VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

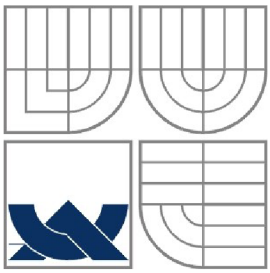
IMPLEMENTACE A VIZUALIZACE KLASICKÉHO GENETICKÉHO ALGORITMU ZA POUŽITÍ METROPOLISOVA ALGORITMU

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

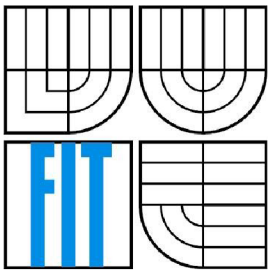
AUTOR PRÁCE
AUTHOR

RADEK MATULA

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

IMPLEMENTACE A VIZUALIZACE KLASICKÉHO GENETICKÉHO ALGORITMU ZA POUŽITÍ METROPOLISOVA ALGORITMU

IMPLEMENTATION AND VISUALIZATION OF CLASSIC GENETIC ALGORITHM USING METROPOLIS ALGORITHM

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE
AUTHOR

RADEK MATULA

VEDOUCÍ PRÁCE
SUPERVISOR

ING. MILOŠ OHLÍDAL

BRNO 2007

Abstrakt

Tato bakalářská práce popisuje využití genetického a Metropolisova algoritmu k řešení problému obchodního cestujícího. Dále popisuje průběh vývoje aplikace POC a vysvětluje problematiku nastavení jednotlivých parametrů algoritmu.

Klíčová slova

Genetický algoritmus, problém obchodního cestujícího, Metropolisův algoritmus, Nicholas Constantine Metropolis, Metropolisovo kritérium, metoda Monte Carlo, simulované žíhání, wxWidgets, křížení, mutace, populace, gen, chromozóm, elitářský mechanismus, generace, C++, python, sed, skript, Mersenne Twister

Abstract

This bachelor's thesis contains description of utilisation genetic and Metropolis algorithm to solution the Traveling Salesman Problem (TSP). Thesis describes process of development aplication POC and explain problems with adjusting parameters of algorithm.

Keywords

Genetic algorithm, traveling salesman problem, TSP, metropolis algorith, Nicholas Constantine Metropolis, Metropolis criterion, method Monte Carlo, simulated annealing, wxWidgets, crossover, mutation, population, gene, chromosome, elitism, generation, C++, python, sed, script, Mersenne Twister

Citace

Radek Matula: Implementace a vizualizace klasického genetického algoritmu za použití Metropolisova algoritmu, bakalářská práce, Brno, FIT VUT v Brně, 2007

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Miloše Ohlídala. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Radek Matula
červenec 2007

Poděkování

Děkuji Ing. Ohlídalovi za vedení a velice přínosné rady při zpracovávání této bakalářské práce.

© Radek Matula, 2007.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod

Účelem této práce bylo, jak je uvedeno v zadání, vytvořit jednoduchou aplikaci sloužící k řešení problému obchodního cestujícího (TSP – Traveling Salesman Problem). Tuto aplikaci jsem podle Problému Obchodního Cestujícího nazval POC. Jádrem tohoto programu je klasický genetický algoritmus a dále se využívá Metropolisova algoritmu. Asi hlavním důvodem, proč jsem si toto téma vybral, bylo, že jsem znal problematiku TSP, ale netušil jsem, že lze řešit i jinak než hrubou silou. Proto mě zajímalo, co je na genetických algoritmech tak zázračného, že dokáží být při řešení této úlohy někdy až mnohonásobně výkonnější, než metoda hrubé síly.

To hlavní, co genetické algoritmy nabízí, je v podstatě aplikace principů (reprodukce, dědičnost, mutace, přirozený výběr, křížení), které příroda úspěšně používá již po tisíciletí. Obor, ze kterého pocházejí, je tedy biologie. Naopak Metropolisův algoritmus má své kořeny ve fyzice, protože vychází z fyzikálního děje, probíhajícího při žíhání tuhých těles. Toto žíhání se používá k odstranění vnitřních defektů. Principy i z takto zdánlivě neslučitelných oborů lze tedy s úspěchem aplikovat v dnešní informatice.

V následující kapitole je rozebrána problematika obchodního cestujícího. Další kapitola této práce se věnuje problematice genetického algoritmu a hlavním jeho operacím jako křížení a mutace. Následuje text věnující se podrobně Metropolisově algoritmu a výpočtu Metropolisova kritéria.

Text pokračuje oddílem věnujícím se samotné aplikaci POC. Je zde popsán vývoj výpočetního jádra, kde jsou zapracovány genetický a Metropolisův algoritmus. Pokračuje se podkapitolou věnující se popisu implementovaných vizualizací. Další podkapitola se věnuje popisu fungování aplikace z globálního hlediska. Text nezabíhá do nějakých implementačních detailů. K tomu slouží samotné zdrojové kódy, které jsou dobře komentované a dokumentace vygenerovaná nástrojem doxygen. Vše se nachází na přiloženém CD. Na tomto CD se taky nacházejí dva skripty které jsem během testování používal pro úpravu vstupních dat. Jejich popisu se věnuje další podkapitola. V závěru tohoto oddílu jsou popsány základní požadavky na cílovou platformu pro bezproblémovou kompilace a provoz aplikace.

Pak následuje kapitola, která se věnuje porovnávání kvality nalezeného řešení v závislosti na nastavení nejrůznějších parametrů.

Předposlední kapitola popisuje adresářovou strukturu CD dodávaného jako příloha.

Závěrečná kapitola pak shrnuje dosažené výsledky práce a přináší další náměty, jak by mohl vývoj této aplikace pokračovat.

2 Problém obchodního cestujícího

Jedná se o diskrétní optimalizační problém, jehož cílem je nalézt nejkratší možnou trasu, procházející všemi zadanými body (městy) a vracející se nazpět do výchozího bodu (města). Tato trasa může vést jen po dostupných cestách spojujících různá města a u nichž známe taky jejich délku[1].

TSP patří mezi tzv. NP-úplné (nedeterministicky polynomiální) úlohy. To znamená, že není známo, jak nalézt přesné řešení v rozumném čase, dokonce není ani známo, zda vůbec může existovat algoritmus, který takové řešení najde v čase úměrném nějaké mocnině počtů bodů. Tento problém má faktoriálovou složitost. Pro N bodů existuje až (N – 1)! možných cest (platí pro variantu, kdy existují cesty mezi všemi body navzájem).

Z toho také vyplývá, že problém je nemožné řešit metodou hrubé síly pro větší počty měst. Například kdybychom byli schopni za 1 s vyhodnotit a porovnat jednu miliardu kombinací, tak už jen při 20 bodech by trvalo prohledání celého stavového prostoru téměř 4 roky. Při 21 bodech by už to bylo 77 let, atd.

V praxi se tedy obvykle využívají algoritmy, které jsou založeny na určitých heuristikách (zde patří i genetický algoritmus a Metropolisův algoritmus). Tyto algoritmy jsou schopné v rozumném čase nalézt řešení, které je velmi blízké optimu. A s ním se v praxi většinou spokojíme.

1 Ve své aplikaci POC předpokládám, že mezi každými body existuje cesta. Tedy že se z libovolného bodu mohu dostat do libovolného jiného bodu.

3 Genetický algoritmus

Genetický algoritmus (dále jen GA) je postup, vycházející z Darwinovy evoluční teorie, který se snaží aplikací principů evoluční biologie nalézt řešení složitých problémů, pro které neexistuje použitelný exaktní algoritmus. GA používají techniky napodobující evoluční procesy známé z biologie (reprodukce, dědičnost, mutace, přirozený výběr, křížení) pro řešení dané úlohy.

Genetické algoritmy jsou poměrně mladou disciplínou a i nyní jsou předmětem intenzivního studia. Jejich zrod je datován do roku 1975 a je spojen zejména se jménem John Holland, který se v té době věnoval studiu buněčných automatů na Michiganské Univerzitě.

Principem práce tohoto algoritmu je postupná tvorba nových generací různých řešení daného problému. Při výpočtu se generuje vždy jen určitý počet řešení, tzv. populace. Každý jedinec této populace tedy představuje jedno řešení daného problému. Řešení je obvykle zakódováno ve vhodném tvaru (např. jako posloupnost písmen nebo čísel), který je závislý na řešeném problému. Takto zakódovanému řešení se v terminologii GA říká chromozóm. Jednotlivé položky, z nichž je chromozóm složen se nazývají geny. Nová populace vzniká výběrem potomků, kteří vznikli křížením chromozómů z původní populace. Tito potomci mohou ještě s určitou pravděpodobností zmutovat. Tento výběr probíhá zpravidla na základě ohodnocení jejich kvality. Toto ohodnocení se nazývá fitness (v biologii je definováno jako relativní schopnost přežití a reprodukce). Čím je tedy daný chromozóm kvalitnější, tím má větší fitness. Takto vzniklá nová populace nahradí populaci původní. Existuje mnoho variant GA, kde jsou hlavní rozdíly ve způsobu výběru chromozómů, postupujících do další populace (vybírá se mezi potomky, vybírá se mezi potomky i rodiči, vybírá se vždy jen jeden postupující atd.). Tento postup se iterativně opakuje, čímž se kvalita řešení v populaci postupně vylepšuje. Proces konverguje k situaci, kdy je populace tvořena jen těmi nejlepšími jedinci. Algoritmus se obvykle zastaví při dosažení postačující kvality řešení, případně po předem dané době.

Křížení tedy znamená, že z populace vybereme náhodný pár chromozómů u nich provedeme vzájemnou výměnu určitých stejně dlouhých částí genetické informace. Podle počtu těchto částí rozlišujeme křížení jednobodové, dvoubodové atd.

Naproti tomu mutace je proces, kdy dochází ke změně pouze v rámci jednoho chromozómu, tedy chromozóm nepřebírá genetickou informaci z cizího chromozómu. Nejčastěji je mutace prováděná změnou hodnoty jednoho nebo více genů chromozómu nebo jejich vzájemnou záměnou.

Jeho velkou výhodou je poměrná jednoduchost. Nepotřebujeme znát žádné bližší informace o hledaném řešení (např. derivace funkce apod.), ale stačí nám pouze umět ohodnotit kvalitu tohoto řešení. Díky mutacím se zabraňuje situaci, kdy by proces hledání sklouznul do oblasti nějakého lokálního optima. GA vykazuje velmi dobré výsledky u problémů s rozsáhlými množinami přípustných řešení. Díky těmto vlastnostem bývá s úspěchem využit při řešení nejrozmanitějších optimalizačních problémům (mezi které patří i TSP).

Mezi nevýhody GA bych zařadil potřebu velkého množství vyhodnocování cílové funkce (tedy ohodnocování chromozómů). V případě, kdy je tato funkce výpočetně velice náročná, může být použití GA horší variantou než použití jiných metod. Další slabinou GA je opakované generování již jednou vygenerovaných jedinců. Algoritmus má tedy problém s nalezením úplně přesného řešení také proto, že nedisponuje ani žádnými mechanismy, jak poznat, že konkrétní řešení je absolutně nejlepší.

3.1 Křížení

Jak již bylo uvedeno výše, během křížení dochází k výměně genetických informací mezi dvěma náhodně vybranými chromozómy. V této kapitole bude popsán postup jednobodového křížení, protože přesně takové jsem použil v mém programu POC.

Definice jednobodového křížení zní takto: Z populace vybereme náhodné páry a náhodně zvolíme pozici k. Tato pozice se nazývá bod křížení. Do prvního potomka zkopírujeme geny 1 až k prvního rodiče a geny k + 1 až n druhého rodiče, kde n je počet genů, a do druhého potomka kopírujeme geny opačně, tedy 1 až k z druhého rodiče a geny k + 1 až n z prvního rodiče.

V mém případě, kdy je chromozóm tvořen permutací souřadnic jednotlivých bodů, musíme ještě provést opravu každého chromozómu. Opravu musíme provést z důvodu porušení permutace. Může totiž vzniknout chromozóm, kde budou některé souřadnice dvakrát a některé budou zcela vypuštěny. K opravě použijeme funkci zobrazení rodiče1 na rodiče2 a funkci zobrazení rodiče2 na rodiče1. Toto zobrazení nám stačí pouze od genů, nacházejících se za bodem křížení. Pokud máme v chromozómu duplicitní hodnotu, vždy opravujeme tu, která se nachází před bodem křížení. Vše bude dobře znázorněno na následujícím příkladu:

Máme chromozómy A = (3 4 5 1 2) , B = (1 3 2 5 4)[2] a bod křížení[3] jsme náhodně vygenerovali jako pozici 2. Provedeme tedy výměnu genetické informace a vzniknou nám tyto chromozómy:

A′ = (3 4 2 5 4) a B′ = (1 3 5 1 2)

Při bližším prozkoumání zjistíme, že zde už byla permutace porušena. Např. v chromozómu A′ je dvakrát hodnota 5, ale zcela tam chybí hodnota 1. Budeme muset tedy použít ony zmíněné funkce zobrazení. Ty vypadají následovně:

$$f_{AB}=\begin{pmatrix}5&1&2\\2&5&4\end{pmatrix} \quad \text{a} \quad f_{BA}=\begin{pmatrix}2&5&4\\5&1&2\end{pmatrix}$$

K opravě chromozómu A′ použijeme zobrazení f_{BA} , k opravě B′ zobrazení f_{AB} a to následovně: Procházíme chromozóm od počátku a hledáme duplicitní geny. Pokud nějaký objevíme, nalezneme si pomocí daného zobrazení jeho obraz a tímto obrazem ho přepíšeme. Poté ale musíme opětovně zkontrolovat, zda i tento obraz se už v chromozómu nenachází. Pokud ano, celou akci

2 Ve skutečnosti je v mém programu gen tvořen strukturou obsahující pozice x a y daných bodů a ne jedním číslem, jak je tomu v tomto příkladě.

3 Pozice genů číslujeme od 1

opakujeme (tentokrát ale s minule nalezeným obrazem) dokud se daný gen nepřepíše na neduplicitní hodnotu. Takto postupně projdeme a opravíme celý chromozóm. U našeho chromozómu A´ máme duplicitně pouze hodnotu 4. Pomocí zobrazení f_{BA} ji nejprve opravíme na hodnotu 2. Bohužel i tato hodnota se již v chromozómu nachází, proto hledáme obraz čísla 2. Obrazem je číslo 5, toto už se ale také v chromozómu nachází, a proto hledáme jeho obraz. Tím je hodnota 1, která v chromozómu dosud chyběla. Našli jsme tedy konečnou hodnotu a protože se v chromozómu nenachází už žádné duplicitní geny, oprava je u konce. Obdobně opravíme gen B´, ale s využitím zobrazení f_{AB} . Vznikli nám tedy dva výslední potomci:

A* = (3 1 2 5 4) a B* = (4 3 5 1 2)

3.2 Mutace

Protože je v mé aplikaci chromozóm realizován jako permutace souřadnic bodů, musí být i po provedení mutace výsledný chromozóm permutací. Proto mutaci provádím pomocí výměny dvou vybraných genů chromozómu mezi sebou. Díky tomu mám jistotu, že vzniklé řešení je regulérní, protože se stále jedná o permutaci stejných bodů. V aplikaci používám dva druhy mutací.

Jedna mutace provádí výměnu dvou náhodně vybraných genů chromozómu mezi sebou. Dále v textu budu tuto mutaci označovat jako mutace1. Druhá mutace je sofistikovanější. Náhodně vybere pouze jeden gen chromozómu. Na základě tohoto výběru potom prohledává celý chromozóm, kde hledá dva nejbližší geny na základě hodnot souřadnic. Tyto dva nalezené geny potom vymění za předchůdce a následníka tohoto náhodně vybraného genu. Tuto mutaci budu dále označovat jako mutace2. U mutace1 se tedy provádí jedna výměna, kdežto u mutace2 se provádí výměny dvě.

4 Metropolisův algoritmus

Jak už jsem uvedl v úvodu, tak inspirací tomuto algoritmu bylo žíhání tuhých těles. Tato operace se provádí k odstranění vnitřních defektů v tělese. Provádí se tak, že se nejprve těleso zahřeje na vysokou teplotu, která se postupně velice pomalu snižuje. Zahřátím získají atomy tělesa energii, která jim umožní překonávat lokální energetické bariéry a dostat se do rovnovážných poloh. Postupné snižování teploty má za následek, že rovnovážné polohy atomů se stabilizují. Pokud je snižování teploty dostatečně pomalé, tak se těleso při každé teplotě nachází v tepelné rovnováze. Tato rovnováha odpovídá Boltzmannovu rozdělení pravděpodobnosti, že při teplotě T je těleso v tepelné rovnováze. Při konečné teplotě žíhání, která bývá podstatně nižší než počáteční, jsou všechny atomy v rovnovážných polohách a těleso již neobsahuje žádné vnitřní defekty.

Řecko-americký matematik Nicholas Constantine Metropolis (1915 – 1999) spolu s kolegy (A. Rosenbluth, M. Rosenbluth, A. Teller a E. Teller) spojil tyto principy s numerickou simulací pomocí metody Monte Carlo. Podle tohoto vědce dostal algoritmus také svůj název. Algoritmus se používá k řešení optimalizačních problémů.

Algoritmus pracuje na základě simulace evoluce systému pomocí generování nových stavů systému tímto způsobem: Na počátku se nastaví teplota na vysokou hodnotu. Aktuální stav systému se mírně naruší (částice jsou libovolně mírně posunuty). Spočte se rozdíl v energii neporušeného stavu a porušeného stavu částic tělesa ($\Delta E = E_{NEPORUŠENÉ} - E_{PORUŠENÉ}$). Pokud je tento rozdíl záporný, proces pokračuje s novým porušeným stavem. Jinak přijmeme porušený stav pouze s určitou pravděpodobností, která je závislá na velikosti vypočteného rozdílu energií a aktuální teplotě systému. Tato pravděpodobnost se nazývá Metropolisovo kritérium. Díky občasnému akceptování i horších stavů je tento algoritmus odolný proti sklouznutí do oblasti lokálního minima. Aplikováním velkého počtu porušení stavů a akceptováním těchto poruch, získáme systém v tepelné rovnováze a rozložení těchto stavů se bude velmi (asymptoticky) blížit Boltzmannovu rozložení. Během výpočtu se postupně snižuje teplota (a díky tomu se i mění pravděpodobnosti přijetí horších stavů). Snižování teploty se obvykle provádí pomocí násobení aktuální teploty určitým koeficientem.

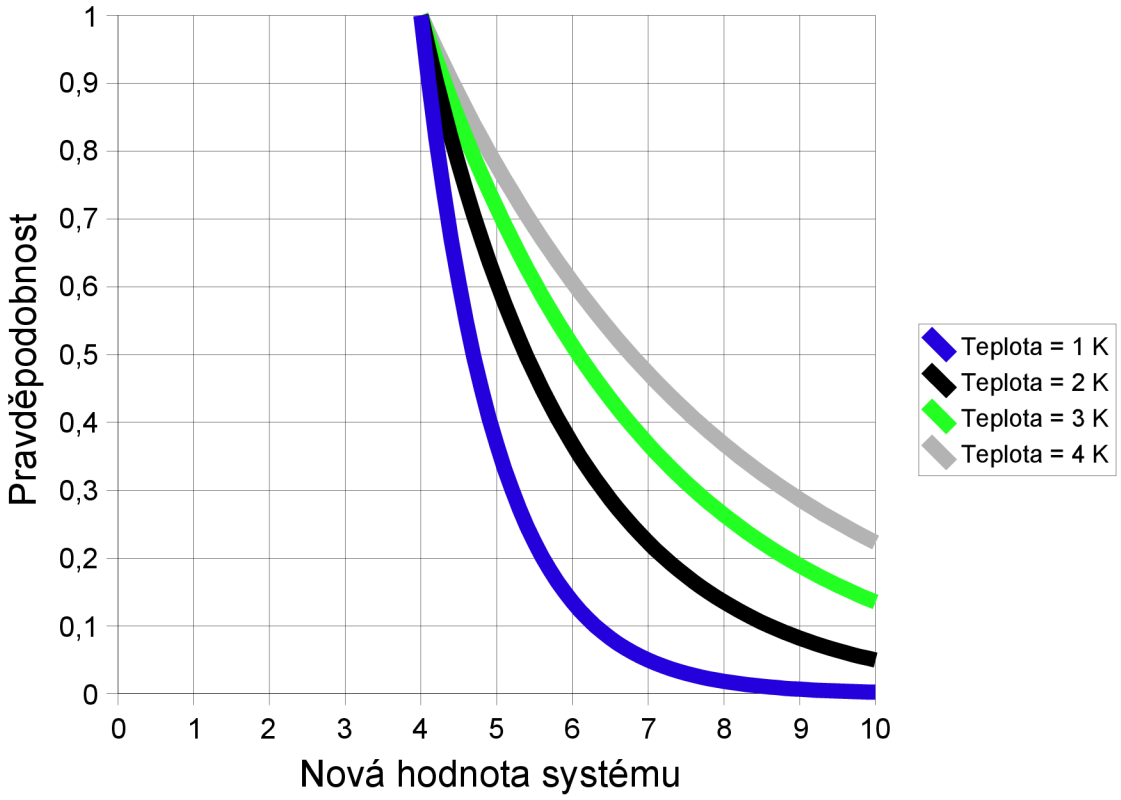
Algoritmus je na správné nastavení parametrů hodně choulostivý. Například pokud by byla maximální teplota příliš nízká, hrozí uváznutí v lokálním minimu. Naopak při nastavení velmi vysoké počáteční teploty, by algoritmus dlouho akceptoval téměř každé řešení. Tím by také postrádal heuristiku, podle které by pracoval a velice by se tak podobal metodě slepého prohledávání stavového prostoru. Podle literatury bývá správnou volbou nastavení počáteční teploty na takovou hodnotu, aby algoritmus přijal přibližně 50 % všech horších stavů systému. Koeficient podle kterého se snižuje teplota se doporučuje nastavit v rozmezí 0,99 – 0,8 (tedy snižovat teplotu v každém kroku o 1 – 20 %).

4.1 Metropolisovo kritérium

Metropolisovo kritérium a je určeno vztahem $e^{-\frac{\Delta E}{kT}}$, kde e je Eulerovo číslo ($e \approx 2{,}718$), ΔE je rozdíl energií v J (viz. výše), k je Boltzmannova konstanta (k = 1,380662.10^{-23} JK^{-1}) a T je aktuální teplota systému ve stupních Kelvina.

V mé aplikaci POC se ale nepracuje se žádnými částicemi a energiemi. Místo energie počítám délku cesty mezi jednotlivými body (cesta obchodního cestujícího), takže ΔE ve vztahu nahrazuje rozdíl vzdáleností. A protože nepočítám s částicemi, vypustil jsem ze vztahu i Boltzmannovu konstantu. Teplota ovšem zůstává, protože je to důležitá veličina, která v průběhu výpočtu ovlivňuje pravděpodobnost akceptování aktuálního stavu. Zde už není nějaká souvislost s teplotou v pravém slova smyslu, a proto můžeme parametr T nazývat například řídícím parametrem.

Na následujícím grafu je znázorněn průběh Metropolisova kritéria pro různé teploty systému[4], pokud měl neporušený stav hodnotu (energie, délky cesty, ...) 4[5]:



Z grafu je patrné, jak s klesající teplotou význačně klesá pravděpodobnost přijetí horšího stavu. Například hodnota 5 má pravděpodobnost přijetí při teplotě 4 K téměř 80 %, kdežto při teplotě 1 K to je už jen necelých 40 %.

4 Z výpočtu pravděpodobnosti už je zcela vypuštěna hodnota Boltzmannovy konstanty

5 Jednotky jsou zde nepodstatné

Jako určitou nevýhodu tohoto algoritmu pro využití v mé aplikaci vidím způsob, jakým počítá změnu energie systému. Jedná se o pouhý rozdíl energií. V mém případě, kdy místo energií počítám s délkou cesty, bych spíš uvítal vztah, kde by se vyskytoval podíl energií (vzdáleností). Jako příklad k odůvodnění by mohla posloužit situace, kdy máme vygenerováno řešení obchodního cestujícího, které má délku 100. Jeho narušením získáme řešení nové o délce 105. Teplota systému je 100 K. Pravděpodobnost akceptování tohoto o 5 % horšího stavu tedy je:

$$P = e^{-\frac{105-100}{100}} = e^{-\frac{5}{100}} \approx 0{,}95$$

Pokud ale použijeme obdobné rozložení bodů, jen jejich souřadnice vynásobíme 10x bude mít aktuální stav hodnotu 1000 a stav po narušení 1050 (tedy stále se jedná o 5% horší řešení, než řešení původní). Teplota zůstane stále na 100 K. Pravděpodobnost akceptování tohoto stavu je:

$$P = e^{-\frac{1050-1000}{100}} = e^{-\frac{50}{100}} \approx 0{,}61$$

Pravděpodobnost tedy výrazně poklesla. Řešení tohoto problému spočívá v nastavení počáteční a koncové teploty na 10x vyšší hodnoty. Poté bude algoritmus pracovat s velice podobnými výsledky jako při předchozím rozložení bodů. Kdyby tedy existoval vztah pro Metropolisovo kritérium využívající podíl kvalit jednotlivých řešení, nebyly by výsledky práce Metropolisova algoritmu tolik ovlivňovány nastavením maximální a minimální teploty systému. Snažil jsem se takový vztah nalézt v literatuře nebo vymyslet, ale bohužel neúspěšně. Vždy mi totiž vycházely mírně odlišné pravděpodobnosti přijetí horšího stavu než u originálního Metropolisova kritéria. Z toho plyne, že by výsledný systém nebyl v tepelné rovnováze a distribuce pravděpodobnosti rozložení stavů by byla výrazně odlišná od Boltzmannova rozdělení pravděpodobnosti.

5 Aplikace POC

Jak už jsem uvedl v úvodu, tak název POC vnikl od problému, který tato aplikace řeší. Aplikaci jsem vyvíjel pod operačním systémem Linux (konkrétně distribuce Kubuntu 7.04) v programovacím jazyce C++.

Pro implementaci grafické nadstavby jsem využil multiplatformní open source knihovny wxWidgets verze 2.8.0 (viz. *www.wxwidgets.org*). Tato knihovna je distribuovaná pod wxWindows licencí, která je velmi blízká L-GPL licenci. Díky využití wxWidgets by měla být aplikace po mírné úpravě zdrojového kódu bez problémů přeložitelná a funkční i na jiné platformě (Windows, Mac, …).

Dále byl při implementaci aplikace využit open source generátor náhodných čísel Mersenne Twister, který je pod BSD licencí. Generátor jsem získal z internetové adresy *http://www-personal.engin.umich.edu/~wagnerr/MersenneTwister.html*. Tento generátor vykazuje větší náhodnost u generované posloupnosti a je přibližně 4x rychlejší než klasický generátor náhodných čísel zabudovaný v jazyce C. Navíc zde odpadá klasická inicializace generátoru (provádí se automaticky ze souboru */dev/urandom* a nebo aktuálním časem) a generátor je schopen generovat jak celé čísla, tak i čísla desetinná.

Ke snadnému zkompilování aplikace slouží již připravený makefile skript.

5.1 Koncepce výpočetního jádra aplikace

V průběhu vývoje aplikace prošlo její výpočetní jádro mnoha změnami.

Jedním z prvních konceptů výpočetního jádra byl tento: Iterativním aplikováním mutace (mutace1) na vstupní chromozóm vznikne vstupní populace. Pořadí genů ve vstupním chromozómu je dáno pořadím vložení jednotlivých bodů uživatelem na simulační plochu. Vzniklá vstupní populace vstupuje do Metropolisova algoritmu, kde se postupně snižuje teplota. S využitím principu genetického algoritmu se náhodně vyberou ze vstupní populace dva rodičovské chromozómy, nad kterými se provede operace křížení a vzniknou tak dva potomci. S určitou pravděpodobností (kterou má možnost určit uživatel aplikace) se ještě nad potomky provede operace mutace (opět mutace1). Takto vzniklí potomci se potom porovnávají s dosud nejlepším nalezeným řešením. Pokud se potomek jeví jako lepší řešení, automaticky postupuje do další generace. V jiném případě se provede ohodnocení Metropolisovým kritériem, vygeneruje se náhodné desetinné číslo v rozsahu 0 – 1 a řešení postoupí do další generace pouze v případě, kdy je toto číslo menší nebo rovno vypočtenému Metropolisově kritériu. Takto se už ohodnocují i oba rodičovské chromozómy, které tak mají určitou šanci být i v další generaci. Tyto operace (křížení, mutace, ohodnocení, ...)se provádějí tak dlouho, dokud se nenaplní výstupní populace. Tato výstupní populace v dalším průchodu Metropolisovým algoritmem (kde je teplota opět snížena) představuje vstupní populaci. Výpočet

končí, když teplota poklesne na předem stanovenou hodnotu. Při ochlazování pomocí násobení aktuální teploty určitým koeficientem se obvykle nepodaří dosáhnout přesné hodnoty. Proto se výpočet zastavuje na mírně vyšší teplotě. Aplikací dalšího ochlazení by totiž teplota poklesla pod minimální teplotu (zadanou uživatelem).

Tento koncept plnil svou úlohu a během výpočtu neustále vylepšoval nalezené řešení. Nicméně i přes toto jsem hledal další způsoby, jak průběh výpočtu urychlit a nacházet tak při stejném nastavení vstupních parametrů algoritmu kvalitnější řešení. Jedním z těchto urychlení bylo zapracování tzv. elitářského mechanismu do výpočtu. Tento mechanismus automaticky posouvá do další generace určitý počet (zadaný uživatelem) nejlepších řešení. Tyto řešení mají totiž velkou pravděpodobnost, že určité částí jejich genetické informace jsou totožné s genetickou informací optimálního řešení. Proto se budou v další populaci hodit jako kvalitní rodičovské chromozómy pro operaci křížení. Výpočetní jádro aplikace bez elitářského mechanismu totiž mohlo tyto kvalitní řešení ignorovat a místo nich poslat do další generace řešení horší. Vše záleželo pouze na tom, zda se tato řešení vyberou pro operaci křížení a také jací potomci z křížení vzejdou. Po zapracování tohoto mechanismu do jádra aplikace se kvalita výsledného řešení zvýšila.

Další zlepšení kvality výsledného řešení přinesla úprava algoritmu, kdy po provedení operace křížení vstupovali potomci do cyklu, kde se prováděla mutace a za pomocí Metropolisova kritéria se rozhodovalo, zda v další iteraci cyklu se bude počítat s původním nebo zmutovaným chromozómem. Počet těchto cyklů měl možnost uživatel měnit. Čím byl počet vyšší, tím se zlepšovala i kvalita řešení. Vliv počtů cyklů na výslednou kvalitu řešení byl ale velice nepatrný. Navíc větší počet cyklů prodlužoval čas výpočtu. Z provedených měření se jako kompromis jevila hodnota počtu cyklů 10. Toto vylepšení výpočetního algoritmu pomohlo zlepšit kvalitu dosahovaných výsledků mnohem více, než zapracování elitářského mechanismu. Původně jsem myslel, že přidáním cyklu např. o deseti průchodech zpomalí výpočet 10x, ale není tomu tak. Je zde totiž změna v systému akceptování nových stavů. V původní verzi algoritmu (bez tohoto cyklu) se totiž při nízkých teplotách a při vzniku horších potomků, vzniklých zkřížením ne úplně kvalitních rodičů, mohlo stát, že Metropolisovo kritérium bylo příliš nízké a neakceptoval se ani jeden potomek nebo rodič. Proto se musela operace křížení a případné mutace opakovat a znovu se vyhodnocovalo, zda tyto nově vzniklé stavy akceptovat nebo ne. Zde se mohla situace klidně opakovat a čistě teoreticky zde mohl vzniknout nekonečný cyklus. S klesající teplotou tak trvalo delší dobu vygenerování nové populace a výpočet se postupně zpomaloval. To u verze algoritmu se zabudovaným cyklem mutací a následným rozhodováním o postupu do další iterace nehrozí. Vždy totiž postupuje do další iterace cyklu a nakonec i do další populace jeden chromozóm. Buď ten původní nebo zmutovaný. Takovýto algoritmus je v porovnání s původní verzí pomalejší při vyšších teplotách, naopak rychlejší při teplotách nižších. S výsledky, které tento algoritmus dodával, jsem už byl docela spokojený, ale bohužel mi zase tato varianta velice zkomplikovala návrh výpočetního jádra aplikace pomocí výpočetního vlákna. Výpočetní vlákno jsem totiž používal jako ochranu před „zamrzáváním“

grafického uživatelského rozhraní během výpočtu. Navíc mi výpočetní vlákno umožňovalo výpočet zastavovat, opětovně obnovovat a ukončovat v jeho průběhu.

Proto jsem toto řešení zavrhl a hledal k němu alternativu. Jako kvalitní řešení se jevila varianta, kdy se z populace vyberou dva náhodné chromozómy a zkříží se. Z těchto čtyř chromozómů se vybere pouze jeden nejlepší na základě délky reprezentované cesty. Zjistí se, zda se jedná o potomka nebo o rodiče a podle toho provedu mutaci. Pokud se jedná o rodiče, mutace se provede vždy. Pokud se jedná o potomka, mutace se provede jen s určitou pravděpodobností (opět záleží na uživatelově nastavení). Dále se takto získaný chromozóm testuje, zda reprezentuje lepší řešení, než oba rodičovské chromozómy, ze kterých vznikl. Pokud ano, tak postupuje do další generace. V opačném případě záleží na jeho ohodnocení Metropolisovým kritériem a aktuálně vygenerovaném náhodném čísle. Tento koncept algoritmu vykazoval zatím nejlepší výsledky z dosud testovaných koncepcí. Implementaci výpočetního vlákna to nijak nezkomplikovalo, ale opět zde vyvstala problematika postupného zpomalení generování nových populací. Zpomalování není ale tak výrazné jako u původního konceptu algoritmu, protože chromozóm má možnost postoupit do nové populace bez použití Metropolisova kritéria již při splnění podmínky, že jeho řešení je kvalitnější než řešení jeho rodičů. U původního konceptu se totiž požadovalo, aby nové řešení bylo absolutně nejlepší z dosud vygenerovaných řešení (tudíž zde byla mnohem menší pravděpodobnost).

Jako další urychlení konvergence výsledného řešení algoritmu směrem ke globálnímu optimu se nabízelo použití sofistikovanější mutace (mutace2). Podrobnosti o její implementaci byly popsány výše. Díky jejímu využití se rychlost konvergence zvýšila, bohužel ale pouze zpočátku výpočtu. Poté začala kvalita řešení téměř stagnovat.

Pro posuzování a porovnávání kvality získaných řešení jsem musel provádět měření na stejných datech a při stejně nastavených vstupních parametrech algoritmu. Testovací data jsem získal z internetové adresy *http://www.tsp.gatech.edu/vlsi/index.html*. Nejčastěji jsem testoval na testovacím souboru o 131 bodech. Velikost nejkratší cesty těchto bodů je 564, v testovacím souboru byly ale zadány v pořadí, které má délku 1384. Protože celý algoritmus je založen na generování náhodných čísel, dával pokaždé jiný výsledek při zpracování stejných dat při nastavených stejných vstupních parametrů. Proto jsem vždy provedl 6 měření a vypočítal aritmetický průměr, aby měly výsledky alespoň nějakou vypovídací hodnotu.

Při používání mutace2 se výsledné řešení zastavilo průměrně na hodnotě 943. Vstupní parametry algoritmu byly následující: Počáteční teplota = 10000, Koncová teplota = 1, Velikost populace = 50, Počet elitních jedinců, kteří automaticky postupují do další generace = 10, Pokles teploty = 5 %, Pravděpodobnost mutace = 20 %. Při používání obyčejné mutace (mutace1) a nastavení stejných parametrů jsem naměřil průměrně 902. Na základě těchto výsledků a faktu, že mutace2 dokázala rychleji vylepšovat kvalitu výsledku v počátku výpočtu, jsem se rozhodl použít mutaci2 pouze k účelu generování vstupní populace. Dále jsem už používal mutaci1. Po této úpravě jsem provedl opětovné měření a naměřený průměr byl už 860.

Závěrečná varianta výpočetního jádra aplikace, u které jsem nakonec zůstal, by se tedy dala popsat takto: Počáteční populace se vygeneruje pomocí aplikování mutace2 na vstupní chromozóm. Takto vzniklá populace vstupuje do Metropolisova algoritmu, kde se postupně snižuje teplota. Zde je zabudován tzv. elitářský mechanismus, který automaticky posílá určitý počet nejlepších řešení do další generace. Generování další populace se provádí aplikováním křížení na dva náhodně vybrané chromozómy. Vzniknou tak dva potomci. Z těchto dvou potomků a dvou rodičů vyberu nejlepšího na kterého aplikuji mutaci1 podle toho zda se jedná o potomka nebo rodiče. Rodiče zmutují vždy, potomci jen na základě určité pravděpodobnosti. Takto vzniklý chromozóm testuji, zda je lepším řešením než rodičovské chromozómy ze kterých vzešel. Pokud ano, postupuje do další generace. Pokud ne, tak do další generace postupuje s pravděpodobností odpovídající Metropolisově kritériu. Pokud nepostoupí, celá procedura se opakuje (náhodný výběr, křížení, …). Takto vygenerovaná populace bude sloužit v dalším průchodu Metropolisovým algoritmem jako vstupní populace. Výpočet se zastaví, když teplota poklesne na teplotu, kdy by dalším poklesem došlo k překročení hranice minimální teploty.

Nakonec jsem nalezl ještě jedno urychlení výpočetního jádra. A to takové, že nebylo nutné zasahovat do samotného algoritmu. Zjistil jsem totiž, že operace ohodnocení kvality řešení je hodně výpočetně náročná (záleží také na počtu genů v chromozómu) a někdy tuto operaci provádím se stejným chromozómem zbytečně vícekrát. Proto jsem upravil datovou strukturu, reprezentující tento chromozóm. Původně se jednalo pouze o posloupnost jednotlivých bodů. Nově jsem tam přidal desetinné číslo, které reprezentovalo kvalitu této posloupnosti. Inicializace probíhá vždy ihned po vytvoření daného chromozómu (operace křížení). Pokud se nad chromozómem provede operace mutace, tato hodnota se aktualizuje. Když se pro takovýto chromozóm počítá Metropolisovo kritérium nemusí se už na tento chromozóm volat ohodnocující funkce, ale stačí jen přečíst hodnotu této položky v jeho datové struktuře.

5.2 Vizualizace průběhu výpočtu

V průběhu výpočtu se provádí vizualizace dosud nejlepšího nalezeného řešení. Každý bod zadaný uživatelem se znázorňuje červenou kružnicí se středem v tomto bodě o poloměru 5 pixelů. Získané řešení, které reprezentuje posloupnost bodů se vykresluje za pomocí černých úseček spojujících tyto body v pořadí daném touto posloupností. Z posledního bodu posloupnosti se ještě vykreslí úsečka končící v prvním bodu posloupnosti. Vždy se totiž musí jednat o smyčku, protože se obchodní cestující musí vrátit do výchozího města.

Pod touto vizualizací se nachází graf, který znázorňuje kvalitu akceptovaných řešení v průběhu výpočtu. Osa y tedy znázorňuje kvalitu řešení ve formě délky cesty. Jednotkou je pixel. Ale záleží pouze na uživateli, co pro něj tento pixel na obrazovce znamená (cm, m, km, …), proto zde jednotky neuvádím. Osa x znázorňuje průběh výpočtu. Nejedná se ale o čas, protože výpočet se s klesající

teplotou zpomaluje (vinou nízké pravděpodobnosti přijetí horšího řešení). Proto zde jednotky také neuvádím. Postupně se zde zakreslují body znázorňující každé akceptované řešení. Další řešení se vykresluje o jeden pixel dále ve směru osy x. Každému akceptovanému řešení tedy odpovídá jeden bod. Aby byl graf lépe čitelný, tak tyto body jsou pospojovány v pořadí odpovídajícím ose x.

Z takto vykreslovaného grafu lze snadno vyčíst, jak se postupně zvyšuje kvalita přijímaného řešení (tedy graf klesá). Velice pěkně je na grafu vidět, jak s klesající teplotou klesá také rozptyl akceptovaných řešení.

Při implementaci tohoto grafu jsem se ale potýkal s problémem velmi vysoké paměťové náročnosti. Vše bylo závislé na nastavených teplotách, na velikosti populace a na poklesu teploty. Například při implicitním nastavení maximální teploty na 10000 K, minimální teploty na 100 K a velikosti populace na 50 chromozómů musel tento graf uchovávat 4600 bodů (vstupní populace o 50-ti chromozómech + 91 generací dalších populací po 50 chromozómech). To ještě není až tak mnoho, ale kdyby uživatel vstupní parametry trochu upravil, tak se klidně může jednat i o miliardy bodů. Tolik uchovávaných bodů už samozřejmě pojme velké množství paměti, což je velice nežádoucí. Navíc se tyto body stejně všechny do grafu nevejdou. Když počet bodů překročil šířku grafu v pixelech starší body se začaly ztrácet z obrazovky a nahrazovaly je postupně body novější. Graf tak byl neustále v pohybu. Pod grafem jsem zobrazil posuvník, aby měl uživatel možnost si prohlédnout starší průběhy grafu, které se už do něj nevešly.

Na tomto konceptu jsem z důvodů již zmíněných možných vysokých paměťových nároků provedl jednoduchou úpravu. Vždy před vložením nového bodu do grafu jsem zkontroloval, zda se tento bod do něj vleze. Pokud ne, umazal jsem jeho první bod a ostatní o jednu pozici posunul směrem k počátku. Graf se tedy také pohyboval směrem k počátku. Tím jsem si uvolnil místo pro nový bod, který jsem mohl následně do grafu bez obav vložit. Uchovával jsem tedy vždy jen maximálně takový počet bodů, který odpovídal šířce grafu v pixelech. Tato varianta byla paměťově velice úsporná, ale bohužel se vytratila možnost prohlížet si historii průběhu. Uchovávalo se tak jen posledních několik bodů a z nich šlo jen velmi špatně vyčíst jak se zvyšuje kvalita přijímaného řešení a jak se snižuje jejich rozptyl (hlavně při velkých populacích).

Proto jsem tento koncept také zamítl a začal pracovat na novém. Požadoval jsem od něj, aby byl co nejméně paměťově náročný a měl vysokou informační hodnotu. Jako kompromis jsem zvolil tuto variantu: Při vkládání bodu do grafu testuji, zda se tam ještě vejde. Pokud ne, provedu přepočet všech bodů v grafu již obsažených. Přepočtem transformuji vždy dva body na jeden, který je jejich aritmetickým průměrem. Tím se počet bodů vždy zmenší na polovinu. Od této chvíle bude pro vykreslení dalšího bodu grafu potřeba přijmout dvakrát více řešení než původně. Vykreslený bod bude aritmetickým průměrem jednotlivých ohodnocení těchto řešení. Pokud opět vyplníme celou šířku grafu, opět dojde na další přepočet bodů a pro vykreslení dalšího bodu budeme muset přijímat dvojnásobek řešení atd. To znamená, že graf se vždy smršťuje na polovinu své šířky a rychlost, kterou

v něm přibývají další body se postupně snižuje[6]. Tímto smršťováním se z grafu sice postupně vytrácejí detaily a celý průběh je postupně aproximován, ale kladnou stránkou této koncepce je její paměťová úspornost. Vždy je vidět celý graf a lze pozorovat rozptyl přijímaných řešení i trend postupného zkvalitňování přijímaných řešení.

Někdy (obvykle při nízkých teplotách) se vykreslovaný průběh jeví jako vodorovná čára, ale ve skutečnosti tomu tak být nemusí. Na vině je malá výška grafu a tím i nedostatečný počet pixelů ve vertikálním směru. Maximální hodnotu osy y se určuje vždy dynamicky podle dosud nejhoršího přijatého řešení. Pozice vykreslovaného bodu na ose y se tedy vypočítá následovně: *ohodnocená_kvalita_řešení * počet_pixelů_grafu_ve_vertikálním_směru / maximální_hodnota_osy_y* Výsledek ještě musíme zaokrouhlit na celé číslo. Z toho vyplývá, že například při maximální hodnotě v ose y 2000, počtem pixelů grafu ve vertikálním směru 200 budou řešení s ohodnocením 1795 - 1804 vykreslovány na stejné pozici 180-ti pixelů. Při provádění přepočtu bodů se provádí i oprava maximální hodnoty osy y. Ta se postupně mírně snižuje a kvalita rozlišení se tak zlepšuje.

5.3 Postup práce s aplikací

Nejprve je nutno zadat body na plochu k tomu určenou (bílá plocha v pravé horní části okna) a nastavit vstupní parametry algoritmu. Zadávání bodů se provádí kliknutím levým tlačítkem myši na vybrané místo plochy nebo načtením rozmístění bodů ze souboru. Získáme tak vstupní chromozóm pro výpočetní algoritmus. Pořadí bodů v chromozómu odpovídá pořadí, jakým byly na tuto plochu umístěny, nebo pořadí, jakým jsou jednotlivé body uloženy v souboru.

Nastavení vstupních parametrů se provádí pomocí ovládacích prvků nacházejících se v levém sloupci aplikačního okna. Aby mohl algoritmus pracovat správně, musí se dodržet určitá pravidla: Nastavená počáteční teplota musí být v rozsahu 2 - 1000000, koncová teplota musí být vždy menší než teplota počáteční, velikost populace musí být v rozsahu 10 - 1000 jedinců, počet uchovávaných elitních jedinců musí být menší než velikost populace, pokles teploty musí být v rozsahu 1 - 50 % a pravděpodobnost mutace musí být v rozsahu 1 -99 %. Pokud se u některého z parametrů jeho pravidlo nedodrží, nastaví se příslušný parametr na implicitní hodnotu na kterou byl nastaven po spuštění aplikace.

Po těchto krocích se už může přistoupit ke spuštění výpočtu. Výpočet probíhá ve zvláštním výpočetním vláknu a díky tomu ho můžeme kdykoliv pozastavit nebo ukončit. Při provedení této akce se ale vždy nejprve dokončí vygenerování nové populace a až poté se provede pozastavení nebo ukončení výpočetního vlákna. Aktuální hodnoty výpočtu se zobrazují ve stavovém řádku (aktuální teplota a hodnota dosud nejkvalitnějšího řešení). Průběžně se také vykresluje dosud nejkvalitnější řešení a vykresluje se také graf znázorňující kvalitu akceptovaných řešení v průběhu výpočtu.

6 Snižuje se také z důvodů malé pravděpodobnosti přijetí horšího řešení při nižších teplotách.

Celá aplikace má velice jednoduché intuitivní ovládání pomocí jednotlivých položek v menu, různých tlačítek nebo klávesových zkratek.

5.4 Skripty dodávané spolu s aplikací

Aplikace umožňuje načítat a ukládat rozmístění bodů z/do externího textového souboru s příponou *poc*. Takovýto soubor má pevně danou strukturu. Musí obsahovat pouze souřadnice jednotlivých bodů (celá nezáporná čísla), nic víc. Každý jeho řádek obsahuje souřadnice jednoho bodu a to nejprve hodnotu souřadnice x, za tou následuje nenulový počet mezer nebo tabulátorů a nakonec následuje hodnota souřadnice y.

Protože jsem průběžné testy prováděl na datech dostupných na internetu (*http://www.tsp.gatech.edu/vlsi/index.html*), připravil jsem si skript na jejich konverzi do mého formátu. Ten se nachází v adresáři skripty na CD. Z výše uvedené adresy si stačí stáhnout datový soubor a ručně z něj umazat první řádky až po řádek začínající číslem 1 (ten už zde necháme) a slovo EOF z konce souboru. Takto upravený soubor můžeme dále upravit skriptem s názvem *uprava.sh*. Skript využívá utility *sed* a výsledek vypisuje na standardní výstup, takže si ho musíme ještě přesměrovat do souboru (s příponou *poc*). Syntaxe je tedy následující:
sh ./uprava.sh název_vstupního_souboru > název_výstupního_souboru
Takto upravený soubor už můžeme načíst v aplikaci POC.

V adresáři skript se nachází ještě jeden soubor s názvem *usporadej.py*. Jedná se o skript v jazyce *python*, který slouží k uspořádání souřadnic bodů vstupního souboru v pořadí, které určuje další vstupní soubor. Takovýto soubor můžeme získat na adrese uvedené výše. Soubor udává pořadí jednotlivých bodů u optimální cesty. Musíme si ale opět nejdříve ručně mírně upravit. Je potřeba odmazat první „nečíselné“ řádky souboru a také poslední 2 řádky souboru, kde se nachází hodnota -1 a EOF. Nyní je počet řádků tohoto souboru totožný s počtem bodů, které uspořádává a můžeme ho použít ve skriptu *usporadej.py*. Výsledek se vypisuje na standardní výstup (opět v *poc* formátu), takže je zapotřebí použít přesměrování do souboru. Syntaxe je následující:
python ./usporadej.py souboru_se_seznamem_bodů soubor_s_pořadím > název_výstupního_souboru

Oba popsané skripty jsem na CD umístil, jen pro případ, kdy by bylo nutné rychle získat obsáhlejší soubory dat, než které jsou s aplikací dodávány.

5.5 Systémové požadavky

Tato aplikace byla vyvíjena a testována pod operačním systémem Linux. Distribuovaná je ve formě zdrojových kódu. Cílová platforma musí mít nainstalovanou grafickou nadstavbu a překladač jazyka C++. Testován byl překladač *g++* verze 4.1.2. Dále musí obsahovat vývojářské balíky s knihovnou

wxWidgets verze 2.8.0 nebo vyšší. Pokud budeme chtít použít pro překlad dodávaný skript makefile, musí být na cílové platformě také nainstalovaná utilita *make*.

Dodávaný makefile je speciálně upravený pro překlad aplikace na školním serveru merlin.fit.vutbr.cz, kde je nainstalováno hned několik verzí knihovny wxWidgets. Proto se v obsahu jeho pravidla pro překlad vyskytuje explicitní uvedení verze:

g++ Poc.cpp `wx-config-2.8 --libs` `wx-config-2.8 --cxxflags` -o Poc

Na jiných platformách, kde je nainstalovaná pouze jedna verze knihovny stačí použít pouze:

g++ Poc.cpp `wx-config --libs` `wx-config --cxxflags` -o Poc

Dále pokud budeme chtít využít dodávané skripty pro úpravu testovacích dat, nacházejících se na internetové adrese *http://www.tsp.gatech.edu/vlsi/index.html* musí být na cílové platformě nainstalovaná utilita *sed* a podpora skriptovacího jazyka *python*.

Pro snadné studování dokumentace zdrojového kódu vygenerované nástrojem *doxygen* z komentářů obsažených ve zdrojovém kódu bude zapotřebí ještě libovolný internetový prohlížeč.

6 Testování parametrů

U výsledné aplikace jsem testoval, jak nastavení různých parametrů výpočtu ovlivní kvalitu nalezeného řešení. Abych mohl jednotlivé výsledky mezi sebou porovnávat, testoval jsem na stejném vstupním souboru dat. Konkrétně na souboru *131.poc*, který se také nachází na CD. Abych alespoň částečně eliminoval rozptyl v dodávaných výsledcích, provedl jsem vždy deset měření a vypočítal aritmetický průměr získaných výsledků.

Zde je příklad čtyř testovacích konfigurací parametrů:

Konfigurace	A	B	C	D
Počáteční teplota	10000	100000	10000	1000
Koncová teplota	10	10	100	10
Velikost populace	50	50	50	50
Velikost elity	10	10	10	10
Pokles teploty	5	5	5	5
Pravděpodobnost mutace	20	20	20	20
Průměrná hodnota výsledku	980	984	1083	994

Když porovnáme výsledky konfigurace A s konfigurací B, zjistíme, že u konfigurace B zvýšení počáteční teploty nebylo ku prospěchu věci. Dokonce se zdá, že se kvalita řešení mírně zhoršila. Navíc při zvýšení počáteční teploty trvá výpočet delší dobu, protože se musí vygenerovat více populací.

Naopak při porovnání výsledků konfigurace C s konfigurací D vidíme, že počet generovaných populací bude u obou konfigurací stejný (u obou je počáteční teplota rovna stonásobku koncové teploty), ale konfigurace D dosahuje daleko lepších výsledků.

Podobnými porovnáváním výsledků různých konfigurací jsem došel k závěru, že dobré parametry algoritmu je téměř nemožné odhadnout a vždy je nutné udělat pár experimentů. Velice pomáhá pozorování grafu. Pokud výpočet nekončí téměř vodorovnou čárou je koncová teplota vysoká a je zapotřebí ji snížit, aby se dosáhlo lepších výsledků. Naopak pokud je v počátku výpočtu velmi velký rozptyl přijímaných řešení (např. když se přijímá 2x horší řešení než je řešení aktuální) je počáteční teplota zbytečně vysoká.

Dále bylo vysledováno, že vyšší pravděpodobnost mutace má spíše negativní vliv. Naopak zvětšení populace nebo elity a snižování rychlosti poklesu teploty má vliv pozitivní. Většina nastavení zlepšující kvalitu dodávaných výsledků má bohužel vedlejší efekt v podobě prodloužení doby výpočtu.

Z nastavování vstupních parametrů se tak stává velká alchymie a neexistuje nějaké univerzální doporučení, které by zohledňovalo veškeré aspekty.

7 Adresářová struktura CD

Obsah CD dodávaného jako příloha je následující:

- Adresář **bp** – technická zpráva bakalářské práce
  - soubor **techzprava.odt** – tato technická zpráva ve formátu odt
  - soubor **techzprava.pdf** – tato technická zpráva ve formátu pdf
- Adresář **data –** datové soubory s popisem rozmístění bodů
  - soubor 1**31.poc** – datový soubor obsahující popis rozmístění 131 bodů
- Adresář **skripty** – pomocné skripty pro úpravu vstupních dat
  - soubor **uprava.sh** – pomocný skript pro přípravu testovacích vstupních souborů
  - soubor **usporadej.py** – pomocný skript v jazyce python pro seřazování obsahu datových souborů
- Adresář **src** – zdrojové kódy aplikace
  - soubor **Poc.h** – hlavičkový soubor aplikace
  - soubor **Poc.cpp** – zdrojový kód aplikace
  - soubor **MersenneTwister.h** – hlavičkový soubor generátoru náhodných čísel
  - soubor **makefile** – *makefile* skript pro překlad aplikace na školním serveru merlin.fit.vutbr.cz
- Adresář **srcdok** – zde se nachází dokumentace zdrojového kódu vygenerovaná nástrojem *doxygen*

8 Závěr

S kvalitou dodávaných řešení algoritmem jsem spokojený. Zpočátku tomu ale tak nebylo a aplikace nebyla schopna vyřešit někdy i úlohu o deseti bodech. Až po postupném zapracování mých jednotlivých vylepšení (elitářský mechanismus, ukládání ohodnocení chromozómu přímo do datové struktury, atd.) se situace rapidně zlepšila. Protože má tato aplikace sloužit pouze k demonstraci práce genetického a Metropolisova algoritmu, myslím že bez problémů splňuje požadavky kladené na tento druh aplikací. Navíc má tato aplikace velkou výhodu v tom, že pokud se uživateli nebudu líbit vypočtený výsledek, může přenastavit vstupní parametry a spustit výpočet nad tímto výsledkem a ne nad původním zadáním.

Jako velkou slabinu vidím velkou citlivost algoritmu na nastavení vstupních parametrů. Proto bych další vývojový stupeň této aplikace viděl v zabudování mechanismů, provádějící automatickou změnu vstupních parametrů algoritmu během výpočtu podle aktuální situace. Uživatel by tak nemusel provádět neustále různé experimenty, aby dovedl odhadnout alespoň přibližné nastavení parametrů.

Během vývoje jsem se také mnohému přiučil. Nejprve studováním principů samotných algoritmů, následně při vývoji jsem se naučil používat knihovny wxWidgets pro vytváření grafické nadstavby programu.

Trochu zklamaný jsem ale z grafické knihovny wxWidgets. Nemile mě překvapilo, že už v demonstračním příkladu, dostupném na internetu, analyzátor paměťových úniků *valgrind* našel několik nesrovnalostí. Paměťové úniky byly už ovšem ve zkompilovaných knihovnách. Navíc mé internetové připojení nedovolilo moc podrobné testování na školním serveru merlin.fit.vutbr.cz. Vinou přenosu všech grafických informací se běh aplikace výrazně zpomalil. Proto jsem většinu času testoval pouze na svém PC.

Literatura

[1] Kvasnička, V., Pospíchal, J., Tiňo, P.: *Evolučné algoritmy*. Bratislava, STU 2000.

[2] Luner, P.: Jemný úvod do genetických algoritmů. Dokument dostupný na URL *http://cgg.ms.mff.cuni.cz/~pepca/prg022/luner.html* (červenec 2007)

[3] Otevřená encyklopedie Wikipedie: *http://cs.wikipedia.org* (červenec 2007)

[4] Vašíček, Z.: Simulované žíhání. Dokument dostupný na URL *http://www.stud.fit.vutbr.cz/~xvasic11/projects/msi.pdf*

[5] Teda, J.: Genetické algoritmy a jejich aplikace v praxi. Dokument dostupný na URL *http://programujte.com/view.php?cisloclanku=2005072601* (červenec 2007)

Seznam příloh

Příloha 1. CD obsahující elektronickou verzi této technické zprávy, zdrojové kódy aplikace, dokumentaci zdrojového kódu a pomocné soubory (viz. kapitola 7)