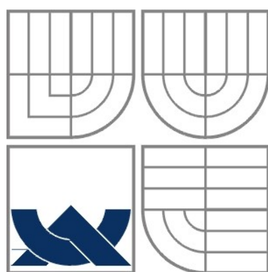
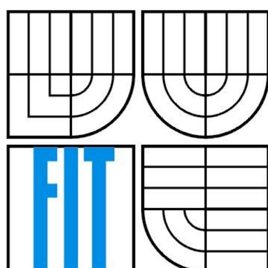


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PROBLÉM OBCHODNÍHO CESTUJÍCÍHO - paralelní řešení pomocí vláken (SMP)

BAKALÁŘSKÁ PRÁCE

AUTOR PRÁCE **Martin Weigner**

VEDOUCÍ PRÁCE **Ing. Tomáš Kašpárek**

BRNO 2009

SEM PATŘÍ ZADÁNÍ

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

Abstrakt

Práce se zabývá řešením problému obchodního cestujícího. Problém je řešen nejprve sériovým přístupem na čtyřech algoritmech, aby byly posléze vybrány dva, které jsou převedeny do paralelního provedení. V závěru jsou shrnuty poznatky o rozdílných parametrech obou přístupů. Práce rovněž čtenáře krátce seznamuje s problematikou programování paralelních aplikací pomocí vláken.

Klíčová slova

Problém obchodního cestujícího, genetický algoritmus, metoda zakázaného prohledávání, metoda simulovaného žíhání, hladový algoritmus, vlákna (SMP).

Abstract

This thesis is focused on solving the problem of traveling salesman. At first, the problem is solved by serial access at four algorithms. There are two of them choosed and transferred to parallel access. In the end there are summarised observations about different parameters of both access. This thesis also introduces questions of programming parallel aplications with threads to the reader.

Keywords

Traveling salesman problem (TSP), genetic algorithm, tabu search, simulated annealing, greedy search, threads (SMP).

Citace

Martin Weigner: PROBLÉM OBCHODNÍHO CESTUJÍCÍHO - paralelní řešení pomocí vláken (SMP), bakalářská práce, Brno, FIT VUT v Brně, 2009

PROBLÉM OBCHODNÍHO CESTUJÍCÍHO - paralelní řešení pomocí vláken (SMP)

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Tomáše Kašpárka.

Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………

Martin Weigner

19. května 2009

Poděkování

Děkuji vedoucímu své bakalářské práce panu Ing. Tomáši Kašpárkovi za cenné rady, podnětné návrhy i za ochotu a trpělivost, se kterou se mi věnoval ve všech fázích přípravy této práce.

© Martin Weigner, 2009.

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

Obsah....5
1. Úvod....7
2. Matematický pohled na problém....8
 2.1 Možné modifikace zadaní....9
 2.2 Využití TSP....9
  2.2.1 V logistice dodávek zboží....9
  2.2.2 Ve výrobě....10
 2.3 Algoritmy použitelné k řešení problému....10
  2.3.1 Genetický algoritmus 10
  2.3.2 Metoda lokálního hledání 11
  2.3.3 Iterativní metoda 11
  2.3.4 Hladový algoritmus 11
  2.3.5 Horolezecký algoritmus 11
  2.3.6 Metoda zakázaného prohledávání 11
  2.3.7 Metoda simulovaného žíhání 12
  2.3.8 Gradientní algoritmus 12
 2.4 Složitosti jednotlivých algoritmů....13
  2.4.1 Postup testování....13
  2.4.2 Tabulka porovnávající výše zmíněné algoritmy....14
3. Implementace sériových řešení....18
 3.1 Výběr algoritmů k sériovému zpracování....18
 3.2 Sériová implementace....18
  3.2.1 Metoda simulovaného žíhání....18
  3.2.2 Metoda zakázaného prohledávání....19
  3.2.3 Hladový algoritmus....20
  3.2.4 Genetický algoritmus....21
  3.2.5 Testování (generator.c)....22
 3.3 Výsledky sériového řešení....23
4. Paralelní program pomocí vláken....27
 4.1 Norma POSIX 1003.1....27
 4.2 Proces nebo vlákno....27
 4.3 Implementace vlákna....27
  4.3.1 Vytvoření vlákna....28
  4.3.2 Data vláken....29
  4.3.3 Synchronizace vláken....29
  4.3.4 Rušení vlákna....32
  4.3.5 Čisticí funkce....32
  4.3.6 Užití vláken....32
5. Implementace paralelních řešení....34
 5.1 Převedení sériových programů na paralelní....34
 5.2 Aplikace změn do algoritmů....34
  5.2.1 Genetický algoritmus....35
  5.2.2 Metoda zakázaného prohledávání....35
  5.2.3 Simulované žíhání a hladový algoritmus....35

5.4 Paralelní řešení na vícejádrovém procesoru....................39
5.5 Modifikace úlohy....................42
5.5.1 Nalezení řešení....................42
5.5.2 Paměťová náročnost....................42
6. Závěr....................43
Literatura....................44
Přílohy....................45

1. Úvod

Problém obchodního cestujícího je definován následovně: Na mapě je zvoleno N měst a jejich vzájemná vzdálenost pro každou z dvojic. Problémem obchodního cestujícího je určit takové pořadí návštěv jednotlivých měst, aby každé město bylo navštíveno právě jednou, výsledná délka (cena, jízdné, ...) byla nejkratší a cestující se vrátil zpět do města, kde cestu začal.

V první kapitole je rozepsán matematický pohled na tento problém. Kapitola pokračuje stručným popisem modifikací zadání této úlohy a je zde zmínka o aplikaci tohoto problému v praxi. Dále jsou tu popsány různé algoritmy, kterými lze daný problém řešit. Kapitola končí tabulkou porovnávající popsané algoritmy z pohledu časové a paměťové složitosti.

Ve druhé kapitole je popsán postup implementace genetického a hladového algoritmu a metod zakázaného prohledávání a simulovaného žíhání. Vše zatím v sériovém provedení. Kapitola je ukončena tabulkou porovnávající dobu trvání, paměťovou složitost a výsledná řešení těchto implementovaných aplikací.

Třetí kapitola je opět teoretická. Velmi stručně je v ní popsáno, co je to norma POSIX. Hlavně je kapitola zaměřena na popis implementace vláken, od jejich vytvoření, přes synchronizaci až k ukončení činnosti vlákna. Kapitola je ukončena popisem možností, jak lze vlákna využít ke zefektivnění aplikace.

Čtvrtá kapitola se věnuje praktické aplikaci vláken na genetický algoritmus a metodu zakázaného prohledávání. Kromě popisu několika možností, jak lze problém převodu na paralelní provedení řešit, je zde popsána i metoda, která byla nakonec aplikována. Tabulka na konci kapitoly porovnává výsledky sériových a paralelních řešení. Navíc jsou zde uvedeny výsledky vrácené osmijádrovým procesorem.

Poslední kapitola je věnována souhrnu získaných poznatků a jiným možnostem přístupu k danému problému.

2. Matematický pohled na problém

Z matematického hlediska jsou vstupní data znázorněna ohodnoceným grafem, kde uzly jsou jednotlivá města a ohodnocené hrany jsou vzdálenosti mezi těmito městy. Úkolem obchodního cestujícího je efektivně nalézt co nejkratší hamiltonovskou kružnici.

V teorii grafů je hamiltonovská kružnice (také hamiltonovský cyklus) grafu taková kružnice v tomto grafu, která projde právě jednou všemi jeho vrcholy. Graf, který obsahuje hamiltonovskou kružnici, se nazývá Hamiltonův graf.

Každý graf nemusí mít nutně hamiltonovskou kružnici. Nutnými (avšak nikoli postačujícími) podmínkami je, že graf musí být souvislý a každý vrchol musí mít stupeň nejméně rovný dvěma (ke každému vrcholu musí vést alespoň 2 hrany).[1] Po jedné z nich do „města“ obchodní cestující přijde a po druhé odejde.

Jedná se o problém z kategorie NP-úplných. *To jsou takové nedeterministicky polynomiální problémy, na které jsou polynomiálně redukovatelné všechny ostatní problémy z NP. To znamená, že třídu NP-úplných úloh tvoří v jistém smyslu ty nejtěžší úlohy z NP.[2]*

Že jde o nedeterministicky polynomiální problém, je patrné z toho, že nedeterministický počítač, umožňující v každém kroku rozvětvit výpočet na libovolný počet větví, by mohl začít v některém „městě“, rozdělit propočet délky trasy na tolik větví, kolik z města vede silnic, a v každém z cílových měst postupovat stejně – s výjimkou tras vedoucích do již navštívených měst. Tak by prohledal všechny možné trasy v n výpočetních krocích, pokud počet měst činí n, a rozvětvil by se maximálně do (n − 1)! větví.[3]

Třída složitosti P obsahuje všechny úlohy, jejichž řešení lze nalézt deterministickým Turingovým strojem v polynomiálním čase. Pro třídu NP platí totéž s tím rozdílem, že se jedná o nedeterministický Turingův stroj. Jsou to ty problémy, jejichž řešení lze ověřit v polynomiálním čase, ovšem nevíme, zda je lze také v polynomiálním čase nalézt.[4]

K problému NP-úplných úloh cituji server www.scienceworld.cz:

Keith Devlin (Anglický matematik a spisovatel pracující na univerzitě ve Stanfordu) *řadí otázku NP úplných úloh a jejich vztah k P úlohám mezi jeden ze sedmi největších problémů, které stojí před matematikou nového tisíciletí. Podle něj jde však současně o jediný z těchto problémů, na jehož řešení má šanci dosáhnout i laik...*

Co se týče problému NP, Devlin je přesvědčen, že se podaří prokázat, že NP je různé od P. Probírat se třeba známými procedurami pro řešení úlohy obchodního cestujícího ovšem podle Devlina nejspíš nikam nepovede. Konec konců nehledáme konkrétní řešící algoritmus fungující v polynomiálním čase (pokud se tedy shodneme na tom, že takový ani neexistuje). Probírání se řadou algoritmů ale těžko může přivést k důkazu, že žádný polynomiální algoritmus nemůže existovat obecně.

Za nejnadějnější pokládá Devlin postup, kdy nejprve budeme konstruovat nějaký nový problém, který ze své podstaty není řešitelný v polynomiálním čase, a poté dokážeme, že je ekvivalentní některému z „velkých" NP úplných problémů. Uznává však, že sám při svých pokusech v tomto směru nijak nepokročil, vlastně ani o píď. Nepřišel totiž na to, jak přímo do podstaty problému zabudovat jeho neřešitelnost v polynomiálně rostoucím čase.

Devlin je ovšem přesvědčen, že problém NP versus P má největší šanci, že jej vyřeší nějaký neznámý amatér. Mj. i proto, že je to jeden z mála současných matematických problémů, kde amatér může vůbec porozumět formulaci úlohy :-). Možná, na rozdíl od ostatních matematických problémů, postačí jeden dobrý nápad...

(To by konec konců stačilo i ve velmi neočekávaném případě, že NP=P; také zde by šlo o nalezení jediného algoritmu.)[5]

2.1 Možné modifikace zadaní

Základní verze problému je taková, že obchodní cestující může navštívit každé město právě jednou, tedy nemůže žádné město vynechat a zároveň se nesmí vrátit do žádného s výjimkou startovacího města, ve kterém musí skončit.

Problém může být upraven tak, aby obchodní cestující procházel opravdu nejkratší možnou cestou všechna města. V takovém případě musí každé město navštívit alespoň jednou. Tzn. může se vrátit do města, ve kterém již byl, pokud to je pro jeho cestu výhodnější.

- *Metrický problém je takový, pokud pro všechna města (i, j a k) platí trojúhelníková nerovnost $C_{ik} <= C_{ij} + C_{jk}$.[6]* Kde C_{ik} je vzdálenost mezi městy i a k.
- U symetrického problému platí, že $C_{ij} = C_{ji}$. Je to situace, kdy potřebujeme zajistit, aby obchodní cestující prošel města po co nejkratší dráze, tzn. hodnoty hran jsou vzdálenost.
- *Euklidovský problém je takový, kde odpovídají hodnoty C_{ij} vzdálenostem bodů v rovině. Euklidovský problém je pochopitelně, jak symetrický, tak metrický.[6]*
- Asymetrický problém se liší v tom, že C_{ij} se nerovná C_{ji}. Lze si to představit jako cestu kopcovitým terénem, kdy cesta z kopce trvá kratší dobu než cesta do kopce. Hodnoty hran jsou v tomto případě doby trvání cesty z jednoho města do druhého.

2.2 Využití TSP

2.2.1 V logistice dodávek zboží

Jak už název problému napovídá, jedno z možných využití je právě pro obchodní cestující, nebo spíše pro dodavatelské firmy. Dobře naplánovaný rozvoz může takové firmě ušetřit nezanedbatelné množství peněz. Samozřejmě dodavatelským firmám v ideálním případě stále přibývají další a další zakázky, což plánování komplikuje.

Na univerzitě v indickém Madrásu se Chandra Sunil Kumar zaměřil právě na tuto podobu problému. Předpokládal, že obchodní cestující ráno vyrazí, řekněme, s tak nějak optimalizovanou trasou, ovšem v průběhu dne budou přibývat další požadavky. Jak nejlépe zareagovat na měnící se situaci? Řekněme, že ve chvíli, kdy přijde první dodatečná žádost, se úloha začne řešit znovu. Vstupem jsou dosud nesplněné úkoly plus úkol nový, vzdálenosti se počítají od místa, kde se náš cesťák zrovna nachází. Takže co teď? Má se prostě zastavit, dokud neobdrží další optimalizovanou

trasu? Nebo má ještě nějaký čas sledovat trasu původní a dodatečné požadavky se budou zpracovávat nějak dávkově? U velké firmy navíc nepůjde ani tak o plán trasy pro konkrétního cestujícího, ale o porovnání všech možných tras, na jejichž základě bude zakázka přidělena konkrétnímu člověku.

Nepřekvapí, že když budete muset plán cesty neustále modifikovat, nakonec urazíte větší vzdálenost, než kdyby vše bylo zadáno předem – takže předešlá úloha vlastně jinými slovy znamená „komu zakázku přidělit, aby mu cestu nejméně nabourala". Výsledek z Madrásu ale říká, že i tak se vyplatí raději provádět přepočítání úlohy než prostě splnit původní plán a aktuální požadavky řešit až dodatečně.[7]

2.2.2 Ve výrobě

Je-li například potřeba do nějaké desky/plochy vytvořit velké množství otvorů na přesně daná místa, tak je výhodné zjistit ideální pořadí vytváření děr, aby se děrovací přístroj nepřesunoval zbytečně z jednoho konce desky na druhý.

2.3 Algoritmy použitelné k řešení problému

Použití deterministického algoritmu je v tomto případě nevhodné. Nebo, lépe řečeno, deterministickým algoritmem dosáhneme přesného řešení (pokud existuje). To by se nám hodilo, jenže doba, kterou by takový algoritmus potřeboval, je exponenciálně závislá na množství bodů a hran. Takže takový algoritmus je možné použít jen na velmi malých grafech. Pro obecný graf se takový algoritmus rozhodně nehodí, proto je potřeba hledat jiné.

Níže jsou popsány různé druhy algoritmů, které hledají řešení mnohem rychleji než deterministické, ale nezaručují ideální řešení. Nicméně řešení, která naleznou, jsou ideálnímu řešení tak blízká, že jsou algoritmy pro náš problém použitelné. Jsou to příklady heuristických algoritmů.

2.3.1 Genetický algoritmus

Princip práce genetického algoritmu je postupná tvorba generací různých řešení daného problému. Při řešení se uchovává tzv. populace, jejíž každý jedinec představuje jedno řešení daného problému. Jak populace probíhá evolucí, řešení se zlepšují. Tradičně je řešení reprezentováno binárními čísly, řetězci nul a jedniček, nicméně používají se i jiné reprezentace (strom, pole, matice, ...). Typicky je na začátku simulace (v první generaci) populace složena z naprosto náhodných členů. V přechodu do nové generace je pro každého jedince spočtena tzv. fitness funkce, která vyjadřuje kvalitu řešení reprezentovaného tímto jedincem. Podle této kvality jsou stochasticky vybráni jedinci, kteří jsou modifikováni (pomocí mutací a křížení), čímž vznikne nová populace. Tento postup se iterativně opakuje, čímž se kvalita řešení v populaci postupně vylepšuje. Algoritmus se obvykle zastaví při dosažení postačující kvality řešení, případně po předem dané době.[8] Tento algoritmus je vhodné využít ve složitém, případně i měnícím se prostředí, kde programátor není schopen dopředu nadefinovat všechny vzniklé případy a správné reakce na ně.[9] S drobnými úpravami je vhodné jej využít pro náš problém v základní modifikaci.

Časová složitost tohoto algoritmu je $O(X^2)$. Paměťová složitost je O(2N).

2.3.2 Metoda lokálního hledání

Je nejjednodušší a také nejméně efektivní heuristickou metodou. Je u ní velmi pravděpodobné, že skončí v lokálním optimu, které nemusí být nutně globálním optimem. Tohoto efektu je možné se částečně vyvarovat spuštěním metody s různými počátečními řešeními a za výsledek brát nejlepší nalezený.[12]

Časová složitost tohoto algoritmu je $O(X^2)$. Paměťová složitost je O(N).

2.3.3 Iterativní metoda

Tato metoda využívá postupného hledání řešení ve stále se zužující oblasti řešení (postupně se z dobrého řešení dopracovává k ještě lepšímu řešení).[10] Je možné ji použít tam, kde víme alespoň přibližné řešení, tedy vhodný startovní bod.

Časová složitost tohoto algoritmu je $O(N^2)$. Paměťová složitost je O(N).

2.3.4 Hladový algoritmus

Tato metoda *je jedním z možných způsobů řešení optimalizačních úloh v matematice a informatice. V každém svém kroku vybírá lokální minimum, přičemž existuje šance, že takto nalezne minimum globální. Hladový algoritmus se uplatní v případě, kdy je třeba z množiny určitých objektů vybrat takovou podmnožinu, která splňuje jistou předem danou vlastnost a navíc má minimální (případně maximální) ohodnocení. Ohodnocení je obvykle reálné číslo w, přiřazené každému objektu dané množiny.[11]* Hladový algoritmus je vhodný pro původní a také pro symetrickou modifikaci.

Časová složitost jeho konkrétní implementace v Kruskalově algoritmu je O(m logN), kde m je počet hran a N je počet uzlů. Paměťová složitost je O(N).

2.3.5 Horolezecký algoritmus

Tento algoritmus *patří do skupiny algoritmů, ve kterých jsou dovoleny i kroky směřující ke zhoršení aktuálního řešení. Podobně jako v metodě lokálního hledání v každém kroku vybíráme nejlepší řešení ze sousedních řešení. Rozdíl je v tom, že podmínkou ukončení algoritmu není nenalezení zlepšujícího řešení mezi sousedními řešeními, ale provedení určitého počtu iterací.[12]* Je možné jej použít tam, kde nevíme, jak přesně problém vypadá (např. neznáme průběh funkce). Lze jej využít v původní modifikaci našeho problému.

Časová složitost algoritmu je O(m logN). Paměťová složitost je O(N).

2.3.6 Metoda zakázaného prohledávání

Tato metoda dokáže najít přijatelné optimum za velmi krátký čas, pokud je přijatelných minim ve sledované funkci více. Při prohledávání funkcí s mnoha minimy, ale jen málo přijatelnými minimy, má tato metoda trochu problémy. Její základní myšlenka totiž spočívá v logickém vylepšení horolezeckého algoritmu (hill climbing algorithm). Jedná se tedy o variantu gradientní metody „bez

gradientu", protože se směr nejprudšího spádu určí prohledáváním okolí. Tento algoritmus proto trpí základní nectností gradientních metod, tj. často skončí v lokálním extrému a nedosáhne globálního minima. Vychází se zde z náhodně navrženého řešení. Pro aktuální navržené řešení se generuje konečným souborem transformací určité okolí a vybere se z tohoto okolí nejlepší minimum. Získané lokální řešení se použije jako „střed" nového okolí, ve kterém se lokální minimalizace opakuje. Tento proces projde předepsaným počtem opakování. V průběhu celé historie algoritmu se zaznamenává nejlepší řešení, které slouží jako výsledné minimum. Aby nedošlo k zacyklení, spustí se algoritmus několikrát s náhodně vygenerovanými počátečními stavy a poté se vybere nejlepší výsledek. V průběhu algoritmu si dočasně zaznamenává tzv. zakázané cesty, do kterých se nebude vracet, čímž se stává odolnějším vůči zacyklení se.[9] Tento algoritmus se výborně osvědčil při řešení problému obchodního cestujícího v původní modifikaci.

Časová složitost tohoto algoritmu je O(N). Paměťová složitost je O(N).

2.3.7 Metoda simulovaného žíhání

Tato metoda *je variantou horolezeckého algoritmu, v němž heuristické kroky směřující k horšímu řešení jsou řízeny určitou pravděpodobností. Přístup simulovaného žíhání je založen na simulování fyzikálních procesů probíhajících při odstraňování defektů krystalické mřížky. Krystal se zahřeje na určitou (vysokou) teplotu a potom se pomalu ochlazuje (žíhá). Defekty krystalické mřížky mají při vysoké teplotě vysokou pravděpodobnost zániku. Pomalé ochlazování systému zabezpečí, že pravděpodobnost vzniku nových defektů klesá. Při žíhání se soustava snaží dostat do takového stavu, ve kterém je její energie minimální – tj. krystal bez defektů. Existuje určitá analogie tohoto přírodního procesu s procesem řešení optimalizačních problémů. Simulované žíhání je schopno nalézt optimální (u složitějších úloh většinou však jen suboptimální) řešení. Ukazuje se, že simulované žíhání poskytuje velmi efektivní algoritmus k řešení kombinatoriálních úloh, které jsou NP-úplné, přičemž získaná řešení jsou buď totožná nebo velmi blízká optimálním řešením.[12]* Algoritmus by se měl uplatnit i u asymetrické modifikace našeho problému.

Algoritmus má časovou složitost O(2N). Paměťová složitost je O(N).

2.3.8 Gradientní algoritmus

Je to nejjednodušší verze informovaného algoritmu. Funguje tak, že expanduje vždy ten nejperspektivnější uzel a nejlepší z následovníků je vybrán k další expanzi. Algoritmus uchovává v paměti vždy jen aktivní uzel, ostatní jsou zapomenuty. To má jisté nevýhody; může skončit i v lokálním extrému (ne jen v globálním) a může se zacyklit. Navíc neumožňuje dočasně zhoršit hodnotu stavu proto, aby se dosáhlo v dalších krocích zlepšení.[14]

Algoritmus má časovou složitost O(N). Paměťová složitost je O(N).

2.4 Složitosti jednotlivých algoritmů

Složitost je způsob klasifikace počítačových algoritmů. Zjišťuje jakým způsobem se bude chování algoritmu měnit v závislosti na změně velikosti (počtu) vstupních dat. Zapisuje se pomocí „velké O notace“ jako O(f(N)) (např. O(N)). Obvykle se používá asymptotická časová a prostorová složitost.[17] Prostorová složitost určuje, kolik paměti algoritmus spotřebovává.

Některé asymptotické složitosti mají i své triviální pojmenování (jsou seřazeny od nejrychlejších k nejpomalejším):

- *O(1) – konstantní*
- *O(log N) – logaritmická*
- *O(N) – lineární*
- *O(N log N) – lineárnělogaritmická*
- *$O(N^2)$ – kvadratická*
- *$O(N^3)$ – kubická*
- *obecně $O(N^X)$ – polynomiální*
- *obecně $O(X^N)$ – exponenciální*
- *O(N!) – faktoriálová [17]*

2.4.1 Postup testování

Výše zmíněných osm algoritmů jsem v první fázi práce neimplementoval. K vyhodnocení jejich základních vlastností jsem převzal kódy a aplikace ze zdrojů na internetu([26], [27], [28], [29], [30]). Většinou se jednalo o java applety. Všechny jsou na přiloženém médiu označeny jako převzaté algoritmy. Z těchto aplikací se mi nejlépe pracovalo s genetickým algoritmem a s metodou zakázaného prohledávání. U těchto dvou bylo ovládání intuitivní a ani nastavování parametrů nebylo příliš komplikované.

Metoda zakázaného prohledávání se ovládala naprosto nejsnadněji, stačilo nastavit počet uzlů. Jejich vytvoření zajistila aplikace sama. U genetického algoritmu bylo potřeba jednotlivé uzly zadat do prostoru, což zabralo chvilku času, ale nijak nepřiměřeně mnoho. Hrany mezi jednotlivými uzly už si aplikace dodala sama, což další applety nezvládaly. Genetický algoritmus si dokonce sám měřil čas, a to i po nalezení ideální cesty. Bylo tedy potřeba odhadnout, do jaké míry bude algoritmus cestu ještě upravovat, a jestli už náhodou není v nejlepším řešení. Tento odhad jsem prováděl tak, že jsem sledoval, kolik hran se mění, pokud se měnila už jen jedna a ještě se pravidelně vracela do původní pozice, považoval jsem výpočet za ukončený.

Zmínil jsem se o dalších appletech, ale to nebylo úplně přesné. Ostatní algoritmy, kromě simulovaného žíhání, prováděl jeden applet. Ten sice dokázal provádět výpočty pomocí několika různých algoritmů, ale jeho ovládání bylo velmi náročné. Bylo potřeba naklikat nejen jednotlivé uzly, ale i všechny hrany mezi nimi. Tato aplikace také nedokáže provádět výpočet problému obchodního cestujícího, ale pouze hledá nejkratší cestu mezi dvěma body. Mým cílem bylo zjistit, jak se budou algoritmy chovat při vzrůstajícím počtu uzlů obecně v jakémkoli problému, proto tento nedostatek nevadil. Vždy jsem stanovil startovací a cílový bod a mezi nimi příslušný počet bodů, které jsem náhodně propojil hranami.

Posledním algoritmem je simulované žíhání. Pro ten jsem si pořídil zdrojový kód v jazyce C++. Není to jediný algoritmus, ke kterému jsem získal zdrojový kód, ale jediný, který se podařilo bez problémů přeložit a spustit. Poté, co jsem prostudoval jeho kód, pustil jsem se do testování na různých datech. Vstupní data se tomuto programu zadávala v podobě matice vzdáleností. Jde tedy

o paměťově náročnou aplikaci, zvlášť při větším objemu dat. Tento algoritmus je tedy jediný, jehož konkrétní implementaci jsem měl příležitost pochopit ještě před výběrem, které algoritmy budu dále zpracovávat. Princip jeho funkce vysvětlím níže, už proto, že z tohoto algoritmu jsem vycházel při tvorbě dalších.

2.4.2 Tabulka porovnávající výše zmíněné algoritmy

Tabulka porovnává rychlost jednotlivých algoritmů a paměť jimi zabranou při vykonávání dané úlohy. Například genetický algoritmus je schopen v prvních několika sekundách najít cestu kvalitní, a tu už těžko zlepšuje, což je důvod jeho vysokých časů hledání. Na druhou stranu, metoda zakázaného prohledávání má velký potenciál k procházení mnoha bodů. Časově také vychází metoda zakázaného prohledávání jako lepší než genetický algoritmus, kterému doba hledání trasy mezi 800 uzly trvá více než půl hodiny. Také metoda simulovaného žíhání dokáže zpracovat počet uzlů přesahující hodnotu 1000.

Celkově vzato lze říci, že pro nízké počty uzlů jsou vyhovující všechny zmíněné algoritmy. Pro větší počet uzlů už to tak jasné není. Některé algoritmy dokáží nalézt řešení poměrně rychle, jiným to trvá déle. Je ovšem potřeba si uvědomit, že to, co je v tabulce označené jako zbytečně dlouhé řešení, může stačit. Máme-li horolezecký algoritmus vypracovaný a nechceme utrácet za programátora, který by nám dodal rychlejší algoritmus, a zároveň nám stačí zjistit ideální cestu jednou za rok, či za dva, jistě nevadí strávit týden nad výpočtem takovéto trasy. Pokud však potřebujeme propočítávat ideální cestu stále, budeme potřebovat nějaký rychlejší algoritmus.

Nebo stroj. Testující počítač není ani nejpomalejší, ale ani nejrychlejší. Na stroji s mnohem silnějším procesorem a také mnohonásobnou pamětí RAM mohou algoritmy trvat i desetkrát kratší dobu. To se sice nijak výrazně neprojeví na malých počtech uzlů, ale na větších už ano.

Pro mou další práci si myslím, že nebude mít smysl pokoušet se pracovat na algoritmech pro počty uzlů menší než 100. Na druhou stranu, budu-li se chtít dostat k nějakým výsledkům v dosažitelné době, nebude mít smysl, až na výjimky, pracovat s počtem uzlů větším než 1000.

Testy probíhaly stroji Intel centino T5550 2x1,83GHz, 2GB RAM, Windows XP SP 2

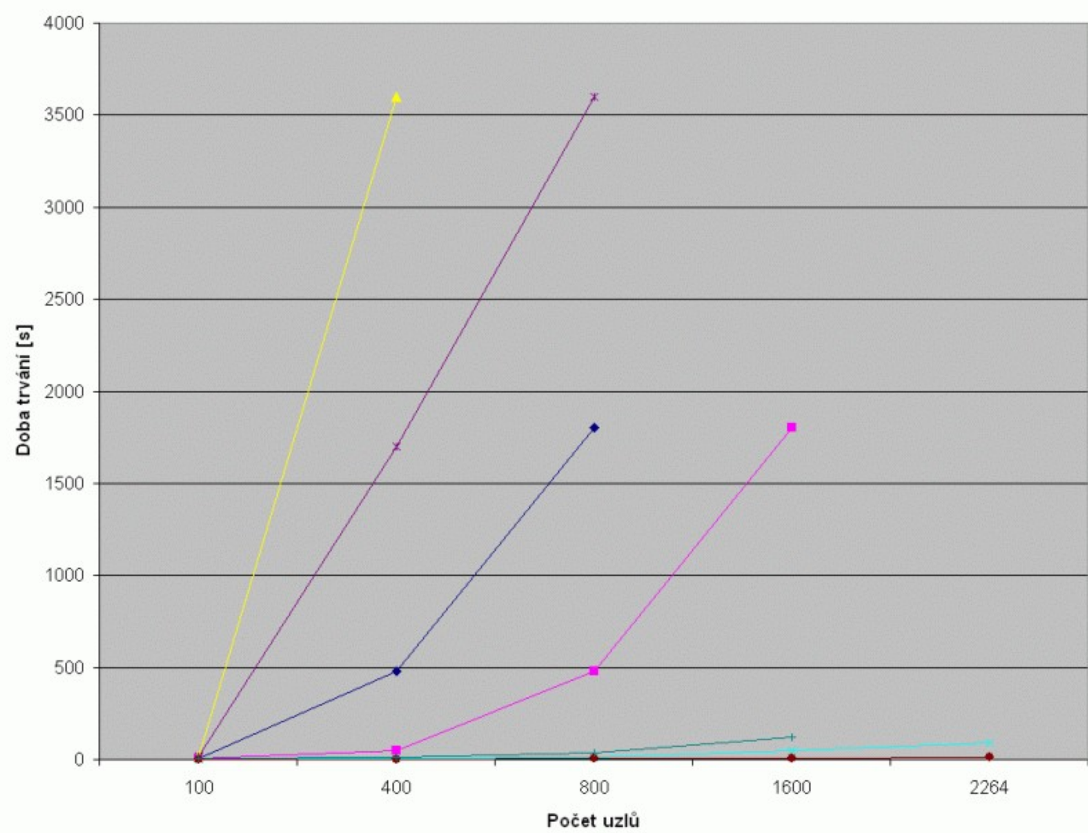
Vysvětlivky:

- N je počet uzlů grafů
- m je počet hran grafu

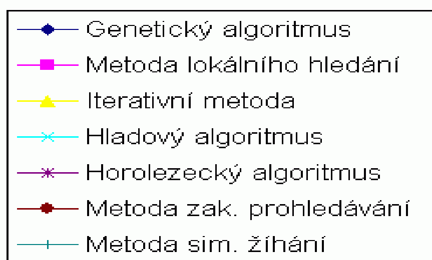
	Genetický algoritmus (GA)	*Metoda lokálního hledání (LS)*	*Iterativní metoda (IDS)*	*Hladový algoritmus (GR)*	*Horolezecký algoritmus (HCA)*	*Metoda zak. prohledávání (TS)*	*Metoda sim. žíhání (SA)*
100: čas	4 s	8 s	20 s	1 s	19 s	1,5 s	6 s
400: čas	8 min	47 s	> 60 min	4 s	26 min	2 s	15 s
800: čas	> 30 min	8 min	------------	14 s	> 60 min	4 s	35 s
1600: čas	------------	> 30 min	------------	48 s	--------------	8 s	2 min
2264: čas	------------	-----------	------------	1,5 min	--------------	12 s	----------
3200: čas	------------	-----------	------------	3 min	--------------	16 s	----------
100: pam.	1 MB	37 MB	44 MB	11 MB	30 MB	2,5 MB	1 MB
400: pam.	4,2 MB	37,5 MB	80 MB	11,5 MB	48 MB	13 MB	1,5 MB
800: pam.	14,5 MB	37,5 MB	120 MB	12 MB	65 MB	22 MB	3,5 MB
1600: pam.	30 MB	38 MB	163 MB	12,5 MB	80 MB	46 MB	10,5 MB
2264: pam.	-----------	-----------	-----------	12,8 MB	-------------	78 MB	----------
3200: pam.	------------	-----------	-----------	13 MB	--------------	112 MB	----------
Časová složitost	$O(N^2)$	$O(N^2)$	$O(N^2)$	O(m logN)	O(m logN)	O(N)	O(2N)
Paměťová složitost	$O(N+N^2)$	O(N)	O(N)	$O(N^2)$	O(N)	$O(N^2)$	$O(N^2)$

Tabulka 1: Porovnání časové a paměťové náročnosti popsaných algoritmů

V tabulce jsou uvedeny časové a paměťové složitosti tak, že u algoritmů, které byly dále implementovány, je taková složitost, kterou má nakonec implementovaná verze. U ostatních jsou to hodnoty odpovídající tendenci v tabulce.

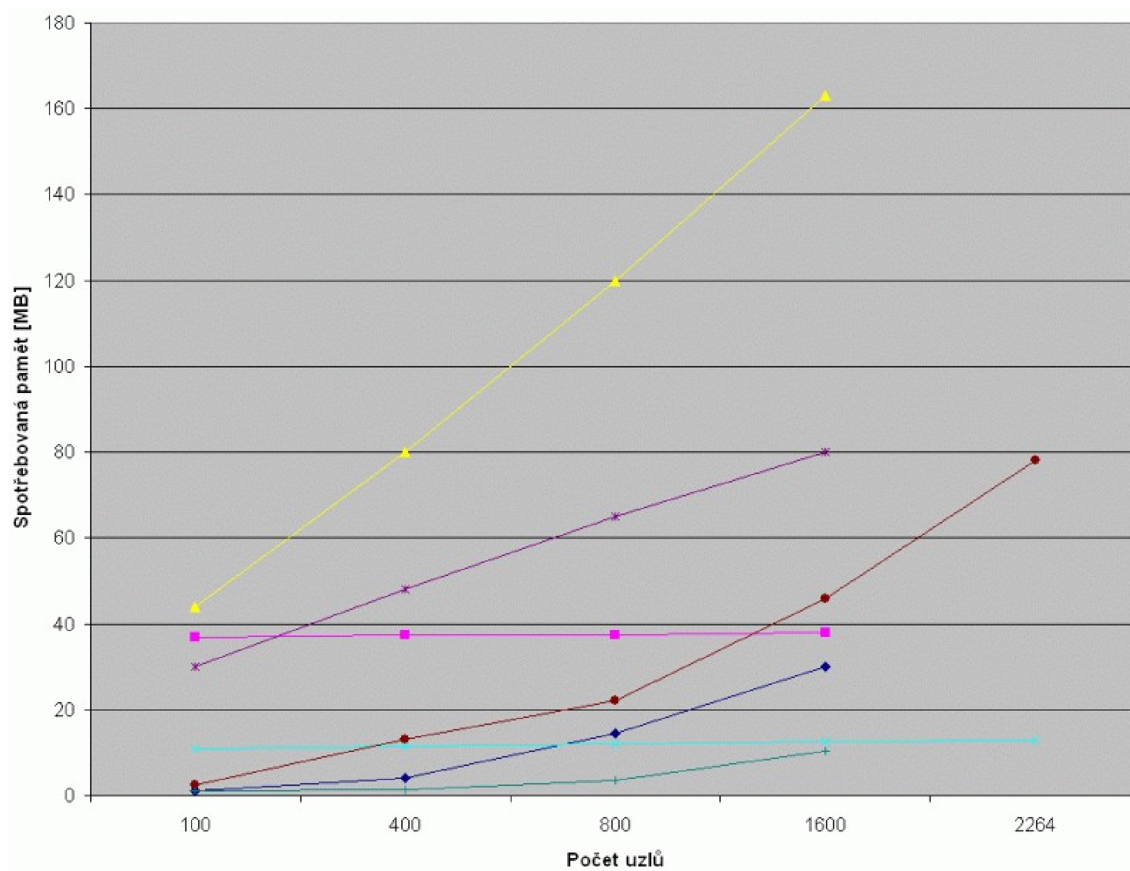


Graf 1: Porovnání časové složitosti popsaných algoritmů



Tabulka 2: Legenda ke grafu 1

Graf 1 pěkně ukazuje odlišnosti časové složitosti jednotlivých algoritmů. V grafu je vynechána poslední hodnota (3200), kvůli lepší čitelnosti grafu. Grafy obou dvou algoritmů, které se k této hodnotě úspěšně dostaly, mají tendenci zůstávat na nízkých hodnotách.



Graf 2: Porovnání paměťové složitosti popsaných algoritmů

Genetický algoritmus
Metoda lokálního hledání
Iterativní metoda
Hladový algoritmus
Horolezecký algoritmus
Metoda zak. prohledávání
Metoda sim. žíhání

Tabulka 3: Legenda ke grafu 2

Graf znázorňuje vývoj paměťové složitosti výše popsaných algoritmů v závislosti na počtu uzlů, tedy velikosti vstupních dat. Opět v grafu z důvodu čitelnosti chybí poslední v tabulce uvedená hodnota pro 3200 uzlů. Opět bylo možno tuto hodnotu vypustit, protože se nijak neměnila stávající tendence.

3. Implementace sériových řešení

3.1 Výběr algoritmů k sériovému zpracování

Na základě výše uvedené tabulky, grafů a popisů vlastností jednotlivých algoritmů jsem vybíral takové, které se osvědčily při řešení problému obchodního cestujícího. Další postup spočíval v tom, že jsem implementoval čtyři vybrané algoritmy sériově. To jsem dělal s předpokladem výběru dvou algoritmů k převodu na paralelní řešení pomocí vláken.

Jakmile jsem je otestoval, pustil jsem se do výběru. Jako první algoritmus jsem vybral genetický, protože se pro můj problém hodí, vzhledem k vysoké rychlosti nalezení poměrně přesného řešení. Kromě toho jsem již tou dobou předpokládal, že i převod ze sériového provedení na paralelní by nemělo být příliš problematické.

Dalším vybraným algoritmem byl algoritmus simulovaného žíhání. Jednak to bylo z důvodu, že jsem měl k dispozici funkční kód, kterému jsem rozuměl, a měl jsem tedy z čeho vycházet pro další práci. Kromě toho se podle zdrojů na internetu[12] osvědčil tento algoritmus nejen při řešení standardního problému obchodního cestujícího, ale i u jeho modifikací, například s asynchronní délkou cest mezi jednotlivými městy.

Po výběru dvou netriviálních algoritmů jsem se rozhodoval mezi některými z jednodušších. Nakonec jsem jako třetí vybral hladový algoritmus. Ten se mi zalíbil, neboť, na rozdíl od ostatních algoritmů testovaných v hůře ovladatelném appletu, vracel výsledky velmi rychle. Nehledě na to, že se k řešení problému obchodního cestujícího podle zdrojů na internetu[11] také hodil.

Posledním algoritmem, který jsem vybral, byla metoda zakázaného prohledávání. Jednak opět díky vysoké rychlosti, s jakou si s daným problémem poradil, jednak protože je podle zdrojů na internetu [9] vhodný pro řešení tohoto problému.

Ve chvíli, kdy jsem měl vybrané algoritmy, mohl jsem se pustit do jejich implementace sériovým přístupem.

3.2 Sériová implementace

Začal jsem, vcelku pochopitelně, prací na algoritmu, o jehož principu jsem měl vcelku přesnou představu, tedy metodou simulovaného žíhání. Pro samotnou implementaci algoritmů jsem zvolil jazyk C, neboť v jeho užití se cítím nejzběhlejší. Kromě toho jsem nepředpokládal, že by byly potřeba nějaké vymoženosti objektově orientovaného přístupu.

3.2.1 Metoda simulovaného žíhání

Vzhledem k tomu, že kód tohoto algoritmu, ačkoli byl v objektově orientovaném jazyce, neobsahoval objekty, bylo jeho přepsání do jazyka C poměrně snadné.

Jak jsem se již zmínil, vstupní data v této implementaci jsou představována maticí vzdáleností mezi jednotlivými městy. Počet měst se určuje z prvního řádku – počet znaků je roven

počtu měst. Stejný by měl tedy být i počet řádků. Matice vzdáleností funguje tak, že vzdálenost mezi dvěma městy je na pozici průsečíku příslušného řádku a sloupce. Chceme-li například zjistit, jaká je vzdálenost mezi městy 5 a 8, podíváme se na pátý řádek do osmého sloupce. Číslo na tomto místě označuje vzdálenost těchto měst. Je-li hodnota 0, hrana mezi těmito městy neexistuje.

Z tohoto popisu jasně vyplývá, že maximální vzdálenost je 9, po úpravě by bylo možné dosáhnout až vzdálenosti 36, pokud bychom jako možnou hodnotu vzdálenosti neakceptovali pouze čísla, ale i písmenné znaky, které by se posléze přepočítaly na celočíselnou hodnotu.

Program po spuštění nastaví určité konstanty pro výpočet. V původním kódu bylo možné je zadat jako parametry spuštění programu. Toto mi v mém případě přišlo poněkud nadbytečné, proto jsem tuto možnost zrušil a konstanty nastavil napevno v kódu. Jde například o koeficient tuhnutí, počet iterací, startovací a koncovou teplotu.

První, co program udělá, je načtení dat (matice vzdáleností) do paměti a alokování této paměti. Z načtených dat zjistí, výše zmíněným způsobem, počet měst, a je tedy možné alokovat paměť i pro pomocná a pracovní pole. Poté se spustí výpočet nejlepší cesty metodou simulovaného žíhání.

Prvním krokem výpočtu je nalezení nějaké úvodní cesty splňující podmínky hamiltonovské kružnice. Funkce přidává do pracovního pole náhodná města, dokud mezi městy existuje hrana. V případě, že hrana neexistuje, zruší se poslední dvě města a funkce se snaží pokračovat v cestě někudy jinudy. Pokud jsou již uložena všechna města, zkontroluje se ještě existence hrany mezi posledním a prvním městem. Jestliže taková hrana existuje, bylo nalezeno úvodní řešení a je možné pokračovat ve výpočtu, konkrétně spustit samotné žíhání.

Samotné žíhání probíhá ve vnořeném cyklu, vnější cyklus běží, dokud se teplota postupným ochlazováním nedostane až na hranici koncové teploty. Vnitřní cyklus probíhá podle na začátku definované konstanty. Ve vnitřním cyklu probíhají pokusy o zlepšení řešení na základě předávané teploty. Nejlepší nalezená cesta je uchovávána pro případ, že by algoritmus dosud nejlepší cestu opustil a žádnou lepší již nenašel. U každého řešení se určuje, zda je lepší nebo horší než stávající nejlepší řešení, je-li lepší, zazálohuje se. Jestliže není lepší, určí se náhodnou funkcí, zda má šanci vést k lepšímu řešení, nebo ne.

Po doběhnutí celého algoritmu se uvolní matice vzdáleností (celou dobu byla v paměti, aby bylo možné zjišťovat vzdálenosti mezi městy), a vypíše se výsledek.

Jakmile jsem dokončil tuto metodu, dospěl jsem k názoru, že všechny další algoritmy budou také potřebovat nějaké úvodní řešení, rozhodl jsem se využít v dalších případech opět výše popsaný algoritmus. Z tohoto kódu tedy vycházím i při implementaci ostatních algoritmů.

3.2.2 Metoda zakázaného prohledávání

Program začíná stejně jako předcházející. Načte data, alokuje pro ně paměť, spustí výpočet. Výpočet zahájí nalezením nějakého úvodního řešení, které se bude postupem času a programu zlepšovat.

Další výpočet probíhá tak, že se algoritmus snaží najít lepší řešení, než jaké našel dosud. V případě, že jej najde, uloží si cestu i její cenu. Jestliže se nenajde cesta lepší, zvýší se počítadlo stavů beze změny. Dosažení určité hodnoty tohoto počítadla je podmínkou ukončení celého algoritmu.

Hledání nového řešení probíhá tak, že se náhodně vygeneruje město, které se umístí do nové cesty. Vybrané město se porovná se všemi městy v tzv. tabu listu, což je seznam všech měst, která byla využita v poslední době a nesmí být použita znovu, dokud budou v tomto seznamu.

Jestliže město v tabu listu je, musí se vygenerovat nové město. Jestliže v tabu listu není, je akceptováno a zařazeno do nově vznikající cesty a zároveň je uloženo do tabu listu, aby nemohlo být hned použito znovu. Velikost tabu listu jsem po několika pokusech stanovil na trojnásobek počtu měst.

Ve chvíli, kdy je splněna výše uvedená podmínka dosažení určitého počtu stavů beze změny, je hledání dalších řešení ukončeno a za nejlepší cestu je považováno dosud nejlepší nalezené řešení. Program již pouze dealokuje paměť, vypíše výsledek a ukončí se.

Počet stavů beze změny, potřebný k určení konce výpočtu, jsem stanovil typicky na 15. K tomuto číslu jsem došel na základě pokusů, kdy právě při této hodnotě algoritmus vracel relativně dobré výsledky v krátkém čase. Při nastavení této podmínky na nižší hodnotu byl algoritmus velmi rychlý, až se doba jeho trvání měřila těžko, ale cesty, které nacházel se od původního, tedy zcela náhodného, řešení příliš nelišily.

Na druhou stranu, při vyšším čísle trval algoritmus výrazně déle. To by teoreticky nevadilo, pokud by nacházel výrazně lepší řešení, ale zlepšení nebylo tak značné jako zpomalení. Například 500 měst bylo pospojováno s podmínkou na standardní hodnotě 15 stavů beze změny za cca 4s a celková vzdálenost byla 1367j (délkových jednotek). Když jsem na identických datech spustil algoritmus s podmínkou nastavenou na 20, trval výpočet až 12s, ale výsledná cesta byla 1359j, tedy toto zpomalení nevedlo k výraznému zlepšení řešení.

3.2.3 Hladový algoritmus

Program začíná stejně jako referenční metoda zakázaného prohledávání. Načte data, alokuje pro ně paměť, spustí výpočet. Výpočet zahájí nalezením nějakého úvodního řešení, které se bude postupem času a programu zlepšovat.

Další výpočet probíhá tak, že se algoritmus snaží najít lepší řešení, než jaké našel dosud. V případě, že jej najde, uloží si cestu i její cenu. Jestliže se nenajde cena lepší, zvýší se počítadlo stavů beze změny. Dosažení určité hodnoty tohoto počítadla je podmínkou ukončení celého algoritmu.

V tomto algoritmu jsem hodnotu této podmínky stanovil na 10, z podobných důvodů jako u výše uvedené metody zakázaného prohledávání.

Hledání lepší cesty v tomto algoritmu probíhá samozřejmě jinak než u předcházejících. Algoritmus vygeneruje náhodné město, které ještě není v nově vytvářené cestě, a snaží se najít takového souseda, který je co nejblíž. Pokusí se najít souseda ve vzdálenosti mezi 1 a 5, jestliže takového souseda nenajde na několik málo pokusů (10), generuje náhodného jiného souseda, již bez ohledu na vzdálenost.

Algoritmus se tedy snaží najít co nejkratší cestu k nějakému sousedovi, ale nezdržuje se tím, a jestliže nemůže žádného vhodného najít, bere zavděk kterýmkoli.

K tomu, aby se zajistilo, že algoritmus nesáhne po sousedovi, který už se mu jednou nehodil, slouží taková obdoba tabu listu. Zkrátka soused, který nesplnil podmínku krátké vzdálenosti je uložen a po vygenerování nového souseda se kontroluje, zda tento soused nebyl již dříve označen za nevyhovujícího.

Po provedení dostatečného počtu iterací se program opět standardně ukončí, tedy dealokuje paměť pro matici a vypíše výsledek.

Všechny dosud popsané algoritmy si byly svou strukturou velmi podobné. Vždy se vygenerovalo jedno původní řešení, a pak se nějakým způsobem vylepšovalo hledáním lepších sousedů. Poslední algoritmus se tohoto postupu nedrží a funguje poněkud odlišně. Jedná se o genetický algoritmus.

3.2.4 Genetický algoritmus

Ze začátku by se mohlo zdát, že bude fungovat opět stejně jako předcházející, ale není tomu tak. Stejným způsobem jsou jen uložena data, alokována matice a generováno původní řešení. To trochu popírá závěr předcházejícího algoritmu, ale rozdíl tu přece jen je.

V genetickém algoritmu je definováno několik konstant. V tuto chvíli je nejzajímavější konstanta POP_SIZE určující počet jedinců v populaci. Populace v problematice obchodního cestujícího je množina jedinců. A jedinec, to je jedna cesta splňující podmínky obchodního cestujícího.

Algoritmus začíná tak, že vytvoří první populaci, tedy několik prvotních cest. K tomu, aby nebyli všichni jedinci v populaci stejní, je využito náhodné generování počátečního města. První krok, který algoritmus provede, je tedy vygenerování tolika cest, kolik jedinců má mít populace.

Další použitou konstantou je GENERATIONS. Ta určuje počet generací, které se mají v době běhu programu vytvořit. Tedy kolikrát se program pokusí vylepšit jedince v populacích, aby dostal co nejlepšího jedince.

K vytvoření nové populace je nutné starou populaci nějakým způsobem modifikovat. K modifikaci populace slouží několik tzv. genetických operátorů.

První z nich je operátor křížení. Ten se provádí téměř vždy. Jak často se provede, to určuje konstanta CROSSOVER. Ta se vynásobí náhodným číslem a podle výsledku se určí, zda se křížení bude provádět nebo ne. Smysl křížení je takový, že se dva sousední jedinci v jedné populaci zkříží, tedy přesunou města jednoho jedince do druhého a naopak, vždy se převádí jen náhodně velká část jedince.

Křížení samotné probíhá tak, že se náhodně vygeneruje několik měst jednoho jedince, která zůstanou na svém pořadí v cestě, ale přesunou se do druhého jedince. V tomto jedinci proběhne kontrola, a zdvojená města jsou nahrazena chybějícími, protože při přesunu mezi jedinci došlo k přepsání původních měst. Města, která se přesunovala z prvního jedince do druhého, si algoritmus zapamatuje, a v náhodném pořadí je rozmístí zpět do původního jedince. Je tedy možné, že ačkoli se jeden jedinec změní téměř úplně, druhý se nemusí změnit skoro vůbec.

Dalším genetickým operátorem je mutace, využívající konstanty MUTATION. Mutace probíhá v 50 % případů. Zda proběhne nebo ne, se určí stejným způsobem, jako se určilo, zda proběhne křížení.

Mutace samotná se týká pouze jednoho jedince, kdy se v něm vyberou náhodná města, a náhodně se opět vrátí zpět na jiné místo. Kolik měst a která se budou vyměňovat, je určeno náhodně. Tato náhoda je ovšem trošku ovlivněna, totiž tak, že pokud se bude mutace provádět, tak se minimálně dvě města prohodí. Není tedy možné, aby se na základě MUTATION určilo, že se jedinec má zmutovat a v mutaci samotné se nemutovalo vůbec.

Posledním genetickým operátorem, který je v programu použit, je selekce. Využívá konstantu ELITISM_VALUE. Selekce probíhá jako poslední operace a probíhá vždy.

Před selekcí se seřadí všichni jedinci ve staré i nově vzniklé populaci od nejlépe hodnoceného po nejhůře hodnoceného. Nová populace pak vznikne z nejlepších jedinců staré populace a nejlepších jedinců dosud vytvořené nové populace. Výše zmíněná konstanta pak určuje, kolik jedinců ze staré populace bude mít to právo dostat se do nové.

Nastavení konstant se zatím osvědčilo dle následujících pravidel:

- POP_SIZE jako stejný počet měst pro malé problémy (do 100 uzlů) a polovina až desetina počtu měst pro větší problémy.

- GENERATIONS jako stejný počet měst pro malé problémy (do 100 uzlů) a jako polovina až pětina POP_SIZE pro větší problémy.
- CROSSOVER nastavena permanentně na 0,9.
- MUTATION nastavena na 0,5.
- ELITISM_VALUE je nastavena na základě pokusů na 10

Při implementaci tohoto algoritmu jsem vycházel z kódu v C++, který byl opravdu objektově orientovaný, a daly se z něj proto vyčíst jednotlivé implementace genetických operátorů.

3.2.5 Testování (generator.c)

Všechny výše popsané algoritmy měly stejný způsob předávání vstupních dat. Kvůli objektivním testům a také kvůli ulehčení práce při vytváření testovacích dat jsem vytvořil jednoduchý program, který vytvořil soubor s testovacími daty. Naplnil matici vzdáleností tak, že vzdálenosti byly generovány náhodně mezi 0 a 9, jen do diagonály se vždy zapsala 0. To kvůli zajištění, aby se nedalo pokračovat z města A do města A. Velikost vytvářené matice se zadává jako parametr spouštění programu.

Díky tomuto programu jsem nemusel věnovat čas vytváření testovacích dat. A navíc jsem měl možnost kontrolovat chování jednotlivých algoritmů na identických datech. Pokud vracely podobné výsledky za podobné časy, usuzoval jsem, že pracují, jak mají.

3.3 Výsledky sériového řešení

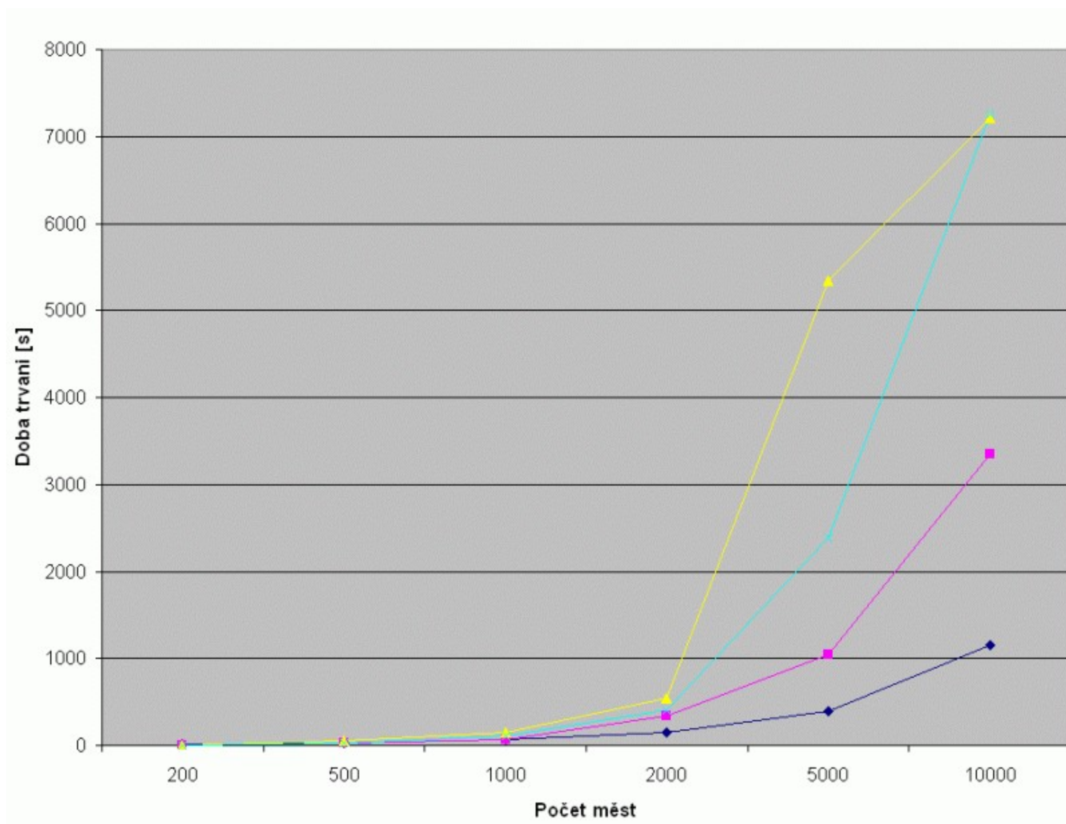
Doba trvání je v tabulce znázorněna vcelku jasně a není k ní snad potřeba žádný obsáhlý komentář. Zmíním se jen o hodnotách nad dvě hodiny. Aby při testování nedocházelo ke zbytečným časovým prodlevám, jsou v algoritmech podmínky délky trvání nastaveny na dvě hodiny. Pokud doba běhu programu tuto hodnotu přesáhne, vypíše aktuální nejlepší výsledek a ukončí se.

Paměťová náročnost je u všech algoritmů velmi podobná, je to z důvodu jednotného zpracování vstupních dat do matice.

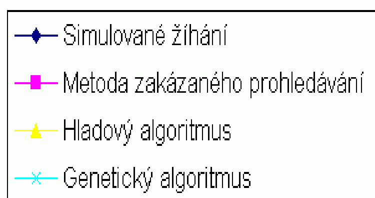
Vzdálenost je udávána v jednotce [j], která symbolizuje délkovou jednotku, ve které jsou zadány hodnoty v matici vzdáleností. Mohou to být milimetry v případě nějakého vrtání děr do desky, nebo kilometry v případě nějakých cest po Zemi.

	Simulované žíhání	*Metoda zakázaného prohledávání*	*Hladový algoritmus*	*Genetický algoritmus*
200: doba trvání	16 s	4 s	6 s	6 s
500: doba trvání	33 s	23 s	50 s	30 s
1000: doba trvání	01:11 min	01:10 min	02:28 min	01:39 min
2000: doba trvání	02:27 min	05:40 min	09:05 min	06:51 min
5000: doba trvání	06:32 min	17:30 min	01:29:00 hod	39:52 min
10000: doba trvání	19:13 min	55:14 min	> 02:00:02 hod	> 02:00:52 hod
200: paměť	0,5 MB	0,5 MB	0,5 MB	2 MB
500: paměť	1,5 MB	1,5 MB	1,5 MB	3,5 MB
1000: paměť	4,5 MB	4,5 MB	4,5 MB	11 MB
2000: paměť	15 MB	15 MB	15 MB	34 MB
5000: paměť	96 MB	96 MB	96 MB	134 MB
10000: paměť	382 MB	382 MB	382 MB	447 MB
200: výsledek	563 j	572 j	538 j	808 j
500: výsledek	1367 j	1446 j	1368 j	2423 j
1000: výsledek	3127 j	3036 j	3049 j	5071 j
2000: výsledek	7448 j	6708 j	6210 j	10430 j
5000: výsledek	22450 j	20404 j	17022 j	26469 j
10000: výsledek	48074 j	45392 j	42743 j	53764 j

Tabulka 4: Porovnání výsledků sériového řešení

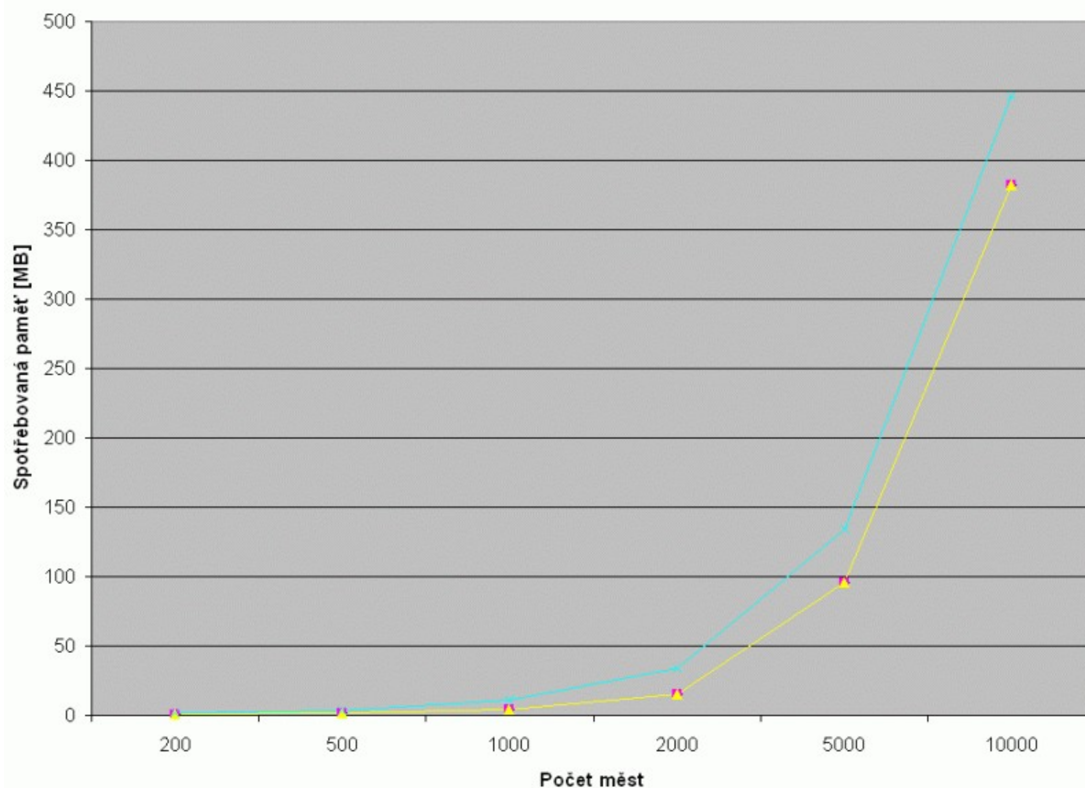


Graf 3: Časová složitost implementovaných algoritmů

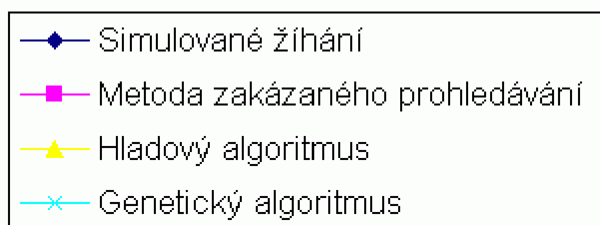


Tabulka 5: Legenda ke grafu 3

Graf časové náročnosti vypadá zcela dle předpokladů, časová náročnost stoupá. Plynulé stoupání narušuje pouze hladový algoritmus, který musel být tvrdě ukončen po dvou hodinách běhu. Stejně byl ukončen i genetický algoritmus, ale vzhledem ke stoupající tendenci jeho grafu, lze předpokládat, že se již blížil k výsledku.



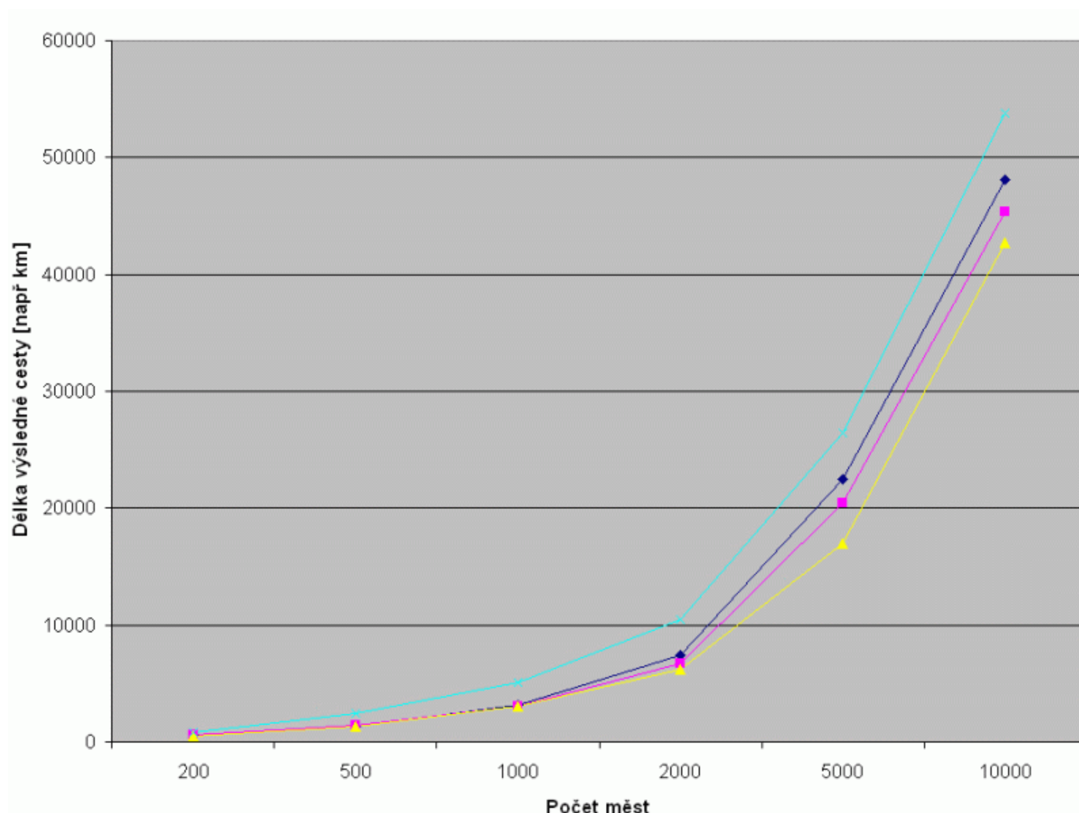
Graf 4: Paměťová složitost implementovaných algoritmů



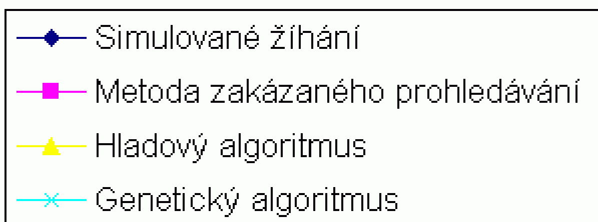
Tabulka 6: Legenda ke grafu 4

Graf paměťové náročnosti názorně ukazuje, že tuto veličinu mají mnou implementované algoritmy identickou. Tedy s výjimkou genetického algoritmu, který má paměťovou složitost vyšší. To je způsobeno tím, že navíc oproti ostatním algoritmům potřebuje ukládat mnohem více cest. Ukládá si vlastně dvě populace, novou a starou, a to se v tomto grafu muselo projevit.

Takto prudce stoupající grafy jsou způsobeny tím, že veškerá vstupní data jsou po celou dobu běhu programu v paměti, v podobě vzdálenostní matice.



Graf 5: Nalezené řešení jednotlivých algoritmů



Tabulka 7: Legenda ke grafu 5

Graf výsledků je zajímavější, zvlášť pokud jej porovnáme s dobou trvání. Ukazuje se, že hladový algoritmus sice hledá velice dlouho, ale na druhou stranu nachází jednoznačně nejlepší řešení. Metoda simulovaného žíhání sice nenachází nejlepší řešení, ale v poměrně rychlých časech nachází velmi dobré řešení. Také je vidět, že genetický algoritmus by se jistě dal zlepšovat, buďto k přesnějším výsledkům nebo k vyšší rychlosti. To bude pravděpodobně způsobeno implementací, která jistě není ideální.

4. Paralelní program pomocí vláken

Dle zadání by měl další postup být takový, že vyberu dva algoritmy z výše zmíněných a převedu jejich sériovou implementaci na paralelní. Tento převod bych měl provádět pomocí vláken na SMP, proto se v této kapitole podrobněji podívám na to, co to vlákna jsou, jak fungují, jak s nimi lze pracovat apod. Jako první ale zjistím, co je to norma POSIX.

4.1 Norma POSIX 1003.1

POSIX .1původně (1990) definoval množinu knihovních funkcí jazyka C pro přístup k základním službám OS, např. open(), read(), fork(). Systémy splňující standard POSIX.1 musejí tyto funkce poskytovat a aplikace chovající se podle POSIX.1 využívají pouze tyto funkce => aplikace jsou na úrovni zdrojových textů přenositelné.

Na rozdíl od většiny standardů (které zahrnují, co si kdo přeje) je POSIX.1 průnikem vlastností SVR3 a BSD[20]

4.2 Proces nebo vlákno

Jako proces je v systému chápán souhrn kódu programu, dat programu, zásobníku, údajů o procesem otevřených souborech, a také informací ohledně zpracování signálů. Tyto všechny informace má každý proces vlastní (privátní) a nemůže je sdílet s jiným procesem, kromě datových oblastí. Při volání jádra fork(2) se pak tyto informace pro nový proces zkopírují, takže jsou pro něj zase privátní.

Jako vlákno si můžeme představit odlehčený proces, tj. pouze kód vlákna a zásobník, vše ostatní je sdíleno s ostatními vlákny téhož procesu. Vlákno je tedy podmnožinou procesu a proces může vlastnit několik vláken. Vlákno samo o sobě v systému existovat nemůže, musí k němu vždy existovat proces, se kterým sdílí všechna data, otevřené soubory, zpracování signálů.[18]

Z toho plyne, že zatímco k tomu, aby spolu mohlo komunikovat více procesů, je potřeba na úrovni systému zajistit možnost předávání informací, je možné v rámci jednoho procesu spustit více vláken, která spolu mohou komunikovat na „půdě“ procesu, který je vytvořil.

Také je jasné, že vlákna, nebo přesněji řečeno vlákno, je používáno i v případě sériového přístupu k problému.

4.3 Implementace vlákna

Existují tři modely implementace vláken. Prvním z nich je ***One-to-one.*** I*mplementace je provedena na úrovni jádra. Každé vlákno je pro jádro samostatný proces, plánovač procesů nečiní rozdíl mezi vláknem a procesem. Nevýhodou tohoto modelu může být velká režie při přepínání vláken.*

Many-to-one. *Implementace je provedena na úrovni uživatele, program si sám implementuje vlákna a vše okolo. Jádro o vláknech v procesech nemá ani tušení. Tento model se nehodí na víceprocesorové systémy, protože vlákna nemohou běžet zároveň (každé na jiném*

procesoru), jeden proces nelze nechat vykonávat na dvou procesorech. Výhodou může být malá režie přepínání vláken.

Many-to-many. *Implementace je provedena na úrovni jádra i uživatele. Tento model eliminuje nevýhody předchozích implementací (velká režie při přepínání procesů, souběžně nemůže běžet více vláken), a je proto použit v mnoha komerčních UNIXech (Solaris, Digital Unix, IRIX).*

V Linuxu je použit model první, velká režie při přepínání vláken je ze značné části eliminována efektivním řešením přepínání.[18]

Podpora vláken může být zabudována přímo do jádra operačního systému, anebo může být implementována na úrovni knihovních funkcí v uživatelském procesu. Podpora vláken na úrovni knihoven a jádra je v různých podobách dostupná na celé řadě systémů. Hlavním problémem, který bránil širšímu využití, byla vzájemná nekompatibilita těchto implementací. Ve světě otevřených operačních systémů nabývají stále většího významu normy IEEE POSIX 1003, které jsou podporovány jak v prostředí systémů Unix, tak ve specializovaných systémech (QNX, Lynx, apod.). V srpnu 1995 byl po dlouhém období upřesňování a doplňování schválen standard POSIX 1003.1c, definující rozhraní pro použití vláken na úrovni jazyka C. Tento standard by měl být implementován ve všech komerčních systémech Unix (AIX 4.1, SOLARIS 2.5, IRIX 6.2, HP-UX) a existují také dvě volně dostupné implementace na úrovní knihoven pro systémy Linux a BSD.

Pokud je podpora vláken zabudována do jádra operačního systému, je procesor přidělován ne na úrovni procesů, ale na úrovni vláken. Každé vlákno je jakoby jedním procesem z klasických operačních systémů. Při volání služeb jádra systému, které pozastavují proces, jako je například read(), není pozastaven proces, ale pouze prováděné vlákno. Termín proces v těchto systémech zahrnuje několik paralelně běžících vláken. Klasický proces odpovídá procesu s jedním aktivním vláknem. Identita procesu, vlastnictví, ochrana, spouštění procesů zůstává zachována, ale vše je interpretováno ve smyslu „identita procesu, jehož součástí je vlákno, které je právě prováděno". Pokud je každé vlákno uvnitř procesu identifikováno a obhospodařováno jádrem systému, jedná se o zobrazení uživatelských vláken na systémové N:N.[19]

4.3.1 Vytvoření vlákna

Vlákno je na úrovni jazyka C tvořeno samostatnou paralelně prováděnou funkcí ve tvaru:

```
void *func(void *arg);
```

Parametr funkce arg je předán při spuštění vlákna a může obsahovat adresu libovolného objektu v programu. Jedna funkce může být samozřejmě spuštěna vícekrát a může tak řídit činnost několika paralelně prováděných vláken. Pro vytvoření a spuštění nového vlákna je určena funkce:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                       void *(*func)(void *), void *arg);
```

Parametr thread je návratový, je v něm vrácena identifikace vytvořeného vlákna. Parametr attr definuje atributy vytvořeného vlákna, pro vytvoření vlákna se standardními atributy lze zadat hodnotu NULL. Funkce func() je funkce řídící běh vlákna. Nově vytvořené vlákno začne svůj běh vyvoláním funkce func() s parametrem arg. Ukončením funkce func() je automaticky ukončeno vlákno a případně předán stav ukončení. Úspěšnost vytvoření vlákna je signalizována výsledkem funkce pthread_create(), podobně jako u jiných funkcí norem POSIX 1003. Hodnota 0 znamená úspěšnost, jiná hodnota je kód chyby. Vlákna jsou na úrovni programu identifikovaná typem pthread_t. Je to obvykle ukazatel na strukturu, ve které je zachycen stav a atributy vlákna. Tento typ a další uvedené typy, prototypy funkcí a makra jsou deklarovány v hlavičkovém souboru <pthread.h>, který musí být vložen na začátku programu.[19]

4.3.2 Data vláken

Všechna vlákna uvnitř jednoho programu, na rozdíl od procesů, sdílejí tentýž adresový prostor. Pokud tedy jedno vlákno modifikuje nějakou část paměti (např. globální proměnnou), je tato změna viditelná v ostatních vláknech. To umožňuje více vláknům operovat se stejnými daty bez vzájemné komunikace.

Každé vlákno má však svůj vlastní zásobník (call stack). Tím je zajištěno, že každé vlákno může realizovat svůj vlastní kód a volat různé další funkce podle svých potřeb. V programu s jediným vláknem platí, že každé volání podprogramu v každém vlákně má svoji vlastní množinu lokálních proměnných, jež jsou uloženy v zásobníku tohoto vlákna.

Někdy je požadováno, aby jistá proměnná byla duplicitně vytvořena tak, aby každé vlákno mělo svou vlastní kopii. Linux podporuje tento požadavek tím, že poskytuje každému vláknu datovou oblast specifickou pro vlákna. Proměnné uložené do této oblasti jsou duplikovány pro každé vlákno, a proto každé vlákno může takovou proměnnou modifikovat, aniž by se změnila její hodnota pro jiná vlákna. Protože jinak vlákna sdílejí tentýž paměťový prostor, nemohou být data ve speciální datové oblasti zpřístupňována normálním způsobem. Pro tento účel se používají speciální funkce, jež umožňují nastavovat a získávat data z této oblasti.

*Speciální data pro vlákna musejí být typu void * a můžete jich vytvořit jakékoliv množství. Každá položka dat se zpřístupňuje pomocí klíče. K vytvoření klíče a tedy k vytvoření nové datové položky je určena funkce pthread_key_create(). Prvním argumentem je ukazatel na proměnnou pthread_key_t. Klíčová hodnota může být použita každým vláknem ke zpřístupnění své vlastní kopie datové položky. Druhý argument funkce pthread_key_create() je tzv. čistící funkce (cleanup function). Pokud pomocí druhého argumentu předáte ukazatel na funkci, provede se její kód a bude jí předána hodnota specifická pro tento klíč.*

*Po vytvoření klíče může každé vlákno nastavit svoji vlastní hodnotu odpovídající tomuto klíči prostřednictvím funkce pthread_setspecific(). Prvním argumentem této funkce je samotný klíč a druhým argumentem je speciální hodnota typu void *, která se má uložit. Ke zpětnému získání dat lze použít funkci pthread_getspecific(), které se jako argument předává klíč.[21]*

4.3.3 Synchronizace vláken

Při práci s vlákny je důležité mít na paměti, že vlákna spuštěná v rámci jednoho procesu pracují na jednom adresovém prostoru. V kombinaci s tím, že jim procesor přiděluje systémové prostředky náhodně, a není tedy jisté, kdy bude které vlákno pracovat a upravovat globální data v paměti, nastává velmi vážný problém, který je potřeba řešit. Problém spočívá v tom, že si jednotlivá vlákna mohou zasahovat do proměnných, se kterými v danou chvíli pracuje jiné vlákno. V tom případě dojde téměř jistě k nějaké chybě, ať už zásahu do nealokované paměti, nebo prostě nebude program fungovat tak, jak má. Jedná se o tzv. problém souběhu.

Vlákna je proto potřeba určitým nějak synchronizovat. Synchronizace může probíhat následujícími způsoby či mechanismy.

Mutex

Řešení problému souběhu při zpracování úloh spočívá v tom, že se dovolí pouze jednomu vláknu v daný okamžik zpřístupnit frontu úloh. Jakmile začne vlákno prohledávat frontu, žádné jiné vlákno k ní nesmí mít přístup, dokud se naše vlákno nerozhodne, zda úlohu zpracovat, a když ano, dokud ji nezpracuje.

Implentace takového řešení vyžaduje podporu operačního systému. Proto Linux poskytuje

tzv. mutexy. Mutex je speciální typ zámku, který v daný okamžik může být uzavřen pouze jediným vláknem. Pokud jedno vlákno uzamkne mutex a pak se druhé vlákno pokusí také mutex uzamknout, zablokuje se. Teprve až první vlákno odemkne mutex, druhé se odblokuje a může dále pokračovat v činnosti. Systémem je garantováno, že se problém souběhu nemůže vyskytnout mezi vlákny pokoušejícími se uzamknout mutex. Vždy pouze jedno vlákno dostane možnost mutex uzamknout a ostatní budou zablokována.

K vytvoření mutexu je určena proměnná typu pthread_mutex_t. Po deklaraci takové proměnné musíte předat její ukazatel funkci pthread_mutex_init(). Druhým argumentem funkce pthread_mutex_init() je ukazatel na atributy mutexu, což je objekt specifikující vlastnosti mutexu. Pokud je ukazatel nulový, předpokládají se implicitní atributy. Proměnná mutexu by měla být inicializována pouze jednou.

Vlákno se může pokusit uzamknout mutex voláním funkce pthread_mutex_lock(). Pokud byl mutex odemknut, uzamkne se a funkce se vrátí okamžitě. Pokud ovšem byl mutex uzamknut jiným vláknem, funkce zablokuje další realizaci vlákna a vrátí se, až bude mutex odemknut. Může být také zablokováno více vláken najednou, ale nelze určit, které se po odemknutí mutexu odblokuje. Toto vlákno může mutex opět uzamknout. Ostatní vlákna zůstanou zablokována.

Voláním funkce pthread_mutex_unlock() se mutex odemkne. Tato funkce by měla být vždy volána z téhož vlákna, které mutex uzamklo.

Použití funkce pthread_mutex_lock() je nevhodné, protože ta se vrátí až po odemknutí mutexu.

Pro tento účel je v Linuxu poskytována funkce pthread_mutex_trylock(). Když tuto funkci zavoláte pro odemknutý mutex, bude uzamknut stejně jako po volání funkce pthread_mutex_lock() a funkce pthread_mutex_trylock() vrátí nulu. Když je však mutex již uzamčen jiným vláknem, funkce pthread_mutex_trylock() volající vlákno nezablokuje. Místo toho se vrátí s návratovým kódem EBUSY. Uzamknutí mutexu jiným vláknem se tím neovlivní. Mutex se můžete pokusit uzamknout později.[23]

Semafory

Mutexy fungují, pokud jsou všechny úlohy předem zařazeny do fronty nebo pokud je nová úloha zařazena do fronty tak rychle, jak rychle vlákna jednotlivé úlohy zpracovávají. Pokud však budou vlákna pracovat příliš rychle, fronta úloh se vyprázdní a vlákna ukončí svoji činnost. Když se potom do fronty zařadí nová úloha, nebude existovat žádné vlákno, které by ji zpracovalo. Bylo by proto lepší mít při vyprazdňování fronty mechanismus pro zablokování vlákna až do doby, než se začne zpracovávat další úloha.

Tímto prostředkem jsou semafory. Semafor je čítač, který může být použit k synchronizaci činnosti více vláken. Podobně jako v případě mutexů je garantováno, že kontrola a modifikace semaforu se provádí bezpečně a že nemůže dojít k souběhu.

Každý semafor je celočíselnou nezápornou hodnotou. Semafory podporují dvě základní operace:

- *Operace wait sníží hodnotu semaforu o jedničku. Pokud je hodnota semaforu nula, operace se zablokuje, dokud nenabude semafor opět kladné hodnoty (v důsledku činnosti jiného vlákna). Jakmile je hodnota semaforu kladná, sníží se o jedničku a operace wait se vrátí.*
- *Operace post zvýší hodnotu semaforu o jedničku. Jestliže byla předtím hodnota semaforu nula a ostatní vlákna byla zablokovaná operací wait pro tento semafor, jedno ze zablokovaných vláken se odblokuje a ukončí se jeho operace wait (čímž se hodnota semaforu nastaví na nulu).*

Operační systém Linux poskytuje dvě mírně odlišné implementace semaforů. Právě

popisovaná odpovídá standardu POSIX. Je vhodné ji používat pro komunikaci mezi vlákny. Pro používání semaforů v kódu je nutné vkládat do zdrojového kódu hlavičkový soubor semaphore.h.

Semafor je reprezentován proměnnou typu sem_t. Před jejím použitím se musí inicializovat pomocí funkce sem_init(). Jako argument se funkci předává ukazatel na proměnnou sem_t. Druhým argumentem je nula (nenulová hodnota by znamenala, že semafor může být sdílen mezi procesy, což není pro tento typ semaforů v Linuxu podporováno) a třetím je počáteční hodnota semaforu. Pokud již semafor není potřeba, dealokuje se pomocí funkce sem_destroy().

Pro čekání na semafor lze použít funkci sem_wait(). Pro operaci post se používá funkce sem_post(). Pro neblokující vlákna lze použít funkci sem_trywait(). Tato funkce je podobná funkci pthread_mutex_trylock() - jestliže má být operace wait blokována, protože hodnota semaforu je nula, vrátí se funkce okamžitě s návratovou hodnotou EAGAIN a volající vlákno se nazablokuje.

Dále je také možné použít funkci pro získání aktuální hodnoty semaforu, sem_getvalue(). Ta uloží hodnotu semaforu do celočíselné proměnné, na kterou ukazuje druhý argument. Neměla by se však hodnota semaforu získaná pomocí této funkce použít k rozhodnutí, zda zvolit operaci post, nebo wait. Mohl by se tak přivodit stav souběhu. Jiné vlákno by totiž během volání funkce sem_getvalue() mohlo změnit hodnotu semaforu. Proto je lepší používat atomické operace post a wait.[24]

Podmíněná proměnná

Zatímco vzájemné vyloučení (mutex) lze použít pouze pro vytváření sdružených kritických sekcí, podmíněné proměnné jsou obecným synchronizačním prostředkem, který lze použít pro pozastavení vlákna a čekání na splnění libovolné podmínky. Podmínka, na kterou vlákno čeká, nemá přímou spojitost s podmíněnou proměnnou, může to být libovolná podmínka, či stav v programu, který je signalizován pomocí podmíněné proměnné. Asociace mezi podmínkou a podmíněnou proměnnou je dána použitím. Protože je podmínka reprezentována nějakým síleným objektem v programu, musí být zajištěno vzájemné vyloučení operací s objektem. Proto je podmíněná proměnná vždy svázána se vzájemným vyloučením operací nad objektem, který reprezentuje podmínku. Vlákno, které čeká na splnění podmínky, musí nejprve získat výhradní přístup k objektu stráženému vzájemným vyloučením. Pak může otestovat stav objektu, zda je daná podmínka splněna, a pokud není, čekat na splnění podmínky pomocí podmíněné proměnné:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

Funkce uvolní jednou nedělitelnou operací výhradní přístup strážený proměnnou mutex a zařadí vlákno do fronty vláken, čekajících na splnění podmínky. Až některé jiné vlákno oznámí, že je podmínka splněna, bude čekání ukončeno a před návratem z funkce pthread_cond_wait() získá původní vlákno opět výhradní přístup strážený proměnnou mutex. Nedělitelnost operace uvolnění a zahájení čekání je podstatnou vlastností tohoto synchronizačního prostředku. Zaručuje, že v době mezi uvolněním výhradního přístupu k objektu a zahájením čekání se nezmění stav objektu a nebude tedy signalizováno splnění podmínky, protože to by se v tomto okamžiku ztratilo.

Cyklus testování stavu podmínky a volání pthread_cond_wait() je nezbytný, protože po ukončení čekání může předběhnout odblokované vlákno při získání výhradního přístupu k objektu jiné vlákno a to může stav podmínky změnit. Splnění podmínky může signalizovat kterékoli vlákno funkcemi:

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Funkce pthread_cond_signal() uvolní z čekání pouze jedno vlákno, zatímco funkce pthread_cond_broadcast() uvolní všechna vlákna čekající na danou podmíněnou proměnnou cond. Není chybou, pokud v daný okamžik nečeká na podmíněnou proměnnou žádné vlákno. Programátor

musí zajistit, aby při uvolnění vlákna byla podmínka skutečně splněna.

Délku čekání na splnění podmínky a uvolnění lze omezit časovým limitem. Čekání s časovým limitem umožňuje funkce pthread_cont_timedwait().[19]

4.3.4 Rušení vlákna

Za normálních okolností končí vlákno buď návratem z funkce vlákna, nebo voláním funkce pthread_exit(). Je ovšem možné požadovat, aby jedno vlákno ukončilo jiné. Takovému procesu se říká zrušení (cancelling) vlákna.

Ke zrušení vlákna slouží funkce pthread_cancel(), které se předává jako parametr identifikační číslo vlákna, jež se má zrušit. Zrušené vlákno může být později spojeno (pokud se nejedná o odloučené vlákno). Návratová hodnota zrušeného vlákna je speciální – PTHREAD_CANCELED.[25]

Vlákna není nezbytně nutné rušit tímto způsobem. Je možné, aby jedno vlákno počkalo, až skončí jiné vlákno se svou prací, a posléze obě vlákna spojit do jednoho. Takto je řešeno ukončování vláken v algoritmech této práce. Spojit vlákna je možné funkcí pthread_join().

4.3.5 Čisticí funkce

Čistící funkce lze využít k dealokaci systémových zdrojů, což je velmi důležité při nepředpokládaném ukončení vlákna. Proto je často užitečné vytvořit čistící funkci i v případě, kdy se nevytváří speciální datová oblast, která má duplikovat jistá data pro každé vlákno. Linux za tímto účelem poskytuje čistící obslužné funkce (cleaning handlers).

*Čistící obslužná funkce je funkce, která by měla být volána v okamžiku ukončení vlákna. Čistící funkce má jeden parametr typu void * a její argument se předkládá při tzv. registraci čistící obslužné funkce. Pak je jednoduché použít tutéž čistící obslužnou funkci pro dealokaci několika systémových zdrojů.*

Čistící funkce je dočasné zařízení používané k dealokaci systémových zdrojů, a to pouze v případě, že vlákno existuje nebo je zrušeno, přičemž se nedokončí jistá část jeho kódu. Za normálních okolností, když vlákno nekončí a není zrušeno, by měly být systémové zdroje dealokovány explicitně a čistící obslužná funkce by měla být odstraněna.

*K registraci čistící funkce se používá funkce pthread_cleanup_push(), které se předává ukazatel na čistící funkci a hodnota argumentu typu void *. Volání funkce pthread_cleanup_push() musí být vyváženo odpovídajícím voláním funkce pthread_cleanup_pop(), která zruší registraci čistící obslužné funkce. Podle zavedené konvence má funkce pthread_cleanup_pop() jeden argument typu int. Je-li hodnota tohoto argumentu nenulová, bude čistící funkce před zrušením její registrace zavolána.[22]*

4.3.6 Užití vláken

Vlákna lze použít jednak jako prostředek paralelního programování pro zvýšení výkonu aplikace, jednak jako prostředek pro dosažení přehlednější a logičtější struktury programu.

Urychlení běhu programu

Využitím více procesorů a paralelním zpracováním lze řadu úloh vyřešit efektivněji. Limitem urychlení je ale počet procesorů v systému. Tento počet nemůže být u architektur se sdílenou pamětí

příliš velký, protože rostou komunikační nároky.

Paralelní vstupní a výstupní operace
Běžné systémy podporují pouze blokující čtení a zápis, proces je pozastaven a po dokončení operace opět spuštěn. Podpora asynchronního čtení a zápisu je sice součástí normy POSIX 1003.1b, ale zatím není příliš rozšířená, případně je omezena jen na některé typy zařízení. Paralelními vlákny lze zahájit velký počet paralelně zpracovávaných vstupních a výstupních operací, což je výhodné jak pro běžné aplikace typu klient/server, tak pro speciální aplikace (zálohování, databáze apod.).

Aplikace s grafickým uživatelským rozhraním
U těchto aplikací musí být zaručeno zpracování vstupních událostí, jinak přestává aplikace reagovat na interakci uživatele. Jakékoli dlouhodobější výpočty, práce se soubory, síťové komunikace jsou bez použití vláken problematické či neřešitelné. Vlákna navíc umožňují přehlednější a čitelnější zápis programu.

Systémy reálného času
Vlákna umožňují jemnější paralelismus než na úrovni nezávislých procesů a efektivnější komunikaci mezi vlákny. Systémy reálného času většinou podporu vláken obsahují a poskytují širší repertoár plánovacích algoritmů a synchronizačních prostředků.[19]

5. Implementace paralelních řešení

Poté, co jsem prostudoval problematiku vláken, uvažoval jsem, které algoritmy, z výše testovaných čtyř, bude nejlepší převést do paralelního řešení.

Vybrat první algoritmus nebyl problém, genetický se přímo nabízí, neboť si každé vlákno může provádět výpočty a operace nad svým jedincem samostatně a je potřeba pouze zajistit, aby si jednotlivá vlákna vzájemně nezasahovala do dat. Nasadit vlákna na tomto algoritmu jsem se rozhodl také kvůli možnosti výrazného navýšení výpočetní rychlosti. Navíc je tímto způsobem možné projít v krátkém čase více různých řešení, což zvyšuje šanci nalezení lepšího řešení.

Problém nastal až při výběru druhého algoritmu. Ty mají všechny vcelku podobnou strukturu kódu, a od počátku bylo jasné, že převádění těchto algoritmů ze sériového provedení na paralelní bude problematické. Po zvážení jsem vybral metodu zakázaného prohledávání, protože se dosud osvědčovala, a chtěl jsem vyzkoušet, jak se na jejích výsledcích projeví zapojení vláken.

Dalším důvodem, proč vybrat právě tento algoritmus, byla skutečnost, že dva ze sériově implementovaných algoritmů jsou vcelku triviální (metoda zakázaného prohledávání a hladový algoritmus), zatímco další dva jsou poměrně náročné. Proto jsem chtěl vybrat jeden z těch náročnějších (genetický) a jeden z triviálnějších.

5.1 Převedení sériových programů na paralelní

Převádět sériový program na paralelní je možné v podstatě dvěma způsoby. Prvním je převod na úrovni algoritmu a druhý na úrovni dat.

Převádění na paralelní řešení na úrovni algoritmu je efektivnější, neboť je při tomto způsobu převodu menší ztráta času na režijních výpočtech. V podstatě jde o to, že se v algoritmu určí funkce, které je možné provádět paralelně a které je nutné provádět sériově. Funkce, které je možné provádět paralelně, se pak spustí v určitém počtu vláken, a algoritmus se tak zrychluje.

Převod na paralelní řešení na úrovni dat probíhá tak, že se vstupní data rozdělí na určitý počet dílů. Na tyto menší díly se pak aplikuje algoritmus celého výpočtu, přičemž každý díl zpracovává jedno vlákno. Například, pokud vstupní data obsahují 500 měst, je možné je náhodně rozdělit na 10 dílů po 50 městech a spustit deset vláken tak, aby každé hledalo nejkratší cestu v oblasti 50 měst. Poté je však potřeba ještě propojit jednotlivé výsledky, což může být opět buďto výpočetně náročné, nebo velmi nepřesné v případě náhodného propojení. Rozhodl jsem se, že tomuto způsobu se pokusím vyhnout a v případě nutnosti trochu upravím první způsob.

5.2 Aplikace změn do algoritmů

Na následujících řádcích bude popsáno, jakým způsobem jsem aplikoval změny vedoucí k paralelnímu provádění vybraných algoritmů, a zmíním se také o problémech, které jsem musel řešit.

5.2.1 Genetický algoritmus

Na genetický algoritmus jsem aplikoval první ze způsobů převodu na paralelní řešení. Jednotlivé genetické operátory mohou být v rámci jedné populace bez problémů prováděny samostatně a tedy paralelně, bez zásahů jiných vláken.

Původně jsem uvažoval o tom, že by si každé vlákno vygenerovalo i své původní řešení, tedy že by se spustilo několik vláken, každé vygenerovalo jedno řešení a poté si každé upravovalo jednoho jedince v jednotlivých populacích. Toto řešení se však ukázalo jako velmi časově náročné (doba provádění výpočtu se tak zvyšovala až desetkrát), a proto jsem od této myšlenky upustil. Všechna původní řešení tedy vytváří hlavní vlákno programu a vlákna jsou pak spouštěna až ve chvíli, kdy se provádí genetické úpravy jednotlivých generací.

Jednotlivá vlákna provádí operace vždy nad svým vlastním jedincem, tedy nejsou potřeba žádné synchronizační nástroje. Při křížení, kdy vlákno potřebuje kromě dat svých i data jiného vlákna, se používají pomocné proměnné, takže vlákna cizí data pouze čtou, ale již je neupravují. K zápisu do nové populace se použije lepší ze vzniklých jedinců.

Po ukončení svých operací se vlákna spojí do jednoho, které provede selekci a posun do další populace, kdy se vlákna opět spustí. Toto by se dalo vyřešit i jinak, například tak, že by všechna vlákna kromě hlavního čekala, až hlavní vlákno provede změny, a pak se zbývající vlákna opět spustila.

5.2.2 Metoda zakázaného prohledávání

U tohoto algoritmu byl zásadní problém, jak jej na úrovni algoritmu převést na paralelní řešení. Již jsem psal, že možnost paralelního řešení na datech jsem zavrhl jako neoptimální.

Nakonec jsem se rozhodl vyřešit tento problém spuštěním algoritmu několikrát. Protože vlákna spolu téměř nekomunikují a nezasahují si příliš do dat, stanovil jsem počet vláken rovný počtu procesorů, které budou danou úlohu zpracovávat. A nakonec vyhodnotit nejlepší z nalezených řešení. Výsledkem je program, který je sice mírně pomalejší než jeho sériová verze, ale na větších datech je přesnější, tj. nalezne lepší řešení. Zpomalení je způsobeno režií vláken.

5.2.3 Simulované žíhání a hladový algoritmus

Oba tyto algoritmy, pokud by se měly převádět na paralelní řešení, by se pravděpodobně převáděly podobně jako metoda zakázaného prohledávání. S podobnými výsledky, takže paralelní řešení by pravděpodobně nacházelo lepší řešení za cenu delšího čekání na výsledek.

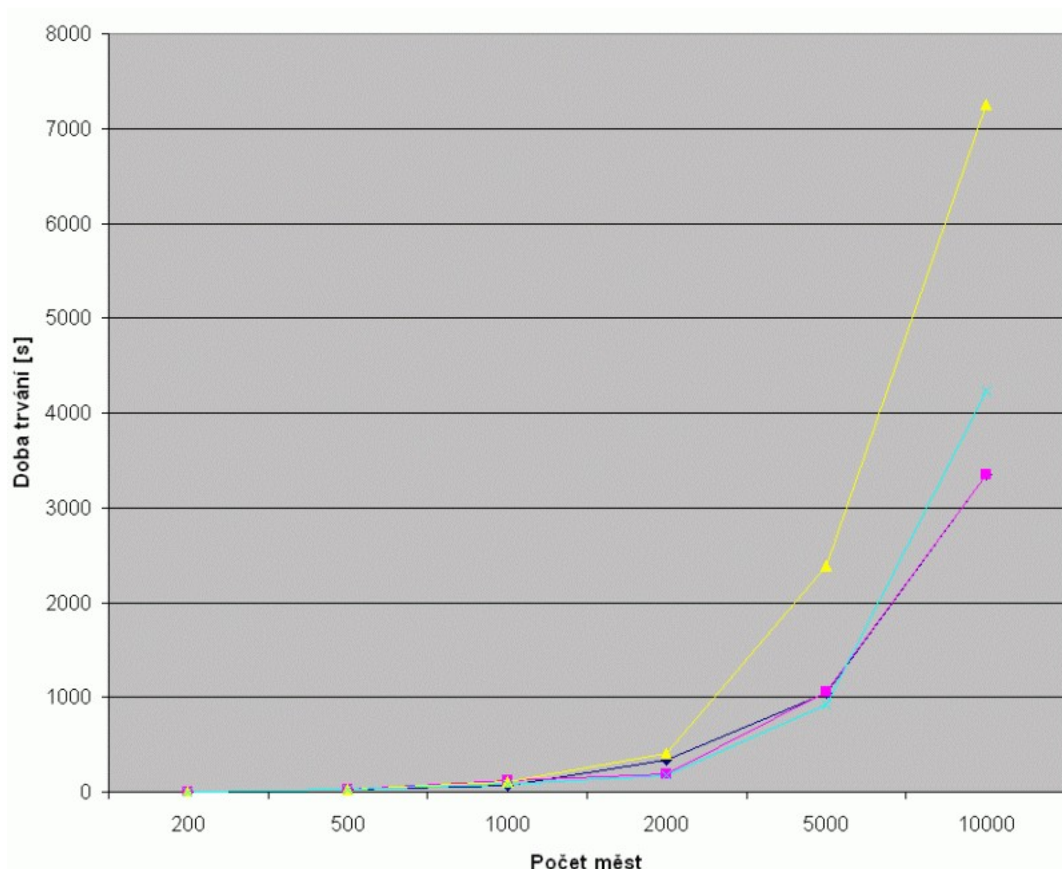
5.3 Výsledky paralelních řešení

Poté, co se podařilo implementovat výše popsaným způsobem oba algoritmy, provedl jsem opět několik pokusů, jejichž výsledky ukazuje následující tabulka. Všechny algoritmy jsou spouštěny na dvoujádrovém procesoru.

	Metoda zakázaného prohledávání (sériově)	*Metoda zakázaného prohledávání (paralelně)*	*Genetický algoritmus (sériově)*	*Genetický algoritmus (paralelně)*
200: doba trvání	4 s	5 s	6 s	6 s
500: doba trvání	23 s	25 s	30 s	24 s
1000: doba trvání	01:10 min	02:04 s	01:39 min	01:19 s
2000: doba trvání	05:40 min	03:12 min	06:51 min	03:02 s
5000: doba trvání	17:30 min	17:31 min	39:52 min	15:31 s
10000: doba trvání	55:14 min	55:33 min	> 02:00:52 hod	01:10:36 hod
200: výsledek	572 j	482 j	808 j	830 j
500: výsledek	1446 j	1298 j	2423 j	2359 j
1000: výsledek	3036 j	2795 j	5071 j	4990j
2000: výsledek	6708 j	6639 j	10430 j	10280 j
5000: výsledek	20404 j	19939 j	26469 j	26504 j
10000: výsledek	45392 j	43455 j	53764 j	54247 j

Tabulka 8: Porovnání sériových a paralelních algoritmů

Tabulka opět vypadá zcela dle předpokladů. Zatímco metoda zakázaného prohledávání trvá v paralelním provedení až dvakrát déle, genetický algoritmus je paralelně velmi rychlý. Na druhou stranu je ještě méně přesný než jeho sériová verze, což byl nejméně přesný sériový algoritmus.



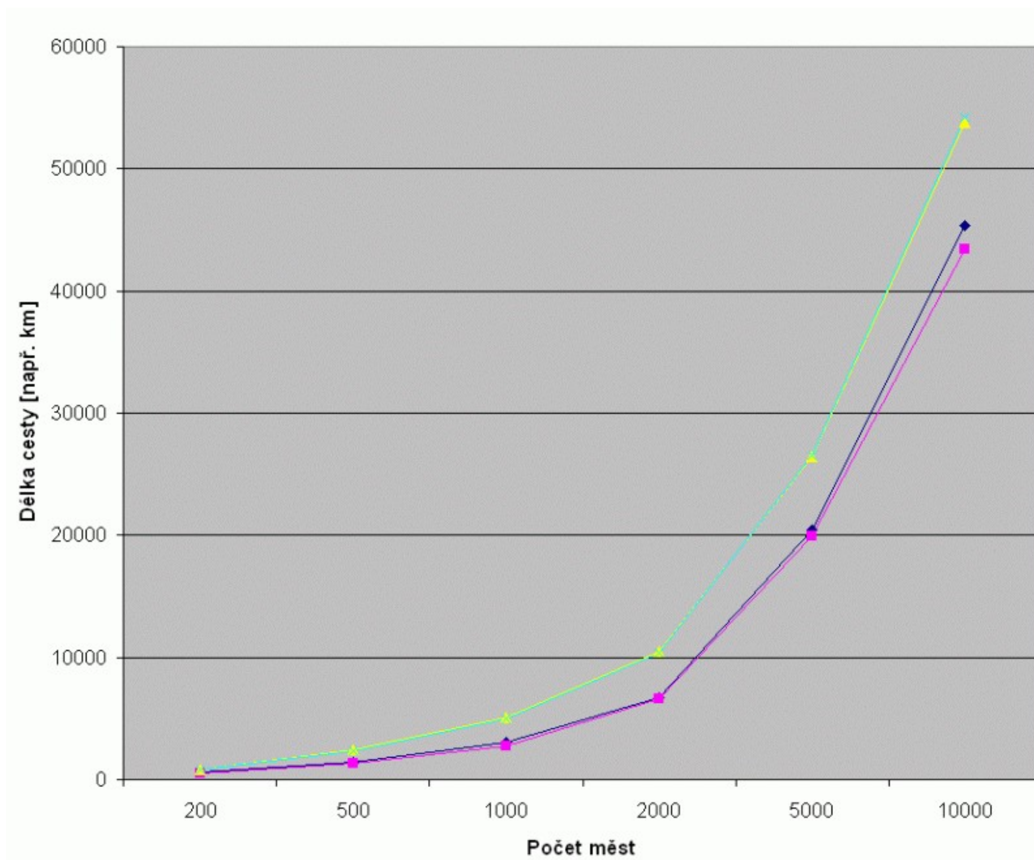
Graf 6: Doba trvání sériových a paralelních algoritmů

Legenda
Metoda zakázaného prohledávání (sériově)
Metoda zakázaného prohledávání (paralelně)
Genetický algoritmus (sériově)
Genetický algoritmus (paralelně)

Tabulka 9: Legenda ke grafu 6

Po aplikaci výše popsaných postupů je vidět, že genetický algoritmus se dvojnásobně zrychlil. To je způsobeno počtem pracujících procesorů. Pokud by se počet ještě zvýšil, doba výpočtu by se ještě zkrátila.

Metoda zakázaného prohledávání prováděla výpočet stejnou dobu. To jsem předpokládal, změna by se měla projevit až v následujícím grafu u přesnosti nalezeného řešení.



Graf 7: Výsledky vrácené sériovými a paralelními algoritmy

Legenda
Metoda zakázaného prohledávání (sériově)
Metoda zakázaného prohledávání (paralelně)
Genetický algoritmus (sériově)
Genetický algoritmus (paralelně)

Tabulka 10: Legenda ke grafu 7

Jak je z grafu vidět, přesnost genetického algoritmu se výrazně nezměnila. Zde záleží také na náhodě, jestli se alespoň jedno vlákno dostane k lepším výsledkům než jiná vlákna.

Na druhou stranu, metoda zakázaného prohledávání se v paralelním řešení poněkud zpřesnila.

Pokud porovnáme tyto dva algoritmy v jejich sériových a paralelních verzích, je vidět, že paralelní verze je v obou případech lepší.

U genetického algoritmu se dostaneme k poměrně dobrému výsledku v kratším čase než u sériového přístupu. Naopak u metody zakázaného prohledávání se za stejný čas dostaneme k výsledku lepšímu.

5.4 Paralelní řešení na vícejádrovém procesoru

Dostal jsem příležitost vyzkoušet své paralelní algoritmy na osmijádrovém procesoru, čehož jsem využil. Následující tabulka ukazuje, jak si s testovacími daty poradily oba algoritmy. Nejprve sériově a poté i v paralelním provedení.

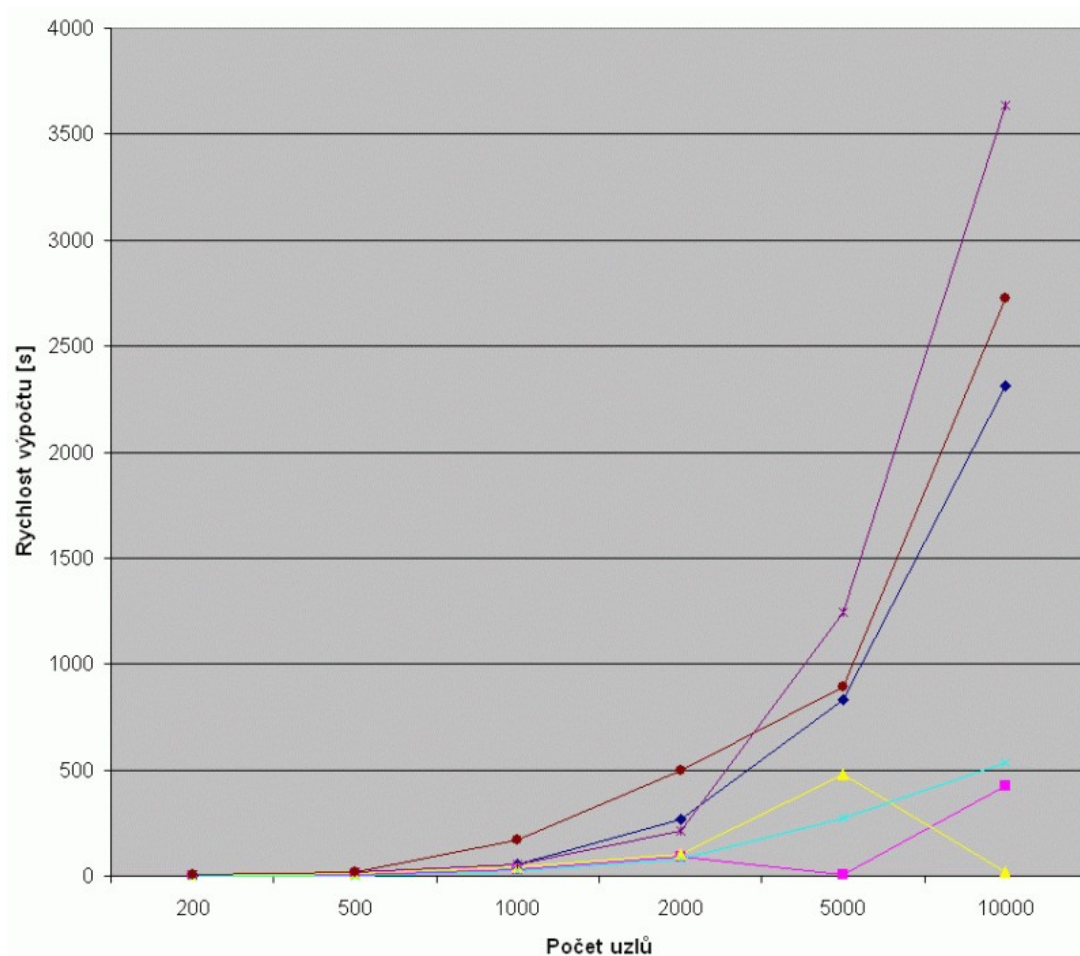
Konkrétně se jedná o stroj *blade44-66 (IBM BladeCenter, 23 modulů HS21 vybavené po 2 čtyřjádrových procesorech Intel Xeon 5345 2,33GHz/1MB, 8-12GB RAM)[31].* Ten je součástí CVT FIT VUT v Brně.

Následující tabulka ukazuje, jak se liší sériové a paralelní řešení na stroji, který má více než jen dvě jádra. Ukazuje, jak se tyto hodnoty vyvíjejí na testovacích datech. Délková jednotka je opět obecně [j] se stejným významem jako u předchozí tabulky.

Tabulka ani grafy nejsou příliš pravidelné, protože se ve výpočtech využívá náhodných prvků, kvůli nimž pak dochází k nalezení horších výsledků i v provedeních, která by měla být teoreticky lepší.

	Metoda zakázaného prohledávání (sériově)	*Metoda zakázaného prohledávání (2 jádra)*	*Metoda zakázaného prohledávání (4 jádra)*	*Metoda zakázaného prohledávání (8 jader)*	*Genetický algoritmus (sériově)*	*Genetický algoritmus (8 jader)*
200: čas	3 s	2 s	3 s	3 s	4 s	4 s
500: čas	17 s	7 s	15 s	0 s	16 s	17 s
1000: čas	56 s	31 s	42 s	22 s	56 s	39 s
2000: čas	04:30 min	01:32 min	01:42 min	01:17 min	03:32 min	02:10 min
5000: čas	13:26 min	07:07 min	07:57 min	04:32 min	20:47 min	14:50 min
10000: čas	38:35 min	08:59 min	13:26 min	08:54 min	>01:00:36	45:23 min
200: cesta	572 j	514 j	490 j	472 j	823 j	845 j
500: cesta	1446 j	1447 j	1269 j	2373 j	2380 j	2457 j
1000: cesta	3063 j	3096 j	2836 j	2977 j	5118 j	5075 j
2000: cesta	6708 j	7187 j	6445 j	6672 j	10473 j	10398 j
5000: cesta	20404 j	20388 j	19945 j	20473 j	26576 j	26324 j
10000: cesta	45392 j	45278 j	42299 j	44770 j	53753 j	53393 j

Tabulka 11: Porovnání paralelních algoritmů na osmijádrovém stroji

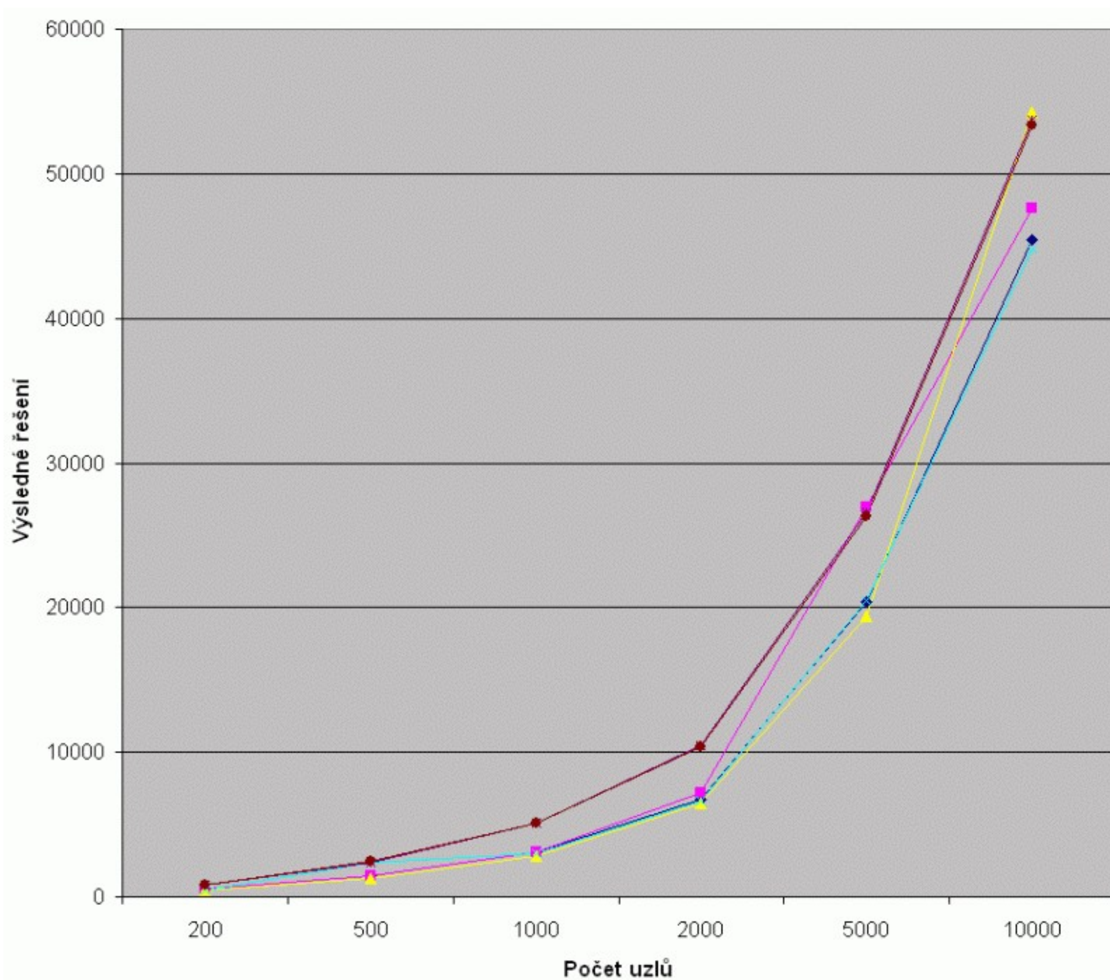


Graf 8: Doba výpočtu na osmijádrovém stroji

Legenda
Metoda zakázaného prohledávání (sériově)
Metoda zakázaného prohledávání (2 jádra)
Metoda zakázaného prohledávání (4 jádra)
Metoda zakázaného prohledávání (8 jader)
Genetický algoritmus (sériově)
Genetický algoritmus (8 jader)

Tabulka 12: Legenda ke grafu 8

Z grafu je vidět, že sériový genetický algoritmus je pomalejší než jeho paralelní verze. Paralelní genetický algoritmus by měl být ještě rychlejší, než je vidět. Toto zpomalení je způsobeno poměrně vysokou režií vytváření a spojování vláken. Metoda zakázaného prohledávání je na druhou stranu velmi rychlá na osmi jádrech. Prudký pokles výpočetního času metody zakázaného prohledávání na čtyřech jádrech je pravděpodobně způsoben zaseknutím v lokálním minimu, kdy se několikrát vrátila jako nejlepší cesta jedna z prvních, a proto došlo k rychlému ukončení. Výsledek potom nebude kvalitní.



Graf 9: Výsledky vrácené paralelními algoritmy na osmijádrovém stroji

Metoda zakázaného prohledávání (sériově)
Metoda zakázaného prohledávání (2 jádra)
Metoda zakázaného prohledávání (4 jádra)
Metoda zakázaného prohledávání (8 jader)
Genetický algoritmus (sériově)
Genetický algoritmus (8 jader)

Tabulka 13: Legenda ke grafu 9

Na grafu je vidět, že mnou implementované paralelní řešení nijak výrazně nezlepšuje výsledky. Tedy že cesta, kterou nalezne sériové řešení, už se příliš vylepšit nedá, alespoň ne tímto způsobem. Co ovšem paralelní přístup zlepšuje, je rychlost výpočtu, neboť je možné stejného výsledku jako u sériového přístupu dosáhnout mnohem rychleji. Na grafu je vidět také ona ztráta v lokálním minimu, do které se dostala metoda zakázaného prohledávání na čtyřech jádrech. Tento výsledek je horší než všechny ostatní.

5.5 Modifikace úlohy

Vstupní data této úlohy byla asymetrická, tedy vzdálenost z místa A do místa B mohla být jiná než vzdálenost z místa B do místa A. Tato skutečnost celou úlohu poněkud komplikuje a prodlužuje dobu výpočtu. Není totiž možné po vyloučení cesty A B C pro přílišnou délku, vyloučit také C B A, protože to může být nejlepší způsob, jak se přes tato tři města dostat.

5.5.1 Nalezení řešení

Ze stejného důvodu se také výrazně zvyšuje počet variant řešení, a je proto nutné brát zavděk i poměrně špatnými řešeními. Protože projít celý stavový prostor a všechny možné cesty je nemožné, nikdy nemůžeme s jistotou určit, že cesta, kterou jsme našli a která nám připadá příliš špatná, nebude nakonec nejlepší, kterou při tomto spuštění najdeme.

Je také dobré myslet na to, že některé operace v těchto algoritmech se řídí náhodou. Je tedy možné spustit algoritmus několikrát na identických datech a pokaždé dostat jiný výsledek, někdy lepší, někdy horší. Tato skutečnost by sice neodpadla ani u symetrické úlohy, ale šance, že se tak stane, by se snížila.

5.5.2 Paměťová náročnost

K tomu, aby se za daných okolností, tedy asymetrickém zadání úlohy, daly určit všechny vzdálenosti, je potřeba zabrat poměrně velkou část paměti. Při použití matice vzdáleností je tak nutné mít uloženou v paměti celou matici.

U symetrického zadání úlohy by stačilo mít v paměti vlastně jen polovinu takové matice. Rozdíl u menších dat by byl vcelku zanedbatelný, ale na desetitisícovém stavovém prostoru Může dojít k problémům na slabších strojích. Na moderních serverech samozřejmě ne.

V případě symetrického řešení by bylo možné hledat cestu proti sobě. Tedy vyrazit z jednoho města na dvě strany a vybírat města v obou směrech zároveň. Bylo by potřeba dávat pozor, aby se v jednom směru neopakovalo město, které je již využito ve druhém a naopak. Bylo by tedy možné, aby na jednom řešení pracovala dvě vlákna, samozřejmě se zajištěním synchronizace, aby obě dvě nepoužila nějaké město zároveň. Lépe řečeno, aby jedno vlákno nepoužilo město, které bylo použito druhým vláknem, ale ještě jím nebylo zneplatněno.

Tento přístup není možný u asymetrického zadání úlohy, protože po spojení obou nově vytvořených cest by nebylo možné určit, kterým směrem je ona nově vytvořená cesta použitelná. Bylo by samozřejmě možné použít tu kratší z obou cest (tam nebo zpátky), ale nemyslím si, že by vzniklé řešení bylo nějak kvalitní. Vzniklo by rychle, ale jeho použitelnost by byla nízká, především proto, že cesta, která byla v polovině stavového prostoru dobrá, bude ve druhé polovině v podstatě neprověřená a spíše náhodná, než cíleně nalezená.

6. Závěr

Všechny algoritmy jsou v tomto provedení paměťově náročné. Je to způsobeno uložením vstupních dat ve vzdálenostní matici po celou dobu výpočtu.

Jinou možností, jak formátovat vstupní data, je mít uloženy jen souřadnice daných uzlů v nějaké souřadné síti. Například předpokládat, že startovací bod je bod [0;0], a mít uloženy informace, že bod A je na souřadnici [1;5], bod B na souřadnici [-2;8] atd. Tento způsob uložení dat by byl jistě méně náročný na data. O to náročnější by byl na výpočetní výkon, neboť by bylo pokaždé nutné přepočítávat vzdálenost mezi dvěma body. Také by to vedlo k výpočtům v plovoucí desetinné čárce, což by výkon opět zpomalovalo. Při uložení dat do matice je možné počítat celočíselně.

V práci jsem se pokusil implementovat několik algoritmů sériovým a paralelním přístupem. Z výsledků testování lze odvodit, že paralelní algoritmy jsou rychlejší a stejně přesné, tedy že za kratší čas vrátí přibližně stejný výsledek. To je vlastně výsledek, který jsem předpokládal, a jsem tedy rád, že testy mé předpoklady potvrdily.

Zjistil jsem, že při přepisování výsledků testů je nutné kontrolovat, zda se algoritmy chovají podle očekávání, a v případě že se tak neděje, je potřeba najít problém. Problém nemusí být nutně v kódu aplikace, může jít jen o ztrátu v lokálním minimu.

Hlavním výstupem práce je dle mého názoru zjištění, že převádět algoritmy na paralelní řešení není snadné, a hlavně že není jeden univerzální způsob, kterým by se dalo řešení daného problému realizovat. Řešení, které jsem aplikoval, není bohužel optimální, ale přesto si myslím, že mi tato práce přinesla nové znalosti i zkušenosti.

Literatura

[1] Internetová encyklopedie wikipedia: http://cs.wikipedia.org/wiki/Hamiltonovský_graf
[2] Internetová encyklopedie wikipedia: http://cs.wikipedia.org/wiki/NP-úplnost
[3] Internetová encyklopedie wikipedia:
http://cs.wikipedia.org/wiki/Problém_obchodního_cestujícího
[4] Internetová encyklopedie wikipedia: http://cs.wikipedia.org/wiki/Problém_P_versus_NP
[5] http://scienceworld.cz/technologie/jak-na-reseni-problemu-zda-np-je-p-1819
[6] http://www.volny.cz/jtuhacek/school/paa_tsp/index.htm
[7] http://www.lupa.cz/clanky/obchodni-cestujici-z-e-shopu-ma-plne-ruce-prace/
[8] Internetová encyklopedie wikipedia http://cs.wikipedia.org/wiki/Genetický_algoritmus
[9] http://www.kiv.zcu.cz/studies/predmety/uir/gen_alg2/E_alg.htm
[10] Internetová encyklopedie wikipedia: http://cs.wikipedia.org/wiki/Heuristické_algoritmy
[11] Internetová encyklopedie wikipedia: http://cs.wikipedia.org/wiki/Hladový_algoritmus
[12] http://majer.czweb.org/scheduling/30Metody_reseni.doc
[13] www.ai.mit.edu/courses/6.034b/searchcomplex.pdf
[14] http://ui.fpf.slu.cz/diplomky/umela_inteligence/UI_Res.htm
[15] Internetová encyklopedie wikipedia: http://cs.wikipedia.org/wiki/Kruskalův_algoritmus
[16] http://alexandria.tue.nl/repository/freearticles/496713.pdf
[17] Internetová encyklopedie wikipedia: http://cs.wikipedia.org/wiki/Asymptotická_složitost
[18] http://www.linux.cz/noviny/1998-0809/clanek11.html
[19] http://www.fit.vutbr.cz/~lampa/papers/vlakna96.html
[20] http://www.kiv.zcu.cz/~luki/vyuka/stare-materialy/os/moje/predn/p2std.d
[21] http://www.root.cz/clanky/programovani-pod-linuxem-specialni-datove-oblasti/
[22] http://www.root.cz/clanky/programovani-pod-linuxem-synchronizace/
[23] http://www.root.cz/clanky/programovani-pod-linuxem-pro-vsechny-mutex/
[24] http://www.root.cz/clanky/programovani-pod-linuxem-pro-vsechny-18/
[25] http://www.root.cz/clanky/programovani-pod-linuxem-identifikacni-cisla/
[26] http://www.tomaskubes.net/CVUT/cestujici.htm
[27] http://www.cs.princeton.edu/courses/archive/spring02/cs226/demo/graph/Graph.html
[28]
http://www.codeproject.com/KB/recipes/tspapp.aspx?fid=2655&df=90&mpp=25&noise=3&sort=Position&view=Quick&fr=51
[29] http://el-tramo.be/software/jsearchdemo
[30] http://www.coin-or.org/Ots/index.html
[31] http://www.fit.vutbr.cz/CVT/servers.html

Přílohy

Příloha 1: CD