PRÁCTICA 1 SSII

Curso 2020/2021

Algoritmo Genético y Búsqueda Primero en Profundidad

Jose Luis Martínez Perea
Curso 3º Grupo 2.1
Fecha de entrega:20/12/2020

ÍNDICE

Algoritmo Genético

Para entender lo que es un algoritmo genético primero debemos saber que es un algoritmo. Un algoritmo se basa en una serie de pasos/instrucciones que nos llevan a resolver un problema en específico. Los algoritmos genéticos son conocidos así por que dichos algoritmos partiendo de una población inicial (conjunto de individuos generados aleatoriamente), les aplica una serie de cambios o acciones aleatorias (mutación/cruce) basándose en un criterio, de forma que los individuos de la población inicial vayan evolucionando, posteriormente se decide que individuos son los más aptos para sobrevivir, esta tarea es de los métodos de selección que en este caso son ruleta y torneo.

Para comenzar un algoritmo genético necesita operar sobre un conjunto de estados que se generan aleatoriamente (pero con un criterio concreto para cada problema), cada estado (individuo o cromosoma), esta codificado de una manera concreta. Cada elemento de un estado (individuo o cromosoma) se denomina gen. Finalmente tenemos el valor heurístico de un estado (individuo o cromosoma) que también podemos denominar fitness (idoneidad o fenotipo), este valor nos indica lo bueno que es un estado por lo que es utilizado para cribar los individuos de una población dada.

Para la codificación de este problema se hace uso de enteros, los cromosomas serán un array de tantas posiciones como columnas tenga el tablero de juego, cada celda del array contendrá un número que indicará la posición en la que está la reina dentro de esa columna (irán numeradas de 1 a N siendo este valor el número de columnas del tablero).

Ahora pasaré a explicar los operadores genéticos que se emplean en este algoritmo:

- Selección: este operador es el encargado de decidir qué individuos van a ser capaces de reproducirse y que individuos van a ser desechados del problema, en este caso particular se emplea GATournamentSelector, este método se encarga de cribar los individuos teniendo en cuenta el valor fitness por lo que se asegura de que solo los mejores individuos van a poder reproducirse.
- Cruce: este operador se encarga de recombinar los genes de los individuos que pueden reproducirse para dar lugar a las futuras generaciones, en este algoritmo se utiliza OnePointCrossover, este método realiza un corte del cromosoma en una parte concreta, por lo que se generan dos partes cada una de un progenitor que serán intercambiadas entre ambos, generando así diversidad entre los individuos.
- Mutación: este operador es el encargado de hacer variaciones en los valores de algunos genes de un individuo concreto, en este algoritmo se emplea FlipMutator, que cambia el valor de un gen dentro concreto dentro del rango de valores que tiene asignados dicho gen.
- Terminación: este operador establece la condición de parada del algoritmo, en este caso se le pasa al algoritmo el número de generaciones que queremos que lleve a cabo por lo tanto ese valor es el que determina cuando el algoritmo no debe seguir generando individuos.

Tabla con valores de la función fitness.

Nº Reinas	pm	F	pm	F
8	0,0125	0	0,025	0
9	0,0125	1	0,025	0
10	0,0125	1	0,025	0
11	0,0125	0	0,025	0
12	0,0125	1	0,025	0
13	0,0125	1	0,025	0
14	0,0125	0	0,025	0
15	0,0125	0	0,025	0
16	0,0125	0	0,025	0
17	0,0125	0	0,025	0
18	0,0125	0	0,025	0
19	0,0125	0	0,025	0
20	0,0125	0	0,025	0
21	0,0125	0	0,025	0
22	0,0125	0	0,025	0
23	0,0125	0	0,025	0
24	0,0125	1	0,025	0
25	0,0125	0	0,025	0
26	0,0125	0	0,025	0
27	0,0125	0	0,025	0
28	0,0125	0	0,025	0
29	0,0125	0	0,025	0
30	0,0125	0	0,025	0
31	0,0125	0	0,025	0
32	0,0125	0	0,025	0
33	0,0125	0	0,025	0
34	0,0125	0	0,025	0
35	0,0125	0	0,025	0
36	0,0125	0	0,025	0
37	0,0125	0	0,025	0
38	0,0125	0	0,025	0
39	0,0125	0	0,025	0
40	0,0125	0	0,025	0
41	0,0125	0	0,025	2
42	0,0125	0	0,025	3
43	0,0125	0	0,025	3
44	0,0125	0	0,025	4
45	0,0125	0	0,025	5
46	0,0125	0	0,025	3
47	0,0125	0	0,025	4
48	0,0125	0	0,025	2
49	0,0125	0	0,025	4
50	0,0125	0	0,025	3

Datos adicionales

- Nº generaciones = 1000
- Método de selección = GATournamentSelector
- Operador de cruce = OnePointCrossover
- Operador de mutación = FlipMutator
- Tamaño de la población (TP)= 100
- pc = 0.8

Por lo que los únicos datos que varían a lo largo de la ejecución son el número de reinas y la probabilidad de mutación (pm), tal y como se puede apreciar en la tabla.

Análisis de pruebas de ajuste

A continuación, adjunto una tabla que recoge los valores de n reinas para los que se obtienes 0 jaques (valor fitness 0).

	pm	TP = 100, Nº generaciones = 1000
pc = 0,8	0,0125	nreinas = {8},{11}, {14-23}, {25-50}
pc = 0,8	0,025	nreinas = {8-40}

Como se puede apreciar con pm = 0,0125, a partir de 25 reinas no se obtiene ningún jaque, mientras que para valores menores se alternarán valores de nº de reinas en la que se obtiene algún jaque y otras en las que no se obtendrá ninguno, pero este valor se estabiliza a 0 jaques a partir de nreinas = 25.

Con pm = 0,025 desde el caso inicial (8 reinas), hasta el caso nreinas = 40 no se produce ningún jaque (valor fitness que nos interesa). Por lo tanto, desde nreinas = 41 hasta nreinas = 50 siempre se obtiene como mínimo un jaque.
Cabe destacar que tanto para valor pm = 0,0125 como para valor de pm = 0,025, siempre se utiliza el mismo método de selección, el mismo operador de cruce y mutación como se indica en el apartado anterior.

Con el ajuste inicial que se ha llevado a cabo, para un tamaño de población de 8 a 50 reinas, debemos utilizar (para 1000 generaciones y tamaño de población 100), los operadores GATournamentSelector, FlipMutator y OnePointCrossover y una pc = 0,8.
Por lo tanto, a partir del ajuste inicial, podemos concluir que para valores pequeños de n reinas (desde 8 hasta 40), se puede usar pm = 0,025, pero para valores superiores, debemos decrementar el valor de pm para así obtener 0 jaques (valor fitness idóneo).
Posteriormente en la sección del manual de asignación se establecerán los valores óptimos para minimizar el uso de recursos de nuestro sistema.
NOTA: No se puede indagar más en el comportamiento del sistema porque los parámetros que empleamos para hacer este estudio son estáticos por lo que las conclusiones que tomamos son en base a la estimación que hacemos de esos datos.

Manual de asignación

A continuación, se incluirán una serie de recomendaciones para llevar a cabo la asignación:

SSII2.exe [nºreinas] [tamaño pobalción] [nºgeneraciones] [pc] [pm]

Como se puede apreciar se adjunta un esquema de la estructura que debe tener una ejecución del algoritmo genético en línea de comandos (en nuestro caso cmd SO Windows). Lo primero que debe aparecer es el nombre del ejecutable que contiene el algoritmo genético (en nuestro caso SSII2.exe), posteriormente se deben incluir los argumentos que el algoritmo genético utilizará a la hora de enfrentar el problema.

A continuación, se adjuntan los valores recomendados de los parámetros que requerirá el sistema para obtener buenos resultados (dichos valores han sido testeados y garantizan obtener buenos resultados en líneas generales).

Valor de n (número de reinas) para el parámetro 1º: [8-50] (valor recomendado)
Valor para el parámetro 2º: [100] (valor recomendado)
Valor para el parámetro 3º: [10000] (valor recomendado)
Valor para el parámetro 4º: [0.8] (valor recomendado)
Valor para el parámetro 5º:

- Si n = 8: 0,0125 (valor recomendado)
- Si $9 \geq n \leq 10$: 0,025 (valor recomendado)
- Si $11 \geq n \leq 13$: 0,025 (valor recomendado)
- Si $14 \geq n \leq 23$: 0,0125 (valor recomendado)
- Si n = 24: 0,025 (valor recomendado)
- Si $25 \geq n \leq 50$: 0,0125 (valor recomendado)

Como se puede apreciar siempre se priorizan los valores más pequeños de pm, puesto que cuanto menor sea la probabilidad de mutación, menos individuos van a ser afectados por esta y por lo tanto se necesitarán menos recursos (complejidad en tiempo y en espacio).

Análisis comparativo

En esta sección vamos a realizar una comparativa entre el algoritmo genético empleado en esta práctica y el algoritmo de búsqueda primero en profundidad que nos ha proporcionado el profesor de la asignatura, para ello tendremos en cuenta una serie de propiedades que nos permitirán realizar una comparativa entre estos algoritmos, dichas propiedades son las siguientes:

- Completitud: Un algoritmo cumplirá esta propiedad si es capaz de encontrar una solución a un problema dado (en caso de existir esta). Si no existe solución, puede que el algoritmo no pueda acabar (en caso de que no se cumpla la condición de parada) o que termine su ejecución sin haber podido encontrar una solución.
- Optimalidad: Un algoritmo será óptimo si puede encontrar la mejor solución posible en base a algún criterio o coste preestablecido.
- Complejidad en tiempo: Esta propiedad hace referencia al tiempo que el algoritmo necesita para encontrar una solución y acabar por lo tanto su ejecución.

- Complejidad en memoria: Esta propiedad hace referencia a la cantidad de espacio en memoria que el algoritmo requiere para hallar la solución.

Búsqueda Primero en Profundidad

i) Completitud

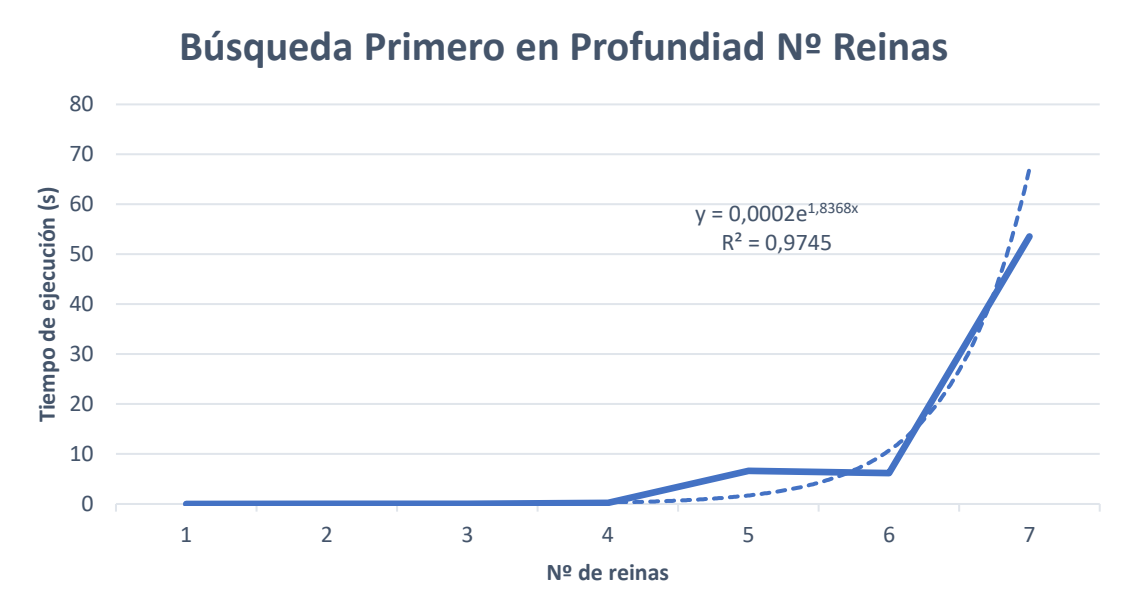
Podemos afirmar que este algoritmo siempre encuentra solución en caso de que exista ya que este algoritmo analiza todos los estados posibles del problema, por lo que si existe una solución acabará encontrándola por la propia naturaleza del algoritmo.

ii) Optimalidad

La solución encontrada es óptima ya que este algoritmo comprueba si TEST-OBJETIVO [problema] aplicado al ESTADO [nodo] es cierto entonces devolver SOLUCION (nodo). Esto nos garantiza que la solución que devuelve es la primera encontrada, puesto que analiza desde la raíz hasta el primer nodo que le ofrece una solución. (Esto asumiendo que al aplicar los operadores de búsqueda estos tendrán un coste uniforme).

iii) Complejidad en tiempo

A continuación, adjunto una tabla y un gráfico construido con los datos de dicha tabla, la línea de tendencia que ofrecía un mejor ajuste es la exponencial, si tomamos como unidad de medida de coste el número de nodos comprobados, se produciría un crecimiento exponencial en base al factor de ramificación y a la profundidad de la solución encontrada. (Coste $O(n^k)$ siendo n el factor de ramificación (número de ramificaciones por nodo) y k (la máxima profundidad del espacio de estados)).



Nº de reinas	5	6	7	8	9	10	11	12
Tiempo (s)	0,001	0,008	0,028	0,221	6,649	6,164	53,56	> 3600

iv) Complejidad en memoria

La complejidad en memoria del algoritmo BPP es O(n*k) puesto que la cantidad de nodos que se deben comprobar son n*k, siendo n el factor de ramificación y k la profundidad necesaria para alcanzar la solución (variando desde 1 hasta el número de reinas). Esto es así porque el algoritmo va comprobando cada rama del árbol individualmente y por lo tanto no debe tener en memoria todo el árbol.

Como n y k es igual al número de reinas se podría afirmar que el algoritmo BPP tiene una complejidad en memoria de $O(n^2)$.

Algoritmo Genético

i) Completitud

Podemos afirmar que este algoritmo es completo puesto que, siempre devuelve una solución, dicha solución variará en base a los parámetros de entrada y a la población inicial, pero podemos afirmar que siempre devolverá una solución al problema.

ii) Optimalidad

Los algoritmos genéticos generan soluciones, para poder comparar estas tenemos el valor fitness, como hemos podido apreciar en las pruebas de ajuste no siempre obteníamos valor fitness 0 (que indica que no hay jaques entre las reinas). El valor fitness del estado solución variará dependiendo de los parámetros del problema, de la población inicial y de otros factores. Por lo que no podemos afirmar que este algoritmo ofrezca siempre la solución óptima, pero haciendo uso del manual de asignación podemos obtener dicha solución bajo unas situaciones concretas.

iii) Complejidad en tiempo

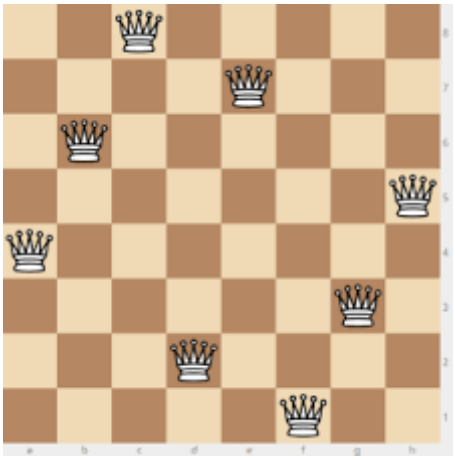
A continuación, adjunto una tabla con los tiempos obtenidos para cada una de las pruebas realizadas, esta tabla es homóloga a la adjuntada anteriormente pero no contiene el valor fitness sino el tiempo de ejecución de cada caso de prueba.
DATO: Los valores de número de generaciones, método de selección, operador de cruce, operador de mutación, tamaño de la población y pc son los mismos que se usaron anteriormente.

Nº Reinas	pm	Tiempo (s)	pm	Tiempo (s)
8	0,0125	0.003	0,025	0.002
9	0,0125	1.161	0,025	0.006
10	0,0125	1.297	0,025	0.005
11	0,0125	0.114	0,025	0.004
12	0,0125	1.56	0,025	0.839
13	0,0125	1.711	0,025	0.033
14	0,0125	0.011	0,025	0.096
15	0,0125	1.073	0,025	0.023
16	0,0125	0.29	0,025	0.153
17	0,0125	0.02	0,025	0.489
18	0,0125	0.186	0,025	2.659
19	0,0125	0.583	0,025	0.053
20	0,0125	0.016	0,025	0.763
21	0,0125	0.031	0,025	0.032
22	0,0125	1.847	0,025	0.457
23	0,0125	0.736	0,025	0.048
24	0,0125	4.03	0,025	0.431
25	0,0125	0.24	0,025	0.325
26	0,0125	0.049	0,025	1.334
27	0,0125	2.929	0,025	0.57
28	0,0125	0.754	0,025	1.805
29	0,0125	0.156	0,025	1.478
30	0,0125	3.297	0,025	0.371
31	0,0125	0.557	0,025	0.946
32	0,0125	1	0,025	2.566
33	0,0125	0.902	0,025	0.59
34	0,0125	0.22	0,025	0.426
35	0,0125	1.343	0,025	0.461
36	0,0125	1.8	0,025	0.229
37	0,0125	1.839	0,025	0.254
38	0,0125	5.179	0,025	2.741
39	0,0125	0.18	0,025	3.117
40	0,0125	0.403	0,025	3.312
41	0,0125	3.462	0,025	10.76
42	0,0125	3.395	0,025	11.222
43	0,0125	1.528	0,025	11.596
44	0,0125	1.593	0,025	12.138
45	0,0125	3.391	0,025	12.622
46	0,0125	1.927	0,025	13.218
47	0,0125	2.117	0,025	13.767
48	0,0125	3.257	0,025	14.137
49	0,0125	9.828	0,025	14.782
50	0,0125	1.568	0,025	15.368

Como se puede apreciar, los tiempos son muy dispares y no siguen una tendencia concreta, lo que podemos afirmar basándonos en estos datos es que si no se consiguen los 0 jaques (condición fitness óptima) el tiempo de ejecución es algo mayor en la mayoría de los casos.

iv) Complejidad en memoria

Primero debemos tener en cuenta como codificamos un estado, en este caso necesitaremos un array de tantas posiciones como número de reinas tenga el problema, dicha posición del array contiene un entero indicando la posición que ocupa en el array viendo cada celda como una columna completa, adjunto un ejemplo para que se pueda apreciar mejor.



Este tablero se codificaría como [4,6,8,2,7,1,3,5], cada casilla del array hace referencia a una columna y dentro de este se guarda un entero que indica en que posición de la columna se encuentra, este valor oscilará entre 1 y el número total de reinas que se le asigna al problema.

Así que inicialmente contamos un coste en memoria de: (nreinas*población_inicial) eso multiplicado por 2 bytes que ocupa un entero.