METAHEURISTICKÉ ALGORITMY

Metaheusiticky jsou obecné algoritmy pro řešení obtížných úloh. Není třeba definovat, jakou úlohu řešíme.

METODA LOCAL SEARCH

1. Zvolíme si libovolné řešení $x \in X$.
2. Definujeme okolí bodu x: $U(x) \subset X$ (například u obchodního cestujícího by to mohly být všechny okruhy s výměnou 2x2).
3. Existuje-li x' takové, že $f(x') < f(x)$ (v případě minimalizace), pak $x = x'$ a pokračujeme bodem 2. Jinak skončíme.

Nevýhodou je, že často skončíme v lokálním optimu (algoritmus hledá lokální extrém funkce). Algoritmus lze přerušit na základě určitého přerušovacího pravidla.

METODA TABU SEARCH

Je podobná metodě Local Srach, ale během hledání se vytváří tzv. TABU seznam, čili seznam bodů, kam se již nevracíme. U metody Local Search totiž není zajištěno, abychom se znovu nevrátili do bodu, který jsme již prohledali.

1. Hledáme nejlepší x' v okolí U(x) **při respektování TABU listu**.
2. Najdeme-li řešení, které není v TABU listu, přidáme ho do něj a porovnáme ho s dosud nejlepším nalezeným řešením x*. Pokud je $f(x') < f(x^*)$ (v případě minimalizace), pak $x^* = x'$.
3. Pokračujeme až do splnění ukončovacího pravidla.

METODA PRAHOVÉ AKCEPTACE

Metoda vylepšuje nedostatek metody Local Search tak, že se nutně nezastaví v lokálním optimu. Definuje se totiž nějaký práh $T > 0$, a prohládává se okolí i takového bodu, který má horší hodnotu účelové funkce než předcházející bod, ale ne o více, než činí prahová hodnota. Práh se postupně snižuje (násobíme jej zvolenou hodntou r z intervalu (0,1)).

1. x je výchozí řešení, definujeme práh $T > 0$
2. Opakujeme n-krát:
    a. zvolíme $x' \in U(x)$, tzn. vezmeme nějaký bod z okolí bodu x
    b. pokud $f(x') - T < f(x)$, pak $x = x'$, tzn. pokud rozdíl účelové funkce v tomto bodě a v původním bodě x nepřesáhne práh, skočíme do něj, když je třeba o něco málo horší, a budeme prohledávat jeho okolí
    c. $f(x') < f(x^*)$, pak $x^* = x'$, tzn. pokud je účelová funkce v tomto bode lepší než dosud nalezené optimum x*, označíme jej jako nové optimální řešení
3. Pokud je splněno ukončovací pravidlo, skončíme, jinak snížíme práh $T = T \cdot r$ a vracíme se k bodu 2.

METODA SIAM (SIMMULATED ANNEALING, SIMULOVANÉ ŽÍHÁNÍ)

Tato metoda vznikla jako analogie procesu žíhání oceli. Je navržena tak, abychom nezůstali v lokálním minimu, takže s určitou pravděpodobností přijmeme nové řešení i v případě, že má horší hodnotu účelové funkce. Tato pravděpodobnost závisí na teplotě. Ta se postupně snižuje, čímž se pravděpodobnost, že skočíme do jiného řešení s horší hodnotou účelové funkce, stává čím dál nižší.

1. Uvažujme minimalizační úlohu, kde x je výchozí řešení. Definujeme teplotu $T > 0$, parametr ochlazování r z intervalu (0,1) a dobu žíhání n.
2. Opakujeme n-krát:
   a. zvolíme $x' \in U(x)$, tedy nějaký bod z okolí bodu x
   b. pokud $f(x') < f(x)$, pak $x = x'$, tedy pokud je účelová funkce v tomto bodě nižší, skočíme do něj a budeme prohledávat jeho okolí
   c. pokud $f(x') \geq f(x)$, pak $x = x'$ s pravděpodobností $e^{-\Delta/T}$, kde $\Delta = f(x') - f(x)$. Tedy i pokud je účelová funkce v tomto novém bodě vyšší, skočíme do něj s určitou pravděpodobností, která závisí na tom, o kolik vyšší tato účelová funkce je (čím horší je nová hodnota účelové funkce proti staré, tím nižší je pravděpodobnost, že do nového bodu skočíme). Prakticky to uděláme tak, že vygenerujeme hodnotu p z rovnoměrného rozdělení (0,1). Pokud je p menší než $e^{-\Delta/T}$, skočíme do tohoto řešení.
   Například pravděpodobnost přechodu do nového řešení, když $f(x) = 10$, ukazuje pro dvě různé teploty následující tabulka:

$f(x')$	T = 50	T = 5
20	0,82	0,14
15	0,90	0,37
11	0,98	0,82
10	1,00	1,00

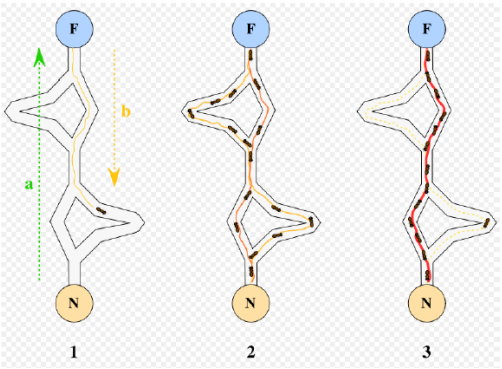
   d. Pokud $f(x') < f(x^*)$, pak $x^* = x'$.
3. Pokud ve druhém kroku zůstaneme v řešení x, říkáme, že proces ztuhl. Jinak snížíme teplotu $T = T \cdot r$ a vracíme se do bodu 2.

GENETICKÉ ALGORITMY

Jsou inpirovány Darwinovou evoluční teorií. Soubor řešení optimalizačního problému se nazývá populace. Fitness value udává, jak dobré je dané řešení ve srovnání s ostatními. Při vytváření další generace se používají tři operace: výběr, křížení (spojení dvou řešení do jednoho) a mutace.

MRAVENČÍ ALGORITMY

Algoritmus vychází z chování mravenců (překvapivě). Ti nejdříve při hledání potravy chodí náhodně, ale když najdou potravu, vrací se do výchozího místa a nechávají za sebou feromonovou stopu. Pokud další mravenec narazí na tuto feromonovou stopu, bude se pravděpodobně pohybovat po ní a vypouštět své feromony, takže stopa se posiluje. Feromony však postupně vyprchávají, takže pokud je nějaká cesta dlouhá, trvá mravencům dlouho, než po ní přejdou, a feromonová stopa vyprchá spíš než u kratší cesty, kde se naopak posiluje.



Zdroj: University of Queensland, MATH3202, 2012

PŘÍKLAD:

Simulované žíhání a genetický algoritmus si ukážeme na příkladu Ztracené mapy (*zdroj zadání: MATH3202, University of Queensland, 2012*).

Uvažujme následující situaci. Máme 9 měst a známe matici jejich vzdáleností. Bohužel jsme však zapomněli, kde tato města leží, tzn. ztratili jsme jejich souřadnice a potřebovali bychom je zpětně najít. Jak na to?

DATA

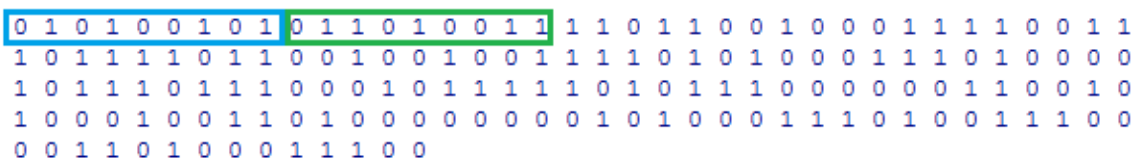
```
######################### DATA  ############################
rm(list = ls()) #cisti pamet
vzdalenosti = matrix(c(
 0, 143, 108, 118, 121, 88, 121, 57, 92,
 143, 0, 35, 63, 108, 228, 182, 73, 162,
 108, 35, 0, 45, 86, 193, 165, 42, 129,
 118, 63, 45, 0, 46, 190, 203, 73, 105,
 121, 108, 86, 46, 0, 172, 224, 98, 71,
 88, 228, 193, 190, 172, 0, 174, 160, 108,
 121, 182, 165, 203, 224, 174, 0, 129, 212,
 57, 73, 42, 73, 98, 160, 129, 0, 117,
 92, 162, 129, 105, 71, 108, 212, 117, 0),ncol = 9)
n = 9 #pocet mest
d = 9 #pocet binarnich promennych kazde souradnice kazdeho mesta,
      #ktere se pak prevedou na skutecne souradnice x,y (viz dale)
      #pr x1 = 110000000 = 1*(2^0) + 1*(2^1) + ... 0*(2*8) = x1 = 3, budou tedy z intervalu 0 az 511
####################################################################
```

ÚČELOVÁ FUNKCE

Než se pustíme do metaheuristik, potřebujeme funkci, která bude umět vygenerovat náhodné řešení, v jehož okolí pak budeme hledat řešení lepší. Dále potřebujeme funkci, která bude schopná počítat pro zadaná řešení hodnotu účelové funkce.

Jednotlivá řešení tedy musíme mít v takové podobě, abychom byli schopni snadno prohledávat jejich okolí. Nabízí se pracovat ve dvojkové soustavě. Pokud budeme mít řešení v podobě řady jedniček a nul, okolní řešení získáme tak, že náhodně některou jedničku vyměníme za nulu či naopak. Každé takové řešení můžeme převést do desítkové soustavy, kde čísla budou představovat souřadnice jednotlivých měst. Pak můžeme pro toto řešení snadno spočítat hodnotu účelové funkce.

Budeme pracovat s mřížkou 511 x 511, kam budeme umisťovat jednotlivé body. Tato velikost by měla stačit. Pro každé z devíti měst budeme potřebovat 2 souřadnice [*x, y*], každou z intervalu <0; 511>. Protože $511 = 2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 + 2^7 + 2^8$, stačí vygenerovat pro každé město 9 bivalentních proměnných určujících souřadnici *x* a 9 bivalentních proměnných určujících souřadnici *y*. Potřebujeme 18 bivalentních proměnných pro 9 různých měst, což je celkem 162 bivalentních proměnných. Například bychom mohli vygenerovat následující řetězec 162 bivalentních proměnných. Ten bychom rozdělili na 18 devítimístných řetězců a každý řetězec bychom převedli do desítkové soustavy, čímž bychom získali 18 čísel představujících 18 souřadnic.



Souřadnice *x* prvního bodu by byla: $x_1 = 0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 0 \cdot 2^4 + 0 \cdot 2^5 + 1 \cdot 2^6 + 0 \cdot 2^7 + 1 \cdot 2^8 = 330$.
Souřadnice *y* prvního bodu by byla: $y_1 = 0 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4 + 0 \cdot 2^5 + 0 \cdot 2^6 + 1 \cdot 2^7 + 1 \cdot 2^8 = 406$.
Všechny souřadnice pro daný řetězec by byly:

```
     x   y
1  330 406
2  155 316
3  379  73
4  175  92
5  372 419
6  471 192
7  138  22
8   80 302
9  195 113
```

Definujeme tedy nejprve funkci *mapa*, která bude převádět zadaný řetězec bivalentních proměnných na souřadnice v desítkové soustavě. Vstupem bude daný řetězec (jako na obrázku výše). Výstupem budou souřadnice pro jednotlivé body v podobě, kterou vidíme výše. Parametry v této funkci jsou n = 9 = počet měst a d = 9 = délka binárního řetězce pro každou souřadnici.

```
#####################  PREVODNI FUNKCE  ######################
#funkce mapa: VSTUPNI hodnota je retezec n*d*2 bivalentnich promennych, VYSTUPNI hodnota je matice souradnic
mapa = function(r) #r je retezec n*d*2 bivalentnich promennych (kazde z n mest musi dostat 2 souradnice(x,y))
{              #tyto souradnice se ziskaji prevodem binarniho retezce delky d do desitkove soustavy - to bude navratova hodnota funkce
 x_bits = NULL   #zalozime si promenne, kam ulozime subretezce delky d
 y_bits = NULL
 for (k in 1:n)  #pro kazde z n mest...
 {
  x_bits = rbind(x_bits,r[(1 + 18*(k-1)) : (d + 18*(k-1))]) #rozsekame retezec r na subretezce po 9
                                                            #z pulky udelame souradnice x
  y_bits = rbind(y_bits,r[((d+1) + 18*(k-1)) : ((2*d) + 18*(k-1))]) #z druhe pulky souradnice y
 }
 x = matrix(rep(NA,(n*d)),ncol = d) #zalozime si matici, v niz prevedeme bivalentni promenne do desitkove soustavy
                                    #radkove soucty v teto matici budou predstavovat souradnice x
 for (i in 1:n)                     #pro kazde z n mest... (pro kazdy radek)
  for(j in 1:d)
  {x[i,j] = (x_bits[i,j])*2^(j-1)} #prevedeme prvek ve sloupci j=1...d do desitkove soustavy
 x = apply(x,1,sum)              #secteme prvky v kazdem radku - dostaneme souradnice x mest 1,2...n

 y = matrix(rep(NA,(n*d)),ncol = d) #stejnym postupem nalezneme souradnice y
 for (i in 1:n)
  for(j in 1:d)
  {y[i,j] = (y_bits[i,j])*2^(j-1)}
 y = apply(y,1,sum)

 souradnice = cbind(c(1:n),x,y) #souradnice X a Y spojime do jedne matice, radky jsou dana mesta
 return(souradnice)          #navratovou hodnotou je tato matice souradnic
}
#########################################################################
```

Máme tedy matici souřadnic. Teď bychom chtěli znát hodnotu účelové funkce tohoto řešení. Co bude účelovou funkcí v případě, že chceme najít takovou matici souřadnic, aby vzdálenosti mezi nimi odpovídaly zadané matici? Bude to **součet rozdílů skutečných** (zadaných) **vzdáleností** a **vzdáleností spočítaných** pro každé dva body na základě vygenerovaných souřadnic daného řešení.
Ze souřadnic daného řešení tedy spočítáme klasickou Pythagorovou větou vzdálenosti mezi každou dvojicí bodů a podíváme se, jak se liší od skutečných vzdáleností. Pokud bychom našli optimální řešení, pak by součet absolutních hodnot rozdílů vzdáleností mezi takto spočítanou maticí a zadanou maticí měl být roven 0. To se pravděpodobně nepodaří na "první pokus" (vlastně se to asi nepodaří vůbec, ale mohli bychom se k nule přiblížit). Budeme tedy souřadnice nepatrně náhodně měnit a zjišťovat, zda se přibližujeme optimu, tedy zda se součet rozdílů začíná blížit nule.

Za tímto účelem definujeme funkci *objective*. Vstupem bude opět řetězec 162 bivalentních proměnných. Ten si funkce *objective* pomocí výše uvedené funkce *mapa* převede na souřadnice, spočítá vzdálenosti mezi nimi Pythagorovou větou a porovná je se vzdálenostmi zadanými v matici *vzdalenosti.* Návratovou hodnotou funkce bude součet rozdílů spočtených a skutečných vzdáleností.

```
#################### UCELOVA FUNKCE ######################
objective = function(r) #r je retezec n*d*2 bivalentnich promennych
{
 souradnice = mapa(r) #retezec prevedeme na souradnice a ulozime do promenne souradnice
 c = matrix(rep(NA,(n*n)),ncol = n)
 for (i in 1:n)
  for(j in 1:n)
  {c[i,j] = sqrt((souradnice[i,2]-souradnice[j,2])^2  + (souradnice[i,3]-souradnice[j,3])^2)}
                #spocitame pythagorovou vetou vzdalenosti bodu se zadanymi souradnicemi
 rozdil = abs(c-vzdalenosti) #zjistime rozdily skutecnych a spocitanych vzdalenosti pro kazdou dvojici mest
 return (sum(rozdil))       #soucet rozdilu vzdalenosti je navratovou hodnotou funkce, mel by byt idealne 0
}
#############################################################
```

Na závěr úvodní části je potřeba poznamenat, že úloha má z logiky věci více řešení, protože souřadnice měst můžeme na mřížce různě otáčet. Pokud bychom chtěli najít skutečné řešení, museli bychom si vzít do jedné ruky obrázek měst umístěných v mřížce (bude výstupem algoritmů níže) a skutečnou mapu Austrálie, v níž tato města leží, a zkusit to nějak spojit. Mně se zatím nepovedlo přijít na to, která města to skutečně jsou.

SIMULOVANÉ ŽÍHÁNÍ

Teď už můžeme skoro začít žíhat! ☺ V procesu žíhání si budeme muset zvolit čtyři parametry:

- parametr ***N*** znamená, **kolikrát snížíme** teplotu
- parametr ***h*** znamená, **kolikrát při dané teplotě prozkoumáme okolí** určitého řešení
- parametr ***T0*** určuje **výchozí teplotu**
- parametr ***s*** z intervalu <0; 1> určuje **rychlost snižování teploty**

K **prozkoumávání okolí daného řešení při dané teplotě *T*** bude sloužit funkce ***jiggle***. Vstupními parametry této funkce jsou teplota, řetězec 162 jedniček a nul a počet prohledávaných řešení. Funkce vezme zadaný řetězec a spočítá pro něj hodnotu účelové funkce. Pak náhodně vygeneruje celé číslo *k* z intervalu <1; 162> a *k*-tý prvek řetězce nahradí prvek opačným, čímž získá jiné řešení z okolí řešení původního, a spočítá pro něj hodnotu účelové funkce. Pokud bude lepší, skočí do něj a bude prohledávat jeho okolí. Ale i pokud bude horší, skočí do něj, avšak pouze s určitou pravděpodobností, která se rovná $e^{-\Delta/T}$, kde Δ = rozdíl účelové funkce nového a původního řešení. Návratovou hodnotou funkce bude řetězec odpovídající řešení, ve kterém při této teplotě nakonec skončíme, a příslušná hodnota účelové funkce tohoto řetězce.

```
 ################################ # JIGGLE ######################################
#VSTUPEM je vychozi reseni pri dane teplote, dana teplota a pocet prohledavanych reseni
#VYSTUPEM je nove nalezene reseni pri dane teplote (lepsi, nebo s urcitou pravdepodobnosti i horsi, ale ne o moc...)
jiggle = function (y0,T,h)   #T je teplota v dane iteraci, h je pocet prohledanych reseni pri teto teplote , y0 je vstupni binarni retezec
{y = y0                  #y nastavime na y0
 c = objective(y)          #spocitame ucelovou funkci pro tento retezec, ulozime ji do promenne c
 for (i in 1:h)           #pro kazde z h opakovani...
 {k = sample(1:(2*n*d),1)                #...vybereme nahodne cele cislo k z <1, delka retezce>
  yp = c(y[1:(k-1)], (1-y[k]), y[(k+1):162])   #k-ty prvek v retezci nahradime opacnym (1 misto 0 ci naopak)
                          #tak ziskame nove reseni z okoli puvodniho reseni
  cp = objective(yp)                  #spocitame ucelovou funkci tohoto noveho reseni
  if(runif(1)< exp(-(cp-c)/T))           #pokud je nahodne cislo z intervalu <0;1> mensi nez e-delta/T, kde delta = (cp-c)...
  {c = cp                          #... pak bude cp novou hodnotou ucelove funkce
   y = yp}                         #... a yp bude nove reseni, jehoz okoli budeme prohledavat
 }                                 #totez opakujeme h-krat
 return (y) }
#################################################################################
```

A konečně můžeme definovat samotnou funkci *anneal*. Tato funkce bude mít čtyři výše uvedené vstupní parametry: *N*, *T0*, *s*, *h*. Nejprve funkce vygeneruje nějaký náhodný řetězec 162 jedniček a nul, který nazvěme *y*. Pak pro každou z *N* iterací udělá následující: prozkoumá pomocí funkce *jiggle* okolí zadaného řešení při zadané teplotě *T0*. Výsledné řešení uloží do matice *reseni* jako další řádek. Řádky matice *řešení* tedy budou představovat výsledná řešení jednotlivých iterací tak, jak je vrátí funkce *jiggle* (tedy nikoli nutně nejlepší nalezená řešení v jednotlivých iteracích, ta by sice bylo možné ukládat do další proměnné, avšak při vhodném nastavení parametrů proces stejně bude k optimálnímu řešení konvergovat). Účelová hodnota výsledného řešení se uloží do proměnné *convergence*, což bude tedy vektor účelových hodnot výsledných řešení jednotlivých iterací. To vše se zároveň zakreslí do grafu. Pak se sníží teplota na *s*-násobek předchozí teploty a opakuje se tentýž postup, a to celkem *N*-krát.

```
######################## ANNEAL ############################
# ANNEAL
reseni = NULL    #matice, jejiz radky budou vysledna reseni z jednotlivych iteraci
convergence =NULL #vektor, jehoz prvky budou hodnoty ucelove funkce vyslednych reseni z jednotlivych iteraci

anneal = function(N,T0,s,h) #VSTUPNI hodnotou bude N=pocet iteraci, T0=vstupni teplota,s=cislo 0 az 1,
                  #ktere rika, jak rychle snizovat teplotu, h = pocet reseni prozkoumavanych funkci jiggle
 #VYSTUPNI hodnotou je list obsahujici vektor hodnot ucelovych funkci z jednotlivych iteraci a souradnice finalniho reseni
{
 x11()  #otevre se graficke okno
 par(mfrow = c(1,2)) #graficky parametr
 randbits =sample(0:1,(2*d*n),replace = TRUE) #vygeneruje se nahodny prvotni retezec jednicek  a nul
 reseni = randbits
 convergence = c(convergence, objective(randbits))

 ### funkce ###
 y = randbits
 T = T0          #teplota prvni iterace je T0
 for (j in 1:N)    #pro kazdou z N iteraci...
 {y = jiggle(y,T,h) #...prozkoumame okoli reseni y pri teplote T, a to celkem h ruznych okolnich reseni
  convergence = c(convergence,objective(y)) #vracenou ucelovou funkci pridame do vektoru convergence
  reseni = rbind(reseni,y)              #vracene reseni pridame jako dalsi radek do matice reseni
  T = s*T                          #snizime teplotu

  ### grafy ###
  #nakreslime si mapu se soucasnym resenim
  grid =0 # Velikost mrizky, kam budeme mesta umistovat
  for (i in 0:(d-1))
   grid = grid + (2^i)
  plot(mapa(y)[,3]~mapa(y)[,2], col = "blue",pch = 16, xlab = "souradnice x", ylab = "souradnice y",main = "mapa",
     xlim = c(0,grid),ylim = c(0,grid))
  grid(nx =32,ny = 32,lwd = 0.01)
  # zakreslime soucasnou hodnotu ucelove funkce
  plot(convergence,xlim=c(1,N+1), pch = ".", ylim = c(0,objective(randbits)),main = "Konvergence",xlab = "iterace",ylab = "Ucelova funkce")
  lines(convergence,xlim = c(1,N),ylim = c(0,objective(randbits)))}
 return(list("convergence" = convergence,"reseni" = mapa(reseni[nrow(reseni),])))
 #navratova hodnota je vektor convergence a matice reseni
}
#########################################################################
```

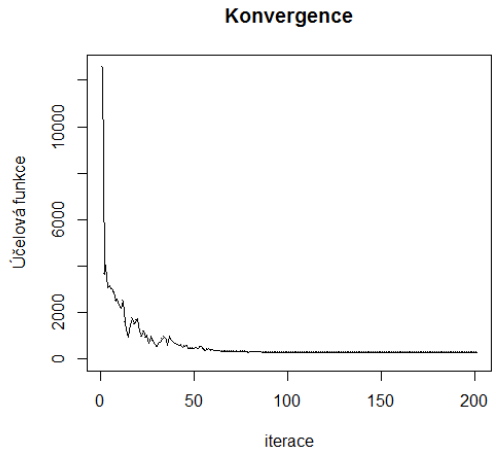
EXPERIMENTY

Pokud už jste porozuměli tomu, co algoritmus dělá, je na čase zkusit si pohrát s jeho vstupními parametry. Pokud se zkoumání algoritmu teď nechcete věnovat, překopírujte všechno do R a hrajte si taky. Příkaz pak spustíme takto:

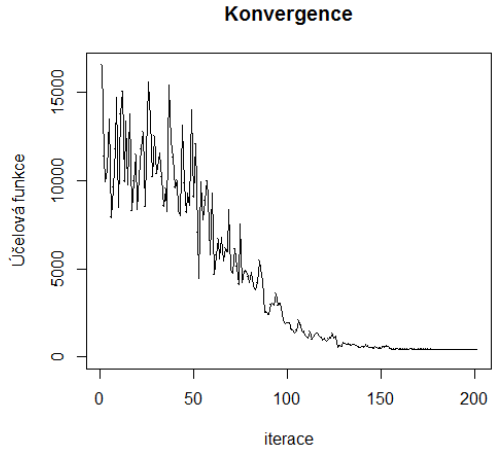
```
############################################################
x = anneal(10,2000,0.95,10)
print(mapa(x$reseni[nrow(x$reseni), ]))
print(paste("Účelová funkce je ", objective(x$reseni[nrow(x$reseni), ])))
############################################################
```

1) *Vliv nastavení původní teploty:*

Podívejme se na vývoj hodnoty účelové funkce v případě, že nastavíme různě vysokou úvodní teplotu. Z obrázků je vidět, že při nízké teplotě účelová hodnota tolik „neskáče," ale je pravděpodobnější, že se zasekneme v lokálním minimu. Proč tomu tak je? Představme si, že máme dvě řešení, jedno má hodnotou účelové funkce 500, druhé 600. Při teplotě 200° máme jen 60 % pravděpodobnost, že se přesuneme do druhého řešení, zatímco při teplotě 20 000° by to bylo celých 95 %. Hodnota účelové funkce původního náhodného řešení však byla kolem 10 000, takže řešení s hodnotou účelové funkce 600 by mohlo být skutečně krokem k lepšímu a příliš nízká teplota nám nedovolí ho prozkoumat.



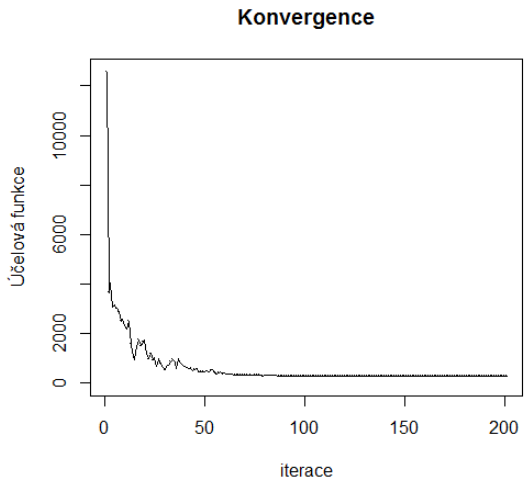
x = anneal(200, **200**, 0.95, 200), úč.funkce: 276

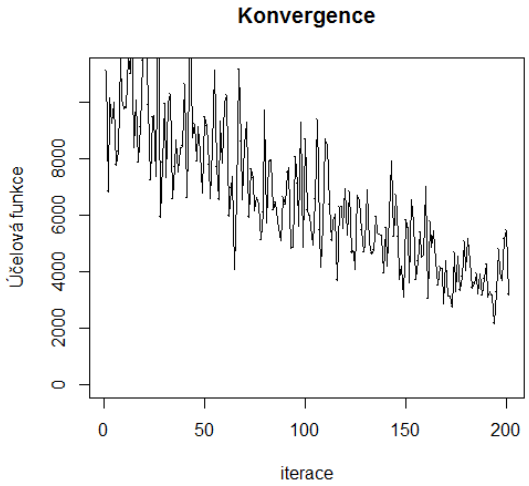


x = anneal(200, **20000**, 0.95, 200), úč. funkce: 405

2) *Vliv nastavení rychlosti ochlazování*

Nyní nastavíme teplotu na 2 000, což se zdá jako rozumný kompromis, a budeme měnit rychlost ochlazování. Podívejme se opět na hodnoty účelové funkce v případě různých hodnot parametru *s*. Pokud je pokles příliš rychlý, jako například při nastavení parametru na 0,8, klesne teplota velice brzy na nulu. To znamená, že už nebudeme moci vůbec prozkoumávat okolní o něco horší řešení, a může se stát, že skončíme v lokálním optimu. Na druhou stranu, příliš vysoký parametr ochlazování znamená, že proces nikdy nebude konvergovat k optimálnímu řešení, protože budeme stále skákat od jednoho řešení ke druhému (pravděpodobnost, že se přesuneme do horšího řešení, bude i na konci procesu stále ještě dost vysoká). Je tudíž vhodné volit parametr ochlazování s ohledem na počet iterací, a to tak, aby se ke konci procesu začala teplota blížit nule.

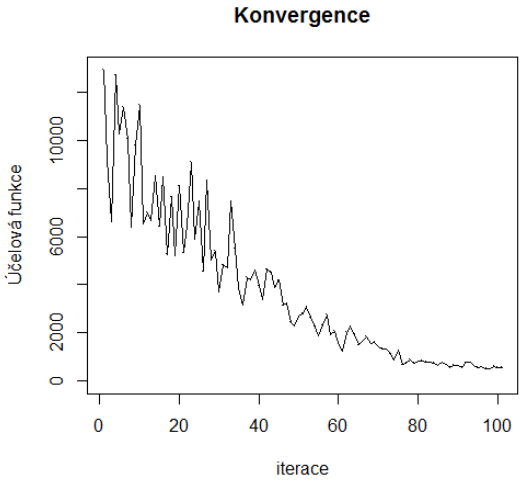


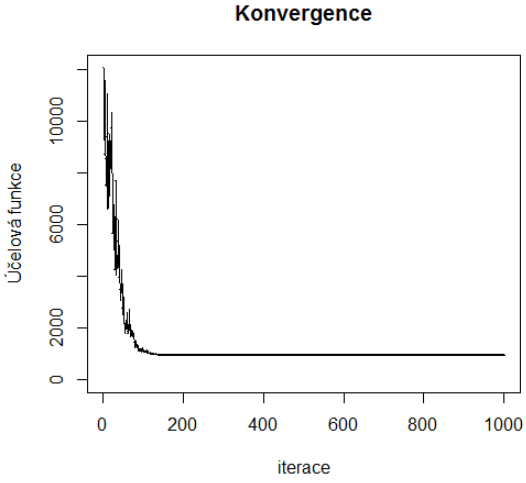


x = anneal(200, 200, **0.8**, 200), úč.funkce: 991 x = anneal(200, 20000, **0.99**, 200), úč.funkce: 3 197

3) *Vliv nastavení počtu iterací*

Poslední, co vyzkoušíme, je vliv počtu iterací a počtu prohledávaných řešení v každé iteraci. Obrázky níže zachycují vývoj účelové funkce v případě, že prohledáváme mnoho řešení, ale jen v málo iteracích (vlevo), a v případě, že prohledáme pouze několik řešení, avšak v mnoha iteracích (vpravo). Zde však hraje velkou roli i nastavená teplota a parametr ochlazování. Například nastavení níže není zrovna ideální, protože v případě tisícovky iterací bude už po první desetině celého procesu teplota téměř na nule.





x = anneal(**100**, 2000, 0.95, **1000**), úč.funkce: 572 x = anneal(**1000**, 2000, 0.95, **100**), úč.funkce: 946

GENETICKÝ ALGORITMUS

Genetický algoritmus funguje následujícím způsobem.

1) Vygeneruje se náhodná populace a vyhodnotí se tzv. fitness value každého jedince (hodnota účelové funkce).
2) Vyberou se nejvhodnější rodiče pro reprodukci, a to tak, aby s větší pravděpodobností byli vybráni rodiče s vyšší fitness value.
3) Křížením dvou rodičů vzniknou dvě děti, a to tak, že se vezme část binárního kódu každého rodiče. Řetězce se rozseknou na náhodném místě. Například rodiče 000000 a 111111 by mohli vytvořit děti 001111 a 110000.
4) U některých dětí dojde k náhodným mutacím (některá jednička se změní na nulu či naopak). To je prevence uvíznutí v lokálním minimu.
5) Totéž se opakuje pro zadaný počet generací.

Budou tedy třeba 4 funkce:

- ➔ funkce parents bude ze zadané matice jedinců dané generace vybírat nejvhodnější rodiče
- ➔ funkce crossover bude pro zadanou dvojici rodičů provádět křížení
- ➔ funkce mutate bude pro zadaného jedince provádět náhodné mutace
- ➔ funkce children budu propojení výše uvedených funkcí. Vybere nejvhodnější rodiče, provede pro jednotlivé dvojice rodičů křížení a u vzniklých dětí udělá náhodné mutace.

A konečně funkce ga (genetic algorithm) zopakuje totéž pro zadaný počet generací.

```
#Funkce parents: VSTUPNI hodnotou je matice jedincu v populaci (1 radek = 1 jedinec = 1 binarni retezec) a hodnota jejich ucelove funkce

#Matice ma tolik radku, kolik je v populaci jedincu.
#Ucelovou funkci fs se rozumi prevracena hodnota ucelove funkce objective, tzn. cim vyssi, tim lepsi.
#VYSTUPNI hodnotou je vektor (jeho delka = pocet jedincu v populaci) s indexy nejvhodnejcich rodicu pro reprodukci
#Funkce tedy ze zadane matice vrati indexy nejvhodnejsich jedincu pro repdodukci vzhledem k jejich hodnote ucelove funkce.
parents = function(p,fs)   #p = matice jedincu v populaci, fs je jejich fitness
{vybrani = NULL            # do matice vybrani budeme ukladat indexy vybranych rodicu
 r = fs/sum(fs)          #kazdemu rodicovi jakoby priradime cast ciselne osy od 0 do 1, tyto hodnoty nazveme r
                    #Napr. kdybychom meli 2 rodice s fitness value 3 a 7, pak r(1) = 0.3, r(2) = 0.7
                    #Pak vygenerujeme nahodne cislo a postupne odecitame hodnoty r jednotlivych rodicu.
                    #Vybran bude ten rodic, po odecteni jehoz hodnoty r spadne y do zapornych cisel.
                    #Nejvice fit rodice tak pochopitelne vybereme vicekrat a nejmene fit rodice treba vubec.
                    #Napr. vygenerujeme-li y = 0.5, odecteme 0.5 - r(1) = 0.2 > 0, avsak 0.2 - r(2) je jiz < 0, vybereme 2.rodice.
                    #Tento postup zajisi, ze budou s vetsi pravdepodobnosti vybrani rodice s vyssi hodnotou fitness.
 for (j in 1:(length(fs)))  #Vyse uvedeny postup dela nasledujici cyklys. Vybrat musíme zase stejny počet rodicu, jako bylo v puvodni populaci
{k = 0
 y = runif(1)
 while(y > 0)
 {k = k + 1
  y = y - r[k]}
 vybrani = c(vybrani, k)}
 return(vybrani)}  #Navratovou hodnotou je vektor s indexy vybranych rodicu.

#VSTUPNI hodnotou funkce crossover jsou 2 rodice. VYSTUPNI hodnotu jsou 2 deti vznikle nahodnym prohozenim casti retezce.
#Vygenerujeme nahodne cele cislo od 1 po delku retezce-1, nazveme ho cut.
#Z prvniho rodice vezmeme binarni promenne od 1 po cut, z druheho od cut po konec retezce.
#Z druheho rodice pak vezmeme  binarni promenne od 1 po cut, z prvniho od cut po konec retezce.
#Priklad: mame rodice 000000 a 111111, cut = 2, pak dite1 = 001111, dite2 = 110000.
crossover = function(par1,par2)
{cut = sample(1:(length(par1)-1),1)
child1 = c(par1[1:cut], par2[(cut+1):length(par2)])
child2 = c(par2[1:cut], par1[(cut+1):length(par2)])
return(rbind(child1,child2))}

#VSTUPNI hodnotou funkce mutate je jeden binarni retezec (=1 jedinec) a pravdepodobnost mutace.
```

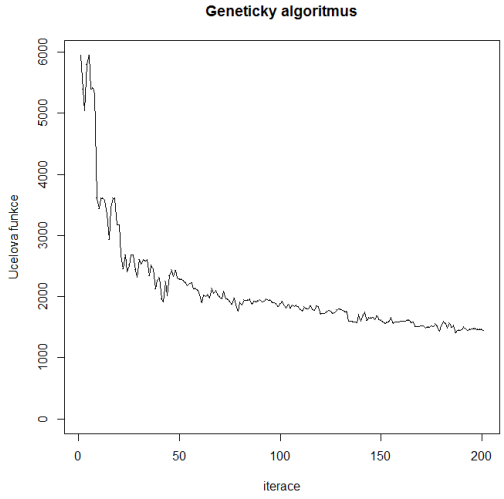
```
#Vybereme nahodne jeden prvek v retezci.
#Pokud je nahodne vygenerovane cislo mensi nez pravdepodobnost mutace, nahradime tento prvek opacnym (1 za 0 ci naopak)
#VYSTUPNI hodnotou je jedinec po mutaci.
mutate = function(v,mp)
{j = sample(1:length(v),1)
 if(runif(1) < mp)
 {v[j] = abs(v[j]-1)}
 return(v)
}

#funkce Children pracuje s funkcemi parents, crossover, mutate.
#VSTUPNI hodnotu je matice rodicu a pravdepodobnost mutace.
#VYSTUPNU hodnotou je matice deti stejneho rozmeru jako matice rodicu (velikost populace se nemeni)
children = function(p,mp) #p je matice rodicu, mp je pravdepodobnost mutace
 {pnew = NULL          #do pnew budeme ukladat deti
  size = nrow(p)/2
  fs = 1/(apply(p,1,objective)) #spocitame prevracenou hodnotu ucelove funkce kazdeho rodice, potrebuje ji totiz funkce Parents
  ps = parents(p,fs) #Vybereme rodice kteri vytvori dalsi generaci, ps je index techto rodicu v zadane matici p.
  for (i in 1:size)  #Vezmeme vzdy dvojice rodicu, tzn. v matici s 10 radky vzdy 1 a 6, 2 a 7... 5 a 10.
  { child12= crossover(p[ps[i],],p[ps[(size+i)],]) #Deti vzniknou krizenim vybranych dvojic rodicu.
    child1 = mutate(child12[1,],mp)          #Nektery gen kazdeho ditete s urcitou pravdepodobnosti zmutuje.
    child2 = mutate(child12[2,],mp)
    pnew = rbind(pnew,child1,child2)
  }
return(pnew)}                          #Navratovou hodnotou funkce je matice deti.

#A konecne samotna funkce ga...
#VSTUPNI hodnotou je rodice = pocet jedincu v populaci, generace = pocet generaci, mp = mutation probability
ga = function(rodice,generace,mp)
{
  x11()
  par(mfrow = c(1,2)) #graficky parametr
  convergence = NULL  #sem se bude ukladat ucelova funkce nejlepsiho reseni v kazde generaci
  reseni = NULL      #sem se bude ukladat nejlepsi reseni v kazde generaci
  p = matrix(sample(0:1,rodice*(d*n*2),replace = TRUE),nrow=rodice) #Vygeneruje se prvotni matice pocet rodicu, pocet radky = pocet rodicu.
  convergence = min(apply(p,1,objective))                  #Spocita se ucelova funkce nejlepsiho rodice.
  for (g in 1:generace) #Pro kazdou z g generaci...
  {
   p = children(p,mp) #Vezmeme matici rodicu p a pomoci funkce children z ni udelame matici deti p.
   nejlepsi = p[which.min(apply(p,1,objective)),] #Z nich vybereme nejlepsi reseni (radek s nejlepsi hodnotou ucelove funkce)
   reseni = rbind(reseni,nejlepsi)            #Nejlepsi reseni ulozime do matice reseni jako dalsi radek.
   convergence = c(convergence, min(apply(p,1,objective))) #Hodnotu jeho ucelove funkce ulozime do vektoru convergence.

   ### grafy ###
   #Nakreslime si mapu se soucasnym nejlepsim resenim.
   grid =0 # Velikost mrizky, kam budeme mesta umistovat
   for (i in 0:(d-1))
    grid = grid + (2^i)
   plot(mapa(nejlepsi)[,3]~mapa(nejlepsi)[,2], col = "blue",pch = 16, xlab = "souradnice x", ylab = "souradnice y",main = "mapa",
        xlim = c(0,grid),ylim = c(0,grid))
   grid(nx =32,ny = 32,lwd = 0.01)
   # Zakreslime soucasnou hodnotu ucelove funkce.
   plot(convergence,xlim=c(1,generace+1), pch = ".", ylim = c(0,convergence[1]),main = "Geneticky algoritmus",xlab = "iterace",ylab =
"Ucelova funkce")
   lines(convergence,xlim = c(1,generace),ylim = c(0,convergence[1]))}
   return(list("convergence" = convergence,"reseni" = mapa(reseni[nrow(reseni),])))
   #Navratova hodnota je vektor convergence a matice souradnic posledniho nejlepsiho reseni.
   }
```

EXPERIMENTY

I genetický algoritmus si zaslouží pár experimentů.

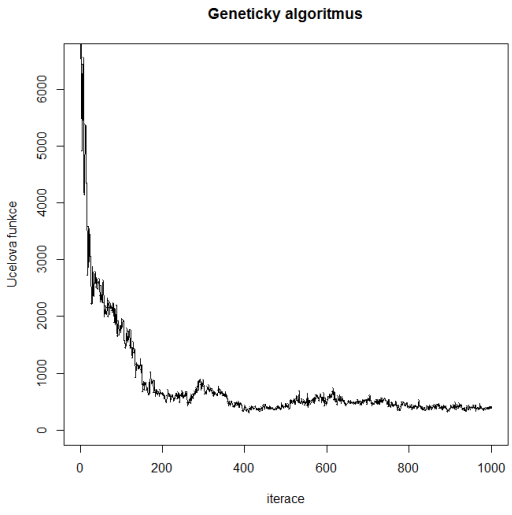
```
#################################################################
y = ga(200,200,0.95)
print(y$reseni
print(paste("Účelová funkce je ", y$convergence[length(y$convergence)]))
#################################################################
```

Uvažujme nejprve situaci, kdy snížíme pravděpodobnost mutace. To může být problém z toho důvodu, že zde hrozí riziko zaseknutí se v lokálním minimu, jak znázorňuje například následující obrázek.

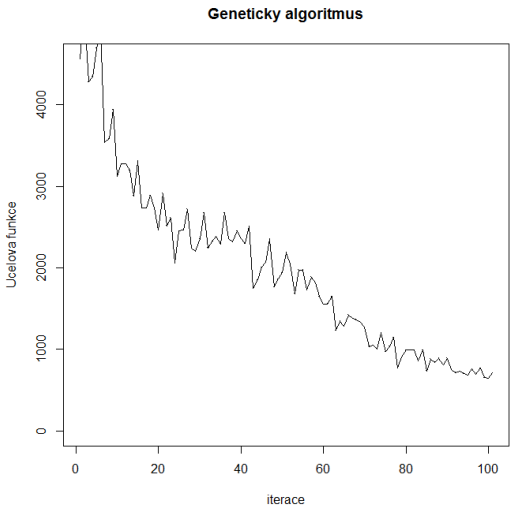


ga(200,200,**0.05**), účelová funkce = 1444

Nyní zkusíme zvýšit počet jedinců v populaci a snížit počet generací a naopak. Zdá se, že méně jedinců v populaci a více generací (obrázek vlevo) vede k lepším výsledkům než více jedinců v populaci a méně generací. Důvod je ten, že když máme málo generací (obrázek vpravo), nemáme dost „času“ na to, abychom vyselektovali ty nejlepší jedince. Při menší populaci může být jako celek populace zpočátku horší, ale v průběhu jednotlivých generací se z ní nakonec vyselektují ti „nejlepší“ jedinci.



ga(**100**,**1000**,0.95), účelová funkce = 409



ga(**1000**,**100**,0.95), účelová funkce = 713

ZDROJE:

Ing. J. Fábry, Ph.D.: přednášky 4EK314 Diskrétní modely, 2011.
University of Queensland, MATH3202, 2012.