BHLN: Populazioan oinarritutako algoritmoak

Borja Calvo, Usue Mori

Laburpena

Aurreko kapituluan 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 oinarrizen 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 [3] eta EDAk [4, 5] aurkezten dira. Bigarren zatian *swarm intelligence* [1] 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.

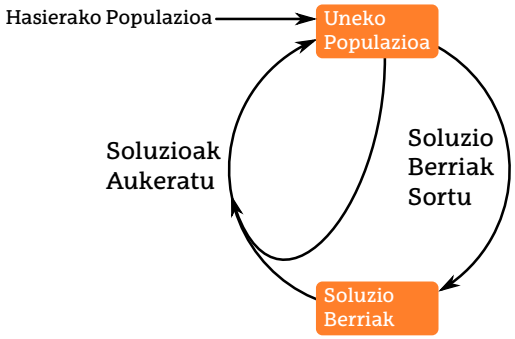
Eboluzioaren teoriak dioenez, belaunalditik belaunaldira zenbait mekanismoen bidez –mutazioak, esate baterako– aldaketak sortzen dira. Aldaketa batzuei esker indibiduoak hobeto egokitzen dira beraien inguruneari eta, ondorioz, bizirik mantentzeko eta, batez ere, ugaltzeko probabilitateak handitzen dira. 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 metahuristikoak diseinatzeko; gaur egun konputazio zientziaren arlo oso bat da konputazio ebolutiboa. Atal honetan bi algoritmo mota aztertuko ditugu, algoritmo genetiko klasikoak [3] eta EDAk (*Estimation of Distribution Algorithms*) [4, 5].

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 behark dira; sorkuntza prozesua aukeratu ditugu soluzioak hartzen ditu abia-puntutzat.

Ikasturtea: 2014/2015
Irakaslea: Borja Calvo, Usue Mori
Telefonoa: 943 01 50 13
E-posta: borja.calvo@ehu.es

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



Irudia 1: Algoritmo ebolutiboen eskema orokorra

Diferentziak diferentzia, algoritmo ebolutiboen 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, zanbait 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– dibertstiatea maximizatzeko. 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.

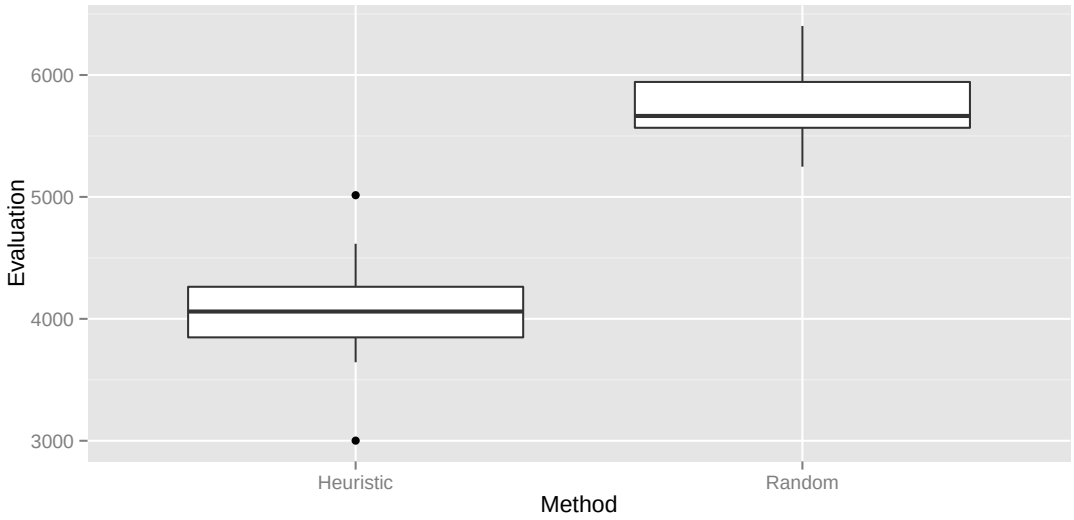
```
> rnd.sol <- function(cl.size = 5){
+   tsp.greedy(cmatrix = cost.matrix , cl.size = cl.size)
+ }
```

`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` hirietatik –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 adieraztен 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 hobeak 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.

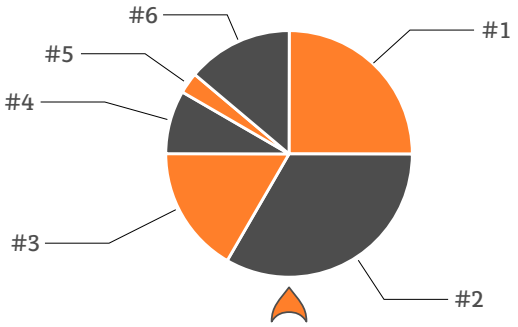
1.1.2 Hautespena

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 dituenez, hautespen honi «elitista» deritzo. Soluzio onak aukeratzea garrantzitsua da baina, dibertsitatea mantentzearren, tarteka soluzio txarrak ere sartzea komenigarria izan daiteke. Hau zuzenea 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, indibiduoen 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	Ebaluzaioa
#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. indibiduoa 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.

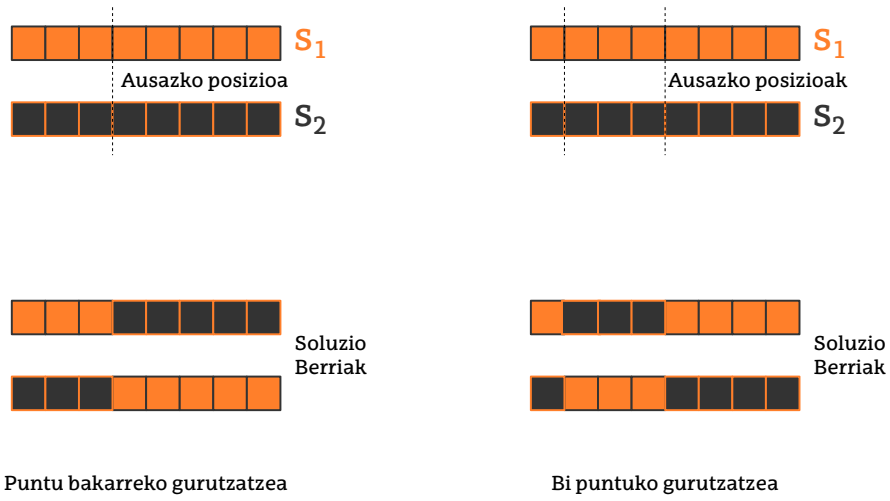
Gelditzeko irizpideak dinamikoak ere izan daitezke, eboluzioaren prozesuari erreparatzen badiogu. Balunaldiz balaundi populazioan dauden soluzioak geroz eta hobeak dira eta, aldi berean, populazioaren dibertsitatea murrizten da. Beste era batean esanda, populazioko soluzioek soluzio bakar batera konbergitzeko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

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

1.1.4 Algoritmo Genetikoak

Atal honetan algoritmo genetikotan [3] 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
- Indibiduoen 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. Ohikoena 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, «gurutze-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 kopiatuz sortuko dugu. Era berean, bigarren soluzio berria s_2-tik lehenengo elementuak eta besteak s_1-etik kopiatuz sortuko dugu. 4 irudiaren ezkerraldean adibide bat ikus daiteke. Irudiak ideia nola orokor daitekeen 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 performantzian 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[2]. Esate baterako, bektore errealak baldin baditugu, bi bektore harturik era ezberdinetan konbina daitezke; adibidez, bataz bestekoa kalkulatuz. 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 gurutatzeko operadore bereziak behar ditugu. Aukera asko izan arren [6], 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 sinple bat hauxe 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. Emaitza, 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 [1].

```
> 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 algoritmoko perturbazioaren antzerakoa da; izan ere, operadore berdinak 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 trukatuko ditugun –.

[1] 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 duden 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 genetikoen 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 behako luke. Bestaldetik, 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 sortzeak 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.

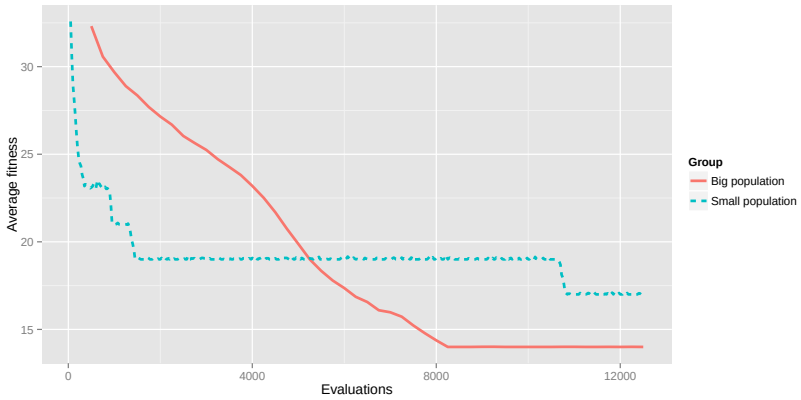
Mutazio operadorea era probabilistikoan aplikatzen da; hau da, ez zaie soluzio guztiei aplikatzen. Hortaz, mutazioari lotutako bi parametro izango ditugu: mutazio probabilitatea eta mutazioaren magnitudea.

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

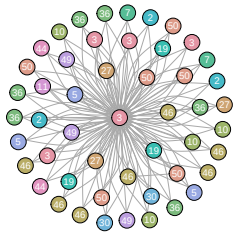
```
> library(igraph)
> n <- 50
> rnd.graph <- aging.ba.game(n = n , pa.exp = 2 ,
+                            aging.exp = 0, m = 3 , directed = F)
> gcp <- graph.coloring.problem (graph = rnd.graph)
```

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 poulazio tamaina ezberdin erabiliz. Ezkerrean, populazio tamaina handiarekin lortutako soluzioa ikus daiteke.

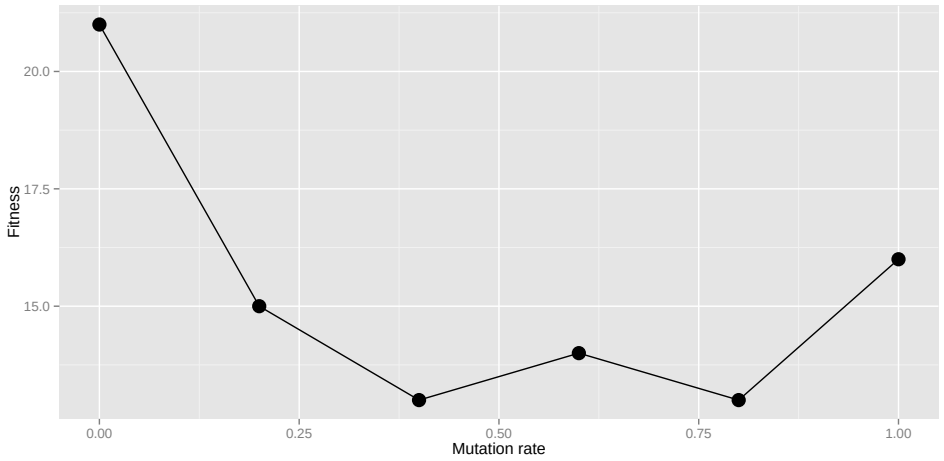
```
> levels <- paste("C" , 1:n , sep = "")
> rnd.sol <- function(x){
+   sol <- factor(paste("C" , sample(1:n , size = n , replace = TRUE) ,
+                       sep = "") , levels = levels)
+   return(gcp$correct(sol))
+ }
> pop.small <- lapply(1:n.pop.small , FUN = rnd.sol)
> pop.big   <- lapply(1:n.pop.big , 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 mutatuko 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 genetikoa 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 irduian 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 bataz besteko *fitness*a 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, popluazio 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 berrriz, arazoa justo 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 probabilistikoak erabiltzea proposatu zuten, populazioaren «esentzia» kapturatzeko helburuarekin. Hauxe da, EDA – *Estimation of Distribution Algorithms* – algoritmotan erabiltzen den ideia.

Algoritmo genetikoen eta EDA motako algoritmoen artean dagoen diferentzia bakarra indibiduo berriak sortzean dago. Gurutzatzea eta mutazioa erabili beharrean, uneko populazioa eredu probabilistiko 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 sinple 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 aukeratuko ditugu probabilitate hauek kontutan hartuz.

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

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

Orain, `univariateMarginals` funtzioa erabiliz probabilitate marginalak kalkulatuko ditugu:

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

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

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

> model@prob.table

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

Sortutako soluzioek 10 elementu dituzte 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 probabilistikoa soluzio berriak sortzeko erabil daiteke, posizioz posizio balioak dagokion marginala laginduz; laginketa `simulate` funtzioaren bitartez egiten da.

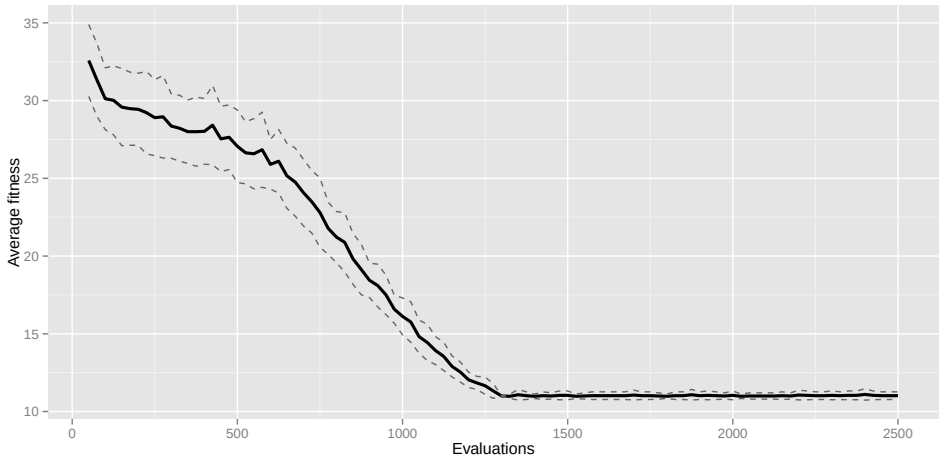
```
> simulate(model , nsim = 2)

## [[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 bataz-besteko *fitness*a adierazten du; marra etenek, berriz, desbiderazio estandarra adierazten dute. Ikus daiteke populazioak konbergitzen duen heinean soluzioen *fitness*aren bariabilidadea txikiagotzen dela.

Marjinalak zuzenean edozein bektoreekin erabil daitezke; alabaina, balio errealak 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 eredua 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.umda <- univariateMarginals(perm.pop)
> simulate(perm.umda)

## [[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.umda@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 eraukteko 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
```

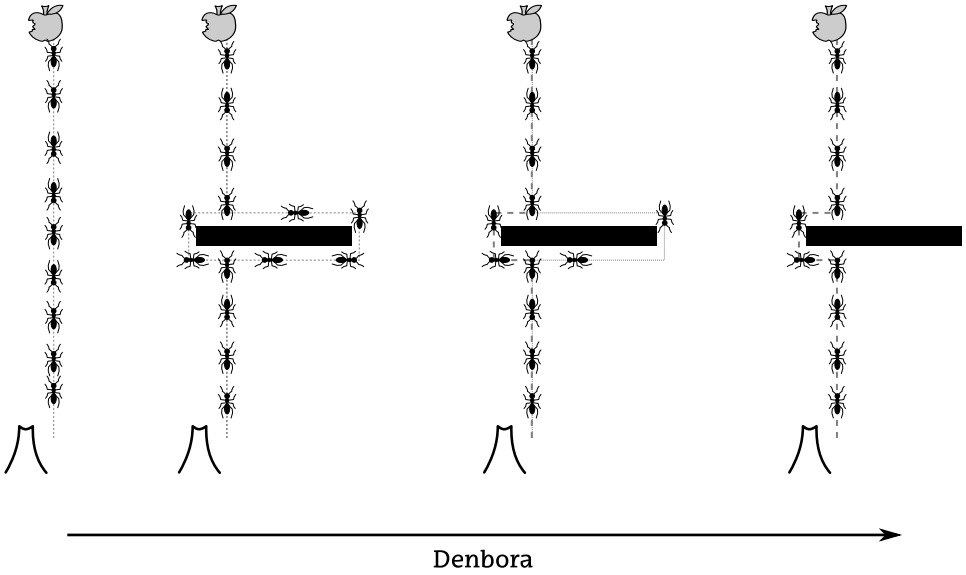
Prozesu honi esker probabilitate marjinalak erabil daitezke permutazioak sortzeko, baina arazo bat dauka; eredua lagintzen dugun bakoitzean probabilitateak aldatzen ditugu eta, hortaz, lagintzen duguna ez da populaziotik ikasi duguna. Beste era batean esanda, populaziotik ateratako «esentzia» gal genezake.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

2 Swarm Intelligence

Eboluzioaz gain, badago populazioetan oinarritzen diren algoritmoen artean beste intuizio edo inspirazio nagusia arrakasta handia lortu duena, *swarm intelligence* deritzona. Naturan badaude hainbat espezie zeinen indibiduoen portaera, indibidualki, oso sinplea den baina, taldeka daudenean, ataza zailak burutzeko gai diren. Intsektuak dira, zalantzarik gabe, adibiderik ezagunena. Hauen artean inurriak, erleak eta termitak dira adibide aipagarrienak.

2.1 Inurri Kolonien Algoritmoak

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 berezi bati esker: feromona.



Irudia 8: Feromonaren erabileraren adibidea. Hasierako egoeran biderik motzena feromonaren bidez markatuta dago. Bidea mozte dugunean, inurriek eskumara edo ezkerrera joango dira, probabilitate berdinarekin feromonarik ez dagoelako. Eskumako bidea luzeagoa da eta, ondorioz, ezkerreko bidearekiko inurrien fluxua txikiagoa da. Denbora igaro ahala eskumako lorratza 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 lorratza guztiz galduko da; koloniak bide motzena topatu du

Inurri bakoitza mugitzen denean feromona-lorratz bat uzten du eta, atzetik datozen inurriak lorratz hori jarraitzeko gai dira. Geroz eta feromona gehiago, orduan eta probabilitate handiagoa datozen inurriak utzitako lorratza jarraitzeko.

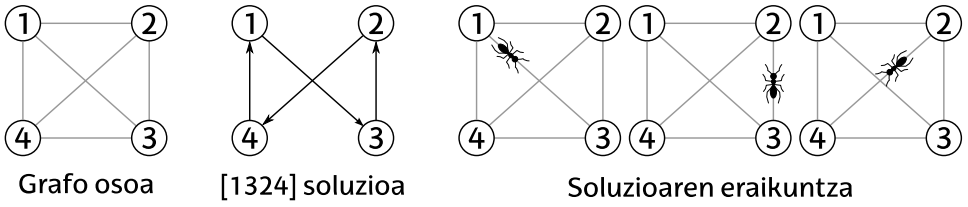
Topatutako elikagai-iturriaren kalitatearen arabera, utzitako feromona kopurua ezberdina da; geroz eta kalitate handiagoa, orduan eta feromona gehiago. Kontutan hartuz feromona lurrunkorra dela, hau da, denborarekin baporatzen dela, koloniak komunikazio sistemari esker biderik motzena topatzeko gai dira.

Mekanismoaren erabilera 8 irudian ikus daiteke. Hasieran bide motzena feromona lorratzaren bidez markatuta dute inurriek. Bidea mozten dugunean, 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 ezkerretik inurri gehiago igaroko dira, ezkerreko bidean feromona gehiago utziz. Ondorioz, datozen inurriak ezkerretik joateko joera handiago izango dute, bide hori indartuz. Eskumako bidean lorratza apurka-apurka baporatuko da eta, denbora nahiko igarotzen bada, zeharo galduko da.

Intuizio hau optimizazio problemak ebazteko erabil daiteke. Ikus dezagun adibide bat.

2.2 Adibidea: *Linear ordering problem*

Linear Ordering Problem, LOP, optimizazio problema klasiko bat da. Matrize karratu bat emanda, honen errenkadak eta zutabeak ordenatu behar dira, aldi berean, diagonaletik gora dauden elementuen batura minimizatzeko. Demagun ondoko matrizea daukagula:



Irudia 9: Soluzioen eraikuntza. Grafo osotik abiatuta, edozein permutazioa ziklo Hamiltoniar baten bidez adieraz daiteke. Inurrien portaera soluzioak eraikitzeko erabil daiteke, pausu bakoitzean inurria uneko erpinetik zein erpinera mugituko den erabakiz.

	Z_1	Z_2	Z_3	Z_4
E_1	9	4	2	3
E_2	2	4	6	1
E_3	1	3	6	3
E_4	7	4	4	8

Edozein ordenazio emanda, matrizeen errenkadak eta zutabeak ordena daitezke, matrize berri bat sortuz. Esate baterako, 2. eta 3. zutabe/errenkada trukatzen baditugu, ondoko matrizea lortuko dugu:

	Z_1	Z_3	Z_2	Z_4
E_1	9	2	4	3
E_3	1	6	3	3
E_2	2	6	4	1
E_4	7	4	4	8

Ordenazio hau LOP-rako soluzio bat da, [1324] permutazioaren bidez adieraziko duguna. Edozein permutazio problemarako soluzio bat izango da, zeinen ebaluazioa jarraian nabarmendua dauden elementuen batura den.

	Z_1	Z_3	Z_2	Z_4
E_1	9	**2**	**4**	**3**
E_3	1	6	**3**	**3**
E_2	2	6	4	**1**
E_4	7	4	4	8

hau da, kasu honetan 16.

Beraz, LOP-rako soluzioak permutazioak dira. n nodoko grafo osoa hartzen badugu, non nodoak zenbatuta dauden, edozein ziklo Hamiltoniarra[2] permutazio bat da. 9 irudian adibideko permutazioari dagokion zikloa ikus daiteke.

Grafoen bidezko permutazioen adierazpidea erabiliz, inurri «artifizialak» erabil ditzakegu LOP-rako soluzioak eraikitzeko. Demagun 1. nodoan inurri bat kokatzen dugula. Inurriak ziklo bat osatzeko zein nodora mugituko den erabaki beharko du; hau da, 1. nodotik 2., 3. edo 4. nodora joango den erabaki beharko du. Demagun 3. nodora joaten dela, hurrengo urratsean 2. eta 4. nodoen artean bat aukeratu beharko du inurriak. 2. nodoa aukeratzen bada, zikloa osatzeko 4. nodora eta, handik 1. nodora joan beharko da. Prozesu hau jarraituz adibideko permutazioa eraiki dezakegu; 9 irudiak prozesua erakusten du.

Inurri artifizialen bidez soluzioak eraiki daitezke baina, nola erabaki uneko nodotik nora abiatu?. Galdera honi erantzuteko naturan gertatzen denari erreparatuko diogu. Egiazko inurriek bidea ausaz aukeratzen dute, baina bide bat edo bestea aukeratzeko probabilitatea feromona kopuruarekiko proportzionala da. Era berean,

[2]Gogoratu ziklo bat Hamiltoniarra dela erpin guztietatik behin eta bakarrik behin pasatzen bada

grafoaren ertz bakoitzari feromona kopuru bat esleitzen badiogu, gure inurri artifizialak bidea erabakitzeko feromona erabili ahal izango du.

Hortaz, uneoro ertz bakoitzean zenbat feromona dagoen gorde beharko dugu F matrizean, non $f_{i,j}$ i nodotik j nodora joateari dagokion feromona kopurua den[3]. Feromona kopurua eguneratu barik, inurri-koloniak ez luke bide motzena topatuko. Era berean, gure problema ebazteko feromona matrizea eguneratu beharko dugu. Naturan bezala, feromona kopurua eguneratzeko bi pausu izango ditugu: lurrunketa eta feromona lagatzea. Hasieran matrizearen posizio guztien feromona kopurua berdina izango da; hortik aurrera inurriek erabilitako bideak kontutan hartu beharko ditugu feromona kopurua eguneratzeko.

Lurrunketa egiteko F matrize posizio guztiak txikiagotuko ditugu, $f_{i,j} = \alpha f_{i,j}$ eginez, non $0 < \alpha < 1$ izango den. Lagatze prozesuari dagokionez, oso era sinplean egin daiteke, soluzio bakoitza sortzean erabilitako ertzei balio finko bat gehituz. Hau da, inurri bat soluzioa eraikitzeko 2. nodotik 3. nodora joaten bada, $f_{2,3}$ eguneratuko dugu balio finko bat d gehituz. 2.3 algoritmoan prozeduraren sasikodea ikus daiteke. Algoritmoa aplikatzeko zenbait funtzio beharko ditugu:

- `initialize_matrix` - Funtzio honek feromona matrizea hasieratzen du, posizio guztiei – diagonala izan ezik – balio finko bat esleituz
- `build_solution(pheromone_matrix)` - `pheromone_matrix` feromona matrizea erabiliz, funtzio honek pausuz pausu soluzio bat eraikitzen du. Lehen pausua nodo bat ausaz aukeratzea da. Gero, urrats bakoitzean uneko nodotik zein nodoetara joan gaitezkeen jakin behar dugu – bisitatu barik daudenak, alegia –. Aukera guztietatik bat ausaz aukeratuko dugu; aukera bakoitzari dagokion probabilitatea feromona matrizean dauden balioak erabiliz kalkulatuko dugu.
- `evaporate(pheromone_matrix)` - Funtzio honek feromona matrizea hartzen du eta posizio bakoitza eguneratzen du, 0 eta 1 tartean dagoen balio bat biderkatuz.
- `add_pheromone (pheromone_matrix,solution)` - Funtzio honek, soluzio bat emanda, soluzio hori eraikitzeko jarraitutako bidean dauden ertz guztiei balio finko bat gehitzen die.

2.3 ACO algoritmoak diseinatzen

Aurreko atalean LOP nola ebatz daitekeen ikusi dugu. Orokorrean, optimizazio problema bat ACO motako algoritmo baten bidez ebatzi nahi badugu, bi elementu nagusi beharko ditugu: soluzioak eraikitzeko prozedura bat[4] eta feromona eredu bat. Algoritmo eraikitzaileetan pausu bakoitzean zenbait aukera izaten ditugu; ACO bat diseinatzeko feromona ereduak aukera bakoitzaren feromona kopurua mantenduko du.

Aurreko atalaren adibidean inurri guztiek era finkoan eguneratzen zuten feromona eredua; edonola era, aukera hau ez da bakarra. Ereduaren eguneraketa diseinatzean bi gauza hartu behar ditugu aintzat. Alde batetik, zein inurriak eguneratuko du eredua; bestetik, nola eredua nola eguneratu. Lehenengo puntuari dagokionez, hiru aukera ditugu:

- **Inurri guztiak** - Soluzio bat eraikitzen den bakoitzean, erabilitako elementuen feromona kopurua handitu.
- **Iterazioko soluziorik onena** - Behin koloniako inurri guztiek beraien soluzioak eraiki, guztietatik zein den onena identifikatu eta bakarrik soluzio hori eraikitzeko erabili diren elementuak eguneratu.
- **Algoritmoak topatutako soluziorik onena** - Bilaketa areagotu nahi badugu, iterazioko soluziorik onena erabili beharrean, aurreko iterazioetan eraiki den soluziorik onena erabil dezakegu.

Adibidean feromona lorratzak eguneratzean inurri guztien ekarpena berdina zen; alabaina, beste zenbait aukera ditugu:

[3] Kontutan hartu naturan ez bezala, bidearen noranzkoa garrantzitsua izan daitekeela; hau da, ez da berdina i-tik j-ra edo j-tik i-ra joatea

[4] Hau dela eta, ACO algoritmoak erraz diseina daitezke ebatzi nahi dugun problema ebazteko algoritmo eraikitzaile onak existitzen badira

Inurri-kolonien algoritmoa

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

Algoritmoa 2.1: Inurri-kolonien algoritmoaren sasikodea

- **Soluzioaren ebaluazioaren arabera -** Naturan inurriak utzitako lorratzaren intentsitatea janari iturriaren kalitatearen araberakoa da; era berean, gure algoritmoan soluzioaren ebaluazioa erabil dezakegu soluzio onenen ekarpena handiagoa izan dadin.
- **Inurrien ranking-aren arabera -** Ebaluazio funtzioaren magnitudea problemaren eta instantziaren araberakoa da[5]. Eskala arazo hauek saihesteko zuzenean ebaluazioa erabili beharrean, soluzioen ranking-a erabil dezakegu; era honetan soluziorik onenaren ekarpena azkenarena baino handiago izango da, baina lehenengo eta azkenaren soluzioen arteko diferentziak murriztuta egongo dira.

2.4 Particle Swarm Optimization

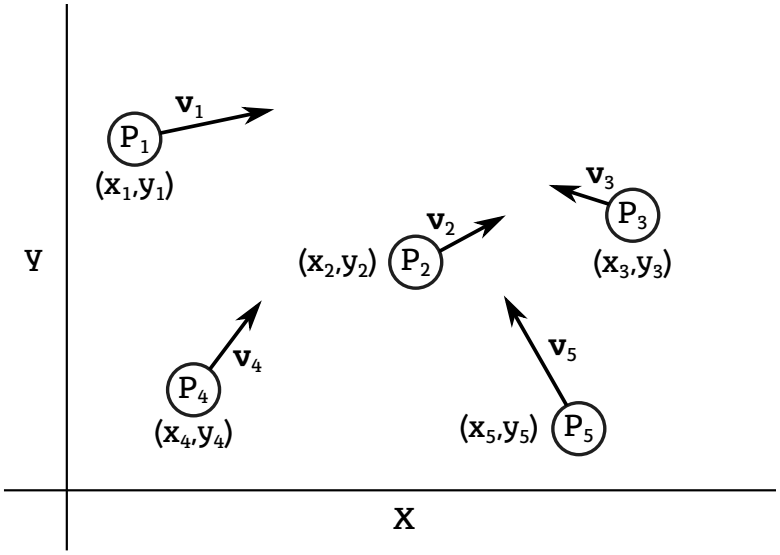
Intsektu sozialen portaera *swarm* adimenaren adibide tipikoak dira, baina ez dira bakarrak; animali handiagotan ere inspirazioa bila daiteke. Esate baterako, txori-saldotan ehunaka indibiduo batera mugitzen dira beraien arteak talka egin gabe. Multzo horietan ez dago indibiduo bat taldea kontrolatzen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

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 sinple bat. 10 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, indibiduoen 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

[5] TSP-an, adibidez, ez da berdina 10 herri Gipuzkoan izatea edo 100 herri Europan zehar banatuta



Irudia 10: PSO algoritmoak erabiltzen dituen partikulen adibidea. Partikula bakoitzak bere kokapena (x_i, y_i) eta bere abiadura $(\mathbf{v}_i)$ du

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 erabil daitezke (eraztunak, 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, partikularen ingurunean dauden beste partikulek lortu duten soluziorik onena, alegia. Soluzio honi ingelesez *global best*[6] 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 kalkulatuko 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 hauetaz gain, bi ausazko aldagai ditugu, ρ_1 eta ρ_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 ezarriz. Kalkulatutako abiaduraren modulua handiagoa bada, balio maximora eramaten da.

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

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

Iterazio bakoitzean lortutako soluzioak ebaluatu eta, beharrezkoa bada, partikulen *personal* eta *global best* eguneratu behar dira. Pasu guzti hauek 2.4 algoritmoan biltzen dira.

[6] 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

Bibliografia

[1] C. Blum and D. Merkle. *Swarm Intelligence: Introduction and Applications*. Springer-Verlag, 2008.

[2] T.D. Gwiazda. *Genetic algorithms reference Volume I Crossover for single-objective numerical optimization problems*. Number v. 1. Lightning Source, 2006.

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

[4] P. Larrañaga and J. A. Lozano. *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*. Kluwer Academic Publishers, 2002.

[5] 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.

[6] El-Ghazali Talbi. *Metaheuristics: From Design to Implementation*. Wiley Publishing, 2009.