IMPLEMENTACIÓN DE UN ALGORITMO EVOLUTIVO PARA RESOLVER EL PROBLEMA DE LAS OCHO REINAS

César Christopher Hernández Aparicio [a]

[a] *Facultad de estadística e informática, Universidad Veracruzana, Xalapa, Veracruz*

Lunes, 28 de abril del 2020

Abstract:

Los algoritmos evolutivos son parte de la Inteligencia Artificial principalmente conocidos como problemas con espacios de búsqueda extensos y no lineales donde otros métodos no son capaces de encontrar soluciones en tiempos razonables. Un algoritmo evolutivo es un método de optimización y búsqueda de soluciones basado en los métodos de la evolución biológica, por lo que se mantiene un conjunto de entidades que representan posibles soluciones, las cuales se mezclan, y compiten entre sí, de tal manera que las más aptas son capaces de prevalecer a lo largo del tiempo, evolucionando hacia mejores soluciones cada vez. Por su capacidad de búsqueda de soluciones en un espacio de búsqueda extenso se adaptará la implementación de este al problema de las ocho reinas para encontrar una solución donde las reinas no colisionen.

Introducción

El problema de las ocho reinas es un juego de ingenio donde se busca ubicar 8 reinas en un tablero de ajedrez de tal forma que estas no se amenacen. En ajedrez, las reinas tienen la posibilidad de moverse en las direcciones horizontal, vertical y diagonales (ver figura 1) dentro del tablero de ajedrez, el cual está compuesto de 64 cuadros ordenados en una matriz de 8 x 8 donde alternan sus colores (como se ve en la Figura 1).

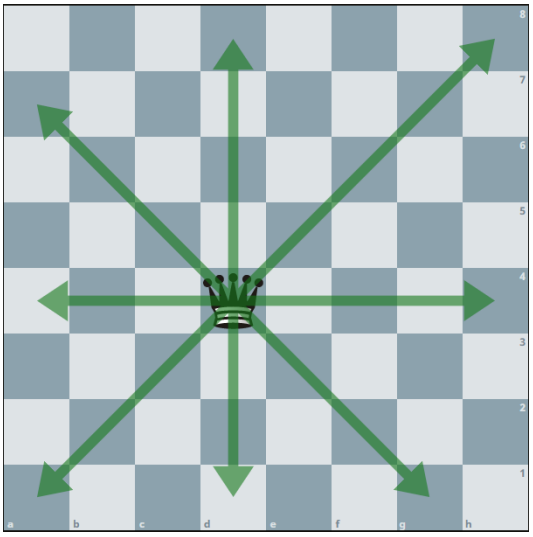


Figura 1 Movimientos de una reina

Para la solución de este problema se utilizará un algoritmo evolutivo codificado en el lenguaje de programación C#. Para ello, el tablero donde se ubicarán las reinas será una lista de enteros donde el índice representará la fila (o numero en la Figura 1) y el valor del entero será el valor de la columna (o letra en la Figura 1), además de un valor de puntuación que será la cantidad de choques que las reinas tendrán (ver Figura 2).

```
4   namespace ProblemaOchoReinas
5   {
        9 references
6       public class Tablero
7       {
            6 references
8           public List<int> PosicionesDeReinas = new List<int>();
            9 references
9           public int Puntuacion;
10
11      }
12
```

Figura 2 Clase tablero

Algoritmo evolutivo

Este algoritmo esta desarrollado en el lenguaje de programación C# y abstraído a 2 clases Tablero y Algoritmo Evolutivo que se pueden encontrar en el repositorio en las referencias (Hernández Aparicio, 2020).

Para representar el comportamiento del algoritmo evolutivo de las 8 reinas se representará en una

clase donde las propiedades de solo lectura serán las propiedades específicas del algoritmo (ver Figura 3 y 4).

Representación	Permutaciones
Tamaño de la población	100
Inicialización	Aleatoria
Selección de Padres	Mejores dos de 5 aleatorios
Número de descendientes	2
Cruza	Cortar y llenar cruzado
Probabilidad de Mutación	0.8
Mutación	Intercambio
Reemplazo	Reemplazar el peor
Condición de Paro	Encontrar la solución o llegar a 10 000 evaluaciones

Figura 3 Propiedades del algoritmo

```
2 references
public class AlgoritmoEvolutivo
{
    3 references
    private readonly int TOTAL_POBLACION = 100;
    1 reference
    private readonly int MAXIMO_DE_ITERACIONES = 10000;
    1 reference
    private readonly float PROBABILIDAD_DE_MUTACION = 0.8f;
    8 references
    private readonly int NUMERO_DE_REINAS = 8;
    12 references
    public List<Tablero> Poblacion = new List<Tablero>();
    2 references
    public int TotalEvaluaciones = 0;
    4 references
    public int TotalIteraciones = 0;
```

Figura 4 Propiedades del algoritmo codificadas

Debido a que el objetivo es que el algoritmo se comporte como la evolución biológica debe seguir el flujo de la supervivencia del mas apto en su lógica (especificado en la Figura 5). Esta implementación lógica esta descrita en el programa de la siguiente forma:

Inicialización del algoritmo

Se generan 100 tableros iniciales con posiciones aleatorias entre 0 y 7 (debido a que todos los índices y valores en este programa serán inicializados en 0) y se añadirán a la Población del algoritmo (ver Figura 6), la generación de las reinas se realizara de manera aleatoria seleccionando de manera iterativa de una lista con las posibles posiciones de las reinas, de esta forma se evitan ciclos infinitos donde se seleccione un numero que no este ya incluido en la lista de posiciones de cada tablero generado.

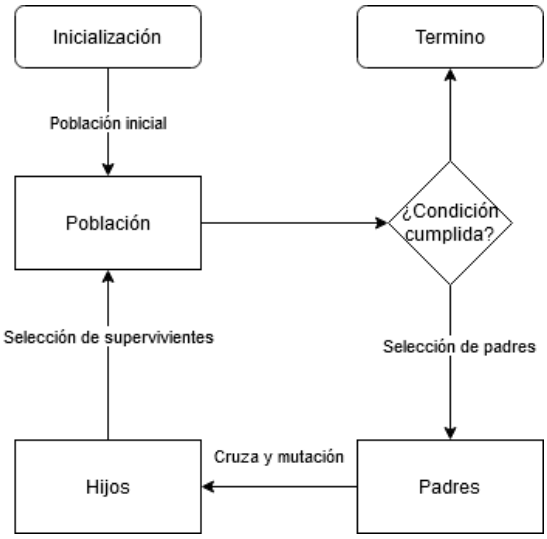


Figura 5 Flujo del algoritmo evolutivo

```
        1 reference
        private void LlenarPoblacion()
        {
            for (int i = 0; i < TOTAL_POBLACION; i++)
            {
                Tablero tablero = new Tablero();
                tablero.PosicionesDeReinas = GenerarListaDeReinasAleatorias();
                tablero.Puntuacion = CalcularNumeroDeChoques(tablero);
                Console.Write("Tablero #{0}, Choques: {1}    ",i,tablero.Puntuacion);
                tablero.Imprimir(true);
                Poblacion.Add(tablero);
            }
        }

        1 reference
        private List<int> GenerarListaDeReinasAleatorias()
        {
            List<int> lista = new List<int>();
            for (int i = 0; i < NUMERO_DE_REINAS; i++)
            {
                lista.Add(i);
            }
            lista = lista.OrderBy(i => new Random().Next()).ToList();

            List<int> listaRandom = new List<int>();
            int j = NUMERO_DE_REINAS;
            while (j > 0)
            {
                int indiceRandom = new Random().Next(j);
                listaRandom.Add(lista[indiceRandom]);
                lista.RemoveAt(indiceRandom);
                j--;
            }

            return listaRandom;
        }
    }
```

Figura 6 Inicialización de la población inicial

Condición de salida

Se valida mediante la función de aptitud si la solución se ha encontrado (ver Figura 7), de tal forma que si las reinas de algún tablero de la población (100 en total) tiene un número de choques igual a 0 significara que la solución se ha encontrado y terminara el algoritmo.

```
2 references
42        private bool SeEncuentraLaSolucion()
43        {
44            bool solucion = false;
45            for (int i = 0; i < Poblacion.Count; i++)
46            {
47                if (Poblacion[i].Puntuacion == 0)
48                {
49                    solucion = true;
50                }
51            }
52            return solucion;
53        }
54
```

Figura 7 Condición de salida del algoritmo

Selección de padres

Debido a que el algoritmo es un ciclo, en cada iteración se seleccionaran 2 padres, cada uno de 5 tableros seleccionados aleatoriamente de la población (ver Figura 9), se puede observar que en cada iteración del ciclo (cada ciclo se puede definir como las líneas 26 a 39 de la figura 8) se escogen 2 padres por la función EscogerMejorDe5Aleatorios (ver Figura 9).

```
1 reference
17        public void Iniciar()
18        {
19            LlenarPoblacion();
20            Console.Write("MEJOR TABLERO EN POBLACION INICIAL: ");
21            ImprimirMejorTablero();
22            Console.WriteLine();
23
24            bool solucionEncontrada = SeEncuentraLaSolucion();;
25
26            while (TotalIteraciones < MAXIMO_DE_ITERACIONES && !solucionEncontrada)
27            {
28                TotalIteraciones++;
29                int padre1 = EscogerMejorDe5Aleatorios();
30                int padre2 = EscogerMejorDe5Aleatorios();
31                List<List<int>> hijos = CruzarPadres(Poblacion[padre1], Poblacion[padre2]);
32                MutarHijos(hijos);
33                ReemplazarPoblacion(hijos);
34
35                Console.Write("ITERACION: {0}  ",TotalIteraciones);
36                ImprimirMejorTablero();
37
38                solucionEncontrada = SeEncuentraLaSolucion();
39            }
40        }
```

Figura 8 Inicio e iteraciones del algoritmo

```
2 references
131       public int EscogerMejorDe5Aleatorios()
132       {
133           int mejor = -1;
134           List<int> listaAleatorios = new List<int>();
135           for (int i = 0; i < 5; i++)
136           {
137               int numero = new Random().Next(TOTAL_POBLACION - 1);
138               if (listaAleatorios.Contains(numero))
139               {
140                   i--;
141               }
142               else
143               {
144                   listaAleatorios.Add(numero);
145               }
146           }
147
148           List<int> listaOrdenada;
149           listaOrdenada = listaAleatorios.OrderByDescending(o => Poblacion[o].Puntuacion).ToList();
150           mejor = listaAleatorios.First();
151
152           return mejor;
153       }
```

Figura 9 Selección aleatoria de padres

Generación de hijos

Una vez por ciclo se generarán hijos por medio de la cruza por permutación de los dos padres seleccionados antes, posteriormente tendrá una probabilidad de mutación, del 80% en este ejemplo que distinguirá a cada hijo para tener un espacio de búsqueda mas extenso. Los conceptos para la generación de nuevos hijos se explicarán a continuación.

Cruza de padres: En esta implementación, la cruza de padres se generará recorriendo la longitud del tablero y añadiendo al primer hijo la primera parte del primer padre (ver línea 65 a 67 de la Figura 10).

Para añadir la segunda parte al hijo del segundo padre se verificará si el hijo ya tiene la parte que se le va a añadir del segundo padre en su tablero, si es así, se intercambiaran los elementos del segundo padre (ver Figura 11) dada la lista del hijo para que se intercambie por uno que hasta el momento el primer hijo no tenga, entonces, se añadirá el elemento al hijo, repitiéndose este proceso para cada reina de la segunda mitad del segundo padre.

Debido a que los elementos del segundo padre se han intercambiado para no repetirse en la lista del primer hijo, estos se encontraran ordenados de tal forma que para el segundo hijo simplemente se añada la primera parte del primer padre y la segunda parte del segundo padre (ver líneas 80 a 88 de la Figura 10).

Mutación de hijos: Se seleccionarán dos índices de la lista de posiciones de reinas del tablero y se intercambiarán con una probabilidad del 80%, debido a que no se puede cambiar un solo índice ya que habría dos reinas en la misma columna (o valor dentro de un índice en la lista), esto pueda dar la posibilidad de que los dos índices a intercambiar sean el mismo (ver Figura 12).

```
1 reference
public List<List<int>> CruzarPadres(Tablero primerPadre, Tablero segundoPadre)
{
    List<int> padre1 = new List<int>(primerPadre.PosicionesDeReinas);
    List<int> padre2 = new List<int>(segundoPadre.PosicionesDeReinas);
    List<int> hijo1 = new List<int>();
    List<int> hijo2 = new List<int>();

    int mitadDeNumeroDeReinas = NUMERO_DE_REINAS / 2;
    for (int i = 0; i < NUMERO_DE_REINAS; i++)
    {
        if (i < mitadDeNumeroDeReinas)
        {
            hijo1.Add(padre1[i]);
        }
        else
        {
            if (hijo1.Contains(padre2[i]))
            {
                IntercambiarElementosDeListaSinRepetirEnOtraLista(padre2, hijo1, i);
            }

            hijo1.Add(padre2[i]);
        }
    }

    for (int i = 0; i < NUMERO_DE_REINAS; i++)
    {
        if (i < mitadDeNumeroDeReinas)
        {
            hijo2.Add(padre1[i + mitadDeNumeroDeReinas]);
        }
        else
        {
            hijo2.Add(padre2[i - mitadDeNumeroDeReinas]);
        }
    }

    List<List<int>> hijos = new List<List<int>>();
    hijos.Add(hijo1);
    hijos.Add(hijo2);

    return hijos;
}
```

Figura 10 Cruza de padres

```
private void IntercambiarElementosDeListaSinRepetirEnOtraLista(List<int> lista,
List<int> listaANoRepetir, int indice)
{
    for (int i = 0; i < lista.Count / 2; i++)
    {
        if (!listaANoRepetir.Contains(lista[i]))
        {
            int elementoACambiar = lista[indice];
            lista[indice] = lista[i];
            lista[i] = elementoACambiar;
            break;
        }
    }
}
```

Figura 11 Intercambiar reinas del padre según un hijo

```
1 reference
public void MutarHijos(List<List<int>> hijos)
{
    foreach (List<int> hijo in hijos)
    {
        float aleatorio = (float) new Random().NextDouble();
        if (aleatorio <= PROBABILIDAD_DE_MUTACION)
        {
            int indiceACambiar = new Random().Next(NUMERO_DE_REINAS);
            int indiceACambiar2 = new Random().Next(NUMERO_DE_REINAS);

            int valorACambiar = hijo[indiceACambiar];
            hijo[indiceACambiar] = hijo[indiceACambiar2];
            hijo[indiceACambiar2] = valorACambiar;
        }

    }
}
```

Figura 12Mutación de hijos

Selección de supervivientes

Esta esta implementada en el método de reemplazar población (ver Figura 13). Este método ordena la población en una lista de índices por su puntuación (que es el resultado de la función de aptitud en la Figura 14), ordena esta lista de mayor a menor y por cada nuevo elemento (o hijo) sobre escribe el último elemento (el peor debido al ordenamiento) por el hijo de la iteración.

```
1 reference
private void ReemplazarPoblacion(List<List<int>> hijos)
{
    List<int> indicesDePoblacion = new List<int>();
    for (int i = 0; i < TOTAL_POBLACION; i++)
    {
        indicesDePoblacion.Add(i);
    }

    indicesDePoblacion =
    indicesDePoblacion.OrderByDescending(i => Poblacion[i].Puntuacion).ToList();
    foreach (List<int> hijo in hijos)
    {
        int indice = indicesDePoblacion.First();
        indicesDePoblacion.Remove(indicesDePoblacion.First());
        Poblacion[indice].PosicionesDeReinas = hijo;
        Poblacion[indice].Puntuacion = CalcularNumeroDeChoques(Poblacion[indice]);
    }
}
```

Figura 13 Reemplazar población con nuevos hijos

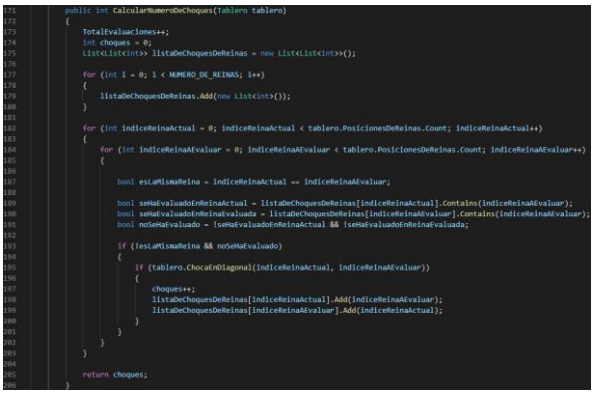
Función de aptitud

Evalúa en una métrica de entero la calidad de un tablero, en este problema sería el numero de choques de reinas (ver Figura 14). Esta representada en el método calcular número de choques, este método calcula el numero de choques entre cada reina con las otras 7 reinas, de tal forma que si, por ejemplo, una reina A (1,2) y otra reina B (2,3), donde se evalúa con quien choca A y se evalúa que A choca con B cuando se evalué B y se descubra que B choca con A solo se cuente como un choque en lugar de dos. De esta forma el numero de choques es la cantidad de choques real.

Para validar el choque de las reinas solo se debe validar los choques diagonales debido a que, cada índice de la lista de reinas es la fila, estas no se pueden repetir, y debido a que en la generación de tableros no se pueden repetir números en la lista las columnas tampoco se repetirán. La validación de choques diagonales se delega al método choca en diagonal del tablero (ver Figura 15), donde por medio de la fórmula:

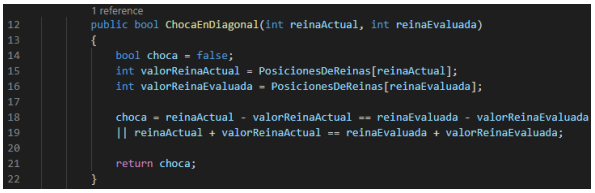
$$i - j = k - l \text{ o } i + j = k + l$$

Donde (i , j) es una reina y (k , l) es otra, se valida que chocan en diagonal debido a que la primera parte de la formula valida los choques en diagonal descendiente y la parte siguiente en diagonal ascendiente.



```
public int CalcularNumeroDeChoques(Tablero tablero)
{
    TotalEvaluaciones++;
    int choques = 0;
    List<List<int>> listaDeChoquesDeReinas = new List<List<int>>();

    for (int i = 0; i < NUMERO_DE_REINAS; i++)
    {
        listaDeChoquesDeReinas.Add(new List<int>());
    }

    for (int indiceReinaActual = 0; indiceReinaActual < tablero.PosicionesDeReinas.Count; indiceReinaActual++)
    {
        for (int indiceReinaAEvaluar = 0; indiceReinaAEvaluar < tablero.PosicionesDeReinas.Count; indiceReinaAEvaluar++)
        {

            bool esLaMismaReina = indiceReinaActual == indiceReinaAEvaluar;

            bool seHaEvaluadoEnReinaActual = listaDeChoquesDeReinas[indiceReinaActual].Contains(indiceReinaAEvaluar);
            bool seHaEvaluadoEnReinaEvaluada = listaDeChoquesDeReinas[indiceReinaAEvaluar].Contains(indiceReinaAEvaluar);
            bool noSeHaEvaluado = !seHaEvaluadoEnReinaActual && !seHaEvaluadoEnReinaEvaluada;

            if (!esLaMismaReina && noSeHaEvaluado)
            {
                if (tablero.ChocaEnDiagonal(indiceReinaActual, indiceReinaAEvaluar))
                {
                    choques++;
                    listaDeChoquesDeReinas[indiceReinaActual].Add(indiceReinaAEvaluar);
                    listaDeChoquesDeReinas[indiceReinaAEvaluar].Add(indiceReinaActual);
                }
            }
        }
    }

    return choques;
}
```

Figura 14 Función de aptitud



```
1 reference
public bool ChocaEnDiagonal(int reinaActual, int reinaEvaluada)
{
    bool choca = false;
    int valorReinaActual = PosicionesDeReinas[reinaActual];
    int valorReinaEvaluada = PosicionesDeReinas[reinaEvaluada];

    choca = reinaActual - valorReinaActual == reinaEvaluada - valorReinaEvaluada
    || reinaActual + valorReinaActual == reinaEvaluada + valorReinaEvaluada;

    return choca;
}
```

Figura 15 Calcular choque en diagonal

Ejecución del algoritmo

Después de construir este algoritmo evolutivo, se decidió probar con 30 ejecuciones para posteriormente evaluar el comportamiento del algoritmo por medio de sus estadísticas (ver Tabla 1) de la cantidad de veces que la función de aptitud fue ejecutada.

Durante la ejecución del algoritmo se encontró que el número de veces que se encontraba la solución en los primeros 100 tableros de la población inicial del algoritmo fue 8, planteando que la probabilidad de que el algoritmo no se ejecute debido a que la solución estaba en la población inicial sea de 26.6666% alejándose por 3.8491% de la probabilidad de encontrar una solución en 100 tableros la cual es 22.8174 % (40320 casos posibles entre 92 soluciones por 100 tableros).

Al analizar dichas estadísticas se puede observar que el algoritmo encontró la solución del tablero todas las iteraciones que se ejecutó, además, se observa que la mejor iteración fue de 100 evaluaciones, ósea las 100 evaluaciones iniciales de la generación de la población inicial, lo que quiere decir que la solución se encontraba en los tableros iniciales y no se ejecutó ninguna vez el algoritmo evolutivo; lo cual concuerda con la probabilidad de encontrar una solución en 100 tableros del 20%

Conclusión

Dentro del análisis de las estadísticas del algoritmo que se recopilaron en la ejecución del algoritmo se observo que la mejor ejecución del algoritmo fue de 100 evaluaciones, lo cual no ejecuto el algoritmo ya que pertenecen a la población inicial, sin embargo, el siguiente mejor fue de 112 evaluaciones, encontrando la solución en 12 evaluaciones extra.

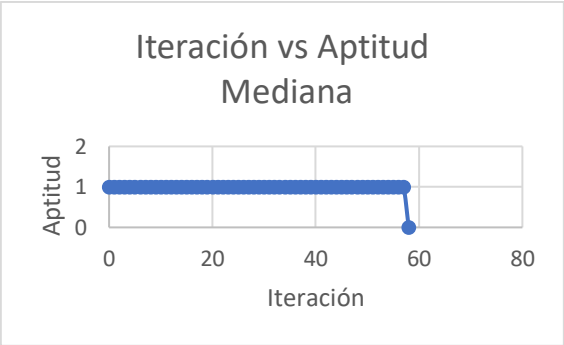
Así mismo observamos que la media de evaluaciones es de 271.33 por ejecución del algoritmo evolutivo y que la peor ejecución fue de 1040 evaluaciones, una decima parte de la cantidad total de evaluaciones permitida, pero encontrando siempre la solución.

Estadísticas del algoritmo por número de evaluaciones	
Número Total de Ejecuciones	30
Número de Ejecuciones Exitosas	30
Mejor	100
Mejor fuera de población inicial	112
Media	271.3333333
Mediana	214
Desviación Estándar	222.976933
Peor	1040

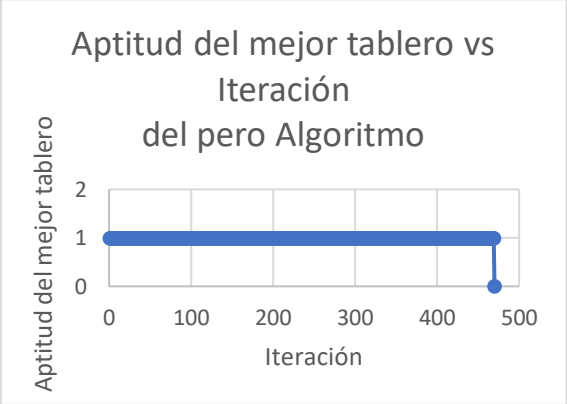
Tabla 1 Estadísticas del algoritmo

Al analizar las estadísticas de la ejecución del algoritmo evolutivo en la mediana podemos observar que este inicia con la aptitud del mejor tablero en 1 y que en 58 iteraciones se encontró la primera solución de 0 choques (ver Gráfica 1).

Por último, podemos observar que en el peor desempeño del algoritmo (1040 evaluaciones) tardo 470 evaluaciones obtener la solución aun empezando con la aptitud de 1 entre los primeros 100 tableros de la población inicial (ver Gráfica 2).



Gráfica 1 Estadísticas del algoritmo en la mediana



Gráfica 2 Estadísticas del algoritmo en el peor

Referencias

- Hernández Aparicio, C. C. (28 de 04 de 2020). *Github*. Obtenido de https://github.com/cesarhdezap/8queen