

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 dago bilaketa prozesurik – eta, hortaz, ez ditugu kapitulu honetan azalduko. Horren ordeztan, 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 ordezkatzeko dugu.

Idea honetan hainbat algoritmo oinarritzen dira, bakoitzak 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 nagusia inguruarena da – *neighborhood* ingelesez – eta, hortaz, problema bat ebazterakoan arreta handiz diseinatu beharreko osagaia da. Ingurune funtzioak edo operadoreak¹, soluzio bakoitzeko, bilaketa espazioaren azpimultzo bat itzultzen du.

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

Soluzio baten ingurunean dauden soluzioak antzerakoak izatea espero dugu; alabaina, ingurune operadoreek soluzioen kodeketarekin dihardute. 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.

¹Programazio testuinguruetan – sasikodetan, adibidez – «operadore» terminoa erabiliko dugu soluzioak maneiatzeko erabiltzen diren funtzioetarako. Definizio matematikoetan, berriz, «funtzio» terminoa erabiliko dugu, gehien bat.

Adibidea 1.1 Lokaltasuna FSS problemen *Datu meatzaritzan, gainbegiraturako datu base bat daukagunean, sailkatzaile funtzioak eraikitzea edo ikastea ataza oso ohikoa da. 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 azpi-multzo bat aukeratzea oso pausu ohikoa da; prozesu hau, ingelesez feature subset selection, FSS deitzen dena, optimizazio problema bat da. FSS problemarako soluzioak sektore bitarren bidez kode daitezke, bit bakoitzak dagokion aldagaia azpi-multzoan dagoenetz adierazten duelarik.

Sektore 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.

Lokaltasun falta dela eta, ingurune operadore ohikoena flip aldaketan oinarritzen dena da; ingurune soluzioak sortzeko uneko soluzioaren posizio batean dagoen balioa aldatzen da, hau da, 0 bada 1ekin ordezkatzeko da eta 1 bada, 0rekin. Operadore honekin FSS problemaren beti bermatzen da lokaltasuna, ingurunean dauden edozein bi soluzio aldagai bakar bateko diferentzia izango baitute.

Soluzioak sektoreen 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 hau da:

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

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

Bektore errealek - 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 errealek direnez infinitu soluzio izango ditugu edozein soluzioen ingurunean. Euklidearra distantziarik ezagunena izan arren, badira beste metrika batzuk ere – Manhattan- edo Chebyshev-distantziak, besteak beste –.

Bektore kategorikoak eta bitarrak - Bektoreetan dauden aldagaiak kategorikoak direnean, bi sektoreen 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 posizio desberdintasunak neurtzen ditu. Hamming-distantzia inguruneak definitzeko erabiltzen denean ohikoena 1 distantziara dauden soluzioak erabiltzea da, hau da:

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

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

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

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

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 trukutzen dira. Adibidez, [21345] permutazioa badugu, 1. eta 3. posizioak trukutzen 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:

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

Operadore honi *2-opt neighborhood* deritzo, bi posizio bakarrik trukutzen baitira. Era berean, operadorea hedatu ahal da trukaketa gehiago eginez. Hedatzeaz gain, operadorea murriztu ere egin ahal da, bakarrik elkarren ondoan dauden posizioak trukatzuz.

Ingurunearen tamainari dagokionez, posizio jarraien trukaketak eginez n ingurune soluzio izango ditugu; edozein bi posizio trukutzen baditugu, berriz, ingurunearen tamaina $n(n-1)$ izango da.

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 haxe izango da:

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

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.

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

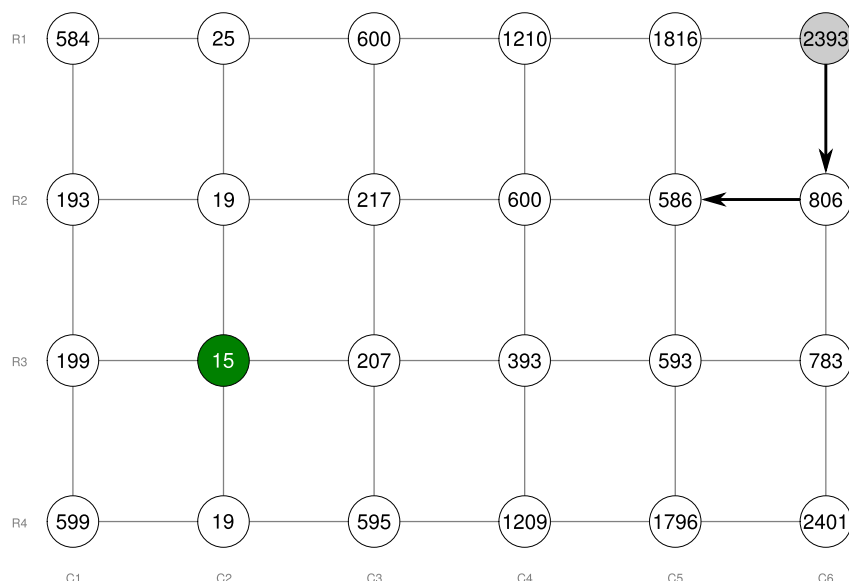
1.2 Optimo lokalak

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

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

Ekuazio hau optimo globalarenaren oso antzerakoa da; 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. Haxe da, hain zuzen, bilaketa lokalaren ezaugarriarik – eta, aldi berean, arazorik – nagusia. Aintzat hartzekoa da optimo lokalak, nahiz eta bere ingurune soluziorik onenak izan, nahiko soluzio txarrak izan daitezkeela, 1 irudian erakusten den bezala. Irudi honetan kapituluaren 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 besteari ingurunean dago.



Irudia 1: Optimo lokalaren adibidea. Goiko eskumako soluziotik abiatzen bada bilaketa – (R1,C6), grisean nabarmendua dagoen soluziotik, alegia –, pausu bakoitzean aukerarik onena aukeratzeko 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.

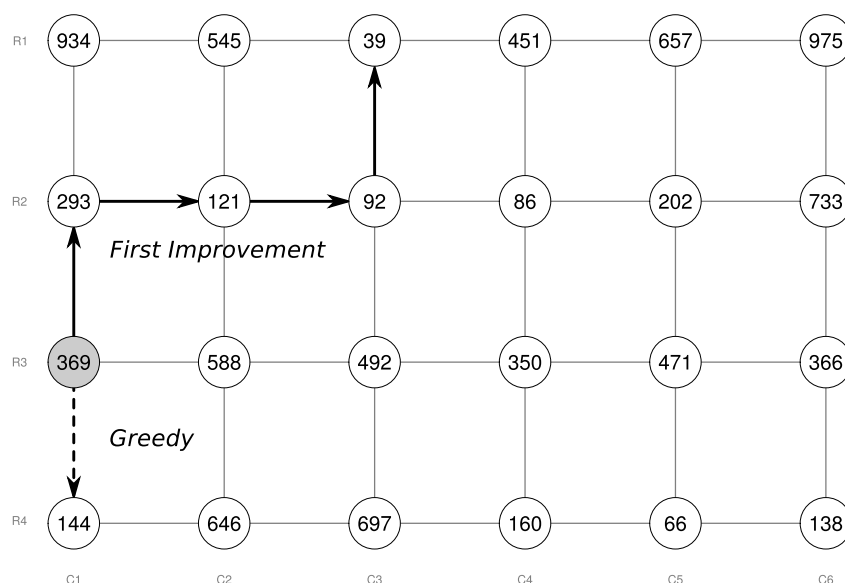
Adibidea 1.2 1 irudian agertzen diren geziek algoritmoak egiten duen bidea erakusten dute, (R1,C6) soluziotik abiatuta. Pausu bakoitzean inguruneko soluziorik onena aukeratzeko badugu, algoritmoa bi pausutan trabatuta geldituko da (R2,C5) soluzioan; soluzio honen ebaluazioa 586 da eta, bere ingurunean 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 gainera.

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

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 eta, gero ikusiko dugun moduan, oso garrantzitsua da problemen zailtasuna aztertzerakoan.

Definizioa 1.2 erakarpen-arroa Izan bitez N ingurune funtzioa, f helburu funtzioa, $A(s; f, N) : \mathcal{S} \rightarrow \mathcal{S}$ 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 aukeratzeko den ezartzen du; ondorioz, egiten dugun ibilbidean eragina handia izan dezake, 2 irudiak ikus daitekeen bezala.



Irudia 2: Inguruneke 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 – inguruneke 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 –; hurrengo soluzioa ingurunean dagoen soluziorik onena izango da beti. Estrategia hau erabiliz pausu bakar batean (R4,C1) optimo lokalera ailegatzen gara

1.3 Optimizazio problemaren «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; 1 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 diseinatzekoan kontuan hartu beharreko elementua da. Zentsu horretan, aintzat hartzekoa da problemaren arabera ez ezik, *landscape*-a problema instantzia konkretuen arabera ere alda daitekeela.

2 Bilaketa lokala

2 algoritmoan oinarrituko 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 uneko 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.

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



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 helburu funtzioa hobetzen duen soluzio bat bilatzen dugu. Horrelakorik badago, uneko soluzioa ordezkatzeko dugu; ez badago, bilaketa amaitzen da.

2.1 Hasierako soluzioa

Lehen aipatu bezala, bilaketa lokala soluzio batetik abiatuko da. Nondik hasi bilaketa oso garrantzitsua da, horren arabera optimo lokal batean edo bestean amaituko baitu bilaketa. Hau 1 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 estrategiak:

- **Ausazko soluzioak erabiltzea** - Ausaz aukeratzen da bilaketa espazioan dagoen soluzio bat. Metodo honen abantaila bere sinpletasuna da, ausazko soluzioak sortzea erraza izan ohi baita². 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 eraikitzea** - 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 aukeratu – ingelesez metodo hauei *constructive greedy* deritze, hau da algoritmo eraikitzaile gutziatsuak edo jaleak –. Nahiko soluzio onak eman arren, hauek ez daukate zergatik optimoak³ 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 Inguruneke soluzioaren aukeraketa

Behin inguruneke soluzioen multzoa definiturik, hurrengo pausua soluzioen aukeraketa irizpidea ezartzea da. Lehen aipatu bezala, bi dira, nagusiki, erabiltzen diren estrategiak: helburu funtzioa hobetzen duen lehenengo soluzioa aukeratzeko edo helburua gehien hobetzen duen soluzioa bilatzeko. 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 inguruneke soluzioen «ordenazioa» oso garrantzitsua da, uneko soluzioa hobetzen duen lehenengo soluziora mugituko baikara.

²Hau ez da beti egia; gure problematan murrizketa asko daudenean ausazko soluzioak sortzea oso zaila izan daiteke

³Ez optimo globalak eta ezta lokalak ere



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, 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 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, inguruneke soluzioen azterketa alde batera edo bestera eginez azken soluzioa ezberdina izan daiteke.

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

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 heuristikoen gain, tamaina handiko inguruneak era eraginkorrean zehazki aztertzeke algoritmoak ere badaude; hauetako adibide bat *dynasearch*[2] algoritmoa da.

3 Bilaketa lokalaren hedapenak

Aurreko atalean ikusi dugun legez, bilaketa lokala soluzioak areagotzeko prozedura egokia izan arren, arazo larri bat du; optimo lokaletan tratatuta gelditzen da. Arazo hau saihesteko – soluzioen dibertsifikazioa suspertzeko, alegia – bilaketa lokalak dituen lau aspektu nagusietan aldaketak sar ditzakegu: hasierako soluzioan, ingurunearen definizioan, inguruneke soluzioen aukeraketan eta helburu funtzioaren definizioan. Hurrengo atalean 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 helduko gara. Ideia hau 3.1 sasikodean inplementatuta dagoena da. Algoritmoan agertzen diren *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 aukeratzeko 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 daiteke. Ausazko soluzio «onak» eraikitzeke prozedura bat izanez gero bi arazo hauek saihas ditzakegu. Hain juxtu, hauxe da GRASP algoritmoan egiten dena; hasierako soluzio multzo sortzeko heuristiko gutziatsua 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.

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  $s = random\_solution$ 
5 while  $\neg stop\_criterion$ 
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  $\neg stop\_criterion$ 
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

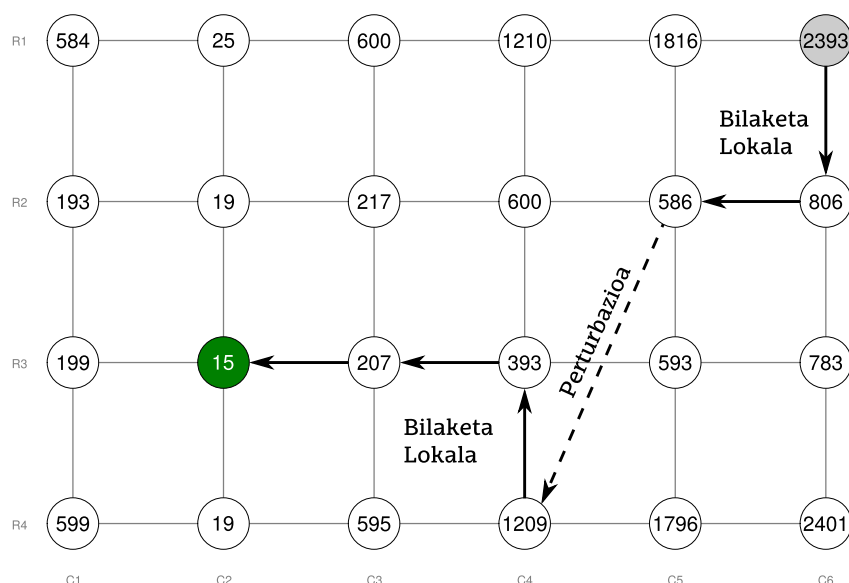
3.1.1 Bilaketa Lokala Iteratua (ILS)

Bilaketa lokala berrabiarazteko 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 3 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 geldituko gara. Kontutan hartzekoa da perturbazio prozedurak itzultitako 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 instantzietatik instantziara alda daiteke.

Perturbazioa definitzean inguruneke soluzioak definitzeko mugimendu mota berberak edo beste batzuk erabil daitezke. Esate baterako, permutazioetan oinarritzen den problema batean *2-opt* ingurune oper-



Irudia 3: ILS algoritmoaren funtzionamendua. Goiko eskumako soluziotik abiatzen bada bilaketa – (R1,C6), grisean nabarmendua dagoen soluziotik, alegia –, (R2,C5) optimo lokalean trabatuta geldituko ginateke. Ego- era desblokeatzeko soluzioa «perturbatzen» dugu, (R3,C4) soluziora eramanez; hortik abiatuta bilaketa lokala aplikatzen dugu, kasu honetan optimo globalera heldu arte

adorea erabiltzen badugu, *k-opt* operadorea erabil daiteke soluzioak perturbatzeko. Perturbazioa trukake- tan 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 aurreratueta bilaketaren «historia» erabil daiteke, soluzioaren zein osagai perturbatu eta zein ez erabakitzeke. Estrategia hauek «memoria» kontzeptua erabiltzen dute; er- abiltzen diren mekanismoak – memoria motak – tabu bilaketan soluzioak areagotzeko eta dibertsifikatzeko erabiltzen diren antzerakoak dira.

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

3.1.2 GRASP algoritmoa

Optimizazio problemak ebazteko ohiko da metodo eraikitzaileak erabiltzea. Aurreko kapitulu- an ikusi genuen bezala, algoritmo hauek soluzio pausuz pausu eraikitzen dute, urrats bakoitzean aukera guztietatik onena hau- tatuz. 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.



Idea 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. Idea 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 motxilararen problema ebatzi nahi dugula. Oso sinplea den algoritmo eraikitzaile bat hauze da: kalkulatu, motxilaran sartzek ditugun elementu bakoitzeko balioa/pisua ratioa eta gero, pausu bakoitzean, motxilaran sartu ahal diren elementuetatik (pisu-muga gaindiarazi ez dutenak) ratorik handiena duena aukeratu.*

Algoritmo hau GRASP algoritmoa implementatzeko egoki daiteke. Algoritmoaren iterazio bakoitzean soluzio bat eraikiko dugu, pausu bakoitzean ratorik handiena duen elementua aukeratu beharrean ratio handienak duten α soluzioen artean bat ausaz aukeratzuz. Behin soluzioa eraikita, bilaketa lokala aplikatuko dugu lortutako soluzioa areagotzeko.

3.2 Inguruneko soluzioen hautaketa

Definizioz, optimo lokalak diren soluzioen inguruko soluzio guztiak bera baino txarragoak dira; hortaz, ez dugu soluzio hoberik aukeratzetik 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. Lehendabizikoa helburua hobetzen ez duen soluzio bat aukeratzeko sortutako «galera» kontutan hartzen da, era probabilistikoa zein determinista. 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 aukeratzeko 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-egituraren 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; Kirkpatrick-ek bere algoritmoari *simulated annealing*, suberaketa simulatua, izena eman zion eta hauze 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, $\Delta E = f(s') - f(s)$.



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

Energia-muga hau gainditzeko sistemak energia behar du; sistemaren energia «tenperatura»ren bidez neur-tuko 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 erabakitze-ko Boltzmann probabilitate-banaketa erabiltzen da:

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

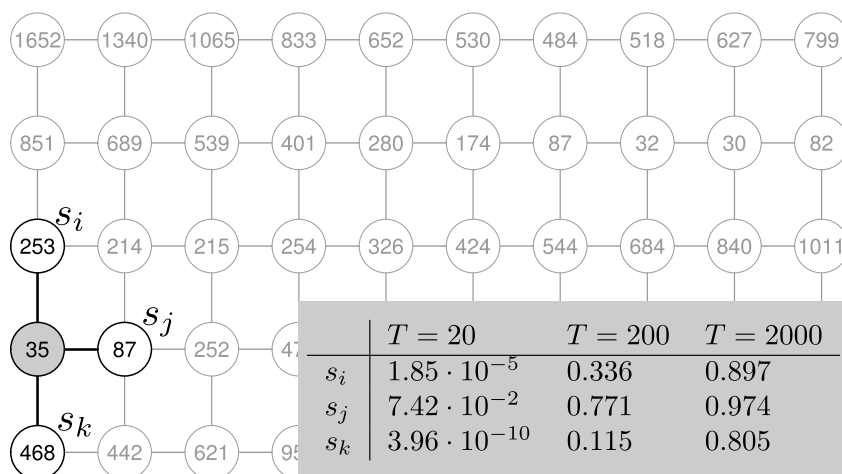
Ikus daitekeenez, distribuzio esponentzial honetan muga gainditzeko probabilitatea –hots, helburua hobet-zen ez duen soluzioa onartzekoarena– temperaturarekiko proportzionala da; energia diferentziarekiko, oster-a, alderantzizko proportzionala da.

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 hobe ba ita, ondorioz, beti onartzen da.

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 txiki-agotuko 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 oinarritzko bilaketa lokala bihurtuko da. 4 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.



Irudia 4: 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 temperatura ezberdinekin. Ikus daitekeenez, temperatura baxua denean edozein soluzioa aukeratzeko probabilitatea oso baxua da; temperatura oso altua denean, berriz, edozein soluzioa hartuta litekeena onartzea da.

Temperatura hasieratzeko bi estrategia ditugu. Lehendabizikoa temperatura 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 temperatura finkatzeko erabil daiteke [6, 1].

Adibidea 3.2 Demagun 10 tamainako TSP-aren instantzia bat ebatzi nahi dugula. Kostu matrizean dagoen baliorik handiena – hau da, bi hiriaren 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, $f_g = 7.28 \cdot 10 = 72.8$ 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 temperatura kalkulatzeko erabil dezakegu, edozein bi soluzioen arteko ebaluazioaren diferentzia $f_g - f_b$ baino txikiagoa izango dela baitakigu:

$$P = 0.75 = e^{-\frac{f_g - f_b}{T_0}} = e^{-\frac{72.8 - 25}{T_0}}$$

$$T_0 = -\frac{72.8 - 25}{\ln(0.75)} = 184.93$$

Beraz, hasierako temperatura 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 10 elementurik handienak batuz

- **Oreka lortzeko iterazio kopurua** - Temperatura balio bakoitzeko zenbait iterazio egin behar dira – hau da, zenbait inguruneako soluzio aztertu behar da – «oreka» lortu arte. Behin oreka lorturik, temperatura eguneratu behar da, berriro oreka bilatzeko. Lehenengo pausua, beraz, oreka lortzeko behar dugun iterazio kopurua ezartzea da. Ohikoena inguruneako tamainaren arabera iterazio kopurua finkatzea da. Beste era



batean esanda, aurretik ρ ratioa finkatuko dugu; gero, temperatura 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 temperatura bakoitzeko aldakorra den. Adibide gisa, temperatura 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, ingurunean soluzio gehienak txarrak badira, iterazio gehiago beharko ditugu uneko soluzio aldatzeko.

- **Temperatura jaitsieraren abiadura** - Hau da, ziurrenik, algoritmoaren osagairik nagusiena. Hainbat formula erabil daitezke temperatura eguneratzeko. Hona hemen batzuk:
 - *Lineala*: $T_i = T_0 - i\beta$, non T_i i iterazioko temperatura den. Eguneraketa honetan temperatura beti positiboa izan behar dela kontrolatu behar dugu, ekuazioak temperatura negatiboak itzuli baitaitezke.
 - *Geometrikoa*: $T_i = \alpha T_{i-1}$. $\alpha \in (0, 1)$ abiadura kontrolatzen duen parametroa da eta, ohikoa, 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 temperatura beti jaisten da. Edonola ere, problema batzuetan funtzio ez-monotonoek hobeto funtziona dezakete⁴.

- **Algoritmoa gelditzeko irizpidea** - Aurreko puntuan ikusi dugu 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, probabilitatea oso txikiarekin bada ere. Hori dela eta, algoritmoa gelditzeko baldintzaren bat beharko dugu. Irizpide hedatuena temperatura 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 implementa 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 diren erabakitzeak temperatura erabili beharrean «deabru» bat erabiltzen da; deabru honek uneoro E_D energia du. Soluzio bakoitza aztertzerakoan, suberaketa simulatua legez, ΔE kalkulatu da eta, une horretan $\Delta E > E_D$ bada, soluzioa onartzen da – suberaketa simulatua 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 hobe denean, deabruak energia irabazten du eta, txarragoa denean, berriz, energia galtzen du – energia nahiko baldin badu, betiere –. Algoritmo honen abantaila sinpletasuna da, Boltzmann distribuzioa ebaluatzeke beharrik ez baitugu.

3.2.2 Tabu bilaketa

Tabu bilaketa – *tabu search*, ingelesez – 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.

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

⁴Temperatura 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



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

inguruneke 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 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 trukatu 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 1a eta bigarrenean 3a egotea tabu egin genetzake.*

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 egokitzeke.

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 eraikitzeke hiri bakoitzetik zein hirira mugituko garen erabaki behar dugu. Algoritmo eraikitzaile tipikoan erabaki hori kostu matrizea begiratzuz 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 eramanez nahi badugu, gutxien erabilitako loturak erabiliz soluzioak sor ditzakegu, bilaketa prozesua bertatik abiatzeko.*

3.3 Bilaketa espazioaren itxura aldaketa

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 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 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* eragiketari oinarritutako ingurunea –. Izan bedi [1432] soluzioa, ingurune eta problema honetarako optimo lokala dena, hau da, edozein bi posizio trukatzuz 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 ingurunerako optimoa ez izatea.

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

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

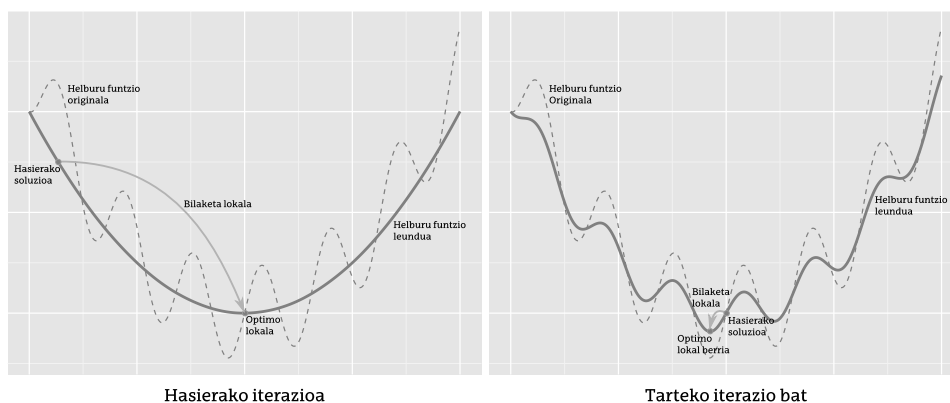
VND algoritmoaren sasikodea

```

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



Irudia 5: *Smoothing* algoritmoaren funtzionamendua. Iterazio bakoitzean hasierako helburu funtzioa maila bateraino leuntzen da eta bilaketa lokala aplikatzen da.

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.

3.4 *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-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 arabera da. Edonola ere, kasu guztietan algoritmo inplemtatu ahal izateko leuntze-parametro bat beharko dugu. Parametro hau handia denean, helburu funtzioa asko



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

Smoothing metodoen 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

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.



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-besteako distantziara gerturatuz, adibidez. Demagun ondoko matrizea definitzen dugula:

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

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-besteak 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.

^aKontutan hartu behar da, 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&A33):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.