ALGORÍTMICA

Prácticas

ALGORITMOS
DE EXPLORACIÓN EN GRAFOS

El Recorrido del Caballo

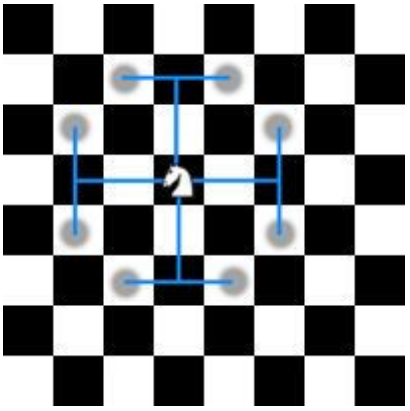
Grupo de prácticas C3
Grupo 2

1. DEFINICIÓN DEL PROBLEMA

Esta práctica consiste en la resolución del problema del **recorrido del caballo** mediante la técnica Backtracking o Branch & Bound.

En un tablero M de tamaño FxC ('F' número de filas y 'C' número de columnas), el caballo de ajedrez situado en una casilla inicial cuya posición es px y py (número de fila y número de columna), tiene que encontrar, si es posible, un recorrido donde recorra todo el tablero de forma que todas las casillas por las que pase, no se pisen 2 veces ni realice cualquier movimiento que lo saque del tablero. Su posición de inicio y final tras el recorrido, es indiferente.

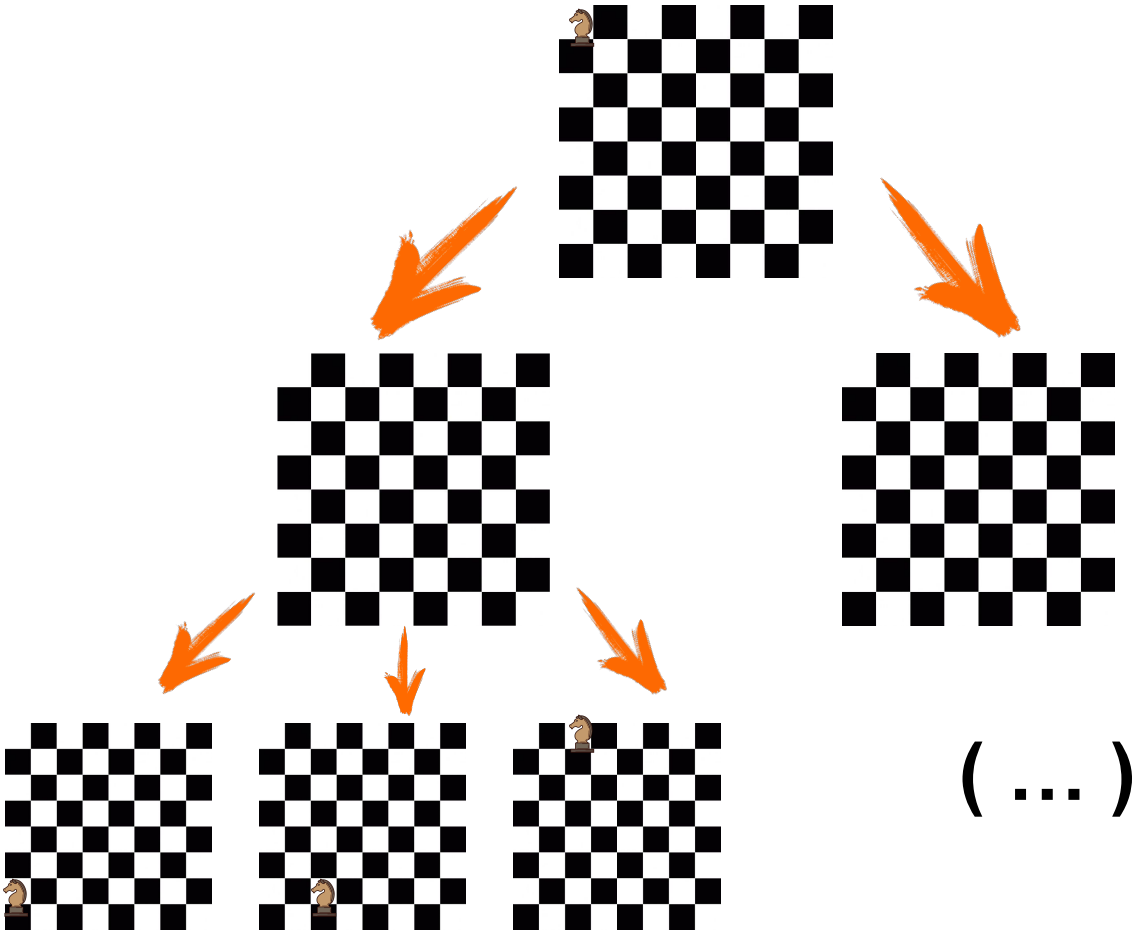
Los movimiento del caballo sólo son posibles en forma de L, como indica la siguiente imagen:



2. ANÁLISIS DEL PROBLEMA

El problema en cuestión plantea una matriz (tablero) de 8x8 donde colocamos una pieza de ajedrez, el caballo, en nuestro caso en la posición (0,0) y a partir de ahí debemos tratar de pasar por todas las casillas y no salirnos del tablero durante las 63 posiciones restantes.

Su implementación se presta al uso de la técnica Backtracking,un método de retroceso o vuelta atrás aplicable en juegos y otros tipos. Se realiza una búsqueda exhaustiva y sistemática en el espacio de soluciones posibles. El resultado es realizar un recorrido en profundidad del árbol de soluciones:



El branch and bound nos permite llegar a la misma solución que el backtracking con menor esfuerzo computacional, pero para llevar a cabo el método de búsqueda inicial debemos establecer una cota inicial, sin embargo, esta debe ser una solución del problema y en nuestro caso no sería posible pues implicaría que ya se ha llegado a una solución óptima. Por ello se ha elegido la estrategia de búsqueda de backtracking.

3. DISEÑO DE LA SOLUCIÓN

Para implementar la solución tenemos que partir de la idea de que necesitamos una función recursiva que nos permita recordar los caminos escogidos desde cada posición y aplicar backtracking para volver al paso anterior de la solución errónea.

La implementación en C++ sería la siguiente:

Definimos un struct con las coordenadas x e y del caballo en cuestión.

```
typedef struct coordenadas {
   int x,y;
```

```
}coordenadas;
```

Función que imprime el tablero 8x8 con la posición actual de la pieza :

```
void imprimirRecorrido(int tablero[8][8]) {
   int i,j;
   for (i = 0; i < 8; i++) {
      for (j = 0; j < 8; j++) {
          cout<<tablero[i][j]<<"\t";
      }
      cout<<endl;
   }
}
```

Función que comprueba que el movimiento de la pieza no provoca salirse del tablero y que esa casilla no se repite :

```
bool esFactible(coordenadas movimiento, int recorrido[8][8]) {
   int i = movimiento.x;
   int j = movimiento.y;
   /*Si está entre la casilla 0 y 8 y no ha pasado por allí nunca*/
   if ((i >= 0 && i < 8) && (j >= 0 && j < 8) && (recorrido[i][j] == 0))
      return true;
   return false;
}
```

Función que encuentra el recorrido que recorre las 63 casillas restantes. Tenemos un caso base en la situación donde el caballo haya pasado por todas las casillas y en caso contrario, comprobamos si el siguiente movimiento está permitido y creamos el recorrido. Los casos donde la posición siguiente es inválida, corregimos la matriz con un 0 y volvemos a la casilla anterior :

```
bool buscarRecorrido(int recorrido[8][8], coordenadas movimientos[],
               coordenadas posicionActual, int numMovimientos) {
   int i;
   coordenadas siguienteMov;

   if (numMovimientos == 8*8-1) {
       return true;
   }

    for (i = 0; i < 8; i++) {
       siguienteMov.x = posicionActual.x + movimientos[i].x;
       siguienteMov.y = posicionActual.y + movimientos[i].y;
```

```
        if(esFactible(siguienteMov, recorrido)) {
                recorrido[siguienteMov.x][siguienteMov.y] = numMovimientos+1;
                if(buscarRecorrido(recorrido,movimientos,siguienteMov, numMovimientos+1) ==
true) {
                return true;
        }
                else {
                recorrido[siguienteMov.x][siguienteMov.y] = 0;
        }
        }
   }
   return false;
}
```

Función que simula el movimiento del caballo por el tablero. Se inicializan los movimientos que puede realizar el caballo y se imprime el recorrido.

```
void recorridoCaballo() {

   int recorrido[8][8];
   int i,j;
   coordenadas movimientos[8] = { {2,1},{1,2},{-1,2},{-2,1},
                            {-2,-1},{-1,-2},{1,-2},{2,-1} };
   for (i = 0; i < 8; i++) {
        for (j = 0; j < 8; j++) {
        recorrido[i][j] = 0;
        }
   }
   coordenadas posicionActual = {0,0};
   if(buscarRecorrido(recorrido, movimientos, posicionActual, 0) == false) {
        cout<<"\nError el recorrido no existe";
   }
   else {
        imprimirRecorrido(recorrido);
   }
}
```

Además, hemos creado la clase auxiliar “Tablero” en la que iremos guardando el recorrido del caballo. Es simplemente una matriz 2D, y cada casilla le vamos asignando el número con el orden por el que se pasó.

Esta sería la implementación en **pseudocódigo** de las principales funciones:

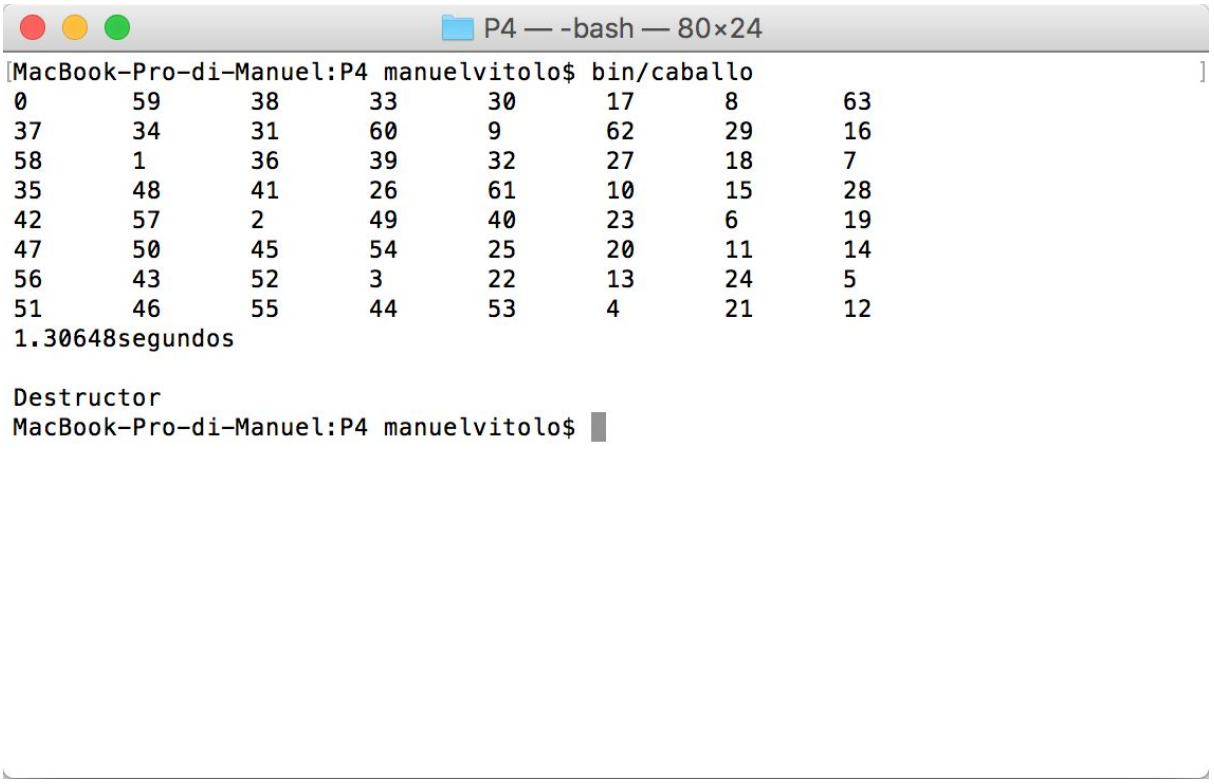
Recorrido

Inicializar Tablero recorrido //Muestra cada casilla por la que pasa indicando con un número el orden
Inicializar vector movimientos = {{2,1},{1,2},{-1,2},{-2,1},
{-2,-1},{-1,-2},{1,-2},{2,-1} };
posicionActual = {0,0}

```
        si (buscarRecorrido(recorrido,movimientos,posicionActual,0))
                imprimir recorrido (recorrido)



buscarRecorrido(Tablero recorrido, movimientos,posicionActual,numeroMovimientosHecho){
        si numeroMovimientosHecho == 63
                return true; //Ya hemos visitado todas casillas

        sino
                Para i = 0 hasta 8 i++{
                        //vamos moviendo para cada casilla, con los 8 movimientos posibles por ejemplo //
                        posición 0.0 + -2.-1
                        siguienteMovimiento.x = posicionActual.x + movimientos[i].x
                        siguienteMovimiento.y = posicionActual.y + movimientos[i.y

                        si es Factible(siguienteMovimiento){
                                asignar posicion al recorrido → recorrido[x][y]=numMovimientos+1
                                si                                      buscarRecorrido(recorrido,
                        movimientos,siguienteMovimiento,numeroMovimientos)
                                        return true
                                else
                                        recorrido asignar posicion 0          //recorrido[pos]=0

                }
}
```

4. EJEMPLO DEL FUNCIONAMIENTO DEL ALGORITMO

Tras la compilación del código y la ejecución del mismo el resultado en el terminal es este:



5. ORDEN DE EFICIENCIA TEÓRICO

En la primera función “imprimirRecorrido” ya tenemos dos bucles for anidados. La complejidad entonces es de $O(n^2)$. En la función principal “recorridoCaballo” también tenemos dos bucles for anidados y complejidad $O(n^2)$. Las demás funciones tienen todas complejidades menores, entonces podemos decir que el algoritmo tiene complejidad $O(n^2)$.

6. CÓMO COMPILAR Y EJECUTAR EL CÓDIGO

La carpeta que incluye el código contiene un archivo makefile que sirve para la compilación. Compilando a través de este se creará un archivo ejecutable en la carpeta /bin que permite la ejecución del código.

7. CONCLUSIONES

Para este problema en el que las soluciones son todas igual de válidas y no cuantifica si una es más óptima que otra, no tiene sentido usar técnicas de ramificación y poda propias del Branch & Bound, ya que los criterios de estas dependen de estimaciones a soluciones al problema.
Por lo tanto la única opción es Backtracking, evaluar todas los nodos del árbol generado hasta llegar a una solución, al usar un método recursivo la búsqueda es en profundidad,

llegando cuanto antes a una solución y así evitar seguir evaluando otros nodos en la gran mayoría de casos.