Taller 4

Camilo Martinez
Alberto Vigna

May 2024

Análisis del Problema

El problema a resolver es la implementación de dos algoritmos para jugar al juego de Buscaminas: uno utilizando una estrategia de fuerza bruta y otro utilizando una heurística. El objetivo es determinar si es posible completar el tablero sin activar ninguna mina.

Diseño del Algoritmo

Algoritmo de Fuerza Bruta

Entrada:

- T: una matriz de tamaño $m \times n$ donde $T[i][j]$ puede ser 'M' (mina) o 'E' (espacio vacío).

Salida:

- resultado: booleano, verdadero si se logró completar el juego sin activar minas, falso en caso contrario.

Precondiciones:

- T es una matriz de tamaño $m \times n$ y contiene únicamente 'M' o 'E'.

Poscondiciones:

- Se han revelado todas las celdas no minadas si el resultado es verdadero.
- No se ha activado ninguna mina si el resultado es verdadero.

Descripción del Algoritmo:

1. Iterar sobre todas las posibles configuraciones de celdas para descubrir.
2. Comprobar si es posible descubrir todas las celdas sin activar una mina.

Algoritmo Heurístico

Entrada:

- T: una matriz de tamaño $m \times n$ donde $T[i][j]$ puede ser 'M' (mina) o 'E' (espacio vacío).

Salida:

- resultado: booleano, verdadero si se logró completar el juego sin activar minas, falso en caso contrario.

Precondiciones:

- T es una matriz de tamaño $m \times n$ y contiene únicamente 'M' o 'E'.

Poscondiciones:

- Se han revelado todas las celdas no minadas si el resultado es verdadero.
- No se ha activado ninguna mina si el resultado es verdadero.

Descripción del Algoritmo:

1. Utilizar información de celdas descubiertas y sus números para deducir las posiciones de las minas.
2. Realizar deducciones lógicas y descubrimientos seguros basados en la información conocida.

1 Pseudocodigo

1.1 Fuerza Bruta

Algorithm 1 Revisar Casilla Completada

Require: i, j

```
num_minas ← MATRIX[i][j]
num, casillas_adyacentes ← adjacent_squares(i, j)
casillas_adyacentes_marcadas ← obtener_marcadas(casillas_adyacentes)
casillas_adyacentes_no_marcadas ← obtener_no_marcadas(casillas_adyacentes)
if len(casillas_adyacentes_marcadas) = num_minas then
    for casilla_vacia in casillas_adyacentes_no_marcadas do
        if MATRIX[casilla_vacia] = "?" then
            if update_board(casilla_vacia) or has_won() then
                if mina_encontrada then
                    reveal_mines()
                end if
                return casilla_vacia
            end if
        end if
    end for
end if
return casillas_adyacentes_no_marcadas[0]
```

Algorithm 2 Detectar Minas

```
opciones ← obtener_opciones()
for casilla in opciones do
    i, j ← casilla
    num_minas, casillas_adyacentes ← adjacent_squares(i, j)
    casillas_reveladas ← obtener_reveladas(casillas_adyacentes)
    for casilla_adyacente in casillas_reveladas do
        adj_i, adj_j ← casilla_adyacente
        num_minas_adyacentes, casillas_adyacentes_adyacentes ←
        adjacent_squares(adj_i, adj_j)
        casillas_desconocidas ← obtener_desconocidas(casillas_adyacentes_adyacentes)
        if len(casillas_desconocidas) = num_minas_adyacentes then
            MINAS_MARCADAS.add(casilla)
        end if
    end for
end for
print draw_board()
```

Algorithm 3 Solver Fuerza Bruta

```
detectar_minas()
casillas_seguras ← []
casillas_pendientes ← []
for i in range(ROWS) do
    for j in range(COLUMNS) do
        if MATRIX[i][j] = "?" then
            casillas_pendientes.append((i, j))
        else
            casilla_completada ← revisar_casilla_completada(i, j)
            if casilla_completada then
                casillas_seguras.append(casilla_completada)
            end if
        end if
    end for
end for
if has_won() then
    return casillas_seguras[0]
end if
detectar_minas()
for tamano_combinacion in range(1, len(casillas_pendientes) + 1) do
    for combinacion in itertools.combinations(casillas_pendientes, tamano_combinacion)
    do
        if validar_combinacion(combinacion) then
            return combinacion[0]
        end if
    end for
end for
return jugador_aleatorio()
```

Algorithm 4 Jugador Aleatorio

```
available_squares ← obtener_cuadrados_disponibles()
return random.choice(available_squares)
```

La complejidad de `solver_fuerza_bruta()` puede describirse aproximadamente como:

$$O(n + \sum_{k=1}^{m} \binom{m}{k} \cdot O(k))$$

Donde:

n es el número de casillas en el tablero.

m es el número de casillas pendientes (desconocidas).

Esto se simplifica a:

$$O(n + m \cdot 2^m)$$

Ya que $\sum_{k=1}^{m} \binom{m}{k} \cdot k$ es $O(m \cdot 2^m)$.

En resumen, la complejidad del solver es exponencial en el peor de los casos debido a la generación y evaluación de combinaciones, lo que hace que sea un enfoque computacionalmente costoso para resolver el juego de Minesweeper.

1.2 Heuristica

Algorithm 5 Revisar Slots

```
Función revisar_slots(i, j)
minas, casillas ← casillas_adyacentes(i, j)
minas_marcadas ← contar_marcadas(casillas)
if minas_marcadas == minas then
    for cada casilla en casillas do
        if casilla == '?' then
            golpe_mina ← actualizar_tablero(casilla, Verdadero)
            if golpe_mina o ganado() then
                return casilla
            end if
        end if
    end for
end if
return Nada
```

Algorithm 6 Detectar Minas

```
Función detectar_minas()
for cada casilla en matriz do
    if casilla != '?' then
        continuar
    end if
    minas, casillas ← casillas_adyacentes(i, j)
    casillas_cubiertas ← [casilla for casilla in casillas if casilla == '?']
    if longitud(casillas_cubiertas) == minas then
        for cada casilla en casillas_cubiertas do
            matriz[casilla[0]][casilla[1]] ← 'F'
        end for
    end if
end for
```

Algorithm 7 Calcular Probabilidades

```
Función calcular_probabilidades()
probabilidades ← {}
for cada casilla en matriz do
    if casilla != '?' then
        continuar
    end if
    minas, casillas ← casillas_adyacentes(i, j)
    casillas_cubiertas ← [casilla for casilla in casillas if casilla == '?']
    if no casillas_cubiertas then
        continuar
    end if
    probabilidad ← minas / longitud(casillas_cubiertas)
    for cada casilla en casillas_cubiertas do
        if casilla en probabilidades then
            probabilidades[casilla] ← max(probabilidades[casilla], probabilidad)
        else
            probabilidades[casilla] ← probabilidad
        end if
    end for
end for
return probabilidades
```

Algorithm 8 Encontrar Slot Seguro

```
Función encontrar_slot_seguro(probabilidades)
probabilidad_minima ← infinito
casilla_segura ← Nada
for cada casilla, probabilidad en probabilidades do
    if probabilidad ¡ probabilidad_minima then
        probabilidad_minima ← probabilidad
        casilla_segura ← casilla
    end if
end for
return casilla_segura
```

Algorithm 9 Heurística

```
Función heurística()
detectar_minas()
while Verdadero do
    for cada casilla en matriz do
        if casilla != '?' y casilla != 'F' then
            continuar
        end if
        resultado ← revisar_slots(i, j)
        if resultado y ganado() then
            return resultado
        end if
    end for
    probabilidades ← calcular_probabilidades()
    casilla_segura ← encontrar_slot_seguro(probabilidades)
    if casilla_segura then
        golpe_mina ← actualizar_tablero(casilla_segura, Verdadero)
        if golpe_mina o ganado() then
            return casilla_segura
        end if
    else
        romper
    end if
end while
while Verdadero do
    i, j ← aleatorio(0, FILAS - 1), aleatorio(0, COLUMNAS - 1)
    if matriz[i][j] == '?' then
        golpe_mina ← actualizar_tablero((i, j), Verdadero)
        if golpe_mina o ganado() then
            return (i, j)
        end if
    end if
end while
```

La función `heuristic` en el código proporcionado es un solucionador de Buscaminas que utiliza una combinación de estrategias deterministas y probabilísticas para realizar movimientos.

La complejidad temporal de la función `heuristic` está determinada principalmente por los bucles anidados que iteran sobre todo el tablero de juego, que es una cuadrícula 2D de tamaño $FILAS \times COLUMNAS$.

- La función `detectar_minas`, que marca todas las minas que puede determinar con seguridad, itera sobre todo el tablero, lo que hace que su complejidad temporal sea $O(FILAS \times COLUMNAS)$.
- El bucle principal de la función `heuristic` también itera sobre todo el

tablero varias veces. En el peor de los casos, podría iterar potencialmente sobre el tablero una vez por cada cuadrado, lo que hace que su complejidad temporal sea $O((FILAS \times COLUMNAS)^2)$.

- La función `calcular_probabilidades`, que calcula la probabilidad de que cada cuadrado no visitado sea una mina, también itera sobre todo el tablero, lo que hace que su complejidad temporal sea $O(FILAS \times COLUMNAS)$.
- La función `encontar_slot_seguro`, que encuentra el cuadrado con la menor probabilidad de ser una mina, itera sobre todas las probabilidades calculadas, que en el peor de los casos es el número de cuadrados, lo que hace que su complejidad temporal sea $O(FILAS \times COLUMNAS)$.

Por lo tanto, la complejidad temporal general de la función `heuristic` es $O((FILAS \times COLUMNAS)^2)$ en el peor de los casos.

La complejidad espacial de la función `heuristic` está determinada por el almacenamiento del tablero de juego y el conjunto de probabilidades. Ambas estructuras de datos pueden almacenar potencialmente una entrada para cada cuadrado en el tablero, lo que hace que la complejidad espacial sea $O(FILAS \times COLUMNAS)$.

Por lo tanto, la complejidad temporal general de la función `heuristic` es $O((FILAS * COLUMNAS)^2)$ en el peor de los casos.

2 Comparación Experimental

Para las pruebas se crearon dos códigos modificados (`fb_sim.py` y `heu_sim.py`), los cuales devolvían cada uno un archivo `.csv` con el tiempo para cada simulación y si lograron ganar. De esas tablas, obtuvimos la siguiente información:

Table 1: Comparación entre Fuerza Bruta y Heurística

	Fuerza Bruta	**Heurística**
Promedio Tiempos (s)	0.003470003605	0.001286444664
Juegos Ganados	61	39
Juegos Perdidos	39	61

1. **Promedio de Tiempos:**
   - Fuerza Bruta: 0.003470003605 segundos
   - Heurística: 0.001286444664 segundos

   La Heurística muestra un tiempo promedio significativamente menor en comparación con la Fuerza Bruta. Esto sugiere que la Heurística es más eficiente en términos de tiempo de ejecución para resolver el tablero de buscaminas y efectivamente se comprueba al ver las complejidades en la sección anterior.

2. **Juegos Ganados:**

   - Fuerza Bruta: 61 juegos ganados
   - Heurística: 39 juegos ganados

   La Fuerza Bruta logró ganar más juegos que la Heurística. Esto podría indicar que la Fuerza Bruta tiene una estrategia más efectiva para resolver ciertos tipos de tableros de buscaminas que la Heurística y teniendo encuenta la logica de esta, tiene sentido que se equivoque de igual manera encuentra la respuesta casi el 40% de las veces.

3. **Juegos Perdidos:**

   - Fuerza Bruta: 39 juegos perdidos
   - Heurística: 61 juegos perdidos

   La Heurística tuvo más juegos perdidos en comparación con la Fuerza Bruta. Esto sugiere que la Heurística puede tener una tasa de fallo más alta en ciertas situaciones o configuraciones de tablero.

En resumen, la Heurística muestra una mayor eficiencia en términos de tiempo de ejecución, pero la Fuerza Bruta logró ganar más juegos. La elección entre utilizar la Heurística o la Fuerza Bruta dependerá de la prioridad dada a la velocidad versus la efectividad en términos de ganar juegos. Tambien se ha de aclarar que con tableros más grandes o con más minas, la complejidad de la opción de fuerza bruta la puede volver inviable a tales escalas.