"El Problema de las N-Reinas"

Andreas Spading Pablo Itaim Ananias

Valparaíso, 01 de Julio del 2005

Resumen

El problema de las n-Reinas (*n-Queens Problem*) es muy antiguo, propuesto por primera vez en el año 1848, consiste en encontrar una asignación a n reinas en un tablero de ajedrez de n x n de modo tal, que éstas no se ataquen. El presente trabajo presenta el problema en cuestión, muestra un ejemplo sencillo de él y profundiza en las diferentes alternativas de modelamiento y métodos de resolución existentes. Además, se muestra un ejemplo de resolución del problema utilizando un algoritmo Branch and Bound.

1. Introducción

El problema de las n-Reinas consiste en encontrar una distribución de n reinas en un tablero de ajedrez de n x n de modo tal, que éstas no se ataquen. Así, no pueden encontrarse dos reinas en la misma fila, columna o diagonal.

Este problema tiene 2 Versiones. La más simple consiste en encontrar exactamente una solución válida para un valor n dado. La otra versión, más difícil, consiste en encontrar todas las soluciones posibles para un valor n.

Fue propuesto para $n = 8$ en el año 1848 en un trabajo anónimo [1], siendo posteriormente atribuido a Max Bezzel. Sin embargo, la publicación detallada más antigua que se conoce fue realizada por Nauck [2] en 1850. Ese mismo año, Gauss postuló la existencia de 72 soluciones para $n = 8$. Posteriormente, en el año 1874, Glaisher [3] probó la existencia de 92 soluciones.

La investigación sobre este tema no ha parado hasta hoy por lo que existe una amplia variedad de algoritmos sugeridos para su resolución. Muchas de las soluciones planteadas se basan en proporcionar una fórmula específica para colocar las reinas o extrapolar conjuntos pequeños de soluciones para proporcionar soluciones para valores de n más grandes.

Observaciones empíricas de problemas de pequeño tamaño muestran que el número de soluciones crece en forma exponencial al ir aumentando el valor de n [4].

Se puede observar que este problema tiene una solución ($Q(1) = 1$) para $n = 1$, no tiene solución para $n = 2$ y $n = 3$ y tiene 2 soluciones para $n = 4$.

2. Definición del Problema:

El Problema de las n-Reinas consiste en encontrar una distribución de n reinas en un tablero de n x n de modo tal, que éstas no se ataquen. Así, no pueden encontrarse dos reinas en la misma fila, columna o diagonal.

Este Problema tiene 2 Versiones. La más simple consiste en encontrar exactamente una solución válida para un valor n dado. La otra versión, más difícil, consiste en encontrar todas las soluciones posibles para un valor n.

Se puede observar que este problema tiene una solución ($Q(1) = 1$) para $n = 1$, no tiene solución para $n = 2$ y $n = 3$ y tiene 2 soluciones para $n = 4$. La Tabla 1 muestra el número total de Soluciones $Q(n)$ para $4 \leq n \leq 20$ [5] [6].

Para solucionar este problema, se han diseñado numerosos Algoritmos basados en algunos como Backtracking, Algoritmos Genéticos, Búsqueda local con resolución de conflictos, programación entera y redes neuronales entre otros. Además, el Problema de las n reinas pertenece a la clase de problemas *NP-completos* [7], pero se resuelve fácilmente en tiempo polinomial cuando solamente se busca una solución [4].

n	$Q(n)$
4	2
5	10
6	4
7	40
8	92
9	352
10	724
11	2.680
12	14.200
13	73.712
14	365.596
15	2.279.184
16	14.772.512
17	95.815.104
18	666.090.624
19	4.968.057.848
20	39.029.188.884

Cuadro 1: Número total de soluciones $Q(n)$ para $4 \leq n \leq 20$

Existen soluciones analíticas donde se da una fórmula explícita para la localización de las reinas o bien se encadenan soluciones obtenidas para valores menores de n [8] [9]. El problema con estas soluciones es que generan un número muy pequeño de soluciones.

Backtracking, en general es ineficiente y para el problema de las n-reinas no es fácil encontrar soluciones para $n > 100$ en tiempos razonables [10], sin embargo, Kalé [11] diseño un algoritmo especializado de backtracking que consigue resolver el problema hasta tamaños del orden de 1.000. Se han realizado numerosas aplicaciones de algoritmos genéticos para resolver el problema de las n-reinas [7] [12] [13] [14], pero se han visto ineficientes aún cuando se hayan probado diversas representaciones del problema y diversos operadores.

Por otra parte, los Algoritmos para Programación Entera son típicamente exponenciales y consecuentemente consiguen resolver problemas de tamaño muy pequeño [15].

Un Algoritmo de Redes Neuronales utilizando un modelo modificado de Hopfield [16] parece funcionar mejor que otros modelos de redes neuronales presentados, pero solo se publican resultados para valores de n pequeños.

La Búsqueda local con minimización de conflictos [4] parece ser el mejor algoritmo encontrado hasta ahora. Este algoritmo de Socic y Gu tiene tiempo de ejecución lineal por lo que es extremadamente rápido. En su articulo, ellos muestran que su algoritmo es capaz de obtener una solución para cualquier valor de $n, n < 1000$ en menos de 0.1 segundo y 55 segundos para $n = 3{,}000{,}000$, en un computador del año 94 (IBM RS 6000/530). Ahora, dada la naturaleza probabilística del algoritmo, no se tiene garantía del tiempo del peor caso del algoritmo, pero en la práctica muestra un excelente desempeño y un comportamiento muy robusto. Hynek [14] diseñó una Heurística basada en una fase de preprocesamiento y posteriormente aplica un operador de mutación basado en búsqueda local. Los resultados computacionales muestran que el algoritmo es efectivo y logra soluciones hasta $n = 1000$ en tiempos razonables.

3. Aplicaciones:

El Problema de las n-reinas es conocido usualmente como uno relacionado a un juego y también como un problema apropiado para probar algoritmos nuevos. Sin embargo, tiene otras aplicaciones ya que se le considera como un modelo de máxima cobertura. Una solución al problema de las n-reinas garantiza que cada objeto puede ser accesado desde cualquiera de sus ocho direcciones vecinas (dos verticales, dos horizontales y cuatro diagonales) sin que tenga conflictos con otros objetos.

Algunas aplicaciones posibles son:

- Control de Tráfico Aéreo
- Sistemas de Comunicaciones

- Programación de Tareas Computacionales
- Procesamiento Paralelo Óptico
- Compresión de Datos
- Balance de Carga en un Computador multiprocesador
- Ruteo de mensajes o datos en un Computador multiprocesador
- etc.

Un ejemplo práctico de una aplicación del problema de las n-reinas es el siguiente: para lograr el mayor ancho de banda posible en un sistema de comunicaciones de banda estrecha y direccional, se deben ubicar los n transceptores de forma tal que no se interfieran entre ellos. Con los n transceptores ubicados en patrones sin conflictos, lo que corresponde a una solución del problema de las n-reinas, cada transceptor puede comunicarse con el mundo exterior en forma libre en 8 direcciones sin que sea inhibido por otro transceptor. La Figura 3 muestra una distribución para 10 transceptores. Esta distribución corresponde a una de las soluciones para el problema de las 10-reinas.

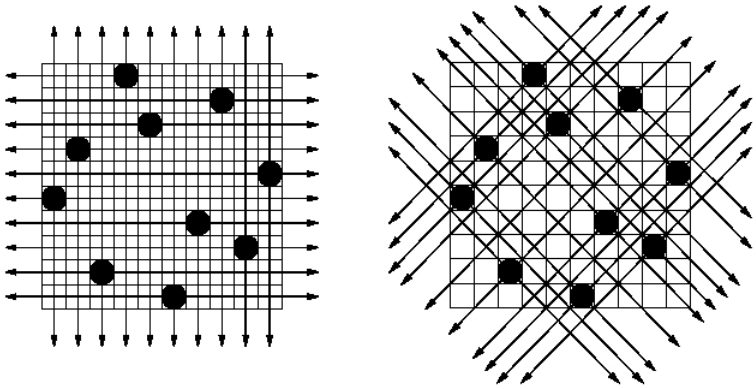

Figura 1: Ubicación de 10 Transmisores/Receptores sin conflicto entre ellos (Cada uno puede comunicarse en cualquier dirección vertical, horizontal o diagonal

4. Ejemplo:

Para visualizarlo mejor, analizaremos el problema de las n reinas para $n = 6$. Como ya hemos visto, esto consiste en ubicar 6 reinas en un tablero de 6x6 sin que ellas se ataquen. Se sabe que para este caso existen 4 soluciones válidas, las que se presentan en los gráficos de la figura 2.

Con algún método, comenzamos la búsqueda de una solución y encontramos la a). A partir de ella podemos encontrar las otras 3 de la siguiente forma:

- La solución b) se obtiene rotando la solución a), 90 grados a la derecha.
- La solución c) es el reflejo de la solución b) respecto al borde derecho de ésta. Es decir, si coloco un espejo en el borde derecho de la solución b), puedo mirar la solución c).
- La solución d) es el reflejo de la solución a) respecto del borde derecho de ésta.

Si bien estos métodos ayudan a encontrar soluciones válidas del problema, no las encuentran todas, por lo que se hace necesario contar con otros métodos para solucionar el problema. Además, es necesario encontrar una solución inicial para poder aplicarlos.

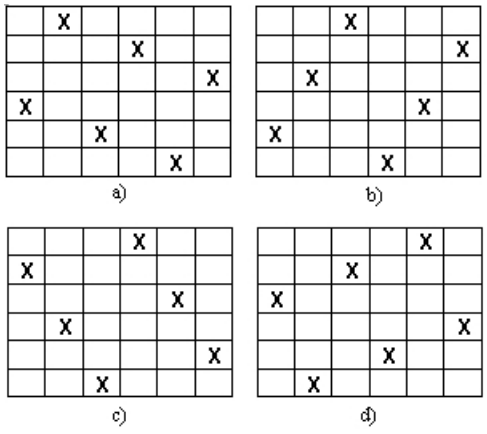

Figura 2: Soluciones al problema de las 6-reinas

5. Modelo en Extenso:

El problema de las n-reinas tiene, básicamente, dos modelos asociados que cumplen con lo siguiente:

- La Función Objetivo no tiene un uso práctico ya que se sabe a priori el valor de n.
- Dos reinas no pueden estar en la misma Fila.
- Dos reinas no pueden estar en la misma Columna.
- Dos reinas no pueden estar en la misma Diagonal.

Para efectos de mejor entendimiento, vamos a dividir las diagonales según el signo de su pendiente. De esta forma, tendremos diagonales positivas y diagonales negativas. Además, vamos a definir el número de reinas y ancho del tablero de ajedrez como N, con $N = \{1, \ldots, n\}$.

Si bien la cantidad de formas de modelar son pocas, estos modelos se resuelven o desarrollan de muchas formas y con varios métodos distintos.

5.1. Modelo 1

Este modelo es el más general y el más fácil de implementar. Utiliza una variable binaria x_{ij} para representar la existencia de una reina en la casilla (i, j). Así, si existe, $x_{ij} = 1$ y si no, vale 0. Las restricciones abarcan las filas, las columnas, diagonales positivas y diagonales negativas.

La restricción de las filas impide que existan dos reinas en la misma fila. Esto se verifica sumando todos los valores de las casillas x_{ij} para una fila determinada. Así, dicha suma debería ser igual a 1 para todas las filas ya que debería haber un solo 1 en cada fila. Esta restricción se puede resumir de la siguiente forma:

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad \forall i \in N$$

Del mismo modo, la restricción de las columnas impide que existan dos reinas en la misma columna. Esto se verifica sumando todos los valores de las casillas x_{ij} para una columna determinada. Así, dicha suma debería ser igual a 1 para todas las columnas ya que debería haber un solo 1 en cada columna. Esta restricción se puede resumir de la siguiente forma:

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad \forall j \in N$$

Las diagonales son un poco más complicadas de verificar por la definición de los valores de recorrido de la variable. En las diagonales, tanto positivas como negativas, debe haber a lo más una reina. Esto se verifica sumando los valores de las casillas que forman las distintas diagonales.

Si analizamos la formación de las diagonales positivas, veremos que éstas estarán definidas por la suma $k = i + j$. Así, todos los pares (i, j) que tengan el mismo valor para k, estarán en la misma diagonal positiva.

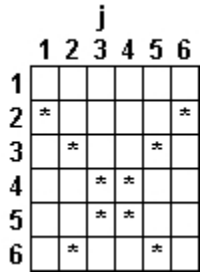

Figura 3: Ejemplo de diagonales positivas y negativas para $n = 6$

Por ejemplo, en la figura 3, podemos observar que todos los pares que forman la diagonal positiva suman 8. Ahora, para el mismo ejemplo, podemos observar que k se mueve dentro del rango $\{2, \ldots, 12\}$. Si no consideramos las dos diagonales de las esquinas ($(1, 1)$ y $(6, 6)$), el valor de k varía en el rango de $\{3, \ldots, 11\}$. De esta forma, definiendo los valores de k podemos recorrer todas las diagonales positivas.

Generalizando, podemos afirmar que $k = i + j$ varía en el rango de $\{2, \ldots, 2n\}$ o $\{3, \ldots, 2n - 1\}$ dependiendo si consideramos las casillas de las esquinas.

De esta forma, podemos definir la restricción para las diagonales positivas de la siguiente forma:

$$\underbrace{\sum_{i=1}^{n}\sum_{j=1}^{n}}_{i+j=k} x_{ij} \leq 1 \qquad \forall k = \{2, \ldots, 2n\} \text{ o } k = \{3, \ldots, 2n - 1\}$$

Análogamente, analizamos la formación de las diagonales negativas. Podemos verificar que éstas están definidas por la resta $k = i - j$. Así, todos los pares (i, j) que tienen el mismo valor para k, están en la misma diagonal negativa.

Por ejemplo, en la figura 3, podemos observar que para todos los pares que forman la diagonal negativa, sus restas valen 1. Ahora, para el mismo ejemplo, podemos observar que k se mueve dentro del rango $\{-5, \ldots, 5\}$. Si no consideramos las dos diagonales de las esquinas ($(6, 1)$ y $(1, 6)$), el valor de k varía en el rango de $\{-4, \ldots, 4\}$. De esta forma, definiendo los valores de k podemos recorrer todas las diagonales negativas.

Generalizando, podemos afirmar que $k = i - j$ varía en el rango de $\{-5, \ldots, 5\}$ o $\{-4, \ldots, 4\}$ dependiendo si consideramos las casillas de las esquinas.

De esta forma, podemos definir la restricción para las diagonales negativas de la siguiente forma:

$$\underbrace{\sum_{i=1}^{n}\sum_{j=1}^{n}}_{i-j=k} x_{ij} \leq 1 \qquad \forall k = \{-5, \ldots, 5\} \text{ o } k = \{-4, \ldots, 4\}$$

El problema de este modelo es el tamaño del espacio de búsqueda, que es 2^{n*n}. Así, para $n = 10$, este sería de 10^{30} aproximadamente, lo cual lo hace muy ineficiente a la hora de buscar todas las soluciones al problema.

5.2. Modelo 2

Otra forma de modelar el mismo problema es considerando otro tipo de variables. Podemos utilizar la variable x_i para representar la posición de la reina en la fila i. En este caso, el dominio de x_i sería $\{1, \ldots, n\}$. A diferencia del modelo anterior, las restricciones abarcan las columnas y todas las diagonales, sin distinguir entre positivas y negativas. Las filas se verifican por defecto, al tener que asignársele un valor a la variable. Es decir, por la forma del modelo, no pueden haber dos reinas en la misma fila, por lo que esa restricción se omite.

La restricción de las columnas impide que existan dos reinas en la misma columna. Esto se verifica observando los valores de las variables. Si estos son distinto, entonces se cumple con la restricción, si son iguales, no. Otra forma análoga es calculando la diferencia entre dos valores de x. De esta forma, si $x_i - x_j = 0$ (con $i \neq j$) las dos reinas están en la misma columna. Esta restricción se puede resumir de la siguiente forma:

$$x_i \neq x_j \qquad \forall i \neq j \text{ con } i \text{ y } j \in N$$

Análogamente:

$$x_i - x_j \neq 0 \qquad \forall i \neq j \text{ con } i \text{ y } j \in N$$

Al igual que en el modelo anterior, en las diagonales, tanto positivas como negativas, debe haber a lo más una reina. Dada la definición del modelo, el valor de la variable me indica la columna y el índice de la variable, la fila. Utilizando esos valores, se puede calcular la pendiente de una recta que pase por los dos casilleros, x_i y x_j que estoy verificando. Sabiendo que todas las diagonales positivas forman un ángulo de $45°$ respecto a la base del tablero y que todas las diagonales negativas forman un ángulo de $-45°$ respecto a la misma base, si calculamos la pendiente y el ángulo resultante es $\pm 45°$, entonces los dos casilleros están en la misma diagonal.

En general, para poder encontrar la pendiente de una recta, basta con conocer dos puntos de esa recta y calcular la diferencia entre las coordenadas de ambos puntos. Por ejemplo, si queremos calcular la pendiente de la recta de la figura 4, calculamos el $\Delta x = x_2 - x_1$ y el $\Delta y = y_2 - y_1$. Después, calculamos $\frac{\Delta y}{\Delta x}$ y al resultado le calculamos la tangente.

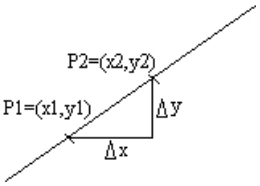

Figura 4: Cálculo de la pendiente de una recta

Como la $\tan 45 = 1$, eso significa que el Δx y el Δy deben ser iguales. En nuestro caso, como no queremos que las reinas estén en la misma diagonal, vamos a exigir que esas diferencias sean distintas.

De esta forma, podemos definir la restricción para las diagonales de la siguiente forma:

$$|x_i - x_j| \neq |i - j| \qquad \forall i \neq j \text{ con } i \text{ y } j \in N$$

Lo bueno de este modelo, comparado con el anterior, es que reduce en forma drástica el espacio de búsqueda, quedando en $n!$. De esta forma, para $n = 10$, el espacio de búsqueda es 10^6 aproximadamente.

6. Modelo General

Considerando los modelos anteriormente descritos, podemos formalizarlos de acuerdo a lo siguiente:

6.1. Modelo 1

Variables :

- $x_{ij} = \begin{cases} 1 & \text{si existe una reina en la posición } (i,j) \\ 0 & \text{si no} \end{cases}$

Dominio : $\{0, 1\}$

Restricciones :

$$\sum_{j=1}^{n} X_{ij} \quad = \quad 1 \qquad ; \forall i = \{1, \ldots, n\} \qquad \text{Restricción en las Filas}$$

$$\sum_{i=1}^{n} X_{ij} = 1 \quad ;\forall j = \{1,\ldots,n\} \qquad \text{Restricción en las Columnas}$$

$$\underbrace{\sum_{i=1}^{n}\sum_{j=1}^{n}}_{i+j=k} X_{ij} \leq 1 \quad ;\forall k = \{3,\ldots,2n-1\} \qquad \text{Restricción en las Diagonales positivas}$$

$$\underbrace{\sum_{i=1}^{n}\sum_{j=1}^{n}}_{i-j=k} X_{ij} \leq 1 \quad ;\forall k = \{1-n,\ldots,n-1\} \qquad \text{Restricción en las Diagonales negativas}$$

6.2. Modelo 2

Variables : x_i = posición de la reina en la fila i.

Dominio : $\{1,\ldots,n\}$

Restricciones :

$$x_i \neq x_j \qquad \forall i \neq j \text{ con } i \text{ y } j \in N \qquad \text{Restricción en las Columnas}$$

$$|x_i - x_j| \neq |i-j| \qquad \forall i \neq j \text{ con } i \text{ y } j \in N \qquad \text{Restricción en las Diagonales}$$

7. Especificación

Para poder resolver los modelos planteados anteriormente, es necesario trasformarlos a algún lenguaje que sea entendido por algún método de resolución. Para el primer método, se utilizó AMPL por su facilidad de programación y por la sencillez de sus restricciones. Sin embargo, dado que el segundo método tiene restricciones más complejas de manejar, no fue posible utilizar el mismo lenguaje, por lo que se utilizó Java. A continuación se muestran los códigos respectivos.

7.1. Modelo 1

Para la formulación de este modelo se utilizo AMPL cuyo archivo respectivo puede verse a continuación:

nreinas.mod

```
 param t := 6; ←ajuste de ancho del Tablero
set FILAS := {1..t};
set COLUMNAS := {1..t};
var x{FILAS,COLUMNAS} binary; ← x_{i,j} es 1 si hay una reina en posición (i,j), sino 0.
ataq_colum {j in COLUMNAS}: sum {i in FILAS} x[i,j] = 1;
ataq_filas {i in FILAS}: sum {j in COLUMNAS} x[i,j] = 1;
diag_positiv {k in 2..2*t}: sum {i in FILAS, j in COLUMNAS: i+j=k} x[i,j] <= 1;
diag_negativ {k in (-t+2)..(t-2)}: sum {i in FILAS, j in COLUMNAS: i-j=k} x[i,j]<= 1;
```

7.2. Modelo 2

Después de formular nuestro primer modelo en AMPL con una representación con n x n variables, descubrimos que la ejecución era muy ineficiente. Al tratar de implementar el modelo 2 en AMPL, nos encontramos con que no contábamos con los operadores necesarios para hacerlo. Seguimos intentando hasta que contactamos a Rob Fourer (uno de los diseñador de AMPL [17]), quien nos respondió que no era posible formular el problem de las n-reinas con la representación actual.

Entregar el problema directamente a lp_solve tampoco era muy conveniente, pues no existen operadores relacionales como "≠" o cuantificadores como "para todos".

Por eso efectuamos una implementación del modelo 2, basados en el algoritmo QueensSearch1 [18]. Hecho sin optimizaciones, el algoritmos nos da fácilmente soluciones para n mucho más grande que lp_solve con la representación del modelo 1.

La idea básica es, representar las variables como un vector y solamente permitir instanciaciones que sean una permutación de n. Con esos "trucos" podemos fijarnos solo en la restricción de las diagonales. La cantidad de violaciones de la restricción para cada reina son almacenadas. Ahora, arbitrariamente, elegimos dos reinas y las intercambiamos. Si la cantidad total de violaciones producidas es menor o igual a la de la solución actual, se mantiene esa como solución actual, si no, se deshace el movimiento.

Además, utilizamos una cota superior que fuerza al algoritmo a terminar aún sin haber encontrado una solución.

En pseudo-código, la implementación sería la siguiente:

```
 n: número de reinas y tamaño de la tabla
crear arreglo filas de largo n
crear arreglo violaciones de largo n
int ejecuciones: las iteraciones
int cota_sup: la cota superior
crear una configuración inicial (permutación de n)
// instanciar el arreglo violaciones
for i<n
    for j+i<n
        si reina i y j están en la misma diagonal
            violaciones[i]++;
            violaciones[j]++;
while(!isEmpty(violaciones) AND ejecuciones <cota_sup)
    elige 2 reinas distintas arbitrariamente
    si swap(i,j) produce menos o igual violaciones
        swap(i,j)
        reinstanciar violaciones (crear el arreglo de nuevo)
    ejecuciones++
```

8. Benchmarks

Como dijimos anteriormente, a lo largo de los años se han desarrollado numerosas formas de plantear y solucionar el problema de las n-reinas. Un Algoritmo de Redes Neuronales utilizando un modelo modificado de Hopfield [16] parece funcionar mejor que otros modelos de redes neuronales presentados, pero sólo se publican resultados para valores de n pequeños. Hynek [14] diseñó una Heurística basada en una fase de preprocesamiento y posteriormente aplica un operador de mutación basado en búsqueda local. Los resultados computacionales muestran que el algoritmo es efectivo y logra soluciones hasta $n = 1000$ en tiempos razonables. Sin embargo, el mejor algoritmo encontrado hasta ahora es la búsqueda local con minimización de conflictos [4, 19]. El algoritmo de Socic y Gu [18] tiene tiempo de ejecución lineal por lo que es extremadamente rápido. En su artículo, ellos muestran que su algoritmo es capaz de obtener una solución para cualquier valor de n, con $n < 1000$, en menos de 0,1 segundo y 55 segundos para $n = 3{,}000{,}000$, en un computador del año 94 (IBM RS 6000/530). Ahora, dada la naturaleza probabilística del algoritmo, no se tiene garantía del tiempo del peor caso del algoritmo, pero en la práctica muestra un excelente desempeño y un comportamiento muy robusto.

9. Resolución

Independiente de lo anterior, se analizaron distintas formas de resolver el problema utilizando técnicas completas de optimización. Las técnicas analizadas fueron Branch and Bound, Branch and Cut, Enumeración Implícita y Algoritmo aditivo, resultando la más adecuada la de Branch and Bound.

9.1. Branch and Bound

Este método es en realidad una variante de Backtracking y se aplica normalmente para resolver problemas de optimización. Branch and Bound, al igual que Backtracking, realiza una enumeración parcial del espacio de soluciones basándose en la generación de un árbol de expansión. Una característica que le hace diferente al diseño anterior es la posibilidad de generar nodos siguiendo distintas estrategias. Recordemos

que Backtracking realiza la generación de descendientes de una manera sistemática y de la misma forma para todos los problemas, haciendo un recorrido en profundidad del árbol que representa el espacio de soluciones.

Branch and Bound en su versión más sencilla puede seguir un recorrido en anchura o en profundidad, o utilizando el cálculo de funciones de costo para seleccionar el nodo que en principio parece más prometedor. Además de estas estrategias, Branch and Bound utiliza cotas para podar aquellas ramas del árbol que no conducen a la solución óptima. Para ello calcula en cada nodo una cota del posible valor de aquellas soluciones alcanzables desde ése. Si la cota muestra que cualquiera de estas soluciones tiene que ser necesariamente peor que la mejor solución hallada hasta el momento no necesitamos seguir explorando por esa rama del árbol, lo que permite realizar el proceso de poda.

Definimos nodo vivo del árbol a un nodo con posibilidades de ser ramificado, es decir, que no ha sido podado. Para determinar en cada momento que nodo va a ser expandido y dependiendo de la estrategia de búsqueda seleccionada, necesitaremos almacenar todos los nodos vivos en alguna estructura que podamos recorrer. Emplearemos un arreglo para almacenar los nodos que se han generado, pero no han sido examinados en una búsqueda en profundidad. Las búsquedas en amplitud utilizan una cola para almacenar los nodos vivos de tal manera que van explorando nodos en el mismo orden en que son creados.

La estrategia de mínimo costo utiliza una función de costo para decidir en cada momento qué nodo debe explorarse, con la esperanza de alcanzar lo más rápidamente posible una solución más económica que la mejor encontrada hasta el momento.

Básicamente, en un Branch and Bound se realizan tres etapas. La primera de ellas, denominada de Selección, se encarga de extraer un nodo de entre el conjunto de los nodos vivos. La forma de escogerlo va a depender directamente de la estrategia de búsqueda que decidamos para el algoritmo. En la segunda etapa, la Ramificación, se construyen los posibles nodos hijos del nodo seleccionado en el paso anterior. Por último, se realiza la tercera etapa, la Poda, en la que se eliminan algunos de los nodos creados en la etapa anterior. Esto contribuye a disminuir en lo posible el espacio de búsqueda y así atenuar la complejidad de estos algoritmos basados en la exploración de un árbol de posibilidades. Aquellos nodos no podados pasan a formar parte del conjunto de nodos vivos, y se comienza de nuevo por el proceso de selección. El algoritmo finaliza cuando encuentra la solución, o bien cuando se agota el conjunto de nodos vivos.

Para cada nodo del árbol dispondremos de una función de costo que nos estime el valor óptimo de la solución si continuáramos por ese camino. De esta manera, si la cota que se obtiene para un nodo, que por su propia construcción debería ser mejor que la solución real (o a lo sumo, igual que ella), es peor que una solución ya obtenida por otra rama, podemos podar esa rama pues no es interesante seguir por ella. Evidentemente no podremos realizar ninguna poda hasta que hayamos encontrado alguna solución. Por supuesto, las funciones de costo han de ser crecientes respecto a la profundidad del árbol, es decir, si h es una función de costo, entonces $h(n) \leq h(n')$ para todo n' nodo descendiente de n.

En consecuencia, y a la vista de todo esto, podemos afirmar que lo que le da valor a esta técnica es la posibilidad de disponer de distintas estrategias de exploración del árbol y de acotar la búsqueda de la solución, que en definitiva se traduce en eficiencia. La dificultad está en encontrar una buena función de costo para el problema, buena en el sentido de que garantice la poda y que su cálculo no sea muy costoso. Si es demasiado simple probablemente pocas ramas puedan ser excluidas. Dependiendo de cómo ajustemos la función de costo mejor algoritmo se deriva. Inicialmente, y antes de proceder a la poda de nodos, tendremos que disponer del costo de la mejor solución encontrada hasta el momento que permite excluir de futuras expansiones cualquier solución parcial con un costo mayor. Como muchas veces no se desea esperar a encontrar la primera solución para empezar a podar, un buen recurso para los problemas de optimización es tomar como mejor solución inicial la obtenida con una técnica incompleta, que no encuentra siempre la solución óptima, pero sí una cercana a ella.

Por último, este Branch and Bound posee la ventaja de poder ser ejecutados en paralelo. Debido a que disponen de un conjunto de nodos vivos sobre el que se efectúan las tres etapas del algoritmo antes mencionadas, nada impide tener más de un proceso trabajando sobre este conjunto, extrayendo nodos, expandiéndolos y realizando la poda. El disponer de algoritmos paralelizables (y estos algoritmos, así como los de Divide y Vencerás lo son) es muy importante en muchas aplicaciones en las que es necesario abordar los problemas de forma paralela para resolverlos en tiempos razonables, debido a su complejidad intrínseca. Sin embargo, todo tiene un precio, sus requerimientos de memoria son mayores que los de los algoritmos Backtracking. Ya no se puede disponer de una estructura global en donde ir construyendo la solución, puesto que el proceso de construcción deja de ser tan "ordenado" como antes. Ahora se necesita que cada nodo sea autónomo, en el sentido que ha de contener toda la información necesaria para realizar los procesos de bifurcación y poda, y para reconstruir la solución encontrada hasta ese momento.

9.2. Resolución del problema de las n-reinas utilizando Branch and Bound

Para construir el árbol de expansión nos basamos en el modelo 2, construyendo un vector solución formado por n enteros positivos distintos, donde el k-ésimo de ellos indica la columna en donde hay que colocar la reina de la fila k del tablero de ajedrez. En cada paso o etapa disponemos de n posibles opciones a priori (las n columnas), pero podemos eliminar aquellas columnas que den lugar a un vector que no sea k-prometedor, esto es, que la nueva reina incorporada amenace a las ya colocadas.

Más formalmente, diremos que el vector S de n elementos es k-prometedor (con $1 \leq k \leq n$) si y solo si para todo par de enteros i y j entre 1 y k se cumple lo siguiente:

$$\begin{aligned} S[i] &\neq S[j] \\ |S[i] - S[j]| &\neq |i - j| \end{aligned}$$

Respecto al proceso de ramificación, cada nodo puede generar hasta n nodos hijos, cada uno con la columna de la reina que ocupa la fila en curso. Sin embargo, se descartan todos aquellos nodos que no den lugar a un vector solución k-prometedor. Asimismo, la poda de los nodos se realiza durante el proceso de ramificación y todos aquellos nodos que se generan son válidos porque o son solución al problema, o conducen a una de ellas.

La medida de lo prometedora que es una situación del tablero o una rama, es una medida heurística y tiene relación con el número de casillas libres del tablero. Es decir, una configuración será mejor cuantas más casillas no alcanzables por las reinas colocadas en el tablero existan. En la figura 5 se presenta un ejemplo de lo anteriormente descrito.

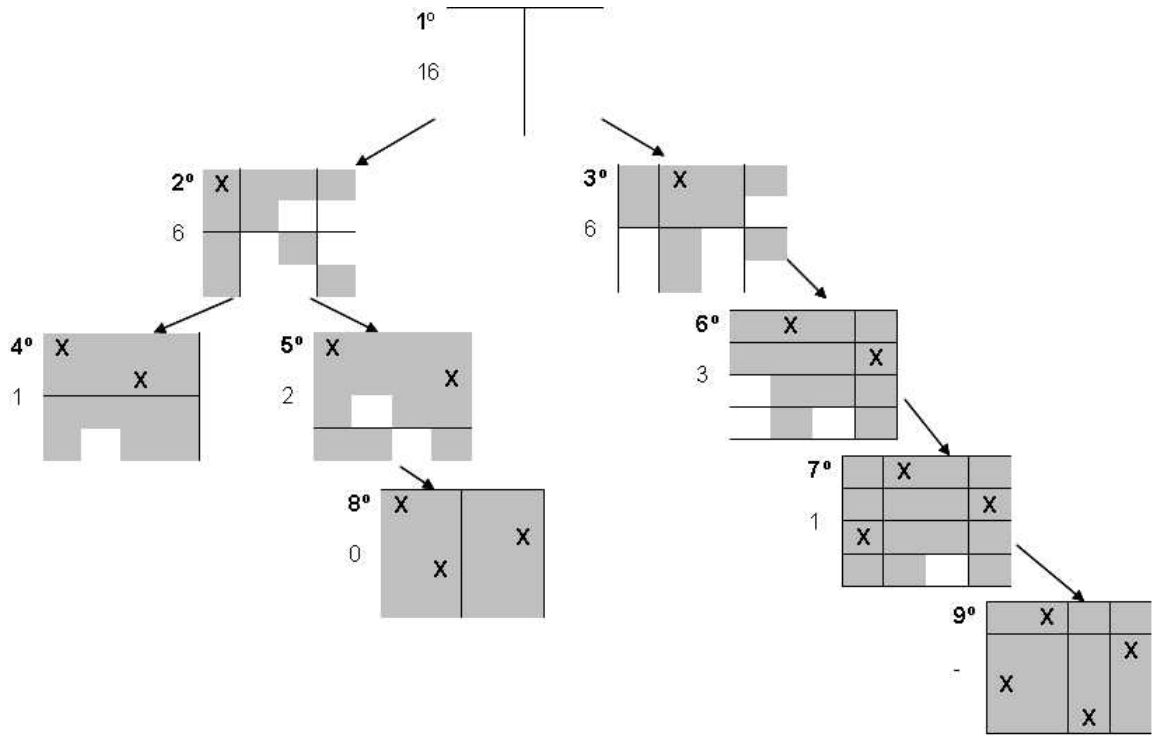

Figura 5: Recorrido del árbol en el problema de las 4 reinas.

Las pruebas realizadas consideraron dos formas de recorrido del árbol: en profundidad y en anchura. Lo anterior se efectuó con el objeto de comparar el comportamiento del algoritmo ante un mismo problema,pero con diferente forma de recorrer el árbol.

10. Análisis de resultados

Los resultados que hemos obtenido utilizando análisis en profundidad y en anchura hasta conseguir encontrar la primera solución del problema son los siguientes:

n	**Tiempos** (en minutos)	
	profundidad	**anchura**
8	0:05	0:53
10	0:15	1:47
15	0:32	4:29
20	0:51	11:34
25	2:05	27:08
30	4:27	>30:00
40	8:13	
45	17:58	
48	>30:00	

Cuadro 2: Búsqueda de la primera solución

Como puede apreciarse en la tabla 2, el recorrido en profundidad del árbol es el más adecuado ya que los tiempos de ejecucion son bastante menores que los del recorrido en anchura.

Por otro lado, es normal que el análisis en anchura tenga que analizar tantos nodos, pues hasta no llegar al último nivel del árbol no encuentra la solución. Obsérvese, además, cómo tiene que analizar todos los nodos hasta el nivel $n-1$ y generar casi todos los nodos del nivel n antes de encontrar la primera solución.

También es fácil analizar cómo se comportan una y otra estrategia cuando lo que le pedimos es que calculen todas la soluciones y no se detengan al encontrar la primera. Los resultados obtenidos se muestran en la tabla 3.

n	**Tiempos** (en minutos)	
	profundidad	**anchura**
8	1:27	1:27
10	2:51	2:51
15	8:33	8:33
20	19:41	19:41
25	>30:00	>30:00
30		
40		
45		
48		

Cuadro 3: Búsqueda de todas las soluciones

Para este caso ambas estrategias (análisis en profundidad y en anchura) obtienen los mismos resultados antes de encontrar todas las soluciones que posee el problema para el n determinado, pues ambas deben recorrer todo el árbol.

11. Perspectivas de Investigación

Sería deseable ahondar más en los métodos de búsqueda local que tan buenos resultados han entregado, basándonos en los trabajos de [19, 4, 18] y con ello poder optimizar el algoritmo presentado en el modelo 2. Asimismo, sería recomendable buscar una mejor representación del modelo 2 para el manejo de violaciones. Además, se podría modificar el algoritmo para que encuentre más de una o todas las soluciones, ante la necesidad de contar con más alternativas en caso que existan costos involucrados. Por ejemplo, ante la necesidad de determinar la posición donde deben instalarse unas antenas, tal que no se interfieran entre ellas, puede darse el caso que una solución sea más costosa que otra, por lo que se puede asociar otra función objetivo de minimización de costos.

Referencias

[1] Anonymous. Unknown. *Berliner Schachgesellschaft*, 3:363, 1848.

[2] Franz Nauck. Schach. *Illustrierter Zeitung*, pages 361–352, 1850.

[3] J.W.L. Glaisher. *Philosophical Magazine*, 18:457–467, 1874.

[4] Rok Sosic and Jun Gu. Efficient local search with conflict minimization: A case study of the n-queens problem. *Knowledge and Data Engineering*, 6(5):661–668, 1994.

[5] Alfredo Candia Véjar and Cesar Astudillo Hernández. Algoritmos para el problema de las n-reinas. In Mauricio Solar, David Fernández-Baca, and Ernesto Cuadros-Vargas, editors, *30ma Conferencia Latinoamericana de Informática (CLEI2004)*, pages 160–167. Sociedad Peruana de Computación, September 2004. ISBN 9972-9876-2-0.

[6] I.Vardi I.Rivin and P. Zimmermann. The n-queens problem. *The American Mathematical Monthly*, 101:629–639, 1994.

[7] K.D. Crawford. Solving the *n*-queens problem using genetic algorithms. *In Proceedings ACM/SIGAPP Symposium on Applied Computing, Kansas City*, pages 1039–1047, 1992.

[8] B. Abramson and M. Yung. Divide and conquer under global constraints: A solution to the n-queens problem. *J. Parallel Distrib. Comput.*, 6(3):649–662, 1989.

[9] B. Bernhardsson. Explicit solution to the n-queens problems for all *n*. *ACM SIGART Bulletin*, 2(7), 1991.

[10] H.S. Stone and J.M. Stone. Efficient local search with conflict minimization: A case study of the n-queens problem. *IBM J. Res. Develop.*, 30(3):242–258, 1986.

[11] L.V.Kalé. An almost perfect heuristic for the n nonattacking queens problem. *Information Processing Lettes.*, 66:375–379, 1992.

[12] A.E. Eiben, P.-E. Raué, and Zs. Ruttkay. Solving constraint satisfaction problems using genetic algorithms. In *Proceedings of the 1st IEEE World Conference on Computational Intelligence*, pages 542–547. IEEE Service Center, 1994.

[13] Abdollah Homaifar, Joseph Turner, and Samia Ali. The *n*-queens problem and genetic algorithms. In *Proceedings IEEE Southeast Conference, Volume 1*, pages 262–267, 1992.

[14] J. Hynek. The n-queens problem revisited. *In Proceedings of the ICSC, Symposia on Intelligent Systems and Applications ISA 2000. ICSC Academic Press.*, 2000.

[15] L.R. Foulds and D.G. Johnson. An application of graph theory and integer programming: Chessboard nonattacking puzzles. *Mathematical Magazine*, 57:95–104, 1984.

[16] I.N. da Silva A.N. Souza and M.E. Bordon. A modified hopfield model for solving the n-queen problem. *IJCNN*, pages 509–514, 2000.

[17] D.M.Gay R.Fourer and B.W.Kernighan. Ampl: A modeling language for mathematical programming. *Duxbury Press, Brooks, Cole Publishing Company*, 2002.

[18] R.Sosic and J.Gu. 3.000.000 queens in less than one minute. *SIGART Bulletin*, 2(2):22–24, 1991.

[19] J.Gu R.Sausic. Fast search algorithms for n-queens problem. *IEEE transactions on Systems, Man, and Cybernetics*, 21(6):1572–1576, Nov/Dec 1991.