Práctica 10: Algoritmo Genético

3175

9 de abril de 2019

1. Introducción

Las fuerzas entre partículas son muchas y variadas, entre las más importantes están las fuerzas electroestáticas proporcionadas por las cargas que éstas poseen la cual determina si se atraen o se repelen y la fuerza de gravedad que afecta a todo aquello que tiene una masa o inclusive una masa aparente como lo es en el caso de algunas partículas como los electrones [**?**].

2. Objetivo

Determinar usando distintas reglas de asignación de valores y pesos la manera mas eficiente de reslover, el algoritmo genético con la mejor selección de valores dentro de un rango de pesos permitido.

3. Metodología

Usando de base el código proporcionado[**?**], se obtiene la selección con la cantidad mas alta de la suma de valores que tengan peso permitido, por medio de la función *knapsack*, posteriormente se modifican las funciones para generar los valores de los pesos y valores: la primera con los valores asignados independientemente uno del otro, la segunda asignado independientemente pero correlacionado con el peso del objeto, esto es, entre mas pesado mas valioso, finalmente la ultima instancia es con los valores inversamente relacionados a los pesos, se verifica la distribución de los valores mediante gráficas de la figura 1.

Mediante el código:

```
#no correlacionado
generador.pesos <- function(cuantos, min, max) {
  return(sort(round(normalizar(rnorm(cuantos)) * (max - min) + min)))
}

generador.valores <- function(pesos, min, max) {
  n <- length(pesos)
  valores <- double()
  for (i in 1:n) {
    media <- pesos[n]
    desv <- runif(1)
    valores <- c(valores, rnorm(1))
  }
  valores <- normalizar(valores) * (max - min) + min
  return(valores)
```

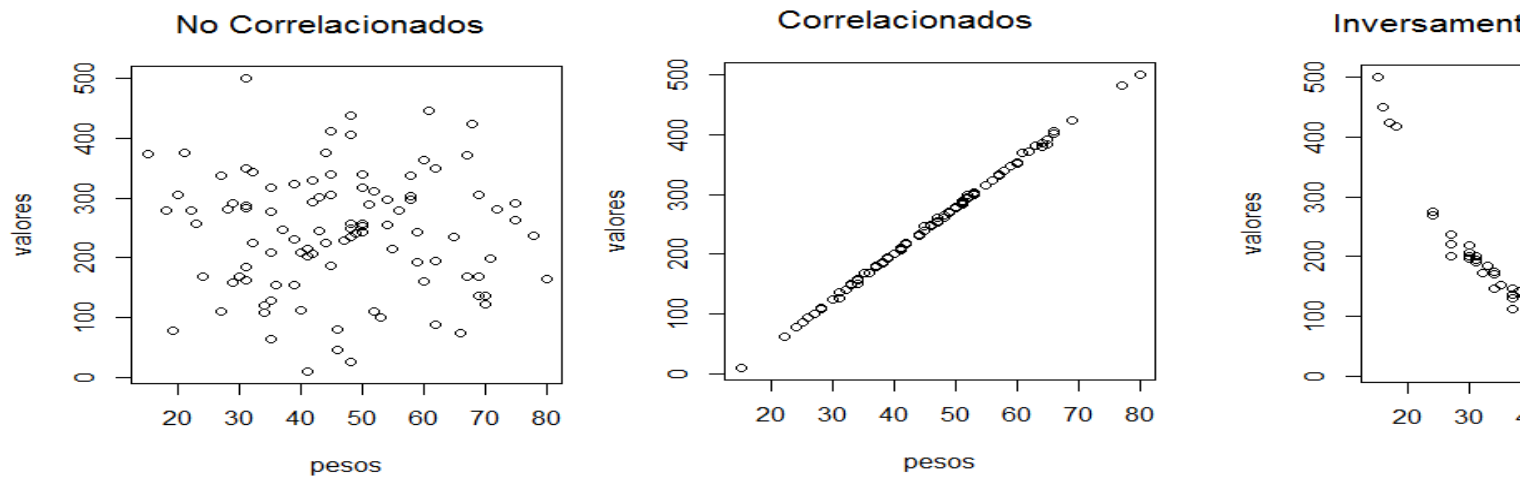


Figura 1: Distribición de valores y pesos de los objetos

```
16  }
17  #CORRELACIONADO
18  generador.pesos <- function(cuantos, min, max) {
19    return(sort(round(normalizar(rnorm(cuantos)) * (max - min) + min)))
20  }
21
22  generador.valores <- function(pesos, min, max) {
23    n <- length(pesos)
24    valores <- double()
25    for (i in 1:n) {
26      media <- pesos[n]
27      desv <- runif(1)
28      valores <- c(valores, (rnorm(1, media, desv)* (pesos[i]))+(rnorm(1)/10))
29    }
30    valores <- normalizar(valores) * (max - min) + min
31    return(valores)
32  }
33      #Inversamente CORRELACIONADO
34    generador.pesos <- function(cuantos, min, max) {
35      return(sort(round(normalizar(rnorm(cuantos)) * (max - min) + min)))
36    }
37
38    generador.valores <- function(pesos, min, max) {
39      n <- length(pesos)
40      valores <- double()
41      for (i in 1:n) {
42        media <- pesos[n]
43        desv <- runif(1)
44        valores <- c(valores, (rnorm(1, media, desv) / (pesos[i]))+(rnorm(1)/10))
45      }
46      valores <- normalizar(valores) * (max - min) + min
47      return(valores)
48    }
```

Finalmente se realiza la ejecición en paralelo del algoritmo genético y de la función *knapsack*

```
library(parallel)
cluster <- makeCluster(detectCores() - 1)
clusterExport(cluster, c("mutar", "mutacion", "reproduccion", "normalizar", "reproducete", "obj
tamao <-seq(50, 100, by= 25)
for (n in tamao){
    inicio1 <-Sys.time()
    pesos <- generador.pesos(tamao, 15, 80)
    valores <- generador.valores(pesos, 10, 500)
    capacidad <- round(sum(pesos) * 0.65)
    print(n)
    optimo <- knapsack(capacidad, pesos, valores)
    final1 <- Sys.time()
    #termina optimo
    inicial1 <- Sys.time()
    init <- 200
    p <- poblacion.inicial(n, init)
    tam <- dim(p)[1]
    assert(tam == init)
    pm <- 0.05
    rep <- 50
    tmax <- 50
    mejores <- double()
    clusterExport(cluster, c("n", "pm", "valores", "pesos", "capacidad", "objetivo"))
    for (iter in 1:tmax) {
      p$obj <- NULL
      p$fact <- NULL
      clusterExport(cluster, "p")
      mute <- parSapply(cluster, 1:tam, mutar)
      for(i in 1:length(mute)){
        if(!is.null(mute[[i]])){
          p <- rbind(p, mute[[i]])
        }
      }
        clusterExport(cluster, c("tam", "p"))
      S <- parSapply(cluster, 1:rep, reproducete) # una cantidad fija de reproducciones

      for (agrego in 1:length(S)) {
        p <- rbind(p, S[[agrego]])
      }
      tam <- dim(p)[1]
      obj <- double()
      fact <- integer()
      clusterExport(cluster, c("p", "tam", "factibilidad", "objet"))
      obj <-  parSapply(cluster, 1:tam, objet)
      fact <- parSapply(cluster, 1:tam, factibilidad)
      p <- cbind(p, obj)
      p <- cbind(p, fact)
      mantener <- order(-p[, (n + 2)], -p[, (n + 1)])[1:init]
      p <- p[mantener,]
      tam <- dim(p)[1]
      assert(tam == init)
      factibles <- p[p$fact == TRUE,]
```

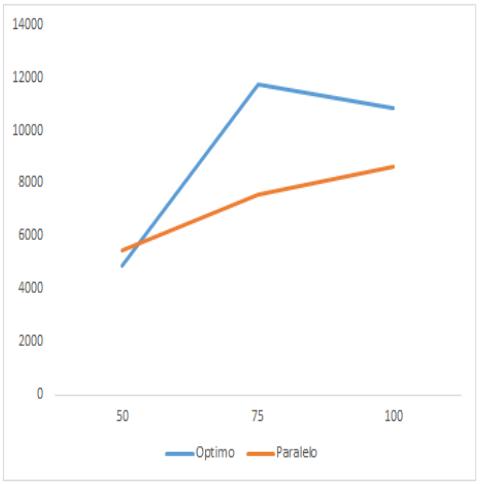
```
53          mejor <- max(factibles$obj)
54          mejores <- c(mejores, mejor)
55        }
56        termino1<- Sys.time()
```

4. Resultados

Los resultados en las figuras 2,3 y 4 muestran que las asignaciones de los valores si influyen en la cantidad de tiempo que le toma resolver el algoritmo genético .

5. Conclusiones

Se puede determinar que los valores determinan en gran medida el tiempo que toma resolver el algoritmo genético y éste mejora contra la funcion *knapsack* conforme aumentan los valores.



Gráfica de valores no correlacionados

Referencias

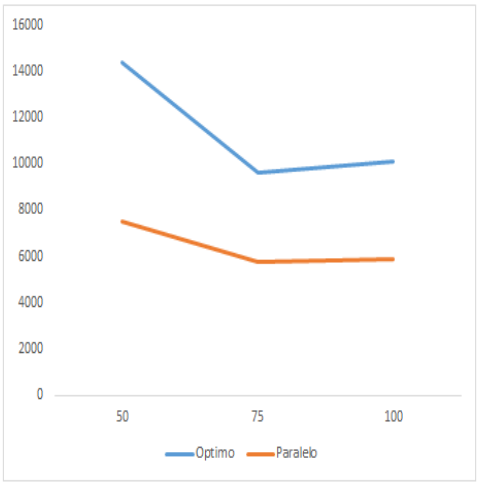


Figura 2: Gráfica valores correlacionados

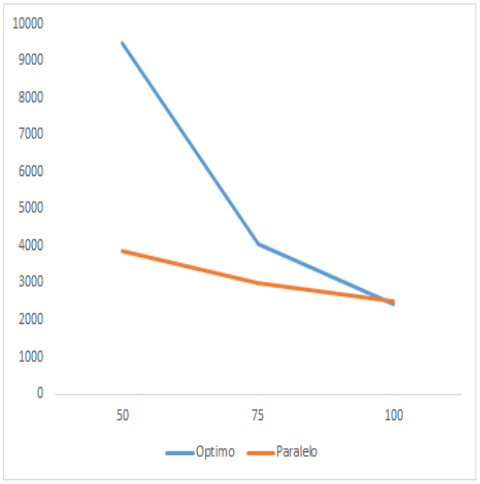


Figura 3: Gráfica de valores inversamente correlacionados