VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

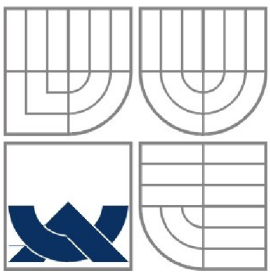
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

PROBLÉM OBCHODNÍHO CESTUJÍCÍHO S VELKÝM POČTEM MĚST

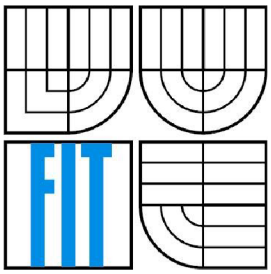
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE LUKÁŠ KUKULA
AUTHOR

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

PROBLÉM OBCHODNÍHO CESTUJÍCÍHO S VELKÝM POČTEM MĚST

LARGE-SCALE TRAVELING SALESMAN PROBLEM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

LUKÁŠ KUKULA

VEDOUCÍ PRÁCE
SUPERVISOR

ING. PAVEL BARTOŠ

BRNO 2011

Abstrakt

Tato bakalářská práce se zabývá problémem obchodního cestujícího s velkým počtem měst. V úvodu je problematika popsána za užití vhodné terminologie. Dále jsou zde popsány nejpoužívanější algoritmy řešící zadaný problém. Detailně je představena heuristika Lin-Kernighan a představen princip genetických algoritmů. Závěr práce je věnován implementaci a vyhodnocení aplikace, která se snaží nalézt nejlepší řešení v co nejkratším čase.

Abstract

This bachelor's thesis deals with large-scale traveling salesman problem. In the beginning of this thesis, the problem is described using suitable terminology. There are also described the most frequently used algorithms for solving given problem. Lin-Kernighan heuristic is shown in detail and the principle of genetic algorithms is presented. The end of thesis is devoted to implementation and evaluation work application that tries to find best solution in the shortest time.

Klíčová slova

Problém obchodního cestujícího s velkým počtem měst, simulované žíhání, zakázané prohledávání, mravenčí kolonie, Inver-over, k-opt, Lin-Kernighan, hybridní genetický algoritmus.

Keywords

Large-scale traveling salesman problem, simulated annealing, tabu search, ant colony optimization, Inver-over, k-opt, Lin-Kernighan, hybrid genetic algorithm.

Citace

Lukáš Kukula: Problém obchodního cestujícího s velkým počtem měst, bakalářská práce, Brno, FIT VUT v Brně, 2011

Problém obchodního cestujícího s velkým počtem měst

Prohlášení

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

.....
Lukáš Kukula
17. 5. 2011

Poděkování

Chtěl bych poděkovat Ing. Pavlu Bartošovi za odbornou pomoc a cenné rady při zpracování této bakalářské práce.

Obsah

1 Úvod

Tématem této bakalářské práce je hledání algoritmu řešícího problém obchodního cestujícího pro velký počet měst. Rozsah problému byl vymezen v řádech tisíců. Nejdůležitějšími faktory ovlivňující výběr algoritmu jsou kvalita výsledků a rychlost jejich nalezení. Symetrická varianta je vhodná pro řešení nejčastějších a zároveň i nejobecnějších problémů. Složkou reálného světa, na kterou lze řešení aplikovat, je především dvourozměrný eukleidovský prostor, ve kterém je většina problémů formulována. Vyjmenovanými směry se ubíraly mé myšlenky a rovněž i tato bakalářská práce.

Problematikou úlohy se bude zabývat druhá kapitola a termíny nutné pro vysvětlení algoritmů řešících problém obchodního cestujícího budou popsány v kapitole třetí. Problém řešící metody jsou obsaženy v následujících pěti kapitolách a vybraný implementovaný model bude popisovat devátá kapitola. Výsledky práce jsou shrnuty v desáté kapitole.

2 Problém obchodního cestujícího

Problém obchodního cestujícího (Traveling Salesman Problem, TSP) je optimalizační úloha, jejíž cílem je nalezení nejlevnější cyklické cesty mezi *N* městy, kde každé město je třeba projít právě jednou. Cena trasy mezi jednotlivými městy nemusí být dána pouze jejich vzájemnou vzdáleností, ale v úvahu může být bráno více odlišných faktorů. Takové ohodnocení může být závislé např. i na čase či na finanční hodnotě.

Od teoretických začátků problému z počátku 19. století se prvním efektivním řešením dá nazývat vyřešená instance o 49 městech z roku 1954. Řešiteli byli G. Dantzing, R. Fulkerson a S. Johnson a od jejich postupu se odvíjí výpočetní techniky i dnes. [2]

2.1 Varianty

Existuje více variant problému obchodního cestujícího. O *asymetrickou* modifikaci problému se jedná za předpokladu, že cena vzdálenosti z města *A* do města *B* není pro libovolný pár měst rovna opačnému směru cesty. Pokud platí rovnost pro *A* a *B*, kde lze za proměnné dosadit libovolná dvě města, jedná se o problém *symetrický*. Když pro libovolné město neexistuje přímá cesta do jiného města, jedná se o variantu *orientovanou*. Podmínku omezující jedinou návštěvu města lze například upravit na hledání takových cest, kde je každé město navštíveno alespoň jednou. Úloha může být také modifikována *dynamičností*, kdy se v průběhu řešení může množina navštěvovaných měst měnit, nebo výskytem tzv. *časových oken*, kde pro vybraná města platí organizační řád, ve kterém se na cestě objevují. Pod pojmem TSP se však v naprosté většině případů uvažuje *symetrická* varianta, která bude předmětem studie této práce.

2.2 Využití

Jedny z prvních algoritmů pro řešení TSP sloužily k podobné činnosti, na jakou se samotný název problému vztahuje. Instance obsahující pár desítek prvků byly opravdu transformované problémy týkající se měst a vzdálenostmi mezi nimi. Motivem pro vývoj takových algoritmů bylo plánování tras autobusů či rozvážejících dodávek.

V dnešní době jsou řešeny problémy až několik tisíckrát větší a jejich řešení jsou aplikována v odvětvích jako jsou například:

- logistika
- sekvenování genů
- skenování/testování řetězců

- vrtání plošných spojů
- míření teleskopů a rentgenových paprsků
- data clustering
- vedení kabeláže
- krystalografie
- řízení robotů
- chronologické řazení

2.3 Matematické zadání symetrické varianty

Matematicky lze symetrický problém obchodního cestujícího vyjádřit jako hledání permutace obsahující všechna města o počtu *N* minimalizující kvantitu $\sum c(x,y)$, kde *c(x,y)* představuje cenu mezi cestami *x* a *y* (*x*, *y* = 1, …, *N*).

V eukleidovském prostoru lze zápis rozepsat na:

$$c(x,y)=\sqrt{\sum_{i=1}^{D}(x_i-y_i)^2}$$, kde *D* značí rozměr prostoru.

Vyhodnocovací funkcí problému lze rozumět součet všech ohodnocených hran na výsledné trase.

Při symetrické variantě platí $c(i,j)=c(j,i)$ pro každá *i* a *j*. Při pohledu na cestu jako permutaci měst, existuje *N*! možných řešení. Následujících pár zápisů je však identických, protože byť cesta začíná v jiném městě, či je cesta nalezena v jiném směru, jedná se vždy o stejný cyklus:

1 – 6 – 5 – 2 – 4 – 3, 5 – 2 – 4 – 3 – 1 – 6, 1 – 3 – 4 – 2 – 5 – 6

Každou cestu lze zapsat 2*N* způsoby, množinu všech možných řešení je tedy možné zredukovat na:

$$|S|=\frac{N!}{(2N)}=\frac{(N-1)!}{2}$$

Množina řešení problému 10 měst obsahuje řádově 10^5 možností, avšak třeba při pětinásobku vstupních měst se množina řešení zvětší na nezvladatelných cca 10^{62} možností.

2.4 Řešené instance

2.4.1 Heuristicky

Aktuálně největším řešeným problémem je hledání nejkratší cesty mezi 1 904 711 městy (a výzkumnými stanicemi na Antarktidě) rozmístěnými po celé zeměkouli. Nejlepší cesta byla nalezena K. Helsgaunem za použití jeho varianty Lin-Kernighan heuristiky. Trasa o délce 7 515 786 987 km byla nalezena 4. dubna 2011. Autor drží rekord v nejlepší nalezené trase již od 16. září 2003. Od té doby jej stihl několikrát vylepšit. Nguyenův hybridní genetický algoritmu našel nejkratší trasu naposledy v roce 2003, kterou dokázaly překonat pouze výsledky K. Helsgauna. Oba způsoby řešení TSP budou probrány v kapitole 8.

Dolní mez světového TSP byla naposledy vypočtena v roce 2007. Její hodnota je 7 512 218 268 km. Aktuálně nejlepší nalezená cesta tudíž dosahuje nejhůře o 0,0477 % horšího výsledku. [21]

2.4.2 Exaktně

Největším exaktně vyřešeným problémem bylo nalezení optimální trasy napříč Švédskem. Trasa byla hledána mezi 24 978 městy a její nalezení by v souhrnném přepočtu na jediný 2,66-GHz Intel Exon procesor trvalo kolem 84 let. [2]

2.5 NP-úplnost

Podle časové složitosti algoritmů určených pro řešení optimalizačních problémů lze algoritmy neformálně rozdělit na dva základní typy. Polynomiální a nepolynomiální.

„*Řekneme, že algoritmus je polynomiální, jestliže jeho časová složitost je* $O(N^k)$ *pro nějakou konstantu k, případně obsahuje logaritmus. Řekneme, že algoritmus je nepolynomiální (resp. exponenciální), jestliže jeho časová složitost není polynomiální.*“ [23]

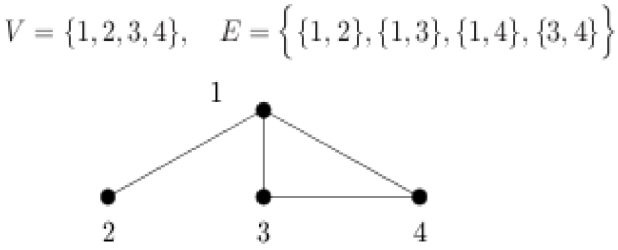
Symetrická varianta TSP je velmi obtížný kombinatorický problém. Pro řešení úloh s velkým počtem měst exponenciálně roste množina možných řešení. Úlohy, pro které nebyl dosud nalezen polynomiální algoritmus a jejichž časová složitost je pravděpodobně exponenciální, jsou řazeny do množiny NP-úplných problému.

Třída složitosti NP je třída všech rozhodovacích problémů, jejichž řešení lze ověřit v polynomiálně omezeném čase nedeterministickým algoritmem. Do NP-úplné subkategorie se navíc řadí nejobtížnější úlohy, do které spadá problém obchodního cestujícího.

3 Teorie grafů

Pro řešení TSP je vhodné zavést zadání do prostředí zobrazeného pomocí grafů. Cílem hledání je nalezení nejkratšího hamiltonovského cyklu v úplném ohodnoceném grafu s *N* uzly.

„Graf (rozšířeně obyčejný či jednoduchý neorientovaný graf) je uspořádaná dvojice G = *(*V, E*), kde* V *je množina vrcholů (resp. uzlů) a* E *je množina hran – množina vybraných dvouprvkových podmnožin množiny vrcholů.“* [15]



Ilustrace 1: Graf s vrcholy V a hranami E. Zdroj: [15]

Hranu mezi vrcholy *u* a *v* značíme jako {*u*, *v*}, nebo zkráceně *uv*. Vrcholy spojené hranou jsou *sousední*. Na množinu vrcholů známého grafu *G* je odkazováno jako na *V (G)*, na množinu hran *E (G)*.

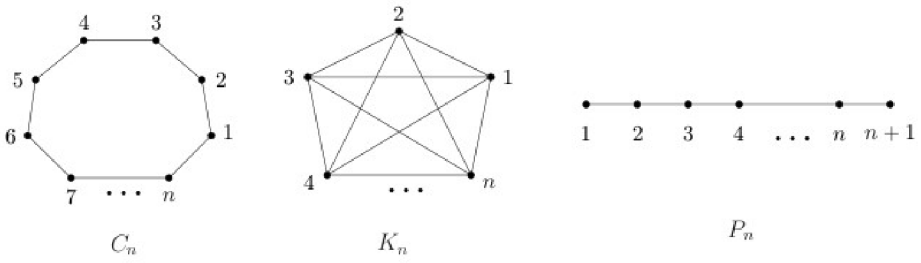
Při hledání řešení symetrického problému obchodního cestujícího jsou důležité následující pojmy: [15]

Kružnice C_n délky *n* má $n \geq 3$ vrcholů spojených do jednoho cyklu *n* hranami.

Hamiltonovská kružnice v grafu *G* je kružnice obsahující všechny jeho vrcholy.

Úplný graf K_n je takový graf o vrcholech $n \geq 1$, ve kterém jsou každé dva vrcholy spojené právě jednou hranou.

Cesta P_n (kde $n \geq 1$) má $n + 1$ vrcholů spojených za sebou *n* hranami.



Ilustrace 2: Kružnice (C_n*), úplný graf (*K_n*) a cesta (*P_n*). Zdroj: [15]*

Podgrafem grafu *G* rozumíme libovolný graf *H* na podmnožině vrcholů $V(H) \subseteq V(G)$, který má za hrany libovolnou podmnožinu hran grafu *G* majících oba vrcholy ve *V (H)*.

Graf *G* je *souvislý*, jestliže pro každé dva vrcholy *x* a *y* v něm existuje cesta z *x* do *y*.

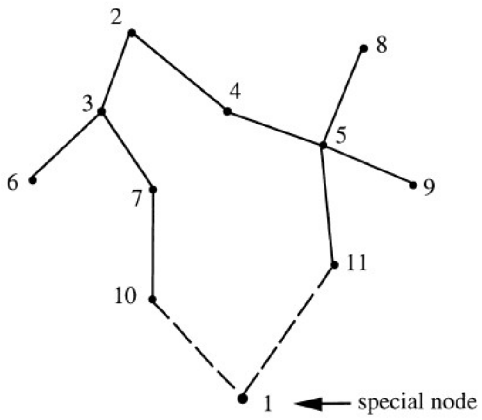
Ilustrace 3: Souvislý (a) a nesouvislý (b) graf. Zdroj: [15]

Je-li *e* hrana grafu, číslo *w*(*e*) se nazývá její ohodnocení, cena nebo váha. Graf se nazývá *ohodnocený*, má-li každá jeho hrana váhu.

Strom je souvislý graf neobsahují kružnici.

Kostra grafu *K* je strom, který je jeho podgrafem a obsahuje všechny vrcholy grafu *G*. *Minimální kostra* ohodnoceného grafu je jeho kostra, kde je součet hran minimální.

1-tree grafu *G* = (*V*, *E*) je kostra nad $V - \{1\}$ hranami spojená s dvěma hranami z *E* obsahující jediný vrchol neobsažený v kostře. Příkladem 1-tree může být ilustrace 4. 1-tree je *minimální*, pokud součet jeho hran je minimální. [9]



Ilustrace 4: 1-tree. Zdroj: [9]

Zadání úlohy TSP by na základě uvedených informací šlo také formulovat jako hledání minimálního 2-regulárního 1-tree.

Pro účely TSP bude dále odkazováno na pojmy trasa, resp. podtrasa. *Trasa* je hamiltonovská kružnice, kde *n* je počet vrcholů instance TSP. *Podtasou* se bude rozumět podgraf trasy.

4 Exaktní řešení

Exaktní algoritmy zaručují nalezení optimálního řešení v ohraničeném počtu kroků. Mezi exaktními algoritmy hraje v současnosti nejdůležitější roli algoritmus Branch and cut. Algoritmus je hybridem metody větve a mezí a Cutting plane. Zmíněny zde budou pouze základní vlastnosti těchto metod. Řešení TSP s velkým počtem měst za pomocí metody Branch and cut totiž v dnešní době zaměstnává procesor výpočetní techniky řádově jednotky až desítky let.

4.1 Metoda větví a mezí

Vyhledávací metoda větví a mezí (branch and bound) vznikla i pro účely studie TSP. Stále se ale jedná o obecný algoritmus pro hledání optimálních řešení optimalizačních problémů spočívající v systematické organizaci množiny všech potenciálních řešení do podskupin rozdělených dle kritérií. Každé větvení dělí prostor řešení do podmnožin za účelem vytvoření podproblémů, které by šly vyřešit snáze než problém původní. Před takovými operacemi je například vypočítat odhad horní a dolní hodnoty optimální cesty. Dalším typickým znakem této metody je totiž i tzv. odsekávání větví, což jsou podmnožiny obsahujících neoptimální řešení. Účelem mezí je zamezení zbytečného prohledávání „neplodných“ větví, které nemohou nalézt lepší řešení, než kterého již bylo dosaženo.

4.2 Cutting plane

Metoda Cutting plane byla navrhnuta R. Gomorym v polovině 20. století pro řešení úloh celočíselného programování. Pro TSP byl přístup lineárního programování přednesen autory G. Dantzigem, R. Fulkersonem a S. Johnsonem. Podstatou metody je rozložení formulace problému na soustavu lineárních nerovnic, které jsou relaxovány. [2]

Cestu skrz N měst lze reprezentovat jako incidenční vektor délky $N(N\text{-}1)/2$, jehož komponenty znázorňují hrany mezi všemi městy. Pokud hrana nabývá hodnoty 1, je součástí trasy. Pokud nabývá nulové hodnoty, tak nikoli. Nechť x je incidenčním vektorem instance o N městech, potom $c^T x$ znázorňuje ohodnocení takové cesty. Označením S množiny incidenčních vektorů všech možných cest lze problém popsat:

$$\text{minimalizovat} \quad c^T x \quad \text{na základě} \quad x \in S \tag{4.1}$$

Příkladem takové relaxace definující hledaný prostor je:

$$0 \leq x_e \leq 1 \quad \text{pro všechny hrany } e \tag{4.2}$$

kde pro všechna $x \in S$ a každou hranu e, nechť x_e značí část x odpovídající e. Spolu s

$$\sum (x_e : v) = 2 \quad \text{pro všechna města } v, \text{ kde } v \text{ je vrcholem v } e \tag{4.3}$$

kde v každé cestě $x \in$ S a pro každé město v, musí $x_e = 1$ pro přesně dvě hrany obsahující v.

Problém tedy lze zapsat jako

$$\text{minimalizovat} \quad c^T x \quad \text{na základě (4.1), (4.2)} \tag{4.4}$$

Ukázkou relaxace (4.1) je rozložení zadání na (4.4). Takhle proces Cutting plane pokračuje až na rozdělení do řešitelných relaxací. [2]

4.3 Held-Karp

Nejpoužívanějším algoritmem pro výpočet dolní meze je Held-Karp. Výpočet dolní meze je pro práci Branch and cut naprosto nezbytný, avšak využití má i v aproximačních metodách. Pro instance, u kterých ještě nebyla nalezena optimální cesta, je cílenou metou právě hodnota dolní meze vypočítána metodou Held-Karp.

Výpočet meze navržený oběma autory je založen na minimální kostře grafu. Ze zadaných N vrcholů je jeden odebrán a ze zbytku vytvořena minimální kostra. Vyjmuté město je k minimální kostře spojeno dvěmi hranami (obsahujícími vrchol v) s minimálním ohodnocením. Cena vzniklého 1-tree však má k optimální trase ještě daleko. Jeho hodnota bývá dokonce i menší než optimální trasa může být. K toužené hodnotě lze dle autorů dosáhnout následujícími kroky.

Pokud je pro každý vrchol v vybráno číslo y_v, které odečteme od obou hran v cestě obsahující vrchol v, trasa se nezmění. Jedna trasa však bude levnější než druhá. Pro libovolnou hranu e skládající se z měst u a v se tak od její hodnoty odečte ($y_u + y_v$). Hodnota nově vzniklé trasy je rovna předešlé minus $\sum 2y_v$, kde $v \in V$ a V značí množinu všech měst. Pokud mez nové trasy nabývá hodnoty B, potom mez původní cesty byla $B + \sum 2y_v$.

Není pravidlem, že z nové trasy sestavený 1-tree se bude rovnat předešlému, protože došlo ke změnám vzdáleností mezi uzly. Tento fakt je základním bodem pro nalezení dolní meze. Při vhodném nastavení hodnot y_v totiž lze vytvořit nový optimální 1-tree, jehož hodnota dosáhne optimální dolní meze. [2]

5 Stochastické algoritmy

Pro problémy s větším počtem měst, kde exaktní algoritmy z časového hlediska selhávají, je vhodné pro nalezení dobrých řešení v krátkém čase použít heuristické algoritmy. Jelikož cílem této práce je nalezení co nejpřesnějšího řešení v co možno nejkratším čase, budou tyto i na náhodě založené nedeterministické algoritmy prozkoumány podrobněji než exaktní metody. Velmi kvalitních cest tak lze dosáhnout za použití nepatrného množství výpočetní techniky.

Pro každou cestu *x* v množině řešeních *S* lze definovat operátory s jejichž použitím lze pro každou cestu najít tzv. cestu *sousední*. Příkladem může být jednoduchý inverzní operátor 2-swap, který zaměňuje pozici vybraného města na trase s pozicí libovolného dalšího města. Na trase mezi městy tak dojde k inverzi podtrasy. S takovými sousedními řešeními pracuje mj. horolezecký algoritmus, ze kterého vychází řada dalších pokročilejších algoritmů zabývajících se problémem obchodního cestujícího.

5.1 Horolezecký algoritmus

Algoritmus prohledává prostor možných řešení ve směru nejmenšího spádu. Obecná verze vychází z náhodné varianty z množiny všech řešení. Pro vybranou cestu je vybráno okolí, z kterého se vyhodnocovací funkcí vybere cesta s nejmenším ohodnocením. Nejlepší dosažený výsledek se ukládá a pro další iteraci algoritmu se může použít buď uložený mezivýsledek, nebo je generována nová cesta.

Byť je algoritmus snadno implementovatelný, jeho kvalita řešení se odvíjí hlavně od inicializované cesty a od množství lokálních optim funkce. Algoritmus totiž často uvázne v lokálním extrému či na tzv. rovinách, kde mají sousedé minimální rozdíl vyhodnocovacích funkcí, a kteří jsou vzdálení od lokálních extrémů. Výběr malého počtu sousedů nebo nepříznivá funkce může také vést k zacyklení. I kvůli této skutečnosti je ukončovací podmínkou algoritmu vypršení stanoveného počtu iterací.

```
iterovaný_horolezecký_algoritmus() {
        t = 0
        best = VALUE_MAX
        while (t < TIME_MAX) {
          local = false
          náhodně vygeneruj trasu v_c
          while (local == false) {
            vyber všechny cesty v sousedství v_c
            vyber z nich cestu v_n nejlpépe splňující vyhodnocovací funkci
            if (eval(v_n) je lepší než eval(v_c))
              v_c = v_n
            else
              local = true
          }
          t++
          if eval(v_c) je lepší než best
            best = v_c
        }
        return best
}
```

Pro obtížnější problémy jako je TSP mohou být vhodnější modifikace horolezeckého algoritmu – a to např. simulované žíhání či tabu search.

Časová složitost: $O(N)$

Paměťová složitost: $O(N)$

5.2 Simulované žíhání (SA)

Stejně jako horolezecký algoritmus i simulované žíhání (simulated annealing) neodmítá pracovat s cestami, které nedosahují minimálních hodnot. Od horolezeckého algoritmu se liší tím, že aktuální řešení iterací jsou schopna překonávat lokální extrém. Místo výběru nejlepšího řešení z množiny sousedních řešení si metoda simulovaného žíhání zvolí jedno řešení z této množiny náhodně.

Metoda je inspirována fyzikálním procesem žíhání tuhého tělesa. Při žíhání kovů s nestabilní krystalovou mřížkou totiž dochází ke stabilizaci volných částic k optimálnímu stavu, tj. k vytvoření stabilní krystalové mřížky. Takový kov má potom lepší vlastnosti. Krystal je zahříván na vysokou teplotu až k bodu tání a poté je velmi pomalu chlazen (žíhán). Nestabilní částice se tak eliminují a kov získá požadovanou optimální kvalitu. [23]

Tento jev je algoritmem simulován výskytem parametru teploty T, která se v každém kroku iterace sníží násobením koeficientem o velikosti obvykle mezi 0,8 a 0,99 (tzv. redukční faktor teploty). V případě, že aktuální řešení iterace není nejlepší dosažené, hraje roli v případném přijetí horšího výsledku právě teplota. Pravděpodobnost přijetí horšího řešení je vypočítána vztahem (5.1).

$$p = \frac{1}{1 + e^{\frac{eval(v_c) - eval(v_n)}{T}}} \tag{5.1}$$

Simulované žíhání nabízí lepší možnost nalezení globálního extrému než horolezecký algoritmus. Pravděpodobnost přijetí umožňuje i přeskočit lokální extrémy a mířit ke globálnímu, což horolezecký algoritmus nezvládne.

```
simulované_žíhání() {
        nastav teplotu T
        náhodně vygeneruj trasu vc
        while (T > T_MIN) {
          for (i = 0; i < počet_opakování_pro_teplotu; i++) {
            náhodně vyber cestu vn ze sousedství
            if (eval(vc) je lepší než eval(vn))
              vc = vn
            else if random(0,1) < e^(eval(vn)-eval(vc)/T)
              vc = vn
          }
          T = redukční_faktor_teploty * T
        }
        return vc
}
```

Tabulka 1 udává pravděpodobnosti přijetí řešení, které je o 13 jednotek horší pro různé teploty.

T	$e^{\frac{-13}{T}}$	***p***
1	0,0000002	1,00
5	0,0743	0,93
10	0,2725	0,78
20	0,52	0,66
50	0,77	0,56
10^{10}	0,9999999	0,50

Tabulka 1: Pravděpodobnost přijetí horšího řešení na základě teploty T. Zdroj: [16]

Dle testů v [11] nedosahovalo simulované žíhání kvalitních výsledků pro příklady obsahující tisíc měst. V tabulce 2 jsou reprodukovány nejdůležitější hodnoty. Číslo v instanci reprezentuje počet měst. Hodnota kvality značí, o kolik procent je vypočtené řešení horší než optimální cesta.

instance	optimum	∅ výsledek	∅ doba trvání (s)	kvalita (%)
rl1304	252 948	573 082	1	126,561
pr2392	378 032	1 258 511	2	232,911
rl5915	565 530	5 377 522	5	850,882

Tabulka 2: Výsledky simulovaného žíhání. Zdroj: [11]

Kvalita = (vypočtená_trasa – nejkratší_zaznamenaná) / nejkratší_zaznamenaná * 100 (%), číselný údaj v instanci je roven počtu vrcholů.

Časová složitost: $O(2N)$

Paměťová složitost: $O(N)$

5.3 Tabu search (TS)

Metoda zakázaného prohledávání (tabu search) je modifikace horolezeckého algoritmu pracující s flexibilní pamětí. Obsahuje navíc také mechanismus snižující nebezpečí zacyklení. Stejně jako simulované žíhání lze algoritmem přijmout cestu s horším ohodnocením než dosavadní minimum. Záleží zde však krom parametru teploty také na historii předchozího prohledávání prostoru řešení. To hraje hlavní roli v rozhodování, zda cestu přijmout či nikoli.

V krátkodobé paměti jsou uchovávány informace posledních transformací, jimiž byly generovány lokálně optimální sousední řešení aktuální cesty. Organizována jsou do FIFO fronty s omezenou délkou, tzv. seznamu zakázaných transformací (tabu list). Malá délka fronty může vést k zacyklení a naopak příliš dlouhá fronta může způsobit, že algoritmus přeskočí nadějná místa vedoucí ke globálnímu řešení.

V dlouhodobé paměti lze uchovávat transformace cest, které budou při vyhledávání ignorovány, např. transformace, které se během běhu algoritmu často opakovaly. Implementovat lze také aspirační kritéria, pomocí kterých je možné (stejně jako u simulovaného žíhání) za určité pravděpodobnosti přijmout i horší řešení. [23]

```
tabu_sarch() {
        t = 0
        best = náhodně generuj trasu v_c
        vytvoř prázdný seznam zakázaných transformací
        while (t <= T_MAX) {
          local = v_c
          for (každá přípustná transformace generující množinu sousedních řešení) {
            v_n = transformace(v_c)
            if (eval(v_n) je lepší než eval(local) && (transformace není mezi zakázanými
            || eval(v_n) je jepší než eval(best)) {
              local =  v_n
              ulož transformaci generující lokální minimum
            }
          }
          if (eval(local) je lepší než eval(best))
            best = local
          if (seznam zakázaných transformací je plný)
            smaž se seznamu nejstarší transformaci
          přidej do seznamu transformaci generující lokální minimum
          t++
          v_c = local
        }
        return best
}
```

Tabu search nepřekonal výsledky simulovaného žíhání a celková doba řešení se dokonce mnohonásobně navýšila. Test v tabulce 3 pro instanci s počtem měst převyšujícím pět tisíc nebyl dokonce kvůli enormní době trvání dokončen. [11]

instance	optimum	∅ výsledek	∅ doba trvání (h)	kvalita (%)
rl1304	252 948	567 439	4,3	124,330
pr2392	378 032	2 216 550	15	486,339
rl5915	565 530	-	-	-

Tabulka 3: Výsledky zakázaného prohledávání. Zdroj: [11]

Kvalita = (vypočtená_trasa – nejkratší_zaznamenaná) / nejkratší_zaznamenaná * 100 (%), číselný údaj v instanci je roven počtu vrcholů.

Časová složitost: $O(N)$

Paměťová složitost: $O(N)$

5.4 Mravenčí kolonie (ACO)

Optimalizace za pomoci mravenčích kolonií (ant colony optimization) je heuristický optimalizační algoritmus inspirovaný chováním skutečných biologických systémů. Je založen na rojové inteligenci, což je technika umělé inteligence založená na studiu chování decentralizovaných samoorganizujících

se systémů. Stejně jako v případě rojové inteligence není mravenčí chování ovlivňováno centrální kontrolou a komunikace probíhá pouze mezi mravenci samotnými. [17]

Činnost algoritmu transformuje chování mravenců v kolonii na kooperující stochastické konstruktivní procedury, které mezi sebou komunikují. Mravenci jsou vypuštění náhodně do vrcholů instance TSP a hledaná trasa je dána jejich chováním. Při simulaci hledání potravy značí feromonem průchod trasy ve směru za potravou a rovněž cestou zpět. Na základě svých biologických vlastností se mravenci začnou pohybovat po kratších trasách. Feromon se dá vyjádřit jako váha hrany. Její hodnota je ovlivňována i dalšími mravenci. Váhy s největším ohodnocením (nejčastěji procházené hrany) skládají stále kratší trasu a silnější feromon přitahuje stále více mravenců. Vliv feromonu je v pseudokódu reprezentován pravděpodobností (výpočet lze dohledat v [23]). Zohledněn je i fakt vypařování feromonů tak, že váhy u jednotlivých spojů během času slábnou. Hledání tak nemůže předčasně konvergovat pouze do jediné cesty. [23]

```
mravenčí_kolonie() {
        t = 0;
        nastavení nulového feromonu pro všechny hrany
        náhodně umísti všech m mravenců
        best = nejkratší na počátku nalezená cesta
        while (t < T_MAX) {
            for (i = 0; i < počet_mravenců; i++) {
               for (j = 0; j < (N - 1); j++) {
                    if (existuje alespoň jedno město ze seznamu kandidátů)
                            vyber následující město dle pravděpodobnosti p
                    aplikuj lokální aktualizaci feromonu
               }
            }
            for (i = 0; i < počet_mravenců; i++) {
              if (eval(trasa_vytvořená_mravencem_i) je lepší než eval(best))
                 best = trasa_vytvořená_mravencem_i
            }
            aktualizuj feromonové stopy pro všechny hrany
            přiřaď nové feromonové stopy všem hranám
            t++
        }
        return best
}
```

ACO si v testování uvedeném v tabulce 4 vedla lépe než předchozí algoritmy, ale „těžší“ problémy nezvládla. Rostly především paměťové nároky a algoritmus nebyl schopen pracovat s problémy nad 7 000 měst. Zdrojem programu bylo [1].

instance	optimum	∅ výsledek	∅ doba trvání (s)	paměťové nároky (B)	kvalita (%)
pr1002	259 045	260 643	12	56 025 354	0,617
pcb3038	137 694	142 109	12	247 948 902	3,206
pla7397	565 530	-	-	-	-

Tabulka 4: Výsledky mravenčích kolonií.

Kvalita = (vypočtená_trasa – nejkratší_zaznamenaná) / nejkratší_zaznamenaná * 100 (%), číselný údaj v instanci je roven počtu vrcholů. Testováno na 2,1GHz PC s 4GB RAM. Program byl přeložen g++ překladačem a běhy probíhaly na OS Ubuntu 10.10 - Maverick Meerkat.

Časová složitost: $O(N)$

Paměťová složitost: $O(N^2)$

5.5 Genetické algoritmy (GA)

Přerušením běhu algoritmu simulovaného žíhání nebo tabu-search lze v libovolné iteraci dosáhnout vždy jednoho aktuálně nejlepšího výsledku. Při cestě za stále lepšími výsledky byla revoluční myšlenkou správa několika řešení současně. Nikoli ve smyslu paralelního běhu stejného programu, ale v soupeření mezi jednotlivými řešeními. S tzv. populacemi řešení pracují evoluční algoritmy. Hlavní motivací GA je snaha simulovat evoluční proces, přírodní výběr a vývoj populace během generací.

Snahy napodobit přírodní jevy by se daly rozdělit dvěma směry. Prvním jsou umělé neuronové sítě napodobující činnost mozku a druhým přístupem jsou genetické algoritmy používané hlavně pro řešení problémů, učení a adaptaci. Problém obchodního cestujícího byl do dnešní doby aplikován do obou přístupů, přičemž výsledky za pomocí heuristik spojených s neuronovými sítěmi nedosahují významných výsledků a jedná se pouze o ilustrativní příklad. Zato genetické algoritmy mají vztah s TSP daleko příznivější.

Genetické algoritmy byly představeny J. Hollandem roku 1975 a dnes patří mezi nejpoužívanější evoluční algoritmy. Stejně jako předcházející metody se i genetické algoritmy řadí mezi základní stochastické optimalizační algoritmy. [23]

5.5.1 Terminologie

Za *jedince* v populaci je považováno jedno možné řešení TSP v podobě reprezentace cyklu obsahujícího všechny vstupní vrcholy. Vrcholy jsou u genetických algoritmů nazývány *geny*, které vytvářejí řetězec nazývaný *chromozóm*. *Populace* je sdružení jedinců, mezi kterými probíhají operace algoritmu. Fenotyp (resp. kvalitu jedince) ovlivňuje *vhodnost*, což je transformovaná hodnota účelové funkce na hledání minima s případnou normalizací do daného intervalu. [16]

5.5.1.1 Vhodnost

Ohodnocení cest mohou nabývat obrovských hodnot s menšími rozdíly, což vede ke stírání podstatných rozdílů mezi vhodnými a nevhodnými jedinci. Z praktických důvodů je vhodné transformovat tato ohodnocení do vhodného intervalu lépe značícího poměr mezi kvalitami jedinců. Transformovanou hodnotou je vhodnost (fitness). Ta ovlivňuje výběr rodičů v populaci pro křížení nebo výběr jedince pro mutaci v reprodukčním cyklu (viz. kapitola 5.5.1.3).

Vhodnost lze počítat:

1. Na základě individuálního ohodnocení jedince:

   Maximální funkční hodnotě je přiřazena maximální hodnota vhodnosti a pro minimální hodnotu účelové hodnoty je přiřazena minimální hodnota vhodnosti. Tím je stanoven interval transformace. Vhodnost všech jedinců lze vypočítat vztahem:

   $$fitness(ind) = \frac{1}{f_{min} - f_{max}} [(1-\epsilon) f(ind) + f_{min}\epsilon - f_{max}] \quad ,$$

   kde f_{max} je maximální hodnota účelové funkce (f_{min} minimální), f(ind) je hodnota účelové funkce aktivního jedince a ϵ je malé kladné číslo (např. 0,01). [23]

2. Na základě řazení jedince v seřazené populaci dle účelové funkce:

   Příkladem rozdělování vhodnosti jedincům na základě pořadí je tzv. *lineární řazení*:

   $$fitness(pos) = 2 - SP + 2 \cdot (SP - 1) \cdot \frac{pos}{N} \quad ,$$

   kde *SP* je výběrový koeficient (selection pressure, resp. selection bias) určující rozdíl vhodnosti mezi sousedními členy v posloupnosti jedinců dle účelové funkce. Nejméně vhodný jedinec má hodnotu *pos* = 0, naopak pro nejvhodnější platí *pos* = *N*-1. Koeficient lineárního řazení leží v intervalu $\langle 1.0, 2.0 \rangle$. [19]

5.5.1.2 Selekce

Operátor výběru, který vrátí jedince s pravděpodobností odpovídající jeho poměrné kvalitě se nazývá *ruleta*. V případě, kdy je vrácen s pravděpodobností odpovídající jeho pořadí v populaci podle kvality se jedná o operátor *ruleta založená na pořadí*. [10]

Při selekci je rovněž možno užití *elitismu*, což je automatické umístění nejlepšího jedince do budoucí populace.

5.5.1.3 Reprodukce

Existují různé modely genetického algoritmu, které se liší v reprodukčním pojetí. V generaci může u jedince dojít k mutaci třeba jen po zkřížení dvou rodičů. V jednom generačním cyklu může nastat libovolný počet těchto operací pro libovolný počet jedinců.

U problému obchodního cestujícího bývá upuštěno od tradiční reprezentace dat v binární formě a pro operace křížení a mutace jsou navrhnuty složitější operátory.

Ke zkombinování dvojice chromozomů slouží operátory křížení jejichž výsledkem může být jediný potomek či dvojice potomků.

Přeuspořádáním genů v chromozomu je charakteristická operace mutace.

5.5.1.4 Modely genetických algoritmů

Uvedené termíny jsou uplatněny v genetických algoritmem několika způsoby. Mezi nejpoužívanější varianty patří algoritmus *Steady-State*, který v jedné generaci může obměnit maximálně jednoho jedince. Algoritmus vytvářející v každé generaci novou populaci se nazývá *Generational*. [10]

```
Steady-State_GA() {
  t = 0
  vytvoř počáteční populaci P
  best = nejlepší jedinec z populace
  while (t < T_MAX) {
    vyhodnoť kvalitu každého jedince v populaci
    vyber jedince z populace P, aplikuj na ně/něj operátor křížení
      a/nebo operátor mutace a vytvoř tak nového jedince
    vyhodnoť kvalitu výsledného jedince a nahraď jím nejhoršího jedince v P
    best = nejlepší jedinec z populace
    t++
  }
  return best
}
```

```
Generational_GA() {
   t = 0
   vytvoř počáteční populaci P
   best = nejlepší jedinec z populace
   while (t < T_MAX) {
      vyhodnoť kvalitu každého jedince v populaci P
      vytvoř novou prázdnou populaci P'
      for (i=0; i< velikost_populace; i++) {
         vyber jedince z populace P, aplikuj na ně operátor
            křížení a/nebo operátor mutace a vytvoř tak nové jedince
         vyhodnoť kvalitu výsledného jedince a vlož jej do P'
      }
      P = P'
      best = nejlepší jedinec z P
      t++
   }
   return best
}
```

5.5.2 Inver-over

Samotné genetické algoritmy na problém obchodního cestujícího příliš nestačí. Neznalost konkrétního problému i spolehnutí pouze na stochastické postupy nesvědčí jak kvalitě výpočtu, tak době řešení. Nejlepším genetickým operátorem se jeví být Inver-over, jehož kvality ční nad ostatními.

Paradoxně se nejedná o žádný ze zmíněných operátorů. Inver-over totiž neoperuje pouze nad jednou či dvěma trasami, nýbrž nad celou populací. Potomek vzniká modifikací jedné trasy na základě informací z populace. Charakteristické prvky modelu pracujícího z operátorem jsou následující: [20]

- Každý potomek soupeří pouze se svým rodičem.
- Použit je právě jediný adaptivní operátor Inver-over.
- Násobnost použití operátoru během generace je proměnlivá.

Algoritmus lze vnímat jako proceduru paralelního horolezeckého algoritmu, kde každý „horolezec“ vykonává proměnlivý počet výměn hran. Podrobněji je operátor popsán nadcházejícím pseudokódem, kde hodnota p leží v intervalu $\langle 0,1 \rangle$.

```
Inver-over() {
   t = 0
   náhodně vytvoř počáteční populaci P
   while (t < T_MAX) {
      for (i=0; i< velikost_populace; i++) {
         S' = jedinec P[i]
         vyber náhodně město c z S'
         while (1) {
            if (rand(0,1) < p)
               vyber město c' ze zbývajících měst v S'
            else {
               náhodně vyber jedince z P
               c' = následující město od c ve vybraném jedinci
            }
            if (c' je sousedním městem k c v jedinci S')
               break
            proveď inverzi jedince S' mezi následujícím městem k c
              a mezi městem c'
            c = c'
         }
         if (eval(S') je lepší než eval(P[i]))
            P[i] = S'
      }
      t++
   }
   return nejlepší jedinec z populace P
}
```

U testů pro případ s 2 392 městy byla v nejlepším případě nalezena trasa o 2,66 % horší než optimum vypočtené algoritmem Held-Karp. Průměrná kvalita byla o 3,56 % horší, podobného výsledku nabyl i test s náhodně generovanými 10 000 vrcholy. Řešení instance s 2 392 městy trvalo přes hodinu. Vzhledem k ostatním genetickým operátorům se jedná o mnohem vyšší rychlostní stupeň, ale na některé pokročilejší metody lokálního prohledávání to nestačí. [20]

6 Cestu vytvářející algoritmy

K rychlému sestavení cest napomáhají algoritmy založené na:

- iterativním přidáváním měst do dílčích podtras.
- spojování vybudovaných fragmentů cest.
- modifikovaných metodách pro hledání minimálních koster.

Prvním případem může být velmi jednoduchý nearest-neighbour algoritmus. Zbylými dvěma body lze charakterizovat přetvořenou metodu Borůvkova algoritmu pro účely řešení TSP. Modifikovaným algoritmem pro hledání minimálních koster je také hladový (greedy) algoritmus. Podobných metod existuje spousta, ale vzhledem k faktu, že metody vytvářející cestu nepatří mezi nejpřesnější, bude stačit se seznámit s vyjmenovanými metodami. Metody slouží především k rychlému vytvoření středně kvalitní cesty pro další cestu zkvalitňující heuristiky. Všechny zmíněné algoritmy vždy vytvoří cestu nad úplným vstupním grafem.

6.1 Nearest-neighbour (NN)

Jeden z nejjednodušších a prvních algoritmů používaných pro řešení TSP.

Postup:

1) Vyber libovolný vrchol tvořící podtrasu.
2) Přidej k poslednímu přidanému vrcholu vrchol, který není součástí již vzniklé podrasy a který se nachází v nejkratší vzdálenosti.
3) Opakuj 2. bod, dokud délka podtrasy neobsahuje všechny vrcholy.

Nejvýraznějším záporem metody jsou velmi dlouhé hrany, které ke konci výsledné permutace značně navyšují celkovou délku trasy.

6.2 Greedy algoritmus

Hladový (greedy) algoritmus je založen na Kruskalovém algoritmu pro hledání minimální kostry. Stejně jako Borůvkův algoritmus neumožňuje připojení k vrcholu se stupněm vyšším než 1.

Postup:

1) Seřaď hrany vstupního grafu dle jejich ohodnocení. Hrana s nejnižší cenou tvoří podtrasu.
2) Vyber hranu s nejnižším ohodnocením, která není součásti již vzniklé podtrasy.
3) Přidej hranu do existující či nové podtrasy, pokud takové přidání
   a) nevytvoří vrchol o stupni vyšším než 2.

b) nezpůsobí zacyklení a to pouze pokud se počet hran ve vzniklé podtrase nerovná počtu vrcholů v grafu.

4) Opakuj 2. a 3. bod, dokud se počet vybraných hran nerovná počtu vrcholů v grafu.

6.3 Borůvkův algoritmus

Algoritmus s moravským původem byl poprvé použit pro konstrukci efektivní elektrické sítě. Výsledky cest jsou mírně horší než u greedy algoritmu, ale při experimentálních pokusech se ukázal být rychlejším a v případech řešení TSP s velkým počtem měst se ukázal v kombinaci s LK být dokonce i výkonnějším než greedy algoritmus. Popsaná verze je modifikací [2] originálního Borůvkového algoritmu pro hledání minimální kostry.

Postup:

1) Pro každý vrchol je nalezena hrana s nejnižším ohodnocením, která vytvoří podtrasu (duplikáty jsou ignorovány).
2) Ke každé vzniklé podtrase je přidána hrana s nejnižším ohodnocením k uzlu se stupněm 1. Hrana nesmí ležet v žádné vzniklé podtrase.
3) Opakuj 2. bod do okamžiku, v kterém se všechny podtrasy sloučí do jediné o počtu hran rovnému počtu vrcholů vstupního grafu.

6.4 Výsledky testů

Při testování se uvedené algoritmy osvědčily daleko lépe než stochastické metody, které obvykle vytvářejí cesty náhodně. Algoritmy byly podrobeny instancím o větším počtu měst, kde dosahovaly relativně kvalitních výsledků ve velmi krátkém čase. Zdrojové kódy jsou čerpány z [21], kde byla pro Borůvkův algoritmus dostupná i modifikovaná verze v tabulce 5 vedená jako QuickBorůvka.

Na základě výsledků testů z tabulky 5 si nejlépe vedly Greedy a Borůvkův algoritmus.

instance	optimum	algoritmus	nejlepší výsledek (10 běhů)	⌀ doba běhu (s)	kvalita (%)
pr1002	259 045	NN	318 250	1	22,855
		Greedy	303 768		17,265
		Borůvka	**295 556**		**14,094**
		QuickBorůvka	303 905		17,317
pcb3038	137 694	NN	170 679		23,955
		Greedy	159 011		15,481
		Borůvka	**158 109**		**14,826**
		QuickBorůvka	162 763		18,206
rl5934	556 045	NN	670 207		20,531
		Greedy	**620 939**		**11,671**
		Borůvka	621 959		11,854
		QuickBorůvka	637 877		14,717
gr9882	300 899	NN	381 453		26,771
		Greedy	358 807		19,245
		Borůvka	**351 256**		**16,736**
		QuickBorůvka	352 787		17,244
usa13509	19 982 859	NN	24 763 131		23,922
		Greedy	**23 225 412**		**16,227**
		Borůvka	23 672 435		18,464
		QuickBorůvka	23 418 153		17,191

Tabulka 5: Výsledky testů cestu vytvářejících algoritmů.

Kvalita = (vypočtená_trasa – nejkratší_zaznamenaná) / nejkratší_zaznamenaná * 100 (%), číselný údaj v instanci je roven počtu vrcholů.

Testováno na 2,1GHz PC s 4GB RAM. Program byl přeložen g++ překladačem a běhy probíhaly na OS Ubuntu 10.10 - Maverick Meerkat.

Časová složitost: $O(N)$

Paměťová složitost: $O(N)$

7 Lokální hledání

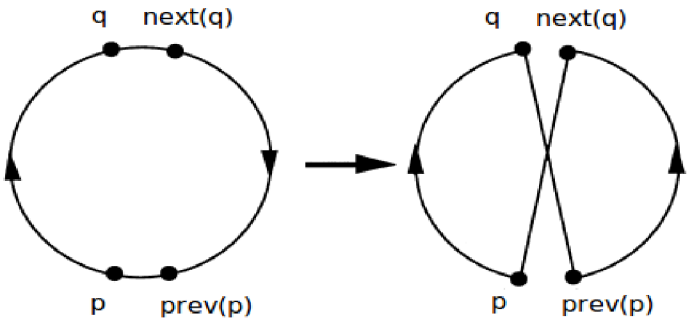
Do kategorie tzv. cestu upravujících algoritmů by se daly zařadit všechny heuristiky z 5. kapitoly. Cílem algoritmů této kapitoly je také postupné zlepšování kvality po jednotlivých iteracích. Největším rozdílem je, že při lokálním hledání se nedbá tolik na náhodu a v každém kroku iterace je čerpáno ze znalosti sousedství.

7.1 *K*-opt

M. Flood při hledání nejkratší cesty po Spojených Státech inicializoval trasu o 49 městech metodou nejbližšího souseda, na kterou v několika krocích aplikoval úpravu lokálních zlepšení. Pro města na trase p a q ($0 \le p \le q < n$) je vyžadováno plnění vztahu (7.1).

$$c(prev(p), p) + c(q, next(q)) \le c(prev(p), q) + c(p, next(q)) \quad . \tag{7.1}$$

Prev(v) značí město nacházející se na trase před městem v a *next*(v) v druhém směru. Pár měst p a q, který podmínku nesplňuje se nazývá *protínající* (intersecting). U takových párů lze provést inverzi podtrasy (nahrazení hran (*prev*(p),p) a (q,*next*(q)) za hrany (*prev*(p),q) a (p,*next*(q)), která vede k cestě s nižším ohodnocením, viz. Ilustrace 5. [2] [9]



Ilustrace 5: Operace swap. Zdroj: [9]

Operace zobrazená na ilustraci 5 se nazývá prohození (swap) měst p a q na trati. Floodův algoritmus končil ve stavu, kdy neexistuje žádný pár měst na trase, který by nesplňoval zadanou podmínku. V podstatě se jedná o podobný problém jako odstraňování křížících se hran u geometrického zadání úlohy. Takové úpravy cesty jsou základní složkou *k*-opt algoritmů.

Algoritmy 2-opt i 3-opt patří mezi nejznámější trasu zlepšující algoritmy. Souhrnně je lze pojmenovat právě jako *k*-opt.

„Trasa je k*-optimální, pokud nelze docílit modifikace trasy způsobené výměnou jejich* k *hran jakýmikoli* k *jinými hranami, z které by vzešla trasa s nižším ohodnocením oproti trase předcházející.*“ [9]

V případě, že je trasa 2D eukleidovské instance 2-optimální, musí být i neprotínající. Dle definice je také zřejmě, že pokud je trasa (k+1)-optimální, musí být i k-optimální. Pro čím větší k je trasa k-optimální, tím větší je pravděpodobnost, že bude trasa optimální. S větším k se ale také zvyšuje prohledávací prostor a složitost roste $O(Nk)$. Nejpoužívanějšími tedy bývají právě algoritmy 2-opt a 3-opt. Experimentálními pokusy bylo navíc zjištěno, že výsledky pro k=4 nebyly viditelně lepší než pro k=3. Je obtížné zjistit, pro jaké k dosáhne algoritmus nejlepšího kompromisu mezi kvalitou řešení a dobou trvání programu. Na základě takových výsledků se příliš nerozšířila modifikace algoritmu pro příliš vysoká k. [9]

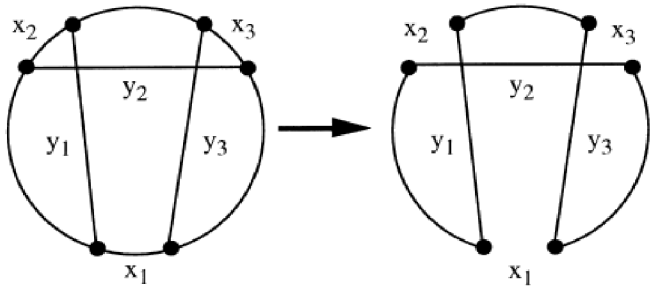
8 Složené algoritmy

V této kapitole budou zmíněny algoritmy kombinované jak z algoritmů cestu vytvářejících, tak nějakým způsobem cestu upravujících. Jelikož předchozí algoritmy mnohdy nenacházely optimální výsledky (zvláště pro velký počet měst), je třeba je zdokonalovat a kombinovat.

8.1 Lin-Kernighan (LK)

Lin-Kernighan patří do kategorie složených optimalizačních algoritmů. V prvním kroku algoritmu se náhodně vygeneruje cesta, která je potom v dalších iteracích konvertována do cest s lepšími ohodnoceními. Heuristika je považována za jednu z nejefektivnějších. Generuje optimu se blížící až optimální řešení symetrického problému obchodního cestujícího. Užíván je při řešení tras obsahujících až desítky tisíc měst.

Jako alternativu ke *k*-opt algoritmům vyvinuli S. Lin a B. W. Kernighan algoritmus nazývaný „proměnný *k*-opt“ [9]. Jádrem je iterativní hledání sekvencí měst, pro které by po aplikaci *k*-opt (*k* je variabilní pro každou iteraci) kroku docházelo ke zkrácení průběžné trasy. V každém kroku iterace algoritmus vyhodnotí, zda by výměna *k* hran vedla k cestě s lepším ohodnocením G_i (cena původní trasy minus cena aktuální trasy). Konkrétně jsou hledány hrany naplňující množiny *X* a *Y* (každá obsahující *k* hran), přičemž se počítá s odstraněním hran z množiny *X* a přidáním z množiny *Y*. Na počátku algoritmu jsou obě množiny prázdné. Koeficient *k* začíná na hodnotě *k*=2 a po každém kroku iterace je inkrementován. Příklad 3-opt transformace je vyobrazen v ilustraci 6.



Ilustrace 6: Sekvenční operace 3-opt. Zdroj: [9]

Pokud je hledání sekvencí hran úspěšné, jsou všechny nalezené podtrasy připravené k výměně uloženy a algoritmus pokračuje vyhledáváním dalších hran. Základní verzi algoritmu lze popsat následujícím pseudokódem: [9]

1) Generuj náhodně cestu *T*.
2) Nastav $i = 1$. Vyber vrchol t_1.

3) Vyber hranu $x_1 = (t_1,t_2) \in T$.
4) Vyber hranu $y_1 = (t_2,t_2) \in T$ aby $G_1 > 0$.

   Pokud nelze najít hranu, která by podmínku splňovala, jdi na krok 12.
5) Nastav $i = i+1$.
6) Vyber hranu $x_i = (t_{2i-1},t_{2i}) \in T$ aby:

   (a) pokud t_{2i} by bylo spojeno s t_1 a provedena výměna hran, výsledek by vytvořil cestu T'.

   (b) hrana x_i byla disjunktní se všemi hranami v Y.

   Pokud je cesta T' kratší než T, nastav $T = T'$ a pokračuj na krok 2.
7) Vyber hranu $y_i = (t_{2i},t_{2i+1}) \in T$ aby:

   (a) $G_i > 0$,

   (b) hrana y_i byla disjunktní se všemi hranami v X.

   (c) x_{i+1} existovalo.

   Pokud lze takovou hranu y_i najít, pokračuj na krok 5.
8) Pokud existuje nevyzkoušená alternativa pro y_2, nastav $i = 2$ a jdi na krok 7.
9) Pokud existuje nevyzkoušená alternativa pro x_2, nastav $i = 2$ a jdi na krok 6.
10) Pokud existuje nevyzkoušená alternativa pro y_1, nastav $i = 1$ a jdi na krok 4.
11) Pokud existuje nevyzkoušená alternativa pro x_1, nastav $i = 1$ a jdi na krok 3.
12) Pokud existuje nevyzkoušená alternativa pro t_1, jdi na krok 2.
13) Konec (nebo jdi na krok 1).

Uvedený algoritmus se od původní verze liší v tom, že novou cestu akceptuje ihned, jakmile je její cena menší než původní. Z podobného modelu vychází především varianta Helsguan Lin-Kernighan. Většina dalších modifikací používá backtracking, pro který by bylo třeba výše uvedený pseudokód změnit v tom smyslu, že by pokračoval v přidávání hran i po nalezení sekvencí hran, jejichž nahrazení by vedlo ke zkrácení cesty.

Jádrem efektivnosti algoritmu je hledání spojů mezi městy, které naplňují množiny X a Y. Za účelem zvýšení efektivnosti je třeba dbát na následující pravidla: [9]

1) **kritérium sekvenční výměny**

   Dvojice hran (x_i, y_i) musí sdílet jedno koncové město. Stejné pravidlo platí i pro každé (y_i, x_{i+1}).
2) **kritérium proveditelnosti**

   Každý vrchol cesty vzniklé transformací musí být 2. stupně.
3) **kritérium pozitivního zisku**

   Striktní pravidlo nutí každou iteraci s novými uzly v množinách X a Y mít vždy pozitivní zisk pro případné provedení transformace.

4) **disjunktní kritérium**

Množiny *X* a *Y* musí být disjunktní.

Obecný algoritmus a jeho základní pravidla jsou neustále aktualizována o nové prvky zvyšující efektivitu algoritmu. Mezi další možné úpravy týkající se omezování prostoru vyhledávání nebo jeho řízení patří např.:

- Pro vyhledávání nahrazovací hrany lze použít pouze omezený počet nejvíce vyhovujících vrcholů.
- Algoritmus může být ukončen ve stavu, kdy je aktuální cesta totožná s předcházející cestou.
- Je vhodné řadit sousedních vrcholy dle efektivity potenciálního využití
- Po dosáhnutí lokálního minima se z něj lze za pomoci nesekvenční 4-opt operace vymanit.
- První cesta není generována náhodně.

8.1.1 Varianta Applegate, Bixby, Chvatal & Cook Lin-Kernighan (LK-ABCC)

Snad nejrozšířenější varianta LK je součástí knihovny Concorde, která patří mezi nejpokročilejší projekty řešící TSP. Od vzniku původního LK algoritmu uplynulo pár desítek let a prošel spousty heuristickými úpravami omezujícími i řídícími vyhledávání. Varianta od výše uvedených tvůrců na základě mnoha testování došla k verzi, která se v mnohém zdá být nejefektivnější. Pojmenování Lin-Kernighan bude v další části textu směrováno právě na variantu LK-ABCC. Klíčové prvky jsou popsány v následující části.

8.1.1.1 Šíře prohledávání

Původní podmínku 2-opt algoritmu, že součet nahrazovaných hran musí být větší než nahrazujících (7.1) je u LK nahrazen podmínkou (8.1).

$$c(base, next(base)) - c(next(base), next(probe)) > 0 \qquad (8.1)$$

Base je vrchol, z kterého se vychází a ke kterému je hledán další vrchol *probe*. Pro operaci prohození (swap) užívají autoři termín *flip*, který nahrazuje hrany následovně:

flip(*next*(*base*), *probe*) nahradí hrany (*base*, *next*(*base*)) a (*probe*, *next*(*probe*)) za (*next*(*base*), *next*(*probe*)) a (*base*, *probe*).

U podmínky (8.1) se jedná o hladový přístup snažící se zlepšit jedinou hranu v cestě. Pro tento případ je také nutné uchovávat informace o průběžném zkrácení cesty v proměnné *delta*. Během přidávání *flipů* u průběžně upravované trasy je požadována platnost (8.2).

$$delta + c(base, next(base)) - c(next(base), next(probe)) > 0 \qquad (8.2)$$

Po provedení operace *flip* se k *delta* přičtou hodnoty nově přidaných hran a odečtou ceny hran zaniklých.

Tzv. *slibný vrchol* (promising vertex) je taková *probe*, pro kterou platí vztah (8.2). Hledání slibných vrcholů je tedy závislé na co nejkratším ohodnocení hrany *c(next(base), next(probe))*, která snižuje velikost průběžného zlepšení cesty *delta*. Jedná se tudíž o uzly ležící v co nejkratší vzdálenosti od *next(base)* na aktuální cestě. Podmínka čistě hledící na vzdálenosti mezi uzly je však krátkozraká a často vede k dlouhým sekvencím nevedoucím ke zkrácení cesty. Místo toho je pro výběr hrany uvažována hrana *(next(base),a)*, (kde *probe* = *prev(a)*). Prioritně jsou vybírány hrany, pro které je *c(prev(a),a) – c(next(base),a)* maximální. Z původní podmínky zůstává alespoň fakt, že *a* jsou hledána v nejbližší vzdálenosti k uzlu *next*(*base)* pro větší efektivnost výpočtu. Tato množina uzlů se nazývá *sousedé* uzlu. [2]

Shrnutě se jedná o hledání sousedů *a* od uzlu *next(base)*, pro které platí, že *prev(a)* je slibným vrcholem. Vyhledávání sousedů ve všech kvadrantech od výchozího vrcholu zvlášť vede k efektivnějšímu prohledávání než nalezení pouhých *k* nejlepších sousedů.

8.1.1.2 Hloubka prohledávání

Algoritmus může pracovat s libovolnou hloubkou prohledávání, standardní nastavení v knihovně Concorde je 25. Příliš hluboké prohledávání podle tvůrců nevede k efektivním výpočtům.

8.1.1.3 Vymanění z lokálního minima

Varianta LK rozlišuje provedení více typů nesekvenčních 4-opt (double-bridge) transformací. Ve všech je vrchol *v* u první hrany vybrán jako ten maximalizující podmínku *c*(*v*,*next*(*v*)) – *c*(*v*,near(*v*)), kde *near(v)* je nejbližší vrchol. První vybraná hrana (*v*,*next*(*v*)) tedy nabývá větší délky, než by měla hrana od vrcholu *v* k jeho nejbližšímu sousedu. Transformace jsou nazývány kopy (kick) a jsou uvažovány následující tři: [3]

Blízké kopy (close kicks) jsou sestaveny z náhodně vybrané množiny vrcholů o velikosti *βn*. Z množiny jsou vybrány 3 vrcholy *w* patřící nejlépe do šestice nejbližších sousedů vrcholu *v*. Hrany z vybraných vrcholů jsou sestaveny jako (*w*,next(*w*)).

U *geometrických kopů* (geometric kicks) jsou vrcholy *w* vybírány náhodně mezi *k* nejbližšími sousedy *v*.

Třetí typ je založen na náhodné procházce o *k*-krocích v grafu sousedů vytvořeném při stanovení šíře prohledávání pro LK. Města, ke kterým taková chůze dosáhla, jsou hledanými vrcholy *w*. Hrany k nalezeným vrcholům jsou sestaveny stejným způsobem jako v předcházejících variantách. Kop se nazývá *náhodná chůze* (random-walk kick).

Při experimentech s těmito nesekvenčními 4-opt transformacemi se ukázal být pro řešení problémů nad 1 000 měst vhodnější geometrický kop než kop blízký. Všeobecně se však ukázala být nejvhodnější varianta kopu náhodné chůze. [3]

8.1.2 Varianta Helsguan Lin-Kernighan (LK-H)

Varianta LK-Helsgaun se od původního LK algoritmu liší především v pokročilejší a časově náročnější vyhledávací strategii vhodných kandidátů pro *k*-opt sekvence. Rozdíly oproti předchozí variantě jsou následující: [9]

8.1.2.1 Šíře prohledávání

Pokud optimální trasa obsahuje hranu, v níž jsou vrcholy příliš vzdálené, nemusí stanovením *k*-nejbližších sousedů algoritmus optimální trasu nikdy najít. Pro zamezení takovým případům je zavedena nová míra vzdálenosti, tzv. α-blízkost analyzující citlivost vyhledávání pomocí minimálních koster grafů.

Minimální kostra grafu obsahuje mnoho společných hran s optimální cestou (70-80 % hran). Hrany náležící k minimálnímu 1-tree nebo ty ležící „poblíž“ jsou s velkou pravděpodobností součástí optimální cesty. Naopak ty, které jsou těmto hranám vzdálené, jsou adepty na nahrazení jinými.

„Nechť T *je minimální 1-tree délky* L*(*T*) a nechť* T+*(*i,j*) značí minimální 1-tree povinně obsahující hranu (*i,j*). Blízkost hrany (*i,j*) je tak definována jako: α(*i,j*)* = L*(*T+ *(*i,j*))* – L*(*T*).“* [9]

Stejně jako předchozí zmíněná verze, tak i LK-H nenaplňuje množinu sousedů vrcholy ležícími v nejkratší vzdálenosti. Sousedství uzlu o *k*-prvcích je naplňováno *k* vrcholy s minimální hodnotou α-blízkosti. Čím menší hodnota α-blízkosti pro hranu spojující vrchol s vrcholem ze sousedství je, tím je slibnější začlenění hrany do trasy. Pro co největší podobnost minimálního 1-tree s optimální cestou, je 1-tree modifikován podobným způsobem jako v případě Held-Karp.

8.1.2.2 Hloubka prohledávání

Základním krokem algoritmu je sekvenční 5-opt operace. Krok je však (stejně jako u pseudokódu) zastaven, jakmile je nalezena trasa s nižším ohodnocením. *K*=5 tedy pouze značí maximální hloubku a algoritmus se skládá z *k*-opt kroků, kde *k*= $\langle 2,5 \rangle$. Kvůli malé hloubce prohledávání LK-H nepoužívá backtracking.

8.1.2.3 Vymanění z lokálního minima

Na rozdíl od originálního algoritmu jsou variantou LK-H používány nesekvenční k-opt častěji a v různých podobách.

8.1.3 Výsledky testů

Na základě výsledku inicializačních metod z tabulky 5 nebyla heuristika LK testována po vytvoření cesty Nearest-neighbour algoritmem. Z porovnání výsledků v tabulce 6 se jeví nejvýhodnějšími metodami pro inicializaci Lin-Kernighan heuristiky algoritmy Greedy a QuickBorůvka. pro LK heuristiku nejvýhodnější. Výsledky testů varianty LK-H se v této práci nacházejí v následující podkapitole, v tabulce 7.

instance	**optimum**	**algoritmus**	**nejlepší výsledek (10 běhů)**	**celková doba běhu (s)**	**kvalita (%)**
pr1002	259 045	Greedy	**259 045**	10	**0,000**
		Borůvka	259 125		0,031
		QuickBorůvka	**259 045**		**0,000**
pcb3038	137 694	Greedy	**137 820**	27	**0,092**
		Borůvka	137 829		0,098
		QuickBorůvka	137 833		0,101
rl5934	556 045	Greedy	558 308	54	0,407
		Borůvka	558 431		0,429
		QuickBorůvka	**557 414**		**0,246**
gr9882	300 899	Greedy	301 534	170	0,211
		Borůvka	301 482		0,194
		QuickBorůvka	**301 142**		**0,081**
usa13509	19 982 859	Greedy	**20 011 166**	263	**0,142**
		Borůvka	20 019 684		0,184
		QuickBorůvka	20 022 366		0,198

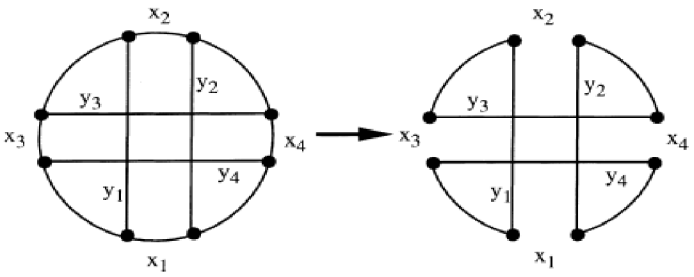
Tabulka 6: Výsledky testů LK heuristiky.

Kvalita = (vypočtená_trasa – nejkratší_zaznamenaná) / nejkratší_zaznamenaná * 100 (%), číselný údaj v instanci je roven počtu vrcholů. Testováno na 2,1GHz PC s 4GB RAM. Program byl přeložen g++ překladačem a běhy probíhaly na OS Ubuntu 10.10 - Maverick Meerkat.

8.1.4 Shrnutí

Lin-Kernighan nalézá velmi kvalitní cesty pro problémy o různých koncentracích měst. Důležitou součástí algoritmu je ale i jeho opakované použití. Samotná modifikace cesty algoritmem je přesně determinovaná, avšak právě inicializace původní cesty je stochastického charakteru. Lze tedy metodu

aplikovat na nespočetné množství cest a uchovávat pouze nejlepší dosažené řešení. Jedná se o tzv. Opakovaný (Repeated) Lin-Kernighan. Druhou možností je výsledky dosažené Lin-Kernighanem modifikovat transformací „dvojitého mostu“ (double-bridge) a výsledek modifikace dále upravovat algoritmem za cílem nalezení ještě kratší cesty. Dvojitý most je speciálním typem 4-opt kroku, který však na rozdíl od dříve zmíněných *k*-opt kroků není sekvenční (viz. Ilustrace 7).



Ilustrace 7: Nesekvenční 4-opt. Zdroj: [9]

Mezivýsledky, které jsou modifikovány transformací dvojitého mostu jsou základem tzv. Iterovaného (Iterated) Lin-Kernighanu. Do této varianty lze také zakomponovat princip ze simulovaného žíhání, ve kterém lze nové výsledky přijímat podle parametru teploty – tzv. Zřetězený (Chained) Lin-Kernighan. Chained Lin-Kernighan je skvělým příkladem toho, že stochastické metody z kapitoly 5 najdou při řešení TSP nejlepšího uplatnění v kombinaci s metodou lokálního hledání.

Délky běhů obou verzí algoritmů rostou přibližně $O(N^{2.2})$. LK-H se jeví efektivnější, což se však znatelně projevuje na mnohem delším běhu aplikace.

8.2 Hybridní genetické algoritmy (HGA)

Zavedou-li se do genetického algoritmu další operace s hlubšími znalostmi problému, které by jeho řešení napomohly, mluví se o hybridních genetických algoritmech (HGA).

Obecné genetické algoritmy nacházely mnohem horší cesty než metody lokálního hledání a to navíc během mnohem delší doby. Vzájemná kombinace spolu s vhodně zvolenými operátory může pomoci k uniknutí z lokálního minima.

Nejdůležitější složky HGA:

- výběr modelu GA
- způsob začlenění lokálního hledání do GA
- rovnováha mezi globálním a lokálním hledáním

Mezi nejefektivnější HGA patří především ty se začleněnou Lin-Kernighan heuristikou. HGA, které byly testovány na příkladech o několika tisících vrcholů, jsou následující:

- GA s LK lokálním prohledáváním (Cga-LK)

- strategie asynchronní paralelní genetické optimalizace (Asparagos)
- natural crossover GA (NGA)
- genetické lokální hledání (GLS)
- edge assembly crossover GA (GA-EAX)
- Nguyen HGA

Asparagos pracuje s populacemi o malém počtu jedinců a při výběrů rodičů pro křížení se snaží vyhledávat podobné rodiče - individua ležící v blízkém sousedství. Takovou selekcí se snaží udržet diversitu v populaci. Největší slabinou je aplikace jednoduchého 2-opt algoritmu pro zlepšení trasy vzniklé po křížení a následné mutaci potomka. V porovnání s ostatními verzemi HGA však (hlavně kvůli nepoužití LK heuristiky pro zkrácení trasy) patří k těm pomalejším. I novější Cga-LK nesplňuje požadavky pro rychlejší běh aplikace, byť užívá sofistikovanější LK heuristiku, kde násobnost jejího užití roste s pokročilejší generací.

Ze zbývajících algoritmů má nejlepší vyhlídky pro řešení TSP s velkým počtem měst Nguyen HGA. Řešení ostatních nedosahují takových kvalit, byť délka běhu může být kratší. Největší výhodou Nguyen HGA oproti metodě LK-H je možnost paralelizovat výpočetní proces do více výpočetních jednotek. Heuristice LK-H navíc nepomáhá přílišná složitost. Délka běhu aplikace roste přibližně $O(N^{2.2})$ což brání jeho efektivnímu použití k problémům nad 100 000 měst. Při experimentálních výsledcích lze z tabulky 7 vyčíst, že při paralelním běhu aplikace pracující s Nguyen HGA (10 výpočetních jednotek, PHGA) roste rychlost běhu pro „těžší“ příklady pomaleji. LK-H je rychlejší pouze při řešení příkladů do 5 000 měst. Pokud Nguyen HGA našel optimální řešení, tak jej v několika bězích našel častěji než LK-H. [18]

instance	**metoda**	**optimum**	**nejlepší výsledek**	**∅ výsledek (10 běhů)**	**∅ doba trvání**
pr1002	PHGA	259 045	optimum	optimum	507 s
	LK-H		optimum	optimum	25 s
pcb3038	PHGA	137 694	optimum	optimum	37 m
	LK-H		137 712	137 753	11 m
pla7397	PHGA	23 260 728	optimum	23 260 814	2,5 h
	LK-H		optimum	23 263 996	4,5 h
d15112	PHGA	1 573 084	1 573 183	1 573 330	9,5 h
	LK-H		1 573 168	1 573 273	10 h
pla33810	PHGA	66 005 185	66 062 314	66 067 083	1,8 dne
	LK-H		66 076 272	66 086 687	13,5 dne

Tabulka 7: Srovnání výsledků metod PHGA a LK-H. Zdroj: [18]

Číselný údaj v instanci je roven počtu vrcholů

8.2.1 Nguyen HGA

Cílem Nguyen HGA [18] bylo vyvinout mocný hybridní algoritmus, který by byl schopen nalézt řešení velmi vysokých kvalit pro problém obchodního cestujícího s velkým počtem měst. Uveřejněny byly minimálně dvě verze, lišící se v inicializační metodě a v použitých operátorech křížení a mutace. Pseudokód tohoto hybridního genetického algoritmu rozvíjející základní genetický algoritmus o možnost migrace mezi pod-populacemi je popsán ilustrací 8.

num_subpop	10
subpop_size	50
max_gen	500 000
mutation_rate	0,1
selection bias	1,25
num_iteration	5
max_nonimproved_gen	2 000
migration_interval	500
num_migrant	3

Tabulka 8: Proměnné Nguyen HGA. Zdroj: [18]

```
program HybridGA
begin
 for (sub = 1; sub <= num_subpop; sub++) {
  for (ind = 1; ind <= subpop_size; ind++) {
   Construct_tour(P[sub][ind]);
   Improve_tour(P[sub][ind]);
  }
 }
 best_so_far = Find_best(P);
 num_nonimproved_gen = 0;
 for (gen = 1; gen <= max_gen; gen++) {
  for (sub = 1; sub <= num_subpop; sub++) {
   if (Rand() < mutation_rate) {
    p1 = Linear_select(P[sub], selection_bias);
    c = p1;
    for (iter = 1; iter <= num_iteration; iter++) {
     cc = Mutate(c);
     Improve_tour(cc);
     if (cc.cost < c.cost)
       c = cc;
    }
   } else {
    p1 = Linear_select(P[sub], selection_bias);
    p2 = Linear_select(P[sub], selection_bias);
    c = Crossover(p1, p2);
    Lock_comon_subtours(p1, p2);
    Improve_tour(c);
    Unlock_common_subtour();
    if (p1.cost < p2.cost)
      Swap(p1, p2);
   }
   if (c.cost < p1.cost)
    Replace(c, p1, P[sub]);
  }
  best = Find_best(P);
  if (best < best_so_far) {
   best_so_far = best;
   num_nonimproved_gen++;
  } else {
   if (++num_nonimproved_gen >= max_nonimproved_gen)
    break;
  }
  if ((gen % migration_interval) == 0)
   Migrate(P, num_migrant);
 }
 Report(best_so_far);
end
```

Ilustrace 8: Pseudokód Nguyen HGA. [18]

8.2.1.1 Tok algoritmu

Pojmy uvedené v tabulce 8 reprezentují proměnné, kterými lze kontrolovat chování algoritmu.

Chování podpopulace vychází ze *Steady-State* modelu GA. Jejich počet a množství jejich jedinců definují proměnné num_subpop a subpop_size. Každý jedinec je zkonstruován funkcí Construct_tour a následně hned vylepšen funkcí Improve_tour.

V každé generaci je vytvořen jediný potomek (buď křížením Crossover anebo mutací Mutate závisle na pravděpodobnosti mutation_rate), který je ve stejné generaci ještě upravován již zmíněnou vylepšující funkcí.

Oproti *Steady-State* modelu zde však není nahrazován nejhorší jedinec v populaci za předpokladu, že je potomek kvalitnější. Nahrazován (funkce Swap) je nýbrž horší z dvojice rodičů. Tato změna udržuje větší rozmanitost populace za cenu pomalejší konvergence. Rodiče jsou vybíráni lineárně, na základě pořadí účelové funkce jedince v populaci. Parametrem výpočtu vhodnosti každého jedince pro lineární řazení je hodnota selection_bias.

Algoritmus končí nejpozději po uplynutí generace č. max_gen. Pokud však nedojde během max_noimproved_gen generací ke změně nejlepšího průběžného řešení best_so_far, končí algoritmus dříve.

Jednou za každých migration_interval generací dochází mezi podpopulacemi k cyklické migraci num_migrant počtu jedinců.

8.2.1.2 Heuristiky lokálního hledání

Pro vytvoření jedince funkcí Counstruct_tour bývají trasy v eukleidovském prostoru vytvořeny Borůvkovým algoritmem, pro geografické TSP algoritmem nejbližšího souseda.

Při zavolání funkce Improve_tour je trasa za pomoci 12 kvadraticky nejvhodnějších vrcholů ke každému městu zdokonalována heuristikou Lin-Kernighan. Hloubka prohledávání je omezena na hodnotu 100. Původní verze pracuje s rychlejší obecnou verzí LK za užití struktury dvouvrstvého seznamu, novější verze pracuje s 5-opt úpravami známými z LK-H.

8.2.1.3 Genetické operátory

Při křížení dvou rodičů vedoucímu ke vzniku jediného potomka byl dříve používaný Greedy Subtour Crossover (GSX2) nahrazen variantou Maximal Preservative Crossover (MPX3). Jak GSX, tak MPX patří do kategorie operátorů, které nevyužívají lokálních informací (např. délka hran) pro vytváření jedinců. GSX operátor pomáhal rychlé konvergenci algoritmu a pracoval dobře s problémy menších rozloh. Kvůli tomu, že s jeho použitím docházelo k uváznutí v lokálním minimu, byl nahrazen alternativou v podobě operátoru MPX3. Ten dvakrát až čtyřikrát zpomaluje chod programu, ale dosahuje kvalitnějších řešení.

Operátor mutace provádí nesekvenční 4-opt operaci (double-bridge), konkrétně verzí náhodné chůze o počtu kroků k=250.

9 Implementace HGA

Z algoritmů uvedených v předchozích kapitolách patří mezi nejvýkonnější heuristiku suverénně Lin-Kernighan. Dosahuje velmi kvalitních výsledků v krátkém čase. Metody LK-H a Nguyen HGA by se daly nazvat nadstavbou této heuristiky. Z nich byla vybrána varianta hybridního genetického algoritmu a to z následujících důvodů: Na rozdíl od LK-H zde existuje možnost paralelismu, doba výpočtu roste s vyšším počtem měst pomaleji a dle testů z [18] nachází optimální řešení častěji než LK-H. Implementovaný model tedy vychází především z Nguyen HGA, s tím rozdílem, že se bude snažit nacházet co nejkvalitnější řešení v co nejkratším čase.

9.1 Model GA

Pokud se v populaci nachází jedinci s širokým rozptylem ohodnocení, může vhodnost na základě ohodnocení způsobit předčasnou konvergenci.

V případě vhodnosti na základě pořadí vede vysoký výběrový koeficient k zaměření selekce především na nejsilnější jedince v populaci. Bohužel se tak rychleji ztrácí rozmanitost populace. Při volbě příliš malého koeficientu (nebo při velkém počtu jedinců v populaci) jsou kvalitnější jedinci upřednostňováni méně a dochází k robustnějšímu prohledávání, což zpomaluje běh algoritmu. Rozmanitost populace tím však netrpí.

Každý způsob selekce má své nedostatky i své výhody, v [22] byly modely:

1) *Steady-State* s vhodností dle ohodnocení jedince
2) *Steady-State* s vhodností dle pořadí (tzv. *GENITOR*)
3) *Generational* s vhodností dle ohodnocení jedince (tzv. *Genesis*)
4) *Generational* s vhodností dle pořadí

testovány pěti různými funkcemi, ve kterých nejvíce vynikal model *GENITOR* [22].

Nguyen HGA je postaven na stejném modelu. Vhodnosti jsou tedy v implementovaném programu počítány na základě kvalitativního pořadí jedinců v populaci. Během generace dochází k výběru jediného jedince, který na základě svých kvalit může nahradit jedině svého rodiče.

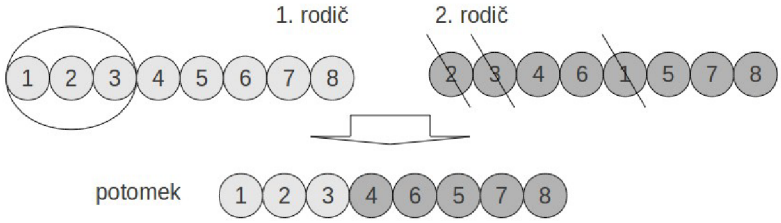
9.2 Operátory křížení

Na základě kvalitních výsledků Nguyen HGA byly implementovány operátory křížení, které tento model obsahuje. Jedná se o MPX2 a alternativní verzi GSX. Doplňujícím operátorem je Sub-tour Recombination Crossover (SRX).

9.2.1 Maximal Preservative Crossover (MPX)

Operátor maximálního zachování byl představen roku 1988 H. Mühlenbeinem. [14] Potomek ze dvou rodičů vznikne následovným křížením:

1) Z 1. rodiče je náhodně vybrána podtrasa o délce *K*, která je vložena do potomka.
2) Z 2. rodiče jsou vyjmuty všechny vrcholy, které jsou obsaženy v potomku a zbytek trasy je do potomka zkopírován.



Ilustrace 9: Průběh křížení operátorem MPX.

Činnost křížení operátorem MPX je ilustrována ilustrací 9. Výhodou operátoru je, že ničí pouze omezený počet hran, jejichž maximální počet je roven *K*. S četností užití operátorů v populaci jedinců se průměrný počet rušících hran snižuje. Délka *K* vybraného podřetězce by dle autora měla být v intervalu <10,*N*/2>.

9.2.1.1 Alternativní varianta (MPX2)

Rozdílem oproti původní verzi je hledání první rozdílné hrany mezi rodiči v 2. kroku, od které se následný zbytek cesty zkopíruje do potomka. [8] Takovou změnou je garantováno, že v případě existujícího rozdílu mezi rodiči, je tato diference obsažena i v potomku, který pak nebude totožný s žádným rodičem.

Například hybridní genetický algoritmus Asparagos používal verzi MPX2 a vybíral z původního řetězce podtrasu o délce *K* z intervalu <*N*/10, *N*/3>. V implementaci vlastního HGA je velikost dána poměrem *N*/5.

9.2.2 Greedy Subtour Crossover (GSX)

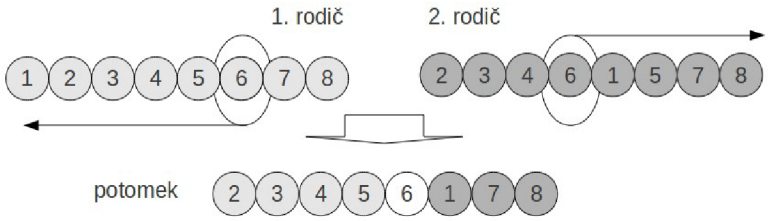
Potomek ze dvou rodičů vznikne následovným křížením: [7]

1) Z rodičů je náhodně vybráno město, které je vloženo do potomka. Do proměnné *P1* je uložena pozice města v prvním potomku. Obdobně je nastavena hodnota proměnné *P2*.
2) Pokud město v prvním rodiči nacházející se na pozici *P1*-1 není obsaženo v potomku, je přidáno do potomka před jeho první vrchol a *P1* = *P1*-1.
3) Pokud město v druhém rodiči nacházející se na pozici *P2*+1 není obsaženo v potomku, je přidáno do potomka za jeho poslední vrchol a *P2* = P2+1.

4) Opakuj 2. a 3. krok. Pokud v jednom z kroků nelze splnit zadanou podmínku, krok dále ignoruj.
5) Opakuj 4. krok dokud lze splnit alespoň jednu podmínku z 2. či 3. kroku.
6) Pokud existují vrcholy, které nejsou obsaženy v potomku, nahodile je k němu přidej za jeho poslední vrchol.

9.2.2.1 Alternativní varianta (altGSX)

Náhodné přidávání kroků ke konci algoritmu nemusí být vždy výhodné. Zamezit tomu lze následujícím způsobem. 2. a 3. krok lze provádět, i když se narazí na vrchol již obsažený v potomku. V tom případě pouze nedojde k opětovnému přidání duplikátu. Algoritmus by pokračoval až do okamžiku, ve kterém by potomek byl naplněn všemi vrcholy, viz ilustrace 10. V implementovaném programu je použit operátor GSX této verze.



Ilustrace 10: Průběh křížení operátorem AltGSX.

9.2.3 Sub-tour Recombination Crossover (SRX)

Operátor rozděluje rodičovské chromozomy do několika podtras, jejichž vzájemnou kombinací vznikne jeden výsledný jedinec.

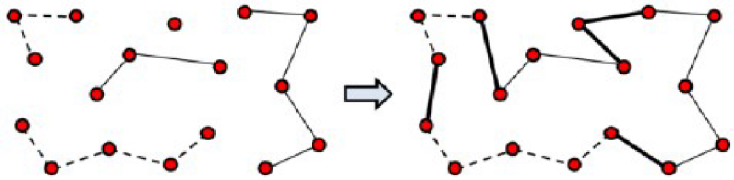
Věří se, že dědění dlouhých podtras z potomků brání k vytvoření špatného potomstva. Hlavní myšlenkou operátoru SRX je snaha dědit poněkud kratší úseky. Autoři operátoru vycházejí z toho, že čím delší trasa zděděná operátorem je, tím méně pravděpodobné zlepšení dalším krokem Lin-Kernighan je. [13]

Operátor SRX dědí z obou rodičů podtrasy následovným postupem:

1) Vybraný rodič pro extrakci je 1. rodič.
2) Je náhodně vybráno startovní město z vybraného rodiče, které se nenachází v potomku. Město tvoří novou podtrasu.
3) Oběma směry je podtrasa prodlužována, dokud se nenarazí na vrchol již umístěný v potomku nebo dokud délka podtrasy je rovna hodnotě *K*.
4) Vybraný rodič pro extrakci je 2. rodič.
5) Opakuje se 2. krok.
6) 1. - 5. kroky jsou opakovány do okamžiku, kdy je potomek naplněn všemi vstupními vrcholy v podobě podtras.

7) Páry koncových měst podtras extrahované z rozdílných rodičů jsou spojeny a utvoří novou cestu.

Operátor byl do implementace přidán na základě nejkvalitnějších výsledků v tabulce 9 a to i přes skutečnost, že test byl prováděn během příliš dlouhého časového rozmezí. [13] Test uvedený v tabulce 9 se skládal z běhu jediné populace s pouze 30 jedinci bez užití operátoru mutace. Použitý hybridní genetický algoritmus dále používal 8 kvadraticky nejbližších sousedů a hloubku prohledávání o hodnotě 25. Jedinci byli inicializováni modifikovaným Borůvkovým algoritmem. Vznik potomku křížením je znázorněn na ilustraci 11.



Ilustrace 11: Průběh křížení operátorem SRX. Zdroj: [13]

Implementovaný operátor se na rozdíl od popsané verze liší v 7. kroku. Operátor se nesnaží skládat podtrasy od rozdílných rodičů, ale snaží se trasy sloučit bez ohledu na jejich původ.

příklad	it16862				vm22775			
operátor křížení	SRX	MPX3	GSX2	ERX6	SRX	MPX3	GSX2	ERX6
Ø kvalita (%)	0,173	1,235	1,566	1,117	0,204	1,217	1,781	1,006
Ø doba běhu (s)	96 337	116 145	125 236	107 794	166 895	227 173	238 013	236 781

Tabulka 9: Test operátorů křížení. Zdroj: [13]

Kvalita = (vypočtená_trasa – nejkratší_zaznamenaná) / nejkratší_zaznamenaná * 100 (%), číselný údaj v instanci je roven počtu vrcholů.

9.3 Optimalizační heuristika

K inicializaci jedinců jsou použity algoritmy uvedené v 6. kapitole. Opravnou heuristikou byla zvolena Lin-Kernighan, varianta LK-ABCC. Oproti LK-H pracuje s jednodušeji určeným okolím kandidátních vrcholů pro případné *k*-opt transformace. Na základě testů z [18] se však jedná o variantu přibližně třikrát rychlejší. Kopy zmíněné v kapitole 8.1. jsou použity jako mutující funkce. Funkce vykonávající tyto uvedené operace jsou převzaty z knihovny Concorde [21] a modifikovány pro účely HGA.

9.4 Datové struktury

Od nejčastějších operací s daty a také od reprezentace cesty se odvíjí výběr datových struktur. Binární reprezentace dat pro problém obchodního cestujícího s velkým počtem měst se z praktických důvodů neprosadila. [12] Pracovalo se především se sousedskou reprezentací (adjacency), ordinální reprezentací a reprezentací cesty (path). Poslední zmíněná se zdá být nejpřirozenější a ukázala se být pro problém i nejvhodnější. S takovou reprezentací pracují i všechny zmíněné operátory křížení.

V reprezentaci cesty musí být program schopen rychle nalézt souřadnice požadovaného města, ale i jeho předchozího a následujícího souseda na trase. Musí být také schopen rychle provádět inverze podtras, jak vyžaduje algoritmus Lin-Kernighan.

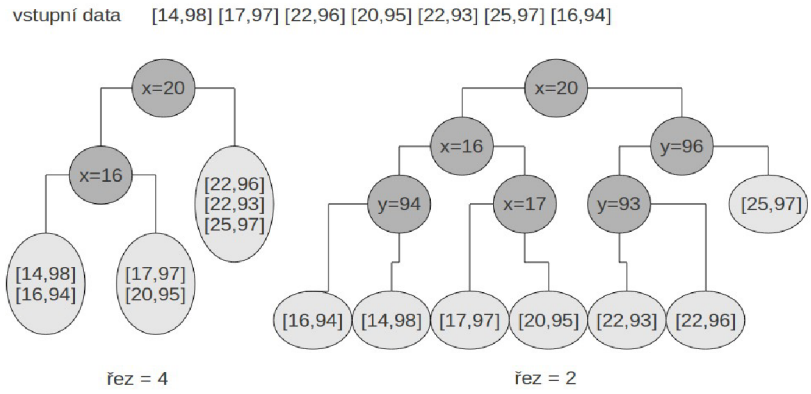
Následující datové struktury výrazně krátí výpočetní dobu běhu aplikace hybridního genetického algoritmu. Jejich implementace je součástí knihovny Concorde. [21]

9.4.1 *K*-dimensionální strom

K-dimenzionální strom (*kd*-tree) reprezentuje množinu *N* bodů v *k*-rozměrném prostoru. Jedná se o binární prohledávací strom umožňující značné urychlení operace hledání nejbližšího souseda k danému bodu. Takové operace jsou při řešení TSP konány cestu vytvářejícími algoritmy, operátorem mutace i heuristikou Lin-Kernighan. Datová struktura zefektivňuje vyhledávání dělením prohledávaného prostoru na podprostory. U eukleidovského dvourozměrného prostoru jsou data organizována do podprostorů podle os *x* a *y*. [4]

U obecného *kd*-stromu je list vázán pouze na jedno město. Lze však definovat koeficient řezu, který stanoví pod jaký počet zbývajících měst v podmnožině se již dále nebude uzel větvit na listy. Standardní hodnotou je 2, v případech pro velký počet měst je tento koeficient navyšován, což vede ke snížení počtu uzlů a tudíž ke snížení paměťových požadavků. Při hledání nejbližšího souseda se v takových případech musí prohledávat i města v jednotlivých listech.

Pro sestavení *kd*-stromu je ze seřazeného náhodného vzorku vybrán medián. Množina je rozdělena dle osy s největším rozptylem, pokud počet jejich prvků není menší než řezný koeficient. Pokud došlo k rozdělení prostoru, algoritmus je aplikován na nově vzniklé podprostory.

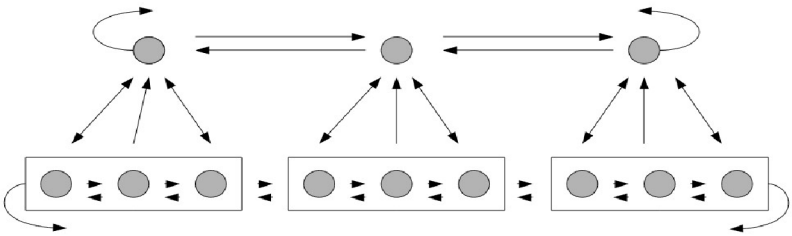


Ilustrace 12: Příklady kd-stromů.

Příklad vytvořeného *kd*-stromu je na ilustraci 12. Horní výpočetní složitost stavby stromu je $O(KN + N\log N)$ [5]. Ke každému městu je přidělen list *kd*-stromu. Při vyhledávání nejbližšího souseda se vychází směrem odspoda-nahoru, při kterém jsou prozkoumáváni otcové počátečního uzlu a uzly okolní. Takové vyhledávání je velmi rychlé, s horní hranicí složitosti pouze $O(1)$.

9.4.2 Dvouvrstvý seznam

Dvouvrstvý seznam (two-layered list, two-level tree) bývá používán pro reprezentaci cest TSP. Při porovnání s dalšími strukturami jako jsou jednoduché pole, jednovrstvé seznamy a binárními stromy, se jevila struktura dvouvrstvého stromu nejefektivnější pro operace algoritmu Lin-Kernighan nad vstupními uzly. [2] Nejčastější činností nad daty u algoritmu LK je inverze sekvencí cest a hledání následujícího či předcházejícího města na trase. Dvouvrstvý seznam umožňuje provádět tyto operace se složitostí maximálně $O(\sqrt{(N)})$, kde N je počet vrcholů. To je umožněno dělením trasy do $\sqrt{(N)}$ bloků. Bloky jsou reprezentovány dvousměrnými seznamy (doubly linked lists). Ke každému bloku je asociován rodičovský uzel. Rodičovské uzly jsou rovněž uspořádány dvousměrným seznamem, viz. ilustrace 13.



Ilustrace 13: Dvouvrstvý seznam.

Prvky bloků na spodním seznamu na ilustraci 13 jsou jednotlivé vrcholy reprezentující města. Rodičovské uzly obsahují tzv. inverzní bit určující orientaci bloku. Jednotlivé bloky seznamu

obsahují ukazatele na předchozí i následující bloky v cyklickém seznamu. Rovněž k městům na konci bloku jsou asociovány ukazatele na předchozí (resp. následující) město nacházející se v sousedním bloku.

Hledání následujícího či předcházejícího města na trase probíhá vyhledáním města v blocích a dle inverzního bitu v rodičovském uzlu se rozhodne, v jakém směru sousedí hledaný vrchol od původního. K provedení inverze trati (flip operace) mezi body *a* a *b* se nejdříve vyhledá bod *a*. Pokud se v bloku nenachází na první pozici (popř. poslední při inverzi), je část bloku předcházející vrcholu *a* z bloku odstraněna a připojena na konec předcházejícího bloku. Pokud má po této operaci blok velikost větší než $2\sqrt{(N)}$ je blok rozdvojen. Podobně proběhne operace pro vrchol *b*. Pro inverzi podtrasy jsou invertovány hodnoty rodičovských uzlů operujícími nad bloky, jejichž vrcholy jsou součástí takové podtrasy.

9.5 Použití programu

Hybridní genetický algoritmus byl implementován objektově orientovaným jazykem C++. Spolu s vybranými moduly z Concorde knihovny [21] se implementace skládá ze tří tříd a hlavního programu, v kterém jsou objekty takových tříd vytvářeny. Jejich vzájemná interakce vede k aproximačnímu řešení problému obchodního cestujícího o velkém počtu měst.

Jednotlivá řešení jsou ukládána v objektech třídy cChromosome. Informace o cestě obsahuje třída v celočíselné permutaci o velikosti *N*. Objekty této třídy uchovávají informace o ohodnocení cesty a hodnotu vhodnosti aktualizující se v každé generaci. Třída také obsahuje pomocné struktury pro efektivní práci heuristiky Lin-Kernighan. Třída cChromosome je schopna mj. kalkulovat ohodnocení trasy na základě permutace a vstupních dat, permutaci inicializovat, křížit se, mutovat a vykonávat iterace heuristiky Lin-Kernighan.

Třída cPopulation je schopna dynamicky vytvářet objekty třídy cChromosome, pro které také uchovává souhrnné informace o jejich počtu a množství podpopulací, ve kterých se sdružují. Po vytvoření objektů třídy cChromosome má také schopnost jedince inicializovat. Třída dále obsahuje metodu pro řízení generačního toku. V něm jsou na základě metody určující výběr jedince pro reprodukci volány jednotlivé metody samotných chromozomů pro mutaci či křížení. Na konci každé generace objekt řadí jedince podle jejich ohodnocení a znovu počítá jejich fitness. Po ukončení programu jsou objektem třídy cPopulation volány metody pro uložení aktuální populace a nejlepšího jedince v ní. Za předpokladu, že je v programu zvolena možnost načtení populace, inicializační cyklus se neuskuteční a data jsou načtena z uložené konfigurace populace.

Po spuštění programu je vytvořen jediný objekt cPopulation spolu s třídou cFileData načítající a organizující vstupní data. Po načtení dat tiskne program každé nové zlepšení nejlepší

vypočtené trasy a uživatele informuje o každé uplynulé tisící generaci. Po splnění ukončovacích podmínek se program ukončí. Rovněž lze program předčasně ukončit stisknutím kombinace kláves CTRL+C. Po obou možnostech ukončení aplikace se aktuální řešení i populace uloží pro další možná použití.

```
pouziti: ./hga [parametr z nabidky] eucl2D_data.tsp
 -r #  inicializovani pseudonahodneho generatoru cislem
 -g #  maximalni pocet generaci [500000]
 -h #  maximalni doba behu aplikace v hodinach [1.5]
 -o f.tour  ulozit vyslednou cestu
 -l f.pop   nacist populaci (nepovinne argumenty budou ignorovany)
---------------------------------------- lin-kernighan --
 -I #  inicializacni metoda [2]
       (0-nahodne, 1-NNeigh, 2-Greedy, 3-Boruvka, 4-QBoruvka)
 -q #  # kvadr. nejblizších vrcholu ve vyhledavacim grafu [12]
 -a #  # nejblizších vrcholu ve vyhledavacim grafu [0]
---------------------------------- geneticky algoritmus --
 -c #  pocet podpopulaci [3]
 -p #  velikost populace <5,x) [30]
 -s #  parametr linearni selekce z intervalu <1.0, 2.0> [1.25]
 -m #  pravdepodobnost mutace pred krizenim <0.0, 1.0> [0.9]
 -M #  mutace [3]
       (0-nahodne, 1-Geometric, 2-Close, 3-Random_Walk)
 -i #  pocet iteraci mutace [5]
 -X #  operator krizeni [0]
       (0-MPX2, 1-AltGSX, 2-SRX)
 -t #  pocet migrujicich chromozomu (0, "-p") [2]
 -f #  frekvence migrace v generacich [1000]
```

Ilustrace 14: Náhled implementovaného programu bez vstupních parametrů.

Činnost programu reprezentuje stejně jako Nguyen HGA ilustrace 8. Možné ukončení programu po daném počtu neúspěšných generací zde bylo nahrazeno nastavením maximální doby běhu aplikace. Samotný běh programu je možno nastavit dle parametrů zobrazených na ilustraci 14, na které se vyskytuje nápověda implementovaného programu. Jediným povinným parametrem je cesta ke instanci TSP uložené v eukleidovském 2D formátu. U ostatních nepovinných parametrů jsou v hranatých závorkách uvedeny přednastavené hodnoty.

Přijímaný vstupní formát je shodný se vzorky z knihovny TSPLIB [6] obsahujícími v hlavičce položku EDGE_WEIGHT_TYPE : EUC_2D. Výsledkem programu je soubor s příponou tour. Součástí výsledného souboru je nejkratší nalezená permutaci měst, ohodnocení výsledné trasy a pojmenování řešeného problému. Formát je rovněž shodný se soubory z knihovny TSPLIB, které obsahují řešení tras.

10 Výsledky testů

Pro všechny testy byla shodná velikost kvadraticky nejbližších sousedů LK (12), hloubka prohledávání LK (25) a random-walk mutace. Tyto hodnoty jsou standardně nastaveny již v knihovně Concorde [21], odkud byla Lin-Kernighan heuristika použita. Rovněž algoritmy pro inicializaci cest, mutace a datové struktury byly modifikovány ze stejné knihovny pro účely hybridního genetického algoritmu. Maximální doba běhu aplikace pro všechny testy byla jednotná - 1,5 hodin.

Při nastavení konfigurace problému je třeba brát v úvahu rozlohu instance, s kterou má hybridní algoritmus pracovat. Během testů na 5 příkladech o počtech cca: 1, 3, 6, 10 a 13 tisíc měst, lze s algoritmem dojít k velmi kvalitním výsledkům. Poznatky k uvedeným pěti testům v pořadí, v kterém testy probíhaly jsou následující:

1) **rl5934** (optimum: 556 045)

   Algoritmus našel optimální řešení ve všech bězích o jediné populaci s 30 jedinci. Při 30 bězích samotné LK heuristiky nebyl program schopen najít optimální řešení a nejlepší běh zaznamenal velikost o 1 000 jednotek delší než optimální trasa. Uvedeného výsledku dosáhl během 40 minut, po kterých již nebyl schopen nalezení lepšího řešení.

   Zato implementovanému HGA se podařilo najít optimální trasu během deseti minut.

2) **pr1002** (optimum: 259 045)

   Předešlá konfigurace se uplatnila i pro řešení této menší instance. Problém pr1002 byl do optima vyřešen během půlminuty.

3) **gr9882** (optimum: 300 899)

   Při použití stejné konfigurace program relativně brzy uvázl na hodnotě (301 878) vzdálené tisíc kilometrů od nejkratší nalezené cestě po řeckých městech. Při zapojení více populací a migračního toku mezi nimi, se úroveň řešení dostala na velmi kvalitní hodnotu 300 926, která na nejlepší výsledek ztrácí pouze 27 km. Lepšího řešení dosaženo v daném časovém termínu nebylo.

4) **usa13509** (optimum: 19 982 859)

   Při hledání nejkratší trasy po amerických městech se na rozdíl od „řecké konfigurace“ ujal model předchozích jednodušších problémů. Program totiž hledá řešení relativně dlouho a hranice 90 minut mu zřejmě nestačí. Proto se ujalo řešení pouze jediné populace. U jedné populace však existuje menší pravděpodobnost, že by pokračování algoritmu vedlo k lepším výsledkům, než by tomu bylo ve více populačním případě.

5) **pcb3038** (optimum: 137 694)

Pro uvedené příklady o počtu měst kolem 1 i 6 tisíc byla optimální trasa nalezena takřka bez problému. Pokud jsou v instanci města uskupena v přílišných shlucích, může být nalezení nejkratší cesty složitější i pro relativně malé příklady. Takovým se dle testování jeví být i třítisícová instance pcb3038. Dokonce ani zapojením více populací ve vyhledávání nebyl HGA schopen najít optimální řešení. K nejkratší nalezené cestě v nejlepších případech chyběly 4 jednotky a konfigurace hledající lepší trasu zatím nalezena nebyla.

Výsledky nejlepších měření jsou umístěny v tabulce 10 spolu s konfigurací, která k nim vedla. Uvedena je také doba, za kterou bylo řešení nalezeno.

instance	pr1002	pcb3038	rl5934	gr9882	usa13509
optimum	259 045	137 694	556 045	300 899	19 982 859
jedinců v populaci	30	20	30	30	20
populací	1	3	1	3	3
inicializováno	QuickBorůvka	Greedy	QuickBorůvka	Greedy	Greedy
časnost migrace (generace)	-	-	-	1 000	1 000
migrantů	-	-	-	2	2
po 30 minutách	259 045	137 698	556 045	301 283	20008 670
po 60 minutách	-	-	-	300 988	19991 902
po 90 minutách	-	-	-	300 926	19990 251
nejvyšší dosažená kvalita (%)	0	0,003	0	0,009	0,0370
doba běhu (minut)	0,5	9	9	90	90

Tabulka 10: Výsledky implementovaného HGA.

Kvalita = (vypočtená_trasa – nejkratší_zaznamenaná) / nejkratší_zaznamenaná * 100 (%), číselný údaj v instanci je roven počtu vrcholů. Testováno na 2,1GHz PC s 4GB RAM. Program byl přeložen g++ překladačem a běhy probíhaly na OS Ubuntu 10.10 - Maverick Meerkat.

Krom již zmíněných parametrů, které byly při testování shodné, se v nejúspěšnějších případech opakovala následující nastavení: pravděpodobnost mutace 0,9 a parametr lineární selekce 1,25. Suverénním se stal operátor křížení MPX2, díky kterému bylo dosaženo všech výsledků uvedených v tabulce 10. Pro práci s omezeným časovým úsekem ostatní operátory zcela převyšoval.

Hodnota pravděpodobnosti mutace byla značně navýšena od hodnoty užívané v Nguyen HGA z důvodu nalezení kvalitního řešení v co nejkratším čase. Při dostatku výpočetního času je doporučováno tuto hodnotu snížit. Nižší hodnota pravděpodobně povede k většímu udržení diversity populace, což by mělo přinášet i kvalitnější výsledky.

11 Závěr

Cílem této práce bylo nalezení efektivního řešení problému obchodního cestujícího pro velký počet měst. Po seznámení s problematikou byly představeny exaktní i heuristické metody zabývající se problémem. Druhá kategorie metod více splňovala podmínky zadání, tudíž byla rozebrána detailněji.

Samotné stochastické metody se neukázaly být vhodným prostředkem ke splnění vytyčeného cíle. I jednoduché cestu vytvářející algoritmy založené na hledání minimální kostry grafu vedly ke kvalitnějším výsledkům než simulované žíhání nebo metoda zakázaného prohledání. Jen ACO a Inver-over operátor pro genetické algoritmy byly schopny dosažení lepších výsledků. Nejkratší cesty dokázaly generovat metody pracující se znalostí svého okolí, konkrétně heuristika Lin-Kernighan.

Ukázalo se, že složením stochastických metod, cestu vytvářejících metod a lokálního hledání lze sestavit mocný nástroj v podobě hybridního genetického algoritmu schopného kvalitních výsledků v krátkém čase. Práce uvádí základní principy všech důležitých komponent a známé optimalizace vedoucí ke snížení časové náročnosti výsledného programu.

Byla vytvořena softwarová implementace hybridního genetického algoritmu pracující s heuristikou Lin-Kernighan. Podařilo se nalézt vhodnou konfiguraci programu, které vedla k optimálním a optimu blížícím se výsledkům. Výběr modelu genetického algoritmu a jeho dílčích součástí se zdařil.

Chod programu by bylo v budoucnosti možné urychlit jeho optimalizací pro více výpočetních jednotek, protože je schopen paralelního běhu.

Literatura

[1] *Adaptivebox.net* [online]. [cit. 20105-11]. Source Code Library : Travelling Salesman Problem (TSP). Dostupné z WWW: <http://www.adaptivebox.net/CILib/code/tspcodes_link.html>.

[2] APPLEGATE, David L., et al. *The traveling salesman problem : A computational study*. Princeton, New Jersey : Princeton University Press, 2006. 593 s. ISBN 978-0-691-12993-8.

[3] APPLEGATE, David; COOK, William; ROHE, André. Chained Lin-Kernighan for Large Traveling Salesman Problems. *INFORMS JOURNAL ON COMPUTING* [online]. Winter 2003, Vol. 15, No. 1, [cit. 2011-05-11], s. 82-92. Dostupný z WWW: <http://www2.isye.gatech.edu/~wcook/papers/clk_ijoc.pdf>.

[4] BENTLEY, Jon Jouis. Fast Algorithms for Geometric Traveling Salesman Problems. *ORSA JOURNAL ON COMPUTING* [online]. Fall 1992, Vol. 4, No. 4, [cit. 2011-05-11], s. 387-411. Dostupný z WWW: <http://joc.journal.informs.org/cgi/content/abstract/4/4/387>.

[5] BENTLEY, Jon Louis. K-d trees for semidynamic point sets. In *Proceedings of the sixth annual symposium on Computational geometry* [online]. New York, NY, USA : ACM, 1990 [cit. 2011-05-14], s. 187-197. Dostupné z WWW: <http://doi.acm.org/10.1145/98524.98564>. ISBN 0-89791-362-0.

[6] *Comopt.ifi.uni-heidelberg.de* [online]. 2008 [cit. 2011-05-11]. TSPLIB. Dostupné z WWW: <http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/>.

[7] *Ethoughts.brinkster.net* [online]. 2011 [cit. 2011-05-14]. TRAVELING SALES-MAN PROBLEM USING GENETIC ALGORITHMS. Dostupné z WWW: <http://ethoughts.brinkster.net/my/tsp/tsp.html>.

[8] GORGES-SCHLEUTER, Martina. *Asparagos96 and the Traveling Salesman Problem*. [online]. [cit. 2011-05-11]. Dostupný z WWW: <http://alumnos.elo.utfsm.cl/~fmontero/referencias/get29PDF.pdf>.

[9] HELSGAUN, Keld. An effective implementation of the Lin-Kernighan traveling salesman heuristic. *European Journal of Operational Research* [online]. 1 October 2000, Volume 126, Issue 1, [cit. 2011-05-11], s. 106-130. Dostupný z WWW: <http://www.sciencedirect.com>.

[10] HORDĚJČUK, Vojtěch. *Voho.cz* [online]. 2011 [cit. 2011-05-11]. Genetické algoritmy. Dostupné z WWW: <http://voho.cz/wiki/geneticke-algoritmy/>.

[11] HRUŠKA, Michal. *Problém obchodního cestujícího - paralelní řešení na SMP (OpenMP)*. Brno, 2008. 44 s. Bakalářská práce. VUT Brno.

[12] HYNEK, Josef. *Genetické algoritmy a genetické programování*. Praha : Grada, 2008. 182 s. ISBN 978-80-247-2695-3.

[13] KURODA, Masafumi, et al. Development of a novel crossover of hybrid genetic algorithms for large-scale traveling salesman problems. *Artificial Life and Robotics* [online]. 2010-12-01, Volume 15, Number 4, [cit. 2011-05-14], s. 547-550. Dostupný z WWW: <http://www.springerlink.com/content/h4804r46qx156327/fulltext.pdf>. ISSN 1433-5298.

[14] LARRAÑAGA, Pedro, et al. Genetic Algorithms for the Travelling Salesman Problem : A Review of Representations and Operators. *Artificial Intelligence Review* [online]. April 1999, Volume 13, Number 2, [cit. 2011-05-14], s. 129-170. Dostupný z WWW: <http://www.dca.fee.unicamp.br/~gomide/courses/EA072/artigos/Genetic_Algorithm_TSPR_eview_Larranaga_1999.pdf>. ISSN 0269-2821.

[15] MATOUŠEK, Jiří; NEŠETŘIL, Jaroslav. *Kapitoly z diskrétní matematiky*. Vyd. 2., opr. Praha : Karolinum, 2000. 377 s. ISBN 8024600846.

[16] MICHALEWICZ, Zbigniew; FOGEL, David B. *How to solve it : modern heuristics*. Berlin : Springer, 2000. 467 s. ISBN 3540660615.

[17] NĚMEC, Miloš. *Milosnemec.cz* [online]. 2003-2011 [cit. 2011-05-11]. Mravenčí kolonie: Zevrubný popis ACO algoritmu pro řešení TSP. Dostupné z WWW: <http://www.milosnemec.cz/clanek.php?id=76>.

[18] NGUYEN, Hung Dinh, et al. Implementation of an Effective Hybrid GA for Large-Scale Traveling Salesman Problems. In *IEEE Transactions on Systems, Man, and Cybernetics* [online]. Part B . 2007, 37, 1, [cit. 2011-05-11]. Dostupný z WWW: <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4067083>.

[19] POHLHEIM, Hartmut. *Geatbx.com* [online]. December 2006 [cit. 2011-05-11]. Evolutionary Algorithms 3 Selection. Dostupné z WWW: <http://www.geatbx.com/docu/algindex-02.html>.

[20] TAO, Guo; MICHALEWICZ, Zbigniew. Parallel Problem Solving from Nature. In P*roceedings of the 5th International Conference on Parallel Problem Solving from Nature*. A. E.Eiben, Thomas Bäck, Marc Schoenauer, Hans-Paul Schwefel [online]. London, UK : Springer-Verlag, 1998 [cit. 2011-05-11]. Inver-over operator for the TSP, s. 803 - 812. Dostupné z WWW: <http://portal.acm.org/citation.cfm?id=645824.668606>. ISBN 3-540-65078-4.

[21] *Tsp.gatech.edu* [online]. April 5, 2011 [cit. 2011-05-11]. World TSP. Dostupné z WWW: <http://www.tsp.gatech.edu/>.

[22] WHITLEY, Darrell. The GENITOR algorithm and selection pressure: why rank-based allocation of reproductive trials is best. In *Proceedings of the third international conference on Genetic algorithms* [online]. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1989 [cit. 2011-05-11], s. 116 - 121. Dostupné z WWW: <http://portal.acm.org/citation.cfm?id=93126.93169>. ISBN 1-55860-006-3.

[23] ZELÍNKA, Ivan, et al. *Evoluční výpočetní techniky - principy a aplikace*. 1. vyd. Praha : Nakladatelství BEN-Technická literatura, 2009. 536 s. ISBN 978-80-7300-218-340.

Seznam příloh

Příloha 1. DVD obsahující technickou zprávu v elektronické podobě, zdrojový kód programu, testovací příklady, záznamy nejúspěšnějších testů, dokumentaci