UNIVERSIDAD
DE GRANADA

Algorítmica

Algoritmos de Vuelta Atrás (Backtracking) para el Senku

UNIVERSIDAD
DE GRANADA

ETSIIT
Escuela Técnica Superior
de Ingenierías Informática
y de Telecomunicación

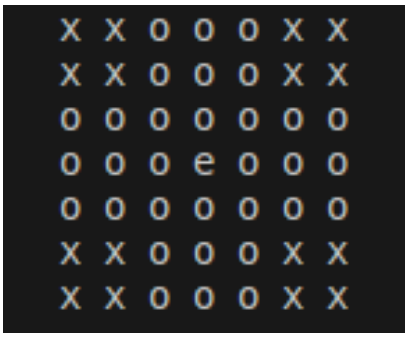
Juan Miguel Acosta Ortega

UNIVERSIDAD
DE GRANADA

UNIVERSIDAD
DE GRANADA

Solo puede quedar uno (Backtracking)

Se trata de un solitario donde se colocan 32 piezas iguales en un tablero de 33 casillas, tal y como se indica en la figura siguiente (las “x” corresponden a posiciones no válidas):

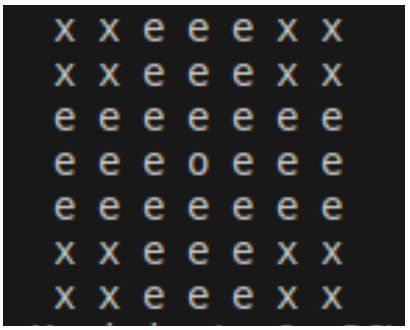


Sólo se permiten movimientos de las piezas en vertical y horizontal. Una pieza solo puede moverse saltando sobre otra y situándose en la siguiente casilla, que debe estar vacía. La pieza sobre la que se salta se retira del tablero. Se consigue terminar con éxito el juego cuando queda una sola pieza en la posición central del tablero (la que estaba inicialmente vacía). Diseñad e implementad un algoritmo vuelta atrás que encuentre una serie de movimientos para llegar con éxito al final del juego. ¿Cuántas soluciones distintas hay?

Planteamiento del ejercicio

El problema se aborda con “Backtracking”, lo que quiere decir que se explorarán diferentes caminos posibles, retrocediendo en los pasos en los que no se cumpla “la función factibilidad”, hasta dar con una o varias soluciones.

El objetivo es realizar acciones factibles con las fichas hasta resultar en el tablero solución, es decir, sólo quedará una ficha en el centro, como el título del juego indica:



Para esto modularizamos el problema en las partes características de un problema de “Backtracking”:

Representación del problema:

```
const int SIZE = 7;
const char EMPTY = 'e';
const char PLAYER = 'o';
const char INVALID = 'x';
bool impreso = false;

enum class Action;

struct Position
{
    int x;
    int y;
    Position(int x, int y) : x(x), y(y) {}

    bool operator <(const Position &p) const
    {
        return (x < p.x) || (x == p.x && y < p.y);
    }
};

enum class Action
{
    UP,
    DOWN,
    LEFT,
    RIGHT
};
```

Representación de la solución: Esta ha de ser una lista de las diferentes decisiones que se van tomando hasta llegar a la solución. Como las fichas (estas se encuentran en una fila “x”, y una columna “y”) , sólo se pueden mover en 4 direcciones posibles (“UP”, “DOWN”, “LEFT”, “RIGHT”), la solución será un “pair” de Acción y Posición.

```
vector<pair<Action, Position>> &decisiones
```

Restricciones implícitas: Cada ficha que se encuentra en una “Position” que se ha de encontrar dentro del tablero en una ficha que no sea “INVALID”. Como ya se ha mencionado anteriormente sólo se pueden mover en 4 direcciones.

Restricciones explícitas: Además para que una ficha se pueda “mover” se ha de tener en cuenta que la casilla siguiente ha de ser una ficha “o”, y que la siguiente de la siguiente ha de estar vacía “e”, además de calcular que no se salga del tablero.

Función factibilidad: Los dos tipos de restricciones desembocan en la “función que dictamina la factibilidad según la posición y la acción”. Estas restricciones están ligadas a la “función de poda” que nos permitirá eliminar exploraciones de ramas inadecuadas.

```
bool isValidMove(vector<vector<char>> &board, const Position &pos, const Action &a)
{ // FUNCIÓN FACTIBILIDAD
    bool valid = false;
    if (board[pos.x][pos.y] == PLAYER) // SI LA POSICIÓN ACTUAL ES UNA FICHA
    {
        switch (a)
        {
        case Action::UP:
            if (pos.x - 2 >= 0 && pos.x - 2 != INVALID)
            { // SIGO EN EL TABLERO TRAS LA ACCIÓN
                if (board[pos.x - 1][pos.y] == PLAYER && board[pos.x - 2][pos.y] == EMPTY)
                { // HAY UNA FICHO EN EL SIGUIENTE MOVIMIENTO, Y DESPUÉS HAY UN ESPACIO VACÍO
                    valid = true;
                }
            }
            break;
```

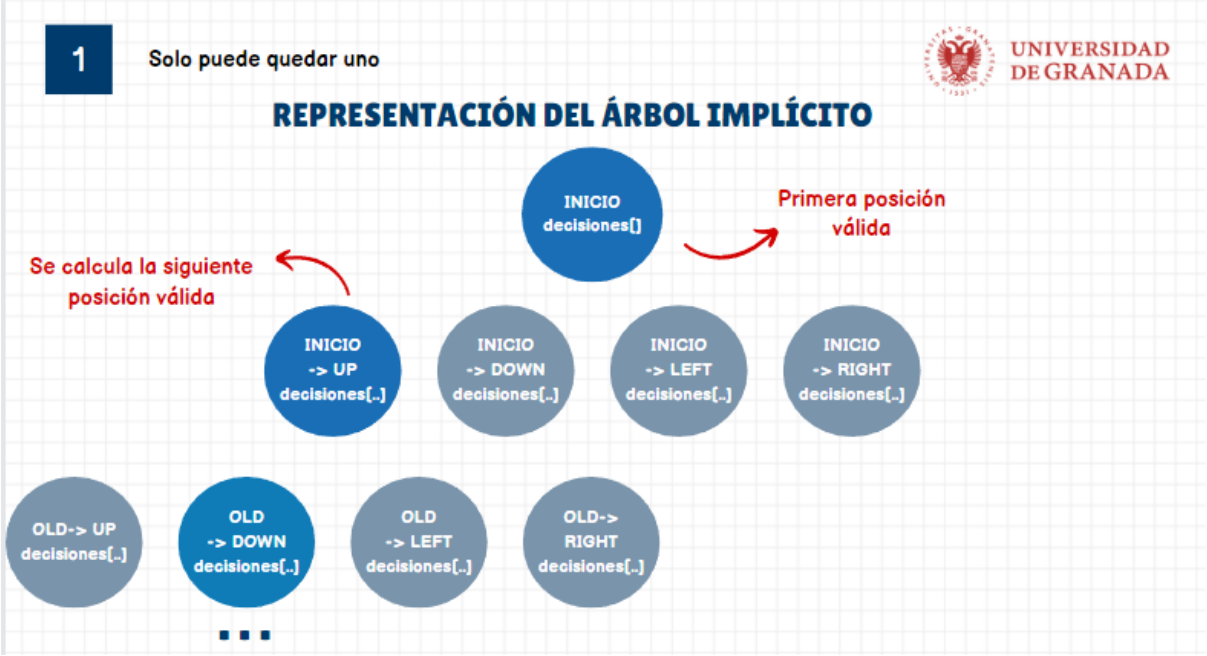
Función objetivo: Cuando el número de decisiones tomadas (fichas movidas), sea de 31, significará que de las 32 casillas sólo queda una, y si además esta se encuentra en el centro del tablero, hemos encontrado una solución.

```
// Caso base: se ha alcanzado la condición de éxito (quedó una sola pieza en el centro)
if (decisiones.size() == 31 && board[SIZE / 2][SIZE / 2] == PLAYER && impreso == false)
{
    imprimeTablero(board);
    imprimePlan(decisiones);
    impreso = true;
    return true; // Éxito

}else if(decisiones.size() > 31) return false;
```

Representación del árbol implícito:

- ***Estado inicial:*** La lista de movimientos y posiciones estará vacía.

- Se buscará en el tablero las **fichas que puedan realizar alguno de los movimientos posibles** y se anotará, es decir, al expandir los nodos (la siguiente posición del tablero factible), son 4 por la posibilidad de acciones que hay.



Además se han implementado diferentes funciones tanto como para **realizar una acción** dada como para hacer **rollBack** de la misma:

```
void executeMove(vector<vector<char>> &board, Position &pos, const Action &a)
{ // FUNCIÓN EJECUCIÓN
    switch (a)
    {
    case Action::UP:
        board[pos.x][pos.y] = EMPTY;
        board[pos.x - 1][pos.y] = EMPTY;
        board[pos.x - 2][pos.y] = PLAYER;
        pos.x = pos.x - 2;
        break;
    case Action::DOWN:
        board[pos.x][pos.y] = EMPTY;
        board[pos.x + 1][pos.y] = EMPTY;
        board[pos.x + 2][pos.y] = PLAYER;
        pos.x = pos.x + 2;
        break;
    case Action::LEFT:
        board[pos.x][pos.y] = EMPTY;
        board[pos.x][pos.y - 1] = EMPTY;
        board[pos.x][pos.y - 2] = PLAYER;
        pos.y = pos.y - 2;
        break;
    case Action::RIGHT:
        board[pos.x][pos.y] = EMPTY;
        board[pos.x][pos.y + 1] = EMPTY;
        board[pos.x][pos.y + 2] = PLAYER;
        pos.y = pos.y + 2;
        break;
    }
}
```

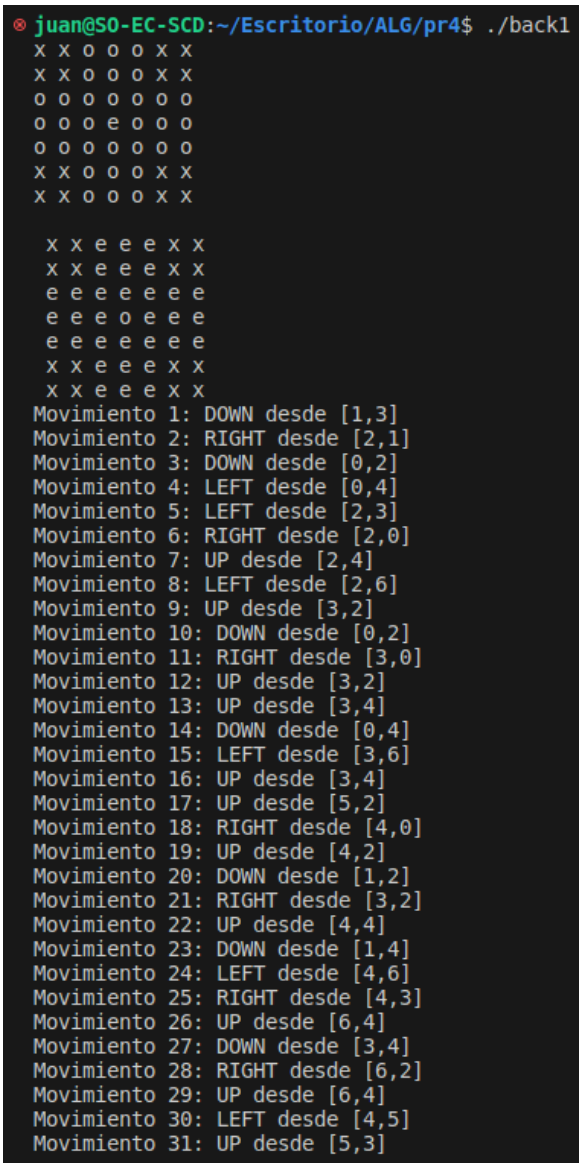
```
void rollBack(vector<vector<char>> &board, Position &pos, const Action &a)
{ // FUNCIÓN RETROCESO
    switch (a)
    {
    case Action::UP:
        board[pos.x][pos.y] = PLAYER;
        board[pos.x - 1][pos.y] = PLAYER;
        board[pos.x - 2][pos.y] = EMPTY;
        pos.x = pos.x - 2;
        break;
    case Action::DOWN:
        board[pos.x][pos.y] = PLAYER;
        board[pos.x + 1][pos.y] = PLAYER;
        board[pos.x + 2][pos.y] = EMPTY;
        pos.x = pos.x + 2;
        break;
    case Action::LEFT:
        board[pos.x][pos.y] = PLAYER;
        board[pos.x][pos.y - 1] = PLAYER;
        board[pos.x][pos.y - 2] = EMPTY;
        pos.y = pos.y - 2;
        break;
    case Action::RIGHT:
        board[pos.x][pos.y] = PLAYER;
        board[pos.x][pos.y + 1] = PLAYER;
        board[pos.x][pos.y + 2] = EMPTY;
        pos.y = pos.y + 2;
        break;
    }
}
```

Algoritmo Backtracking

En última instancia el algoritmo backtracking hace una síntesis de lo comentado anteriormente, y es que recorre el tablero buscando fichas, y comprueba si esta puede realizar alguna de las acciones disponibles. Si puede realizar alguna se ejecuta, se guarda la información en la solución (decisiones), y se llama recursivamente a la función. Si no encuentra ningún movimiento válido se hace “rollBack” y se descarta esa decisión en la solución final.

```cpp
bool backTrackingSolitaire(vector<pair<Action, Position>> &decisiones, vector<vector<char>> &board, int contador)
{
    // Caso base: se ha alcanzado la condición de éxito (quedó una sola pieza en el centro)

    // Recorremos el tablero
    for (int i = 0; i < SIZE; i++)
    {
        for (int j = 0; j < SIZE; j++)
        {
            if (board[i][j] == PLAYER)
            {
                // Recorremos las direcciones
                Position current = {i, j};
                for (Action direccion : {Action::UP, Action::DOWN, Action::LEFT, Action::RIGHT})
                {

                    // Si el movimiento es válido
                    if (isValidMove(board, current, direccion))
                    {
                        // Realizar el movimiento
                        Position old = current;
                        executeMove(board, current, direccion);

                        // imprimeTablero(board);

                        // Agregar la acción y la posición al vector de decisiones
                        decisiones.push_back(make_pair(direccion, old));
                        // cout << "Decisiones: " << decisiones.size() << endl;
                        // cout << "Movemos la casilla [" << old.x << "][" << old.y << "] en la dirección " << direccion << endl;

                        // Llamada recursiva
                        if (backTrackingSolitaire(decisiones, board, contador))
                            return true;

                        // Retroceso
                        rollBack(board, old, direccion);
                        decisiones.pop_back();
                    }
                }
            }
            if (decisiones.size() == 31 && board[SIZE / 2][SIZE / 2] == PLAYER && impreso == false)
            {

                impreso = true;
                return true; // Éxito
            }
            else if (decisiones.size() > 31)
                return false;
        }
    }
}
```

Una ejecución del algoritmo daría una solución del siguiente tipo:



Conclusiones

Eficiencia: La eficiencia de un algoritmo Backtracking suele ser de tipo exponencial, y se ha de tener en cuenta:

- El tiempo necesario para calcular la siguiente componente de la solución
- El número de componentes que satisfacen las restricciones explícitas
- El tiempo de determinar la función de factibilidad

- El número de componentes que satisfacen la función de factibilidad (número de nodos generados). Es la única que varía de unos casos a otros.

- Si en el peor de los casos el número de nodos en el espacio solución es d^n o n! el tiempo de este caso sería O(p(n)*d^n) u O(q(n)*n!) con p y q polinomios en n.

Analizando la eficiencia podemos concluir la importancia del backtracking para resolver casos con grandes valores de n en un tiempo reducido.

¿Cuántas soluciones tendría el juego?

El tablero es simétrico horizontal, vertical y diagonalmente, por lo que cada solución genera otras 7 por simetría. El juego tiene cierta dificultad. Notar que hay 2^32 = 4.294.967.296 posiciones posibles de peonzas en el tablero, aunque no todas pueden ser obtenidas como una sucesión de jugadas legales.

El número total de sucesiones de juego posibles es 577.116.156.815.309.849.672, de las cuales **40.861.647.040.079.968 son soluciones** (esto se hace por búsquedas a fuerza bruta con computadoras).

Es decir que la proporción de juegos en que se gana sobre el total es 40.861.647.040.079.968 / 577.116.156.815.309.849.672 = 0,00007080315 o sea de solamente un 0,00708315 %.

Es por esto que nuestro algoritmo es rápido para dar una solución, pero sería mucho más tardado y costoso computacionalmente hablando dar todas las posibilidades.