Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Študijný odbor: INFORMATIKA

Peter Ledňa

EVOLUČNÉ ALGORITMY

Algoritmus simulovaného žíhania pre úlohu obchodného cestujúceho na ortogonálnej štvorcovej mriežke

Zadanie

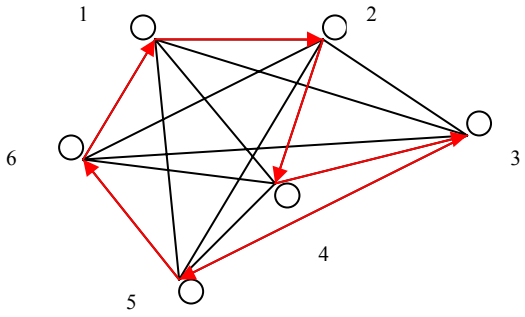
Použite oba prístupy na kódovanie cesty, tak priamo pomocou permutácie, ako aj pomocou binárneho reťazca (pozri cvičenie 7.1 z kapitoly Kombinatoriálne optimalizačné problémy). Porovnajte efektívnosť týchto dvoch rôznych prístupov.

Úvod do problematiky

Úloha obchodného cestujúceho sa dá naformulovať nasledovne:
Majme graf ohodnotený G, ktorý má **N** vrcholov a je úplný(existuje spojnica z každého mesta do každého). Hrany grafu sú ohodnotené, teda vzdialenosť medzi vrcholmi **i, j** je hodnotou hrany medzi nimi. Ak by v grafe neexistovala spojnica medzi vrcholmi **i,j**, tak dĺžku cesty medzi oboma vrcholmi dodefinujeme ako najkratšiu cestu, ktorá ich spája a vedie cez iné vrcholy. Úlohou obchodného cestujúceho je navštíviť každé mesto 1 krát, a vrátiť sa späť do mesta, z ktorého začal (reps. nájsť Hamiltonovskú kružnicu).

Na určenie počtu permutácií miest, v ktorom ich môže obchodný cestujúci navštíviť slúži nasledujúca úvaha:
Máme N miest. Obchodný cestujúci sa nachádza v jednom z nich – vo východisku. Ako prvé mesto, ktoré navštívi si môže zvoliť N-1 miest, keď je v druhom meste, tak si ako tretie vyberá z N-2 miest. Z toho vyplýva, že počet permutácií je (N-1)! . Relevantných permutácií je však iba $\frac{(N-1)!}{2}$, lebo nezáleží, či sa začalo v prvom meste alebo poslednom a išlo sa opačným smerom.
Celkovú dĺžku cesty obchodného cestujúceho vypočítame ako súčet dĺžky ciest medzi mestami, ktoré v danom poradí navštívil. Teda ak funkcia **d(i,j)** vráti dĺžku medzi mestami **i** a **j**, potom celková dĺžka cesty (hodnoty permutácie P) je:

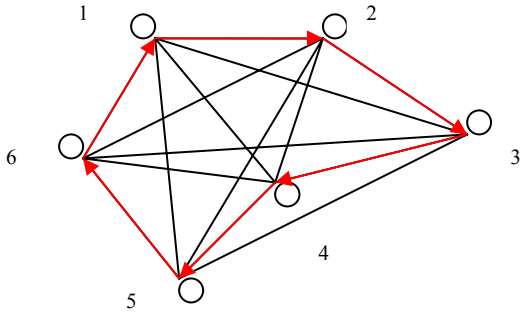
$$f(P) = d(p_1, p_N) + \sum_{i=2}^{N} d(p_{i-1}, p_i)$$



Obr.4 cesta obchodného

permutácia, ktorá reprezentuje cestu je : P(1,2,4,3,5,6)
úlohou optimalizácie je nájsť takú uzavretú cestu medzi mestami, aby bola čo najkratšia. Teda nájsť takú permutáciu miest, aby platilo: $f_{opt} = f(P_{opt})$.

Na problém obchodného cestujúceho existuje mnoho heuristických metód. Niektoré z nich nájdu riešenie väčšinou iba suboptimálne riešenie. Jednou z nich je napr. algoritmus založený na tzv. vonkajšej ceste **[2]**. Tá je založená na poznatku pohybovať sa po takej ceste, ktorej úseky sa nikdy nekrižujú. Jej výhoda je pomerne veľká rýchlosť, ale na nešťastie nám nezaručuje, že výsledná cesta bude optimálna.



Obr.5 Riešenie podľa vonkajšej cesty

Iný výhodný postup, ktorý možno aplikovať, je algoritmus založený na princípe výberu nasledujúceho mesta (uzlu) podľa kritéria „chod do najbližšieho nenavštíveného mesta“ **[2]**. Tento algoritmus má vysokú efektivitu, lebo riešenie sa nájde v jedinej iterácii. Bohužiaľ, ani tento algoritmus vo všeobecnom prípade nezaručuje nájdenie optimálneho riešenia. Väčšinou však pôjde o lepšie riešenie, ako pomocou vonkajšej cesty.

Existujú však heuristické metódy, ktoré sú úplné. Jednou z nich je **Metóda dynamického programovania nad podmnožinami** (Dynamic programming across the subset) **[3]**. Táto metóda umožňuje nájsť optimálne riešenie so zložitosťou $O(N^2 2^N)$, čo je výrazné zlepšenie oproti triviálnemu riešeniu $O(N!)$. Jej hlavnou myšlienkou je generovanie stavov iba každej „zaujímavej " podmnožiny množiny všetkých miest. Obchodný cestujúci začína v meste 1, a je celkovo N miest. Potom sa hľadá minimum cesty, ktorá začína v meste **1** a končí v nejakom meste **j** (odtiaľ sa vracia opäť do 1) a prechádza cez zvyšné mestá podmožiny, z ktorej bolo predtým aj mesto j.
Kardinalita príslušnej podmnožiny sa postupne zvyšuje. Preto si je nutné pamätať pomerne veľké množstvo už vykreovaných ciest.

V roku 1993 publikoval R.Z Hwang a kolektív, heuristickú metódu na riešenie tzv. Euklidovský problém obchodného cestujúceho. Jeho zložitosť je subexponenciálna $O(c^{\sqrt{n}\log n}), c > 1$. Euklidovský problém obchodného cestujúceho je špeciálnym prípadom problému obchodného cestujúceho, kedy sú mestá položené na Euklidovskej ploche, a vzdialenosť medzi dvoma mestami je daná pomocou Euklidovskej metriky $d(i,j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$. Metóda je založená na planárnych oddeľovačoch štruktúr, a preto ju nemožno aplikovať na všeobecný problém obchodného cestujúceho .

Kódovanie stavov

Pri kódovaní stavov pomocou permutácie miest sú mestá reprezentované ako čísla a každé mesto má svoje vlastné jedinečné číslo. Keďže prvé mesto, ktoré má číslo 0 je nemenné, nie je možné v operátoroch s ním manipulovať. Prvé mesto je zároveň aj posledným, aby bola cesta uzavretá. S posledným miestom nie je taktiež možné v žiadnom operátore manipulovať. Na nasledujúcom obrázku je znázornený možný stav:

0	15	28	4	61	49	74	.....	47	59	8	0

Obr.7 Návrh stavu pre TSP

Pri kódovaní stavov pomocou reťazcov nie sú mestá reprezentované priamo v reťazci ale v reťazci sa nachádzajú iba čísla reprezentujúce „operátory“ (posun doľava, doprava, hore a dolu). Na nasledujúcom obrázku je znázornený možný stav:

1	1	1	2	3	3	1	.....	4	4	2	2

Obr.7 Návrh stavu pre TSP

Aby bolo možné vypočítať vzdialenosti medzi jednotlivými mestami je nutné vytvoriť dátovú štruktúru, ktorá niečo podobné umožňuje. Ako najjednoduchšie riešenie sa javí vytvorenie **matice susednosti,** ktorej prvky s indexom **i,j** budú reprezentovať vzdialenosť medzi mestami **i** a **j**.

Algoritmy

Základom metódy simulovaného žíhania je tzv. **Metropolisov algoritmus.** Metropolisov algoritmus je kráčavým algoritmom v priestore stavov pre danú teplotu. T Metropolisov algoritmus v pseudokóde vyzerá nasledovne **[1]**:

```
void Metropolis_algorithm(stav x_init, stav x_out ,int k_max,double T){
  int k=0;
  double Pr;
  stav  x',x=x_init;
  for (k=0;k< k_max;k++){
            x'= Permut(x);
            Pr= min(1,exp(-(E(x')-E(x))/T));
            if (Rnd() <Pr)  x=x';
      }
 x_out=x;
}
```

1. Funkcia *Permut(x)* vráti nejakú permutáciu (zmenený stav oproti pôvodnému) stavu x

2. Funkcia f(x), je ľubovolná nezáporná funkcia, ktorá nám „ohodnotí“ stav x (jeho „energiu“).
3. k_{max} – určuje maximálny počet pokusov o permutáciu stavov, toto číslo musí byť dostatočne veľké, aby bola dosiahnutá termodynamická rovnováha.
4. T aktuálna hodnota teploty
5. Funkcia Rnd() reprezentuje generátor náhodných čísel rovnomerného rozdelenia z intervalu $\langle 0,1)$
6. x’ je porušený stav stavu x.
7. x_{init} je počiatočný stav
8. x_{out} je koncový stav

Metóda Simulovaného žíhania spočíva v tom, že „zreťazíme“ postupnosť Metropolisových algoritmov, kedy vstupom jedného je výstup druhého. Algoritmus je začína pri teplote T_{max} , ktorá sa volí tak, aby približne bolo pri nej akceptovaných 50% stavov **[1]**. Pri každom zreťazení dvoch po sebe idúcich Metropolisových algoritmov nepatrne znížime teplotu, a to pomocou funkcie *schedule(T)*. Predpokladáme, že pri každom Metropolisovom algoritme je dosiahnutá tepelná rovnováha, preto je nutné zvoliť správne hodnotu k_{max}. Algoritmus pokračuje, až kým teplota nedosiahne alebo neklesne pod Tmin. Výstup z posledného Metropolisovho algoritmu by malo byť nami hľadané minimum funkcie **[1]**.

```
stav Simulated_annealing(double Tmin,double Tmax,int kmax){
xinit=vygeneruj náhodný stavový vektor;
      T=Tmax;
      while (T>Tmin ){
          Metropolis_algorithm(xinit,xout,kmax,T);
            xinit=xout;
          T=schedule(T);
      }
      return xinit;
}
```

Keďže pri riešení špecifických problémov, pôvodná metóda simulovaného žíhania nedosahovala optimálne výsledky, boli navrhnuté varianty tejto metódy. Hlavný rozdiel medzi pôvodnou metódou simulovaného žíhania spočíva v zmene funkcie predstavujúcej Metropolisov algoritmus.

Simulované žíhanie s elitizmom

Základným rozdielom pri Metropolisovom algoritme je spôsob akceptácie porušeného stavu. Kým v originálnej verzii simulovaného žíhania sa spôsob akceptácie porušeného stavu opieral o veľkosť pravdepodobnosti :

$$p(\text{aktualny} -> \text{poruseny}) = \exp\left(-\frac{\Delta f}{T}\right)$$

v metóde simulovaného žíhania s elitizmom, bude akceptácia porušeného stavu možná len v prípade ak je funkčná hodnota (energia) porušeného stavu menšia alebo rovná ako funkčná hodnota (energia) stavu aktuálneho. Teda základný predpoklad simulovaného žíhania bude teraz modifikovaný tak, že ďalší Metropolisov algoritmus je inicializovaný **najlepším** riešením, ktoré bolo získané v priebehu predchádzajúceho Metropolisovho algoritmu pri vyššej teplote. Modifikovaná funkcia Metropolisovho algoritmu je **[1]**:

```
void Ellitism_Metropolis_algorithm(stav x_init, stav x_out,
int k_max,double T_t)
  int k=0;
  double Pr;
  stav  x',x=x_init;
  for (k=0;k< k_max;k++){
            x'= Permut(x);
            if(f(x')<= f(x)){
               x=x';
            }
 x_out=x;
}
```

Ak sa pozrieme na algoritmus bližšie, tak vo všeobecnosti, simulované žíhanie s elitizmom nespĺňa dôležitú podmienku metódy, a to akceptáciu aj horších stavov. Pretože v každom kroku zníženia teploty je Metropolisov algoritmus inicializovaný najlepším riešením, ktoré bolo zaznamenané v celých predchádzajúcich behoch Metropolisových algoritmov. Môže nastať taká situácia, že výsledné riešenie produkované posledným behom Metropolisovho algoritmu odpovedá lokálnemu minimu, ktoré je podstatne odlišné od globálneho minima. Pri elitickom Simulovanom žíhaní môže funkcia veľmi rýchlo spadnúť do lokálneho minima, z ktorého sa už nedostane, a to vďaka spôsobu akceptácie porušených stavov, ktorých funkčná hodnota (energia) musí byť menšia ako už predtým dosiahnutá. Pri elitickom simulovanom žíhaní ide vlastne o „slepé " hľadanie minima, a to pomocou generovania nových stavov a akceptáciou toho najlepšieho, takže sa stráca pôvab pôvodnej metódy, ktorá umožňovala takéto lokálne minimum preskočiť.

Paralelné simulované žíhanie

Táto variácia simulovaného žíhania je založená na súčasnom aplikovaní simulovaného žíhania na množinu stavov $x,x',x'',...x^{(n)}$, že nad nimi bežia Metropolisové algoritmy s rovnakou teplotou. K tomu, aby stavy medzi sebou interagovali, zaviedla sa medzi stavmi operácia kríženia podobným spôsobom ako v genetických algoritmoch. Kríženie nastane pri medzi dvoma stavmi nastane len s malou pravdepodobnosťou (odporúča sa 0,5%). Ako bolo uvedené v predošlom príklade, zmena oproti pôvodnému algoritmu simulovaného žíhania nastane len v funkcii Metropolis_algorithm, kde je nutné implementovať novú funkciu **Crossover**(x1, x2), ktorá vygeneruje dva nové stavy vektorov, tak, že vymení ich obsah nejakých ich podreťazcov.
Modifikovaná funkcia Metropolisovho algoritmu je nasledovná **[1]**:

```
void Metropolis_algorithm(stav x_init, stav x_out, int k_max,double T_t){
  int k=0;
  double Pr;
  stav  x[N]=x_init;
```

```
  for (k=0;k< k_max;k++){
       if(Pcross<Rnd()){
          for(int i=0; i< N; i++){
               x'= Permut(x[i]);
                    Pr= min(1,exp(-(f(x')-f(x[i]))/T));
                    if (Rnd() <Pr)
                     x[i]=x';
          }
     }
        else{
           int first= RandN(0,N-1);
           int second = RandN(0,N-1);
           (x',x')= Crossover(x[first], x[second])
                 Pr_1=min(1,exp(-(f(x')-f(x[first]))/T));
                 Pr_2=min(1,exp(-(f(x'')-f(x[second]))/T));
                 If (Rnd()<Pr_1) x[first]= x';
                 If (Rnd()<Pr_2 ) x[second]= x'';

     }
}
x=best(x[]);
x_out=x;
}
```

Funkcia *best* nájde najoptimálnejšie riešenie spomedzi generovaných riešení v poli stavov x[.].*RandN(0,N-1)* vygeneruje náhodné číslo v intervale <0,N-1>. Ešte tu musíme podotknúť, že tu musí byť spravené opatrenie aby index-i *first* a *second* neboli rovnaké a nekrížili sa rovnaké stavy.

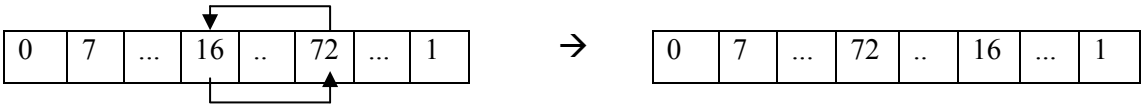
Táto metóda je veľmi efektívna a úspešne sa používa pri riešení problémov s veľkým počtom premenných, kde pôvodná metóda simulovaného žíhania nedokázala nájsť optimálne riešenie.

Operátory

Operátor Permut(x) je vlastne symbolický zápis troch operátorov, z ktorých sa vyberie a aplikuje práve jeden z nich. Výber operátorov je náhodný a deje sa na základe generovania náhodného čísla z intervalu (0,1). Jednotlivým operátorom sú pridelené určité intervaly pravdepodobnosti, a keď do neho náhodne vygenerované číslo padne, tak sa aplikuje daný operátor.

Operátor switching(x)

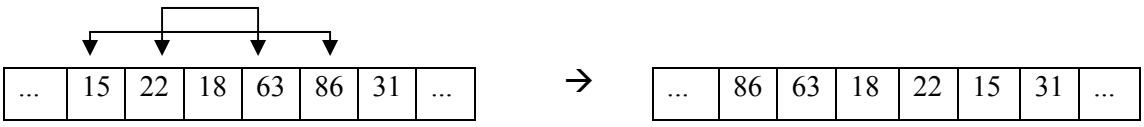
Tento operátor má dve varianty. Prvý variant je, že pre daný stav, ktorý je reprezentovaný ako postupnosť miest vymení poradie dvoch miest. Druhý variant je, že sa vymenia poradia miest niekoľkonásobne, a to v cykle. Pravdepodobnosť aplikácií verzií jednotlivých typov toho operátora som zvolil 90% pre prvú variantu a 10% pre druhú variantu. Samotný výber miest, ktoré sa idú vymieňať je náhodný, ale pritom musí byť zaručené, že druhé mesto nie je totožné s prvým. Na obrázku je vidieť stav pred a po výmene v prípade použitia prvej varianty.



Obr.9 aplikácia operátora switching

Operátor reversing(x)

Tento operátor prevráti podreťazec medzi dvoma náhodne zvolenými mestami v stave x. Náhodne zvolené mestá musia byť vzdialené aspoň o 2, lebo ak by neboli, išlo by o aplikáciu operátora switching. Na nasledujúcom obrázku je znázornená aplikácia tohto operátora.



Obr.10 aplikácia operátora reversing

Operátor transporting(x)

Tento operátor premiestni vybraný podreťazec na vopred zvolené miesto. Premiestňovanie sa deje nasledujúcim spôsobom, najprv sa zvolia náhodne dva index-i v aktuálnom stave (teda cesty), prvý bude prvé mesto v podreťazci, ktoré sa bude premiestňovať, a druhý určuje index v ceste kam sa tento podreťazec bude premiestňovať. Potom sa určí maximálna dĺžka podreťazca –dlzkahranica tak, aby bolo možné daný podretazec premiestniť. Potom sa vygeneruje celé náhodné číslo z intervalu <2, dlzkahranica> ktoré bude reprezentovať dĺžku podreťazca, ktorý sa bude premiestňovať na určené miesto. Pri premiestňovaní reťazca z jedného miesta na druhé, môže nastať prekryv častí prenášaného reťazca samého so sebou. Na nasledujúcom obrázku je znázornená aplikácia tohto operátora.

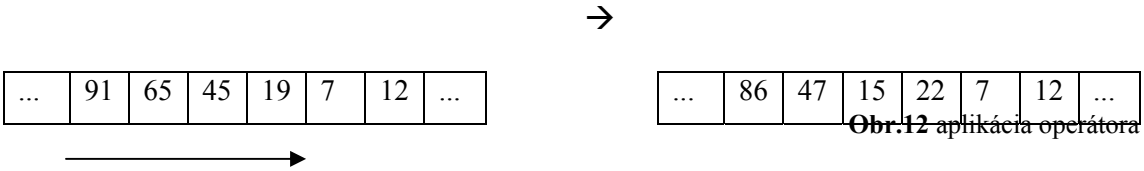
Obr.11 aplikácia operátora transporting

Operátor crossover(x1,x2)

Tento operátor, ktorý bol „požičaný“ z Genetických algoritmov, má za argumenty dva stavy (cesty). Jeho úlohou je výmena určitých podreťacov v oboch cestách x1 a x2. Algoritmus funguje tak, že sa najprv zvolí bod kríženia a potom dĺžka reťazca, ktorý sa ide krížiť Pri jeho implementácií je nutné brať do úvahy to, aby sa daných reťazcoch pri krížení zachovalo pravidlo, že každé mesto sa nachádza v danom stave (ceste) práve jeden krát. To sa dá docieliť tým, že ešte pred samotnou výmenou podreťazcov sa spravia kópie pôvodných stavov. Potom sa pôvodné reťazce krížia práve s touto kópiou susedného stavu (prvý stav by sa zmenil pri krížení s druhým a potom by v druhom vymieňal ten istý reťazec, ktorý pochádza z neho samého, takže by neprišlo ku žiadnej zmene). Pri samotnej výmene sa vybrané mesto v kópii reťazca hľadá v druhom stave, potom sa aplikuje tá istá procedúra ako pri operátore switching, teda dané nájdené mesto, ktoré má inú polohu akú má mať sa vymení s mestom, ktoré sa momentálne nachádza v žiadanej polohe v reťazci. Na nasledujúcom obrázku je znázornená aplikácia tohto operátora.

...	86	47	15	22	18	63	...

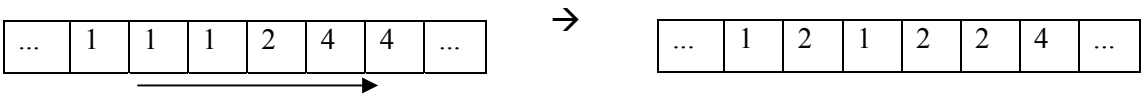
...	91	65	45	19	18	63	...



Obr.12 aplikácia operátora crossover

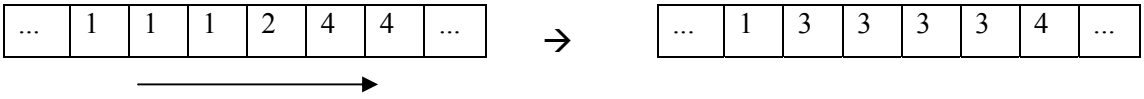
Operátor Mutation(x,pr)

Tento operátor sa používa iba v prípade kódovania pomocou binárneho reťazca. Operátor náhodne vyberie určitý podreťazec v reťazci cesty a následne pre každý prvok z tohto podreťazca vykoná zmenu s pravdepodobnosťou pr. Na nasledujúcom obrázku je znázornená aplikácia tohto operátora.



Operátor Mutation2(x)

Tento operátor sa používa iba v prípade kódovania pomocou binárneho reťazca. Operátor náhodne vyberie určitý podreťazec v reťazci cesty a následne pre každý prvok z tohto podreťazca vykoná zmenu na vopred určený smer pohybu. Na nasledujúcom obrázku je znázornená aplikácia tohto operátora.



Výsledky experimentov

Permutácie miest

Vstupná cesta jednotlivých algoritmov je náhodná. Vstupom programu je 100 miest Tmax=6, Tmin=0,05 a multiplikatívny rozvrh teploty bez paralelného výpočtu

kmax	**alfa**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**6**	**9**	**10**	**∑opt**
5000	**0,99**	100	100	100	100	100	100	100	100	100	100	**10**
5000	**0,97**	102	100	100	104	100	102	100	102	100	100	**6**
2500	**0,99**	102	100	100	102	100	100	100	102	102	102	**5**
2500	**0,98**	102	102	102	100	102	104	102	104	100	104	**2**
1000	**0,995**	100	100	100	100	100	100	102	102	100	100	**8**
1000	**0,99**	100	100	100	100	104	102	100	102	100	102	**6**
500	**0,995**	104	100	106	102	102	102	100	100	104	102	**3**
500	**0,99**	102	104	104	104	106	104	104	100	106	104	**1**

Pre porovnanie Adaptívne simulované žíhanie

kmax	**alfa**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**∑opt**
500	**0,99**	100	100	100	100	100	100	100	100	100	100	10

Ako je vidieť adaptívna metóda simulovaného je ďaleko úspešnejšia ako hľadaní optimálnej cesty ako obyčajná metóda simulovaného žíhania s multiplikatívnym rozvrhom teploty.

Ako demonštráciu sily paralelného simulovaného žíhania použijem nasledujúci príklad. Kdeže by nebolo celkom v poriadku porovnávať výsledky z behu jedného jedného vlákna s behom viacerých, porovnáme výsledky paralelnej metódy s možnosťou kríženia 3% a bez možnosti kríženia. Bolo vytvorených 5 vlákien.

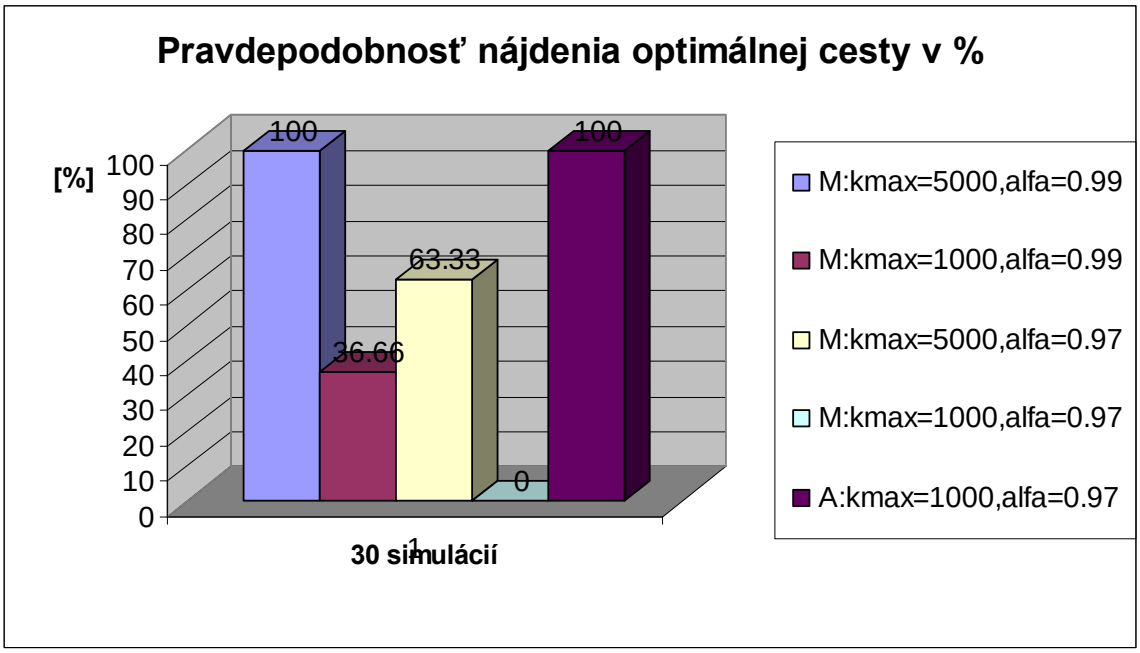
bez kríženia

Kmax	**alfa**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**∑opt**
500	**0,99**	102	102	102	104	102	100	100	100	104	102	3

s križením 3%

kmax	**alfa**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**∑opt**
500	**0,99**	100	102	102	104	102	102	100	100	100	100	5

Z tohoto výsledku vyplýva, že pokiaľ v prvom prípade išlo iba paralelné prehľadávanie stavového priestoru, v druhom prípade bola úspešne použitá operácia kríženia.



Permutácie miest

Vstupná cesta jednotlivých algoritmov je náhodná. Vstupom programu je 100 miest Tmax=10, Tmin=0,05 a multiplikatívny rozvrh teploty bez paralelného výpočtu. Penalizácia za návštívené mesto je 5.

kmax	**Alfa**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**∑opt**
10000	**0,99**	100	110	110	100	100	100	105	113	115	116	4
	Počet nenavš. miest	0	2	2	0	0	0	1	3	3	4	

Zhodnotenie

Výsledky simulovaného žíhania pri použití kódovania pomocou permutácie sú môžem ohodnotiť ako viac než uspokojivé. Pri nastavení kmax=5000 a alfa=0,99 sa dosahuje optima so 100% pravdepodobnosťou. Implementovaný program dokáže pri vhodnom nastavení nájsť aj optimum pre viac miest (až 900 miest viď príloha). Pri použití kódovania pomocou binárneho reťazca sú výsledky pri hľadaní optimálnej

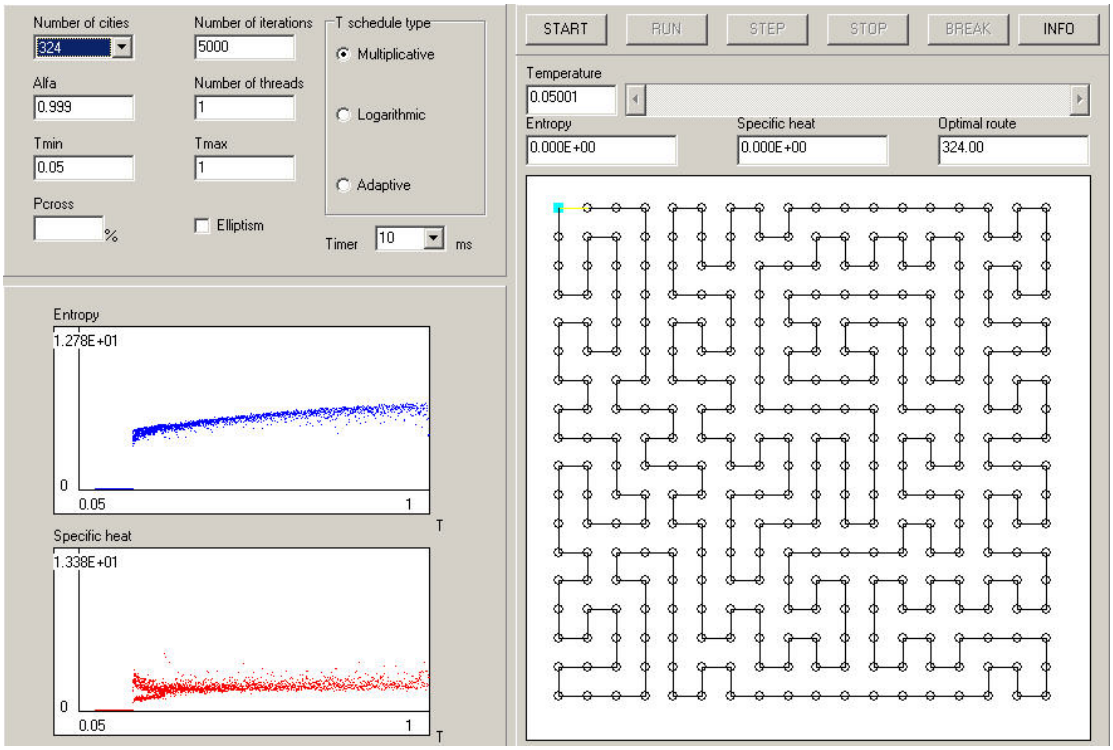
cesty horšie a na nájdenie optima je potrebné väčšie množstvo iterácií. Hlavný problém pri kódovaní pomocou binárneho reťazca vidím najmä v tom, že táto metóda nezaručuje, že každé mesto sa bude nachádzať v skutočnej ceste iba raz resp. že sa vôbec bude vo vytvorenej ceste nachádzať. Tieto výsledky je možné vylepšiť pri použití paralelnej verzie simulovaného žíhania, ale za cenu spomalenia výpočtu .

Zo skúseností so simulovaným žíhaním musím poznamenať, že je dôležitý výber konštánt, ktoré určujú spomaľovanie teploty (alfa) a počet iterácií v Metropolisovom algoritme(kmax).

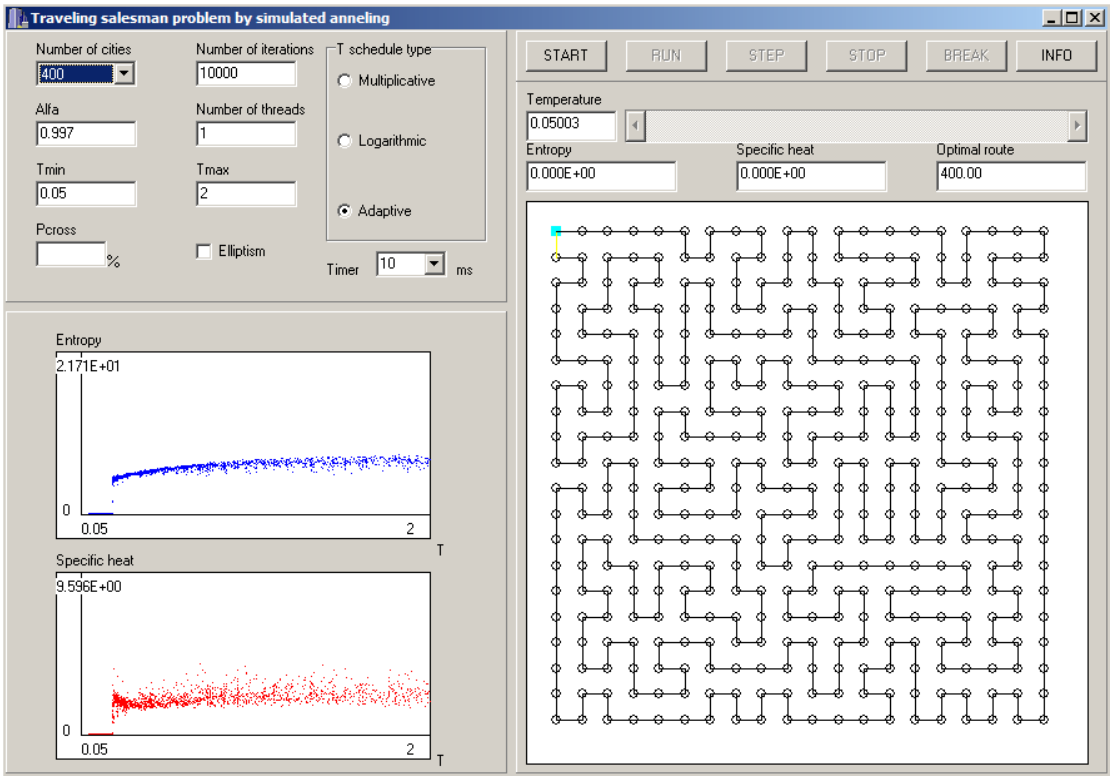
Príloha

V tejto časti sú výsledky jednotlivých experimentov s viac než 100 mestami.

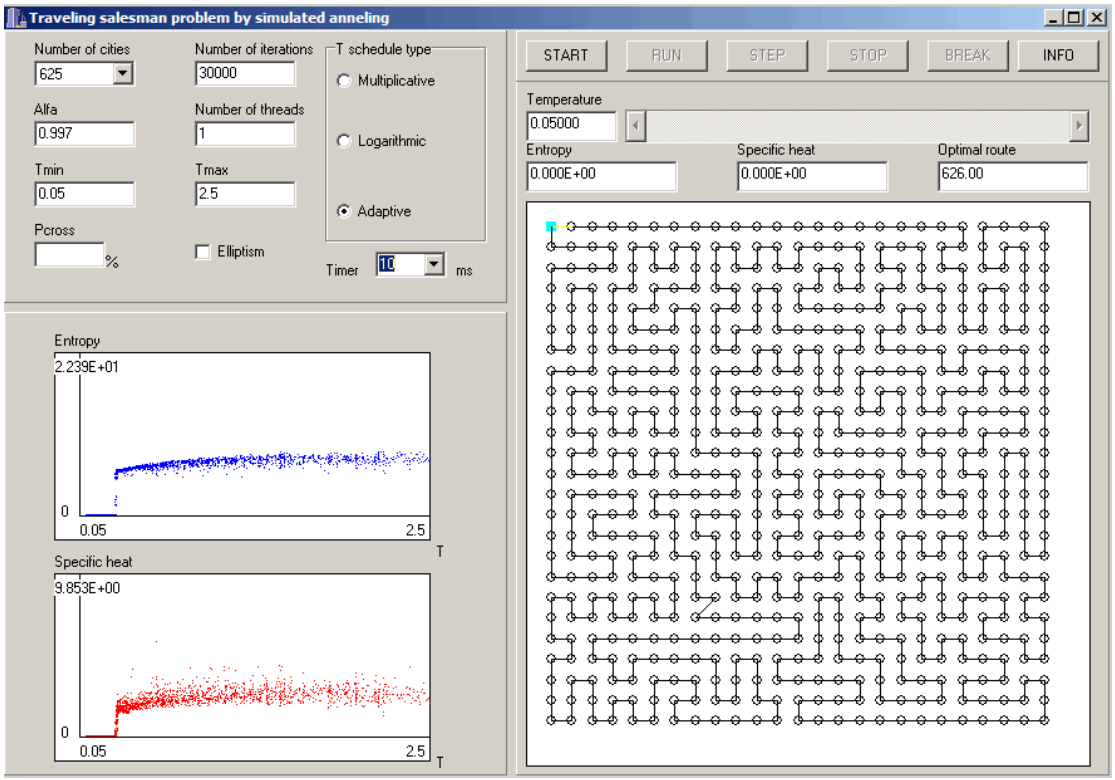
Mriežka 18x18 (324 miest) - nájdené optimum 324



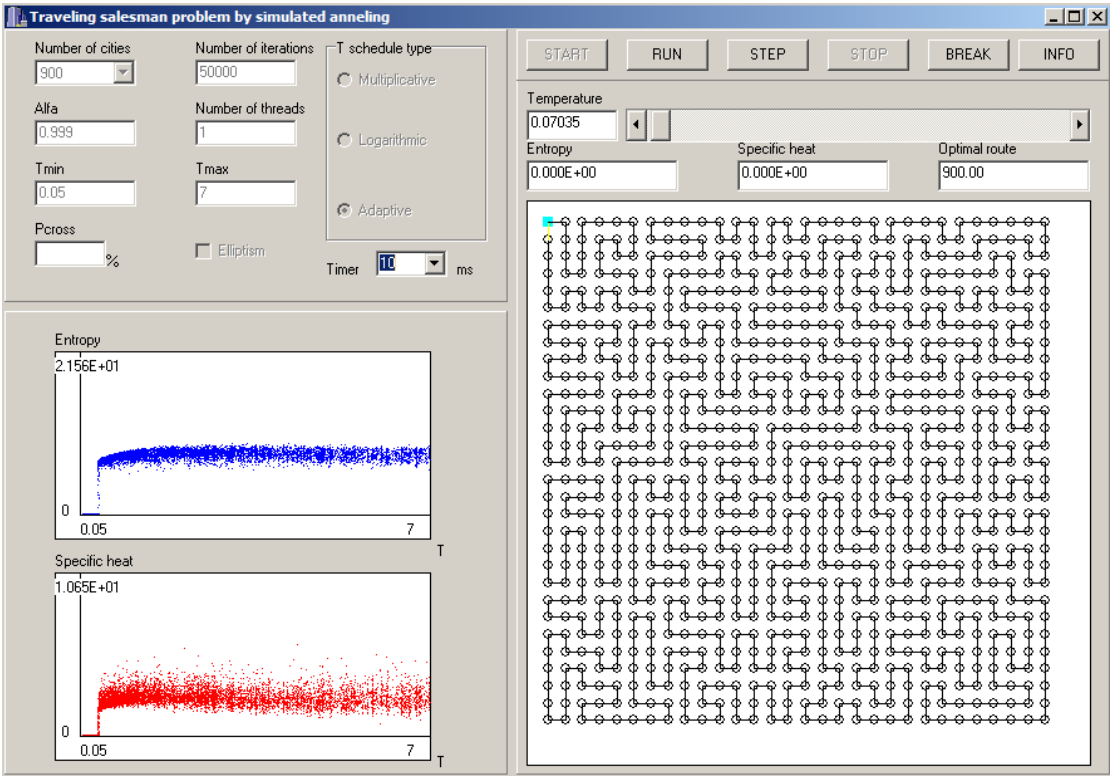
Mriežka 20x20 (400 miest) - nájdené optimum 400



Mriežka 25x25 (625 miest) - nájdené optimum 626



Mriežka 30x30 (900 miest) - nájdené optimum 900



Použitá literatúra

[1] Vladimír Kvasnicka, Jiří Posíchal, Peter Kiňo : *Evolučné algoritmy,* 77-91 (*2000*)

[2] Pavol Návrat , Mária Bieliková , Ľubica Beňušková, Ivan Kapustík, Milan Unger: Umelá inteligencia, 30-31(2002)

[3] Gerhar J. Woeginger: Exact algorithms for NP –hard problems http://www.win.tue.nl/~gwoegi/papers/exact.pdf