Tecnología Digital V: Diseño de Algoritmos

Tabla de Contenidos

- Solución con Programación Dinámica - Bottom Up

Recursión

La recursión es un método de resolución de problemas que consiste en dividir el problema en subproblemas más pequeños, resolverlos y combinar las soluciones para obtener la solución del problema original.

Para esto se necesita un caso base, que es un caso simple que se puede resolver directamente, y un caso recursivo, que es un caso más complejo que se puede resolver dividiéndolo en subproblemas más pequeños.

Ejemplo Recursivo: Factorial

El factorial de un número n es el producto de todos los números enteros positivos menores o iguales a n.

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

Lo podemos definir recursivamente como:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{si } n > 0 \end{cases}$$

Donde el caso base es $n = 0$ y el caso recursivo es $n > 0$.

Se puede implementar en Python como:

```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Ejemplo Recursivo: Fibonacci

Un algoritmo puede tener más de un caso base y más de un caso recursivo.

La sucesión de Fibonacci es una sucesión infinita de números naturales que comienza con 0 y 1, y cada término es la suma de los dos anteriores.

$$f(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ f(n-1) + f(n-2) & \text{si } n > 1 \end{cases}$$

Su implementación en Python es:

```python
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Complejidad Algorítmica

La complejidad de un algoritmo es una medida de la cantidad de recursos que necesita para resolver un problema.

Complejidad temporal

La complejidad temporal es una medida de la cantidad de tiempo que necesita un algoritmo para resolver un problema.

Se puede medir contando la cantidad de operaciones que realiza el algoritmo.

Por ejemplo, el factorial de un número n se puede calcular con el siguiente algoritmo:

```python
def factorial(n):
    resultado = 1
    for i in range(1, n + 1):
        resultado *= i
    return resultado
```

Este algoritmo realiza n multiplicaciones, por lo que su complejidad temporal es $O(n)$.

Para los algoritmos recursivos, la complejidad temporal se puede calcular con la siguiente fórmula:

$$T(n) = \begin{cases} c & \text{si } n = 0 \\ a + T(n-1) & \text{si } n > 0 \end{cases}$$

Para un algoritmo como el factorial recursivo, $a = 1$ y $c = 1$, por lo que su complejidad temporal es $O(n)$.

En cambio para fibonacci, $a = 2$ y $c = 1$, por lo que su complejidad temporal es $O(2^n)$.

En general los algortimos recursivos tienen complejidad temporal $O(a^n)$ salvo que se trate de un ciclo recursivo, en cuyo caso la complejidad temporal es $O(n)$.

Para ver mas sobre complejidad temporal, ver Complejidad Algorítimica en Resumen TD III

Complejidad espacial

La complejidad espacial es una medida de la cantidad de memoria que necesita un algoritmo para resolver un problema.

Se puede medir contando la cantidad de variables que utiliza el algoritmo.

Ejemplo Complejidad Espacial: Fibonacci En la implementación recursiva más simple, la función de Fibonacci se llama recursivamente con argumentos $n-1$ y $n-2$ para calcular los términos $n-1$ y $n-2$ de la secuencia, y luego suma estos valores para obtener el término n.

Para resolver la recursión, la función de Fibonacci necesita almacenar los valores de $n-1$ y $n-2$ en la pila de llamadas. Eventualmente se llegan a las hojas de la recursión, donde $n = 0$ o $n = 1$, y se puede calcular el valor de la función.

Como $fib(1)$ y $fib(0)$ son constantes e iguales a 1, la cantidad de hojas es igual a la suma de los resultados de todas las hojas, es decir $fib(n)$, por lo que la complejidad espacial de Fibonacci es $O(fib(n))$.

Esto trae un problema al calcularlo que se puede resolver usando programación dinámica.

Fuerza Bruta

La fuerza bruta es un método de resolución de problemas que consiste en probar todas las posibles soluciones y quedarse con la mejor.

Ejemplo Fuerza Bruta: Knapsack 0-1

El problema de la mochila 0-1 consiste en elegir un subconjunto de objetos de forma que la suma de sus pesos sea menor o igual a la capacidad de la mochila y la suma de sus valores sea máxima.

Se puede resolver con fuerza bruta probando todas las posibles combinaciones de objetos y quedándose con la mejor.

```
def knapsack_0_1(capacidad, pesos, valores, n):
    if n == 0 or capacidad == 0:
        return 0
    if pesos[n - 1] > capacidad:
        return knapsack_0_1(capacidad, pesos, valores, n - 1)
    else:
        return max(
            valores[n - 1] + knapsack_0_1(
                                capacidad - pesos[n - 1],
                                pesos,
                                valores,
                                n - 1
                            ),
            knapsack_0_1(capacidad, pesos, valores, n - 1)
        )
```

Backtracking

El backtracking es un método de resolución de problemas que consiste en probar todas las posibles soluciones y descartar las que no cumplen con ciertas condiciones.

A estas condiciones se las llama podas o *pruning*.

Podas

Existen distintos tipos de podas:

- Podas de factibilidad: descartan soluciones que no cumplen con ciertas condiciones.
- Podas de optimalidad: descartan soluciones que no son mejores que la mejor solución encontrada hasta el momento.

Ejemplo Backtracking: N-Queens - Podas de Factibilidad

El problema de las N-Reinas consiste en colocar N reinas en un tablero de ajedrez de forma que ninguna reina pueda atacar a otra.

Se puede resolver con backtracking probando todas las posibles combinaciones de reinas y descartando las que no cumplen con la condición de que ninguna reina pueda atacar a otra.

```
def n_queens(n, tablero, fila):
    if fila >= n:
        return True
    for columna in range(n):
        if es_posicion_valida(tablero, fila, columna):
            tablero[fila][columna] = 1
            if n_queens(n, tablero, fila + 1):
                return True
            tablero[fila][columna] = 0
    return False
```

Ejemplo Backtracking: Knapsack 0-1 - Podas de Optimalidad

En el caso de la mochila 0-1, se puede aplicar una poda de optimalidad para descartar soluciones que no son mejores que la mejor solución encontrada hasta el momento.

```
def knapsack_0_1(capacidad, pesos, valores, n, mejor_valor):
    if n == 0 or capacidad == 0:
        return 0
    if pesos[n - 1] > capacidad:
        return knapsack_0_1(capacidad, pesos, valores, n - 1, mejor_valor)
    else:
        return max(
            valores[n - 1] + knapsack_0_1(
                                capacidad - pesos[n - 1],
                                pesos,
                                valores,
                                n - 1,
                                mejor_valor),
            knapsack_0_1(capacidad, pesos, valores, n - 1, mejor_valor)
        )
```

Programación Dinámica

La programación dinámica es un método de resolución de problemas que se encarga de resolver cada subproblema una sola vez y guardar su resultado para no tener que volver a calcularlo.

Existen dos tipos de programación dinámica:

- **Top-down**: Se resuelven los subproblemas de forma recursiva y se guardan los resultados en una tabla. Tambien conocido como *memoización*.
- **Bottom-up**: Se resuelven los subproblemas de forma iterativa y se guardan los resultados en una tabla. Tambien conocido como *tabulación* o *Tableau Filling*.

Ejemplo Programación Dinámica: Fibonacci

La programación dinámica se puede aplicar a la función de Fibonacci de la siguiente manera:

Top-down Fibonacci Se crea un diccionario para guardar los resultados de los subproblemas.

```
def fibonacci(n, memo):
    if n in memo:
        return memo[n]
    elif n == 0:
        resultado = 1
    elif n == 1:
        resultado = 1
    else:
        resultado = fibonacci(n - 1, memo) + fibonacci(n - 2, memo)
    memo[n] = resultado
    return resultado
```

Bottom-up Fibonacci Se crea una lista para guardar los resultados de los subproblemas. Se plantea de forma tal que `memo[i]` dependa de algun valor `memo[j]` con $j < i$.

```python
def fibonacci(n):
    memo = [1, 1]
    for i in range(2, n + 1):
        memo.append(memo[i - 1] + memo[i - 2])
    return memo[n]
```

Otros ejemplos de programación dinámica

Grafos

Un grafo es un conjunto de nodos y aristas que los conectan.

$$G = (V, E)$$

Donde V es el conjunto de nodos y E es el conjunto de aristas.

Propiedades de los grafos

Grado de un nodo El grado de un nodo es la cantidad de aristas que inciden en él. Lo representamos como $d(v)$.

Como regla, la cantidad de aristas de un grafo es:

$$\sum_{v \in V} d(v) = 2|E|$$

Camino Un camino es una secuencia de nodos conectados por aristas. La longitud de un camino es la cantidad de aristas que tiene.

Por ejemplo, el camino $A \to B \to C \to D$ tiene longitud 3.

Ciclo Un ciclo es un camino que comienza y termina en el mismo nodo.

Un grafo con n nodos y m aristas tiene a lo sumo $n - m + 1$ ciclos.

Tipos de grafos

Los grafos pueden ser dirigidos o no dirigidos. En los grafos dirigidos las aristas tienen una dirección, mientras que en los no dirigidos no la tienen.

Los grafos pueden ser ponderados o no ponderados. En los grafos ponderados las aristas tienen un peso, mientras que en los no ponderados no lo tienen.

El peso de un camino es la suma de los pesos de las aristas que lo componen.

Por ejemplo, el peso del camino $A \to^2 B \to^3 C \to^1 D$ es 6.

Grafos conexos Un grafo es conexo si existe un camino entre todos los pares de nodos. Es decir, si se puede llegar desde cualquier nodo a cualquier otro nodo.

Un grafo no es conexo si no existe un camino entre todos los pares de nodos. Es decir, si no se puede llegar desde cualquier nodo a cualquier otro nodo.

Grafos bipartitos Un grafo es bipartito si sus nodos se pueden dividir en dos conjuntos disjuntos, de tal forma que todas las aristas conectan un nodo de un conjunto con un nodo del otro conjunto.

Grafos completos Un grafo es completo si todos los nodos están conectados entre sí.

La cantidad de aristas de un grafo completo es:

$$|E| = \frac{n(n-1)}{2}$$

Si el grafo es dirigido, la cantidad de aristas es $|E| = n(n-1)$.

Grafo complementario El grafo complementario de un grafo G es un grafo G' que tiene los mismos nodos que G, pero tiene una arista entre dos nodos si y solo si G no tiene una arista entre esos dos nodos.

Es decir $G = (V, E)$ y $G' = (V, E')$, donde E' es el conjunto de aristas que no están en E. Ademas:

$$G \cap G' = \emptyset G \cup G' = K_n$$

Donde K_n es un grafo completo con n nodos.

Implementación de grafos

Listas de adyacencia Se puede implementar un grafo como una lista de adyacencia, donde cada nodo tiene una lista de nodos adyacentes.

```
grafo = {
    'A': ['B', 'C'],
    'B': ['A', 'C', 'D'],
    'C': ['A', 'B', 'D', 'E'],
    'D': ['B', 'C', 'E', 'F'],
    'E': ['C', 'D'],
    'F': ['D']
}
```

Ademas si es un grafo ponderado, se puede guardar el peso de cada arista en la lista de adyacencia.

```
grafo = {
    'A': [('B', 1), ('C', 2)],
    'B': [('A', 1), ('C', 2), ('D', 3)],
    'C': [('A', 2), ('B', 2), ('D', 2), ('E', 3)],
    'D': [('B', 3), ('C', 2), ('E', 1), ('F', 3)],
    'E': [('C', 3), ('D', 1)],
    'F': [('D', 3)]
}
```

Matriz de adyacencia O también como una matriz de adyacencia, donde la posición i, j es 1 si existe una arista entre los nodos i y j.

```
grafo = [
    [0, 1, 1, 0, 0, 0],
    [1, 0, 1, 1, 0, 0],
    [1, 1, 0, 1, 1, 0],
    [0, 1, 1, 0, 1, 1],
    [0, 0, 1, 1, 0, 0],
    [0, 0, 0, 1, 0, 0]
]
```

Y si es un grafo ponderado, se puede guardar el peso de cada arista en la matriz de adyacencia.

```
grafo = [
    [0, 1, 2, 0, 0, 0],
    [1, 0, 2, 3, 0, 0],
    [2, 2, 0, 2, 3, 0],
    [0, 3, 2, 0, 1, 3],
    [0, 0, 3, 1, 0, 0],
    [0, 0, 0, 3, 0, 0]
]
```

Recorridos en grafos

Recorrido en profundidad (DFS) El recorrido en profundidad (DFS) es un algoritmo que recorre todos los nodos de un grafo. Para esto, el algoritmo visita un nodo y luego visita recursivamente todos los nodos adyacentes a este nodo. El algoritmo se detiene cuando no hay más nodos por visitar.

```
def dfs(grafo, nodo, visitados):
    if nodo not in visitados:
        visitados.append(nodo)
        for nodo_adyacente in grafo[nodo]:
            dfs(grafo, nodo_adyacente, visitados)
    return visitados
```

Recorrido a lo ancho (BFS) El recorrido a lo ancho (BFS) es un algoritmo que recorre todos los nodos de un grafo. Para esto, el algoritmo visita un nodo y luego visita todos los nodos adyacentes a este nodo. El algoritmo se detiene cuando no hay más nodos por visitar.

```
def bfs(grafo, nodo, visitados):
    cola = [nodo]
    while cola:
        nodo = cola.pop(0)
        if nodo not in visitados:
            visitados.append(nodo)
            for nodo_adyacente in grafo[nodo]:
                cola.append(nodo_adyacente)
    return visitados
```

Algoritmo de Dijkstra

El algoritmo de Dijkstra es un algoritmo que encuentra el camino más corto entre dos nodos de un grafo. Para esto, el algoritmo calcula la distancia más corta desde un nodo inicial a todos los demás nodos del grafo.

Implementación El algoritmo de Dijkstra se puede implementar utilizando una cola de prioridad.

1. Inicializamos las distancias d de todos los vértices al infinito excepto por el vértice de origen s, que tiene distancia 0:

$$d(v) = \begin{cases} 0 & \text{si } v = s \\ \infty & \text{en otro caso} \end{cases}$$

2. Creamos un conjunto Q con todos los vértices del grafo. En cada iteración del algoritmo, extraeremos el vértice u de Q con la distancia más pequeña.

3. Para cada vecino v del vértice u que aún se encuentre en Q, verificamos si la distancia desde s hasta v pasando por u es más pequeña que la distancia actualmente registrada en $d(v)$. Si es así, actualizamos la distancia $d(v)$ y registramos a u como el vértice predecesor de v en el camino más corto desde s a v:

$$\begin{aligned} \text{si } d(v) &> d(u) + w(u,v) \\ d(v) &\leftarrow d(u) + w(u,v) \\ \text{predecesor}(v) &\leftarrow u \end{aligned}$$

Donde $w(u,v)$ es el peso de la arista que conecta a u y v.

A este proceso se le llama relajación de aristas.

En términos generales, la relajación de una arista consiste en actualizar la información de un vértice v (por ejemplo, su distancia más corta desde el origen) en función de la información de otro vértice u (por ejemplo, la distancia más corta conocida desde el origen hasta u) y la arista (u,v) que los conecta.

4. Repetimos el paso 2 y 3 hasta que el conjunto Q esté vacío.

Una vez finalizado el algoritmo, las distancias $d(v)$ para cada vértice v son los pesos de los caminos más cortos desde s hasta v, y los predecesores registrados nos permiten recuperar dichos caminos.

Es importante notar que el algoritmo de Dijkstra solo funciona correctamente si los pesos de las aristas son no negativos. Si hay aristas con peso negativo en el grafo, el algoritmo puede fallar y se debe utilizar otro enfoque, como el algoritmo de Bellman-Ford.

```python
def dijkstra(grafo, nodo_inicial):
    # Inicializamos las distancias de todos los nodos al infinito
    distancias = {}
    for nodo in grafo:
        distancias[nodo] = float('inf')

    # La distancia del nodo inicial a sí mismo es 0
    distancias[nodo_inicial] = 0

    # Inicializamos la cola con el nodo inicial
    cola = [nodo_inicial]

    # Mientras la cola no esté vacía
    while cola:
        # Extraemos el nodo con la distancia más pequeña
        nodo = cola.pop(0)

        # Para cada nodo adyacente al nodo extraído
        for nodo_adyacente, peso in grafo[nodo]:
            # Calculamos la distancia desde el nodo hasta el nodo adyacente
            distancia = distancias[nodo] + peso

            # Si la distancia es menor a la distancia actualmente registrada
            if distancia < distancias[nodo_adyacente]:
                # Actualizamos la distancia
                distancias[nodo_adyacente] = distancia

                # Registramos al nodo como el predecesor del nodo adyacente
                cola.append(nodo_adyacente)
    return distancias
```

Complejidad de Dijkstra

Arboles

Un árbol es un grafo no dirigido, conexo y acíclico. Un árbol con n nodos tiene $n-1$ aristas.

Propiedades de los árboles

- Un árbol con n nodos tiene $n-1$ aristas.
- Un árbol tiene al menos dos nodos de grado 1.
- Al agregar una arista a un árbol, se forma un ciclo.

Caminos en árboles Como un árbol es un grafo conexo, para cada par de nodos u y v existe un camino entre ellos. Además, como un árbol es un grafo acíclico, este camino es único.

Para encontrar un camino entre dos nodos u y v en un árbol, podemos utilizar el algoritmo de búsqueda en profundidad (DFS) o el algoritmo de búsqueda en amplitud (BFS).

Si queremos encontrar el camino máximo posible, podemos utilizar el algoritmo de búsqueda en profundidad (DFS) o el algoritmo de búsqueda en amplitud (BFS) desde un nodo cualquiera u para encontrar el nodo más lejano v. Luego, podemos utilizar denuevo DFS o BFS desde v para encontrar el nodo más lejano w. El camino entre v y w es el camino máximo posible en el árbol.

Árbol binario Un árbol binario es un árbol en el que cada nodo tiene a lo sumo dos hijos.

Árbol binario completo Un árbol binario completo es un árbol binario en el que todos los niveles están completamente llenos, excepto posiblemente el último nivel, que se llena de izquierda a derecha.

Árbol binario de búsqueda Un árbol binario de búsqueda es un árbol binario en el que para cada nodo, todos los nodos en el subárbol izquierdo tienen un valor menor que el valor del nodo y todos los nodos en el subárbol derecho tienen un valor mayor que el valor del nodo.

Árbol AVL Un árbol AVL es un árbol binario de búsqueda en el que la diferencia de altura entre los subárboles izquierdo y derecho de cada nodo es a lo sumo 1.

Más algoritmos

Algoritmo A*

El algoritmo A* es un algoritmo de búsqueda de caminos en grafos que se utiliza para encontrar el camino más corto entre dos nodos. A diferencia del algoritmo de Dijkstra, el algoritmo A* utiliza una heurística para estimar la distancia desde un nodo hasta el nodo destino.

El algoritmo A* utiliza una función $f(n)$ para estimar la distancia desde el nodo inicial hasta el nodo destino pasando por el nodo n. Esta función se define como:

$$f(n) = g(n) + h(n)$$

Donde $g(n)$ es la distancia desde el nodo inicial hasta el nodo n y $h(n)$ es la distancia estimada desde el nodo n hasta el nodo destino.

Para estimar la distancia desde el nodo n hasta el nodo destino, se utiliza una heurística. Una heurística es una función que estima la distancia desde el nodo n hasta el nodo destino. Una heurística común es la distancia euclidiana entre los nodos n y destino. También se puede utilizar la distancia de Manhattan, que es la suma de las distancias horizontales y verticales entre los nodos n y destino.

El algoritmo A* utiliza una cola de prioridad para almacenar los nodos que se van a visitar. En cada iteración, el algoritmo extrae el nodo con la menor distancia estimada desde el nodo inicial hasta el nodo destino. Luego, el algoritmo visita todos los nodos adyacentes al nodo extraído y actualiza sus distancias estimadas si es necesario.

Complejidad de A* Checkear, Lo escribió GH Copilot

La complejidad del algoritmo A* depende de la heurística utilizada. Si la heurística es una función constante, el algoritmo A* es equivalente al algoritmo de Dijkstra. Si la heurística es una función que sobreestima la distancia desde el nodo n hasta el nodo destino, el algoritmo A* es equivalente al algoritmo de Dijkstra con una cola de prioridad.

Si la heurística es una función que subestima la distancia desde el nodo n hasta el nodo destino, el algoritmo A* es equivalente al algoritmo de Dijkstra con una cola de prioridad y una cota superior para la distancia desde el nodo inicial hasta el nodo destino.

Ejercicio Subset Sum

Dado un conjunto de números S y un número k, determinar si existe un subconjunto de S cuya suma sea k.

```python
def subset_sum(S, k):
    # Si k es 0, existe un subconjunto cuya suma es k
    if k == 0:
        return True

    # Si S es vacío y k no es 0, no existe un subconjunto cuya suma es k
    if not S:
        return False

    # Poda por optimalidad
    # Si el último elemento de S es mayor que k no lo podemos utilizar
    if S[-1] > k:
        return subset_sum(S[:-1], k)

    # Si el último elemento de S es menor o igual que k podemos o no utilizarlo
    return subset_sum(S[:-1], k) or subset_sum(S[:-1], k - S[-1])
```

Solución Recursiva

Solución con Programación Dinámica - Bottom Up Completar