2019

Memoria TP6 – Parte 1

INTELIGENCIA ARTIFICIAL 2019-2020

PEDRO TAMARGO ALLUÉ (758267)

Contenido

Parte 1: Algoritmos de búsqueda local 2
Algoritmo Hill Climbing 2
nQueensHillClimbingSearch_Statistics 2
nQueensRandomReestartHillClimbing 2
Algoritmo Simulated Annealing 3
Tabla comparativa del porcentaje de éxito con distintos parámetros del Scheduler 3
nQueensSimulatedAnnealingSearch_Statistics 3
nQueensHillSimulatedAnnealingRestart 4
Algoritmo Genetic 4
Tabla comparativa midiendo el tiempo y las iteraciones y variando la población 5
Tabla comparativa midiendo el tiempo y las iteraciones y variando la probabilidad de mutación 5
nQueenGeneticAlgorithmSearch 6
Parte 2: Resolución de Sudokus mediante propagación de restricciones. 6
Ejecución de la búsqueda 7
Parte 3: Propagación de restricciones y búsqueda local 7
Ejecución del problema 8

Parte 1: Algoritmos de búsqueda local

Para la realización de esta tarea se han elegido los algoritmos: Hill Climbing, Simulated Annealing y Genetic Algorithm para resolver el problema de las N-reinas (N = 8).

Algoritmo Hill Climbing

Este algoritmo busca optimizar la solución ya que, partiendo de un estado inicial completo, busca el sucesor que maximice el resultado. Es un algoritmo de búsqueda local ya que solamente se fija en sus sucesores para encontrar el resultado óptimo.

nQueensHillClimbingSearch_Statistics

```
NQueens HillClimbing con 10000 estados iniciales diferentes
Fallos: 0.86
Coste medio fallos: 2.73
Exitos: 0.13
Coste medio Exitos: 3.72
```

Captura de pantalla del resultado de la ejecución del algoritmo Hill Climbing, partiendo de 10000 estados iniciales distintos.

nQueensRandomReestartHillClimbing

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


Numero de intentos:4
Fallos:3
Coste medio de fallos: 3
Coste de exito:5
Coste medio de exito:5
-------------------------
```

Captura de pantalla del resultado de la ejecución del algoritmo Hill Climbing con reinicio, cuando no encuentra una solución, busca un nuevo estado inicial y reinicia el experimento.

Algoritmo Simulated Annealing

Este algoritmo es un símil de la industria metalúrgica, donde, durante los procesos de templado (de aceros), la aleación se enfría drásticamente para maximizar el empaquetamiento de átomos de carbono en la estructura cristalina, lo cual implica mayor resistencia y dureza, así como fragilidad.

En este algoritmo, se utiliza una función $F(T) = k * e^{-\delta T}$, siendo δ, k constantes del problema. Para determinar si se pasa de un cierto estado a otro sucesor se utiliza una función de probabilidad dependiente del incremento de los costes entre el nuevo estado y el actual, $Inc(C) = Coste(nuevo) - Coste(actual)$. Por lo tanto, definimos la función de probabilidad como: $P[aceptacion] = e^{\frac{Inc(C)}{F(T)}}$, si $Inc(C) > 0$, siendo T la variable que se corresponde con el paso de iteración en la ejecución, o $P[aceptacion] = 1$, si $Inc(C) < 0$.

Tabla comparativa del porcentaje de éxito con distintos parámetros del Scheduler

NQUEENS Simulated Annealing			(midiendo % éxito | limit = 2000)		
∂ \ k	**500**	**550**	**600**	**650**	**700**
0,005	60%	62%	62%	60%	57%
0,01	82%	85%	86%	84%	86%
0,05	91%	91%	90%	90%	90%
0,1	92%	90%	91%	90%	90%
0,5	88%	85%	85%	86%	87%

Para medir el éxito en esta tabla se ha utilizado una *"temperatura"* límite de 2000.

Para la realización de las pruebas se han utilizado los parámetros:

$$\partial = 0.05, k = 650, T_{Limite} = 2000$$

nQueensSimulatedAnnealingSearch_Statistics

```
NQueens Simulated Annealing con 1000 estados iniciales diferentes
Parametros Scheduler: Scheduler(650, 0,050000, 2000)
Fallos: 0.10
Coste medio fallos: 210.
Exitos: 0.89
Coste medio Exitos: 164,4372
```

Captura de pantalla del resultado de la ejecución del algoritmo Simulated Annealing, partiendo de 1000 estados iniciales distintos.

nQueensHillSimulatedAnnealingRestart

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


Numero de intentos:1
Fallos:0
Coste de exito:207
```

Captura de pantalla del resultado de la ejecución del algoritmo Simulated Annealing, cuando no encuentra una solución, busca un nuevo estado inicial y reinicia el experimento.

Algoritmo Genetic

Este algoritmo se basa en las mutaciones y cruces entre individuos dentro de una población. Su funcionamiento consiste en que, dada una población inicial y una probabilidad de mutación, seleccionamos a los individuos, usando una $Fitness\ Function$ y una probabilidad (los individuos cuyo resultado después de aplicar la $Fitness\ Function$ sea mayor, tendrán más probabilidades de ser elegidos), que transmitirán sus genes a la siguiente generación. Tras seleccionar a los individuos, procedemos a realizar combinaciones, para ello, seleccionamos (de forma aleatoria) el punto de cruce ($crossover\ point$), dividiendo el individuo en 2 partes. Después procedemos a realizar la mutación, creando dos nuevos individuos combinando las 4 partes de los individuos que se han seleccionado en base a la $Fitness\ Function$. En algunos casos pueden ocurrir mutaciones (dependiendo de la probabilidad de mutación), esto implica que alguno de los elementos generados en el paso anterior, cambie alguno de sus “genes” para mantener la diversidad de la población.

Tabla comparativa midiendo el tiempo y las iteraciones y variando la población

Población	Tiempo (ms)	Iteraciones
10	4869	14452
20	3878	3020
30	3614	1222
40	3857	732
50	3728	449
60	5634	475
70	8795	531
80	5723	262
90	7713	290
100	6464	335

Para la realización de las mediciones, se ha establecido una probabilidad de mutación de 0,15

Tabla comparativa midiendo el tiempo y las iteraciones y variando la probabilidad de mutación

Probabilidad	Tiempo (ms)	Iteraciones
0,15	2095	1214
0,25	3229	1846
0,35	3626	2116
0,45	2776	1522
0,55	3743	2087
0,65	4835	2541
0,75	3946	2071
0,85	5319	2756
0,95	5726	3276
1,05	2071	1024
1,15	5343	3117

Para la realización de las mediciones, se ha establecido una población de 30 elementos, ya que, como resultado de la prueba anterior, ha mostrado los mejores resultados.

nQueenGeneticAlgorithmSearch

```
GeneticAlgorithm
Parametros iniciales:    Poblacion:30, Probabilidad mutacion:0,150000)

Goal Test Best Individual=
---Q----
-----Q--
-------Q
--Q-----
Q-------
------Q-
----Q---
-Q------

Board Size              = 8
Fitness                 = 28.0
Es objetivo             = true
Tamaño de la poblacion  = 30
Iteraciones             = 324
Tiempo                  = 1033ms.
```

Captura de pantalla del resultado de la ejecución del algoritmo Genetic Algorithm.

Parte 2: Resolución de Sudokus mediante propagación de restricciones.

Las clases a realizar en esta parte son: $SudokuVariable, SudokuConstraint\ y\ SudokuApp$. En $SudokuVariable$ hereda de $Variable$ y representa una celda del sudoku, tiene $valores\ x, y, value$. La clase $SudokuConstraint$ hereda de $Constraint$ e impide que una asignación de variables sea igual a otra (como $NotEqualConstraint$). La clase $SudokuApp$ contiene el código del método $main$, en este se leen los ficheros de sudokus y se resuelven.

Ejecución de la búsqueda

```
---------
....7..2.
8.......6
.1.2.5...
9.54....8
.........
3....85.1
...3.2.8.
4.......9
.7..6....
SUDOKU INCOMPLETO - RESOLVIENDO
{Cell at [0][7]=2, Cell at [8][1]=7, Cell at [0][
594876123
823914756
617235894
965421378
781653942
342798561
159342687
436587219
278169435
Sudoku solucionado correctamente
+++++++++
Se han resuelto 156 sudokus
```

Captura de pantalla de la resolución de los 156 Sudokus, ubicados en los 3 ficheros proporcionados junto con el enunciado del trabajo.

Parte 3: Propagación de restricciones y búsqueda local.

Para la realización de esta parte se han reutilizado las clases creadas en la parte 2, con modificaciones.

La clase $NQueensVariable$ hereda de $Variable$, y contiene 2 enteros, que simbolizan la fila y la columna.

La clase $NQueensConstraint$ hereda de $Constraint$, y comprueba que una asignación de variables cumple con las restricciones (no son la misma reina, no están en la misma fila y no están en la misma diagonal).

La clase $NQueensMinConflictApp$, contiene el método $main$ que utiliza $MinConflictsStrategy$ para resolver el tablero.

Ejecución del problema

```
Limite: 10 -- Se han resuelto 4 tableros en 0.0 segundos
Limite: 20 -- Se han resuelto 14 tableros en 0.0 segundos
Limite: 30 -- Se han resuelto 22 tableros en 0.0 segundos
Limite: 40 -- Se han resuelto 23 tableros en 0.0 segundos
Limite: 50 -- Se han resuelto 33 tableros en 0.0 segundos
Limite: 60 -- Se han resuelto 32 tableros en 0.0 segundos
Limite: 70 -- Se han resuelto 33 tableros en 0.0 segundos
Limite: 80 -- Se han resuelto 37 tableros en 0.0 segundos
Limite: 90 -- Se han resuelto 41 tableros en 0.0 segundos
Limite: 100 -- Se han resuelto 44 tableros en 0.0 segundos
Limite: 110 -- Se han resuelto 40 tableros en 0.0 segundos
Limite: 120 -- Se han resuelto 38 tableros en 0.0 segundos
Limite: 130 -- Se han resuelto 35 tableros en 0.0 segundos
Limite: 140 -- Se han resuelto 41 tableros en 0.0 segundos
Limite: 150 -- Se han resuelto 42 tableros en 0.0 segundos
Limite: 160 -- Se han resuelto 43 tableros en 0.0 segundos
Limite: 170 -- Se han resuelto 40 tableros en 0.0 segundos
Limite: 180 -- Se han resuelto 41 tableros en 0.0 segundos
Limite: 190 -- Se han resuelto 45 tableros en 0.0 segundos
Limite: 200 -- Se han resuelto 45 tableros en 0.0 segundos
Limite: 210 -- Se han resuelto 44 tableros en 0.0 segundos
Limite: 220 -- Se han resuelto 46 tableros en 0.0 segundos
Limite: 230 -- Se han resuelto 45 tableros en 0.0 segundos
Limite: 240 -- Se han resuelto 47 tableros en 0.0 segundos
Limite: 250 -- Se han resuelto 46 tableros en 0.0 segundos
```

Captura de pantalla del resultado de la ejecución de la búsqueda CSP con la estrategia *"MinConflicts"*.

Podemos observar como resultado de la ejecución, que con un límite de 250 pasos podemos obtener la solución para un 92% de los casos.