UNIVERSIDAD NACIONAL DEL ALTIPLANO

Facultad Ingeniería Mecánica, Eléctrica, Electrónica y Sistemas

Escuela Profesional de Ingeniería de Sistemas

Trabajo 4 :

EJERCICIO DE LAS 5 REINAS

DOCENTE:

ING. FLORES ARNAO ALODIA

PRESENTADO POR:

CHOQUE CHURA OLIVER FRAN

SEXTO SEMESTRE 2024-I

PUNO - PERÚ

2024

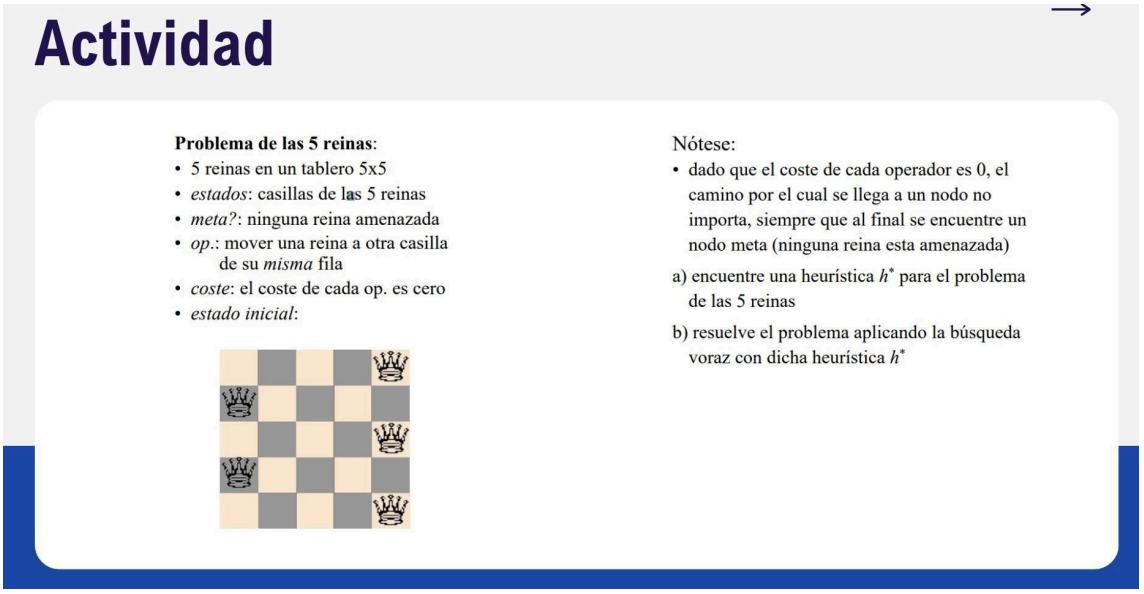
Índice

Introducción **3**
Descripción del Problema **4**
Configuración del Tablero 4
Definiciones Clave 4
Estado Inicial 4
Objetivos **5**
Heurística Utilizada **5**
Generación de Estados Vecinos **5**
Representación Visual del Tablero **5**
Proceso de Búsqueda Voraz **6**
Proceso de Búsqueda Detallado 6
Implementación en C++ **7**
Estructura del Código 7
Descripción Detallada de Componentes Clave 7
Heurística 7
Generación de Vecinos 8
Interfaz Gráfica 8
Búsqueda Voraz 8
Resultados 8
Conclusiones **9**
Futuras Direcciones **9**
Referencias **10**
PROBLEMA DE LAS 5 REINAS EN C++ **10**
Código 10
RESULTADO **17**
POSICIÓN INICIAL 17
POSIBILIDADES **18**
PRIMERA SOLUCIÓN 18
SEGUNDA SOLUCIÓN 18
TERCERA SOLUCIÓN 19
CUARTA SOLUCIÓN 19
QUINTA SOLUCIÓN 20
SEXTA SOLUCIÓN 20
SÉTIMA SOLUCIÓN 21
OCTAVA SOLUCIÓN 21
NOVENA SOLUCIÓN 22
DÉCIMA SOLUCIÓN 22

LINK GITHUB - Informe y Código

https://github.com/OliverChoque/5-Reinas.git

EJERCICIO



Problema de las 5 Reinas y su Resolución Heurística con Implementación en C++

Introducción

El problema de las N reinas es un problema clásico en la teoría de la computación y la inteligencia artificial. Este problema implica colocar N reinas en un tablero de ajedrez de NxN de tal manera que ninguna reina pueda atacar a otra. En este informe, se aborda una versión específica conocida como el problema de las 5 reinas, donde se deben colocar cinco reinas en un tablero de 5x5. El objetivo es aplicar una búsqueda heurística para encontrar una solución óptima donde ninguna reina se encuentre amenazada por otra.

Descripción del Problema

Configuración del Tablero

- **Tamaño del Tablero**: 5x5
- **Número de Reinas**: 5

Definiciones Clave

- **Estado**: Representa la configuración actual de las reinas en el tablero. Cada estado se define por la posición de las reinas en cada fila del tablero.
- **Meta**: Una configuración en la que ninguna reina está en posición de atacar a otra. Esto significa que no comparten la misma fila, columna o diagonal.
- **Operaciones**: Mover una reina a otra casilla dentro de la misma fila. Este movimiento genera un nuevo estado.
- **Costo**: El costo de cada operación se considera cero para simplificar el modelo. Esto implica que la longitud del camino para llegar a un nodo no es relevante, solo el estado final importa.

Estado Inicial

Se presenta una configuración inicial para las reinas:

- Configuración: {4, 0, 4, 0, 4}

Esta configuración sirve como punto de partida para la exploración del espacio

de estados en la búsqueda de una solución óptima.

Objetivos

1. **Encontrar una Heurística h*h^*h***: Identificar una función heurística adecuada que guíe la búsqueda hacia una solución donde ninguna reina esté amenazada.
2. **Resolver el Problema**: Aplicar la búsqueda voraz utilizando la heurística h*h^*h* para encontrar una configuración válida de las 5 reinas.

Heurística Utilizada

La heurística seleccionada es el número de reinas que están amenazadas en la configuración actual del tablero. Específicamente, la heurística cuenta el número de pares de reinas que se encuentran en la misma fila, columna o diagonal. Esta elección es intuitiva, ya que una configuración con menos reinas amenazadas está más cerca de la solución óptima.

Generación de Estados Vecinos

Para explorar el espacio de estados, se generan vecinos de la configuración actual moviendo una reina a cualquier otra posición en su fila. Esto permite explorar todas las posibles configuraciones derivadas de un solo movimiento de reina.

Representación Visual del Tablero

Se implementa una interfaz gráfica utilizando la API de Windows para

representar visualmente el tablero de ajedrez y las posiciones de las reinas. Esta visualización es fundamental para comprender mejor el estado actual del problema y los movimientos realizados durante la búsqueda de la solución.

Proceso de Búsqueda Voraz

El algoritmo de búsqueda voraz se utiliza para explorar el espacio de estados. Este método selecciona iterativamente el estado vecino que minimiza la heurística, es decir, que reduce el número de reinas amenazadas. Si no se encuentra un mejor estado, se realiza un reordenamiento aleatorio de las reinas para escapar de posibles mínimos locales.

Proceso de Búsqueda Detallado

1. **Inicialización**: Comienza con la configuración inicial de reinas.
2. **Evaluación Heurística**: Calcula la heurística de la configuración actual.
3. **Generación de Vecinos**: Genera todos los posibles estados vecinos moviendo una reina en su fila.
4. **Selección de Mejor Estado**: Selecciona el vecino con la menor heurística.
5. **Actualización del Estado**: Si se encuentra un estado con menor heurística, se actualiza la configuración actual a este nuevo estado. Si no, se reordena aleatoriamente la configuración.
6. **Detección de Solución**: Si la heurística alcanza cero, se ha encontrado una solución y el proceso termina.

Implementación en C++

La implementación del algoritmo se realiza en C++ utilizando la API de Windows para la interfaz gráfica. A continuación, se describe la estructura y componentes principales del código.

Estructura del Código

1. **Definición de la Heurística** La función heurística calcula el número de amenazas entre reinas en una configuración dada del tablero.
2. **Generación de Vecinos** La función para generar vecinos produce todas las configuraciones posibles moviendo una reina a otra posición en su fila.
3. **Dibujo del Tablero** La función de dibujo crea una representación visual del tablero y las reinas utilizando gráficos de la API de Windows.
4. **Control del Proceso de Búsqueda** La lógica para iniciar, detener y reiniciar la búsqueda está controlada mediante eventos de Windows y botones de la interfaz gráfica.

Descripción Detallada de Componentes Clave

Heurística

La función heurística evalúa cada configuración del tablero y cuenta el número de pares de reinas que se amenazan entre sí. Esto se logra comparando las posiciones de las reinas en términos de filas,

columnas y diagonales. La heurística es fundamental para guiar el proceso de búsqueda hacia configuraciones óptimas.

Generación de Vecinos

La generación de vecinos implica mover cada reina a todas las demás posiciones posibles en su fila y almacenar estas nuevas configuraciones. Esto crea un conjunto de estados vecinos que pueden evaluarse para encontrar el siguiente mejor movimiento.

Interfaz Gráfica

La interfaz gráfica muestra el tablero y permite al usuario iniciar y controlar la búsqueda de una solución. Utilizando la API de Windows, se dibuja el tablero con casillas de diferentes colores y se posicionan las reinas en sus respectivas ubicaciones. La interfaz incluye botones para iniciar la búsqueda, reiniciar la configuración y actualizar la visualización del tablero.

Búsqueda Voraz

La búsqueda voraz explora el espacio de estados moviendo iterativamente hacia la configuración con menor número de amenazas, utilizando la heurística como guía. Si no se encuentra una mejora, se realiza un reordenamiento aleatorio para evitar mínimos locales.

Resultados

El algoritmo de búsqueda voraz, combinado con la heurística del número de reinas amenazadas, fue capaz de encontrar soluciones efectivas al problema de las 5 reinas. La visualización gráfica ayudó a entender mejor el proceso y los resultados. Se encontraron soluciones en las cuales ninguna reina estaba amenazada, demostrando la efectividad del método aplicado.

Conclusiones

El problema de las 5 reinas se puede resolver eficazmente mediante la aplicación de una búsqueda heurística voraz. La heurística basada en el número de reinas amenazadas proporciona una guía adecuada para reducir el espacio de búsqueda y encontrar configuraciones válidas. Este enfoque es aplicable no solo a este problema específico, sino también a otras variaciones y problemas similares en el campo de la optimización y la inteligencia artificial.

Futuras Direcciones

1. **Extensión a N Reinas**: Aplicar y adaptar el algoritmo para resolver el problema general de las N reinas.
2. **Optimización del Algoritmo**: Investigar mejoras en la heurística y en las técnicas de búsqueda para aumentar la eficiencia.
3. **Comparación de Métodos**: Comparar la búsqueda voraz con otros métodos de búsqueda y optimización como algoritmos genéticos o recocido simulado.
4. **Implementaciones en Diferentes Lenguajes**: Explorar implementaciones del

algoritmo en otros lenguajes de programación y entornos para evaluar su desempeño.

Referencias

- Libros de texto sobre inteligencia artificial y teoría de la computación.
- Artículos académicos sobre el problema de las N reinas.
- Documentación y guías sobre programación de interfaces gráficas en C++.
- Recursos en línea sobre algoritmos de búsqueda y optimización heurística.

PROBLEMA DE LAS 5 REINAS EN C++

Código

```
#include <windows.h>
#include <vector>
#include <algorithm>
#include <string>
#include <ctime>
#include <cstdlib>

#define SIZE 5

// Heur?stica: n?mero de reinas amenazadas
int heuristic(const std::vector<int>& queens) {
    int threats = 0;
    for (int i = 0; i < SIZE; ++i) {
        for (int j = i + 1; j < SIZE; ++j) {
            if (queens[i] == queens[j] || abs(queens[i] - queens[j]) == abs(i
- j)) {
```

```
                threats++;
            }
        }
    }
    return threats;
}

// Genera estados vecinos moviendo una reina en la misma fila
std::vector<std::vector<int>> getNeighbors(const std::vector<int>& queens) {
    std::vector<std::vector<int>> neighbors;
    for (int i = 0; i < SIZE; ++i) {
        for (int j = 0; j < SIZE; ++j) {
            if (queens[i] != j) {
                std::vector<int> newQueens = queens;
                newQueens[i] = j;
                neighbors.push_back(newQueens);
            }
        }
    }
    return neighbors;
}

// Dibuja el tablero y las reinas
void drawBoard(HDC hdc, const std::vector<int>& queens) {
    RECT rect;
    for (int i = 0; i < SIZE; ++i) {
        for (int j = 0; j < SIZE; ++j) {
            rect.left = j * 100;
            rect.top = i * 100;
            rect.right = rect.left + 100;
            rect.bottom = rect.top + 100;
```

```
            // Cambia los colores del tablero como en la imagen
             HBRUSH brush = CreateSolidBrush(((i + j) % 2 == 0) ? RGB(233, 174, 95) : RGB(128, 128, 128));
            FillRect(hdc, &rect, brush);
            DeleteObject(brush);

            // Dibuja las reinas
            if (queens[i] == j) {
                Ellipse(hdc, rect.left + 20, rect.top + 20, rect.right - 20, rect.bottom - 20);
            }
        }
    }
}

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {
    static std::vector<int> initialQueens1 = {4, 0, 4, 0, 4};
        static std::vector<int> queens = initialQueens1; // Empezamos con initialQueens1
    static bool searching = false;
    static int solutions = 0;

    switch (uMsg) {
        case WM_PAINT: {
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(hwnd, &ps);
            drawBoard(hdc, queens);
            EndPaint(hwnd, &ps);
            return 0;
        }
        case WM_COMMAND: {
            if (LOWORD(wParam) == 1) { // ID del bot�n "Iniciar B�squeda"
                searching = true;
```

```
                solutions = 0;
                 SetTimer(hwnd, 1, 500, NULL); // Iniciar b�squeda al presionar
el bot�n
            } else if (LOWORD(wParam) == 2) { // ID del bot�n "Reiniciar"
                    queens = initialQueens1; // Reiniciar con la primera
configuraci�n inicial
                searching = false;
                InvalidateRect(hwnd, NULL, TRUE); // Redibujar el tablero
            }
            return 0;
        }
        case WM_TIMER: {
            if (!searching) {
                return 0;
            }
            int minHeuristic = heuristic(queens);
            std::vector<int> nextQueens = queens;

            for (const auto& neighbor : getNeighbors(queens)) {
                int h = heuristic(neighbor);
                if (h < minHeuristic) {
                    minHeuristic = h;
                    nextQueens = neighbor;
                }
            }

            if (minHeuristic == heuristic(queens)) {
                std::random_shuffle(nextQueens.begin(), nextQueens.end());
            }

            queens = nextQueens;
            InvalidateRect(hwnd, NULL, TRUE);
```

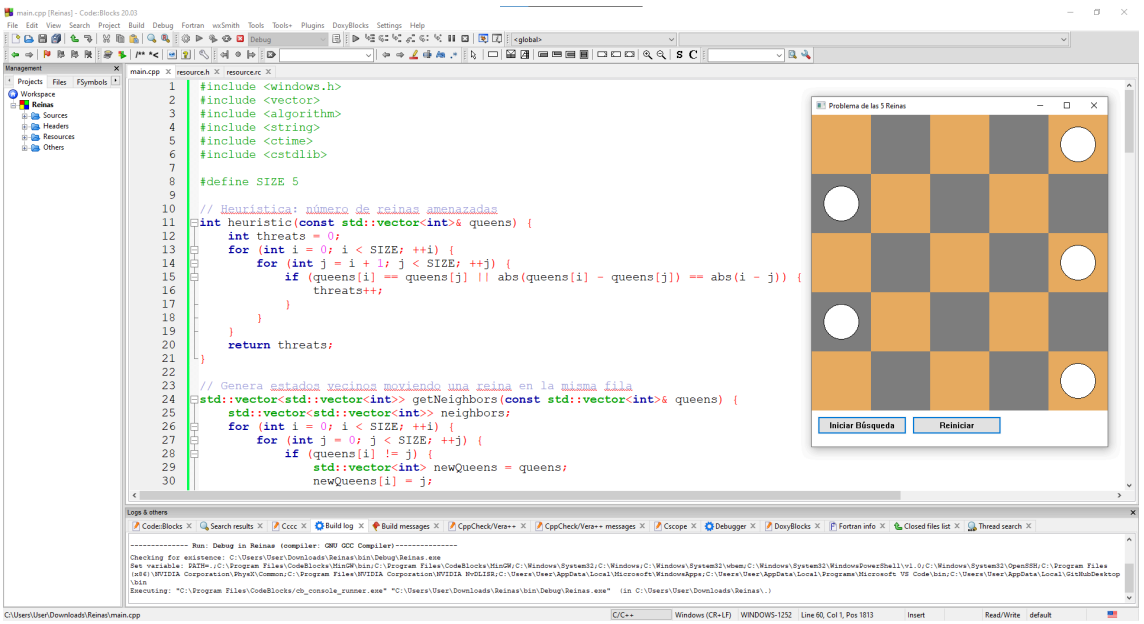
```
            if (minHeuristic == 0) {
                solutions++;
                KillTimer(hwnd, 1);
                searching = false;
                  MessageBox(hwnd, (std::to_string(solutions) + " soluci\x{fffd}n(es) encontrada(s)").c_str(), "Resultado", MB_OK);
            }

            return 0;
        }
        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
        default:
            return DefWindowProc(hwnd, uMsg, wParam, lParam);
    }
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {
    const char CLASS_NAME[] = "FiveQueensClass";

    WNDCLASS wc = {};
    wc.lpfnWndProc = WindowProc;
    wc.hInstance = hInstance;
    wc.lpszClassName = CLASS_NAME;

    RegisterClass(&wc);

    HWND hwnd = CreateWindowEx(
        0,
        CLASS_NAME,
        "Problema de las 5 Reinas",
```

```
        WS_OVERLAPPEDWINDOW,

          CW_USEDEFAULT, CW_USEDEFAULT, SIZE * 100 + 16, SIZE * 100 + 100, //
Ajuste del tama�o de la ventana

        NULL,

        NULL,

        hInstance,

        NULL

    );


    if (hwnd == NULL) {

        return 0;

    }


    // Crear bot�n "Iniciar B�squeda"

    CreateWindow(

        "BUTTON",

        "Iniciar B�squeda",

        WS_TABSTOP | WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON,

        10, SIZE * 100 + 10,

        150, 30,

        hwnd,

        (HMENU)1,

        (HINSTANCE)GetWindowLongPtr(hwnd, GWLP_HINSTANCE),

        NULL);


    // Crear bot�n "Reiniciar"

    CreateWindow(

        "BUTTON",

        "Reiniciar",

        WS_TABSTOP | WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON,

        170, SIZE * 100 + 10,

        150, 30,

        hwnd,
```

```
        (HMENU)2,
        (HINSTANCE)GetWindowLongPtr(hwnd, GWLP_HINSTANCE),
        NULL);

    ShowWindow(hwnd, nCmdShow);

    MSG msg = {};
    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    return 0;
}
```

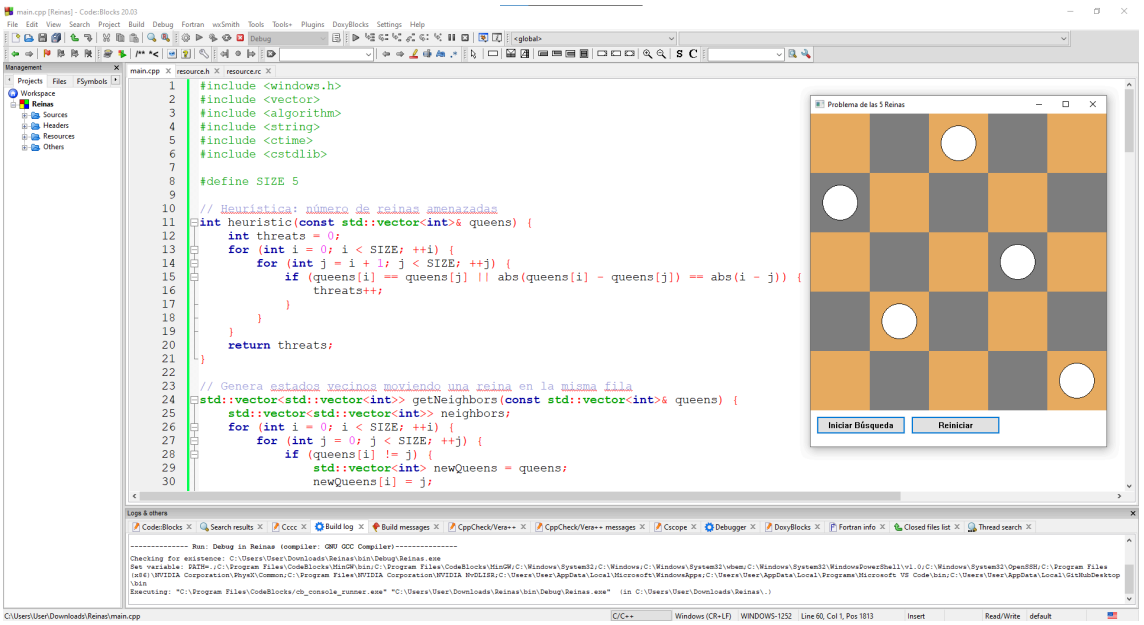
RESULTADO

POSICIÓN INICIAL

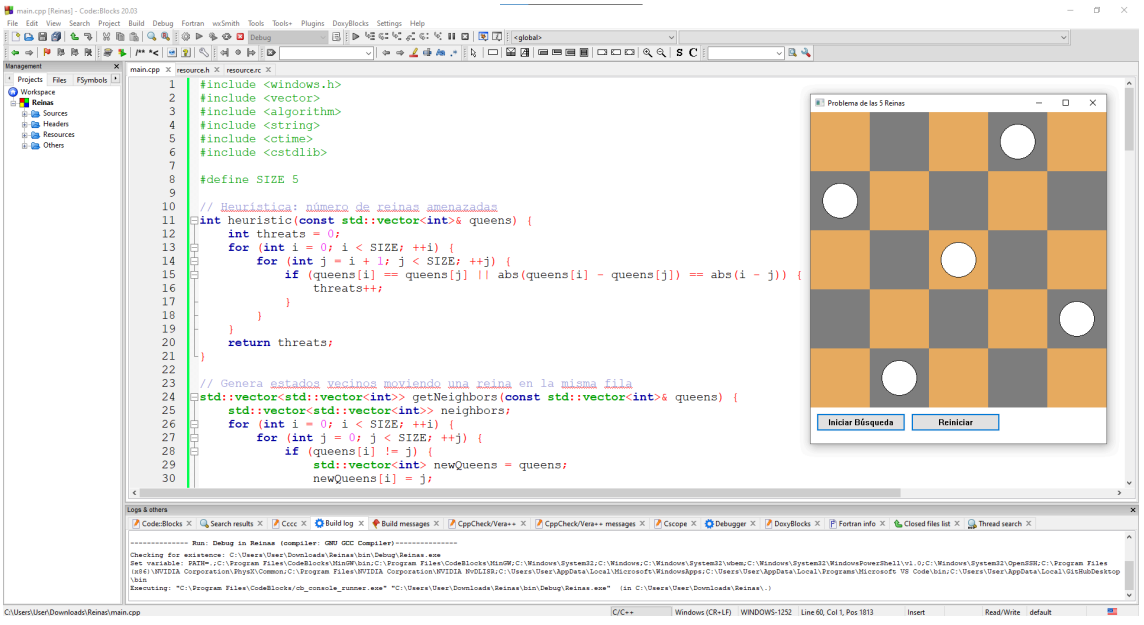


POSIBILIDADES

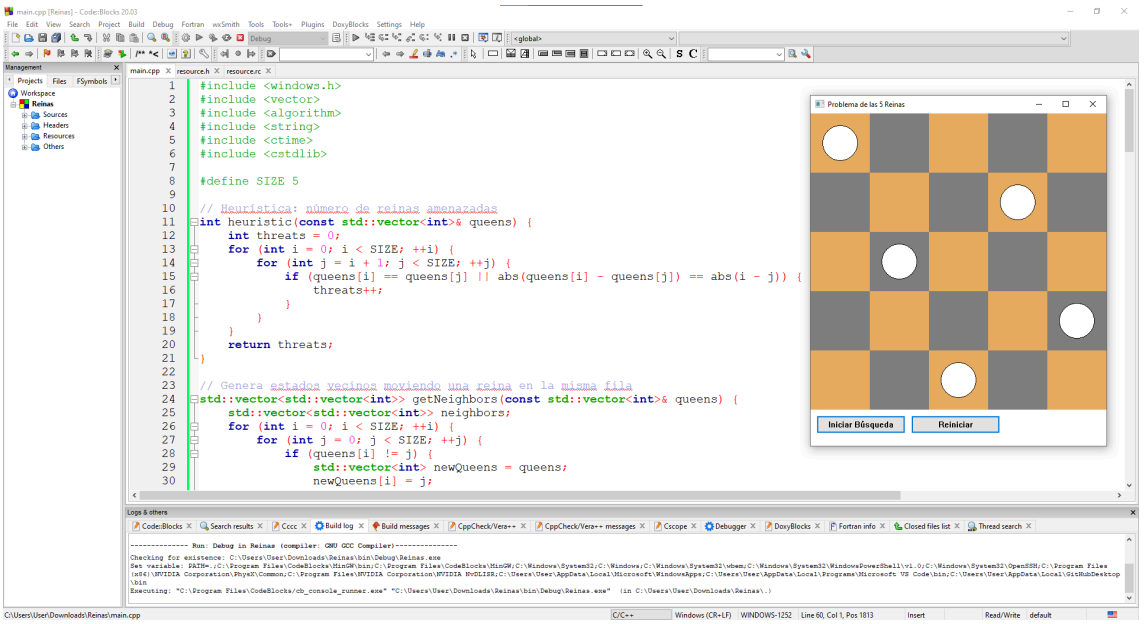
PRIMERA SOLUCIÓN



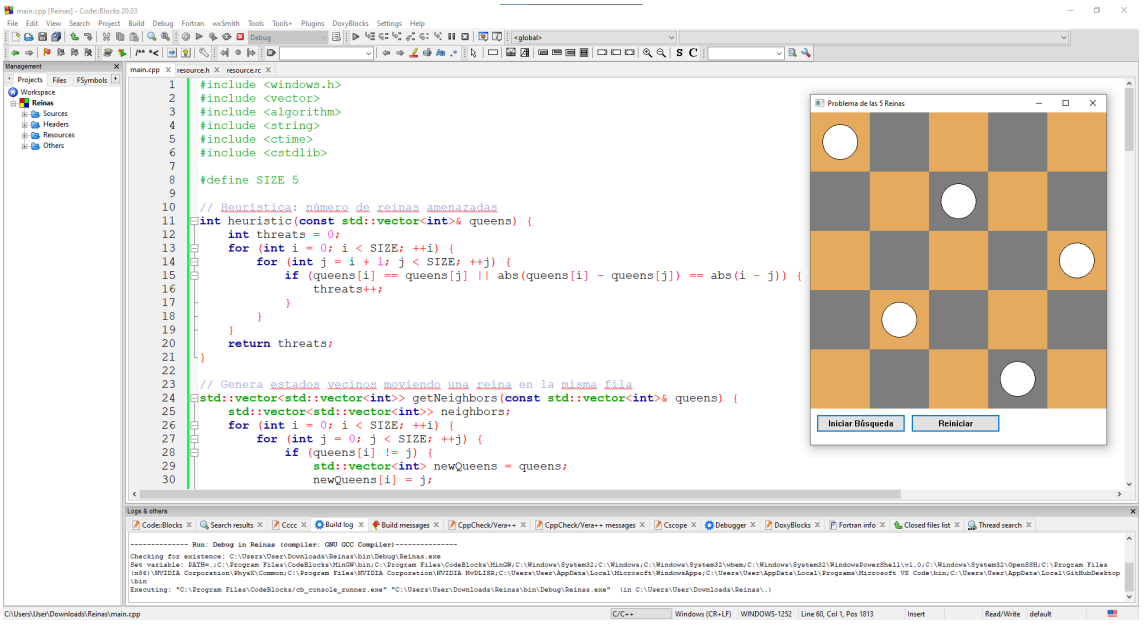
SEGUNDA SOLUCIÓN



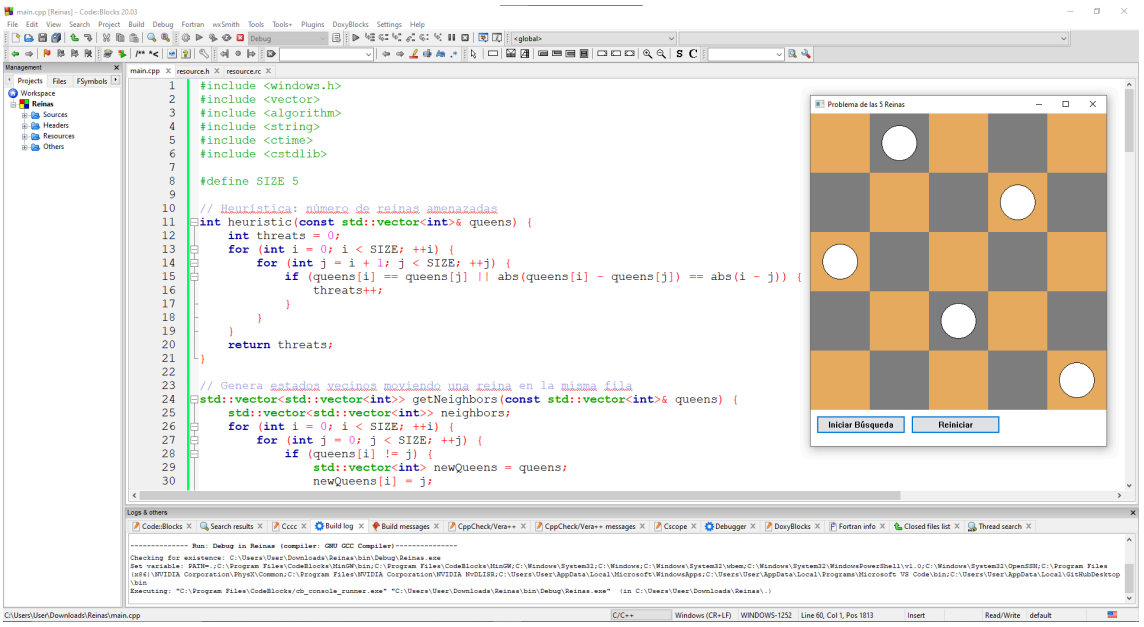
TERCERA SOLUCIÓN



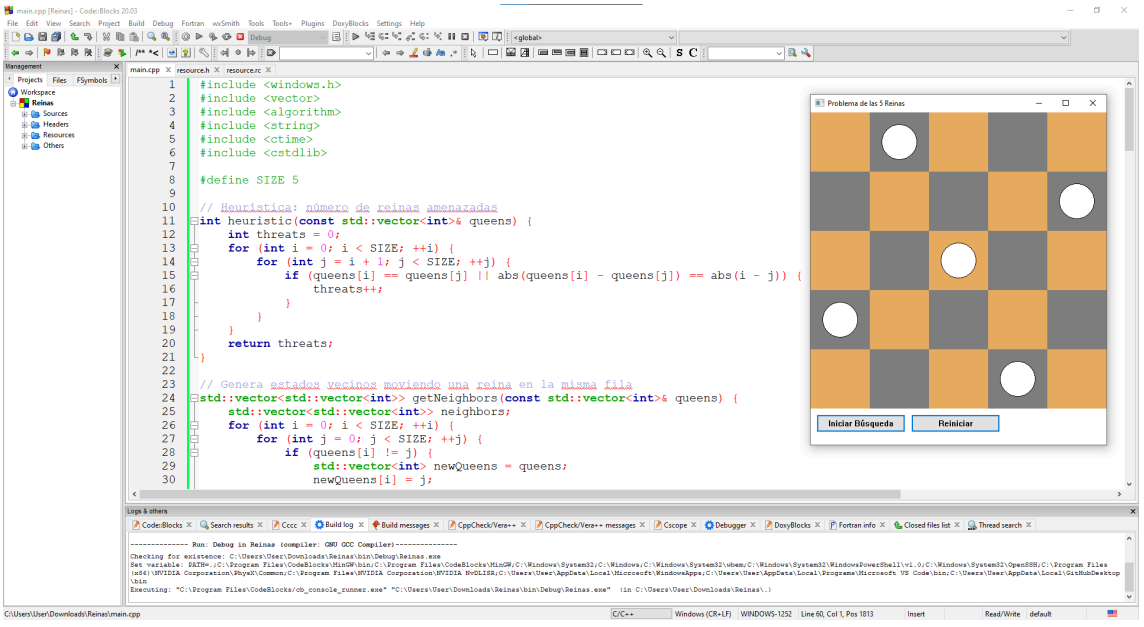
CUARTA SOLUCIÓN



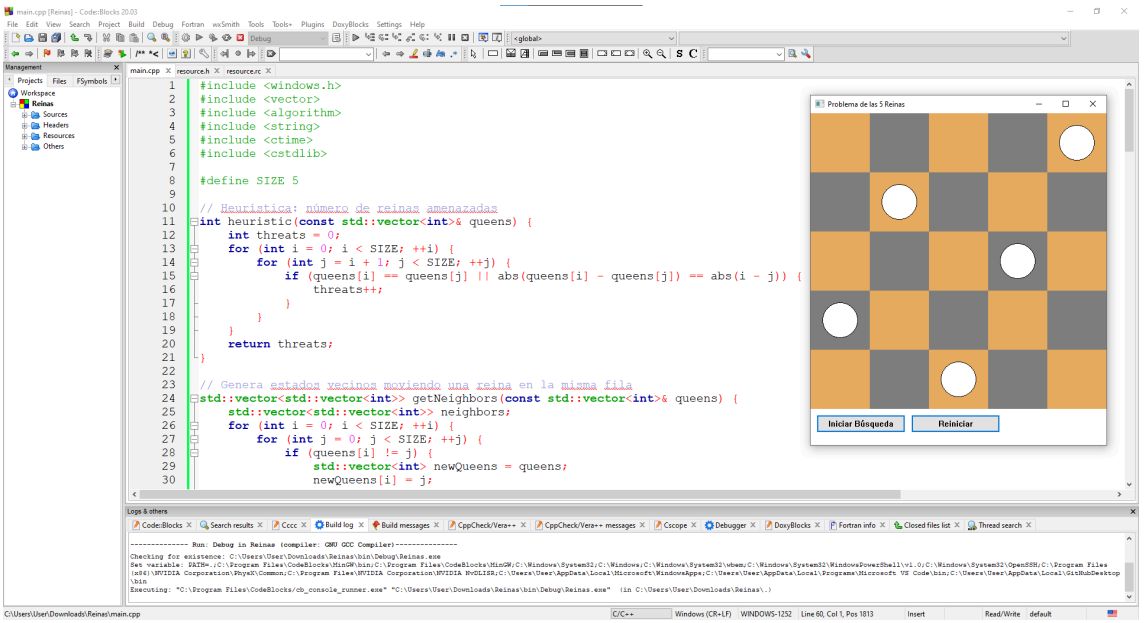
QUINTA SOLUCIÓN



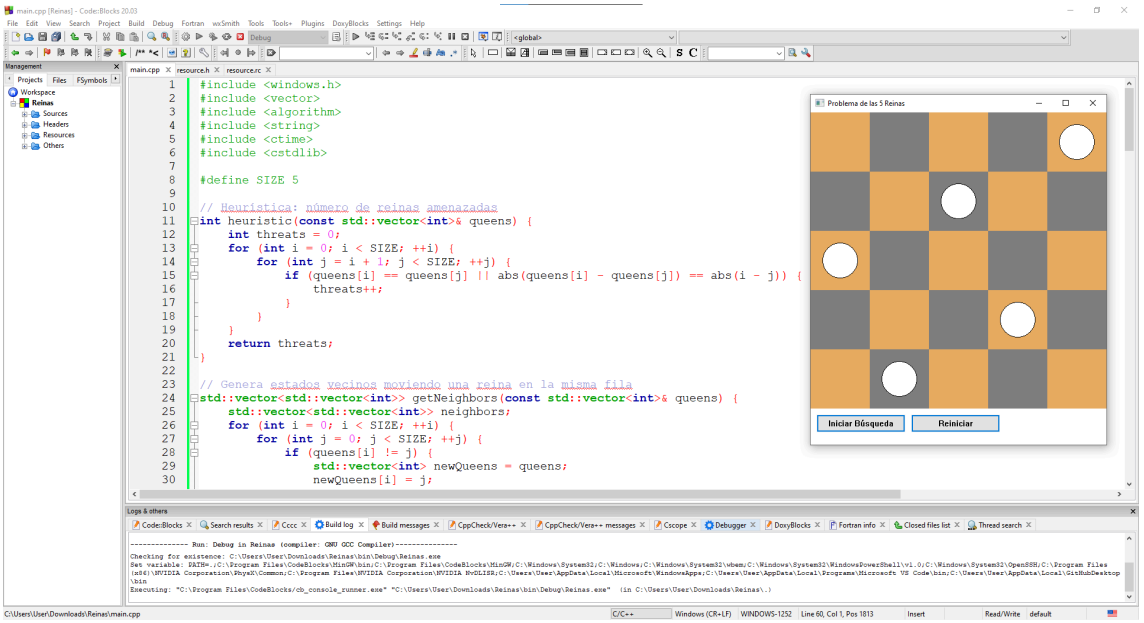
SEXTA SOLUCIÓN



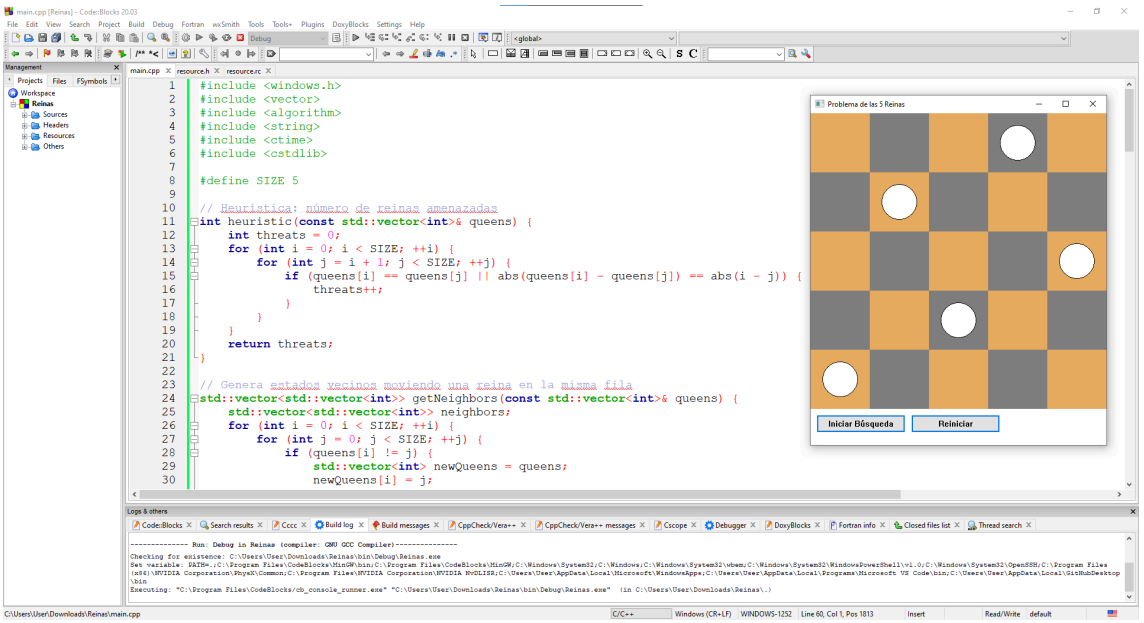
SÉTIMA SOLUCIÓN



OCTAVA SOLUCIÓN



NOVENA SOLUCIÓN



DÉCIMA SOLUCIÓN