1. Algoritmos Voraces

Los algoritmos ávidos o voraces (*greedy algorithms*) son algoritmos que toman decisiones de corto alcance, basadas en información inmediatamente disponible, sin importar consecuencias futuras. En cada paso toman la mejor decisión posible en ese momento hasta conseguir una solución para el problema.

Es importante destacar que los algoritmos voraces no siempre encuentran una solución óptima. Sin embargo, muchas veces esta pérdida de precisión es aceptable, siempre y cuando el algoritmo encuentre soluciones *lo suficientemente buenas*.

1.1. Forma general de resolución

Para resolver un problema con un enfoque codicioso, solo debemos identificar:

- Un función `es_solucion` que decida si la solución es válida.
- Una función `elegir_candidato` que nos indique cómo elegir un candidato de forma adecuada.
- Una función `es_factible` que nos indique si la solución propuesta tiene sentido.

1.2. Ejemplos

1.2.1. Red de ciudades

Retomemos un ejemplo ya visto. Consideremos una red de ciudades, y una empresa constructora codiciosa. La empresa constructora es contratada para trazar vías férreas entre las ciudades, de modo que cualquier ciudad puede ser alcanzada desde cualquier otra. El contrato, sin embargo, no especifica ningún criterio, tal como la necesidad de rutas directas, o un máximo número de ciudades intermedias entre dos ciudades. Por lo tanto, la constructora codiciosa, está interesada en cumplir con el objetivo de la forma más económica, eligiendo las combinación de rutas más barata. Se asume que no todas las ciudades se pueden conectar con segmentos directos por vías, debido a características físicas del terreno (por ejemplo, la existencia de un río). En la Figura 1 se presenta el grafo que muestra este problema. Los vértices representan a las ciudades, y las aristas indican las rutas que pueden trazarse entre estas, con sus respectivos costos.

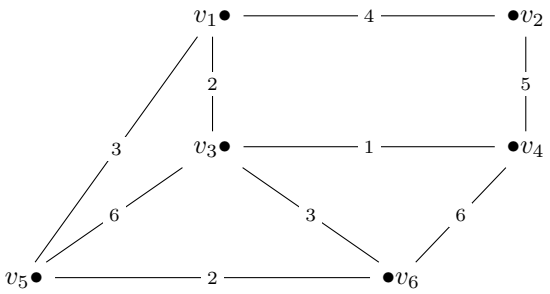

Figura 1: Ciudades y posibles vías que pueden trazarse entre ellas.

Notemos que lo que la empresa constructora necesita es encontrar el árbol de expansión mínimo. El algoritmo de Prim visto en la unidad anterior es un ejemplo de algoritmo avaro: en cada paso, el algoritmo agrega la arista de menor peso que cumpla con ciertas propiedades (un vértice de la arista debe estar dentro de los nodos agregados, mientras que el otro no). Es decir, en cada paso, toma la mejor decisión posible hasta el momento, de modo de construir la solución final que, en este caso, es la óptima.

En la Figura 2 se presenta un posible árbol de expansión mínimo (¡no es único!), que se obtiene al aplicar el algoritmo de Prim.

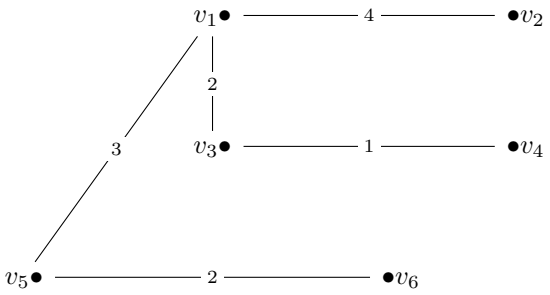

Figura 2: Red de vías de mínimo costo.

Retomando las funciones mencionadas en la sección 1.1, para pensar una implementación del algoritmo de Prim, podríamos definir:

- Un función `es_solucion` que determine si un conjunto de aristas constituye una red de vía férreas que conecta a todas las ciudades.
- Una función `elegir_candidato` que devuelva la arista de peso mínimo hasta el momento.
- Una función `es_factible` que indique si la solución parcial no contiene ciclos.

Así, la implementación del algoritmo de Prim, debería ir agregando aristas (elegidas mediante la llamada a `elegir_candidato`), chequeando que las soluciones parciales no contengan ciclos (mediante la llamada a `es_factible`), hasta que se obtenga un conjunto de aristas que conecta a todas las ciudades (que puede ser corroborado mediante la función `es_solucion`).

1.2.2. Monedas

En el país Albariocoque la moneda oficial es el fiji. Existen billetes de 100 fijis, y monedas de 4 fijis, 3 fijis, y 1 fiji. Supongamos que un cliente gasta 94 fijis, paga con 100 fijis y se debe determinar la mejor manera de dar el vuelto, cuando se quiere minimizar la cantidad de monedas que se entregan.

Algunas formas posibles de dar el vuelto son:

- seis monedas de un fiji.
- tres monedas de un fiji y una moneda de tres fijis.
- dos monedas de tres fijis
- una moneda de cuatro fijis y dos monedas de un fiji.

Este proceso, de buscar todas las posibilidades y ver cuál conviene utilizar, es una búsqueda exhaustiva. Podemos ver que la solución óptima es elegir entregar dos monedas de 3 fijis.

Podríamos intentar definir un algoritmo voraz para resolver este problema. Una idea simple sería siempre entregar el billete de mayor denominación posible, sin sobrepasar el valor que debe devolverse. Si luego de elegir esta billete se sigue debiendo dinero, se repite el proceso con el monto actualizado.

Al aplicar dicho algoritmo avaro en la devolución de dinero se entregaría una moneda de 4 fijis y dos monedas de 1 fiji, dando una solución subóptima. Sin embargo, este algoritmo voraz es intuitivo, y sencillo tanto de describir como de implementar, por lo que en algunas circunstancia podría considerarse una buena aproximación a la solución óptima.

1.3. Conclusión

Ventajas

- Este tipo de algoritmos es útil en problemas de optimización, es decir, cuando hay una variable que maximizar o minimizar.

Desventajas

- Hay que tener cuidado. No siempre la solución encontrada será óptima.
- Si necesitamos una solución óptima, se puede intentar demostrar matemáticamente que el algoritmo voraz es correcto, o se puede intentar con una técnica más avanzada, como la programación dinámica.

2. Búsqueda exhaustiva

La búsqueda exhaustiva, a veces llamada búsqueda por fuerza bruta es una técnica trivial pero a menudo usada, que consiste en enumerar sistemáticamente todos los posibles candidatos para la solución de un problema, con el fin de chequear si dicho candidato satisface la solución al mismo.

2.1. Forma general de resolución

Para obtener un algoritmo de búsqueda exhaustiva necesitamos:

- Una función `es_solución` que verifique si un intento es correcto.
- Un generador `candidatos` que devuelva uno a uno los candidatos con los que hay que probar.

2.2. Ejemplo

Supongamos que necesitamos encontrar un divisor (distinto de 1) de un número natural $n > 1$. Una posible forma de lograrlo consiste en enumerar todos los enteros desde 2 hasta $n - 1$, chequeando en cada paso si divide a n sin dejar resto.

2.2.1. Implementación

Para implementar el algoritmo mencionado anteriormente, debemos definir las funciones dadas en la sección 2.1.

La función `es_solución` debe verificar si un candidato x es correcto. En nuestro caso, deberá determinar si x divide a n sin dejar resto.

```
def es_solucion(n, x):
    return n % x == 0
```

Por otro lado, debemos definir la función `candidatos` que retorna los candidatos, a partir de los cuales haremos la búsqueda exhaustiva de la solución. En este caso, los candidatos serán los números naturales entre 2 y n (inclusive).

```
def candidatos(n):
    return [i for i in range(2,n+1)]
```

Luego, podemos escribir la función `divisor` que, dado un número natural, devuelve un divisor del mismo. Notemos que el divisor que devuelve es el primero que encuentra y que, por el orden en que se dan los candidatos, es el mínimo.

```
def divisor(n):
    for x in candidatos(n):
        if es_solucion(n, x):
            return x
```

2.2.2. Implementación para obtener todas las soluciones

Supongamos que, en lugar de encontrar un único divisor de n distinto de 1, queremos encontrarlos todos. Notemos que, en este caso, las funciones `es_solución` y `candidatos` no deberían modificarse, ya que lo único que difiere es que ahora los candidatos que son solución deben acumularse (en lugar de retornar solo el primero).

```
def divisores(n):
    divisores = []
    for x in candidatos(n):
        if es_solucion(n, x):
            divisores.append(x)
    return divisores
```

2.2.3. El módulo `itertools`

En las implementaciones anteriores, los candidatos fueron generados como una lista por comprensión. Si bien esto es razonable para un ejemplo tan simple como el que vimos, para casos más complejos es preferible usar herramientas específicas que sean rápidas y eficientes en cuanto a memoria utilizada. Para esto, Python trae incluido un módulo llamado `itertools`, el cual estandariza una serie de herramientas que se pueden utilizar para escribir generadores de forma más sencilla.

Tarea. Investigar la documentación oficial de este módulo.

Ejercicio. Escribir un algoritmo de Búsqueda Exhaustiva que ordene una lista de números.

2.3. Encontrando la mejor solución

Muchos problemas admiten varias soluciones. Sin embargo, muchas veces cada uno de estas posibles soluciones tiene un costo asociado - no da lo mismo elegir cualquiera.

En dichos casos, se puede adaptar la estrategia de búsqueda exhaustiva para encontrar la **mejor** solución posible.

Para esto necesitamos una nueva función `calcular_costo` que dado un candidato a solución, nos devuelva un número que cuantifique cuán buena es nuestra solución de alguna manera.

De este modo, cuando iteramos por los candidatos a solución, lo hacemos de forma tal que, si tenemos solución, calculamos el costo de la solución y comparamos con la mejor solución encontrada al momento.

2.4. Conclusión

Ventajas

- Es una técnica fácil de implementar y de leer, por lo que se suele utilizar cuando se quiere tener implementaciones fáciles de probar y depurar.
- Hay problemas que admiten muchas soluciones. Cuando se usa una técnica de búsqueda exhaustiva, es fácil adaptar el programa para encontrar *todas* las soluciones del problema, o una cantidad K de soluciones, o buscar soluciones hasta haber consumido cierta cantidad de recursos (tiempo, memoria, CPU, etc.).

Desventajas

- La cantidad de posibles soluciones a explorar por lo general crece exponencialmente a medida que crece el tamaño del problema.
- Depende mucho del poder de cómputo de la máquina para resolver el problema.