Facultad de Ciencias de la Ingeniería UACh

Informe Grupo 12

Subiendo la Escalera con PD - Ruta en Grafos Mediante Algoritmos Greedy

Asignatura:

- Diseño y Análisis de Algoritmos - INFO145

Académico responsable:

- Dr. Hector Ricardo Ferrada Escobar

Integrantes:

- Cristián Alfonso Cárdenas Morales
- Carolina Andrea Obreque Higueras
- Benjamín Enrique Parra Barbet
- Pascal Gabriel Salinas Schmidt

1. Resumen

Existen diversas técnicas para el diseño de algoritmos que pueden marcar la diferencia en la eficiencia de un programa. En este informe se analiza el comportamiento de dos estrategias: Programación dinámica (PD), aplicada a un problema combinatorio y algoritmos greedy, aplicados a un problema de grafos. Para contrastar la diferencia, se implementaron y analizaron algoritmos de fuerza bruta. Para el primer problema, el algoritmo de fuerza bruta posee una complejidad asintótica exponencial, frente al algoritmo PD cuya complejidad es lineal logarítmica. Para el segundo problema, fuerza bruta posee complejidad factorial, versus Greedy con complejidad lineal logarítmica. Esto se ve reflejado en los experimentos, donde se puede notar la gran diferencia en los tiempos de ejecución entre los algoritmos de fuerza bruta y los que aplican una técnica para el diseño de algoritmos. Lo anterior lleva a que, si se escoge la estrategia correcta para crear un algoritmo, su rendimiento empírico puede marcar una enorme diferencia en la ejecución.

2. Introducción

El diseño y análisis de algoritmos es un área muy estudiada que permite discriminar y otorgar métricas para algoritmos con el fin de obtener soluciones eficientes, que sean capaces de ejecutarse en un tiempo factible en una computadora. Existen diversas técnicas para el diseño de algoritmos aplicables a distintos problemas de la vida cotidiana. En el presente informe se presenta un análisis detallado de dos técnicas ampliamente utilizadas en el mundo de los algoritmos: Programación Dinámica y Algoritmos Greedy.

En la fase temprana de creación de algoritmos, generalmente se opta por la realización de programas que, como mínimo, resuelvan el problema propuesto sin considerar el rendimiento como factor determinante en su funcionamiento. En problemas que requieren de una alta optimización debido a su naturaleza, esto incurre en tiempos de ejecución elevados, en algunos casos llegando a la total infactibilidad. Es por esto que el uso de estrategias adecuadas de programación es imperioso en este tipo de problemas.

En este trabajo se presentan dos problemas con las características mencionadas anteriormente y que requieren de un alto grado de optimización: el primero, obtener el número de formas posibles de subir una escalera, dando saltos que cubran cierto número de escalones. El segundo, hallar el camino óptimo (de menor costo) entre dos capitales, cuya representación se realiza a través de grafos. Se plantea resolver el problema 1 y 2 utilizando programación dinámica y algoritmos greedy, respectivamente. A cada problema se le añaden ciertas restricciones, para demostrar adaptabilidad que poseen los métodos tratados.

El objetivo de este informe es demostrar que la utilización de las técnicas correctas para el diseño de algoritmos puede marcar una gran diferencia con respecto a una versión que resuelva los problemas anteriores por fuerza bruta.

3. Metodología

3.1. Subiendo la escalera

A continuación se presenta la metodología utilizada para resolver el problema "Subiendo la escalera" propuesto inicialmente. Constará de una solución por fuerza bruta y una solución eficiente utilizando programación dinámica.

3.1.1. Algoritmo de fuerza bruta:

Para el algoritmo de fuerza bruta primero se inicializan los 3 vectores con los valores correspondientes, luego se invoca al método *fuerza_bruta_cont()*, que cuenta el total de soluciones existentes para el problema, una por una.

Pseudocódigo:

Input:
E ← Arreglo de booleanos, representando si un escalón se puede pisar o no.
X ← Arreglo de enteros, con todos los saltos que puede dar Super Mario.

Output:
t ← Número total de soluciones.

```
setup(E, X){
    c = 0 //escalón inicial
    return fuerzaBruta(E, X, c)
}

fuerzaBruta(E, X, c){
        t = 0
        foreach (x in X) do {  //O(k)
            if (c + x < n ∧ E[c+x] == True) then
                t = t + fuerzaBruta(E, X, c+x)
            else if (c + x == n) then
                t = t + 1
    }
    return t
}
```

Costo asintotico:

Para el peor de los casos, se considera por simplicidad que no existen peldaños rotos.

La ecuación de recurrencia del algoritmo es:

$$T(n) = \sum_{i=0}^{k} T(n - p^{i}) + c, T(1) = T(0) = c;\ T(m) = 0,\ \forall m < 0$$

donde $k = |X| = \lfloor log_p n \rfloor$ es el número total de posibles saltos que puede realizar Súper Mario en un momento dado.

Luego, para definir una cota superior simple:

$$T(n) <= \sum_{i=0}^{k} T(n-1) + c = kT(n-1) + c = k^2 T(n-2) + kc + c$$
$$= k^3 T(n-3) + c\left(k^2 + k + 1\right) = \ldots = k^n + c\sum_{i=0}^{n-1} k^i$$

Esto concluye que el costo asintótico (worst case) seria:

$$T(n) = O\left(k^n + c\sum_{i=0}^{n-1} k^i\right) = O\left(k^n\right)$$

Lo cual en términos de n queda como:

$$T(n) = O\left(\lfloor log_p n \rfloor^n\right)$$

Hipótesis:

Este algoritmo es muy ineficiente, puesto que evalúa todas las posibilidades sin considerar otras soluciones obtenidas con anterioridad, por lo que, complementado con el costo asintótico obtenido previamente, se estima que será el algoritmo más lento para valores grandes de n.

3.1.2. Algoritmo con Programación Dinámica (PD):

Una aproximación más eficiente para resolver este problema, es utilizando programación dinámica. Si solo interesa saber la cantidad de caminos posibles, únicamente se requieren las soluciones obtenidas para llegar a todos los escalones previos a cada salto.

Los pasos a seguir son:

1) Por cada salto, guardamos en el arreglo de escalones (arreglo inicializado en 0) un 1, ya que hay una manera de llegar hasta allí desde el escalón 0.
2) Por cada escalón destruido, guardamos en el arreglo un -1
3) Por cada escalón [0..n-1] iteramos 4,5,6 y 7) :
4) Si el escalón tiene un -1, pasamos al siguiente escalón, sino vamos a 5).
5) Por cada salto iteramos 6):
6) Si la escalera en el escalón previo a ese salto existe y no está destruido, hacemos el paso 7)
7) Sumamos el valor (cantidad de soluciones) de ese escalón al actual.
8) Devolvemos el valor guardado en el arreglo de escalones, en la última posición.

Pseudocódigo:

Input:

```
S: Arreglo [0..k-1] con todos los saltos potencias de p
E: Arreglo [0..n-1] con los escalones, como enteros.
D: Arreglo [0..r-1] con los escalones rotos.
```

Output:

```
E[-1]: Cantidad de caminos posibles que llegan al último escalón.
```

```
escalera64(S="Arreglo con los saltos", E="Arreglo con los peldaños",
D="Arreglo con los peldaños rotos"){
    foreach s ∈ S                                          // O(k)
        E[s-1] = 1                                         // O(1)
    foreach d ∈ D                                          // O(r)
        E[d-1] = -1                                        // O(1)

    for (i=0; i<n; i++)                                    // O(n)
        if (E[i] ≠ -1)
            foreach s ∈ S                                  // O(k)
                if (i-s >= 0 ∧ E[i-s] ≠ -1)
                    E[i] += E[i-s]                         // O(1)
    return E[-1]
}
```

Costo asintotico:

Luego de combinar los costos en notación O expresados en el pseudocódigo, se obtiene:

$$T(n) = O(k + r + kn),$$

donde sabemos que $r \leq n \Rightarrow r < kn,\ k > 0$, resultando:

$$T(n) = O(kn)$$

Entonces, cómo sabemos que $k = \lfloor log_p n \rfloor$:

$$T(n) = O(n * \lfloor log_p n \rfloor) \leq O(nlog_p n) \leq O(nlog_2 n)$$

ya que $p \in Z,\ p \geq 2$.

Hipótesis:

Este algoritmo aprovecha de adquirir resultados de escalones superiores en base a soluciones de escalones inferiores, por lo que solamente es necesario conocer, a lo más, k soluciones anteriores para resolver cualquier escalón. Se trata entonces de un algoritmo altamente eficiente, que resolverá el problema con tiempos de ejecución muy bajos, siendo mucho más eficiente que el de fuerza bruta.

Comentarios:

- ¿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?

Las variaciones del parámetro r no deberían afectar de forma significativa el rendimiento del algoritmo de programación dinámica, puesto este valor solo indica por qué escalones no se puede pisar, sin requerir un procedimiento extra aparte de una condición. En cuanto al valor de p, se espera que mientras menor sea su valor, más disminuya su rendimiento, esto debido a que Súper Mario puede realizar un mayor número de saltos en cada escalón, por lo que se depende de un mayor número de soluciones previas.

Esto se justifica con los costos asintóticos obtenidos anteriormente, donde se aprecia que el crecimiento asintótico no depende de r, sin embargo, sí depende de k, que es un valor calculado a partir de p y que alcanza su máximo cuando p = 2.

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

Si se analiza de forma profunda el funcionamiento del algoritmo de fuerza bruta, se puede observar que éste se encuentra constantemente calculando soluciones que ya se encontraron anteriormente, generando un árbol de recurrencia extremadamente extenso y repetitivo. Como se depende de soluciones obtenidas anteriormente, utilizar programación dinámica para guardar estas soluciones y no tener que volver a calcularlas resulta útil, transformando la complejidad asintótica del problema de exponencial a lineal logarítmica.

3.1.3. Variaciones del problema:

Variación: Guardar todas las soluciones

Podemos tener todas las soluciones para cada escalón empleando programación dinámica para hacer que cada escalón guarde su propio camino, de esta manera, si en un programa necesitaremos acceder a varios escalones, podemos calcular todos y luego bastaría con acceder al arreglo y a tendríamos los caminos y valores en cada escalón, entonces ejecutandolo una sola vez lentamente, podríamos acelerar procesos futuros.

El algoritmo necesita una forma para almacenar los caminos, así que basta con hacer que cada escalón sea una estructura de la forma (value, road) donde se guarda la cantidad de maneras de llegar hasta ese escalón en E[i] y en road todas las maneras, respectivamente. Por último podemos eliminar la recursividad para hacer que sea más fácil de estimar los costos asintóticos.

Los pasos a seguir son:

Partimos el arreglo de escalones desde [0..n-1], tenemos un set con los escalones destruidos, y usamos un set para almacenar los números de escalones destruidos otros para los saltos posibles:

1) Iteramos 2), 3) y 4) desde 0 por cada escalón:
2) Si el escalón (que parte desde 0) + 1 no está destruido (Revisamos usando un conjunto), realizamos 3 y 4.
3) Si el escalón es además un salto posible desde 0, le sumamos uno al valor (inicializado en 0) y le agregamos a las rutas ese camino unitario.
4) Revisamos los escalones previos los cuales pueden llegar al actual solamente con un salto, le sumamos su valor y agregamos su camino, mas el numero de escalon actual, al escalón E[i]
5) Retornamos al último escalón.

Pseudocódigo:

Input:
S: Set [0..k-1] con los saltos potencias de p menores al largo de E
E: Arreglo [0..n-1] con todos los escalones
D: Set que guarda los peldaños rotos.
n: Largo de E

Un set puede ser implementado como una tabla hash, por lo que en el peor caso son O(n), pero esto solo se da cuando hay una muy grande cantidad de colisiones, de otra manera son O(1).

```
escalera64(S: Saltos, E: Escalones, D: Escalones rotos, n: Largo de E){
    for (i=0; i<n; i++) do {                    // O(n)
        if (i+1 ∉ D) then {                     // O(r), Avg(1)
            if(i+1 ∈ S) then {                   // O(k), Avg(1)
                E[i].value += 1;
                E[i].road.add(i + 1);
            }
            foreach s ∈ S do {                      // O(k)
                if (i-s >= 0){
                    E[i].value += E[i-s].value;
                                foreach path ∈ E[i-s].road do         //
O(value[n-1])
                            E[i].road.add(path + [i + 1]);    // O(1)
                }
            }
        }
    }
    return E;
}
```

Complejidad asintótica:

Peor caso: O(n*(r + (k) + k*Value[n]))
pero ya comprobamos que k <= logp(n), además value[n] crece mucho más rápido que k y r, ya que son las soluciones posibles para llegar a ese escalón, por lo que en el peor caso sería:

$$T(n) = O(n \bullet log_p n \bullet max\{value\})$$

El algoritmo mejora mientras la potencia p es más grande, pero value[n-1] crece muy rápido mientras aumenta n, por lo que con n muy grandes, tanto la complejidad como la memoria necesaria se vuelve un problema serio. Esta solución hace que con p pequeño y n grandes el algoritmo sea impráctico, al igual como ocurrió con la solución de fuerza bruta debido a su naturaleza exponencial.

Variación: Guardar un camino cuyo largo sea el menor

En esta variación, solo guardamos la solución más corta para cada escalón. Pensando en optimizar el rango de números posibles y reducir la cantidad de accesos a memoria, marcaremos con INF ("infinito") previamente en el valor del escalón cuando están rotos, de esa manera podemos usar el bit de signo para almacenar más números y no debemos acceder a una estructura adicional para revisar si están rotos, además, si tenemos los saltos ordenados, ya que de igual manera debemos pasar por todos los previos, podemos ir desde el menor hasta detenernos cuando el salto sea mayor al número del escalón [i] actual, de esta manera nos ahorramos una búsqueda, aunque el comportamiento asintótico será el mismo.

Los pasos serán los siguientes:

1) Por cada salto s, vemos el escalón correspondiente E[S], si no está marcado como infinito, le ponemos un 1 al valor y guardamos el camino [s].
2) Por cada escalón E[i], iteramos 3 y 4:
3) Si el escalón no es infinito, y mientras el salto s sea menor que el escalón actual:
4) Si el escalón previo al salto no es ni 0 ni infinito, le sumamos su valor y le agregamos su camino más el escalón [i], si el camino hasta [i] no ha sido calculado o el camino actual hace menos saltos que el que tenía guardado.
5) Se retornan los contenidos del último escalón.

Pseudocódigo:

```
Input:
 S[0..k-1]: Arreglo ordenado con los saltos posibles,
 E[0..n-1]: Arreglo de Escalones: (value:0,road:[]),
 n: cantidad de escalones a subir,
 k: cantidad de saltos distintos

Output:
La información del último escalon (E[n]):
   -Un camino más corto (Como un vector, por ejemplo).
   -Cantidad de formas distintas de llegar a ese escalón.
```

```
escalera64(S: Arreglo de saltos, E: Arreglo de escalones, n, k) {
    for (j = 0; j < k; j++) do {                                    // O(k)
        if (E[S[j]-1].value != INF) then {
            E[S[j]-1].value = 1;
            E[S[j]-1].road.push_back(S[j]);                         // O(1)
        }
    }
    for (i = 0; i < n; i++) do {                                    // O(n)
        j = 0;
        if (E[i].value ≠ INF) then {
            while (j < k ∧ (s=S[j++]) <= i) do {                    // O(k)
                if (E[i-s].value != 0 ∧ E[i-s].value != INF){
                    E[i].value += E[i - s].value;
                    r = E[i - s].road; // r: camino del salto anterior
                    r.add(i+1);
                    if (E[i].road.empty() ∨ r.size() < E[i].road.size()+1) then
                        E[i].road = r;                              // O(1)
                }
            }
        }
    }
    return E[n-1];
}
```

Complejidad asintótica:

Twc = O(k + n(k)) Pero como vimos anteriormente, k <= logp(n) entonces:*

$= O(log_p n + n*log_p n)$

Twc = O$(n \bullet log_p n)$

Esta mejora es debido a que al solo guardar un elemento, no necesitamos acceder a todas las soluciones de los escalones previos al salto, lo que lo hace escalar inmensamente mejor a la solución anterior O(n*logp(n)*value[n]) tanto en memoria como en la cantidad de operaciones necesarias.

3.2. Ruta en Grafos

3.2.1. Algoritmo con Fuerza Bruta:

El algoritmo de fuerza bruta en palabras simples evalúa todas las

combinaciones posibles de caminos que permitan llegar de la capital continental a la capital del archipiélago, almacenando el camino que obtuvo la menor suma de costos (óptimo), que será la solución al problema. Debido a que se están evaluando combinaciones de caminos sin más, el algoritmo tendría una complejidad factorial, volviéndolo extremadamente ineficiente, por lo que al considerar un caso con varios puertos e islas, esta aproximación no es factible.

3.2.1. Algoritmo Greedy:

Se realizan los siguientes pasos para resolver el problema:

1. Ejecutar Dijkstra para $G = (V, E)$, ω, partiendo desde S. Así tendremos las rutas de costo mínimo para llegar, desde S, a cada puerto p_i, con $1 \leq i \leq n$. Denotamos el costo de S a p_i como $C(S, p_i)$. Sea $Puertos$ el conjunto con los n costos, $Puertos = \{C(S, p_1), C(S, p_2), \ldots, C(S, p_k)\}$. Esto se logra en un tiempo de $O((n + |E|)logn)$.

2. Ejecutar $logm$ veces una modificación a Breadth-First Search (que funcione con los pesos de las aristas del grafo, en lugar de la cantidad de nodos) para $G' = (V', E'), \omega'$, partiendo de cada una de las islas q_j, $1 \leq j \leq logm$. Así tendremos las rutas de costo mínimo para llegar desde cada isla q_j a Z. Denotamos el Costo de q_j a S como $C(q_j, Z)$. Sea $Islas$ el conjunto con los $logm$ costos, $Islas = \{C(q_1, Z), C(q_2, Z), \ldots, C(q_t, Z)\}$, con $t = \lfloor logm \rfloor$, en un tiempo de $O((m + |E'|)logm)$.

3. Dados $Puertos = \{C(S, p_1), \ldots, C(S, p_k)\}$ e $Islas = \{C(q_1, Z), \ldots, C(q_t, Z)\}$, determinar el par (i, j) que minimice el costo $C(S \to p_i \to q_j \to Z)$, tal que:

$C(S \to p_i \to q_j \to Z) = min\{C(S, p_i) + costoBarco(p_i, q_j) + C(q_j, Z)\}$; esto se hace fácilmente con un doble for para los i en k y para los j en $logm$; en un tiempo de $O(k\ logm)$.

Pseudocódigo:

```
Input:
    Puertos: Grafo Dirigido de tamaño n que representa los
    puertos el cual guarda su símbolo, una relación a otro
    puerto y el costo
    P = ('s','A',3)
    s: Char que representa la capital continental del país
    Islas:Grafo no Dirigido de tamaño m que representa las
```

Islas el cual guarda su símbolo, una relación a otra isla y el costo.
I =(('z','1',4)
z: Char que representa la capital regional del archipiélago
costoBarco: Arreglo que contiene los costos de los puertos habilitados a las islas habilitadas
Puertos_Habilitados:Arreglo[0..k] con los puertos habilitados
Islas_Habilitadas: Arreglo[0..log2(m)] con las islas habilitadas

Output:
min_cost: *El costo mínimo de viajar de una capital a otra*
besti: *La posición en Puertos_Habilitados del puerto que se usó.*
bestj: *La posición en Islas_Habilitadas de la isla que se usó.*

puertosIslas*(Puertos,s,Islas,z,costoBarco,Puertos_Habilitados, Islas_Habilitadas){*
Usamos Dijkstra desde s en el grafo dirigido (Puertos), guardamos los costes en puertos_cost que es una estructura de la forma (puerto_1:coste_1, puerto_2:coste_2, … , puerto_n:coste_n).

Islas_cost = ∅
foreach isla in Islas_Habilitadas {
C(Isla , z) = extraer el costo desde cada isla habilitada para recibir barcos para llegar a z, usando un algoritmo Breadth-First-Search-Modificado(G',q_j,ω') que funcione con pesos en lugar de número de nodos.
Islas_cost = Islas ∪ C(q_j , z)
}

minCost = +inf
best$_i$ = 0
best$_j$ = 0
for i = 1 to k do {
for j = 1 to log m do {
costo = C(s,p_i) + costoBarco(p_i , q_j) + C(q_j ,z)
if (costo < minCost){
minCost = costo
best$_i$ = i
best$_j$ = j
}
}
}
return {minCost, best$_i$, best$_j$ }
// se extrae el camino de s a best$_i$ de la matriz generada en

```
Dijkstra(G, s, ω)
// y el camino desde bestj a z de la matriz generada en
Dijkstra(G', q_j , ω')
}
```

Algoritmo BFS modificado (Variante del BFS que tiene en cuenta los pesos de las aristas):

```
BFS-pesos(G=(V,E), s){
     visitados = ∅
     queue = ∅
     costes = ∅

     queue.add([s,0])
     sea 'costes' un arreglo [1..|V|] de enteros
     costes[s] = 0

     while (queue != ∅){
          costo = queue.pop()
          nodo = costo[1]
          dist = costo[2]
          visitados.add(nodo)

          foreach vecino in G.Ad[nodo] {
               nodoVecino = vecino[1]
               pesoViaje = vecino[2]

               if (nodoVecino in visitados){
                    queue.add([nodoVecino, dist + pesoViaje])
                    visitados.add(nodoVecino)
                    costes[nodoVecino] = dist + pesoViaje
               }
          }
     }
     return costes
}
```

Costo asintotico:

$$T(n, m, k) = O((n + |E|)logn + (m + |E'|)logm + k\,logm)$$

Hipótesis:

Si bien el algoritmo se complejiza al subdividirlo en tres partes, su análisis asintótico sigue siendo con creces mejor que el análisis resultante del algoritmo de fuerza bruta. Por tanto, se espera que este algoritmo presente un buen tiempo de ejecución, permitiendo un número grande de ciudades e islas.

Comentarios:

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

El paso 3 es el único en el que el tiempo se ve afectado por k, ya que si aumenta k aumenta el número de iteraciones que debe realizar el algoritmo. Sin embargo, si el aumento de k no está acompañado de un aumento en el tamaño de n o m, el impacto en el tiempo de ejecución puede no ser notable.

- ¿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:

Si hay muy pocos puertos p_i, entonces la primera parte de ejecutar Dijkstra en la ciudad, partiendo de s y llegando a todas las otras ciudades, se puede optimizar en que 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 acelerará la ejecución, terminando esta primera parte apenas se alcancen los k puertos.

4. Experimentación y Resultados

4.1. Subiendo la escalera

Analisis de graficos - PD:

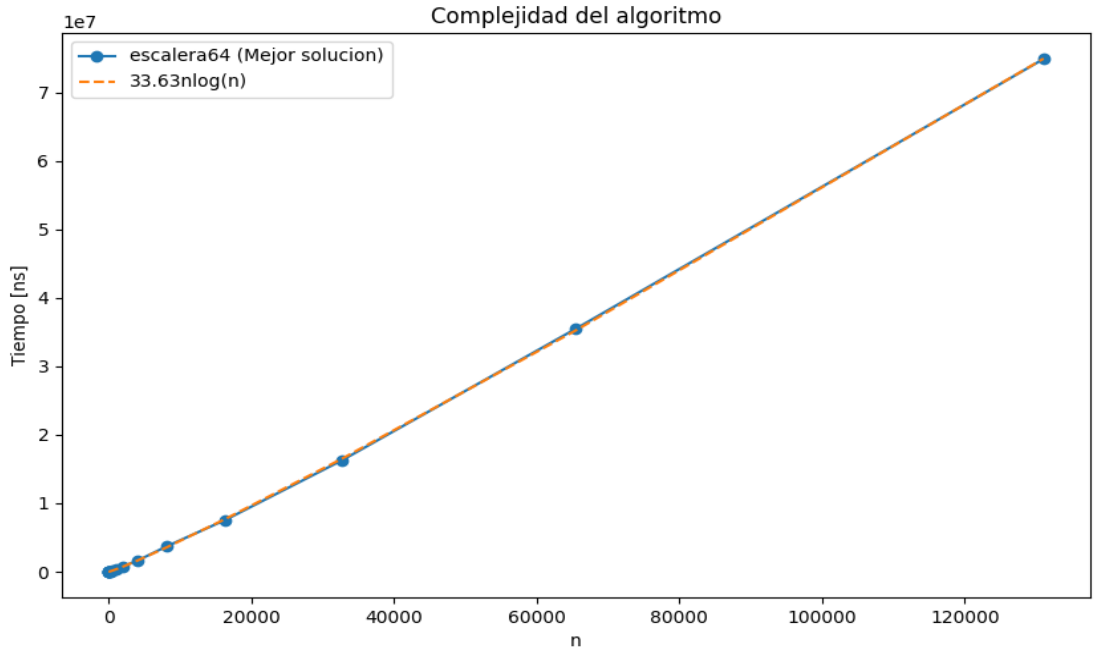
Escalera64 (Mejor solución) con n grandes:

Al fijar r=0 y p=2, estresando el algoritmo lo más posible, podemos ver su comportamiento con valores de n grandes, en concreto, potencias de 2. En la práctica, con c++, es imposible que en un caso normal se busque una escalera de

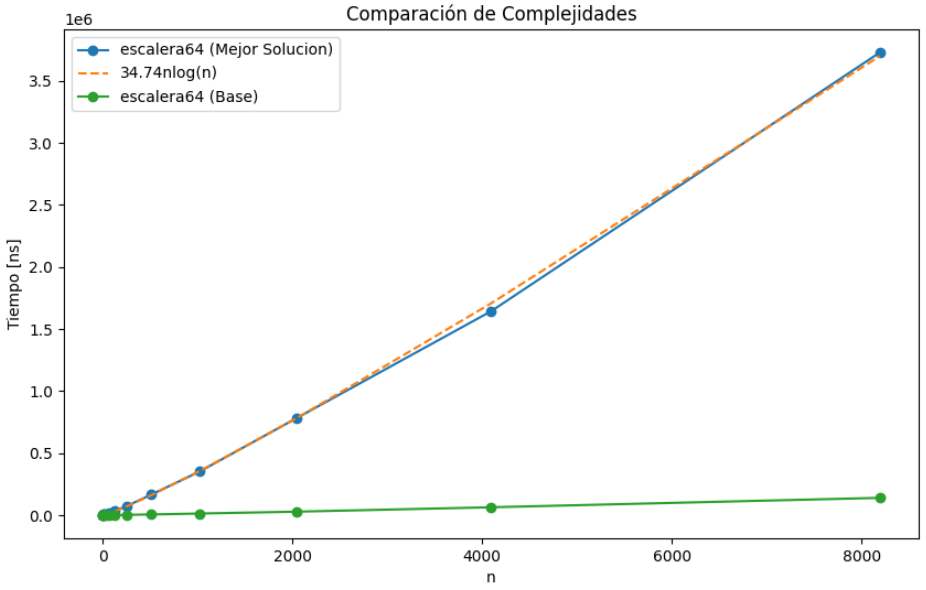
este tamaño, ya que a partir de cierto escalón (si se representa con int, alrededor del número 42) los valores comienzan a hacer overflow y la salida ya no tiene sentido, pero para efectos de ver el rendimiento bruto de esta solución, en el gráfico se ignorara este fenómeno.

Escalera64 (Mejor solución) y su complejidad:

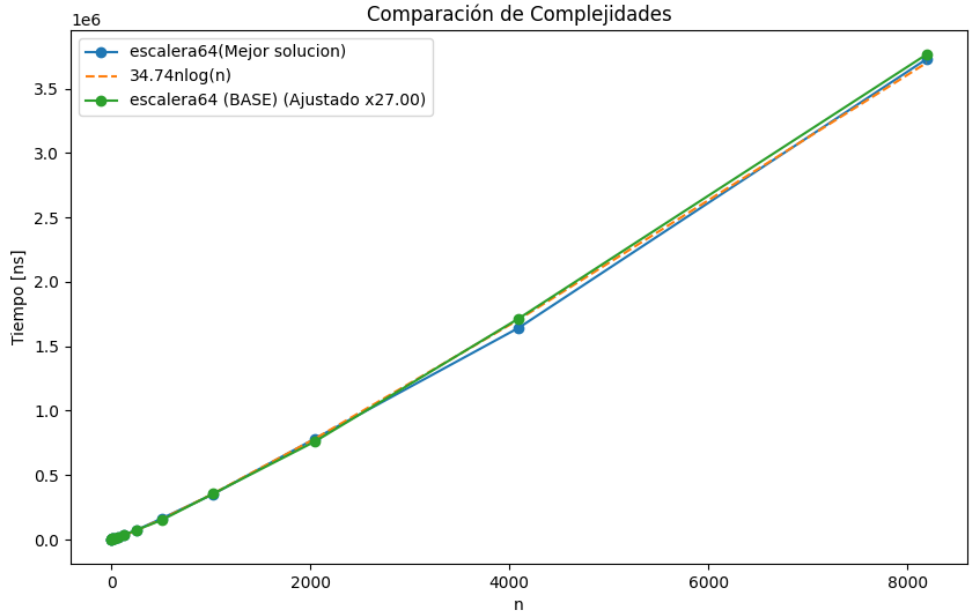
Para ilustrar el orden del algoritmo, en este gráfico se sobrepondrá una curva T(n)=c*n*lg(n), con c = 33.63. Como podemos observar, se aprecia una gran similitud, esto debido a que la complejidad calculada fue de O(n lg n), entonces sabemos que c*n*lg n es una cota igual o superior a la curva definida por el algoritmo, con valores grandes de n.



Por otro lado, al comparar rendimiento con el algoritmo que no guarda las soluciones queda el siguiente gráfico:



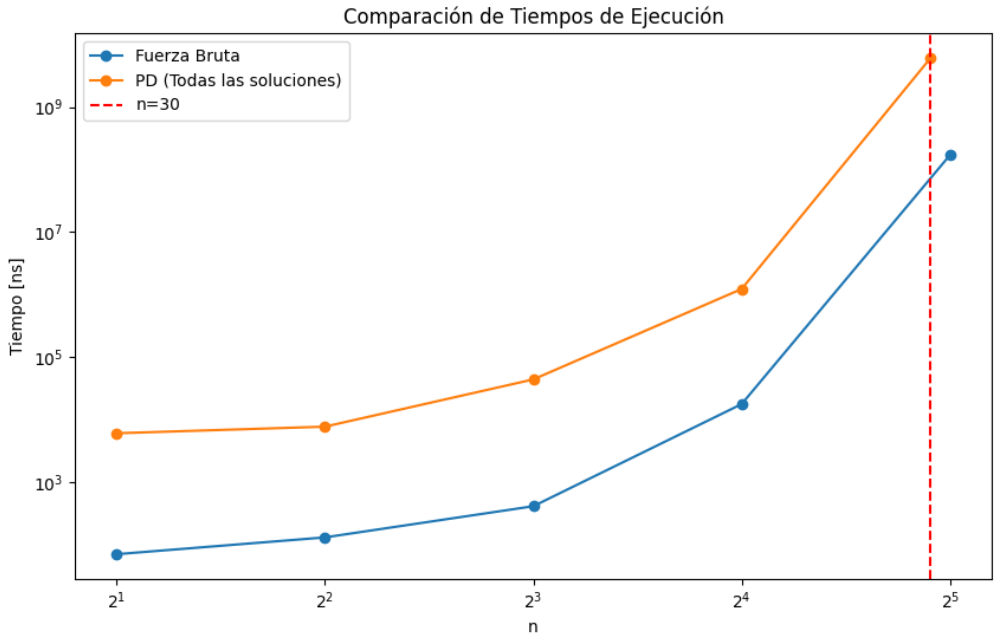
Aunque parezcan muy distintos a simple vista, si se multiplica por una constante los tiempos del algoritmo base, la historia es muy distinta:



Esto ocurre ya que el algoritmo base tiene la misma complejidad que el algoritmo que guarda la mejor solución.

Fuerza bruta vs PD guardando todas las soluciones:

Aunque guardar todas las soluciones en memoria puede parecer mucho peor, el siguiente gráfico ilustra las diferencias en la práctica, usando r=0 y p=2.



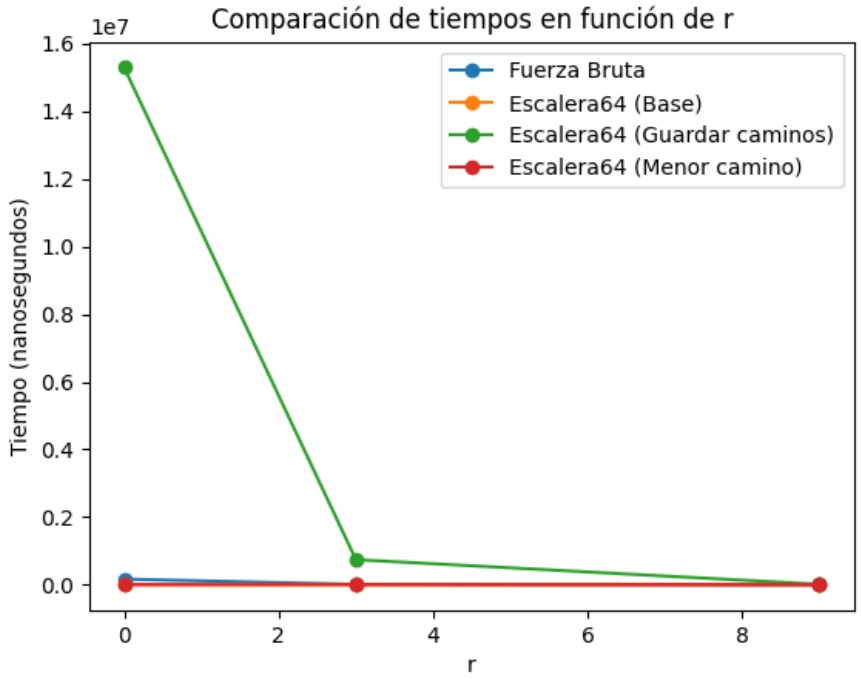
El rendimiento visto no es tanto peor, de hecho, se puede observar como su comportamiento es similar al de fuerza bruta, pero al guardar tantas soluciones en memoria el programa tiene un límite, en este caso, el computador sobre el que se experimentaba, el cual contaba con 16 gb de ram, no podía llegar a guardar más de 30 escalones. Aunque parezca la peor solucion, todavia se puede plantear llegar a utilizarla, ya que una vez calculadas todas las soluciones que se necesiten, no será necesario calcularlas de nuevo, entonces si es necesario tener todos los caminos (y no solo el más corto), la aplicación de este algoritmo, puede seguir siendo viable.

Si se analiza el funcionamiento del algoritmo PD que guarda todas las soluciones, es claro que hay que tener todas las soluciones previas, entonces en realidad, el rendimiento no puede ser mejor que el de fuerza bruta para n pequeños.

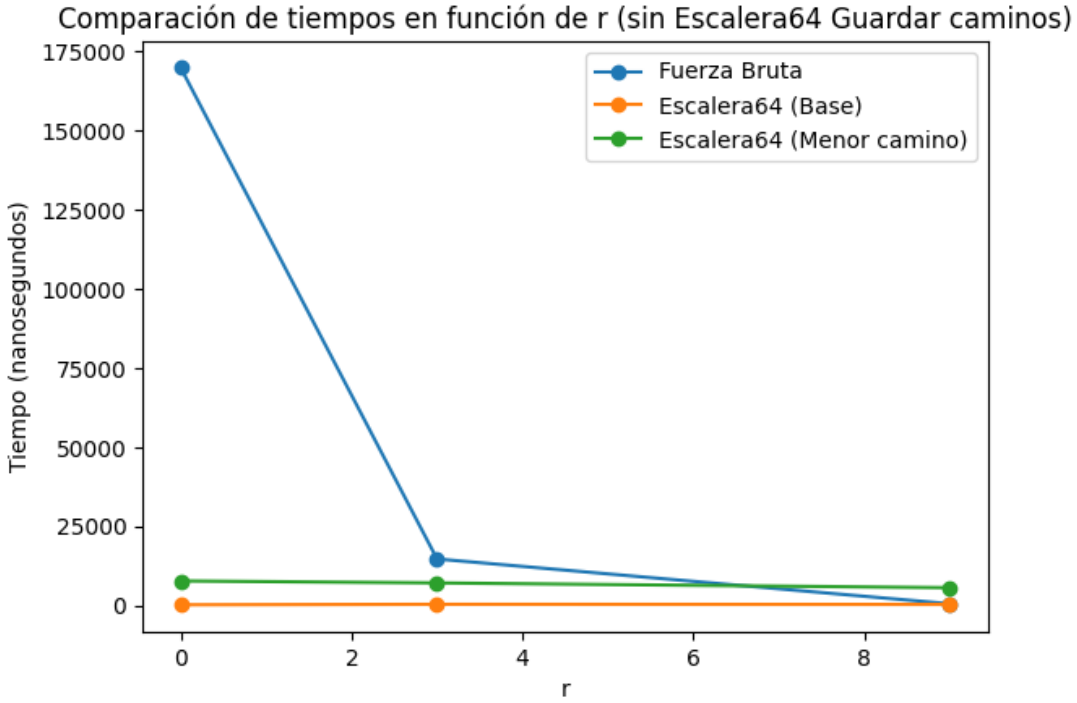
Tampoco puede ser peor que el de fuerza bruta, ya que el de fuerza bruta debe construir el escalón previo (cuando p^i = 1) para poder calcular el actual, entonces, de todas maneras, debe calcular todas las soluciones.
De esto podemos concluir que sus complejidades son theta de cada una, lo que en el gráfico anterior queda muy marcado.

Comportamiento en función de r:

Según lo calculado en los costos asintóticos, la cantidad de escalones destruidos no importa al momento de ver la complejidad de los algoritmos, pero en fuerza bruta si, ya que no debe construir esos números, lo mismo ocurre con la variación que guarda todas las soluciones. Si fijamos n=20 y p=2, los resultados son los siguientes:



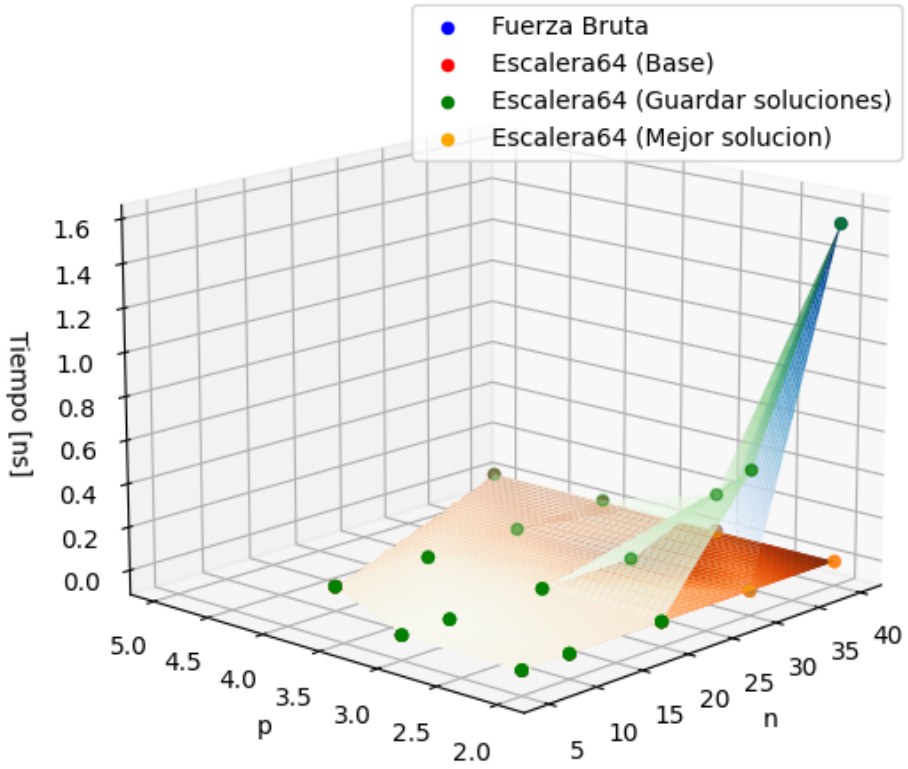
Aquí se muestra como la variación de guardar caminos es la más impactada, ya que no necesita llenar esas posiciones con todos sus caminos posibles.
Si se ignora esa variación, para ver más de cerca el gráfico, el resultado es el siguiente:



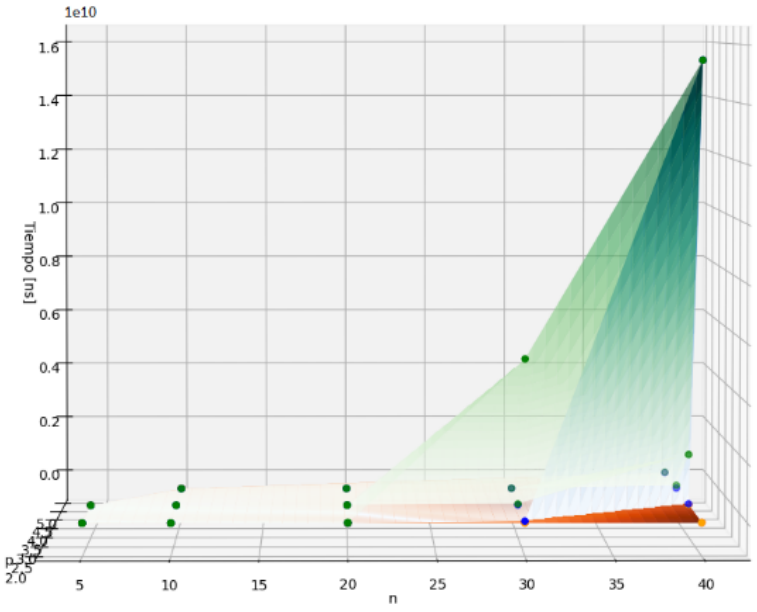
Al escalador base no le importa cuales escalones están destruidos, solo revisa los que necesita y los suma, lo mismo ocurre con la variación de menor camino.

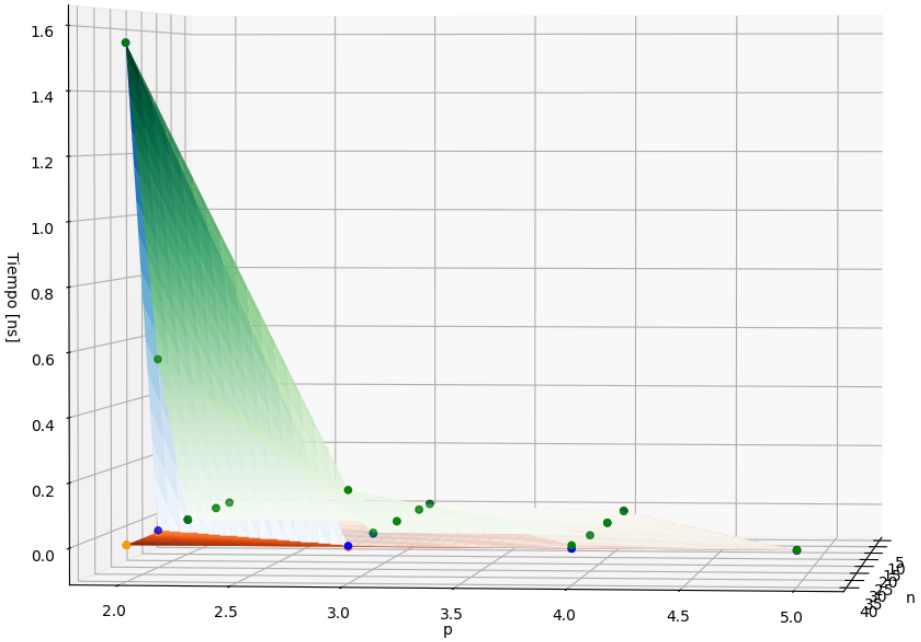
Comparativa general

Aquí se observan claramente las diferencias de rendimiento entre los algoritmos propuestos y los impactos que generan cambiar los valores de p y n. Los siguientes dos gráficos realizan una proyección sobre los planos para simplificar las cosas. Notar que en estos gráficos, el plano verde debería detenerse, ya que luego de n=30 y p=2 el programa no pudo seguir ejecutándose. La escala del eje y está multiplicada por e^10



Proyecciones en los planos NxT y PxT :



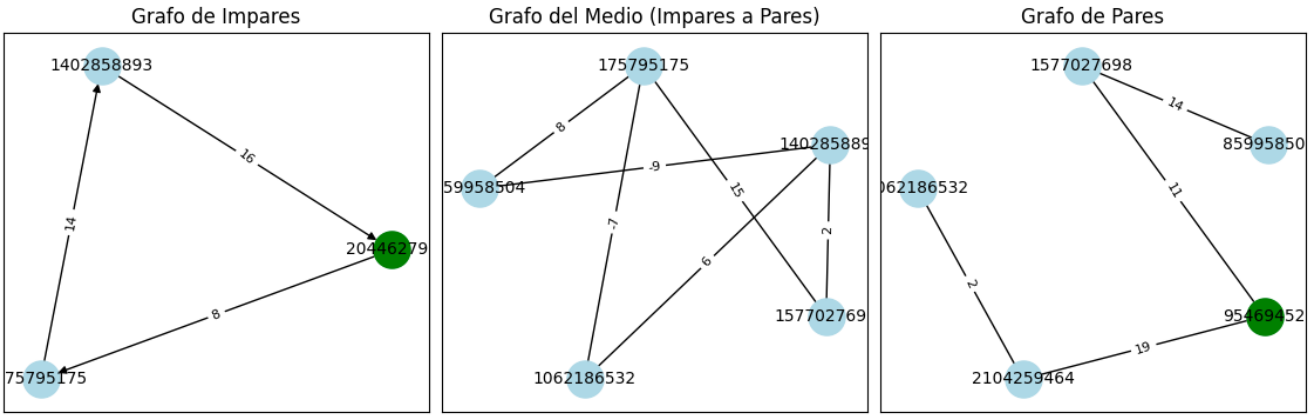


Aquí se observa que aumentar el parámetro ‘p’ genera una tendencia a reducir los tiempos de ejecución especialmente en el algoritmo de fuerza bruta y el algoritmo que guarda todas las soluciones, esto es debido a que se reduce la cantidad de escalones que hay que revisar para poder calcular el valor y caminos que hay que revisar para poder determinar las características del escalón actual. Que el impacto sea más significativo con el de fuerza bruta y la variación antes mencionada, se debe a que el de fuerza bruta debe construir muchos más números y la variación debe almacenar muchas más soluciones posibles.

4.2. Ruta en Grafos

En solGreedyAleatorio.cpp se creó un programa para crear los grafos aleatorios, con n, k y m ingresados por el usuario, esto representando a las ciudades con números impares y las islas con números pares, para después calcular la ruta más corta. Se utilizó un programa de python llamado graficador.py para dibujar las ciudades con sus puertos e islas generadas al azar y comprobar que se crean correctamente.
Este es un ejemplo del grafo generado con n=10 ,k=9 , m=10 y seed=93931230

Grafo de impares representa las ciudades, la capital está destacada con verde
Grafo del Medio representa los viajes en barco como se puede ver hay 9 nodos impares los cuales representan los puertos y 3 nodos pares que representan las islas habilitadas

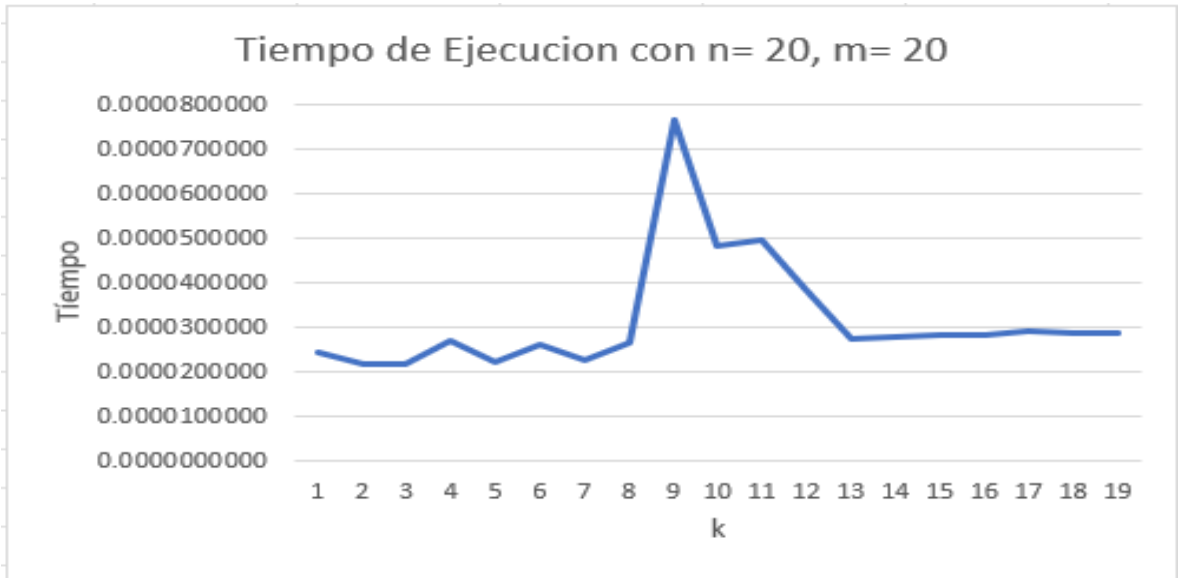


Grafo de pares representa las islas, la capital está destacada con verde
A continuación se ven los resultados entregados por el programa solGreedyAleatorio.cpp:

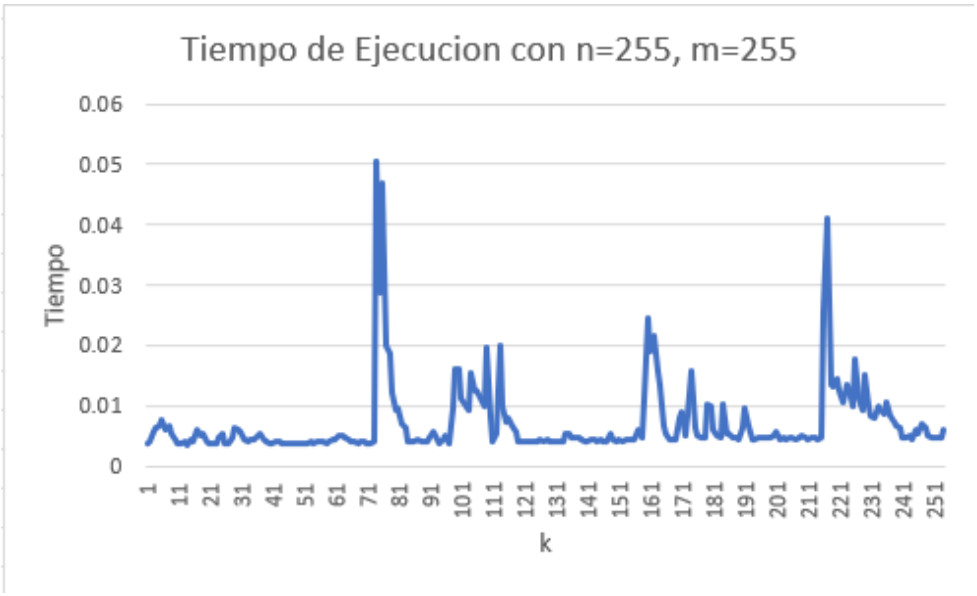
```
('1402858893', '204462795', 16),
('175795175', '1402858893', 14),
('204462795', '175795175', 8),
('1577027698', '95469452', 11),
('2104259464', '95469452', 19),
('1062186532', '2104259464', 2),
('859958504', '1577027698', 14),
('175795175', '1577027698', 15),
('175795175', '1062186532', -7),
('175795175', '859958504', 8),
('1402858893', '1577027698', 2),
('1402858893', '1062186532', 6),
('1402858893', '859958504', -9),
Nodo inicial: 204462795
Nodo final: 95469452

Costo Mínimo: 22
Mejor puerto habilitado: 17579517
Mejor isla habilitada: 1062186532
```

En los siguientes gráficos se prueban variaciones de n, m, k para el algoritmo Greedy. En este primer gráfico en el cual se usan n y m pequeños se puede observar ver que el algoritmo se ejecuta extremadamente rápido y podemos ver que los valores de k no varían mucho el resultado. Se consideran valores pequeños de n y m para simular un problema “realista”, sin embargo, para testear el verdadero potencial del algoritmo se deja como propuesta utilizar valores grandes para estos parámetros.

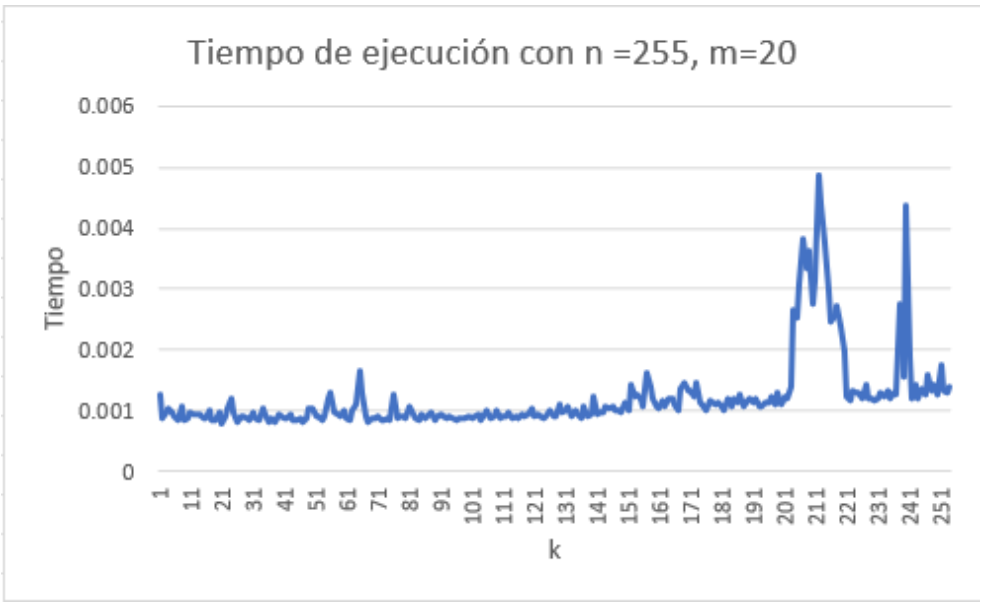


En comparación, cuando n y m son grandes , el código se ejecuta más lentamente,como se puede observar en el eje representado por el tiempo, y al igual que en el anterior las variaciones en los valores de k no producen más que oscilaciones en los tiempos de ejecución, pero rondan el mismo rango de valores. De esto podemos concluir que los valores de k son despreciables para el tiempo de ejecución del algoritmo y solo aportan una leve tendencia a aumentar el tiempo. Esto tiene sentido ya que en el tiempo asintótico n y m se suman a su cantidad de aristas, lo cual produce que crezcan más rápido que k. Los picos en los tiempos pueden haber sido producidas por interrupciones en el equipo donde se hicieron las mediciones, ya que el ordenador contaba con un procesador mononúcleo.

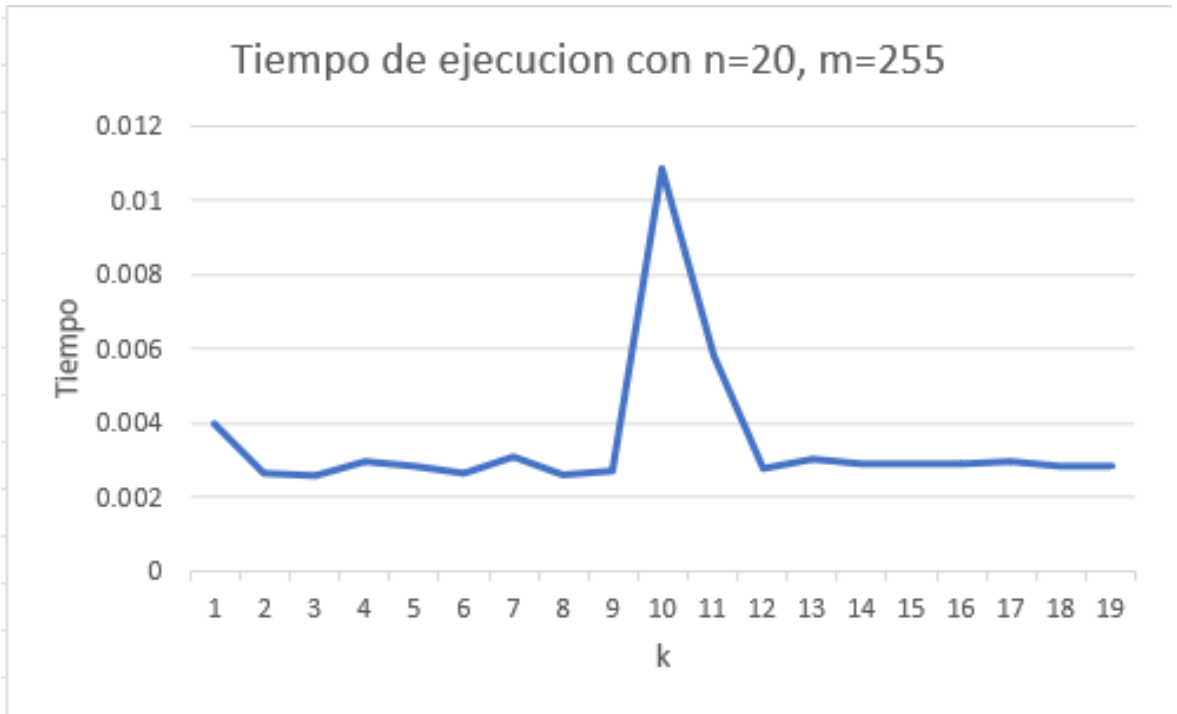


En este se prueba con un n grande pero un m pequeño, sigue la misma lógica que los anteriores, pero se puede destacar que se ejecuta un orden de

magnitud más rápido que en el gráfico anterior, lo que queda claro al comparar las escalas y con estos dos gráficos.



Por último probamos con un n pequeño y un m grande, podemos ver menos oscilaciones en los valores que el anterior y que los tiempos de ejecución se duplican. Esto nos indica que los valores de m tienen mayor significancia para el tiempo de ejecución del algoritmo que n.



5. Conclusión

Escaleras:

Después de revisar todos los códigos podemos ver varias cosas:

1. El código que guarda todas las soluciones es el peor de todos con diferencia, esto a causa de que la cantidad de soluciones crece exponencialmente, lo que se traduce en un enorme usa de memoria terminando en la ralentización del código al punto de quedar por detrás del de fuerza bruta, e incluso limitado a la ram del usuario, por lo que no suele ser una solución viable.
2. El código de Fuerza Bruta sería el segundo peor. Con una solución exhaustiva, y de crecimiento exponencial, es esperable que su desempeño se quede atrás a medida que aumenta el tamaño de la entrada.
3. Luego vendría la variación guarda el camino más corto. Es mucho más eficiente que los anteriores aunque no llega a ser el mejor por el tiempo usado en encontrar la solución más corta. Sin embargo se puede considerar la mejor solución ya que su tiempo es muy cercano al algoritmo más rápido pero a su vez te devuelve información útil, siendo esta la solución más corta.
4. Finalmente quedaría el más rápido, siendo la solución de programación dinámica que no guarda soluciones. Su velocidad en comparación a la anterior solución se da gracias a que no guarda ninguna solución, por lo que su tiempo de cómputo se reduce al cálculo del número de posibles soluciones.
   El problema de esta solución es que no da otros datos aparte del número de soluciones posibles, lo que no resulta muy útil para saber como resolver el problema

Grafos:

Se puede concluir que la solución greedy reduce enormemente el coste de ejecución, pasando de un coste factorial a uno lineal logarítmico. Luego, dentro del algoritmo greedy sus tiempos de ejecución están dados por n y m, siendo m el más significativo de los dos. La variación de los valores de k solo produce oscilaciones en los tiempos pero no es significativo para el tiempo de ejecución del algoritmo.