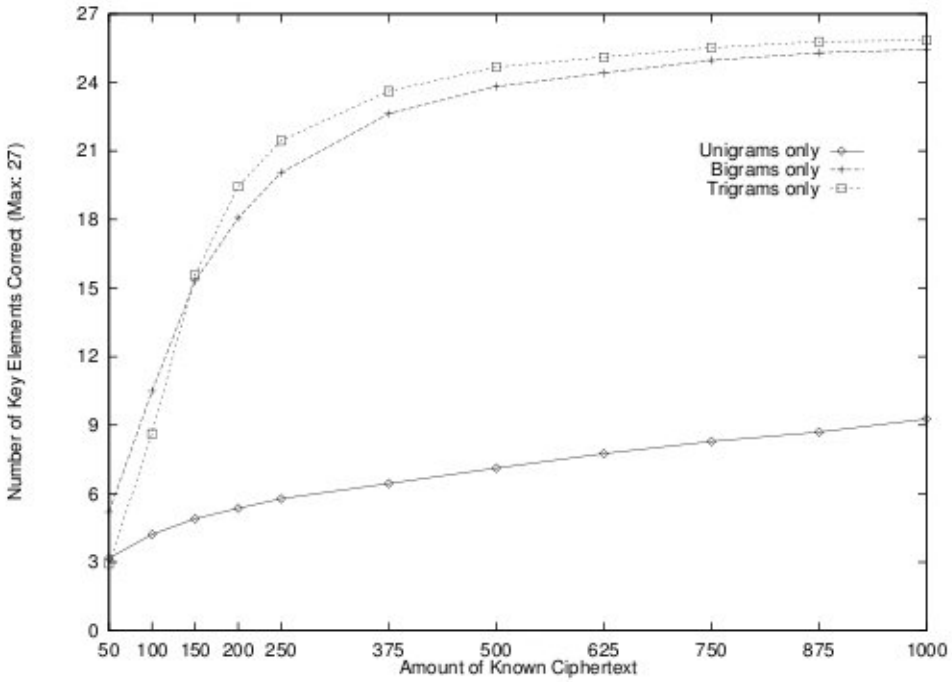
$$\sum_i \sum_j \sum_k |C_{i,j,k} - D_{i,j,k}| \ ,$$

kde $C_{i,j,k}$ a $D_{i,j,k}$ jsou frekvence trigramu ijk v dešifrovaném textu, resp. korpusu.

Je na místě otázka, zda si takto oproti používání frekvence bigramů pomůžeme. Graf na obrázku 4.1 ukazuje, že počítat trigramy se vyplatí na delších textech, zatímco pro krátké je lepší používat bigramy. Bod, ve kterém se používání trigramů „vyplatí“, je přibližně 150 znaků, avšak i za tímto bodem je rozdíl mezi kvalitou těchto účelových funkcí minimální. Nás zajímají především texty krátké. Pro nás tedy využití frekvencí trigramů nepřináší užitek. Navíc, počítání kvality klíče vyžaduje nutně 26-krát více operací.

Podívejme se nyní, proč je funkce nepřesná na krátkých textech. Počet trigramů nad anglickou abecedou je 17576. Spousta z nich je tak výjimečných, že se v korpusech s největší pravděpodobností nevyskytnou. Takovéto trigramy mohou být například xxx, yiy nebo qwq. Zpráva délky n může obsahovat maximálně n-2 unikátních n-gramů. Je zřejmé, že krátké texty o desítkách či stovkách znaků nemohou všechny tyto trigramy obsahovat.

Pokud budeme pracovat s texty v jiném jazyce, dá se předpokládat, že nepřesnost funkce na krátkých textech bude jiná. Například česká abeceda



Obrázek 4.1: Úspěšnost účelových funkcí využívající unigramy, bigramy a trigramy v závislosti na délce textu. Převzato z [5]

se skládá z 42. písmen. Z těch se dá poskládat až 74088 trigramů, což je ve srovnání počtu trigramů nad anglickou abecedou velké číslo.

4.1.4 Kombinace předchozích - obecné N-gramy

Výpočet účelové funkce pro obecné N-gramy má složitost $\mathcal{O}(a^b)$, kde a je velikost abecedy (v našem případě 26) a b je maximální n, pro které počítáme n-gramy. Vzhledem k tomu nemá v praxi smysl zvažovat počítání 4-gramů.

Avšak má smysl kombinovat hodnoty předchozích třech funkcí. Funkce lineárně kombinující dříve popsané funkce vypadá následovně:

$$\alpha * \sum_i |c_i - d_i| + \beta * \sum_i \sum_j |C_{i,j} - D_{i,j}| + \gamma * \sum_i \sum_j \sum_k |R_{i,j,k} - S_{i,j,k}| \ ,$$

kde α, β, γ jsou parametry, $R_{i,j,k}$ je frekvence trigramu ijk v dešifrovaném textu a S_{ijk} je frekvence trigramu v korpusu.

Tato funkce je zobecnění třech předchozích. Pokud $\alpha = 1, \beta = \gamma = 0$, pak jde o frekvenci jednotlivých písmen. Pokud $\alpha = \gamma = 0, \beta = 1$, jde o frekvenci bigramů a obdobně lze získat funkci počítající frekvenci trigramů.

4.1.5 Vyhlazování

Výše zmíněné funkce jsou často používané a povětšinou dostačující. Stává se, že nějaký málo se vyskytující n-gram v trénovacím textu zcela chybí.

Trénovací text, použitý k vytvoření tabulky 4.1, se skládal ze třech textů o celkové délce 2 306 634 znaků anglické abecedy. Přesto vůbec neobsahoval 71 bigramů, což je přes 10%. Pokud se takový bigram nachází i v otevřeném textu, je správnému klíči a náležitému dešifrovanému textu přiřazeno neodpovídající ohodnocení.

Techniky vyhlazení (smoothing) se tento problém snaží řešit. Těmito technikami se zabýval Chen a Goodman [3]. Nám postačí ta nejjednodušší nazvaná *ad hoc*. Při použití ad hoc metody přiřadíme n-gramům, které se nevyskytují v korpusu, pravděpodobnost $exp(-8.0)$.

4.2 Převod problému na prohledávání grafu

Algoritmy ACO dosahují dobrých výsledků na problému obchodního cestujícího (dále TSP). Na první pohled by se mohlo zdát, že se jedná o problém blízký našemu. I v případě TSP hledáme permutaci na pevném počtu prvků.

Z optimalizačního hlediska je v TSP důležité pořadí prvků ve smyslu návaznosti („Který prvek následuje za prvkem a?“), zatímco v našem případě se ptáme „Který prvek je na i-tém místě?“ Proto musíme při aplikaci ACO dojít ke zcela jinému konstrukčnímu grafu.

Označme si abecedu písmenem K. Náš graf potom bude úplný graf nad množinou vrcholů $K \times K$. Pokud cesta prochází vrcholem (a, b), interpretujeme to jako „Písmeno a se šifruje na písmeno b.“ Dále musíme položit řadu omezení. Není možné, aby se jeden znak šifroval na více znaků. Dále není možné, aby se na nějaký znak šifrovalo více znaků.

4.3 Výběr konkrétního ACO algoritmu

Vzhledem k povaze konstrukčního grafu jsme zvolili asociování feromonů s vrcholy místo s hranami, jak se tradičně dělá. Je to proto, že nám záleží na

tom, kterých 26 vrcholů mravenec navštíví. Nezáleží na tom v jakém pořádí, jako je tomu např. u problému obchodního cestujícího.

Pro náš problém se Elitist Ant System a Rank-Based Ant System jeví jako ideální ACO algoritmy. Elitist Ant System z toho důvodu, že úloha nemá mnoho lokálních maxim. Proto je výhodné si držet to nejlepší nalezené řešení mezi iteracemi a snažit se hledat lepší řešení v jeho okolí. Rank-based Ant System je vhodný proto, že při jeho použití nemusíme hledat způsob, jak přepočítávat fitness mravence na množství feromonu, který pokládá. Už jen název fitness je trochu zavádějící. Fitness se překládá jako vhodnost nebo způsobilost. Avšak v našem případě platí, že čím nižší fitness řešení má, tím lepší je. Pro aplikaci ACO potřebujeme umět každé hodnotě fitness přiřadit odpovídající množství feromonů, které mravenec položí. Při použití Rank-based Ant Systému není množtví závislé přímo na kvalitě řešení, ale na ranku mravence. Jinými slovy podle toho, kolik mravenců je lepších.

Proto jsme se rozhodli sloučit tyto dvě strategie. Mravenci pokládají feromon podle pravidla Rank-based Ant-Systému. Navíc nejlepší mravenec položí množství e dodatečného feromonu. Pro použití čistého Rank-based Ant Systém stačí nastavit $e = 0$. Aktualizaci feromonu na hraně (a, b) můžeme zapsat následovně:

$$\tau_{a,b} \leftarrow \tau_{a,b} + \sum_{r=1}^{w-1}(w-r)\Delta\tau_{a,b}^{k^r} + (w+e)\Delta\tau_{a,b}^{bs}$$

Mravencům nedovolujeme konstruovat nepřípustná řešení.

4.4 Beta heuristika

Beta heuristika pomáhá mravencům při konstrukci řešení, avšak nevyužívá feromonů. Posuzuje vhodnost přesunu z vrcholu (a, b) do vrcholu (c, d). Tuto informaci hledáme v trénovacím a šifrovém textu. Označme si $\phi_{k,l}$ frekvenci bigramu kl v trénovacím textu a $\varphi_{k,l}$ frekvenci bigramu kl v šifrovém textu. Hodnota heuristiky pro hranu ((a,b),(c,d)) potom bude

$$1 + \frac{add_1 + add_2}{4},$$

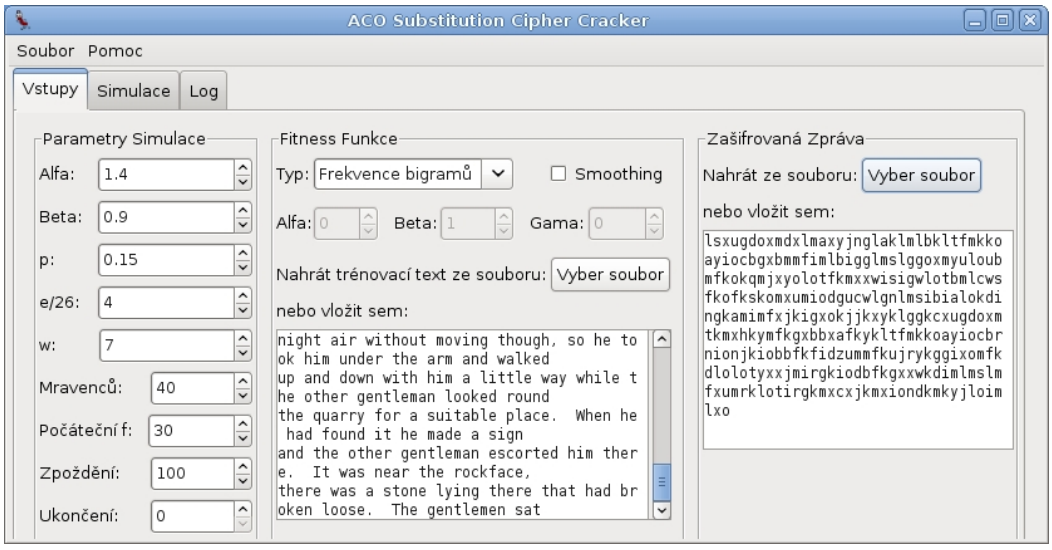
kde $add_1 = \frac{min(\phi_{b,d},\varphi_{a,c})}{max(\phi_{b,d},\varphi_{a,c})}$, pokud $min(\phi_{b,d}, \varphi_{a,c}) > 0$, jinak 0 a $add_2 = \frac{min(\phi_{d,b},\varphi_{c,a})}{max(\phi_{d,b},\varphi_{c,a})}$, pokud $min(\phi_{d,b}, \varphi_{c,a}) > 0$, jinak 0.

Hodnota této heuristiky bude vždy v intervalu $[1, \frac{3}{2}]$.

Kapitola 5

Implementace

V této kapitole popíšeme průběh a způsob implementace doprovodného programu. Program je popsán shora. Nejedná se o referenční příručku k API. Ta se nachází na CD v adresáři doc_devel/javadoc a popisuje detailně jednotlivé metody. Tato kapitola není ani uživatelskou příručkou k programu. Práce s programem a grafické rozhraní (obrázek 5.1) jsou popsány příloze A.



Obrázek 5.1: Grafické uživatelské rozhraní programu

Program jsem implementoval v jazyce Java verze 6. Ke spuštění je tedy potřeba Java Virtual Machine. Spustitelný .jar soubor i zdrojové kódy jsou

přiložené na CD. Dále jsou ke stažení na následujících adresách:

`http://sourceforge.net/projects/acosubstcracker/`
`http://sourceforge.net/users/lubos_turek/`

Program vydávám pod licencí BSD dovolující každému program libovolně upravovat nebo využívat části programu ve svých dílech.

5.1 Grafické uživatelské rozhraní

Cílem bylo, aby uživatel mohl kvalitně sledovat průběh výpočtu, výpočet zastavit, zkoumat jednotlivé mravence a nějakým způsobem „vidět“ rozložení feromonů. Dále jsem požadoval možnost měnit parametry bez nutnosti restartu simulace. K programu jsem se proto rozhodl implementovat grafické uživatelské rozhraní(obrázek 5.1), které všechny tyto vlastnosti umožňuje.

5.2 Pohled shora

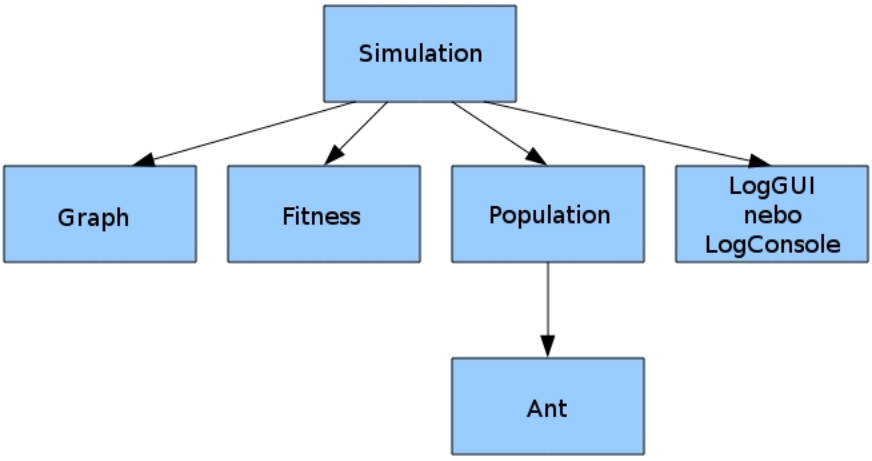
Hlavní main metoda se nachází ve třídě ACOSubstCracker. Její úkoly jsou parsování parametrů předané z příkazové řádky a spuštění zprostředkovatele simulace. To může být ConsoleRunner, pokud je nalezen přepínač $-nogui$, nebo ACOSubstCrackerApp, pokud požadujeme spuštění v grafickém uživatelském rozhraní. Třída ACOSubstCracker byla vygenerována v Netbeans.

Nyní si odmyslíme grafické uživatelské rozhraní a jiné pomocné třídy a budeme se soustředit pouze na jádro ACO algoritmu. Třída simulace zastřešuje simulaci. Uchovává parametry výpočtu, vytváří populaci mravenců, a obstarává hlavní smyčku. Výpočet probíhá ve vlastním vlákně. Zjednodušený diagram tříd je na obrázku 5.2.

Pokud dojde k nestandardní situaci související přímo se simulací, je vyhozená výjimka SimulationException. Ta je odchycena v Simulation. V takovém případě je výpočet ukončen a informace obsažené ve výjimce jsou zapsány do Logu.

5.3 Třída Simulation

Simulaci jako takovou zprostředkovává třída Simulation, běžící ve vlastním vlákně. Zvolený návrhový vzor je singleton. Simulace se vytvoří pomocí



Obrázek 5.2: Zjednodušený diagram tříd

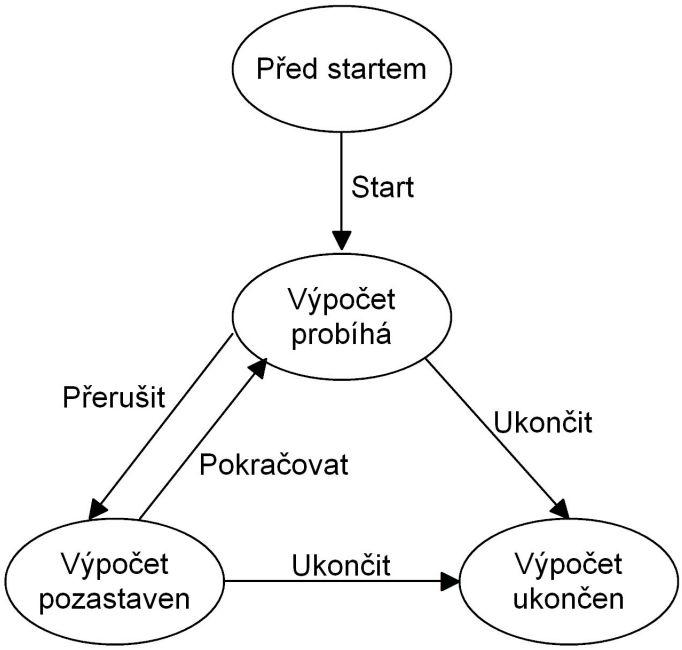
statické metody createSimulation. Průběh simulace se řídí statickými metodami startSimulation, pauseSimulation, continueSimulation a finishSimulation. Aktuální instance simulace se získá pomocí statické třídy getSimulation. K parametrům simulace (Alfa, Beta, P, E, numberOfAnts) existují gettery a settery.

Simulace může být ve čtyřech stavech:

- Před startem,
- probíhající výpočet,
- výpočet pozastaven,
- ukončeno.

Nová simulace vytvořená metodou *createSimulation* se nachází ve stavu před startem. Do stavu, kdy probíhá výpočet, se dostane po volání metody *runSimulation*. Ve stavu probíhajícího výpočtu není možné měnit parametry simulace. K tomu je potřeba výpočet pozastavit. Pokud k tomu zadáme povel metodou *pauseSimulation*, nejprve se dokončí právě probíhající iterace. Nějaké atributy simulace není možné upravovat ani pokud je výpočet pozastaven. Tyto atributy jsou šifrový text a fitness funkce. Pozastavený výpočet je možné převést do stavu probíhajícího výpočtu. Ukončený výpočet

již nikoliv. Stavy výpočtu a přechody mezi nimi jsou znázorněny na obrázku 5.3.



Obrázek 5.3: Stavy výpočtu a přechody mezi nimi

5.4 Třída Population

Třída Population reprezentuje populaci mravenců. Při startu simulace se vytvoří pole mravenců. Mravenci se vytvářejí v konstruktoru. Řešení hledají po volání metody *generatePaths*. Třída dále obsahuje metodu *getAnts*, která vrátí pole všech mravenců v populaci.

V každé iteraci se vytváří nová populace. Reference na objekt odpovídající staré populaci se ztratí.

5.5 Třída Graph

Třída Graph reprezentuje použitý konstrukční graf. Uvědomil jsem si, že jediné, co po této třídě požaduji, je udržování informace o množství feromonu na vrcholech a udržování heuristické informace. Tudíž není potřeba nijak reprezentovat hrany ani vrcholy. Množství feromonů na vrcholech a hodnoty heuristické funkce se udržují v poli. Stačí nám následující funkce:

- *getPheromone*, která pro daný pár písmen vrátí množství feromonu na odpovídajícím vrcholu
- *addPheromone*, která na vrchol odpovídajícímu dané dvojici písmen přidá, resp. odebere (pokud je argument záporný) určité množství feromonu
- *pheromoneEvaporate*, která zajišťuje odpařování feromonu
- *getHeuristic*, která vrací pro daný pár vrcholů heuristickou informaci β

Jelikož heuristická informace je statická (nemění se v čase), rozhodl jsem se její hodnoty předpočítat a uložit již v konstruktoru. Funkce *getHeuristic* tedy nic nepočítá, ale pouze vrací hodnotu uloženou v poli. Časová náročnost je tedy minimální.

5.6 Třída Ant

Když mravenec začíná konstruovat řešení, nenachází se na žádném vrcholu. První vrchol cesty si vybírá podle stejného pravidla jako další vrcholy. Cestování po grafu při konstrukci řešení odpovídá přidávání prvků do pole key. Toto pole reprezentuje klíč šifry. Jakmile je klíč vytvořen, můžeme používat metodu *decipher*, která dešifruje text za pomocí nalezeného klíče. Třída se dále stará o pokládání feromonů při volání funkce depositPheromone.

5.7 Třída Fitness

Fitness je třída reprezentující fitness funkci pro naší aplikaci.

Před startem simulace si můžeme vybrat, kterou fitness funkci použijeme. Výběr je ze všech popsaných v sekci 4.1, tedy: frekvence jednotlivých písmen,

frekvence bigramů, frekvence trigramů, obecné N-gramy. Ve skutečnosti je implementována pouze funkce obecných N-gramů, která je zobecněním všech zmíněných. Je třeba rozlišovat parametry alfa a beta patřící k simulaci od parametrů alfa, beta a gama patřící k fitness funkce. Ty první ovlivňují mravence a to, jak si vybírají vrcholy grafu při konstrukci cesty. Dále si uživatel může vybrat, zda se použije ad hoc smoothing, či ne.

Vybranou fitness funkci již nejde po spuštění simulace měnit.

Složitost ohodnocení klíče je lineární vzhledem k šifrovému textu. Po ohodnocení mravence je výsledek uložen v cache. Pokud je později hodnocen nějaký mravenec se stejným klíčem, hodnota fitness je nalezena v cache v konstantním čase. V prvních desítkách iterací simulace je poměr klíčů nalezených v cache minimální. Ke konci simulace je v cachi nalezeno okolo 80% klíčů, avšak konkrétní hodnota je velmi závislá na parametrech a konkrétním textu.

Hodnota heuristické informace β se také počítá zde, avšak hodnoty patřící jednotlivým uzlům jsou uloženy v *Graph* a je k nim přistupováno přes $Graph.getHeuristic$.

5.8 Výjimky

Jakákoliv výjimka, která vznikne z podstaty simulace, je instancí $SimulationException$. Pokud je taková výjimka vyhozena, je odchycena až v třídě Simulation. V tomto případě se ukončí výpočet a chyba je zapsána do Logu. Druhou definovanou výjimkou je $UnknownParameterException$, která se vyhazuje pouze v parseru parametrů předaných z příkazové řádky. Parserem je vždy i odchycena. Při odchycení se na terminál vytiskne správné užití parametrů a program končí.

5.9 Třída ACOSubstCrackerView

Tato třída se stará o vykreslování grafického uživatelského rozhraní. Většina kódu byla vytvořena automaticky Netbeans plug-inem GUI Builder.

5.10 Třída Log a její potomci

Abstraktní třída Log slouží k logování událostí. Její potomci jsou LogGUI a LogConsole zajištující zobrazení logovaných událostí v grafickém, resp.

řádkovém režimu.

Při použití LogGUI se logovací zprávy tisknou do textového pole *textAreaLog*. Pokud je z nějakého důvodu textové pole *textAreaLog* nedostupné (např. pokud nebylo inicializováno nebo pokud došlo k chybě programu), zpráva je vytištěna na standardní výstup aplikace.

Při použití řádkového režimu není možné úroveň logování za běhu měnit, a tak je nastavení LogConsole realizováno parametrizací konstruktoru. Všechny logované události jsou vypisovány na standardní výstup.

5.11 Třída SubstitutionCipher

Jedná se o třídu se samými statickými metodami, které jsou nápomocné v ostatních třídách a mají souvislost se substituční šifrou. Uvnitř programu jsou písmena abecedy reprezentovány čísly. A odpovídá nule a Z odpovídá číslu 25. SubstitutionCipher implementuje metody k převodu mezi těmito formáty. Tyto funkce využívají hashovací tabulky.

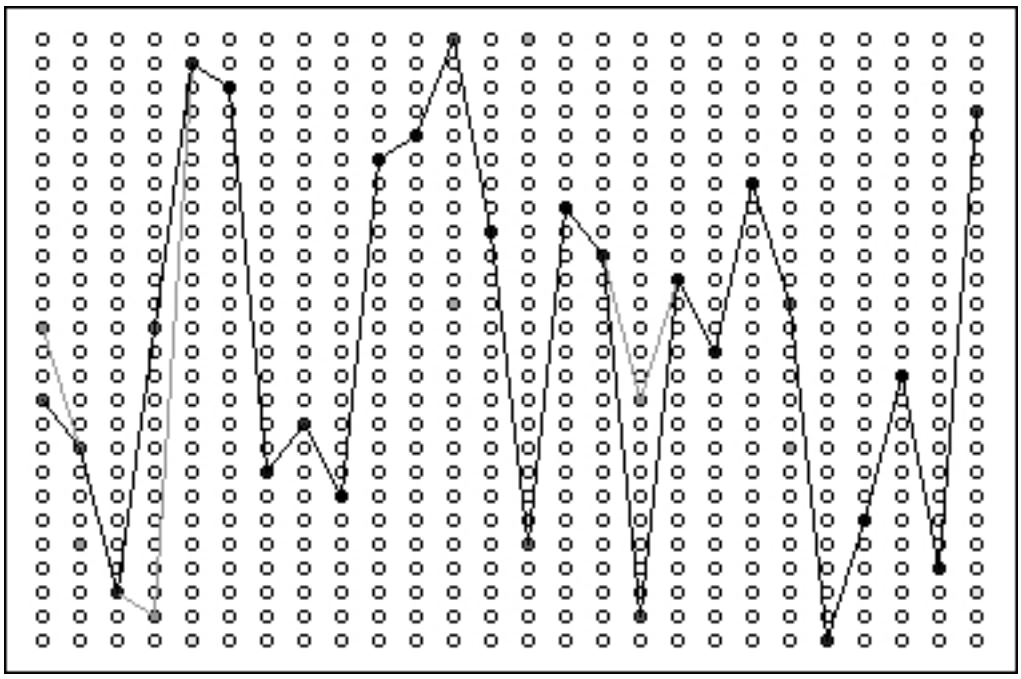
Sice pracujeme pouze s texty obsahující písmena anglické abecedy, ale je třeba umět zpracovat i texty ze života. Například jako trénovací text fitness funkci můžeme předhodit nějakou knihu. K tomu se nám hodí následující statické metody:

- isEnglishChar(char c) zjišťuje, zda je znak *c* znakem anglické abecedy.
- stripNonEnglishChars(String text) vrací String, který vznikne z řetězce na vstupu vynecháním písmen, které nepatří do anglické abecedy.

5.12 Třída GraphVisualization

GraphVisualization je grafická komponenta, kterou jsem napsal. Jejím úkolem je graficky zobrazovat množství feromonu na uzlech grafu. Komponenta je potomek *javax.swing.JPanel*.

Zobrazován je spíše klíč, který mravenec nalezl, než cesta, po které mravenec šel. Klíč je zobrazován po sloupcích. Mravenec však mohl naleznout tento klíč procházkou po vrcholech v kterémkoliv pořadí.



Obrázek 5.4: Vizualizace konstrukčního grafu, nejlepších nalezených cest a feromonů

Kapitola 6

Výsledky

Tato kapitola se zabývá metodikou, kterou jsme dospěli k vhodným parametrům. Takto nastavený program byl testován na různých textech. Výsledky porovnáváme s genetickým algoritmem, který je popsán v příloze B.

6.1 Hledání optimálních parametrů

ACO metaheuristika je na nastavení parametrů obecně velmi citlivá. V případě naší aplikace na prolomení jednoduché substituční šifry to není jednoznačně patrné, protože účelová funkce je velmi hladká a tak nalezení optima není vždy obtížný úkol. Tedy i simulace se špatnými parametry s velkou pravděpodobností nalezne kvalitní řešení. Na druhou stranu se zvyšuje malá pravděpodobnost, že nalezené řešení bude špatné. Proto bylo k nalezení optimálních parametrů potřeba simulaci spustit mnohokrát a výsledky statisticky zpracovat.

Na počítači s procesorem Intel Core2 Duo 2,00GHz a 2GB 800MHz DDR2 pamětí trvala jedna simulace cca 3 minuty. Při zapnutém logování cest mravenců trval výpočet až pětkrát déle. Z časových důvodů nebylo možné vyzkoušet mnohokrát každou kombinaci parametrů a tak bylo postupováno podle následující strategie: nejprve empiricky najít rozumnou sadu parametrů, a pak pro každý parametr testovat všechny jeho rozumné hodnoty. Vytvořil jsem dávkový skript, jehož vykonání trvalo 11 dní.

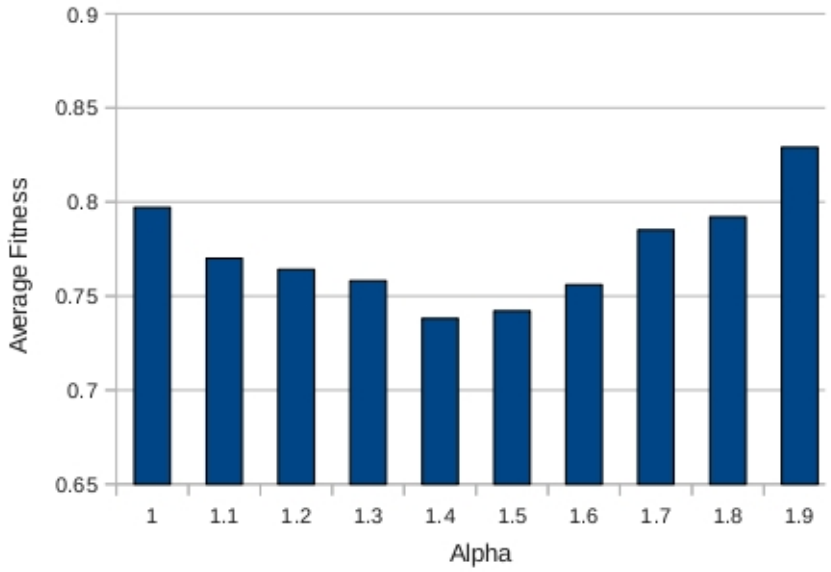
Empiricky nalezené výchozí hodnoty byly následující:

- Alfa: 1.5
- Beta: 1

- p: 0.10
- e/26: 7
- w: 7
- Mravenců: 40
- Počáteční f: 20

6.1.1 Optimální α

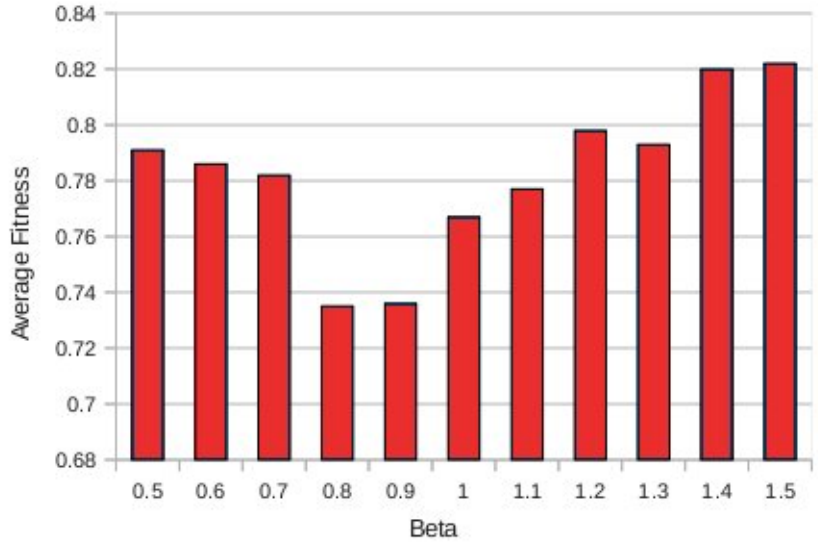
Pro každou hodnotu α z množiny $\{1.0, 1.1, ..., 1.9\}$ bylo vykonáno 200 simulací na textech o 360 znacích. Ve srovnání vyšla najlépe hodnota 1.4 s průměrným fitness nalezeného řešení 0.738. Původní empiricky nalezená hodnota 1.5 byla jen o málo horší s průměrným fitness 0.742. Graf 6.1 znázorňuje výsledky pro všechny zkoušené hodnoty parametru α.



Obrázek 6.1: Průměrný fitness řešení pro různé hodnoty parametru α

6.1.2 Optimální β

Parametr β jsme hledali obdobně. Pro všech 11 hodnot z množiny $\{0.5, 0.6, ..., 1.5\}$ bylo opět vykonáno 200 simulací prolamujících šifrové texty o 360 znacích. Nejlepší se ukázaly být hodnoty 0.8 a 0.9 s průměrným fitness 0.735, resp. 0.736. Průměrný fitness všech zkoušených hodnot znázorňuje graf 6.2.



Obrázek 6.2: Průměrný fitness pro různé hodnoty β

6.1.3 Další parametry

Hledání ideálních parametrů p, e, w, počtu mravenců a počátečního feromonu je složitější, jelikož tyto parametry se navzájem více ovlivňují. Rozhodl jsem se pro každý z těchto parametrů zvolit množinu rozumných hodnot a spustit 5 simulací pro každý prvek z kartézského součinu těchto množin. Zkoušené množiny hodnot parametrů jsou tyto:

- p: $\{0.075, 0.1, 0.125, 0.15\}$
- e/26: $\{3, 4, 5, 6, 7\}$
- w: $\{4, 7, 12, 15\}$

- Mravenců: $\{25, 32, 40\}$
- Počáteční f: $\{10, 20, 30\}$

Z těchto celkem 720ti možných konfigurací nebyla žádná, která by výrazně převyšovala všechny ostatní. Některé konfigurace se však ukázaly být nevhodné. Jsou to kombinace vysokého elitismu s nízkým faktorem odpařování, což vede k dominanci lokálního minima. Nevhodné se také ukázalo malé množství počátečního feromonu v kombinaci s vysokým faktorem odpařování.

Popsanými metodami jsme dospěli k následujícím hodnotám parametrů, které jsme nastavili v programu jako výchozí:

- Alfa: 1.4
- Beta: 0.9
- p: 0.15
- e/26: 4
- w: 7
- Mravenců: 40
- Počáteční f: 30

6.2 Srovnání s genetickým algoritmem

Za účelem srovnání představeného ACO algoritmu s jiným biologicky motivovaným algoritmem jsme navrhli a implementovali program řešící stejný problém pomocí genetických algoritmů. Návrh programu a jeho ovládání jsou popsány v příloze B.

Srovnání proběhlo postupným spuštěním obou programů na čtyřiceti různých textech. Tyto texty se nacházejí na přiloženém CD v adresáři texts/batch.

Aby bylo zpracování výsledků jednodušší, algoritmu jsme podstrčili otevřené texty jako šifrové texty a očekávali od něj, že nalezne klíč $ABCDEFGHIJKLMNOPQRST$. Algoritmy jsme srovnávali z hlediska správně uhodnutých písmen v klíči. Problém tohoto přístupu je, že se nějaké písmeno v šifrovém textu nemusí vyskytovat. V takovém případě nemá na výsledný rozluštěný text vliv a v našich testech jej zanedbáváme.

Algoritmy našly v naprosté většině případů stejné klíče. Pokud byla kvalita jednoho řešení horší, program nešťastně uvázl v lokálním minimu a při opětovném spuštění již našel lepší klíč. Oba programy se zasekly v lokálním minimu přibližně ve stejném procentu případů. Důvody, proč genetické algoritmy i ACO vykazují stejné výsledky, jsou následující:

Zaprvé při bystrém pohledu na algoritmy zjistíme, že dělají téměř to samé. V obou případech se náhodně vygeneruje množina řešení, která se ohodnotí a poté se řešení navzájem kombinují s tím, že dobře ohodnocené řešení má větší váhu. Rozdíl je v tom, že při použití genetických algoritmů jsme museli explicitně naprogramovat operace mutace a křížení, zatímco ACO tyto operace dělá samo a obecněji. V tom spatřujeme obecnou výhodu ACO oproti genetickým algoritmům. Mutace odpovídá situaci, kdy mravenec následuje hlavní cestu a jednou z ní odbočí. Při použití genetického algoritmu je možná pouze jedna mutace a jedno křížení dvou řešení během jedné iterace. Naproti tomu mravenec může z hlavní cesty odbočit mnohokrát, což odpovídá mnohonásobné mutaci, nebo libovolně kombinovat mnoho řešení svých předchůdců, což odpovídá křížení libovolného počtu jedinců.

Za druhé průběh účelové funkce je hladký. Najít optimální klíč vzhledem k fitness funkci by neměl být velký problém pro žádný sofistikovaný algoritmus. V naprosté většině případů, kdy algoritmus nenašel správný klíc, se ukázalo, že nalezený klíč má lepší fitness než správný klíč. Z toho vyplývá, že slabším článkem je fitness funkce, kvůli které oba algoritmy chybují.

Kapitola 7

Závěr

V této práci jsem nastudoval a popsal problematiku jednoduché substituční šifry a metaheuristiku Ant Colony Optimization. Dále jsem navrhl a implementoval program prolamující jednoduchou substituční šifru pomocí této metaheuristiky. Pro tento program jsem zjistil vhodné parametry spuštěním celkem 7800 simulací. S vhodnými parametry program téměř vždy našel optimální klíč vzhledem k fitness funkci.

Poté jsem navrhl a implementoval genetický algoritmus sloužící ke stejnému účelu. Tyto programy jsem navzájem srovnal. Zjistilo se, že programy nacházejí stejná řešení a dělají stejné chyby. To není překvapivé, jelikož v principu jsou si velmi podobné a navíc sdílejí nejslabší článek, kterým je fitness funkce. Výhodu ACO oproti genetickým algoritmům vidím v tom, že se provádí větší škála mutací a křížení, aniž by je bylo potřeba explicitně programovat.

Pokud bychom chtěli program ještě vylepšit tak, aby našel správné klíče na ještě kratších šifrových textech, měli bychom se zaměřit na vylepšení fitness funkce. Další způsob, jak na práci navázat, by bylo hledání strategie pozměňování parametrů za běhu simulace, aby docházelo ke střídání fáze explorace s fází exploatace tak, jak to dělá např. simulované žíhání. Tím by se zamezilo možnosti uváznutí v lokálním minimu.

Práce mě ujistila v tom, že metaheuristiky a kombinatorická optimalizace je odvětví, kterému bych se rád věnoval.

Podle očekávání nedošlo k žádnému průlomu v tom, jak umíme substituční šifru luštit.

Příloha A

Uživatelský manuál k programu

A.1 Spuštění aplikace v grafickém režimu

Aplikace je napsaná v programovacím jazyce java 6 a distribuovaná jako jar soubor. Ke spuštění je potřeba mít nainstalované Java Runtime Environment. To je ke stažení na adrese http://www.java.com/en/download/. V unixu se program spouší příkazem java -jar ACOSubstCracker.jar. Ve Windows se dá spustit program přímo dvojklikem.

A.2 Záložka Vstupy

Po spuštění aplikace se zobrazí záložka nadepsaná „Vstupy“. Zde zadáme všechny parametry a vstupní texty potřebné ke startu simulace. Parametry jsou následující:

- Alfa: Čím větší alfa, tím více se mravenci řídí podle feromonů
- Beta: Čím větší beta, tím více se mravenci řídí podle heuristické informace
- p: Faktor odpařování. Množství feromonu na vrcholech je po každé iteraci násobeno $1 - p$
- e/26: Množství dodatečného feromonu, který se položí na každý vrchol dosud nejlepší nalezené cesty

- w: Počet mravenců, kteří pokládají feromon v každé iteraci
- Mravenců: Počet mravenců v iteraci
- Počáteční f: Počáteční množství feromonu na každém vrcholu
- Zpoždění: Doba v milisekundách, na kterou je pozastaveno vlákno simulace po každé iteraci. Je pouze doporučením
- Ukončení: Simulace skončí, pokud proběhne tento počet simulací bez vylepšení řešení. Zadejte 0 pro nekonečnou simulaci

Dále je nutné vybrat druh fitness funkce a zadat šifrový a trénovací text. Pomocí tlačítka „Vyber soubor“ je možné tyto texty načíst ze souboru. Trénovací text by měl mít podobnou strukturu jako předpokládaný otevřený text. Minimálně by měl být ve stejném jazyce a měl by být dostatečně dlouhý. Několik dobrých anglických trénovacích textů je na přiloženém CD v adresáři texts/training.

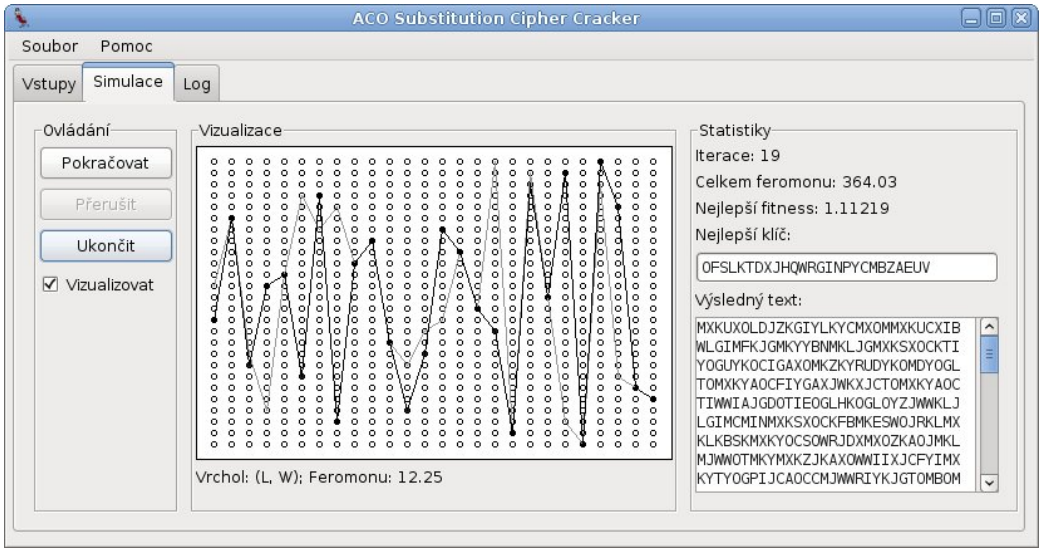
Zadané hodnoty není možné měnit za běhu simulace. K výměně vstupů obsažených v kartě Parametry Simulace je potřeba simulaci zastavit. Vstupy v kartách Fitness Funkce a Zašifrovaná zpráva není možné upravovat po startu simulace. Jedinou možností je simulaci ukončit. Poté je možné tyto vstupy měnit a začít nový výpočet.

A.3 Záložka Simulace

Po vyplnění vstupů můžeme přejít na záložku Simulace (obrázek A.1). V této založce se simulace startuje, přerušuje a ukončuje. Konstrukční graf, feromony na něm, a 2 dosud nejlepší nalezené cesty mohou být vizualizovány. Tato vizualizace se dá vypnout odškrtnutím checkboxu Vizualizovat.

Čím více feromonu je naneseno na vrcholu, tím je tmavší. Po najetí myši na nějaký vrchol je vypsáno, kterému bigramu vrchol odpovídá a množství feromonu na něm. Vrcholy, kterými prošel nejlepší dosud nalezený mravenec, jsou propojeny černou úsečkou. Vrcholy na cestě druhého nejlepšího mravence jsou spojeny šedivou čarou. Tyto cesty se často překrývají.

Během výpočtu se také zobrazují některé statistiky. Konkrétně iterace, ve které se simulace nachází. Dále celkový počet feromonů na grafu, fitness a cesta dosud nejlepšího nalezeného mravence a text, který vznikne použitím řešení, které tento mravenec nalezl.



Obrázek A.1: Záložka simulace

A.4 Záložka Log

Některé události se během průběhu simulace logují. V kartě ovládání si můžeme vybrat které. Kromě zaškrtnutých událostí se logují i vyhozené výjimky a parametry nově vzniklé simulace. Logování simulaci výrazně zpomaluje. Proto je dobré logovat jen události, které nás skutečně zajímají. Logování lze úplně vypnout odznačením checkboxu „Zobrazovat logy“.

Aktuálně zobrazený log je možné smazat nebo uložit do souboru.

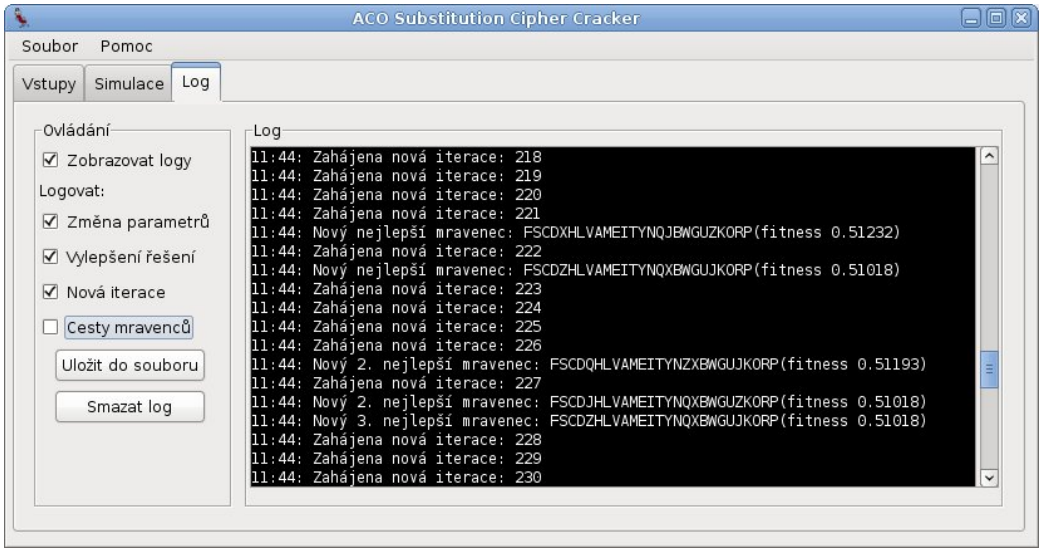
A.5 Konzolový režim

Program je možné spouštět i v konzolovém módu pomocí přepínače $-nogui$. V takovém případě se parametry simulace předávají z příkazové řadky. Pokud jsou specifikovány parametry, ale není použit přepínač $-nogui$, program se spustí v grafickém režimu a specifikované parametry jsou nastaveny jako výchozí hodnoty.

Význam parametrů a přepínačů je následující:

-help: vypíše, jak se program používá, a skončí

-nogui: Spustí simulaci v konzolovém režimu



Obrázek A.2: Záložka Logy

-nofinish: Nastaví nekonečnou simulaci

-finish (INTEGER): Simulace skončí, pokud ve specifikovaném počtu iterací vylepšeno řešení

-logiterations: Zapne logování iterací

-logpaths: Zapne logování cest všech mravenců

-logbest: Zapne logování nejlepších řešení

-alpha (DOUBLE): Parametr simulace α. Čím větší alfa, tím více se mravenci řídí podle feromonů

-beta (DOUBLE): Parametr simulace β. Čím větší beta, tím více se mravenci řídí podle heuristické informace

-e (DOUBLE): Množství dodatečného feromonu, který se položí na každý vrchol dosud nejlepší nalezené cesty

-initpheromone (DOUBLE): Nastaví počáteční množství feromonu na každém vrcholu

-ants (INTEGER): Nastaví počet mravenců v každé iteraci

-w (INTEGER): Počet mravenců, kteří pokládají feromon

-delay (INTEGER): Doba v milisekundách, na kterou je pozastaveno vlákno simulace po každé iteraci. Je pouze doporučením

-fitalpha (DOUBLE): Parametr α fitness funkce. Defaultní hodnota je 0

-fitbeta (DOUBLE): Parametr β fitness funkce. Defaultní hodnota je 1

-fitgama (DOUBLE): Parametr fitness funkce. Defaultní hodnota je 0

-fitsmoothing: Zapne ad hoc vyhlazování bigramů popsané v 4.1.5

Program může být spuštěn např. takto:

```
java -jar acosubstcracker.jar -nogui -alpha 1.2 -beta 0.7
-delay 20 -e 6 -ants 32 -ciphertext sifrovana_zprava.txt
-trainingtext kafka.txt
```

Příloha B

Genetický algoritmus

V této příloze je popsán genetický algoritmus, který jsem navrhl a implementoval za účelem srovnání s ACO algoritmem. Snažím se být maximálně stručný, jelikož detailní popis by se rozsahem vyrovnal zbytku práce.

B.1 Zakódování klíče do chromosomu

Pojmem chromosom myslíme uspořádanou 26tici znaků tak, že každý znak se v ní vyskytuje právě jednou. Chromosom tedy odpovídá klíči. Jednotlivé prvky této 26tice označujeme jako geny. Chromosomy jsou hodnoceny fitness funkcí. Jako fitness funkci jsem použil frekvenční analýzu bigramů.

B.2 Selekce

Proces selekce je výběr populace pro další generaci. Náhodně se vybere k jedinců z populace (k je jeden z parametrů). Tyto jedinci se seřadí podle fitness funkce. Nejlepší jedinec je vybrán s pravděpodobností p, kde p je další parametr. N-tý nejlepší jedinec je vybrán s pravděpodobností $p * (1-p)^{n-1}$.

Takto vybraný jedinec se buď do nové generace přidá rovnou nebo (s určitou pravděpodobností určenou parametrem) se kříží s jiným jedincem. Druhý jedinec potřebný ke křížení je vybrán obdobně. Jejich potomek s nějakou pravděpodobností (opět parametrizovanou) mutuje. Tento potomek se přidá do nové generace.

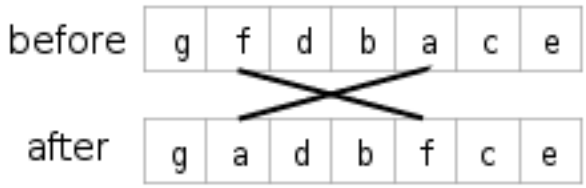
Tento proces se opakuje dokud nová generace neobsahuje požadovaný počet jedinců.

Algoritmus začal být výrazně efektivnější poté, co jsem implementoval princip elitářství. To znamená, že určité procento nejlepších jedinců je beze změny zachováno do generace další. Takto vybraní jedinci se nekříží a nemutují.

Dále určité malé procento jedinců v každé generaci je vytvořeno náhodně. Tímto se snažím zabránit možnému zabřednutí v lokálním minimu, hlavně při malé velikosti populace.

B.3 Mutace

Pro naše účely se hodí Swap Mutation Operator odpovídající transpozici. Dva náhodně vybrané prvky v chromosomu se prohodí. Proces mutace je vyobrazen na obrázku B.1.

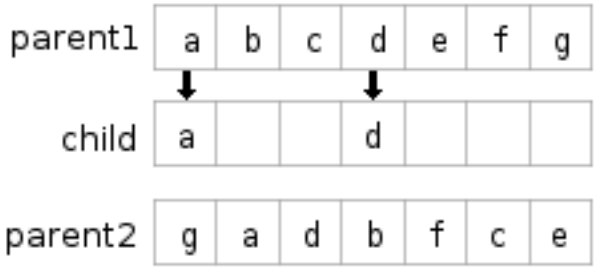


Obrázek B.1: Příklad mutace

B.4 Křížení

Křížení je již více komplikované. One Point Crossover nebo Two Point Crossover není možné použít, jelikož požadujeme, aby ve výsledném chromosomu byl každý prvek pouze jednou. Pro takové situace se používá Position Based Crossover, který se však tentokrát také nehodí. Proto jsem navrhl řešení vlastní. Mnou implementované křížení probíhá ve čtyřech krocích:

1. Každý gen z prvního předka se s pravděpodobností 1/3 zkopíruje do chromosomu potomka na stejné místo. První fáze křížení je znázorněna na obrázku B.2.



Obrázek B.2: První fáze křížení

2. Geny, které mohou (v potomku se takový gen ještě nevyskytuje a příslušná pozice je volná), se zkopírují z druhého předka. Tato fáze je znázorněna na obrázku B.3.

Obrázek B.3: Druhá fáze křížení

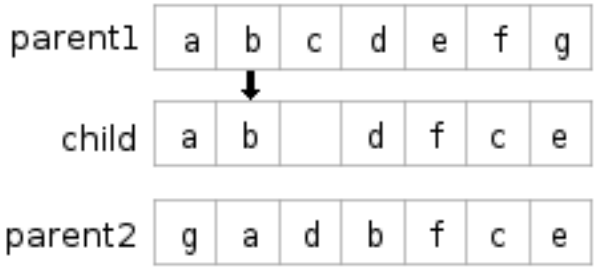
3. Geny, které mohou (v potomku se takový gen ještě nevyskytuje a příslušná pozice je volná), se zkopírují z prvního předka. Tato třetí fáze křížení je znázorněna na obrázku B.4.

4. Některé pozice v chromosomu potomka mohou zůstat prázdné. Obvykle se jedná o jednu až čtyři pozice. Zjistíme, jaké geny se v chromosomu ještě nevyskytují a náhodně je na takovéto pozice rozmístíme.

B.5 Používání

Popsané řešení jsem implementoval v jazyce Java. Program jsem pojmenoval gasc (Genetic Algorithm - Substitution Cipher).

Gasc se spouští z příkazové řádky. Má dva povinné argumenty. První je cesta k referenčnímu textu. Druhý je cesta k šifrovému textu.



Obrázek B.4: Třetí fáze křížení

Program se dá spustit takto: `java -jar gasc.jar kafka-the_trial.txt encrypted3884`

Další volitelné parametry jsou:

crossoverprob (Crossover probabilty): Pravděpodobnost, že se vybraný jedinec bude křížit

mutationprob (Mutation probability): Pravděpodobnost, se kterou nově vzniklý potomek bude mutovat

populationsize (Population size): Počet jedinců v populaci (na generaci)

tournamentsize (Tournament size): Velikost turnaje (parametr k ze sekce Selekce)

tournamentp (Tournament winner probability): Parametr p ze sekce Selekce

elitarism (Elitarism): Podíl jedinců, kteří se do kazdé generace dostanou principem elitářství (0.2 znamená 20%)

seeding (Seeding): Podíl jedinců, kteří jsou v každé generaci nahodně vygenerováni (0.02 znamená 2%)

generations (Number of generations): Počet generací, které se vygenerují

Pokud se některé parametry neuvedou, použijí se implicitní hodnoty:

- Crossover probabilty: 0.85
- Mutation probability: 0.05

- Population size: 500
- Tournament size: 20
- Tournament winner probability: 0.7
- Elitarism: 0.2
- Seeding: 0.02
- Number of generations: 100

Spuštění může vypadat například takto:

```
java -jar gasc.jar kafka-the_trial.txt encrypted3884 -crossoverprob
0.7 -mutationprob 0.03 -populationsize 250 -tournamentsize 15 -tournamentp
0.8 -elitarism 0.1 -seeding 0.01 -generations 250
```

Příloha C

Obsah CD

K této práci je přiloženo CD. Nachází se na něm tyto soubory a adresáře:

- bakalarska-prace.pdf Tato bakalářská práce ve formátu PDF
- ACOSubstCracker.jar Spustitelný program. Ke spuštění je potřeba Java Virtual Machine. V unixu se program spouští následujícím příkazem:

  ```
  java -jar ACOSubstCracker.jar
  ```

  Ve Windows se dá soubor spustit přímo
- readme Textový soubor obsahující informace o obsahu CD
- src Adresář obsahující zdrojové kódy
- doc Adresář obsahující uživatelskou dokumentaci k programu
- doc_devel Adresář obsahující vývojářskou dokumentaci k programu
- lib Adresář obsahující používané knihovny
- texts Adresář obsahuje několik textů ve formátu plaintext. V podadresáři training se nachází dlouhé anglické texty vydané jako public domain. V podadresářích ciphertexts a openmessages jsou zašifrované, respektive otevřené zprávy. V podadresáři batch jsou texty, které jsem použil pro srovnání s genetickým algoritmem. Texty použité pro hledání optimálních parametrů se nacházejí v podadresáři 360
- gasc Adresář obsahující genetický algoritmus

Literatura

[1] Blum C., Roli A., Dorigo M.: *HC-ACO: The Hyper-Cube Framework for Ant Colony Optimization*, Proceedings of MIC'2001 – Metaheuristics International Conference, vol. 2, 399-403 , 2001.

[2] Bullnheimer B., Hartl R. F., Strauss C.: *A New Rank-Based Version of the Ant System: A Computational Study*, Central European Journal for Operations Research and Economics, **7(1)**, 25-38, 1999.

[3] Chen S. F., Goodman J.: *An empirical study of smoothing techniques for language modeling*, Proceedings of the Thirty-Fourth Annual Meeting of the Association for Computational Linguistics, 1996, 310-318.

[4] Chu S. C., Roddick J. F., Pan J.S., Su C. J.: *Parallel Ant Colony Systems*, Foundations of Intelligent Systems, Lecture Notes in Computer Science 2871, Springer Berlin / Heidelberg, 279-284, 2003.

[5] Clark A. J.: *Optimisation Heuristics for Cryptology*, PhD Thesis, Queensland University of Technology, 1998.

[6] Deneubourg J.-L., Aron S., Goss S., Pasteels J.-M.: *The Self-Organizing Exploratory Pattern of the Argentine Ant*, Journal of Insect Behavior **(3)**, 159-168, 1990.

[7] Diaconis P.: *The Markov Chain Monte Carlo Revolution*, Bulletin of the American Mathematical Society **46**(2009), 179-205.

[8] Dorigo M.: *Optimization, Learning and Natural Algorithms*, PhD thesis, Politecnico di Milano, 1992.

[9] Dorigo M., Birattari M., Stützle T.: *Ant Colony Optimization: Artificial Ants as a Computional Intelligence Technique*, Technical Report Number TR/IRIDIA/2003-023, IRIDIA, Université Libre de Bruxelles.

[10] Dorigo M., Gambardella L. M.: *Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem*, IEEE Transactions on Evolutionary Computation, **6(4)**, 317-365, 1997.

[11] Dorigo M., Stützle T.: *Ant Colony Optimization*, The MIT Press, 2004.

[12] Dorigo M., Stützle T.: *The Ant Colony Optimization Metaheuristic: Algorithms, Applications, and Advances*, Handbook of Metaheuristics, volume 57 of International Series in Operations Research & Management Science, Kluwer Academics Publishers, 2002, 251-285.

[13] Garici M. A., Drias, H.: *Cryptanalysis of Substitution Ciphers Using Scatter Search*, Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach, Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2005, 31-40.

[14] Hasinoff S. W.: *Solving Substitution Ciphers*, Technical Report, University of Toronto, Department of Computer Science, 2003.

[15] Jakobsen T.: *A Fast Method for Cryptoanalysis of Substitution Ciphers*, Cryptologia, **19(3)** (1995) 265–274.

[16] Johnson D. S., McGeoch L. A.: *The travelling salesman problem: A case study in local optimization*, Local Search in Combinatorial Optimization, John Wiley & Sons, 1997, 215-310.

[17] Lewand R. E.: *Cryptological Mathematics*, The Mathematical Association of America, 2000.

[18] Maniezzo V.: *Exact and Approximate Nondeterministic Tree-Search Procedures for the Quadratic Assignment Problem*, INFORMS Journal on Computing, **11(4)**, 358-369, 1999.

[19] Mao W.: *Modern Cryptography: Theory & Practice*, Prentice Hall, 2003.

[20] Russell M., Clark J. A., Stepney S.: *Using Ants to Attack a Classical Cipher*, Genetic and Evolutionary Computation - GECCO 2003, Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2003, 199.

[21] Singh S.: *Kniha kódů a šifer*, Dokořán, Argo, 2003.

[22] Stützle T., Hoos H. H.: $\mathcal{MAX} - \mathcal{MIN}$ *Ant System*, Future Generation Computer Systems **16(8)**, 2000, 889-914.

[23] Verma A. K., Dave M., Joshi R. C.: *Genetic Algorithm and Tabu Search Attack on the Mono-Alphabetic Substitution in Adhoc Networks*, Journal of Computer Science **3(3)**, Science Publications, 2007, 134-137.

[24] Vondruška P.: *Kryptologie, šifrování a tajná písma*, Albatros, 2006.