Universidad Católica del Norte
Departamento de Ingeniería de Sistemas y Computación
Magíster en Ingeniería Informática

TAREA 2:
Ocho Reinas
Resolución de Problemas Basados en Búsqueda

15 de Mayo, 2014

Fecha de Entrega:	15 de Mayo, 2014
Autor:	Exequiel Fuentes Lettura
Asignatura:	Sistemas Inteligentes
Profesor:	Juan Bekios-Calfa

1. Introducción

El problema de las 8 Reinas fue propuesto por el entusiasta ajedrecista Max Bezzel en 1848. Este consiste en situar 8 Reinas en un tablero de ajedrez de 8x8 sin que ninguna Reina se ataque [1, pág 71]. Una Reina ataca a otra Reina cuando se encuentra en la misma fila, columna o en diagonal. La figura 1 muestra los posibles movimientos de una Reina.

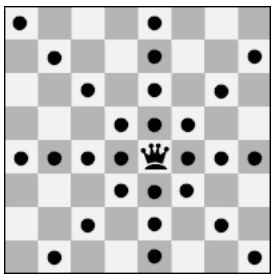

Figura 1: Movimientos de una Reina

Entonces, dado un tablero de 8x8 debemos situar las 8 Reinas sin que ninguna ellas coincida en fila, columna o en diagonal, la figura 2 muestra una de las 92 soluciones al problema de las 8 Reinas.

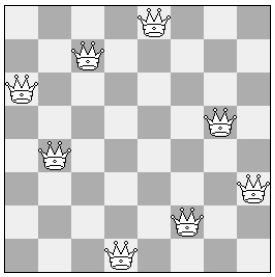

Figura 2: Una de las 92 soluciones al problema

El estado inicial es un tablero vacío, la inserción de la primera Reina no requiere comparaciones, a diferencia de las siguientes inserciones que requieren verificar la regla del juego. La figura 3 muestra el árbol de tableros que se va generando en la medida que se va insertando una Reina en el tablero. Nótese que cada nivel representa todas las posibles posiciones de las Reinas. Si una configuración de un tablero no es válido este será eliminado. Un tablero hoja con las 8 Reinas con posiciones validas será una solución.

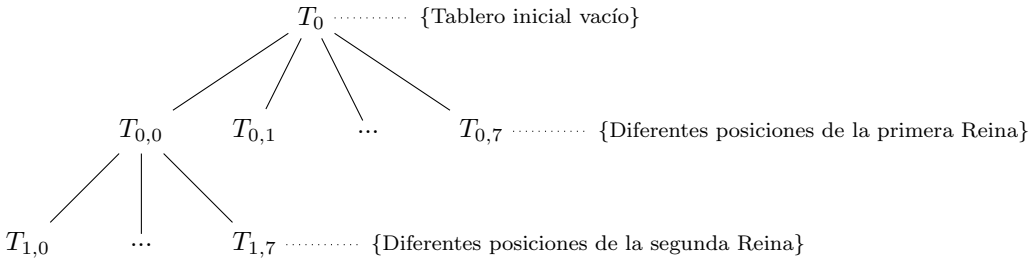

Figura 3: Árbol de tableros. Índices indican posición de la Reina en el tablero

Este problema puede ser resuelto a través de varias metodologías. Este estudio particularmente utiliza tres algoritmos de búsqueda para localizar las soluciones: 1. Breadth-First Search, 2. Depth-First Search y 3. Iterative Deepening Depth-First Search. Cada uno de estos algoritmos de búsqueda utiliza una estructura de datos especifica para buscar la propiedad deseada en el tablero.

1.1. Objetivos

Primer Objetivo

El objetivo principal de este estudio es modelar y formular una solución al problema de las 8 Reinas usando los algoritmos de búsqueda seleccionados.

Segundo Objetivo

El segundo objetivo es determinar la eficiencia de los algoritmos implementados.

2. Métodos implementados

Primero, el algoritmo de búsqueda debe ser diseñado para buscar un tablero que satisfaga las condiciones del juego, esto es que ninguna de las 8 Reinas pueda atacarse. Por ejemplo, R_1 inicialmente no ataca a ninguna pieza porque es la única en el tablero, entonces se inserta en cualquier columna de la primera fila. Cuando se quiere insertar R_2 primero se debe verificar en que columna de la segunda fila puede insertarse para cumplir

con la regla, eso debería ser algo mas o menos sencillo. Ahora, insertar la siguiente R_3 en la tercera fila es algo mas difícil, ya que no debe atacar a R_1 y R_2. Es posible que al insertar R_3 y/o las siguientes piezas sea imposible con la configuración del tablero actual, entonces se debe volver al estado en que las Reinas en el tablero son válidas descartando la configuración no válida. Considerando lo anterior, los siguientes métodos de búsqueda serán utilizados.

2.1. Algoritmos de Búsqueda

En general, un algoritmo de búsqueda toma un estado inicial y retorna una solución del espacio de búsquedas. El espacio de búsquedas es un conjunto de todas las posibles soluciones. Para encontrar el estado final debemos recorrer desde un estado inicial hasta el estado final de meta, a este camino se le denomina espacio de estados. Es útil pensar en la búsqueda como la construcción de un árbol como muestra la figura 3. Entonces, básicamente la solución en pseudocódigo usando algoritmos de búsqueda sería:

```
# Creacion del tablero inicial
estado_inicial <- tablero.nuevo()
# Dependiendo del tipo de algoritmo de b squeda que se use
# debe ser elegida la estructura de datos
estados_abiertos <- cola.nuevo() # pila.nuevo()
While length(estados_abiertos) != 0
    tablero_actual <- estados_abierto.pop()
    If tablero_actual <- es valido?
        Imprimir <- tablero_actual
    Else
        nuevos_tableros <- generar_nuevos_tableros()
        estados_abiertos <- agregar nuevos_tableros
```

2.1.1. Definición formal del problema

Primero, definamos nuestras reglas:

a. 2 Reinas no pueden estar en la misma fila

b. 2 Reinas no pueden estar en la misma columna

c. 2 Reinas no pueden estar en la misma celda

d. 2 Reinas no pueden estar en la misma diagonal

Las reglas a, b y c pueden ser formalmente definidas como:

$$\forall i, j; \quad R_i \neq R_j$$

Y la regla d puede ser formalizada:

$$\forall i, j; SiR_i = xyR_j = y$$
$$Entoncesi - j \neq y - x$$

Ahora, el tablero inicial debe ser vacío, por lo tanto:

$$T_0 = \emptyset$$

Un tablero que cumpla con todas las reglas debe ser de la siguiente forma:

$$T_f = R_1, R_2, \ldots, R_n$$
$$Donde 1 \leq n \leq 8 y f es nuestro tablero final$$

Entonces, nuestro espacio de soluciones se definiría como:

$$S = T_{f1}, T_{f2}, \ldots, T_{fk}$$

Donde k son todas las soluciones válidas

Estado Inicial

Como se ha mencionado, el estado inicial será un tablero vacío, esto es 0 Reinas en el tablero. Un tablero lleno será un tablero con las 8 Reinas en una posición valida.

Acciones

Tenemos las siguientes acciones definidas para este problema:
1. Insertar Reinas si esta cumple con las reglas definidas anteriormente.
2. Generar los siguientes sucesores.

Modelo de transición

Para el estado inicial, siguiendo la acción 1, tendremos la primera Reina insertada en cualquiera columna de la fila 1. Entonces, tendremos un tablero con una Reina. Ahora, el siguiente paso es la generación de los sucesores. Nuestro modelo deberá retornar todos los posibles tableros donde la segunda Reina haya sido insertada en la segunda fila siguiendo las reglas del juego. Y así sucesivamente hasta conseguir el objetivo.

Prueba de meta (objetivo)

El objetivo será alcanzado cuando se haya insertado las 8 Reinas en el tablero. Nótese que al llegar al tablero con las 8 Reinas ya insertadas este será válido porque el modelo de transición permite sólo insertar una Reina si esta cumple con las reglas del juego.

Costo del Camino

El costo del camino puede ser medido en término de movimientos porque no tenemos información adicional de los costos asociados a búsqueda. Entonces, podríamos asignarle el valor 1 a cada movimiento. Este valor puede contarse cada vez que se genere los nuevos sucesores.

2.1.2. Explicación de los algoritmos de búsqueda

El problema se resolvió a través de 3 algoritmos de búsqueda:

1. Breadth-First Search
Comienza con el tablero raíz, en este caso un tablero vacío. Luego, se exploraría sus nodos vecinos, pero como es la raíz entonces descendería a través de sus hijos. A continuación para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el árbol. Esto es realizado a través de una cola (FIFO). Es una búsqueda exhaustiva, esto significa que termina hasta recorrer todo el árbol.

2. Depth-First Search
Al igual que Breadth-First Search, este algoritmo comienza con un tablero vacío. A diferencia de Breadth-First Search, este se va expandiendo por cada nodo hijo de forma recurrente. Cuando no

encuentra más nodos que visitar, entonces regresa al padre y visita el siguiente hijo, así repitiendo el mismo proceso. Esto es realizado a través de una pila (LIFO). Y de la misma manera, es una búsqueda exhaustiva, esto significa que termina hasta recorrer todo el árbol.

3. Iterative Deepening Depth-First Search
Básicamente es la misma implementación que Depth-First Search, la única diferencia que se agrega un límite de carácter constante a las búsquedas. Se inicia con un valor 0 y se va incrementando gradualmente en la medida que necesita internarse en el árbol. Es posible que no encuentre una solución para el límite dado.

2.2. Código Documentado

A continuación se adjunta las partes esenciales del código. Se utilizo Orientación a Objetos para modelar el tablero y la Reina.

La clase Reina esta definida de la siguiente manera:

```
    class Queen:
    """La clase Queen representa a una reina"""

    def __init__(self):
        """Crea una instancia de la clase Queen."""
        self.row = -1
        self.column = -1

    def is_valid(self):
        """Verifica si la reina tiene asignada una posicion"""
        if self.row != -1 and self.column != -1:
            return True
        else:
            return False

    def is_attacking(self, queen):
        """Determina si la actual reina esta atacando a la reina pasada

        :param queen: La reina a comparar con la actual reina
        """
        if self.row == queen.row or self.column == queen.column: return True
        if abs(self.row-queen.row) == abs(self.column-queen.column): return True
        return False
```

La clase Tablero esta definida como:

```
    class Board:
    """La clase Board representa al tablero"""
    def __init__(self, board_size):
        """Crea una instancia de Board.

        :param board_size: Tamano del tablero
```

```
        """
        self.board_size = board_size
        self.queens = []
        self.depth_index = 0 # indica la profundidad en el arbol

    def set_queen(self, row, column):
        """Asigna una posicion a la reina si no ataca a otra

        :param row: Fila en el tablero
        :param column: Columns en el tablero
        """
        tmp_queen = Queen()
        tmp_queen.row = row
        tmp_queen.column = column

        for queen in self.queens:
            if queen.is_valid() and tmp_queen.is_attacking(queen):
                return False

        self.queens.append(tmp_queen)

        return True

    def is_valid(self):
        """Verifica si el tablero es valido"""
        if len(self.queens) == self.board_size: return True
        return False
```

Luego, se definio una clase general llamada Queens para generar los sucesores e implementar los algoritmos de búsqueda.

```
    class Queens:
    """La clase Queens define varios metodos para resolver el problem de
    las 8 reinas. Esta solucion comienza con un tablero vacio, en cada
    action se agrega unareina al tablero si esta no ataca a las otras reinas.
    """

    def __init__(self, options):
        """Crea una instancia de la clase Queens

        :param options: Valores opcionales para inicializar las variables
        de clase
        """
        self.board_size = 8
        self.options = options

    def __get_successors(self, current_board):
        """Retorna los siguientes sucesores
```

```
        :param current_board: Es el actual tablero
        """
        new_board_successors = []

        for col in xrange(self.board_size):
            board_successor = copy.deepcopy(current_board)
            if board_successor.set_queen(board_successor.get_depth(), col):
                board_successor.increment_depth()
                new_board_successors.append(board_successor)

        return new_board_successors

    def depth_first_search(self):
        """Encuentra todas las posibles soluciones usando el algoritmo
        Depth-First.
        """
        initial_state = self.__create_board()
        open_states = []
        open_states.append(initial_state)
        valid_boards = []

        while open_states:
            current_board = open_states.pop()
            if current_board.is_valid():
                valid_boards.append(current_board)
            else:
                # Genera proximos estados
                new_boards = self.__get_successors(current_board)
                for new_board in new_boards:
                    open_states.append(new_board)

        # Imprime las soluciones
        print "Fueron encontradas:", len(valid_boards), "soluciones"
        if self.options.show: self.__print_board(valid_boards)

    def __ids_helper(self, max_depth):
        """Basicamente es el mismo algoritmo DFS pero con un limite

        :param max_depth: Indica la profundidad maxima
        """
        initial_state = self.__create_board()
        open_states = []
        open_states.append(initial_state)
        valid_boards = []

        current_depth = 0
        while open_states:
            current_board = open_states.pop()
            if current_board.is_valid():
```

```
                print "Solucion encontrada en la profundidad:", current_depth
                valid_boards.append(current_board)
            else:
                # Genera proximos estados pero con limite
                if current_depth < max_depth:
                    new_boards = self.__get_successors(current_board)
                    for new_board in new_boards:
                        open_states.append(new_board)
                    current_depth += 1


        return valid_boards

    def iterative_deepening_search(self):
        """Encuentra todas las posibles soluciones usando el algoritmo
        Iterative Deepening.
        """
        valid_boards = []

        # La profundidad maxima sera el tamano del tablero
        for i in xrange(self.board_size):
            temp_boards = self.__ids_helper(i)
            if temp_boards: valid_boards.append(temp_boards)

        # Imprime las soluciones
        print "Fueron encontradas:", len(valid_boards), "soluciones"
        if self.options.show: self.__print_board(valid_boards)

    # BFS is based on queue data structure.
    def breadth_first_search(self):
        """Encuentra todas las posibles soluciones usando el algoritmo
        Breadth-First.
        """
        initial_state = self.__create_board()
        open_states = []
        open_states = deque([])
        open_states.append(initial_state)
        valid_boards = []

        while open_states:
            current_board = open_states.popleft()
            if current_board.is_valid():
                valid_boards.append(current_board)
            else:
                # Genera proximos estados
                new_boards = self.__get_successors(current_board)
                for new_board in new_boards:
                    open_states.append(new_board)
```

```
        # Imprime las soluciones
        print "Fueron encontradas:", len(valid_boards), "soluciones"
        if self.options.show: self.__print_board(valid_boards)
```

2.3. Resultados Obtenidos

La siguiente tabla muestra los resultados obtenidos tras la ejecución del los algoritmos de búsqueda. Nótese que existen 92 soluciones para el problema de las 8 Reinas. Los algoritmos Depth-First-Search y Breadth-First-Search encontraron las soluciones, a diferencia del algoritmo Iterative Deepening Search que no encontró soluciones para el limite igual a 8, sólo aumentando el límite se logro encontrar soluciones, pero aumentando el tiempo de ejecución.

Algoritmo	**Tiempo [seg]**	**Soluciones**
Depth-First-Search	1.394023	92
Breadth-First-Search	1.391231	92
Iterative-Deepening-Search	0.011623	0 (límite 8)
Iterative-Deepening-Search	5.865025	18 (límite 113)

3. Resultados y conclusiones

Como puede verse en la tabla anterior, los algoritmos Depth-First-Search y Breadth-First-Search encontraron las 92 soluciones que tiene el problema de las 8 Reinas. El algoritmo Iterative-Deepening-Search no encontró soluciones para el límite igual a 8. Para esta implementación, el algoritmo Iterative-Deepening-Search encuentra 18 soluciones para un limite igual a 113.

Considérese la definición formal de los tiempos de ejecución de los algoritmos:

Breadth-First-Search, donde d es la profundidad en el árbol:

$$\mathrm{O}\left(b^d\right)$$

Depth-First-Search, donde m son la cantidad de nodos visitados:

$$\mathrm{O}\left(b^m\right)$$

Iterative-Deepening-Search, donde d es la profundidad en el árbol:

$$\mathrm{O}\left(b^d\right)$$

Si analizamos la definición y comparamos los tiempos medidos en segundos de cada algoritmo podemos ver que hay una correspondencia. El algoritmo que toma menos tiempo de ejecución es Breadth-First-Search con 1.391231 segundos, esto nos demuestra que no se necesita llegar profundo en el árbol para llegar a las soluciones, por lo tanto este algoritmo es mas eficiente en tiempo de ejecución que el algoritmo Depth-First-Search.

Por último el algoritmo Iterative-Deepening-Search con límite igual a 113 fue capaz de encontrar 18 soluciones con un tiempo de ejecución igual a 5.865025, pero esto no quiere decir que sea ineficiente, solo que tuvo que repetir 113 veces el algoritmo Depth-First-Search para encontrar las soluciones. Quizás para un próximo estudio se podría medir el espacio de memoria utilizado por los algoritmos, de esta manera realizar una comparación en tiempo de ejecución versus uso de la memoria.

Referencias

[1] S. Russell *Artificial Intelligence* 2010: A Modern Approach.