

BHLN: Soluzio bakarrean oinarritutako algoritmoak

Borja Calvo, Josu Ceberio, Usue Mori

Laburpena

Kapitulu honetan soluzio bakarrean oinarritzen diren algoritmoak aztertuko ditugu. Algoritmo mota hauen adibiderik esanguratsuenak, eta erabilienak, bilaketa lokalean oinarritutako algoritmoak dira. Horregatik, kapituluaren lehenengo atalean algoritmo haue funtzionamendua eta erlazionatuta dauden kontzeptu batzuk azalduko ditugu. Bilaketa lokalaren desabantailarik handienetako bat, optimizazio prozesua optimo lokalak diren soluzioetan tratatuta geratzea izan ohi da. Ondorioz, kapituluaren bigarren atalean, arazo hau saihesten ahalegintzen diren estrategiatzuk aztertuko ditugu, esate baterako, suberaketa simulatua eta tabu bilaketa algoritmoak.

1 Kontzeptu orokorrak

Bilaketa lokalaren atzean dagoen intuizioa oso sinplea da: soluzio bat emanda, bere «inguruan» dauden soluzioen artean soluzio hobeak bilatzea. Ideia hau bilaketa prozesu bihurtzeko, uneoro problemarako soluzio (bakar) bat mantenduko dugu eta, pausu bakoitzean, uneko soluzio horren ingurunean dagoen beste soluzio batekin ordezkatzeko dugu.

Hainbat algoritmo dira ideia honetan oinarritzen direak, bakoitza bere berezitasunekin, noski. Diferentziak diferentzia, zenbait elementu komunak dira algoritmo guztietan; atal honetan kontzeptu hauek aztertzeari ekingo diogu.

1.1 Soluzioen inguruneak

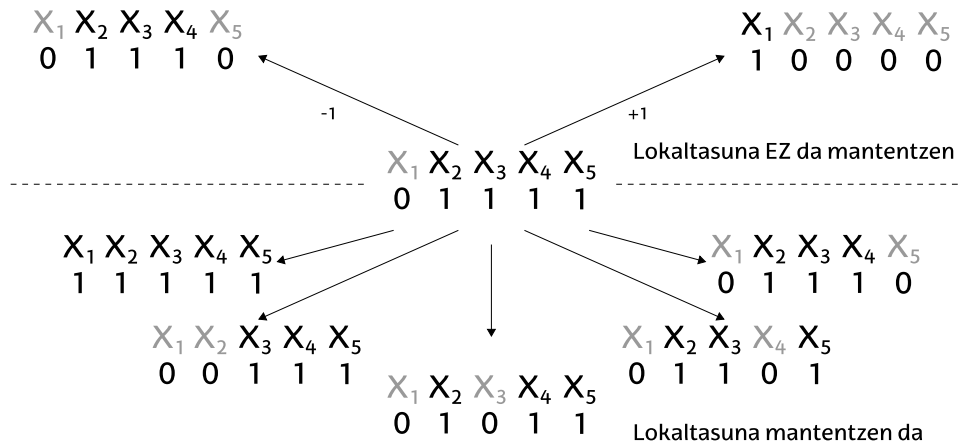
Bilaketa lokalean dagoen kontzepturik garrantzitsuen ingurunearena da – *neighborhood*, ingelesez – eta, hortaz, problema bat ebazterakoan arreta handiz diseinatu beharreko osagaia da. Ingurune funtzioak edo operadoreak¹, soluzio bakoitzeko, bilaketa espazioaren azpimultzo bat definitzen du.

Definizioa 1.1 *Izan bitez \mathcal{S} bilaketa espazioa eta $N : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ funtzioa zeinak, $s \in \mathcal{S}$ soluzioa emanda, $N(s) \subset \mathcal{S}$ bilaketa espazioaren azpimultzo bat itzultzen duen. Orduan, N funtzioa ingurune funtzioa dela diogu.*

Nahiz eta ingurune funtzioaren definizioa oso orokorra izan, errealitatean, soluzio baten ingurunean dauden soluzioen artean –bizilagunak, alegia– nolabaiteko «antzekotasuna» mantentzea interesatzen zaigu. Soluzio baten bizilagunak, orokorrean, ingurure operadore baten bidez lortzen dira, eta beraz, hurrengo adibidean ikusiko dugun legez, antzekotasuna ez dago halabeharrez bermatuta, kodeketaren menpekoea baita.

Laburbilduz, bilaketa lokal bat diseinatzean berebizikoa da «kodeketa - ingurune» bikotea modu egokian aukeratzea, ingurunean dauden soluzioak antzerakoak izan daitezen. Ezaugarri honi lokaltasuna –*locality* ingelesez– esaten zaio, eta emaitza onak lortzeko funtsezkoa da.

¹Programazio testuinguruetan – sasikodetan, adibidez – soluzioak maneiatzeko erabiltzen diren funtzioei «operadore» deritze eta, hortaz, «funtzio» eta «operadore» terminoak baliokide gisa erabiliko ditugu.



Irudia 1: Bi ingurune ezberdinen adibidea. Goikoan bektore bitarrei 1 gehitzen/kentzen diogu inguruneko soluzioak lortzeko. Eskuman dagoen soluzioan ikus daitekeen bezala, lokaltasuna ez da mantentzen, soluzioak elkarrengandik oso ezberdinak direlako. Beheko ingurunea *flip* eragiketetan oinarritzen da. Kasu honetan sortutako soluzio guztiak antzerakoak dira.

Adibidea 1.1 Lokaltasuna *feature subset selection* problemaren Datu meatzaritzan, sailkatzaile funtzioak eraikitzea edo ikastea oso ataza ohikoa da. Funtzio hauek, beraien izenak adierazten duen moduan, datu berriak sailkatzeko erabiltzen dira.

Oro har, datuetan agertzen diren aldagaiek iragarpenak egiteko gaitasun ezberdinak dituzte; are gehiago, aldagai batzuk eragin negatiboa izan dezakete sailkatzailearen funtzionamenduan. Hori dela eta, aldagaien azpi-multzo bat aukeratzea oso pausu ohikoa da; prozesu hau, ingelesez *feature subset selection*, FSS deitzen den optimizazio problema bat da. FSS problemarako soluzioak bektore bitarren bidez kodetu daitezke, bit bakoitzak dagokion aldagaia azpi-multzoan dagoenetz adierazten duelarik.

Bektore bitarrak zenbaki oso moduan interpreta daitezke eta, hortaz, inguruneko soluzioak lortzeko modu bat horiei balio txikiak gehitzea/kentzea da. Adibidez, (01001) soluzioak 9 zenbakia adierazten duela suposatuko dugu, eta hortaz, antzerako soluzioak lor ditzakegu 1 zenbakia (00001) gehituz – 10 zenbakia (01010) lortzen delarik – edo kenduz – 8 zenbakia (01000) dugularik –. Adibide honetan lortutako soluzioak nahiko antzerakoak dira; lehendabiziko kasuan azken aldagaia azken-aurrekoarekin ordezkatu dugu eta bigarren kasuan azken aldagaia kendu dugu. Edonola ere, beste kasu batzuetan lokaltasuna ez da mantentzen; (1000000) soluzioari 1 kentzen badiogu, (0111111) lortuko dugu, hau da, aukeratuta zegoen aldagai bakarra – lehendabizikoa – kendu eta beste gainontzeko guztiak sartuko ditugu. Beste era batean esanda, FSS problemarako ingurune definizio hau ez da oso egokia.

Azpi-multzo problematan ingurune operadore ohikoena *flip* aldaketan oinarritzen da; inguruneko soluzioak sortzeko uneko soluzioaren posizio bateko balioa aldatzen da, hau da, 0 bada 1ekin ordezkatzan da eta 1 bada, 0rekin. Operadore honekin FSS problematan beti bermatzen da lokaltasuna, ingurunean dauden edozein bi soluzioak aldagai bakar bateko diferentzia izango baitute. 1 irudiak adibidea grafikoki erakusten du.

Soluzioak bektoreen bidez kodetzen direnean, inguruneak definitzeko «distantzia» kontzeptua erabili ohi da, esplizituki zein inplizituki. Era honetan, bi soluzio bata bestearen ingurunean daudela esango dugu baldin eta beraien arteko distantzia finkaturiko kopuru bat baino txikiagoa bada. Edozein bi soluzio, s eta s' arteko distantzia, $d(s, s')$ funtzioaz adierazten badugu, ingurunearen definizio orokorra ondorengoa izango da:

$$N(s; k) = \{s' \in \mathcal{S} \mid d(s, s') \leq k\} \quad (1)$$



Bektore motaren arabera distantzia ezberdinak erabil ditzakegu. Jarraian adibide hedatuenak ikusiko ditugu.

Bektore errealak - Bektore errealekin dihardugunean euklidearra da gehien erabiltzen den distantzia

$$d_e(s, s') = \sqrt{\sum_{i=1}^n (s'_i - s_i)^2}$$

Ingurune tamainari erreparatuz, zenbaki errealekin dihardugunez, infinitu soluzio izango ditugu edozein soluzioren ingurunean. Euklidearra distantziarik ezagunena izan arren, badira beste metrika batzuk ere bektore errealean arteko distantzia neurtzeko – Manhattan- edo Chebyshev-distantziak, besteak beste –.

Bektore kategorikoak eta bitarrak - Bektoreetan dauden aldagaiak kategorikoak direnean, bi bektoreen arteko distantzia neurtzeko metrikarik ezagunena Hamming-ek proposatutakoa da: $d_h(s, s') = \sum_{i=1}^n I[s_i \neq s'_i]$, non I funtzioa adierazlea den, eta 1 balioa hartzen duen bere argumentua egia denean, eta 0 beste kasuan. Hor-taz, Hamming-distantziak posizioz posizio desberdintasunak neurtzen ditu. Hamming-distantzia inguruneak definitzeko erabiltzen denean, ohikoena 1 distantziara dauden soluzioetara mugatzea da, hau da:

$$N_h(s; 1) = \{s' \in \mathcal{S} \mid d_h(s, s') = 1\} \quad (2)$$

Algoritmoak diseinatzean oso garrantzitsua da ingurunearen tamaina aztertzea. Aurreko operadorea n tamainako bektore bitar bati aplikatzen badiogu, s soluzioaren bizilagun kopurua $|N(s)| = n$ izango da, posizio bakoitza aldatzeko aukera bakarra baitaukagu. Bektore kategorikoetan, posizio bakoitzean r_i balio hartzeko aukera dagoenean, ingurunearen tamaina $|N(s)| = \sum_{i=1}^n (r_i - 1)$ izango da.

Bestalde, ondoko ekuazioak, edozein distantziarako ingurune funtzio orokor bat adierazten duen distantzia maximoa bektorearen tamaina dela kontutan hartuz, betiere –:

$$N_h(s; k) = \{s' \in \mathcal{S} \mid d_h(s, s') \leq k\} \quad (3)$$

Ikus dezagun adibide bat, **metaheuR** paketea erabiliz. Motxilaren problema erabiliko dugu eta, horretarako, lehenengo, ausazko problema bat eta soluzio bat sortuko ditugu. Pisia eta balioa korrelaturik egoteko, elementu bakoitzaren pisua lortzeko, haren balioari ausazko kopuru bat gehituko diogu. Gero, motxilaren kapazitatea definitzeko ausaz aukeratutako $\frac{n}{2}$ elementuen pisuak batuko ditugu. Azkenik, elementu bakoitza aukeratzeko probabilitatea heren batean ezarri, ausazko soluzio bat sortuko dugu.

```
> library(metaheuR)
> n <- 10
> rnd.value <- runif(n) * 100
> rnd.weight <- rnd.value + runif(n) * 50
> max.weight <- sum(sample(rnd.weight, size = n/2, replace = FALSE))
> knp <- knapsack.problem(weight = rnd.weight, value = rnd.value, limit = max.weight)
> rnd.sol <- runif(n) < 1/3
```

Kontutan hartu behar da motxilaren probleman soluzio guztiak –azpi-multzo guztiak, alegia– ez direla bideragarriak. Eta beraz, ausazko soluzio bat sortzean, elementu edo objektu bat aukeratzeko probabilitatea handitzen badugu, soluzio bideraezinak lortzea probableagoa izango da. Edozein kasutan, sortutako soluzioa bideraezina izan daitekeenez, lehenengo pausua soluzioa zuzentzea izango da. Gero, «flip» operadorea erabiliko dugu inguruneako soluzioak sortzeko. Operadore honek Hamming-en bat distantziara dauden soluzioak esleituko dizkio inguruneari. Nahiz eta uneko soluzioa bideragarria izan, inguruneako bideraezinak izan daitezke, beraz, pausu bakoitzean inguruneako soluzioa bideragarria den ala ez aztertu beharko dugu.



```
> rnd.sol <- knp$correct(rnd.sol)
> which(rnd.sol)

## [1] 2 9

> flip.ngh <- flipNeighborhood(base = rnd.sol , random = FALSE)
> while(has.more.neighbors(flip.ngh)){
+   ngh <- next.neighbor(flip.ngh)
+   isvalid <- ifelse (knp$is.valid(ngh) , "bideragarria" , "bideraezina")
+   mssg <- paste("Inguruneke soluzio ",isvalid,": " ,
+                 paste(which(ngh),collapse = ",") , sep="")
+   cat (mssg , "\n")
+ }

## Inguruneke soluzio bideragarria: 1,2,9
## Inguruneke soluzio bideragarria: 9
## Inguruneke soluzio bideragarria: 2,3,9
## Inguruneke soluzio bideragarria: 2,4,9
## Inguruneke soluzio bideragarria: 2,5,9
## Inguruneke soluzio bideragarria: 2,6,9
## Inguruneke soluzio bideragarria: 2,7,9
## Inguruneke soluzio bideragarria: 2,8,9
## Inguruneke soluzio bideragarria: 2
## Inguruneke soluzio bideragarria: 2,9,10
```

Goiko kodean ikus daitekeen bezala, `metaheuR` paketea inguruneak korritzeko bi funtzio aurki ditzakegu: `has.more.neighbors` eta `next.neighbor`. Izenek adierazten duten bezala, lehenengo funtzioak ingurunean oraindik bisitatu gabeko soluzioaren bat dagoen esaten digu eta, bigarrenak, bisitatu gabeko hurrengo soluzioa itzultzen du. Horrez gain, badago beste funtzio bat, `reset.neighborhood`, ingurune objektua berrabiarazteko. Informazio gehiago lor dezakezu R-ko terminalean `?reset.neighborhood` teklatuz.

Permutazioak - Permutazioen arteko distantziak neurtzeko metrikak existitu arren, ingurune operadore klasikoak ez dituzte zuzenean erabiltzen. Horren ordez, permutazioetan definitutako eragiketak erabili ohi dira, trukaketa eta txertaketa batik bat.

Trukaketan – *swap* ingelesez –, permutazioaren bi posizio hartzen dira eta beraien balioak trukutzen dira. Adibidez, 21345 permutazioaren 1. eta 3. posizioak trukutzen baditugu 31245 permutazioa lortuko dugu. Formalki, trukaketa funtzioa, $t_r(s; i, j)$, defini dezakegu non $s' = t_r(s; i, j)$ bada $s'(i) = s(j)$, $s'(j) = s(i)$ eta $\forall k \neq i, j, s'(k) = s(k)$ beteko den. Funtzio honetan oinarriturik, ondoko ingurunea defini dezakegu:

$$N_{2opt}(s) = \{t_r(s; i, j) \mid 1 \leq i, j \leq n, \forall i > j\} \quad (4)$$

Ingurune honi *2-opt* deritzo, bizilagun bakoitzea bi posizio bakarrik trukutzen baitira. Era berean, operadorea hedatu daiteke trukaketa gehiago eginez. Azkenik, hedatzeaz gain, operadorea murriztu ere egin ahal da, trukaketak elkarren ondoan dauden posizioetara soilik mugatuz. *2-opt* ingurunearen trukaketa operadorea `ExchangeNeighborhood` klaseak inplementatzen du, eta ondoz-ondoko trukaketetara murriztutako bertsioa `SwapNeighborhood` klasearen bidez erabil daiteke.

Ingurunearen tamainari dagokionez, ondoz-ondoko posizioetan soilik trukaketak eginez $n-1$ ingurune soluzio izango ditugu; edozein bi posizio trukutzen baditugu, berriz, ingurunearen tamaina $n(n-1)$ izango da. Hau, jarraian dagoen adibidean ikus daiteke.

```
> n <- 10
> rnd.sol <- random.permutation(length = n)
```



```
>
> swp.ngh <- swapNeighborhood(base = rnd.sol)
> exchange.count <- 0
> swap.count <- 0
> while(has.more.neighbors (swp.ngh)){
+   swap.count <- swap.count + 1
+   next.neighbor(swp.ngh)
+ }
>
> ex.ngh <- exchangeNeighborhood(base = rnd.sol)
> exchange.count <- 0
> while(has.more.neighbors (ex.ngh)){
+   exchange.count <- exchange.count + 1
+   next.neighbor(ex.ngh)
+ }
>
> swap.count

## [1] 9

> exchange.count

## [1] 45
```

Txertaketan – *insert* ingelesez –, elementu bat permutaziotik atera eta beste posizio batean sartzen dugu. Adibidez, 54123 permutaziotik abiatuta, bigarren elementua laugarren posizioan txertatzen badugu, emaitza 51243 izango da. Eragiketa $t_x(s; i, j)$ funtzioaren bidez adieraziko dugu $-i$ elementua j posizioan txertatu –, eta ingurunearen definizioa hauxe izango da:

$$N_{in}(s) = \{t_x(s; i, j) \mid 1 \leq i, j \leq n, \forall i \neq j\} \quad (5)$$

Trukaketan bakarrik bi posiziotako balioak aldatzen dira; txertaketan, berriz, bi indizeen artean dauden posizio guztietako balioak aldatzen dira. Hori dela eta, ingurune operadore bakoitzaren erabilgarritasuna problemaren arabera izango da. Operadore hau ere *metaheur* paketearen aurki dezakegu, *InsertNeighborhood* klasean inplementaturik.

Bi ingurune operadore hauetaz gain, badiran literaturan beste zenbait operadore, inbertsio eragiketaren oinarritutakoak esate baterako.

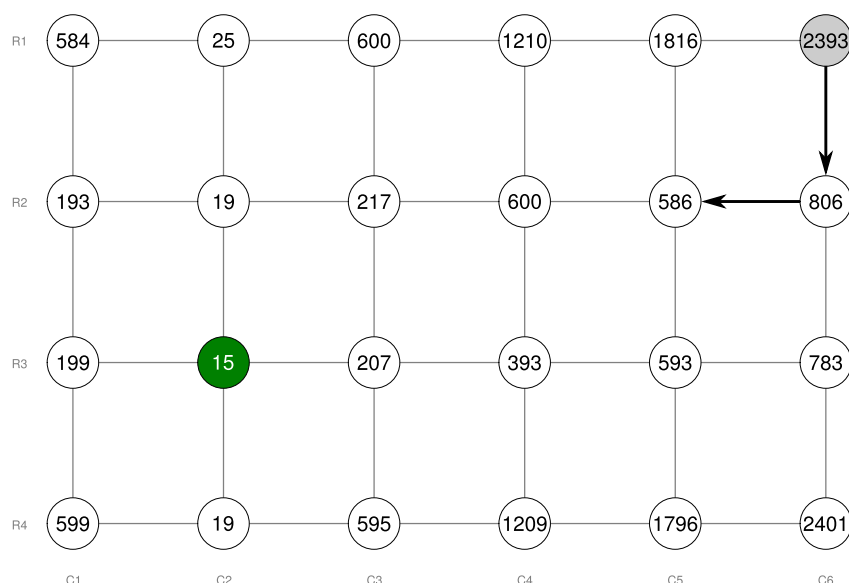
1.2 Optimo lokalak

Bilaketa lokalean – oinarritzko bertsioan, behintzat – soluzio batetik beste batera mugitzeko, uneko soluzioaren ingurunean, helburu funtzioaren balioa hobetzen duen bizilagun bat egon behar da². Ingurune soluzio guztien fitness-a txarragoa baldin bada, orduan soluzio hau *optimo lokala* dela esango dugu, eta bilaketa amaitu egingo da. Formalki, s^* optimo lokala da baldin eta ondoko baldintza betetzen baldin badu:

$$\forall s \in N(s^*) \quad f(s^*) \leq f(s)$$

Aurreko ekuazioa S osorako betetzen baldin bada, hau da, bilaketa espazio osorako, orduan, s^* *optimo globala* dela esango dugu.

²Helburu funtzioak soluzio jakin batean hartzen duen balioa, ingelesezko *fitness* hitzarekin izendatzea ohikoa da. Horregatik, kapituluaren zehar, bi adierazpideak erabiliko ditugu baliokide gisa.



Irudia 2: Optimo lokalaren adibidea. Goiko eskumako soluziotik abiatzen bada bilaketa – (R1,C6), grisean nabarmendua dagoen soluziotik, alegia –, eta pausu bakoitzean aukerarik onena aukeratzen badugu, geziek markatzen duten bidea jarraitu eta, bi pausutan, (R2,C5) soluzioan trabatuta geldituko gara. Soluzio hau optimo lokala da, bere inguruneke soluzio guztiak txarragoak baitira. Optimo lokalaren ebaluazioa 586 da, oso txarra optimo globalarekin alderatzen badugu –(R3,C2) soluzioa–.

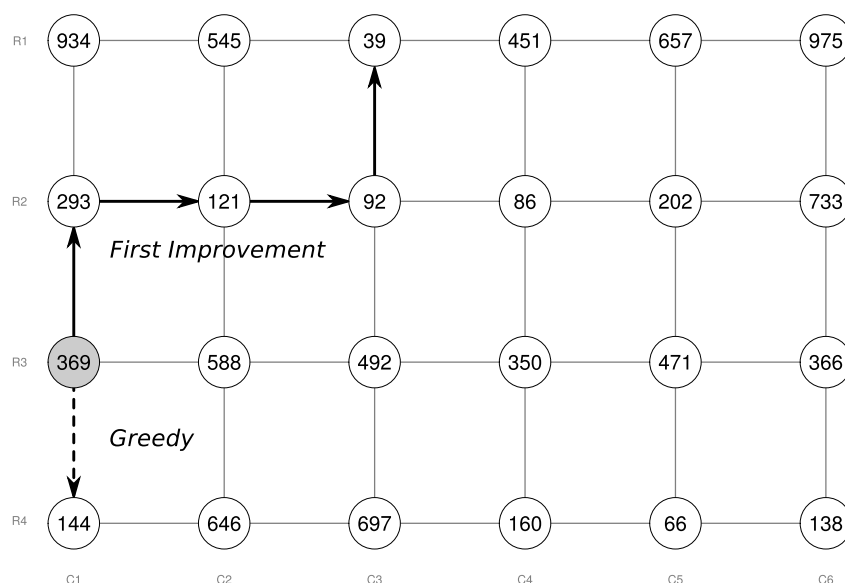
Definizio hauek kontutan hartuz, bilaketa lokala optimo lokal batean amaitzen dela beti ondorioztatzen dugu. Hauxe da, hain zuzen, bilaketa lokalaren ezaugarriarik – eta, aldi berean, desabantailarik – nagusia. Aintzat hartzekoa da optimo lokalak, nahiz eta bere inguruneke soluziorik onenak izan, nahiko soluzio txarrak izan daitezkeela, 2 irudian erakusten den bezala. Irudi honetan kapituluaren zehar maiz erabiliko dugun grafiko mota bat ikus daiteke. Grafikoan, fikziozko problema baterako soluzio guztiak jasotzen dira, bakoitza borobil baten bidez adierazita; borobilaren barruan soluzio bakoitzaren fitness-a dago idatzita. Ingurune funtzioa soluzioak lotzen dituzten marren bidez adierazten da; hala, bi soluzio lotuta badaude, bata besteari ingurunean daudela diogu – bizilagunak direla, alegia.

Adibidea 1.2 2 irudian agertzen diren geziek algoritmoak (R1,C6) soluziotik abiatuta egiten duen bidea erakusten dute. Pausu bakoitzean inguruneke soluziorik onena aukeratzen badugu, algoritmoa bi pausutan trabatuta geldituko da (R2,C5) soluzioan; soluzio honen fitness-a 586 da eta, bere ingurunean dauden soluzioen fitness-ak handiagoa direnez – 1816, 806, 593 eta 600 –, ez dago helburu funtzioaren balioa hobetzen duen soluziorik. (R2,C5) optimo lokal bat da eta, optimo globalaren – (R3,C2) – fitness-a 15 dela kontutan hartuz, nahiko soluzio txarra ere bai.

Optimo lokaletatik ateratzeko hainbat estrategia planteatu dira literaturan, bilaketa lokalaren puntu batean edo bestean aldaketak proposatuz. Hauexek izango dira, hain justu, 3. atalean aztergai izango ditugunak.

Ikusi dugunez, edozein soluziotik abiatuta, bilaketa lokala beti optimo lokal batean amaitzen da. Are gehiago, posible da bi soluzio ezberdinetatik hasita, bilaketa lokala soluzio berdinean amaitzea. Izan ere, optimo lokalek soluzioak «erakartzen» dituzte, zulo beltzak balira bezala. Ideia hau «erakarpen-arroa» – *basin of attraction*, ingelesez – deritzon kontzeptuan formalizatzen da.

Definizioa 1.2 erakarpen-arroa Izan bitez N ingurune funtzioa, f helburu funtzioa, $A(s; f, N) : \mathcal{S} \rightarrow \mathcal{S}$ bilaketa lokaleko algoritmoa eta s^* optimo lokala (N ingurunerako eta f funtziorako). s^* optimo lokalaren erakarpen-arroa $\{s \in \mathcal{S} / A(s; N, f) = s^*\}$ soluzio multzoa da.



Irudia 3: Ingurunekeko soluzioaren aukeraketaren efektua. Irudiak, soluzio berdinetik abiatuta – (R3,C2), grisean nabarmendua – bi estrategia ezberdin erabiliz egindako ibilbideak erakusten ditu. Lehenengo estrategia *first improvement* motakoa da, hau da, helburu funtzioa hobetzen duen lehenengo soluzioa aukeratzen dugu – ingurunekeko soluzioen ordena goikoa, eskumakoa, behekoa eta ezkerrekoa izanik –. Irizpide hau erabiliz egindako ibilbidea (R2,C1), (R2,C2), (R2,C3), (R1,C3) da, azken soluzio hau optimo lokala izanik. Bigarren estrategia gutziatsua da – *greedy*-a, alegia –; aukeratzen dugun hurrengo soluzioa ingurunean dagoen onena izango da beti. Estrategia hau erabiliz pausu bakar batean (R4,C1) optimo lokalera ailegatzeko gara.

Erakarpen-arroa, definizioan ikus daitekeen legez, helburu funtzioaren, ingurunearen eta algoritmoaren araberakoa da. Alde batetik, ingurunearen eta helburu funtzioaren eragina begi bistakoa da. Bestetik, algoritmoak, ingurunea nola aztertzen den eta, bereziki, zein soluzio aukeratzen den ezartzen du; ondorioz, egiten dugun ibilbidean eragin handia izan dezake, 3 irudian aukeraketa estrategiek duten eragina ilustratzen da.

2 Bilaketa lokala

Bilaketa lokalean oinarritutako edozein algoritmoren errendimendua, kodeketaren eta ingurunearen aukeraketaz gain, beste zenbait elementutan ere oinarritzen da. Lehenik eta behin, hasierako soluzioa nola aukeratzen dugun erabakitzea garrantzitsua da, aurreko atalean ikusi dugun moduan horren arabera optimo lokal batean edo bestean amaituko baita bilaketa. Bigarren oinarritzko elementua ingurunekeko soluzioen aukeraketa egiteko aplikatzen den estrategia da. Uneko soluzioaren ingurunean hainbat soluzio izango ditugu, baina, zein aukeratu dugu hurrengo soluzioa izateko? Azkenik, gelditze irizpideak ere kontutan hartu beharreko faktoreak dira. Bilaketa lokala optimo lokal bat topatzen dugunean amaitzen da; dena dela, beste edozein algoritmotan bezala, denboran edota ebaluazio kopuruan oinarritutako gelditze irizpideak ere proposa ditzakegu³. 2 algoritmoan oinarritzko bilaketa lokalaren sasikodea ikus daiteke.

Sasikodean dagoen algoritmoa `metaheuR` paketeko `basic.local.search` funtzioak inplementatzen du. Funtzio honek zenbait parametro ditu, batzuk algoritmoarekin zerikusia dutenak eta beste batzuk problemari eta exekuzioari lotuta daudenak. Paketearen dauden metaheuristika guztiek antzerako egitura izango dutenez, pausuz pausu aztertuko ditugu parametro hauek. Gauzak honela, parametroak hiru motakoak dira:

- Problemari lotutako parametroak - Bilaketa gidatzeko helburu funtzio bat behar dugu. Funtzio hau `evaluate` parametroaren bidez pasatuko diogu algoritmoari. Algoritmoen inplementazioa orokorra denez,

³Informazio gehiago R-ren laguntzan duzu; `?basic.local.search` teklatu laguntza zabaltzeko.



Oinarrizko bilaketa lokala

```
1 input:  $f$  helburu funtzioa,  $s_0$  hasierako soluzioa,  $N$  ingurune funtzioa
2 output:  $s^*$  soluzio optimoa
3  $s^* = s_0$ 
4 do
5    $H = \{s' \in N(s^*) | f(s') < f(s^*)\}$ 
6   if  $|H| > 0$ 
7     Aukeratu  $H$ -n dagoen soluzio bat  $s'$ 
8      $s^* = s'$ 
9   fi
10 while ( $|H| > 0$ )
```

Algoritmoa 2.1: Oinarrizko bilaketa lokalaren sasikodea. Uneko soluzioaren ingurunean fitness-a hobetzen duen soluzio bat bilatzen dugu. Horrelakorik badago, uneko soluzioa ordezkatzen dugu; ez badago, bilaketa amaitzen da.

gerta daiteke problema batzuetarako bideraezinak diren soluzioak agertzea. Problema mota hauekin lan egin ahal izateko, **metaheuR** packageak soluzioen bideragarritasuna aztertu eta soluzio bideraezinak konpontzeko funtzioak parametro gisa sartzea ahalbidetzen du. Funtzio hauek problema bakoitzeko ezberdinak izango dira eta algoritmoari **valid** eta **correct** parametroen bidez pasatuko dizkiogu, hurrenez hurren.

- Exekuzio kontrola - Badaude exekuzioaren zenbait aspektu kontrola ditzakegunak. Lehenik eta behin, algoritmoari baliabide konputazionalak mugatu diezazkiokegu, denbora, soluzio berrien ebaluazio kopurua edota iterazio kopurua finkatuz. Hau **CResource** objektuen **cresource** parametroaren bidez kontrola dezakegu, bertan algoritmoak eskuragarri dituen baliabideak definituz. Horrez gain, **basic.local.search** funtzioak, algoritmoak gauzatzen duen bilaketaren progresioa bistaratzeko aukera ematen digu **verbose** parametroaren bidez. Era berean, progresioa taula batean gorde dezakegu, **do.log** parametroaren bidez.
- Bilaketaren parametroak - Bilaketa lokala aplikatzeko hiru gauza behar ditugu, hasierako soluzioa, ingurune definizio bat eta inguruneko soluzio bat aukeratzeko prozedura. Hiru elementu hauek **initial.solution**, **neighborhood** eta **selector** parametroen bidez ezarri beharko ditugu. Horez gain, soluzio bideraezinak daudenean, hiru aukera ditugu, bideraezin diren soluzioak onartu, deskartatu edo konpontzea. Zein aukera erabili **non.valid** parametroaren bidez adiraziko dugu.

Jarraian **basic.local.search** funtzioaren eta bere parametro guztien erabilera adibide baten bidez aztertuko dugu. Adibiderako grafoen koloreztetze-problema bat sortuko dugu **graph.coloring.problem** funtzioa erabiliz eta ausazko grafo bat hautatuz. Honela, **gcp** objektuak problemaren ebaluazio funtzioa eta soluzio bideragarriekin tratatzeko funtzioak gordeko ditu.

```
> n <- 25
> rnd.graph <- random.graph.game(n = n , p.or.m = 0.25)
> gcp <- graph.coloring.problem (graph = rnd.graph)
```

Orain, algoritmoari emango dizkiogun baliabideak mugatuko ditugu. Gehienez, algoritmoak 10 segundu, helburu funtzioaren $100n^2$ ebaluazio edo algoritmoaren $100n$ iterazio erabili ahalko ditu.

```
> resources <- cresource(time = 10 , evaluations = 100*n^2 , iterations = 100*n)
```

Azkenik, bilaketarekin loturiko parametroei dagokienez, hasierako soluzio gisa soluzio tribiala sortuko dugu, non nodo bakoitzak kolore bat duen. Horrez gain, Hamming distantzian oinarritutako ingurune objektua sortuko dugu. Objektu honek ingurunea aztertu eta harekin lan egiteko beharrezko funtzioak gordeko ditu.



```
> colors <- paste("C",1:n,sep="")
> initial.solution <- factor(colors, levels = colors)
> h.ngh <- hammingNeighborhood(base = initial.solution)
```

Dena prest daukagu bilaketa abiaratzeko ...

```
> bls <- basic.local.search(evaluate = gcp$evaluate , valid = gcp$is.valid ,
+                           correct = gcp$correct, initial.solution = initial.solution ,
+                           neighborhood = h.ngh , selector = first.improvement.selector ,
+                           non.valid = 'correct' , resources = resources)
```

```
## Running iteration 1 . Best solution: 25
## Running iteration 2 . Best solution: 24
## Running iteration 3 . Best solution: 23
## Running iteration 4 . Best solution: 22
## Running iteration 5 . Best solution: 21
## Running iteration 6 . Best solution: 20
## Running iteration 7 . Best solution: 19
## Running iteration 8 . Best solution: 18
## Running iteration 9 . Best solution: 17
## Running iteration 10 . Best solution: 16
## Running iteration 11 . Best solution: 15
## Running iteration 12 . Best solution: 14
## Running iteration 13 . Best solution: 13
## Running iteration 14 . Best solution: 12
## Running iteration 15 . Best solution: 11
## Running iteration 16 . Best solution: 10
## Running iteration 17 . Best solution: 9
## Running iteration 18 . Best solution: 8
## Running iteration 19 . Best solution: 7
## Running iteration 20 . Best solution: 6
```

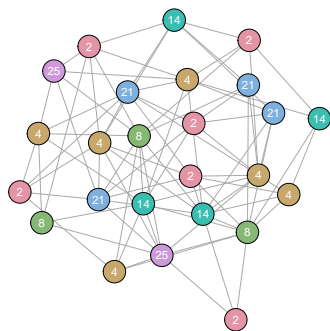
```
> bls
```

```
## RESULTS OF THE SEARCH
## Best solution's evaluation: 6
## Algorithm: Basic Local Search
## Resource consumption:
## Time: 3.850525
## Evaluations: 6339
## Iterations: 19
## None of the resources completely consumed
##
## You can use functions 'optima', 'parameters' and 'progress' to get the list of optimal
solutions, the list of parameters of the search and the log of the process, respectively
```

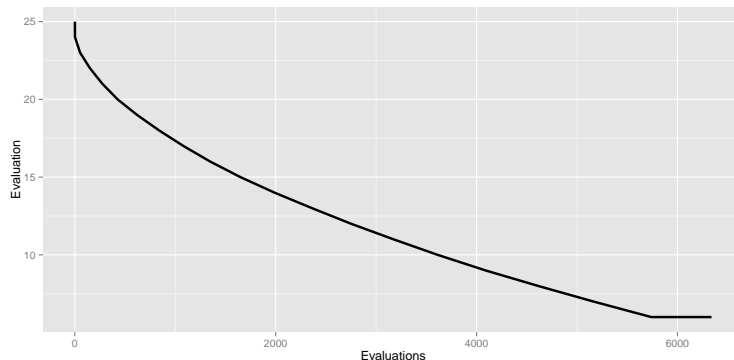
```
> final.solution <- optima(bls)[[1]]
> as.character(unique(final.solution))

## [1] "C2" "C4" "C8" "C14" "C21" "C25"
```

```
> plot.gpc.solution <- gcp$plot
> plot.gpc.solution(solution = final.solution , node.size = 15 , label.cex = 0.8)
```



(a) Problemaren soluzioa



(b) Bilaketaren progresioa

Irudia 4: Grafoen koloreztatzeko-problemarako bilaketa lokalak topatutako soluzioa eta egindako bilaketaren progresioa. Bigarren grafiko honetan, X ardatzak ebaluazio kopurua adierazten du eta Y ardatzak uneko soluzioaren ebaluazioa –soluzioak erabiltzen dituen kolore kopurua, alegia–.

Bilaketa lokalak –eta, oro har, beste gainontzeko algoritmoek– objektu berezi bat itzultzen dute, `MHResult` klasekoa. Bertan dagoen informazioa funtzio sorta baten bitartez lor daiteke; este baterako, `optima` funtzioak lortutako soluzio optimoa(k) itzultzen du, zerrenda batean. Bilaketa lokalak soluzio bakarra itzultzen duenez, soluzio hori listaren 1. posizioan egongo da. Soluzioa grafikoki bistaratzeko `graph.coloring.problem` funtzioak itzultzen duen `plot` funtzioa erabil daiteke. Gainera, bilaketaren progresioa ere bistara dezakegu, `plot.progress` funtzioa erabiliz. 4 irudiak problemarako soluzioa eta bilaketaren progresioa jasotzen ditu.

```
> plot.progress(bls , size=1.1 ) + labs(y="Evaluation")
```

Jarraian, bilaketa lokalean berebiziko garrantzia duten bi aspektu landuko ditugu: hasierako soluzioaren esleipena eta ingurunearen aukeraketa.

2.1 Hasierako soluzioaren aukeraketa

Lehen aipatu bezala, bilaketa lokala soluzio batetik abiatuko da beti. Bilaketa nondik hasten den oso garrantzitsua da, horren arabera optimo lokal batean edo bestean amaituko baita bilaketa. Adibidez, hau argi ikusten da 2 irudian; (R1,C6) soluziotik hasten badugu bilaketa (C2,R5) soluzioan amaituko da. Gauza bera gertatzen da optimo lokaletik bertatik – (C2,R5) –, goian dagoen soluziotik – (C1,R5) – edo bere eskuinean dagoen soluziotik – (C2,R6) – hasten bada prozesua. Beste edozein soluzio aukeratzen badugu, berriz, optimo globalera helduko gara.

Hau ikusirik, bi dira bilaketa lokala hasieratzeko erabiltzen diren estrategia ohikoenak:

- **Ausazko soluzioak sortu** - Ausaz aukeratzen da bilaketa espazioan dagoen soluzio bat eta hortik hasten da bilaketa. Metodo honen abantaila bere sinpletasuna da, ausazko soluzioak sortzea, kasu orokorrean, erraza izaten baita. Problema murritzeta asko dituen kasuetan ordea, premisa honek ez du balio, baliozko ausazko soluzioak sortzea asko zaildu baitaiteke. Hala ere, estrategia honek alde txarrak ere baditu; alde batetik, hasierako soluzioa txarra bada, bilaketa prozesua luzea izan daiteke eta, bestetik, algoritmoa aplikatzen dugun bakoitzean emaitza, oro har, ezberdina izango da.
- **Soluzio onak eraiki** - Lehenengo kapituluan ikusi genuen problema bakoitza ebazteko metodo heuristiko espezifikoak diseina daitezkeela. Oro har, metodo hauek pausuz pausu eraikitzen dituzte soluzioak, urrats

bakoitzean aukera guztietatik onena aukeratuz – ingelesez metodo hauei *constructive greedy* deritze, hau da algoritmo eraikitzaile gutziatsuak edo jaleak –. Nahiko soluzio onak lortu arren, hauek ez dira zertan optimoak izan⁴ eta, beraz, lortutako soluzioak bilaketa lokala hasieratzeko erabil daitezke. Estrategia hauek ausazko soluzioetatik abiatzeak baino emaitza hobeak lortzen ditu normalean, bilaketak iterazio gutxiago behar izaten baititu; konputazionalki ordea, garestia da.

2.2 Inguruneke soluzioaren aukeraketa

Behin inguruneke soluzioen multzoa definiturik dugula, hurrengo pausua soluzio horien artean bat aukeratzeko irizpidea ezartzea da. Lehen aipatu bezala, bi dira, nagusiki, erabiltzen diren estrategiak. Lehenengo estrategian inguruneke soluzioak banan banan analizatzen dira eta fitness-a hobetzen duen lehenengo soluzioa aukeratzen da. Hurbilketan honetan, inguruneke soluzioen «ordenazioa» oso garrantzitsua da, uneko soluzioa hobetzen duen lehenengo soluziora mugituko baikara. Bigarren estrategiak ingurune osoa arakatzen du eta fitness-a gehien hobetzen duen soluzioa aukeratzen du. Oro har, inguruneak txikiak direnean bigarren hurbilketa da interesgarriena baina, inguruneak handiak direnean, kostu konputazionala dela eta, bideraezina gerta daiteke estrategia hau.

Adibidea 2.1 *Demagun problema baterako soluzioak bektore bitarren bidez kodetzen ditugula. Uneko soluzioa (0, 1, 1, 0, 1) da dagokion fitness-a 25 delarik. Ingurunea definitzeko (2) ekuazioan dagoen funtzioa erabiltzen badugu, inguruneke soluzioak hauek izango dira:*

- $s_1 = (1, 1, 1, 0, 1); f(s_1) = 30$
- $s_2 = (0, 0, 1, 0, 1); f(s_2) = 24$
- $s_3 = (0, 1, 0, 0, 1); f(s_3) = 5$
- $s_4 = (0, 1, 1, 1, 1); f(s_4) = 27$
- $s_5 = (0, 1, 1, 0, 0); f(s_5) = 29$

Inguruneke soluzioak lortzeko posizio bakoitzeko balioa banan-banan aldatu behar dugu. Lehenengo posiziotik abiatzen bagara, s_2 soluzioa izango da fitness-a hobetzen duen lehenengo soluzioa, bere ebaluazioa 24 baita. Azken posiziotik abiatzen bagara, berriz, s_3 soluzioarekin geldituko ginateke, ebaluazioa 5 baita. Kasu bakoitzean soluzio ezberdina aukeratu dugu lehenengo pausu honetan, hortaz, hurrengo urratsean izango dugun ingurunea ere ezberdina izango da. Hori dela eta, inguruneke soluzioen azterketa orden desberdinetan eginez, azken soluzioa ezberdina izan daiteke. Inguruneke soluziorik onena aukeratzen ordea, ordenak ez du garrantziarik eta beti soluzio berdina topatuko dugu, berdinketarik ez badago betiere.

Adibidean, soluzioen kodeketarekin zerikusia duen ordena erabiltzen da inguruneke soluzioak lortzeko. Horren ordez, esplorazioa ausaz ere egin daiteke.

Jarraian azaltzen den kodean ingurunearen azterketan ordenak duen eragina erakusten da. Lehenik eta behin, TSPLib repositorioan dagoen problema bat kargatuko dugu, **metaheuR** paketeko **tsplib.parser** funtzioa erabiliz. TSPLib repositorioan TSP problemaren zenbait adibide ezberdin topa ditzakegu. Erabiliko dugun problemak Babariako 29 hiri izango ditugu. Hasierako soluzio gisa identitate permutazioa hartuko dugu.

```
> url <- system.file("bays29.xml.zip", package = "metaheuR")
> cost.matrix <- tsplib.parser(url)
```

```
## Processing file corresponding to instance bays29: 29 cities in Bavaria, street distances
(Groetschel, Juenger, Reinelt)
```

```
> n <- dim(cost.matrix)[1]
```

⁴Ez optimo globalak eta ezta lokalak ere.



```
> tsp.babaria <- tsp.problem(cost.matrix)
> csol <- identity.permutation(n)
> eval <- tsp.babaria$evaluate(csol)
```

Problema definitu ostean, hasierako soluzioaren ingurunea sortuko dugu. *2-opt* ingurunea aukeratu dugu, ausazko eta ez-ausazko esplorazioak hautatuz. Lehenengoan inguruneko soluzioak ausazko orden batean aztertuko dira eta bigarrenetan, ordean, kodeketaren arabera orden zehatz bat jarraituko da ingurunea arakatzeko.

```
> ex.nonrandom <- exchangeNeighborhood(base = csol , random = TRUE)
> ex.random <- exchangeNeighborhood(base = csol , random = FALSE)
```

Bilaketaren pausu bakoitzean inguruneko helburu funtzioa hobetzen duen lehenengo soluzioa aukeratzen badugu, hautatutako bi ordenazioak erabiliz emaitza ezberdinak lortuko ditugu. Ingurunea modu honetan arakatzeko, *first.improvement.selector* funtzioa erabili dezakegu. Funtzio honek, uneko soluzioaren ingurunea arakatu du eta fitness-a hobetzen duen lehenengo soluzioa itzultzen du.

```
> first.improvement.selector(neighborhood = ex.nonrandom , evaluate = tsp.babaria$evaluate ,
+                             initial.solution = csol , initial.evaluation = eval)$evaluation
## [1] 5684
```

```
> first.improvement.selector(neighborhood = ex.random , evaluate = tsp.babaria$evaluate ,
+                             initial.solution = csol , initial.evaluation = eval)$evaluation
## [1] 5478
```

Inguruneko soluziorik onena aukeratzen badugu, hautatutako bi ordenazioak erabiliz emaitza bera lortuko dugu, kasu guztietan inguruneko soluzio guztiak aztertzen baitira. Ingurunea aztertzeko estrategia hau *greedy.selector* funtzioak inplementatzen du.

```
> greedy.selector(neighborhood = ex.nonrandom , evaluate = tsp.babaria$evaluate ,
+                 initial.solution = csol , initial.evaluation = eval)$evaluation
## [1] 5034
```

```
> greedy.selector(neighborhood = ex.random , evaluate = tsp.babaria$evaluate ,
+                 initial.solution = csol , initial.evaluation = eval)$evaluation
## [1] 5034
```

Inguruneak handiak direnean ordenak oso eragin handia izan dezake eta, hortaz, heuristikoak erabil daitezke esplorazioa egiteko. Adibide gisa, Fred Glover-ek TSP problemarako proposatutako *ejection chains* [10] aipa daitezke. Gainera, metodo heuristikoez gain, tamaina handiko inguruneak era eraginkorrean zehazki aztertzeko algoritmoak ere badaude; hauetako adibide bat *dynasearch*[2] algoritmoa da.

2.3 Bilaketa lokalaren elementuen eragina

Ikusi dugun bezala, bilaketa lokal arrunt bat aplikatzeko hiru aspektu aztertu behar ditugu. Lehenengoa, hasierako soluzioaren aukeraketa, bigarrena, erabiliko dugun ingurunearen diseinua eta, azkena, inguruneko soluzioaren aukeraketa. Atal honetan aspektu hauen eragina aztertuko dugu adibide baten bidez.

Zehazki, aurreko atalean aurkeztutako Babariako hirien problema erabiliko dugu. Problema honetarako bi hasierako soluzio erabiliko ditugu, bat ausazkoa eta bestea algoritmo eraikitzaileak itzultzen duena.



```
> rnd.sol <- random.permutation(n)
> greedy.sol <- tsp.greedy(cmatrix = cost.matrix)
> tsp.babaria$evaluate(rnd.sol)
```

```
## [1] 6360
```

```
> tsp.babaria$evaluate(greedy.sol)
```

```
## [1] 2307
```

Ikusi daitekeenez, algoritmo eraikitzaileak (`tsp.greedy`) ematen duen soluzioa ausazkoa baino askoz ere hobea da. Bi soluzio hauek hasierako soluzio gisa hartuz, bilaketa lokala aplikatu ahal dugu, *swap* ingurunea erabiliz. Gainera, pausu bakoitzean, inguruneko soluziorik onena aukera dezakegu (*greedy*) edo, bestela, uneko soluzioaren fitness-a hobetzen duen lehenengo soluzioa hartu (*first improvement (fi)*). Honenbestez, lau bilaketa ezberdin exekutatu ditugu.

```
> eval <- tsp.babaria$evaluate
> swp.ngh.rnd <- swapNeighborhood (base = rnd.sol)
> swp.ngh.greedy <- swapNeighborhood (base = greedy.sol)
> swap.greedy.rnd.sol <- basic.local.search(evaluate = eval , initial.solution = rnd.sol ,
+                                          neighborhood = swp.ngh.rnd ,
+                                          selector = greedy.selector , verbose = FALSE)
>
> swap.fi.rnd.sol <- basic.local.search(evaluate = eval , initial.solution = rnd.sol ,
+                                     neighborhood = swp.ngh.rnd ,
+                                     selector = first.improvement.selector, verbose = FALSE)
>
> swap.greedy.greedy.sol <- basic.local.search(evaluate = eval , initial.solution = greedy.sol ,
+                                             neighborhood = swp.ngh.greedy ,
+                                             selector = greedy.selector, verbose = FALSE)
>
> swap.fi.greedy.sol <- basic.local.search(evaluate = eval , initial.solution = greedy.sol ,
+                                         neighborhood = swp.ngh.greedy ,
+                                         selector = first.improvement.selector, verbose = FALSE)
```

Azter dezagun zer nolako hobekuntza lortu dugun bilaketa lokalarekin, ausazko soluziotik abiatzen garenean.

```
> tsp.babaria$evaluate(rnd.sol) - evaluation(swap.greedy.rnd.sol)
```

```
## [1] 1936
```

```
> tsp.babaria$evaluate(rnd.sol) - evaluation(swap.fi.rnd.sol)
```

```
## [1] 1317
```

Ikusi daitekeenez bilaketa lokalaren bidez, topatutako soluzioa hasierakoa baino askoz ere hobea da. Are gehiago, bilaketa prozesuko pausu bakoitzean inguruneko soluziorik onena aukeratzen badugu, hobekuntza handiagoa da. Halere, kontutan hartu behar da ebaluazio kopurua ere handiagoa dela kasu honetan.

```
> consumed.evaluations(resources(swap.greedy.rnd.sol))
```

```
## [1] 337
```

```
> consumed.evaluations(resources(swap.fi.rnd.sol))
```

```
## [1] 276
```



Algoritmo eraikitzailearekin lortutako hasierako soluzioa, ausazko soluzioa baina hobea dela ikusi dugu. Hala ere, soluzio horri bilaketa lokala aplikatuz, soluzio oraindik hobea lortuko dugu, nahiz eta kasu honetan hobekuntza hain handia ez izan.

```
> tsp.babaria$evaluate(greedy.sol) - evaluation(swap.greedy.greedy.sol)

## [1] 56
```

Inguruneke soluzio guztietatik onena hartzeak emaitza hobeak ematen ditu –ausazko soluziotik abiatzen garenean, behintzat–, bilaketa espazioa sakonago aztertzen delako. Bilaketa sakonagoa egiteko beste era bat, *swap* ingurunearen ordez *2-opt* ingurunea erabiltzean datza. Izan ere, lehen ikusi dugun bezala, *2-opt* ingurunea *swap* ingurunea baino askoz ere handiagoa da.

```
> ex.ngh.rnd <- exchangeNeighborhood (base = rnd.sol)
> ex.greedy.rnd.sol <- basic.local.search(evaluate = eval , initial.solution = rnd.sol ,
+                                       neighborhood = ex.ngh.rnd ,
+                                       selector = greedy.selector , verbose = FALSE)
>
> tsp.babaria$evaluate(rnd.sol) - evaluation(ex.greedy.rnd.sol)

## [1] 3951

> consumed.evaluations(resources(ex.greedy.rnd.sol))

## [1] 11775
```

Ikus daitekeen bezala, hobekuntza handiagoa da baina, inguruneak handiagoak direnez, baita ebaluazio kopurua ere.

3 Bilaketa lokalaren hedapenak

Aurreko atalean ikusi dugun legez, bilaketa lokala soluzioak areagotzeko prozedura egokia izan arren, desabantaila handi bat du; optimo lokaletan trabatuta gelditzen da. Arazo hau saihesteko –soluzioen dibertsifikazioa suspertzeko, alegia – bilaketa lokalak dituen lau aspektu nagusietan aldaketak sar ditzakegu bilaketan zehar: hasierako soluzioan, ingurunearen definizioan, inguruneke soluzioen aukeraketan eta helburu funtzioaren definizioan. Hurrengo ataletan hauetako elementu bakoitzean aldaketak egiten dituzten algoritmo batzuk aurkeztuko ditugu.

3.1 Hasieraketa anizkoitza

Bilaketa lokala aplikatzean, soluzio bakoitzetik abiatuz optimo lokal batera heltzen gara; soluzio ezberdinetatik abiatzen bagara, optimo lokal ezberdinetara heldu gaitzke. Ideia hau da, hain zuzen ere, hasieraketa-anizkoitzeko bilaketa lokalak – *Multistart Local Search*, ingelesez– inplementatzen duen (3.1 sasikodean ikusi daiteke). Algoritmoan agertzen diren *generate_random_solution* eta *local_search* funtzioetan dago prozeduraren mamia, beraiek karakterizatuko baitute algoritmoaren performantzia. Lehenengoak, bilaketa espazioaren esplorazioa burutzen du. Bigarrena, aldiz, bere izenak adierazten duen bezala, soluzioen areagotzeaz arduratzen da.

Hasteko, ausazko soluzioak sortzeko hainbat aukera ditugu. Horietako bat, soluzioak uniformeki ausaz sortzea da, hots, iterazio bakoitzean probabilitate berdinarekin espazioko edozein soluzio aukeratu dugu eta bilaketa lokala soluzio horretatik hasiko dugu. Uniformeki ausazko soluzioetatik abiatzea, gehienetan, aukera erraza da; alabaina, ez da oso estrategia adimentsua. Gainera, murritzeta askoko problemetan ausazko soluzio bideragarriak sortzea zaila izan daiteke. Bilaketa hasieratzeko soluzio «onak» eraikitzeke prozedura bat izanez gero, bi arazo hauek saihestu ditzakegu. Hain juxtu, haxe da hurrengo atalean ikusiko ditugun ILS eta GRASP algoritmoak egiten dutena.



Hasieraketa-anizkoitzeko bilaketa lokal orokorra

```

1 input:  $f$  helburu funtzioa
2 input:  $random\_solution$ ,  $stop\_criterion$  eta  $local\_search$  funtzioak
3 output:  $s^*$  soluzioa optimoa
4  $s = generate\_random\_solution$ 
5 while  $!stop\_criterion$ 
6      $s' = random\_solution$ 
7      $s'' = local\_search(s')$ 
8     if  $(f(s'') < f(s))$   $s = s''$ 
9 done

```

Algoritmoa 3.1: Hasieraketa anizkoitza erabiltzen duen bilaketa lokalaren hedapenaren sasikode orokorra

Bilaketa Lokala Iteratua (ILS)

```

1 input:  $f$  helburu funtzioa
2 input:  $accept$ ,  $perturb$ ,  $stop\_criterion$  eta  $local\_search$  funtzioak
3 input:  $s_0$  hasierako soluzioa
4 output:  $s^*$  soluzioa
5  $s = local\_search(s_0)$ 
6  $s^* = s$ 
7 while  $!stop\_criterion$ 
8      $s' = perturb(s)$ 
9      $s'' = local\_search(s')$ 
10    if  $(accept(s''))$   $s = s''$ 
11    if  $(f(s'') < f(s^*))$   $s^* = s''$ 
12 done

```

Algoritmoa 3.2: Bilaketa Lokala Iteratuaren (ILS) sasikodea

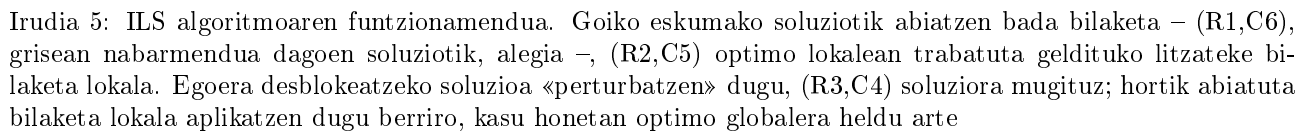
3.1.1 Bilaketa Lokala Iteratua (ILS)

Bilaketa lokala berrabiarazteko uniformeki ausazko soluzioak erabili beharrean, ILS – *Iterated Local Search*, ingelesez – algoritmoak uneko optimo lokala hartuko du oinarritzat. Ideia oso sinplea da; optimo lokal batean trabaturik gelditzen garenean, uneko soluzioa «perturbatu» eta bertatik bilaketarekin jarraituko dugu. Optimo lokal berri batera heltzen garenean, soluzio hau onartuko dugunetz erabaki behar dugu. 3.2 algoritmoan ILS-aren sasikode orokorra ikusi daiteke.

Algoritmo hau zehazteko bi prozedura berri definitu behar ditugu:

- **Perturbazioa** - Hasteko, optimo lokal batean trabaturik geratzean, hau nola perturbatuko den erabaki behar da. Soluzio baten inguruneke soluzio guztiak antzekoak direnez, optimo lokaletatik edo zehazki haien erakarpen-arroetatik ateratzeko, aldaketa nabarmenak egin behar dira (5 irudian adibide bat proposatzen da). Uneko soluzioa (R2,C5) izanik, perturbazio txikiegia egingo bagenu – (R1,C5) soluziora mugitzea, adibidez –, berriro optimo lokal berdinean amaituko litzateke bilaketa. Perturbazioa nahiko handia bada, uneko optimo lokalaren erakarpen-arrotik aterako gara eta, definizioz, beste optimo lokal batean trabaturik geldituko gara. Kontutan hartzekoa da ordea, perturbazio prozedurak itzultitako soluzioa erabat ausazkoa bada –hau da, perturbazioa oso handia bada –, ILS algoritmoa ausazko hasieraketa anizkoitzeko algoritmoa bilakatuko dela. Beraz, perturbazio tamainaren aukeraketak eragina izango du algoritmoaren portaeran.

Perturbazioa definitzean inguruneke soluzioak definitzeko erabiltzen diren operazio mota berberak edo



Gainera, soluzioak perturbatzeko prozedura aurreratuetan, bilaketaren «historia» ere erabil daiteke, soluzioaren zein osagai perturbatu eta zein ez erabakitzeko. Estrategia hauek «memoria» kontzeptua erabiltzen dute eta memoria mota ezberdinak soluzioak areagotzeko eta dibertsifikatzeko balio dezakete.

- metaheur paketea, `ILS-a iterated.local.search` funtzioan dago inplementatuta. Funtzio honen parametro gehienak `basic.local.search` funtzioaren berberak dira, inplementazioa funtzio horretan oinarritzen baita. Algoritmo honek ordea, hiru parametro berri izango ditugu:

- 16



parametro bat izan beharko du, *delta*, soluzio berriaren eta zaharraren fitness balioen arteko diferentzia jasoko duena.

- `num.restarts` - Optimo lokalak perturbatuz, bilaketa zenbat alditan berrabiarazi behar dugun esaten duen zenbaki osoa da.

Ikus dezagun adibide bat, TSPLib-eko problema bat erabiliz. Problema honetan Burma-ko 14 hirien arteko distantziak izango ditugu. Problema ebazteko ausazko soluzio batetik abiatuko dugu bilaketa. Ingurune gisa *2-opt* erabiliko dugu (trukaketa orokorrak, ez bakarrik elkar-ondokoak) eta soluzioak perturbatzeko eragiketa bera erabiliko dugu baina behin baino gehiagotan aplikatuz; soluzio bati operazio hau ausaz aplikatzeko `shuffle` funtzioa erabil dezakegu.

```
> f <- paste("http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95"
+           ,"/XML-TSPLIB/instances/burma14.xml.zip",sep="")
> burma.mat <- tsplib.parser(f)

## Processing file corresponding to instance burma14: 14-Staedte in Burma (Zaw Win)

> n <- ncol(burma.mat)
> burma.tsp <- tsp.problem(burma.mat)
>
> init.sol <- random.permutation(n)
> ngh <- exchangeNeighborhood(init.sol)
> sel <- greedy.selector
```

Segidan, optimo lokalak onartzeko irizpideak definituko ditugu. Kasu honetan, soluzioen arteko diferentzia 0 baina handiagoa izan beharko da, optimo lokal berria onartzeko. Hau da, optimo lokal berria aurrekoa baina hobea izan behar da.

```
> th.accpet <- threshold.accept
> th <- 0
```

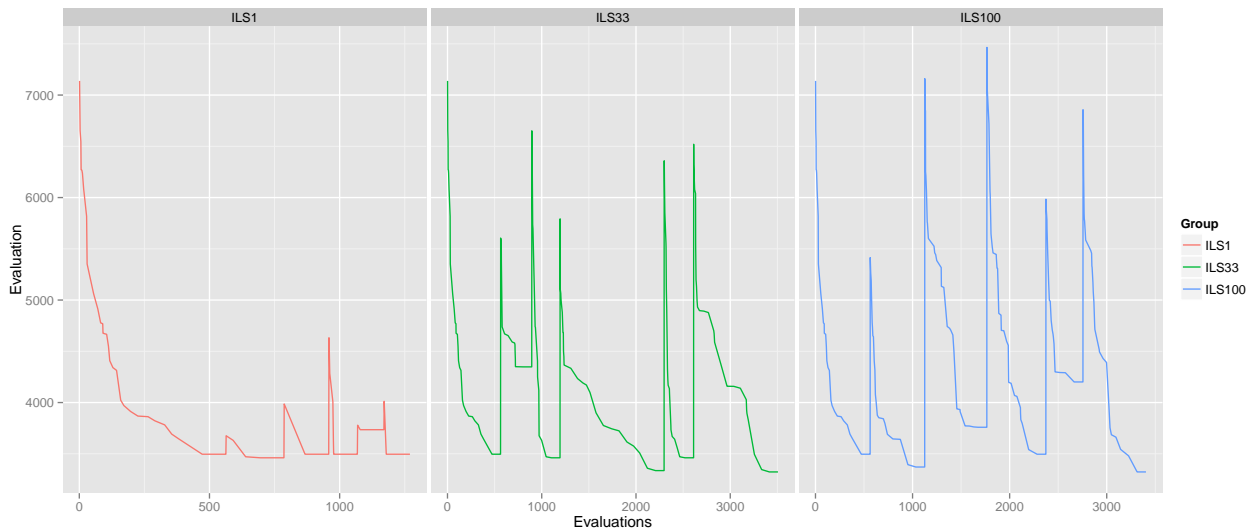
Perturbazio maila soluzioari aplikatuko dizkiogun trukaketa kopuruaren bidez kontrolatuko dugu. Beraz, trukaketa kopurua, gure perturbazio funtzioaren parametro bat izan beharko da⁵.

```
> set.seed(1)
> perturb.2opt <- function(solution , ratio , ...)
+   shuffle(permutation = solution , ratio = ratio)
```

Honekin guztiarekin ILS algoritmoa exekutatu dezakegu perturbazio maila ezberdinekin:

```
> ratio <- 0.01
> ils.1 <- iterated.local.search(evaluate = burma.tsp$evaluate ,
+                               initial.solution = init.sol ,
+                               neighborhood = ngh , selector = sel ,
+                               perturb = perturb.2opt , ratio = ratio ,
+                               accept = th.accpet , th = th ,
+                               num.restarts = r)
> ratio <- 0.25
> ils.25 <- iterated.local.search(evaluate = burma.tsp$evaluate ,
+                                initial.solution = init.sol ,
```

⁵`iterated.local.search` funtzioarekin (eta paketearen beste hainbat funtzioekin) arazorik ez izateko, pasatutako funtzioak ... argumentua izan behar du gutxienez, nahiz eta gero barruan ez erabili.



Irudia 6: ILS algoritmoaren progresioa 14 hiriko TSP problema batean. Ezkerretik eskuinera, perturbazioaren ratioak 0.01, 0.25 eta 1 dira.

5

```
+ neighborhood = ngh , selector = sel ,
+ perturb = perturb.2opt , ratio = ratio ,
+ accept = th.accpet , th = th ,
+ num.restarts = r)
> ratio <- 1
> ils.100 <- iterated.local.search(evaluate = burma.tsp$evaluate ,
+ initial.solution = init.sol ,
+ neighborhood = ngh , selector = sel ,
+ perturb = perturb.2opt , ratio = ratio ,
+ accept = th.accpet , th = th ,
+ num.restarts = r)
>
> plot.progress(result = list(ILS1 = ils.1 , ILS33 = ils.25 , ILS100 = ils.100)) +
+ facet_grid(. ~ Group , scales = 'free_x') + labs(y="Evaluation")
```

Hiru bilaketa hauen progresioak 5 irudian ikus daitezke. Hirurak soluzio berdinetik hasten dira eta, pausu bakoitzean inguruko soluziorik onena aukeratzen dutenez, lehenengo jaitsiera berdina da hiru grafikoetan. Lehenengo optimo lokala topatzen den momentuan, ordea, diferentziak hasten dira. Ezkerretik eskuinera, lehenengo grafikoan soluzioak posizioen %1 trukatzuz perturbatzen dira; guztira soluzioak 14 posizio dituztenez, trukaketa bakar bat egiten da. Erdiko grafikoan perturbazioa %25ekoa da, 3 trukaketa egiten dira, alegia. Azken grafikoan, berriz, 14 trukaketa egiten dira (posizioen %100). Perturbazio txiki bat erabiltzen dugunean, lortutako soluzioaren ebaluazioa optimo lokalaren antzerakoa da (agertzen diren jauziak ez dira oso altuak, alegia); geroz eta perturbazio handiagoa orduan eta diferentzia handiagoa soluzio berria eta optimoaren artean. Azken kasuan, perturbazioa oso handia da eta, beraz, optimo lokaletatik ateratzeko, soluzioak guztiz ausaz aukeratzen dira, hasieraketa anizkoitzeko algoritmoan bezala.

3.1.2 GRASP algoritmoa

Optimizazio problemak ebazteko ohikoa da metodo eraikitzaileak erabiltzea. Aurreko kapituluaren ikusi genuen bezala, algoritmo hauek soluzioa pausuz pausu eraikitzen dute, urrats bakoitzean aukera guztietatik onena

hautatuz. Era honetan, soluzio onak sortzen dira baina, hauek ez dute zertan optimoak izan, ez globalki eta ezta lokalki ere. Hori dela eta, behin soluzioa sortuta, bilaketa lokal bat erabil daiteke soluzioa areagotzeko. Alabaina, berdinketak egon ezean, metodo eraikitzaileek instantzia bakoitzeko soluzio bakarra eta beti berdina lortzen dute eta beraz hasieraketa bakarra ahalbidetzen dute.

Idea hau apur bat landuz, metodo eraikitzaileak soluzio bakarra sortu beharrean soluzio multzo bat sortzeko egoki ditzakegu. Eta ondoren, 3.1 algoritmoan agertzen den *random_solution* metodoak multzo horretatik ausazko soluzioak aterako ditu, bilaketa lokala hasieratzeko. Idea hau GRASP *Greedy Randomized Adaptive Search Procedure* algoritmoaren atzean dagoena da [4].

Ausazko soluzio onak eraikitzeko, pausu bakoitzean aukerarik onena aukeratu beharrean «hautagai zerrenda» bat izango dugu – *candidate list*, ingelesez –; algoritmoak zerrenda horretan dauden osagaiak ausaz aukeratuko ditu hasierako soluzioak eraikitzeko.

Adibidea 3.1 *Demagun motxilararen problema ebatzi nahi dugula. Oso sinplea den algoritmo eraikitzaile bat ondorengoa da: lehenik eta behin, kalkulatu motxilaran sartzen ditugun elementu bakoitzaren balioa/pisua ratioa eta gero, pausu bakoitzean, pisu-muga gaindiaz ez duten elementuetatik, ratorik handiena duena aukeratu.*

Algoritmo hau GRASP algoritmoaren ideia modu errezean egokitu daiteke. Algoritmoaren iterazio bakoitzean hasierako soluzio bat eraikiko dugu pausu bakoitzean, ratorik handiena duen elementua aukeratu beharrean ratio handiena duten α soluzioen artetik bat ausaz aukeratuz. Behin soluzioa eraikita, bilaketa lokala aplikatuko dugu lortutako soluzioa areagotzeko.

Edozein problemari GRASP algoritmoa aplikatzeko, adibidean planteatzen den hasierako soluzio onak sortzeko estrategiaren antzerako prozedura bat diseinatu eta inplementatu beharko dugu. Funtzio honen parametro gehienak problema bakoitzarentzat ezberdinak izango dira baina, gainera, hautagaien zerrendaren luzeera α proportzio baten bidez adierazi beharko dugu.

Adibide gisa, *metaheur* paketea motxilararen problema GRASP algoritmoaren bitartez ebatzi ahal izateko, *knap.GRASP* funtzioa izango dugu. Funtzio honetan, hasteko, zenbait datu atera eta aldagai batzuk hasieratzen dira. Besteak beste, elementurik gabeko soluzio «hutsa» sortuko dugu.

```
knap.GRASP <- function (weight , value , limit , cl.size = 0.25){
  size <- length(weight)
  ratio <- value / weight
  solution <- rep(FALSE , size)
  finished = FALSE
```

Soluzioa sortzeko, elementuak banan banan sartuko ditugu motxilaran, eta honetarako, begizta bat izango dugu, motxila beteta ez dagoen bitartean errepikatuko dena. Begiztaren lehenengo pausuan, motxilaran oraindik sartu gabeko elementuei atzematen diegu eta, uneko iterazioan, gure hautagai zerrendak izango duen tamaina kalkulatzeko (gogoratu hautagai zerrendaren tamaina urrats bakoitzean ditugun aukera kopuruaren proportzio bat bezala definitu dugula).

```
while (!finished){
  non.selected <- which(!solution)
  cl.n <- max(1, round(length(non.selected)*cl.size))
```

Orain, ratioak ordenatu ondoren, lehenengo elementuak hartzen ditugu hautagai zerrenda gisa, eta horietatik bat ausaz aukeratzen dugu.

```
cl <- sort(ratio[non.selected] , decreasing=TRUE)[1:cl.n]
selected <- sample(cl,1)
```

Bukatzeko, aukeratutako elementua soluzioan sartu eta aurrera jarraitzen da motxila betetzen ez den bitartean. Motxila bete egiten bada, sartutako azkeneko elementua atera eta begizta bukatu egingo da.



```

aux <- solution
aux[ratio == selected] <- TRUE
if (sum(weight[aux]) < limit){
  solution <- aux
}else{
  finished <- TRUE
}
}
solution
}

```

Sortu dugun funtzioa, GRASP algoritmoa, guztiz ausazkoa den hasieraketa anizkoitzarekin alderatzeko erabil dezakegu, `cl.size` parametroarekin jolastuz. Lehenik eta behin, ausazko motxilaren problema bat sortuko dugu. Kasu honetan 50 elementu izango ditugu eta hauen balioa bi multzotan banatuko dugu. Elementu erdien balioak 0 eta 10 arteko ausazko zenbakiak izango dira; beste erdien balioak, 0 eta 25 tartetik ausaz hautatuko ditugu. Kasu guztietan pisua balioarekiko proportzionala izango da, faktorea ausazko zenbaki bat izanik. Limite gisa, ausaz aukeratutako 10 elementuen pisuaren batura erabiliko dugu.

```

> n <- 50
> values <- c(runif(n/2) * 10, runif(n/2)*25)
> weights <- values * rnorm(n,1,0.05)
> limit <- sum(sample(weights , n/5))

```

GRASP eta hasieraketa anizkoitzeko bilaketa lokala erabiltzeko, `multistart.local.search` funtzioa erabil dezakegu. Orain arte ikusitakoen antzerakoa da funtzio hau, baina argumentu moduan hasierako soluzioa pasatu beharrean, soluzioak sortzeko funtzio bat pasatu behar diogu. Funtzioak parametro asko dituenez, sarrerako balioak antolatzeke beste era bat erabiliko dugu kodea irakurterrezagoa izateko. Zerrenda batean sartuko ditugu, izenak erabiliz eta gero R-ren `do.call` funtzioa erabiliko dugu.

```

> knp.problem <- knapsack.problem(weights , values , limit)
>
> args$evaluate <- knp.problem$evaluate
> args$valid <- knp.problem$is.valid
> args$correct <- knp.problem$correct
> args$non.valid <- 'discard'
>
> args$neighborhood <- flipNeighborhood(base = rep(FALSE , n))
> args$selector <- greedy.selector
>
> args$generate.solution <- knap.GRASP
> args$num.restarts <- 25
> args$weight <- weights
> args$value <- values
> args$limit <- limit

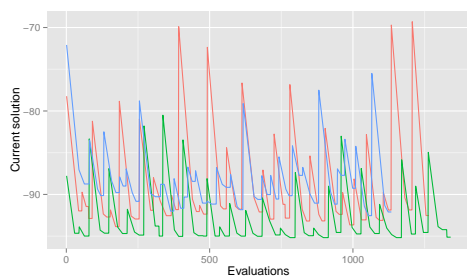
```

Behin parametro guztiak ezarrita, `cl.size` parametroari 3 balio ezberdin esleituko dizkiogu, 1, 0.5 eta 0.1. Lehenengo kasuan hautagai zerrendan aukera guztiak egongo direnez, soluzioak guztiz ausaz sortuko ditu, hots, ausazko hasieraketa anizkoitza erabiliko du bilaketa lokalak. Beste kasuetan, berriz, aukera guztietatik %50 eta %10 erabiliko ditu, hurrenez hurren.

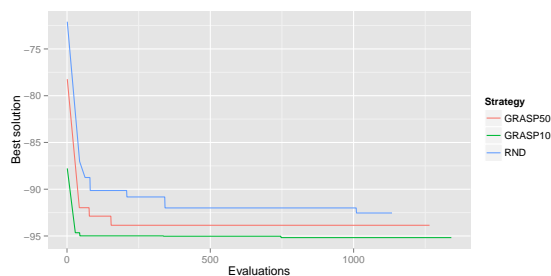
```

> rnd.ls <- do.call (multistart.local.search , args)
>

```



(a) Uneko soluzioa



(b) Soluziorik onena

Irudia 7: Hasieraketa anizkoitza estrategia ezberdinen konparaketa, motxilaren problema batean. Ezkerreko grafikoan uneko soluzioaren progresioa ikus daiteke. Eskumakoan, berriz, uneko soluziorik onenaren progresioa erakusten da.

```
> args$cl.size <- 0.50
> GRASP.50 <- do.call (multistart.local.search , args)
>
> args$cl.size <- 0.1
> GRASP.10 <- do.call (multistart.local.search , args)
```

7 irudian bilaketen progresioak ikus daitezke. Ezkerreko grafikoan uneko soluzioaren eboluzioa jasotzen da. Ikusi dezakegunez, berabiaratze bakoitzean, bilaketa soluzio txarragoetatik hasten da (gorago daudenak). GRASP algoritmoan, berriz, hasierako soluzioak hobeak dira, eta baita lortzen den emaitza ere. Hori argi ikusten da eskuineko grafikoan, non uneko soluziorik onenaren eboluzioa jasotzen den.

3.2 Inguruneko soluzioen hautaketa

Definizioz, optimo lokalen inguruko soluzio guztiak optimo lokala bera baino okerragoak dira; hortaz, bilaketak soluzio hauetara eramaten gaituenean, ez dugu soluzio hoberik aukeratzetik eta optimizazioa tratatuta gelditzen da. Egoera hauetan, bilaketa prozesuari amaiera ez emateko, inguruneko soluzioen hautaketa estrategia alda genezake, soluzio hoberik egon ezean, helburu funtzioa hobetzen ez duten soluzioak ere onartuz. Estrategia honi esker, okerragoak diren soluzio batzuetatik pasatuz, bilaketa espazioaren eskualde berrietara ailega gintezke.

Soluzio «txarragoak» aukeratzeko bi estrategia daude. Lehendabizikoa, fitness-a hobetzen ez duen soluzio bat aukeratzeko, sortutako «galera» kontutan hartzen da (era probabilistikoa zein deterministan egin daiteke). Algoritmorik ezagunena *suberaketa estokastikoa* –*simulated annealing* [7, 11] ingelesez– izenekoa da, zeinek probabilitate-banaketa parametrikoki bat erabiltzen duen aukeraketa egiteko. Algoritmo honetan inspiratutako beste hainbat algoritmo proposatu dira literaturan, *demon algorithm* [9] eta *threshold accepting* [3, 8] algoritmoak, besteak beste.

Soluzio txarrak aukeratzeko bigarren estrategia mota Gloverrek 1986an proposaturiko *tabu bilaketa* [5] –*tabu search* ingelesez– algoritmoak erabiltzen duena da. Kasu honetan, fitness balioa hobetzen ez duten soluzioak aukeratzeko dira, baina bakarrik ingurune osoan helburua hobetzen duen soluziorik ez badago. Estrategia honen arriskua dagoeneko bisitatu ditugun soluzioak berriro bisitatzea da. Hala, zikloak saihesteko, tabu bilaketak azken aldiak bisitatutako soluzioak «memoria» batean gordetzen ditu.

Jarraian, bi algoritmo hauek (suberaketa simulatua eta tabu bilaketa), sakonki aztertuko ditugu.

3.2.1 Suberaketa Simulatua

Metalezko tresnen edo piezen sorrera prozesuan, metalek hainbat propietate gal ditzakete, euren kristal-egituraren eragindako aldaketak direla eta. Propietate horiek berreskuratzeko metalurgian «suberaketa» prozesua erabiltzen da; metal pieza behar adina berotzen da, gero astiro-astiro hozten uzteko. Temperatura igotzean meta-

Suberaketa Simulatua - Simulated Annealing

```

1 input: random_neighbor operadorea
2 input: update_temperature, equilibrium, stop_condition operadoreak
3 output:  $s^*$  topatutako soluziorik onena
4  $s^* = s$ 
5  $T = T_0$ 
6 while  $\neg stop\_condition$ 
7   while  $\neg equilibrium$ 
8      $s' = random\_neighbor(s)$ 
9      $\Delta E = f(s') - f(s)$ 
10    if  $\Delta E < 0$ 
11       $s = s'$ 
12      if  $(f(s) < f(s^*))$   $s^* = s$ 
13    fi
14    else
15       $e^{-\frac{\Delta E}{T}}$  probabilitatearekin  $s = s'$ 
16    done
17     $T = update\_temperature(T)$ 
18 done

```

Algoritmoa 3.3: Suberaketa Simulatuaren sasikodea

laren atomoen energia handitzen da eta, hortaz, beraien artean sortzen diren indar molekularrak apurtzeko gai dira; mugitzeko askatasun handiagoa dute, alegia.

Metala oso azkar hozten bada –tenplatzean egiten den bezala, adibidez– molekulak zeuden tokian «izoztuta» gelditzen dira. Honek metala gogortzen du, baina hauskorrakoa bihurtzen du, aldi berean. Suberatzean, berriz, metala poliki-poliki hozten da eta, ondorioz, molekulak astiro galtzen dute beraien energia –hots, abiadura–. Hozketa-abiadura motelari esker, molekulak euren kristal-egituraren «kokapen optimora» joaten dira, hau da, energia minimoko kristal-egitura sortzen da.

1983an Kirkpatrick-ek [7] eta bi urte geroago Cerny-k [11], suberaketaren prozesuan inspiratuta, optimizazio algoritmoak proposatu zituzten; Kirkpatrick-ek bere algoritmoari *simulated annealing*, suberaketa simulatua, izena eman zion eta haxe da gaur egun hedatuen dagoena.

Algoritmoaren funtzionamendua sinplea da oso. s soluzio batetik txarrakoa den s' soluzio batera mugitzeko, «energia» behar dugu; behar den energia bi soluzioen ebaluazioen arteko diferentzia izango da, hau da, $\Delta E = f(s') - f(s)$.

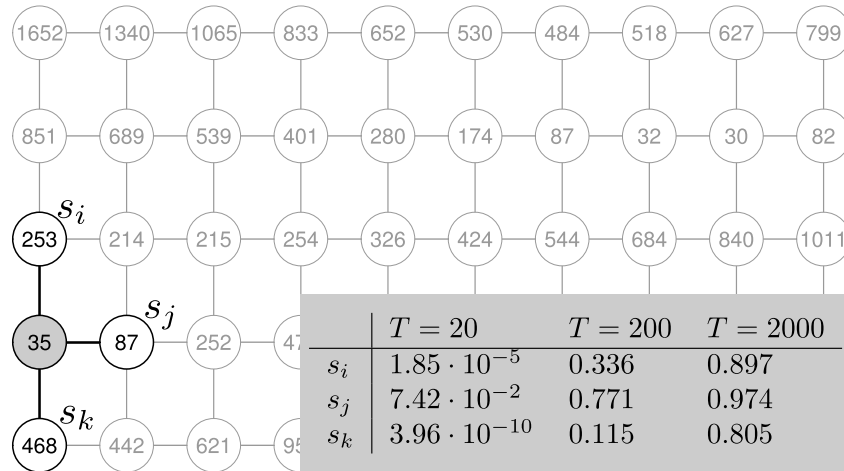
Energia-muga hau gainditzeko, sistemak energia behar du eta sistemaren energia «tenperatura»k neurtuko du. Beste era batean esanda, uneoro, sistemak T tenperatura izango du eta, zenbat eta tenperatura altuagoa, orduan eta errazagoa izango da energia-mugak gainditzea. Zehazki, soluzio batetik bestera mugitzeko behar den energia-muga gainditzen denetz erabakitzeo Boltzmann probabilitate-banaketan oinarritutako funtzio bat erabiltzen da:

$$P(\Delta E, T) = e^{-\frac{\Delta E}{T}}$$

Ekuaziotik ondorioztatu daitekeenez, uneo soluzioa baino fitness okerrakoa duen soluzio bat onartzeko probabilitatea, tenperaturarekiko proportzionala da, eta energia diferentziarekiko alderantziz proportzionala.

Aintzat hartzekoa da $\Delta E < 0$ denean funtzioaren balioa 1 baino handiagoa dela. Izatez, ekuazioa bakarrik energia diferentzia positiboa denean erabiltzen da, negatiboa bada, s' soluzioa hobe baia eta, ondorioz, beti onartzen da.

Suberaketaren gakoa hozte-abiaduran datza, hau da, tenperaturaren eguneraketan. Izan ere, hasieran T



Irudia 8: Soluzioak onartzeko probabilitateen adibideak. Uneko soluzioa grisez nabarmendua dagoena izanik, hiru soluzio ditugu ingurunean, s_i , s_j eta s_k . Taulak hiru temperatura ezberdinekin soluzio bakoitza onartzeko probabilitateak jasotzen ditu. Ikus daitekeenez, temperatura baxua denean, edozein soluzio aukeratzeko probabilitatea oso baxua da; temperatura oso altua denean, berriz, edozein soluzio hartuta litekeena oso probalea da.

balio handiak erabiliko ditugu, ia edozein soluzio onartu ahal izateko, eta gero, astiro-astiro, T txikiagotuko dugu, gelditzeko baldintza bete arte. 3.3 algoritmoan suberaketa simulatuaren sasikodea ikus daiteke.

Suberaketa simulatua oinarritutako algoritmoak diseinatzean lau aspektu hartu behar dira kontutan:

- **Hasierako temperatura** - Altuegia bada, hasierako iterazioetan *random walk*, hau da, ausazko ibilbide bat jarraituko dugu; baxuegia bada, berriz, bilaketa oinarritzko bilaketa lokala bihurtuko da. 8 irudian adibide bat ikus daiteke. $T = 20$ denean, nahiz eta fitness-en arteko diferentzia txikia izan, soluzioa onartzeko probabilitatea txikia da; $T = 2000$ denean ebaluazioen arteko diferentzia handia izan arren, oso probalea da soluzioak onartzea. $T = 200$ denean, berriz, optimo lokaletik atera gaitezke, probabilitate handiarekin, s_j aukeratuz baina tarteko temperatura honekin oso soluzio txarrak onartzea zaila izango da.

Tenperatura hasieratzeko bi estrategia erabili ohi dira. Lehendabizikoa hasierako temperatura oso altua aukeratzeko da. Dibertsifikazio ikuspegitik interesgarria izan arren, estrategia honekin bilaketak asko luzatu daitezke eta konputazionalki garestia izan daiteke. Bigarren strategiaren funtsa bilaketa espazioaren itxura aztertzean datza, ingurunean dauden soluzioen arteko diferentziak nolakoak diren jakiteko. Informazio hau onarpen ratio edo probabilitate ezagun bat lortzeko behar dugun temperatura finkatzeko erabili daiteke [6, 1], goi eta behe mugekin egin degun antzera.



Adibidea 3.2 Demagun 10 tamainako TSP-aren instantzia bat ebatzi nahi dugula. Kostu matrizeko baliorik handiena – hau da, bi hirien arteko distantziarik handiena – 7.28 da. Problemarako soluzio guztietan 10 hiri izango ditugu eta, hortaz, soluzio guztien ebaluazioa matrizean dauden 10 elementuren batura izango da. Hori dela eta, $f_g = 7.28 \cdot 10 = 78.2$ problema honen fitness-aren goi-muga bat da^a. Era berean, matrizeko distantziarik txikiena 2.5 izanik, behe-muga kalkula dezakegu: $f_b = 2.5 \cdot 10$.

Demagun, edozein soluzio aukeratzeko hasierako probabilitatea 0.75 dela. Orduan, bi muga hauek, f_g eta f_b , hasierako tenperatura kalkulatzeko erabil ditzakegu, edozein bi soluzioen arteko fitness-en diferentzia $f_g - f_b$ baino txikiagoa izango dela baitakigu:

$$P = 0.75 = e^{-\frac{f_g - f_b}{T_0}} = e^{-\frac{78.2 - 25}{T_0}}$$

$$T_0 = -\frac{78.2 - 25}{\ln(0.75)} = 184.93$$

Hasierako tenperatura 185 balioan finkatzen badugu, badakigu hasierako iterazioetan edozein soluzio aukeratzeko probabilitatea %75 edo handiagoa izango dela.

^aKontutan hartuz aipatutako baturan matrizeko elementuak ezin direla errepikatu, goi-muga birfindu daiteke matrizeko 10 elementurik handienak batuz

Algoritmoaren erabilera erakusteko 2.2 ataleko adibidea erabiliko dugu (Bavariako hiriena).

Goiko adibidean egin dugun bezala, fitness balio maximo eta minimo posibleak kalkulatu ditugu eta beraien arteko diferentzia hasierako tenperatura definitzeko erabiliko dugu. Horretarako, kostu matrizearen goiko triangeluanzenbaki minimoak eta maximoak bilatu beharko ditugu. Gainera, adibidean ez bezala, kasu honetan, hozketa prozesua gehiegi ez luzatzeko hasierako tenperatura kalkulatzeko diferentzia maximoaren probabilitatea 0.5-en finkatuko dugu.

```
> n <- ncol(cost.matrix)
> distances <- cost.matrix[upper.tri(cost.matrix)]
> ebal.max <- sum(sort(distances , decreasing = TRUE)[1:n])
> ebal.min <- sum(sort(distances , decreasing = FALSE)[1:n])
> probability <- 0.5
>
> init.t <- -1*(ebal.max - ebal.min) / log(probability)
> init.t

## [1] 14760.21
```

- **Oreka lortzeko iterazio kopurua** - Tenperatura eguneratzen den bakoitzean, balio honekin zenbait iterazio egiten diren – hau da, inguruneke soluzio batzuk aztertu behar diren – «oreka» lortu arte. Behin oreka lorturik, tenperatura berriro eguneratzen da. Lehenengo pausua, beraz, oreka lortzeko behar dugun iterazio kopurua ezartzea da. Ohikoena inguruneke tamainaren araberrako iterazio kopuru bat finkatzea da. Horretarako, p parametroa erabiliko dugu (0 eta 1 tartean hartuko ditu balioak), tenperatura bakoitzeko $\rho|N(s)|$ soluzio ebaluatuko ditugularik. Aurreko atalean azaldu dugun bezala, $|N(s)|$ -k uneko soluzioaren bizilagun kopurua adierazten du.

Beste estrategia batzuek, tenperatura bakoitzeko iterazio kopuru desberdina ezartzea proposatzen dute. Adibide gisa, tenperatura soluzio berri bat onartzen dugun bakoitzean alda dezakegu; Soluzioen onarpena haien fitness balioaren araberrako probabilitate balio baten menpekoea denez, batzuetan ebaluatzen dugun lehendabiziko soluzioa onartuko dugu eta, bestetan, hainbat soluzio probatu beharko ditugu, bat onartu arte. Kasu honetan, beraz, iterazio kopurua aldakorra da.



- **Temperatura jaitsieraren abiadura** - Hau da, ziurrenik, algoritmoaren elementurik garrantzitsuenak. Hainbat formula erabil daitezke temperatura eguneratzeko. Hona hemen batzuk:

- *Lineala*: $T_i = T_0 - i\beta$, non T_i i . iterazioko temperatura den. Eguneraketa mota honetan temperatura beti positiboa izan behar dela kontrolatu behar dugu, ekuazioak temperatura negatiboak itzuli baititzake.
- *Geometrikoa*: $T_i = \alpha T_{i-1}$. $\alpha \in (0, 1)$ abiadura kontrolatzen duen parametroa da eta, ohikoa, 0.5 eta 0.99 tarteko balio bat aukeratzea da.
- *Logaritmikoa*: $T_i = \frac{T_0}{\log(i)}$. Abiadura hau oso motela da eta, nahiz eta praktikan oso erabilgarria ez izan, interes teorikoa du suberaketa simulatu algoritmoaren konbergentzia demostratuta baitago ekuazio honekin.

Funtzio guzti hauek monotonoak dira, hau da, iterazio bakoitzean temperatura beti jaisten da. Edonola ere, problema batzuetan funtzio ez-monotonoek hobeto funtziona dezakete⁶.

- **Algoritmoa gelditzeko irizpidea** - Aurreko puntuan ikusi dugun legez, iterazioak aurrera egin ahala temperatura zero baliora hurbiltzen da baina, kasu gehienetan, ez da inoiz heltzen. Honek esan nahi du beti optimo lokaletatik ateratzeko aukera izango dugula, probabilitate oso txikiarekin bada ere. Hori dela eta, algoritmoa gelditzeko baldintzaren bat definitu beharko dugu. Irizpide hedatuena temperatura minimo bat finkatzea da; bestela, denbora edota helburu funtzioaren ebaluazio kopuru maximo bat ere finka ditzakegu.

Suberaketa simulatuaren erabilera erakusteko Bavariako TSP adibidearekin jarraituko dugu. Algoritmoa `simulated.annealing` funtzioak inplementatzen du. Funtzio honek, ohiko argumentuez gain, beste zenbait parametro berezi ditu:

- `cooling.scheme` - Temperaturaren eguneraketa funtzioa. Funtzioak uneko temperatura jasoko du argumentu gisa eta hurrengo temperatura itzuliko du.
- `initial.temperature` - Hasierako temperatura.
- `final.temperature` - Amaierako temperatura. Hau algoritmoa gelditzeko irizpidea definitzeko erabiltzen da.
- `eq.criterion` - Oreka baldintza zeren arabera den adierazten duen *string* motako parametro bat da. Bi balio posible har ditzake, `'evaluations'` eta `'acceptances'`. Lehenengoa erabiltzen bada, oreka baldintza ebaluazio kopuru jakin batera iristean beteko da. Bigarren kasuan, osteraz, uneko soluzioaren fitness-a hobetzen ez duten soluzio kopuru finko bat onartzen denean beteko da.
- `eq.value` - `eq.criterion` parametroa zehaztuko duen ebaluazio edo onarpen kopurua.

Hasierako temperatura kalkulatu dugu, baina ez amaierakoa. Aukera posible bat, ausaz soluzioak sortu eta beraien fitnessen arteko diferentzien behe-muga kalkulatzeko da.

Estrategia hau gauzatzeko 500 ausazko soluzio sortuko eta ebaluatuko ditugu. Gero, edozein bi soluzioen arteko diferentzia balio absolutuan konputatuko dugu. Azkenik, 0 ez diren diferentzien artean txikiena hartuko dugu behe-muga gisa.

```
> rep <- 500
> rnd.eval <- sapply(1:rep, FUN = function(x) tsp.babaria$evaluate(random.permutation(n)))
> pairs <- do.call(rbind, lapply(2:rep, FUN = function(i) cbind(i-1, i:rep)))
> diffs <- apply(pairs, MARGIN = 1, FUN = function(p) abs(rnd.eval[p[1]] - rnd.eval[p[2]]))
> min.delta <- min(subset(diffs, diffs != 0))
> min.delta

## [1] 1
```

⁶Temperatura igotzen denean dibertsifikazioan gailentzen da; temperatura jaistean, berriz, areagotze prozesua indartzen da. Hau kontutan hartuz, funtzio ez-monotonoak dibertsifikazio/areagotze prozesuen arteko oreka kontrolatzeko erabil daitezke



Fitness balioen diferentzia horri probabilitate txiki bat esleituz, tenperatura minimoa kalkula dezakegu.

```
> probability <- 0.1
> final.t <- -min.delta / log(probability)
> final.t

## [1] 0.4342945
```

Gauza berdina egin dezakegu tenperatura maximoa kalkulatzeko, lehen kalkulaturako goi-mugarekin alderatu ahal izateko.

```
> init.t

## [1] 14760.21

> max.delta <- max(diffs)
> probability <- 0.5
> init.t <- -max.delta / log(probability)
> init.t

## [1] 4068.4
```

Ikus daitekeen bezala, simulazio bitartez kalkulaturako diferentzietan oinarritzen bagara, hasierako tenperatura askoz ere txikiagoa da, teorikoki kalkulaturako goi-muga laginketan lortu ditugun diferentziak baino handiagoa delako.

Informazio guztiarekin funtzioaren argumentuak pausuz pausu munta ditzakegu. Ingurunea definitzeko, ondoz-ondoko trukaketak erabiliko ditugu eta bilaketa ausazko soluzio batetik abiatuko dugu.

```
> set.seed(90)
>
> args$evaluate <- tsp.babaria$evaluate
> args$initial.solution <- random.permutation(n)
> args$neighborhood <- swapNeighborhood(args$initial.solution)
```

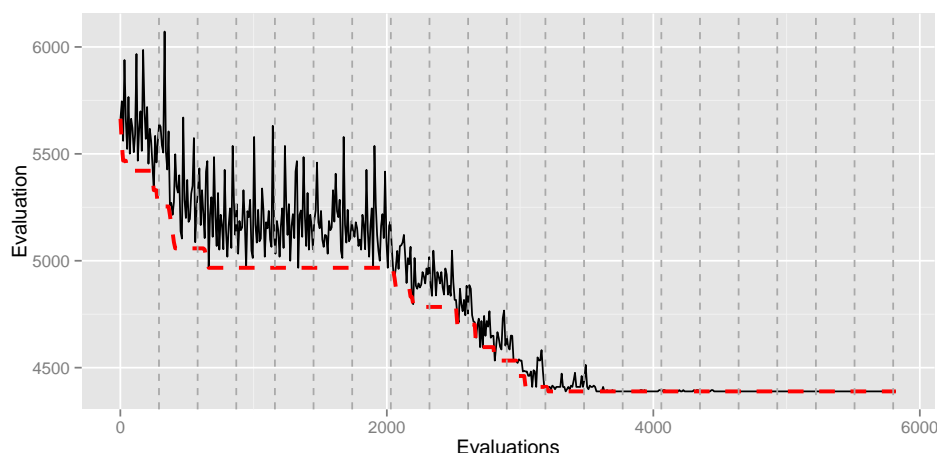
Hozketa funtzioa sortzeko `geometric.cooling` funtzioa erabiliko dugu –hozketa geometrikoa inplementatzen duena–. Funtzio honi hasierako eta amaierako tenperaturak eman behar dizkiogu. Horrez gain, hasierako tenperaturatik amaierakora zenbat eguneraketa behar diren ere adierazi beharko diogu.

```
> steps <- 20
> args$cooling.scheme <- geometric.cooling(init.t , final.t , steps)
> args$initial.temperature <- init.t
> args$final.temperature <- final.t
```

Eguneraketa funtzioari tenperatura bat emanik, hurrengo tenperatura emango digu.

```
> init.t
> next.t <- args$cooling.scheme(init.t)
> next.t
> next.t <- args$cooling.scheme(next.t)
> next.t
```

Oreka egoera, soluzio berrien ebaluazio kopuru baten arabera kontsideratuko dugu. Zehazki, 10 bider ingurunearen tamaina (hiri kopurua) izango da.



Irudia 9: Simulated annealing algoritmoaren progresioa TSP problema batean. Marra jarraikiak uneko soluzioaren progresioa adierazten du eta etenak, berriz, arkitutako soluziorik onena. Marra bertikalek tenperaturaren aldaketak erakusten dituzte.

```
> args$log.frequency      <- 10
> args$eq.criterion       <- 'evaluations'
```

Amaitzeko, algoritmoa exekutatzen dugu.

Bilaketaren eboluzioa 9 irudian dago jasota. Irudiak, uneko soluzioaren eta soluziorik onenaren progresioak erakusten ditu (beltzez eta gorrix, hurrenez hurren). Grafikoan marra bertikal bat agertzen denean, tenperatura aldaketa bat egon dela esan nahi du. Uneko soluzioaren eboluzioan ikus daiteke ebaluazioa, hasieran, oso aldakorra dela. Hau tenperaturaren eraginaren ondorioz da, tenperatura altuak soluzio txarrak aukeratzeko probabilitatea handitzen baitu. Hozketa prozesuaren zehar, tenperatura jaisten den heinean, soluzio txarrak onartzeko probabilitatea txikitu egiten da eta, ondorioz, soluzioen arteko aldakortasuna murriztu egiten da. Amaieran, tenperatura oso txikia denean, ia ezinezkoa da soluzio okerragoak onartzea eta, hortaz, uneko soluzioa ez da ia aldatzen.

Grafikan ere ikus daiteke algoritmoak oso azkar konbergitzen duela. Bilatu duen soluzioa optimo globala bada, honek esan nahi du algoritmoak oso ondo funtzionatu duela. Aldiz, tenperatura azkarregi jeitsi badugu, posible da algoritmoak optimo globalaren ingurua ez den leku batean intensifikatu izana bilaketa. Hori ez gertatzeko, eta algoritmoak gehiago dibertsifikatzeko, azken tenperatura handiagoa jarri daiteke.

Ikusitako algoritmoetan soluzioen onarpena probabilistikoa da; alabaina, suberaketa simulatuaren kontzeptua era deterministan ere implementa daiteke. Honen adibidea da Deabru Algoritmoa [9] – *Demon Algorithm*, ingelesez –. Hasiera batean Creutz-ek simulazio molekularrak egiteko proposatu zuen algoritmo hau, baina optimizazio problemak ebazteko ere egoki daiteke.

Algoritmoan soluzioak onartuko diren erabakitzeak, tenperatura erabili beharrean «deabru» bat erabiltzen da; deabru honek uneoro E_D energia kopurua dauka. Soluzio bakoitza aztertzerakoan, suberaketa simulatuan legez, ΔE kalkulatu da eta, une horretan $\Delta E > E_D$ bada, soluzioa onartzen da. Gainera, suberaketa simulatuan bezala, $\Delta E < 0$ denean ere soluzioa onartu egiten da.

Algoritmoaren gakoa deabruaren energia eguneratzean datza; soluzio bat onartzen den bakoitzean, deabruaren energia $E_D + \Delta E$ izatera pasatzen da, hau da, «sistemaren» energia aldaketa deabruak jasotzen du. Onartutako soluzioa hobea denean, deabruak energia irabazten du eta, okerragoa denean, berriz, energia galtzen du – nahiko energia baldin badu betiere –. Algoritmo honen abantaila sinpletasuna da, Boltzmann distribuzioa ebaluatzeko beharrik ez baitago.



Tabu Bilaketa

```

1 input: intensify eta diversify operadoreak
2 input: intensify_condition, diversify_condition eta stop_condition baldintzak
3 input:  $\mathcal{N}$  ingurune operadorea eta  $s_0$  hasierako soluzioa
4 output:  $s^*$  topatutako soluziorik onena
5  $s^* = s_0$ 
6  $s = s_0$ 
7 Hasieratu tabu zerrenda, epe erdiko memoria eta epe luzeko memoria
8 while !stop_condition
9     Topatu  $\mathcal{N}(s)$ -n dagoen soluzio onargarririk onena  $s'$ 
10     $s = s'$ 
11    Eguneratu tabu lista
12    if intensify_condition
13        intensify
14    fi
15    if diversify_condition
16        diversify
17    fi
18 done

```

Algoritmoa 3.4: Tabu bilaketaren sasikodea

3.2.2 Tabu bilaketa

Tabu bilaketa – *tabu search*, ingelesez – izango da, ziurrenik, bilaketa lokalaren aldaerarik hedatuena. Duen eraginkortasuna eta sinpletasuna dela eta, optimizazio konbinatorioan asko erabiltzen da eta, zenbait problematan, emaitza onenak ematen dituen metaheuristikoa da.

1977an proposatu zen lehenengo aldiz eta optimo lokaletan trabaturik ez geratzeko, bilaketa soluzio okerragoetara bideratzea baimentzen du. Estrategia hau hutsean erabiliz gero, prozesua amaigabeko ziklo batean sartzeko arriskua dago, egindako bidea behin eta berriro errepikatzeak aukera baitago. Beraz, arazo hau saihesteko, tabu bilaketak, bisitatutako soluzioen historikoa gordetzen du.

Oinarritzko tabu bilaketa, bilaketa lokal gutiziatsuan oinarritzen da, baina «tabu zerrenda» deituriko bisitatutako soluzioen multzoa gordeko da uneoro. Urrats bakoitzean tabu ez diren – bideragarriak diren, alegia – inguruneako soluzioetatik onena aukeratuko dugu, helburu funtzioa hobetzen duen ala ez kontutan hartu barik. Tabu ez diren soluzioak bakarrik hartzen ditugunez aintzat, ez da ziklorik sortuko.

Alabaina, bisitatutako soluzio guztiak gordetzen dituen zerrenda mantentzea ez da bideragarria; hori dela eta, tabu zerrendan bisitatutako azkeneko soluzioak bakarrik gordeko ditugu. Algoritmoaren iterazio bakoitzean, aukeratutako soluzioa tabu zerrendan sartuko da, eta zerrendatik soluzio bat aterako da – tabu zerrendak FIFO pilak dira, hau da, sartzen lehendabizikoa den elementua ateratzen ere lehendabizikoa izango da. Bisitatutako azken soluzioak bakarrik gordetzen direnez tabu zerrendari epe laburreko memoria ere deitzen zaio.

Bigarren estrategia mota honekin, tabu zerrendaren tamaina k bada k tamainako zikloak ekiditeko gai izango gara. Edonola ere, eraginkortasuna dela eta, soluzio osoak maneiatzeak kostu handia ekar dezake. Hori dela eta, aukera egokiagoak ere aurki ditzakegu literaturan, soluzioen atributu batzuk soilik gordetzea, adibidez. Atributuak soluzioen zatiak, ezaugarriak, edo soluzioen arteko desberdintasunak izan ohi dira. Hauek, ebazten ari garen problemaren menpekoak dira eta, hortaz, aukera ugari proposatu daitezke, kasu bakoitzerako tabu lista eredu desberdin bat inplementatuz. Ikus dezagun adibide bat.

Adibidea 3.3 Demagun permutazioetan oinarritutako problema batean tabu bilaketa bat inplementatu nahi dugula. Bilaketa lokalak 2-opt ingurunea erabiltzen badu, soluzio batetik bestera mugitzeko i eta j posizioak trukatzeko ditugu. Era honetan, tabu zerrendan trukatzeko ditugun bi posizioak gorde ditzakegu, alderantzizko trukaketa tabu bihurtuz. Esate baterako, uneko soluzioa 13245 bada eta 31245 soluziora mugitzen bagara, hurrengo urratsetan lehenengo eta bigarren posizioak trukatzea debekatua izango dugu. Problemaren arabera, beste irizpide batzuk erabil genitzake. Adibide gisa, lehenengo posizioan 1a eta bigarrenengan 3a egotea debekatu genezake.

Soluzioen atributuak erabiltzen ditugunean memoria gutxiago behar dugu eta, hortaz, tabu zerrenda handiagoak erabil ditzakegu; edonola ere, kontuan eduki behar da estrategia honekin tabu zerrenda baino txikiagoak diren zikloak ager daitezkeela. Horrez gain, diseinatutako atributuak oso zehatzak izan behar dira, bisitatu gabeko soluzio onak baztertu ez ditzagun. Ildo honetan *aspiration criteria* deritzen irizpideak erabili ohi dira bilaketa prozesuan tabu diren soluzioak onartzeko. Esate baterako, uneko soluziotik, tabu den mugimendu bat erabiliz orain arte topatutako soluziorik onena topatzen badugu, soluzio horretara bai pasatuko gara.

Tabu zerrendaren tamainak, tabu bilaketaren portaera definitzen du; txikia baldin bada, espazioko eremu txikietan zentratuko da; handia bada, berriz, algoritmoak eremu zabalagoetara bideratuko du bilaketa, soluzio asko tabu izango baitira. Ohikoa, lista tamaina aldakor bat erabiltzea da, algoritmoaren portaera kasu bakoitzeko beharretara egokitu ahal izateko.

Tabu zerrendaz gain, bestelako aukera konplexuagoak ere proposatu dira. Epe motzeko memoria erabiltzeaz gain, bilaketa prozesuan zehar jasotako informazioa ere oso baliotsua izan daiteke algoritmoa gidatzeko. Informazio hau epe erdiko edota epe luzeko memorian gorde daiteke. Lehendabiziko kasuan soluzio onenen informazioa bakarrik gordeko dugu, bilaketa areagotzeko asmoarekin. Bigarren kasuan, berriz, bilaketa osoan zehar soluzioen osagaien frekuentziak gordeko ditugu; frekuentzia hauek bisitatu ez ditugun eremuei atzemateko erabil daitezke – hau da, bilaketa dibertsifikatzeko –.

Adibidea 3.4 TSP-rako soluzioak eraikitzeke, hiri bakoitzetik zein hirira mugituko garen erabaki behar dugu. Algoritmo eraikitzaile tipikoan, erabaki hori kostu matrizea begiratzuz hartzen da, uneko hiritik bisitatu gabeko hirietatik gertuen dagoena aukeratuz. Era berean, epe erdiko eta epe luzeko memoriak matrize karratu batean inplementa ditzakegu. Matrize hauetan, bisitatutako zenbat soluzioetan i hiritik j hirira joaten garen gordeko dugu. Epe erdiko memorian azken k soluzio onenen informazioa bakarrik gordeko dugu, areagotze prozesuan gehien erabili direnak finkatzeko eta bilaketa falta diren loturretan zentratzeko. Epe luzeko memorian, berriz, bisitatu ditugun soluzio guztien informazioa gordeko dugu. Era honetan, bilaketa esploratu gabeko eremuetara eramane nahi badugu, gutxienez erabilitako loturak erabiliz soluzioak sor ditzakegu, bilaketa prozesua bertatik abiatzeko.

3.3 Optimizazio problemen «itxura»aldaketa

Bilaketa lokalean uneko soluziotik honen inguruan dagoen soluzio batera mugitzen gara beti, hau da, soluzio bakoitzetik soluzio kopuru mugatu batetara mugitu gaitzeko soilik. Hori dela eta, bilaketa espazioa grafo baten bidez adieraz daiteke, non erpinak soluzioak diren eta ertzek mugimendu posibleak adierazten dituzten; 2 irudiak horrelako grafo bat adierazten du.

Bilaketa espazioaren definizioari soluzioen ebaluazioa gehitzen badiogu, optimizazio problemaren «itxura» – *landscape*-a, ingelesez – daukagu. Problemaren itxuraren eragina berebizikoa da algoritmoen performantzia eta, beraz, algoritmoak diseinatzerakoan kontuan hartu beharreko elementua da. Zentzu horretan, kontuan hartu behar da problema motaren arabera ez ezik, *landscape*-a instantzia konketuaren arabera aldatzen dela. Optimizazio problemen "itxuraren" ideia intuikorra, gailurrez, mendikatez eta bailarez osatutako paisaia baten ilustrazioa da, non gailurrek optimo lokalak erreprezentatzen dituzten eta gailurrik altuena optimo globala den. Gailur baten erakarpene-arroa, mendi osoa da, oinarritik gailurrera. Intuitiboki, zenbat eta gailur gehiago, orduan eta zailagoa izango da instantzia baten optimo globalera heltzea bilaketa lokalean oinarritutako algoritmo batekin.



VND algoritmoaren sasikodea

```

1 input:  $\mathcal{N} = \{\mathcal{N}_1, \dots, \mathcal{N}_k\}$  ingurune funtzioak
2 input:  $s$  hasierako soluzioa
3  $i = 1$ 
4  $s^* = s$ 
5 while  $i \leq k$  do
6   Bilatu  $s'$ ,  $\mathcal{N}_i(s^*)$  inguruneke soluziorik onena
7   if  $f(s') < f(s^*)$ 
8      $s^* = s'$ 
9      $i = 1$ 
10  else
11     $i = i + 1$ 
12  fi
13 done

```

Algoritmoa 3.5: VND algoritmoaren sasikodea

Soluzio bakarrean oinarritzen diren algoritmoekin amaitzeko, bilaketa prozesuan zehar, *landscape*-a eraldatzen dituzten algoritmoak aztertuko ditugu. Zehazki, bi algoritmo ikusiko ditugu. Lehenengoak, VNS-ak, ingurune definizio ezberdinak erabiltzen ditu optimo lokaletatik ateratzeko. Bigarrenak, berriz, helburu funtzio berriak sortzen ditu optimo lokalen kopurua murrizteko.

3.3.1 Variable Neighborhood Search algoritmoa

Bilaketa lokalean optimo lokal batean trabaturik gelditzen gara, definizioz bere ingurunean helburu funtzioa hobetzen duen soluziorik ez dagoelako. Baina, zer gertatuko litzateke ingurunearen definizioa aldatuko bagenu?. Adibide gisa, demagun optimizazio problema batean soluzioak permutazioen bidez kodetzen ditugula. Bilaketa lokala aplikatzeko *2-opt* operadorea erabiliko dugu – hau da, *swap* eragiketan oinarritutako ingurunea –. Izan bedi 1432 soluzioa, ingurune eta problema honetarako optimo lokala dena. Definizioz, soluzio honen edozein bi posizio trukatzuz lortutako soluzioak okerragoak izango dira. Alabaina, txertaketan oinarritzen den ingurunea erabiliz *2-opt* ingurunean ez dauden soluzioak lor ditzakegu – lehenengo elementua azken elementuaren ostean txertatuz lortzen den 4321 soluzioa, esate baterako –. Beraz, gerta daiteke *2-opt* ingururako optimo lokala den gure soluzioa txertaketak definitzen duen ingurunerako optimoa ez izatea.

Idea hau *Variable Neighborhood Descent* (VND) algoritmoan erabiltzen da bilaketa lokala optimo lokaletan trabaturik geratzea ekiditeko. 3.5 algoritmoan VND-aren sasikodea ikus daiteke.

Algoritmoan ikusten den bezala, VND-an ingurune funtzio bakarra izan beharrean hauen multzo bat dago. Lehenengo ingurunea erabiliz, uneko soluzioaren ingurunea arakatu eta soluzio onena aukeratuko dugu. Inguruneke soluzio guztiak okerragoak direnean – topatutako soluzioa uneko ingurunerako optimo lokala bada, alegia – hurrengo ingurune definiziora pasatuko gara; honela, ingurune funtzio guztiak erabili arte. Gainera, iterazio bakoitzean, inguruneke soluzio berri batera pasatzen garenean, berriro ere hasierako ingurune definiziora itzuliko gara.

Bilaketa amaitzeko baldintza kontutan hartuz, algoritmo honek itzultzen duen soluzioa *ingurune definizio guztietarako optimo lokala* izango da.

Variable Neighborhood Search algoritmoa VND-aren hedapen bat da, non iterazio bakoitzean, uneko ingurune definizioa erabiliz, bilaketa lokala amaiera arte eramaten den. Hau da, fitness-a hobetzen duen soluzio bat topatu arren, uneko ingurune definizioa mantentzen dugu optimo lokal batera heldu arte. Behin optimo lokal batera heldutakoan, hurrengo ingurune definizioa erabiltzera pasatuko gara, VND-an legez, lortutako soluzioa ingurune guztietarako optimo lokala izan arte. 3.6 algoritmoak VNS-aren sasikodea erakusten du.

Oinarrizko VNS algoritmoaren sasikodea

```

1 input: local_search bilaketa algoritmoa
2 input:  $\mathcal{N} = \{\mathcal{N}_1, \dots, \mathcal{N}_k\}$  ingurune funtzioak
3 input:  $s$  hasierako soluzioa
4  $i = 1$ 
5  $s^* = s$ 
6 while  $i \leq k$  do
7   Aukeratu ausaz soluzio bat  $s' \in \mathcal{N}_i(s^*)$ 
8    $s'' = \text{local\_search}(s^*, \mathcal{N}_i)$ 
9   if  $f(s'') < f(s^*)$ 
10      $s^* = s''$ 
11      $i = 1$ 
12   else
13      $i = i + 1$ 
14   fi
15 done

```

Algoritmoa 3.6: VNS algoritmoaren sasikodea

Oinarrizko *Smoothing* algoritmoaren sasikodea

```

1 input: smoothing ( $f, \alpha$ ) helburu funtzioa eraldatzeko funtzioa
2 input: local_search( $s, f$ ) bilaketa lokala
3 input: update( $\alpha$ ) faktorea eguneratzeko funtzioa
4 input:  $s$  hasierako soluzioa;  $\alpha_0$  hasierako faktorea;  $f$  helburu funtzioa
5  $s^* = s$ ;  $\alpha = \alpha_0$ 
6 while  $\alpha > 1$  do
7    $f' = \text{smoothing}(f; \alpha)$ 
8    $s^* = \text{local\_search}(s^*, f')$ 
9    $\alpha = \text{update}(\alpha)$ 
10 done

```

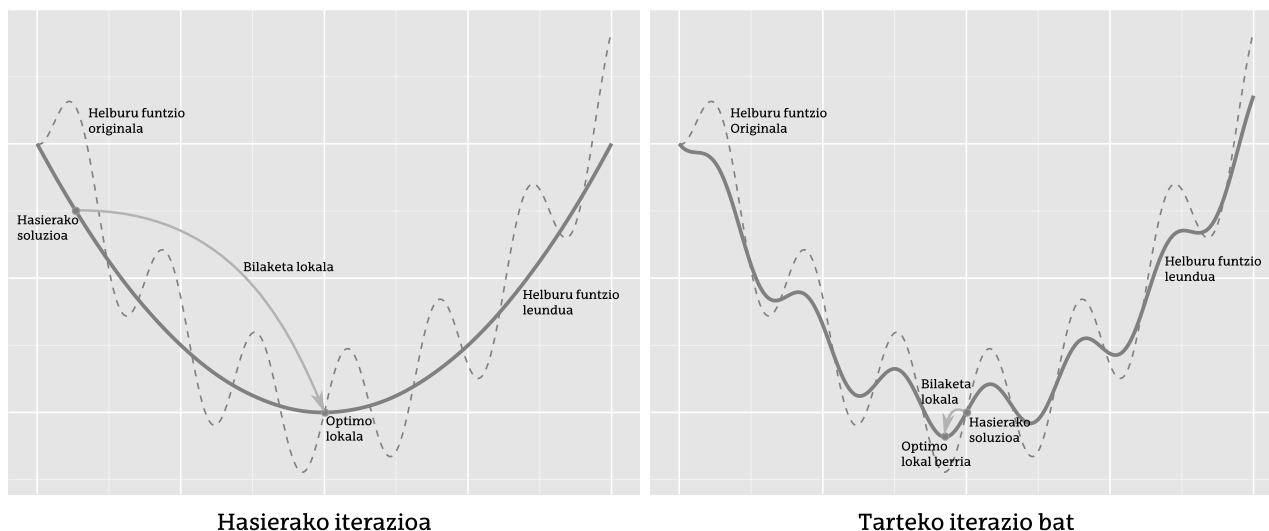
Algoritmoa 3.7: *Smoothing* algoritmoaren sasikodea

3.4 *Smoothing* algoritmoak

Optimizatu behar dugun funtzioak optimo lokal asko dituenean, bilaketa lokalak ez dira oso metodo egokiak, globala ez den optimo batean trabaturik gelditzeko probabilitatea oso altua delako. Leuntze-metodoekin – *smoothing methods*, ingelesez – iterazio bakoitzean jatorrizko helburu funtzioa eraldatu – leundu – egiten da, optimo lokal kopurua gutxitzeko asmoz; eta helburu funtzio berria erabiliz, bilaketa lokala aplikatzen da. Bilaketa trabaturik geratzen denean – optimo lokal batean –, helburu funtzioa berriro aldatzen da, aurreko iterazioan baino gutxiago leunduz. Helburu funtzio berri honekin eta aurreko iterazioan lortutako optimoarekin, bilaketa lokala aplikatzen da, optimo berri bat lortuz.

Iterazioz iterazio leuntze-maila geroz eta txikiagoa bihurtuz, azken iterazioan problemaren jatorrizko helburu funtzioa erabiliko dugu, problemarako soluzioa topatzeko.

Helburu funtzioa nola leundu problemaren arabera erabakia da. Edonola ere, kasu guztietan, algoritmoa inplementatu ahal izateko leuntze-parametro bat definitu beharko dugu. Parametro hau handia denean, helburu funtzioa asko leunduko dugu; parametroa 1 denean, berriz, helburu funtzioa ez da bat ere aldatuko. Hau aintzat hartuz, 3.7 algoritmoan metodoaren sasikodea definituta dago. Ikus dezagun adibide bat.



Hasierako iterazioa

Tarteko iterazio bat

Irudia 10: *Smoothing* algoritmoaren funtzionamendua. Iterazio bakoitzean hasierako helburu funtzioa maila bateraino leuntzen da eta bilaketa lokala aplikatzen da.

Adibidea 3.5 *TSP-an helburu funtzioa kalkulatzeko distantzien matrizea erabiltzen dugu. Matrize horretan edozein bi hirien arteko distantzia dago jasota. Helburu funtzioa leuntzeko, matrizea hau eralda daiteke, distantzia guztiak batez-besteiko distantziara hurbilduz, adibidez. Demagun ondoko matrizea definitzen dugula:*

$$d_{ij}(\alpha) = \begin{cases} \bar{d} + (d_{ij} - \bar{d})^\alpha & \text{baldin eta } d_{ij} \geq \bar{d} \\ \bar{d} - (\bar{d} - d_{ij})^\alpha & \text{baldin eta } d_{ij} < \bar{d} \end{cases} \quad (6)$$

non \bar{d} distantzien batez-bestekoa eta d_{ij} jatorrizko matrizearen elementuak diren. Distantzia matrizea normalizatuta badago – distantzia guztiak 1 edo txikiagoak badira^a – α parametroa oso handia denean distantzia guztiak batez-besteak hurbilduko zaizkio, $0 \leq (d_{ij} - \bar{d}), (\bar{d} - d_{ij}) < 1$ baita. Muturreko kasu horretan, soluzioa tribiala da, soluzio guztiak berdinak baitira.

Iterazioz iterazio α parametroa gutxituko dugu, 1 baliora heldu arte. Goiko ekuazioan ikus daitekeen bezala, $\alpha = 1$ denean distantzia matrizea jatorrizkoa da.

^aKontutan hartu behar da, matrizea normalizatuta ere, soluzio optimoa, hau da, balio minimoa duena, ez dela aldatzen.

Bibliografia

- [1] E. H. L. Aarts and P. J. M. Laarhoven, editors. *Simulated Annealing: Theory and Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1987.
- [2] R. K. Congram. Polynomially searchable exponential neighborhoods for sequencing problems in combinatorial optimization.
- [3] G. Dueck and T. Scheuer. Threshold accepting-a general-purpose optimization algorithm appearing superior to simulated annealing. *Journal of Computational Physics*, 90:161–175, 1990.



- [4] Thomas A Feo and Mauricio G.C Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8(2):67 – 71, 1989.
- [5] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533–549, 1986.
- [6] M.D. Huang, F.Romeo, and A.L. Sangiovanni-Vincentelli. An efficient general cooling schedule for simulated annealing. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 381–384, Santa Clara C.A., 1986.
- [7] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.
- [8] P. Moscato and J.F. Fontanari. Convergence and finite-time behavior of simulated annealing. *Advances in Applied Probability*, 18:747–771, 1990.
- [9] J. W. Pepper, B.L. Golden, and E.A. Wasil. Solving the traveling salesman problem with demon algorithms and variants. Technical report, Smith School of Business, University of Maryland, College Park, Maryland, 2000.
- [10] Erwin Pesch and Fred Glover. TSP ejection chains. *Discrete Applied Mathematics*, 76(1&2&3):165 – 181, 1997.
- [11] Vlado Černý. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45(1):41–51, January 1985.