Algoritmos para el problema de las n-reinas

Alfredo Candia Véjar
Universidad de Talca, Dpto. de Ingeniería de Sistemas,
Curicó, Chile
acandia@utalca.cl

and

César Astudillo Hernández
Universidad de Talca, Dpto. de Ingeniería de Sistemas,
Curicó, Chile
castudillo@utalca.cl

Abstract

The n-queens problem is to find the different ways to assign n nonattacking queens in a $n \times n$ chessboard. This work analizes the application of an algorithm based on Local Search for the resolution of the n-queens problem.

Empirical results show that, large size instances of the problem are well solved by the Local Search algorithm in comparison to more sofisticated algorithms like Genetic Algorithms.

Keywords: n-queens problem, Local search.

Resumen

El problema computacional de las n-reinas consiste en encontrar las diferentes formas de asignar n reinas a un tablero $n \times n$, de manera que éstas no se ataquen. Este trabajo analiza la aplicación de algoritmos basados en Búsqueda Local para la resolución del problema de las n-reinas.

La experimentación computacional efectuada con instancias de gran tamaño muestra que se obtienen interesantes resultados con el algoritmo de Búsqueda Local en comparación con algoritmos más sofisticados tal como Algoritmos Genéticos.

Palabras claves: Problema de las n-reinas, Búsqueda Local.

1. INTRODUCCION

El problema de las n-reinas (**N-Q**) consiste en encontrar una asignación de n reinas a un tablero $n \times n$ de modo que éstas no se ataquen. El problema (para $n = 8$) fue propuesto en el año 1848 en un trabajo anónimo [2], y que posteriormente fue atribuído a Max Bezzel. La publicación detallada más antigua que se conoce fue realizada por Nauck en 1850 [15]. Ese mismo año, según unas cartas publicadas en 1929, Gauss conjeturó que existían 72 soluciones para $n = 8$; en el año 1874, Glaisher [7] probó que existían en realidad 92 soluciones. La investigación sobre este tema no ha cesado hasta hoy y luego existe una amplia variedad de algoritmos sugeridos para su resolución.

N-Q tiene dos versiones. La más simple consiste en encontrar exactamente una solución al problema para un valor dado de n. La versión más difícil consiste en encontrar todas las soluciones para un valor dado de n. Notamos que N-Q tiene una solución para $n = 1$, no tiene para $n = 2$ y $n = 3$ y tiene dos soluciones para $n = 4$ (ver fig. 1). Para mayores valores de n, N-Q tiene una función estrictamente creciente de soluciones y más aún su crecimiento es exponencial siguiendo un análisis empírico. Algunos matemáticos han estudiado el problema de encontrar todas las soluciones y hasta ahora no han encontrado una fórmula cerrada para tal problema de enumeración. La Tabla 1 ilustra el número total de soluciones $Q(n)$ para $4 \leq n \leq 20$ [16].

Numerosos algoritmos se han diseñado para N-Q tales como backtracking, algoritmos genéticos, búsqueda local con resolución de conflictos, programación entera, dividir para conquistar y redes neuronales, entre las más conocidas. N-Q pertenece a la clase de problemas computacionales NP-completos [4], pero se resuelve

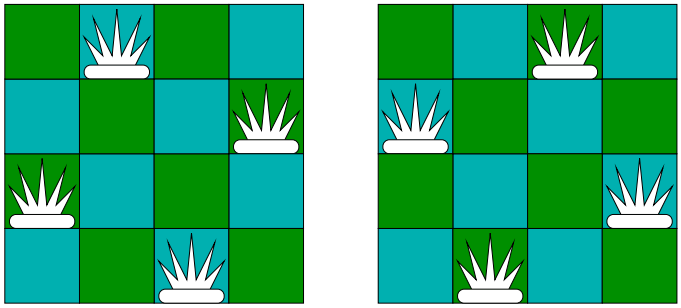

Figura 1: Soluciones para n=4

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

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

fácilmente, en tiempo polinomial observado, cuando se requiere solamente una solución [17]. Existen soluciones analíticas para N-Q donde se da una fórmula explícita para la localización de las reinas o bien se encadenan soluciones obtenidas para valores menores de n [3][1]. El problema con estas soluciones es que generan un número muy pequeño de soluciones. Backtracking, en general es ineficiente y para N-Q no es fácil encontrar soluciones para $n > 100$ en tiempos razonables, ver Stone y Stone [18], sin embargo, Kalé [13] diseñó 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 a N-Q [5], [4], [9], [10], pero se han visto ineficientes aún cuando se hayan probado diversas representaciones del problema y diversos operadores. Algoritmos para Programación Entera son típicamente exponenciales y consecuentemente consiguen resolver problemas de tamaño muy pequeño [6]. Un algoritmo de redes neuronales utilizando un modelo modificado de Hopfield [11] parece funcionar mejor que otros modelos de redes neuronales presentados [14][12] , pero se reportan resultados solamente para valores muy pequeños de n. Búsqueda local con minimización de conflictos [17] parece ser el mejor algoritmo encontrado hasta ahora. Este algoritmo de Sosic y Gu tiene tiempo de corrida (observado) lineal y luego es extremadamente rápido. En su artículo, 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 una máquina rápida del año 94 (IBM RS 6000/530). Dada la naturaleza probabilística del algoritmo, no se tiene una garantía del tiempo del peor caso del algoritmo pero exhibe en la práctica un excelente desempeño y un comportamiento muy robusto. Hynek [10] 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.

N-Q es conocido usualmente como un problema relacionado a un juego y también como un problema apropiado para probar algoritmos nuevos. Sin embargo, N-Q tiene también algunas aplicaciones; de hecho, se le considera como un modelo de máxima cobertura. Una solución a N-Q garantiza que cada objeto se puede

accesar de cualquiera de sus ocho direcciones vecinas (dos verticales, dos horizontales, y cuatro diagonales). Algunas aplicaciones posibles son: control de tráfico aéreo, sistemas de comunicación modernos, programación de tareas computacionales, procesamiento paralelo óptico, compresión de datos y balance de carga en un computador multiprocesador, ver [17] para mayores detalles.

La sección 2 discute representaciones para el problema, la definición de posibles vecindades a usar en algoritmos basados en búsqueda local y la definición de la función de costo a utilizar. La sección 3 describe un algoritmo de búsqueda local simple para N-Q y se discuten resultados computacionales para diversos valores de n. Adicionalmente, se discute la problemática de obtener un conjunto de soluciones diferentes mediante la aplicación repetida de este algoritmo y se compara con el algoritmo aleatorizado de Gu [17]. Finalmente, comentamos algunas mejorías al algoritmo propuesto y también presentamos una extensión del problema.

2. REPRESENTACION, VECINDADES Y COLISIONES

Una representación natural para N-Q es aquella de utilizar matrices para representar el tablero $n \times n$. En este caso, un elemento de la matriz es un número $a_{ij}, 1 \leq i, j \leq n$, donde $a_{ij} \in \{0,1\}$, $a_{ij} = 1$, si una reina se localiza en la posición i, j, y $a_{ij} = 0$ en caso contrario. Evidentemente que si localizamos al azar n reinas representadas por el valor 1 en la matriz A, entonces pueden aparecer colisiones horizontales, verticales y diagonales. Con el objetivo de trabajar solamente con colisiones diagonales, y dado que queremos minimizar el número de colisiones (una solución debe tener 0 colisiones) es más usada la representación denominada *permutación*. En una permutación, usamos un vector de longitud n, donde cada posición j $(1 \leq j \leq n)$, contiene un número a_j indicando que existe una reina en la posición (j, a_j). Por ejemplo, para $n = 4$, el vector $(2, 4, 1, 3)$ indica que existe una reina en la posición $(1, 2)$, otra en la posición $(2, 4)$, otra en la posición $(3, 1)$, y la última en la posición $(4, 3)$. De esta forma, la representación hace uso del concepto de permutación de los números 1 al n, y con la propiedad que solamente podemos encontrar colisiones diagonales disminuyendo así fuertemente el espacio solución. La cantidad de permutaciones de los números 1 al n crece exponencialmente con el tamaño de n, pero lo sorprendente es que la cantidad de soluciones, esto es, permutaciones con 0 colisiones, también crece a un ritmo exponencial aunque obviamente menor que la cantidad de permutaciones, como se ilustró en la introducción.

Estamos interesados en diseñar algoritmos de búsqueda local para N-Q. Un elemento fundamental en búsqueda local es la definición de un esquema de vecindades ya que el algoritmo se basa en explorar el espacio de soluciones siguiendo una trayectoria definida por puntos obtenidos en una vecindad de la solución actual. El algoritmo se detiene cuando no es posible encontrar en la vecindad del punto actual un elemento de mejor calidad.

Para el problema N-Q varios esquemas de vecindades se podrían establecer. En general, existe un compromiso en la definición del tamaño de la vecindad dado que vecindades de gran tamaño pueden generar puntos de buena calidad pero a un costo computacional alto; recíprocamente, vecindades reducidas serán examinadas rápidamente pero podría ser difícil encontrar mejores soluciones; de esta forma, normalmente se eligen vecindades que no consuman demasiado tiempo en la búsqueda pero que generen buenas expectativas de encontrar soluciones mejores.

Nuestra proposición de vecindad es muy simple. Dada una n permutación, se trata de escoger aleatoriamente dos elementos distintos del n-vector e intercambiarlos; esto es equivalente a intercambiar dos columnas en la disposición de un tablero dado. Por ejemplo, para $n = 5$, un tablero particular lo representamos por $(2, 3, 1, 4, 5)$. Si intercambiamos las columnas 2 y 4 obtenemos el vector $(2, 4, 1, 3, 5)$. La figura 2 ilustra esta operación. Notamos que, en este caso, el vector resultante es una solución al problema de las 5 reinas.

En búsqueda local, necesitamos evaluar el costo de las soluciones. Este costo estará dado por el número de colisiones entre reinas en un tablero particular. En el ejemplo anterior, el vector $(2, 3, 1, 4, 5)$ tiene colisiones, en particular la reina en la columna 1 colisiona con la reina en la columna 2 y la reina en la columna 4 colisiona con aquella en la columna 5, y luego existen dos colisiones. Luego, en general, dado un n-vector interesa calcular en forma rápida el número de colisiones que posee. Crawford [4] diseñó la siguiente estrategia, se construyen dos arreglos llamados positivo y negativo. Cada uno de estos arreglos tiene un tamaño de $2n - 1$ elementos, uno por cada diagonal ya sea positiva o negativa, y cada uno de estos elementos contabiliza, para un tablero dado, el número de reinas que se encuentra en esa diagonal. Cada vez que una reina se posiciona en la fila i y columna σ_i, un contador se incrementa en la posición $n + i - \sigma_i$ del arreglo negativo para contabilizar esta reina. Análogamente, para la posición $i + \sigma_i - 1$ del arreglo positivo. Si $\pi = \langle \sigma_1, ..., \sigma_n \rangle$ es una permutación denotamos por π^- al arreglo negativo y por π^+ al arreglo positivo, de modo que el número total de colisiones está dado por:

$c(\pi) = \sum_{i=1}^{2n-1} [(\pi_i^+ - 1) + (\pi_i^- - 1)]$, donde se considera para las sumas solamente aquellos valores $\pi_i > 1$.

Esta forma de evaluación del número de colisiones asociadas a una permutación se puede implementar en tiempo $O(n)$ [4].

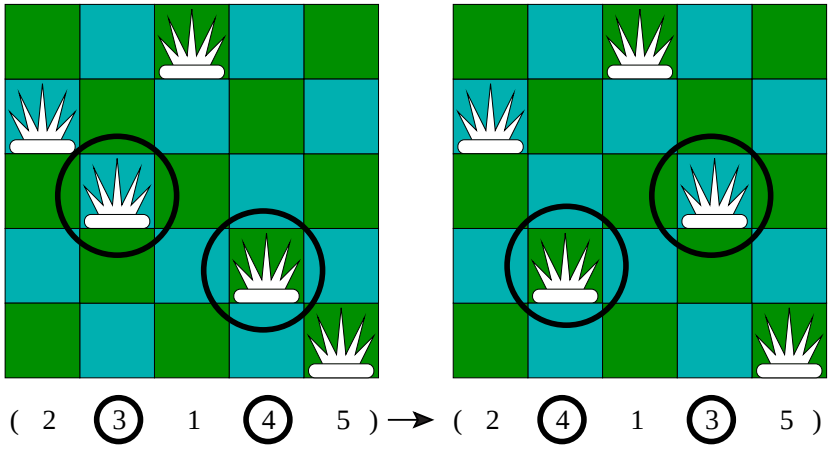

Figura 2: Ejemplo de vecindad para n=5

Notamos que diferentes permutaciones pueden tener diferente costo asociado, luego el objetivo de resolver el problema N-Q se traduce, entonces, en encontrar un valor de π tal que $c(\pi) = 0$. Es decir, una solución a N-Q es cualquier permutación de costo nulo. Es interesante comentar que N-Q, con esta formulación, es un problema de optimización discreta con valor óptimo conocido y tal que existe un número de soluciones que crece exponencialmente con n, para un n dado.

Existen varios algoritmos modernos que utilizan búsqueda local tales como simulated annealing, algoritmos genéticos o tabu search y ya se comentó en la introducción de su aplicación a N-Q. Sin embargo, discutiremos a continuación una heurística de búsqueda local simple.

3. BUSQUEDA LOCAL SIMPLE PARA N-Q

Búsqueda Local (BL) ha demostrado, durante varias décadas, su poder como un algoritmo muy simple y rápido que permite resolver, en forma aproximada, algunos problemas de optimización de alta complejidad. Uno de los casos más importantes es su aplicación al problema del Vendedor Viajero, donde heurísticas basadas en BL han logrado resolver instancias con miles de ciudades. Adicionalmente, este algoritmo también sirve como base para el diseño de metaheurísticas que en los últimos veinte años han provocado un impacto mayor en la resolución de problemas complejos, ver Glover y Kochenberger [8], para una serie de artículos sobre aspectos teóricos y prácticos ligados a BL.

El algoritmo BL parte de un punto inicial, típicamente generado aleatoriamente, y recorre parte del espacio solución generando en cada iteración un vecino del punto actual que tenga un costo menor (*mejoría iterativa*) o bien explorando completamente la vecindad de modo de alcanzar el vecino de menor costo (*máximo descenso*). El algoritmo se detiene cuando, a partir del punto actual no es posible encontrar, en su vecindad, un punto de costo menor. La salida del algoritmo es un punto factible cuyo costo representa una cota superior (para un problema de mínimo) para el valor óptimo del problema. BL tiene algunas variantes que permiten eventualmente mejorar su desempeño tales como partidas múltiples y considerar varias vecindades.

Para N-Q, BL partirá con una permutación aleatoria y repetidamente intentará encontrar un vecino mejor para la solución actual. Un punto crucial entonces será definir la vecindad. Una forma simple consiste en intercambiar dos elementos elegidos al azar del vector permutación, lo cual podría resultar en una disminución del número de colisiones. Dado que esto podría no ocurrir, necesitamos seguir buscando dentro de la vecindad hasta encontrar una permutación mejor. En el peor caso, podríamos explorar completamente la vecindad y no tener éxito; en este caso, la búsqueda termina y da como salida la permutación actual. Notamos que si buscamos completamente la vecindad entonces el número de búsquedas por un vecino mejor es igual al número de pares diferentes de componentes que podemos elegir en la n-permutación, o sea el tamaño de la vecindad es $\binom{n}{2} = O(n^2)$. Nuestra implementación considera un máximo de n^2 intentos notando que, dado el caracter aleatorio de las elecciones, es posible que existan repeticiones en la búsqueda de un vecino mejor. Luego, la regla de detención será la detección de una permutación con costo nulo, o bien alcanzar el máximo número de intentos n^2. La figura 3 ilustra el algoritmo de búsqueda local propuesto (BLNQ).

Con el objetivo de acelerar la búsqueda, también experimentamos con L =Máximo número de intentos igual a $O(n)$, en particular, $L = 3n$. La tabla 2 muestra los resultados obtenidos por el algoritmo BLNQ, en términos de colisiones promedio sobre 100 ejecuciones, y considerando $L = n^2$ y $L = 3n$.

```
PROCEDURE BLQ()
BEGIN
intentos :=  0;
L := n^2;
x := permutacion(n);
cx := costo(x);
IF (cx=0) THEN
        RETURN;
WHILE(b=false)
        BEGIN
        y := vecino(x);
        cy := costo(y);
        IF (cy < cx) THEN
                BEGIN
                x := y;
                cx := cy;
                IF(cx=0) THEN
                        b:=true;
                ELSE
                        intentos := 0;
                END
        ELSE
                BEGIN
                intentos := intentos + 1;
                IF (intentos > L OR cx = 0) THEN
                        b := true;
                END
        END
END
```

Figura 3: algoritmo BLNQ

n	$L = n^2$	$L = 3n$
10	0,97	1,46
20	1,27	2,39
30	1,14	3,19
40	1,06	4,06
50	0,97	4,47
60	0,9	5,35
70	1	5,54
80	0,77	6,14
90	0,93	6,37
100	0,91	7,24
200	0,31	10,04
300	0,17	12,72
400	0,05	14,22
500	0,02	16,38
600	0,03	17,08
700	0,01	18,74
800	0	18,81
900	0	20,31
1000	0	20,73
2000	0,00	26,5
3000	0,00	29,4
4000	0,00	31,6
5000	0,00	33,0
6000	0,00	32,2

Tabla 2: Número promedio de colisiones tras 100 ejecuciones, considerando $L = n^2$ y $L = 3n$

Además probamos con factores mayores que 3n, en particular $L = 10n$, pero el costo final no mejoró sustancialmente. Para $L = 3n$, notamos que BLNQ no funciona bien aumentando el número de colisiones con el tamaño de n. Sin embargo, para $L = n^2$ BL converge rápidamente a encontrar soluciones y, a partir de $n = 700$, las 100 ejecuciones obtienen una solución.

También realizamos experimentos con BL pero usando una representación matricial y con una vecindad de tamaño mayor, donde dada una configuración de reinas en el tablero obtenemos un vecino simplemente cambiando una reina a otra posición desocupada dentro del tablero.

Los resultados obtenidos fueron muy pobres dando origen a un número creciente de colisiones con el tamaño de n.

BLNQ usando $L = n^2$ se ve muy robusto en relación a su desempeño en función del tamaño de la entrada n. La tabla 3 muestra información adicional relacionada con el desempeño del algoritmo. En particular, para cada n, se muestran los resultados promedios (sobre 100 corridas) para el costo inicial, el costo final y la cantidad de mejorías alcanzadas en la búsqueda. Los resultados verifican la idea que los cambios locales permiten disminuir levemente el costo y, en promedio, se tiene poco más de una unidad en el costo por mejoría lograda.

Diversos trabajos se han realizado implementando algoritmos genéticos [4, 9, 10]. Sin embargo, a pesar de probar con una serie de operadores, resulta muy dificil de encontrar soluciones en tiempos razonables. Se pueden generar tableros con pocas colisiones muy rápidamente, pero llegar a una solución se puede tornar un proceso lento.

Realizamos experimentos con algoritmos genéticos probando diversas variaciones tales como operadores de cruzamiento, mutación, tamaño de la población, obteniendo resultados muy similares a los mencionados por Crawford [4] y ratificados por Hynek [10], respecto al hecho que el cruzamiento entre dos tableros no necesariamente generan un hijo de mejor calidad, obligando a reducir las potencialidades de un algoritmo genético a la capacidad de la componente mutación.

Esto último puede justificar, en parte, que la estrategia de algoritmos genéticos sea superada por una estrategia más simple de BL.

El desempeño de BLNQ hace recordar el algoritmo de Sosic y Gu (SG) [17], un algoritmo probabilístico muy rápido, que privilegia una fase de construcción de una solución, y posteriormente, una fase de reparación de la solución parcial encontrada en la primera fase. En términos de la obtención de soluciones, los dos algoritmos, SG y BLNQ tienen un desempeño similar en el sentido que, a partir de un cierto valor de n (n=700), alcanzan una solución con posibilidad muy cercana a 1.

n	Costo Inicial	Vecinos Analizados	Mejoras Realizadas	Costo Final
10	5,3	140,4	3,5	0.97
20	10,2	475,6	7,6	1.27
30	14,4	1099,1	10,7	1.14
40	21,6	1863,8	17,5	1.06
50	26,5	3103,2	21,9	0.97
100	55,6	12098,5	45,2	0.91
200	104,4	35430,1	88,7	0.31
300	158,2	39866,8	130,5	0.17
400	212,3	86344,7	177,4	0.05
500	262,7	94892,2	218,9	0.02
600	307,5	167051,5	259,3	0.03
700	367,9	120751	312,3	0.01
800	424,5	148108,4	353,46	0
900	476,4	171108,5	399,7	0
1000	530,5	172121,3	444,8	0

Tabla 3: Valor promedio para los distintos valores de N usando BL, considerando 100 corridas y $L = N^2$

Adicionalmente, estudiamos el problema de obtener un conjunto de soluciones diferentes para N-Q. Realizamos experimentos con SG y BLNQ concluyendo que cuando estos algoritmos alcanzan soluciones, todas éstas son diferentes. Esta diversidad alcanzada es muy interesante ya que no es posible observarla con los algoritmos explícitos comentados en la sección 2.

n	50	100	200	300	400	500	600	700	800	900	1000	2000
Soluciones	0	6	13	19	20	19	20	20	20	20	20	20

Tabla 4: Soluciones encontradas para el algoritmo de Gu después de 20 pruebas para cada valor de n

4. CONCLUSIONES

Hemos diseñado un algoritmo de búsqueda local para N-Q, el primero de este tipo según nuestro conocimiento. El algoritmo tiene complejidad computacional exponencial en el peor caso, como es típico para esta clase de algoritmos, pero tiene tiempo de corrida observado $C = n^3$, donde C es una constante aproximada al valor 1/200,000. Del punto de vista de la calidad de la solución, el desempeño del algoritmo es claramente superior al de algoritmos genéticos propuestos, y podría ser aún mejorado tanto en tiempo de ejecución (por ejemplo, calculando el costo en tiempo constante), como en el aumento de la probabilidad de obtención de soluciones para valores de n, $n \leq 700$ (por ejemplo, haciendo L depender de n).

5. AGRADECIMIENTOS

Los autores agradecen los comentarios de los revisores y, en particular, el comentario referido a la complejidad computacional del algoritmo propuesto.

Referencias

[1] B. Abramson and M. Yung. Divide and conquer under global constraints: A solution to the n-queens problems. *J. Parallel Distrib. computing*, 6:649–662, 1989.

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

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

[4] 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.

[5] 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.

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

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

[8] F. Glover and G. Kochenberger. *Handbook of Metaheuristics*. Kluwer, 2002.

[9] 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.

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

[11] A.N. Souza I.N. da Silva and M.E. Bordon. A modified hopfield model for solving the n-queen problem. pages 509–514. IJCNN'00, 2000.

[12] S. Lee and J. Park. Dual-mode dynamics neural networks for combinatorial optimization, 1994.

[13] L.V.Kalé. An almost perfect heuristic for the n nonattacking queens problem. *Information Processing Letters*, 34:173–178, 1990.

[14] J. Mańdziuk and B. Macukow. A neural network designed to solve the n-queens problem. *Biological Cybernetics*, 66:375–379, 1992.

[15] Franz Nauck. Schach. *Illustrierter Zeitung*, 361:352, 1850.

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

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

[18] H.S. Stone and J.M. Stone. Efficient search techniques - an empirical study of the n-queens problem. *IBM J. Res. Develop.*, 30(3):242–258, 1986.