

# BHLN: Populazioan oinarritutako algoritmoak

Borja Calvo, Usue Mori

## Laburpena

Aurreko kapitulan soluzio bakarrean oinarritzen diren zenbait algoritmo ikusi ditugu. Algoritmo hauek oso portaera ezberdina izan arren, badute ezaugarri komuna: bilaketa prozesua soluzio batetik bestera mugitzen da, soluzioak banan-banan aztertuz. Beraz, oso algoritmo egokiak dira bilaketa espazioaren eskualdea interesgarriak arakatzeko –bilaketa areagotzeko, alegia–. Alabaina, hainbat kasutan emaitza honak lortzeko bilaketa dibertsifikatzea ere beharrezkoa izan liteke. Izan ere, bilaketa lokalean oinarritzen diren algoritmo batzuk dibertsifikatzeko zenbait estrategia darabilte; honen adibide nabarmena tabu bilaketaren epe-luzeko memoria da.

Kapitulu honetan soluzioak banan-banakako azterketa alde batera utzita, multzoka erabiltzeari ekingo diogu, horixe baita, hain juxtu, populazioetan oinarritzen diren algoritmoen filosofia. Une oro, soluzio bakar bat izan beharrean soluzio multzo bat izango dugu. Testu inguru batzuetan multzo horri «soluzio-populazio» deritzo eta, hortik, algoritmo hauen izena. Bilaketa prozesuan multzo hori aldatuz joango da, helburu funtzioaren gidapean.

Bi dira, nagusiki, populazioetan oinarritzen diren algoritmoen hurbilketak: algoritmo ebolutiboak eta *swarm intelligence*. Lehenengo kategoriako algoritmoek, teknika ezberdinak erabiliz, populazioa eboluzionarazten dute, geroz eta soluzio hobeak izan ditzan. Adibiderik ezagunena algoritmo genetikoak dira. Bigarren algoritmo mota, berriz, zenbait animalien portaeran oinarritzen da. Hauen arteko adibiderik ezagunenak, esate baterako, inurriek, janaria eta inurritegiaren arteko distantziarik motzena topatzeko darabilten mekanismoa imitatu egiten du.

Kapitulua bi zatitan banaturik dago. Lehenengoan algoritmo ebolutioben eskema orokorra ikusi ondoren, algoritmo genetikoak [6] eta EDAk [7, 8] aurkeztzen dira. Bigarren zatian *swarm intelligence* [2] arloan proposaturiko bi algoritmo aztertuko dira.

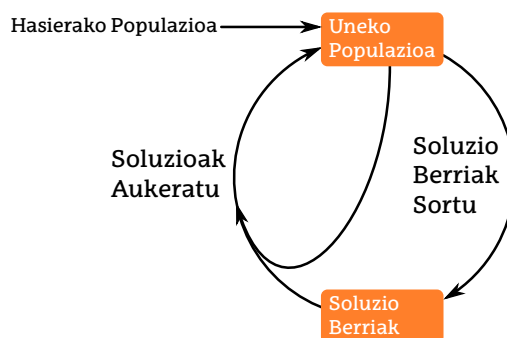
## 1 Algoritmo Ebolutiboak

1859. urtean Charles R. Darwin *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 honetan Darwinek hautespen naturalaren teoria aurkeztu zuen.

Eboluzioaren teoriak dioenez, belaunalditik belaunaldira zenbait mekanismoen bidez –mutazioak, esate baterako– aldaketak sortzen dira. Aldaketa batzuei esker indibiduoak hobeto egokitzen dira beraien inguruneari eta, ondorioz, bizirik mantentzeko eta, batez ere, ugaltzeko probabilitateak handitzen dira. Era berean, noski, aldaketa batzuk kaltegarriak izan daitezke, bizitzeko aukerak murriztuz. Kontutan hartuz aipatutako aldaketak heredatu egiten direla, ezaugarri onak generazioz generazio pasatzen dira; kaltegarriak direnek, ostera, galtzeko joera izaten dute. Mekanismo honen bidez, espezieak beraien ingurunera egokitzeko gai dira.

Hirurogeigarren hamarkadan ikertzaileek Darwinaren lana inspiraziotzat hartu zuten optimizazio metahuristikoa diseinatzeko; gaur egun konputazio zientziaren arlo oso bat da konputazio ebolutiboa. Atal honetan bi algoritmo mota aztertuko ditugu, algoritmo genetiko klasikoak [6] eta EDAk (*Estimation of Distribution Algorithms*) [7, 8].

Algoritmo ebolutibotan bi dira giltzarri diren elementuak: hautespena eta soluzio berrien sorkuntza. Naturan bezala, soluzio onak aukeratuko ditugu hurrengo belaunaldiara pasatzeko. Soluzio txarrenak deusestatzen ditugunez, populazioa osatzeko soluzio berriak beharko dira; sorkuntza prozesua aukeratu ditugu soluzioak hartzen dituen abia-puntutzat.



Irudia 1: Algoritmo ebolutiboaren eskema orokorra

Diferentziak diferentzia, algoritmo ebolutiboaren eskema orokorra defini daiteke (ikusi 1 irudia). Algoritmoaren sarrera-puntua hasierako populazioa izango da; populazio horretatik abiatuz, algoritmoa begizta nagusian sartzen da, non bi pausu tartekatzen dira. Lehenik, uneko populazioan dauden soluzioetatik batzuk aukeratzen dira. Ondoren, soluzio horiek erabiliz populazio berri bat sortzen da. Begizta nagusia etengabekoa denez, zantzen irizpide erabiltzen dira algoritmoa amaitutzat emateko.

Hurrengo ataletan eskema orokor hau nola gauzatzen den ikusiko dugu. Dena dela, zenbait pausu orokorrak diren legez, algoritmo konkretuak ikusi aurretik aztertuko ditugu.

## 1.1 Urrats orokorrak

Badaude zenbait pausu algoritmo ezberdinetan oso antzerakoak direnak. Izan ere, ikusiko ditugun algoritmoen arteko diferentzia nagusia soluzio berrien osaketan datza. Atal honetan gaionontzeko urratsak aztertuko ditugu eta hurrengo ataletan soluzioen sorkuntzari ekingo diogu.

### 1.1.1 Populazioaren hasieraketa

Hasierako populazioa da algoritmoaren abia-puntua eta, hortaz, bere sorkuntza oso pausu garrantzitsua da. Izan ere, askotan garrantzi gutxiegi ematen zaio pausu honi, nahiz eta oso eragin handia izan azken emaitzan.

Algoritmoen xedea soluzio honak topatzea izanda, pentsa dezakegu hasierako populazio on bat sortzeko soluzio onak behar ditugula; alabaina, dibertsitatea soluzioen kalitatea bezain garrantzitsua da. Izan ere, oso soluzio antzerakoak baldin baditugu, onak izan arren, populazioaren eboluzioa oso zaila izango da eta algoritmoaren konbergentzia goiztiarra gertatuko da.

Hortaz, hasierako populazioa sortzean bi aspektu izan behar ditugu kontutan: kalitatea eta dibertsitatea. Kasu gehienetan ausazko hasieraketa erabiltzen da lehen populazioa sortzeko, hau da, ausazko soluzioak sortzen dira populazioa osatu arte. Estrategia hau erabiliz dibertsitate handiko populazioa sortuko dugu, baina kalitatea ez da handia izango.

Populazioa guztiz ausaz sortzen badugu dibertsitatea handia izan arren, badaude beste estrategia batzuk –sasiausazkoak direnak– dibertsitatea maximizatzen. Hori dela eta, proposatu dira beste prozedura batzuk populazioak sasi-ausaz sortzeko dibertsitatea maximizatuz. Esate baterako dibertsifikazio sekuentzialean soluzio berri bat onartzeko populazioan dauden soluzioekiko distantzia minimo batera egon behar da. Adibide moduan, demagun 10 tamainako populazio bat sortu nahi dugula non soluzioak 25 tamainako bektore bitarrak diren. Dibertsitatea bermatzeko beraien arteko Hamming distantzia minimoa 10 izan behar dela inposa dezakegu. Jarraian dagoen kodeak horrelako populazioak sortzen ditu:

```

> hamm.distance <- function (v1 , v2){
+   d <- sum(v1!=v2)
+   return(d)
+ }
>

```



```
> rnd.binary <- function(n){  
+   return (runif(n) > 0.5)  
+ }
```

Lehenik, Hamming distantzia neurtzeko eta ausazko bektore bitarrak sortzeko funtzioak sortzen ditugu. Gero, soluzioak asuaz sortzen ditugu eta, distantzia minimoko baldintza bete ezean, deusestatzen dira; prozedura nahi ditugun soluzio kopurua lortu arte exekutatzen da.

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

Hau prozedura ez da bat ere eraginkorra, zenbait kasutan soluzio asko sortu beharko batitugu populazioa sortu arte. Beste alternatiba bat dibertsifikazio paraleloa da. Kasu honetan bilaketa espazioa zatitu egiten da eta azpi-espazio bakoitzetik ausazko soluzio bat erauzten da.

Orain arte dibertsitateari bakarrik erreparatu diogu. Hasierako popuazioaren kalitatea handitu nahi izanez gero, hasieraketa heuristikoak erabil daitezke. Hau lortzeko era sinple bat, GRASP algoritmoetan ausazko soluzioak sortzeko erabiltzen diren prozedurak erabiltzea da. Ikus dezagun adibide bat TSP problemarako; lehenik, Bavierako hirien problema kargatuko dugu.

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

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

Orain, tsp.greedy funtzioan oinarrituta ausazko soluzio onak sortzeko funtzio bat definitzen dugu.

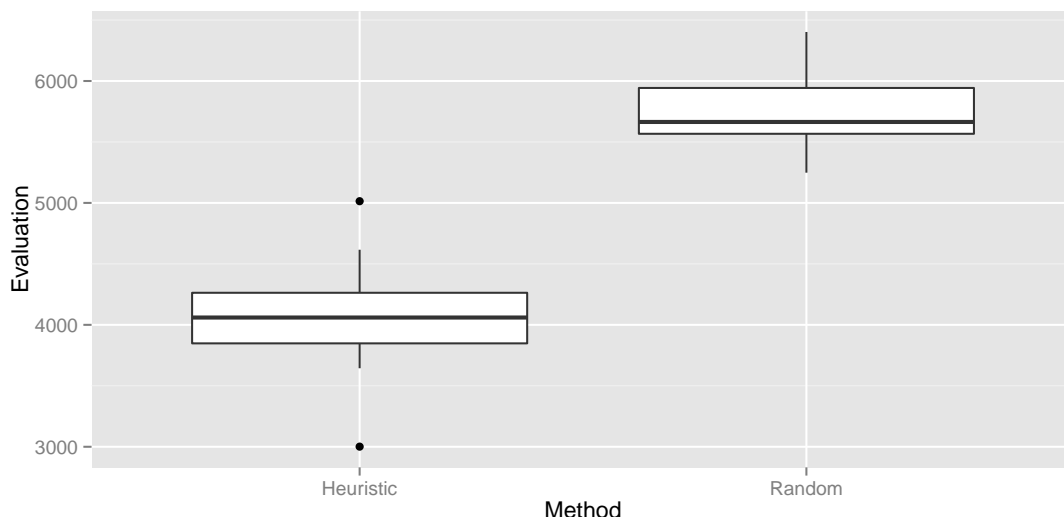
```
> rnd.sol <- function(cl.size = 5){  
+   tsp.greedy(cmatrix = cost.matrix , cl.size = cl.size)  
+ }
```

tsp.greedy funtzioak TSP-rako algoritmo eraikitzaile bat inplementatzen du; pausu bakoitzean, uneko hiritik zein hirira mugituko garen erabakitzen da. Hiria aukeratzeko gertuen dauden cl.size hiritetatik -5, gure kasuan- bat ausaz aukeratzen da. Populazioa sortzeko funtzio hau erabiliko dugu.

```
> pop.size <- 25  
> population <- lapply (1:pop.size , FUN = function(x) rnd.sol())
```

Hautagaien zerrenda problemaren tamainaraino handitzen badugu, pausu bakoitzean aukera guztietatik bat ausaz hartuko dugu, hots, guztiz ausazkoak diren soluzioak sortuko ditugu. Hau eginda populazioaren kalitatea goiko kodearekin lortutakoa baino txarragoa izango da:

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



Irudia 2: Ausazko hasieraketa eta hasieraketa heuristikoaren arteko konparaketa. Y ardatzak metodo bakoitzarekin sortutako soluzioen *fitness*-a adierazten du.

Bi populazioen ebaluazioak *boxplot* baten bidez aldera dezakegu.

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

2 irudiak lortutako emaitzak erakusten ditu; argi eta garbi ikus daiteke heuristikoa erabiliz sortutako soluzioak hobekiago direla.

Soluzioak sortzeko metodoak ez ezik, populazioaren tamainak ere badu eragin handia azken emaitzan. Izan ere, populazioaren tamaina egokitu behar den oso parametro garrantzitsua izango da. Populazioak txikiegiak badira, dibertsitatea mantentzea oso zaila izango da eta, hortaz, belaunaldi gutxitan algoritmoa konbergituko da. Beste aldetik, populazioa handiegia bada, konbergentzia abiadura motelduko da baina kostu konputazionala handituko da; hurrengo atalean adibide baten bidez ikusiko dugu hau. Ez dago irizpide finkorik tamaina ezartzeko, problema bakoitzeko egokitu beharko da. Edonola ere, irizpide orokor gisa esan dezakegu populazio azkar konbergitzen badu –hots, soluzioen arteko distantzia azkar txikitzen bada–, konponbidea populazioare tamaina handitzea izan daitekeela.

### 1.1.2 Hautespena

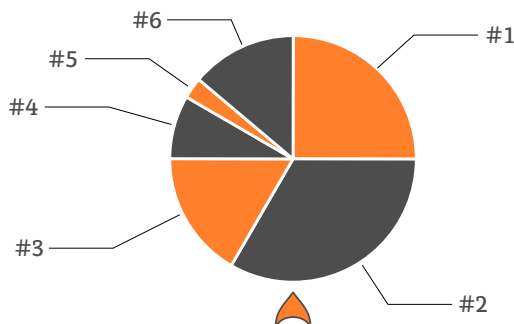
Algoritmo ebolutibotan populazioaren murriztea urrats garrantzitsua da, populazioaren eboluzioa kontrolatzen duen prozesua baita. Orokorrean, populazioan dauden soluziorik onenak hautatzea interesatuko zaigu eta hori da, hain zuzen, gehien erabiltzen den hautespena; bakarrik soluziorik onenak aukeratzen dituzenez, hautespen honi «elitista» deritza. Soluzio onak aukeratzea garrantzitsua da baina, dibertsitatea mantentzearen, tarteka soluzio txarrak ere sartzea komenigarria izan daiteke. Hau zuzenean egiterik egon arren, badaude beste hautespen metodoak era probabilistikoan egiten dutenak.

Erruleta-hautespena, (*Roulette Wheel selection*, ingelesez) deritzon estrategian soluzioak erruleta batean kokatzen dira; soluzio bakoitzari dagokion erruletaren zatia bere ebaluazioarekiko proportzionala izango da. 3 irudian ikus daitekeen bezala, erruleta jaurtitzen den bakoitzean indibiduo bat hautatzen da; hautatua izateko probabilitatea erruleta zatiaren tamaina eta, hortaz, indibiduoaren ebaluazioarekiko proportzionala da.

Indibiduo bat baino gehiago aukeratu behar baldin badugu, behar ditugun erruletaren jaurtiketak egin ditzakegu. Alabaina, honek alborapenak sor ditzake; efektu hau saihesteko erruletan puntu bakar bat markatu



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



Irudia 3: Erruleta-hautespena. Indibiduo bakoitzaren erruletaren zatia bere ebaluazioarekiko proportzionala da. Erruleta jaurtitzen den bakoitzean indibiduo bat aukeratzen da, bere *fitness*arekiko proportzionala den probabilitatearekin. Adibidean, 2. indibiduo da hautatu dena.

beharrean (gezia, 3 irudian), behar ditugun puntuak finkatu ahal ditugu. Era honetan, jaurtiketa bakar batekin nahikoa da behar ditugun indibiduo guztiak hautatzeko. Teknika hau populazioa murrizteko zein indibiduoak gurutzatzeko hautespenean erabil daiteke.

*Fitness*aren magnitudea problema eta, are gehiago, instantzien araberakoa da. Hori dela eta, probabilitateak zuzenean helburu funtzioaren balioarekiko proportzionalak badira, oso distribuzio radikalak izan ditzakegu. Arazo hau ekiditeko, helburu funtzioaren balioa zuzenean erabili beharrean soluzioen ranking-a erabili ahal da.

Beste hautespen probabilistiko bat Lehiaketa-hautespena da. Estrategia honetan hautespena bi pausutan egiten da. Lehenengo urratsean indibiduo guztietatik azpi-multzo bat aukeratzen da, guztiz ausaz (ebaluazioa kontutan hartu barik). Ondoren, aukeratu ditugun indibiduoetatik onena hautatzen dugu. Azpi-multzoa ausaz aukeratzen denez, bertan oso soluzio txarrak leudeke eta, hortaz, nahiz eta onena hautatu, soluzioa txarra izan daiteke.

### 1.1.3 Gelditze Irizpideak

Lehen apiatu bezala, algoritmo ebolutioben begizta nagusia amaigabea da eta, beraz, algoritmoa gelditzeko irizpideren bat ezarri behar dugu, bilaketari amaiera emateko. Hurbilketarik sinpleena irizpide estatikoak erabiltzea da, hala nola, denbora maximoa ezartzea, ebaluazioak mugatzea, etab.

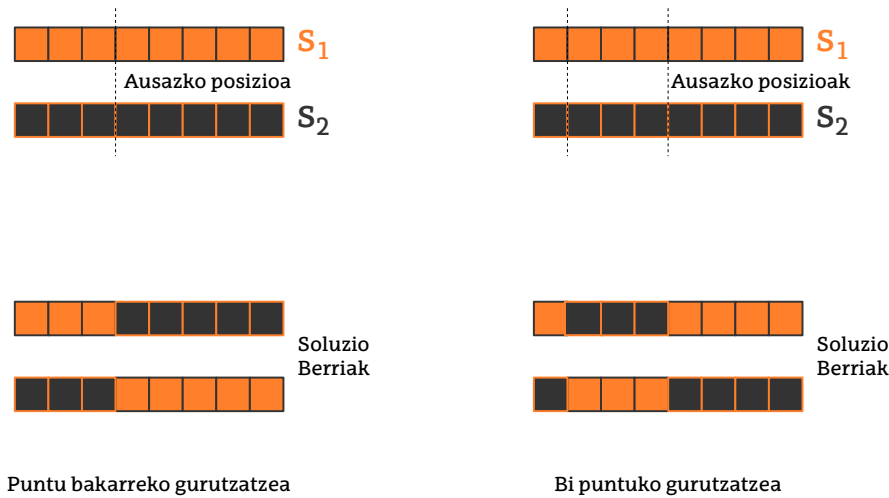
Gelditzeko irizpideak dinamikoak ere izan daitezke, eboluzioaren prozesuari erreparatzen badiogu. Balu-naldiz balaundi populazioan dauden soluzioak geroz eta hobeak dira eta, aldi berean, populazioaren dibertsitatea murrizten da. Beste era batean esanda, populazioko soluzioek soluzio bakar batera konbergitzeko joera dute.

Hori gertatzen denean eboluzioa moteltzen denez, populazioaren dibertsitatea gelditzeko irizpide gisa erabili ahal da. Dibertsitatea soluzioei zein beraien *fitness*-ari erreparatuz neur daiteke. Esate baterako, soluzioen arteko distantzia neurtzerik badago, indibiduen arteko batz besteko distantzia minimo bat ezar dezakegu.

### 1.1.4 Algoritmo Genetikoak

Atal honetan algoritmo genetikoetan [6] soluzio berriak algoritmo genetikoetan nola sortzen diren ikusiko dugu. Algoritmo hauek naturan espeziekin gertatzen dena imitatzen dute. Era honetan, zenbait paralelismo ezar daitezke:

- Espezieen indibiduoak = Problemaren soluzioak
- Indibiduen egokitasuna *-fitness-*a, alegia = Soluzioaren ebaluazioa



Irudia 4: Gurutzatze-operadoreak bektoreen bidezko kodeketarekin erabiltzeko

- Espeziearen populazioa = Soluzio multzoa
- Ugalketa = Soluzio berrien sorkuntza

Beraz, algoritmo genetikoetan soluzio berriak sortzeko indibiduen ugalketan oinarrituko gara. Ugalketa prozesuaren xedea zenbait indibiduo emanda –bi, normalean–, indibiduo gehiago sortzea da. Ohikoa prozesu hau bi pausutan banatzea da: soluzioen gurutzaketa eta mutazioa. Lehenaren helburua «guraso» -soluzioek duten ezaugarriak soluzio berriei pasatzea da. Bigarrenarena, berriz, sortutako soluzioetan ezaugarri berriak sartzea da. Jarraian soluzioak maneiatzeko bi operadore hauek aztertuko ditugu.

### 1.1.5 Gurutzaketa

Bi soluzio –edo gehiago– gurutzatzen ditugunean euren propietateak sortutako soluzioei transmititzea da gure helburua. Hori lortzeko, soluzioei operadore mota berezi bat aplikatuko diegu, «gurutzte-operadorea» –*crossover*, ingelesez–. Operadore honek soluzioen kodeketarekin dihardu eta, beraz, operadorea hautatzean soluzioak nola adieratzen dugun aintzat hartu beharko dugu.

Badaude zenbait operadore kodeketa klasikoekin erabil daitezkeenak. Ezagunena puntu bakarreko gurutzatzea –*one-point crossover*, ingelesez –deritzona da. Demagun soluzioak bektoreen bidez kodetzen ditugula. Bi soluzio,  $s_1$  eta  $s_2$  parametro gisa hartuz, operadore honek beste bi soluzio berri sortzen ditu. Horretarako, lehenik eta behin, ausazko posizio bat  $i$  aukeratu behar da. Gero, lehenengo soluzio berria  $s_1$  soluziotik lehenengo  $i$  elementuak eta  $s_2$  soluziotik beste gainontzekoak kopiauz sortuko dugu. Era berean, bigarren soluzio berria  $s_2$ -tik lehenengo elementuak eta besteak  $s_1$ -etik kopiauz sortuko dugu. 4 irudiaren ezkerrean adibide bat ikus daiteke. Irudiak ideia nola orokor daitezkeen ere erakusten du, puntu bakar bat erabili beharrean bi, hiru, etab. puntu erabiliz.

Operadore hau `metaheuR` liburutegian inplementaturik dago, `k.point.crossover` funtzioan. Ikus ditzagun 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"

> k.point.crossover(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"

> k.point.crossover(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"

> k.point.crossover(A.sol , B.sol , 20)

## Warning in k.point.crossover(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  $n - 1$  puntu erabil ditzakegu operadore honetan; edonola ere, balio handiagoa erabiltzen badugu funtzioak abisu bat emango du eta parametroa bere balio maximoan ezarriko du. Balio maximoarekin jatorrizko soluzioen elementuak tartekatuko dira; operadore honi uniform crossover deritzo.

Erabiliko dugun puntu kopurua eragin handia izan dezake algoritmoaren performantzia eta, hortaz, egokitu behar den algoritmoaren parametrotzat hartu beharko genuke.

Ikusitako operadorea nahiko orokorra da, ia edozen bektoreari aplikatu ahal baitzaio. Hala eta guztiz ere, kodeketa batzuetan operadore bereziak erabil daitezke[5]. Esate baterako, bektore errealek baldin baditugu, bi bektore harturik era ezberdinetan konbina daitezke; adibidez, batz bestekoa kalkulatu. Ikus dezagun operadore hau nola inplementa daitekeen R-n:

> mean.crossover <- 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

> mean.crossover(s1 , s2)

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

Permutazioak ere bektoreak dira baina, kasu honetan, *k point crossover* operadorea ezin da erabili, permutazioetan ditugun murrizketak direla eta. Hortaz, permutazioak gurutzatzeko operadore bereziak behar ditugu. Aukera asko izan arren [9], hemen puntu bakarreko gurutzatze operadorearen baliokidea ikusiko dugu. Lehenik eta behin, ikus dezagun zergatik puntu bakarreko operadorea ezin da zuzenean aplikatu permutazioei. Izan bitez bi permutazio,  $s_1 = 12345678$  eta  $s_2 = 87654321$ , eta gurutzatze 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  $s' = 12354321$  izango zen baina, zoritxarrez, hau ez da permutazio bat.

Nola saihestu daiteke arazo hau? Soluzio simple bat haxe da: lehenengo posizioak zuzenean soluzio batetik kopiatzea; besteak zuzenean beste soluziotik kopiatu beharrean, ordena bakarrik erabiliko dugu. Hau da, soluzio berria sortzeko  $s_1$ -etik lehenengo 3 elementuak kopiatuko ditugu, 123, eta falta direnak, 45678,  $s_2$ -an agertzen den ordenean kopiatuko ditugu, hots, 87654. Eraitza, beraz,  $s' = 12387654$  izango da eta beraz, orain bai, permutazio bat lortu dugu. Era berean, beste soluzio berri bat sor daiteke  $s_2$ -tik lehenengo hiru posizioak kopiatuz (876) eta beste gainontzeko guztiak  $s_1$ -an duten ordenean kopiatuz (12345); beste soluzioa, beraz, 87612345 izango da. Operadore honi «Order crossover» deritzo eta *metaheuR* liburutegian inplementaturik dago, *order.crossover* funtzioan <sup>1</sup>.

```
> sol1 <- random.permutation(10)
> sol2 <- identity.permutation(10)
> as.numeric(sol1)

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

> as.numeric(sol2)

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

> new.solutions <- order.crossover(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
```

### 1.1.6 Mutazioa

Naturan bezala, gure populazioa eboluzionatzeko dibertsitatea garrantzitsua da. Hori dela eta, behin soluzio berriak lortuta gurutzatze-operadorearen bidez, soluzio hauetan ausazko aldaketak eragin ohi da; aldaketa hauek mutazio operadorearen bidez sortzen dira.

Mutazioaren kontzeptua ILS algoritmoa perturbazioaren antzerakoa da; izan ere, operadore berdina erabil daitezke. Esate baterako, permutazio bat mutatzeko ausazko trukaketak erabil ditzakegu. ILS-an bezala, algoritmoa diseinatzean erabaki behar dugu zenbat aldaketak sortuko ditugun – adibidean, zenbat posizio trukatu ditugun –.

<sup>1</sup>Funtzio honetan inplementatuta dagoena *2-point crossover* operadorea da. Hau da, bi puntu erabiltzen dira eta, soluzioak eraikitzeko, bi puntuen artean dagoena soluzio batetik kopiatu ondore, beste gainontzeko elementuak beste soluzioan duten ordenean erabiltzen dira.





### 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

```

---

#### Algoritmoa 1.1: Algoritmo genetikoaren sasikodea

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 beharko luke. Bestalde, bilaketa prozesua soluzio bideragarrien espazio osoa arakatzeko gaitasuna izan beharko luke eta hori bermatzeko mutazio operadoreak edozein soluzio sortzeko gai izan behar du. Hau da, edozein soluzio hartuta mutazio operadorearen bidez beste edozein soluzioa sortzeko posible izan beharko luke. Amaitzeko, lokaltasuna ere mantendu behar da –alegia, mutazioak eragindako aldaketa txikia izan behar da–, bestela gurasoengandik heredatutako ezaugarriak galdu egingo dira.

Mutazio operadorea era probabilistikoan aplikatzen da; hau da, ez zaie soluzio guztiei aplikatzen. Hortaz, mutazioari lotutako bi parametro izango ditugu: mutazio probabilitatea eta mutazioaren magnitudea.

Algoritmo genetiko orokorra 1.1 irudian ikus daiteke. Algoritmoan dagoen sasikodea `basic.genetic.algorithm` funtzioan implementaturik dago. Ikus dezagun nola erabil daitekeen funtzio hau *graph coloring* problema bat ebazteko. Lehenik, ausazko grafo bat sortuko dugun problemaren instantzia sortzeko.

```

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

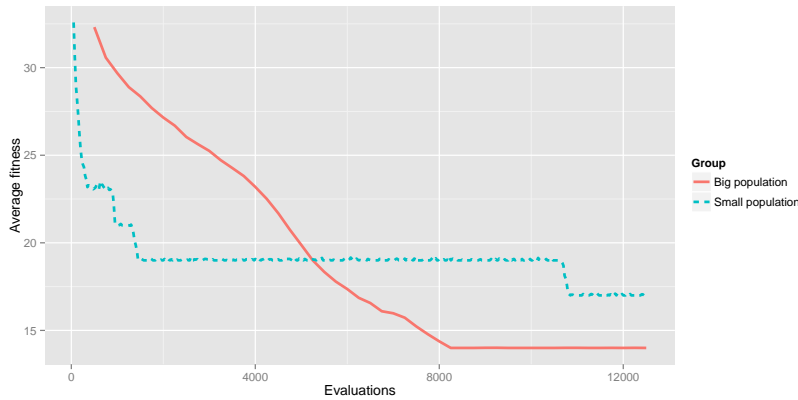
```

Orain zenbait elementu definitu behar ditugu. Lehenengoa, hasierako populazioa izango da. Populazioa sortzeko bere tamaina ezarri behar dugu. Hau algoritmoaren parametro garrantzitsu bat denez, bi balio erabiliko ditugu, emaitzak alderatzeko:  $n$  eta  $10n$ . Soluzioak ausaz sortuko ditugu eta, gero, bideragarriak ez badira, zuzenduko ditugu.

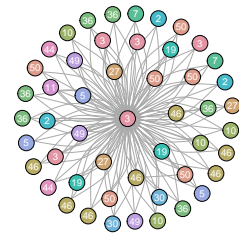
```

> n.pop.small <- n
> n.pop.big <- 10*n

```



(a) Algoritmo genetikoaren progresioa



(b) Lortutako soluzioa

Irudia 5: Algoritmo genetikoaren progresioa *graph coloring* problema batean, bi populazio tamaina ezberdin erabiliz. Ezkerrean, populazio tamaina handiarekin lortutako soluzioa ikus daiteke.

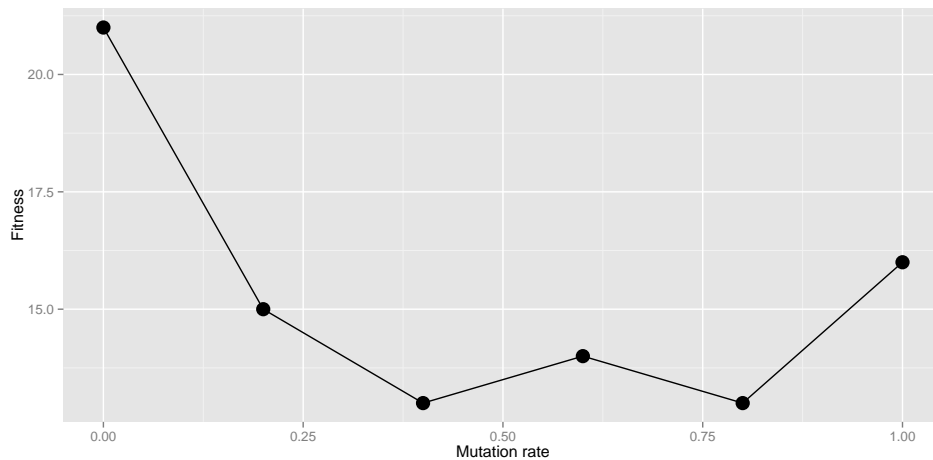
```
> levels <- paste("C" , 1:n , sep = "")
> rnd.sol <- function(x){
+   sol <- factor(paste("C" , sample(1:n , size = n , replace = TRUE) ,
+     sep = "")) , levels = levels)
+   return(gcp$correct(sol))
+ }
> pop.small <- lapply(1:n.pop.small , FUN = rnd.sol)
> pop.big <- lapply(1:n.pop.big , FUN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

Hasierako populazioaz gain, ondoko parametro hauek ezarri behar ditugu:

- Hautespen operadoreak - Hurrengo belaunaldira pasatuko diren soluzioak aukeratzeko hautespen elitista erabiliko dugu, populazio erdia aukeratuz; zein soluzio gurutzatuko diren aukeratzeko, berriz, lehiaketa hautespena erabiliko dugu.
- Mutazioa - Soluzioak mutatzeko `factor.mutation` soluzio erabiliko dugu. Funtzio honek zenbait posizio ausaz aukeratzen ditu eta bere balioak ausaz aldatzen ditu. Funtzioak parametro bat du, `ratio`, zeinek aldatuko diren posizioen ratioa adierazten duen. Gure kasuan 0.1 balioa erabiliko dugu, alegia, %10 posizio aldatuko dira mutazioa aplikatzen denean. Zein probabilitatearekin mutatu ditugun soluzioak ere finkatu behar da, `mutation.rate` parametroaren bidez; gure kasuan probabilitatea bat zati populazioaren tamaina izango da.
- Gurutzaketa - Soluzioak gurutzatzeko *k point crossover* erabiliko,  $k = 2$  finkatuz.
- Beste parametro batzuk - Algoritmo genetikoaren parametroaz gain, beste bi parametro finaktuko ditugu, `non.valid = 'discard'`, bideraezina diren soluzioak baztertu behar direla adierazteko, eta `resources`, gelditze irizpidea finkatzeko ( $5n^2$  ebaluazio kopuru maximoa erabiliko dugu).

Jarraian parametro hauek erabiliz algoritmo genetikoaren exekutatzeko kodea dago.

```
> args <- list()
> args$evaluate <- gcp$evaluate
> args$initial.population <- pop.small
> args$select.subpopulation <- elitist.selection
```



Irudia 6: Mutazioaren probabilitatearen eragina algoritmo genetikoaren emaitzan. Irudian ikus daitekeen bezala, soluziorik onenak ematen duen balioa 0.5 inguruan dagoena da (zehazki, 0.6).

```
> args$selection.ratio      <- 0.5
> args$select.cross        <- tournament.selection
> args$mutate              <- factor.mutation
> args$ratio               <- 0.1
> args$mutation.rate       <- 1 / length(args$initial.population)
> args$cross               <- k.point.crossover
> args$k                   <- 2
> args$non.valid           <- 'discard'
> args$resources           <- cresource(evaluations = 5*n^2)
>
> bga.small <- do.call(basic.genetic.algorithm , args)
>
> args$initial.population  <- pop.big
> args$mutation.rate       <- 1 / length(args$initial.population)
>
> bga.big <- do.call(basic.genetic.algorithm , args)
> plot.progress(list("Big population" = bga.big , "Small population" = bga.small) ,
+               size = 1.1) + labs(y = "Average fitness") + aes(linetype = Group)
```

5 irudian bi populazio erabiliz bilaketaren progresioa ikus daiteke. Populazioa txikia denean algoritmoak oso azkar konbergitzen du 19 kolore darabiltzan soluzio batera. Populazioko soluzio gehienak oso antzerakoak direnean soluzio berriak sortzeko bide bakarra mutazioa da, baina prozesu hori oso motela denez, grafikan ikus daiteke soluzioen batzaz besteko *fitnessa* ez dela aldatzen.

Populazioaren tamaina handitzen dugunean konbergentzia zailagoa da eta grafikan ikus daiteke helbur funtzioaren balioaren eboluzioa motelagoa izan arren, lortutako soluzioa hobea dela.

Populazioaren tamaina ez ezik, beste hainbat parametro eragin handia izan dezakete algoritmoaren emaitzan; esate baterako, mutazioaren probabilitatea. Adibide gisa, populazio tamaina txikia erabiliz mutazio probabilitate ezberdinak erabiliko ditugu, eta, ondoren, bakoitzarekin lortutako emaitzak alderatuko ditugu.

```
> args$initial.population  <- pop.small
> args$verbose             <- FALSE
> args$resources           <- cresource(evaluations = n^2)
```



```
>
> test.mutprob <- function (rate){
+   args$mutation.rate <- rate
+   res <- do.call(basic.genetic.algorithm , args)
+   evaluation(res)
+ }
>
> ratios <- seq(0,1,0.2)
> evaluations <- sapply(ratios , FUN = test.mutprob)
>
> df <- data.frame("Mutation_rate" = ratios , "Fitness" = evaluations)
> ggplot(df , aes(x=Mutation_rate , y = Fitness)) + geom_line() + geom_point(size = 5) +
+   labs(x = "Mutation rate")
```

6 irudian konparaketaren emaitzak ikus daitezke. Grafikoan agerian dago probabilitate txikiegiak zein handiegiak kaltegarriak direla bilaketarako; izan ere, probabilitate egokiena 0.5 ingurukoa da. Edonola ere, kontutan hartu behar da probabilitate txikien kasuan emaitzak txarrak direla konbergentzia goiztiarra dugulak baina, probabilitate handiekin berriz, arazoa justu kontrakoa izan daiteke, alegia, jarri dugun muga dela eta populazioa konbergitzen duela. Hau egiaztatzeko, errepika dezakezu goiko experimentua ebaluazio kopuru maximoa handituz.

## 1.2 Estimation of Distribution Algorithms

Algoritmo genetikoetan uneko populazioa indibiduo berriak sortzeko erabiltzen da. Prozesu honetan, naturan inspiratutako operadoreen bidez burutzen dena, populazioan dauden ezaugarriak mantentzea espero dugu. Edonola ere, prozesua korapilatsua izan daiteke. Hori dela eta, zenbait ikertzaile ideia hau hartu eta ikuspen matematikotik birformulatu zuten; gurutzatze-operadoreak eta mutazioa erabili beharrean, eredu probabilistikoa erabiltzea proposatu zuten, populazioaren «esentzia» kapturatzeko helburuarekin. Hauxe da, EDA – *Estimation of Distribution Algorithms* – algoritmotan erabiltzen den ideia.

Algoritmo genetikoaren eta EDA motako algoritmoen artean dagoen diferentzia bakarra indibiduo berriak sortzean dago. Gurutzatzea eta mutazioa erabili beharrean, uneko populazioa eredu probabilistikoa bat «ikasteko» erabiltzen da. Ondoren, eredu hori laginduko dugu nahi ditugun indibiduo adina sortzeko.

EDA algoritmoen gakoa, beraz, eredu probabilistikoa da. Ildo honetan, esan beharra dago eredua soluzioen kodeketari lotuta dagoela, soluzio adierazpide bakoitzari probabilitate bat esleitu beharko diolako.

Konplexutasun ezberdineko eredu probabilistikoen erabilera proposatu da literaturan, baina badago hurbilketa simple bat oso hedatua dagoena: UMDA – *Univariate Marginal Distribution Algorithm* –. Kasu honetan soluzioaren osagaiak – bektore bat bada, bere posizioak – independenteak direla suposatuko dugu eta, hortaz, osagai bakoitzari dagokion probabilitate marjinala estimatu beharko dugu. Gero, indibiduoak sortzean osagaiak banan-banan aukeratzeko ditugu probabilitate hauek kontutan hartuz.

Probabilitate marjinalak maneiatzeko *metaheuR* paketeko *UnivariateMarginals* objektua erabil dezakegu. Bere erabilera ikusteko, populazio txiki bat sortuko dugu eta probabilitate marjinalak kalkulatzeko ditugu.

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

Orain, *univariateMarginals* funtzioa erabiliz probabilitate marjinalak kalkulatzeko 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 eta populazioak 5 soluzio ditu. Lehenengo elementuari erreparatzen badiogu, 5 soluzioetatik lehenengo biak 1 balioa, laugarrenak 2 balioa eta beste biak 3 balioa dute. Hortaz, elementu horretarako 1 eta 3 balioen probabilitatea 0.4 izango da  $-\frac{2}{5}$ , alegia— eta 2 balioaren probabilitatea 0.2 izango da, marginaleen taulan ikus daitekeen bezala.

Ikasitako eredu probabilitistikoa soluzio berriak sortzeko erabil daiteke, posizioz posizio balioak dagokion marginala laginduz; laginketa `simulate` funtzioaren bitartez egiten da.

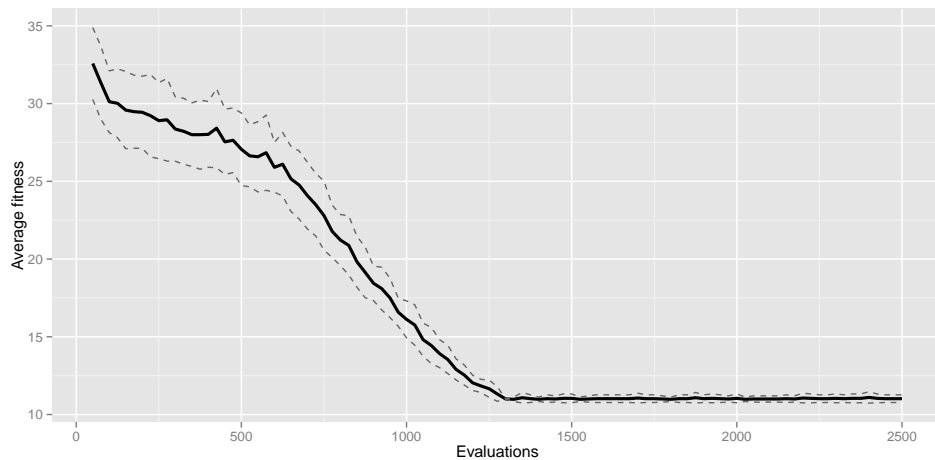
```
> simulate(model , nsim = 2)
```

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

UMDA erabiliz aurreko ataleko problema ebatz dezakgu. Horretarako, bakarrik hautespen operadoreak eta ereduak ikasteko funtzioak zehaztu behar ditugu —betiko parametroz gain—.

```
> args <- list()
> args$evaluate <- gcp$evaluate
> args$initial.population <- pop.small
> args$select.subpopulation <- elitist.selection
> args$selection.ratio <- 0.5
> args$learn <- univariateMarginals
> args$non.valid <- 'discard'
> args$resources <- cresource(evaluations = n^2)
>
> umda <- do.call(basic.eda , args)
>
> plot.progress(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")
```

Bilaketaren progresioa 7 irudian erakusten da. Marra etenek populazioan dauden soluzioen *fitness*aren desbiderazioa erakusten dute. Eboluzioak aurrera egiten duen heinean, populazioaren dibertsitatea murrizten da eta hau desbiderazioaren txikiagotzean islatzen da; bilaketak 11 kolore darabiltzan soluzio batera konbergitzen du.



Irudia 7: UMDA algoritmoaren progresioa *graph coloring* adibidean. Marra jarraituak populazioko soluzioen batzaz-beste *fitness* adierazten du; marra etenek, berriz, desbiderazio estandarra adierazten dute. Ikus daiteke populazioak konbergitzen duen heinean soluzioen *fitness*aren bariabilitadeari txikiagotzen dela.

Marjinalak zuzenean edozein bektoreekin erabil daitezke; alabaina, balio errealek baditugu, marjinalak zein probabilitate distribuzioarekin modelatuko ditugun erabaki beharko dugu –distribuzio normala, adibidez–. Dena dela, soluzioek murrizketak dituztenean gauzak konplika daitezke; honen adibidea permutazioak dira.

Permutazio multzo bat badugu, posible da marjinalak estimatzea, baina eredu lagintzen dugunean ez ditugu permutazioak lortuko, balio errepikatuak ager baitaitezke. Hona hemen adibide bat:

```
> n <- 5
> perm.pop <- lapply(1:50, FUN = function(x) factor(as.numeric(random.permutation(n)),
+                                               levels = 1:n))
> perm.umd <- univariateMarginals(perm.pop)
> simulate(perm.umd)

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

Arazo hau sahiesteko, soluzio berriak lagintzean permutazioetan ditugun murrizketak aintzat hartu behar dira. Beraz, laginketa prozesuan lehenengo elementua ausaz aukeratuko dugu, marjinala erabiliz.

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

Orain, bigarren elementua aukeratu aurretik, lehenengo posizioan egongo den elementua kendu behar dugu aukeretatik eta marjinalak eguneratu behar ditugu –erabili dugun elementuaren probabilitatea kendu eta normalizatu, gelditzen diren elementuen probabilitateen batura 1 izan dezan–:

```
> 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)
```

Prozesua 3. eta 4. elementuak erauzteko errepika dezakegu.

```
> 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 gelditzen den elementu bakarra izango da.

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

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

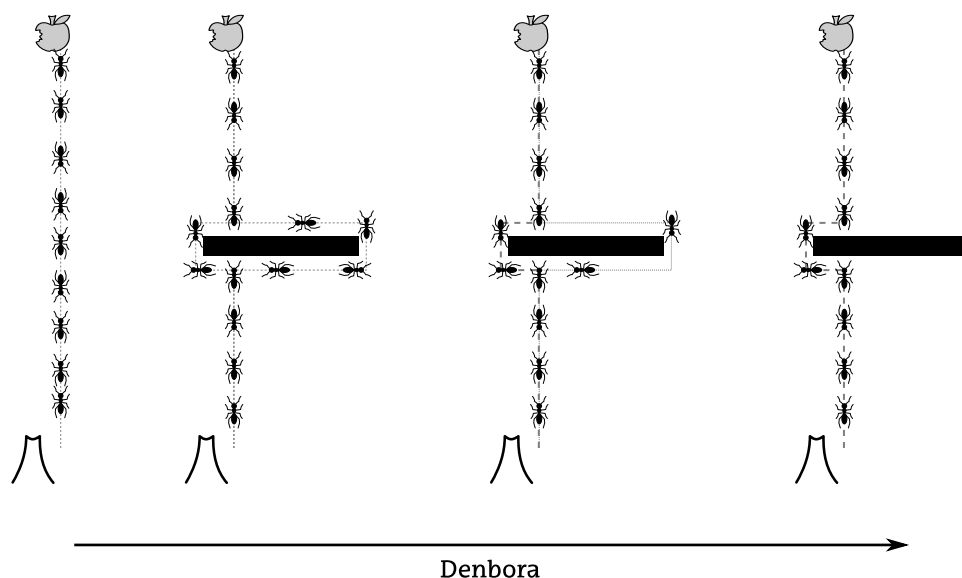
Prozesu honi esker probabilitate marjinalak erabil daitezke permutazioak sortzeko, baina arazo bat dauka; ereduia lagintzen dugun bakoitzean probabilitateak aldatzen ditugu eta, hortaz, lagintzen duguna ez da populaziotik ikasi duguna. Beste era batean esanda, populaziotik ateratako «esentzia» gal genezake. Hau ez gertatzeko, permutazio espazioetan definitutako probabilitate distribuzioak erabil ditzakegu; esate baterako, Mallows ereduia, *metaheuR* paketea dagoen *MallowsModel* objektuak inplementatzen duena.

## 2 Swarm Intelligence

Eboluzioaren bidez naturak indibiduen diseinuak «optimizatzeko» gai da; alabaina, hau ez da naturak erabiltzen duen estrategia bakarra. Optimizazio algoritmoak inspiratu dituen beste adibidea bat animalia sozialena da. Zenbait espezieetan –intsektuak, batik bat– banan-banan hartuta, indibiduoak oso izaki sinpleak dira baina, multzoka, ataza konplexuak burutzeko gai dira. Esate baterako, inurriek eta erleek elikagai-iturri onenak aukeratzeko gai dira eta, hauek agortzen direnean, ingurunea esploratzen duten indibiduen kopurua handi dezakete iturri berriak lortzeko; era berean, bizitzeko toki berriak bilatzean toki egokienak aukeratzeko gai dira.

Erabaki guzti horiek ez dira era zentralizatuan hartzen –alegia, ez dago «nagusi» bat agintzen dunea–. Portaera hauek indibiduen portaera sinpleen eta, batez ere, indibiduen arteko komunikazioari esker agertzen dira. Beste era batean esanda, mekanismo sinplei esker kolonia batean dauden indibiduoak autoantolatzen dira.

Portaera hauek dira *swarm intelligence* deritzon arloaren inspirazioa. Kontzpetua lehenengo aldiz robotika arloan proposatu zen [1], baina laister optimizazio mundura hedatu zen. Izan ere, 90. hamarkadan inurri kolonien optimizazioa –*Ant Colony Optimization*, ingelesez– proposatu zen [4, 3].



Irudia 8: Feromonaren erabilera. Hasierako egoeran biderik motzena feromonaren bidez markatuta dago. Bidea mozten dugunean, inurriek, eskumara edo ezkerreara joango dira, probabilitate berdinarekin, feromonarik ez baitago ez ezkerrean ez eskuinean. Eskumako bidea luzeagoa da eta, ondorioz, ezkerreko bidearekiko inurri-fluxua txikiagoa da. Denbora igaro ahala, eskumako lorratzak ahulduko da; ezkerrekoa, berriz, indartuko da. Honek inurrien erabakia baldintzatuko du, ezkerretik joateko joera handiago sortuz eta, ondorioz, bi bideen arteko diferentziak handituz. Denbora nahiko igarotzen denean eskumako lorratzak guztiz galduko da eta inurriak bide motzetik bakarrik joango dira.

Hurrengo bi ataletan arlo honetan dauden bi algoritmo ezagunenak aztertuko ditugu, inurri kolonien optimizazioa eta *particle swarm optimization*.

## 2.1 Ant Colony Optimization

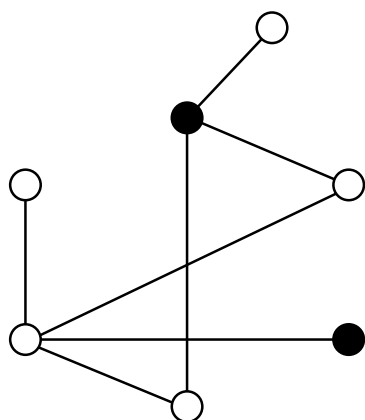
Inurriek, janaria topatzen dutenean, beraien koloniatik janarira biderik motzena topatzeko gaitasuna dute. Inurri bakar batek ezin du horrelakorik egin baina, taldeka, komunikazio mekanismo sinpleei esker ataza burutzeko gai dira. Erabiltzen den komunikabidea zeharkakoa da, darien molekula mota berezi bati esker: feromonak. Inurriek mugitzen direnean beste inurriek jarrai dezaketen feromona-lorratz bat uzten dute; geroz eta feromona gehiago, orduan eta probabilitate handiagoa datozen inurriak utzitako lorratzak jarraitzeko.

Bestaldetik, inurri batek elikagai-iturri bat topatzen duenean, bidetik uzten duen feromona kopurua iturriaren kalitateari egokitzen du; geroz eta kalitate handiagoa, orduan eta feromona gehiago. Sistemaren funtzionamendua ulertzeko kontutan hartu behar dugu feromonak molekula lurrunkorak direla, hots, denborarekin utzitako lorratzak galtzen direla.

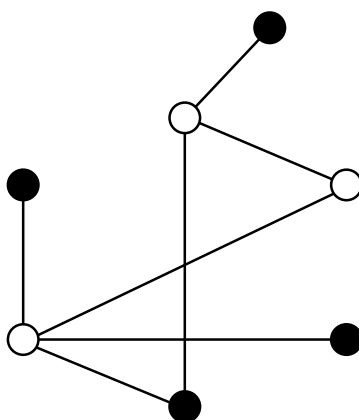
Arau sinple hauek erabiliz inurriek elikagai-iturri onenak aukeratzeko gai dira; are gehiago, elikagai eta inurritegiaren arteko biderik motzena topa dezakete. Mekanismoaren funtzionamendua 8 irudian ikus daiteke. Hasieran bide motzena feromona lorratzaren bidez markatuta dute inurriek. Bidea mozten badugu, eskuman eta ezkerrean ez dago feromonarik eta, hortaz, inurri batzuk eskumatik eta beste batzuk ezkerretik joango dira, probabilitate berdinarekin. Ezkerreko bidea motzagoa denez, denbora berdinean inurri gehiago igaroko dira, ezkerreko bidean feromona gehiago utziz. Ondorioz, datozen inurriak ezkerretik joateko joera handiago izango dute, bide hori indartuz. Eskumako bidean lorratzak apurka-apurka baporatuko da eta, denbora nahiko igarotzen bada, zeharo galduko da.

*Ant Colony Optimization* (ACO) deritzon metaheuristika inurrien mekanismoa hartzen du intuiziotzat. Bere inguruetik pasiatu beharrean, inurri artifizialak soluzio «sortzaileak» dira. Beraz, inurri artifizialek soluzioak sortuko dituzte, baina ez edonolakoak; soluzioak aurreko inurriek utzitako lorratzak jarraituz eraikitzen dira.

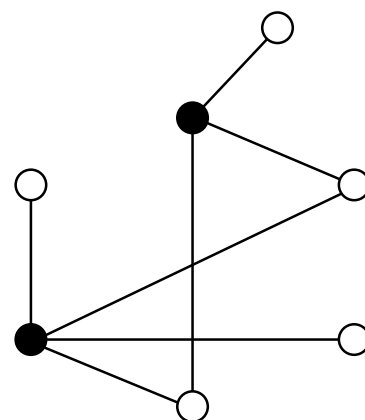




(a) Soluzio hau ez da menderatze-multzoa



(b) 4 tamainako menderatze-multzoa



(c) 2 tamainako menderatze-multzoa

Irudia 9: Irudiak MDS problemarako 3 soluzio jasotzen ditu –beltzez adierazita dauden nodoak–. Lehenengoa (ezkerrean dagoena), ez da bideragarria, soluzioan dagoen nodo bat soluzioan dauden nodeoi konektaturik ez baitago. Bigarren soluzioa bideragarria da, baina ez optimoa. Azken soluzioa optimoa da, ez baitago 1 tamainako soluzio bideragarririk.

Lorrazak feromona ereduen bidez adierazten dira eta bi eratan eguneratzen dira. Alde batetik, inurriek sortutako soluzioen kalitatea –hots, helburu funtzioaren balioa– feromona gehitzeko erabiltzen da. Bestaldetik, iterazioz iterazio feromona kopuruak txikituko ditugu, molekularen baporazioa simulatuz.

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

Demagun MIS problema bat ebatzi nahi dugula. Problema honetarako soluzioak bektore bitarren bidez kodetzen ditugu eta, hortaz, bektorearen posizioak soluzioen osagitzat har ditzakegu. Posizio bakoitzen bi balio posible daude, 0 edo 1. Inurri batek soluzio bat sortu behar duenenan, lehenik, bektorearen lehenengo posizioko balioa, 0 edo 1, aukeratu beharko du. Horretarako, feromona kopurua hartuko du aintzat, probabilitate handiago esleituz feromona gehiago duen balioari; behin lehenengo posizioko balioa finkaturik, bigarren posiziora pasatuko da eta, era antzerakoan, balio bat esleituko dio. Prozesua soluzio osoa sortu arte errepikatuko da.

Beraz, gure adibidean feromona eredu matrize sinple baten bidez implementa daiteke non zutabe bakoitzean bektorearen posizio bat izango dugun; matrizeak bi errenkada bakarrik izango ditu, posizio bakoitzeko 0 eta 1 balioek duten feromona kopurua gordetzeko. Feromona eredu mota hau **metaheuR** paketearen implementaturik dago, **VectorPheromone** klasean. Ereduen erabilera argitzeko *Maximum Dominating Set* (MDS) problema erabiliko dugu. Laburki, 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; MDS problematik, grafo bat emanik, kardinalitate minimoko menderatze-multzoa topatzean datza. 9 irudian problema honetarako hiru soluzio ikus daitezke.

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

Feromona eredu sortzeko matrizea hasieratu behar dugu. Ohikoena balio finko batekin hasieratzea da. Era honetan osagai guztiak balio berdina dute eta, beraz, guztien probabilitatea berdina izango da. Gogoratu gure adibidean matrizeak bi errenkada izan behar dituela, soluzioak bektore bitarrak baitira.



```
> init.trail <- matrix (rep(1 , 2*n) , ncol = n)
> evaporation <- 0.9
> pheromones <- vectorPheromone(binary = TRUE , initial.trail = init.trail ,
+                               evaporation.factor = evaporation)
> 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
```

Goiko kodean ikus daitekeen bezala, badago beste parametro bat finkatu behar dena, `evaporation.factor`. Parametro horretan pasatzen den balioa lurrunketa faseetan erabiltzen da, matrizean dauden balio guztiak txikiagotzeko; baporazioa `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 eraikitzeke erabiltzen dute. Soluzioak `build.solution` funtzioaren bitartez egiten da.

```
> build.solution(pheromones , 1)

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

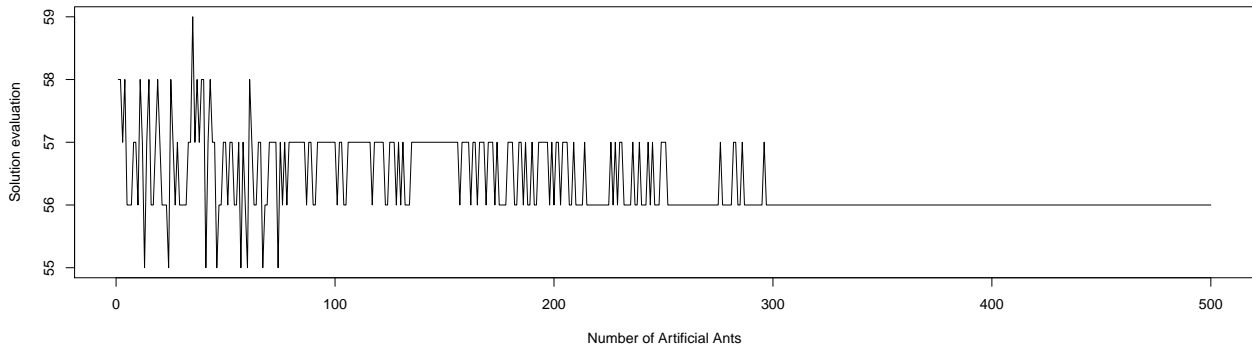
Inurriek ingurunetik ibiltzen diren heinean feromona uzten dute eta, era berean, inurri artifizialek soluzioak eraikitzen dutenean feromona kopurua handitzen dute; ereduaren eguneraketa hau `update.trail` funtzioa erabiliz egiten da.

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

## [1] 4

> update.trail(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
```



Irudia 10: ACO sinplearen eboluzioa MDS problema batean.

Ikus daitekeen bezala, zutabe bakoitzean errenkada bati soluzioaren helburu funtzioaren balioa gehitu zaio, hori baita inurriek utzitako lorratza. Elementu guzti hauek batzen baditugu, ACO sinple bat sor dezakegu. Lehenik, problema berri bat sortuko dugu.

```
> n <- 100
> rnd.graph <- aging.ba.game (n , 0.5 , 0 , 3 , directed = FALSE)
> mdsp <- mds.problem(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 «bidetik» feromona utziko du. Inurri bakoitza simulatu ondoren feromona «lurrundu» egiten dugu.

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

10 irudiak algoritmoaren eboluzioa erakusten du. Irudian ikus daiteke hasieran oso bariabildade handia dagoela sortutako soluzioen helburu funtzioaren balioan. Alabaina, inurri kopurua handitzen den heinean bariabildadea murrizten da eta, amaieran, prozedura soluzio bakar batera konbergitzen du. Edonola ere, soluzio hori ez da hasierakoak baino hobea. Kontua da, nahiz eta naturan horrela izan, optimizazioaren ikuspegitik inurri guztiek feromona eredua eguneratzea ez dela hurbilketarik onena; algoritmoa hobetzeko hautespen prozesu bat sartzea komenigarria da.

Ikusi dugun algoritmo sinpleak ez du ondo funtzionatzeko, inurri guztiek feromona eguneratzen dutelako. Hori dela eta, beste estrategia erabili ohi da. Inurriak banan-banan simulatu ordez, inurri-kolonia tamaina bat definitzen da eta iterazio bakoitzean inurritegiko inurri guztiek soluzio bat sortzen dute. Gero, bi estrategia ezberdin erabil daitezke:

- Iterazioko soluziorik onena erabili - Inurriek uneko iterazioan sortutako soluzioen artean onena aukeratzen da eta soluzio hori bakarrik erabiltzen da feromona eredua eguneratzeko. Kasu honetan helburu

## Inurri-kolonien algoritmoa

---

```

1 input: build_solution, evaporate, add_pheromone , initialize_matrix eta stop_criterion oper-
  adoreak
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

```

---

### Algoritmoa 2.1: Inurri-kolonien algoritmoaren sasikodea

funtzioaren arabera egitea ez da beharrezkoa, bakarrik soluziorik onena erabiltzen baita eguneraketan. Hori dela eta, ohikoa da balio finko bat erabiltzea eguneraketan.

- Bilaketan topatutako soluziorik onena - Hainbat kasutan bilaketa areagotzea interesatuko zaigu. Kasu horietan, feromonaren eguneraketa bilaketan zehar topatu den soluziorik onena erabiliz egin daiteke. Aurreko puntuan bezala, eguneraketak ez dauka zergatik izan helburu funtzioaren balioarekiko proportzionala.

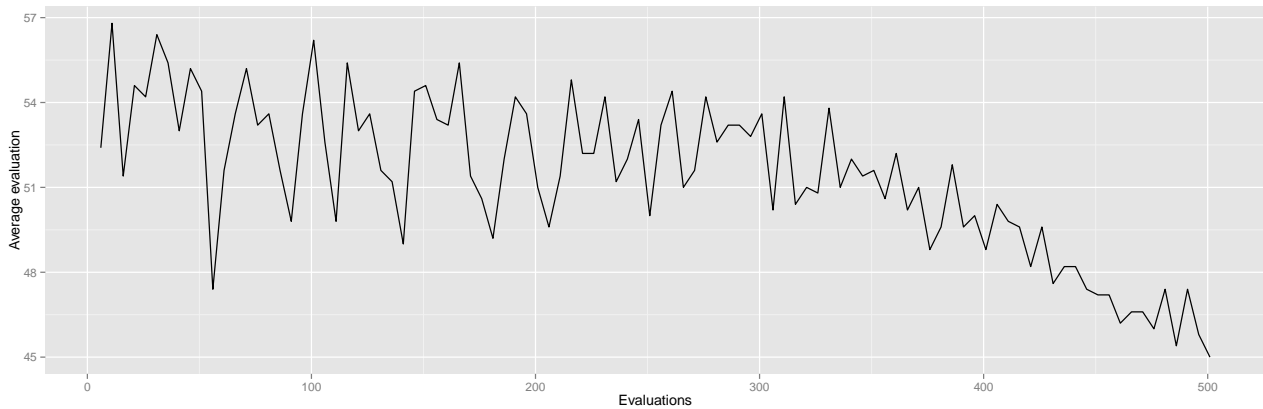
Aldaketa honekin oinarritzko ACO algoritmoaren sasikodea defini dezakegu (ikusi 2.1 algoritmoa). **basic.aco** funtzioak Oinarritzko ACO-a implementatzen du eta, hortaz, goiko problema ebazteko erabil dezakegu. Ohiko parametroaz gain, argumentu hauek zehaztu behar dira.

- **nants** - Zenbat inurri artifizial izango ditugu gure kolonian
- **pheromones** - Feromona eredua.
- **update.sol** - Nola eguneratuko dugu feromona eredua. Hiru aukera daude: **'best.it'**, iterazio bakoitzean sortutako soluziorik onena; **'best.all'**, bilaketan zehar lortutako soluziorik onena edo **'all'**, sortutako soluzio guztiak.
- **update.value** - Balio bat finkatzen bada, eguneraketa guztietan balio hori gehitzen da; NULL bada, helburu funtzioa erabiltzen da. Kontutan hartu problema batzuetan helburu funtzioak negatiboak direla, baina feromona eredu batzuetan balio positiboak eta negatiboak ezin dira nahastu, bestela arazoak egon daitezke probabilitateak kalkulatzeko. Hori dela eta, helburu funtzioaren zeinua kontutan hartu behar da feromona eredua hasieratzeko.

```

> args <- list()
> args$evaluate      <- mdsp$evaluate
> args$nants         <- 5
> init.value         <- 1
> initial.trail <- matrix(rep(init.value , 2*n ) , nrow=2)
> evapor <- 0.9
> pher <- vectorPheromone(binary = TRUE , initial.trail = initial.trail ,

```



Irudia 11: Oinarritzko ACO algoritmoaren eboluzioa MDS problema batean.

```
+                               evaporation.factor = evapor)
> args$pheromones               <- pher
> args$update.sol               <- 'best.it'
> args$update.value             <- init.value / 10
> args$non.valid               <- 'correct'
> args$valid                   <- mdsp$is.valid
> args$correct                 <- mdsp$correct
>
> args$resources                <- cresource(iterations = 100)
> args$verbose                 <- FALSE
>
> results.aco <- do.call(basic.aco , args)
> plot.progress(results.aco) + labs(y="Average evaluation")
```

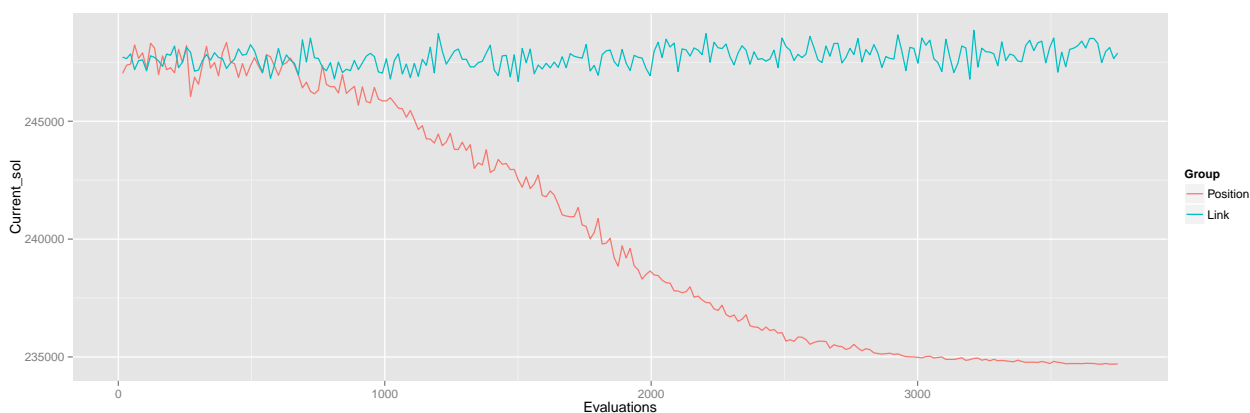
11 irudiak oinarritzko ACO algoritmoaren progresioa jasotzen du. Algoritmo honek soluzio kopuru berdina sortu du, baina aurrekoa ez bezala, iterazioz iterazio soluzioa hobetuz doa. Grafikoan batazbesteko *fitness*-ak bariabilidade handia duela ikus daiteke. Hau da oso kolonia txikia erabili dugulako –5 inurri bakarrik–. Balio hori handitzen badugu, progresioa ez da hain zatatsua izango –eta, ziurrenik, emaitzak hobeak izango dira–, baina ebaluazio gehiago beharko ditugu.

ACO algoritmoen mamia soluzio eraikuntza da eta, hortaz, soluzioen osagaien definizioa oso garrantzitsua da; osagaiek problemaren natura kontutan hartzen ez badute, feromona ereduak ez du soluzioen informazioa behar den bezala jasoko. Hau agerian gelditzen da jarraian dagoen adibidean.

Demagun LOP problema bat ebazteko ACO algoritmo bat erabili nahi dugula. Problema honetarako soluzioak permutazioen bidez kodetzen ditugunez, adierazpide honekin diharduen feromona eredu bat behar dugu. Soluzioen eraikuntzan permutazioak sortzeko behar diren aldaketak eginez gero, sektoreekin erabilitako eredu permutazioekin ere erabil dezakegu. Hau da, matrize karratu bat gordeko dugu non posizio bakoitzeko, balio bakoitzari dagokion feromona kopurua gordeko dugu. Gero, soluzioak osatzerakoan, urrats bakoitzean aukeraturik gabe dauden balioetatik bat aukertuko dugu, aukera guztien arteko feromona kopurua kontutan hartuz. Eredu honek UMDA-n ikusi genuen oso matrize antzerakoa erabiltzen du.

Dena dela, hau ez da aukera bakarra. TSP problemari ikusi genuen permutazioek grafo osoko ziklo Hamiltoniarrak adierazten dituztela. Hau da,  $n$  nodoko grafo oso bat badugu, edozein permutazio  $n$  nodoak behin eta bakarrik behi bisitatzen dituen ibilbide bat adierazten du. Beraz, permutazioak osatzeko nodoak lotzen dituzten ertzak erabil ditzakegu.

Idea hau erabiliz beste feromona eredu bat planteatu dezakegu. Eredu honek ere matrize karratu bat erabiliko du, baina matrizearen interpretazioa –eta, hortaz, soluzio osaketa– ezbedina da. Kasu honetan, matrizeak



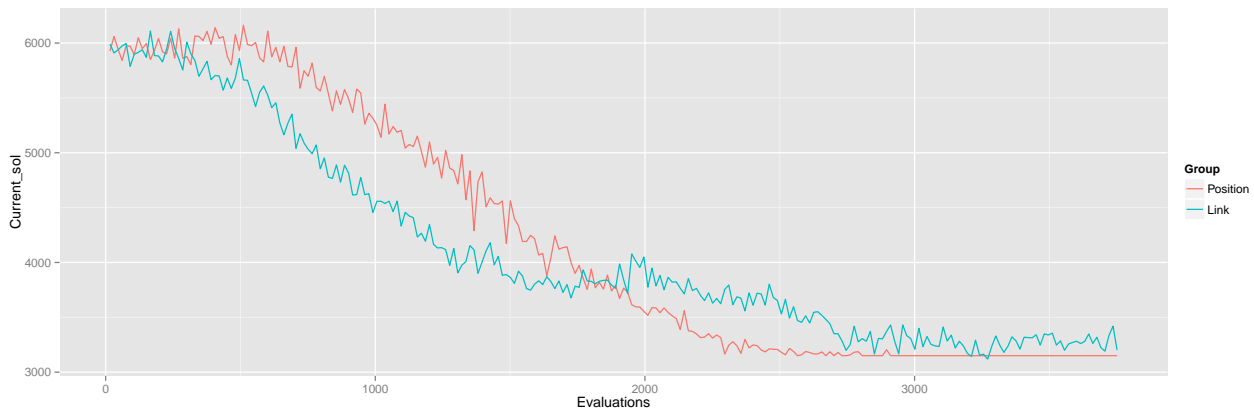
Irudia 12: Oinarrizko ACO algoritmoaren eboluzioa LOP problema batean.

grafoaren ertzak adierazten ditu, alegia,  $(i, j)$  posizioan  $i$  nodotik  $j$  nodora joateari lotutako feromona kopurua izango dugu. Adibidez, 3421 permutazioa badugu, soluzio honi dagozkion matrizeko posizioak dira  $(3, 4)$ ;  $(4, 2)$  eta  $(2, 1)$  –problemaren arabera,  $(1, 3)$  posizioa ere erabiltzea komenigarria izan daiteke–.

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

```
> n <- 100
> rnd.mat <- matrix(round(runif(n^2)*100) , n)
> lop <- lop.problem(matrix = rnd.mat)
>
> args <- list()
> args$evaluate <- lop$evaluate
> args$nants <- 15
> init.value <- 1
> initial.trail <- matrix(rep(init.value , n^2 ) , n)
> evapor <- 0.9
> pher <- permuLinkPheromone(initial.trail = initial.trail ,
+                             evaporation.factor = evapor)
> args$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(basic.aco , args)
>
> args$pheromones <- permuPosPheromone(initial.trail , evapor)
> aco.pos <- do.call(basic.aco , args)
>
> plot.progress (list("Position" = aco.pos , "Link" = aco.links))
```

12 irudiak experimentuaren emaitza erakusten du. Grafikan argi ikus daiteke noden arteko lotruak osagaitzat hartzen direnean, bilaketak ez du aurrera egiten. Osagaiak posizioak direnean, berriz, iterazioz iterazio soluzioa hobetzen da. Honen arrazoia sinplea da: LOP problematik posizio absolutuak dira garrantzitsuena, eta ez zein elementu dagoen zeinen ondoan.



Irudia 13: Oinarritzko ACO algoritmoaren eboluzioa TSP problema batean.

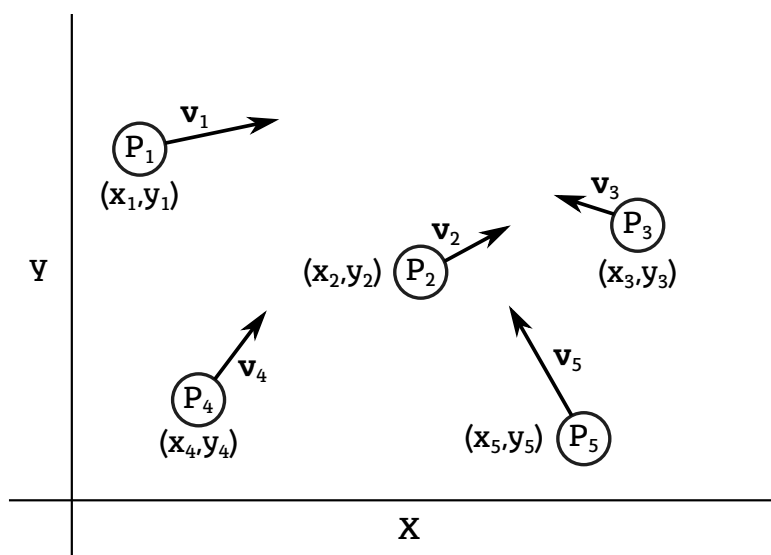
TSP probleman justu kontrakoa gertatzen da, hau da, informazio garrantzitsuena da zein hiri dagoen zeinen ondoan. Hori dela eta, printzipioz, ertzetan oinarritzen den feromona ereduak soluzio hobeak lortuko ditu. Jarraitu experimentu hori egingo dugu; emaitzak 13 irudian daude.

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

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

> n <- ncol(cost.matrix)
> tsp <- tsp.problem(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(basic.aco , args)
>
> args$pheromones <- permuPosPheromone(initial.trail , evapor)
> aco.pos <- do.call(basic.aco , args)
>
> plot.progress (list("Position" = aco.pos , "Link" = aco.links))
```

Orain arte ikusi ditugu adibide guztietan feromonak bakarrik erabiltzen dira soluzioak eraikitzerakoan. Oinarritzko algoritmoan horrela izan arren, problema konkretu bat ebatzi behar denean, posible bada, infor-



Irudia 14: PSO algoritmoak erabiltzen dituen partikulen adibidea. Partikula bakoitzak bere kokapena  $(x_i, y_i)$  eta bere abiadura  $(\mathbf{v}_i)$  du

mazio heuristikoa sartu ohi da. Adibide gisa, TSPrako algoritmo eraikitzaile tipikoan hurrengo hiri hautatzeko hirien arteko distantzia erabiltzen da. Beraz, goiko adibidean hiri batetik bestera joateari dagokion feromona kopurua soilik erabili beharrean, bi hiri horien arteko distantzia ere kontutan har dezakegu.

## 2.2 Particle Swarm Optimization

Intsektu sozialen portaera *swarm* adimenaren adibide tipikoak dira, baina ez dira bakarrak; animali handiagotan ere inspirazioa bila daiteke. Esate baterako, txori-saldotan ehunaka indibiduo batera mugitzen dira beraien arteak talka egin gabe. Multzo horietan ez dago indibiduo bat taldea kontrolatzen duena, txori bakoitzak bere ingurune txorien portaera aztertzen du, berea egokitzeke. Era horretan, arau simple batzuk (txori batetik gertuegi banago, urrundu egiten naiz, adibidez) besterik ez dira behar sistema osoa antolatzeke.

*Particle Swarm Optimization* (PSO) algoritmoaren inspirazioa animali-talde hauen mugimenduak dira. Gure sisteman bilaketa espazioan mugitzen diren zenbait partikula izango ditugu; partikula bakoitzak kokapen eta abiadura konkretuak izango dituzte. Partikularen kokapenak berak partikulari dagokion soluzioa izango da; abiadurak, hurrengo iterazioan nora mugituko den esango digu. Ikus dezagun adibide simple bat. 14 irudian bi dimentsioko bilaketa espazio bat adierazten da. Bertan, bost partikula ditugu; bakoitzak problemarako soluzio bat adierazten du. Adibidez,  $p_1$  partikulak  $X = x_1; Y = y_1$  soluzioa adierazten du.

PSO algoritmoan partikulek bilaketa espazioa aztertzen dute, posizio batetik bestera mugituz. Beraz, iterazio bakoitzean partikula guztien kokapena eguneratzen da, beraien abiadura erabiliz. Partikulen abiadurak finko mantentzen baditugu, partikula guztiak infinitura joango dira. Hori ez gertatzeko, iterazio bakoitzean abiadura ere eguneratu behar da; eguneraketa honetan datza, hain zuzen, algoritmoaren gakoa. PSO *swarm intelligence*-ko algoritmo bat denez, indibiduen arteko (partikulen arteko, kasu honetan) komunikazioa ezinbestekoa da. Komunikazio hau abiadura eguneratze-prozesuan erabiltzen da, partikula bakoitzak bere abiadura eguneratzeko ingurunean dauden partikulak aintzat hartuko baititu.

Beraz, algoritmoa aplikatzeko ingurune kontzeptua definitu behar dugu. PSO-n, ingurune kontzeptua ez da bilaketa lokalean erabiltzen den berdina, partikula bakoitzaren ingurunea aurrez aurretik ezarritakoa baita; ez du partikularen kokapenarekin zerikusirik, alegia. Partikula bakoitzaren ingurunea grafo baten bidez adieraz daiteke, non bi partikula konektatuta dauden baldin eta bakarrik baldin bata bestearen ingurunean badaude. Lehenengo hurbilketa grafo osoa erabiltzea da, hots, edozein partikularen ingurunean beste gainontzeko partikula guztiak egongo dira; grafo osoa erabili beharrean, beste zenbait topologia ere erabili daitezke (eratzunak, izarrak, toroideak, etab.).



Partikula baten abiadura eguneratzeko bi elementu erabiltzen dira. Alde batetik, partikula horrek bisitatu duen soluziorik onena, hau da, bere «arrakasta pertsonala». Soluzio honi ingelesez *personal best* deritzo, eta  $\mathbf{p}_i$  sinboloaren bidez adieraziko dugu. Bestaldetik, ingurunean dauden partikulen arrakasta ere kontutan hartzen da, partikularene ingurunean dauden beste partikulek lortu duten soluziorik onena, alegia. Soluzio honi ingelesez *global best*<sup>2</sup> deritzo, eta  $\mathbf{p}_g$  sinboloaren bidez adieraziko dugu.

Hau dena kontutan hartuta, *i.* partikulak *t* iterazioan erabiliko duen abiadura aurreko iterazioan erabilitakoa ondoko ekuazioaren bidez kalkulatu dugu:

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

Ekuazioan bi konstante ditugu,  $C_1$  eta  $C_2$ ; balio hauek partikulak eta ingurunean topatutako soluzioen eragina kontrolatzeko erabiltze dira. Konstante hauek gain, bi ausazko aldagai ditugu,  $\rho_1$  eta  $\rho_2$ . Bi aldagai hauek ausazko balioak hartzen dituzte  $[0, 1]$  tartean.

Lortutako abiadura bektore bat da. Arazoak saihesteko, bektore horren modulua mugatuta dago, aurrez aurretik abiadura maximoa ezarritik. Kalkulatutako abiaduraren modulua handiagoa bada, balio maximora eramaten da.

Behin uneko iterazioaren abiadura kalkulatu, abiadura partikularene kokapena eguneratzeko erabiltzen da:

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

Iterazio bakoitzean lortutako soluzioak ebaluatu eta, beharrezkoa bada, partikulen *personal* eta *global best* eguneratu behar dira. Pasu guzti hauek 2.2 algoritmoan biltzen dira.

## Bibliografia

- [1] G. Beni. The concept of cellular robotic system. In *Proceedings of the IEEE International Symposium on Intelligent Control*, pages 57–62, 1988.
- [2] C. Blum and D. Merkle. *Swarm Intelligence: Introduction and Applications*. Springer-Verlag, 2008.
- [3] M. Dorigo, V. Maniezzo, and A. Coloni. Ant system: Optimization by a colony of cooperating agents. *Trans. Sys. Man Cyber. Part B*, 26(1):29–41, 1996.
- [4] Marco Dorigo. *Optimization, Learning and Natural Algorithms*. PhD thesis, Politecnico di Milano, 1992.
- [5] T.D. Gwiazda. *Genetic algorithms reference Volume I Crossover for single-objective numerical optimization problems*. Number v. 1. Lightning Source, 2006.
- [6] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, USA, 1975.
- [7] P. Larrañaga and J. A. Lozano. *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*. Kluwer Academic Publishers, 2002.
- [8] Jose A. Lozano, Pedro Larrañaga, Iñaki Inza, and Endika Bengoetxea. *Towards a New Evolutionary Computation: Advances on Estimation of Distribution Algorithms (Studies in Fuzziness and Soft Computing)*. Springer-Verlag New York, Inc., 2006.
- [9] El-Ghazali Talbi. *Metaheuristics: From Design to Implementation*. Wiley Publishing, 2009.

<sup>2</sup>Ingurunea definitzeko grafo osoa erabiltzen ez bada, partikula baten ingurunean lortutako soluziorik onenari *global best* baino *local best* esaten zaio



### 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

```

---

Algoritmoa 2.2: *Particle Swarm Optimization* algoritmoaren sasikodea