Backtracking

Backtracking

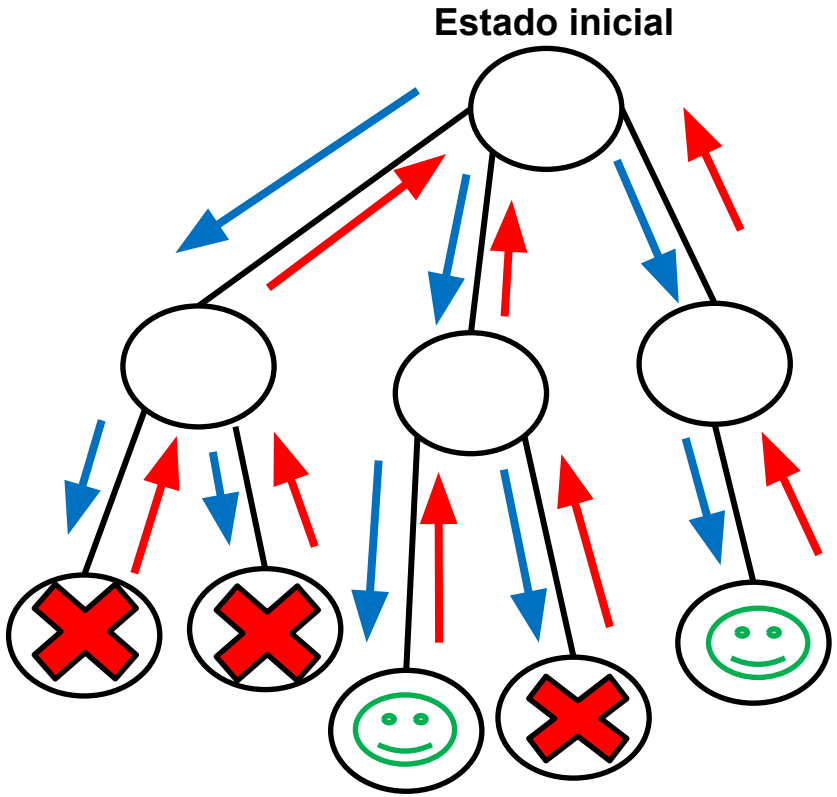
- Técnica de fuerza bruta basada en la recursividad.
- Permite iterar por todas las posibles configuraciones de un espacio de búsqueda.
- Se usa cuando es necesario obtener una configuración que cumpla ciertas reglas y los constraints dados son pequeños.

Objetivos comunes del backtracking

- Encontrar 1 configuración válida (o determinar si no existe ninguna)
- Encontrar todas las configuraciones válidas.
- Contar cuántas configuraciones válidas hay
- Encontrar la mejor de las configuraciones válidas (bajo determinado criterio)

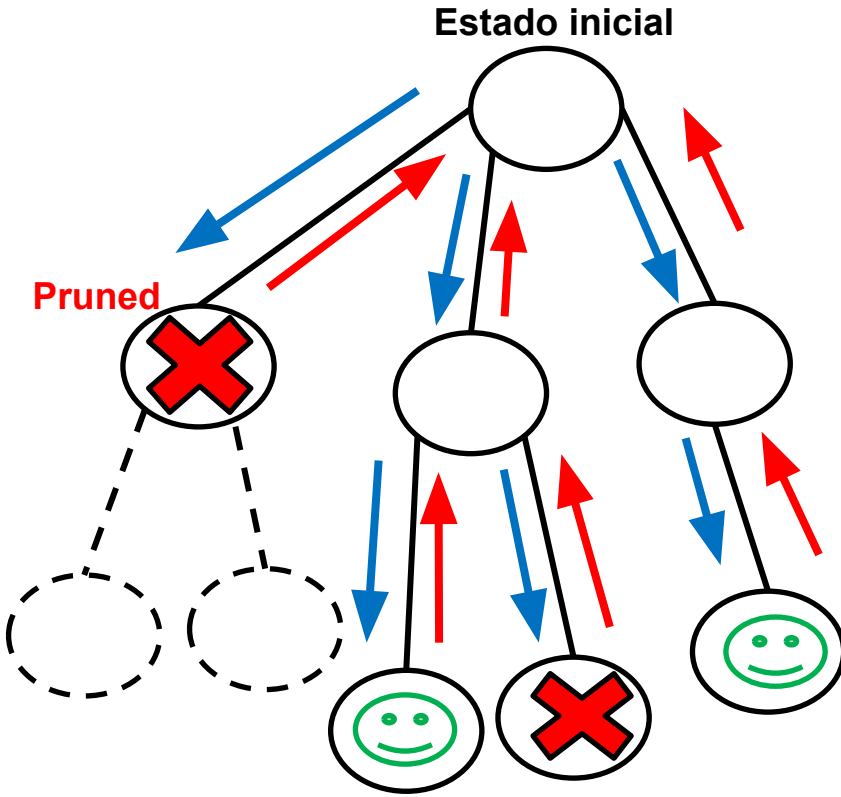
Backtracking

- Aprovecha el recorrido en profundidad que tienen las funciones recursivas
- En esta técnica, se van construyendo recursivamente las soluciones, armando poco a poco las configuraciones candidatas, basándose en el estado actual de la configuración.
- Cuando se determina que la configuración candidata no es una solución válida, el algoritmo retrocede (backtrack), para seguir generando otras configuraciones a partir de un estado anterior.



Pruning

- Es una especie de “validación temprana” que se realiza para optimizar el backtracking.
- Esta validación nos permite saber si una configuración está siendo formada correctamente en lugar de esperar a que se haya completado toda la configuración y recién ahí validar.
- Con el pruning ya no es necesario explorar todas las alternativas.



Estructura general del backtracking

El procedimiento recursivo general es el siguiente:

- (Opcional: Prunning): Si detecto que estoy en un estado actual que no puede generar configuraciones válidas, retorno la llamada recursiva al estado anterior.
- Si me encuentro en un caso base en donde ya conseguí una configuración, la añado a mis configuraciones encontradas
- Si todavía me falta seguir construyendo la configuración, pruebo todas las opciones posibles a partir de la configuración actual.
  - Por cada posible opción, armo una parte más de la configuración (**pruebo** la opción)
  - Luego, avanzo recursivamente al siguiente paso
  - Cuando termino de evaluar recursivamente esa opción, **deshago** la opción utilizada para volver al estado anterior y continuar con la siguiente opción.

Estructura general del backtracking

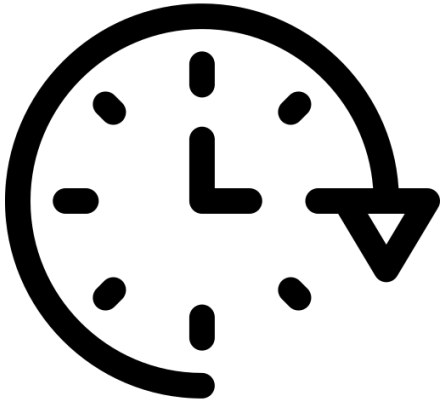
Pseudocódigo general para encontrar todas las configuraciones que cumplen cierta condición.

```
vector<Configuration> configurations;

bool canPrune(Configuration conf, Option option);
bool isValid(Configuration conf);
bool isComplete(Configuration conf);
void make(Configuration conf, Option option);
void undo(Configuration conf, Option option);

void generateAllConfigurations(Configuration conf) {
    if (isComplete(conf)) {
        if (isValid(conf)) configurations.push_back(conf);
        return;
    }
    vector<Option> options;
    for (auto option : options) {
        if (canPrune(conf, option)) continue; // pruning
        make(conf, option);
        generateAllConfigurations(conf);
        undo(conf, option);
    }
}
```

Time Complexity

Para calcular el time complexity podemos hacerlo con un análisis similar a como se hizo con las funciones recursivas en general: estimando el número de estados y transiciones (nodos y aristas en nuestro grafo de recursión).

En backtracking, las complejidades suelen ser exponenciales o factoriales. Las podas también pueden reducir la complejidad, aunque en otras ocasiones solo mejoran la constante.



Permutaciones

Generar todas las permutaciones de los números del 1 al n.

```
bool isPermutation(vector<int> &permutation) {
    int n = permutation.size();
    vector<bool> used(n + 1);
    for (int x : permutation) {
        if (used[x]) return false;
        used[x] = true;
    }
    return true;
}
```

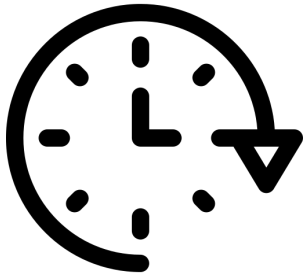
```
vector<vector<int>> allPermutations;

void generatePermutations(vector<int> &permutation, int n) {
    if (permutation.size() == n) {
        if (isPermutation(permutation)) {
            allPermutations.push_back(permutation);
        }
        return;
    }
    for (int x = 1; x <= n; x++) {
        permutation.push_back(x); // make
        generatePermutations(permutation, n);
        permutation.pop_back(); // undo
    }
}
```

Permutaciones

En el ejemplo anterior, no se usó ninguna poda. Siempre utilizamos las n posibles opciones, por lo que el número de estados totales de recursión es igual a $1 + n + n^2 + n^3 + \cdots + n^n = O(n^{n+1})$. Como cada iteración del ***for*** nos lleva a un estado válido, entonces la complejidad en suma de todos los bucles utilizados en la función recursiva es también igual a $O(n^{n+1})$

En los estados finales de nuestro de árbol de recursión, utilizamos $isPermutation$ que es $O(n)$, como hay n^n estados finales, la complejidad total de los casos bases también es $O(n^{n+1})$.

Sumando todo esto, podemos concluir que la complejidad final del algoritmo es $O(n^{n+1})$



Permutaciones

Podemos mejorar la complejidad usando poda. En este caso, revisaremos en todo momento si la permutación es válida (no tiene elementos repetidos).

```cpp
vector<vector<int>> allPermutations;
const int MX = 20;
bool used[MX];

void make(vector<int> &permutation, int x) {
    permutation.push_back(x);
    used[x] = true;
}

void undo(vector<int> &permutation, int x) {
    permutation.pop_back();
    used[x] = false;
}
```

```cpp
void generatePermutations(vector<int> &permutation, int n) {
    if (permutation.size() == n) {
        allPermutations.push_back(permutation);
        return;
    }
    for (int x = 1; x <= n; x++) {
        if (used[x]) continue; // pruning
        make(permutation, x);
        generatePermutations(permutation, n);
        undo(permutation, x);
    }
}
```

Permutaciones

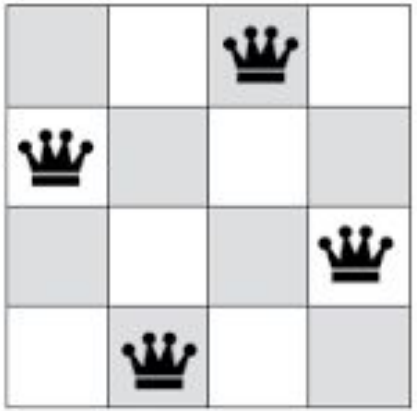
El número de estados en esta función recursiva es ahora $1 + n + n(n-1) + \cdots + n! = O(n \times n!)$. Teniendo en cuenta que cada estado siempre se hacen n iteraciones, donde algunas de ellas se podan, entonces la complejidad total, contabilizando los bucles, sería $O(n^2 \times n!)$.

De todas formas, esto es una mejora con respecto a la complejidad anterior, sin poda, que era $O(n^{n+1})$.

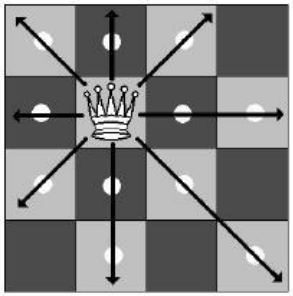
Bonus: Si en vez de un arreglo booleano, utilizamos un bitmask, podemos iterar directamente sobre todos los elementos que tengan todavía bits prendidos, bajando la complejidad hasta $O(n \times n!)$**.**

N Reinas

Dado un tablero de $n \times n$ se desea colocar n reinas de tal forma que ninguna de ellas pueda atacar a las otras.

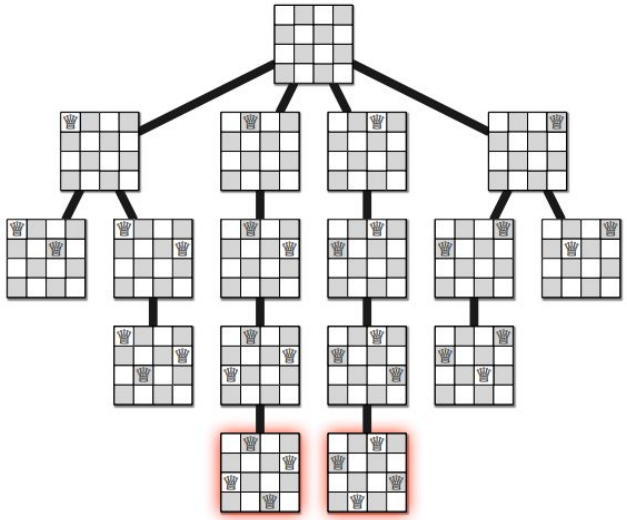


Una reina puede moverse a lo largo de una fila, columna o diagonal de la posición en que se encuentra.



N Reinas

En el algoritmo backtracking necesitaremos mantener el estado del tablero del ajedrez de forma que podamos hacer podas y evitar caer en posiciones donde haya reinas que se ataquen.



Fuente: Algorithms. Jeff Erickson [1]

N reinas

Necesitamos guardar la siguiente información:

- Si una fila ya está ocupada por una reina
- Si una columna ya está ocupada por una reina
- Si una diagonal principal ya está ocupado por una reina
- Si una diagonal secundaria ya está ocupada por una reina

Podemos usar 4 arreglos booleanos. Identificar filas y columnas es sencillo ya que podemos usar su número. Sin embargo, para poder identificar las diagonales necesitamos darnos cuentas de ciertas propiedades geométricas.

En una diagonal principal se cumple que la diferencia de las coordenadas $(fila - columna)$ es constante, mientras que en una diagonal secundaria, la suma de coordenadas $(fila + columna)$ es constante.

Diagonal principal

Fila /col	0	1	2	3
0	0	-1	-2	-3
1	1	0	-1	-2
2	2	1	0	-1
3	3	2	1	0

Diagonal secundaria

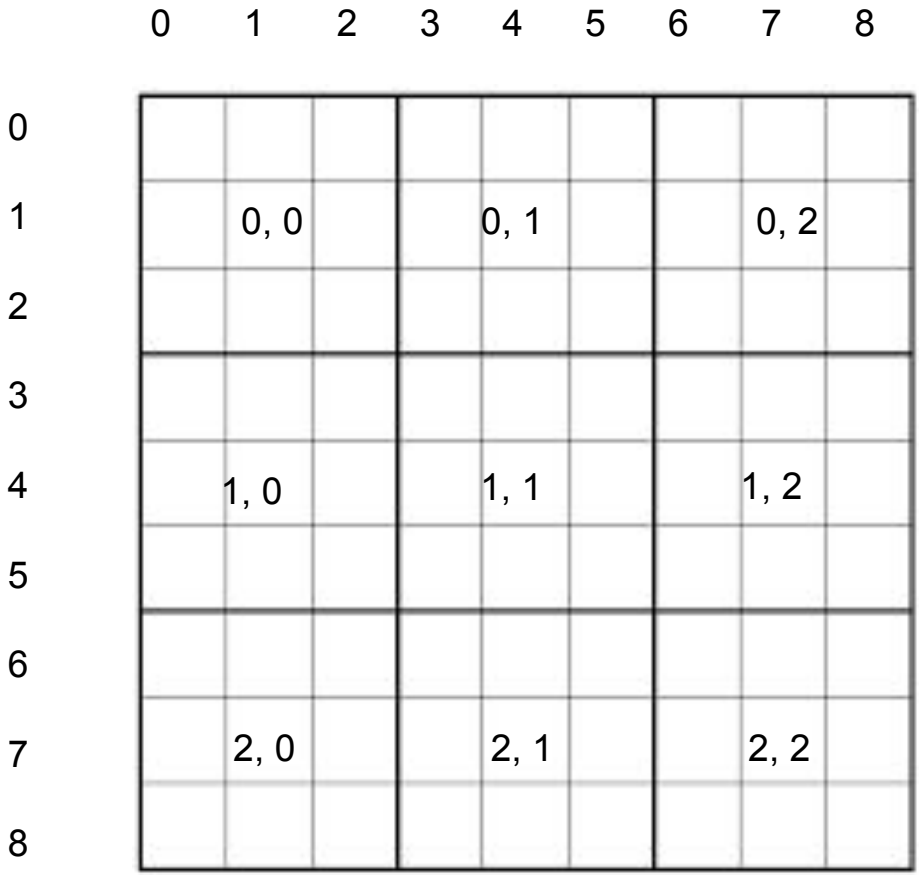
Fila /col	0	1	2	3
0	0	1	2	3
1	1	2	3	4
2	2	3	4	5
3	3	4	5	6

Sudoku

- Juego matemático que se presenta como un tablero de 9 x 9, compuesto por subtableros de 3 x 3 denominados regiones.
- El tablero presenta algunas celdas vacías y otras con números del 1 al 9.
- El objetivo es llenar con números las celdas vacías, tal que los números del 1 al 9 aparezcan exactamente una vez en cada fila, columna y región.

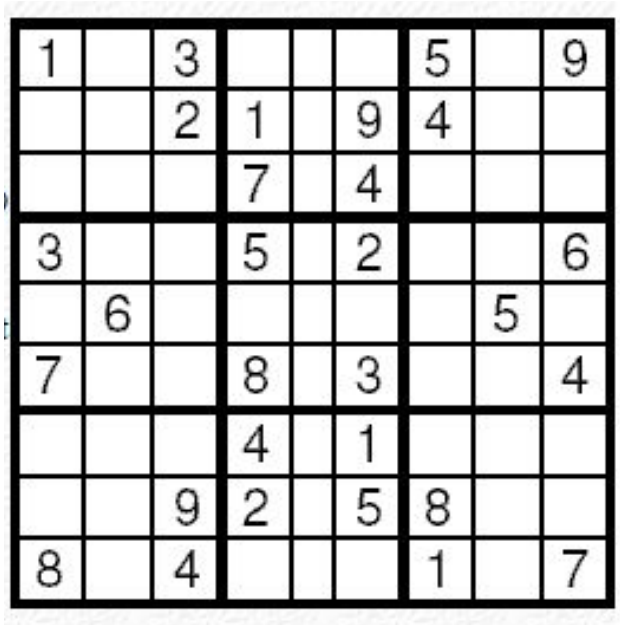
Sudoku

Filas, columnas
y regiones



Sudoku

Completar el siguiente sudoku



1		3				5		9
		2	1		9	4		
			7		4			
3			5		2			6
	6						5	
7			8		3			4
			4		1			
		9	2		5	8		
8		4				1		7

Referencias

- Jeff Erickson. Algorithms. Chapter 2 – Backtracking. https://jeffe.cs.illinois.edu/teaching/algorithms/book/02-backtracking.pdf

¡ Good luck and have fun !