```
##
##
## processing file:  02_Soluzio_bakarrak.Rnw
## Error in parse_block(g[-1], g[1], params.src):  duplicate label 'GC_1 '
```

BHLN: Soluzio bakarrean oinarritutako algoritmoak

Borja Calvo, Josu Ceberio, Usue Mori

Laburpena

Kapitulu honetan soluzio bakarrean oinarritzen diren algoritmoak aztertuko ditugu. Algoritmo eraikitzaileak mota honetakoak izan arren, teknikoki ez dira bilaketa heuristikoak – ez baitago bilaketa prozesurik – eta, hortaz, ez ditugu kapitulu honetan azalduko. Horren ordez, bilaketa lokalari erreparatuz, lehenengo atalean bere inguruan agertzen diren kontzeptuak azalduko ditugu. Bilaketa lokalaren arazorik handiena optimo lokaletan trabatuta geratzea denez, kapituluaren bigarren zatian arazo hau saihesteko zenbait estrategia aztertuko ditugu.

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, bilaketaren pausu bakoitzean, uneko soluzio horren ingurunean dagoen beste soluzio batekin ordezkatuko dugu.

Ideia honetan hainbat algoritmo oinarritzen dira, bakoitzak bere berezitasunekin. Diferentziak diferentzia, zenbait elementu komunak dira algoritmo guztietan; atal honetan kontzeptu hauek aztertzeari ekingo diogu.

1.1 Soluzioen inguruneak

Bilaketa lokalean dagoen kontzepturik nagusiena inguruarena da – *neighborhood*, ingelesez – eta, hortaz, problema bat ebazterakoan arreta handiz diseinatu beharreko osagaia da. Ingurune funtzioak edo operadoreak[1], soluzio bakoitzeko, bilaketa espazioaren azpimultzo bat definitzen du.

Definizioa 1.1 *Izan bedi $\mathcal{S}$ bilaketa espazioa; $N : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ funtzioari, $s \in \mathcal{S}$ soluzioa emanda $N(s) \subset \mathcal{S}$ bilaketa espazioaren azpimultzo itzultzen duenari, ingurune funtzioa deritzo*

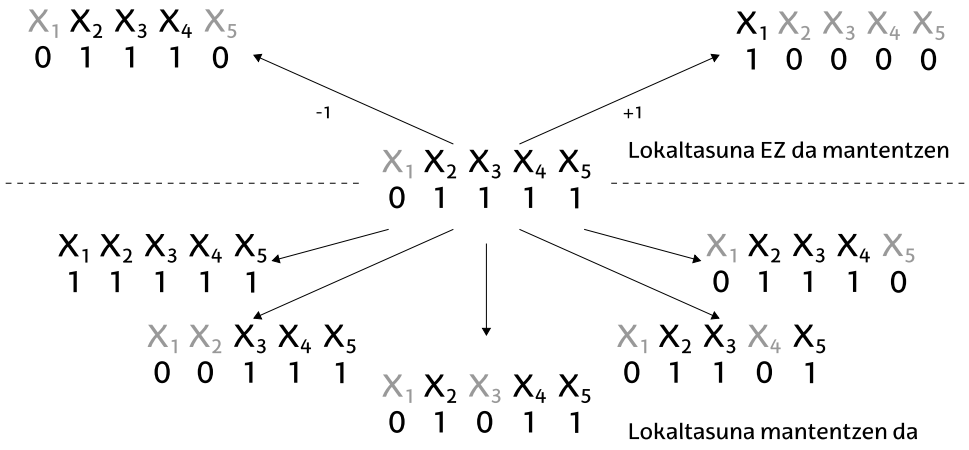
Soluzio baten ingurunean dauden soluzioak antzerakoak izatea espero dugu; alabaina, ingurune operadoreek soluzioen kodeketarekin dihardute eta, hortaz, antzekotasuna ez dago beti bermatuta.

Beraz, bilaketa lokal bat diseinatzean ondo aukeratu behar da kodeketa/ingurune bikotea, ingurunean dauden soluzioak antzerakoak izan daitezen. Ezaugarri honi lokaltasuna –*locality* ingelesez– esaten zaio, eta emaitza onak lortzeko funtsezkoa da.

[1] Programazio testuinguruetan – sasikodetan, adibidez – «soluzioak maneiatzeko erabiltzen diren funtzioei operadore» deritze eta, hortaz, «funztio» eta «terminoa» terminoak erabiliko ditugu baliokideak gisa.

Ikasturtea: 2013/2014
Irakaslea: Borja Calvo, Usue Mori
Telefonoa: 943 01 50 13
E-posta: borja.calvo@ehu.es

GIPUZKOAKO CAMPUSA
CAMPUS DE GUIPUZCOA
Paseo Manuel de Lardizabal, 1
20018 Donostia (Gipuzkoa)



Irudia 1: Bi ingurune adibide. Goikoan bektore bitarri 1 gehitzen/kentzen diogu inguruneko soluzioak sortzeko. Eskuman dagoen soluzioan ikus daitekeen bezala, lokaltasuna ez da mantentzen. Beheko ingurunea *flip* eragiketan onairritzen da. Kasu honetan sortutako soluzio guztiak antzerakoak dira.

Adibidea 1.1 Lokaltasuna FSS probleman *Datu meatzaritzan, gainbegiratutako datu base bat daukagunean, sailkatzaile funtzioak eraikitzea edo ikastea ataza ohikoa da oso. Funtzio hauek, beraien izenak adierazten duen moduan, datu berriak sailkatzeko erabiltzen dira.*
Oro har, datuetan dauden aldagaiek iragarpenak egiteko gaitasun ezberdinak dituzte; are gehiago, aldagai batzuk eragin negatiboa izan dezakete sailkatzailearen funtzionamenduan. Hori dela eta, aldagaien azpimultzo bat aukeratzea oso pausu ohikoa da; prozesu hau, ingelesez feature subset selection, FSS *deitzen dena, optimizazio problema bat da. FSS problemarako soluzioak bektore bitarren bidez kode daitezke, bit bakoitzak dagokion aldagaia azpi-multzoan dagoenetz adierazten duelarik.*
Bektore bitarrek zenbaki osoak moduan interpreta daitezke eta, hortaz, antzerako soluzioak zenbaki horiei balio txikiak gehituz/kenduz sor ditzakegu. Adibidez, (01001) *soluzioa badugu* 9 *zenbakia moduan interpreta dezakegu eta, hortaz, antzerako soluzioak lor ditzakegu 1 gehituz –* (01010)*, hau da,* 10 *zenbakia – edo kenduz –* (01000)*,* 8 *zenbakia –. 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 sartu. Beste era batean esanda, FSS problemarako ingurune definizio hau ez da oso egokia.*
Azpi-multzo problematan ingurune operadore ohikoena flip *aldaketan oinarritzen dena da; inguruneko soluzioak sortzeko uneko soluzioaren posizio batean dagoen balioa aldatzen da, hau da, 0 bada 1ekin ordezkatzen da eta 1 bada, 0rekin. Operadore honekin FSS probleman beti bermatzen da lokaltasuna, ingurunean dauden edozein bi soluziok aldagai bakar bateko diferentzia izango baitute. 1.1 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 kopurua baino txikiagoa bada. Bi soluzioen arteko distantzia $d(s, s')$ funtzioaz adierazten badugu, ingurunearen definizio orokorra hauxe izango da:

$$N(s; k) = \{s' \mid d(s, s') \leq k\} \tag{1}$$

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

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

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

Ingurune tamainari erreparatuz, zenbaki errealak direnez infinitu soluzio izango ditugu edozein soluzioren inguruenean. Euklidearra distantziarik ezagunena izan arren, badira beste metrika batzuk ere – Manhanttan-edo Chevyshev-distantziak, besteak beste –.

Bektore kategorikoak eta bitarrak - Bektoreetan dauden aldagaiak kategorikoak direnean, bi bektoreen arteko distantzia neurtzeko metrikarik ezagunena Hamming-ek proposatutakoa da: $d_h(s, s') = \sum_{i=1}^{n} I(s_i \neq s'_i)$ da, non I funtzioak 1 balioa hartzen du bere argumentua egia denean eta 0 beste kasuan; hau da, Hamming-distantziak posizioz posizioko desberdintasunak neurtzen ditu. Hamming-distantzia inguruneak definitzeko erabiltzen denean ohikoena 1 distantziara dauden soluzioak erabiltzea da, hau da:

$$N_{h1}(s) = \{s' \in \mathcal{S} \mid d_h(s, s') = 1\} \tag{2}$$

Algoritmoak diseinatzean oso garrantzitsua da ingurunearen tamaina aztertzea. Aurreko operadorea n tamainako bektore bitar bati aplikatzen badiogu, $|N(s)| = n$ 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.

Ikus dezagun adibide bat, `metaheuR` paketea erabiliz. Motxilaren problema erabiliko dugu eta, horretarako, lehenengo ausazko problema bat eta soluzio bat sortuko ditugu. Pisua eta balioa korrelaturik egoteko, balioari ausazko kopuru bat gehituko diogu. Gero, motxilaren kapazitatea definitzeko ausazko $\frac{n}{2}$ elementuen pisua batuko ditugu. Ausazko soluzioa sortzen dugu; elementu bakoitza aukeratzeko probabilitatea heren bat izango da.

```
> 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
```

Sortutako soluzioa bideraezina izan daitekeenez, lehenengo pausua soluzioa zuzenduko dugu. Gero, «flip» operadorea erabiliko dugu inguruneko soluzioak sortzeko. Operadore honek Hamming distantzia 1era dauden soluzioak sortuko ditu.

```
> 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("Inguruneko soluzio ",isvalid,": " ,
+                 paste(which(ngh),collapse = ",") , sep="")
+   cat (mssg , "\n")
+ }
```

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

Goiko kodean ikus daitekeen bezala, badaude paketean bi funtzio inguruneak korritzeko: `has.more.neighbors` eta `next.neighbor`. Izenek adierazten duten bezala, lehenengo funtzioak ingurunean bisitaturik gabeko soluzioren bat badagoen 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 `?reset.neighborhood` tekleatuz R-ko terminalean.

Kontutan hartu behar da motxilaren probleman soluzio guztiak –azpi-multzo guztiak, alegia– ez direla bideragarriak; nahiz eta uneko soluzioa bideragarria izan, ingurunekoak bideraezinak izan daitezkee. Adibide bat ikusteko, proba ezazu ausazko soluzioa sortzean item bat aukeratzeko probabilitate handitzen.

Ingurune funtzioa orokor daiteke edozein distantziarako – distantzia maximoa bektorearen tamaina dela kontutan hartuz, betiere –.:

$$N_{hk}(s;k) = \{s' \in \mathcal{S} \mid d_h(s,s') \leq k\} \tag{3}$$

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 trukatzen dira. Adibidez, [21345] permutazioa badugu, 1. eta 3. posizioak trukatzen baditugu [31245] permutazioa lortuko dugu. Formalki, trukaketa funtzioa defini dezakegu $t_r(s;i,j)$ 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 onetan oinarriturik, ondoko operadorea defini dezakegu:

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

Operadore honi *2-opt neighborhood* deritzo, bi posizio bakarrik trukatzen baitira. Era berean, operadorea hedatu ahal da trukaketa gehiago eginez. Hedatzeaz gain, operadorea murriztu ere egin ahal da, bakarrik elkarren ondoan dauden posizioak trukatuz. 2-opt operadorea `ExchangeNeighborhood` klaseak inplementatzen du, eta elkar ondoko trukaketari murriztutako bertsioa `SwapNeighborhood` klasearen bidez erabil daiteke.

Ingurunearen tamainari dagokionez, posizio jarraien trukaketak eginez $n-1$ ingurune soluzio izango ditugu; edozein bi posizio trukatzen 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
```

Txertaketa egitean 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 $t_x(i,j)$ funtzioaren bidez adieraziko dugu – i elementua j posizioan txertatu –, ingurunearen definizioa hauxe izango da:

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

Trukaketan bakarrik bi posizio aldatzen dira; txertaketan, berriz, bi indizeen artean dauden posizio guztiak aldatzen dira. Hori dela eta, ingurune operadore bakoitzaren erabilgarritasuna problemaren araberakoa izango da. Operadore hau ere `metaheuR` paketean dago, `InsertNeighobrhood` klasean inplementaturik.

Bi inguru operadore hauetaz gain, literaturan beste zenbait topa daitezke, inbertsio eragiketan oinarritutakoak, adibidez.

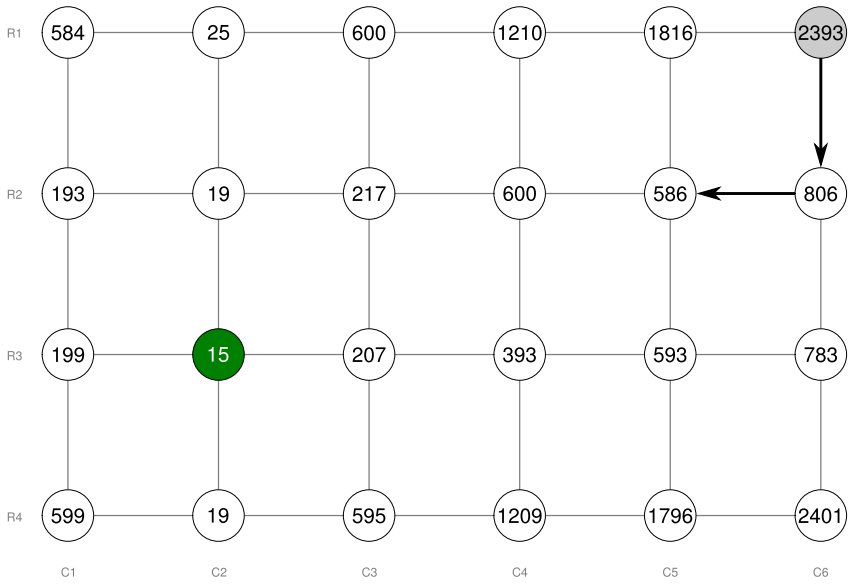
1.2 Optimo lokalak

Bilaketa lokalean – oinarrizko bertsioan, behintzat – soluzio batetik bestera mugitzeko helburu funtzioa hobetu behar da eta, beraz, algoritmoa bukatzean uneko soluzioak (s^*) ondoko baldintza beteko du:

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

Ekuazio hau optimo globalarenaren oso antzerakoa da –aurreko kapituluan, 2.1 definizioa–; alabaina, optimo globalaren kasuan, bilaketa espazio osoan bete behar da eta hemen, berriz, bakarrik s^* soluzioaren ingurunean daudenentzat. Soluzio guztietarako betetzen bada, s^* *optimo globala* dela esaten da; ildo berean, inguruko soluzioentzat bakarrik betetzen baldin bada, s^* *optimo lokala* dela esango dugu.

Ondorioz, bilaketa lokala optimo lokal batean amaitzen da beti. Hauxe da, hain zuzen, bilaketa lokalaren ezaugarririk – eta, aldi berean, arazorik – nagusiena. Aintzat hartzekoa da optimo lokalak, nahiz eta bere inguruneko soluziorik onenak izan, nahiko soluzio txarrak izan daitezkeela, 2 irudian erakusten den bezala. Irudi honetan kapituluan zehar erabiliko dugun grafiko mota bat ikus daiteke. Grafikoan fikziozko problema baterako soluzio guztiak jasotzen dira, bakoitza borobil baten bidez adierazita; borobilen barruan soluzio bakoitzaren helburu funtzioaren balioa idatzita dago. Ingurunearen funtzioa soluzioak lotzen dituzten marren bidez adieraziko da; bi soluzio lotuta badaude, bata bestearen ingurunean dago.



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

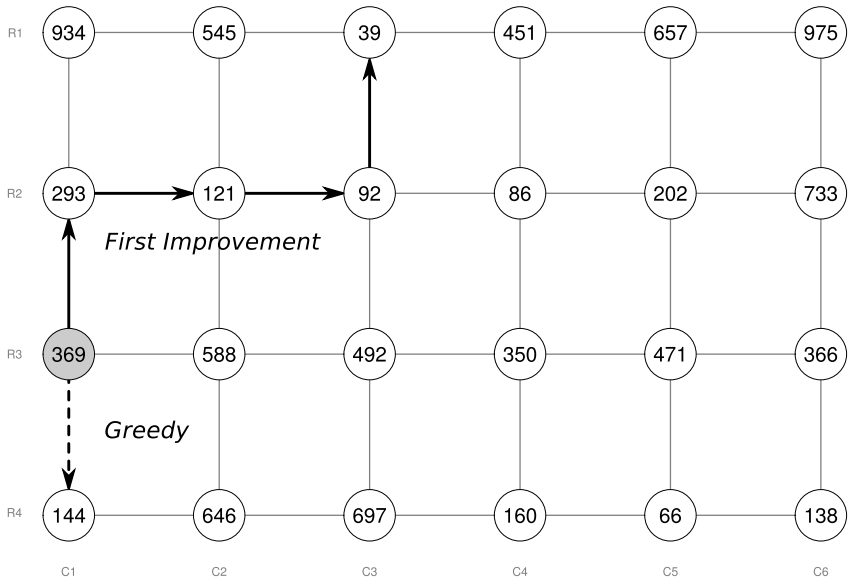
Adibidea 1.2 *2 irudian agertzen diren geziek algoritmoak egiten duen bidea erakusten dute, (R1,C6) soluziotik abiatuta. Pausu bakoitzean inguruneko soluziorik onena aukeratzen badugu, algoritmoa bi pausutan trabatuta geldituko da (R2,C5) soluzioan; soluzio honen ebaluazioa 586 da eta, bere ingurunean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

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

Ikusi dugunez, edozein soluziotik abiatuta, bilaketa lokala beti optimo lokal batean amaitzen da. Are gehiago, bi soluzio ezberdinetatik hasita, bilaketa lokala soluzio berdinean amaitu ahal da. Beste era batean esanda, 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, $A(s; f, N) : \mathcal{S} \to \mathcal{S}$ bilaketa lokala 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.*

Erakarpen-arroa, definizioan ikus daitekeen legez, helburu funtzioaren, ingurunearen eta algoritmoaren araberakoa da. Ingurunearen eta helburu funtzioaren eragina begi bistakoa da. Algoritmoari dagokionez, ingurunea nola aztertzen den eta, bereziki, zein soluzio aukeratzen den ezartzen du; ondorioz, egiten dugun ibilbidean eragina handia izan dezake, 3 irudiak ikus daitekeen bezala.



Irudia 3: Inguruneko 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 – inguruneko 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 gutiziatsua da – *greedy*-a, alegia –; hurrengo soluzioa ingurunean dagoen soluziorik onena izango da beti. Estrategia hau erabiliz pausu bakar batean (R4,C1) optimo lokalera ailegatzen gara

2 Bilaketa lokala

2 algoritmoan oinarrizko bilaketa lokalaren sasikodea ikus daiteke. Zenbait gauza nabarmen daitezke algoritmo honetan. Lehenik eta behin, bilaketa soluzio batetik hasten da. Soluzio hau nola aukeratzen dugun erabakitzea garrantzitsua da, aurreko atalean ikusi dugun legez horren arabera optimo lokal batean edo bestean amituko baita bilaketa. Uneko soluzioaren ingurunean hainbat soluzio izango ditugu, beraz, zein aukeratuko dugu hurrengo soluzioa izateko? Azkenik, sasikodean dagoen prozedura optimo lokal bat topatzen dugunean amaitzen da; dena dela, beste edozein algoritmotan bezala, denboran edota ebaluazio kopuruan oinarritutako gelditze irizpideak proposa ditzakegu.[2]

Sasikodean dagoen algorithmoa `metaheuR` paketeko `basic.local.search` funtzioak inplementatzen du. Funtzio honek zenbait parametro ditu, batzuk algoritmoarekin zerikusia dutenak eta beste batzuk problemari eta exekuzioari lotuta daudena. Paketean dauden metaheuristika guztietan zerbait antzerakoa gertatuko den legez, pausuz pausu aztertuko ditugu parametro hauek. Esan dugun bezaka, parametroak hiru motakoak dira:

- Problemako parametroak - Bilaketa gidatzeko helburu funtzioa behar dugu. Funtzio hau `evaluate` parametroaren bidez pasatuko diogu algoritmoari. Algoritmoen inplementazioa orokorra denez, gerta daiteke problema baterako bideraezin diren soluzioak egotea. Problema horiekin lan egin ahal izateko soluzioak bideragarriak direnetz jakiteko eta soluzioak konpontzeko funtzioak erabil ditzakete. 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 konputazional murriztuak eman ahal dizkiogu, denbora, ebaluazio kopurua edota iterazio kopurua mugatuz. Hau `CResource` objektuen bidez egin dezakegu, `cresource` parametroaren

[2] Informazio gehiago R-ren laguntzan duzu; `?basic.local.search` tekleatu laguntza zabaltzeko

Oinarrizko bilaketa lokala

input: f helburu funtzioa, s_0 hasierako soluzioa, N ingurune funtzioa
output: s^* soluzio optimoa
$s^* = s_0$
do
  $H = \{s' \in N(s^*) | f(s') < f(s^*)\}$
  if $|H| > 0$
    Aukeratu H-n dagoen soluzio bat s'
    $s^* = s'$
  fi
while $(|H| > 0)$

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

bidez pasatuz algoritmoak eskuragarri dituen baliabideak. Jarraian dagoen adibidean klase hau nola erabili ikusiko dugu. Horrez gain, algoritmoak bilaketaren progresioa bistara dezake. Bistaraketa aktibatzeko `verbose` parametroa dugu. Era berean, progresioa taula batean gorde nahi izanez gero, `do.log` parametroaren bidez adieraz diezaiokegu funtzioari.

- 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 daudeneanhiru aukera ditugu, bideraezin diren soluzioak onartu, deskartatu edo konpondu. Zein aukera erabili `non.valid` parametroaren bidez adiraziko dugu.

Ikus dezagun funtzioaren erabilera adibide baten bidez. Adibiderako grafoen koloreztetze-problema erabiliko dugu, ausazko grafo bat sortuz.

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

Orain, hasierako soluzio gisa soluzio tribiala sortuko dugu, non nodo bakoitzak kolore bat duen. Horrez gain, Hamming distantzian oinarritutako ingurune objektua sortuko dugu.

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

Algoritmoari emango dizkiogun baliabideak mugatuko ditugu. Gehienez, algoritmoak 10 segundu edo $100n^2$ ebaluazio edo $100n$ iterazio erabili ahalko ditu.

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

Dena prest daukagu bilaketa abiaratzeko ...

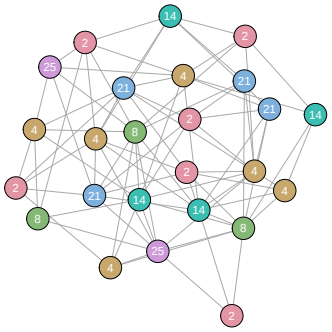
```
> bls <- basic.local.search(evaluate = gcp$evaluate , valid = gcp$is.valid ,
+                          correct = gcp$correct, initial.solution = initial.solution ,
+                          neighborhood = h.ngh , selector = first.improvement.selector ,
+                          non.valid = 'correct' , 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: 3.861327
## 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)
```

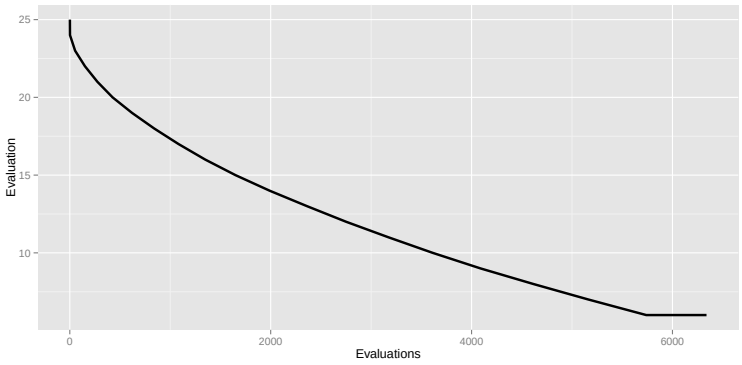
Bilaketa lokalak –eta, oro har, beste gainontzeko algoritmoek– objektu berezi bat itzultzen du, `MHResult` klasekoa. Bertan dagoen informazioa zenbait funtzio erbiliz lor daiteke; este baterako, `optima` funtzioak lortutako soluzio optimo(ak) itzultzen du, zerrenda batean. Nola bilaketa loklak soluzio bakarra itzultzen du, soluzio hori listaren 1. posizioan egongo da. Soluzioa grafikoki bistaratzeko `graph.coloring.problem` funtzioak itzultzen duen `plot` funtzioa erabil daiteke.

Bilaketare progresioa ere bistara dezakegu, `plot.progress` funtzioa erabiliz. 4 irudiak soluzioa eta progresioa jasotzen ditu.

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



(a) Problemaren soluzioa



(b) Bilaketaren progresioa

Irudia 4: Soluzioa grafoen koloreztatze-problemarako eta bilaketaren progresioa. X ardatzak ebaluazio kopurua adierazten du eta Y ardatzak uneko soluzioaren ebaluazioa –soluzioak erabiltzen dituen kolore kopurua, alegia–.

Jarraian algoritmoaren bi aspektu garrantzitsu aztertuko ditugu: hasierako soluzioa eta inguruneko soluzioaren aukeraketa.

2.1 Hasierako soluzioa

Lehen aipatu bezala, bilaketa lokala soluzio batetik abiatuko da beti. Nondik hasi bilaketa oso garrantzitsua da, horren arabera optimo lokal batean edo bestean amaituko baitu bilaketa. Hau 2 irudian agerikoa da; (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.

Bi dira bilaketa lokala hasieratzeko erabiltzen diren estrategia nagusiak:

- **Ausazko soluzioak sortu -** Ausaz aukeratzen da bilaketa espazioan dagoen soluzio bat. Metodo honen abantaila bere sinpletasuna da, ausazko soluzioak sortzea erraza izan ohi baita[3]. 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. Azken puntu hau optimo lokalen arazoa ekiditeko estrategiak diseinatzeko erabil daiteke, 3.1 atalean ikusiko dugun legez.
- **Soluzio onak eraiki -** Lehenengo kapituluan ikusi genuen problemak ebazteko metodo heuristiko espezifikoak diseina daitezkeela. Oro har, metodo hauek pausuz pausu soluzio onak eraikitzen dituzte, urrats bakoitzean aukera guztietatik onena aukeratuz – ingelesez metodo hauei *constructive greedy* deritze, hau da algoritmo eraikitzaile gutiziatsuak edo jaleak –. Nahiko soluzio onak eman arren, hauek ez daukate zergatik optimoak[4] izan behar eta, beraz, lortutako soluzioak bilaketa lokala hasieratzeko erabil daitezke. Estrategia hauek emaitza hobeak lortu ohi dituzte eta bilaketak iterazio gutxiago behar izaten dituzte; desabantaila nagusia, ordea, kostu konputazionala da.

2.2 Inguruneko soluzioaren aukeraketa

Behin inguruneko soluzioen multzoa definiturik, hurrengo pausua soluzioen aukeraketa irizpidea ezartzea da. Lehen aipatu bezala, bi dira, nagusiki, erabiltzen diren estrategiak: helburu funtzioa hobetzen duen lehenengo

[3] Hau ez da beti egia; gure probleman murrizketa asko daudenean ausazko soluzioak sortzea oso zaila izan daiteke
[4] Ez optimo globalak eta ezta lokalak ere

soluzioa aukeratzea edo helburua gehien hobetzen duen soluzioa bilatzea. Oro har, inguruneak txikiak direnean bigarren hurbilketa da interesgarriena; hala eta guztiz ere, inguruneak handiak direnean, kostu konputazionala dela eta, bideraezina gerta daiteke.

Lehenengo hurbilketan inguruneko soluzioen «ordenazioa» oso garrantzitsua da, uneko soluzioa hobetzen duen lehenengo soluziora mugituko baikara.

Adibidea 2.1 *Demagun problema baterako soluzioak bektore bitarren bidez kodetzen ditugula. Uneko soluzioa, zeinen helburu funtzioa* 25 *den,* $(0, 1, 1, 0, 1)$ *da. Ingurunea definitzeko* (2) *ekuazioan dagoen funtzioa erabiltzen badugu, inguruneko soluzioak hauexek izango dira:*

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

Inguruneko soluzioak esploratzeko posizioak banan-banan aldatu behar ditugu. Lehenengo posiziotik abiatzen bagara, s_2 *soluzioa izango da helburu funtzioa hobetzen duen lehenengoa, bere ebaluazioa* 24 *baita. Azken posiziotik abiatzen bagara, berriz,* s_3 *soluzioarekin geldituko ginateke, ebaluazioa* 5 *baita. Helburu funtzioa ez ezik, soluzioa ezberdina da eta, hortaz, hurrengo urratsean izango dugun ingurunea ezberdina izango da. Hori dela eta, inguruneko soluzioen azterketa alde batera edo bestera eginez azken soluzioa ezberdina izan daiteke. Inguruneko soluziorik onena aukeratzen badugu, logikoki, ez du axola nondik hasi, beti soluzio berdina topatuko dugu, berdinketarik ez badaude betiere.*

Adibidean inguruneko soluzioak kodeketarekin zerikusia duen ordena erabiltzen da. Horren ordez, esplorazioa ausaz egin daiteke, helburua hobetzen duen soluzioetatik ausazko bat aukeratuz.

Jarraian kodean ingurune azterketaren ordenak duen eragina ikus daiteke. Lehenik eta behin, TSPlib repositorioan dagoen problema bat kargatuko dugu, `tsplib.parser` funtzioa erabiliz. Erabiliko dugun probleman Babariako 29 hiri izango ditugu.

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

Orain, 2-opt ingurunea sortzen dugu, ausazko eta ez-ausazko esplorazioa erabiliz.

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

Inguruneko helburu funtzioa hobetzen duen lehenengo soluzioa aukeratzen badugu, bi ordenazio erabiliz emaitza ezberdina da. Aukeratzeko prozesu hau `first.improvement.selector` funzioaren bidez egin dezakegu.

```
> 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, bi ordenazioak erabiliz emaitza izango da, kasu guztietan inguruneko soluzio guztiak aztertzen baitira. Aukeraketa 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.

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 soluzioa, bigarrena, erebiliko dugun ingurunea eta, azkena, inguruneko soluzioaren aukeraketa. Atal honetan aspektu hauen eragina aztertuko dugu adibide bat erabiliz.

Aurreko ataleko problema (Babariako hiriena) 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
```

Ikus daitkeenez, algoritmo eraikitzaileak (`tsp.greedy`) ematen duen soluzioa ausazkoa baino askoz ere hobea da. Bi soluzio hauei bilaketa lokala aplikatu ahal diegu, swap ingurunea erabiliz. Inguruneko soluziorik onena aukera dezkaegu edo, bestela, hobetzen duen lehenengo soluzioa.

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

Azter dezagun zein nolako hobekuntza lortu dugun bilaketa lokalarekin, ausazko soluziotik abitazen 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
```

Ikus daitekeenez bilaketak topatutako soluzioa hasierakoa baino askoz hobea da. Hare gehiago, bilaketa prozesuan inguruneko soluziorik onenan aukeratzen badugu, hobekuntza handiagoa da. Halere, kontutan hartu behar da ebaluazio kopurua ere handiagoa dela.

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

## [1] 337

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

## [1] 276
```

Algoritmo eraikitzailearekin lortutako soluzioa ausazko soluziotik abitatutako bilaketa lokalak lortutakoak baino hobea da. Halere, soluzio horri bilaketa lokal bat aplikatzen badiogu, soluzioa hobetuko dugu, nahiz eta gutxi izan.

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

## [1] 56
```

Bilaketa greedy hurbilketa erabiliz emaitza hobeak ematen ditu –ausazko soluziotik abitazen garenean, behintzat–, bilaketa espazioa sakonkiago aztertzen delako. Beste era bat bilaketa sakonago egiteko, swap inguruneren ordez 2-opt ingurunea erabiltzea da, lehen ikusi dugun bezala, azken ingurunea hau swap baino askoz ere handiagoa baita.

Hasiera-anizkoitza 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

```
> 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, arazo larri bat du; optimo lokaletan trabatuta gelditzen da. Arazo hau saihesteko – soluzioen dibertsifikazioa suspertzeko, alegia – bilaketa lokalak dituen lau aspektu nagusietan aldaketak sar ditzakegu: hasierako soluzioan, inguruneren definizioan, inguruneko soluzioen aukeraketan eta helburu funtzioaren definizioan. Hurrengo ataletan elementu bakoitzean aldaketak sartzen dituzten algoritmo batzuk aurkeztuko ditugu.

3.1 Hasieraketa anizkoitza

Soluzio bakoitzetik abiatzen garenean optimo lokal batera heltzen gara; soluzio ezberdinetatik abitatzen bagara, optimo lokal ezberdinetara heldu gaitezke. Ideia hau 3.1 sasikodean inplementatuta dagoena da. Algoritmoan agertzen diren *generate_random_solution* eta *local_search* funtzioetan dago prozeduraren mamia. Batak espazioaren esplorazioa egiten du eta besteak, bere izenak adierazten duen bezala, soluzioen areagotzeaz arduratzen da.

Ausazko soluzioak sortzeko hainbat aukera ditugu. Lehendabizikoa uniformeki ausaz sortzea da, hots, pasu bakoitzean probabilitate berdinarekin espazioaren edozein soluzioa aukeratuko dugu, bilaketa lokala hasieratzeko.

Uniformeki ausazko soluzioetatik abiatzea problema gehienetan aukera erraza da; alabaina, ez da oso estrategia adimentsua. Gainera, murrizketa askoko problemetan ausazko soluzio bideragarriak sortzea zaila izan

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

daiteke. Ausazko soluzio «onak» eraikitzeko prozedura bat izanez gero bi arazo hauek saihes ditzakegu. Hain juxtu, hauxe da GRASP algoritmoan egiten dena; hasierako soluzio multzo sortzeko heuristiko gutiziatsu bat erabili, gero *random_solution* funtzioak multzo horretatik soluzioak ausaz ateratzeko. Ausazko soluzioak sortzeko beste aukera bat aurreko pausuan lortutako optimo lokaletik abiatzea da, ILS algoritmoan egiten den legez.

3.1.1 Bilaketa Lokala Iteratua (ILS)

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

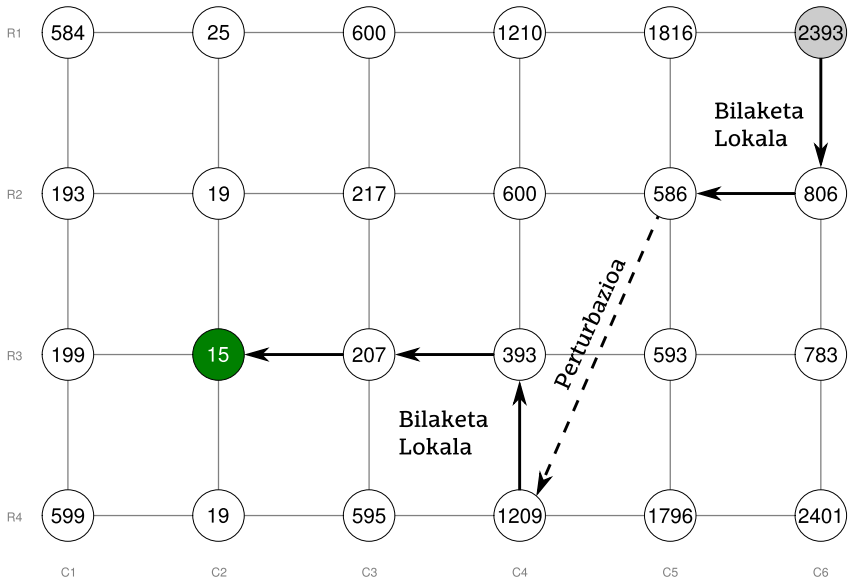
Bi prozedura dira algoritmo honetan definitu behar ditugunak:

- **Perturbazioa** - Bilaketa lokalean soluziotik soluziora mugitzeko aldaketa txikiak egiten dira. Optimo lokaletatik ateratzeko, beraz, egin behar diren aldaketak handiagoak izan beharko dira. Hau 5 irudiak ikus daiteke. Perturbazioa txikiegia izango bazen – (R1,C5) soluziora mugitzea, adibidez –, berriro optimo lokal berdinean amaituko zen bilaketa. Perturbazioa nahiko handia bada, uneko optimo lokalaren erakarpen-arrotik aterako gara eta, definizioz, beste optimo lokal batean trabaturik gelditu gara. Kontutan hartzekoa da perturbazio prozedurak itzulitako soluzioa erabat ausazkoa bada –hau da, perturbazioa oso handia bada –, ILS algoritmoa ausazko hasieraketa anizkoitz algoritmoa bilakatuko dela. Perturbazioaren tamaina egokitzea ez da erraza, problemaren *landscape*-arekin zerikusia baitu eta, lehen ikusi dugun moduan, hau instantziatik instantziara alda daiteke.

  Perturbazioa definitzean inguruneko soluzioak definitzeko mugimendu mota berberak edo beste batzuk erabil daitezke. Esate baterako, permutazioetan oinarritzen den problema batean *2-opt* ingurune operadorea erabiltzen badugu, *k-opt* operadorea erabil daiteke soluzioak perturbatzeko. Perturbazioa trukaketan oinarritu beharrean, txertaketa ere erabil dezakegu, k elementu hartu eta ausazko posizioetan sartuz.

  Perturbazioaren tamaina aurretik finkatu eta bilaketaren zehar aldatu barik mantendu ahal da. Estrategia estatiko honen ordez, estrategia dinamikoak erabil daitezke, non tamaina bilaketaren zehar aldatzen den.

  Soluzioak perturbatzeko prozedura aurreratuetan bilaketaren «historia» erabil daiteke, soluzioaren zein osagai perturbatu eta zein ez erabakitzeko. Estrategia hauek «memoria» kontzeptua erabiltzen dute; erabiltzen diren mekanismoak – memoria motak – tabu bilaketan soluzioak areagotzeko eta dibertsifikatzeko erabiltzen direnen antzerakoak dira.



Irudia 5: ILS algoritmoaren funtzionamendua. Goiko eskumako soluziotik abiatzen bada bilaketa – (R1,C6), grisean nabarmendua dagoen soluziotik, alegia –, (R2,C5) optimo lokalean trabatuta gelditzuko ginateke. Egoera desblokeatzeko soluzioa «perturbatzen» dugu, (R3,C4) soluziora eramanez; hortik abiatuta bilaketa lokala aplikatzen dugu, kasu honetan optimo globalera heldu arte

- **Optimo lokalak onartzeko irizpideak** - Uneko optimo lokala perturbatu ondoren bilaketa lokala aplikatzen da, optimo (berri) bat sortuz. Hurrengo iterazioan lortutako optimo berria edo optimo zaharra perturbatuko dugun erabaki behar da. Bi hurbilketa estremo plantea daitezke: beti optimo berria onartu edo unekoa baino hobea denean bakarrik onartu. Lehendabiziko estrategiak dibertsifikazioa suspertzen du; bigarrenak, berriz, areagotzeko egokia da. Ohikoena tarteko zerbait erabiltzea da, optimo zaharraren eta berriaren arteko ebaluazioaren diferentzia kontutan hartuz. Esate baterako, optimoak era probabilistikoan onar daitezke, Boltzmann-en distribuzioa erabiliz, gero *simmulated annealing* algoritmoan ikusiko dugun bezala.

ILS-a `iterated.local.search` funtzioan inplementatuta dago. Funtzio honen gainontzeko parametroak `basic.local.search` funtzioaren berberak dira, inplmentazioa funtzio horretan oinarritzen baita. Horrez gain, hiru parametro berri izango digutu:

- `perturb` - Parametro honen bidez soluzioak perturbatzeko erabili behar den funtzioa pasatuko diogu funtzioari. Funtzio honek parametro bakarra izan beharko du, perturbatu behar den soluzioa, eta itzultzen duen emaitza perturbatutako soluzioa izan beharko da.

- `accept` - Parametro hau funtzio bat izan behar da, soluzio berriak noiz onartzen ditugun definitzen duena. Gutxienez parametro bat izan beharko du, `delta`, soluzio berria eta zaharraren ebaluazioen arteko diferentzia. `num.restarts` - Bilaketa optimo lokal batean trabatuta gelditzen denean, zenbat alditan berrabiarazi behar dugun bilaketa, perturbatutako soluziotik.

Ikus dezagun adibide bat, TSP 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, hau da, trukaketa; eragiketa hau ausaz egiteko `shuffle` funtzioa erabil dezakegu.

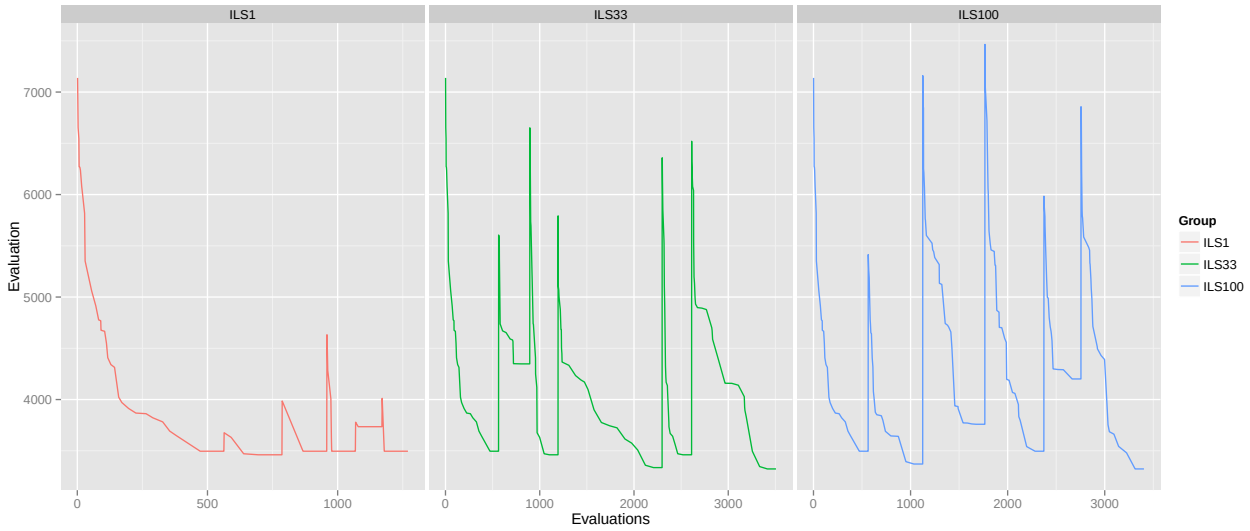
```
> f <- paste("http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95"
+          ,"/XML-TSPLIB/instances/burma14.xml.zip",sep="")
> burma.mat <- tsplib.parser(f)

## 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 <- first.improvement.selector
> th.accpet <- threshold.accept
> th <- 0
```

Soluzioen perturbazioa kontrolatzeko zenbat trukaketa egiten diren kontrola dezakegu. Kopuru hori aldatu ahal izateko, gure perturbazio funtzioaren parametro bat izan beharko da[5].

```
> perturb.2opt <- function(solution , ratio , ...)
+   shuffle(permutation = solution , ratio = ratio)
>
> r <- 5
> 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 ,
+                                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 bilaketen progresioa 5 irudian ikus daitezke. Hirurak soluzio berdinatik 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 trukatuz perturbatzen dira; guztira 14 posizioak direnez, trukaketa bakar bat egiten da.

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



Irudia 6: ILS algoritmoaren progresioa 14 hiriko TSP problema batean. Ezkerretik eskuinera, perturbazioaren ratioa 0.01, 0.25 eta 1 da.
5

Erdiko grafikoan perturbazioa %25ekoa da, 3 trukaketa, alegia. Azeken grafikoan, berriz, 14 trukaketa egiten dira (posizioen %100). Perturbazio txiki bat erabiltzen dugunean, lortutako soluzioaren ebaluazioa optimo lokalaren antzerakoa da (agertzen diren "pikoak" ez dira oso altuak, alegia); geroz eta perturbazio handiagoa orduan eta diferentzia handiagoa soluzio berria eta optimoaren artean.

3.1.2 GRASP algoritmoa

Optimizazio problemak ebazteko ohiko da metodo eraikitzaileak erabiltzea. Aurreko kapituluan ikusi genuen bezala, algoritmo hauek soluzio pausuz pausu eraikitzen dute, urrats bakoitzean aukera guztietatik onena hautatuz. Era honetan, soluzio onak sortzen dira baina, berdinketak egon ezean, instantzia bakoitzeko soluzio bakarra lortzen dugu.

Metodo hauen bidez lortutako soluzioak onak izan arren ez daukate zergatik optimoak izan, ez globalki eta ezta lokalki ere. Hori dela eta, behin soluzioa sortuta, bilaketa lokal bat erabil daiteke soluzioa areagotzeko.

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

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

Adibidea 3.1 *Demagun motxilaren problema ebatzi nahi dugula. Oso sinplea den algoritmo eraikitzaile bat hauxe da: kalkulatu, motxilan sartzek ditugun elementu bakoitzeko balioa/pisua ratioa eta gero, pausu bakoitzean, motxilan sartu ahal diren elementuetatik (pisu-muga gaindiarazi ez dutenak) ratiorik handiena duena aukeratu.*
Algoritmo hau GRASP algoritmoa inplementatzeko egoki daiteke. Algoritmoaren iterazio bakoitzean soluzio bat eraikiko dugu, pausu bakoitzean ratiorik handiena duen elementua aukeratu beharrean ratio handienak duten %α soluzioen artetik bat ausaz aukeratuz. Behin soluzioa eraikita, bilaketa lokala aplikatuko dugu lortutako soluzioa areagotzeko.

Adibidean planteatzen den estrategia ausazko soluzio onak sortzeko procedura bat inplementatu beharko dugu.

Funtzioaren pamateroak problemarenak izango dira, gehi kandidatoen zerrendaren luzeera, ratio baten bidez adieraziko duguna. Hasteko, zenbait datu atera eta aldagai batzuk hasieratzen ditugu. Besteak beste, soluzio "hutsa" sortzen dugu, elementurik ez duena.

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

Soluzioan elementuak sartzeko begizta bat izango dugu, motxila beteta ez dagoen bitartean errepikatuko dena. Begiztaren lehenengo pausua sarturik gabeko elementuei antzemate diegu eta, uneko iterazioan, gure hautagai zerrenda izan behar duen tamaina kalkulatu egiten dugu (gogoratu tamaina urratsean ditugun aukera kopuruaren araberakoa dela).

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

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

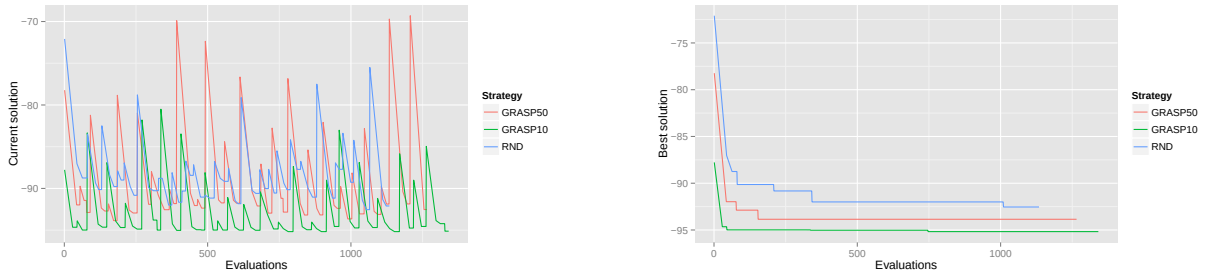
```
    cl <- sort(ratio[non.selected] , decreasing=TRUE)[1:cl.n]
    selected <- sample(cl,1)
```

Amaitzeko, aukeratutako elementua soluzioan sartzen dugu eta, motxila betetzen ez bada, aurrera jarraitzen dugu. Bestela, soluzioa deusestatzen dugu eta begizta amaitzen dugu.

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

Sortu dugun futnzioa GRASP algoritmoa guztiz ausazkoa den hasieraketa anizkoitzarekin alderatzeko erabil dezakegu, `cl.size` parametroarekin jolastuz. Lehenik eta behin, ausazko problema bat sortuko dugu. Kasu honetan 50 item izango ditugu eta hauen balioa bi multzotan banatuko dugu. Elementuen erdirako balioa 0 eta 10 arteko ausazko zenbakiak dira; beste erdirako, ausazko zenbakiak 0 eta 25 tartean. Kasu guztietan pisua balioarekiko proportzionala da, faktorea ausazko zenbakia 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))
```



(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.

GRASP eta hasieraketa anizkoitzan oinarritzen den beste edozein algoritmo erabiltzeko [**?**] funtzioa daukagu. Orain arte ikusitakoen antzerakoa da funtzio hau, baina hasierako soluzioa argumentu moduan pasatu beharrean, soluzioak sortzeko funtzio bat pasatu behar diogu. Parametro asko direnez, funtzioen argumentuak antolatzeko beste era bat erabiliko dugu. 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 (ia) parametro guztiak ezarrita, `cl.size` parametroari 3 balio ezberdin esleituko diogu, 1, 0.5 eta 0.1. Lehenengo kasuan hautagai zerrendan aukera guztiak egongo direnez, funtzioak ausazko soluzioak sortuko ditu, hots, ausazko hasieraketa anizkoitza erabiliko du bilaketa lokalak. Beste kasuetan, berriz, aukera guztietatik %50 eta %10 erabiltzen dira bakarrik, hurrenez hurren.

```
> rnd.ls <- do.call (multistart.local.search , args)
>
> 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 bilaketaren progresioa ikus daiteke. Ezkerreko grafikoan uneko soluzioaren eboluzioa jasotzen da. Ikus daitekeen bezala, berabiarazte bakoitzean bilaketa soluzio txarragoetatik hasten da (gorago daudenak).

GRASP algoritmoan, berriz, hasierako soluzioak hobeak dira, eta baita ere lortzen den emaitza. Hori argi ikusten da eskumako grafikoan, non uneko soluziorik onenaren eboluzioa erakusten den.

3.2 Inguruneko soluzioen hautaketa

Definizioz, optimo lokalak diren soluzioen inguruko soluzio guztiak bera baino txarragoak dira; hortaz, ez dugu soluzio hoberik aukeratzerik eta, ondorioz, bilaketa lokala trabatuta gelditzen da. Bilaketa prozesuari amaiera ez emateko, soluzio hoberik egon ezean, helburu funtzioa hobetzen ez duten soluzioak onar ditzakegu. Estrategia honi esker, txarragoak diren soluzio batzuetatik pasatuz gero bilaketa espazioaren eskualde hobeetara ailega gintezke.

Soluzio «txarrak» aukeratzeko bi estrategia daude. Lehendabizikoan helburua hobetzen ez duen soluzio bat aukeratzean sortutako «galera» kontutan hartzen da, era probabilistikoan zein deterministan. Algoritmorik ezagunena *suberaketa estokastikoa –simulated snnealing* [7, 11] ingelesez– izenekoa da, zeinek probabilitate-banaketa parametriko bat erabiltzen duen aukeraketa egiteko. Algoritmo honetan inspiratutako beste hainbat algoritmoak proposatu dira, *demon algoritm* [9] eta *threshold accepting* [3, 8] algoritmoak, besteak beste.

Estrategia honen beste interpretazioa Gloverrek 1986an proposaturiko *tabu bilaketa* [5] –*tabu search* ingelesez– algoritmoarena da. Kasu honetan helburu funtzioa hobetzen ez duten soluzioak aukeratzen dira baldin eta soilik baldin ingurune osoan helburua hobetzen duen soluziorik ez badago. Estrategia honen arriskua dagoeneko bisitatu ditugun soluzioak bisitatzea den legez, «memoria»erabili beharra dago zikloak saihesteko.

3.2.1 Suberaketa Simulatua

Tresnak edo piezak sortzeko prozesatzen direnean, metalek hainbat propietate gal dezakete, euren kristal-egituran eragindako aldaketak direla eta. Propietate horiek berreskuratzeko metalurgian «suberaketa» prozesua erabiltzen da; metal pieza oso tenperatura handira berotzen da, gero astiro hozten uzteko. Tenperatura igotzean metalaren 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 hauskorragoa 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-egiturako «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; Kirpatrick-ek bere algoritmoari *simmulated annealing*, suberaketa simulatua, izena eman zion eta hauxe da gaur egun hedatuen dagoena.

Algoritmoaren funtzionamendua sinplea da oso. Soluzio batetik (s) txarragoa den beste soluzio batera (s') 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; sistemaren energia «tenperatura»ren bidez neurtuko dugu. Beste era batean esanda, uneoro sistemak T tenperatura izango du eta, zenbat eta tenperatura altuago, orduan eta errazago izango da energia-mugak gainditzea.

Zehazki, soluzio batetik bestera mugitzeko gainditu behar den energia-muga gainditzen denetz erabakitzeko Boltzmann probabilitate-banaketa erabiltzen da:

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

Ikus daitekeenez, distribuzio esponentzial honetan muga gainditzeko probabilitatea –hots, helburua hobetzen ez duen soluzioa onartzekoarena– tenperaturarekiko proportzionala da; energia diferentziarekiko, ostera, alderantzizko proportzionala da.

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

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 !stop_condition
7     while !equilibrium
8        s' = random_neighbor(s)
9        ΔE = f(s') − f(s)
10       if ΔE < 0
11          s = s'
12          if (f(s) < f(s*)) s* = s
13       fi
14       else
15          e^(−ΔE/T) probabilitatearekin s = s'
16    done
17    T=update_temperature(T)
18 done
```

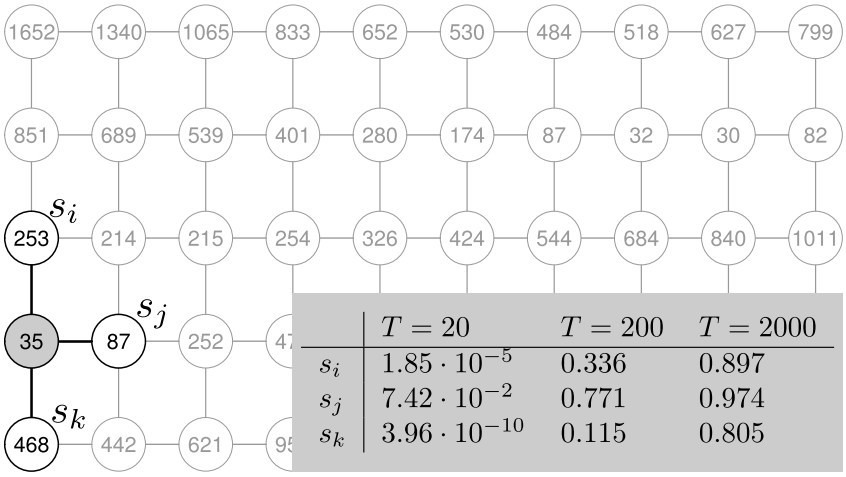
Algoritmoa 3.3: Suberaketa Simulatuaren sasikodea

Suberaketaren gakoa hozte-abiadura da eta, era berean, suberaketa simulatuaren zati garrantzitsuena izango da. Izan ere, hasieran T balio handiak erabiliko ditugu, ia edozein soluzio onartzeko, eta gero, astiro, T txikiagotuko dugu, optimo lokal batean trabatuta gelditu arte. 3.3 algoritmoan suberaketa simulatuaren sasikodea ikus daiteke.

Suberaketa simulatuaren motako algoritmoak diseinatzean lau aspektu hartu behar ditugu kontutan:

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

  Tenperatura hasieratzeko bi estrategia ditugu. Lehendabizikoa tenperatura oso altua erabiltzea da; dibertsifikazio ikuspegitik interesgarria izan arren, estrategia hau konputazionalki garestia izan daiteke. Beste estrategian bilaketa espazioaren itxura aztertuko dugu, ingurunean dauden soluzioen arteko diferentziak nolakoak diren jakiteko. Gero, informazio hau onarpen ratio edo probabilitate ezagun bat lortzeko behar dugun tenperatura finkatzeko erabil daiteke [6, 1].



	$T = 20$	$T = 200$	$T = 2000$
s_i	$1.85 \cdot 10^{-5}$	0.336	0.897
s_j	$7.42 \cdot 10^{-2}$	0.771	0.974
s_k	$3.96 \cdot 10^{-10}$	0.115	0.805

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

Adibidea 3.2 *Demagun 10 tamainako TSP-aren instantzia bat ebatzi nahi dugula. Kostu matrizean dagoen 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 elementuen batura izango da. Hori dela eta, $f_g = 7.28 \cdot 10 = 78.2$ problema honetarako ebaluazio funtzioaren goi-muga bat da*[a]*. Era berean, matrizeko distantziarik txikiena 2.5 izanik, behe-muga kalkula ditzakegu: $f_b = 2.5 \cdot 10$. Beraz, edozein soluzio aukeratzeko hasierako probabilitatea finkatzen badugu – 0.75, adibidez –, bi muga hauek hasierako tenperatura kalkulatzeko erabil dezakegu, edozein bi soluzioen arteko ebaluazioaren diferentzia $f_g - f_b$ baino txikiagoa izango dela baitakigu:*

$$P = 0.75 = e^{-\frac{f_g - f_b}{T_0}} = e^{-\frac{78.2-25}{T_0}}$$
$$T_0 = -\frac{78.2 - 25}{\ln(0.75)} = 184.93$$

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

[a] Kontutan hartuz aipatutako baturan matrizeko elementuak ezin direla errepikatu, goi-muga birfindu daiteke 10 elementurik handienak batuz

- **Oreka lortzeko iterazio kopurua -** Tenperatura balio bakoitzeko zenbait iterazio egin behar dira – hau da, zenbait inguruneko soluzio aztertu behar da – «oreka» lortu arte. Behin oreka lorturik, tenperatura eguneratu behar da, berriro oreka bilatzeko. Lehenengo pausua, beraz, oreka lortzeko behar dugun iterazio kopurua ezartzea da. Ohikoena inguruneko tamainaren araberako iterazio kopurua finkatzea da. Beste era batean esanda, aurretik ρ ratioa finkatuko dugu; gero, tenperatura bakoitzeko $\rho|N(s)|$ soluzio ebaluatuko ditugu, non $|N(s)|$ uneko soluzioaren ingurunearen tamaina den.

  Estrategia honetan iterazio kopurua finkatuta dago, baina badaude beste estrategiak non kopurua tenperatura bakoitzeko aldakorra den. Adibide gisa, tenperatura soluzio berri bat onartzen dugun bakoitzean alda dezakegu; batzuetan ebaluatzen dugun lehendabiziko soluzioa onartuko dugu eta, bestetan, hainbat

soluzio probatu beharko ditugu, bat onartu arte. Kasu honetan, beraz, iterazio kopurua aldakorra da. Ingurunean soluzio on asko badaude, iterazio gutxi beharko ditugu; kontrako kasuan, inguruneko soluzio gehienak txarrak badira, iterazio gehiago beharko ditugu uneko soluzio aldatzeko.

- **Tenperatura jaitsieraren abiadura -** Hau da, ziurrenik, algoritmoaren osagairik nagusiena. Hainbat formula erabil daitezke tenperatura eguneratzeko. Hona hemen batzuk:
  - *Lineala*: $T_i = T_0 - i\beta$, non T_i i iterazioko tenperatura den. Eguneraketa honetan tenperatura beti positiboa izan behar dela kontrolatu behar dugu, ekuazioak tenperatura negatiboak itzuli baitaitezke.
  - *Geometrikoa*: $T_i = \alpha T_{i-1}$. $\alpha \in (0, 1)$ abiadura kontrolatzen duen parametroa da eta, ohikoena, 0.5 eta 0.99 tartean egotea da.
  - *Logaritmikoa*: $T_i = \frac{T_0}{log(i)}$. Abiadura hau oso motela da baina, nahiz eta praktikan oso erabilgarria ez izan, algoritmoaren konbergentzia demostratuta dago ekuazio hau erabiltzen denean.

  Funtzio guzti hauek monotonoak dira, hau da, iterazio bakoitzean tenperatura beti jaisten da. Edonola ere, problema batzuetan funtzio ez-monotonoek hobeto funtziona dezakete[6].

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

Ikusitako algoritmoetan soluzioen onarpena probabilistikoa da; alabaina, suberaketa simulatuaren kontzeptua era deterministan ere inplementa daiteke. Honen adibidea da Deabru Algoritmoa [9] – *Deamon Algorithm*, ingelesez –. Hasiera batean Creutz-ek algoritmoa simulazio molekularrak egiteko proposatu zuen, baina optimizazio problemak ebazteko ere egoki daiteke. Algoritmoan soluzioak onartuko direnetz erabakitzeko tenperatura erabili beharrean «deabru» bat erabiltzen da; deabru honek uneoro E_D energia du. Soluzio bakoitza aztertzerakoan, subaeraketa simulatuan legez, ΔE kalkulatzen da eta, une horretan $\Delta E > E_D$ bada, soluzioa onartzen da – suberaketa simulatuan bezala, $\Delta E < 0$ denean soluzioa onartzen 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, txarragoa denean, berriz, energia galtzen du – energia nahiko baldin badu, betiere –. Algoritmo honen abantaila sinpletasuna da, Boltzmann distribuzioa ebaluatzeko beharrik ez baitugu.

3.2.2 Tabu bilaketa

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

1977an lehenengo aldiz proposatuta, tabu bilaketak, optimo lokaletan trabaturik ez geratzeko, bilaketa soluzio txarragoetara bideratzea baimentzen du. Neurri honek prozesua amaigabeko ziklo batean sartzeko arriskua sortzen du, egindako bidea berregiteko aukera baitago. Arazo hori saihesteko, tabu bilaketak, bisitatutako soluzioen historikoa gordetzen du.

Oinarrizko tabu bilaketa, bilaketa lokal gutiziatsuan oinarritzen da, non «tabu zerrenda» deitzen den bisitatutako soluzio multzo bat gordeko den uneoro. Urrats bakoitzean tabu ez diren – bideragarriak diren, alegia – inguruneko 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.

Bisitatutako soluzio guztiak gordetzen dituen zerrenda mantentzea ez da bideragarria; hori dela eta, tabu zerrendan azkenengoz bisitatutako soluzioak bakarrik gordeko ditugu. Algoritmoaren iterazio bakoitzean, aukeratutako soluzioa tabu zerrendara gehituko da, eta zerrendatik soluzio bat aterako da – tabu zerrendak FIFO

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

Tabu Bilaketa

```
1  input: intensify eta diversify operadoreak
2  input: intensify_condition, diversify_condition eta stop_condition baldintzak
3  input: 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 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

pilak dira, hau da, sartzen lehendabizikoa den elementua ateratzeko ere lehendabizikoa izango da–. Azken soluzioak bakarrik gordetzen direnez tabu zerrendari epe laburreko memoria esaten zaio.

Tabu zerrendan soluzioak gordetzen baditugu, eta zerrendaren tamaina k bada k tamainako zikloak ekiditeko gai izango gara. Edonola ere, eraginkortasuna dela eta, soluzio osoak maneiatzeak kostu handia lekarke. Hori dela eta, aukera egokiagoak ere aurki ditzakegu literaturan, atributuak gordetzea, besteak beste. Atributuak soluzioen zatiak, mugimenduak, edo soluzioen arteko desberdintasunak izan ohi dira. Soluzioen elementu hauek ebazten ari garen problemaren menpekoak dira eta, hortaz, aukera ugari proposatu daitezke, kasu bakoitzeko 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 trukatuko ditugu. Era honetan, tabu zerrendan bi posizio hauek gorde ditzakegu, alderantzizko trukaketa tabu bihurtuz. Esate baterako, uneko soluzioa* [13245] *bada eta* [31245] *soluziora mugitzen bagara, hurrengo urratsetan lehenengo eta bigarren posizioak trukatzea debekatuko dugu. Problemaren arabera, beste irizpide batzuk erabil genitzake. Adibide gisa, adibidean lehenengo posizioan* 1*a eta bigarrenean* 3*a egotea tabu egin genezake.*

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

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

Tabu zerrendaz gain, bestelako aukera konplexuagoak, ere proposatu dira. Epe motzeko memoria erabiltzeaz gain, bilaketa prozesuan 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 soluzioen osagaien frekuentziak gordeko ditugu; frekuentzia hauek bisitatu ez ditugun eremuei antzemateko erabil daitezke – hau da, bilaketa dibertsifikatzeko –.

Adibidea 3.4 *TSP-rako soluzioak eraikitzeko hiri bakoitzetik zein hirira mugituko garen erabaki behar dugu. Algoritmo eraikitzaile tipikoan erabaki hori kostu matrizea begiratuz hartzen da, uneko hiritik bisitatu gabeko gertuen dagoen hira 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 loturetan zentratzeko. Epe luzeko memorian, berriz, bisitatu ditugun soluzio guztien informazioa gordeko dugu. Era honetan, bilaketa esploratu gabeko eremuetara eraman nahi badugu, gutxien erabilitako loturak erabiliz soluzioak sor ditzakegu, bilaketa prozesua bertatik abiatzeko.*

3.3 Bilaketa espazioaren itxura aldaketa

3.4 Optimizazio problemen «itxura»

Bilaketa lokalean uneko soluziotik honen inguruan dagoen soluzio batera mugitzen gara beti, hau da, soluzio bakoitzetik zenbait soluzioetara mugi gaitezke. Hori dela eta, bilaketa espazioa grafo baten bidez adieraz daiteke non erpinak soluzioak dira eta ertzek mugimendu posibleak adierazte dituen; 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 performantzian eta, beraz, algoritmoak diseinatzerakoan kontuan hartu beharreko elementua da. Zentsu horretan, aintzat hartzekoa da problemaren arabera ez ezik, *landscape*-a problema instantzia konkretuen arabera ere alda daitekeela.

Soluzio bakarrean oinarritzen diren algoritmoekin amaitzeko bilaketa espazioaren *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 murriztekotan.

3.4.1 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 ingurune definizioa aldatuko bagenuke?. Adibide gisa, demagun optimizazio problema bat dugula non soluzioak permutazioen bidez kodetzen ditugun. 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, hau da, edozein bi posizio trukatuz lortutako soluzioak txarragoak dira. 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 txertaketa erabiltzen duen inguru nerako optimoa ez izatea.

Ideia hau *Variable Neighborhood Descent* (VND) algoritmoan erabiltzen da bilaketa lokala optimo lokaletan trabaturik gera ez dadin. 3.5 algoritmoan VND-aren sasikodea ikus daiteke.

Algoritmoan ikusten den bezala, VND-an ingurune funtzio bakarra izan beharrean multzo bat daukagu. Lehenengo funtzioarekin hasita, iterazio bakoitzean uneko soluzioaren inguruneko soluziorik onena bilatzen dugu. Inguruneko soluzio guztiak txarragoak badira – soluzioa uneko ingurunerako optimo lokala bada, alegia – hurrengo ingurune funtziora pasatzen gara; ingurune funtzio gehiagorik egon ezean bilaketa amaitzen da.

VND algoritmoaren sasikodea

```
1  input: N = {N_1, ..., N_k} ingurune funtzioak
2  input: s hasierako soluzioa
3  i = 1
4  s* = s
5  while i ≤ k do
6     Bilatu s', N_i(s*) inguruneko 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

Iterazio batean inguruneko soluzio berri batera pasatzen garen bakoitzean, berriro hasierako ingurune definiziora bueltatzen 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, helburu funtzioa hobetzen duen soluzio bat topatu arren uneko ingurune definizioa mantentzen dugu optimo lokal batera heldu arte. Behin optimo lokal batean, hurrengo ingurune definiziora mugitzen gara, VND-an legez, lortutako soluzioa ingurune guztietarako optimo lokala izan arte. 3.6 algoritmoak VNS-aren sasikodea erakusten du.

3.5 *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-metodoetan – *smoothing methods*, ingelesez – iterazio bakoitzean jatorrizko helburu funtzioa eraldatu – leundu – egiten da, optimo lokal kopurua gutxitzeko asmoz; helburu funtzio berria erabiliz bilaketa lokala aplikatzen da. Behin bilaketa trabaturik – 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 iterazioa leuntze-maila geroz eta txikiago eginez, azkenengo iterazioan problemaren jatorrizko helburu funtzioa erabiliko dugu, problemarako soluzioa topatzeko.

Helburu funtzioa nola leundu problemaren araberakoa da. Edonola ere, kasu guztietan algoritmo inplementatu ahal izateko leuntze-parametro bat 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.

Oinarrizko VNS algoritmoaren sasikodea

```
1  input: local_search bilaketa algoritmoa
2  input: N = {N_1, ..., N_k} ingurune funtzioak
3  input: s hasierako soluzioa
4  i = 1
5  s* = s
6  while i ≤ k do
7      Aukeratu ausaz soluzio bat s' ∈ N_i(s*)
8      s'' =local_search(s*,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

Smoothing metodoen sasikodea

```
1  input: smoothing (f,α) helburu funtzioa eraldatzeko funtzioa
2  input: local_search(s,f) bilaketa lokala
3  input: update(α) faktorea eguneratzeko funtzioa
4  input: s hasierako soluzioa; α_0 hasierako faktorea; f helburu funtzioa
5  s* = s; α = α_0
6  while α > 1 do
7      f' =smoothing(f;α)
8      s* =local_search(s*,f')
9      α =update(α)
10 done
```

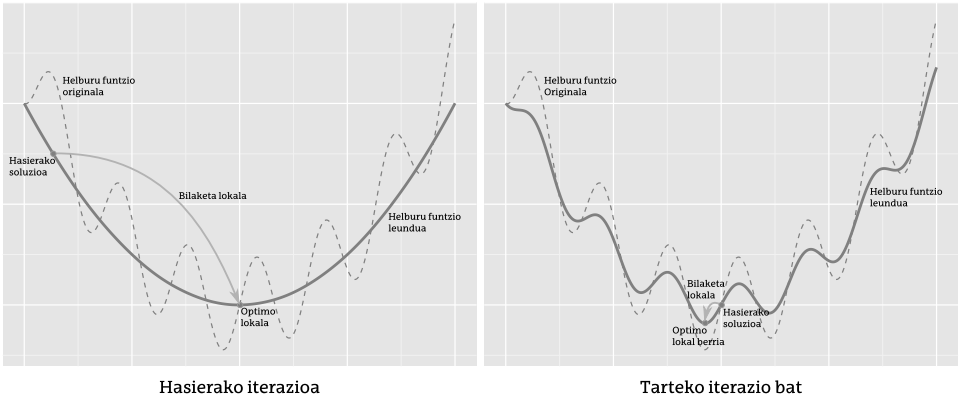
Algoritmoa 3.7: *Smoothing* algoritmoaren sasikodea

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

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

non $\bar{d}$ distantzien batez-bestekoa eta d_{ij} jatorrizko matrizearen elementuak diren. Distantzia matrizea normalizaturik badago – distantzia guztiak 1 edo txikiagoak badira[a]*, alegia – α parametroa oso handia denean distantzia guztiak batez-bestekoari hurbiltzen dira, $0 \leq (d_{ij} - \bar{d}), (\bar{d} - d_{ij}) < 1$ baita. Limite horretan soluzio 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.

[a] Kontutan hartu behar da, normalizatuta ere soluzio optimoa, hau da, balio minimoa duena, ez dela aldatzen



Irudia 9: *Smoothing* algoritmoaren funtzionamendua. Iterazio bakoitzean hasierako helburu funtzioa maila bateraino leuntzen da eta bilaketa lokala aplikatzen da.

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 Computationl 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(1âĂŞ3):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.