Práctica3 . Curso 2021-2022

Búsqueda local.

1. Objetivo de la práctica

El objetivo de la práctica es familiarizarse con el código en java para resolver problemas de búsqueda local y problemas de optimización. Utilizaremos el código del paquete aima.core.search.local. Tu trabajo consistirá en familiarizarte con los algoritmos **Hill-Climbing**, **SimulatedAnnealing** y **GeneticAlgorithm** utilizando el problema de las 8-reinas. Puedes partir del código en la clase **NQueensDemo** del paquete aima.gui.demo.search en el que se muestra, entre otros, la resolución del problema de las 8-reinas mediante algoritmos de Enfriamiento Simulado (**SimulatedAnnealing)**, Escalada (**Hill-Climbing**), y Algoritmos genéticos (**GeneticAlgorithm)**. La demo del algoritmo de búsqueda local **Hill-Climbing** no está correctamente implementada y deberás corregirla atendiendo a las notas de este guion

2. Tareas

HillClimbingSearch / Escalada

1. Implementa el método nQueensHillClimbingSearch_Statistics(int numExperiments) que realice numExperimets veces la búsqueda Hill-Climbing y muestre el porcentaje de éxitos, fallos, media de pasos al fallar y media de pasos en éxito. Realiza 10000 experimentos a partir de estados iniciales aleatorios no repetidos de las 8-reinas con la búsqueda **HillClimbingSearch.** Copia en la memoria de la práctica la traza de ejecución con los resultados:

```
NQueens HillClimbing con 10000 estados iniciales diferentes -->
Fallos:**.**
Coste medio fallos:*.**
Exitos:**.**
Coste medio Exitos:*.**
```

2. Implementa el método nQueensRandomRestartHillClimbing() que reinicia el estado inicial hasta que se obtiene el éxito. Muestra número de reintentos, solución y estadísticas. Copia en la memoria de la práctica la traza de ejecución con los resultados:

```
Search Outcome=SOLUTION_FOUND
Final State=
-----Q--
-Q------
------Q-
Q-------
---Q----
-------Q
----Q---
--Q-----
```

```
Numero de intentos:*
Fallos:*
Coste medio fallos:*
Coste éxito:*
Coste medio éxito:*
```

Nota 1: En la clase `NQueensDemo` se muestra un ejemplo de uso de la clase `HillClimbingSearch`. Pero ten cuidado porque usan las funciones que de forma incremental añaden una reina en cada columna a partir de un tablero vacío : *`NQueensFunctionFactory.getIActionsFunction()`*. Para hacer una búsqueda con `HillClimbingSearch` debemos partir de un estado completo y cambiar la posición de las reinas. Deberás utilizar las funciones en `NQueensFunctionFactory.`*`getCActionsFunction`*`()`.

Nota 2: Ten cuidado con como representa los tableros en el programa. A continuación, tienes ejemplos de matrices que representan los tableros correctamente y como los visualiza.

```
/**
 * X--> increases left to right with zero based index
 * Y--> increases top to bottom with zero based index | |
 * V [colum,left to right x][row, top-down y]
 */
  static NQueensBoard EjemploSolucion = new NQueensBoard(
                        // f1 f2 f3 f4 f5 f6 f7 f8
               new int[][] {{0, 0, 0, 0, 1, 0, 0, 0},   //c1
                            {0, 0, 1, 0, 0, 0, 0, 0},   //c2
                            {1, 0, 0, 0, 0, 0, 0, 0},   //c3
                            {0, 0, 0, 0, 0, 0, 1, 0},   //c4
                            {0, 1, 0, 0, 0, 0, 0, 0},   //c5
                            {0, 0, 0, 0, 0, 0, 0, 1},   //c6
                            {0, 0, 0, 0, 0, 1, 0, 0},   //c7
                            {0, 0, 0, 1, 0, 0, 0, 0}}); //c8
/* ejemplo solución representado
 *
--Q-----  3
----Q---  5
-Q------  2
-------Q  8
Q-------  1
------Q-  7
---Q----  4
-----Q--  6
*/

static NQueensBoard EjemploEstadoCompleto = new NQueensBoard(
          //f1 f2 f3 f4 f5 f6 f7 f8
new int[][] {{1, 0, 0, 0, 0, 0, 0, 0},   //c1
               {0, 0, 1, 0, 0, 0, 0, 0},   //c2
               {1, 0, 0, 0, 0, 0, 0, 0},   //c3
               {0, 0, 0, 0, 0, 0, 1, 0},   //c4
               {0, 1, 0, 0, 0, 0, 0, 0},   //c5
               {0, 0, 0, 0, 0, 0, 0, 1},   //c6
               {0, 0, 0, 0, 0, 1, 0, 0},   //c7
               {0, 0, 0, 1, 0, 0, 0, 0}}); //c8

/* EjemploEstadoCompleto
Q-Q-----
----Q---
-Q------
-------Q
--------
------Q-
---Q----
-----Q--
*/
```

SimulatedAnnealing / Enfriamiento Simulado

El algoritmo Simulated annealing (SA) genera un estado aleatorio sucesor: (i) si el coste del nuevo estado es mejor, el cambio se acepta; (ii) por el contrario, si se produce un incremento en la función de evaluación, el cambio será aceptado con una cierta probabilidad (Ver el pseudo-código del algoritmo). El SA acepta con mayor probabilidad estados peores al principio, pero según se va "enfriando", la probabilidad de aceptar estados peores es menor. Esta forma de aceptar con alta probabilidad estados peores al principio y con menos probabilidad al final permite evitar óptimos locales. El nombre e inspiración viene del proceso de templado ("*annealing*" en inglés) en metalurgia, una técnica que consiste en calentar y luego enfriar controladamente un metal para aumentar el tamaño de sus cristales y reducir sus defectos. El calor causa que los átomos se salgan de sus posiciones iniciales (se encuentran en un mínimo local de energía) y se muevan aleatoriamente; el enfriamiento lento les da mayores probabilidades de encontrar configuraciones con menor energía que la inicial.

Pseudo-código del Algoritmo Simulated Annealing

```
function Simulated_Annealing(T0, k, Tfinal)
T <- T0
Sactual <- Solución inicial
for T=T0 to Tfinal do
      if T = 0 then return Sactual
      else Snuevo <- Nueva solución aleatoria
      Inc(C) <- Coste(Snuevo)-Coste(Sactual)
      if (Inc(C) > 0) then Sactual <- Snuevo
      else Sactual <- Snuevo con probabilidad Exp(Inc(C) /F(T))
                                    donde F(T) = k Exp(-delta*T)
   T <- Evolucion(T)
```

donde, Inc(C) = [Coste(nuevo estado) − Coste(estado anterior)]

- Cuando Inc(C) < 0:la probabilidad de cambio tiene una P [aceptación] = 1
- Cuando Inc(C) > 0: la probabilidad de cambio tiene una P[aceptación] = $\exp(\frac{Inc(c)}{F(T)})$

  donde F (T) = k · exp(−δ T)

Los **parámetros** del algoritmo significan lo siguiente: (i) los valores que toma la **variable T** se corresponden con el paso de iteración en la ejecución, (ii) el **parámetro k** determina la velocidad que tarda en comenzar a decrecer la temperatura, y (iii) el **parámetro δ** mide lo rápido que desciende. En las Figura 1 y Figura 2 se puede ver el comportamiento del algoritmo para Inc(C) = 1, k= {10, 50, 500, 5000, 50000} manteniendo δ = 0,01, y δ= {0.0005, 0.001, 0.01, 0.1} manteniendo k = 10.

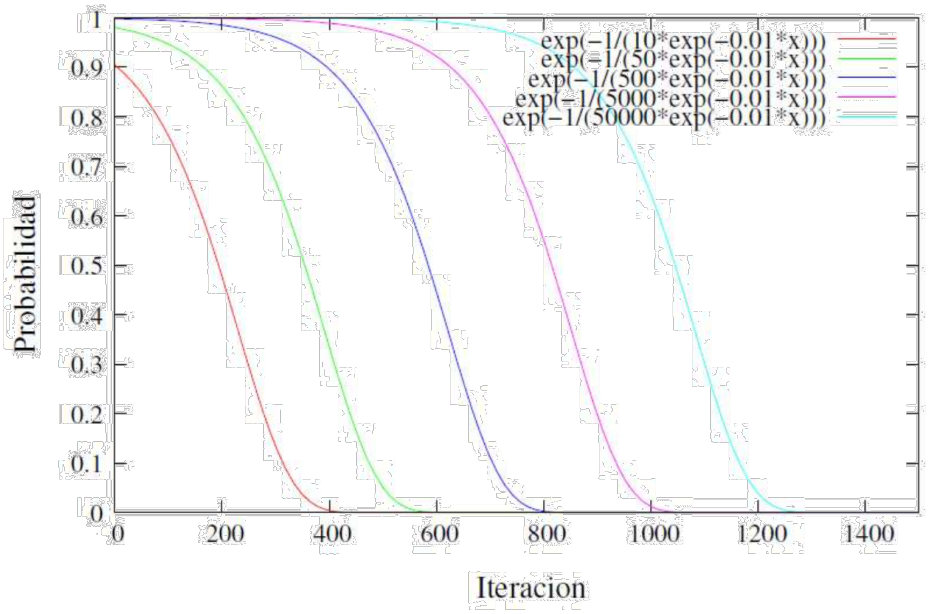

Figura 1: *Simulated annealing*: variación en función de k

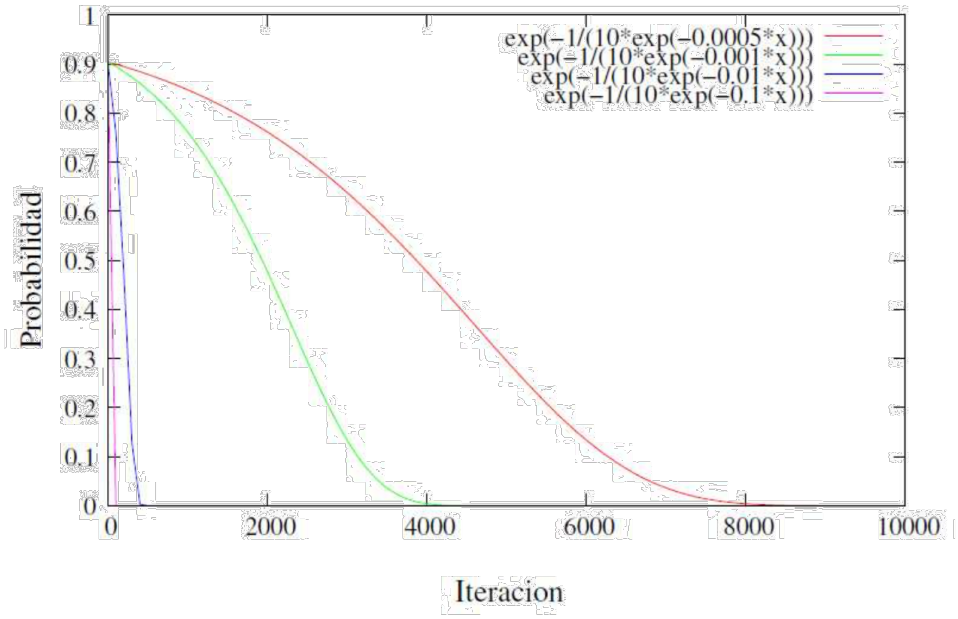

Figura 2: *Simulated annealing*: variación en función de

La mayor dificultad del uso del algoritmo SA es el ajuste de los parámetros ya que la única indicación que tenemos son los resultados que obtenemos experimentalmente. Esto nos obliga a hacer diversas pruebas para ver cómo se comporta el algoritmo con el problema dado. En general, es importante guiarnos por lo siguiente:

- Al aumentar el valor de k se aleja el punto en el que la P [aceptación] se anula, y al disminuir el valor de δ se alarga el tiempo necesario para converger
- Fijados unos valores de (k, δ), el número de iteraciones mínimo que necesitamos para T, será el de aquella iteración en que la P [aceptación] se anule. A partir de esta iteración, el algoritmo sólo aceptará mejores soluciones: "cómo de grande debe ser T es una cuestión a la que sólo se puede responder experimentalmente.

3. Implementa `nQueensSimulatedAnnealing_Statistics (int numExperiments)` que realice numExperimets y muestre el porcentaje de éxitos, fallos, media de pasos al fallar y media de pasos en éxito. Realiza 1000 experimentos a partir de estados iniciales aleatorios no repetidos de las 8-reinas con la búsqueda `SimulatedAnnealingSearch`. Utiliza los parámetros que creas son más adecuado y muéstralos. Copia en la memoria de la práctica la traza de ejecución con los resultado:

```
NQueensDemo Simulated Annealing con 1000 estados iniciales diferentes -->
Parámetros Scheduler: Scheduler (*,*,*);

Fallos:**.**
Coste medio fallos:*.**
Exitos:**.**
Coste medio Exitos:*.**
```

4. Implementa el método `nQueensHillSimulatedAnnealingRestart()` que reinicia el estado inicial hasta que se obtiene el éxito con la búsqueda `SimulatedAnnealingSearch`. Muestra número de reintentos, solución y estadísticas. Copia en la memoria de la práctica la traza de ejecución con los resultados, por ejemplo:

```
Search Outcome=SOLUTION_FOUND
Final State=
--Q-----
-----Q--
-------Q
Q-------
----Q---
------Q-
-Q------
---Q----

Numero de intentos:*.*
Fallos:*.*
Coste  Éxito:*.*
```

GeneticAlgorithm / Algoritmos Genéticos

5. Realiza experimentos con la búsqueda `GeneticAlgorithm` y ajusta los parámetros de población inicial y probabilidad de mutación hasta que consideres dan los mejores resultados. Implementa el método `nQueensGeneticAlgorithmSearch()` con los parámetros elegidos . Copia en la memoria de la práctica la traza de ejecución con los resultados:

```
GeneticAlgorithm
Parámetros iniciales:       Población: *, Probabilidad mutación: *
 Mejor individuo=
 -----Q--
 ---Q----
 ------Q-
 Q-------
 -------Q
 -Q------
 ----Q---
 --Q-----

 Tamaño tablero      = 8
 Fitness             = 28.0
 Es objetivo         = true
 Tamaño de población = *
 Iteraciones         = *
 Tiempo              = *ms.
```

Entrega una clase `NQueensLocal` **con los métodos solicitados implementados** y la ejecución de estos algoritmos en el main**.** En la **memoria** que debes entregar, muestra los resultados y comentarios de los experimentos realizados. Puedes incluir cualquier clase o método adicional que hayas implementado para la realización del trabajo.