Bilaketa Heuristikoak

Teoria eta Adibideak R Lengoaian

Borja Calvo
Josu Ceberio
Usue Mori

Universidad del País Vasco
Euskal Herriko Unibertsitatea

Borja Calvo · Josu Ceberio · Usue Mori

BILAKETA HEURISTIKOAK

Teoria eta adibide praktikoak R lengoaian

CIP. Unibertsitateko Biblioteka

Calvo Molinos, Borja

Bilaketa heuristikoak [Recurso electrónico]: teoria eta adibide praktikoak R lengoaian / Borja Calvo, Josu Ceberio, Usue Mori. – Datos. – Bilbao : Servicio Editorial. Universidad del País Vasco / Euskal Herriko Unibertsitatea, [2017]. – 1 recurso en línea : PDF (138 p.)
Modo de acceso: World Wide Web
ISBN: 978-84-9082-688-1

1. Heurística. 2. Optimización combinatoria. 3. Algoritmos. 4. R (Lenguaje de programación)
I. Ceberio Uribe, Josu, coaut. II. Mori Carrascal, Usue, coaut.

(0.034)007.7
(0.034)519.1

UPV/EHUko Euskara Zerbitzuak sustatua eta zuzendua, Euskarazko ikasmaterialgintza sustatzeko deialdiaren bitartez.

© Servicio Editorial de la Universidad del País Vasco
Euskal Herriko Unibertsitateko Argitalpen Zerbitzua

ISBN: 978-84-9082-688-1

Gaien aurkibidea

1 Oinarrizko kontzeptuak 1
1.1 Sarrera 1
1.1.1 Hastapen historikoa 1
1.2 Optimizazio-problemak 3
1.2.1 Problemen konplexutasuna 4
1.2.2 Problema klasikoak 8
1.3 Optimizazio-problemak ebazten 16
1.3.1 Helburu-funtzioa 19
1.3.2 Bilaketa-espazioa: soluzioen kodeketa 20
1.3.3 Bilaketa-espazioa: murrizketak 22
1.3.4 Algoritmoak 25

2 Soluzio bakarrean oinarritutako algoritmoak 29
2.1 Kontzeptu orokorrak 29
2.1.1 Soluzioen inguruneak 30
2.1.2 Optimo lokalak 36
2.2 Bilaketa lokala 39
2.2.1 Hasierako soluzioaren aukeraketa 43
2.2.2 Inguruneko soluzioaren aukeraketa 43
2.2.3 Bilaketa lokaleko elementuen eragina 46
2.3 Bilaketa lokalaren hedapenak 48
2.3.1 Hasieraketa anizkoitza 48
2.3.2 Inguruneko soluzioen hautaketa 58
2.3.3 Optimizazio-problemen «itxuraaldaketa» 69
2.3.4 *Smoothing* algoritmoak 71

3 Populazioetan oinarritutako algoritmoak 75
3.1 Algoritmo ebolutiboak 76
3.1.1 Urrats orokorrak 77
3.1.2 Algoritmo genetikoak 82
3.1.3 Estimation of Distribution Algorithms 90
3.2 Swarm Intelligence 94
3.2.1 *Ant Colony Optimization* 95

3.2.2 Particle Swarm Optimization 105

4 Algoritmoen konparazio enpirikoa 111
4.1 Problemaren instantzien aukeraketa 112
4.2 Konparazioaren baldintzak 113
4.3 Parametroen aukeraketa 114
4.4 Algoritmoak exekutatu eta emaitzak aztertu 118
4.5 Konparazio grafikoa 121
4.6 Test estatistikoak 125

1 Oinarrizko kontzeptuak

Lehenengo kapitulu honetan optimizazioaren ikuspegi globala aurkeztuko dugu, oinarrizko kontzeptuak azalduz. Kapitulua bi zatitan dago banatuta; lehenengoan optimizazio-problemak izango dira aztergai eta, bereziki, horien konplexutasuna, horixe baita metodo heuristikoak erabiltzeko motibaziorik garrantzitsuena. Heuristikoak dira, bigarren zatiaren protagonistak. Zehazki, heuristikoak diseinatzeko eta erabiltzeko kontuan eduki beharreko kontzeptuak aztertuko ditugu.

Sarrera

Optimizazioa oso kontzeptu hedatua da, askotan baitarabilgu –konturatu barik bada ere–. Problema baten aurrean *soluzio bat baino gehiago* daudenean, soluzio horien *kalitatea* neurtzeko eraren bat izanez gero, *soluziorik onena* bilatzea izango da optimizazioaren helburua. Definizio orokor horren barruan problema mota asko sartzen diren arren, liburu honetan, *optimizazio konbinatorio*ko problemetan zentratuko gara batez ere. Optimizazio-problemak aspalditik aztertuak izan diren arren, matematika aplikatuan duela ez askorik bereizi den ikerkuntza-arloa da optimizazioa.

Hastapen historikoa

Lehen Mundu Gerra amaitu zenean, garaileek Alemaniari oso baldintza gogorrak inposatu zizkioten Versalles-eko itunean; besteak beste, Alemaniak armada izatea zeharo debekatuta zeukan. Are gehiago, itun horrek jasotzen zi-

tuen baldintza ekonomikoen ondorioz 1920ko hamarkadan Alemaniako egoera nahiko larria izan zen; hala ere, krisi-momentu horretan, pertsona batek promes egin zuen herrialdea larrialdi horretatik aterako zuela... Pertsona hori Hitler zen eta 1933an hauteskundeak irabaziz boterea lortu zuen; orduan, Bigarren Mundu Gerra ekarriko zuen gertakizun-sekuentzia bat hasi zen.

1934. urtean Hitler-ek Alemaniako berrarmatze-prozesua agindu zuen, eta 1935eko udaberrirako bere aireko armada –Luftwaffea– Britainia Handikoaren parekoa zela aldarrikatzen hasi zen. Nazien hegazkin bonbaketarien mehatxua arazo larri bihurtu zen Britainiar Gobernuarentzat, eta, hortaz, airedefentsa antolatzeari ekin zion. Aireko estatu-idazkariak «Imperial College of Science and Technology»ko errektoreari aireko defentsaren arazoa aztertuko zuen batzordea sortzeko eskatu zion; batzorde horren lanaren ondorioz, 1935eko udan, Robert Watson-Watt-ek radarra asmatu zuen. Tresna oso erabilgarria izan arren, etekin guztia ateratzeko, defentsa-sisteman –behatokiak, ehiza-hegazkinak, artilleria antiaereoa, ...– integratu beharra zegoen, eta arduradunak laster konturatu ziren ez zela lan erraza izango. Izan ere, defentsa operazioen antolakuntza, bere osotasunean, oso arazo konplexua zen. Hori dela eta, problema ikuspegi matematikotik ikertzen hasi ziren; eta hortik, gaur egun ezaguna den Ikerkuntza Operatiboa sortu zuten.

Bigarren Mundu Gerran zehar operazioen antolakuntza «matematikoa»k arrakasta handia izan zuen britainiar armadan, eta, hortik, Estatu Batuetako armadara hedatu zen. 1945ean, gerra amaitu zenerako, mila pertsonatik gora zeuden Ikerkuntza Operatiboan lanean britainiar armadan.

Gerra amaitu ostean, Europako herrialdeen egoera oso larria zen: herrialdeak suntsituta, baliabideak gerran xahututa ... Hurrengo urteetan, herrialdeak berreraikitzeko erronkari aurre egiteko, gerrak iraun bitartean operazio militarrak antolatzeko garatutako metodologia matematikoak bereziki egokiak zirela konturatu ziren agintariak. Adibide gisa, Dantzigek –gerran AEBko armadan ziharduena– 1947an Ikerkuntza Operatiboko algoritmorik ezagunena, Simplex algoritmoa, proposatu zuen.

Arlo berri honek ikertzaileen arreta erakarri zuen, eta gerraosteko garaian hedapen nabarmena izan zuen. Hasierako urteetan planteatzen ziren algoritmoek soluzio zehatzak lortzea zuten helburu baina teknika hauek problema mota konkretu batzuk ebazteko erabil zitezkeen bakarrik –problema linealak, adibidez–. 1970eko hamarkadan zientzialariek problemen konplexutasuna aztertzeari ekin zioten. Bestalde, konputagailu pertsonalak agertu ziren merkatuan. Problema batzuetan *konplexutasun - tamaina*-ren konbinazioaren ondorioz, soluzio zehatza lortzea ezinezkoa zen. Hori dela eta, merkatuan algoritmo heuristikoak agertzen hasi ziren, zeinek, soluzio onena ez bermatu arren, soluzio onak ematen zituzten denbora laburrean.

Metodo heuristikoak oso interesgarriak izan arren, desegokiak ziren problema berrietan berrerabiltzeko. Hori dela eta, 1975etik aurrera, bilaketa heuristiko edo metaheuristika deritzen hurbilketak hasi ziren garatzen zientzialariak. Hona hemen adibide eta data batzuk:

- 1975 - John Hollandek algoritmo genetikoak proposatu zituen

- 1977 - Fred Gloverrek *scatter search* algoritmoa proposatu zuen
- 1983 - Kirkpatrick eta lankideek *simulated annealing* edo suberaketa simulatua proposatu zuten
- 1986 - Fred Gloverrek tabu-bilaketa algoritmoa proposatu zuen
- 1986 - Gerardo Beniek eta Jing Wangek *swarm intelligence* kontzeptua proposatu zuten
- 1992 - Marco Dorigoek *Ant Colony Optimization* (ACO) algoritmoa proposatu zuen
- 1996 - Muhlenbeinek eta Paassek *Estimation of Distribution Algorithms* (EDAs) kontzeptua proposatu zuten

Optimizazio-problemak

Egunero erabakiak hartzen dira nonahi; enpresetan, zientzian, industrian, administrazioan... Geroz eta konpetitiboagoa den mundu honetan, erabakiak hartzeko prozesu hori arrazionalki hurbildu beharra daukagu.

Erabaki-hartzea hainbat pausotan bana daiteke. Lehendabizi, problema formalizatu behar da, gero matematikoki modelatu ahal izateko. Behin problema modelaturik, soluzio onak topatu behar ditugu problemarentzat –soluzio optimoa zein den erabaki, alegia–.

Problema erreal batean dihardugunean, hartutako erabaki optimoak praktikan jarri beharko genituzke, egiaztatzeko ea funtzionatzen dutenetz; arazoren bat egonez gero, atzera joz problemaren formulazioa berrikusteko.

1.1 adibidea. *Demagun plastikozko piezak ekoizten dituen enpresa bateko logistika-sailean lan egiten dugula. Lantegian zenbait makina, lehengaiak eta ekoitzitako piezak biltzeko biltegi bat, eta abar daude. Igandero, asteko eskaera aztertuz, plangintza egin behar dugu; zer pieza ekoitzi lehenago, zer makinatan, noiz bidali bezeroei... Plangintza era eraginkorrean eginez gero, eskaera gehiago asetzeko gai izango gara, eta, hortaz, diru gehiago irabaziko du enpresak.*
Plangintza optimoa bilatzeko, lantegiak dituen ezaugarriak –biltegiaren tamaina, makinen berezitasunak, denborak,...– aztertu eta problema formalizatu egin behar dugu, zeren matematikoki nola hurbildu eta ebatz daitekeen erabakitzeko ezinbestekoa baita.

Optimizazio-problemak formalizatzean bi elementuri atzeman beharko diegu. Lehenik eta behin, problemaren soluzio guztien multzoari, *soluzio bideragarrien espazio* edo *bilaketa-espazio* deiturikoari. Eta, bigarren, soluzio optimoa topatzeko optimotasuna definitzen duen *helburu-funtzioari*. Soluzio bideragarrien multzoa S sinboloa erabiliz adieraziko dugu, eta helburu funtzioa, berriz, f erabiliz:

$$f : S \to \mathbb{R}$$

Optimizazio-problemak *optimo globala* –hau da, soluziorik onena– topatzean dautza. Optimotasuna helburu-funtzioaren araberakoa izan arren, beti bi aukera izango ditugu: funtzioa maximizatzea edo minimizatzea. Hemendik aurrera, azalpen guztiak bateratzeko asmoarekin, xedea helburu-funtzioa *minimizatzea* dela joko dugu[1].

1.1 definizioa. Minimizatze-problema. *Izan bedi S soluzio bideragarrien multzoa eta $f : S \to \mathbb{R}$ helburu-funtzioa. Minimizazio-problema optimo globala $s^* \in S$ topatzean datza non $\forall s \in S, f(s^*) \leq f(s)$*

Optimizazio-problema bat era eraginkorrean ebazteko, hiru ezaugarriok aztertu beharko ditugu:

- Problemaren tamaina
- Problemaren konplexutasuna
- Eskuragarri ditugun baliabideak (denbora, konputazio-baliabideak, etab.)

Problemak ebazteko behar den denborari erreparatuz –baliabide garrantzitsuena izan ohi dena–, problema mota oso ezberdinak aurkitu ditzakegu. Hala nola, kasu batzuetan denbora oso murriztuta egongo da, kontrol-problemetan gertatzen den legez[2]. Beste muturrean diseinu-problemak ditugu, non helburua ahalik eta soluziorik onena topatzea den denborari erreparatu gabe. Optimizazio-problema gehienak bi kasu horien erdibidean kokatzen dira, eta beraz, denbora-muga bat izango dugu ebazteko.

Problemaren konplexutasuna edozein izanda ere, beti tamaina batetik aurrera ezinezkoa izango da metodo zehatzen bidez ebaztea. Aurrerago ikusiko dugun bezala, kasu horietan metodo heuristikoetara jotzea beharrezkoa izango da.

Problemen konplexutasuna

Problema baten konplexutasuna da hura ebazteko existitzen den algoritmo eraginkorrenaren konplexutasuna da. Algoritmoak problemak pausoz pauso ebazteko erabiltzen diren prozedurak dira. Problema mota bakoitzetik *instantzia* ezberdinak izan ditzakegu, eta instantzia horiek *tamaina* bat izango dute. Konplexutasunak problemaren tamaina handitzen den heinean ebazteko behar den denbora edota memoria nola handitzen den neurtzen du; hau da, behar diren baliabideen hazkundearen abiadura tamainarekiko.

---

[1] Gure helburu-funtzioa maximizatu nahi izanez gero, funtzio berri bat definituko dugu, $g = -f$.

[2] Problema hauei *real-time optimization* deritze ingelesez.

1.2 adibidea. *Demagun hiri batzuen zerrenda daukagula zeinen koordenatu geografikoak ezagutzen ditugun. Hirien arteko distantzia kalkulatzea problema konputazional bat da. Hiri zerrenda bakoitza problemaren instantzia bat izango da; adibidez, zerrendan Donostia, Bilbo eta Gasteiz baditugu, 3 tamainako instantzia bat izango dugu.*

Oso era orokorrean, algoritmoen konplexutasunak n tamainako problema bat ebazteko behar den pauso kopurua neurtzen du. [3]

Konplexutasunari buruz hitz egiten denean, kontrakorik esan ezean, *kasurik txarrena* aztertu ohi da; halere, kasurik onena eta batezbestekoa ere aztertzen dira, algoritmoen portaeraren irudi zehatzagoa lortzeko. Konplexutasuna neurtzean, pauso kopurua zehatza baino gehiago, kopuru horrek problemaren tamainarekiko nola «eskalatzen» duen interesatzen zaigu.

1.3 adibidea. *Jo dezagun aurreko adibidean distantzia euklidearra erabil dezakegula hirien arteko distantziak kalkulatzeko. Problema ebazteko, edozein bi hirien arteko diferentzia kalkulatzeko bi biderketa, batuketa bat eta erro karratu bat beharko ditugu; hau da, bikote bakoitzeko lau eragiketa beharko ditugu. Gure problemaren tamaina n bada –zerrendan n hiri baditugu–, $\frac{n(n-1)}{2}$ distantzia kalkulatu beharko ditugu –kontuan hartuz i eta j hirien arteko distantzia behin bakarrik kalkulatu behar dugula eta hiri batetik hiri berdinera dagoen distantzia ez dugula kalkulatu behar–. Beraz, guztira, $4\frac{n(n-1)}{2} = 2n(n-1)$ eragiketa beharko ditugu.*

Esan dugun legez, balio zehatza ez da garrantzitsuena. Esate baterako, ez du garrantzirik pauso kopurua $10n^2$ edo $0.5n^2$ izateak, kontua eragiketa kopuruaren progresioa tamainarekiko koadratikoa dela baizik; ideia hori O notazioaren bidez adierazi ohi da.

1.2 definizioa. O notazioa. *Algoritmo batek $f(n) = O(g(n))$ konplexutasuna dauka, n_0 eta c konstante positiboak existitzen badira zeinentzat $\forall n > n_0, f(n) \leq c \cdot g(n)$ betetzen den.*

Beraz, aurreko adibiderako $g(n) = n^2$ izatea nahikoa da definizioa betetzeko; izan ere, $2n^2 > 2n(n-1)$ eta, ondorioz, $c = 2$ bada, ekuazioa beteko da, $n > 0$ bada betiere. Hori kontuan hartuz, beraz, distantzien matrizea kalkulatzeko algoritmoaren konplexutasuna $O(n^2)$ dela esango dugu.

Konplexutasun-maila ezberdinak defini daitezke; 1.1 taulak maila ohikoenak biltzen ditu. Oinarrizko operazio bakoitzak milisegundo behar duela jotzen badugu, taulan, n tamaina ezberdinetarako, konplexutasun desberdinetako problemen exekuzio denborak daude kalkulatuta.

[3] Denboran ez ezik, espazioan ere neur daiteke konplexutasuna; kasu horretan, pauso kopurua baino gehiago, behar dugun memoria aztertu beharko genuke. Edonola ere, optimizazio-problematan denbora-konplexutasuna aztertu ohi da.

1.1 taula. Konplexutasun-mailak gehi exekuzio-denbora tamainaren arabera. Adibide gisa, erreferentzia-operazioaren iraupena milisegundo bat da.

Maila	Notazioa	$n = 1$	$n = 5$	$n = 10$	$n = 20$
Lineala	$O(n)$	0.001 seg	0.005 seg	0.01 seg	0.020 seg
Koadratikoa	$O(n^2)$	0.001 seg	0.025 seg	0.100 seg	0.4 seg
Kubikoa	$O(n^3)$	0.001 seg	0.125 seg	1 seg	8 seg
Esponentziala	$O(2^n)$	0.002 seg	0.032 seg	1.024 seg	17.4 min
Faktoriala	$O(n!)$	0.001 seg	0.12 seg	1 ordu	7.7 milurteko
Hiper-esponentziala	$O(n^n)$	0.001 seg	3.12 seg	115 urte	*

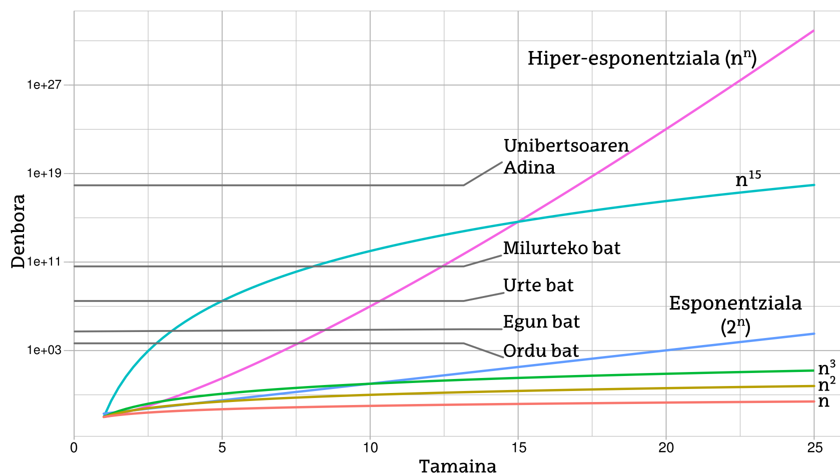
* $3.23 \cdot 10^8$ aldiz unibertsoaren adina

Aintzat hartzekoa da konplexutasun-analisiak $n \to \infty$ limitean dugun portaera adierazten duela. Izan ere, n finitua denean konplexutasun-mailek zentzua gal dezakete, hurrengo adibidean ikus daitekeen bezala.

1.4 adibidea. *Demagun problema bat ebazteko hiru algoritmo ditugula: P, polinomikoa, $O(n^{10})$; E, esponentziala, $O(5^n)$, eta H, hiperesponentziala, $O(n^n)$. Dakigunez, $n \to \infty$ kostuaren araberako ordena $P < E < H$ da, baina zer gertatzen da n finitua denean?.*
$n = 7$ bada, algoritmoen kostuak hauexek izango dira: $P = 7^{10}; E = 5^7; H = 7^7$. Hau da, kostuaren arabera ordenatzen baditugu, kostu txikiena duena E da, eta ez P –izatez, P kosturik handiena duena da–. Hau $5 < n < 10$ betetzen da, $n < 5$ denean egoera zertxobait ezberdina baita. Demagun $n = 2$ dela eta, beraz, kostuak $P = 2^{10}; E = 5^2; H = 2^2$ direla. Kasu horretan E-ren ordez H, algoritmo hiperesponentziala, da kosturik txikienekoa. Hau da, kostuaren araberako ordena konplexutasunarekiko alderantzizkoa da, $H < E < P$.

1.1 irudiak konplexutasun-ordena batzuen funtzioak erakusten ditu. Y ardatzak denbora segundotan adierazten du eta eskala logaritmikoan dago. Grafikoan ikus dezakegun bezala, funtzio linealak, koadratikoak eta kubikoak ordubeteko mugatik behera mantentzen dira, eta, haien hazkunde-abiadura ikusita, hor mantenduko dira problema-tamaina (n) handietarako ere. Halere, konplexutasun polinomikoaren kasuan, berretzailea handitzen den heinean, denboraren kurba geroz eta azkarrago hazten da, batez ere problemaren tamaina txikia denean. Adibide gisa, problemaren tamaina txikia denean –n, 15 baino txikiagoa denean, zehazki–, $O(n^{15})$ da konplexutasunik «garestiena» hiper-esponentzialaren gainetik. Dena dela, esan dugun moduan, konplexutasuna aztertzean abiadura da interesatzen zaiguna; hau da, grafikoan agertzen diren kurben deribatua edo malda. Grafikoan argi ikusten da, $n \to \infty$ denean, funtzio hiper-esponentzialaren hazkunde abiadura dela handiena, gero esponentzialarena eta polinomikoena.

Hasieran esan dugun legez, problema baten konplexutasuna problema hori ebazteko ezagutzen den algoritmorik eraginkorrenaren konplexutasuna da.



1.1 irudia. Konplexutasun-ordena tipikoen hazkunde-abiadura n-rekiko.

Adibidez, bektore bat ordenatzeko ezagutzen den metodorik eraginkorrena –kasurik txarrenean– $O(n \log n)$ ordenakoa da, eta, ondorioz, bektoreen ordenazioa $O(n \log n)$ mailakoa dela esaten da.

Problema konputazionalak bi klasetan banatzen dira, konplexutasunaren arabera: P, problema polinomikoak, eta NP, problema ez-polinomikoak.

- **P klasea** - Klase honetan dauden problementzat badago algoritmo determinista bat problemaren edozein instantzia denbora polinomikoan ebazten duena.
- **NP klasea** - Klase honetan dauden problementzat ez da existitzen algoritmo deterministarik problema denbora polinomikoan ebazten duenik.[4]

NP klasean, NP-osoa deritzon azpiklase bat definitzen da (*NP-complete*, ingelesez). Problema bat NP-osoa dela esango dugu baldin eta *edozein* NP problema, denbora polinomikoan, problema hori bihurtu baldin badaiteke.

Sailkapen hori erabaki-problemei zuzendua egon arren, optimizazio-problementzat ere erabiltzen da; hau da, optimizazio-problema bat P izango da (era berean, NP) dagokion erabaki-problema P bada (edo NP). Era berean, NP-osoa terminoa erabaki-problementzat erabiltzen denean; optimizazio-problemetan, aldiz, NP-zaila terminoa (*NP-hard*, ingelesez) erabiltzen da.

Intuizio gisa, problema bat NP-zaila dela esaten denean, hura ebazteko zailtasuna nabarmena dela adierazi nahi da. Jarraian, adibide gisa aipatuko ditugun problema guztiak, azpiklase honen parte dira.

[4] Problema hauek polinomikoak diren algoritmo *estokastikoak* erabiliz ebatz daitezke; beste era batean esanda, soluzioak denbora polinomikoan ebalua daitezke.

Problema klasikoak

Errealitatean ebatzi behar izaten diren optimizazio-problema guztiak ezberdinak dira, kasu bakoitzak bere murrizketa edota baldintza bereziak baititu. Diferentziak diferentzia, problema askoren mamia bertsua da; hala nola, garraio-problemak, esleipen-problemak, antolakuntza-problemak, etab. Hori dela eta, optimizazioan, askotan, problema teorikoak aztertzea izaten da ohikoena, hain zuzen errealitatean topa ditzakegun problemen abstrakzioak edota sinplifikazioak. Atal honetan optimizazio-problema teoriko klasiko batzuk aztertuko ditugu.

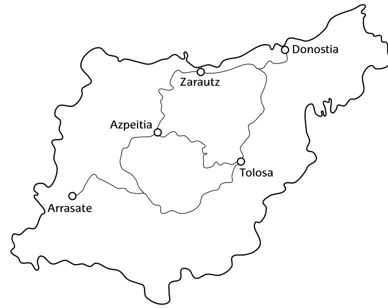
Garraio-problemak

Merkantzien edota pertsonen garraioarekin zerikusia duten optimizazio-pro blemak Ikerkuntza Operatiboan aztertu izan diren lehenetarikoak dira. Oinarrizko garraio-probleman, iturburu-puntuak eta helburu-puntuak ditugu; halaber, iturburu-puntu bakoitzetik helburu-puntu bakoitzera merkantzia garraiatzeko kostua ezaguna da, eta kostu-matrizean jasotzen da. Iturburu-puntu bakoitzaren eskaintza eta helburu-puntu bakoitzaren eskaera ezagunak dira. Problemaren helburua eskaera guztiak asetzeko kostu minimoko garraio-eskema topatzea da, iturburu-puntuak dituen mugak (eskaintzak) gainditu barik.

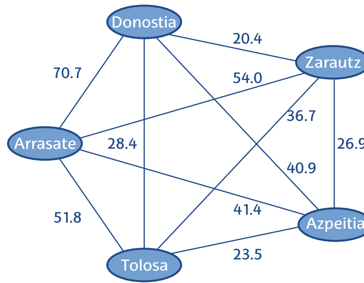
Problema sinple hau eredu linealen bidez modela daiteke, eta, hortaz, badaude algoritmo eraginkorrak ebazteko. Alabaina, badaude garraioari buruzko beste hainbat problema klasiko NP-zailak direnak. Ezagunena, *Travelling Salesman Problem* [16] da –hau da, Saltzaile Bidaiariaren Problema–.

1.5 adibidea. Saltzaile Bidaiariaren Problema (Travelling Salesman Problem, TSP) *Demagun saltzaile bidaiariak garela eta egunero hainbat bezero bisitatu behar ditugula. Bezero bakoitza herri batean bizi da, eta, edozein bi herriren arteko distantzia ezaguna izanda, gure helburua da herri guztietatik behin eta bakarrik behin pasatzen den ibilbiderik motzena topatzea.*

Problema honen hastapenak Irlandan daude, Hamilton matematikariaren lanetan. Problema formalizatzeko grafo oso bat eraiki dezakegu non erpinak herriak diren eta edozein bi erpinen artean pisu konkretu bateko ertz bat definitzen den; ertzen pisua, lotzen dituen bi herrien arteko distantzia da (ikusi 1.2 irudia). Horrela ikusita herri bakoitzetik bakarrik behin igaro nahi badugu, problemaren soluzioak ziklo hamiltoniarrak izango dira –hots, nodo guztiak behin eta bakarrik behin agertzen diren zikloak–. Ziklo hamiltoniar guztien artean pisu total minimoa duena bilatu nahi dugu.



(a) Mapa



(b) Dagokion grafoa

1.2 irudia. TSParen adibide bat, bost herrirekin

Arrasate, Zarautz, Tolosa, Azpeitia, Donostia ibilbidea, irudian agertzen den problemarentzako soluzio posible bat da; eta haren ebaluazioa ondorengoa izango da:

- Arrasate - Zarautz: 54.0
- Zarautz - Tolosa: 36.7
- Tolosa - Azpeitia: 23.5
- Azpeitia - Donostia: 40.9
- Donostia - Arrasate: 70.7

Hortaz, ibilbidearen distantzia totala 225.8 da. Ikus dezagun adibidea R-n. Lehenik eta behin, **metaheuR** paketea kargatu, eta problemaren helburu-funtzioa sortuko dugu:

```
> library("metaheuR")
> cost.matrix <- matrix(c(0,    20.4, 40.9, 28.4, 70.7,
+                         20.4, 0,    26.9, 36.7, 54,
+                         40.9, 26.9, 0,    23.5, 41.4,
+                         28.4, 36.7, 23.5, 0,    51.8,
+                         70.7, 54,   41.4, 51.8, 0), nrow=5)
> city.names <- c("Donostia", "Zarautz", "Azpeitia",
+                 "Tolosa", "Arrasate")
> colnames(cost.matrix) <- city.names
> rownames(cost.matrix) <- city.names
> cost.matrix

##          Donostia Zarautz Azpeitia Tolosa Arrasate
## Donostia      0.0    20.4     40.9   28.4     70.7
## Zarautz      20.4     0.0     26.9   36.7     54.0
## Azpeitia     40.9    26.9      0.0   23.5     41.4
## Tolosa       28.4    36.7     23.5    0.0     51.8
## Arrasate     70.7    54.0     41.4   51.8      0.0

> tsp.example <- tspProblem(cmatrix=cost.matrix)
```

Orain, lehen aipatutako soluzioa sortu eta ebaluatuko dugu.

```
> solution <- permutation(c(5, 2, 4, 3, 1))
> tsp.example$evaluate (solution)
```

```
## [1] 225.8
```

Hona hemen pentsatzeko galdera batzuk:

- Distantzia totala 225.8km-koa da, baina ibilbide hau distantzia minimokoa al da?
- Proba egin ezazu, adibidez, Azpeitia eta Tolosa trukaturik. Soluzio berri hori hobea ala okerragoa da?
- Zenbat ibilbide daude bost herri hauek behin eta bakarrik behin bisitatzen dituztenak?

1.1 ariketa. *Inplementa ezazu funtzio bat n tamainako TSP problema bat eta beraren ebaluazio-funtzioa emanda, soluzio guztiak ebaluatuz bide motzena topatzen duena.*

Ikus dezagun nola formaliza daitekeen TSP problema matematikoki:

1.3 definizioa. TSP problema - *Izan bedi* $H = h_1, \ldots, h_n$ *kokapen zerrenda eta* $C \in \mathbb{R}^{n \times n}$ *distantzia-matrizea, non* c_{ij} h_i *eta* h_j *kokapenen artean dagoen distantzia den. TSP problema ibilbide optimoa topatzean datza, hau da, kokapen guztietatik behin eta bakarrik behin igarotzen diren ibilbideetatik motzena.*

TSParen definizioan kostu-matrizea simetrikoa dela jotzen da. Hala ere, kasu errealetan, posible da norabide batean istripu bat egotea edo bidea noranzko batean eta bestean ezberdinak izatea. Egoera horietan bideen kostuak simetrikoak direla jotzea ez da problema modelatzeko aukerarik logikoena. Hortaz, bi TSP problema mota defini daitezke: simetrikoa eta asimetrikoa.

TSPa oso erabilia da algoritmoak probatzean, eta TSPLIB liburutegian [31] hainbat instantzia eskuragarri daude, orain arte lortutako soluziorik onenen informazioarekin batera.

Esleipen-problemak

Matematikako eta, batez ere, Ikerkuntza Operatiboko oinarrizko problemak dira. Esleipen-problemetan aldaera konbinatorio asko aurkitu ditzakegun arren, funtsean denak ondorengo ideian oinarritzen dira:

Demagun n agente ditugula, m ataza burutzeko. Agente bakoitzak kostu konkretu bat du ataza bakoitza burutzeko, eta ataza bakoitza betetzeaz agente bakar bat arduratu behar da. Esleipen-problemaren helburua ataza bakoitzari agente bat esleitzean datza, ataza guztiak burutzeko kostu totala minimizatuz.

Problema honen kasu nabariena Esleipen Problema Lineala da –*Linear Assignment Problem*, ingelesez–, non agente eta ataza kopurua berdina den, eta esleipen kostu totala eta agente bakoitzaren kostuen batura totala ere berdinak diren. 1955ean. Harold Kuhn-ek Algoritmo Hungariarra proposatu zuen, zeinak Esleipen Problema Lienala denbora polinomialean ($O(n^4)$) ebazten duena, n agente kopurua izanik. Badago, ordea, esleipen-problema mota bat, NP-zaila dena eta liburuan adibide gisa erabiliko duguna: Esleipen Problema Koadratikoa –*Quadratic Assignment Problem*, QAP, [8] ingelesez–.

1.4 definizioa. QAP Problema *Izan bitez n lantegi, n kokapen posible, $H \in \mathbb{R}^{n \times n}$ lantegien arteko fluxuen matrizea, eta $D \in \mathbb{R}^{n \times n}$ kokapenen arteko distantzien matrizea. QAP problemaren helburua lantegi bakoitza kokapen batean finkatzean datza, kostu totala minimizatuz.*

Antolakuntza-problemak

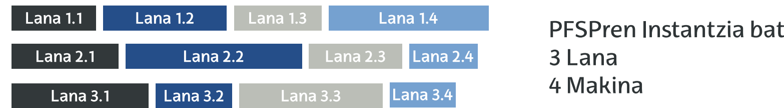
Prozesu, lan edota atazen antolakuntza eta zerbitzatzearekin zerikusia duten optimizazio-problemak dira. Antolakuntza-problemen helburua kudeatzaileak jasotzen dituen lan-eskariak modurik eraginkorrenean zerbitzatzea da. Eraginkortasunaren neurria problemaren araberakoa da; hala ere, eskariak ahalik eta denbora/kostu txikienean asetzea da irizpide ohikoena.

Hasiera batean, antolakuntza-problema gehientsuenak industriarekin zerikusia zuten eremuetan proposatu ziren. Produktuaren ekoizpenerako makineria erabiltzen zen enpresetan, etekina maximizatzea zen helburua, makineriaren eta baliabideen erabilera, mantentze-kostuak, eta abar optimizatuz. Aldi berean, langileen ordutegien planifikazioan antzerako ezaugarriak zituzten problemak ere proposatu ziren.

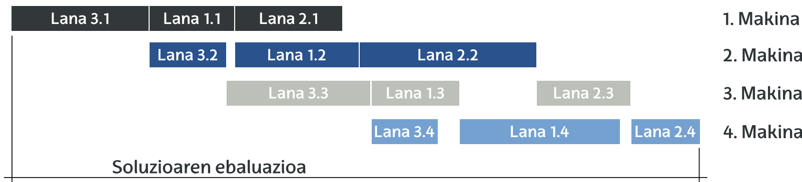
Gaur egun, ordea, problema horiek edozein arlotara daude hedatuta. Esate baterako, konputazioan, konputagailuen Prozesurako Unitate Zentralak (PUZ) *scheduler* eta *dispatcher* izeneko tresnak erabiltzen ditu, jasotzen dituen prozesuak erarik eraginkorrenean zerbitzatzeko, denboraren eta memoriaren erabilera minimizatuz.

Adibide gisa, jarraian, antolakuntza-problemetan oso ezaguna den muntaketa-kateko plangintza-problema –*Permutation Flowshop Scheduling Problem*, PFSP [17] ingelesez– aztertuko dugu:

1.5 definizioa. PFSP problema. *Izan bitez n lan, m makina eta $P \in \mathbb{R}^{n \times m}$ prozesatze-denboren matrizea, non p_{ij} i lanak j makinan prozesatzeko behar duen denbora adierazten duen. Lan bakoitza burutzeko m prozesu aplikatu behar dira, bakoitza makina batean; hau da, j-garren operazioa, j-garren makinan egingo da. Behin i lana j makinan sartzen denean prozesatzeko, eten barik prozesatuko da, eta denbora konkretu bat emango du bertan, p_{ij}. i lana j makinatik irten denean, $j+1$ makinara pasatuko da hurrengo prozesua burutzera, baldin eta makina hori libre badago. PFSParen gakoa da lanak*



Problemarako soluzio bat: 3. lana, 1. lana, 2. lana



1.3 irudia. PFSP problemaren adibide bat. Goiko partean problemaren definizioa dago, hau da, lan bakoitza burutzeko makina bakoitzean behar den denbora. Beheko partean, soluzio bat eta beraren interpretazioa erakusten dira. Helburua denbora totala minimizatzea bada, 3.1 lana hasten denetik 2.4 lana amaitzen den arte igarotzen den denbora da soluzioaren ebaluazioa

prozesatzeko denbora totala minimizatzen duen n lanen sekuentzia optimoa aurkitzea.

1.3 irudian problemaren instantzia bat ikus daiteke. Adibide honetan 3 lan burutu behar dira, 4 makinatan. Irudiaren goiko partean lan bakoitzak makina bakoitzean igaro behar duen denbora adierazten da, lauki zuzenen bidez. Irudiaren beheko partean soluzio bat proposatzen da; soluzio horretan, lanak 3,1,2 ordenan prozesatuko dira. Horrek esan nahi du 3. lana 1. makinan sartuko dela. Behar duen denbora pasatzen denean, 2. makinan sartuko da, eta 1. makinan 1. lana sartuko da. Prozesu guztiaren eskema irudian ikus daiteke. Problemaren helburua denbora totala minimizatzea bada, soluzio horren helburu-funtzioaren balioa 3.1 lana hasten denetik 2.4 lana amaitzen den arte igarotzen den denbora izango da.

PFSP antolakuntza-problemen murriztapenik gabeko problema teorikoa da. Problema errealetan, lan kopurua ez da finitua, etengabekoa baizik; kasu horietan, makinen okupazio maila altua izatea denbora murriztea bezain garrantzitsua izaten da. Zentzu horretan, antzeko problemen aukera oso zabala da [34].

Azpimultzo-problemak

Demagun objektu multzo bat dugula, eta multzo horretatik objektu batzuk aukeratu behar ditugula. Aukeraketa ez da ausaz egingo, baizik eta irizpide eta murriztapen batzuei jarraituz. Azpimultzo-problemetan, helburua auke-

raketak ematen dizkigun onurak maximizatzean datza, definitutako murriztapenak betez.

Azpimultzo-problemen hedapena oso zabala da, logistikako alorretan, gehienbat: kargarako garraiobideen betetzea, aurrekontuaren kontrola, finantzen kudeaketa, industriako materialen ebaketa, besteak beste.

Azpimultzo-problemen artean adibide erakusgarriena – eta bidenabar sinplifikagarriena – ingelesez *0-1 Knapsack problem* deritzon 0-1 Motxilaren problema [21] da. Jarraian zehazki azalduko dugu problema hori:

1.6 definizioa. 0-1 motxilaren problema. *Izan bitez n objektu, c motxilaren edukiera maximoa, eta $P \in \mathbb{R}^n$ eta $W \in \mathbb{R}^n$ objektuen balio- eta pisu-bektoreak hurrenez hurren. Problema honetan, n objektuen artetik aukeratzen ditugun elementuen balio totala maximizatzea dugu helburu, motxilaren edukiera maximoa gainditu gabe betiere.*

Grafoei buruzko problemak

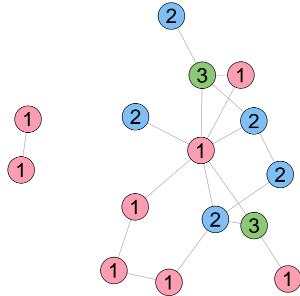
Grafoak matematikan eta informatikan funtsezko egiturak dira. Oro har grafoak objektuen arteko erlazioak adierazteko erabiltzen dira eta, beraz, haien erabilera edozein eremutan da aplikagarria. Hori dela eta, grafoei loturiko problemen multzoa oso zabala da. Esaterako, berriki ikusi ditugun TSP eta QAPak grafo-problema gisa formaliza ditzakegu. Garraiobide- eta esleipen-problemez gain, sareko informazio-trukaketa edota grafoen teoriako problema ugari (deskonposizio-problemak, azpimultzo-problemak, estaldura-problemak, etc.) ebazteko erabili ohi dira.

Atal honetan, ingelesez *Graph Coloring* deritzon grafoen koloreztatze-problema izango dugu aztergai.

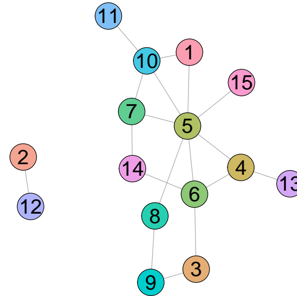
1.7 definizioa. Grafoen koloreztatze-problema. *Izan bedi $G = (V, E)$ grafoa non V eta E bektoreak grafoaren erpin eta ertz multzoak diren, hurrenez hurren; $e_{ij} \in E$ existitzen bada, v_i eta v_j erpinen artean ertza dago. Erpin bakoitzari kolore bat esleitu behar diogu, kontuan hartuz $e_{ij} \in E$ existitzen bada v_i eta v_j nodoek ezin dutela kolore bera izan; hau da, ertz baten bidez lotutako erpinak kolore ezberdinekin koloreztatu behar dira. Problemaren helburua kolore kopuru minimoa erabiltzen duen koloreztatzea topatzean datza.*

Ikus dezagun adibide pare bat, ausaz sortutako grafo bat erabiliz. Jarraian dagoen `R` kodean 15 erpin dituen ausazko grafo bat sortu ondoren, koloreztatze-problema bat sortuko dugu, **metaheuR** paketean dauden funtzioak erabiliz.

```
> library("igraph")
> set.seed(1623)
> n <- 15
> rnd.graph <- random.graph.game(n, p.or.m=0.2)
> gcol.problem <- graphColoringProblem(rnd.graph)
```



(a) Soluzio bideraezina



(b) Soluzio bideragarria

1.4 irudia. Grafoen koloreztatze-problemaren bi adibide. Lehenengoa, (a), bideraezina da, elkar ondoan dauden nodo batzuek kolore berdina baitute. Bigarrena, (b), soluzio bideragarri tribiala da, nodo bakoitzak kolore ezberdin bat baitu.

Ausazko soluzio bat sortzen dugu, bakarrik 3 kolore erabiliz. Kontuan hartu behar da soluzioak `factor` motako bektore bat izan behar duela, non balio posibleen kopuruak nodo kopuruaren berdina izan behar duen (kasurik txarrenean, grafo osoa denean, nodo bakoitzak kolore ezberdin bat izan beharko du). Ausazko soluzioa ea bideragarria denetz egiaztatuko dugu, problema definitzean `valid` sortutako funtzioa erabiliz.

```
> l <- paste("c", 1:n, sep="")
> rnd.sol <- factor(sample(l[1:3], size=n, replace=TRUE), levels=l)
> rnd.sol

##  [1] c1 c1 c1 c3 c1 c2 c2 c1 c1 c3 c2 c1 c1 c2 c2
## Levels: c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15

> gcol.problem$valid(rnd.sol)

## [1] FALSE

> gcol.problem$plot(rnd.sol, node.size=20, label.cex=2)
```

Soluzioa bideraezina da, 1.4 (a) irudian ikus daitekeen legez. Soluzio bideragarri bat sortzeko era sinple bat badago: nodo bakoitzari kolore ezberdin bat esleitu (ikusi 1.4 (b) irudia).

```
> trivial.sol <- factor (l, levels=l)
> trivial.sol

##  [1] c1  c2  c3  c4  c5  c6  c7  c8  c9  c10 c11 c12 c13 c14 c15
## Levels: c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15
```

```
> gcol.problem$valid(trivial.sol)

## [1] TRUE

> gcol.problem$plot(trivial.sol,  node.size=20, label.cex=2)
```

Karaktere-kateko problemak

Optimizazio konbinatorioko karaktere-kateen arteko erlazioak aurkitzeaz arduratzen diren problemen multzoa da. Problema ezagunenetariko bat azpisekuentzia komun luzeenaren problema da –*Longest Common Subsequence Problem*, LCSP– da. Izenak berak adierazten duen bezala, ditugun karaktere-kateen edo sekuentzien artean komuna den azpisekuentzia luzeena bilatzen duen problema da.[5]

> **1.6 adibidea.** *Demagun hiru DNA sekuentzia ditugula; CACGACGCGT, CGTTTCGCAG eta CTTGCGCGA. Hiru sekuentzietan dagoen azpisekuentzia komun luzeena CGCGCG da; hau da, caCGaCGCGt, CGtttCGCaG eta CttGCGCGa dauzkagu. Beste edozein letra sartuz gero, azpisekuentzia ez da sarrerako hiru sekuentzietan agertuko.*

LCSPa formalizatzeko lehendabizi azpisekuentzia kontzeptua definitu behar dugu.

1.8 definizioa. *Izan bedi Σ alfabetoan definitutako n tamainako sekuentzia $S = (s_1, \ldots, s_n)$, hau da, $\forall i = 1, \ldots, n$, $s_i \in \Sigma$. $S' = (s_{a_1}, \ldots, s_{a_m})$ S-ren azpisekuentzia da, baldin eta soilik baldin $a_i \in \{1, \ldots, n\}$ eta $\forall j = 2 \ldots, m$, $a_{i-1} < a_i$*

Hau da, azpisekuentzia batean jatorrizko sekuentzian dauden elementuen azpimultzo bat izango dugu, *jatorrizko sekuentzian agertzen diren ordena berdinarekin*. Sekuentzia bat emanda, bere zenbait elementu ezabatuz lor ditzakegu azpisekuentziak.

Bi sekuentzia besterik ez badugu –m eta n tamainakoak– programazio dinamikoa erabiliz[6] $O(mn)$ konplexutasunarekin ebatz daiteke LCSP-a; sekuentzia kopurua finkaturik gabe badago, ordea, problema NP-zaila da.

---

[5] Kontutan hartu azpisekuentzia eta karaktere azpi-kate kontzeptuak ezberdinak direla, bigarrenean hautatutako elementuak jarraian egon behar baitira jatorrizko sekuentzian baina lehenengoan ez. Hau da, $CTTTCGTCATA$ sekuentzia badugu, $TCGTCA$ bai azpisekuentzia eta baita azpi-katea ere bada, baina GTA azpisekuentzia izan arren ez da azpi-katea

[6] Problema hau sekuentziak lerrokatzean agertzen da; kasu horretan, nahiz eta algoritmo polinomikoa izan, sekuentziak milioika elementu izan dezakeenez, programazio dinamikoaz gain metodo heuristikoak erabiltzen dira; metodorik ezagunena ([2]) izenekoa da

1.9 definizioa. Longest Common Subsequence Problem (LCSP) *Izan bitez Σ alfabetoan definitutako tamaina ezberdineko k sekuentzia $S_1, \ldots, S_k$. Izan bedi k sekuentzien azpisekuentzia diren sekuentzia multzoa $\mathcal{C}$. LCSPren helburua $C^* \in \mathcal{C}$ sekuentzia topatzea da, non $\forall C_i \in \mathcal{C}$ $|C_i| > |C^*|$.*

LCSP bezalako problemak ohikoak izaten dira terminaleko komandoetan, adibidez, `diff` edo `grep` komandoetan.

Azken hamarkadetan, ordea, Bioinformatika arloak entzute handia lortu duela-eta, karaktere-kateko problema ugari proposatu dira. Horietako bat, sekuentzia-zati muntaketa-problema –*Fragment Assembly Problem*, FAP, ingelesez– da. DNA-ren sekuentziaziorako teknologia ez dago hain aurreratua, eta gaur egun oraindik ezinezkoa da genomen karaktere-kateak osorik irakurtzea. Hori dela eta, DNA zatitxoak irakurtzea ahalbidetzen duten bestelako teknikak erabiltzen dira.

Testuinguru horretan, sekuentzia-zati mutaketa-problemaren helburua, azpisekuentzietatik abiatuta, sekuentzia bakar bat osatzea da. LCSPn antzera, azpisekuentzia komunak modu eraginkorrean detektatzea ezinbestekoa da, prozesuaren bukaeran DNA-sekuentzia fidagarri bat lortzeko.

Optimizazio-problemak ebazten

Ikerkuntza Operatiboaren hasierako urteetan hainbat problemaren soluzio zehatzak topatzeko algoritmoak proposatu ziren. Adibiderik ezagunena 1947an proposatutako Simplex algoritmoa da.

Hurbilketa hori –algoritmo zehatzak erabiltzea, alegia– problema sinpleentzat egokia izan arren, problemaren konplexutasuna edota tamaina handitzen denean bideraezin bihur daiteke. Konplexutasunak algoritmoa aplikatzeko behar dugun denboraren hazkunde-abiadura adierazten du; ordena handiagoa edo txikiagoa izan daiteke, baina abiadura beti positiboa da;

hau da, zenbat eta problema handiagoa orduan eta denbora gehiago beharko dugu problema ebazteko. Hori kontuan hartuz, denbora maximoa finkatzen badugu, beti problema-tamaina maximo bat izango dugu; problema handiagoa bada, finkatutako denboran ebazterik ez da egongo.

Demagun 1.1 irudiak problema baten soluzio zehatza lortzeko zenbait algoritmok behar duten denbora adierazten duela; era berean, demagun soluzioa lortzeko ordu bat besterik ez dugula. Grafikoan agerian dago algoritmo hiperesponentzialarekin $n > 7$ tamainako problemak, ordu batean, ebaztezinak direla; algoritmo esponentzialarekin, ostera, hogei tamainatako problemak ebazteko gai izango ginateke. Algoritmoak polinomikoak izateak ez du esan nahi algoritmoak edozein tamainatako problema ebatz ditzatekeenik. Hori argi eta garbi ikusten da $O(n^{15})$ kasuan, non ordu bateko mugarekin tamaina maximoa $n = 2$ den. Beste algoritmo polinomikoentzat ere denbora-kurbek, nahiz eta oso motel, gora egiten dute, eta, beraz, nonbait ordu bateko muga gurutzatuko dute.

TSP problemarako heuristikoa

```
1  input: n × n tamainako C kostu-matrizea (herrien arteko distantziak)
2  input: i_1, ibilbideko lehendabiziko herria
3  output: I = (i_1, ..., i_n) ibilbidea
4  Sartu H multzoan herri guztiak, i_1 izan ezik
5  k = 2
6  while H ≠ ∅ do
7     i_k = arg min_j {c_{i_{k-1},j} | j ∈ H}
8     Kendu i_k H multzotik
9  done
```

1.1 algoritmoa. TSPrako soluzio onak eraikitzeko metodo heuristikoa

Kasu horietan teknika klasikoak baliogabeak direnez, beste aukeraren bat bilatu beharko dugu; soluzio zehatza lortzerik ez badago, daukagun baliabideekin eta denborarekin *albait soluziorik onena* topatzeko algoritmoak diseinatu behar ditugu. Xede hori lortzeko algoritmo *heuristikoak* «intuizioan» oinarritutako metodoak erabiltzea da ohikoena. Algoritmo horien bidea optimo globala topatzea bermatuta ez egon arren, oro har soluzio onak topa ditzakegu.

Literaturan proposaturiko lehendabiziko metodoak problemaren intuizioan oinarritzen ziren. Ikus dezagun adibide bat, 1.2.2.1 atalean deskribaturiko TSP problema ebazteko.

Demagun badakigula abiapuntuko herria zein den, hau da, zein den ibilbideko lehendabiziko herria; soluzioa pausoz pauso eraikiko dugu, urrats bakoitzean aurreko urratsean aukeratu dugun herritik gertuen dagoen herria aukeratuz. Metodoaren sasikodea 1.1 algoritmoan ikus daiteke.

Ebatz dezagun, algoritmo hau erabiliz, 1.2 irudian dagoen TSParen instantzia, Arrasatetik abiatuz. Arrasatetik gertuen dagoen herria Azpeitia denez, horixe izango da gure ibilbideko bigarren herria. Ondoren, Azpeititik Tolosara joango gara, hura baita gertuen dagoena eta handik Donostiara. Bisitatu barik dauden herrietatik Zarautz da Donostiatik gertuen dagoena –eta, berez, bakarra–. Ibilbidea ixteko Zarautzetik Arrasatera itzuli beharko dugu. Laburbilduz, heuristiko sinple hau erabiliz ondoko soluzioa daukagu: Arrasate, Azpeitia, Tolosa, Donostia, Zarautz, Arrasate; soluzio horren helburu-funtzioa 41,4 + 23,5 + 28,4 + 20,4 + 54,0 = 167,7 da. Ez dugu inolaz bermaturik soluzio hori optimoa izatea, baina soluzio ona da, eta, batez ere, denbora koadratikoan lortu dugu.

Algoritmo hori **metaheuR** paketean inplementaturik dago, baina bi diferentzia ditu. Alde batetik, erabiltzaileak lehendabiziko hiria sartu beharrean, algoritmoak aukeratzen du, matrizean dagoen distantziarik txikienari erreparatuz. Bestetik, funtzioak ez du beti aukerarik onena aukeratzen: aukera

batzuen artean bat ausaz aukeratzen du. Azken puntu horrek gero ikusiko dugun algoritmo batekin (GRASP) zerikusia du.

Hemen funtzio horren bertsio sinplifikatu bat inplementatuko dugu, adibide gisa. Funtzioak parametro bakar bat jasotzen du, `cmatrix`, kostu-matrizea dena; hasteko, aukeraezinak diren balioak adierazteko `NA`-k (*not available*) txertatuko ditugu matrizean. Kontuan hartuz diagonalean dauden balio guztiak ezin direla aukeratu (ez du zentzurik hiri batetik hiri berberara joateak), aukeraezin gisa finkatzen ditugu eta, gero, matrizean dagoen elementurik txikiena(k) aukeratuko d(it)ugu.

```
tsp.constructive <- function(cmatrix){
  diag(cmatrix) <- NA
  best.pair <- which(cmatrix == min(cmatrix, na.rm=TRUE),
                     arr.ind=TRUE)
```

Orain `best.pair` aldagaian matrizearen errenkada bakoitzean elementu baten koordenatuak izango ditugu: lehenengo zutabean bere errenkada eta bigarrenean bere zutabea. Aintzat hartzekoa da elementu bat baino gehiago izan ditzakegula (izan ere, matrizea simetrikoa bada, beti izango ditugu, gutxienez, bi elementu). Lehenengo elementua bakarrik hautatuko dugu, eta horrek markatuko ditu ibilbideko lehendabiziko bi hiriak. Algoritmoarekin jarraitzeko jakin behar dugu sartu dugun lehenengo hiritik (`best.pair[1]`) ezin garela berriz igaro. Hori dela eta, matrizean dagokion errenkada aukeraezin moduan markatu behar dugu. Era berean, hurrengo urratsetan ezin dugu aukeratu sartu ditugun hirietan amaitzen den elementurik; hots, hiri horiei dagozkien zutabeak ere bideraezin gisa finkatu beharko ditugu.

```
  solution <- c(best.pair[1], best.pair[2])
  cmatrix[best.pair[1], ] <- NA
  cmatrix[, best.pair] <- NA
```

Gure soluzioak, momentuz, bakarrik bi hiri ditu. Soluzio osoa eraikitzeko, urrats bakoitzean, azken pausoan sartutako hiritik aukeratu gabe dauden hirien artean gertuen dagoena aukeratu beharko dugu. Behin aukeraturik, matrizea eguneratu beharko dugu aukeraezin diren elementuei `NA` esleituz.

```
  for(i in 3:nrow(cmatrix)){
    next.city <- which.min(cmatrix[solution[i-1], ])
    solution <- append(solution, next.city)
    cmatrix[solution[i-1], ] <- NA
    cmatrix[,next.city] <- NA
  }
```

Une honetan `solution` bektoreak eraikitako soluzio bat gordetzen du. Dena dela, soluzioa hirien permutazio baten bidez kodetu nahi dugunez, funtzioaren amaieran ondoko kodea izango dugu.

```
  names(solution) <- NULL
  return(permutation(vector=solution))
}
```

Inplementatutako funtzioa gure problemari aplikatzen badiogu, hona hemen emaitza:

```
> greedy.solution <- tsp.constructive(cost.matrix)
> tsp.example$evaluate(greedy.solution)

## [1] 167.7

> colnames(cost.matrix)[as.numeric(greedy.solution)]

## [1] "Zarautz"  "Donostia" "Tolosa"   "Azpeitia" "Arrasate"
```

Aurreko adibidearekin alderatuta, lortzen dugun soluzioa ezberdina da, baina beraren kostua, ordea, berdina. Izan ere, nahiz eta soluzioa ezberdina izan, definitzen duen zikloa berdina da, beste noranzkoan izanda ere. Beste era batean esanda, goiko kodean dugun soluzioa atzetik aurrera irakurtzen badugu, lehen bilatutako soluzio berbera dugu!

Metodo heuristikoak oso interesgarriak dira, baina zailak «birziklatzeko» –pentsa ezazu nola egoki daitekeen goiko algoritmoa grafoen koloreztatze-problema ebazteko, adibidez–. Eragozpen horri aurre egiteko, *bilaketa heuristikoak* edo *metaheuristikoak* proposatu ziren. Metodo horiek ere intuizioan oinarritzen dira, baina ez problemaren intuizioan, optimizazio-prozeduraren intuizioan baizik; hori dela eta, metodo hauek edozein problema ebazteko egoki daitezke. Hainbat metaheuristika existitzen dira, hala nola, bilaketa lokala, algoritmo genetikoak edo inurri-kolonia algoritmoak esaterako. Horiek guztiak hurrengo kapituluen izango ditugu aztergai.

Optimizazio-problema baten aurrez aurre gaudenean, kontuan izan beharreko hainbat gauza daude: alde batetik, problemaren formalizazio berean agertzen diren elementuak –helburu-funtzioa eta soluzioen espazioa, alegia– eta, bestaldetik, problema ebazteko erabil daitezkeen algoritmoak. Hurrengo ataletan alderdi horiek guztiak banan-banan komentatuko ditugu.

Helburu-funtzioa

Aurreko atalean ikusi dugu optimizazio problema bat definitzeko bi elementu behar ditugula, horietako bat helburu funtzioa izanik. Helburu funtzioa soluzioen optimotasuna ebaluatzeko erabiliko dugu eta, hortaz, funtzio horrek optimo globala zein den zehaztuko du.

Optimizazio-problema bat formalizatu behar dugunean argi izan behar dugu soluzioak nola ebaluatuko diren. TSPan, adibidez, distantzia edo kostua nahi dugu minimizatu; hori dela eta, ibilbide bat emanda helburu funtzioak horren distantzia edo kostu totala neurtuko du. Adibide honetan darabilgun funtzioa tribiala da eta zuzenean aplika daiteke; hori ordea, ez da beti horrela izaten. Zenbait kasutan soluzioen ebaluazioa konplexua izan daiteke. Hona hemen adibide batzuk:

- **Helburu-funtzioa simulazio-prozesu bat denean**. Adibidez, ekuazio diferentzial sistema bat daukagunean eta euren parametroak optimizatu nahi ditugunean, parametro sorta bakoitza ebaluatzeak sistema ebaztea inplikatzen du.
- **Optimizazio interaktiboan**([35]). Problema batzuetan ezin da formula matematiko bat sortu, eta soluzioak ebaluatzeko erabiltzailearen elkarrekintza behar da –erakargarritasuna, zaporea eta horrelakorik aztertu behar denean, besteak beste–.
- **Soluzioa ebaluatzeko algoritmo bat aplikatu behar denean**. Eredu grafiko probabilistikoetan (esate baterako, grafo bat eraikitzeko), aldagaiak ordenatuta badaude, algoritmo deterministak erabil daitezke. Kasu horietan optimizazio-problema aldagaien ordena optimoa topatzean datza; alabaina, ordenarekin bakarrik ezin dugu soluzioa ebaluatu, grafo osoa behar baitugu. Hortaz, soluzioak ebaluatu ahal izateko, algoritmo deterministaren bat aplikatu beharko dugu ordena ezagututa grafoa sortzeko.
- **Programazio genetikoan**. Programazio genetikoan soluzioak atazaren bat burutzeko programa-diseinuak dira. Hori dela eta, soluzioak ebaluatzeko horiek *exekutatu* egin behar dira, ea espero dena egiten duten egiaztatzeko.

Bilaketa-espazioa: soluzioen kodeketa

Problemak formalizatzeko soluzioak nola kodetuko ditugun erabakitzea ezinbestekoa da; izan ere, helburu-funtzioa ezin da zehaztu pauso hau burutu arte.

Kodeketa on bat diseinatzeko zenbait alderdi aztertu behar ditugu. Lehendabizikoa *osotasuna* da, hau da, edozein soluzio adierazteko gaitasuna. Edozein bi soluzio hartuta, batetik bestera joateko bidea badagoela ziurtatzea ere garrantzitsua da, *konexutasuna* izan ezean bilaketa-espazioko eremu batzuk helezinak gerta daitezkeelako.[7] Amaitzeko, bilaketa-prozesuan soluzioak manipulatzeko hainbat funtzio edo operadore erabiliko ditugu; beraz, darabilgun soluzioen kodeketak *operadoreekiko eraginkorra* izan behar du.

Literaturan hainbat kodeketa *estandar* topa ditzakegu. Ikus ditzagun horietako batzuk, 1.2. atalean deskribatutako problemen soluzioak adierazteko erabil daitezkeenak.

TSP problemarako soluzioak herrien zikloak dira; hau da, herri bakoitza behin eta bakarrik behin agertzen diren zerrendak. Esklusibotasun hori dela eta, *permutazioak* TSParen soluzioak kodetzeko adierazpide oso egokiak dira. Soluzio guztiak kodetu daitezke permutazioen bidez, eta permutazio guztiek

[7] Gaitasun hau bermatzeko soluzioen kodetzea ez ezik, soluzioak maneiatzeko darabiltzagun operadoreak eta murrizketak kudeatzeko estrategiak ere aintzat hartu behar ditugu.

soluzio bideragarriak kodetzen dituzte.[8] Adierazpide berdina beste hainbat problemetan erabil daiteke, hala nola *scheduling* problema batzuetan, beste *routing* problemetan, ordenazio-problemetan, ...

Azpimultzo problemetan (ikus. 1.2.2.4 atala), baldintza edo murrizketa batzuk betetzen dituen azpimultzo optimoa topatzea da helburua. Multzoekin dihardugunean *bektore bitarrak* aukera egokiak dira oso. Motxilaren probleman, esate baterako, n elementu baldin baditugu edozein soluzio n tamainako bektore bitar baten bidez adieraz dezakegu, non i. posizioak i elementua motxilan dagoenetz adieraziko duen. Edozein soluzio n tamainako bektore bitar baten bidez kodetu daiteke; alderantzizkoa, ostera, ez da beti beteko, bektore bitarrek motxilaren kapazitatea gainditzen duten soluzioak adierazi ahal baitituzte.

Bektore bitarren kontzeptua aldagai kategorikoetara heda daiteke; hau da, soluzioak *bektore kategoriko* baten bidez adieraz ditzakegu. Kodeketa hori esleipen-problema orokorrean –*Generalized Assignment Problem*, GAP [33], ingelesez– erabili ohi da.

1.10 definizioa. GAP problema *Izan bitez n ataza, m agente, $C \in \mathbb{R}^{m \times n}$ kostu-matrizea eta $P \in \mathbb{R}^{m \times n}$ etekin-matrizea; c_{ij} eta p_{ij} elementuek j ataza i agenteari esleitzeari dagokion kostua eta lortutako etekina adierazten dituzte, hurrenez hurren. i agentearen lan-karga gorenekoa l_i bada eta ataza bakoitza agente bakar batek egin dezakeela kontuan hartuz, GAP problemaren helburua esleipen-kostu totala minimizatzen duen esleipena topatzean datza, agenteen lan-karga maximoa gainditu gabe, betiere.*

GAP problemarako soluzioak adierazteko n tamainako bektore kategorikoak erabil daitezke; posizio bakoitzean dauden balioak $\{1, \ldots, m\}$ tartean egongo dira. Bektorearen i. posizioak i. ataza zer agenteak egingo duen adieraziko du eta; kodeketa horren bidez soluzio-kode erlazioa bijektiboa da; hau da, soluzio bakoitzeko kode bakarra dago, eta kode bakoitzak soluzio bakarra kodetzen du.

Bektoreetan oinarritutako kodeketarekikoak amaitzeko, ideia zenbaki errealetara ere hedatu daiteke; hau da, zenbait problematan soluzioak bektore errealen bidez kodetu daitezke. Simulazio edo bestelako prozesuen parametroen optimizazioa soluzio-kodeketa horren bitartez egin daiteke, parametroak jarraituak badira betiere.

Orain arte ikusi ditugun adierazpideak *linealak* deritzenak dira, soluzioak bektore baten bidez kodetzen baitira. Adierazpide horiek maneiatzeko errazak izan arren, ez dira hainbat soluzio mota kodetzeko gai: adibidez programazio genetikoko soluzioak. Kasu horietan, oso hedatuta dauden adierazpideak erabiltzen dira: grafoak eta, bereziki, zuhaitzak. Kodetze mota ezberdinen

[8] Berez arazotxo bat badago. Ibilbide bat emanda, norabide batean edo bestean egiteak ez du inongo eraginik helburu-funtzioan –problema simetrikoa bada betiere– eta, hortaz, ziklo bakoitzeko bi permutazio izango ditugu zeinentzat helburu-funtzioa berdina den.

Motxilaren problemarako konponketa-algoritmoa

```
1  input: bideraezina den s soluzioa
2  input: s' soluzio bideragarria
3  s' = s
4  while s' bideraezina den do
5      Kendu motxilatik erabilgarritasuna zati pisua ratioa (u_i/w_i) minimizatzen duen e_i
       elementua
6      s' = s \ e_i
7  done
```

1.2 algoritmoa. Motxilaren problemarako bideragarriak ez diren soluzioak konpontzeko prozedura bat

konbinaketa ere asko erabiltzen den beste estrategia bat da; lehen aipaturiko parametro optimizazioan, esate baterako, parametro jarraituak eta diskretuak ditugunean balio errealak eta diskretuak dituzten bektoreak erabili genitzake. Edonola ere, kodeketa bat diseinatzean atalaren hasieran aipaturiko ezaugarriak aintzat hartu beharko ditugu.

Adierazpideen eta soluzioen artean dagoen erlazioari erreparatuz, hiru aukera ditugu:

- **Kode bat soluzio bakoitzeko**. Aukerarik ohikoena da, soluzio bakoitzeko kode bat daukagu, eta kode bakoitzak soluzio bakarra adierazten du.
- **Kode anitz soluzio bakoitzeko**. Kasu honetan «erredundantzia» daukagu, eta horrek bilaketaren eraginkortasuna kaltetu dezake. Nahikoa ez eta, bilaketa-espazioa behar baino handiagoa da.
- **Soluzio anitz kode berdinarekin**. Kodeketa honekin bereizmen murriztua daukagu –soluzioen xehetasuna gal dezakegu, alegia–. Bestalde, bilaketa-espazioa txikiagoa da eta horrek bilaketari lagun diezaioke. Adierazpide mota honetan deskodeketa-prozesu bat egon ohi denez, zeharkodetzea deritzogu.

Bilaketa-espazioa: murrizketak

Askotan, bideragarritasunaren definizioak hainbat murrizketa dakartza, eta, hortaz, murrizketak nola kudeatu erabaki beharko dugu; hori lortzeko hainbat aukera ditugu:

- **Soluzioen kodeketaren edota operadoreen bidez bideragarritasuna mantendu** - Problema ebazteko soluzioen kodeketa diseinatu beharko dugu. Posible denean, kodeketa horrek murrizketak integratuko ditu; alegia, sor daitezkeen kode guztiek soluzio bideragarriak adieraziko dituzte.

Soluzioen kodeketa ez ezik, soluzioak maneiatzeko darabiltzagun funtzio matematikoak ere bideragarritasuna mantentzeko erabil daitezke. Estrategia hau zenbait problematan –TSPn, besteak beste– murrizketak beteko direla ziurtatzeko bide zuzena da.

- **Soluzio bideraezinak baztertzea** - Estrategiarik sinpleena da; soluzio bat bideragarria izan ezean, baztertu egiten da bilaketa-prozesuan. Sinplea izan arren, eragin handia izan dezake bilaketa-prozesuan, espazioko zenbait eskualde «isolaturik» gera baitaitezke.
- **Soluzio bideraezinak zigortu** - Gerta daiteke soluzio bideragarrien espazioa etena izatea; hau da, soluzio bideragarri batetik bestera joateko soluzio bideraezinetatik pasatzea ezinbestekoa izatea. Gauzak horrela, soluzio bideraezinak baztertzeak edo konpontzeak ez du oso irtenbide egokia ematen. Soluzio bideraezinak erabil daitezke bilaketa-prozesuan, helburu-funtzioan zigortze-termino bat sartuz.

$$f'(s) = f(s) + \alpha c(s)$$

  $f'(s)$ zigorra duen helburu-funtzio berria da, eta $f(s)$, berriz, helburu-funtzio «kanonikoa». c funtzioak soluzioaren kostua –hau da, bideragarritasun eza– neurtzen du. Kostua neurtzeko hainbat aukera daude; hala nola, betetzen ez diren murrizketa kopurua, konponketaren kostua, etab. α parametroa zigorra kontrolatzeko erabil daiteke; zigortze-maila txikiegia bada, bilaketak topatzen dituen soluzioak bideraezinak izan daitezke; handiegia bada, berriz, soluzio bideraezinak baztertzeak dituen arazoak errepika daitezke. Hori dela eta, parametro hau estatikoa izan beharrean, dinamikoki alda dezakegu.[9]
- **Soluzio bideraezinak konpondu** - Bilaketa-prozesuan soluzio bideraezin bat topatzean, posible bada betiere, soluzioa «konpondu» egin dezakegu. Estrategia hori erabilgarria izan dadin, erabiltzen diren konponketa-algoritmoek eraginkorrak izan behar dute, bilaketaren kostu konputazionalean albait eragin gutxien izan dezaten. Adibide gisa, motxilaren probleman kapazitate-murrizketa dugu; hori dela eta, soluzio batek kapazitate-muga gainditzen badu, bideraezina izango da. Soluzioa konpontzeko banan-banan atera ditzakegu elementuak murrizketa bete arte.

Azken hurbilketa hori motxilaren probleman erabil dezakegu. Adibide gisa, ikus dezagun knapsack problemarako soluzioak konpontzeko algoritmo bat. Lehenik eta behin, soluzioen bideragarritasuna aztertzeko funtzio bat inplementatuko dugu. Gogoratu soluzio bat bideragarria dela baldin eta motxilan sartutako elementuen pisua motxilaren muga baino txikigoa bada.

[9] Oro har, bilaketa hasierako iterazioetan zigortze-koefiziente txikiak erabiliko ditugu, eta gero handitu –gogoratu bilaketa amaitzen denean soluzio bideragarria nahi dugula–. Bilaketaren progresioari buruzko informazioa erabil daiteke zigortzea egokitzeko, «adaptative penalizing» deritzen estrategiak erabiliz.

```
> valid <- function(solution, weight, limit) {
+     return(sum(weight[solution]) <= limit)
+ }
```

Orain, funtzio hori kontuan hartuz, soluzioak zuzentzeko funtzioa inplementa dezakegu. Funtzioak inplementatzen duen sasi-kodea 1.2 algoritmoan ikus daiteke. Oso algoritmoa sinplea da; soluzioa bideraezina den bitartean, pisu/balio ratio handiena duen elementua motxilatik aterako da.

```
> correct <- function(solution, weight, value, limit) {
+   wv.ratio <- weight / value
+   while(!valid(solution, weight, limit)) {
+     max.in <- max(wv.ratio[solution])
+     id <- which(wv.ratio == max.in & solution)[1]
+     solution[id] <- FALSE
+   }
+   return(solution)
+ }
```

Ikus dezagun adibide bat. Demagun $P = \{2, 6, 3, 6, 3\}$, $W = \{1, 3, 1, 10, 2\}$ eta $c.m = 5$ balio-bektorea, pisu-bektorea eta motxilaren edukiera maximoa direla hurrenez hurren. Motxilan elementu guztiak sartzen dituen soluzioa bideraezina da, pisu totala (8) muga baino handiagoa baita.

```
> p <- c(2, 6, 3, 6, 3)
> w <- c(1, 3, 1, 10, 2)
> c.m <- 5
> solution <- rep(TRUE, times=5)
> valid(solution=solution, weight=w, limit=c.m)

## [1] FALSE

> w / p

## [1] 0.5000000 0.5000000 0.3333333 1.6666667 0.6666667
```

Azken lerroan ikus daiteke ratiorik handiena duena 4. elementua dela; beraren balioa handia da, baina baita pisua ere. Beraz, elementu hori motxilatik aterako dugun lehena izango da. Dena dela, aldaketa horrekin bakarrik ez dugu soluzio bideragarri bat lortuko. Hori ikusirik, ratiorik handiena duen bigarren elementua kenduko dugu: 5. elementua. Orain, bi aldaketa horiek egin ostean lortu dugun soluzioa bideragarria da.

```
> solution

## [1] TRUE TRUE TRUE TRUE TRUE

> valid(solution=solution, weight=w, limit=c.m)

## [1] FALSE

> corrected.solution <- correct(solution=solution, weight=w,
+                                value=p, limit=c.m)
```

```
>
> corrected.solution

## [1]  TRUE  TRUE  TRUE FALSE FALSE

> valid(solution=corrected.solution, weight=w, limit=c.m)

## [1] TRUE
```

Algoritmoak

Ikerkuntza Operatiboaren lehenengo urteetan ikertzaileek problema mota partikularrentzat soluzio optimoa lortzeko algoritmoak garatzen ziharduten. Problema horiek «programazio matematiko» deritzon alorrean sartzen dira[10] eta ebazteko hainbat algoritmo zehatz planteatu dira; hala nola, dagoeneko aipatu dugun Simplex algoritmoa, edo barne-puntu metodoa, adarkatze- eta bornatze-algoritmoak, eta abar.

Metodo horiek problema mota konkretuak ebazteko diseinaturik daude, eta hortaz, ezin dira edozein optimizazio-problema ebazteko erabili. Nahikoa ez eta, nahiz eta problemen konplexutasun maila handia ez izan, problemaren tamaina handiegia bada ere, algoritmoa hauek erabilezinak gerta daitezke.

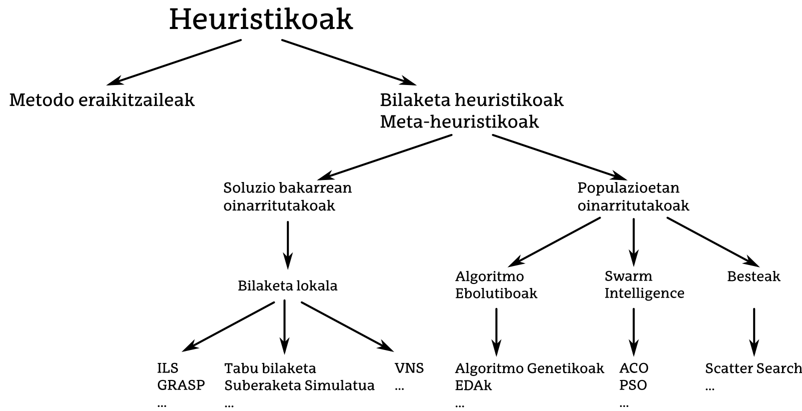
Gauzak horrela, ez dago hainbat egoeratan problemaren optimo globala topatzerik. Kasu horretan, zer egin dezakegu? Eskuragarri ditugun baliabideekin soluzio onena topatzea ezinezkoa bada, ahalik eta soluziorik onena topatzen saiatuko gara; alegia, optimo globaletatik albait gertuen dagoen soluzio bat topatzen saiatuko gara.

Soluzio hurbilduak lortzeko bi aukera ditugu: metodo hurbilduak, zeinek hurbilketa maila bermatzen duten, eta metodo heuristikoak, ezer bermatzen ez dutenak.

Metodo heuristikoak intuizioan oinarritzen dira problema bat optimizatzerakoan. Intuizioa bi motatakoa izan daiteke:

- **Problemari buruzko intuizioa** - Posible denean, problemaren ezaugarriak soluzioa topatzeko *ad hoc* metodo heuristikoak diseinatzeko erabiliko ditugu. Ohikoena algoritmo horiek «eraikitzaileak» izatea da, hau da, soluzioa pausoz pauso eraikitzen duten metodoak. Are gehiago, pauso bakoitzean aukera guztietatik onena hartzea da ohikoena; irizpide horri jarraitzen dioten algoritmoei «gutiziatsu» edo «jale» –*greedy*, ingelesez– esango diegu. Atal honen hasieran TSP problemak ebazteko horrelako algoritmo bat ikusi dugu. Metodo heuristikoak problemaren mamiari egokituta daude, eta horrek alde onak eta txarrak ditu. Oro har algoritmo eraginkorrak izan ohi dira, baina bestelako problemetan berrerabiltzeko desegokiak –edo aplikaezinak– izan daitezke.

[10] Programazio lineala eta programazio osoa hauen adibideak dira.



1.5 irudia. Metodo heuristikoen eskema

- **Bilaketa-prozesuari buruzko intuizioa** - *Ad hoc* diseinaturiko metodoak zailak dira birziklatzeko problemari buruzko intuizioan oinarritzen direlako; horren ordez, intuizioa bilaketa-prozesuan bertan bilatzen badugu, edozein problema ebazteko egoki daitezkeen metodoak diseina genitzake. Algoritmo horiei, *bilaketa heuristikoak* edo *meta-heuristikoak* deritze eta heuristikoak sortzeko txantiloi moduan ikus daitezke. Beste era batean esanda, problema bakoitza ebazteko egokitu behar diren eskemak dira.

Meta-heuristiko mota asko daude, bakoitza bere intuizioan oinarritutakoa. Metodoak sailkatzeko era asko egon arren, sailkapen hedatuenak bi multzotan banatzen ditu:

- **Soluzio bakarrean oinarritutako metaheuristikoak** - Metodo hauetan uneoro soluzio bakar bat mantentzen dugu, eta soluzio horretatik abiatuta beste batera mugitzen saiatuko gara; mugimenduak nola egiten diren desberdintzen du algoritmoak. Bilaketa lokala da metodo hauen arteko ezagunena; alabaina, metodo horrek arazo larri bat du: optimo lokaletan trabaturik gelditzen da. Arazo hori saihesteko zenbait hedapen proposatu dira; hala nola tabu-bilaketa [15], GRASP algoritmoa [14], suberaketa simulatua [23, 37], etab.
- **Populazioetan oinarritutako meta-heuristikoak** - Algoritmo hauetan, soluzio bakar bat izan beharrean soluzio-«populazio» bat –multzo bat, alegia– mantentzen dugu; iterazioz iterazio populazioari aldaketak egingo zaizkio, eta hala horren «eboluzioa» ahalbidetzen, eta geroz eta soluzio hobeagoak lortzen da. Atal honetan dauden algoritmo askok naturan bilatzen dute inspirazioa; era horretan, adibidez, algoritmo genetikoak [19] ditugu, eboluzioan oinarritutakoak, edo inurri-kolonia algoritmoa [11], inurrien talde-portaeran oinarritzen dena.

1.5 irudian optimizaziorako heuristikoen eskema orokor bat ikus daiteke. Meta-heuristiko asko daude, baina denak gauza berdina egiteko diseinatuta daude: bilaketa-espazioa miatzeko. Miaketa-prozesuan bi estrategia erabil –eta, batez ere, konbina– daitezke: *dibertsifikazioa* eta *areagotzea*. Dibertsifikatzean espazioko eremu handiak aztertzen ditugu, baina xehetasun handirik gabe; helburua bilaketa-espazioko eremu interesgarriei antzematea da –espazioa esploratzea, alegia–. Areagotzeak, berriz, topatu ditugun eremu interesgarri horiek sakonki arakatzea dauka helburutzat; batzuetan esaten den legez, eremu onak esplotatzea. Oro har, bilaketa lokal motako algoritmoetan areagotzeari garrantzi handiagoa ematen zaio; populazioetan oinarritutako algoritmoetan, berriz, dibertsifikazioa da helburu nagusia.[11]

[11] Izan ere, azken kapituluan ikusiko dugun bezala, teknikak nahas daitezke, bilaketa lokalean oinarritutako areagotze-pausoak populazioetan oinarritutako metodoei gehituz.

2
Soluzio bakarrean oinarritutako algoritmoak

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

Kontzeptu orokorrak

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

Hainbat algoritmo dira ideia horretan oinarritzen direnak, bakoitza bere berezitasunak izanik, noski. Diferentziak diferentzia, zenbait elementu berdinak dira algoritmo guztietan; atal honetan kontzeptu horiek aztertzeari ekingo diogu.

Soluzioen inguruneak

Bilaketa lokalean dagoen kontzepturik garrantzitsuena ingurunea 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.

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

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

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

[1] Programazio-testuinguruetan – sasikodeetan, adibidez– soluzioak erabiltzeko erabiltzen diren funtzioei «operadore» deritze, eta, hortaz, «funtzio» eta «operadore» terminoak baliokide gisa erabiliko ditugu.

2.1 adibidea. Lokaltasuna *feature subset selection* probleman *Datu-meatzaritzan, sailkatzaile-funtzioak eraikitzea edo ikastea oso ataza ohikoa da. Funtzio horiek, izenak berak adierazten duen moduan, datu berriak sailkatzeko erabiltzen dira.*

Oro har, datuetan agertzen diren aldagaiek iragarpenak egiteko gaitasun ezberdinak dituzte; are gehiago, aldagai batzuek eragin negatiboa izan dezakete sailkatzailearen funtzionamenduan. Hori dela eta, aldagaien azpimultzo bat aukeratzea oso pauso ohikoa da; prozesu hori, ingelesez feature subset selection (FSS) *deitzen den optimizazio-problema bat da. FSS problemetarako soluzioak bektore bitarren bidez kodetu daitezke, eta bit bakoitzak dagokion aldagaia azpimultzoan dagoenetz adierazten du.*

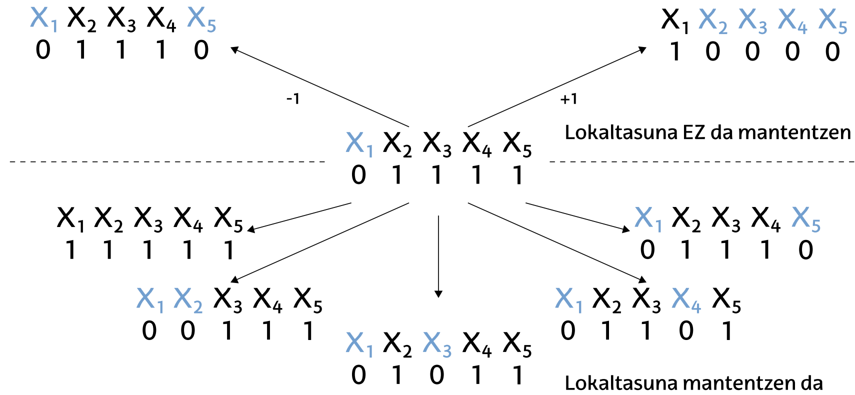
Bektore bitarrak zenbaki oso moduan interpreta daitezke, eta, hortaz, inguruneko soluzioak lortzeko modu bat horiei balio txikiak gehitzea/kentzea da. Adibidez, (01001) *soluzioak* 9 *zenbakia adierazten duela joko dugu, eta, hortaz, antzerako soluzioak lor ditzakegu 1 zenbakia* (00001) *gehituz –*10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

Azpimultzo problemetan ingurune-operadore ohikoena flip *aldaketan oinarritze; hau da, 0 bada 1ekin ordezkatzen da, eta 1 bada, 0rekin. Operadore horrekin FSS probleman beti bermatzen da lokaltasuna, ingurunean dauden edozein bi soluziok aldagai bakar bateko diferentzia izango baitute. 2.1 irudiak adibidea grafikoki erakusten du.*

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

$$N(s;k) = \{s' \in \mathcal{S} \mid d(s,s') \leq k\} \tag{2.1}$$

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



2.1 irudia. Bi ingurune ezberdinen adibidea. Goikoan bektore bitarrei 1 gehitzen/kentzen diegu inguruneko soluzioak lortzeko. Eskuman dagoen soluzioan ikus daitekeen bezala, lokaltasuna ez da mantentzen, soluzioak beren artean oso ezberdinak direlako. Beheko ingurunea *flip* eragiketan oinarritzen da. Kasu horretan sortutako soluzio guztiak antzerakoak dira.

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

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

Ingurune-tamainari erreparatuz, zenbaki errealekin dihardugunez, infinitu soluzio izango ditugu edozein soluzioren ingurunean. Euklidearra distantziarik ezagunena izan arren, badira beste metrika batzuk ere bektore errealen arteko distantzia neurtzeko – 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: $d_h(s, s') = \sum_{i=1}^{n} I[s_i \neq s'_i]$, non I funtzioa adierazlea den, eta 1 balioa hartzen du bere argumentua $s_i \neq s'_i$ egia denean, eta 0 beste kasuan. Hortaz, Hamming distantziak posizioz posizioko desberdintasunak neurtzen ditu. Hamming distantzia inguruneak definitzeko

erabiltzen denean, ohikoena 1 distantziara dauden soluzioetara mugatzea da, hau da:

$$N_h(s;1) = \{s' \in \mathcal{S} \mid d_h(s,s') = 1\} \quad (2.2)$$

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

Bestalde, ondoko ekuazioak edozein distantziatarako ingurune-funtzio orokor bat adierazten du –distantzia maximoa bektorearen tamaina dela kontuan hartuz, betiere–:

$$N_h(s;k) = \{s' \in \mathcal{S} \mid d_h(s,s') \leq k\} \quad (2.3)$$

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

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

Kontuan hartu behar da motxilaren probleman soluzio guztiak –azpimultzo guztiak, alegia– ez direla bideragarriak. Eta, beraz, zorizko soluzio bat sortzean, elementu edo objektu bat aukeratzeko probabilitatea handitzen badugu, soluzio bideraezinak lortzea probableagoa izango da. Edozein kasutan, sortutako soluzioa bideraezina izan daitekeenez, lehenengo pausoa soluzioa zuzentzea izango da. Gero, «flip» operadorea erabiliko dugu inguruneko soluzioak sortzeko. Operadore horrek Hamming-en bat distantziara dauden soluzioak esleituko dizkio inguruneari. Nahiz eta uneko soluzioa bideragarria izan, ingurunekoak bideraezinak izan daitezke; beraz, pauso bakoitzean inguruneko soluzioa bideragarria den ala ez aztertu beharko dugu.

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

## [1] 2 9

> flip.ngh <- flipNeighborhood(base=rnd.sol, random=FALSE)
> while(hasMoreNeighbors(flip.ngh)) {
+   ngh <- nextNeighbor(flip.ngh)
+   is.valid <- ifelse(test=knp$valid(ngh),
+                      yes="bideragarria",
+                      no="bideraezina")
+   message("Inguruneko soluzio ", is.valid, ": ",
+           paste(which(ngh), collapse=","))
+ }

## 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, **metaheuR** paketean inguruneak korritzeko bi funtzio aurki ditzakegu: `hasMoreNeighbors` eta `nextNeighbor`. Izenek adierazten duten bezala, lehenengo funtzioak ingurunean oraindik bisitatu gabeko soluzioren bat dagoen esaten digu, eta, bigarrenak, bisitatu gabeko hurrengo soluzioa itzultzen du. Horrez gain, badago beste funtzio bat, `resetNeighborhood`, ingurune-objektua berrabiarazteko. Informazio gehiago lor dezakezu R-ko terminalean `?resetNeighborhood` tekleatuz.

Permutazioak - Permutazioen arteko distantziak neurtzeko metrikak badiren 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 haien balioak trukatzen dira. Adibidez, 21345 permutazioaren 1. eta 3. posizioak trukatzen baditugu 31245 permutazioa lortuko dugu. Formalki, trukaketa-funtzioa, $t_r(s;i,j)$, defini dezakegu non $s' = t_r(s;i,j)$ bada $s'(i) = s(j)$, $s'(j) = s(i)$ eta $\forall k \neq i,j$, $s'(k) = s(k)$ beteko den. Funtzio horretan oinarriturik, ondoko ingurunea defini dezakegu:

$$N_{2opt}(s) = \{t_r(s; i, j) | \; 1 \leq i, j \leq n, \forall i > j\} \quad (2.4)$$

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

Ingurunearen tamainari dagokionez, ondoz-ondoko posizioetan soilik trukaketak eginez $n-1$ ingurune soluzio izango ditugu; edozein bi posizio trukatzen baditugu, berriz, ingurunearen tamaina $n(n-1)$ izango da. Hori adibide honetan ikus daiteke.

```
> n <- 10
> rnd.sol <- randomPermutation(length=n)
>
> swp.ngh <- swapNeighborhood(base=rnd.sol)
> exchange.count <- 0
> swap.count <- 0
> while(hasMoreNeighbors(swp.ngh)) {
+   swap.count <- swap.count + 1
+   nextNeighbor(swp.ngh)
+ }
>
> ex.ngh <- exchangeNeighborhood(base=rnd.sol)
> exchange.count <- 0
> while(hasMoreNeighbors(ex.ngh)) {
+   exchange.count <- exchange.count + 1
+   nextNeighbor(ex.ngh)
+ }
>
> swap.count

## [1] 9

> exchange.count

## [1] 45
```

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

$$N_{in}(s) = \{t_x(s; i, j) | \; 1 \leq i, j \leq n, \forall i \neq j\} \quad (2.5)$$

Trukaketan bakarrik bi posiziotako balioak aldatzen dira; txertaketan, berriz, bi indizeen artean dauden posizio guztietako balioak aldatzen dira. Hori

dela eta, ingurune operadore bakoitzaren erabilgarritasuna problemaren araberakoa izango da. Operadore hau ere **metaheuR** paketean aurki dezakegu, `InsertNeighborhood` klasean inplementaturik.

Bi ingurune-operadore horietaz gain, badira literaturan beste zenbait operadore, inbertsio-eragiketan oinarritutakoak esate baterako.

Optimo lokalak

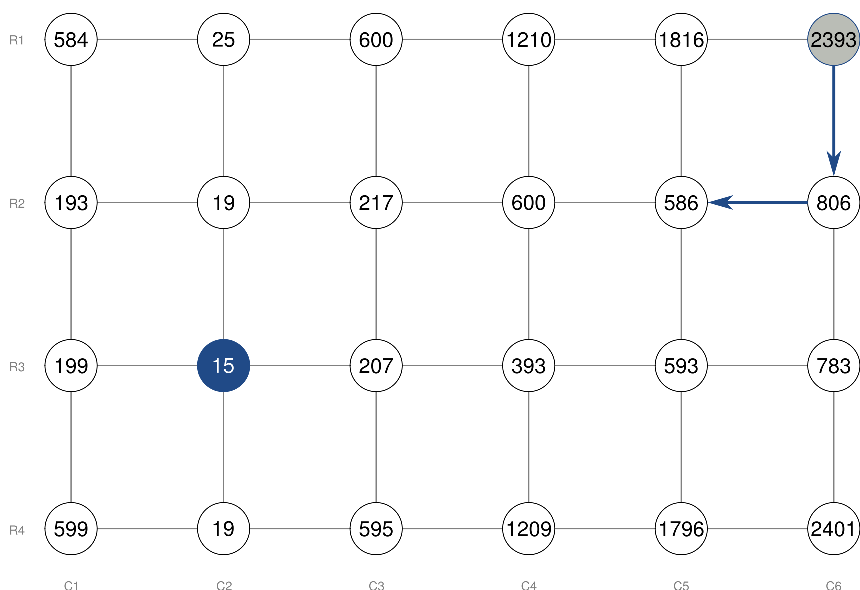
Bilaketa lokalean –oinarrizko bertsioan, behintzat– soluzio batetik beste batera mugitzeko, uneko soluzioaren ingurunean, helburu-funtzioaren balioa hobetzen duen bizilagun batek egon behar du[2]. Inguruneko soluzio guztien fitness-a txarragoa baldin bada, orduan soluzio hori *optimo lokala* dela esango dugu, eta bilaketa amaitu egingo da. Formalki, s^* optimo lokala da, baldin eta ondoko baldintza betetzen baldin badu:

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

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

Definizio horiek kontuan hartuz, bilaketa lokala beti optimo lokal batean amaitzen dela ondorioztatzen dugu. Hauxe da, hain zuzen, bilaketa lokalaren ezaugarri –eta, aldi berean, desabantaila– nagusia. Aintzat hartzekoa da optimo lokalak, nahiz eta beren inguruneko soluziorik onenak izan, nahiko soluzio txarrak izan daitezkeela, 2.2 irudian erakusten den bezala. Irudi horretan, kapituluan zehar maiz erabiliko dugun grafiko mota bat ikus daiteke. Grafikoan, fikziozko problema baterako soluzio guztiak jasotzen dira, bakoitza biribil baten bidez adierazita; biribilen barruan soluzio bakoitzaren fitness-a dago idatzita. Ingurune-funtzioa soluzioak lotzen dituzten marren bidez adierazten da; hala, bi soluzio lotuta badaude, bata bestearen ingurunean daudela diogu –bizilagunak direla, alegia.

[2] Helburu-funtzioak soluzio jakin batean hartzen duen balioa, inglesezko *fitness* hitzarekin izendatzea ohikoa da. Horregatik, kapituluan zehar, bi adierazpideak erabiliko ditugu baliokide gisa.

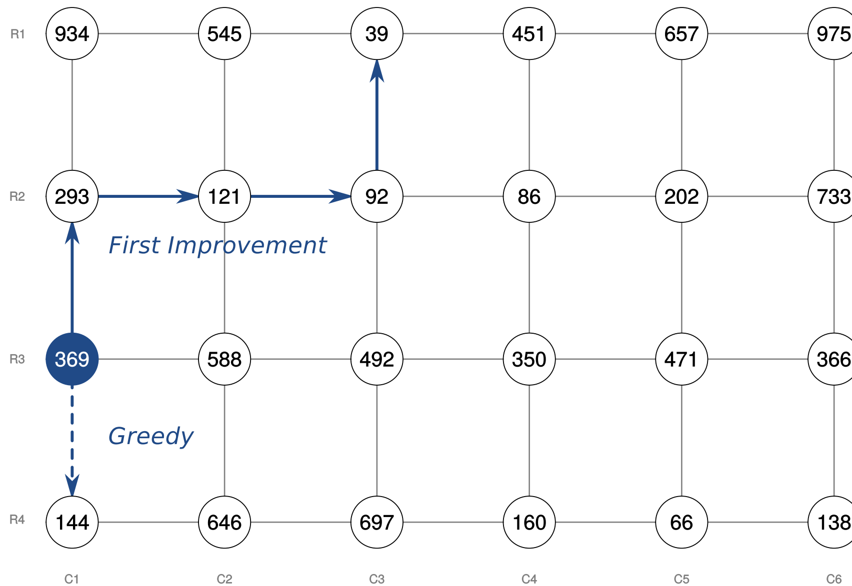


2.2 irudia. Optimo lokalaren adibidea. Goiko eskumako soluziotik abiatzen bada bilaketa –(R1,C6), grisean nabarmendua dagoen soluziotik, alegia–, eta pauso bakoitzean aukerarik onena aukeratzen badugu, geziek markatzen duten bideari jarraitu eta, bi pausotan, (R2,C5) soluzioan trabatuta gelditukо gara. Soluzio hori optimo lokala da, bere inguruneko soluzio guztiak txarragoak baitira. Optimo lokalaren ebaluazioa 586 da, oso txarra optimo globalarekin alderatzen badugu –(R3,C2) soluzioa–.

2.2 adibidea. *2.2 irudian agertzen diren geziek algoritmoak (R1,C6) soluziotik abiatuta egiten duen bidea erakusten dute. Pauso bakoitzean inguruneko soluziorik onena aukeratzen badugu, algoritmoa bi pausotan trabatuta geldituko da (R2,C5) soluzioan; soluzio horren fitness-a 586 da, eta, bere ingurunean dauden soluzioen fitness-ak handiagoa direnez –1816, 806, 593 eta 600–, ez dago helburu-funtzioaren balioa hobetzen duen soluziorik. (R2,C5) optimo lokal bat da, eta, optimo globalaren –(R3,C2)– fitness-a 15 dela kontuan hartuz, nahiko soluzio txarra ere bai.*

Optimo lokaletatik ateratzeko hainbat estrategia planteatu dira literaturan, bilaketa lokalaren puntu batean edo bestean aldaketak proposatuz. Haueexek izango dira, hain justu, 2.3. atalean aztergai izango ditugunak.

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



2.3 irudia. Inguruneko soluzioaren aukeraketaren efektua. Irudiak, soluzio berdinetik abiatuta –(R3,C2), grisean nabarmendua– bi estrategia 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 hori erabiliz egindako ibilbidea (R2,C1), (R2,C2), (R2,C3), (R1,C3) da, azken soluzio hori optimo lokala izanik. Bigarren estrategia gutiziatsua da –*greedy*-a, alegia–; aukeratzen dugun hurrengo soluzioa ingurunean dagoen onena izango da beti. Estrategia hori erabiliz pauso bakar batean (R4,C1) ailegatzen gara optimo lokalera.

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

Erakarpen-arroa, definizioan ikus daitekeen legez, helburu-funtzioaren, ingurunearen eta algoritmoaren araberakoa da. Alde batetik, ingurunearen eta helburu-funtzioaren eragina begi bistakoa da. Bestetik, algoritmoak ingurunea nola aztertzen den eta, bereziki, zer soluzio aukeratzen den ezartzen du; ondorioz, egiten dugun ibilbidean eragin handia izan dezake. 2.3 irudian, aukeraketa-estrategiek duten eragina ilustratzen da.

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' ∈ 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)
```

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

Bilaketa lokala

Bilaketa lokalean oinarritutako edozein algoritmoren errendimendua, kodeketaren eta ingurunearen aukeraketaz gain, beste zenbait elementutan ere oinarritzen da. Lehenik eta behin, hasierako soluzioa nola aukeratzen dugun erabakitzea garrantzitsua da, zeren aurreko atalean ikusi dugun moduan, horren arabera optimo lokal batean edo bestean amaituko baita bilaketa. Bigarren oinarrizko elementua inguruneko soluzioen aukeraketa egiteko aplikatzen den estrategia da. Uneko soluzioaren ingurunean hainbat soluzio izango ditugu, baina zein aukeratuko dugu hurrengo soluzioa izateko? Azkenik, gelditze-irizpideak ere kontuan hartu beharreko faktoreak dira. Bilaketa lokala optimo lokal bat topatzen dugunean amaitzen da; dena dela, beste edozein algoritmotan bezala, denboran edota ebaluazio kopuruan oinarritutako gelditze-irizpideak ere proposa ditzakegu[3]. 2.1 algoritmoan oinarrizko bilaketa lokalaren sasikodea ikus daiteke.

Sasikodean dagoen algoritmoa **metaheuR** paketeko `basicLocalSearch` funtzioak inplementatzen du. Funtzio horrek zenbait parametro ditu, batzuk algoritmoarekin zerikusia dutenak eta beste batzuk problemari eta exekuzioari lotuta daudenak. Paketean dauden metaheuristika guztiek antzerako egitura izango dutenez, pausoz pauso aztertuko ditugu parametro horiek. Gauzak horrela, parametroak hiru motatakoak dira:

- Problemari lotutako parametroak - Bilaketa gidatzeko helburu-funtzio bat behar dugu. Funtzio hori `evaluate` parametroaren bidez pasatuko diogu algoritmoari. Algoritmoen inplementazioa orokorra denez, gerta daiteke problema batzuetarako bideraezinak diren soluzioak agertzea. Problema

[3] Informazio gehiago R-ren laguntzan duzu; `?basicLocalSearch` tekleatu laguntza zabaltzeko.

mota horiekin lan egin ahal izateko, **metaheuR** paketeak soluzioen bideragarritasuna aztertu eta soluzio bideraezinak konpontzeko funtzioak parametro gisa sartzea ahalbidetzen du. Funtzio horiek problema bakoitzeko ezberdinak izango dira, eta algoritmoari `valid` eta `correct` parametroen bidez pasatuko dizkiogu, hurrenez hurren.

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

Jarraian `basicLocalSearch` funtzioaren eta bere parametro guztien erabilera aztertuko dugu adibide baten bidez. Adibiderako grafoen koloreztatze-problema bat sortuko dugu, `graphColoringProblem` funtzioa erabiliz eta zorizko grafo bat hautatuz. Horrela, `gcp` objektuak problemaren ebaluazio-funtzioa eta soluzio bideragarriekin tratatzeko funtzioak gordeko ditu.

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

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

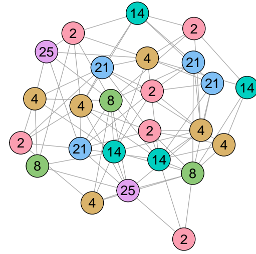
```
> resources <- cResource(time=10, evaluations=100 * n^2,
+                         iterations=100 * n)
```

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

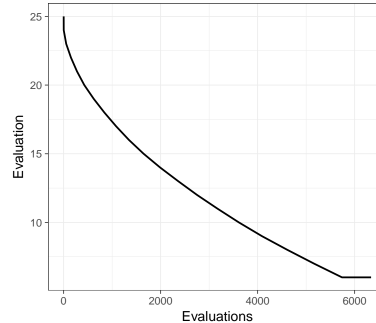
```
> colors <- paste("C", 1:n, sep="")
> initial.solution <- factor (colors, levels=colors)
> h.ngh <- hammingNeighborhood(base=initial.solution)
```

Dena prest daukagu bilaketa abiarazteko ...

```
> args <- list()
> args$evaluate          <- gcp$evaluate
> args$valid             <- gcp$valid
> args$correct           <- gcp$correct
> args$initial.solution  <- initial.solution
> args$neighborhood      <- h.ngh
> args$selector          <- firstImprovementSelector
> args$non.valid         <- "correct"
> args$resources         <- resources
>
> bls <- do.call(basicLocalSearch, args)

## 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: 3.05213379859924
## Evaluations: 6339
## Iterations: 19
## None of the resources completely consumed
## You can use functions 'getSolution', 'getParameters' and
## 'getProgress' to get the list of optimal solutions,
## the list of parameters of the search and the log of
## the process, respectively
```



(a) Problemaren soluzioa



(b) Bilaketaren progresioa

2.4 irudia. Grafoen koloreztatze-problemarako bilaketa lokalak topatutako soluzioa eta egindako bilaketaren progresioa. Bigarren grafiko horretan, X ardatzak ebaluazio kopurua adierazten du, eta Y ardatzak uneko soluzioaren ebaluazioa –soluzioak erabiltzen dituen kolore kopurua, alegia–.

```
> final.solution <- getSolution(bls)
> as.character(final.solution)

## [1] "C2" "C2" "C4" "C4" "C2" "C4" "C8" "C8" "C2" "C4" "C2"
## [12] "C2" "C14" "C14" "C4" "C14" "C4" "C14" "C8" "C21" "C21"
## [23]  "C21" "C21" "C25" "C25"

> gcp$plot(solution=final.solution, node.size=20,
+          label.cex=1.25) + thm.bh

## NULL
```

Bilaketa lokalak –eta, oro har, gainontzeko algoritmoek– objektu berezi bat itzultzen dute, `mHResult` klasekoa. Bertan dagoen informazioa funtzio sorta baten bitartez lor daiteke; este baterako, `optima` funtzioak lortutako soluzio optimoa(k) itzultzen ditu, zerrenda batean. Bilaketa lokalak soluzio bakarra itzultzen duenez, soluzio hori zerrendako 1. posizioan egongo da. Soluzioa grafikoki bistaratzeko `graphColoringProblem` funtzioak itzultzen duen `plot` funtzioa erabil daiteke. Gainera, bilaketaren progresioa ere bistara dezakegu, `plotProgress` funtzioa erabiliz. 2.4 irudiak problemarako soluzioa eta bilaketaren progresioa jasotzen ditu.

```
> plotProgress(bls, size=1.1) + labs(y="Evaluation") + thm.bh
```

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

Hasierako soluzioaren aukeraketa

Lehen aipatu bezala, bilaketa lokala soluzio batetik abiatuko da beti. Bilaketa nondik hasten den oso garrantzitsua da, horren arabera optimo lokal batean edo bestean amaituko baita bilaketa. Adibidez, hori argi ikusten da 2.2 irudian; (R1,C6) soluziotik hasten badugu bilaketa, (C2,R5) soluzioan amaituko da. Gauza bera gertatzen da optimo lokaletik bertatik –(C2,R5)–, goian dagoen soluziotik –(C1,R5)– edo bere eskuinean dagoen soluziotik –(C2,R6)– hasten bada prozesua. Beste edozein soluzio aukeratzen badugu, berriz, optimo globalera helduko gara.

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

- **Zorizko soluzioak sortu** - Zoriz aukeratzen da bilaketa-espazioan dagoen soluzio bat, eta hortik hasten da bilaketa. Metodo honen abantaila bere sinpletasuna da, zorizko soluzioak sortzea, eskuarki, erraza izaten baita. Problemak murrizketa asko dituen kasuetan, ordea, premisa honek ez du balio, baliozko zorizko soluzioak sortzea asko zaildu baitaiteke. Hala ere, estrategia honek alde txarrak ere baditu; alde batetik, hasierako soluzioa txarra bada, bilaketa-prozesua luzea izan daiteke, eta, bestetik, algoritmoa aplikatzen dugun bakoitzean emaitza, oro har, ezberdina izango da.
- **Soluzio onak eraiki** - Lehenengo kapituluan ikusi genuen problema bakoitza ebazteko metodo heuristiko espezifikoak diseina daitezkeela. Oro har, metodo horiek pausoz pauso eraikitzen dituzte soluzioak, urrats bakoitzean aukera guztietatik onena aukeratuz –ingelesez metodo horiei *constructive greedy* deritze, hau da algoritmo eraikitzaile gutiziatsuak edo jaleak–. Nahiko soluzio onak lortu arren, ez dute zertan optimoak izan[4], eta, beraz, lortutako soluzioak bilaketa lokala hasieratzeko erabil daitezke. Estrategia hauek zorizko soluzioetatik abiatzeak baino emaitza hobeak lortzen ditu normalean, bilaketak iterazio gutxiago behar izaten baititu; konputazionalki, ordea, garestiagoa da.

Inguruneko soluzioaren aukeraketa

Behin inguruneko soluzioen multzoa definiturik dugula, hurrengo pausoa soluzio horien artean bat aukeratzeko irizpidea ezartzea da. Lehen aipatu bezala, bi dira, nagusiki, erabiltzen diren estrategiak. Lehenengo estrategian inguruneko soluzioak banan-banan analizatzen dira eta fitness-a hobetzen duen lehenengo soluzioa aukeratzen da. Hurbilketan horretan, inguruneko soluzioen «ordenazioa» oso garrantzitsua da, uneko soluzioa hobetzen duen lehenengo soluziora mugituko baikara. Bigarren estrategiak ingurune osoa

[4] Ez optimo globalak eta ezta lokalak ere.

arakatzen du, eta fitness-a gehien hobetzen duen soluzioa aukeratzen du. Oro har, inguruneak txikiak direnean bigarren hurbilketa da interesgarriena, baina, inguruneak handiak direnean, kostu konputazionala dela eta, bideraezina gerta daiteke estrategia hori.

2.3 adibidea. *Demagun problema baterako soluzioak bektore bitarren bidez kodetzen ditugula. Uneko soluzioa* $(0, 1, 1, 0, 1)$ *da, dagokion fitness-a* 25 *izanik. Ingurunea definitzeko* (2.2) *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 lortzeko posizio bakoitzeko balioa banan-banan aldatu behar dugu. Lehenengo posiziotik abiatzen bagara, s_2 *soluzioa izango da fitness-a hobetzen duen lehenengo soluzioa, beraren ebaluazioa* 24 *baita. Azken posiziotik abiatzen bagara, berriz,* s_3 *soluzioarekin geldituko ginateke, ebaluazioa* 5 *baita. Kasu bakoitzean soluzio ezberdina aukeratu dugu lehenengo pauso honetan. Hortaz, hurrengo urratsean izango dugun ingurunea ere ezberdina izango da. Hori dela eta, inguruneko soluzioen azterketa ordena desberdinetan eginez, azken soluzioa ezberdina izan daiteke. Inguruneko soluziorik onena aukeratzen, ordea, ordenak ez du garrantzirik, eta beti soluzio berdina topatuko dugu, baldin eta berdinketarik ez badago.*

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

Jarraian azaltzen den kodean ingurunearen azterketan ordenak duen eragina erakusten da. Lehenik eta behin, TSPlib errepositorioan dagoen problema bat kargatuko dugu, **metaheuR** paketeko `tsplibParser` funtzioa erabiliz. TSPlib errepositorioan TSP problemaren zenbait adibide topa ditzakegu. Erabiliko dugun probleman Bavariako 29 hiri izango ditugu. Hasierako soluzio gisa identitate-permutazioa hartuko dugu.

```
> url <- system.file("bays29.xml.zip", package = "metaheuR")
> cost.matrix <- tsplibParser(url)
> n <- dim(cost.matrix)[1]
> tsp.babaria <- tspProblem(cost.matrix)
> csol <- identityPermutation(n)
> csol

## An object of class "Permutation"
## Slot "permutation":
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
```

```
## [24] 24 25 26 27 28 29
```

```
> eval <- tsp.babaria$evaluate(csol)
> eval
```

```
## [1] 5752
```

Problema definitu ostean, hasierako soluzioaren ingurunea sortuko dugu. *2-opt* ingurunea aukeratu dugu, zorizko eta ez-zorizko esplorazioak hautatuz. Lehenengoan inguruneko soluzioak zorizko ordena batean aztertuko dira, eta bigarrenean, ordea, kodeketaren araberako ordena zehatz bati jarraituko zaio ingurunea arakatzeko.

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

Bilaketaren pauso bakoitzean inguruneko helburu-funtzioa hobetzen duen lehenengo soluzioa aukeratzen badugu, hautatutako bi ordenazioak erabiliz emaitza ezberdinak lortuko ditugu. Ingurunea modu horretan arakatzeko, `firstImprovementSelector` funtzioa erabil dezakegu. Funtzio horrek uneko soluzioaren ingurunea arakatzen du, eta fitness-a hobetzen duen lehenengo soluzioa itzultzen du.

```
> firstImprovementSelector(neighborhood=ex.nonrandom,
+                          evaluate=tsp.babaria$evaluate,
+                          initial.solution=csol,
+                          initial.evaluation=eval)$evaluation
```

```
## [1] 5684
```

```
> firstImprovementSelector(neighborhood=ex.random,
+                          evaluate=tsp.babaria$evaluate,
+                          initial.solution=csol,
+                          initial.evaluation=eval)$evaluation
```

```
## [1] 5478
```

Inguruneko soluziorik onena aukeratzen badugu, hautatutako bi ordenazioak erabiliz emaitza bera lortuko dugu, kasu guztietan inguruneko soluzio guztiak aztertzen baitira. Ingurunea aztertzeko estrategia hori **metaheuR** paketeko `greedySelector` funtzioak inplementatzen du.

```
> greedySelector(neighborhood=ex.nonrandom,
+                evaluate=tsp.babaria$evaluate,
+                initial.solution=csol,
+                initial.evaluation=eval)$evaluation
```

```
## [1] 5034
```

```
> greedySelector(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* ([30]) delakoak aipa daitezke. Gainera, metodo heuristikoez gain, tamaina handiko inguruneak era eraginkorrean zehazki aztertzeko algoritmoak ere badaude; horietako adibide bat *dynasearch* ([10]) algoritmoa da.

Bilaketa lokaleko elementuen eragina

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

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

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

## [1] 6360

> tsp.babaria$evaluate(greedy.sol)

## [1] 2307
```

Ikus daitekeenez, algoritmo eraikitzaileak (`tspGreedy`) ematen duen soluzioa zorizkoa baino askoz ere hobea da. Bi soluzio horiek hasierako soluzio gisa hartuz, bilaketa lokala aplikatzen ahal dugu, *swap* ingurunea erabiliz. Gainera, pauso bakoitzean, inguruneko soluziorik onena aukera dezakegu (*greedy*) edo, bestela, uneko soluzioaren fitness-a hobetzen duen lehenengo soluzioa hartu (*first improvement (fi)*). Horrenbestez, lau bilaketa exekutatuko ditugu.

```
> eval <- tsp.babaria$evaluate
> swp.ngh.rnd <- swapNeighborhood(base=rnd.sol)
> swp.ngh.greedy <- swapNeighborhood(base=greedy.sol)
>
> args <- list()
> args$evaluate          <- eval
> args$initial.solution <- rnd.sol
> args$neighborhood     <- swp.ngh.rnd
> args$selector         <- greedySelector
```

```
> args$verbose          <- FALSE
>
> swap.greedy.rnd.sol <- do.call(basicLocalSearch, args)
>
> args$selector   <- firstImprovementSelector
> swap.fi.rnd.sol <- do.call(basicLocalSearch, args)
>
> args$initial.solution <- greedy.sol
> swap.fi.greedy.sol    <- do.call(basicLocalSearch, args)
>
> args$selector          <- greedySelector
> swap.greedy.greedy.sol <- do.call(basicLocalSearch, args)
```

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

```
> init.sol.eval <- tsp.babaria$evaluate(rnd.sol)
> init.sol.eval - getEvaluation(swap.greedy.rnd.sol)

## [1] 1936

> init.sol.eval - getEvaluation(swap.fi.rnd.sol)

## [1] 1317
```

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

```
> rsc <- getResources(swap.greedy.rnd.sol)
> getConsumedEvaluations(rsc)

## [1] 337

> rsc <- getResources(swap.fi.rnd.sol)
> getConsumedEvaluations(rsc)

## [1] 276
```

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

```
> init.sol.eval <- tsp.babaria$evaluate(greedy.sol)
> init.sol.eval - getEvaluation(swap.greedy.greedy.sol)

## [1] 56
```

Inguruneko soluzio guztietatik onena hartzeak emaitza hobeak ematen ditu –zorizko soluziotik abiatzen garenean, behintzat–, bilaketa-espazioa sakonago

aztertzen delako. Bilaketa sakonagoa egiteko beste era bat *swap* ingurunearen ordez *2-opt* ingurunea erabiltzean datza. Izan ere, lehen ikusi dugun bezala, *2-opt* ingurunea *swap* ingurunea baino askoz ere handiagoa da.

```
> ex.ngh.rnd <- exchangeNeighborhood (base=rnd.sol)
>
> args <- list()
> args$evaluate          <- eval
> args$initial.solution <- rnd.sol
> args$neighborhood     <- ex.ngh.rnd
> args$selector         <- greedySelector
> args$verbose          <- FALSE
>
> ex.greedy.rnd.sol <- do.call(basicLocalSearch, args)
>
> tsp.babaria$evaluate(rnd.sol) - getEvaluation(ex.greedy.rnd.sol)

## [1] 3951

> getConsumedEvaluations(getResources(ex.greedy.rnd.sol))

## [1] 11775
```

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

Bilaketa lokalaren hedapenak

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

Hasieraketa anizkoitza

Bilaketa lokala aplikatzean, soluzio bakoitzetik abiatuz optimo lokal batera heltzen gara; soluzio ezberdinetatik abiatzen bagara, optimo lokal ezberdinetara hel gaitezke. Ideia hori da, hain zuzen ere, hasieraketa anizkoitzeko bilaketa lokalak – *Multistart Local Search*, inglesez– inplementatzen duena (2.2 sasikodean ikus daiteke). Algoritmoan agertzen diren *generate_random_solution* eta *local_search* funtzioetan dago prozeduraren mamia,

Hasieraketa anizkoitzeko bilaketa lokal orokorra

```
1  input: f helburu-funtzioa
2  input: random_solution, stop_criterion eta local_search funtzioak
3  output: s* soluzioa optimoa
4  s=generate_random_solution
5  while !stop_criterion
6     s' = random_solution
7     s'' = local_search(s')
8     if (f(s'') < f(s)) s = s''
9  done
```

2.2 algoritmoa. Hasieraketa anizkoitza erabiltzen duen bilaketa lokalaren hedapenaren sasikode orokorra

haiexek karakterizatuko baitute algoritmoaren performantzia. Lehenengoak, bilaketa-espazioa miatzen du. Bigarrena, aldiz, bere izenak adierazten duen bezala, soluzioen areagotzeaz arduratzen da.

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

Bilaketa Lokal Iteratua (ILS)

Bilaketa lokala berrabiarazteko uniformeki zorizko soluzioak erabili beharrean, ILS –*Iterated Local Search*, ingelesez– algoritmoak uneko optimo lokala hartuko du oinarritzat. Ideia oso sinplea da; optimo lokal batean trabaturik gelditzen garenean, uneko soluzioa «perturbatu» eta bertatik bilatzen jarraituko dugu. Optimo lokal berri batera heltzen garenean, soluzio hori onartuko dugunetz erabaki behar dugu. 2.3 algoritmoan ILSaren sasikode orokorra ikus daiteke.

Algoritmo hori zehazteko bi prozedura berri definitu behar ditugu:

- **Perturbazioa** - Hasteko, optimo lokal batean trabaturik geratzean, hura nola perturbatuko den erabaki behar da. Soluzio baten inguruneko soluzio guztiak antzekoak direnez, optimo lokaletatik edo zehazki haien erakarpen-arroetatik ateratzeko, aldaketa nabarmenak egin behar dira (2.5 irudian adibide bat proposatzen da). Uneko soluzioa (R2,C5) izanik, perturbazio

Bilaketa Lokal Iteratua (ILS)

```
1  input: f helburu-funtzioa
2  input: accept, perturb, stop_criterion eta local_search funtzioak
3  input: s0 hasierako soluzioa
4  output: s* soluzioa
5  s = local_search(s0)
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
```

2.3 algoritmoa. Bilaketa Lokal Iteratuaren (ILS) sasikodea

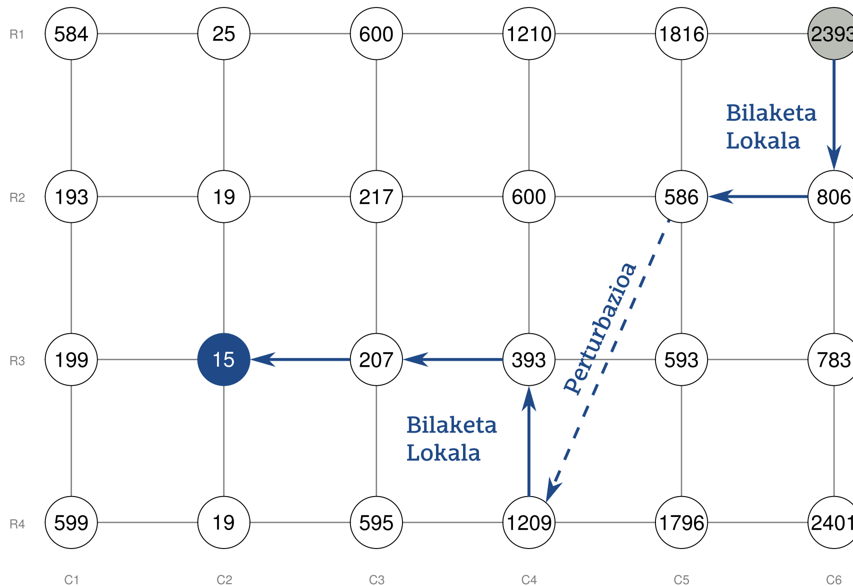
txikiegia egingo bagenu –(R1,C5) soluziora mugitzea, adibidez–, berriro optimo lokal berdinean amaituko litzateke bilaketa. Perturbazioa nahiko handia bada, uneko optimo lokalaren erakarpen-arrotik aterako gara eta, definizioz, beste optimo lokal batean trabaturik geldituko gara. Kontuan hartzekoa da, ordea, perturbazio-prozedurak itzulitako soluzioa erabat zorizkoa bada –hau da, perturbazioa oso handia bada–, ILS algoritmoa zorizko hasieraketa anizkoitzeko algoritmoa bilakatuko dela. Beraz, perturbazio-tamainaren aukeraketak eragina izango du algoritmoaren portaeran.

Perturbazioa definitzean inguruneko soluzioak definitzeko erabiltzen diren operazio 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 hartuz eta zorizko posizioetan sartuz.

Perturbazioaren tamaina aurrez finka daiteke eta bilaketan zehar aldatu barik mantendu edo, bestela, estrategia dinamikoak erabil daitezke, non perturbazioaren maila bilaketan zehar aldatzen den.

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

- **Optimo lokalak onartzeko irizpideak** - Uneko optimo lokala perturbatu ondoren, bilaketa lokala aplikatzen da, optimo (berri) bat sortzeko. Hurrengo iterazioan, lortutako optimo berria edo berriro optimo zaharra perturbatuko dugun erabaki behar da. Bi muturreko hurbilketa plantea



2.5 irudia. ILS algoritmoaren funtzionamendua. Goiko eskumako soluziotik abiatzen bada bilaketa –(R1,C6), grisean nabarmendua dagoen soluziotik, alegia–, (R2,C5) optimo lokalean trabatuta geldituko litzateke bilaketa lokala. Egoera desblokeatzeko soluzioa «perturbatzen» dugu, (R3,C4) soluziora mugituz; hortik abiatuta bilaketa lokala aplikatzen dugu berriro, kasu honetan optimo globalera heldu arte.

daiteke: beti optimo berria onartu, edo soilik unekoa baino hobea denean onartu. Lehendabiziko estrategiak dibertsifikazioa suspertzen du; bigarrena, berriz, soluzioak areagotzeko egokia da. Ohikoena tarteko zerbait erabiltzea da, optimo zaharraren eta berriaren ebaluazioen arteko diferentzia kontuan hartuz. Esate baterako, optimoak era probabilistikoan onar daitezke, Boltzmann-en distribuzioa erabiliz, gero *simmulated annealing* algoritmoan ikusiko dugun bezala.

metaheuR paketean, ILSa `iteratedLocalSearch` funtzioan dago inplementatuta. Funtzio horren parametro gehienak `basicLocalSearch` funtzioaren berberak dira, inplementazioa funtzio horretan oinarritzen baita. Algoritmo horretan ordea, hiru parametro berri izango ditugu:

- `perturb` - Parametro honen bidez soluzioak perturbatzeko erabiliko den funtzioa adieraziko diogu algoritmoari. `perturb` funtzioak parametro bakarra izango du, perturbatu behar den soluzioa, eta perturbazioa aplikatuz lortzen den soluzioa itzuliko du.
- `accept` - Parametro hau soluzio berriak noiz onartzen ditugun definitzen duen funtzioa da. Gutxienez parametro bat izan beharko du, `delta`,

soluzio berriaren eta zaharraren fitness-balioen arteko diferentzia jasoko duena.
- `num.restarts` - Optimo lokalak perturbatuz, bilaketa zenbat alditan berrabiarazi behar dugun esaten duen zenbaki osoa da.

Ikus dezagun adibide bat, TSPlib-eko problema bat erabiliz. Problema horretan Myanmarreko 14 hiriren arteko distantziak izango ditugu. Problema ebazteko zorizko soluzio batetik abiatuko dugu bilaketa. Ingurune gisa *2-opt* erabiliko dugu (trukaketa orokorrak, ez bakarrik ondoz-ondokoak), eta soluzioak perturbatzeko eragiketa bera erabiliko dugu baina behin baino gehiagotan aplikatuz; soluzio bati operazio hau zoriz aplikatzeko `shuffle` funtzioa erabil dezakegu.

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

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

> n <- ncol(burma.mat)
> burma.tsp <- tspProblem(burma.mat)
>
> init.sol <- randomPermutation(n)
> ngh      <- exchangeNeighborhood(init.sol)
> sel      <- greedySelector
```

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

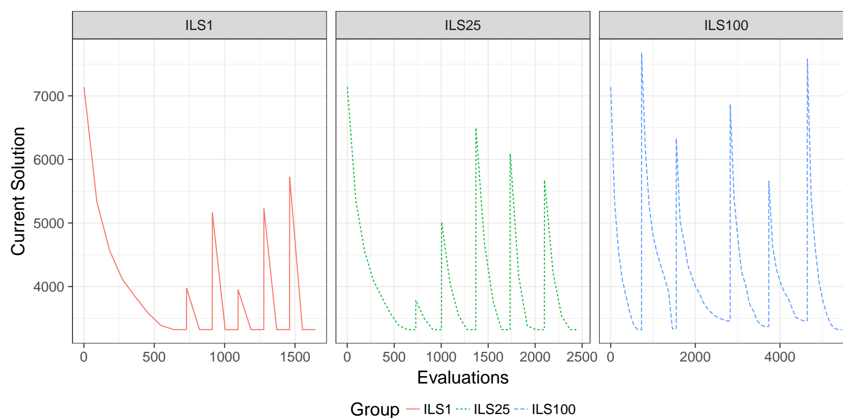
```
> th.accpet <- thresholdAccept
> th <- 0
```

Perturbazio maila soluzioari aplikatuko dizkiogun trukaketen kopuruaren bidez kontrolatuko dugu. Beraz, trukaketa kopuruak, gure perturbazio funtzioaren parametro bat izan beharko du[5].

```
> set.seed(1)
> perturbShuffle <- function(solution, ratio, ...) {
+   return(shuffle(permutation=solution, ratio=ratio))
+ }
```

Honekin guztiarekin ILS algoritmoa exekutatu dezakegu perturbazio maila ezberdinekin:

[5] `iteratedLocalSearch` funtzioarekin (eta paketearen beste hainbat funtziorekin) arazorik ez izateko, pasatutako funtzioak `...` argumentua izan behar du gutxienez, nahiz eta gero barruan ez erabili.



2.6 irudia. ILS algoritmoaren progresioa 14 hirien TSP problema batean. Ezkerretik eskuinera, perturbazioaren ratioak 0.01, 0.25 eta 1 dira.

```
> ratio <- 0.01
>
> args <- list()
> args$evaluate          <- burma.tsp$evaluate
> args$initial.solution  <- init.sol
> args$neighborhood      <- ngh
> args$selector          <- sel
> args$perturb           <- perturbShuffle
> args$ratio             <- ratio
> args$accept            <- th.accpet
> args$th                <- th
> args$num.restarts      <- r
> args$verbose           <- FALSE
>
> ils.1 <- do.call(iteratedLocalSearch, args)
>
> args$ratio <- 0.25
> ils.25 <- do.call(iteratedLocalSearch, args)
>
> args$ratio <- 1
> ils.100 <- do.call(iteratedLocalSearch, args)
>
> plotProgress(result=list(ILS1=ils.1, ILS25=ils.25,
+                          ILS100=ils.100)) +
+   facet_grid(. ~ Group, scales="free_x") +
+   aes(linetype=Group) + labs(y="Current Solution") +
+   thm.bh
```

Hiru bilaketa horien progresioak 2.6 irudian ikus daitezke. Hirurak soluzio berdinetik hasten dira, eta, pauso bakoitzean inguruko soluziorik onena aukeratzen dutenez, lehenengo jaitsiera berdina da hiru grafikoetan. Lehenen-

go optimo lokala topatzen den momentuan, ordea, diferentziak hasten dira. Ezkerretik eskuinera, lehenengo grafikoan soluzioak posizioen %1 trukatuz perturbatzen dira; guztira soluzioek 14 posizio dituztenez, trukaketa bakar bat egiten da. Erdiko grafikoan perturbazioa %25ekoa da; 3 trukaketa egiten dira, alegia. Azken grafikoan, berriz, 14 trukaketa egiten dira (posizioen %100). Perturbazio txiki bat erabiltzen dugunean, lortutako soluzioaren ebaluazioa optimo lokalaren antzerakoa da (agertzen diren jauziak ez dira oso altuak, alegia); zenbat eta perturbazio handiagoa orduan eta diferentzia handiagoa soluzio berriaren eta optimoaren artean. Azken kasuan, perturbazioa oso handia da, eta, beraz, optimo lokaletatik ateratzeko, soluzioak zori hutsez aukeratzen dira, hasieraketa anizkoitzeko algoritmoan bezala.

GRASP algoritmoa

Optimizazio-problemak ebazteko ohikoa da metodo eraikitzaileak erabiltzea. Aurreko kapituluan ikusi genuen bezala, algoritmo horiek soluzioa pausoz pauso eraikitzen dute, urrats bakoitzean aukera guztietatik onena hautatuz. Era horretan, soluzio onak sortzen dira, baina horiek ez dute zertan optimoak izan, ez globalki eta ezta lokalki ere. Hori dela eta, behin soluzioa sortuta, bilaketa lokal bat erabil daiteke soluzioa areagotzeko. Alabaina, berdinketak egon ezean, metodo eraikitzaileek instantzia bakoitzeko soluzio bakarra eta beti berdina lortzen dute, eta beraz hasieraketa bakarra ahalbidetzen dute.

Ideia hori apur bat landuz, metodo eraikitzaileak soluzio bakarra sortu beharrean soluzio multzo bat sortzeko egokitu ditzakegu. Eta, ondoren, 2.2 algoritmoan agertzen den *random_solution* metodoak multzo horretatik zorizko soluzioak aterako ditu, bilaketa lokala hasieratzeko. Ideia hori da GRASP *Greedy Randomized Adaptative Search Procedure* algoritmoaren atzean dagoena [14].

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

2.4 adibidea. *Demagun motxilaren problema ebatzi nahi dugula. Oso sinplea den algoritmo eraikitzaile bat ondorengoa da: lehenik eta behin, kalkulatu motxilan sartzen ditugun elementu bakoitzaren balioa/pisua ratioa, eta gero, pauso bakoitzean, pisu-muga gaindiarazi ez duten elementuetatik ratiorik handiena duena aukeratu.*
Algoritmo hori GRASP algoritmoaren ideiara modu errezean egokitu daiteke. Algoritmoaren iterazio bakoitzean hasierako soluzio bat eraikiko dugu pauso bakoitzean, ratiorik handiena duen elementua aukeratu beharrean ratio handiena duten $\%\alpha$ *soluzioen artetik bat zoriz aukeratuz. Behin soluzioa eraikita, bilaketa lokala aplikatuko dugu lortutako soluzioa areagotzeko.*

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

Adibide gisa, **metaheuR** paketean motxilaren problema GRASP algoritmoaren bitartez ebatzi ahal izateko, `graspKnapsack` funtzioa izango dugu. Funtzio horretan, hasteko, zenbait datu atera eta aldagai batzuk hasieratzen dira. Besteak beste, elementurik gabeko soluzio «hutsa»sortuko dugu.

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

Soluzioa sortzeko, elementuak banan-banan sartuko ditugu motxilan, eta, horretarako, begizta bat izango dugu, motxila beteta ez dagoen bitartean errepikatuko dena. Begiztaren lehenengo pausoan, motxilan oraindik sartu gabeko elementuei atzematen diegu, eta, uneko iterazioan, gure hautagai zerrendak izango duen tamaina kalkulatzen dugu (gogoratu hautagai zerrendaren tamaina urrats bakoitzean dugun aukera kopuruaren proportzio gisa definitu dugula).

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

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

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

Bukatzeko, aukeratutako elementua soluzioan sartu, eta aurrera jarraitu motxila beterik ez badago. Motxila bete egiten bada, sartutako azkeneko elementua atera, eta begizta bukatu egingo da.

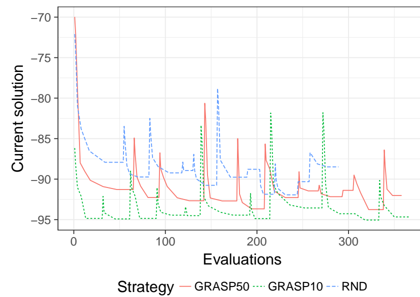
```
    aux <- solution
    aux[ratio == selected] <- TRUE
    if (sum(weight[aux]) < limit) {
      solution <- aux
    }else{
      finished <- TRUE
    }
  }
  return(solution)
}
```

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

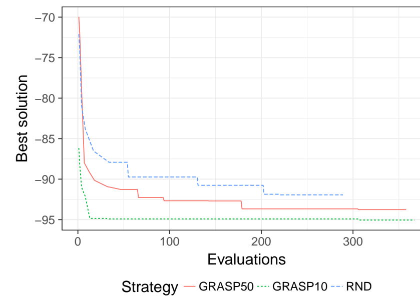
```
> n <- 50
> values  <- c(runif(n / 2) * 10,  runif(n / 2) * 25)
> weights <- values * rnorm(n, 1, 0.05)
> limit   <- sum(sample(weights, n / 5))
```

GRASP eta hasieraketa anizkoitzeko bilaketa lokala erabiltzeko funtzio berdina daukagu: `multistartLocalSearch`. Orain arte ikusitakoen antzerakoa da funtzio hori, baina argumentu moduan, hasierako soluzioa pasatu beharrean, soluzioak sortzeko funtzio bat pasatu behar diogu. Funtzioak parametro asko dituenez, sarrerako balioak antolatzeko beste era bat erabiliko dugu kodea irakurterrazagoa izateko. Zerrenda batean sartuko ditugu, izenak erabiliz, eta gero R-ren `do.call` funtzioa erabiliko dugu.

```
> knp.problem <- knapsackProblem(weights, values , limit)
>
> args$evaluate  <- knp.problem$evaluate
> args$valid     <- knp.problem$valid
> args$correct   <- knp.problem$correct
> args$non.valid <- "discard"
> args$verbose   <- FALSE
>
> args$neighborhood <- flipNeighborhood(base=rep(FALSE, n))
> args$selector     <- firstImprovementSelector
>
> args$generateSolution <- graspKnapsack
> args$num.restarts     <- 10
```



(a) Uneko soluzioa



(b) Soluziorik onena

2.7 irudia. Hasieraketa anizkoitzeko estrategien konparaketa, motxilaren problema batean. Ezkerreko grafikoan uneko soluzioaren progresioa ikus daiteke. Eskumakoan, berriz, uneko soluziorik onenaren progresioa erakusten da.

```
> args$weight          <- weights
> args$value           <- values
> args$limit           <- limit
```

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

```
> rnd.ls <- do.call(multistartLocalSearch, args)
>
> args$cl.size <- 0.50
> GRASP.50 <- do.call (multistartLocalSearch, args)
>
> args$cl.size <- 0.1
> GRASP.10 <- do.call (multistartLocalSearch, args)
```

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

Inguruneko soluzioen hautaketa

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

Soluzio «txarragoak» aukeratzeko bi estrategia daude. Lehendabizikoan, fitness-a hobetzen ez duen soluzio bat aukeratzean, sortutako «galera» kontuan hartzen da (era probabilistikoan zein deterministan egin daiteke). Algoritmorik ezagunena *suberaketa estokastikoa* –*simulated annealing* [23, 37] ingelesez– izenekoa da, zeinek probabilitate-banaketa parametriko bat erabiltzen duen aukeraketa egiteko. Algoritmo horretan inspiratutako beste hainbat algoritmo proposatu dira literaturan, *demon algorithm* [29] eta *threshold accepting* [13, 27] algoritmoak, besteak beste.

Soluzio txarrak aukeratzeko bigarren estrategia mota Gloverrek 1986an proposaturiko *tabu-bilaketa* [15] –*tabu search* ingelesez– algoritmoak erabiltzen duena da. Kasu horretan, fitness-balioa hobetzen ez duten soluzioak aukeratzen dira, baina bakarrik ingurune osoan helburua hobetzen duen soluziorik ez badago. Estrategia horren arriskua dagoeneko bisitatu ditugun soluzioak berriro bisitatzea da. Hala, zikloak saihesteko, tabu-bilaketak azken aldian bisitatutako soluzioak «memoria» batean gordetzen ditu.

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

Suberaketa simulatua

Metalezko tresnen edo piezen sorrera-prozesuan, metalek hainbat propietate gal ditzakete, beren kristal-egituran eragindako aldaketak direla eta. Propietate horiek berreskuratzeko metalurgian «suberaketa» prozesua erabiltzen da; metal-pieza behar adina berotzen da, gero astiro-astiro hozten uzteko. Tenperatura igotzean metalaren atomoen energia handitzen da, eta, hortaz, haien 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. Horrek metala gogortzen du, baina hauskorragoa bihurtzen du, aldi berean. Suberatzean, berriz, metala poliki-poliki hozten da, eta, ondorioz, molekulek astiro galtzen dute beren energia –hots, abiadura–. Hozketa-abiadura motelari esker, molekulak beren kristal-egituraren «kokapen optimora» joaten dira; hau da, energia minimoko kristal-egitura bat sortzen da.

1983an Kirkpatrick-ek [23] eta bi urte geroago Cerny-k [37], suberaketaren prozesuan inspiratuta, optimizazio-algoritmoak proposatu zituzten; Kirpatrick-ek bere algoritmoari *simmulated annealing*, suberaketa simulatua, izena eman zion eta horixe da gaur egun hedatuen dagoena.

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

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

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

Ekuaziotik ondorioztа daitekeenez, uneko soluzioa baino fitness okerragoa duen soluzio bat onartzeko probabilitatea tenperaturarekiko proportzionala da, eta energia diferentziarekiko alderantziz proportzionala.

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

Suberaketaren gakoa hozte-abiaduran datza, hau da, tenperaturaren eguneraketan. Izan ere, hasieran T balio handiak erabiliko ditugu, ia edozein soluzio onartu ahal izateko, eta gero, astiro-astiro, T txikiagotuko dugu, gelditzeko baldintza bete arte. 2.4 algoritmoan, suberaketa simulatuaren sasikodea ikus daiteke.

Suberaketa simulatuan oinarritutako algoritmoak diseinatzean lau alderdi hartu behar dira kontuan:

- **Hasierako tenperatura** - Altuegia bada, hasierako iterazioetan *random walk* (hau da, ausazko ibilbide bat) jarraituko dugu; baxuegia bada, berriz, bilaketa oinarrizko bilaketa lokala bihurtuko da. 2.8 irudian adibide bat ikus daiteke. $T = 20$ denean, nahiz eta fitness-en arteko diferentzia txikia izan, soluzioa onartzeko probabilitatea txikia da; $T = 2000$ denean ebaluazioen arteko diferentzia handia izan arren, oso probablea da soluzioak onartzea. $T = 200$ denean, berriz, optimo lokaletik atera gaitezke, probabilitate handiarekin, s_j aukeratuz, baina tarteko tenperatura horrekin oso soluzio txarrak onartzea zaila izango da.

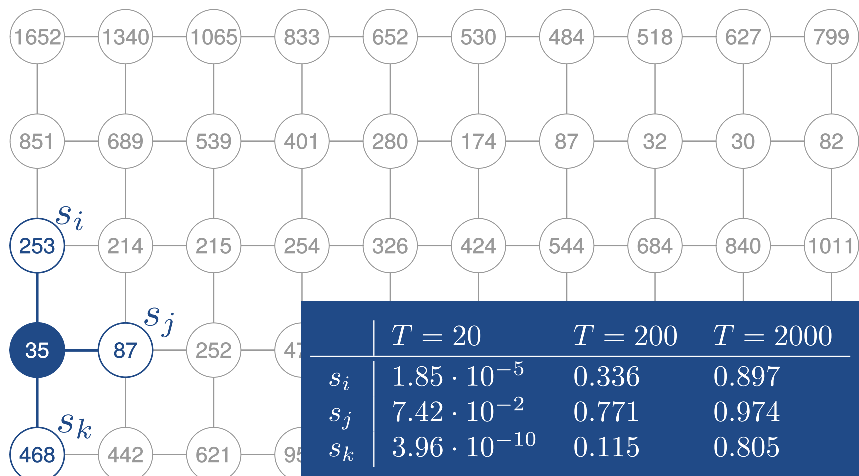
  Tenperatura hasieratzeko bi estrategia erabili ohi dira. Lehendabizikoa hasierako tenperatura oso altua aukeratzea da. Dibertsifikazioaren ikuspegitik interesgarria izan arren, estrategia honekin bilaketak asko luzatu daitezke, eta konputazionalki garestia izan daiteke. Bigarren estrategiaren funtsa bilaketa-espazioaren itxura aztertzean datza, ingurunean dauden soluzioen arteko diferentziak nolakoak diren jakiteko. Informazio hori

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
```

2.4 algoritmoa. Suberaketa simulatuaren sasikodea

onarpen-ratio edo probabilitate ezagun bat lortzeko behar dugun tenperatura finkatzeko erabil daiteke [20, 1], goi- eta behe-mugekin egin dugun antzera.



	$T = 20$	$T = 200$	$T = 2000$
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

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

2.5 adibidea. *Demagun 10 tamainako TSParen instantzia bat ebatzi nahi dugula. Kostu-matrizeko baliorik handiena –hau da, bi hirien arteko distantziarik handiena– 7.28 da. Problemarako soluzio guztietan 10 hiri izango ditugu, eta, hortaz, soluzio guztien ebaluazioa matrizean dauden 10 elementuren batura izango da. Hori dela eta, $f_g = 7.28 \cdot 10 = 78.2$ problema honen fitness-aren goi-muga bat da*[a]*. Era berean, matrizeko distantziarik txikiena 2.5 izanik, behe-muga kalkula dezakegu: $f_b = 2.5 \cdot 10$.*
Demagun, edozein soluzio aukeratzeko hasierako probabilitatea 0.75 dela. Orduan, bi muga hauek, f_g eta f_b, hasierako tenperatura kalkulatzeko erabil ditzakegu, edozein bi soluzioren arteko fitness-en diferentzia $f_g - f_b$ baino txikiagoa izango dela baitakigu:

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

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

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

Algoritmoaren erabilera erakusteko 2.2.2 ataleko adibidea erabiliko dugu (Bavariako hiriena).

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

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

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

## [1] 14760.21
```

- **Oreka lortzeko iterazio kopurua** - Tenperatura eguneratzen den bakoitzean, balio honekin zenbait iterazio egiten da –hau da, inguruneko soluzio batzuk aztertu behar dira– «oreka» lortu arte. Behin oreka lorturik, tenperatura berriro eguneratzen da. Lehenengo pausoa, beraz, oreka lortzeko behar dugun iterazio kopurua ezartzea da. Ohikoena ingurunekoo tamainaren araberako iterazio kopuru bat finkatzea da. Horretarako, ρ parametroa erabiliko dugu (0 eta 1 tartean hartuko ditu balioak), eta tenperatura bakoitzeko $\rho|N(s)|$ soluzio ebaluatuko ditugu. Aurreko atalean azaldu dugun bezala, $|N(s)|$-k uneko soluzioaren bizilagun kopurua adierazten du.
  Beste estrategia batzuek tenperatura bakoitzeko iterazio kopuru desberdina ezartzea proposatzen dute. Adibide gisa, tenperatura soluzio berri bat onartzen dugun bakoitzean alda dezakegu; soluzioen onarpena haien fitness-balioaren araberako probabilitate-balio baten menpekoa denez, batzuetan ebaluatzen dugun lehendabiziko soluzioa onartuko dugu eta, bestetan, hainbat soluzio probatu beharko ditugu bat onartu arte. Kasu horretan, beraz, iterazio kopurua aldakorra da.
- **Tenperatura jaitsieraren abiadura** - Hau da, ziurrenik, algoritmoaren elementurik garrantzitsuena. Hainbat formula erabil daitezke tenperatura eguneratzeko. Hona hemen batzuk:
  – *Lineala*: $T_i = T_0 - i\beta$, non T_i i. iterazioko tenperatura den. Eguneraketa mota honetan tenperatura beti positiboa izango dela kontrolatu behar dugu, ekuazioak tenperatura negatiboak itzul baititzake.

  - *Geometrikoa*: $T_i = \alpha T_{i-1}$. $\alpha \in (0,1)$ abiadura kontrolatzen duen parametroa da, eta ohikoena 0.5 eta 0.99 tarteko balio bat aukeratzea da.
  - *Logaritmikoa*: $T_i = \frac{T_0}{log(i)}$. Abiadura hau oso motela da eta, nahiz eta praktikan oso erabilgarria ez izan, interes teorikoa du suberaketa simulatuaren algoritmoaren konbergentzia demostratuta baitago ekuazio honekin.

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

Suberaketa simulatuaren erabilera erakusteko Bavariako TSParen adibidea lantzen jarraituko dugu. Algoritmoa `simulatedAnnealing` funtzioak inplementatzen du. Funtzio horrek, ohiko argumentuez gain, beste zenbait parametro berezi ditu:

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

Hasierako tenperatura kalkulatu dugu, baina ez amaierakoa. Aukera posible bat da zoriz soluzioak sortu eta haien fitnessen arteko diferentzien behemuga kalkulatzea.

---

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

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

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

## [1] 1
```

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

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

## [1] 0.4342945
```

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

```
> init.t

## [1] 14760.21

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

## [1] 3794.288
```

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

Informazio guztiarekin funtzioaren argumentuak pausoz pauso munta ditzakegu. Ingurunea definitzeko, ondoz ondoko trukaketak erabiliko ditugu, eta bilaketa zorizko soluzio batetik abiatuko dugu.

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

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

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

Eguneraketa-funtzioari tenperatura bat emanik, hurrengo tenperatura emango digu.

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

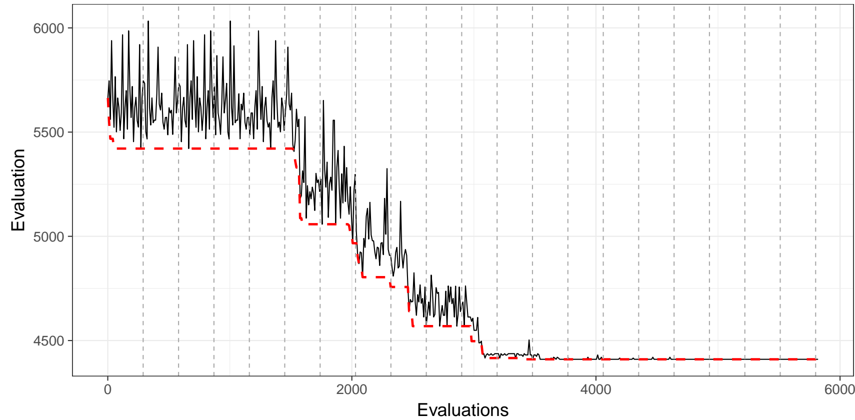
Oreka-egoera soluzio berrien ebaluazio kopuru baten arabera kontsideratuko dugu. Zehazki, 10 bider ingurunearen tamaina (hiri kopurua) izango da.

```
> args$log.frequency <- 10
> args$eq.criterion  <- "evaluations"
> args$verbose       <- FALSE
```

Amaitzeko, algoritmoa exekutatzen dugu.

```
> sa <- do.call(simulatedAnnealing, args)
```

Bilaketaren eboluzioa 2.9 irudian dago jasota. Irudiak uneko soluzioaren eta soluziorik onenaren progresioak erakusten ditu (beltzez eta gorriz, hurrenez hurren). Grafikoan marra bertikal bat agertzen denean, tenperatura-aldaketa bat egon dela esan nahi du. Uneko soluzioaren eboluzioan ikus daiteke ebaluazioa hasieran oso aldakorra dela. Hori tenperaturaren eraginaren ondorioz da, tenperatura altuak soluzio txarrak aukeratzeko probabilitatea handitzen baitu. Hozketa-prozesuaren zehar, tenperatura jaisten den heinean, soluzio txarrak onartzeko probabilitatea txikitzen da, eta, ondorioz, soluzioen



2.9 irudia. Suberaketa simulatuaren algoritmoaren progresioa TSP problema batean. Marra jarraikiak uneko soluzioaren progresioa adierazten du, eta etenak, berriz, aurkitutako soluziorik onena. Marra bertikalek tenperaturaren aldaketak erakusten dituzte.

arteko aldakortasuna murrizten. Amaieran, tenperatura oso txikia denean, ia ezinezkoa da soluzio okerragoak onartzea eta, hortaz, uneko soluzioa ez da ia aldatzen.

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

Ikusitako algoritmoetan soluzioen onarpena probabilistikoa da; alabaina, suberaketa simulatuaren kontzeptua era deterministan ere inplementa daiteke. Horren adibidea da Deabru Algoritmoa [29] –*Demon Algorithm*, ingelesez–. Hasiera batean Creutz-ek simulazio molekularrak egiteko proposatu zuen algoritmo hori, baina optimizazio-problemak ebazteko ere egoki daiteke.

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

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

Tabu-Bilaketa

```
1  input: intensify eta diversify operadoreak
2  input: intensify_condition, diversify_condition eta stop_condition baldintzak
3  input: N ingurune operadorea eta s0 hasierako soluzioa
4  output: s* topatutako soluziorik onena
5  s* = s0
6  s = s0
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
```

2.5 algoritmoa. Tabu-bilaketaren sasikodea

horren abantaila sinpletasuna da, Boltzmann distribuzioa ebaluatzeko beharrik ez baitago.

Tabu-bilaketa

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

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

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

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

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

2.6 adibidea. *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 horretan, tabu-zerrendan trukatzen ditugun bi posizioak gorde ditzakegu, alderantzizko trukaketa tabu bihurtuz. Esate baterako, uneko soluzioa* 13245 *bada eta* 31245 *soluziora mugitzen bagara, hurrengo urratsetan lehenengo eta bigarren posizioak trukatzea debekatua izango dugu. Problemaren arabera, beste irizpide batzuk erabil genitzake. Adibide gisa, lehenengo posizioan* 1*a eta bigarrenean* 3*a egotea debeka genezake.*

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

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

Tabu-zerrendaz gain, bestelako aukera konplexuagoak ere proposatu dira. Epe motzeko memoria erabiltzeaz gain, bilaketa-prozesuan zehar jasotako informazioa ere oso baliotsua izan daiteke algoritmoa gidatzeko. Informazio hori

epe erdiko edota epe luzeko memorian gorde daiteke. Lehendabiziko kasuan, soluzio onenen informazioa bakarrik gordeko dugu, bilaketa areagotzeko asmoarekin. Bigarren kasuan, berriz, bilaketa osoan zehar soluzioen osagaien maiztasunak gordeko ditugu; maiztasun horiek bisitatu ez ditugun eremuei antzemateko erabil daitezke –hau da, bilaketa dibertsifikatzeko–.

2.7 adibidea. *TSP-rako soluzioak eraikitzeko, hiri bakoitzetik zer hiritara mugituko garen erabaki behar dugu. Algoritmo eraikitzaile tipikoan, erabaki hori kostu-matrizeari begiratuz hartzen da, uneko hiritik bisitatu gabeko hirietatik gertuen dagoena aukeratuz. Era berean, epe erdiko eta epe luzeko memoriak matrize karratu batean inplementa ditzakegu. Matrize horietan, gordeko dugu bisitatutako zenbat soluzietan joaten garen i hiritik j hirira. Epe ertaineko 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 horretan, bilaketa esploratu gabeko eremuetara eraman nahi badugu, gutxien erabilitako loturak erabiliz soluzioak sor ditzakegu, bilaketa-prozesua bertatik abiatzeko.*

Optimizazio-problemen «itxuraaldaketa»

Bilaketa lokalean uneko soluziotik honen inguruan dagoen soluzio batera mugitzen gara beti; hau da, soluzio bakoitzetik soluzio kopuru mugatu batetara mugi gaitezke soilik. Hori dela eta, bilaketa-espazioa grafo baten bidez adieraz daiteke, non erpinak soluzioak diren eta ertzek mugimendu posibleak adierazten dituzten; 2.2 irudiak horrelako grafo bat erakusten 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. Zentzu horretan, kontuan hartu behar da problema motaren arabera ez ezik, *landscape*a instantzia konkretuaren arabera ere aldatzen dela. Optimizazio-problemen "itxuraren"ideia intuiziozkoa gailurrez, mendikatez eta bailaraz osatutako paisaia baten ilustrazioa da, non gailurrek optimo lokalak errepresentatzen dituzten eta gailurrik altuena optimo globala den. Gailur baten erakarpen-arroa mendi osoa da, oinarritik gailurrera. Intuitiboki, zenbat eta gailur gehiago, orduan eta zailagoa izango da instantzia baten optimo globalera heltzea bilaketa lokalean oinarritutako algoritmo batekin.

Soluzio bakarrean oinarritzen diren algoritmoekin amaitzeko, bilaketa-prozesuan zehar, *landscape*a eraldatzen dituzten algoritmoak aztertuko ditugu. Zehazki, bi algoritmo ikusiko ditugu. Lehenengoak, VNSak, ingurune

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
```

2.6 algoritmoa. VND algoritmoaren sasikodea

definizio ezberdinak erabiltzen ditu optimo lokaletatik ateratzeko. Bigarrenak, berriz, helburu-funtzio berriak sortzen ditu optimo lokalen kopurua murrizteko.

Variable Neighborhood Search algoritmoa

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

Ideia hori *Variable Neighborhood Descent* (VND) algoritmoan erabiltzen da, bilaketa lokala optimo lokaletan trabaturik geratzea ekiditeko. 2.6 algoritmoan VNDaren sasikodea ikus daiteke.

Algoritmoan ikusten den bezala, VNDan ingurune-funtzio bakarra izan beharrean horien multzo bat dago. Lehenengo ingurunea erabiliz, uneko soluzioaren ingurunea arakatu eta soluzio onena aukeratuko dugu. Inguruneko soluzio guztiak okerragoak direnean –topatutako soluzioa uneko ingurunera-

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 zoriz 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
```

2.7 algoritmoa. VNS algoritmoaren sasikodea

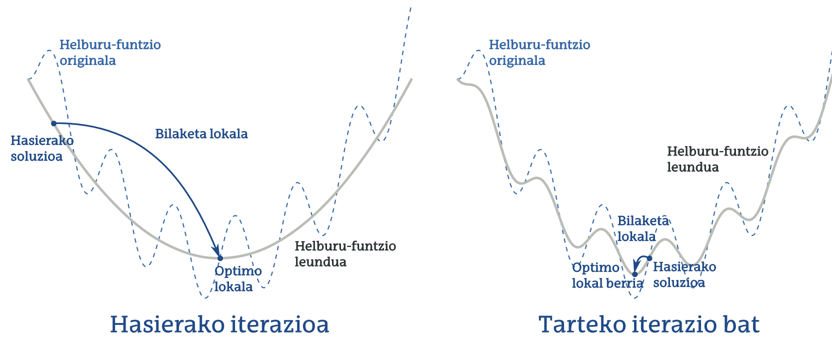
ko optimo lokala bada, alegia– hurrengo ingurune-definiziora pasatuko gara; horrela, ingurune-funtzio guztiak erabili arte. Gainera, iterazio bakoitzean, inguruneko soluzio berri batera pasatzen garenean, berriro ere hasierako ingurune-definiziora itzuliko gara.

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

Variable Neighborhood Search algoritmoa VNDaren hedapen bat da, non iterazio bakoitzean, uneko ingurune definizioa erabiliz, bilaketa lokala amaiera arte eramaten den. Hau da, fitnessa hobetzen duen soluzio bat topatu arren, uneko ingurune-definizioa mantentzen dugu optimo lokal batera heldu arte. Behin optimo lokal batera heldutakoan, hurrengo ingurune-definizioa erabiltzera pasatuko gara, VNDan legez, lortutako soluzioa ingurune guztietarako optimo lokala izan arte. 2.7 algoritmoak VNSaren sasikodea erakusten du.

Smoothing algoritmoak

Optimizatu behar dugun funtzioak optimo lokal asko dituenean, bilaketa lokalak ez dira oso metodo egokiak, globala ez den optimo batean trabaturik gelditzeko probabilitatea oso altua delako. Leuntze-metodoekin –*smoothing methods*, ingelesez– iterazio bakoitzean jatorrizko helburu-funtzioa eraldatu –leundu– egiten da, optimo lokalen kopurua gutxitzeko asmoz; eta helburu-funtzio berria erabiliz, bilaketa lokala aplikatzen da. Bilaketa trabaturik ge-



2.10 irudia. *Smoothing* algoritmoaren funtzionamendua. Iterazio bakoitzean hasierako helburu-funtzioa maila bateraino leuntzen da, eta bilaketa lokala aplikatzen da.

Oinarrizko *smoothing* algoritmoaren 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 $s^* = s$; $\alpha = \alpha_0$
6 **while** $\alpha > 1$ **do**
7 $\quad f' = smoothing(f;\alpha)$
8 $\quad s^* = local_search(s^*, f')$
9 $\quad \alpha = update(\alpha)$
10 **done**

2.8 algoritmoa. *Smoothing* algoritmoaren sasikodea

ratzen denean –optimo lokal batean–, helburu-funtzioa berriro aldatzen da, aurreko iterazioan baino gutxiago leunduz. Helburu-funtzio berri horrekin eta aurreko iterazioan lortutako optimoarekin, bilaketa lokala aplikatzen da, eta horrela optimo berri bat lortzen da.

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

Helburu-funtzioa nola leundu problemaren araberakoa da. Edonola ere, kasu guztietan, algoritmoa inplementatu ahal izateko leuntze-parametro bat definitu beharko dugu. Parametro hori handia denean, helburu-funtzioa asko leunduko dugu; parametroa 1 denean, berriz, helburu-funtzioa ez da batere aldatuko. Hori aintzat hartuz, 2.8 algoritmoan metodoaren sasikodea definituta dago. Ikus dezagun adibide bat.

2.8 adibidea. *TSPan helburu-funtzioa kalkulatzeko distantzien matrizea erabiltzen dugu. Matrize horretan edozein bi hiriren arteko distantzia dago jasota. Helburu-funtzioa leuntzeko, matrizea hori eralda daiteke, distantzia guztiak batez besteko distantziara hurbilduz, adibidez. Demagun ondoko matrizea definitzen dugula:*

$$d_{ij}(\alpha) = \begin{cases} \bar{d} + (d_{ij} - \bar{d})^{\alpha} & \textit{baldin eta } d_{ij} \geq \bar{d} \\ \bar{d} - (\bar{d} - d_{ij})^{\alpha} & \textit{baldin eta } d_{ij} < \bar{d} \end{cases} \tag{2.6}$$

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

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

[a] Kontuan hartu behar da, matrizea normalizatuta ere, soluzio optimoa (hau da, balio minimoa duena), ez dela aldatzen.

3 Populazioetan oinarritutako algoritmoak

Aurreko kapituluan soluzio bakarrean oinarritzen diren zenbait algoritmo ikusi ditugu. Algoritmo hauek oso portaera ezberdina izan arren, badute ezaugarri komun bat: bilaketa-prozesuan zehar, une oro, soluzio bakar bat dute, eta operadore desberdinak erabiliz soluzio batetik bestera mugitzen dira. Hori dela eta, algoritmo hauek oso egokiak dira bilaketa-espazioaren eskualde zehatzak modu exhaustiboan arakatzeko –bilaketa areagotzeko, alegia–. Optimizazio-prozedura honek baditu, ordea, bere eragozpenak, izan ere, ez ditu bilaketa-espazioko eskualde bananduak bisitatzen. Horregatik kasu gehienetan bilaketaren dibertsifikazioa bultzatzea beharrezkoa izaten da. Horren adibide dira bilaketa lokalean oinarritzen diren algoritmo batzuk dibertsifikatzeko erabiltzen dituzten zenbait estrategia. Adibidez, tabu-bilaketaren epe-luzeko memoria.

Kapitulu honetan, soluzio bakarrean oinarritutako algoritmoak alde batera utzi, eta soluzio multzoak erabiltzeari ekingo diogu, hori baita, hain justu, populazioetan oinarritzen diren algoritmoen filosofia. Algoritmo horiek, pauso bakoitzean soluzio bakar bat izan beharrean, soluzio multzo baten gainean egiten dute lan. Testuinguru batzuetan soluzio multzo horri *soluzio-populazio* deritzo eta, hortik, algoritmo hauen izena. Populazioetan oinarritutako algoritmoetan, bilaketa-prozesuan zehar, soluzio multzo hori aldatuz joango da helburu-funtzioaren gidaritzapean, gelditze-irizpide bat bete arte.

Oro har, populazioan oinarritutako algoritmoak bi multzotan banatu ditzakegu: algoritmo ebolutiboak eta *swarm intelligence*an oinarritutakoak. Lehenengo kategoriako algoritmoek, teknika desberdinak erabiliz, populazioa eboluzionatzen dute, horrek geroz eta soluzio hobeak izan ditzan. Adibiderik ezagunenak algoritmo genetikoak dira. Bigarren motako algoritmoak, berriz, zenbait animaliak duten portaera kolektiboan oinarritzen dira. Hori adibide-

rik ezagunena inurri-kolonien algoritmoak dira. Algoritmo mota honek imitatzen du inurriek janariaren eta inurritegiaren arteko distantziarik motzena topatzeko darabilten mekanismoa.

Kapitulua bi zatitan banaturik dago, bakoitza populazioan oinarritutako algoritmo mota bati eskainia. Lehenengo zatian, algoritmo ebolutiboen eskema orokorra ikusi ondoren, algoritmo genetikoak [19] eta EDAk (*Estimation of Distribution Algorithms*) [24, 26] aurkeztuko dira. Bigarren zatian, *swarm intelligence* [6] arloan proposaturiko *Ant Colony Optimization* eta *Particle Swarm Optimization* algoritmoak aztertuko dira.

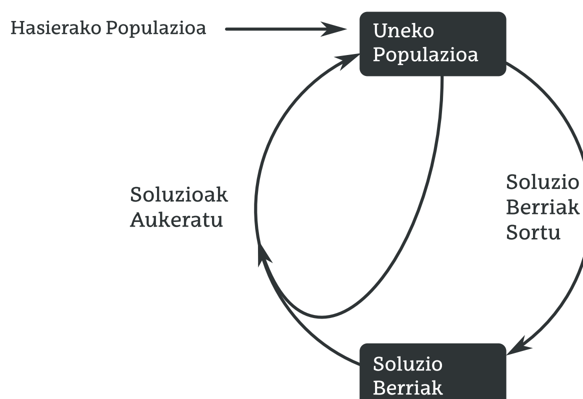
Algoritmo ebolutiboak

1859. urtean Charles R. Darwinek *On the Origin of the Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life* liburua argitaratu zuen. Tituluak berak adierazten duen bezala, liburu horretan Darwinek hautespen naturalaren teoria aurkeztu zuen.

Eboluzioaren teoriak dioenez, belaunalditik belaunaldira zenbait mekanismoren bidez –mutazioak, esate baterako– aldaketak gertatzen dira espezieetan. Aldaketa horietako batzuei esker indibiduoak hobeto egokitzen dira haien ingurunera, eta, ondorioz, bizirik mantentzeko eta, batez ere, ugaltzeko probabilitateak handitzen dira. Era berean, noski, aldaketa batzuk kaltegarriak izan daitezke, bizitzeko aukerak murrizten baitituzte. Kontuan hartuz aipatutako aldaketak heredatu egiten direla, ezaugarri onak belaunaldiz belaunaldi mantentzen dira; kaltegarriak direnek, ordera, galtzeko joera izaten dute. Prozesu horren bidez, espezieak haien ingurunera geroz eta hobeto egokitzeko gai dira.

Hirurogeiko hamarkadan, ikertzaileek Darwinen lana inspiraziotzat hartu zuten optimizazio metaheuristikoak diseinatzeko, eta, geroztik, konputazio ebolutiboa konputazio zientzien arlo bereizia bilakatu da. Atal honetan bi algoritmo mota aztertuko ditugu, algoritmo genetiko klasikoak [19] eta EDAk (*Estimation of Distribution Algorithms*) [24, 26].

Diferentziak diferentzia, algoritmo ebolutibo guztiek 3.1 irudiko eskema orokorra betetzen dute. Eskema orokor horretan, bi elementu dira giltzarriak: soluzio berrien sorkuntza eta soluzioen hautespena. Algoritmoaren abia-puntua hasierako populazioa izango da; populazio horretatik hasiz, algoritmoa begizta nagusian sartzen da, non bi pauso txandakatzen diren. Lehenik, uneko populazioko soluzioetatik abiatuz, soluzio multzo berri bat sortuko da. Ondoren, soluzio berri horiek eta uneko populazioko soluzioak kontuan hartuz, naturan bezala, hurrengo belaunaldiara pasatzeko soluzio onak aukeratuko ditugu, eta populazio berri bat sortuko dugu. Optimizazioa bukatutzat emango da, konbergentzia edo iterazio kopuruari loturiko irizpide konkretu batzuk betetzen direnean.



3.1 irudia. Algoritmo ebolutiboen eskema orokorra

Hurrengo atalean, algoritmo orokor honen urratsak sakonago aztertuko ditugu. Hasteko, algoritmo ebolutibo guztietan komunak diren pausoak azalduko ditugu, eta, aurrerago, bi algoritmo ezberdinen xehetasunetan jarriko dugu arreta.

Urrats orokorrak

Esan bezala, atal honetan algoritmo ebolutibo guztiek komunean dituzten urratsak azalduko dira banan banan, **metaheuR** paketeko funtzioen adibideekin lagundurik.

Populazioaren hasieraketa

Nahiz eta askotan garrantzi gutxi eman, hasierako populazioa da algoritmoaren abiapuntua eta, hortaz, beraren sorkuntza oso pauso garrantzitsua da, eragin handia izaten baitu lortutako azken emaitzean.

Algoritmoen xedea soluzio onak topatzea denez, pentsa dezakegu hasierako populazio on batek soluzio onez osatuta egon behar duela; alabaina, soluzioen dibertsitatea haien kalitatea bezain garrantzitsua da. Populazioa antzekoak diren soluzioez osatuta badago, orduan horren eboluzioa oso zaila izango da, eta algoritmoak azkarregi konbergitu dezake optimoa ez den soluzio batera.

Hortaz, hasierako populazioa sortzean bi alderdi izan behar ditugu kontuan: kalitatea eta dibertsitatea. Kasu gehienetan zorizko hasieraketa erabiltzen da lehen populazioa sortzeko; hau da, zorizko soluzioak sortzen dira

populazioa osatu arte. Estrategia hori erabiliz dibertsitate handiko populazioa sortuko dugu, baina kalitatea ez da handia izango.

Zorizko laginketak lortutako dibertsitatea baino handiagoa bermatu nahi bada, "sasizorizko"deritzen prozedurak existitzen dira. Metodo horiek, populazioko soluzioen dibertsitatea maximizatzeko erabiltzen dira. Horren adibide da dibertsifikazio sekuentziala. Algoritmo honek, soluzioak banan-banan sartzen ditu populaziora, baldin eta soluzioa populazioko guztien distantzia minimo batera badago. Adibide moduan, demagun 25 tamainako bektore bitarren 10 soluzioko populazio bat sortu nahi dugula. Dibertsitatea bermatzeko populazioko soluzioen arteko Hamming distantzia minimoak 10 izan behar duela ezarriko dugu.

Jarraian dagoen kodeak horrelako populazioak sortzen ditu. Lehenik, Hamming distantzia neurtzeko eta zorizko bektore bitarrak sortzeko funtzioak definituko ditugu:

```
> hammDistance <- function (v1, v2) {
+   d <- sum(v1 != v2)
+   return(d)
+ }
>
> createRndBinary <- function(n) {
+   return (runif(n) > 0.5)
+ }
```

Gero, soluzioak zoriz sortzen ditugu, eta, distantzia minimoko baldintza bete ezean, deuseztatu egiten ditugu; prozedura errepikatu egingo da nahi dugun soluzio kopurua lortu arte.

```
> sol.size <- 25
> pop.size <- 10
> min.distance <- 10
> population <- list(createRndBinary(sol.size))
> while (length(population) < pop.size) {
+   new.sol <- createRndBinary(sol.size)
+   distances <- lapply(population,
+                       FUN=function(x) {
+                         return(hammDistance (x, new.sol))
+                       })
+   if (min(unlist(distances)) <= min.distance) {
+     population[[length(population) + 1]] <- new.sol
+   }
+ }
```

Zenbait kasutan, prozedura hau ez da batere eraginkorra, zenbait kasutan soluzio asko aztertu behar izaten baitira populazioa osatu arte. Eragozpen horri aurre egiteko dibertsifikazio paraleloa proposatu zen. Teknika horrek bilaketa-espazioa zatitu egiten du eskualde bakoitzetik zorizko soluzio bat erauziz. Kontuan hartu behar da azken teknika hau ezin dela beti aplikatu. Bilaketa-espazioa bitarra denean, ez dago eragozpen nabarmenik; bilaketa-espazioa permutazioz osaturikoa denean ordea, eskualdeak bilatzea ez da tri-

biala. Soluzioen kalitateari dagokionez, askotan dibertsifikazio-teknikek kalitate oneko soluzioak sortzea galarazten dute. Horretarako, hasieraketa heuristikoak erabiltzea izaten da ohikoena. Era sinple bat da GRASP algoritmoetan zorizko soluzioak sortzeko erabiltzen diren prozedurak erabiltzea. Ondoko lerroetan Bavariako hirien TSP problemarako adibide bat ikus dezakegu. Lehenik, problema kargatuko dugu.

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

Orain, `tspGreedy` funtzioan oinarrituta, zorizko soluzio onak sortzeko funtzio bat definitzen dugu.

```
> createRndSolution <- function(cl.size=5) {
+   tspGreedy(cmatrix=cost.matrix, cl.size=cl.size)
+ }
```

Aurreko kapituluetan azaldu bezala, `tspGreedy` funtzioak TSPrako algoritmo eraikitzaile bat inplementatzen du; pauso bakoitzean, uneko hiritik zein hiritara mugituko garen erabakitzen da, gertuen dauden `cl.size` hirietatik –5, gure kasuan– bat zoriz aukeratuz. Horretan oinarrituz, populazioa sortzeko funtzio hau erabiliko dugu.

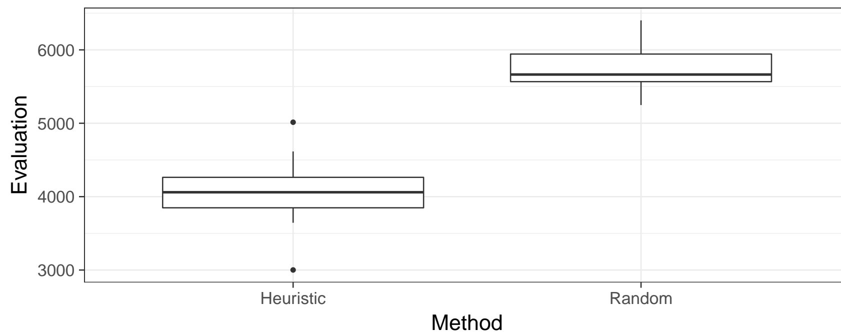
```
> pop.size <- 25
> population <- lapply(1:pop.size,
+                      FUN=function(x) {
+                        return(createRndSolution())
+                      })
```

Hautagaien zerrendaren tamainari (`cl.size`) problemaren tamainaren balioa ezartzen badiogu, pauso bakoitzean, aukera guztietatik bat zoriz hartuko dugu; hots, guztiz zorizkoak diren soluzioak sortuko ditugu. Azken aukera horrekin, populazioaren kalitatea goiko kodearekin lortutakoa baino okerragoa izango da:

```
> rnd.population <- lapply (1:pop.size,
+                           FUN=function(x) {
+                             cls <- ncol(cost.matrix)
+                             sol <- createRndSolution(cl.size=cls)
+                             return(sol)
+                           })
> tsp <- tspProblem(cost.matrix)
> eval.heur <- unlist(lapply(population, FUN=tsp$evaluate))
> eval.rnd <- unlist(lapply(rnd.population, FUN=tsp$evaluate))
```

Bi populazioen ebaluazioak *boxplot* baten bidez aldera ditzakegu:

```
> df <- rbind(data.frame(Method="Heuristic",
+                        Evaluation=eval.heur),
+             data.frame(Method="Random",
+                        Evaluation=eval.rnd))
```



3.2 irudia. Zorizko hasieraketaren eta hasieraketa heuristikoaren arteko konparaketa. Y ardatzak metodo bakoitzarekin sortutako soluzioen *fitness*a adierazten du.

```
> ggplot(df, aes(x=Method, y=Evaluation)) +
+   geom_boxplot() + thm.bh
```

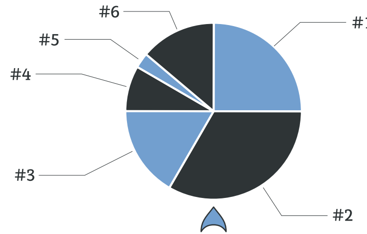
3.2 irudiak lortutako emaitzak erakusten ditu. Helburua minimizazioa dela kontuan hartuta, argi eta garbi ikus daiteke heuristikoa erabiliz sortutako soluzioak hobeak direla.

Soluzioak sortzeko metodoak ez ezik, populazioaren tamainak ere badu eragin handia azken emaitzan, eta hori ere egokitu beharreko parametro garrantzitsua da. Algoritmoak darabiltzan populazioak txikiegiak badira, dibertsitatea mantentzea oso zaila izango da, eta, hortaz, belaunaldi gutxitan algoritmoak konbergitu egingo du, ziurrenik optimoa ez den soluzio batera. Bestalde, populazioak handiegiak badira, konbergentzia-abiadura motelagoa izango da, eta, ondorioz, kostu konputazionala ere handiagoa bilakatuko da (3.5 irudian portaera honen adibide bat ilustratzen da). Horrenbestez, ez dago irizpide finkorik populazioen tamaina ezartzeko, eta problema bakoitzerako balio egoki bat bilatu beharko da. Edonola ere, irizpide orokor gisa esan dezakegu populazioak azkar konbergitzen badu –hots, soluzioen arteko distantzia azkar txikitzen bada–, soluzio hobeak lortzeko modua populazioaren tamaina handitzea izan daitekeela.

Hautespena

Algoritmo ebolutiboetan soluzioen hautespena izango da seguraski urratsik garrantzisuena, horrek kontrolatzen baitu populazioaren eboluzioa. Orokorrean, populazioan dauden soluziorik onenak hautatzea da gehien erabiltzen den hautespen-irizpidea: hautespen «elitista» Ȧlabaina, soluzio onak aukeratzea garrantzitsua bada ere, dibertsitatea mantentzearren, tarteka soluzio ez hain onak sartzea ere komenigarria izaten da. Teknika hori zuzenean aplikatu

Indibiduo	Ebaluzaioa
#1	899
#2	1204
#3	598
#4	313
#5	95
#6	500



3.3 irudia. Erruleta-hautespena. Indibiduo bakoitzaren erruletaren zatia bere ebaluazioarekiko proportzionala da. Erruletari eragiten zaion bakoitzean indibiduo bat aukeratzen da, bere *fitness*arekiko proportzionala den probabilitatearekin. Adibidean, 2. indibiduoa da hautatu dena.

daitekeen arren, badaude aukeraketa-metodo egokiago batzuk soluzio txarrak estrategia probabilistikoak erabiliz aukeratzen dituztenak.

Erruleta-hautespena, (*Roulette Wheel selection*, ingelesez) deritzon estrategian soluzioak erruleta batean kokatzen dira; soluzio bakoitzari bere ebaluazioarekiko proportzionala den erruletaren zati bat esleituko zaio. Hori horrela, 3.3 irudian ikus daitekeen bezala, erruleta jaurtitzen den bakoitzean indibiduo bat hautatzen da. Hautatua izateko probabilitatea erruleta zatiaren tamainarekiko eta, hortaz, indibiduoen ebaluazioarekiko proportzionala da. Indibiduo bat baino gehiago aukeratu behar baldin baditugu, behar adina erruleta-jaurtiketa egin ditzakegu.

Azkenik, esan beharra dago *fitness*aren magnitudea problemaren eta, batez ere, instantzien araberakoa dela. Hori dela eta, erruleta banatzeko probabilitateak zuzenean helburu-funtzioaren balioak erabiliz kalkulatzen badira, oso distribuzio erradikalak izan ditzakegu. Arazo hori ekiditeko, helburu-funtzioaren balioa zuzenean erabili beharrean soluzioen rankinga erabil daiteke.

Beste hautespen probabilistiko mota bat lehiaketa-hautespena da. Estrategia horrekin soluzioen aukeraketa bi pausotan egiten da. Lehenengo urratsean indibiduo guztietatik azpimultzo bat aukeratzen da, guztiz zoriz (ebaluazioa kontuan hartu barik). Ondoren, azpimultzo horretatik soluziorik onena hautatzen dugu. Azpimultzoen eraketa zori hutsez egiten denez, horietako batzuk oso soluzio txarrez osatuta egon daitezke. Kasu horietan, nahiz eta onena aukeratu, populazio berrirako gordeko dugun soluzioa ez da ona izango, eta, horrenbestez, soluzio on eta txarren aukeraketa ahalbidetzen du hautespen metodo honek.

Gelditze-irizpideak

Lehen apiatu bezala, algoritmo ebolutiboen begizta nagusia amaigabea da, eta, beraz, gelditzeko irizpideren bat ezarri behar dugu, bilaketa gelditzeko. Hurbilketarik sinpleena irizpide estatikoak erabiltzea da, hala nola bilaketarako denbora maximoa ezartzea, ebaluazioak mugatzea, etab.

Irizpide estatikoez gain, eboluzioaren prozesuari erreparatzen dioten irizpide dinamikoak ere erabil daitezke. Belaunaldiz belaunaldi populazioan dauden soluzioak geroz eta hobeak dira, eta, aldi berean, populazioaren dibertsitatea murrizten da, soluzio batera konbergitzeko joera izanik. Gauzak horrela, populazioaren dibertsitatea gelditze-irizpideak eraikitzeko ere erabili ohi da.

Dibertsitatea soluzioei zein haien *fitness*ari erreparatuz neur daiteke. Esate baterako, soluzioen arteko distantzia neurtzerik badago, indibiduoen arteko batez besteko distantzia minimo bat ezar dezakegu gelditze-irizpide gisa.

Algoritmo genetikoak

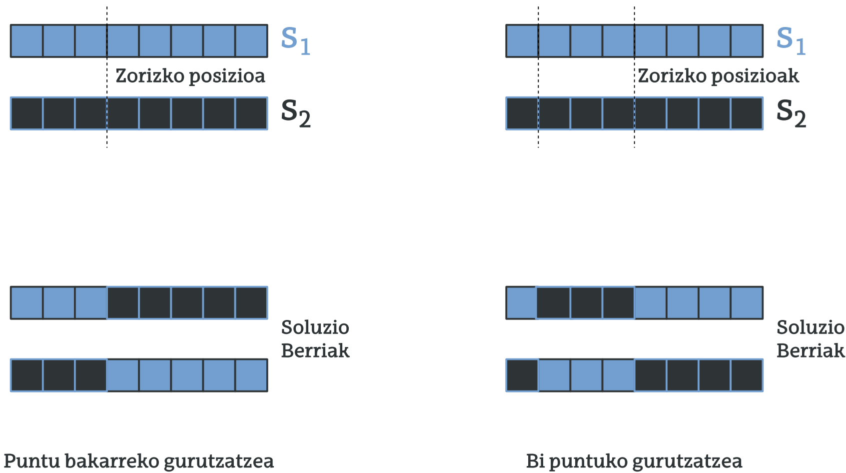
Algoritmo genetikoak [19] algoritmo ebolutiboen adibide ezagunenak eta erreferentziatuenak dira. Naturan gertatzen den espezieen eboluzioa imitatuz, algoritmo genetikoen osagaiak naturako fenomeno horren harira definitzen dira:

- Espezie bateko indibiduoak = Problemaren soluzioak
- Indibiduoen egokitasuna (*fitness*a, alegia) = Soluzioaren ebaluazioa
- Espeziearen populazioa = Soluzio multzoa/populazioa
- Ugalketa = Soluzio berrien sorkuntza

Algoritmo genetikoetan soluzio berriak sortzeko estrategiak diseinatzean naturako indibiduen ugalketa-prozesuan oinarrituko gara. Ugalketa prozesuaren xedea zenbait indibiduo emanda –bi, gehienetan–, indibiduo berriak sortzea da. Ohikoena prozesu hori bi pausotan banatzea da: soluzioak gurutzatzea eta mutatzea. Lehenaren helburua *guraso*-soluzioek dituzten ezaugarriak (geneak) soluzio berriei pasatzea da, espezieen gurutzaketan jasotzen den bezala. Bigarrenarena, berriz, sortutako soluzio berriei, *semeei*, ezaugarri berriak eranstea da. Jarraian soluzio berriak sortzeko bi operadore horiek aztertuko ditugu.

Gurutzaketa

Bi soluzio –edo gehiago– gurutzatzen ditugunean euren propietateak sortutako soluzio berriei transmititzea da helburua. Optimizazio arloan, soluzioen arteko gurutzaketak *gurutzaketa-operadore*-en –*crossover*, ingelesez– bidez



3.4 irudia. Gurutzaketa-operadoreak bektoreen bidezko kodeketarekin erabiltzeko

egiten dira. Operadore horiek soluzioen kodeketarekin dihardute, eta, beraz, gurutzaketa operadore zehatz bat hautatzean soluzioak nola adieratzen ditugun aintzat hartu beharko dugu.

Badaude kodeketa klasikoekin erabil daitezkeen zenbait oinarrizko gurutzaketa-operadore. Ezagunena puntu bakarreko gurutzaketa –*one-point crossover*, ingelesez– deritzona da. Demagun soluzioak bektoreen bidez kodetzen ditugula. Bi soluzio/guraso, s_1 eta s_2 izanik, operadore horrek bi soluzio berri/seme sortzen ditu (3.4 irudian operadore horien adibide bat ilustratzen da). Horretarako, lehenik eta behin, zorizko posizio bat, i, aukeratu behar da. Hau egin ahala, lehenengo soluzio berria s_1 soluziotik lehenengo i elementuak eta s_2 soluziotik gainontzekoak ($i+1$-tik aurrerakoak) kopiatuz sortuko dugu. Era berean, bigarren soluzio berria s_2-tik lehenengo i elementuak eta s_1-etik $i+1$ posiziotik aurrerako elementuak kopiatuz sortuko dugu. 3.4 irudiaren ezkerraldean, puntu bakarreko gurutzaketa (*one-point crossover*) operazioaren aplikazioaren adibide bat ikus daiteke. Horrez gain, eskuinaldean operadore hori nola orokortu daitekeen erakusten da, puntu bakar bat erabili beharrean bi, hiru, etab. puntu erabiliz.

Azken operadore orokorrago horri *k-point crossover* deritzo eta **metaheuR** liburutegiko `kPointCrossover` funtzioan dago inplementaturik. Ikus ditzagun beraren erabileraren adibide batzuk:

```
> A.sol <- rep("A", 10)
> B.sol <- rep("B", 10)
> A.sol

## [1] "A" "A" "A" "A" "A" "A" "A" "A" "A" "A"

> B.sol
```

```
##  [1] "B" "B" "B" "B" "B" "B" "B" "B" "B" "B"

> kPointCrossover(A.sol, B.sol, 1)

## [[1]]
##  [1] "A" "A" "A" "A" "A" "A" "A" "A" "B" "B"
##
## [[2]]
##  [1] "B" "B" "B" "B" "B" "B" "B" "B" "A" "A"

> kPointCrossover(A.sol, B.sol, 5)

## [[1]]
##  [1] "A" "A" "A" "B" "A" "A" "A" "B" "A" "B"
##
## [[2]]
##  [1] "B" "B" "B" "A" "B" "B" "B" "A" "B" "A"

> kPointCrossover(A.sol, B.sol, 20)

## Warning in kPointCrossover(A.sol, B.sol, 20): The length of
## the vectors is 10 so at most there can be 9 cut points.
## The parameter will be updated to this limit

## [[1]]
##  [1] "A" "B" "A" "B" "A" "B" "A" "B" "A" "B"
##
## [[2]]
##  [1] "B" "A" "B" "A" "B" "A" "B" "A" "B" "A"
```

Azken adibidean ikus daitekeen bezala, n tamainako bektore bat izanik, gehienez $n-1$ puntuko gurutzaketa aplika dezakegu; edonola ere, balio handiago bat aukeratzen badugu, funtzioak abisu bat emango du eta puntu kopuruaren parametroa balio maximoan ezarriko du. Balio maximoa aukeratuz gero, jatorrizko *guraso*-soluzioen elementuak tartekatuta agertuko dira soluzio berrietan; operadore horri *uniform crossover* deritzo.

Erabiliko dugun puntu kopuruak eragin handia izan dezake algoritmoaren performantzian, eta, hortaz, egokitu beharreko algoritmoaren parametroa da.

k-point crossover operadorea nahiko orokorra da, ia edozen bektoreri aplikatu ahal baitzaio. Hala eta guztiz ere, kodeketa batzuetan beste operadore espezifikoago batzuk erabiltzea egokiagoa izan daiteke [18]. Esate baterako, soluzioak bektore errealen bidez kodetuta badaude, bi soluzio era askotan konbina daitezke; adibidez, batez bestekoa kalkulatuz. Ikus dezagun operadore hori nola inplementa daitekeen R-n:

```
> meanCrossover <- function(sol1, sol2) {
+   new.solution <- (sol1 + sol2) / 2
+   return(new.solution)
+ }
>
> s1 <- runif(10)
> s2 <- runif(10)
> s1
```

```
## [1] 0.97872844 0.49811371 0.01331584 0.25994613 0.77589308
0.01637905
## [7] 0.09574478 0.14216354 0.21112624 0.81125644
```

```
> s2
```

```
## [1] 0.03654720 0.89163741 0.48323641 0.46666453 0.98422408
0.60134555
## [7] 0.03834435 0.14149569 0.80638553 0.26668568
```

```
> meanCrossover(s1, s2)
```

```
## [1] 0.50763782 0.69487556 0.24827612 0.36330533 0.88005858
0.30886230
## [7] 0.06704457 0.14182962 0.50875588 0.53897106
```

Soluzioak permutazioen bidez kodetzen direnean, bete beharreko murrizketak direla eta, *k-point crossover* operadorea ezin da erabili kodeketa mota horrekin. Demagun bi permutazio ditugula, $s_1 = 12345678$ eta $s_2 = 87654321$, eta gurutzaketa puntu bat, $i = 3$. Lehenengo soluzio berria lortzeko s_1 soluziotik lehendabiziko hiru posizioak kopiatuko ditugu; hau da, 123, eta besteak s_2-tik, hots, 54321. Hortaz, lortutako soluzioa $s' = 12354321$ da; zoritxarrez, ez da permutazio bat. Hori dela eta, permutazioak gurutatzeko operadore bereziak behar ditugu.

Permutazioen murrizketak kontuan hartzen dituzten aukera asko izan arren [36], hemen puntu bakarreko gurutzaketa-operadorearen baliokidea ikusiko dugu. Puntu bateko gurutzaketan bezala, hasteko, puntu bat aukeratuko dugu zoriz, i. Ondoren, lehenengo soluzio berria sortzeko, *guraso*-soluzio baten lehenengo i posizioetako balioak zuzenean kopiatuko ditugu; gainontzeko balioak zuzenean beste *guraso* soluziotik kopiatu beharrean, ordena bakarrik hartuko dugu kontuan. Hau da, aurreko adibidera itzuliz, soluzio berri bat sortzeko s_1-etik lehenengo 3 elementuak zuzenean kopiatuko ditugu, 123, eta falta direnak, 45678, s_2-an agertzen diren ordenan kopiatuko ditugu, hots, 87654. Emaitza, beraz, $s' = 12387654$ izango da, eta, kasu honetan bai, permutazio bat. Era berean, bigarren soluzio berri bat sor daiteke s_2-tik lehenengo hiru posizioak kopiatuz (876) eta gainontzekoak s_1-n agertzen diren ordenan kopiatuz (12345); beste semea, beraz, 87612345 izango da. Operadore horri *Order crossover* deritzo, eta **metaheuR** liburutegian `orderCrossover` funtzioan[1] dago inplementaturik.

```
> sol1 <- randomPermutation(10)
> sol2 <- identityPermutation(10)
> as.numeric(sol1)
```

```
##  [1]  3  7  8  2  1  4 10  5  9  6
```

[1] Funtzio honetan inplementatuta dagoena *2-point crossover* operadorea da. Hau da, bi puntu erabiltzen dira, eta, soluzioak eraikitzeko, bi puntuen artean dagoen soluzio zatia soluzio batetik zuzenean kopiatu ondoren, gainontzeko elementuak beste *guraso*-soluzioan agertzen diren ordenan ezartzen dira.

```
> as.numeric(sol2)

## [1]  1  2  3  4  5  6  7  8  9 10

> new.solutions <- orderCrossover(sol1, sol2)
> as.numeric(new.solutions[[1]])

## [1]  1  3  4  2  5  6  7  8  9 10

> as.numeric(new.solutions[[2]])

## [1]  3  7  8  4  2  1 10  5  9  6
```

Mutazioa

Esan bezala, populazioak eboluzionatu ahal izateko soluzioak desberdinak izatea berebizikoa da. Hori dela eta, behin gurutzaketa-operadorearen bidez soluzio berriak lortzen ditugunean, zorizko aldaketak eragin ohi dira mutazio-operadorearen bidez.

Mutazioaren kontzeptua ILS algoritmoko perturbazioaren antzerakoa da, eta, kasu hartan bezala, operadore ezberdinak erabil daitezke mutazioa burutzeko. Hala nola, algoritmoa diseinatzean erabaki behar dugu zenbaterainoko aldaketak eragingo ditugun soluzioetan. Esate baterako, permutazio bat mutatzeko zorizko trukaketak erabil ditzakegu, baina zenbat posizio trukatuko ditugun aldez aurretik erabaki beharko dugu. Parametro horri mutazioaren magnitude deritzogu.

Mutazio-operadorea era probabilistikoan aplikatzen da eskuarki; hau da, ez zaie soluzio guztiei aplikatzen. Hortaz, mutazio-operadoreari lotutako bigarren parametro bat ere izango dugu: mutazio-probabilitatea.

Mutazio-operadorea aukeratzean –eta baita diseinatzean ere– hainbat gauza hartu behar dira kontuan. Hasteko, soluzioen bideragarritasuna mantentzea garrantzitsua da; hau da, mutazio-operadorea bideragarria den soluzio bati aplikatuz gero, emaitzak soluzio bideragarria izan behar du. Bestalde, algoritmoak soluzio bideragarrien espazio osoa arakatzeko gaitasuna izan behar du, eta, beraz, mutazio-operadoreak edozein soluzio sortzeko gaitasuna izan behar du. Hau da, edozein soluzio hartuta, mutazio-operadorearen hainbat aplikazioren bidez beste edozein soluzio sortzea posible izan behar du. Amaitzeko, lokaltasuna ere mantendu behar da –hau da, mutazioak eragindako aldaketak, txikia izan behar du–, gurasoengandik heredatutako ezaugarriak galdu ez daitezen.

Horrenbestez, algoritmo genetikoaren eskema orokorra 3.1 sasikodean ikus daiteke, eta **metaheuR** paketeko `basicGeneticAlgorithm` funtzioan dago inplementaturik. Ikus dezagun funtzio horren erabileraren adibide bat *graph coloring* problemaren instantzia bat ebazteko. Lehenik, zorizko grafo bat sortuko dugu problemaren instantzia sortzeko.

Algoritmo genetikoak

```
1  input: evaluate, select_reproduction, select_replacement, cross,
   mutate eta !stop_criterion operadoreak
2  input: init_pop hasierako populazioa
3  input: mut_prob mutazio-probabilitatea
4  output: best_sol
5  pop=init_pop
6  while stop_criterion do
7    evaluate(pop)
8    ind_rep = select_reproduction(pop)
9    new_ind = reproduce(ind_rep)
10   for each n in new_ind do
11     mut_prob probabilitatearekin egin mutate(n)
12   done
13   evaluate(new_ind)
14   if new_ind multzoan best_ind baino hobea den soluziorik badago
15     Eguneratu best_sol
16   fi
17   pop=select_replacement(pop,new_ind)
18 done
```

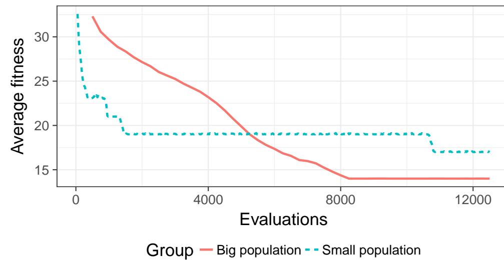
3.1 algoritmoa. Algoritmo genetikoen sasikodea

```
> library(igraph)
> n <- 50
> rnd.graph <- aging.ba.game(n=n, pa.exp=2, aging.exp=0, m=3,
+                            directed=FALSE)
> gcp <- graphColoringProblem(graph=rnd.graph)
```

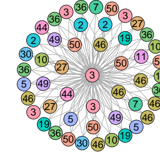
Jarraian algoritmoaren zenbait elementu definituko ditugu. Hasierako populazioa sortzeko, lehenik, bere tamaina erabaki behar dugu. Parametro horren garrantzia kontuan hartuta, bi baliorekin probatuko dugu, emaitzak alderatzeko: n eta $10n$. Behin hasierako populazioaren tamaina definituta, bertako soluzioak zoriz sortuko ditugu, eta, bideragarriak ez badira, zuzenduko egingo ditugu `gcp` objektuaren `correct` funtzioa erabiliz.

```
> n.pop.small <- n
> n.pop.big <- 10 * n
> levels <- paste("C", 1:n, sep="")
> createRndSolution <- function(x) {
+   sol <- factor(paste("C", sample(1:n, size=n, replace=TRUE),
+                       sep=""), levels=levels)
+   return(gcp$correct(sol))
+ }
> pop.small <- lapply(1:n.pop.small, FUN=createRndSolution)
> pop.big   <- lapply(1:n.pop.big, FUN=createRndSolution)
```

Hasierako populazioaz gain, ondoko parametro hauek ezarri behar ditugu:



(a) Algoritmo genetikoaren progresioa



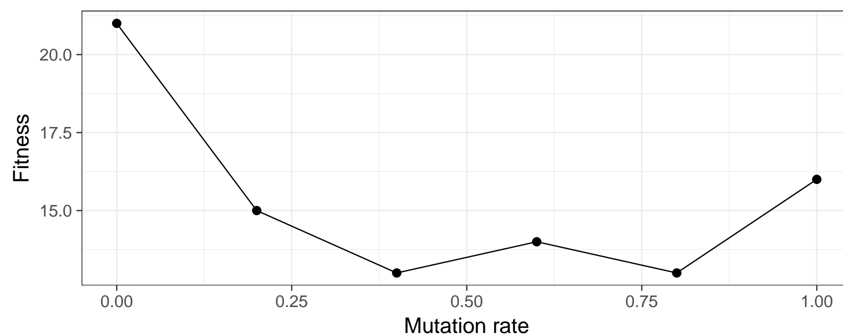
(b) Lortutako soluzioa

3.5 irudia. Definitutako algoritmo genetikoaren progresioa *graph coloring* problemaren instantzia batean, bi populazio tamaina ezberdin erabiliz. Ezkerrean, tamaina handiko populazioarekin lortutako soluzioa ikus daiteke.

- Hautespen-operadoreak - Hurrengo belaunaldira zuzenean pasatuko diren soluzioak aukeratzeko hautespen elitista erabiliko dugu, populazio erdia aukeratuz; zein soluzio gurutzatuko diren aukeratzeko, berriz, lehiaketa-hautespena erabiliko dugu.
- Mutazioa - Soluzioak mutatzeko `factorMutation` funtzioa erabiliko dugu. Funtzio horrek zenbait posizio zoriz aukeratzen ditu, eta bertako balioak zoriz aldatzen ditu. Funtzioak parametro bat du, `ratio`, aldatuko diren posizioen ratioa adierazten duena. Gure kasuan 0.1 balioa erabiliko dugu; alegia, posizioen %10 aldatuko da mutazioa aplikatzen denean. Soluzioak ze probabilitaterekin mutatuko ditugun finkatu behar da `mutation.rate` parametroaren bidez. Gure kasuan, probabilitatea bat zati populazioaren tamaina izango da.
- Gurutzaketa - Soluzioak gurutzatzeko *k-point crossover* operadorea erabiliko dugu, $k = 2$ finkatuz.
- Beste parametro batzuk - Algoritmo genetikoaren parametroaz gain, beste bi parametro finkatuko ditugu: `non.valid = 'discard'`, bideraezinak diren soluzioak baztertu behar direla adierazteko, eta `resources`, gelditze-irizpidea finkatzeko. Kasu horretan, $5n^2$ ebaluazio egingo dira.

Jarraian parametro hauek erabiliz algoritmo genetikoa exekutatzeko kodea ikus dezakegu.

```
> args <- list()
> args$evaluate              <- gcp$evaluate
> args$initial.population    <- pop.small
> args$selectSubpopulation   <- elitistSelection
> args$selection.ratio       <- 0.5
> args$selectCross           <- tournamentSelection
> args$mutate                <- factorMutation
> args$ratio                 <- 0.1
> args$mutation.rate         <- 1 / length(args$initial.population)
> args$cross                 <- kPointCrossover
```



3.6 irudia. Mutazio-probabilitatearen eragina algoritmo genetikoaren azken emaitzan. Irudian ikus daitekeen bezala, soluziorik onena ematen duen mutazio-probabilitatearen balioa 0.5 inguruan dago (zehazki, 0.6).

```
> args$k                      <- 2
> args$non.valid              <- "discard"
> args$resources              <- cResource(evaluations=5 * n^2)
>
> bga.small <- do.call(basicGeneticAlgorithm, args)
>
> args$initial.population     <- pop.big
> args$mutation.rate          <- 1 / length(args$initial.population)
>
> bga.big <- do.call(basicGeneticAlgorithm, args)
>
> plotProgress(list("Big population"=bga.big,
+                   "Small population"=bga.small), size=1.1) +
+   labs(y="Average fitness") + aes(linetype=Group) + thm.bh
```

3.5 irudian bi populazio-tamaina erabiliz lortutako progresioa ikus daiteke. Populazioa txikia denean algoritmoak oso azkar konbergitzen du 19 koloreko soluzio batera. Populazioko soluzio gehienak oso antzerakoak direnean, soluzio berriak sortzeko bide bakarra mutazioa da, baina, prozesu hori oso motela denez, grafikoan ikus daiteke soluzioen batez besteko *fitness*a ez dela aldatzen.

Populazioaren tamaina handitzen dugun heinean konbergentzia motelagoa da, baina lortutako soluzioa, aldiz, hobea.

Populazioaren tamainak ez ezik, beste hainbat parametrok ere eragin handia izan dezakete algoritmoaren emaitzan; esate baterako, mutazioaren probabilitateak. Adibide gisa, populazio-tamaina txikia erabiliz mutazio-probabilitate ezberdinak probatuko ditugu, eta bakoitzarekin lortutako emaitzak alderatuko ditugu.

```
> args$initial.population     <- pop.small
> args$verbose                <- FALSE
```

```
> args$resources            <- cResource(evaluations=n^2)
>
> testMutProb <- function (rate) {
+   args$mutation.rate <- rate
+   res <- do.call(basicGeneticAlgorithm, args)
+   return(getEvaluation(res))
+ }
>
> ratios <- seq(0,1,0.2)
> evaluations <- sapply(ratios , FUN = testMutProb)
>
> df <- data.frame("Mutation_rate"=ratios, "Fitness"=evaluations)
> ggplot(df, aes(x=Mutation_rate, y=Fitness)) +
+   geom_line() +
+   geom_point(size=3) +
+   labs(x="Mutation rate") + thm.bh
```

3.6 irudian lortutako emaitzak ikus daitezke. Grafikoak agerian uzten du mutazio-probabilitate txikiegiak zein handiegiak ezartzea kaltegarria dela bilaketa-prozesuarenzat, 0.5 ingurukoa izanik probabilitate eraginkorrena. Probabilitate txikiak ditugunean, populazioaren dibertsitatea baxua da, eta beraz konbergentzia goiztiarra gertatzen da. Aldiz, mutazio-probabilitate handiak ditugunean, bilaketak ia zorizkoak dirudi, eta algoritmoak ez du ia konbergitzen.

Estimation of Distribution Algorithms

Aurreko atalean ikusi dugun bezala, algoritmo genetikoetan indibiduo berriak naturan inspiratutako gurutzaketa- eta mutazio-operadoreak aplikatuz lortzen dira. Prozesu horren bitartez, populazioan dauden ezaugarriak mantentzea espero da.

Zenbait ikertzailek ideia hori hartu eta ikuspuntu matematikotik birformulatu zuten. Horrela, gurutzaketa eta mutazioa erabili beharrean, eredu probabilistikoak erabiltzea proposatu zuten, populazioaren *esentzia* jasotzeko helburuarekin. Horixe da EDA –*Estimation of Distribution Algorithms*– algoritmoen ideia nagusia.

Algoritmo genetikoen eta EDAen artean dagoen diferentzia bakarra indibiduo berriak sortzeko erabiltzen den estrategia da. Gurutzaketa eta mutazioa erabili beharrean, eredu probabilistiko bat doitzen da uneko populazioaren gainean. Ondoren, eredua behar adina aldiz laginduz lortuko ditugu indibiduo berriak.

EDA algoritmoen esentzia, beraz, eredu probabilistikoa da. Ildo horretan, ereduak soluzio-adierazpide bakoitzari probabilitate bat esleituko dionez, soluzioen kodeketa eredua diseinatzeko orduan puntu kritikoa bihurtuko da.

Konplexutasun ezberdineko eredu probabilistikoak darabiltzaten EDAk proposatu dira literaturan, UMDA –*Univariate Marginal Distribution Algo-*

rithm– izanik hurbilketa sinple eta hedatuena. Kasu horretan soluzioaren osagaiak –bektore bat bada, beraren posizioak– independenteak direla joko da, eta osagai bakoitzari dagokion bazter-probabilitatea estimatuko da. Gero, indibiduoak sortzean soluzioaren osagaiak banan-banan laginduko ditugu probabilitate horiei jarraituz.

Bazter-probabilitateak maneiatzeko `UnivariateMarginals` objektua erabil dezakegu. Beraren erabilera ikusteko, populazio txiki bat sortuko dugu, eta bazter-probabilitateak kalkulatuko ditugu.

```
> population <- lapply(1:5,
+                      FUN=function(x) {
+                        res <- factor(sample(1:3, 10, replace=TRUE),
+                                      levels=1:3)
+                        return(res)
+                      })
```

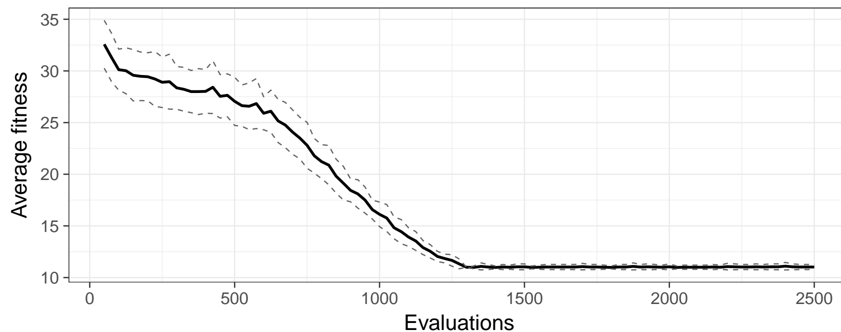
Orain, `univariateMarginals` funtzioa erabiliz bazter-probabilitateak kalkulatuko ditugu:

```
> model <- univariateMarginals(data=population)
> do.call(rbind, population)

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    1    2    2    3    1    3    3    2    2     1
## [2,]    1    1    3    2    3    2    3    3    2     3
## [3,]    3    1    2    1    1    2    1    2    3     2
## [4,]    2    2    2    1    3    3    3    1    3     2
## [5,]    3    2    3    2    2    3    1    2    3     3

> model@prob.table

##   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## 1  0.4  0.4  0.0  0.4  0.4  0.0  0.4  0.2  0.0   0.2
## 2  0.2  0.6  0.6  0.4  0.2  0.4  0.0  0.6  0.4   0.4
## 3  0.4  0.0  0.4  0.2  0.4  0.6  0.6  0.2  0.6   0.4
```

Sortutako soluzioek 10 elementu dituzte (10 posizioko bektore kategorikoak dira), eta populazioak 5 soluzio ditu. Lehenengo elementu edo posizioari erreparatzen badiogu, 5 soluzioetatik lehenengo biak 1 balioa dute, laugarrenak 2 balioa eta beste biek 3 balioa. Hortaz, elementu horretarako, 1 eta 3 balioen probabilitatea 0.4 izango da $-\frac{2}{5}$, alegia–, eta 2 balioaren probabilitatea 0.2 izango da, bazter-probabilitateen taulan ikus daitekeen bezala.

Estimatutako eredu probabilistikoa soluzio berriak sortzeko erabil daiteke, posizioz posizioko laginketa eraginez. Prozesu hori `simulate` funtzioaren bidez egiten da.

```
> simulate(model, nsim=2)

## [[1]]
##  [1] 3 1 2 2 1 3 3 2 2 3
## Levels: 1 2 3
##
```



3.7 irudia. UMDA algoritmoaren progresioa *graph coloring* problemaren instantzia batean aplikatuta. Marra jarraituak populazioko soluzioen batezbesteko *fitness*a adierazten du; marra etenek, berriz, desbideraketa estandarra adierazten dute. Ikus daitekeenez, populazioak konbergitzen duen heinean soluzioen *fitness*aren aldakortasuna murrizten da.

```
## [[2]]
##  [1] 2 2 2 2 3 3 3 3 3 1
## Levels: 1 2 3
```

UMDA algoritmoa **metaheuR** paketeko `basicEda` funtzioaren bidez exekuta dezakegu, eta, segidan, aurreko ataleko *graph coloring* problema ebazteko erabiliko dugu. Horretarako, bakarrik hautespen-operadoreak eta ereduak estimatzeko funtzioak zehaztu behar ditugu –algoritmo genetikoekin bat datozen parametroez gain–.

```
> args <- list()
> args$evaluate              <- gcp$evaluate
> args$initial.population    <- pop.small
> args$selectSubpopulation   <- elitistSelection
> args$selection.ratio       <- 0.5
> args$learn                 <- univariateMarginals
> args$non.valid             <- "discard"
> args$resources             <- cResource(evaluations = n^2)
>
> umda <- do.call(basicEda, args)
>
> plotProgress(umda, size=1.1) +
+   geom_line(aes(y=Current_sol + Current_sd), col="gray40",
+             linetype=2) +
+   geom_line(aes(y=Current_sol - Current_sd), col="gray40",
+             linetype=2) +
+   labs(y="Average fitness") + thm.bh
```

Bilaketaren progresioa 3.7 irudian erakusten da. Marra etenek populazioan dauden soluzioen *fitness*aren desbideraketa erakusten dute, eta marra jarraikiek, ordea, populazioko soluzioen *fitness*aren batezbestekoa. Popula-

zioak eboluzionatu ahala, populazioaren dibertsitatea murrizten dela ikus dezakegu desbideraketaren murrizketari erreparatuz. Amaieran, bilaketak 11 kolore darabiltzan soluzio batera konbergitzen du.

Bazter-probabilitateak kalkulatzeko estrategian ia edozein bektore erabil daiteke; alabaina, balio errealak baditugu, bazter-probabilitateak zein probabilitate-banaketarekin modelatuko ditugun erabaki beharko dugu aurrez. Bektore jarraikietatik haratago joanda, soluzioek murrizketak dituztenean, permutazioetan kasu, gauzak konplikatu egiten dira.

Populazioa, permutazio multzo bat denean, posible da bazter-probabilitateak bektore kategorikoekin bezala estimatzea, baina, ondoren, eredua lagintzen dugunean ez ditugu halabeharrez permutazioak lortuko, balio errepikatuak ager baitaitezke. Hona hemen adibide bat:

```
> n <- 5
> rndPop <- lapply(1:50,
+                  FUN=function(x) {
+                  rnd.perm <- as.numeric(randomPermutation(n))
+                  res <- factor(rnd.perm, levels=1:n)
+                  return(res)
+                 })
> perm.umda <- univariateMarginals(rndPop)
> simulate(perm.umda)

## [[1]]
## [1] 3 2 1 4 4
## Levels: 1 2 3 4 5
```

Arazo hori saihesteko, soluzio berriak lagintzean, permutazioak dakartzen murrizketak aintzat hartu behar dira. Horrela, laginketa-prozesuan lehenengo elementua zoriz aukeratuko dugu, zuzenean bazter-probabilitatea erabiliz.

```
> marginals <- perm.umda@prob.table
> remaining <- 1:n
> probabilities <- marginals[,1]
> new.element <- sample(remaining, size=1, prob=probabilities)
> new.solution <- new.element
```

Ondoren, bigarren elementua lagindu aurretik, lehenengo posiziorako aukeratu dugun elementua kendu beharko dugu aukera posibleetatik eta bazter-probabilitateak horren arabera eguneratu –erabili dugun elementuaren probabilitatea kendu eta normalizatu, gelditzen diren elementuen probabilitateen batura 1 izan dadin–:

```
> id <- which(remaining %in% new.element)
> remaining <- remaining [-id]
> marginals <- marginals[-id, ]
> probabilities <- marginals[, 2]
> probabilities <- probabilities / sum(probabilities)
> new.element <- sample(remaining, size=1, prob=probabilities)
> new.solution <- c(new.solution, new.element)
```

Estrategia berbera aplikatzen dugu 3. eta 4. elementuak erauzteko.

```
> id <- which(remaining %in% new.element)
> remaining <- remaining [-id]
> marginals <- marginals[-id, ]
> probabilities <- marginals[, 3]
> probabilities <- probabilities / sum(probabilities)
> new.element <- sample(remaining, size=1, prob=probabilities)
> new.solution <- c(new.solution, new.element)
>
> id <- which(remaining %in% new.element)
> remaining <- remaining [-id]
> marginals <- marginals[-id, ]
> probabilities <- marginals[, 3]
> probabilities <- probabilities / sum(probabilities)
> new.element <- sample(remaining, size=1, prob=probabilities)
> new.solution <- c(new.solution, new.element)
```

Amaitzeko, permutazioaren azken elementua definitzeko, soberan geratzen den elementua aukeratuko dugu zuzenean.

```
> id <- which(remaining %in% new.element)
> remaining <- remaining [-id]
> new.solution <- c(new.solution, remaining)
> new.solution
```

```
## [1] 4 3 5 1 2
```

Prozesu horrekin bazter-probabilitateak erabiliz permutazioen soluzioak betetzen dituzten soluzioak sortzen ditugu. Arazo bat dauka, ordea: eredua lagintzen dugun bakoitzean probabilitateak eguneratu egiten ditugu eta, ondorioz, lagintzen duguna ez da zehazki estimatutako eredu probabilistikoak adierazten duena. Beste era batera esanda, populaziotik ateratako *esentzia* gal dezakegu. Halakorik ez gertatzeko, permutazio-espazioetan definitutako Mallows eredua edo antzeko probabilitate-banaketak erabil ditzakegu. Mallows eredua **metaheuR** paketean dagoen `MallowsModel` objektuak inplementatzen du.

Swarm Intelligence

Eboluzioaren bidez natura indibiduoen diseinua *optimizatzeko* gai da; alabaina, algoritmo genetikoez gain, naturan optimizazio-estrategiak beste hainbat egoeratan ere ageri dira. Adibidez, animalia sozialen portaera eta jokabideak optimizazio-algoritmoak sortzeko inspirazio-iturri izan dira sarritan.

Zenbait espezietako indibiduoak –intsektuak, batik bat– banan-banan hartuta, oso izaki sinpleak dira, baina, taldeka lan egiten dutenean, ataza konplexuak era oso eraginkorrean egiten dituzte. Esate baterako, inurriak eta erleak elikagai-iturri onenak aukeratzeko gai dira, eta, haien kapazitate eta egoeraren arabera, ingurunea esploratzen duten indibiduoen kopurua egoki-

tzen dute iturri berriak lortu ahal izateko; era berean, bizitzeko toki egokienak aukeratzeko gai dira.

Horrelako bizidun multzoetan erabakiak ez dira era zentralizatuan hartzen –alegia, ez dago agintzen duen *nagusi*rik –. Beraz, mekanismo sinple batzuei jarraituz eta, batez ere, beren arteko komunikazioari esker, kolonia bateko indibiduoak elkarrekin antolatzeko gai dira, inolako koordinazio zentralizaturik gabe.

Mota horretako portaerak dira zehazki *swarm intelligence* deritzon arloaren inspirazio-iturria. *Swarm intelligence* algoritmoak lehenengo aldiz 1988. urtean robotika arloan proposatu ziren [5], baina urte gutxi batzuetan optimizazio-mundura hedatu ziren. Horrela, 90eko hamarkadan inurri-kolonien optimizazioa –*Ant Colony Optimization*, ingelesez– proposatu zen [11, 12].

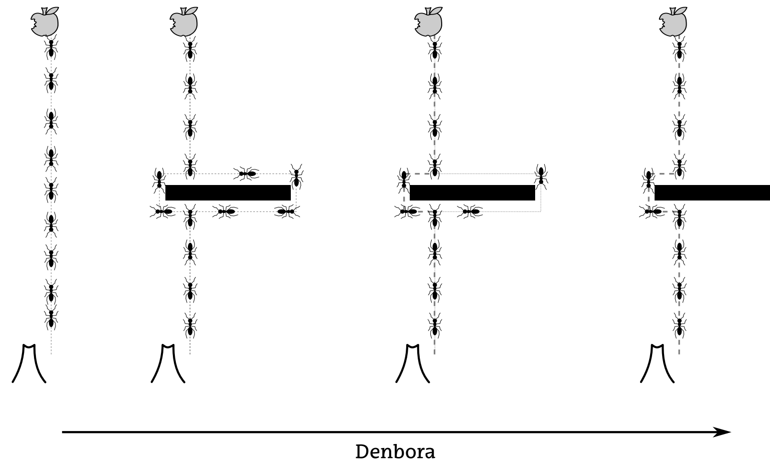
Hurrengo bi ataletan *swarm intelligence* arloan dauden bi algoritmo ezagunenak aztertuko ditugu, inurri-kolonien optimizazioa eta *particle swarm optimization*.

Ant Colony Optimization

Inurriek, elikagai-iturri bat topatzen dutenean, inurritegitik bertara dagoen biderik motzena topatzeko gaitasuna dute. Inurri bakar batek ezin du horrelakorik lortu; taldeka, ordea, komunikazio-mekanismo sinpleei esker, ataza konplexu hori egiteko gai dira. Erabiltzen duten komunikazio mota zeharkakoa da, inurriek jariatzen duten molekula mota berezi baten bidez gauzatzen dena: feromona.

Inurriek toki batetik bestera mugitzean ibili diren bidean feromona-lorratz bat uzten dute. Inurriak beren kolonia-kideek utzitako feromona lorratzak detektatzeko gai dira, eta horretaz baliatzen dira haien ibilbideak aukeratzeko. Hala, zenbat eta feromona gehiago egon bide batean, orduan eta probabilitate handiagoa dago bertatik igarotzen diren inurriek bide horretatik jarraitzeko. Bestalde, inurri batek, elikagai-iturri bat topatzen duenean, bidetik uzten duen feromona kopurua iturriaren kalitatearen arabera egokitzen du; zenbat eta kalitate handiagoa, orduan eta feromona kopuru handiagoa jariatzen du. Azkenik, feromona lurrunkorra da, alegia, denborarekin lurrundu egiten da.

Arau sinple horiek erabiliz inurriak elikagai-iturri onenak aukeratzeko gai dira; are gehiago, elikagaiaren eta inurritegiaren arteko biderik motzena ere topa dezakete. Mekanismo honen funtzionamendua hobeto ulertzeko 3.8 irudiari erreparatuko diogu. Hasieran, bide motzena feromona-lorratzaren bidez markaturik dator. Bidea moztean, eskuineko eta ezkerreko bideetan ez dago feromonarik, eta, hortaz, inurri batzuk eskuinetik eta beste batzuk ezkerretik joango dira, probabilitate berdinarekin. Ezkerreko bidea motzagoa denez, denbora berdinean inurri gehiago igaroko dira, eta horrenbestez ezkerreko bideko feromona-lorratza indartsuagoa izango da. Denbora igaro ahala, datozen



3.8 irudia. Feromonaren erabilera. Hasierako egoeran biderik motzena feromona-lorratzak zehazten du. Bidea mozten dugunean, inurriek, eskuinetik ala ezkerretik joatea erabaki beharko dute. Hasieran erabaki hori probabilitate berdinarekin hartuko dute, feromonarik ez baitago ez ezkerrean eta ez eskuinean ere. Alabaina, eskuineko bidea luzeagoa da eta, ezkerreko bidearekin alderatuta, inurri-fluxua txikiagoa izango da, inurriek denbora gehiago beharko baitute elikadura-iturrira joanetorria egiteko. Arrazoi horregatik, denbora igaro ahala, eskuineko lorratza ahuldu egingo da eta ezkerrekoa, berriz, indartu. Horrek guztiak puntu honetara iristen diren inurrien erabakia baldintzatuko du, ezkerreko bidea aukeratzeko joera areagotuko eta bi bideen arteko diferentzia handituko. Denbora nahikoa igaroz gero eskuineko lorratza guztiz desagertuko da, eta inurriek bide motzena bakarrik aukeratuko dute.

inurriek ezkerretik joateko joera handiagoa izango dute, eta horrek bide hori are gehiago indartuko du. Eskuineko bidean lorratza apurka-apurka lurrundu egingo da erabat desagertu arte.

Laburbilduz, inurriek ez dituzte bi bideak konparatzen, baina, hala eta guztiz ere, azkenean, bide motza soilik erabiltzea lortzen dute. *Ant Colony Optimization* (ACO) deritzon metaheuristikak inurrien portaera hori hartzen du intuiziotzat, eta inurri artifizialak erabiltzen ditu optimizazio-problemaren soluzioak sortzeko. Soluzio horiek ez dira edonolakoak izango; izan ere, naturan bezalaxe, aurretik igarotako inurriek utzitako lorratzei jarraituz sortuko dira.

Lorratzak feromona ereduen bidez adierazten dira, eta eredu horiek optimizazio-algoritmoaren pauso bakoitzean bi eratan eguneratzen dira. Alde batetik, inurriek sortutako soluzioen kalitatea –hots, helburu-funtzioaren balioa– feromona kopurua areagotzeko erabiltzen da. Bestalde, iterazioz iterazio feromona kopurua murriztuko da, naturan gertatzen den lurrunketa simulatuz. Feromona eredua zehazteko, pauso bakoitzean feromonaren areagotzea eta murrizketa nola egin erabaki behar dugu.

Beraz, bi gauza behar dira ACO algoritmo bat diseinatzeko: feromona eredu bat, eta soluzioak sortzeko algoritmo bat. Soluzioak sortzeko era sinpleena osagaietan oinarritzen den algoritmo eraikitzaile bat erabiltzea da. Ikus dezagun adibide bat.

Demagun *Maximal Independent Set* (MIS) problema ebatzi nahi dugula. Problema horrek soluzioak bektore bitarren bidez kodetzen ditugu, eta, hortaz, bektoreen posizioak soluzioen osagaitzat har ditzakegu. Posizio bakoitzak bi balio posible har ditzake, 0 edo 1. Soluzio bat sortzeko, inurri artifizial bakoitzak, hasteko, soluzio-bektorearen lehenengo posizioko balioa 0 edo 1 den aukeratu beharko du. Horretarako, uneko feromona kopurua hartuko du aintzat, probabilitate handiagoa esleituz feromona gehiago duen aukerari; behin lehenengo posizioko balioa finkaturik, bigarren posizioari jarriko dio arreta, eta, lehenengo pausoan bezala, balio bat (0 edo 1) probabilistikoki aukeratuko du feromona kopuruan oinarrituz. Prozesu bera soluzio osoa sortu arte errepikatuko da.

Adibide honetan, feromona eredua matrize sinple baten bidez inplementa daiteke, zutabe bakoitzak soluzio-bektorearen posizio bat adierazten duelarik. Bestalde, matrizeak bi errenkada izango ditu, posizio bakoitzean 0 eta 1 balioek duten feromona kopurua gordetzeko.

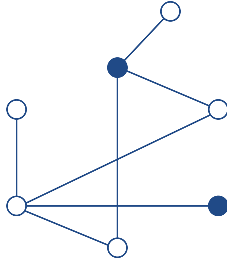
Feromona eredu honen erabilera argitzeko, MIS problema erabili beharrean, antzerakoa den beste problema bat erabiliko dugu: *Minimum Dominating Set* (MDS). Laburki azalduz, grafo bat emanda, nodoen azpimultzo bat menderatze multzoa da –*dominating set*, ingelesez– baldin eta azpimultzoan ez dauden nodo guztiak, gutxienez, azpimultzoko nodo bati konektatuta badaude. Grafo bat emanik, kardinalitate minimoko menderatze multzoa topatzean datza MDS problemak. 3.9 irudian problema horretarako hiru soluzio ikus daitezke.

Lehenik, zorizko grafo bat eraikiko dugu MDS problemaren instantzia bat definitzeko.

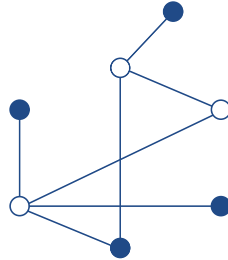
```
> n <- 10
> rnd.graph <- aging.ba.game (n, 0.5, 0, 2, directed=FALSE)
> mdsp <- mdsProblem(graph=rnd.graph)
```

Bigarren pausoan, feromona eredua eraikitzeko matrizea hasieratu behar dugu. Feromona eredu mota hau **metaheuR** paketeko `VectorPheromone` klasearen bidez dago inplementaturik. Ohikoena balio finko batekin hasieratzea da. Era horretan osagai guztiak balio berdina dute, eta, beraz, aukera posible guztien probabilitatea berdina izango da. Gogoratu gure adibidean feromona ereduaren matrizeak bi errenkada dituela, soluzioak bektore bitarrak direlako.

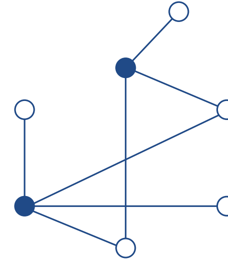
```
> init.trail <- matrix (rep(1, 2*n), ncol=n)
> evaporation <- 0.1
> pheromones <- vectorPheromone(binary=TRUE,
+                               initial.trail=init.trail,
+                               evaporation.factor=evaporation)
```



(a) Soluzio hau bideraezina da, menderatze multzoa ez baita



(b) 4 tamainako menderatze multzoa



(c) 2 tamainako menderatze multzoa

3.9 irudia. Irudiak MDS problemarako 3 soluzio jasotzen ditu –beltzez adierazita dauden nodoak–. Lehenengoa (ezkerrean dagoena) ez da bideragarria, soluziokoa ez den nodo bat ez baitago konektatuta soluzioko nodo batekin ere. Bigarren soluzioa bideragarria da, baina ez optimoa. Azken soluzioa optimoa da, ez baitago 1 tamainako soluzio bideragarririk.

```
> pheromones@trail
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    1    1    1    1    1    1    1    1    1     1
## [2,]    1    1    1    1    1    1    1    1    1     1
```

Aurreko kodean ikus daitekeen bezala, feromona eredua guztiz zehazteko, `evaporation.factor` parametroa finkatu behar da. Parametro hori feromonaren lurrunketarekin dago erlazionatuta, eta matrizean dauden balioak pauso bakoitzean zenbat murriztuko diren adierazten du; balio hori kontuan izanik, lurrunketa `evaporate` funtzioa erabiliz egiten da.

```
> evaporate(pheromones)
> pheromones@trail
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]  0.9  0.9  0.9  0.9  0.9  0.9  0.9  0.9  0.9   0.9
## [2,]  0.9  0.9  0.9  0.9  0.9  0.9  0.9  0.9  0.9   0.9
```

```
> evaporate(pheromones)
> pheromones@trail
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,] 0.81 0.81 0.81 0.81 0.81 0.81 0.81 0.81 0.81  0.81
## [2,] 0.81 0.81 0.81 0.81 0.81 0.81 0.81 0.81 0.81  0.81
```

Esan dugun bezala, inurriek feromona ereduak soluzioak eraikitzeko erabiltzen dituzte. Soluzioak `buildSolution` funtzioaren bitartez lortzen dira.

```
> buildSolution(pheromones, 1)

## [[1]]
##  [1]  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE  TRUE  TRUE FALSE
```

Inurriek ingurunetik ibiltzen diren heinean feromona uzten dute, eta, era berean, inurri artifizialek soluzioak eraikitzen dituztenean feromona kopurua handitzen dute; eredua eguneratzeko `updateTrail` funtzioa erabiltzen da.

```
> solution <- buildSolution(pheromones, 1)[[1]]
> eval <- mdsp$evaluate(solution)
> eval

## [1] 4

> updateTrail(object=pheromones, solution=solution, value=eval)
> pheromones@trail

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,] 4.81 4.81 0.81 0.81 4.81 4.81 4.81 0.81 0.81  4.81
## [2,] 0.81 0.81 4.81 4.81 0.81 0.81 0.81 4.81 4.81  0.81
```

Ikus daitekeen bezala, zutabe bakoitzean errenkada bati (inurriak aukeratutako balioari dagokionari, hain zuzen) soluzioaren helburu-funtzioaren balioa gehitu zaio, inurriek utzitako lorratza irudikatzeko.

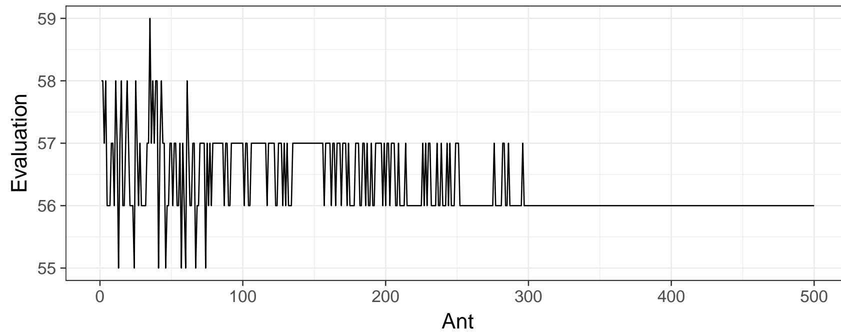
Elementu horiekin guztiekin, ACO sinple bat sor dezakegu. Baina lehenik eta behin, MDSaren instantzia handi bat sortuko dugu.

```
> n <- 100
> rnd.graph <- aging.ba.game(n, 0.5, 0, 3, directed=FALSE)
> mdsp <- mdsProblem(graph=rnd.graph)
> init.trail <- matrix (rep(1, 2*n), ncol=n)
> pheromones <- vectorPheromone(binary=TRUE,
+                                initial.trail=init.trail,
+                                evaporation.factor=evaporation)
```

Orain, 500 inurri simulatuko ditugu; bakoitzak soluzio bat sortuko du, eta egindako *bidean* feromona utziko du. Algoritmoaren pauso bakoitzean inurri bat simulatuko dugu, eta haren ibilbidea amaitzean feromona *lurrundu*arazi egingo dugu.

```
> num.ant <- 500
> sol.evaluations <- vector()
> for (ant in 1:num.ant) {
+   solution <- mdsp$correct(buildSolution(pheromones, 1)[[1]])
+   eval <- mdsp$evaluate(solution)
+   updateTrail(pheromones, solution, eval)
+   evaporate(pheromones)
+   sol.evaluations <- c(sol.evaluations, eval)
+ }
```

3.10 irudiak algoritmoaren eboluzioa erakusten du. Irudian, lehenengo inurriek sortutako soluzioen helburu-funtzioaren balioak oso ezberdinak direla



3.10 irudia. ACO sinplearen eboluzioa MDS problema batean.

ikus daiteke. Alabaina, simulatutako inurri kopurua handitzen den heinean bariantza murriztu egiten da, eta, amaieran, prozedurak soluzio bakar batera konbergitzen du. Edonola ere, soluzio hori ez da hasieran lortutakoak baino hobea.

Portaera hori sakonago aztertuz gero ondokoa ondorioztatzen da: nahiz eta naturan honela gertatu, optimizazioaren ikuspegitik, inurri guztiek feromona eredua eguneratzea ez da hurbilketarik onena. Hori dela eta, eskuarki beste estrategia bat erabili ohi da. Inurriak banan-banan simulatu ordez, tamaina zehatz bateko inurri-kolonia bat sortzen da, eta, iterazio bakoitzean, inurritegiko inurri guztiek sortutako soluzioetatik bakar bat erabiliko da feromona kopurua eguneratzeko. Soluzio bakar hori aukeratzeko bi estrategia erabil daitezke:

- Iterazioko soluziorik onena aukeratu. Inurriek uneko iterazioan sortutako soluzioetatik onena aukeratzen da, eta soluzio hori bakarrik erabiltzen da feromona kopurua eguneratzeko. Kasu honetan helburu-funtzioaren arabera egitea ez da beharrezkoa, bakarrik soluziorik onena erabiltzen baita eguneraketan. Hori dela eta, ohikoa da balio finko bat erabiltzea.
- Bilaketa-prozesu osoan topatutako soluziorik onena aukeratu. Hainbat kasutan, bilaketa soluzio on baten inguruan areagotzea interesatuko zaigu. Kasu horietan, feromona ereduaren eguneraketa bilaketa-prozesu osoan zehar topatu den soluziorik onena erabiliz egin daiteke. Aurreko puntuan bezala, eguneraketak ez du zertan helburu-funtzioaren balioarekiko proportzionala izan.

Aldaketa horrekin oinarrizko ACO algoritmoaren sasikodea defini dezakegu (ikusi 3.2 algoritmoa). `basic.aco` funtzioak oinarrizko ACOa inplementatzen du eta, hortaz, lehen sortutako MDS problema ebazteko erabil dezakegu. Ohiko parametroez gain, `basic.aco` zehazteko, argumentu hauek definitu behar dira:

- `nants` - Kolonia zenbat inurri artifizialek osatuko duten.

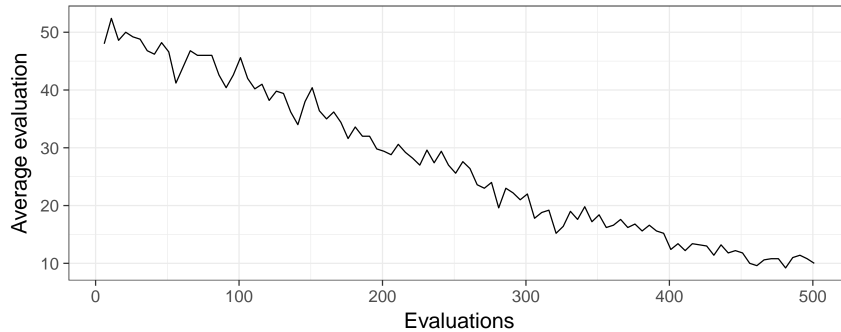
Inurri-kolonien algoritmoa

```
1  input: build_solution, evaporate, add_pheromone ,
   initalize_matrix eta stop_criterion operadoreak
2  input: k_size koloniaren tamaina
3  output: opt_solution
4  pheromone_matrix = initialize_matrix()
5  while !stop_criterion()
6    for i in 1:k_size
7      solution = build_solution(pheromone_matrix)
8      pheromone_matrix = add_pheromone(pheromone_matrix,solution)
9      if solution opt_solution baino hobea da
10       opt_solution=solution
11     fi
12   done
13   pheromone_matrix = evaporate(pheromone_matrix)
14 done
```

3.2 algoritmoa. Inurri-kolonien algoritmoaren sasikodea

- `pheromones` - Feromona eredua.
- `update.sol` - Nola eguneratuko dugun feromona eredua. Hiru aukera daude: `'best.it'`, iterazio bakoitzean sortutako soluziorik onena erabili; `'best.all'`, bilaketan zehar lortutako soluziorik onena erabili, edo `'all'`, sortutako soluzio guztietaz baliatu.
- `update.value` - Balio bat finkatzen bada, eguneraketa guztietan feromona-balio hori gehitzen zaio ereduari; `NULL` bada, helburu-funtzioa erabiltzen da. Kontuan hartu problema batzuetan helburu-funtzioak negatiboak direla eta feromona eredu batzuetan balio positiboak eta negatiboak ezin direla nahastu arazoak egon daitezkeelako probabilitateak kalkulatzean. Hori dela eta, helburu-funtzioaren zeinua kontuan hartu behar da feromona eredua hasieratzerakoan.

```
> args <- list()
> args$evaluate         <- mdsp$evaluate
> args$nants            <- 5
>
> init.value    <- 1
> initial.trail <- matrix(rep(init.value, 2*n), nrow=2)
> evapor        <- 0.1
>
> pher <- vectorPheromone(binary=TRUE,
+                         initial.trail=initial.trail,
+                         evaporation.factor=evapor)
>
> args$pheromones   <- pher
> args$update.sol   <- "best.it"
> args$update.value <- init.value / 10
```



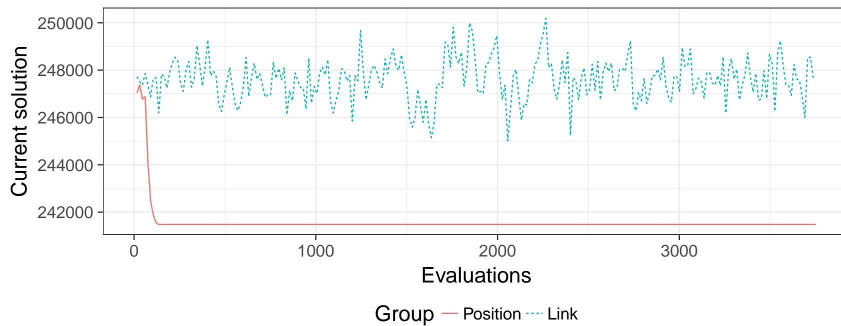
3.11 irudia. Oinarrizko ACO algoritmoaren eboluzioa MDS problema batean.

```
> args$non.valid    <- "correct"
> args$valid        <- mdsp$is.valid
> args$correct      <- mdsp$correct
>
> args$resources    <- cResource(iterations=100)
> args$verbose      <- FALSE
>
> results.aco <- do.call(basicAco, args)
> plotProgress(results.aco) + labs(y="Average evaluation") +
+   thm.bh
```

3.11 irudiak oinarrizko ACO algoritmoaren progresioa irudikatzen du. Algoritmo horrek lehen inplementatu dugun ACO sinpleak aztertzen duen soluzio kopuru berdina aztertzen du, baina, aurrekoa ez bezala, iterazioz iterazio soluzioa hobetuz doa. Grafikoan batez besteko *fitness*ak aldakuntza handia duela ikus daiteke. Hori oso kolonia txikia erabili dugulako da –5 inurri bakarrik–. Balio hori handitzen badugu, progresioa leunagoa izango da –eta, ziurrenik, emaitzak hobeak izango dira–, baina ebaluazio gehiago beharko ditugu.

ACO algoritmoen mamiak soluzioen eraikuntzan datza, eta, hortaz, soluzioen osagaien definizioa oso garrantzitsua da; osagaiek ez badute problemaren izaera kontuan hartzen, feromona ereduak ez du soluzioen informazioa behar bezala jasoko eta zentzua galduko du. Hori agerian gelditzen da jarraian dagoen adibidean.

Demagun LOP problema bat ebazteko ACO algoritmo bat erabili nahi dugula. Problema horretarako soluzioak permutazioen bidez kodetzen ditugunez, kodeketa mota honentzat egokia den feromona eredu bat behar dugu. Lehen ideia gisa, bektoreekin erabilitako eredu bera erabil dezakegu, soluzioen eraikuntzan permutazioak sortzeko behar diren aldaketak egiten baditugu betiere. Hau da, feromona eredua matrize karratu batean gordeko dugu. Bertan, soluzioen posizio bakoitzeko (errenkadak) balio posible bakoitzari (zutabeak) dagokion feromona kopurua gordeko dugu. Gero, solu-



3.12 irudia. Oinarrizko ACO algoritmoaren eboluzioa LOP problema batean.

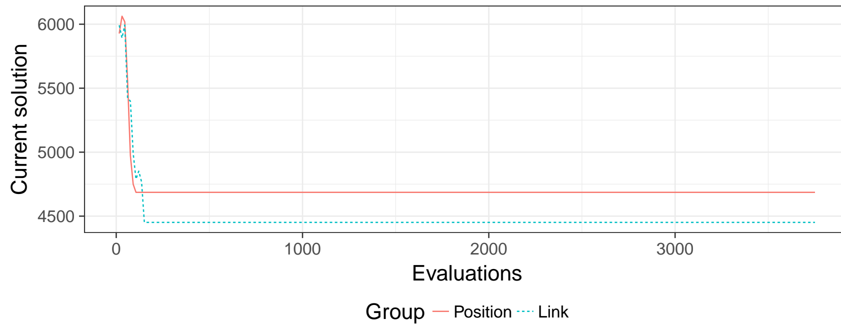
zioak osatzerakoan, urrats bakoitzean, aurretik aukeratu gabeko balioetatik bat aurkeztuko dugu, bakoitzaren feromona kopurua kontuan hartuz, noski. Eredu horrek UMDA definitzeko erabili genuen matrizearen antzerako bat erabiltzen du.

Dena dela, hau ez da feromona eredu posible bakarra. TSP probleman ikusi genuen permutazioek grafo osoko ziklo hamiltoniarrak adierazten dituztela. Hau da, n nodoko grafo oso bat izanik, edozein permutaziok n nodoak behin eta soilik behin bisitatzen dituen ibilbide bat adierazten du. Beraz, permutazioak osatzeko, nodoak lotzen dituzten ertzak erabil ditzakegu.

Ideia hori erabiliz beste feromona eredu bat plantea dezakegu. Eredu horrek ere matrize karratu bat erabiliko du, baina matrizearen interpretazioa –eta, hortaz, soluzioen eraikuntza– ezberdina da. Kasu horretan, matrizeak grafoaren ertzak adierazten ditu. Alegia, matrizearen (i, j) posizioan i nodotik j nodorako bideari dagokion feromona kopurua gordeko dugu. Adibidez, 3421 permutazioari dagozkion matrizeko posizioak $(3, 4)$; $(4, 2)$ eta $(2, 1)$ dira –problemaren arabera, $(1, 3)$ posizioa ere erabiltzea interesgarria izan daiteke, baina guk ez dugu aintzat hartuko gure adibidean–.

Bi eredu horiek `PermuPosPheromone` eta `PermuLinkPheromone` objektuetan inplementaturik daude. Jarraian, bi eredu horiek LOP problema bat ebazteko erabiliko ditugu, eta emaitzak alderatuko ditugu.

```
> n <- 100
> rnd.mat <- matrix(round(runif(n^2)*100), n)
> lop      <- lopProblem(matrix=rnd.mat)
>
> args <- list()
> args$evaluate <- lop$evaluate
> args$nants    <- 15
>
> init.value    <- 1
> initial.trail <- matrix(rep(init.value, n^2 ), n)
> evapor        <- 0.9
```



3.13 irudia. Oinarrizko ACO algoritmoaren eboluzioa TSP problema batean.

```
>
> pher <- permuLinkPheromone(initial.trail=initial.trail,
+                            evaporation.factor=evapor)
> args$pheromones   <- pher
> args$update.sol   <- "best.it"
> args$update.value <- init.value / 10
> args$resources    <- cResource(iterations=250)
> args$verbose      <- FALSE
>
> aco.links <- do.call(basicAco, args)
>
> args$pheromones <- permuPosPheromone(initial.trail, evapor)
> aco.pos <- do.call(basicAco, args)
>
> plotProgress(list("Position"=aco.pos, "Link"=aco.links)) +
+   aes(linetype=Group) + labs(y="Current solution") + thm.bh
```

3.12 irudiak esperimentuaren emaitza erakusten du. Grafikoan argi ikus daiteke noden arteko loturak soluzioen osagaitzat hartzen direnean, bilaketak ez duela aurrera egiten. Soluzioaren osagaiak posizioak direnean, berriz, iterazioz iterazio soluzioa hobetu egiten da. Horren arrazoia sinplea da: LOP probleman, soluzioko osagaien posizio absolutuak dira alderdi garrantzitsuena, eta ez osagaien arteko auzokidetasuna.

TSP problemarako, ordea, ertzetan oinarritzen den eredua egokia da; izan ere, zein hiritatik zein hiritara joan behar dugun interesatzen zaigu. Jarraian hori frogatzeko esperimentu bat egingo dugu; emaitzak 3.13 irudian daude.

```
> url <- system.file("bays29.xml.zip", package="metaheuR")
> cost.matrix <- tsplibParser(url)
> n <- ncol(cost.matrix)
> tsp <- tspProblem(cmatrix=cost.matrix)
>
> args <- list()
> args$evaluate <- tsp$evaluate
```

```
> args$nants    <- 15
>
> init.value    <- 1
> initial.trail <- matrix(rep(init.value, n^2 ), n)
> evapor        <- 0.9
>
> pher <- permuLinkPheromone(initial.trail=initial.trail,
+                            evaporation.factor=evapor)
> args$pheromones   <- pher
> args$update.sol   <- "best.it"
> args$update.value <- init.value / 10
> args$resources    <- cResource(iterations=250)
> args$verbose      <- FALSE
>
> aco.links <- do.call(basicAco, args)
>
> args$pheromones <- permuPosPheromone(initial.trail, evapor)
> aco.pos <- do.call(basicAco, args)
>
> plotProgress (list("Position"=aco.pos, "Link"=aco.links))  +
+   aes(linetype=Group) + labs(y="Current solution") + thm.bh
```

Ikus daitekeen bezala, TSParen kasuan bi ereduak problemaren informazioa ondo adierazteko gai dira, eta, hortaz, bilaketak bi kasuetan aurrera egiten du.

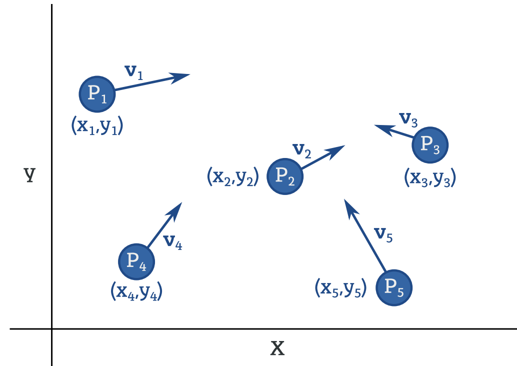
Orain arte ikusi ditugun adibide guztietan feromonak bakarrik erabili ditugu soluzioak eraikitzeko. Oinarrizko algoritmoan horrela izan arren, problema zehatz bat ebatzi behar denean, informazio heuristikoa ere sartu ohi da, posible den kasuetan. Adibide gisa, TSPrako algoritmo eraikitzaile gutiziatsu tipikoan, hurrengo hiria hautatzeko hirien arteko distantzia erabiltzen da. Beraz, goiko adibidean, soluzioak eraikitzean, hiri batetik bestera joateari dagokion feromona kopurua soilik erabili beharrean, hirien arteko distantzia ere kontuan har dezakegu osagai bakoitzaren probabilitatea definitzerakoan.

Particle Swarm Optimization

Intsektu sozialen portaera *swarm* adimenaren adibide tipikoa da, baina ez da bakarra; animalia handiagoetan ere inspirazioa bilatu izan da sarritan. Esate baterako, txori-saldetan ehunka indibiduo era sinkronizatuan mugitzen dira beren artean talkarik egin gabe. Multzo horietan ez dago taldea kontrolatzen duen indibiduorik; txori bakoitzak bere inguruan dauden txorien portaera aztertzen du, eta horren arabera berea egokitzen du. Era horretan, arau sinple batzuk[2] besterik ez dira behar sistema osoa antolatzeko.

Animalia talde horien portaera inspiraziotzat hartuta, 1995ean Kennedy eta Eberhartek *Particle Swarm Optimization* (PSO) algoritmoa proposatu

[2] Txori batetik gertuegi banago, urrundu egiten naiz, adibidez



3.14 irudia. PSO algoritmoak erabiltzen dituen partikulen adibidea. Partikula bakoitzak bere kokapena (x_i, y_i) eta bere abiadura $(\mathbf{v}_i)$ ditu.

zuten [22], optimizazio numerikoko problemak ebazteko.[3] Algoritmoaren ideia sinplea da oso; bilaketa-espazioan barrena mugitzen den partikula multzo bat erabiltzen da bilaketa aurrera eramateko.

Une oro partikula bakoitzak kokapen eta abiadura zehatz bat izango du, 3.14 irudian erakusten den bezala. Partikula bakoitzaren posizioak problemarako soluzio bat adieraziko du. Irudiko adibidean bilaketa-espazioak bi aldagai besterik ez du (X eta Y), eta sisteman 5 partikula daude: P_1, P_2, P_3, P_4 eta P_5.

Bilaketa gauzatzeko, PSO algoritmoaren iterazio bakoitzean partikula guztien kokapena eguneratzen da, haien abiadurak erabiliz. Partikulen abiadurak finko mantenduz gero, denak infinitura joango lirateke. Hori ez gertatzeko, iterazio bakoitzean abiadura ere eguneratu behar da; azken eguneraketa horretan datza, hain zuzen, algoritmoaren gakoa.

Lehen aipatu bezala, algoritmoaren inspirazioa txori-saldoen portaera da. Txori bakoitzak, nora mugitu behar duen erabakitzeko, bere ingurunean dauden txoriei erreparetzen die. Era berean, algoritmoan partikula baten abiadura eguneratzeko, partikula horrek duen informazioa ez ezik, inguruneko partikulek dutena ere erabiltzen da. Hain zuzen ere, i. partikularen abiadura eguneratzeko ondoko ekuazioa aplikatzen da:

$$\mathbf{v}_i(t) = \mathbf{v}_i(t-1) + C_1\rho_1[\mathbf{p}_i - \mathbf{x}_i(t-1)] + C_2\rho_2[\mathbf{p}_g - \mathbf{x}_i(t-1)]$$

Ekuazio hori hiru osagaiz dago osatuta:

- $\mathbf{v}_i(t-1)$ - i. partikulak aurreko iterazioan zeukan abiadura; termino honek partikularen inertzia adirazten du.

[3] Hau da, atal honetan ikusiko dugun algoritmoak bektore errealekin dihardu. Edonola ere, konbinatoriako problemak ebazteko PSO bertsioak ere aurki daitezke.

- $C_1\rho_1[\mathbf{p}_i - \mathbf{x}_i(t-1)]$ - Termino honetan $\mathbf{p}_i$-k i. partikulak bilaketa-prozesuan topatu duen soluziorik onena adierazten du –ingelesez *personal best* deritzona–. Termino honek eguneraketaren alderdi *kognitiboa* adierazten du; hots, partikulak berak jasotako informazioa. C_1 konstantea terminoaren eragina definitzeko erabiltzen da, eta ρ_1 soluzioaren tamainako zorizko bektore bat da.
- $C_2\rho_2[\mathbf{p}_g - \mathbf{x}_i(t-1)]$ - Termino honetan $\mathbf{p}_g$-k i. partikularen inguruan dauden partikulek bilaketa-prozesuan topatu duten soluziorik onena adierazten du –ingelesez *global best* deritzona–. Termino honek eguneraketaren alderdi *soziala* adierazten du; hots, beste partikuletatik jasotako informazioa. C_2 konstantea terminoaren eragina definitzeko erabiltzen da, eta ρ_2 soluzioaren tamainako zorizko bektore bat da.

Ekuazioaren azken terminoak partikulen arteko elkarrekintza simulatzen du. Horretarako, partikulen ingurune-egitura definitu behar da. Kasu honetan, ingurune kontzeptua ez da bilaketa lokalean erabiltzen den berdina, partikula bakoitzaren ingurunea aurrez ezarritakoa baita; ez du partikularen kokapenarekin zerikusirik, alegia. Partikula bakoitzaren ingurunea grafo baten bidez adieraz daiteke, non bi partikula konektatuta dauden baldin eta soilik baldin bata bestearen ingurunean badaude.

Lehenengo hurbilketa grafo osoa erabiltzea da; hots, edozein partikularen ingurunean gainontzeko partikula guztiak egongo dira; hurbilketa sofistikatuago batzuk, grafo osoa erabili beharrean, beste zenbait topologia erabiltzen dituzte (eraztunak, izarrak, toroideak, etab.).

Abiaduren eguneraketari dagokionez, aurreko ekuazioa zuzenean erabiltzen bada, abiadurak dibergitzeko joera izaten du; alegia, abiaduraren modulua gero eta handiagoa izango da. Arazo hori ekiditeko, kalkulatutako abiadurari muga bat ezartzea ohikoa izaten da.

Behin uneko iterazioaren abiadura kalkulatuta, abiadura, partikularen kokapena eguneratzeko erabiltzen da:

$$\mathbf{x}_i(t) = \mathbf{x}_i(t-1) + \mathbf{v}_i(t-1)$$

Iterazio bakoitzean lortutako soluzio –hots, posizio– berriak ebaluatu eta, behar izanez gero, partikulen *personal* ($\mathbf{p}_i$) eta *global best* ($\mathbf{p}_g$) balioak eguneratu behar dira. Urrats horiek guztiak **??** algoritmoan biltzen dira.

Algoritmo hori **metaheuR** paketeko `basicPso` funtzioan inplementaturik dago. Erabilera erakusteko, optimizazio numerikoan *benchmark* gisa erabiltzen den Rosenbrock funtzioa erabiliko dugu; problema sortzeko, `rosenbrockProblem` funtzioa erabiliko dugu.

```
> n <- 10
> rsb.problem <- rosenbrockProblem(size=n)
```

PSO algoritmoa

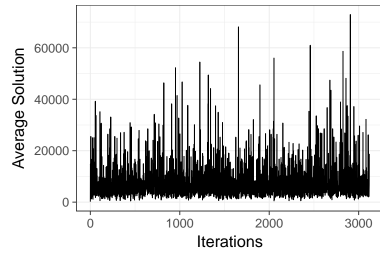
```
1  input: initialize_position, initialize_velocity,
   update_velocity, evaluate eta stop_criterion operadoreak
2  input: num_particles partikula kopurua
3  output: opt_solution
4  gbest = p[1]
5  for each i in 1:num_particles do
6    p[i]=initialize_position(i)
7    v[i]=initialize_velocity(i)
8    pbest[i]=p[i]
9    if evaluate(p[i])<evaluate(gbest)
10     gbest = p[i]
11   fi
12 done
13 while !stop_criterion() do
14   for each i in particle_set
15   do
16     v[i] = update_velocity(i)
17     p[i] = p[i] + v[i]
18     if evaluate(p[i])<evaluate(pbest[i])
19       pbest[i]=p[i]
20     fi
21     if evaluate(p[i])<evaluate(gbest)
22       gbest=p[i]
23     fi
24   done
25 done
26 opt_solution = gbest
```

3.3 algoritmoa. *Particle Swarm Optimization* algoritmoaren sasikodea

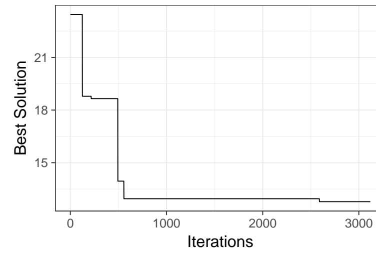
Algoritmoa aplikatu ahal izateko, partikula kopurua, hasierako kokapenak eta abiadurak, abiadura maximoa, *personal best* koefizientea eta *global best* koefizientea ezarri behar ditugu:

```
> nparticles <- 100
> ipos <- lapply(1:nparticles,
+                FUN=function (i) {
+                  return(runif(n))
+                })
> args <- list()
> args$initial.positions  <- ipos
> args$initial.velocity   <- 0
> args$max.velocity       <- 5
> args$c.personal         <- 2
> args$c.best             <- 4
```

Horrez gain, helburu-funtzioa eta baliabide konputazionalak ere finkatu behar ditugu.



(a) Batazbesteko soluzioaren progresioa



(b) Soluzio onenaren progresioa

3.15 irudia. PSO algoritmoaren progresioa Rosenbrock probleman

```
> args$evaluate  <- rsb.problem$evaluate
> args$resources <- cResource(time=10)
>
> res.pso <- do.call(basicPso, args)
>
> plotProgress(res.pso, x="iterations", y="best") +
+   labs(y="Best Solution") + thm.bh
> plotProgress(res.pso, x="iterations") +
+   labs(y="Average Solution") + thm.bh
```

3.15 irudiak bilaketaren progresioa erakusten du. Ezkerreko grafikoan, partikulen batez besteko ebaluazioa erakusten da, iterazioz iterazio; eskubikoan, berriz, bilaketan zehar topatutako soluziorik onenaren *fitness*aren progresioa erakusten da. Beste algoritmoetan ez bezala, partikulen helburu-funtzioen balioek ez dute konbergitzen; hala eta guztiz ere, bilaketak aurrera egiten du, eta, azkenean, optimotik oso hurbil gelditzen da –Rosenbrock funtzioaren balio minimoa 0 da–.

4 Algoritmoen konparazio enpirikoa

Aurreko kapituluetan ikasi dugu problemak formalizatzen eta horiek optimizatzeko erabiltzen diren algoritmo heuristiko eta metaheuristiko esanguratsuenak implementatzen. Problema erreal baten aurrean gaudenean, ordea, ezagutzen ditugun algoritmo guztietatik zein da egokiena (eraginkorrena)? Nola egin behar dugu aukeraketa? Kapitulu honen xedea galdera horiei erantzutea izango da.

No free lunch teoremak [39] dio ez dagoela optimizazio-problema guztiak ebazteko onena den algoritmorik. Beste era batera esanda, problema bakoitzean algoritmo bat izan daiteke egokiena. Horrez gain, algoritmoen parametroei esleitzen dizkiegun balioek ere eragin handia dute haien portaeran eta ondorioz emaitzetan.

Hori dela eta, algoritmo berriak proposatzen direnean, edo problema berrien aurrean gaudenean, algoritmoen eraginkortasuna aztertu behar dugu. Algoritmoen eraginkortasuna konparatzeko bide ohikoenetako bat esperimentazioa da. Ondorengo ataletan optimizazio-arloan esperimentazio-prozesua nola egiten den azalduko dugu, eta kontuan izan beharreko urratsak aztertuko ditugu:

- Problemaren instantzia sorta bat bildu
- Konparazioaren baldintzak definitu
- Parametroen aukeraketa egin
- Algoritmoak exekutatu eta emaitzak aztertu
- Konparazio grafikoa egin
- Analisi estatistikoa gauzatu

Esperimentazio-prozesua errazago ulertzeko, jarraian azaltzen den adibidearekin ilustratuko ditugu urrats guztiak.

4.1 adibidea. *Demagun Saltzaile Bidaiariaren Problema (TSP) optimizatzeko algoritmorik eraginkorrena aukeratu nahi dugula. Zehazki, instantzia txikientzako (100 hiri baino gutxiagokoak) emaitzarik onena itzultzen digun algoritmoa aukeratu nahi dugu.*
Aditu batek bilaketa lokala (LS), algoritmo genetikoak (GA) eta inurri-kolonien optimizazio-algoritmoak (ACO) konparatzeko esan digu, TSPn oso eranginkorrak direla argudiatuz. Adituaren gomendioa aintzat hartuta, 3 algoritmo horietatik onena zein den erabakitzen lagunduko digun konparazio esperimentala egingo dugu.

Problemaren instantzien aukeraketa

Esperimentazio-prozesuan, lehenik eta behin, algoritmoak konparatzeko erabiliko ditugun problemaren instantziak aukeratu behar ditugu. Sarreran esan dugun bezala, algoritmoen eraginkortasuna aldatu egiten da problema mota batetik bestera; gauza bera gertatzen da problema baten instantzia batetik bestera. Algoritmo bat TSP problema bat ebazteko oso ona izan arren, aukera txarra izan daiteke QAP problema bat ebazteko. Era berean, 100 tamainako TSP simetriko bat ebazteko algoritmo onenak ez du zertan onena izan 1000 tamainako TSP asimetriko bat optimizatzeko. Hortaz, esperimentuak egin aurretik, zer nolako instantziak ebatzi nahi ditugun erabaki behar dugu, hau da, ezaugarri batzuk finkatu.

Behin ezaugarriak erabakita, eredugarria den instantzia multzo bat bildu behar da, ebatzi nahi dugun instantzia motaren antzeko problema-aleak aukeratuz, ahal den neurrian. Problemak testuinguru erreal batean ebatzi behar badira, testuinguru horretan sortutako benetako instantziak erabiltzea da egokiena. Instantzia errealak lortzeko aukerarik eduki ezean, bi aukera daude:

1. **Instantziak simulatzea.** Aukeretako bat, instantzia errealen antzerakoak artifizialki sortzea da. Kasu horretan, benetakoen ahal bezain antzerakoak diren instantziak simulatu nahi ditugunez, problema sakonki aztertu beharko dugu.
2. ***Benchmark*ak erabiltzea.** Bigarren aukera, publikoak diren instantzien bildumak edo *benchmark*ak erabiltzea da. Aukera horrek abantailak eta desabantailak dauzka. Alde batetik, *benchmark*ak asko erabiltzen dira eta, hortaz, bertan dauden instantzientzat oso soluzio onak (optimoak, kasu batzuetan) ezagutzen dira. Ikerketa-arloan, algoritmo berri baten portaera aztertzea da *benchmark*ak erabiltzeko testuinguru egokia. Bestalde, bilduma horietan dauden instantziak problema klasikoenak dira, eta, hortaz, testuinguru errealetarako ez dira oso erabilgarriak izango.

4.2 adibidea. *Gure adibidean, aukeratutako hiru algoritmoek –bilaketa lokalak (LS), algoritmo genetikoak (GA) eta inurri-kolonia algoritmoak (ACO)– 100 tamainako edo gutxiagoko instantzietan duten portaera aztertu nahi dugu. Testuinguru teoriko batera mugatuko gara, eta, beraz, TSPrako instantziak biltzen dituen TSPlib* benchmark*era joko dugu zuzenean.*[a] *Zehazki, instantzia hauek aukeratu ditugu:*

gr24	*bays29*	*att48*
eil51	*berlin52*	*brazil58*
st70	*eil76*	*rat99*

[a] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

Instantziak zuzenean TSPlib biltegitik kargatzeko **metaheuR** paketeko `tsplibParser` funtzioa erabiliko dugu, aurreko kapituluetan bezala:

```
> instances <- c("gr24.xml.zip", "bayg29.xml.zip", "att48.xml.zip",
+           "eil51.xml.zip", "berlin52.xml.zip","brazil58.xml.zip",
+           "st70.xml.zip","eil76.xml.zip","rat99.xml.zip")
>
> tsplib.problems <- lapply (instances,
+     FUN=function(x){mat <- tsplibParser(
+       paste("http://comopt.ifi.uni-heidelberg.de/software",
+             "/TSPLIB95/XML-TSPLIB/instances/", x, sep=""))
+       return(tspProblem(mat))
+     })
```

Horrenbestez, TSP problemaren 9 instantzia horiek `tsplib.problems` izeneko zerrendan gorde ditugu, gero algoritmoak aplikatu ahal izateko.

Konparazioaren baldintzak

Optimizazio-algoritmo batzuk konparatzen hasi baino lehen, ezinbestekoa da zenbait baldintza ezartzea, alderaketa bidezkoa izan dadin.

Hasteko, esperimentazioaren helburua algoritmoak konparatu eta egokiena aukeratzen laguntzea da, baina zer esan nahi du algoritmo bat egokia izateak? Galdera horren erantzuna ez da bistakoa, testuinguru bakoitzean beharrak ezberdinak izaten baitira. Diseinu-problemetan, esaterako, lortutako soluzioa (diseinua) behin eta berriz erabiliko da, eta, beraz, denbora gehiago eman dezakegu optimotik albait hurbilen dagoen soluzio bat lortzeko. Kontrol-problemetan, aldiz, oso soluzio onak lortzea baino garrantzitsuagoa izaten da ahalik eta denbora laburrenean 'nahiko onak' diren soluzioak lortzea.

Gainera, algoritmoen errendimendua modu bidezkoan neurtzeko, esperimentazioa diseinatzeko garaian, **algoritmo guztientzat gelditze-irizpide**

berdina finkatu behar da. Mundu errealeko aplikazioetan denbora izan ohi da irizpide erabiliena. Irizpide hori, ordea, erabilitako programaziolengoaiaren, hardwarearen eta inplementazioaren kalitatearen menpekoa da. Horregatik, ikerketa-arloan, helburu-funtzioaren ebaluazio-kopuru maximo bat erabili ohi da gelditze-irizpide gisa.

Azkenik, esperimentazioaren baldintzak guztiz definituta geratzeko, **algoritmoak zein metrikaren arabera konparatuko ditugun erabaki behar dugu**. Ohikoena, algoritmoek itzultzen dituzten azken emaitzak (*fitness*ak) konparatzea da. Batzuetan, ordea, soluzioaren *fitness*a gordinean erabili beharrean, beste erreferentziazko balio batekiko diferentzia erlatiboa kalkulatzen da (erreferentzia-algoritmo bat dugunean edo soluzio optimoa ezaguna denean, esate baterako).

Demagun algoritmo bat instantzia bati aplikatu diogula eta lortu duen *fitness*-balioa 125 dela. Instantzia horretarako orain arte ezagutzen den soluziorik onena 130 bada, gure algoritmoaren performantzia $\frac{130-125}{130} = 0.038$ edo, ehunekotan jarrita, %3.8 da. Metrika horren arabera gure algoritmoak ezagutzen den soluziorik onena %3.8 hobetu du.

Aurreko planteamendua zuzena izango da konparatzen ditugun algoritmoak deterministak baldin badira, hau da, nahi adina aldiz exekutatuta ere, beti emaitza bera itzultzen badute. Baina zer gertatuko da algoritmoak estokastikoak badira? Algoritmo mota horiek zorizko osagai bat dute, eta, beraz, exekuzio bakoitzean emaitza desberdin bat eman dezakete. Hori dela eta, algoritmo estokastikoak konparatzerakoan hainbat errepikapen egitea beharrezkoa da, jasotako emaitzak estatistikoen bitartez laburbilduz. **Algoritmo estokastikoak zenbat aldiz exekutatu behar diren ere aurrez erabaki beharko dugu**.

4.3 adibidea. *Gelditze-irizpidearen aukeraketari zein errepikapen-kopuruari dagokienez, ez dago arau estandarrik. Gure adibidean, algoritmoak (LS, GA eta ACO) helburu-funtzioaren 1000 ebaluazioren ondoren geldituko ditugu, eta algoritmo/instantzia bikote bakoitza 10 aldiz exekutatuko dugu. Ebaluazio-metrika gisa fitness gordina erabiliko dugu, eta baita emaitza onenak lortu dituen algoritmoarekiko fitness normalizatua ere.*

Parametroen aukeraketa

Behin baino gehiagotan esan dugu, algoritmoen parametroentzako aukeratzen ditugun balioek eragin handia dutela haien portaeran. Horrnebestez, parametro egokien aukeraketa, edo *tuning* ingelesez, esperimentazioaren ezinbesteko urratsa da. Estrategia sinpleena (guk adibidean erabiliko duguna) parametro-balio batzuk eskuz finkatzea da, aurretiko ezagutza edo aditu ba-

ten iritzia erabiliz. Alabaina, zaila izaten da parametroen balioen eragina aldez aurretik jakitea. Hori dela eta, aukeraketa mota hori ez da oso egokia eta ez da normalean erabiltzen.

Izan ere, parametroen aukeraketa egiteko hainbat estrategia sofistikatu eta egokiagoak aurki ditzakegu literaturan [3, 4, 25]. Horien konplexutasuna dela eta, liburu honetan soilik pare bat adibide aipatuko ditugu laburki.

Estrategiarik ohikoena hainbat parametro-konbinazio probatzea eta alderatzea da. Kontuan hartuko diren parametro konbinazio posibleak aukeratzeko hainbat estrategia existitzen dira, ohikoena ingelesez *full factorial* deritzona izanik. Parametro bakoitzerako aukera zerrenda bat egin ostean, parametro guztien konbinazio guztiak aztertzen dira. Diseinu horrekin parametroen espazioa ondo arakatzen dugu, baina behar diren proben kopurua esponentzialki hazten da. Beste aukera bat parametroen egokitzapena optimizazio-problema bat bezala tratatzea da. Hala, badaude optimizazio-prozedura bereziak parametroen *tuning*a egiteko. Horietako bat **iRace**[28] paketean dago inplementatuta.

Metodo sofistikatuagoak erabiltzen baditugu, parametroen aukeraketa esperimentazio-prozesu orokorraren azpiesperimentazio bat bezala ikus dezakegu, eta, beraz, horretarako instantziak behar ditugu. Alabaina, ezin ditugu esperimentazio orokorrean erabiliko ditugun instantzia berdinak erabili, horrek emaitzetan alborapena ekar baitezake.[1] Beraz, instantzia ezberdinak erabili behar dira, baina ahal den neurrian esperimentazio orokorrean erabiliko ditugunen antzerakoak. Gehiago sakondu nahi duen irakurleak ondoko erreferentziako lanetara jo dezake [4, 25].

4.4 adibidea. *Gure adibidean aukeraketa mota simpleena egingo dugu, hau da, parametro batzuk eskuz finkatuko ditugu, inolako esperimentu gehigarririk egin gabe. Hala ere, kontuan izan metodo horrek ez dituela inolaz ere emaitzarik efizienteenak lortuko, eta beraz, ikerketa-testuinguru edo problema erreal bat optimizatu nahiko bagenu, beste hurbilketa aurreratuago bat erabili beharko genuke. Egindako aukerak ondokoak dira:*

- ***Local Search algoritmoa:***
  - *Hasierako soluzioa:* `randomPermutation` *funtzioaren bidez sortutako zorizko permutazio bat izango da.*
  - *Ingurunearen definizioa: ondoz ondoko trukaketetan oinarritutako ingurunea erabiliko dugu (*`SwapNeighborhood`*).*

[1] Instantzia berdinak erabiliz gero, aukeratutako parametroak justu instantzia horientzat egokienak izango dira. Beraz, lortuko ditugun emaitzak beste edozein instantzia orokorretan izango liratekeen baina hobeak izango dira ziurrenik, eta emaitzen orokortasuna murriztuko da.

  - *Inguruneko soluzio bat aukeratzeko prozedura: prozedura gutiziatsua aplikatuko dugu, beti inguruneko soluziorik onena aukeratzen duena (*`greedySelector` *klasearen bidez).*
- ***Algoritmo genetikoa:***
  - *Hasierako populazioa: zorizko 100 permutaziok osatuko dute.*
  - *Hautespen-operadorea: aukeraketa elitista aplikatuko dugu.*
  - *Mutazioa: soluzioak 0.01eko probabilitatearekin mutatuko dira, eta* `swapMutation` *klasea erabiliko da mutazioa egiteko, permutazio bateko posizioen %20 trukatuz.*
  - *Gurutzaketa: populazioaren erdia aukeratuko da lehiaketa bidez, eta horiek gurutzatuko dira permutazioentzat berezia den* `orderCrossover` *klasea erabiliz. Irakurri paketearen laguntza-orriak gurutzaketa horren nondik norakoak ulertzeko.*
- ***Ant Colony optimization:***
  - *Inurri-kopurua: 100 inurri.*
  - *Feromona-eredua:* `VectorPheromone` *klasea erabiliko dugu, hasierako feromona-kopurua konstantea izanik eta 0.1eko lurrunketa maila ezarriz.*
  - *Feromona-ereduaren eguneraketa: iterazioko soluziorik onena aukeratuko da eta eguneraketa guztietan 0.1eko balio finkoa gehituko zaio feromona-ereduari.*

Hiru kasuetan, baliagarriak ez diren soluzioak baztertuko ditugu `non.valid` *parametroari* `"discard"` *balioa emanez.*

Orain, aurreko ataleko erabaki guztiak kontuan hartuz, aukeratutako hiru algoritmoak konparatzeko esperimentazioa egiteko kodea implementatuko dugu. Hasteko, bilaketa lokalaren esperimentazioa kodetzen da:

```
> num.rep <- 10
> num.evaluations <- 1000
> resources <- cResource(evaluations=num.evaluations)
> runLS <- function (problem) {
+   # Arguments to run the local search with swap neigh.
+   args <- list()
+   args$evaluate     <- problem$evaluate
+   args$selector     <- greedySelector
+   args$non.valid    <- "discard"
+   evaluations       <- vector()
+   args$do.log       <- FALSE
+   args$verbose      <- FALSE
+   args$resources    <- resources
+   evaluations <- vector()
+   for (i in 1:num.rep) {
+     init.sol <- randomPermutation(problem$size)
+     args$initial.solution <- init.sol
```

```
+     args$neighborhood <- swapNeighborhood(init.sol)
+     message("Running Swap LS, repetition #", i)
+     res <- do.call(basicLocalSearch, args)
+     evaluations <- c(evaluations, getEvaluation(res))
+   }
+   return(evaluations)
+ }
```

Era berean, antzerako kodeak erabiliko ditugu algoritmo genetiko eta inurri-kolonia algoritmoen kasuan:

```
> runGA <- function (problem) {
+ n <- problem$size
+ initial.population.size <- 100
+   # Arguments to run the genetic algorithm.
+   args <- list()
+   args$evaluate              <- problem$evaluate
+   args$selectSubpopulation  <- elitistSelection
+   args$selection.ratio       <- 0.5
+   args$selectCross           <- tournamentSelection
+   args$mutate                <- swapMutation
+   args$ratio                 <- 0.2
+   args$mutation.rate         <- 1 / length(initial.population.size)
+   args$cross                 <- orderCrossover
+   args$non.valid             <- "discard"
+   args$do.log                <- FALSE
+   args$verbose               <- FALSE
+   args$resources             <- resources
+   evaluations                <- vector()

+   for (i in 1:num.rep) {
+     initial.pop         <- lapply(1:initial.population.size,
+                                FUN=function(x) {
+                                rnd.perm <- randomPermutation(n)
+                                return(rnd.perm)
+                                })
+     args$initial.population <- initial.pop
+     message("Running GA, repetition #", i)
+     res <- do.call(basicGeneticAlgorithm, args)
+     evaluations <- c(evaluations, getEvaluation(res))
+   }
+   return(evaluations)
+ }
>
> runACO <- function (problem) {
+   n <- problem$size
+   # Arguments to run the ant colony optimization.
+   args <- list()
+   args$evaluate      <- problem$evaluate
+   args$nants         <- 100
+   init.value         <- 1
+   initial.trail      <- matrix(rep(init.value, n*n), nrow=n)
+   evapor             <- 0.1
```

```
+   pher <- permuPosPheromone(initial.trail=initial.trail,
+                              evaporation.factor=evapor)
+   args$pheromones   <-  pher
+   args$update.sol   <- "best.it"
+   args$update.value <- init.value / 10
+   args$non.valid    <- "discard"
+   args$do.log       <- FALSE
+   args$verbose      <- FALSE
+   args$resources    <- resources
+   evaluations       <- vector()
+   for (i in 1:num.rep) {
+     message("Running ACO, repetition #", i)
+     res <- do.call(basicAco, args)
+     evaluations <- c(evaluations, getEvaluation(res))
+   }
+   return(evaluations)
+ }
```

Algoritmoak exekutatu eta emaitzak aztertu

Esperimentazioa egiteko beharrezkoak diren alderdiak finkatu ditugunean, aukeratutako algoritmo guztiak aplikatuko dizkiogu, ondoko kodea erabiliz:

```
> res.ls  <- lapply(tsplib.problems, FUN=runLS)
> res.ga  <- lapply(tsplib.problems, FUN=runGA)
> res.aco <- lapply(tsplib.problems, FUN=runACO)
```

Esperimentazio egokia egin badugu, exekuzio kopurua oso handia izango da (algoritmoak $\times$ instantziak $\times$ errepikapenak). Horrenbestez, lortutako emaitzetatik ondorioak atera ahal izateko, datu gordinak prozesatu eta laburbildu beharko ditugu. Horretarako, lehenik eta behin, emaitza guztiak (behar dugun informazio gehigarriarekin batera) *data frame* batean gordeko ditugu.

Hasteko, bilaketa lokalaren emaitzak bilduko ditugu, lerro bakoitzean instantzia eta errepikapen baten datuak gordeaz:

```
> instance.names <- c("gr24", "bays29", "att48", "eil51",
+                     "berlin52", "brazil58", "st80",
+                     "eil76", "rat99")
>
> aux.ls <- lapply(1:length(res.ls),
+                  FUN=function(i){
+                    r <- cbind(instance.names[i],
+                               res.ls[[i]],
+                               1:length(res.ls[[i]]))
+                    return(r)
+                  })
> res.ls <- do.call(rbind, aux.ls)
> head(res.ls)
```

```
##      [,1]   [,2]   [,3]
## [1,] "gr24" "3006" "1"
## [2,] "gr24" "2543" "2"
## [3,] "gr24" "2594" "3"
## [4,] "gr24" "2696" "4"
## [5,] "gr24" "2704" "5"
## [6,] "gr24" "2842" "6"
```

Ondoren, antzeko prozesu bati jarraituko diogu beste bi algoritmoen emaitzak biltzeko:

```
> aux.ga <- lapply(1:length(res.ga),
+                  FUN=function(i){
+                    r <- cbind(instance.names[i],
+                               res.ga[[i]],
+                               1:length(res.ga[[i]]))
+                    return(r)
+                  })
> res.ga <- do.call(rbind, aux.ga)
>
> aux.aco <- lapply(1:length(res.aco),
+                   FUN=function(i){
+                     r <- cbind(instance.names[i],
+                                res.aco[[i]],
+                                1:length(res.aco[[i]]))
+                     return(r)
+                   })
> res.aco <- do.call(rbind, aux.aco)
```

Azkenik, hiru algoritmoentzat sortutako taulak `data.frame` bakar batean elkartuko ditugu:

```
> results.df <- rbind(data.frame(res.ls,
+                                Algorithm="LS"),
+                     data.frame(res.ga,
+                                Algorithm="GA"),
+                     data.frame(res.aco,
+                                Algorithm="ACO"))
> names(results.df) <- c("Instance", "Fitness",
+                        "Repetition", "Algorithm")
> head(results.df)
```

```
##   Instance Fitness Repetition Algorithm
## 1     gr24    3006          1        LS
## 2     gr24    2543          2        LS
## 3     gr24    2594          3        LS
## 4     gr24    2696          4        LS
## 5     gr24    2704          5        LS
## 6     gr24    2842          6        LS
```

Horrenbestez emaitza guztiak egitura batean gorde ditugu eta horiek aztertzen has gaitezke. Halere, kontuan hartu behar dugu sortutako egituran

zutabe guztiak kategorikoak direla. Alabaina, guk, emaitzak aztertu ahal izateko, "Fitness"izeneko zutabea `numeric` motakoa izatea behar dugu:

```
> aux <- as.character(results.df$Fitness)
> results.df$Fitness <- as.numeric(aux)
```

Esan bezala, datuak zuzenean interpretatzea ez da berehalakoa izaten, batez ere exekuzio-kopurua handia bada. Hori hala izanik, lehenengo urratsa datuak laburbiltzea izango da. Hasteko, algoritmo bakoitzaren emaitzak zutabe batean jarriko ditugu:

```
> id.LS <- results.df$Algorithm=="LS"
> data.test <- results.df[id.LS, c(1,2)]
> names(data.test)[2] <- "LS"
> id.GA <- results.df$Algorithm=="GA"
> data.test$GA <- results.df[id.GA, 2]
> id.ACO <- results.df$Algorithm=="ACO"
> data.test$ACO <- results.df[id.ACO, 2]
> head(data.test)
```

```
##   Instance   LS   GA  ACO
## 1     gr24 3006 2229 2710
## 2     gr24 2543 2264 2346
## 3     gr24 2594 2291 2658
## 4     gr24 2696 2335 2524
## 5     gr24 2704 2222 2430
## 6     gr24 2842 2557 2663
```

Gainera, algoritmo eta instantzia bakoitzarentzat, 10 errepikapenetan lortutako helburu-funtzioen mediana kalkulatuko dugu, **scmamp** paketeko `summarizeData` funtzioa erabiliz [9]:

```
>  data.summarized <- summarizeData(data=data.test,
+                                   fun=median,
+                                   group.by="Instance")
> data.summarized
```

```
##     groups        LS        GA        ACO
## 1     gr24  2700.500  2277.500   2653.000
## 2   bays29  3586.000  3239.000   3536.000
## 3    att48 37535.500 35552.000  38598.000
## 4    eil51  1302.304  1290.575   1353.275
## 5 berlin52 23897.218 22696.687  24639.090
## 6 brazil58 98160.000 90810.500 101175.000
## 7     st80  2917.966  2923.613   3080.199
## 8    eil76  2021.326  2046.807   2160.971
## 9    rat99  7219.396  6709.807   7140.604
```

Taula honetan errazago ikus dezakegu zein algoritmok lortzen dituen emaitzarik onenak instantzia bakoitzean. Adibidez, ikus dezakegu GA algoritmoak beste biek baino emaitza hobeak lortzen dituela instantzia guztietan `eil51` izenekoan izan ezik.

Konparazio grafikoa

Aurreko atalean emaitzak laburbiltzeko eta bistaratzeko tresna erabilgarri batzuk ezagutu ditugu: taulak. Hala ere, esperimentazioan erabilitako instantzia kopurua oso altua denean (100 edo 1000, adibidez), grafikoak egitea taulak erabiltzea baino erabilgarriagoa izan daiteke. Grafikoek, informazioa laburbiltzeko malgultasun handia eskaintzen dute, eta, askotan, emaitzen arteko erlazioei edota patroiei antzemateko ezinbesteko tresnak dira.

Grafiko bat egin aurretik, argi izan behar dugu zer erakutsi nahi dugun, alegia, zein den grafikoaren helburua. Horren arabera, grafikoaren hainbat alderdi finkatu beharko ditugu: zein datu erabiliko ditugu?, zein motatako grafikoa izango da?, grafiko bakarra edo grafiko multzo bat egingo dugu?

Atal honetan grafikoak **ggplot2** paketearen bidez egingo ditugu. Pakete honen atzean dagoen filosofia nahiko berezia da. Alde batetik, erakutsi nahi ditugun aldagaiak ditugu (emaitzak, denbora, etab.). Bestetik, grafiko motak definituta dakartzan elementu estetikoak ditugu (kokapena, koloreak, etab.). Grafikoa egiteko aldagaiak eta elementu estetikoak "mapeatu" behar ditugu, alegia, aukeratutako aldagai bakoitzari elementu estetiko bat esleitu.

4.5 adibidea. *Adibide gisa, lehen egin dugun esperimentazioaren emaitzak bistaratzeko, instantzia bakoitzeko hiru algoritmoen medianak erakusten dituen barra-grafiko bat egin dezakegu. Kasu honetan, barra bakoitzak hiru ezaugarri estetiko izango ditu: barraren kokapena OX ardatzean, haren kolorea eta altuera (zabalera ere erabil genezake, baina adibide honetan finko mantenduko dugu).*
Beraz, hiru aldagai ditugu (instantzia, algoritmoa eta lortutako emaitzen mediana) eta hiru elementu estetiko (kokapena, kolorea eta altuera) eta ondoko eran egingo dugu "konexioa":

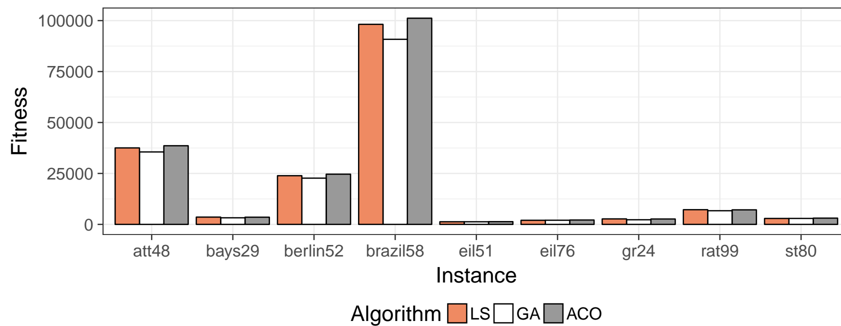
Algoritmoa-Kolorea
Instantzia-OX ardatzean kokapena
Emaitzen mediana-Altuera

ggplot2 paketearen bitartez aldagaien eta ezaugarri estetikoen esleipena modu esplizituan egin dezakegu honela:

```
> map <- aes(x=Instance, y=Fitness, fill=Algorithm)
```

Behin hori eginda, grafikoa bera irudikatzeko ondoko kodea exekutatu beharko dugu:

```
> ggplot(data=results.df, mapping=map) +
+   geom_bar(stat="summary", fun.y=median,
+            position="dodge", color="black") +
+   scale_fill_brewer(palette="RdGy") + thm.bh
```



4.1 irudia. LS, GA eta ACO algoritmoek TSPko instantzietan lortutako batez besteko emaitzak.

Sortutako grafikoa 4.1 irudian erakusten da. Ikus dezakegunez, instantzietan lortutako emaitzak oso ezberdinak dira. Aukeratutako instantzien tamaina antzerakoa izan arren, datuak desberdinak dira, eta, ondorioz, lortutako emaitzak eskala ezberdinetan daude; hots, ez dira zuzenean konparagarriak. Instantzietan lortutako balioak konparatu ahal izateko, emaitzak normalizatu egiten dira erreferentziazko balio batekiko. Kasu batzuetan (*benchmark* klasikoetan, adibidez), ezagutzen den soluziorik onena hartzen da erreferentziatzat (batzuetan soluzio hori optimoa bera da). Horrelako erreferentziarik ez balego, beste algoritmo baten emaitzak (problema hori ebazteko ezagutzen den algoritmorik onena, adibidez) erabili ohi dira.

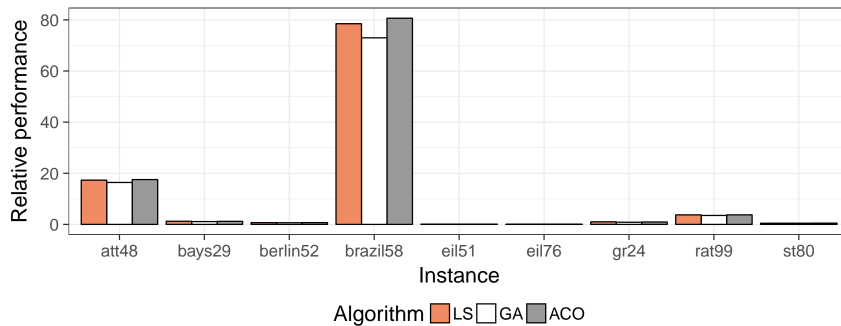
4.6 adibidea. *Aurreko grafikoaren eta aurreko ataletan ikusitako taulen arabera, GA algoritmoa eraginkorrena dela ematen du. Beraz, erreferentzia gisa instantzia bakoitzean GA algoritmoak 10 errepikapenetan lortutako emaitzarik onena hartuko dugu.*

Datuak erreferentzia horrekiko normalizatzeko, **scmamp** paketeak dakarren `summarizeData` funtzioaz baliatuko gara berriro:

```
> aux <-summarizeData(results.df[results.df$Algorithm=="GA", 1:2],
+                      fun=min, group.by=1)
> best.results <- aux[,2]
> names(best.results) <- aux[,1]
```

Balioen atzipena errazteko, sortu dugun bektorearen elementuei izenak emango dizkiegu. Eta, ondoren, emaitzak normalizatuko ditugu.

```
> results.df.norm <- results.df
> results.df.norm$Fitness <- results.df.norm$Fitness /
+   best.results[results.df.norm$Instance]
```



4.2 irudia. LS, GA eta ACO algoritmoek TSPko instantzietan lortutako batez besteko emaitzak normalizatuta onenarekiko.

Eta grafikoa berriro sortuko dugu, oraingoan normalizatutako emaitzak erabiliz:

```
> map <- aes(x=Instance, y=Fitness, fill=Algorithm)
```

Grafiko berria 4.2 irudian erakusten da. Orain instantzietan lortutako emaitzak konparagarriak dira.[2] Argi ikusten da errorerik handiena `brazil58` instantzian lortzen dutela 3 algoritmoek. Bestalde, GA algoritmoak emaitzarik onenak lortzen ditu oro har.

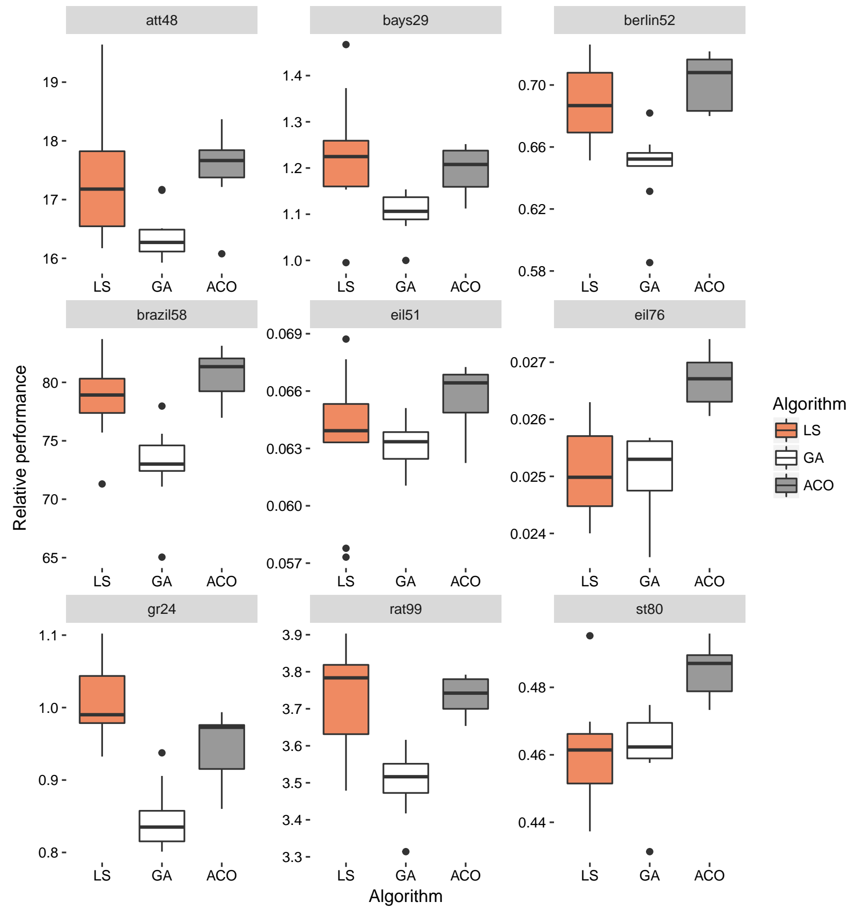
Orain arte, algoritmoen eraginkortasuna errepikapenetan lortutako emaitzen medianaren bidez neurtu dugu. Batzuetan, ordea, joera zentraleko estatistikoez gain (mediana, moda, batezbestekoa), errepikapenetako emaitzen sakabanapena aztertzea ere interesgarria izaten da. Horrelako kasuetan oso ohikoa da *boxplot*ak edo kutxa-grafikoak erabiltzea. Grafiko horiek emaitzen inguruko informazio gehiago deskribatzeko ahalmena dute, mediana, kuartilak eta outlierrak erakusten baitizkigute.

4.7 adibidea. *Gure datuen kutxa-diagramak egiteko, berriro ere estetikoen eta aldagaien arteko "konexioa" egin behar dugu.*

Algoritmoa-Kolorea eta OX ardatzean posizioa
Helburu-funtzioaren balioen banaketa-OY aldagaian kutxaren itxura

Beraz, algoritmo bakoitzaren emaitzak alboz albo agertuko diren koloretako kutxen bidez adieraziko dira. Gainera, kasu honetan grafiko bakar bat egin beharrean, instantzia bakoitzerako grafiko bat egingo dugu.

[2] Emaitzak hobeto bistaratzeko, y ardatza eskala logaritmikoan jar dezakegu, grafikoa sortzen duen kodeari `scale_y_log10()` gehituz.



4.3 irudia. LS, GA eta ACO algoritmoek TSPko instantzietan lortutako batez besteko emaitzak.

Ondoko kodearekin aurreko adibideko irudia eraiki dezakegu:

```
> map <- aes(x=Algorithm, y=Fitness, fill=Algorithm)
> ggplot(data=results.df.norm, mapping=map) +
+   geom_boxplot(position="dodge") +
+   labs(y="Relative performance") +
+   scale_fill_brewer(palette="RdGy") +
+   facet_wrap(~Instance, scales = "free") + thm.bh
```

4.3 irudiak kodearen emaitza jasotzen du. Esan bezala, kutxa-diagramak oso grafiko erabilgarriak dira algoritmoen emaitzak zenbateraino sakabanatuta dauden aztertzeko. Irudiaren arabera, *GA* algoritmoak emaitza onenak izateaz gain, sakabanapen txikia du. Aldiz, *LS* algoritmoaren kutxak nahiko

4.1 taula. Test estatistiko bateko bi errore motak.

	H_0 errefusatzea	H_0 ez errefusatzea
H_0 egia	I motako errorea (negatibo faltsua)	Erabaki egokia
H_1 egia	Erabaki egokia edo potentzia	II motako errorea (positibo faltsua)

handiak dira, emaitzak sakabanatuagoak daudela adieraziz (ziurrenik, errepikapenetan algoritmoa optimo lokal desberdinetara heltzen delako).

Kapitulu honetan, barra eta kutxa-grafikoetan zentratu gara bakarrik, baina badira grafiko mota. Gai honetan sakontzeko, **ggplot2** paketearen liburua [38] gomendatzen dugu.

Test estatistikoak

Aurreko ataletan ikusi ditugun taula zein grafikoek algoritmoen eraginkortasuna ilustratzen dute, eta ondorioak ateratzen laguntzen digute. Ildo beretik, badira arlo estatistikoan emaitzak aztertzen lagunduko diguten beste tresna batzuk ere; aipagarrienak test estatistikoak dira.

Test estatistikoen helburua, lagin edo datu sorta bat oinarritzat hartuta, hipotesi bat testatzea da; alegia, hipotesi hori ontzat hartuko dugun edo ez erabakitzea.

4.1 definizioa. *Hipotesi-konstraste bat bi hipotesik osatuko dute:*

H_0*: ezeztatu nahi dugun hipotesia da (hipotesi nulua).*
H_1*: H_0-ren kontrako hipotesia da (hipotesi alternatiboa).*

Bi hipotesi horiek kontuan hartuz, test estatistiko bat aplikatzean bi errore mota egiteko arriskua dago (ikusi 4.1 taula), eta, errore mota horietatik abiatuz, ondoko kontzeptuak definitzen dira:

$$\alpha = \text{I motako errorearen probabilitatea edo adierazgarritasuna}$$

$$\beta = \text{II motako errorearen probabilitatea}$$

Teorian, hoberena α eta β txikienak jaulkitzen dituzten testak erabiltzea litzateke. Alabaina, I eta II motako erroreak kontrajarriak dira. Beraz, normalean α (edo adierazgarritasuna) finkatu ohi da, eta, ondoren, posible denean, β ahalik eta txikiena duen testa aukeratzen da. Izanez, β balioa barik $1-\beta$ edo testaren potentzia erabiltzen da, hori baita H_1 egia denean hipotesi nulua errefusatzeko probabilitatea.

Hipotesiak egiaztatzeko test estatistiko mota ugari existitzen dira [32]. Beraz, datuak (emaitzak) aztertzerakoan, helburuaren arabera egokia den testa aukeratu beharko da. Gainera, test mota bakoitzak datuen gaineko hainbat suposizio egiten ditu. Horiek horrela, bi test familia bereizten dira bereziki: parametrikoak eta ez-parametrikoak. Lehenengoek datuak (emaitzak) banaketa probabilistiko ezagun batetik datozela jotzen dute; adibidez, banaketa normal batetik. Mota horretako testik ezagunenak *t-test* edo ANOVA testak dira. Bigarrenek, ordea, ez dute datuen banaketaren inguruko baldintzarik ezartzen.

Gure kasuan, bi algoritmoren edo gehiagoren emaitzak alderatzeko erabiliko ditugu testak. Oro har, optimizazioko esperimentuetan test parametrikoak aplikatzeko beharrezko baldintzak nekez betetzen dira, eta horregatik test ez-parametrikoak erabili ohi dira, datuen banaketaren inguruko suposiziorik egiten ez dutelako.

Zehazki, gure helbururako egokiak diren bi test ez-parametriko ikusiko ditugu atal honetan: Wilcoxon-en testa [7], bi lagin (emaitza) konparatzeko, eta Friedman-en testa [7], bi lagin baino gehiago alderatzeko.

Bi emaitza-bektore alderatzen

Instantzia multzo baten gainean bi algoritmo exekutatzen ditugunean, bakoitzeko, emaitza sorta bat lortzen dugu. Algoritmoen portaera berdina bada (antzerako emaitzak itzultzen badituzte, alegia), bi balio sorta horiek probabilitate-banaketa berdinetik datozela esan dezakegu. Hori da, hain zuzen, Wilcoxon-en testak aztertzen duena (H_0=bi algoritmoek portaera berdina dute).

Egin nahi dugun konparazioaren arabera, emaitzak bi motatakoak izan daitezke, askeak edo binakakoak. Lehenengo kasuan, algoritmo baten emaitza bakoitzari beste algoritmoaren emaitza bat eta bakarra dagokio. Hori gertatzen da, adibidez, emaitzak problemaren instantzia desberdinen soluzioak direnean (hau da, instantzia bakoitzeko bi algoritmoen emaitzak ditugunean). Emaitzak instantzia bakar baterako bi algoritmoen zorizko errepikapenetatik lortutako balioak direnean, ordea, lagin askeak ditugula esango dugu, bien artean ez baitago erlaziorik edo parekatzeko aukerarik.

Bi egoera horiek Wilcoxonen testarekin azter daitezke (datu ez-parekatuen kasuan, Mann-Whitney izena ematen zaio batzuetan). Rn `wilcox.test` funtzioak bi test horiek aplikatzea ahalbidetzen digu. Ikus dezagun, adibide pare batekin, testa nola aplikatu. Lehenengo adibidean banaketa uniforme berdinari jarraitzen dioten bi zorizko lagin sortuko ditugu (beraz, testak diferentziarik ez dagoela esan beharko liguke), eta bigarrenean, laginak banaketa desberdinetatik aterako ditugu: bat uniformea eta bestea normala. Bi kasue-

tan lagin askeak konparatzen ari gara, ez baitago haien artean erlaziorik edo parekatzeko aukerarik.[3]

```
> sample.1 <- runif(30)
> sample.2 <- runif(30)
> wilcox.test(sample.1, sample.2)

##
##  Wilcoxon rank sum test
##
## data:  sample.1 and sample.2
## W = 421, p-value = 0.6757
## alternative hypothesis: true location shift is not equal to 0

> sample.3 <- rnorm(30, 1, 1)
> wilcox.test(sample.1, sample.3)

##
##  Wilcoxon rank sum test
##
## data:  sample.1 and sample.3
## W = 226, p-value = 0.00073
## alternative hypothesis: true location shift is not equal to 0
```

Testak diferentziarik dagoela dioenetz jakiteko, p-balioari erreparatuko diogu. H_0 egiazkoa dela jota, p-balioa zera da: lortutako datuak edo horiek baino muturrekoagoak/bakanagoak zoriz lortzeko probabilitatea. Beraz, lortutako p-balioa "txikia" izateak, H_0 egia izanik, halako datuak lortzeko probabilitatea txikia dela adierazten digu, eta horrek H_0 errefusatzera edo baztertzera eramaten gaitu. Aldiz, p-balio "handia" lortzen badugu, hipotesi nuluaren kontrako ebidentziarik ez dugu, eta, beraz, H_0 ezin errefusa daitekeela esango dugu.

Baina nola erabakiko dugu p-balioa "handia" ala "txikia" den? Lehen esan dugun moduan, esangura-maila (α edo H_0 egia izanik baztertzeko probabilitatea), aldez aurretik finkatu behar izan dugu, eta, beraz, hori erabiliko dugu muga-balio gisa. Praktikan erabiltzen diren α-ren balio ohikoenak 0.05 eta 0.01 dira.

Adibidean, $\alpha = 0.05$ finkatzen badugu, ikus dezakegu lehenengo kasuan ezin dugula diferentziarik dagoenik esan (p-balioa 0.676 da); bigarrenean, aldiz, diferentziak daudela esan dezakegu (p-balioa 0.00073 da).

4.8 adibidea. *Wilcoxonen testa erabiliz gure esperimentazio txikian lortutako emaitzak azter ditzakegu. Adibidez, bi algoritmo aukeratuz ikus dezakegu ea, lehenengo instantzian lortutako emaitzen arabera, LSren eta GAren emaitzak berdinak diren.*

Aurreko testa ondoko kodearen bitartez egin dezakegu:

[3] Binakako datuak izango bagenitu, `wilcox.test` funtzioari `paired=TRUE` aukera gehitu beharko genioke.

```
> id.LS <- results.df$Instance=="gr24" &
+           results.df$Algorithm=="LS"
> id.GA <- results.df$Instance=="gr24" &
+             results.df$Algorithm=="GA"
> sample.LS <- results.df$Fitness[id.LS]
> sample.GA <- results.df$Fitness[id.GA]
> wilcox.test(sample.LS, sample.GA)

##
##  Wilcoxon rank sum test
##
## data:  sample.LS and sample.GA
## W = 99, p-value = 2.165e-05
## alternative hypothesis: true location shift is not equal to 0
```

Goiko kodean lortutako p-balioa txikia dela ikusten dugu. Beraz, $\alpha = 0.05$ hartuaz, bi algoritmoen portaera ezberdina dela esan dezakegu.

4.9 adibidea. *Ondorio hori 1. instantziari soilik dagokio, baina gauza bera egin dezakegu beste 8 instantzietarako. Are gehiago, GA algoritmoa erreferentzia gisa hartzen badugu, ACO algoritmoekin lortutako emaitzak ere aldera ditzakegu. Hortaz, 16 konparazio egin ditzakegu eta, bakoitzeko, p-balio bat izango dugu.*

Aurreko adibideko testak egiteko garaian gogoratu, α-k I motako akats bat egiteko probabilitatea adierazten digula, baina probabilitate hori test bakar bati dagokio; 16 test independente egiten baditugu, zoriz, I motako akats bat egitea espero dezakegu (0.05*16=0.8 da eta).

Arazo hori saihesteko, alegia, I motako akats bat egiteko probabilitatea kontrolpean mantentzeko, *post hoc* izeneko zuzenketa batzuk aplikatu behar dira, esangura-mailan edo p-balioen kalkuluan. Metodorik sinpleena Bonferroni da, non p-balioak egindako test-kopuruaz biderkatzen diren. Metodo horren arazoa potentzia da; hots, oso zaila da H_0 errefusatzea (H_1 egia izanik ere) eta, beraz, diferentziei antzematea. Hori dela eta, badira beste metodo sofistikatuago batzuk: Finner, Shaffer, Holm, etab.

Algoritmoen emaitzen analisi estatistikoa gauzatzeko **scmamp** paketea erabil dezakegu [9]. Horretarako, lehenik eta behin emaitzak formatu egokian jarri behar ditugu, hau da, zutabe bakoitzeko algoritmo bat. Hori aurretik egin dugu jada beste helburu batzuekin, eta `data.test` objektuan dugu gordeta emaitza. Beraz berrerabili egingo dugu:

```
> head(data.test)

##   Instance   LS   GA  ACO
## 1     gr24 3006 2229 2710
## 2     gr24 2543 2264 2346
## 3     gr24 2594 2291 2658
## 4     gr24 2696 2335 2524
## 5     gr24 2704 2222 2430
```

```
## 6     gr24 2842 2557 2663
```

Orain Wilcoxonen testa aplikatuko dugu, instantziaz instantzia, LS eta ACO erreferentzia gisa hartu dugun GA-ren emaitzekin alderatzeko; p-balioak zuzentzeko Finner metodoa erabiliko dugu.

```
> a <- list()
> a$data <- data.test
> a$algorithms <- 2:4
> a$group.by <- 1
> a$test <- wilcox.test
> a$paired <- FALSE
> a$control <- "GA"
> a$correct <- "finner"
> test.results <- do.call(postHocTest, a)
> test.results$raw.pval[, -3]
```

```
##   Instance           LS          ACO
## 1     gr24 2.165018e-05 3.247526e-04
## 2   bays29 2.089242e-03 7.252809e-04
## 3    att48 1.149624e-02 7.252809e-04
## 4    eil51 3.526814e-01 1.504687e-03
## 5 berlin52 2.089242e-03 7.577562e-05
## 6 brazil58 2.089242e-03 4.330035e-05
## 7     st80 4.812509e-01 4.330035e-05
## 8    eil76 7.393644e-01 1.082509e-05
## 9    rat99 3.886207e-03 1.082509e-05
```

```
> test.results$corrected.pval[, -3]
```

```
##   Instance           LS          ACO
## 1     gr24 0.0001948337 0.0008348652
## 2   bays29 0.0034164865 0.0016311424
## 3    att48 0.0137795839 0.0016311424
## 4    eil51 0.3869331036 0.0027068068
## 5 berlin52 0.0034164865 0.0002273096
## 6 brazil58 0.0034164865 0.0001948368
## 7     st80 0.5008970922 0.0001948368
## 8    eil76 0.7393643508 0.0001948337
## 9    rat99 0.0049937749 0.0001948337
```

Goiko kodearen emaitzean lortutako p-balioak eta zuzendutako p-balioak ikus daitezke. Azken horiei erreparatuz, ikus dezakegu GA algoritmoa ACO algoritmoarekin alderatzean p-balio guztiak 0.05 baino txikiagoak direla. Horrek ziurtatzen digu bien artean ezberdintasun adierazgarriak daudela, eta, aurreko emaitzak kontuan hartuta, GAren emaitzak hobeak direla ondorioztatu dezakegu. Aldiz, LS algoritmoa eta GA alderatzean, lortutako p-balioak aurreko konparaziokoak baino handiagoak dira eta 3 kasutan (`att48`, `st80`, `eil76`) 0.05 baino altuagoak. Beraz, bi algoritmo horien artean ezberdintasunak badaude, baina ez hain nabariak.

Bi lagin baino gehiago konparatzen

Atal honen sarreran esan bezala, badira hainbat optimizazio-algoritmoren emaitzak aldi berean konparatzeko gai diren test estatistikoak. Test horiek bi fasetan garatzen dira: lehenengo omnibus-test bat aplikatzen da. Horrek, algoritmoren batek besteekiko portaera desberdina duen edo ez testatuko du (H_0=algoritmo guztien portaera berdina da). H_0 errefusatzeko nahiko ebidentzia badago, orduan binakako konparazio guztiak egiten dira diferentziak zehazki zein algoritmoren artean dauden ikusteko, *post hoc* deritzen testen bidez.

Gure kasuan, algoritmoen alderaketa egiteko garaian, normalean Friedmanen test ez-parametrikoa erabiltzen da. Test horrek binakako datuen tratamentua bi lagin baino gehiagotara hedatzen du, eta parekatutako laginak (3 edo gehiago) alderatzeko balio digu.[4]

4.10 adibidea. *Goiko metodologia gure emaitzei aplikatuko diegu, baina kasu honetan instantzia ezberdinetan lortutako emaitzak batera aztertuko ditugu. Horretarako, errepikapen guztien medianak erabiliko ditugu.*[a] *Hasteko, omnibus-test bat egingo dugu, Friedmanen testa hain zuzen. Horrek algoritmo guztien artean baten bat ezberdina den ikusteko balioko digu. Ondoren,* post hoc *test bat egingo dugu (Shaffer-en testa) ezberdintasunak zehazki zeinen artean dauden ikusteko.*

[a] Batez bestekoa ere erabil daiteke, baina estatistiko horrek emaitzak unimodalak direnean du bakarrik zentzua. Askotan, optimizazio-problemetan, emaitzek bi moda edo gehiago dituzte, eta, hortaz, beste estatistiko batzuk (mediana, batik bat) erabili ohi dira.

Konparazioa egiteko, `data.summarized` taula erabiliko dugu, non algoritmo eta instantzia bakoitzeko lortutako *fitness*-balioen mediana daukagun gordeta:

```
> head(data.summarized)
```

```
##     groups        LS        GA       ACO
## 1     gr24  2700.500  2277.500  2653.000
## 2   bays29  3586.000  3239.000  3536.000
## 3    att48 37535.500 35552.000  38598.000
## 4    eil51  1302.304  1290.575  1353.275
## 5 berlin52 23897.218 22696.687  24639.090
## 6 brazil58 98160.000 90810.500 101175.000
```

Orain, hiru algoritmoetatik baten bat ezberdina den jakiteko Friedman testa erabiliko dugu.

[4] Lagin askeak izango bagenitu Kruskal Wallis izeneko testa erabil genezake.

```
> multipleComparisonTest(data=data.summarized,
+                        algorithms=2:4,
+                        test="friedman")

##
##  Friedman's rank sum test
##
## data:  data.multipleComparisonTest
## Friedman's chi-squared = 9.5556, df = 2,
## p-value = 0.008415
```

Testaren arabera, algoritmoren baten emaitzak estatistikoki desberdinak dira (p-balioa 0.003). Beraz, *post hoc* test bat erabiliko dugu algoritmoen binakako konparazio guztiak egiteko, Shaffer-en testa hain zuzen ere.

```
> postHocTest(data=data.summarized,
+             algorithms=2:4,
+             test="friedman",
+             correct="shaffer")

## $summary
##            LS       GA      ACO
## [1,] 19926.69 18616.28 20481.79
##
## $raw.pval
##             LS          GA         ACO
## LS          NA 0.059346439 0.238592829
## GA  0.05934644          NA 0.002183045
## ACO 0.23859283 0.002183045          NA
##
## $corrected.pval
##             LS          GA         ACO
## LS          NA 0.059346439 0.238592829
## GA  0.05934644          NA 0.006549134
## ACO 0.23859283 0.006549134          NA
```

Zuzendutako p-balioak aztertzen baditugu, esan dezakegu, espero genuen bezala, *GA*ren eta beste bi algoritmoen arteko diferentziak estatistikoki esanguratsuak direla. Aurreko emaitzak ikusirik, ondorioztatu dezakegu *GA*ren emaitzak beste bienak baino oro har hobeak direla. Aldiz, beste bi algoritmoen artean ez dago ezberdintasun adierazgarririk.

Bibliografia

1. Aarts, E.H.L., Laarhoven, P.J.M. (eds.): Simulated Annealing: Theory and Applications. Kluwer Academic Publishers, Norwell, MA, USA (1987)
2. Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. Journal of Molecular Biology **215**(3), 403–410 (1990)
3. Ansótegui, C., Sellmann, M., Tierney, K.: A gender-based genetic algorithm for the automatic configuration of algorithms. In: I.P. Gent (ed.) Principles and Practice of Constraint Programming - CP 2009, *Lecture Notes in Computer Science*, vol. 5732, pp. 142–157. Springer Berlin Heidelberg (2009)
4. Bartz-Beielstein, T., Lasarczyk, C., Preuss, M.: The sequential parameter optimization toolbox. In: T. Bartz-Beielstein, M. Chiarandini, L. Paquete, M. Preuss (eds.) Experimental Methods for the Analysis of Optimization Algorithms, pp. 337–360. Springer-Verlag, Berlin, Germany (2010)
5. Beni, G.: The concept of cellular robotic system. In: Proceedings of the IEEE International Symposium on Intelligent Control, pp. 57–62 (1988)
6. Blum, C., Merkle, D.: Swarm Intelligence: Introduction and Applications. Springer-Verlag (2008)
7. Bonnini, S., Corain, L., Marozzi, M., Salmaso, L.: Nonparametric Hypothesis Testing: Rank and Permutation Methods with Applications in R. Wiley (2014)
8. Burkard, R.E., Çela, E., Pardalos, P.M., Pitsoulis, L.S.: The quadratic assignment problem (1998)
9. Calvo, B., Santafe, G.: scmamp: Statistical comparison of multiple algorithms in multiple problems. The R Journal **8**(1), 248–256 (2016)
10. Congram, R.K.: Polynomially searchable exponential neighborhoods for sequencing problems in combinatorial optimization
11. Dorigo, M.: Optimization, learning and natural algorithms. Ph.D. thesis, Politecnico di Milano (1992)
12. Dorigo, M., Maniezzo, V., Colorni, A.: Ant system: Optimization by a colony of cooperating agents. Trans. Sys. Man Cyber. Part B **26**(1), 29–41 (1996)
13. Dueck, G., Scheuer, T.: Threshold accepting-a general-purpose optimization algorithm appearing superior to simulated annealing. Journal of Computationl Physics **90**, 161–175 (1990)
14. Feo, T.A., Resende, M.G.: A probabilistic heuristic for a computationally difficult set covering problem. Operations Research Letters **8**(2), 67 – 71 (1989)
15. Glover, F.: Future paths for integer programming and links to artificial intelligence. Computers & Operations Research **13**(5), 533–549 (1986)
16. Goldberg, D.E., Lingle, R.: Alleles Loci and the Traveling Salesman Problem. In: ICGA, pp. 154–159 (1985)
17. Gupta, J.N., Stafford, E.F.: Flow shop scheduling research after five decades. European Journal of Operational Research (169), 699–711 (2006)
18. Gwiazda, T.: Genetic algorithms reference Volume I Crossover for single-objective numerical optimization problems. v. 1. Lightning Source (2006)

19. Holland, J.H.: Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, MI, USA (1975)
20. Huang, M., F.Romeo, Sangiovanni-Vincentelli, A.: An efficient general cooling schedule for simulated annealing. In: Proceedings of the IEEE International Conference on Computer-Aided Design, pp. 381–384 (1986)
21. Kellerer, H., Pferschy, U., Pisinger, D.: Knapsack problems. Springer (2004)
22. Kennedy, J., Eberhart, R.C.: Particle swarm optimization. In: Proceedings of the IEEE International Conference on Neural Networks, pp. 1942–1948 (1995)
23. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science **220**, 671–680 (1983)
24. Larrañaga, P., Lozano, J.A.: Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation. Kluwer Academic Publishers (2002)
25. Lopez-Ibañez, M., Dubois-Lacoste, J., Stützle, T., Birattari, M.: The irace package: Iteratec racing for automatic algorithm configuration. Tech. Rep. TR/IRIDIA/2011-004, IRIDIA (2011)
26. Lozano, J.A., Larrañaga, P., Inza, I., Bengoetxea, E.: Towards a New Evolutionary Computation: Advances on Estimation of Distribution Algorithms (Studies in Fuzziness and Soft Computing). Springer-Verlag New York, Inc. (2006)
27. Moscato, P., Fontanari, J.: Convergence and finite-time behavior of simulated annealing. Advances in Applied Probability **18**, 747–771 (1990)
28. nez, M.L.I., Dubois-Lacoste, J., Pérez Cáceres, L., St utzle, T., Birattari, M.: The irace package: Iterated racing for automatic algorithm configuration. Operations Research Perspectives **3**, 43–58 (2016). DOI 10.1016/j.orp.2016.09.002
29. Pepper, J.W., Golden, B., Wasil, E.: Solving the traveling salesman problem with demon algorithms and variants. Tech. rep., Smith School of Business, University of Maryland, College Park, Maryland (2000)
30. Pesch, E., Glover, F.: TSP ejection chains. Discrete Applied Mathematics **76**(1âĂŞ3), 165 – 181 (1997)
31. Reinelt, G.: TSPLIB - A t.s.p. library. Tech. Rep. 250, Universität Augsburg, Institut für Mathematik, Augsburg (1990)
32. Sheskin, D.J.: Handbook of Parametric and Nonparametric Statistical Procedures. CRC Press (2003)
33. Shmoys, D.B., Tardos, É.: An approximation algorithm for the generalized assignment problem. Mathematical Programming **62**(1-3), 461–474 (1993)
34. Taillard, E.: Benchmarks for basic scheduling problems. European Journal of Operational Research **64**(2), 278–285 (1993)
35. Takagi, H.: Interactive evolutionary computation: fusion of the capabilities of ec optimization and human evaluation. Proceedings of the IEEE **89**(9), 1275–1296 (2001)
36. Talbi, E.G.: Metaheuristics: From Design to Implementation. Wiley Publishing (2009)
37. Černý, V.: Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. Journal of Optimization Theory and Applications **45**(1), 41–51 (1985)
38. Wickham, H.: ggplot2: elegant graphics for data analysis. Springer New York (2009)
39. Wolpert, D.H., Macready, W.G.: No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation **1**(1), 67–82 (1997)

UNIBERTSITATEKO ESKULIBURUAK
MANUALES UNIVERSITARIOS

INFORMAZIOA ETA ESKARIAK • INFORMACIÓN Y PEDIDOS

UPV/EHUko Argitalpen Zerbitzua • Servicio Editorial de la UPV/EHU
argitaletxea@ehu.eus • editorial@ehu.eus
1397 Posta Kutxatila - 48080 Bilbo • Apartado 1397 - 48080 Bilbao
Tfn.: 94 601 2227 • www.ehu.eus/argitalpenak

Universidad del País Vasco
Euskal Herriko Unibertsitatea