BHLN: Oinarrizko Kontzeptuak

Borja Calvo, Josu Ceberio, Usue Mori

Laburpena

Lehenengo kapitulu honetan optimizazioaren ikuspen globala aurkeztuko dugu, oinarrizko kontzeptuak azalduz. Kapitulua bi zatitan dago banatuta; lehenengoan optimizazio problemak izango dira aztergai eta, bereziki, hauen konplexutasuna, hau baita metodo heuristikoak erabiltzeko motibaziorik garrantzitsuena. Bigarren zatian, metodo heuristikoak aurkeztuko ditugu, hauek diseinatzeko eta erabiltzeko kontuan izan beharreko kontzeptuei erreparatuz.

1 Sarrera

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

1.1 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 honek jasotzen zituen baldintza ekonomikoen ondorioz hogeigarren hamarkadan Alemaniako egoera nahiko larria 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 Gerran amaituko zuen gertakizun sekuentzia hasi zen.

1934. urtean Hitler-ek Alemaniako berrarmatze prozesua agindu zuen, eta 1935eko udaberrirako bere aireko armada –Luftwaffea– Britainiakoaren parekoa zela aldarrikatzen hasi zen. Nazien hegazkin bonbaketarien mehatxua arazo larri bihurtu zen Britaniko Gobernuarentzat eta, hortaz, aire defentsa antolatzeari ekin zion. Aireko estatu-idazkariak «Imperial College of Science and Technology»-ko errektoreari aireko defentsaren arazoa aztertuko zuen batzordea sortzeko eskatu zion; batzorde honen lanaren ondorioz, 1935eko udan, Robert Watson-Watt-ek radarra asmatu zuen. Tresna oso erabilgarria izan arren, zuen ahalmen guztia ateratzeko, defentsa sisteman –behatokiak, ehiza-hegazkinak, antiaereoko artilleria, ...– integratu beharra zegoen, eta arduradunak laster konturatu ziren ez zela lan erreza izango. Izan ere, defentsa operazioen antolakuntza, bere osotasunean, oso arazo konplexua zen. Hori dela eta, problema hau 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 Britaniko armadan eta, hortik, Estatu Batuetako armadara hedatu zen. 1945ean, gerra amaitu zenerako, mila pertsonatik gora zeuden Ikerkuntza Operatiboan lanean Britaniko armadan.

Gerra amaitu ostean, Europako herrialdeen egoera oso larria zen: herrialdeak suntsituta, baliabideak gerran xahututa, ... Hurrengo urteetan, herrialdeak berreraikitzeko erronkari aurre egiteko, gerra bitartean operazio militarrak antolatzeko garatutako metodologia matematikoak bereziki egokiak zirela konturatu ziren agintariak.

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

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

Adibide gisa, Dantzigek –gerran US armadan ziharduena– 1947an Ikerkuntza Operatiboko algoritmorik ezagunena, Simplex algoritmoa, proposatu zuen.

Arlo berri honek ikertzaileen arreta erakarri zuen, gerraosteko garaian hedapen nabarmena edukiz. Hasierako urteetan planteatzen ziren algoritmoek soluzio zehatzak lortzea zuten helburu baina teknika hauek problema mota konkretu batzuk ebazteko erabili zitezkeen bakarrik–problema linealak, adibidez–. Hirurogeita hamarreko hamarkadan zientzialariek problemen konplexutasuna aztertzeari ekin zioten. Bestalde, konputagailu pertsonalak agertu ziren merkatuan. Problema batzuetan *konplexutasun* - *tamaina*-ren konbinaketaren 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 heuristikoak edo metaheuristikak 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

2 Optimizazio problemak

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

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

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

Adibidea 2.1 *Demagun plastikozko piezak ekoizten dituen enpresa bateko logistika sailean lana egiten dugula. Lantegian zenbait makina, biltegi bat lehengaiak eta ekoiztutako piezak biltzeko, eta abar daude. Igandero, asteko eskaera aztertuz, plangintza egin behar dugu; zein piezak ekoiztu lehenago, zein 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, matematikoki nola hurbildu eta ebatzi daitekeen erabakitzeko ezinbestekoa baita.

Optimizazio problemak formalizatzean bi elementuri atzeman beharko diegu. Lehenik eta behin, problemaren soluzio guztien multzoari, *soluzio bideragarrien espazioa* edo *bilaketa espazioa* 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 suposatuko dugu[1].

Definizioa 2.1 Minimizatze-problema. *Izan bedi S soluzio bideragerrien 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, ondorengo hiru ezaugarriak 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-problematan 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 hauen 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 hauetan metodo heuristikoetara jotzea beharrezkoa izango da.

2.1 Problemen konplexutasuna

Problema baten konplexutasuna, hau ebazteko existitzen den algoritmo eraginkorrenaren konplexutasuna da. Algoritmoak problemak pausuz-pausu ebazteko erabiltzen diren prozedurak dira. Problema mota bakoitzetik *instantzia* ezberdinak izan ditzakegu, eta instantzia hauek *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.

Adibidea 2.2 *Demagun hiri 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 pausu kopurua neurtzen du[3].

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

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

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

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

Taula 1: 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

Adibidea 2.3 *Suposatu 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 –kontutan 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 pausu kopurua $10n^2$ edo $0.5n^2$ izateak, kontua eragiketa kopuruaren progresioa tamainarekiko koadratikoa dela baizik; ideia hau O notazioaren bidez adierazi ohi da.

Definizioa 2.2 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. Hau kontutan hartuz, beraz, distantzien matrizea kalkulatzeko algoritmoaren konplexutasuna $O(n^2)$ dela esango dugu.

Konplexutasun maila ezberdinak defini daitezke; 1. taulak maila ohikoenak biltzen ditu. Oinarrizko operazioa milisegundoa hartuz, taulan n tamaina ezberdinetarako exekuzio denborak daude kalkulatuta.

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 ikusi daitekeen bezala.

Adibidea 2.4 *Demagun problema bat ebazteko hiru algoritmo ditugula: P, polinomikoa, $O(n^{10})$; E, esponentziala, $O(5^n)$ eta H, hiper-esponentziala, $O(n^n)$. 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 honetan E-ren ordez H, algoritmo hiper-esponentziala, da alegia, kosturik txikienekoa. Hau da, kostuaren araberako ordena konplexutasunarekiko alderantzizkoa da, $H < E < P$.

1 irudiak konplexutasun ordena batzuen funtzioak erakusten ditu. Y ardatzak denbora segundotan adierazten du eta eskala logaritmikoan dago. Grafikoan ikus ditzakegun bezala, funtzio linealak, koadratikoak eta kubikoak ordu mugatik behera mantentzen dira eta, beraien 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. 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 determinista rik problema denbora polinomikoan ebazten duena[4].

NP klasean, NP-osoa deritzon azpi-klase 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 hau erabaki-problemei zuzendua egon arren, optimizazio problementzat ere erabiltze 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 problematan, aldiz, NP-zaila terminoa (*NP-hard*, ingelesez) erabiltzen da.

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

2.2 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 beretsua da; hala nola, garraio-problemak, esleipen-problemak, antolakuntza-problemak, etab. Hori dela eta, optimizazioan , askotan, problema teorikoak aztertzea izaten da ohikoena, errealitatean topa ditzakegun problemen abstrakzioak edota sinplifikazioak direlarik. Atal honetan optimizazio problema teoriko klasiko batzuk aztertuko ditugu.

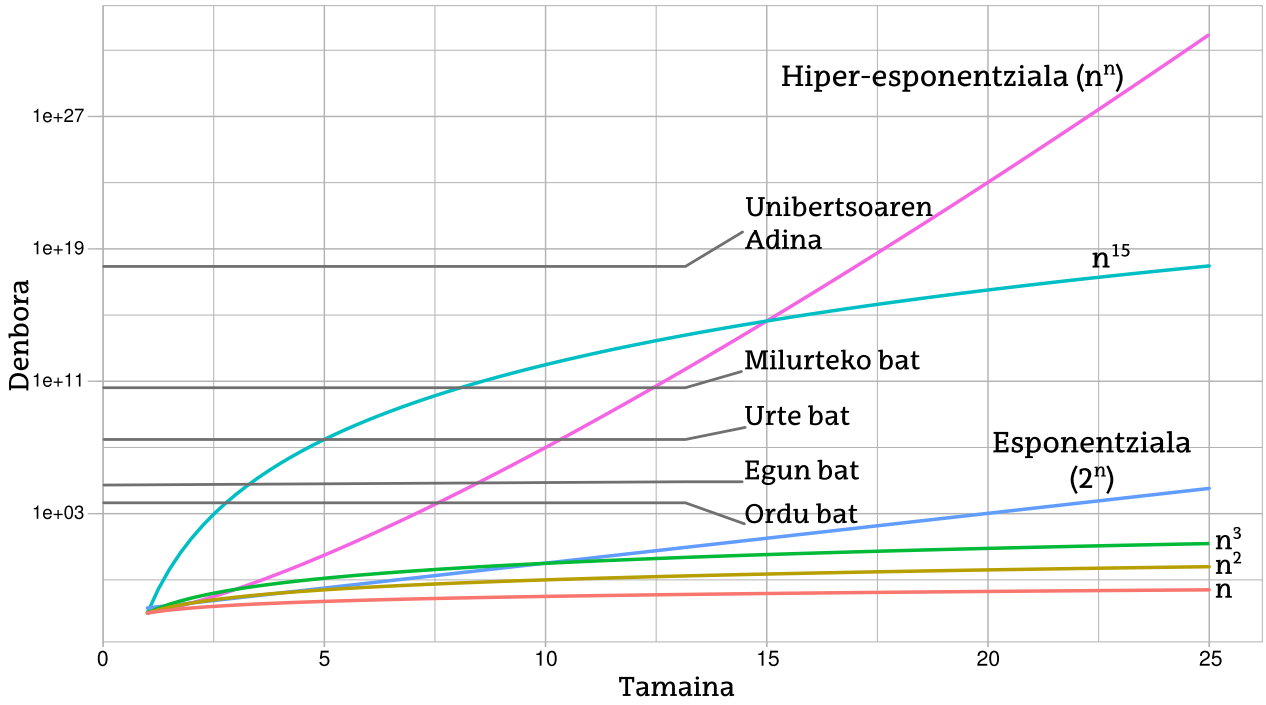
2.2.1 Garraio-problemak

Merkantziak edota pertsonen garraioarekin zerikusia duten optimizazio problemak 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-puntuek 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*-a [6] da –hau da, Saltzaile Bidaiariaren Problema–.

Adibidea 2.5 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 herrien arteko distantzia ezaguna izanda, gure helburua herri guztietatik, behin eta bakarrik behin, pasatzen den ibilbiderik motzena topatzea da.*

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



Irudia 1: Konplexutasun ordena tipikoen hazkunde abiadura n-rekiko.

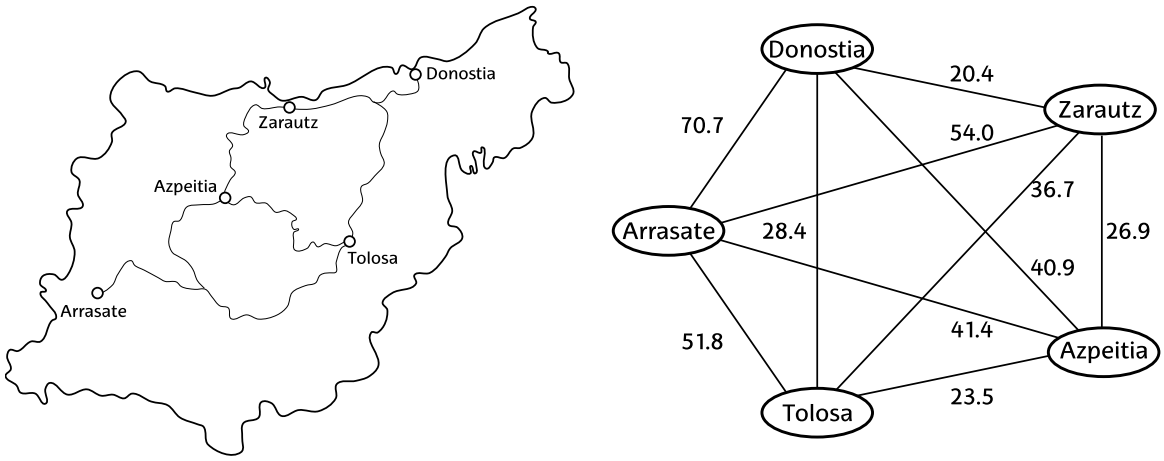
Problema honen hastapenak Irlandan daude, Hamilton matematikariaren lanetan. Problema formalizatzeko grafo oso bat eraiki dezakegu non erpinak herriak diren eta edozein bi erpinen artean pisu konkretu bateko ertz bat definitzen delarik; ertzen pisua, lotzen dituen bi herrien arteko distantzia da (ikusi 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.

Arrasate, Zarautz, Tolosa, Azpeitia, Donostia ibilbidea, irudian agertzen den problemarentzako soluzio posible bat da; eta bere 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)
```



(a) Mapa (b) Dagokion grafoa

Irudia 2: TSP-aren adibide bat, bost herriekin

```
> 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 ebaluatu egingo 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 ezazu, adibidez, Azpeitia eta Tolosa trukatzen. Soluzio berri hau hobea ala okerragoa da?.
- Zenbat ibilbide daude bost herri hauek behin eta bakarrik behin bisitatzen dituztenak?

Ariketa 2.1 *Inplementa ezazu funtzio bat n tamainako TSP problema bat eta bere ebaluazio funtzioa emanda, soluzio guztiak ebaluatuz bide motzena topatzen duena.*

Ikus dezagun nola formaliza daitekeen TSP problema matematikoki:

Definizioa 2.3 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.*

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

TSP-a oso erabilia da algoritmoak probatzean, eta TSPLIB liburutegian [11] hainbat instantzia eskuragarri daude, orain arte lortutako soluziorik onenen informazioarekin batera.

2.2.2 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, zeinek 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, [2] ingelesez–.

Definizioa 2.4 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*

2.2.3 Antolakuntza-problemak

Prozesu, lan edota ataza desberdinen 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.

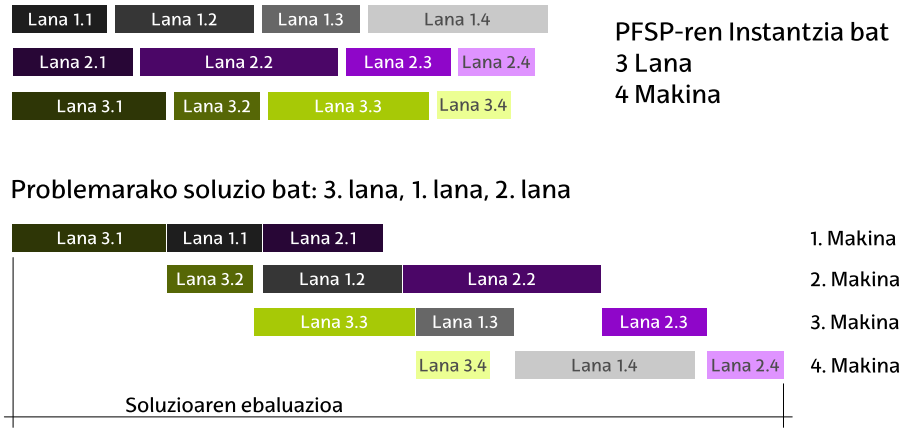
Hasiera batean, antolakuntza-problema gehientsuenak industriarekin zerikusia zuten domeinuetan proposatu ziren. Produktuaren ekoizpenerako makineria erabiltzen zen enpresetan, etekina maximizatzea zen helburua, makineriaren eta baliabideen erabilera, mantentze-kostuak, eta abar optimizatuz. Aldi berean,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

Gaur egun ordea, problema hauek 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, denbora eta memoriaren erabilera minimizatuz.

Adibide gisa, jarraian, antolakuntza problemetan oso ezaguna den Muntaketa-kateko Plangintza-Problema –*Permutation Flowshop Scheduling Problem*, PFSP [7] ingelesez– aztertuko dugu:

Definizioa 2.5 PFSP problema. *Izan bitez, n lan, m makina eta $P \in \mathbb{R}^{n\times m}$ prozesamendu denboren matrizea, non p_{ij} i lanak j makinan prozesatzeko behar duen denbora adierazten duen. Lan bakoitza burutzeko, m prozesu desberdin 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 pasako da hurrengo prozesua burutzera, baldin eta makina hau libre badago. PFSPa lanak prozesatzeko denbora totala minimizatzen duen n lanen sekuentzia optimoa aurkitzean datza.*

3 irudian problemaren instantzia bat ikus daiteke. Adibide honetan 3 lan burutu behar dira, 4 makina ezberdinetan. Irudiaren goiko partean lan bakoitzak makina bakoitzean igaro behar duen denbora adierazten da, lauki zuzenen bidez. Irudiaren beheko partean soluzio bat proposatzen da; soluzio honetan, lanak 3,1,2 ordenean prozesatuko dira. Honek 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



Irudia 3: PFSP problemaren adibide bat. Goiko partean problemaren definizioa dago, hau da, lan bakoitza burutzeko makina bakoitzean behar den denbora. Beheko partean soluzio bat eta bere interpretazioa erakusten da. Helburua denbora totala minimizatzea bada, 3.1 lana hasten denetik 2.4 lana amaitzen den arte igarotzen duen denbora da soluzioaren ebaluazioa

daiteke. Problemaren helburua denbora totala minimizatzea bada, soluzio honen 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 honetan, antzeko problemen aukera oso zabala da [13].

2.2.4 Azpimultzo-problemak

Demagun objektu multzo bat dugula, eta multzo honetatik, objektu batzuk aukeratu behar ditugula. Aukeraketa, ez da ausaz egingo, baizik eta irizpide eta murriztapen batzuei jarraituz. Azpimultzo-problematan, helburua aukeraketak ematen dizkigun onurak maximizatzean datza, definitutako murriztapenak betetzen direlarik.

Azpimultzo-problemen hedapena oso zabala da, logistikako alorretan, gehien bat: 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 [9] da. Jarraian zehazki azalduko dugu problema hau:

Definizioa 2.6 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.*

2.2.5 Grafoei buruzko problemak

Grafoak, matematika eta informatikan funtsezko egiturak dira. Orokorrean grafoak objektuen arteko erlazioak adierazteko erabiltzen dira eta, beraz, beraien erabilera edozein domeinutan da aplikagarria. Hori dela eta, grafoei loturiko problemen multzoa oso zabala da. Esaterako, berriki ikusi ditugun TSP eta QAP-ak 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.

Definizioa 2.7 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 dagoen 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)
```

Ausazko soluzio bat sortzen dugu, bakarrik 3 kolore erabiliz. Kontutan hartu behar da soluzioa `factor` motako bektore bat izan behar dela, non balio posibleen kopurua nodo kopuruaren berdina izan behar den (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.

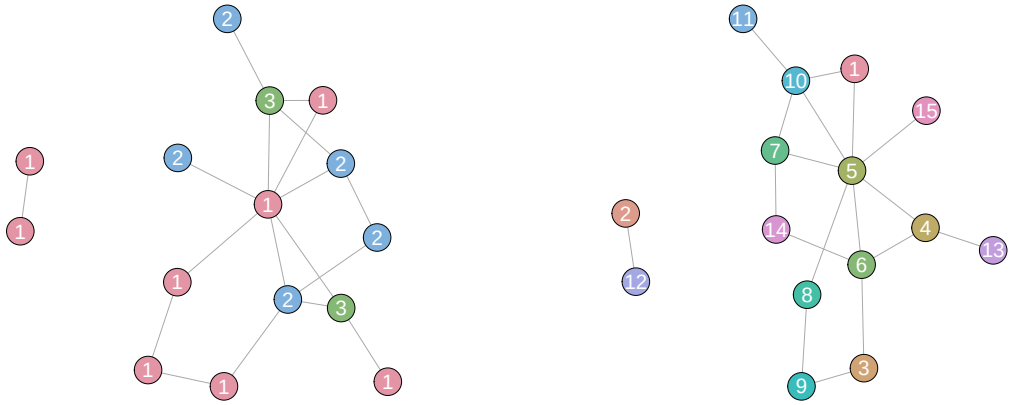
```
> l <- paste("c", 1:n, sep="")
> rnd.sol <- factor(sample(l[1:3], size=n, replace=TRUE), levels=l)
> rnd.sol

##  [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=15, label.cex=1.5)

## Loading required package:  colorspace
```

Soluzioa bideraezina da, 4 (a) irudian ikus daitekeen legez. Soluzio bideragarri bat sortzeko era sinple bat badago: nodo bakoitzari kolore ezberdin bat esleitu (ikusi 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=15, label.cex=1.5)
```

2.2.6 Karaktere-kate problemak

Optimizazio konbinatorioko karaktere-kateen arteko erlazioak aurkitzeaz arduratzen diren problemen multzoa da. Problema ezagunenetariko bat Azpisekuentzia Komun Luzeenaren Problema –Longest Common Subsequce



(a) Soluzio bideraezina

(b) Soluzio bideragarria

Irudia 4: Grafoen koloreztatze-problemaren bi adibide. Lehenengoa, (a), bideraezina da, elkar ondoan dauden nodo batzuk kolore berdina baitute. Bigarrena, (b), soluzio bideragarri tribiala da, nodo bakoitzak kolore ezberdin bat baitu.

Problem, LCSP– da. Izenak berak adierazten duen bezala, ditugun karaktere-kateen edo sekuentzien artean komuna den azpisekuentzia luzeena bilatzen duen problema da[5].

Adibidea 2.6 *Demagun hiru DNA sekuentzia ditugula; CACGACGCGT, CGTTTCGCAG eta CTTGCGCGA. Hiru sekuentzietan dagoen azpisekuentzia komun luzeena CGCGCG da; hau da, caC-GaCGCGt, CGtttCGCaG, eta CttGCGCGa daukagu. Beste edozein letra sartuz gero, azpisekuentzia ez da sarrerako hiru sekuentzietan agertuko.*

LCSP-a formalizatzeko lehendabizi azpisekuentzia kontzeptua definitu behar dugu.

Definizioa 2.8 *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– LCSP-a programazio dinamikoa erabiliz[6] $O(mn)$ konplexutasunarekin ebatz daiteke; sekuentzia kopurua finkaturik gabe badago, ordea, problema NP-zaila da.

Definizioa 2.9 Longest Common Subsequence Problema (LCSP) *Izan bitez Σ alfabetoan definitutako tamaina ezberdineko k sekuentzia $S_1, \ldots, S_k$. Izan bedi k sekuentzien azpisekuentzia diren sekuentzia multzoa $\mathcal{C}$. LCSP-aren helburua $C^* \in \mathcal{C}$ sekuentzia topatzea da, non $\forall C_i \in \mathcal{C}$ $|C_i| > |C^*|$.*

[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 honetan, nahiz eta algoritmo polinomikoa izan, sekuentziak milioika elementu izan dezaketenez, programazio dinamikoaz gain metodo heuristikoak erabiltzen dira; metodorik ezagunena BLAST[1] izenekoa da

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
```

Algoritmoa 3.1: TSPrako soluzio onak eraikitzeko metodo heuristikoa

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

Azken hamarkadetan ordea, Bioinformatika arloak lortu duen arretarekin, karaktere-kate 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 duen bestelako teknikak erabiltzen dira.

Testuinguru honetan, Sekuentzia-zati Mutaketa Problemaren helburua, azpisekuentzietatik abiatuta, sekuentzia bakar bat osatzea da. LCSP-an antzera, azpisekuentzia komunak modu eraginkorrean detektatzea ezinbestekoa da, prozesuaren bukaeran DNA sekuentzia fidagarri bat lortzeko.

3 Optimizazio problemak ebazten

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

Hurbilketa hau –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. Hau kontutan hartuz, denbora maximoa finkatzen badugu, beti problema tamaina maximo bat izango dugu; problema handiagoa bada, finkatutako denboran ebazterik ez da egongo.

Demagun 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 hiper-esponentzialarekin $n > 7$ tamainako problemak, ordu batean, ebaztezinak direla; algoritmo esponentzialarekin, ostera, hogei tamainako problemak ebazteko gai izango ginateke. Algoritmoak polinomikoak izateak ez du esan nahi algoritmoak edozein tamainako problema ebatz ditzatekeenik. Hau 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.

Kasu hauetan 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 hau burutzeko algoritmo *heuristikoak*, «intuizioan» oinarritutako metodoak, erabiltzea da ohikoena. Algoritmo hauekin 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, 2.2.1 atalean deskribaturiko TSP problema ebazteko.

Demagun badakigula abiapuntuko herria zein den, hau da, zein den ibilbideko lehendabiziko herria; soluzioa pausuz-pausu eraikiko dugu, urrats bakoitzean aurreko urratsean aukeratu dugun herritik gertuen dagoen herria aukeratuz. Metodoaren sasikodea 3.1 algoritmoan ikus daiteke.

Ebatz dezagun, algoritmo hau erabiliz, 2 irudian dagoen TSParen instantzia, Arrasatetik abiatuz. Arrasatetik gertuen dagoen herria Azpeitia den legez, hauxe izango da gure ibilbideko bigarren herria. Ondoren, Azpeititik Tolosara joango gara, hau baita gertuen dagoena eta bertatik Donostiara. Bisitatu barik dauden herrietatik Zarautz da Donostiatik gertuen dagoena –eta, berez, bakarra–. Ibilbidea ixteko Zarautzetik Arrasatera itzuli beharko gara. Laburbilduz, heuristiko sinple hau erabiliz ondoko soluzioa daukagu: Arrasate, Azpeitia, Tolosa, Donostia, Zarautz, Arrasate; soluzio honen helburu funtzioa 41,4 + 23,5 + 28,4 + 20,4 + 54,0 = 167,7 da. Ez dugu inongo bermerik soluzio hau optimoa denik, baina soluzio ona da eta, batez ere, denbora koadratikoan lortu dugu.

Algoritmo hau `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 aukerari onena aukeratzen, aukera batzuen artean bat ausaz aukeratzen du. Azken puntu honek gero ikusiko dugun algoritmo batekin (GRASP) zerikusia du.

Hemen funtzio honen bertsio sinplifikatu bat inplementatuko dugu, adibide gisa. Funtzio honek parametro bakar bat jasotzen du, `cmatrix`, kostu matrizea dena; hasteko, aukeraezinak diren balioak adierazteko `NA`-k (*not available*) txertatuko ditugu matrizean. Kontutan hartuz diagonalean dauden balio guztiak ezin direla aukeratu (ez du zentzurik hiri batetik hiri berberara joatea), 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 dezakegula (izan ere, matrizea simetrikoa bada beti izango ditugu, gutxienez, bi elementu). Lehenengo elementua bakarrik hautatuko dugu, eta honek markatuko ditu ibilbideko lehendabiziko bi hiriak. Algoritmoarekin jarraitzeko jakin behar dugu sartu dugun lehenengo hiritik (`best.pair[1,1]`) 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 hauei dagozkien zutabeak ere bideraezin gisa finkatu beharko ditugu.

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

Gure soluzioak, momentuz, bakarrik bi hiri ditu. Soluzio osoa eraikitzeko, urrats bakoitzean, azken pausuan 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 bere 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 honi aurre egiteko, *bilaketa heuristikoak* edo *metaheuristikoak* proposatu ziren. Metodo hauek 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. Hauek guztiak hurrengo kapituluen izango ditugu aztergai.

Optimizazio problema bati aurrez aurre gaudenean, kontuan izan beharreko hainbat gauza daude: alde batetik, problemaren formalizazio beran agertzen diren elementuak –helburu funtzioa eta soluzioen espazioa, alegia– eta, bestaldetik, problema ebazteko erabil daitezkeen algoritmoak. Hurrengo ataletan aspektu guzti hauek banan-banan komentatuko ditugu.

3.1 Helburu funtzioa

Aurreko atalean ikusi dugu optimizazio problema bat definitzeko bi elementu behar ditugula, horietako bat helburu funtzioa delarik. Helburu funtzioa soluzioen optimotasuna ebaluatzeko erabiliko dugu eta, hortaz, funtzio honek 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 honen distantzia edo kostu totala neurtuko du. Adibide honetan darabilgun funtzioa tribiala da eta zuzenean aplika daiteke; hau 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**[14]. 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 hauetan 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 hauek *exekutatu* egin behar dira, ea espero dena egiten duen egiaztatzeko.

3.2 Bilaketa espazioa: soluzioen kodeketa

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

Kodeketa on bat diseinatzeko zenbait aspektu 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 hauetako batzuk, 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. Elkar-esklusibotasun hori dela eta, *permutazioak* TSParen soluzioak kodetzeko adierazpide oso egokiak dira. Soluzio guztiak kodetu daitezke permutazioen bidez eta permutazio guztiek soluzio bideragarriak kodetzen dituzte[8]. Adierazpide berdina beste hainbat problematan erabil daiteke, hala nola, *scheduling* problematan, beste *routing* problematan, ordenazio problematan, ...

Azpi-multzo problematan (ikus. 2.2.4 atala), baldintza edo murrizketa batzuk betetzen dituen azpimultzo optimoa topatzea da helburua. Multzoekin diharduguneen *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 hau Esleipen Problema-Orokorrean –*Generalized Assignment Problem*, GAP [12], ingelesez– erabili ohi da.

> **Definizioa 3.1 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 dute, hurrenez hurren. i agentearen lan-karga gorenekoa l_i bada eta ataza bakoitza agente bakar batek egin dezakeela kontutan 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 zein agenteak egingo duen adieraziko du eta; kodeketa honen bidez soluzio-kode erlazioa bijektiboa da, hau da, soluzio bakoitzeko kode bakarra dago eta kode bakoitzak soluzio bakarra kodetzen du.

Bektoreetan oinarritutako kodeketarekin 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 honen bitartez egin daiteke, parametroak jarraituak badira betiere.

Orain arte ikusi ditugun adierazpideak *linealak* deritzonak dira, soluzioak bektore baten bidez kodetzen baitira. Adierazpide hauek 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 konbinaketa ere asko erabiltzen den beste estrategia bat da; lehen aipaturiko parametro optimizazioan, esate baterako, parametro jarraituak eta diskretuak ditugunean balio

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

[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
```

Algoritmoa 3.2: Motxilaren problemarako bideragarriak ez diren soluzioak konpontzeko prozedura bat

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 honek bilaketaren eraginkortasuna kaltetu dezake. Hau gutxi balitz, 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, zehar-kodetzea deritzogu.

3.3 Bilaketa espazioa: murrizketak

Askotan, bideragarritasunaren definizioak hainbat murrizketa dakarzki 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 honek murrizketak integratuko ditu, alegia, sor daitezkeen kode guztiek soluzio bideragarriak adieraziko dituzte. Soluzioen kodeketa ez ezik, soluzioak maneiatzeko darabilzkigun funtzio matematikoak ere bideragarritasuna mantentzeko erabili daitezke. Estrategia hau zenbait problematan –TSPan, besteak beste– murrizketak beteko direla ziurtatzeko bide zuzena da.
- **Soluzio bideraezinak baztertzea** - Estrategiarik sinpleena da; soluzio bat bideragarria izan ezean, bilaketa prozesuan baztertu egiten da. 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 bilaketa prozesuan erabil daitezke, 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 hau erabilgarria izan dadin, erabiltzen diren konponketa-algoritmoak eraginkorrak izan behar dira, 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 hau 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.

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

Orain, funtzio hau kontutan hartuz, soluzioak zuzentzeko funtzioa inplementa dezakegu. Funtzioak inplementatzen duen sasi-kodea 3.2 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) limitea 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; bere balioa handia da, baina baita bere pisua ere. Beraz, elementu hori motxilatik aterako dugun lehena izango da. Dena dela, aldaketa horrekin

[9] Oro har, bilaketa hasierako iterazioetan penalizazio koefiziente txikiak erabiliko ditugu, geroz eta handiagoa eginez –gogoratu bilaketa amaitzen denean soluzio bideragarria nahi dugula–. Bilaketaren progresioari buruzko informazioa erabil daiteke penalizazioa egokitzeko, «adaptative penalizing» deritzen estrategiak erabiliz.

bakarrik ez dugu soluzio bideragarri bat lortuko. Hau ikusirik, ratiorik handiena duen bigarren elementua kenduko dugu: 5. elementua. Orain, bi aldaketa hauek 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
```

3.4 Algoritmoak

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

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

Gauzak horrela, hainbat egoeratan problemaren optimo globala topatzerik ez dago. Kasu honetan, 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.

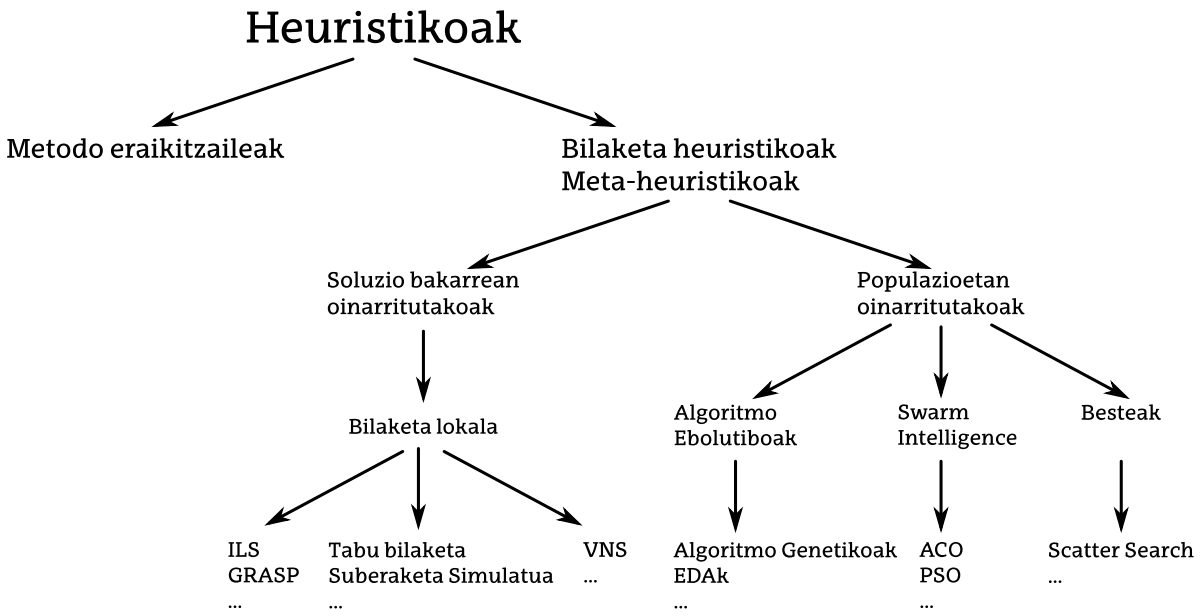
Soluzio hurbilduak lortzeko bi aukera ditugu: metodo hurbilduak, zeinek hurbilketa maila bermatzen duten, eta metodo heuristikoak, ezer bermatzen ez dutenak.

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

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

- **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 hauei, *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.

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



Irudia 5: Metodo heuristikoen eskema

Meta-heuristiko mota asko daude, bakoitza bere intuizioan oinarritutakoa. Metodoak sailkatzeko era asko egon arren, partiketa 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 ditu algoritmoak. Bilaketa lokala da metodo hauen arteko ezagunena; alabaina, metodo honek arazo larri bat du: optimo lokaletan trabaturik gelditzen da. Arazo hau saihesteko zenbait hedapen proposatu dira; hala nola, tabu bilaketa [5], GRASP algoritmoa [4], Suberaketa Simulatua [10, 15], 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 honen «eboluzioa» ahalbidetuz, eta geroz eta soluzio hobeagoak lortuz. Atal honetan dauden algoritmo askok naturan bilatzen dute inspirazioa; era horretan, adibidez, algoritmo genetikoak [8] ditugu, eboluzioan oinarritutakoak, edo inurri-kolonia algoritmoa [3], inurrien talde-portaeran oinarritzen dena.

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 ezberdin erabil –eta, batez ere, konbina– daitezke: *dibertsifikazioa* eta *areagotzea*. Dibertsifikatzean espazioko eremu handiak aztertzen ditugu, baina xehetasun handirik gabe; helburua bilaketa espazioko eremu interesgarriei atzematea 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 lokala 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 pausuak populazioetan oinarritutako metodoei gehituz

Bibliografia

[1] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[2] Rainer E. Burkard, Eranda Çela, Panos M. Pardalos, and Leonidas S. Pitsoulis. The quadratic assignment problem, 1998.

[3] Marco Dorigo. *Optimization, Learning and Natural Algorithms*. PhD thesis, Politecnico di Milano, 1992.

[4] Thomas A Feo and Mauricio G.C Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8(2):67 – 71, 1989.

[5] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533–549, 1986.

[6] David E. Goldberg and Robert Lingle. Alleles Loci and the Traveling Salesman Problem. In *ICGA*, pages 154–159, 1985.

[7] Jatinder N.D. Gupta and Edward F. Stafford. Flow shop scheduling research after five decades. *European Journal of Operational Research*, (169):699–711, 2006.

[8] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, USA, 1975.

[9] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack problems*. Springer, 2004.

[10] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[11] G. Reinelt. TSPLIB - A t.s.p. library. Technical Report 250, Universität Augsburg, Institut für Mathematik, Augsburg, 1990.

[12] David B. Shmoys and Éva Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming*, 62(1-3):461–474, 1993.

[13] Eric Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, January 1993.

[14] H. Takagi. Interactive evolutionary computation: fusion of the capabilities of ec optimization and human evaluation. *Proceedings of the IEEE*, 89(9):1275–1296, Sep 2001.

[15] Vlado Černý. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45(1):41–51, January 1985.