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 komun bat: bilaketa prozesua soluzio batetik bestera mugitzen da, soluzioak banan-banan aztertuz. Ezaugarri hau dela eta, algoritmo hauek oso algoritmo egokiak dira bilaketa espazioaren eskualde interesgarriak arakatzeko –bilaketa areagotzeko, alegia–. Alabaina, hainbat kasutan emaitza onak lortzeko bilaketa dibertsifikatzea ere beharrezkoa da. Hau honela izanik, bilaketa lokalean oinarritzen diren algoritmo batzuk dibertsifikatzeko zenbait estrategia darabilte, adibide nabarmenena tabu bilaketaren epe-luzeko memoria izanik.

Kapitulu honetan soluzioen banan-banakako azterketa alde batera utzita, soluzio multzoak erabiltzeari ekingo diogu, horixe baita, hain justu, populazioetan oinarritzen diren algoritmoen filosofia. Algoritmoaren pausu bakoitzean, soluzio bakar bat izan beharrean, soluzio multzo bat izango dugu. Testuinguru batzuetan soluzio multzo honi «soluzio-populazioa» deritzo eta, hortik, algoritmo hauen izena. Algoritmo hauekin, bilaketa prozesuan zehar, soluzio multzo hori aldatuz joango da helburu funtzioaren gidaritzapean, gelditze irizpide bat bete arte.

Oro har, populazioan oinarritutako algoritmoak bi multzotan banatu ditzakegu: algoritmo ebolutiboak eta *swarm intelligence*-an oinarritutakoak. Lehenengo kategoriako algoritmoek, populazioa eboluzionarazten dute, teknika ezberdinak erabiliz, honek geroz eta soluzio hobeak izan ditzan. Adibiderik ezagunenak algoritmo genetikoak dira. Bigarren motako algoritmoak, berriz, zenbait animalien portaeran oinarritzen dira. Hauen arteko adibiderik ezagunenak, esate baterako, inurriek, janaria eta inurritegiaren arteko distantziarik motzena topatzeko darabilten mekanismoa imitatzen du.

Kapitulua bi zatitan banaturik dago, bakoitza populazioan oinarritutako algoritmo mota bati eskeinita. Lehenengo zatian, algoritmo ebolutiboen eskema orokorra ikusi ondoren, algoritmo genetikoak [6] eta EDAk [8, 9] aurkeztuko dira. Bigarren zatian ordea, *swarm intelligence* [2] arloan proposaturiko bi algoritmo aztertuko dira.

1 Algoritmo Ebolutiboak

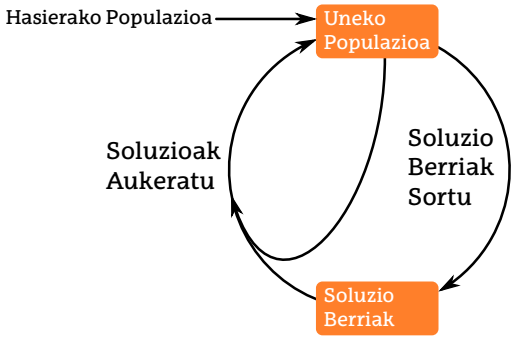
1859. urtean Charles R. Darwinek *On the Origin of the Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life* liburua argitaratu zuen. Tituluak berak adierazten duen bezala, liburu honetan Darwinek hautespen naturalaren teoria aurkeztu zuen.

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

Hirurogeigarren hamarkadan ikertzaileek Darwinen lana inspiraziotzat hartu zuten optimizazio metahuristikoak diseinatzeko eta geroztik, konputazio ebolutiboa konputazio zientzien arlo bereizia bilakatu da. Atal honetan bi algoritmo mota aztertuko ditugu, algoritmo genetiko klasikoak [6] eta EDAk (*Estimation of Distribution Algorithms*) [8, 9].

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 ebolutibo guztiek 1 irudiko eskema orokorra jarraitzen dute. Eskema orokor honetan, bi dira giltzarri diren elementuak: soluzio berrien sorkuntza eta soluzioen hautespena. Algoritmoaren sarrera-puntua hasierako populazioa izango da; populazio horretatik abiatuz, algoritmoa begizta nagusian sartzen da, non bi pausu txandakatzen diren. Lehenik, uneko populazioko soluzioetatik abiatuz, soluzio berri multzo bat sortuko da. Ondoren, soluzio berri hauek eta uneko populazioko soluzioak kontutan hartuz, naturan bezala, hurrengo belaunaldiara pasatzeko soluzio onak aukeratuko ditugu, eta populazio berri bat sortuko dugu. Begizta nagusia etengabekoa denez, zenbait irizpide ezberdin proposatu dira algoritmoa amaitutzat emateko.

Hurrengo atalean, algoritmo orokor honen urratsak sakonago aztertuko ditugu. Hasteko, algoritmo ebolutibo guztietan antzerakoak diren pausuak azalduko ditugu eta ondoren, bi algoritmo ezberdinen xehetasunetan jarriko dugu arreta.

1.1 Urrats orokorrak

Ondorengo ataletan ikusiko ditugun algoritmoen arteko diferentzia nagusia soluzio berrien sorkuntzan datza. Gaionontzeko urratsak era antzekoan burutzen dira algoritmo ezberdinetan eta horiei buruz hitz egingo dugu atal honetan.

1.1.1 Populazioaren hasieraketa

Nahiz eta askotan garrantzi gutxi eman, hasierako populazioa da algoritmoaren abia-puntua eta, hortaz, bere sorkuntza oso pausu garrantzitsua da, eragin handia izaten duelako algoritmoak lortutako azken emaitzan.

Algoritmoen xedea soluzio onak topatzea izanda, pentsa dezakegu hasierako populazio on bat soluzio onez osatuta egon behar dela; alabaina, dibertsitatea soluzioen kalitatea bezain garrantzitsua da. Nahiz eta onak izan, populazioa oso soluzio antzerakoez osatuta badago, populazioaren eboluzioa oso zaila izango da eta algoritmoak azkarregi konbergitu dezake optimoa ez den soluzio batera.

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.

Ausazko laginketak lortutakoa baina dibertsitate handiagoa bermatu nahi badugu, –sasiausazko– prozedura batzuk erabil ditzakegu. Hori dela eta, proposatu dira beste prozedura batzuk populazioak sasi-ausaz sortzeko dibertsitatea maximizatuz. Esate baterako, dibertsifikazio sekuentziala aplikatu dezakegu, zeinak soluzio berri bat onartzen duen soilik populazioan dauden soluzioekiko distantzia minimo batera badago. Adibide moduan, demagun 25 tamainako bektore bitarren 10 tamainako populazio bat sortu nahi dugula. Dibertsitatea bermatzeko populazioko soluzioen arteko Hamming distantzia minimoak 10-ekoa izan behar duela inposa dezakegu.

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

```
> hamm.distance <- function (v1 , v2){
+   d <- sum(v1!=v2)
+   return(d)
+ }
>
> rnd.binary <- function(n){
+   return (runif(n) > 0.5)
+ }
```

Gero, soluzioak ausaz sortzen ditugu eta, distantzia minimoko baldintza bete ezean, deusestatu egiten ditugu; 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
+ }
```

Prozedura hau ez da batere eraginkorra, zenbait kasutan soluzio asko aztertu beharko baititugu populazioa osatu arte. Hala, beste alternatiba eraginkorrago 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. Aldiz, hasierako popuazioaren kalitatea hobetu nahi izanez gero, hasieraketa heuristikoak erabil daitezke. Hau lortzeko era sinple bat, GRASP algoritmoetan ausazko soluzioak sortzeko erabiltzen diren prozedurak erabiltzea da. Ondoko lerroetan Bavierako hirien TSP problemarako adibide bat ikus dezakegu. Lehenik, 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)
+ }
```

Aurreko kapituluetan azaldu bezala, `tsp.greedy` funtzioak TSP-rako algoritmo eraikitzaile bat inplementatzen du; pausu bakoitzean, uneko hiritik zein hirira mugituko garen erabakitzen da, gertuen dauden `cl.size` hirietatik –5, gure kasuan– bat ausaz aukeratuz. Honetan oinarrituz, populazioa sortzeko funtzio hau erabiliko dugu.

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

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

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

```
> 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))
```

Bi populazioen ebaluazioak *boxplot* baten bidez aldera ditzakegu:

```
> 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. Helburua minimizazioa dela kontutan hartuz, argi eta garbi ikus daiteke heuristikoa erabiliz sortutako soluzioak hobeak direla.

Soluzioak sortzeko metodoak ez ezik, populazioaren tamainak ere badu eragin handia azken emaitzan, eta egokitu beharreko oso parametro garrantzitsua da. Algoritmoak darabiltzan populazioak txikiegiak badira, dibertsitatea mantentzea oso zaila izango da eta, hortaz, belaunaldi gutxitan algoritmoak konbergitu egingo du, ziurrenik optimoa ez den soluzio batera. Beste aldetik, populazioak handiegiak badira, konbergentzia abiadura motelagoa izango da baina, ondorioz, kostu konputazionala ere handiagoa bilakatuko da; hurrengo atalean adibide baten bidez ikusiko dugu hau. Honenbestez, ez dago irizpide finkorik populazioen tamaina ezartzeko eta problema bakoitzerako balio egoki bat bilatu beharko da. Edonola ere, irizpide orokor gisa esan dezakegu populazioak azkar konbergitzen badu –hots, soluzioen arteko distantzia azkar txikitzen bada–, soluzio hobeak lortzeko modua populazioaren tamaina handitzea izan daitekeela.

1.1.2 Hautespena

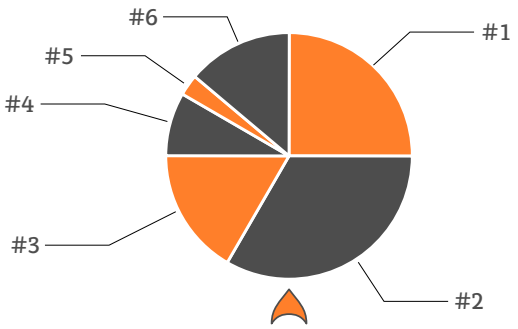
Algoritmo ebolutibotan populazioko soluzio batzuen aukeraketa urrats garrantzitsua da, populazioaren eboluzioa kontrolatzen duen prozesua baita. Orokorrean, populazioan dauden soluziorik onenak hautatzea da gehien erabiltzen den hautespen irizpidea: hautespen «elitista» Àlabaina, soluzio onak aukeratzea garrantzitsua bada ere, dibertsitatea mantentzearren, tarteka soluzio txarrak sartzea ere komenigarria izaten da. Hau zuzenean egin daiteke, baina badaude aukeraketa metodo egokiago batzuk soluzio txarrak estrategia probabilistikoak erabiliz aukeratzen dituztenak.

Erruleta-hautespena, (*Roulette Wheel selection*, ingelesez) deritzon estrategian soluzioak erruleta batean kokatzen dira; soluzio bakoitzari, bere ebaluazioarekiko proportzionala den, erruletaren zati bat esleituko zaio. Hau honela, 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 adina erruleta jaurtiketa egin ditzakegu.

Azkenik, esan beharra dago, *fitness*aren magnitudea problema eta, batez ere, instantzien araberakoa dela. Hori dela eta, erruleta banatzeko probabilitateak zuzenean helburu funtzioaren balioak erabiliz kalkulatzen badira, oso distribuzio erradikalak izan ditzakegu. Arazo hau ekiditeko, helburu funtzioaren balioa zuzenean erabili beharrean soluzioen ranking-a erabiltzea posible da.

Beste hautespen probabilistiko mota bat Lehiaketa-hautespena da. Estrategia honekin soluzioen aukeraketa bi pausutan egiten da. Lehenengo urratsean indibiduo guztietatik azpi-multzo bat aukeratzen da, guztiz ausaz (ebaluazioa kontutan hartu barik). Ondoren, azpi-multzo honetatik soluziorik onena hautatzen dugu. Azpi-multzoen eraketa guztiz ausaz egiten denez, hauetako batzuk, oso soluzio txarrez osatuta egon daitezke. Kasu hauetan, nahiz eta onena aukeratu, populazio berrirako gordeko dugun soluzioa ez da ona izango eta, honenbestez, soluzio on eta txarren aukeraketa baimentzen du hautespen metodo honek.

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.

1.1.3 Gelditze Irizpideak

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

Bestalde, gelditzeko irizpide dinamikoak, hau da, eboluzioaren prozesuari erreparatzen diotenak ere erabili daitezke. Balunaldiz belaunaldi populazioan dauden soluzioak geroz eta hobeak dira eta, aldi berean, populazioaren dibertsitatea murrizten da, soluzio batera konbergitzeko joerarekin. Hau hala izanik, populazioaren dibertsitatea gelditze irizpide dinamikoak eraikitzeko erabil daiteke.

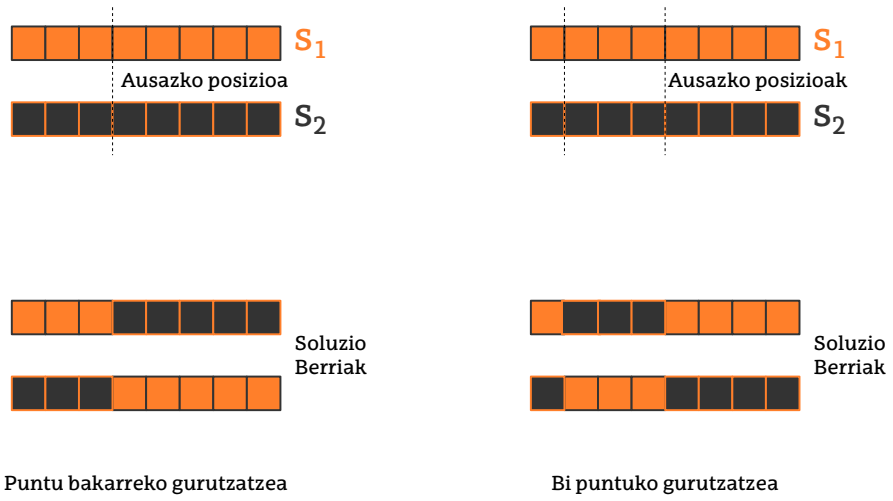
Dibertsitatea soluzioei zein beraien *fitness*-ari erreparatuz neur daiteke. Esate baterako, soluzioen arteko distatzia neurtzerik badago, indibiduoen arteko bataz besteko distantzia minimo bat ezar dezakegu gelditze irizpide gisa.

1.1.4 Algoritmo Genetikoak

Atal honetan algoritmo genetikoetan [6] soluzio berriak nola sortzen diren ikusiko dugu. Algoritmo genetikoak naturan espezieen eboluzioarekin gertatzen dena imitatzen dute eta, beraz, fenomeno honekin zenbait paralelismo ezar daitezke:

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

Beraz, algoritmo genetikoetan soluzio berriak sortzeko estrategiak diseinatzean indibiduen ugalketa prozesuan oinarrituko gara. Ugalketa prozesuaren xedea zenbait indibiduo emanda –bi, normalean–, indibiduo berriak sortzea da. Ohikoena prozesu hau bi pausutan banatzea da: soluzioen gurutzaketa eta mutazioa. Lehenaren helburua «guraso» -soluzioek dituzten ezaugarriak soluzio berriei pasatzea da, espezieen gurutzaketan jazotzen den bezala. Bigarrenarena, berriz, sortutako soluzio berriei ezaugarri berriak eranstea da. Jarraian soluzioak maneiatzeko bi operadore hauek aztertuko ditugu.



Irudia 4: Gurutzatze-operadoreak bektoreen bidezko kodeketarekin erabiltzeko

1.1.5 Gurutzaketa

Bi soluzio –edo gehiago– gurutzatzen ditugunean euren propietateak sortutako soluzio berriei transmititzea da helburua. Optimizazio arloan, soluzioen arteko gurutzaketak «gurutzaketa-operadore» -en –*crossover*, ingelesez– bidez egiten dira. Operadore hauek soluzioen kodeketarekin dihardute eta, beraz, gurutzatze operadore zehatz bat hautatzean soluzioak nola adieratzen ditugun aintzat hartu beharko dugu.

Badaude kodeketa klasikoekin erabil daitezkeen zenbait oinarrizko gurutzaketa operadore. Ezagunena puntu bakarreko gurutzaketa – *one-point crossover*, ingelesez – deritzona da. Demagun soluzioak bektoreen bidez kodetzen ditugula. Bi soluzio, s_1 eta s_2 hartuz, operadore honek beste bi soluzio berri sortzen ditu. Horretarako, lehenik eta behin, ausazko posizio bat, i, aukeratu behar da. Hau egin ahala, lehenengo soluzio berria s_1 soluziotik lehenengo i elementuak eta s_2 soluziotik gainontzekoak ($i+1$-tik aurrerakoak) kopiatuz sortuko dugu. Era berean, bigarren soluzio berria s_2-tik lehenengo i elementuak eta s_1-etik $i+1$ posiziotik aurrerako elementuak kopiatuz sortuko dugu. 4 irudiaren ezkerraldean *one-point crossover* operazioaren aplikazioaren adibide bat ikus daiteke. Gainera, eskuineko irudiak operadore hau nola orokortu daitekeen erakusten du, puntu bakar bat erabili beharrean bi, hiru, etab. puntu erabiliz.

Azken operadore orokorrago honi, *k-point crossover* deritzo eta `metaheuR` liburutegiko `k.point.crossover` funtzioan inplementaturik dago. Ikus ditzagun bere erabileraren adibide batzuk:

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

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

> B.sol

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

> 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 $n-1$ puntuko gurutzaketa aplikatu dezakegu; edonola ere, balio handiago bat aukeratzen badugu funtzioak abisu bat emango du eta puntu kopuruaren parametroa bere balio maximoan ezarriko du. Balio maximoa aukeratuz gero, jatorrizko «guraso» soluzioen elementuak tartekatuta agertuko dira soluzio berrietan; operadore honi *uniform crossover* deritzo.

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

k-point crossover operadorea nahiko orokorra da, ia edozen bektoreari aplikatu ahal baitzaio. Hala eta guztiz ere, kodeketa batzuetan beste operadore espezifikoagoak erabiltzea egokiagoa izan daiteke[5]. Esate baterako, soluzioak bektore errealen bidez kodetuta badaude, bi soluzio era ezberdin askotan 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, bete beharreko murrizketak direla eta, *k-point crossover* operadorea ezin da erabili kodeketa mota honekin. Ikus dezagun hau argiago ikusten lagunduko digun adibide bat. 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. Hortaz, permutazioak gurutatzeko operadore bereziak behar ditugu.

Aukera asko izan arren [10], hemen puntu bakarreko gurutzatze operadorearen baliokidea ikusiko dugu. Puntu bateko gurutzaketan bezala, hasteko, puntu bat aukeratuko dugu ausaz, i. Ondoren, lehenengo soluzio berria sortzeko, «guraso» soluzio baten lehenengo i posizioetako balioak zuzenean kopiatuko ditugu; gainontzeko balioak zuzenean beste «guraso» soluziotik kopiatu beharrean, ordena bakarrik hartuko dugu kontutan. Hau da, aurreko adibidera itzuliz, soluzio berria sortzeko s_1-etik lehenengo 3 elementuak zuzenean kopiatuko ditugu, 123, eta falta direnak, 45678, s_2-an agertzen diren ordenean kopiatuko ditugu, hots, 87654. Emaitza, beraz, $s' = 12387654$ izango da eta, kasu honetan bai, permutazio bat. Era berean, bigarren soluzio berri bat sor daiteke s_2-tik lehenengo hiru posizioak kopiatuz (876) eta gainontzekoak s_1-n agertzen diren ordenean kopiatuz (12345); beste soluzioa, beraz, 87612345 izango da. Operadore honi «Order crossover» deritzo eta `metaheuR` liburutegian `order.crossover` funtzioan [1]. inplementaturik dago.

```
> 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 populazioak eboluzionatu ahal izateko dibertsitatea garrantzitsua da. Hori dela eta, behin gurutzatze-operadorearen bidez soluzio berriak lortuta, hauetan ausazko aldaketak eragin ohi dira mutazio operadorearen bidez.

Mutazioaren kontzeptua ILS algoritmoko perturbazioaren antzerakoa da eta kasu horretan bezalaxe, operadore ezberdinak erabil daitezke mutazioa burutzeko. Hala nola, ILS-an bezala, algoritmoa diseinatzean erabaki behar dugu zenbateko aldaketak eragingo ditugun soluzioetan. Esate baterako, permutazio bat mutatzeko ausazko trukaketak erabil ditzakegu baina zenbat posizio trukatuko ditugun aldez aurretik erabaki beharko dugu.

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.

Mutazio operadorea aukeratzean –eta baita diseinatzean ere– hainbat gauza hartu behar dira kontuan. Hasteko, soluzioen bideragarritasuna mantentzea garrantzitsua da, hau da, mutazio operadorea bideragarria den soluzio bati aplikatuz gero, emaitzak soluzio bideragarria izan behar du. Bestalde, bilaketa prozesuak soluzio bideragarrien espazio osoa arakatzeko gaitasuna izan behar du eta, hori bermatzeko, mutazio operadoreak

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

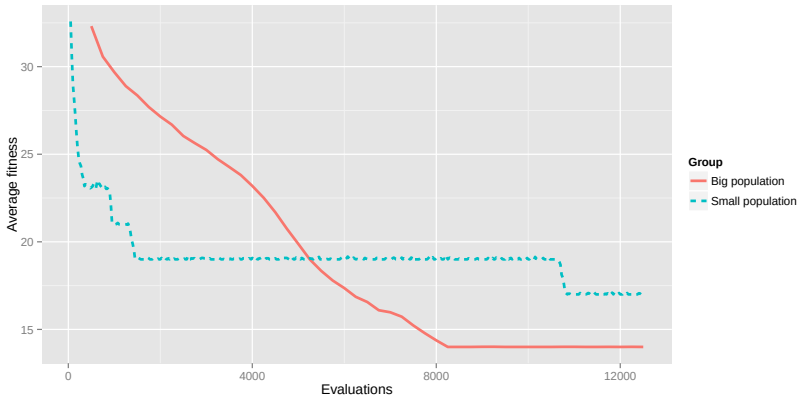
edozein soluzio sortzeko gai izan behar du. Hau da, edozein soluzio hartuta, mutazio operadorearen bidez beste edozein soluzio sortzea posible izan behar du. Amaitzeko, lokaltasuna ere mantendu behar da –hau da, mutazioak eragindako aldaketa txikia izan behar da–, gurasoengandik heredatutako ezaugarriak galdu ez daitezen.

Honenbestez, algoritmo genetiko orokorra 1.1 irudian ikus daiteke eta sasikode hay `metaheuR` paketeko `basic.genetic.algorithm` funtzioan inplementaturik dago. Ikus dezagun funtzio honen erabilpenaren adibide bat *graph coloring* problemaren instantzia bat ebazteko. Lehenik, ausazko grafo bat sortuko dugu problemaren instantzia sortzeko.

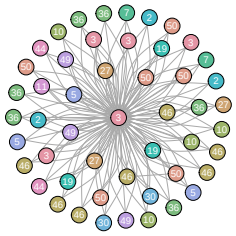
```
> library(igraph)
> n <- 50
> rnd.graph <- aging.ba.game(n = n , pa.exp = 2 ,
+                            aging.exp = 0, m = 3 , directed = F)
> gcp <- graph.coloring.problem (graph = rnd.graph)
```

Orain zenbait elementu definitu behar ditugu. Lehenengoa, hasierako populazioa izango da eta, sortu ahal izateko, lehenik, bere tamaina ezarri behar dugu. Hau algoritmoaren parametro garrantzitsu bat denez, bi balio ezberdinekin probatuko dugu, emaitzak alderatzeko: n eta $10n$. Behin hasierako populazioaren tamaina definituta bertako soluzioak ausaz sortuko ditugu eta, bideragarriak ez badira, zuzenduko egingo ditugu `gcp` objektuaren `correct` funtzioa erabiliz.

```
> n.pop.small <- n
> n.pop.big <- 10*n
> levels <- paste("C" , 1:n , sep = "")
> rnd.sol <- function(x){
+   sol <- factor(paste("C" , sample(1:n , size = n , replace = TRUE) ,
+                       sep = "") , levels = levels)
```



(a) Algoritmo genetikoaren progresioa



(b) Lortutako soluzioa

Irudia 5: Definitutako algoritmo genetikoaren progresioa *graph coloring* problemaren instantzia batean batean, bi populazio tamaina ezberdin erabiliz. Ezkerrean, tamaina handiko populazioarekin lortutako soluzioa ikus daiteke.

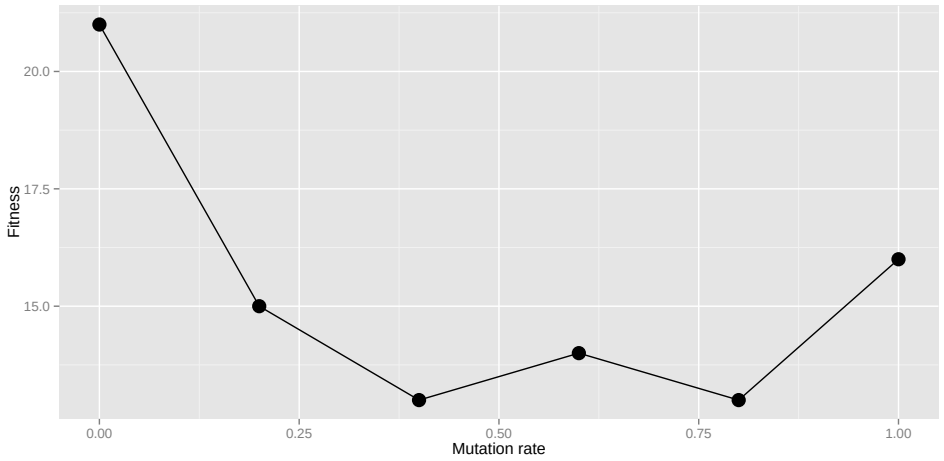
```
+   return(gcp$correct(sol))
+ }
> pop.small <- lapply(1:n.pop.small , FUN = rnd.sol)
> pop.big   <- lapply(1:n.pop.big , FUN = rnd.sol)
```

Hasierako populazioaz gain, ondoko parametro hauek ezarri behar ditugu:

- Hautespen operadoreak - Hurrengo belaunaldira zuzenean pasatuko diren soluzioak aukeratzeko hautespen elitista erabiliko dugu, populazio erdia aukeratuz; zein soluzio gurutzatuko diren aukeratzeko, berriz, lehiaketa hautespena erabiliko dugu.
- Mutazioa - Soluzioak mutatzeko `factor.mutation` funtzioa erabiliko dugu. Funtzio honek zenbait posizio ausaz aukeratzen ditu eta bertako balioak ausaz aldatzen ditu. Funtzioak parametro bat du, `ratio`, aldatuko diren posizioen ratioa adierazten duena. Gure kasuan 0.1 balioa erabiliko dugu, alegia, posizioen %10-a aldatuko da mutazioa aplikatzen denean. Soluzioak zein probabilitatearekin mutatuko ditugun soluzioak ere aurrez finkatu behar da, `mutation.rate` parametroaren bidez. Gure kasuan, probabilitatea bat zati populazioaren tamaina izango da.
- Gurutzaketa - Soluzioak gurutzatzeko *k-point crossover* operadorea erabiliko dugu, $k = 2$ finkatuz.
- Beste parametro batzuk - Algoritmo genetikoaren parametroaz gain, beste bi parametro finkatuko ditugu, `non.valid = 'discard'`, 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 ikus dezakegu.

```
> args <- list()
> args$evaluate             <- gcp$evaluate
> args$initial.population   <- pop.small
> args$select.subpopulation <- elitist.selection
> args$selection.ratio      <- 0.5
> args$select.cross         <- tournament.selection
> args$mutate               <- factor.mutation
```



Irudia 6: Mutazio probabilitatearen eragina algoritmo genetikoaren azken emaitzan. Irudian ikus daitekeen bezala, soluziorik onena ematen duen mutazio probabilitatearen balioa 0.5 inguruan dago (zehazki, 0.6).

```
> args$ratio                <- 0.1
> args$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 tamaina ezberdin 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 helburu funtzioaren balioaren eboluzioa motelagoa izan arren, lortutako soluzioa hobea dela.

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

```
> args$initial.population   <- pop.small
> args$verbose              <- FALSE
> args$resources            <- cresource(evaluations = n^2)
>
> test.mutprob <- function (rate){
+   args$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 lortutako emaitzak ikus daitezke. Grafikoak agerian uzten du mutazio probabilitate txikiegiak zein handiegiak ezartzea kaltegarria dela bilaketa prozesuarenzat, probabilitate egokiena 0.5 ingurukoa izanik. Jokabide honen zergatia bilatzen badugu, konturatzen gara probabilitate txikien kasuan emaitzak txarrak direla, populazioaren dibertsitatea txikia delako eta, ondorioz, konbergentzia goiztiarra ematen delako. Probabilitate handiekin berriz, arazoa justo kontrakoa izan daiteke, alegia, bilaketa ia ausazkoa dela eta beraz konbergentzia oso zaila dela. Hau egiaztatzeko, goiko experimentua errepika dezakezu ebaluazio kopuru maximoa handituz.

1.2 Estimation of Distribution Algorithms

Algoritmo genetikoetan uneko populazioa indibiduo berriak sortzeko erabiltzen da, naturan inspiratutako gurutzatze eta mutazio operadoreak aplikatuz. Prozesu honen bitartez, populazioan dauden ezaugarriak mantentzea espero dugu.

Zenbait ikertzailek ideia hau hartu eta ikuspuntu matematikotik birformulatu zuten. Honela, gurutzatze eta mutazioa erabili beharrean, eredu probabilistikoak erabiltzea proposatu zuten, populazioaren «esentzia» jasotzeko helburuarekin. Hauxe da, EDA – *Estimation of Distribution Algorithms* – algoritmoen ideia nagusia.

Algoritmo genetikoen eta EDA motako algoritmoen artean dagoen diferentzia bakarra indibiduo berriak sortzeko erabiltzen den estrategia da. Gurutzaketa eta mutazioa erabili beharrean, uneko populazioa eredu probabilistiko bat doitzeko erabiltzen da. Ondoren, eredu hori laginduko dugu behar adina indibiduo 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 estimatuko dugu. Gero, indibiduoak sortzean soluzioaren osagaiak banan-banan aukeratuko ditugu probabilitate hauek jarraituz.

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

Ikasitako eredu probabilistikoa soluzio berriak sortzeko erabil daiteke, posizioz-posizio dagokion marginala laginduz; laginketa `simulate` funtzioaren bidez egiten da.

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

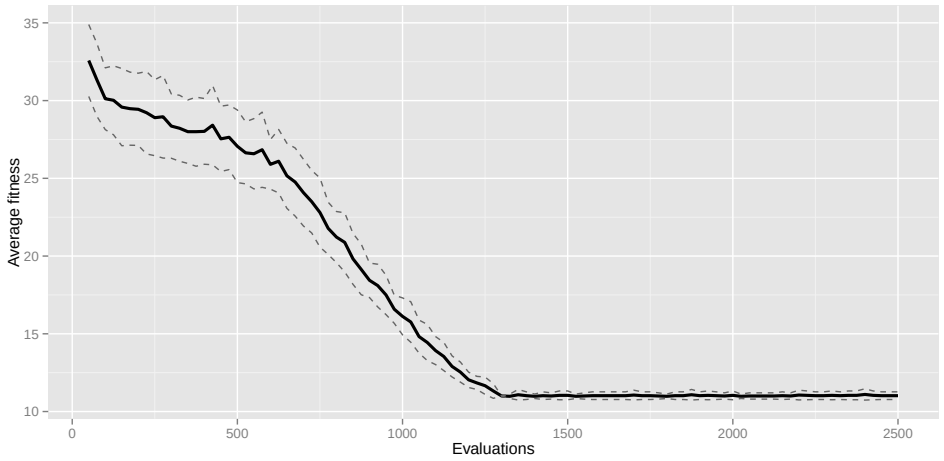
```
## [[1]]
##  [1] 3 1 2 2 1 3 3 2 2 3
## Levels: 1 2 3
##
## [[2]]
##  [1] 2 2 2 2 3 3 3 3 3 1
## Levels: 1 2 3
```

UMDA `basic.eda` funtzioaren bidez exekutatu dezakegu eta, segidan, aurreko ataleko problema ebazteko erabiliko dugu. Horretarako, bakarrik hautespen operadoreak eta ereduak ikasteko funtzioak zehaztu behar ditugu –algoritmo genetikoekin komunak diren parametroez 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 desbiderapena erakusten dute eta marra jarraikiak, ordea, populazioko soluzioen fitness-aren batez-bestekoa. Populazioak eboluzionatu ahala, populazioaren dibertsitatea murrizten dela ikus dezakegu desbiderapenaren murrizketan erreparatuz. Amaieran, bilaketak 11 kolore darabiltzan soluzio batera konbergitzen du.

Marjinalak kalkulatzeko estrategia ia edozein bektore motarekin erabil daitezke zuzenean; alabaina, balio errealak baditugu, marjinalak zein probabilitate distribuzioarekin modelatuko ditugun erabaki beharko dugu aurrez. Aukera ezberdin asko daude baina ohiko distribuzio bat distribuzio normala da. Gainera, soluzioek murrizketak dituztenean, permutazioetan kasu, gauzak konplikatu egiten dira.



Irudia 7: UMDA algoritmoaren progresioa *graph coloring* problemaren istantzia batean aplikatuta. 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 aldakortasuna murrizten dela.

Permutazio multzo bat izanik, posible da marjinalak bektore kategorikoekin bezala estimatzea baina, ondoren, eredua lagintzen dugunean ez ditugu halabeharrez 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, permutazioak dakarzten murrizketak aintzat hartu behar dira. Honela, laginketa prozesuan lehenengo elementua ausaz aukeratuko dugu, zuzenean 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
```

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

```
> id <- which(remaining %in% new.element)
> remaining <- remaining [-id]
> marginals <- marginals[-id , ]
> probabilities <- marginals[ , 2]
> probabilities <- probabilities / sum(probabilities)
```

```
> new.element <- sample(remaining , size = 1 , prob = probabilities)
> new.solution <- c(new.solution , new.element)
```

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

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

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

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

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

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

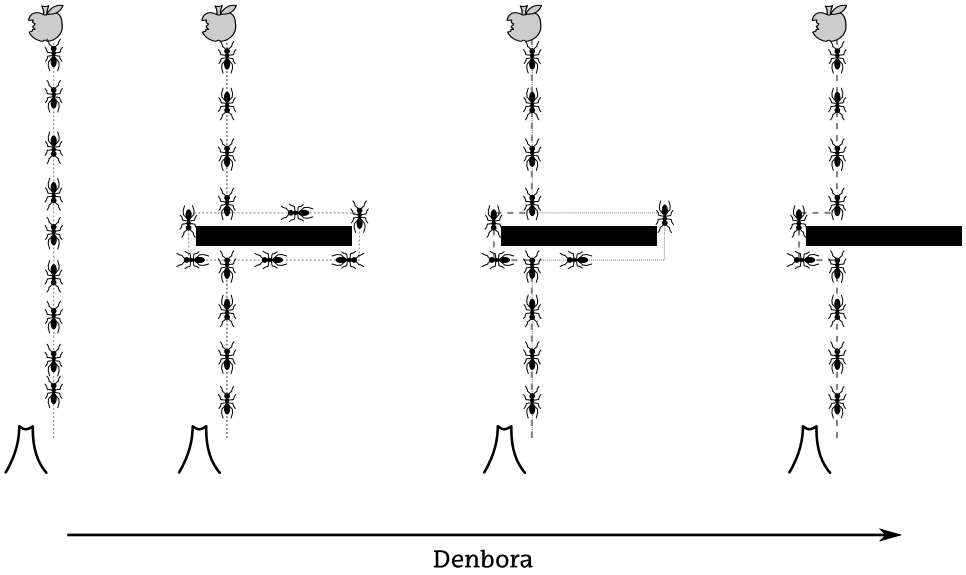
2 Swarm Intelligence

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 espezietan –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 indibiduoen portaera sinpleen eta, batez ere, indibiduoen 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].

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



Irudia 8: Feromonaren erabilera. Hasierako egoeran biderik motzena feromonaren bidez markatuta dago. Bidea mozten dugunean, inurriek, eskumara edo ezkerrera 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 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 eta inurriak bide motzetik bakarrik joango dira.

2.1 *Ant Colony Optimization*

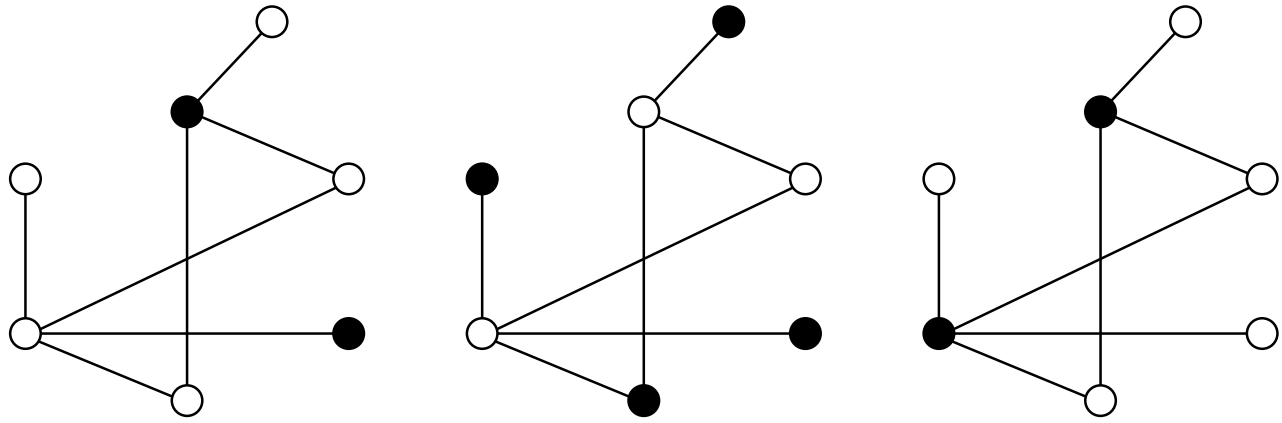
Inurriek, janaria topatzen dutenean, beraien koloniatik janarira biderik motzena topatzeko gaitasuna dute. Inurri bakar batek ezin du horrelakorik egin baina, taldeka, komunikazio mekanismo sinpleei esker ataza burutzeko gai dira. Erabiltzen den komunikabidea zeharkakoa da, darien molekula mota berezi 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 lorratza 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 lurrunkorrak 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 lorratza apurka-apurka baporatuko da eta, denbora nahiko igarotzen bada, zeharo galduko da.

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

Lorratzak 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.



(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 nodoei konektaturik ez baitago. Bigarren soluzioa bideragarria da, baina ez optimoa. Azken soluzioa optimoa da, ez baitago 1 tamainako soluzio bideragarririk.

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 osagaitzat 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 eredua matrize sinple baten bidez inplementa 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` paketean inplementaturik dago, `VectorPheromone` klasean. Ereduaren erabilera argitzeko *Maximum Dominating Set* (MDS) problema erabiliko dugu. Laburki, grafo bat emanda, nodoen azpimultzo bat menderatze-multzoa da –*dominating set*, ingelesez– baldin eta azpimulzoan ez dauden nodo guztiak gutxienez azpimultzoko nodo bati konektatuta badaude; MDS probleman, 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 eredua 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.fractor`. 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
```

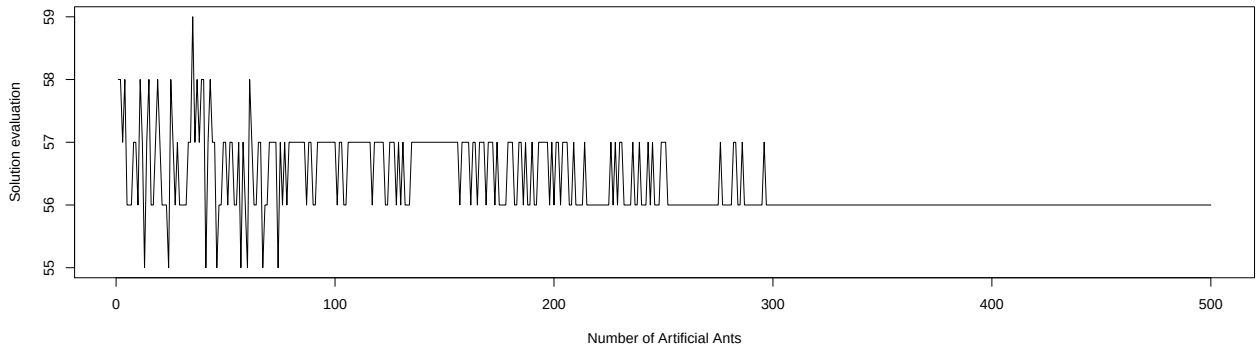
Esan dugun bezala, inurriek feromona ereduak soluzioak eraikitzeko erabiltzen dute. Soluzioak `build.solution` funtzioaren bitartez egiten da.

```
> 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
```

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.



Irudia 10: ACO sinplearen eboluzioa MDS problema batean.

```
> 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 bariabilidade handia dagoela sortutako soluzioen helburu funtzioaren balioan. Alabaina, inurri kopurua handitzen den heinean bariabilidadea 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 funtzionatzen, 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 funtzioaren arabera egitea ez da beharrezkoa, bakarrik soluziorik onena erabiltzen baita eguneraketan. Hori dela eta, ohikoa da balio finko bat erabiltzea eguneraketan.

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
```

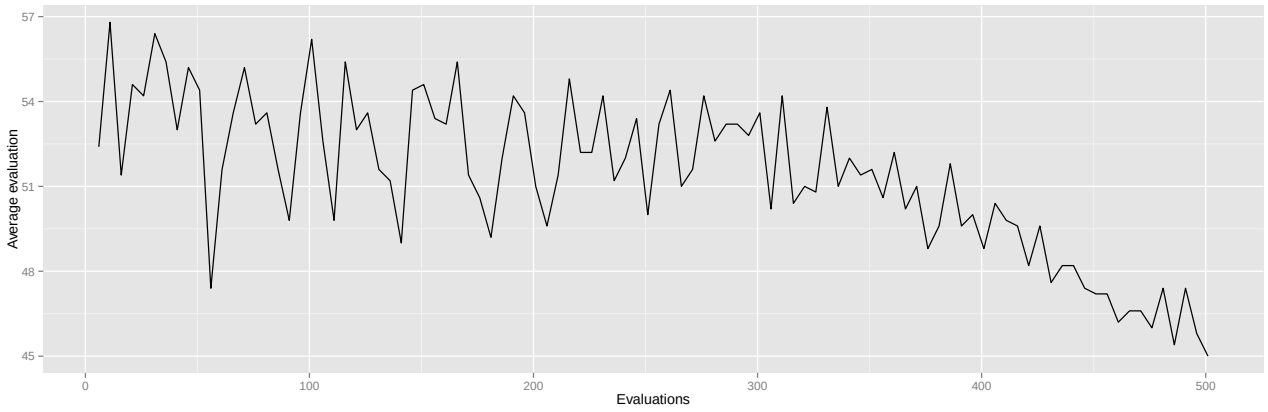
Algoritmoa 2.1: Inurri-kolonien algoritmoaren sasikodea

- 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 oinarrizko ACO algoritmoaren sasikodea defini dezakegu (ikusi 2.1 algoritmoa). `basic.aco` funtzioak Oinarrizko ACO-a inplementatzen 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 kalkulatzean. Hori dela eta, helburu funtzioaren zeinua kontutan hartu behar da feromona eredua hasieratzeko.

```
> args <- list()
> args$evaluate         <- mdsp$evaluate
> args$nants            <- 5
> init.value            <- 1
> initial.trail <- matrix(rep(init.value , 2*n ) , nrow=2)
> evapor <- 0.9
> pher <- vectorPheromone(binary = TRUE , initial.trail = initial.trail ,
+                         evaporation.factor = evapor)
> args$pheromones       <- pher
> args$update.sol       <- 'best.it'
```



Irudia 11: Oinarrizko ACO algoritmoaren eboluzioa MDS problema batean.

```
> 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 oinarrizko 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, prograsioa ez da hain zaratatsua 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, bektoreekin erabilitako eredua 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 probleman 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.

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

Irudia 12: Oinarrizko ACO algoritmoaren eboluzioa LOP problema batean.

Bi eredu hauek `metaheuR` librutegian 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))
```

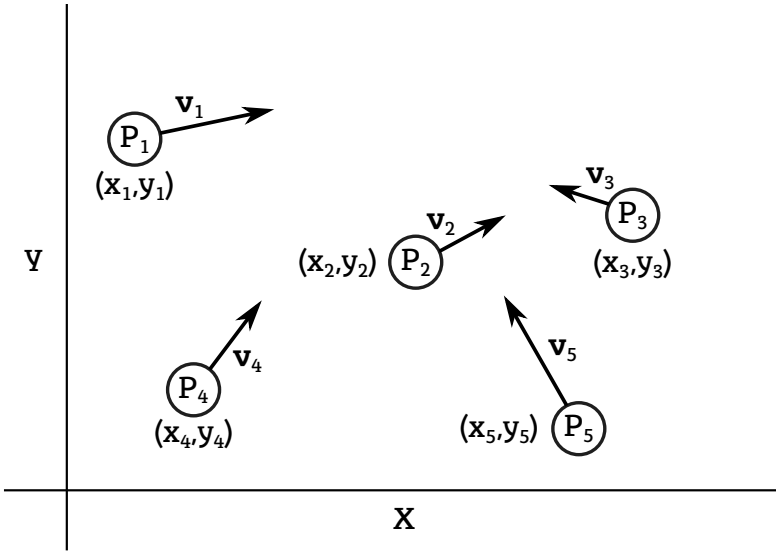
12 irudiak experimentuaren emaitza erakusten du. Grafikan argi ikus daiteke noden arteko lotruak osagaitzat hartzen direnean, bilaketak ez du aurrera egiten. Osagaiak posizioak direnean, berriz, iterazioz iterazio soluzioa hobetzen da. Honen arrazoia sinplea da: LOP probleman posizio absolutuak dira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

TSP probleman justo 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. Jarrain experimentu hori egingo dugu; emaitzak 13 irudian daude.

Irudia 13: Oinarrizko ACO algoritmoaren eboluzioa TSP problema batean.

```
> 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 ditugun adibide guztietan feromonak bakarrik erabiltzen dira soluzioak eraikitzerakoan. Oinarrizko algoritmoan horrela izan arren, problema konkretu bat ebatzi behar denean, posible bada, informazio heuristikoa sartu ohi da. Adibide gisa, TSPrako algoritmo eraikitzaile tipikoan hurrengo hiria 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.



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

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 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.

Animali talde hauen portaera inspiraziotzat hartuz, 1995ean Kennedy eta Eberhart *Particle Swarm Optimization* (PSO) algoritmoa proposatu zuten [7], optimizazio numerikoko problemak ebazteko[2]. Algoritmoaren ideia sinplea da oso; bilaketa espaziotik mugitzen diren partikulak erabiltzen direa bilaketa egiteko. Beraz, algoritmoaren izenak adierazten duen legez, hainbat partikula izango ditugu; partikula bakoitzaren posizioak problemarako soluzio bat adieraziko du.

Uneoro partikula bakoitzak kokapen eta abiadura zehatz bat izango du, 14 irudian erakusten den bezala. Irudiko adibidean bilaketa espazioak bi aldagai besterik ez ditu (X eta Y), eta sisteman 5 partikula daude, P_1-etik P_5-era.

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.

Lehen esan dugu algoritmoaren inspirazioa txori-maldoen portaera dela. Txori bakoitzak nora mugitu behar den erabakitzeko bere ingurunean dauden txoriei erreparetzen die. Era berean, algoritmoan partikula baten abiadura eguneratzean partikula horrek duen informazioa ez ezik, inguruneko partikulek duten informazioa ere erabiltzen da. Hain zuzen, i. partikularen abiadura eguneratzeko ondoko ekuazioa erabiltzen da:

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

Ekuazio honetan hiru termin daude:

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

[2]Hau da, atal honetan ikusiko dugun algoritmoak bektore errealekin dihardu. Edonola ere, problema konbinatorialak ebazteko PSO bertsiak aurki daitezke.

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

Ekuazioaren azken terminoak partikulen arteko elkarekintzak simulatzen ditu. Horretarako, partikuelen ingurune-egitura definitu behar da. Alabaina, 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 erabil daitezke (eraztunak, izarrak, toroideak, etab.).

Goian dagoen ekuazioa erabiltzen bada, abiadurak dibergitzeko joera izango du, alegia, abiaduraren modulua gero eta handiagoa izango da. Arazo hau ekiditeko kalkulatutako abiadurari muga bat ezartzen zaio.

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* ($\mathbf{p}_i$) eta *global best* ($\mathbf{p}_g$) eguneratu behar dira. Pasu guzti hauek 2.2 algoritmoan biltzen dira.

Algoritmo hau `metaheuR` paketeko `basic.pso` funtzioan inplementaturk dago. Erabilera erakusteko, optimizazio numerikoan *benchmark* gisa erabiltzen den Rosenbrock funtzioa erabiliko dugu; problema sortzeko `rosenbrock.problem` funtzioa erabili behar dugu.

```
> n <- 10
> eval <- rosenbrock.problem(size = n)$evaluate
```

```
## Error in eval(expr, envir, enclos):  no se pudo encontrar la funciÃşn "rosenbrock.problem"
```

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

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

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

```
> args$evaluate   <- eval
> args$resources  <- cresource(time=10)
>
> res.pso <- do.call(basic.pso , args)
```

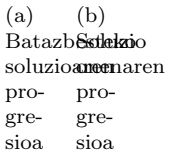
PSO algoritmoa

```
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

```
## Error in do.call(basic.pso, args):  objeto 'basic.pso' no encontrado

> plot.progress(res.pso , x = 'iterations' , y = 'best') + labs(y="Best Solution")

## Error in plot.progress(res.pso, x = "iterations", y = "best"):  error in evaluating the argument
'result' in selecting a method for function 'plot.progress':  Error:  objeto 'res.pso' no encontrado

> plot.progress(res.pso , x = 'iterations' ) + labs(y="Average Solution")

## Error in plot.progress(res.pso, x = "iterations"):  error in evaluating the argument 'result'
in selecting a method for function 'plot.progress':  Error:  objeto 'res.pso' no encontrado
```

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



Irudia 15: PSO algoritmoaren progresioa Rosenbrock probleman

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. Colorni. 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] J. Kennedy and R. C. Eberhart. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 1942–1948, 1995.

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

[9] 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.

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