Subiendo la Escalera con PD - Ruta en Grafos Mediante Algoritmos Greedy Diseño y Análisis de Algoritmos - INFO145

Integrantes: Cristóbal Felipe Rebolledo Oyanedel, Renton Daniel Tapia Yefi, Rodrigo Alejandro Erlandsen Cárcamo, Patricio Alberto Lobos Ojeda

Profesor responsable: Héctor Ferrada

Universidad Austral de Chile

Problema 1: Subiendo la escalera con programación dinámica.

Teniendo un arreglo *E[1..n]* representando a la escalera donde, si *E[i]* = 0 representa que el escalón en la posición *i* está roto, mientras que si *E[i]* = 1 representa que este escalón está bueno. También tenemos el arreglo *saltos[1..k+1]* que contiene los p^k saltos tal que p^k <= *n*, *p* es un valor dado por el usuario que indica la base de los saltos y *n* indica el tamaño de la escalera.

Para la solución exhaustiva queremos encontrar todas las posibles sumas de valores de *saltos[]* que nos den exactamente *n* sin pisar un escalón roto, sean *saltos[i]* los valores del *saltos desde i=1..k+1*, si *ac* == *n* entonces, incrementamos en uno la cantidad de soluciones posibles, si no, nos aseguramos de que *ac + saltos[i]* == 1 y resolvemos el mismo problema para *ac + saltos[i]* y sumamos el resultado del subproblema a la solución final, siempre y cuando *E[ac+saltos[i]]* == 1, es decir, no pise un escalón roto.

Para la solución utilizando programación dinámica utilizaremos un arreglo auxiliar *A[1..n]* que almacenará las soluciones de los subproblemas más pequeños desde un enfoque *bottom-up*, para *j=1..k+1*, guardará un *1* en la *A[saltos[j]]* indicando que puede llegar dando solo un salto, luego, iteraremos *i=1..n* y guardaremos en *A[i]* el subproblema de tamaño *i*. Si *i - saltos[j] >0* y *E[i-saltos[j]]* != 0 entonces, lo consideraremos como un salto válido por lo que agregaremos a *A[i]* las soluciones almacenadas en *A[i-saltos[j]]* para que la solución final se encuentre en *A[n]*.

Pseudocódigo Fuerza bruta:

Input:

E[1..n] = Arreglo de tamaño n, 1 y 0s, representando si se puede o no pisar un escalón.

saltos[1..k+1] = Vector de enteros de tamaño *k*, con los saltos en potencias de $p^{0..k}$.

pisar[1..q] = Vector de enteros tamaño *q*, con el camino más corto para llegar al escalón *n*.

menor[ ] = Vector de enteros de tamaño dinámico, que guardará el camino menor.

count = Variable global.

n = Tamaño del arreglo E.

ac = Acumulador.

Output:

Vector *menor[ ]* con el camino más corto para que Súper Mario llegue a la estrella.

Cantidad de maneras en las que Súper Mario puede llegar a la estrella.

```
pisandoEscalones(E, saltos, pisar, n, ac, menor){
    if (ac == n) then {                                              //O(c)
        if (menor.size( ) = 0 OR pisar.size( ) < menor.size( ) ) then {  // O(c)
            menor = pisar
        ac = 0
        count++
        }
    }
    else{
        for i = 1 to k do {                                          // O(k)
            if(ac + saltos[i] <= n AND E[ac+saltos[i]-1] = 1) then {  // O(c)
                pisar.push(ac + saltos[i])
                pisandoEscalones (E, saltos, pisar, n, ac + saltos[i], menor)
                pisar.pop()
            }
        }
    }
}
```

Ya que la condición de corte es que la función reciba ac + saltos[i]=n, entonces,
ac = n - saltos[i].
Los elementos de *saltos[ ]* están dados por potencias de *p* desde *0* hasta *k* tomando en cuenta que p^k <=n entonces el tiempo para el peor caso (r = 1) sería:

T(n) = k +T(n-1) + T(n-p^2) +...+ T(n-p^k)

Lamentablemente resolver esta ecuación se nos hizo imposible, sin embargo, aún podemos concluir de ella, tomando en cuenta que se iterará $p^i - p^k$ veces (con i desde *0..k*) para cada valor de *saltos[i],* podemos decir que para valores pequeños de *p* nos generará un *k* muy grande, pudiendo llegar a provocar un tiempo de ejecución exponencial. (Esta conclusión viene de la similaridad que tiene nuestra ecuación con la ecuación de recursividad de fibonacci).

Pseudocódigo Programación Dinámica:

Input:

E[1..n] = vector de tamaño *n*, *1* y *0s*, representando si se puede o no pisar un escalón.

saltos[1..k]= Vector de enteros de tamaño *k*, con los saltos en potencias de p^k.

n = Tamaño del arreglo E.

k = Tamaño del vector saltos.

Output:

El vector A en la posición *n-1*, es decir, la cantidad de formas en las que Súper Mario puede llegar a *n*.

```
pisandoEscalonesPD(E, saltos, n , k) {
    Sea A[1..n] un arreglo de tamaño n.
    for (i = 1 to n) do                                          // O (n)
        A[i] = 0

    for (j = 1 to k ) do {                                       // O(k)
        if (saltos[j] <= n && E[saltos[j]] != 0) then {          // O(c)
            A[saltos[j]] = 1
        }
    }
    for(i = 1 to n) do {                                         // O(n)
        for (j= 1 to k) then {                                   // O(k)
            if (i – saltos[j] >= 0 && E[i – saltos[j]] = 1) then { // O(c)
                A[i] = A[i] + A[i – saltos[j]]
            }
        }
    }
    return A[n]                                                  // O(c)
}
```

El tiempo de ejecución de este código es:

T(n) = n*k => O(n*k), para una gran cantidad de casos el k será un valor pequeño por lo que podríamos considerar O(n)

Con k igual a la parte entera de $\log_p n$

¿Es posible esperar un comportamiento muy diferente en la eficiencia de la PD si el valor de r es muy pequeño o muy grande? Del mismo modo: ¿Qué se espera en los casos que el valor de la potencia p sea muy grande o pequeño?

El tamaño del r no afecta al algoritmo PD, ya que este lo usa solamente para comparaciones que utilizan tiempo constante, por lo que el orden sigue siendo O(k*n), por otro lado, si el p es muy pequeño entonces aumentaría el valor de k, ya que k es la parte entera de $\log_p n$, a su vez aumentando el tiempo de ejecución, así mismo, si p es muy pequeño disminuiría el valor de k, disminuyendo el tiempo de ejecución.

Explique de manera clara y objetiva por qué la utilización de la programación dinámica resulta beneficiosa para la solución de este problema.

Usar un algoritmo que aplica programación dinámica es beneficioso por varios motivos, uno de estos motivos es que, considerando otros algoritmos como solución, como es el caso del algoritmo que aplica fuerza bruta, su tiempo asintótico podría llegar a ser exponencial, y esto para un 'n' grande es bastante deficiente puesto que se demora demasiado en entregar la solución. Por lo que, al aplicar un algoritmo con programación dinámica, su tiempo asintótico se reduce a 'k*n', lo cual es bastante significativo. Otros motivos por lo que el algoritmo PD es beneficioso viene por el hecho de guardar en otra estructura de datos, como lo son los vectores, los resultados de los subproblemas anteriores, ahorrando así, una gran cantidad de tiempo. Ahora bien, esto se puede apreciar mejor con los siguientes gráficos:

1)


Gráfico segundos vs n con p = 2 y r = 2
Tiempo fuerza bruta:
Tiempo PD:
Segundos
160
140
120
100
80
60
40
20
0
133.434
42.4781
2.37796
0.0073983
1.90E-06
n = 25
2.30E-06
n = 35
2.10E-06
n = 40
2.40E-06
n = 42
n

2)

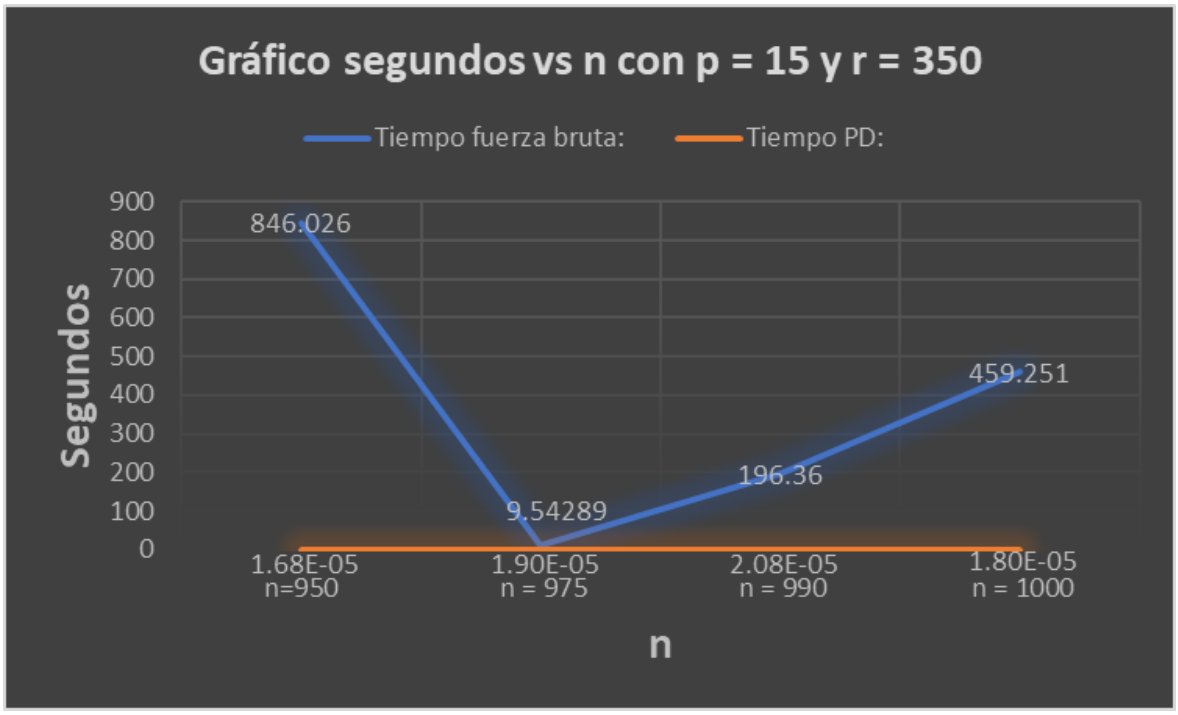


Gráfico segundos vs n con p = 15 y r = 350
Tiempo fuerza bruta:
Tiempo PD:
Segundos
900
800
700
600
500
400
300
200
100
0
846.026
459.251
196.36
9.54289
1.68E-05
n=950
1.90E-05
n = 975
2.08E-05
n = 990
1.80E-05
n = 1000
n

Para toda la experimentación presentada se utilizó un programa escrito en C++ que implementa las soluciones y genera de forma pseudoaleatoria los escalones rotos, este programa solicita los datos *n*, *p* y *r* mostrados en el gráfico además de una semilla, nosotros utilizamos la semilla "3" para nuestros experimentos.

De los gráficos anteriores podemos concluir que, en la práctica la solución con fuerza bruta puede alcanzar un comportamiento similar a exponencial para valores de *n* muy grandes, por otro lado, la solución PD analizada gráficamente, tiene un comportamiento muy cercano a ser constante en valores de n pequeños. (Se infiere del gráfico 1)

Además de que el valor de *n* afecta directamente al tiempo de ejecución, otro caso que no habíamos considerado y gracias al analizar el gráfico 2), pudimos concluir que el valor de *r* y la generación pseudoaletoria de los escalones rotos afecta bastante a los tiempos de ejecución del algoritmo de fuerza bruta, puesto que no solo depende del valor de *n* y *r*, sino, también en dónde se encuentran los escalones rotos.

Si bien no pudimos seguir analizando otras entradas, debido a que al utilizar *n* y *r* muy grandes el algoritmo de fuerza bruta demoraba demasiado o entregaba 0 soluciones, entonces eran casos muy extremos y por este motivo, no pudimos probar el algoritmo de fuerza bruta como lo teníamos previsto.

Problema 2: Ruta en grafos mediante algoritmos Greedy.

Para el algoritmo de fuerza bruta, solo sería necesario hacer todas las comparaciones posibles, de modo que vaya preguntando la ruta óptima desde la capital hasta cada puerto de forma exhaustiva, luego, las rutas óptimas desde cada isla habilitada para recibir barcos hacia la capital regional, una vez hecho esto sumaremos para cada puerto todas las rutas marítimas para llegar a todas las islas habilitadas, la menor de estas sumas será nuestra ruta óptima.

Para la solución Greedy, utilizaremos el algoritmo de Dijkstra para el grafo de las ciudades, así obtendremos rápidamente el costo para los puertos, luego, también utilizaremos el algoritmo de Dijkstra para los archipiélagos considerando que un grafo no dirigido puede ser un grafo dirigido donde del nodo q_i al nodo q_j tienen aristas en ambas direcciones con el mismo peso, esto nos daría el camino óptimo para llegar desde la capital hasta las islas capaces de recibir los barcos, posteriormente, al igual que la solución con fuerza bruta, para cada puerto sumaríamos el peso de todas las rutas marítimas hacia todos los puertos con el fin de obtener la ruta óptima.

¿Cómo afecta la variación de valores de k en el tiempo de ejecución del pseudocódigo?

La variación de los valores de *k* afectaría directamente a la cantidad de iteraciones de 'u' operaciones a ejecutar, haciendo un mayor número de comparaciones y cálculos, siendo directamente proporcional al valor *k* entregado, además, de forma general, este valor sirve para crear los puertos y para definir los costos.

¿Es posible aplicar alguna mejora a su propuesta Greedy a fin de acelerar el tiempo de ejecución de su algoritmo para ciertos casos especiales? Justifique su respuesta

Cuando la cantidad de puertos 'p_i' sea cercana a *1*, al ejecutar Dijkstra en las ciudades, se puede optimizar una vez que el algoritmo de Dijkstra haya expandido los *k* puertos, este no continúe con su ejecución. Así, si bien el tiempo asintótico es el mismo, en la práctica, este acelerara su ejecución, terminando esta primera parte apenas se alcancen los *k* puertos.

Pseudocódigo solución Greedy:

Input:

Costo[1..k][1..q] = vector de costos asociados a los barcos, o sea CostoBarcos.

p[1..u] = vector que almacena los puertos de las n ciudades.

a[1..w] = vector que almacena las islas aptas para recibir barcos.

DijC[1..r] = vector que almacena Dijkstra entre 's' y las 'n' ciudades con

x = |E|.

DijI[1..k] = vector que almacena Dijkstra entre 'z' y las 'm' islas con

y = |E'|.

r = cantidad de islas que aceptan Barcos.

k = cantidad de ciudades con puertos.

Output:

Camino más corto entre la ciudad 's' y la isla 'z'.

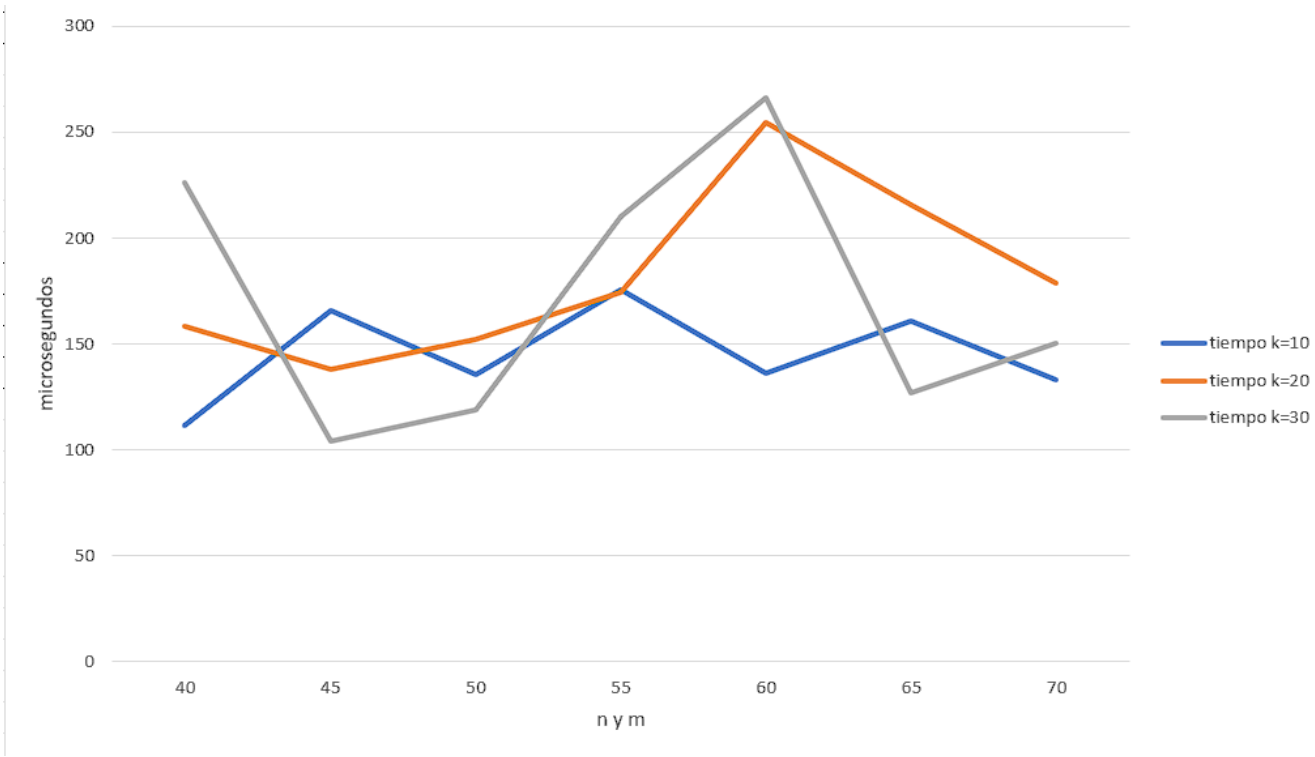
```
Solucion(Costo,p, a, DijC, DijI, r, k) {
        Habiendo calculado Dijkstra de 's' a las n ciudades almacenado en el
         vector DijC, y de 'z' a las 'm' islas almacenado en el arreglo DijI.

        PuertoEco = DijC[p[0]]
        NP = p[0]
        IslaEco = diJi[a[0]]
        NI = a[0]
        BarcoEco = Costo[0][0]
        suma = 0
        menor = PuertoEco + IslaEco + BarcoEco
        for (i = 1 to k) do { // O(k)
                for (j = 1 to r) do { // O(r)
                        suma = DijC[p[i]] + Costo[i][j] + DijI[a[j]]
                        if (suma < menor) then { // O(c)
                                PuertoEco = DijC[p[i]]
                                IslaEco = DijI[a[j]]
                                BarcoEco = Costo[i][j]
                                NP = p[i]
                                NI = a[j]
                                menor = PuertoEco + IslaEco + BarcoEco
                        }
                }
        }
        print("la ruta más economica es: ")
        print("Ciudad: ", NP, " con costo de: ", PuertoEco)
        print ("Isla: ", NI, " con un costo de ", IslaEco)
        print ("El costo del barco fue: ", BarcoEco)
        print ("Costo total: ", PuertoEco + IslaEco+ BarcoEco)
}
```

T(n) = (c*k*r) + [n + nlog(n) + xlog(n)] + [m + mlog(m) + ylog(m)]

= O(kr + (n + x)log(n) + (m + y)log(m)).

Gráfico microsegundos vs n y m



Para la experimentación no estamos tomando en cuenta el tiempo la creación de los grafos. Los experimentos se llevaron a cabo en un programa escrito en C++ que recibe *n*, *m*, *k* y una semilla como argumentos, para nuestra experimentación nosotros utilizamos la semilla "1000".

Gracias a este gráfico, podemos concluir que, como nuestros grafos tienen una generación pseudoaleatoria, la solución no se ve afectada significativamente por los valores de *n*, *m* y *k* al momento de la ejecución de nuestra solución Greedy, enfrentándose de manera bastante eficaz ante los tamaños dados y sin variaciones significativas de los resultados independientemente de estos. Cabe resaltar, que el tiempo tienden a aumentar a la hora de aumentar el número de k esto explica los picos que tiene en ciertos valores, aun así, los resultados están medidos en microsegundos haciendo que los valores extremos no sean tan significativos.

Se probó con valores de 30000 *n* y *m* con un *k* pequeño y grande, pero no se apreció una diferencia considerable en la velocidad del resultado (unos 6 segundos cada uno).