



Instituto Politécnico Nacional
Escuela Superior de Cómputo



Análisis de Algoritmos, Sem: 2022-2, 3CV12, Práctica 3, Fecha: 30-03-2022

PRÁCTICA 5: ALGORITMOS GREEDY

López Cabagné Oscar Eduardo

olopezc1402@alumno.ipn.mx

Resumen: En esta practica se aplicará un algoritmo Greedy para resolver una problemática específica: Dado un arreglo C, donde C representará los días en que un granjero puede adquirir fertilizante, encontrar un conjunto S que contenga los días en que el granjero deberá desplazarse al pueblo antes de quedarse sin fertilizante.

Palabras Clave: C++, Análisis, Gráfica, Algoritmo, Greedy.

1 Introducción

Un Algoritmo Greedy (También llamado Algoritmo Voraz) es una estrategia que busca encontrar la solución óptima a un problema general, dividiéndolo en sub-problemas más pequeños, idénticos en naturaleza al problema original, donde la solución a cada uno de estos también deberá ser óptima.

Este tipo de algoritmos es utilizado para resolver problemas de optimización, pues toma decisiones de acuerdo a la información disponible al momento (sin información previa o posterior). Una vez que una opción ha sido tomada, no se volverá a cuestionar.

Es importante considerar que, aunque los algoritmos Greedy son sencillos y rápidos, no garantizan obtener siempre una solución óptima, por lo que siempre será necesario corroborar la respuesta obtenida o verificar que el problema en cuestión puede ser resuelto utilizando únicamente esta estrategia. Sin embargo, consiga o no una solución, el coste invertido (tanto en tiempo como en procesamiento) es muy pequeño, pues este algoritmo solo generará una única solución de entre todas las posibles.

La técnica más común para demostrar la optimalidad de la solución obtenida es Demostración por Inducción.

Un algoritmo Greedy tiene múltiples aplicaciones posibles, por ejemplo:

- Almacenamiento
- Devolución de cambio
- Rutas más cortas
- Compresión de datos
- Árboles de decisión

Como se mencionó antes, un algoritmo Greedy podría ser capaz de no encontrar una solución óptima. Esto podría deberse a que hay mejores soluciones si se consideran alternativas que Greedy desecha desde un principio. Por esto, a pesar de resolver un sub-problema de manera óptima, podría no llevar a la solución óptima general.

A pesar de esto, a veces aún así resultaría útil, si se busca obtener una solución rápida aunque no eficiente.

En esta práctica se buscará implementar un algoritmo Greedy para resolver un problema. De esta manera confirmaremos su efectividad, comprobaremos su complejidad temporal y veremos cómo se desempeña en situaciones diferentes.

Para realizar estas pruebas, se pondrá a prueba la implementación utilizando arreglos de diferentes tamaños generados al momento con valores aleatorios, ordenados de menor a mayor (donde todo arreglo deberá comenzar siempre con un valor 0, que representará el primer día en que el granjero compró fertilizante). Con los datos obtenidos, se realizará una gráfica que nos permitirá visualizar mejor su comportamiento.

2 Conceptos Básicos

Dado un conjunto C , donde C representará los valores de entrada, Greedy devolverá un conjunto S , donde S represente los elementos de la solución óptima encontrada. Además, se debe cumplir que: $S \subset C$.

En cada paso, Greedy elige al mejor candidato $x \in C$, donde x se vuelve el elemento prometedor y es eliminado de C . Luego, revisará si al incluirlo en el conjunto de soluciones S se llega a una solución óptima.

Si Greedy decide que este elemento no da una solución óptima, lo desecha. Si Greedy decide que este elemento proporciona una solución óptima, es añadido a S .

Este proceso se llevará a cabo para cada elemento candidato, iterando el bucle una y otra vez hasta llegar a la solución general o si el conjunto de candidatos C se vacía, en cuyo caso, Greedy no habría sido capaz de encontrar una solución.

A continuación se enlistan las condiciones que deben cumplir los problemas que son candidatos a ser resueltos usando un algoritmo Greedy:

1. Debe tratarse de un problema de optimización. Es decir, debe existir una función a minimizar o maximizar, según sea el caso.
2. la función debe contar con su respectivo dominio.
3. Existe un conjunto de reglas que definen condiciones al dominio de la función.

4. La solución obtenida debe poder ser representada como una secuencia de decisiones.

Pseudo-código de una función Greedy:

```
Greedy (conjunto de candidatos C): solución S
S =  $\emptyset$ 
while (S no sea una solución y  $C \neq \emptyset$ ) {
  x = selección(C)
  C = C - {x}
  if (S  $\cup$  {x} es factible)
    S = S  $\cup$  {x}
}
if (S es una solución)
  return S;
else
  return "No se encontró una solución";
```

Figure 1: Pseudo-código Greedy

Problema 1. El Granjero: Para poder aplicar Greedy, así como cualquier otra estrategia de programación, primero deberemos comprender bien el problema.

Para comenzar, deberemos ser capaces de generar un arreglo con valores aleatorios que puedan tener suficiente distancia entre ellos y no repetirse. Además, este arreglo deberá estar ordenado de menor a mayor, y deberemos asegurar que siempre el primer elemento sea cero.

```
int* generarArreglo(int n)
{
    int *A;
    int i;
    A = (int*)malloc(n * sizeof(int));
    if(A == NULL)
    {
        printf(">: Ocurrio un error. Memoria insuficiente.\n");
        exit(1);
    }

    srand(time(NULL));
    for(i = 0; i < n; i++)
    {
        A[i] = (rand())%(3*n));
    }
    A[0] = 0;

    quicksort(A, 0, n-1);

    return A;
}
```

Figure 2: Generar Arreglo

En la figura 2 se muestra la función encargada de generar estos arreglos aleatorios. Vemos que los valores con los que se llenarán llegaría hasta $3n$, asegurando así una probable distancia entre días mayor a f .

Para forzar que el primer valor sea siempre cero, basta con sobrescribir el valor generado en la primera posición con un cero.

Finalmente observamos que para ordenar estos arreglos nuevos se hace uso del algoritmo quicksort visto anteriormente en clase.

Para resolver el problema del granjero, se propone una función como la siguiente:

```

Granjero(n):
  For i = 1 to i <= n:
    f= S[j] + r
    j++
    while(C [i] <= f)
      S[j] = C[i]
      i++

```

Figure 3: Función Granjero para resolver el problema.

Esta función recorrerá el arreglo C elemento por elemento y verificará que la duración del fertilizante del granjero aún abarque al valor de días en C . Si encuentra un valor que lo cumpla, lo guardará y verificará si alguno siguiente cumple la condición. Al final obtiene un nuevo arreglo que contiene los valores de la solución general.

Ahora se realizarán algunas pruebas con arreglos de tamaño 20 y diferentes valores de r .

```

>: Valor de r = 30

>: A[20] = 0 5 5 9 12 16 16 19 21 23 26 26 36 37 41 42 45 46 53 55
>: S[20] = 0 26 55
PS C:\Users\Oscar\Desktop\ESCOM\10mo\Análisis de Algoritmos\Análisis-de-Algoritmos\Practica5>

```

Figure 4: Salida de la función Granjero con $r = 30$.

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

>: Valor de r = 24

>: A[20] = 0 0 1 4 5 9 10 10 11 13 14 16 20 27 32 33 46 51 56 58
>: S[20] = 0 20 33 56
PS C:\Users\Oscar\Desktop\ESCOM\10mo\Análisis de Algoritmos\Análisis-de-Algoritmos\Practica5>

```

Figure 5: Salida de la función Granjero con $r = 24$.

Además se realizó una prueba con el arreglo del ejemplo.

```
>: Valor de r = 30  
>: A[12] = 0 29 36 50 52 66 71 85 100 117 127 129  
>: S[12] = 0 29 52 71 100 130  
PS C:\Users\Oscar\Desktop\ESCOM\10mo\Análisis de Algoritmos\Análisis-de-Algoritmos\Practica5>
```

Figure 6: Salida de la función Granjero con el ejemplo.

Aquí el único detalle es que cambió el último valor de 129 a 130 sin motivo aparente. Sin embargo, este error parece solo ocurrir con este arreglo en particular. La única diferencia respecto al resto es que este arreglo no es generado por la función previamente explicada, sino que es declarado de forma estática.

3 Experimentación y Resultados

Problema 1. El Granjero Para realizar las pruebas de esta función con mayor velocidad y poder hacer varias en una sola ejecución se definió previamente un valor $r = 30$.

Se decidió realizar un total de 100 pruebas diferentes, en donde cada una cambió el tamaño del vector desde 0 a 100, a continuación se muestran los resultados obtenidos:


x_1	 y_1
0	9
1	9
2	20
3	26
4	26
5	26
6	32
89 más filas Mostrar todos	
96	322
97	328
98	322
99	325

Figure 7: Datos obtenidos.

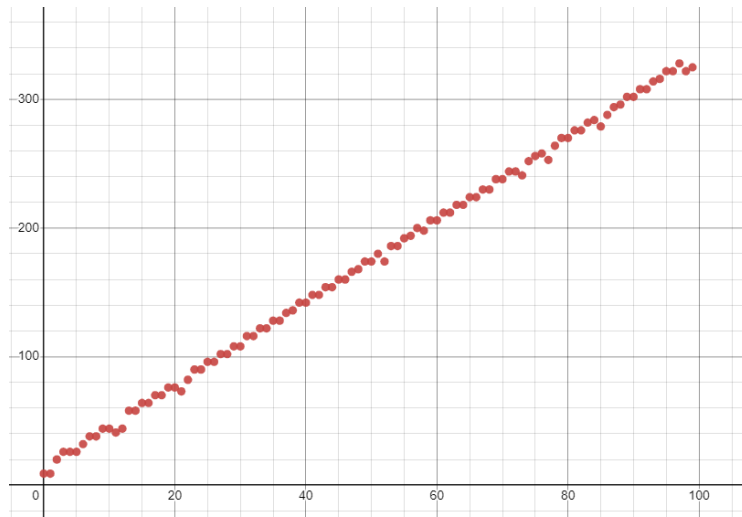


Figure 8: Gráfica obtenida tras las pruebas.

En la figura 8 se muestra en color rojo los puntos obtenidos por la función $\text{Granjero}(n)$ con valores del 0 al 100. Comprobamos de manera visual que se trata de una función lineal $O(n)$

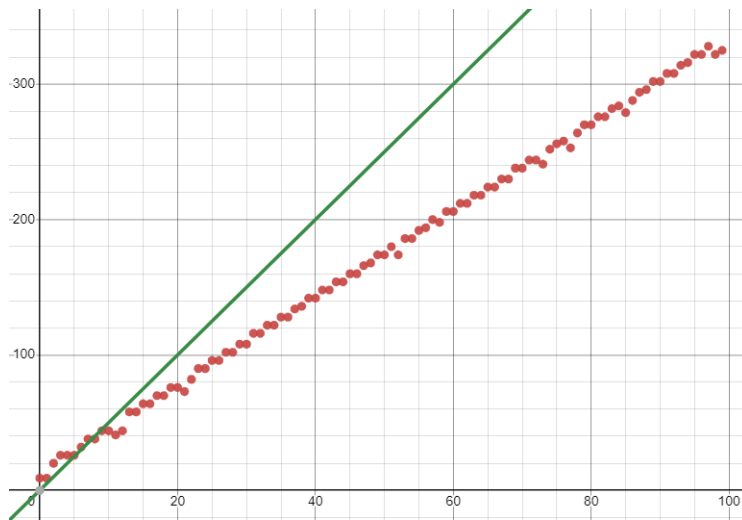


Figure 9: Gráfica obtenida tras las pruebas.

En la figura 9 se muestra en color verde la función propuesta de acotación por arriba $f(x) = 5x$ con un $bn = 20$.

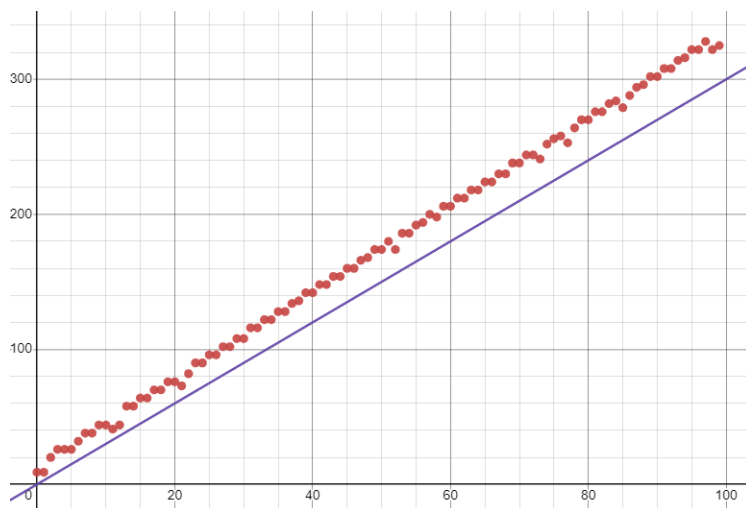


Figure 10: Gráfica obtenida tras las pruebas.

En la figura 10 se muestra en color morado la función propuesta de acotación por debajo $f(x) = 3x$ con un $bn = 20$.

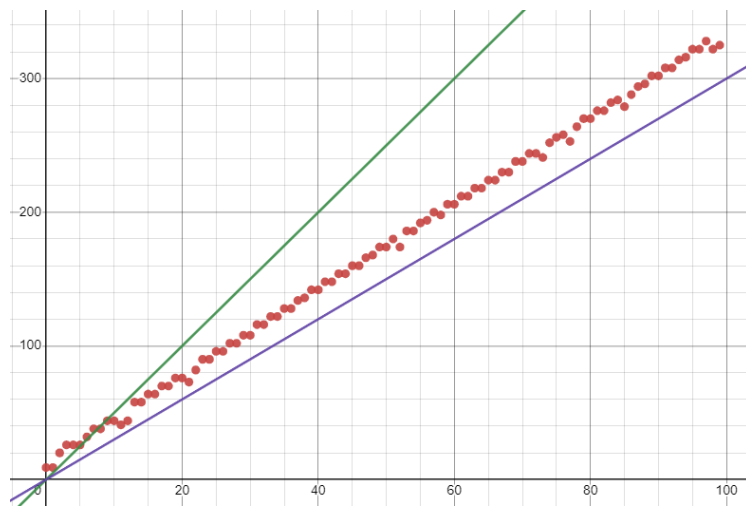


Figure 11: Gráfica obtenida tras las pruebas.

Finalmente en la figura 11 se muestra la vista general de los puntos obtenidos por la función $Granjero(n)$ comparados con las funciones de acotación propuestos. Se mantiene el valor propuesto de $bn = 20$.

4 Conclusiones

En esta práctica logramos comprender de mejor manera el funcionamiento y aplicación de los algoritmos Greedy, además que aprendimos a implementarlo sobre un problema y obtener una solución óptima para este.

Conclusiones López Cabagné Oscar Eduardo. Esta práctica me pareció interesante, pues regresamos al uso y manipulación de arreglos para obtener información y, a través de un algoritmo obtener la solución buscada.



Figure 12: López Cabagné Oscar Eduardo

5 Anexo

Orden de complejidad de Huffman.

Sabemos que la extracción de una cola de prioridad tiene complejidad $O(\log n)$. Como el algoritmo recorrerá cada uno de los elementos a codificar, tendrá también Complejidad $O(n)$. Por lo tanto, el orden de complejidad de todo el algoritmo iterativo de Huffman será $O(n \log n)$.

Orden de Complejidad del Algoritmo de Kruskal.

Tomando n =al n'umero de vertices de un grafo y a =el número de vertices, ordenando las aristas de menor a mayor peso se obtiene un orden de complejidad temporal resulta $O(a \log n)$.

Algoritmo de Prim.

Este algoritmo encuentra un conjunto de aristas que forman un árbol pasando por todos los vértices de un grafo. El peso total de las aristas será el mínimo. Este algoritmo tiene orden de complejidad temporal $O(n^2)$.

```
Prim (Grafo G)
/* Inicializamos todos los nodos del grafo.
La distancia la ponemos a infinito y el padre de cada nodo a NULL
Encolamos, en una cola de prioridad
    donde la prioridad es la distancia,
    todas las parejas <nodo, distancia> del grafo*/
por cada u en V[G] hacer
    distancia[u] = INFINITO
    padre[u] = NULL
    Añadir(cola,<u, distancia[u]>)
distancia[u]=0
Actualizar(cola,<u, distancia[u]>)
mientras !esta_vacia(cola) hacer
    // OJO: Se entiende por mayor prioridad aquel nodo cuya distancia[u] es menor.
    u = extraer_minimo(cola) //devuelve el mínimo y lo elimina de la cola.
    por cada v adyacente a 'u' hacer
        si ((v ∈ cola) && (distancia[v] > peso(u, v)) entonces
            padre[v] = u
            distancia[v] = peso(u, v)
            Actualizar(cola,<v, distancia[v]>)
```

Figure 13: Pseudo-código Algoritmo de Prim.

Orden de Complejidad del Algoritmo de Dijkstra.

Tomando n = número de vértices de un grafo, el algoritmo de Dijkstra tiene orden de complejidad $O(n^2)$.

```

DIJKSTRA (Grafo  $G$ , nodo_fuente  $s$ )
  para  $u \in V[G]$  hacer
    distancia[ $u$ ] = INFINITO
    padre[ $u$ ] = NULL
    visto[ $u$ ] = false
  distancia[ $s$ ] = 0
  adicionar (cola, ( $s$ , distancia[ $s$ ]))
  mientras que cola no es vacía hacer
     $u$  = extraer_mínimo(cola)
    visto[ $u$ ] = true
    para todos  $v \in \text{adyacencia}[u]$  hacer
      si  $\neg$  visto[ $v$ ]
        si distancia[ $v$ ] > distancia[ $u$ ] + peso ( $u$ ,  $v$ ) hacer
          distancia[ $v$ ] = distancia[ $u$ ] + peso ( $u$ ,  $v$ )
          padre[ $v$ ] =  $u$ 
          adicionar(cola, ( $v$ , distancia[ $v$ ]))

```

Figure 14: Pseudo-código Algoritmo de Dijkstra.

6 Bibliografía

- Programación Competitiva. Algoritmos Greedy. Recuperado 24 de Mayo de 2022, de <https://aprende.olimpiada-informatica.org/algoritmia-voraz>.
- Algoritmos Voraces. Recuperado 24 de Mayo de 2022, de <https://www.cs.upc.edu/>.
- Algoritmia/Algoritmos voraces. Recuperado 24 de Mayo de 2022, de https://es.wikibooks.org/wiki/Algoritmia/Algoritmos_voraces.
- Programación Competitiva. Recuperado 24 de Mayo de 2022, de <http://programacioncompetitivaufps.github.io/slides/6-Greedy/index.html/>.
- El Algoritmo Voraz. Recuperado 24 de Mayo de 2022, de <https://www.neoteo.com/el-algoritmo-voraz/>.
- Algoritmo de Prim. Recuperado 24 de Mayo de 2022, de <https://sites.google.com/si>