Licenciatura en Ciencia de Datos

Algoritmos II

RECURSIÓN

Backtracking

Introducción

- Existen problemas en las ciencias de la computación en los que se necesita realizar una búsqueda de soluciones **exhaustivas y finitas** que cumplan con sus **condiciones.**
- Algunas de las **características** de ellos:
  - **Espacio de soluciones finito:** podemos enumerar todas las soluciones posibles.
  - **Restricciones o condiciones explícitas**: restricciones claras y bien definidas, tales como validez de la solución, limitaciones de tiempo o recursos, entre otras.
  - **Solución satisfactoria:** en algunos casos se busca la **solución óptima,** mientras que en otros **cualquier solución válida.**

Estrategias: Fuerza bruta

-Enfoque que implica **probar cada opción disponible** de manera exhaustiva **hasta encontrar la solución** deseada.

-Válida si el espacio de soluciones es **reducido:** incluso podemos **generar todas las soluciones posibles** y **no hay** una **forma más eficiente** de resolver el problema.

-**Problema** cuando el espacio de soluciones es grande: es **costoso** o **inviable** generar **todas las soluciones posibles.**

Ejemplo de ejercicio de estrategia de Fuerza bruta

```
def es_primo(numero: int) -> bool:

    if numero <= 1:
        return False  # Los números menores o iguales a 1 no son primos

    for i in range(2, numero):
        if numero % i == 0:
            return False  # Si se encuentra un divisor, el número no es
primo

    return True  # Si ningún divisor fue encontrado, el número es primo
```

Estrategia: Backtracking

-Es una técnica de resolución de problemas que se basa en la **exploración** sistemática de **todas las posibles soluciones** para encontrar aquellas que **cumplen** con ciertas **restricciones o condiciones.**

-Def: es una estrategia de resolución no determinística de problemas (el resultado no puede ser predicho con certeza, incluso si se proporcionan las mismas entradas en diferentes ejecuciones; no existe un curso de acción que nos permita siempre llegar a la solución) que utiliza la **recursión.**

Ventajas del backtracking

- Útil cuando se enfrentan **problemas combinatorios** o de **búsqueda (espacio de soluciones grande)**

- **Eficiencia:** puede **descartar soluciones parciales** que sabemos que no construirán una solución. => permite **reducir el conjunto completo** de soluciones posibles.

Backtracking

-En cada **llamada recursiva** exploramos una **rama del árbol de posibles soluciones.**

-Si llegamos a un **punto en el que no se satisfacen ciertas condiciones**, **retrocedemos** (backtrack) y **probamos otra opción**.

-Resolución por medio de **prueba y error**.

Pasos fundamentales

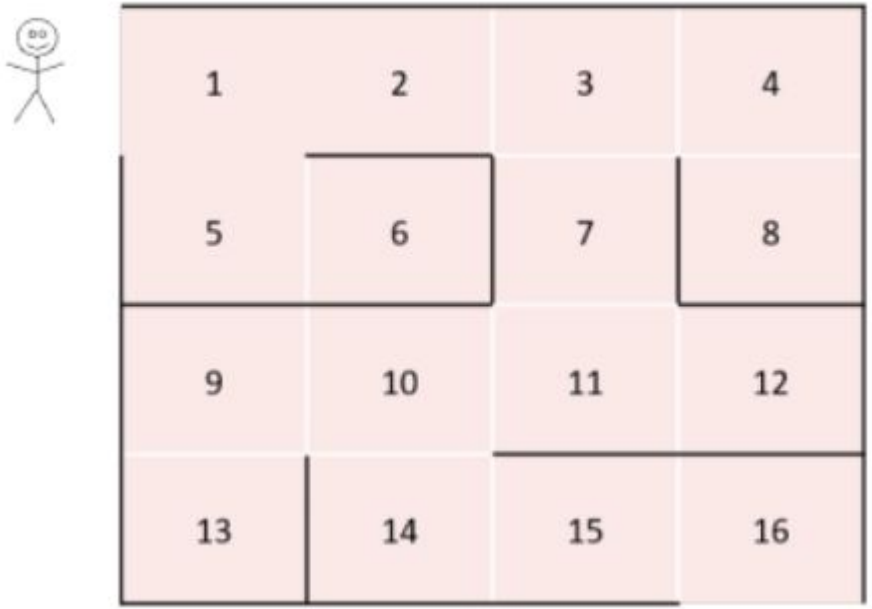
1) **Elección**: Se toma una decisión en el nivel actual de búsqueda, lo que determina cómo se ramificará el árbol de búsqueda.
2) **Exploración**: Se explora recursivamente las opciones disponibles a partir de la elección tomada.
3) **Validación:** Se verifica si la elección actual lleva a una solución válida. Si no es así, se realiza el retroceso (backtrack).
4) **Retroceso** (Backtrack): Si la elección actual no lleva a una solución válida, se revierte a un estado anterior y se realiza una nueva elección.

Ejemplo: laberinto

Estrategia:

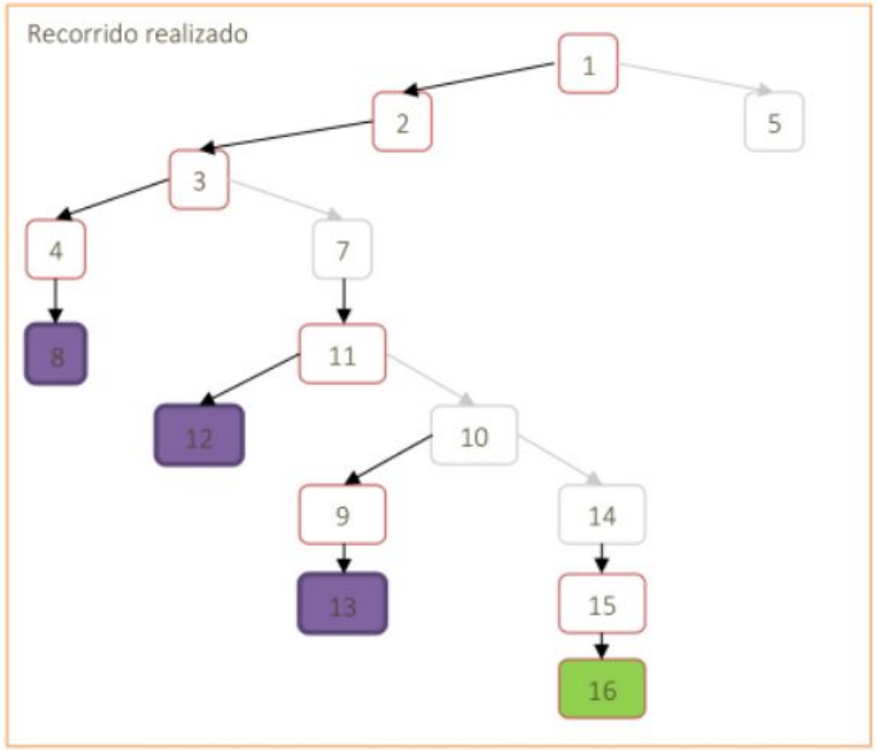
1) Avanzar en una dirección determinada (norte, sur, este, oeste).
2) En cada bifurcación, recorrer todos los caminos posibles.
3) Si llegué a un final sin salida o un lugar ya visitado, vuelvo hacia atrás a probar otro camino.

Ejemplo: laberinto con una sola solución



Ejemplo: laberinto con una sola solución

Árbol de soluciones:



Implementación:

-Requiere mantener en cada instancia de recursión una **copia de la solución parcial** para construir a partir de esta las **siguientes posibles soluciones**

-Sino no se puede **“volver atrás”**

Posible solución

```
def recorrer(camino_previo: list[Posicion]) -> (bool, list[Posicion]):
    posicion_actual = camino_previo[-1]
    if es_salida(posicion_actual):
        return True, camino_previo
    else:
        salida_encontrada = False
        solucion = camino_previo
        direcciones = ['N', 'S', 'O', 'E']
        while direcciones and not salida_encontrada:
            nueva_posicion = avanzar(posicion_actual, direcciones.pop())
            if hay_paso(nueva_posicion) and nueva_posicion not in camino_previo:
                camino_actual = camino_previo.copy()
                camino_actual.append(nueva_posicion)
                salida_encontrada, solucion = recorrer(camino_actual)

        return salida_encontrada, solucion
```

Aclaraciones de posible solución

-Recibe como **camino_previo** una lista sólo con la **posición inicial** del laberinto (la entrada).

-**posicion_actual** se determina con el último elemento de ese parámetro, ya que nos indica el camino realizado desde la entrada hasta la situación actual.

-Operaciones que **abstraemos**:

es_salida(): devuelve verdadero si la posición es una salida del laberinto.

avanzar(): dada una posición en el laberinto y una dirección, devuelve la nueva posición que surge de ir en esa dirección desde la posición original.

hay_paso(): valida si la posición actual es válida para avanzar (si no tiene un muro)

Aclaraciones sobre posible solución

-**Caso base**: estamos parados en la salida.

-**Caso recursivo:**

1) Asume que aún **no encontramos la salida**.
2) Planteamos la **estrategia de búsqueda** con las **4 direcciones posibles**.
3) **Para cada dirección** probamos avanzar si tenemos paso por esa dirección y validando que no estemos regresando por donde vinimos:
   a) si no podemos avanzar, se **descarta** esa dirección.
   b) Si avanzamos **copiamos el camino_previo como camino_actual**, le agregamos la **posición actual al final** y **continuamos** el recorrido utilizando **camino_actual.**

Ejercicio: permutaciones

Definir la función permutaciones, que dada una lista de enteros, retorne una lista de listas de enteros, donde cada lista es cada una de las posibles permutaciones de la lista original, usando backtracking.