Ayudantía 6 - Algoritmos y Complejidad
Algoritmos Voraces

Sassy Complexes

1. Introducción

Un algoritmo voraz es un algoritmo que sigue la estrategia (heurística, como quiera llamarle) de elegir *la mejor opción local* y **nunca reconsiderar las elecciones previas**. En términos mortales, nos casamos con la mejor opción vecina o lo mejor que este a nuestro alcance y *le echamos pa'elante*, con la inocente esperanza de llegar al óptimo global. Intuitivamente se habrá dado cuenta de que no siempre se encontrará el mejor resultado usando esta técnica, pero la gracia de greedy es llegar a soluciones *aproximadas* en un **tiempo razonable**, lo veremos en ejemplos mas adelante.

1.1. Notese

La estrategia voraz para resolver el Traveling Salesman Problem (TSP) es : "En cada iteración, visite una ciudad no visitada vecina a la ciudad actual (NN algorithm)". Esta heurística no encuentra la mejor solución pero termina en una cantidad de iteraciones razonable. Encontrar un verdadero óptimo aquí tardaría una cantidad de pasos cuestionable.

2. Elementos

En general un algoritmo voraz esta compuesto de 5 cosas:

⋆ **Conjunto de Candidatos** de estos saldra la solución.

⋆ **Función de Selección** que elije el mejor candidato para añadir a la solución (solo considera los que no se han considerado).

⋆ **Función de Factibilidad** determina si el candidato no destruye la solución.

⋆ **Función de Solución** verifica si los candidatos seleccionados resuelven el problema, no necesariamente óptima.

⋆ **Función objetivo** Otorga valor a la solución, esto es lo que queremos optimizar.

3. Ejemplo ilustrativo

Problema Encontrar la ruta con suma máxima desde la raíz a las hojas.

Algoritmo Voraz: Elije el nodo vecino que tenga el número mas grande.

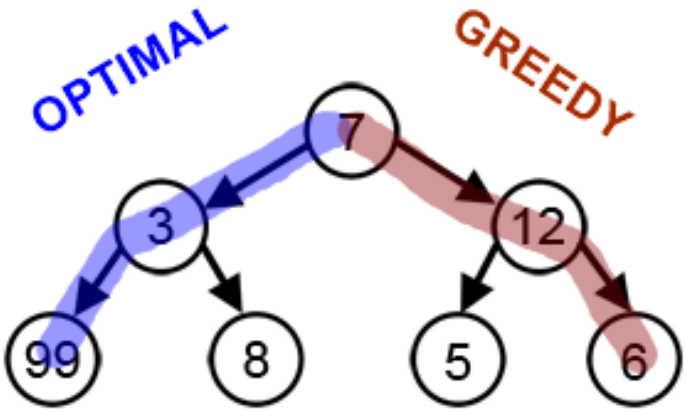

Figura 1: Con el objetivo de alcanzar la suma mas alta, el algoritmo greedy elegirá la que parece ser la solucion óptima inmediata, entonces elegirá 12 en vez de 3 en el segundo paso, y fallara en encontrar la mejor solución que contiene 99. Vea Wikipedia.

Conjunto de Candidatos $C = \{7, 3, 12, 99, 8, 5, 6\}$

Función de Selección "Elijo el mayor"

Función de Factibilidad "Es factible si sigue el camino"

Función de Solución "Si llegamos a las hojas, es solución"

Función objetivo $\max : \sum C_i$

4. Demostrar Optimo

Cuando estas tratando de probar un algoritmo, generalmente estas tratando de probar dos cosas. Primero, necesitas demostrar que el algoritmo produce una solución factible, o sea, que el algoritmo entrega una solución aplicable al mundo real del problema. Después debes probar que el algoritmo produce la solución óptima, esto es, maximizar o minimizar una cantidad.

Generalmente es mas fácil probar factibilidad que optimalidad, pero escribir una prueba formal de validez implica demostrar **Factibilidad** y **Optimalidad**. Estas pruebas funcionan por inducción, mostrando que en cada paso, la *Greedy Choice* no viola las restricciones del mundo del problema y el algoritmo termina con una solución correcta.

4.1. Aquí en algoco, hacemos cumplir estas 3 propiedades
(Que son inducción por debajo)

4.1.1. Greedy choice

Para toda instancia P del problema, hay una solución óptima global $\Pi^\star$ que incluye el primer elemento elegido $\hat{p}$.

Podemos demostrar por reducción al absurdo: Suponiendo que de todas las soluciones óptimas posibles ninguna incluye $\hat{p}$ y mostrar, por ejemplo, que se podría obtener una solución mejor agregando $\hat{p}$ contradiciendo la optimalidad de las originales.
$\star$ A veces es posible demostrar que todas las soluciones óptimas deben contener a $\hat{p}$, en este caso se estaría demostrando más de lo necesario, pero es suficiente.

4.1.2. Inductive Substructure

Dada la elección $\hat{p}$, recortamos el problema P a un problema menor P$'$. Entonces si Π' es solución factible de P$'$, $\{\hat{p}\} \cup \Pi'$ es solución factible de P. O sea: P$'$ no tiene *"restricciones externas"*

$\star$ Los gringos suelen saltarse esta parte, por que la hacen en Greedy Choice. Ahí aseguran que la elección voraz deje P$'$ sin depender de elecciones voraces de pasos anteriores.

Lo importante aquí es elegir P$'$ correctamente. No siempre basta con elegir P$' =$ P $- \{\hat{p}\}$ a veces hay que eliminar más cosas de manera que no queden *objetos* que sean afectados de alguna manera por la elección de voraz. De alguna forma asegure que $\hat{p}$ deje a P$'$ sin depender de $\hat{p}$s anteriores!!

4.1.3. Optimal Substructure

Un problema tiene subestructura óptima si **una** solución óptima $\Pi^\star$ puede ser construida por la unión de alguna solución óptima y sus subproblemas. Es mostrar que el algoritmo es óptimo a cada paso.

Demuestre que $\Pi = \{\hat{p}\} \cup \Pi'^\star$ es la solución óptima al problema P demostrando que $\Pi'^\star$ es la solución óptima al subproblema P$'$.

Generalmente podremos demostrar por reducción al absurdo, suponiendo que existe una solución mejor. Si se trata de un problema donde "mejor" está definido por la cardinalidad de la solución, habría que demostrar que no es posible una solución con menos (o más) de $|\Pi'^\star| + 1$ elementos para el problema P.

Nada nos impide apoyarnos en lo demostrado en *elección voraz* para lograr nuestro cometido.

5. Ejemplo Clásico - Minimum coin change problem

Tratando de dar el vuelto (el cambio), queremos entregar la menor cantidad de monedas posibles.

Plantee un algoritmo: Elija la moneda de mayor denominación que no supere la cantidad que se debe entregar n, continué así hasta haber entregado todo el vuelto.

Un poco mas formal: El algoritmo consiste en agregar la moneda k de mayor valoración tal que $k \leq n$, luego se sigue, resolviendo el problema para $n - k$. Cuando llegamos n sea igual a alguna moneda, agregamos dicha moneda y terminamos.

5.1. En Inglaterra

En la mayoría de los países, suelen ocupar denominaciones divisibles por 5 o 10. La libra esterlina inglesa ocupa el **set de denominaciones** $\{1,2,5,10,20,50\}$.
$\star$ Demuestre que dar $\$n$ usando el algoritmo propuesto con estas monedas entrega la menor cantidad de monedas.

5.1.1. Greedy choice property

Supongamos que existe una solución $\Pi_n^\star$ óptima que no elige la moneda de mayor valor m (tal que $m \leq n$), entonces, posee un subconjunto de monedas que suman m (por los tipos de monedas que hay), si reemplazamos ese subconjunto por la moneda m, tendríamos una solución mejor, pero $\Pi_n^\star$ era óptima, contradicción.

5.1.2. Inductive Substructure

Tras elegir la moneda m, resulta el subproblema P_{n-m} de representar $n-m$ con monedas, El conjunto de monedas $\Pi_{n-m} \cup \{m\}$, es solución factible para P_n ya que suman m y no hay restricciones externas al resolver P_{n-m} (la moneda elegida no afecta para nada cómo resolveremos P_{n-m}).

5.1.3. Optimal Substructure

Debemos demostrar que $\Pi_n = \Pi_{n-m}^\star \cup \{m\}$, es óptima, osea, no se puede resolver el problema P_n con menos monedas que $|\Pi_{n-m}^\star| + 1$.

Supongamos lo contrario, entonces es posible reemplazar un subconjunto de monedas de $\Pi_{n-m}^\star$ por otro subconjunto de menos o igual cantidad de monedas de manera que la suma de estas monedas aumente en m.

Si hacemos este cambio queda una solución para P_n con $|\Pi_{n-m}^\star|$ monedas, por lo visto en Greedy Choice, existe una solución igual de óptima que tiene una moneda m, si sacamos esa moneda, tendríamos una solución para P_{n-m} que requeriría $|\Pi_{n-m}^\star| - 1$ monedas y por lo tanto $\Pi_{n-m}^\star$ no sería óptima. contradicción.

5.2. En Otro Lugar

En el lejano pais de "*Horstlandia*", donde la maldad es infinita, utilizan el sistema "*VonBrand Coins*" con denominaciones $\{1,4,9,25\}$. Demuestre o Refute que el algoritmo propuesto es óptimo para esta denominación.

5.2.1. Greedy choice property

$\star$ Ya debería sospechar que para los *VonBrand Coins* el algoritmo falla.

Por contra-ejemplo (si, así de fácil, **basta que UN caso falle para que deje de ser optimo**). Para representar $n = 37$, el algoritmo elegiría: $[25,9,1,1,1]$ 5 monedas, siendo que el óptimo es 4 monedas: $[25,4,4,4]$, por lo tanto, este algoritmo no es óptimo por contraejemplo

Listing 1: Codigo con ejemplo de uso

```
def makechange(N, Coins, sol):
    if not Coins: return sol
    return makechange(N % Coins[0], Coins[1:], sol + [N//Coins[0]])

# Ordered Set
Sterlings = [50, 20, 10, 5, 2, 1]
VonBrandCoins = [25, 9, 4, 1]

makechange(37, Sterlings, [])
```

Listing 2: Solución alternativa

```
def makechange_alt(N, Coins):
    sol = [0] * len(Coins)

    for i in range(len(Coins)):
        sol[i] = N//Coins[i]
        N = N % Coins[i]

    return sol
```

6. Ejercicios:

6.1. Memory ALLOCation

Se tiene una cinta de memoria con algunos espacios quemados, va llegando una secuencia infinita de *malloc's* (que vienen en un orden inalterable, no se puede *allocar* m_n si no se ha *allocado* m_{n-1}).
Se pide diseñar un algoritmo *greedy* que permita *allocar* la mayor cantidad de posible de memoria de la secuencia. Demuestre o Refute si su algoritmo es óptimo.

respuesta

Un algoritmo greedy intentaría siempre colocar el siguiente bloque de memoria en el espacio blanco mas pequeño que lo pueda contener.

Modelamos el problema representando los espacios de memoria contiguos disponibles en la cinta como $\{b_i = d_i\}$ y la secuencia de solicitudes como $\{t_i = s_i\}$ donde d_i es la cantidad de memoria disponible en el bloque y s_i la memoria solicitada por el malloc.
Suponga el caso en el que se tienen los bloques de memoria $b_1 = 5 y b_2 = 7$, la secuencia de solicitudes parte con $t_1 = 3, t_2 = 3, t_3 = 5$ ...
La elección voraz de nuestro algoritmo elegirá b_1 para satisfacer la solicitud de t_1, ya satisfecha el problema queda con $b_1 = 2 y b_2 = 7$, para satisfacer a t_2 la elección voraz sera b_2, nos deja el problema en $b_1 = 2 y b_2 = 4$.
Quedando t_3 por satisfacer, ya no existe memoria suficiente, por lo tanto logramos satisfacer 2 solicitudes. Para $[t_1, t_2, t_3, ...]$ nuestra solución es $[b_1, b_2]$, logrando 2 mallocs, pero si hubiéramos

elegido $[b_2, b_2, b_1]$ habríamos satisfecho 3 solicitudes, con esto mostramos que la elección voraz falla en encontrar la solución óptima, puesto que la solución óptima contiene elecciones no voraces.

6.2. Haciendo anuncios

De alguna forma (no muy legal) conseguiste acceso al sistema de anuncios de tu escuela porque hay un anuncio muy importante que quieres hacer (cuando se hace un anuncio todas las clases en sesión van a oírlo). Así que quieres asegurarte de repetirlo tantas veces para que todas las clases lo oigan, pero como hacer un anuncio implica meterte en problemas de cierta forma quieres minimizar el número de veces que repites el anuncio.
Supón que tienes la lista de clases con sus horas de inicio y termino:

1) Describe un algoritmo que te permita encontrar el momento en cual hacer los anuncios tal que todas las clases lo oigan.

respuesta

La rutina que minimiza los anuncios es hacer el anuncio al final de la clase que termina mas pronto, luego sacar del problema todas las clases en las que se oyó el anuncio.

2) Demuestre que su algoritmo encuentra el uso mínimo de anuncios.

respuesta

Llamaremos P a un conjunto de clases, Π es solución y un subconjunto de P. Π siempre es viable, buscamos $|\Pi|$ mínimo. Llamaremos p a una clase.

> **Greedy Choice** Para todo P, hay una solución óptima que incluye la elección voraz $\hat{p}$.
> Sea $\hat{p}$ la primera clase elegida donde se hará el anuncio al final de ella y Π^* una solución óptima de P. Si $\hat{p} \in \Pi^*$ estamos listos.
> En caso contrario, suponga que no se elige el final de la tarea que termina mas temprano. Si se agrega una posición diferente de la $\hat{p}$, entonces $\hat{p} \in \Pi^*$, estamos listos.
> Si se elije el final de una tarea diferente a la de final mas temprano, $\hat{p}$ quedaría fuera y necesariamente se debe hacer un anuncio extra para asegurar $\hat{p}$ reciba un anuncio, la tarea de final mas temprano necesariamente va a estar en una solución y por ende, en la solución óptima.
>
> **Inductive Substructure** El elegir $\hat{p}$ nos deja un problema P' sin restricciones externas. El problema $P \setminus \{\hat{p}\}$ contiene clases en las que ya se oyó el anuncio, como no hay restricciones al numero de veces que el anuncio se puede repetir durante la misma clase, el problema P' resultante de eliminar $\hat{p}$ no tiene restricciones externas, y toda solución viable de P' puede combinarse con $\hat{p}$, de todas formas conviene eliminar todas las clases que se solapan con $\hat{p}$.
>
> **Optimal Substructure** Si P' queda después de elegir $\hat{p}$ y eliminar clases que se solapan, y Π' es óptima para P', entonces $\Pi' \cup \{\hat{p}\}$ es óptima para P.
> Por Estructura Inductiva $\Pi' \cup \{\hat{p}\}$ es viable para P, $|\Pi' \cup \{\hat{p}\}| = |\Pi'| + 1$. Sea $\Pi^\star$ una solución óptima para P que contiene $\hat{p}$ por elección voraz, entonces $\Pi^\star \setminus \hat{p}$ es solución óptima para P', si hubiese una de tamaño menor, daría una solución menor que $\Pi^\star$ para P, pero:

$$
\begin{aligned}
|\Pi'| &= |\Pi^\star \setminus \{\hat{p}\}| \\
&= |\Pi^\star| - 1
\end{aligned}
$$

O sea:

$$|\Pi' \cup \{\hat{p}\}| = |\Pi^\star|$$

Y como dijimos $\Pi' \cup \{\hat{p}\}$ es óptima.