**Backtracking**

1. **Introducción**

El backtracking es una técnica algorítmica utilizada para encontrar soluciones a problemas computacionales, especialmente en situaciones donde se requiere buscar todas las posibles soluciones dentro de un espacio de búsqueda. Se basa en la idea de construir sistemáticamente todas las posibles configuraciones, descartando aquellas que no cumplen ciertas condiciones o restricciones, y retrocediendo (backtracking) cuando se llega a un punto donde ya no es posible seguir adelante sin violar alguna restricción.

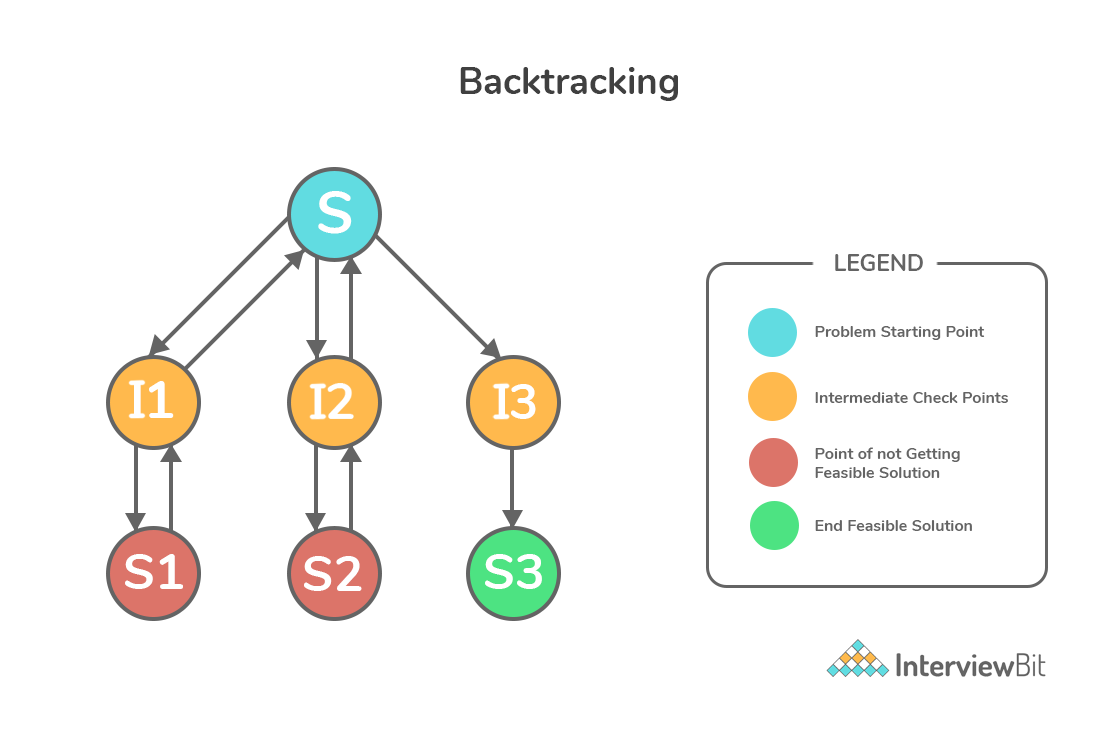
1. **Funcionamiento**

El algoritmo de retroceso normalmente funciona de la siguiente

manera:

* Elija una solución inicial y colóquela en la raíz del árbol de búsqueda.
* Examine la siguiente opción, seleccione una y agréguela a la solución actual.
* Si la solución actual es válida, pase a la siguiente opción y repita el segundo paso
* Si la solución actual no es válida, retroceda hasta el último punto de decisión y pruebe una opción diferente.

Continúe el proceso hasta que se encuentre una solución o se hayan examinado todas las soluciones posibles.



***Figura 1:*** *Funcionamiento del Backtracking*

1. **Ventajas**

* Permite encontrar todas las soluciones posibles a un problema.
* Es una técnica de resolución de problemas general que se puede aplicar a muchos tipos de problemas diferentes.
* El backtracking es una técnica fácil de entender y de implementar.
* El backtracking puede ser muy efectivo cuando se aplica correctamente.

1. **Desventajas**

* Puede ser ineficiente para problemas con un gran número de soluciones posibles.
* La complejidad temporal del backtracking puede ser muy alta en algunos casos, lo que hace que el algoritmo sea impráctico o incluso imposible de aplicar.
* Puede requerir una gran cantidad de memoria para almacenar el árbol de decisiones completo.
* No garantiza encontrar la solución óptima a un problema, a menos que se utilicen técnicas de poda y optimización adicionales.

1. **Análisis**

Complejidad temporal:

* La complejidad temporal del algoritmo de backtracking depende en gran medida del tamaño del espacio de búsqueda y de la cantidad de restricciones y condiciones que se aplican a cada paso.
* En el peor de los casos, donde se exploran todas las posibles combinaciones, la complejidad temporal puede ser exponencial, O(b^d), donde 'b' es el factor de ramificación (número de opciones disponibles en cada paso) y 'd' es la profundidad del árbol de decisiones (número de pasos).
* Sin embargo, con técnicas de poda y optimización adecuadas, la complejidad temporal puede reducirse significativamente.

Complejidad espacial:

* La complejidad espacial del algoritmo de backtracking también depende del tamaño del espacio de búsqueda y de la cantidad de memoria necesaria para almacenar las soluciones parciales y las restricciones.
* En el peor de los casos, la complejidad espacial puede ser lineal, O(d), donde 'd' es la profundidad del árbol de decisiones y representa la cantidad máxima de memoria necesaria para almacenar las soluciones parciales en un momento dado.

1. **Aplicaciones**

* Resolución de problemas de búsqueda exhaustiva: El backtracking se utiliza para encontrar todas las posibles soluciones a problemas donde se necesita explorar todo el espacio de búsqueda, como la generación de permutaciones, combinaciones, subconjuntos, etc.
* Problemas de optimización: En muchos problemas de optimización, el backtracking se puede utilizar para encontrar la mejor solución posible explorando exhaustivamente todas las posibles configuraciones y eligiendo la que optimice ciertas métricas o criterios.
* Problemas de satisfacción de restricciones (CSP): El backtracking se utiliza ampliamente en problemas de satisfacción de restricciones, donde se busca encontrar una solución que cumpla con un conjunto de restricciones o condiciones, como el problema de asignación de horarios, el problema de coloración de grafos, etc.
* Juegos y rompecabezas: El backtracking se utiliza en la resolución de juegos y rompecabezas, como el sudoku, el juego del 8, el ajedrez, etc., donde se busca encontrar una secuencia de movimientos que conduzca a una solución o victoria.
* Compiladores y análisis de código: En la compilación y el análisis de código, el backtracking se utiliza en la resolución de problemas de análisis sintáctico y semántico, como la generación de árboles de sintaxis abstracta, la resolución de conflictos de reducción y desplazamiento en el análisis LR, etc.

1. **Ejemplo – N Reinas**

Tablero.h

#pragma once

**class** **Tablero** {

**public:**

Tablero();

Tablero(**int**);

**void** **set\_TAM**(**int**);

**int** **get\_TAM**();

**void** **set\_piezas\_strings**(std::string\*\*\*);

std::string\*\*\* get\_piezas\_strings();

**void** **mostrar\_tablero**();

**bool** **esta\_ocupado**(**int**, **int**);

~Tablero();

**private:**

**int** TAM;

std::string\*\*\* piezas\_strings;

};

Tablero.cpp

#include <iostream>

#include "Tablero.h"

#include <iomanip>

Tablero::Tablero() {

TAM = **0**;

piezas\_strings = nullptr;

}

Tablero::Tablero(**int** \_TAM) {

//ENCERO MI MATRIZ DE PUNTEROS A strings CON DIRECCIONES NULAS

TAM = \_TAM;

piezas\_strings = (std::string\*\*\*)malloc(TAM \* **sizeof**(std::string\*\*));

**if** (piezas\_strings != nullptr) {

**for** (**int** i = **0**; i < TAM; i++) {

\*(piezas\_strings + i) = (std::string\*\*)malloc(TAM \* **sizeof**(std::string\*));

}

**for** (**int** i = **0**; i < TAM; i++) {

**if** (\*(piezas\_strings + i) != nullptr) {

**for** (**int** j = **0**; j < TAM; j++) {

\*(\*(piezas\_strings + i) + j) = **new** std::string("-");

}

}

}

}

}

**void** Tablero::set\_TAM(**int** \_TAM) {

TAM = \_TAM;

}

**int** Tablero::get\_TAM() {

**return** TAM;

}

**void** Tablero::set\_piezas\_strings(std::string\*\*\* \_piezas\_strings) {

piezas\_strings = \_piezas\_strings;

}

std::string\*\*\* Tablero::get\_piezas\_strings() {

**return** piezas\_strings;

}

**bool** Tablero::esta\_ocupado(**int** \_x, **int** \_y) {

**if** (\*(\*(piezas\_strings +\_x)+\_y) != nullptr) {

**return** true;

}

**return** false;

}

**void** Tablero::mostrar\_tablero() {

**if** (piezas\_strings != nullptr) {

**for** (**int** i = **0**; i < TAM; i++) {

**if** (\*(piezas\_strings + i) != nullptr) {

**for** (**int** j = **0**; j < TAM; j++) {

std::cout << std::setw(**10**) << \*(\*(\*(piezas\_strings + i) + j)) << std::setw(**10**);

}

}

printf("**\n**");

}

}

}

Tablero::~Tablero() {

}

Problema\_Reinas.h

#pragma once

#include "Tablero.h"

**bool** **validar\_posiciones**(**int**, **int**\*, **int**, **int**\*, **int**);

**void** **fun**(Tablero& tablero, **int** x, **int**\* \_x, **int**, **int**\*, **int**&);

**bool** **validar\_posiciones**(**int** x, **int**\* \_x, **int** y, **int**\* \_y, **int** TAM) {

**for** (**int** i = **0**; i < TAM; i++) {

**if** (y == \*(\_y + i) || std::abs((x - \*(\_x + i)) == std::abs(y - \*(\_y + i)))) {

**return** false;

}

}

**return** true;

}

**void** **fun**(Tablero& tablero, **int** x, **int**\* \_x, **int** y, **int**\* \_y, **int**& contador) {

**if** (x != tablero.get\_TAM() && y != tablero.get\_TAM()) {

**if** (validar\_posiciones(x, \_x, y, \_y, tablero.get\_TAM())) {

\*(\*(\*(tablero.get\_piezas\_strings() + x) + y)) = "Reina";

\*(\_x + x) = x;

\*(\_y + x) = y;

fun(tablero, x + **1**, \_x, **0**, \_y, contador);

**if** ((x+**1** != tablero.get\_TAM() && \*(\_y + tablero.get\_TAM() - **1**) == **10**)){

\*(\*(\*(tablero.get\_piezas\_strings() + x) + y)) = "-";

}

**else** {

contador++;

}

}

fun(tablero, x, \_x, y + **1**, \_y, contador);

**if** (\*(\_y + tablero.get\_TAM()-**1**) != **10**) {

tablero.mostrar\_tablero();

printf("SOL: %i, ------------------------------------------------------------------**\n**", contador);

}

\*(\*(\*(tablero.get\_piezas\_strings() + x) + y)) = "-";

\*(\_y + x) = **10**;

\*(\_x + x) = **20**;

}

}

N\_Reinas.cpp

#include <iostream>

#include <conio.h>

#include "windows.h"

#include "Menu.h"

#include "Tablero.h"

#include "validaciones.h"

#include "Problema\_Reinas.h"

**int** **main**()

{

**int** opcion = **1**;

**char** tecla;

**int** contador = **0**;

**int** TAM;

Tablero\* ajedrez = nullptr;

**int**\* \_x = nullptr;

**int**\* \_y = nullptr;

HANDLE consoleHandle = GetStdHandle(STD\_OUTPUT\_HANDLE);

CONSOLE\_CURSOR\_INFO cursorInfo;

GetConsoleCursorInfo(consoleHandle, &cursorInfo);

cursorInfo.bVisible = false;

SetConsoleCursorInfo(consoleHandle, &cursorInfo);

**do** {

mostrar(opcion);

tecla = \_getch();

**switch** (tecla) {

**case** **72**:

**if** (opcion > **1**) opcion--;

**break**;

**case** **80**:

**if** (opcion < **2**) opcion++;

**break**;

**case** **13**:

**if** (opcion == **1**) {

cursorInfo.bVisible = true;

SetConsoleCursorInfo(consoleHandle, &cursorInfo);

system("cls");

**do** {

TAM = ingresar\_enteros("Ingrese el Tamano: (4-9)");

printf("**\n**");

**if** (TAM < **4** || TAM > **9**) {

printf("Ese tamano no es valido. Intente de nuevo.**\n**");

}

**else** {

**break**;

}

} **while** (true);

ajedrez = **new** Tablero(TAM);

**if**(ajedrez->get\_piezas\_strings() != nullptr){

\_y = (**int**\*)malloc(TAM \* **sizeof**(**int**));

**if** (\_y != nullptr) {

**for** (**int** i = **0**; i < TAM; i++) {

\*(\_y + i) = **10**;

}

\_x = (**int**\*)malloc(TAM \* **sizeof**(**int**));

**if** (\_x != nullptr) {

**for** (**int** j = **0**; j < TAM; j++) {

\*(\_x + j) = **20**;

}

fun(\*ajedrez, **0**, \_x, **0**, \_y, contador);

}

}

}

**delete** ajedrez;

**delete** \_x;

**delete** \_y;

contador = **0**;

cursorInfo.bVisible = false;

SetConsoleCursorInfo(consoleHandle, &cursorInfo);

system("pause");

}

**else** **if** (opcion == **2**) {

system("cls");

std::cout << "Saliendo del programa." << std::endl;

exit(**0**);

}

**break**;

**default:**

**break**;

}

} **while** (true);

}

1. **Bibliografía**

* GeeksforGeeks. (n.d.). Backtracking algorithms. Retrieved March 10, 2023, from https://www.geeksforgeeks.org/backtracking-algorithms/
* Estructura de datos en C++, Luis Joyanes e Ignacio Zahonero.

‌