PRÁCTICA FINAL

ALGORITMIA

Jonathan Salisbury
Joan Vilella Candia

Contents

Jerarquía de clases

Para la realización de la práctica hemos usado el patrón de arquitectura de software de MVC (Model-View-Controller). Debido a esto tenemos 3 grandes paquetes donde se engloba el código:

- Bootstrap: Inicio de aplicación.
- Core: Donde reunimos aquellas características comunes a las clases.
- Modules: Donde tratamos en concreto cada uno de los problemas. Que a su vez cada problema tendrá su Bootsrap, su controlador, el modelo y su visualizador.

Core

En esta carpeta tenemos dividido por una parte la carpeta que contendrá la clase abstracta de "Problem" que deberán extender aquellas clases que se encarguen de resolver el problema propiamente dicho. Por eso no es de extrañar que tenga un método abstracto "solve()".

También está la subcarpeta de "chesspieces" donde todas heredan las características de la clase "Chesspiece". Lo más remarcable en comparación a la anterior práctica es que incorporan el método "checkKill()" donde cada clase en particular revisará sus movimientos atendiendo a las reglas del juego.

Finalmente, la subcarpeta "view", la clase abstracta que tiene "ProblemViewer" es la representación visual de un problema de ajedrez. Es por esto que cada problema por separado deberá extender esta clase. Además de dos clases más, por una parte la del menú de inicio y por otro lado la clase necesaria para poder visualizar el tablero.

Modules

Esta carpeta se subdivida en partes iguales por cada uno de los problemas y ya que tienen una gran similitud comentaremos la estructura general. Cada problema tiene un:

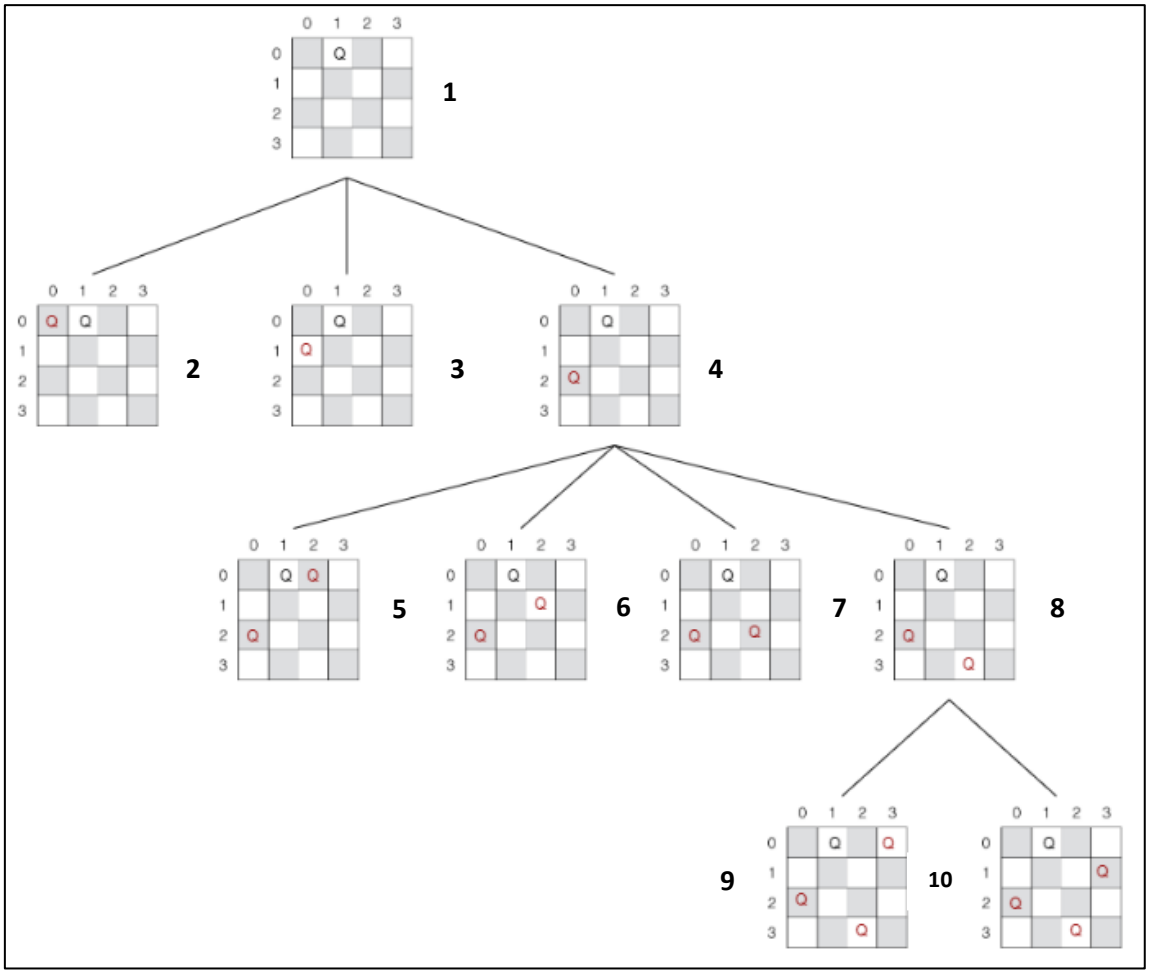
- Bootstrap: En esta clase es donde se carga todo lo relacionado con el problema.
- Model: En esta subcarpeta se encuentra la clase que resuelve el problema de recursividad.
- Viewer: En esta subcarpeta se encuentra la clase para poder visualizar el problema.

Diseño de problemas

NQueens Problem

La principal decisión de diseño en la resolución de este algoritmo y la razón por la cual simplificamos en gran medida el número de casos que debemos tener en cuenta es que restringimos la posición de las reinas a una por columna y por ende solo revisaremos las filas. Una decisión bastante intuitiva ya que es impensable que haya más de una reina por columna.

A continuación, mostramos el árbol recursivo del problema. Hemos elegido un caso sencillo (4x4) y una configuración de reina que conocemos que tiene solución para poder mostrar gran parte del funcionamiento del algoritmo. Aunque el ejemplo sea simple es suficientemente explícito y es extrapolable a tableros más grandes.



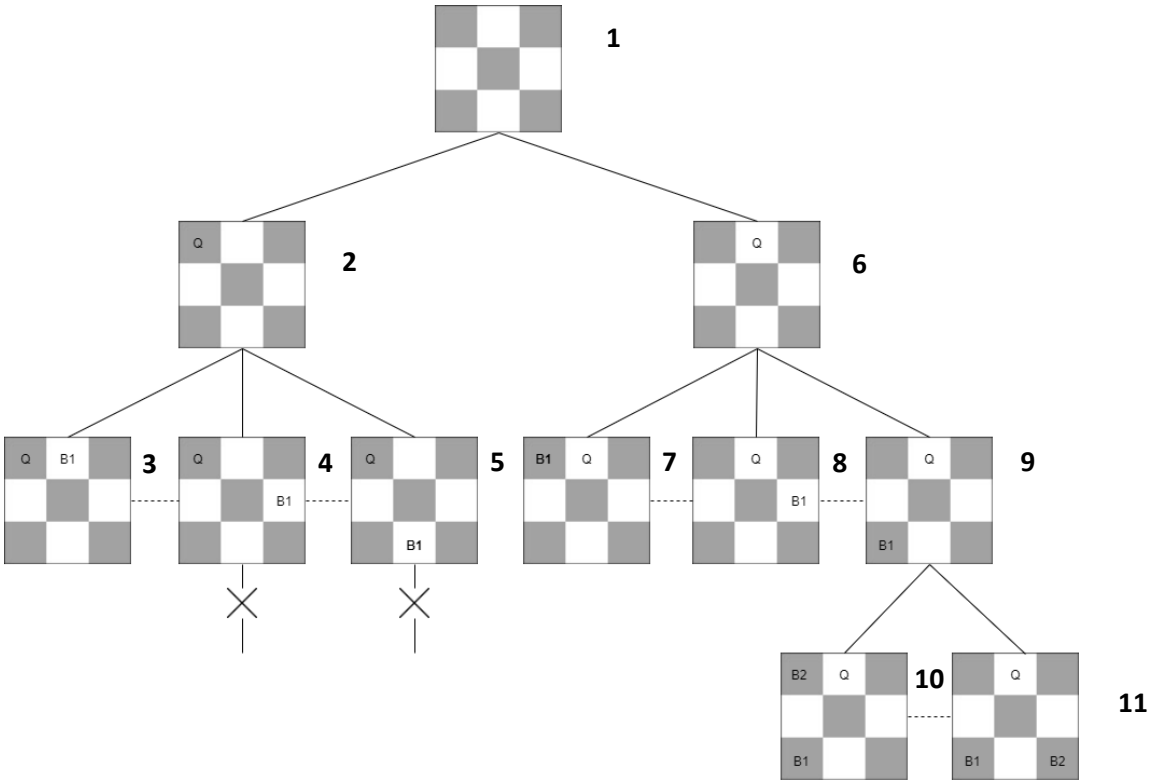
- Figura 1: Aquí es donde comienza nuestro programa, recibimos la reina situada por el usuario, En este caso (0, 1).
- Figura 2: En esta primera llamada recursiva situamos la primera candidata de reina en la primera casilla de la primera columna (0, 0), como podemos observar no será una configuración válida, por lo que deshacemos el cambio.
- Figura 4: No es hasta esta figura donde encontramos la primera posición válida de la reina en la primera columna (2, 0). Como es una posición correcta hacemos una nueva llamada recursiva y bajamos un nivel de profundidad en el árbol.
- Figura 5: Como la segunda columna (columna 1) está ocupada por la reina que nos da el usuario, saltamos a la siguiente. En esta columna volvemos a comprobar fila a fila hasta que encontramos una posición válida.
- Figura 8: Recién en esta figura es donde encontramos una posición válida para la reina, por lo que haremos una nueva llamada recursiva.

- Figura 9: Sólo nos queda la última columna por comprobar, por lo que colocamos la reina en la primera fila (0, 3). Como podemos observar no es un movimiento correcto, por lo que deshacemos el cambio que acabamos de realizar y probamos con la siguiente posición.
- Figura 10: Hemos llegado a una configuración de reinas aceptadora, por lo que el algoritmo acabaría.

SaveBoardProblem

Este problema es muy similar al de las reinas, pero no podemos restringir una pieza por columna, ya que hay piezas que si que permitirán que haya varias en una misma columna (de hecho, la gran mayoría). Por lo que, aunque haya una pieza en una columna, deberemos revisar igualmente si podemos situar otra pieza. Además, el usuario no nos da la posición de ninguna de las piezas, sólo el orden de ellas. Finalmente, la última de las diferencias es que esta vez realizamos el recorrido a nivel de fila y no de columna, esto se verá mejor reflejado en el árbol.

Para el esquema recursivo hemos recurrido a una de las pruebas que se nos propone para superar, la cual es situar una reina y dos alfiles en un tablero 3x3.



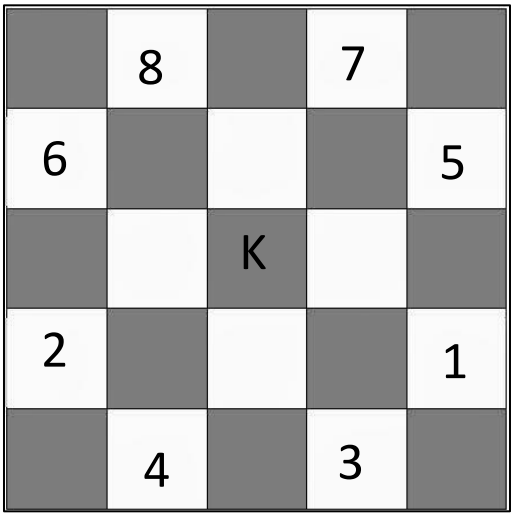
- Figura 1: Este sería el inicio de nuestro programa, donde tenemos el tablero vacío y el usuario nos ha pasado una serie de figuras a colocar.
- Figura 2: Colocamos la reina en la primera casilla (0, 0) y como es una posición aceptadora, realizamos una llamada recursiva para colocar el alfil.
- Figura 3: Esta es la primera de las posiciones aceptadoras para situar el primer alfil, posición (1, 2). Pero como podemos observar a simple vista, no hay ni una sola posición disponible para el segundo alfil. Es por esta razón por la que colocamos una flecha con una 'X' a continuación. Ya que no hay ninguna llamada recursiva que pueda ser aceptadora.

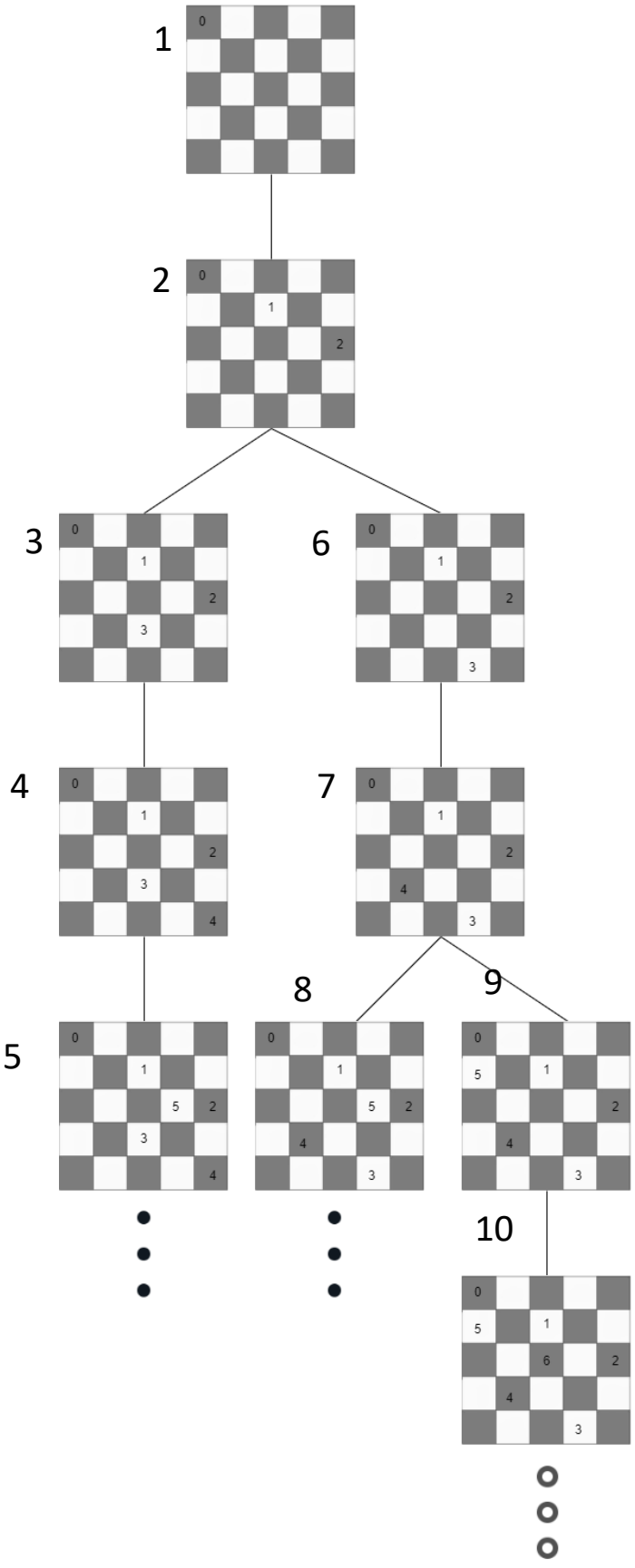
- Figura 5: Después de deshacer los cambios realizados entre las posiciones entre las figuras 3 y 4 llegamos a la posición (2, 1). Pero al igual que en la figura 4 podemos observar que no va a haber ninguna posición aceptable para el segundo alfil.
- Figura 6: Llegamos a esta posición después de hacer dos backtrackings, ya que hemos agotado todas las posibilidades deshacemos los cambios realizados en la figura 5 y en la 2. Por ende situamos la reina el la siguiente posición posible (0, 0).
- Figura 9: Llegamos a este estado después de agotar todas las posibilidades de colocar el primer alfil, de ahí los puntos suspensivos entre figuras. Finalmente podemos poner el primer alfil en la posición (2, 0). Por lo que volveremos a realizar una llamada recursiva buscando la última posición favorable.
- Figura 11: Después de agotar todas las posibilidades para la posición del último alfil, finalmente es en la casilla (2, 2) donde llegamos a una configuración de piezas aceptadora.

Knigth's tour

Para entender el esquema recursivo que mostraremos a continuación debemos explicar brevemente como funciona el algoritmo para entender las decisiones que toma.

Nosotros tenemos marcados los posibles movimientos del caballo por el tablero con dos arrays. Una de ellas indica cuanto ha de incrementar o decrementar la posición de las filas y la otra lo mismo con las columnas. Pero claro, a la hora de configurar estos arrays hay un factor de decisión por parte del programador ya que aquellos movimientos que se pongan antes en los arrays serán los primeros que pruebe. Por lo que el orden que sigue "nuestro caballo" a la hora de revisar posiciones es el siguiente.





- Para este ejemplo el usuario a situado el caballo en la posición (0, 0) y es aquí donde arranca nuestro algoritmo.
- Figura 2: Dictado por el movimiento antes definido del caballo los siguientes movimientos serán los indicados en la figura. Por lo que podemos observar que estamos aumentando siempre en una fila y dos columnas
- Figura 3: Puesto que el primer movimiento más prioritario no es posible (ya que nos saldríamos del tablero), el algoritmo prueba con el segundo más prioritario.
- Figura 5: Aquí llegamos desde la posición más prioritaria de la figura 4. Aunque aquí aún no esté claro acabaremos llegando a una situación en la que el caballo no tendrá más movimientos posibles. Por lo que deberemos deshacer todos los pasos.
- Figura 6: Después de haber probado todas las combinaciones con el primer movimiento más prioritario, el algoritmo seguirá con el segundo.
- Figura 7: Aquí en este caso hay una gran mayoría d movimientos bloqueantes por lo que tenemos que recurrir al movimiento de prioridad 6.
- Figura 8: Aquí, al igual que en la figura 5 no habrá posibilidades de acabar en un aposición aceptadora, por lo que cuando lleguemos al último paso donde el caballo ya no se puede mover, tarde o temprano acabará deshaciendo este cambio.
- Figura 9: Aquí acabamos recurriendo al movimiento de prioridad 8 y como es una posición aceptadora, haremos una nueva llamada recursiva.
- Figura 10: Sabemos con certeza que hasta aquí el algoritmo ha estado siguiendo los pasos adecuados. De los vistos hasta ahora es el único (con nuestra implementación) que podría llegar hasta una buena configuración.

Todas esas suposiciones que hemos dado por buenas en el apartado anterior las conocemos porque tenemos una de las soluciones para cuando el caballo comienza en esa casilla.

0	13	18	7	24
5	8	1	12	17
14	19	6	23	2
9	4	21	16	11
20	15	10	3	22

Es exactamente la solución dada por nuestro algoritmo. De alguna manera hemos hecho un poco de "trampa" en el apartado anterior. Pero es que comprobar a mano todos los casos posibles era inviable. Especialmente en este ejercicio. Ya que el tablero mínimo para que se pueda cumplir este recorrido es en un tablero 5x5.

Soluciones adoptadas

Como la representación de estados es común a todos los problemas y todos basan su implementación alrededor de esta, preferimos comentarla al margen de las soluciones.

A diferencia de muchas soluciones de backtracking vistas en clase, en lugar de usar un vector solución, hemos reflejado en todo momento el estado del cómputo en el array bidimensional de piezas que representa el tablero. Esta decisión la hemos tomado porque nos parecía mucho más sencillo implementar toda la parte gráfica del proyecto.

N-Queens Problem

La principal poda que hemos realizado a nivel sistemático por cómo está programado el algoritmo es el de limitar el número de reinas por columna a una. Además, tenemos el método "checkKill()" que es un método de la clase "Queen" el cual hereda de la clase abstracta "Chesspiece" en la que comprobamos las 4 diagonales, las líneas horizontales y verticales a la hora de colocar una reina.

SaveBoard Problem

A diferencia del anterior problema, en este no podemos limitar la cantidad de piezas por columna por lo que deberemos recorrer todas y cada una de las casillas. Es por esta razón que en el método recursivo tenemos un bucle anidado. Nuestro caso base en el problema es que finalizamos cuando no quedan más piezas por poner. La única poda que implementamos son los métodos que comprueban la disponibilidad a la hora de colocar una pieza.

Para este problema hemos tenido que implementar dos “checkKills”, el primero por parte de la pieza que colocamos que revisa que no esté atacando a ninguna otra pieza y otro método que comprueba que ninguna pieza esté atacando a la pieza que se va a colocar en el tablero.

Knight’s Tour

En este problema, nuestro caso base es cuando alcanzamos el total de casillas del tablero. Una vez recibimos la posición del caballo del usuario vamos probando los distintos movimientos definidos por el doble array y es gracias a la función de “possibleMove()” con la que realizamos la poda.

Coste computacional de los algoritmos implementados

N-Queens

Analizando el árbol recursivo y teniendo en cuenta que por cada columna solo podemos situar una reina podemos obtener las siguientes conclusiones.

- Para la primera reina que situemos, en el peor de los casos tendremos $N - 1$ casos posibles, ya que una de esas filas estará atacada por la reina dada por el usuario.
- Si seguimos con este ejemplo la segunda reina que situemos tendrá por cada uno de los $N - 1$ casos de la primera reina $N - 2$ casos disponibles.

Podemos ver que claramente estamos antes un caso de $(n-1)!$ Pero como a la hora de calcular la complejidad lo que es realmente importantes es el orden, podemos decir que la complejidad del problema es:

$$O(n!)$$

Saveboard

Teniendo en cuenta:

Theorem (Reduction by subtraction) *The recurrence $T(n)$ that is described below defines a function $T(n) : \mathbb{N} \to \mathbb{R}^+$ whose growth depends on its parameters as follows:*

$$T(n) = \begin{cases} bn^\alpha & \text{if } n < c \\ aT(n-c) + bn^\alpha & \text{if } n \geqslant c \end{cases}$$

when $a < 1$: $T(n) \in \Theta(n^\alpha)$
when $a = 1$: $T(n) \in \Theta(n^{\alpha+1})$
when $a > 1$: $T(n) \in \Theta((a^{1/c})^n)$

where $a, b, c > 0$ and $\alpha \geqslant 0$.

Podemos observar en nuestro algoritmo reducimos la complejidad en una unida (cada vez que ocupamos una casilla), realizamos una única llamada recursiva y nuestro caso base al ser constante tenemos que:

- $c = 1,\ a = 1$ y $bn^\alpha = 1$

Por lo que nos encontramos en el segundo casi y nuestra complejidad sería igual a $O(n^2)$ pero claro, esta llamada recursiva está dentro de un doble bucle. Por lo que tendríamos sería:

$$O(n^2) * O(n^2) = O(n^4)$$

Knight's tour

Hemos investigado sobre la complejidad de este problema, ya que nuestros cálculos basados en el "Master Therorem" no nos parecían los correctos. Finalmente hemos llegado a la siguiente conclusión:

Teniendo en cuenta que tenemos un total de N^2 casillas en el tablero y que el máximo número de movimientos del caballo son 8, podemos concluir que la complejidad será de 8 movimentos por cada celda en el peor de los casos. Que es lo mismo que:

$$O(8^{N^2})$$

Pero claro, el caballo no siempre tiene todas estas casillas disponibles por lo que podríamos decir que estará multiplicado por un valor entero k.

$$O\left(k^{N^2}\right)$$

Fuentes:

https://www.geeksforgeeks.org/the-knights-tour-problem-backtracking-1/

https://www.codesdope.com/course/algorithms-knights-tour-problem/

Manual de usuario

Introducción

El programa que hemos desarrollado da solución a tres problemas basados sobre un tablero de ajedrez. El primero de estos problemas es el conocido popularmente como las N-Reinas, otro de los problemas es el del camino del caballo (Knights's Tour) y finalmente el denominado SaveBoard. Además de añadir la práctica parcial para una mayor completitud del programa. A continuación, describiremos brevemente los problemas:

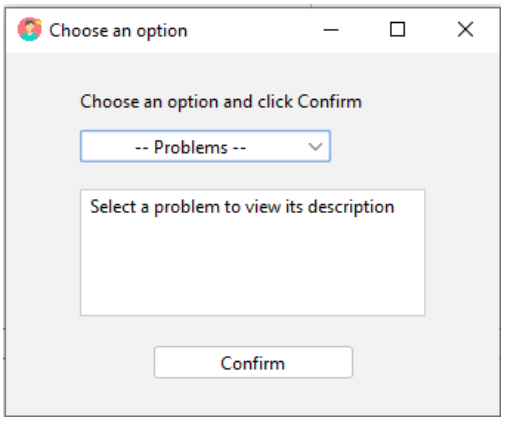
- N-Reinas: Situada una reina por el usuario en el tablero, el programa debe calcular una pasible configuración de las reinas restantes en las que no se maten.
- SaveBoard: Similar al problema anterior, pero en el que el usuario selecciona un conjunto de piezas a situar en el tablero y el programa encuentra una configuración en la que no se mate ninguna pieza. En caso de no poderse el programa lo informará.
- Knight's Tour: Dada una casilla por parte del usuario el programa calcula un camino seguido por el caballo en el que no repite ninguna casilla. En caso de no poderse el programa lo informará.

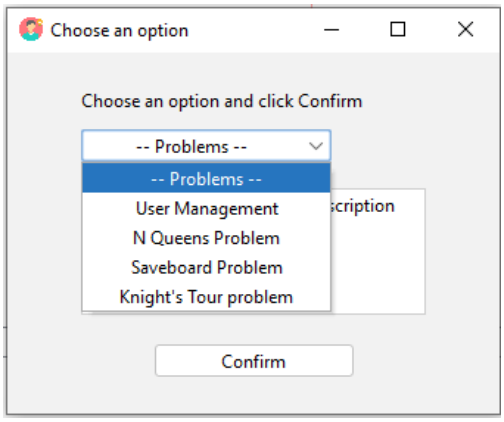
Además, se ha usado una GUI para hacerlo más ameno de cara al usuario.

Uso de funciones básicas

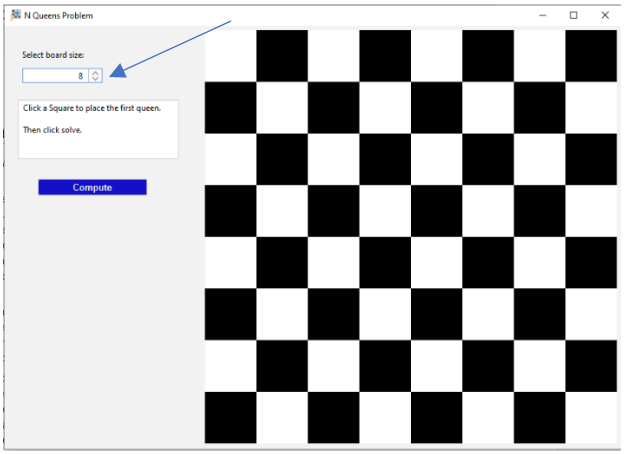
Inicio del programa

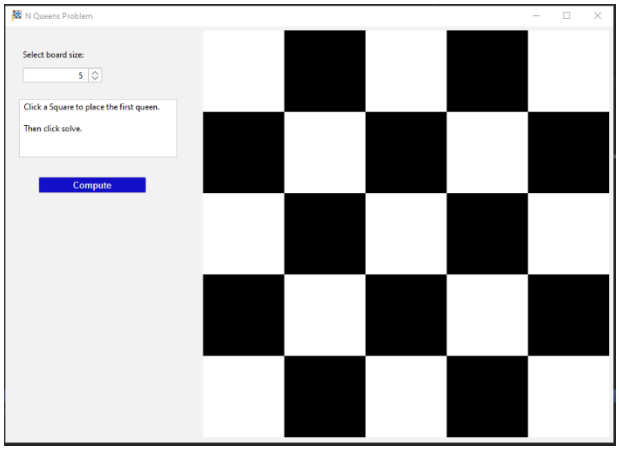
Nada más ejecutar el programa visualizaremos el siguiente panel, donde el usuario deberá de elegir del desplegable la opción que quiera utilizar. En el cuadro de diálogo se mostrará una pequeña descripción del problema y finalmente haciendo click en el botón "Confirm" se pasará al problema.





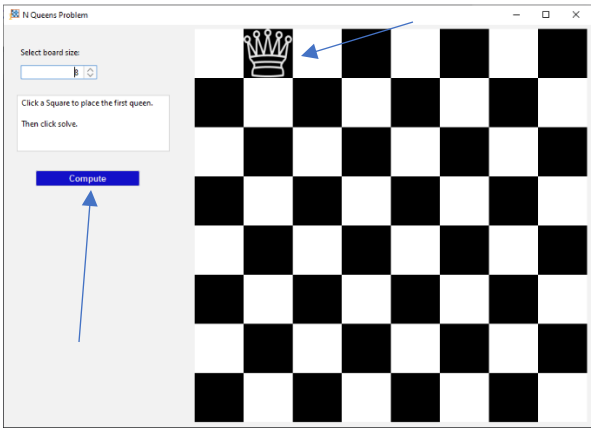
Vamos a analizar cada uno de los problemas por separado ya que la interfaz difiere ligeramente en cada problema. Aunque independientemente del problema podremos modificar el tamaño del tablero, por esta razón indicaremos como hacerlo en este apartado.

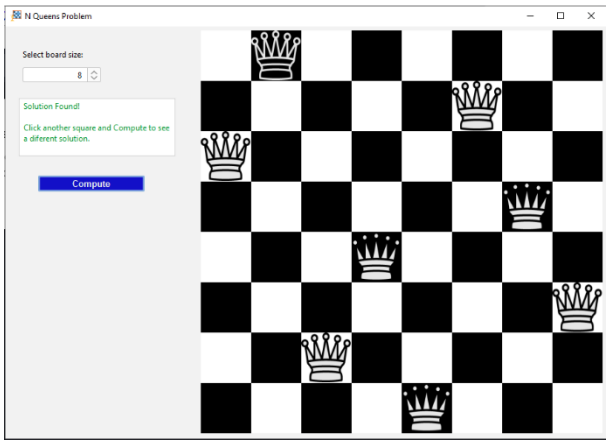




Este es el tablero inicial donde en primer lugar deberemos indicar el tamaño del tablero en el recuadro correspondiente. Haciendo click en las flechas podremos ajustarlo al tamaño deseado.

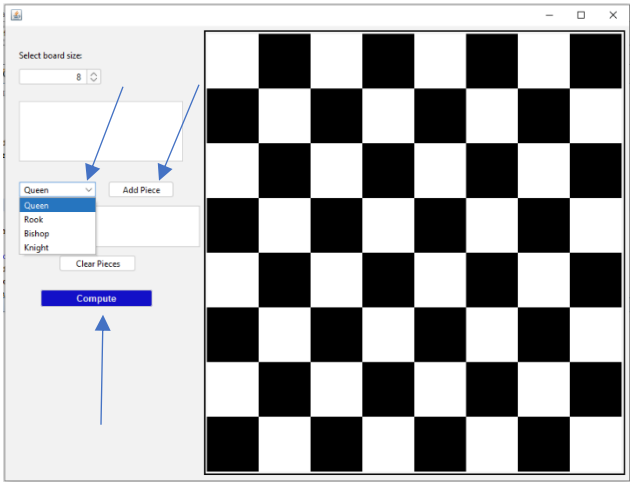
N-Queens

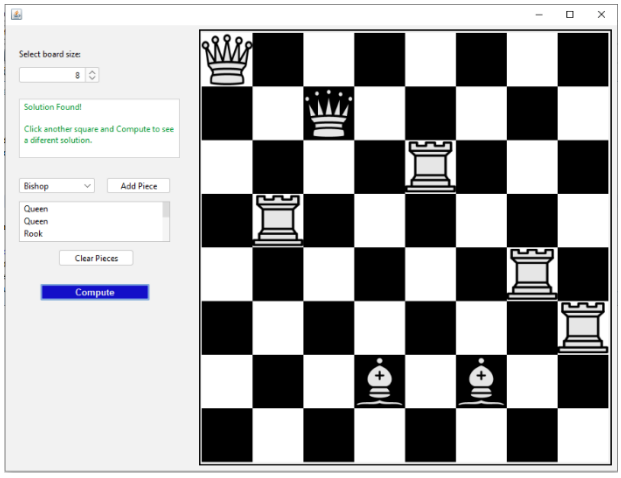




Deberemos seleccionar la casilla donde queremos situar nuestra reina inicial, ésta se distinguirá del resto porque tiene un color distinto. Después haremos click en el botón “Compute” y el programa mostrará una configuración de reinas en la que ninguna ataque a otra. En caso de que no se pueda lo mostrará por el cuadro de texto.

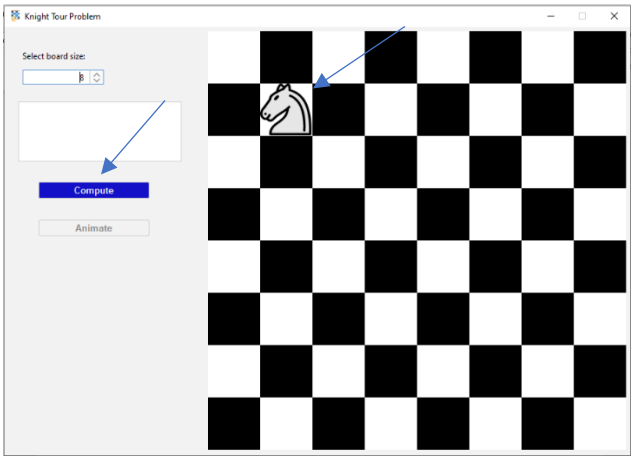
SaveBoard Problem

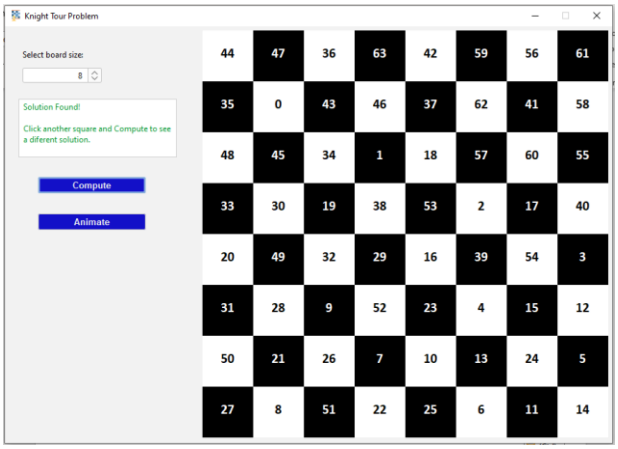




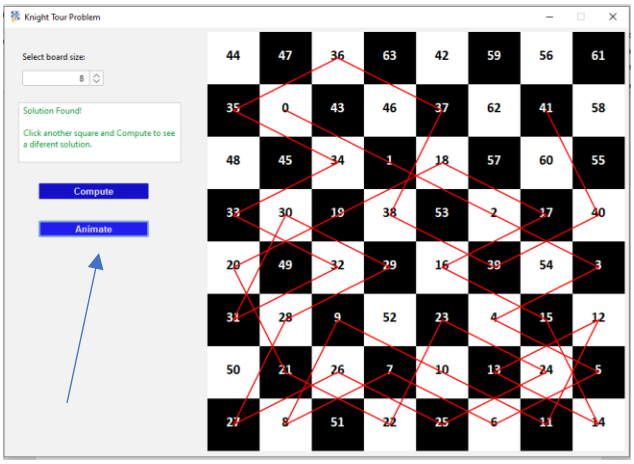
Del desplegable que se muestra en pantalla iremos eligiendo la pieza que queramos y con el botón “add piece” la añadiremos a nuestra selección. Siempre podemos borrar nuestra selección el el botón “clear pieces”. Cuando lo demos al botón “Compute” nos mostrará (en caso de que exista) una de las posibles configuraciones para las piezas. En caso de que no sea posible se indicará en el cuadro de texto.

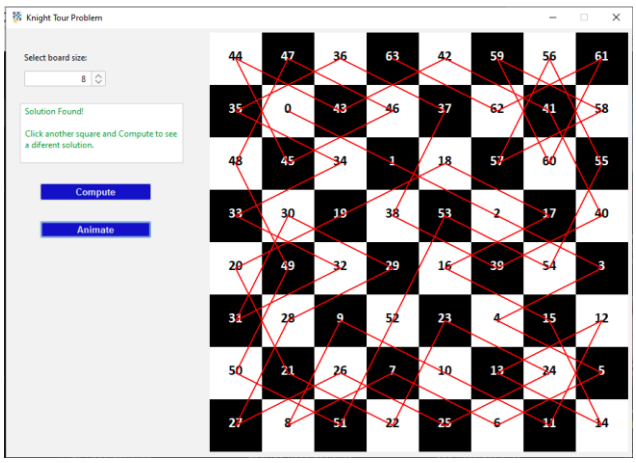
Knight’s tour





Aquí elegiremos sobre el tablero la posición inicial del caballo, una vez seleccionada la casilla dándole al botón de “compute” se mostrarán (en caso de que se pueda) los índices de las posiciones que puede recorrer el caballo en una de sus posibles soluciones.

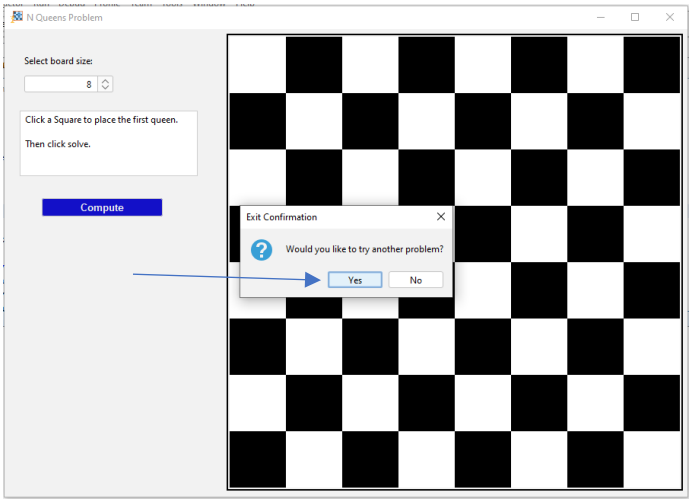


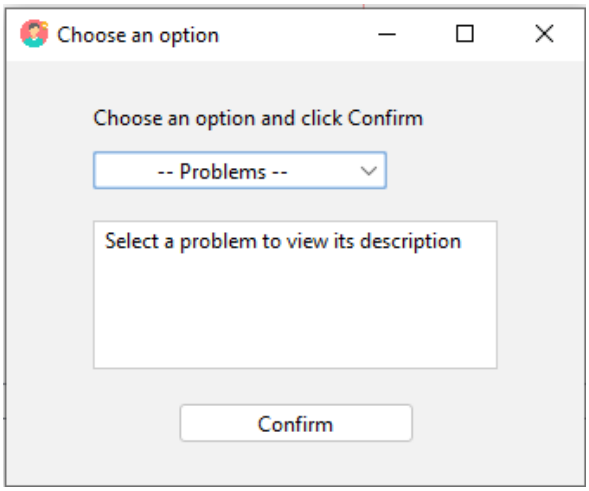


Una vez encontrada una posible solución se nos desbloqueará el botón de “Animate” que haciendo click en él, comenzará una animación donde una línea irá de casilla en casilla mostrando de manera más gráfica el recorrido del caballo.

Cambiar de problema

En caso de que queramos cambiar de problema, la interfaz del programa permite no tener que cerrar el programa por completo. Para ello seguiremos los siguientes pasos.





Una vez le damos le damos a la ‘X’ superior derecha nos aparecerá el siguiente cuadro, si seleccionamos la opción “Yes” nos llevará de nuevo al cuadro inicial donde elegir el programa, en caso de marcar “No” cerraremos el programa.

Solución de problemas

- Bloqueo de aplicación al computar: Este es un error que puede suceder principalmente en los problemas del caballo del SaveBoard. Esto se debe a que los algoritmos implementados no son los más eficientes y en ciertas configuraciones el tiempo de cálculo puede ser excesivo para el usuario. En estos casos la única opción que hay disponible es la de reiniciar la aplicación.

Aclaraciones

Para el apartado gráfico utilizamos una librería añadida, no debería de haber ningún tipo de problema ya que está incluida en el propio proyecto. En caso de que hubiera algún inconveniente por favor contacte con nosotros.