PRÁCTICA 4: PROGRAMACIÓN DINÁMICA

- Ahmed El Moukhtari Koubaa
- Damián Marín Fernández
- Eduardo Segura Richart
- Jesús Martín Zorrilla

ÍNDICE

1. Descripción del problema.
2. Estudio del ejercicio guiado LCS.
3. Problema del viajante de comercio.
4. Conclusiones.

1. Descripción del problema.

Para esta práctica se nos encomiendan dos tareas principalmente. Una de ellas es hacer un breve estudio de un ejercicio guiado y ya resuelto con el fin de que adquiramos las nociones y los conocimientos sobre programación dinámica para resolver otros problemas. Dicho ejercicio guiado es el LCS. El otro problema que se nos pide resolver es el problema del viajante de comercio (TSP por sus siglas en inglés). Este es un problema de combinatoria cuya solución por fuerza bruta tiene un coste computacional tremendo. Así pues, se nos pide dar una solución a dicho problema mediante programación dinámica.

Ya descritos los problemas a resolver, describamos que más se nos pedía hacer con respecto a estos problemas. Se nos pide hallar la eficiencia teórica, es decir, mediante el análisis del código, también se nos pide la eficiencia empírica consistente en realizar ejecuciones con diversos tamaños (los tamaños van a depender de cada algoritmo) y la eficiencia hibrida, la cual se basa en las previas eficiencias. Además de lo citado previamente también se nos pide describir un caso de ejecución y demostrar así que se ha dado una solución correcta.

Aunque no es parte del problema, en el guion de la práctica también se nos pide realizar esta memoria y preparar una presentación resumida para su exposición en clase.

2. Estudio del ejercicio guiado LCS.

Descripción

Se nos pide en el guion de la práctica estudiar la metodología usada para la resolución de este problema (el LCS a partir de ahora) mediante técnicas de programación dinámica. El LCS consiste en encontrar la subsecuencia común en ambas cadenas de izquierda a derecha y no necesariamente contigua.

Análisis

Analizando la solución que se nos pide estudiar, nos damos cuenta que aplicando técnicas como DyV la eficiencia teórica del algoritmo para resolver este problema es $O(2^n)$, lo cual es inviable en tiempo. Es por motivos como este que debemos usar la programación dinámica para resolver este ejercicio.

Analizando el problema vemos que es aplicable la programación dinámica, pues ciertamente este es un problema multietápico donde en cada etapa se está buscando un carácter de una cadena que coincida con otro carácter de otra cadena. También se cumple el POB ya que la búsqueda de una subsecuencia óptima incluye la búsqueda de una subsecuencia de la anterior también óptima. Este problema también nos garantiza que cuando encontremos la solución será la óptima.

El algoritmo PD diseñado para resolver este problema funciona de una forma muy intuitiva. Se crea una matriz de enteros que se corresponde con una especie de producto de las cadenas a procesar. En esta matriz cada elemento nos indica si hay coincidencia o no entre el elemento de una cadena y la de la otra. Se recorren ambas cadenas (bucle anidado) y se van comparando, si estas coinciden se suma uno a los elementos que se encuentren en la esquina superior izquierda, es decir, en una diagonal. Finalmente, el último elemento de la matriz contiene el número de caracteres coincidentes.

		A	B	C	D	A
	0	0	0	0	0	0
A	0	1	1	1	1	1
C	0	1	1	2	2	2
B	0	1	2	2	2	2
D	0	1	2	2	3	3
E	0	1	2	2	3	3
A	0	1	2	2	3	4

LCS - "ACDA"

Para obtener esta subsecuencia basta con situarse en la esquina inferior derecha e ir recorriendo los elementos diagonalmente, como se indica en la siguiente imagen.

Una pequeña comparación

Hemos implementado un pequeño algoritmo que intenta dar solución a este problema. Dicho algoritmo es muy simple, recorre una cadena comparando todos sus elementos con la otra y si coinciden se añaden a una cadena resultado. Este algoritmo es de orden O(n^2) y para unas cadenas como son las del ejemplo de la solución ("**ACBDEA**" y "**ABCDA**") nos da exactamente el mismo resultado ("**ACDA**"). Atendiendo a los tiempos de ejecución y a los gastos de memoria, vemos que nuestro algoritmo tarda 0.00001 y consume 1524 KB mientras que la versión PD consume 0.08 segundos y 28756 KB que es bastante más. Sin embargo, para un caso de ejecución mucho mayor que este vemos que los resultados son diferentes, lo cual nos indica que nuestro algoritmo no es correcto y que la solución óptima es la del algoritmo PD.

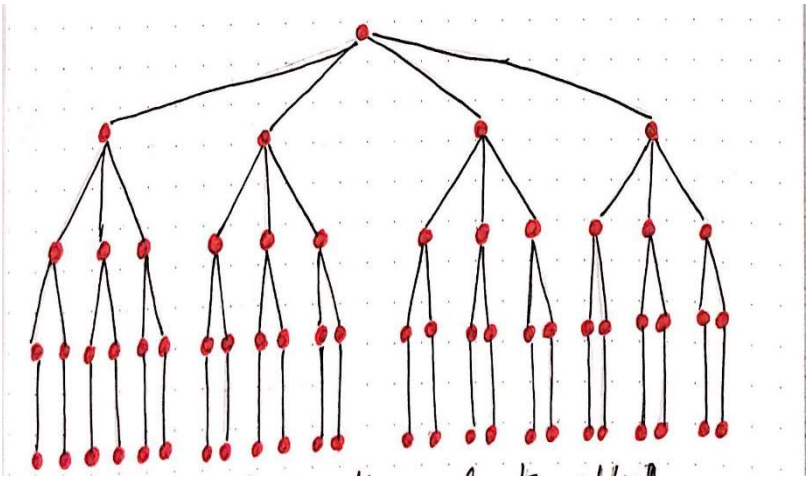
Conclusión

Las técnicas de programación dinámica no son aplicables en el 100% de los casos. Cuando se aplican dichas técnicas se consiguen resultados óptimos. Vemos que en algunos casos y para algunos problemas en concreto mejoran bastante la eficiencia teórica y los tiempos de ejecución, sin embargo, nada nos garantiza que esto sea así para todos los problemas a resolver ni que la eficiencia obtenida sea "buena" (una cuadrática será mejor que una cúbica, pero peor que una lineal, hay que juzgar dependiendo del caso).

3. Problema del viajante de comercio.

Descripción detallada del algoritmo.

Intentar dar una solución al problema del viajante de comercio, en la que el recorrido obtenido sea el óptimo implicaría resolver un problema de combinatoria en el que el número de operaciones crece exponencialmente, es por esto que este es un problema NP-duro.

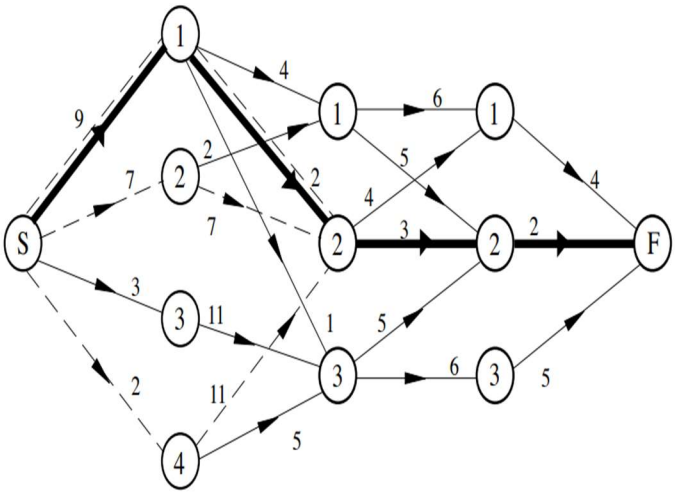


Una solución que diese un recorrido óptimo sería por fuerza bruta, pero como ya dijimos antes la complejidad crece exponencialmente y con esa solución nos encontraríamos con una eficiencia teórica factorial, es decir, O(n!). Con dicha solución para 20 ciudades a recorrer el computador más potente del mundo tardaría 4 veces el tiempo de vida del universo, así pues, esta es una solución inviable.

Sin embargo, observando la solución descrita anteriormente nos damos cuenta de que hay bastantes operaciones y cálculos que se repiten, nos preguntamos ahora: ¿Sería posibles conservar los resultados de esos cálculos y reutilizarlos? La respuesta es sí, mediante la programación dinámica.

Analicemos si podemos usar programación dinámica en este problema.

1. Es un problema multietápico. Formado por vértices (ciudades) particionados en etapas, con arcos adyacentes entre etapas.
2. Cumple el POB. El recorrido mínimo está formado por subrecorridos que son mínimos.
3. Para resolver un problema hay que resolver sus subproblemas.



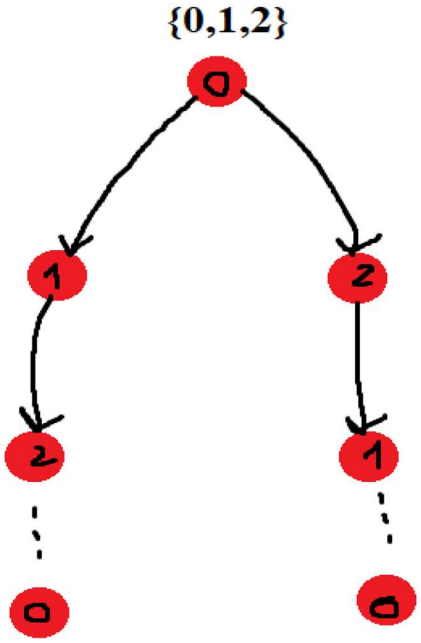
Una vez aclarados todos los puntos citados previamente, podemos comenzar a describir el funcionamiento de nuestro algoritmo. Nuestro algoritmo usa dos matrices, una en la que se encuentran las distancias entre las ciudades y otra con las distancias mínimas entre ciudades y los recorridos, además de un conjunto de ciudades y la ciudad desde la que se va a iniciar dicho recorrido. Así pues, lo que se hace es partir de una ciudad de origen y recorrer el conjunto de ciudades restantes si este no se recorrió previamente (en ese caso nos quedamos con esa distancia y esas ciudades que se encuentran en la matriz de distancias mínimas) para cada una de ellas aplicamos el algoritmo de manera recursiva buscando la distancia mínima, siendo esta vez la ciudad de origen la ciudad actual y pasándole un conjunto de ciudades en el que no se encuentra ella. Repetimos lo anterior hasta llegar a la ciudad de la que partimos inicialmente.

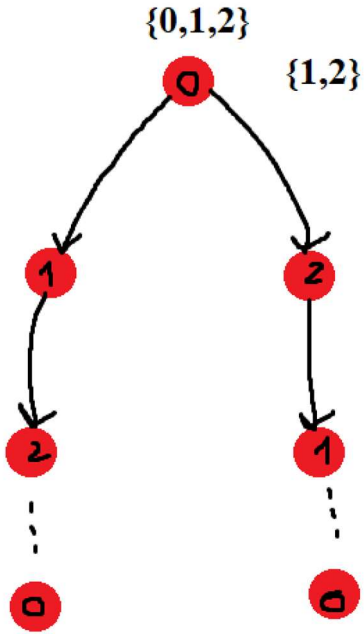
Más detalladamente:

1. Si quedan ciudades por recorrer
    a. Si ese conjunto no se recorrió antes
        i. Recorrerlo ahora aplicando de manera recursiva el algoritmo, que recibirá ahora el conjunto sin la ciudad que se esté procesando y siendo esta la nueva ciudad de origen.
        ii. Guardar la distancia mínima y asignarla a la matriz de distancias mínimas (donde el índice fila es la ciudad que nos da el mínimo recorrido y el índice columna el conjunto).
    b. Si fue recorrido antes, asignar a la distancia mínima el valor de la matriz de distancias mínimas (donde el índice fila es la ciudad que se haya pasado como origen y el índice columna el conjunto de ciudades a recorrer).
2. Si no quedan ciudades por recorrer, asignar a la distancia mínima el valor de la matriz de distancias (donde el índice fila es la ciudad actual y el índice columna es la ciudad de la que partimos principalmente).
3. Devolver la distancia mínima que hayamos encontrado.

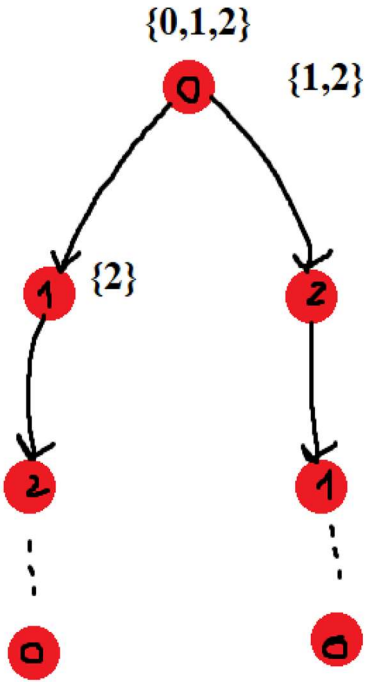
Es posible que esta explicación no haya sido lo suficientemente clara y demasiado abstracta, por ello haremos una representación gráfica del problema y de cómo nuestro algoritmo lo va solucionando. Para representar gráficamente esto podemos usar un grafo orientado o bien un simple árbol partiendo del algoritmo por fuerza bruta. Resolveremos el caso más básico y simple en donde hay alguna elección a realizar, el tsp para 3 ciudades.

Partimos del conjunto de todas las ciudades y quitamos la que escojamos como ciudad de origen, en nuestro ejemplo 0.

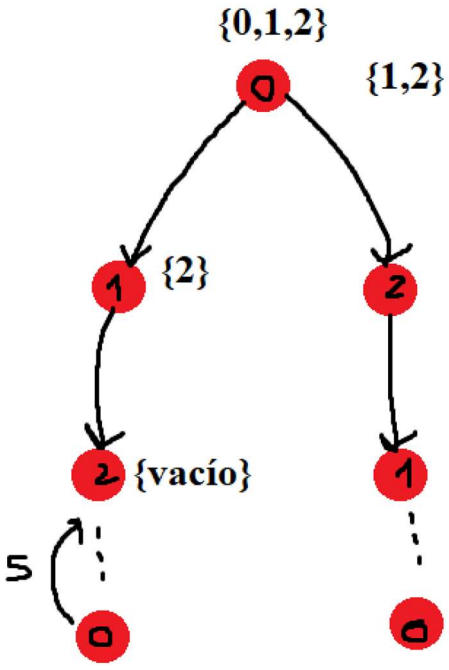




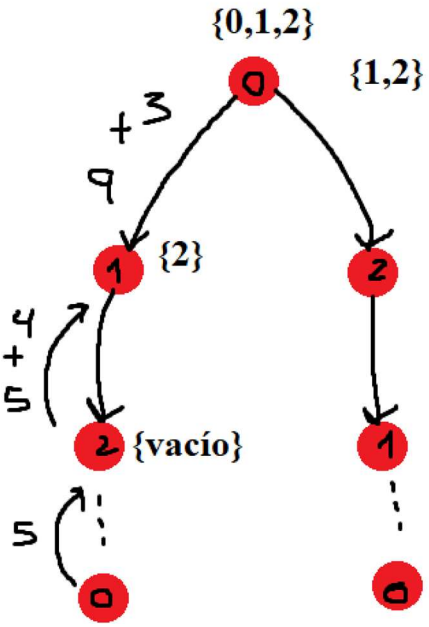
Nos quedan ahora dos ciudades que recorrer. Recorremos ese conjunto aplicando el algoritmo sobre cada una de las ciudades,



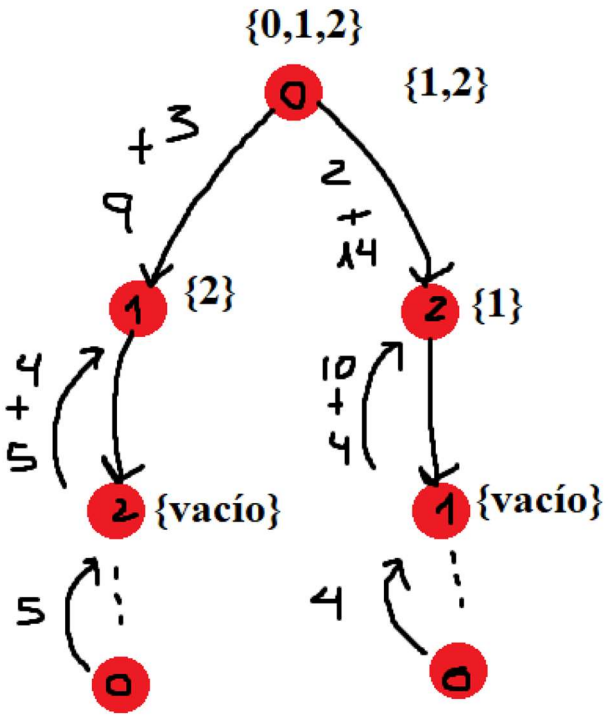
Seleccionando “1” como la siguiente ciudad y quitándola así del conjunto nos queda 2 como única ciudad a la que ir, volvemos a aplicar el algoritmo nuevamente.



Nos queda ahora un conjunto vacío y debemos volver a la ciudad de origen (“0”), haciendo esto devolvemos la iteración anterior el coste de ese camino (en nuestro ejemplo el coste es (2→0) = 5).



Vamos devolviendo las distancias mínimas a las ciudades de las que procedemos y sumamos para obtener el coste del camino.



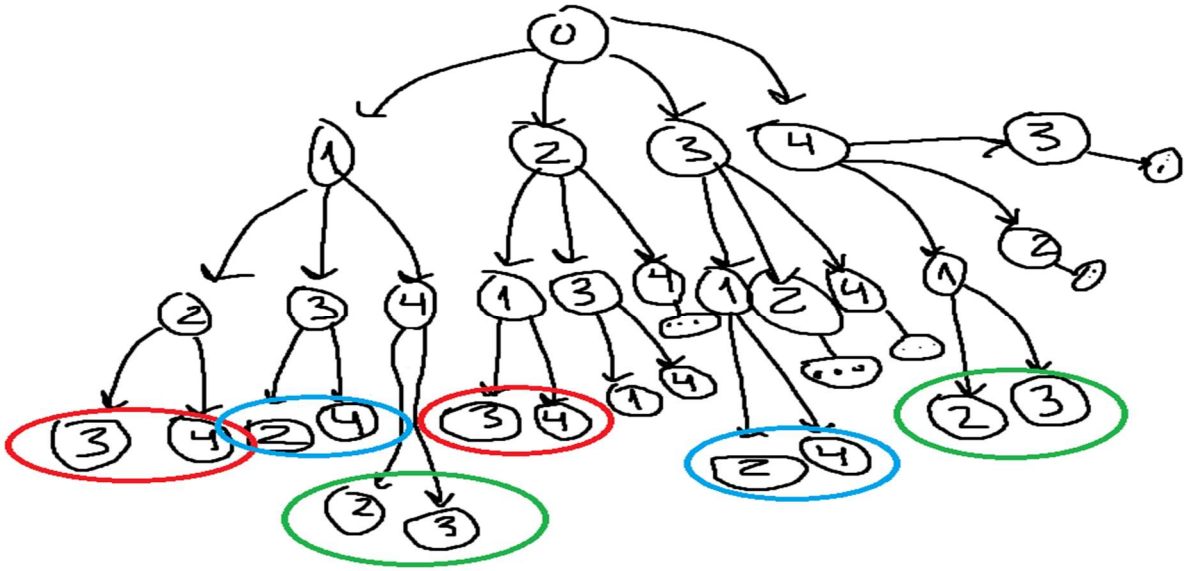
Repitiendo lo mismo en la otra "rama" nos quedarían dos recorridos con dos distancias.

{0,1,2} = 12

{0,2,1} = 16

Nos quedamos, una vez más, con el mínimo de estos dos recorridos.

En el ejemplo tan básico que se acaba de mostrar no se expone toda la potencia del algoritmo ni la diferencia principal con la solución por fuerza bruta (O(n!)). Veamos un breve ejemplo de esta diferencia.



Como vemos en la anterior imagen, hay varios conjuntos ya resueltos que se repiten de los cuales ya sabemos cuál es la mejor opción. Esto se da a mayor escala cuando hay más ciudades y cuando estamos en "niveles" más cercanos a la "raíz" (hacemos referencia a la ciudad de origen, de la que partimos principalmente). Con la solución por fuerza bruta se repiten todos estos cómputos, pero con programación dinámica los realizamos una vez y los guardamos para reutilizarlos más tarde.

Para finalizar la descripción de este algoritmo. Este algoritmo calcula la distancia del recorrido de menor coste, pero no nos da directamente ese recorrido. Para obtenerlo podemos hacerlo de dos formas principalmente (todo esto dependiendo de la implementación). Una forma consistiría en recorrer la matriz de distancias mínimas comparando los conjuntos para saber cuál es la ciudad que se recorrió antes. Otra forma consiste en recorrer toda la matriz e ir buscando nuevamente el mínimo recorrido, solo que en esta ocasión todos los conjuntos necesarios están resueltos y guardados en la matriz, conforme encontramos una ciudad la guardamos en una lista y devolvemos esa lista. Ambas funciones están implementadas en el código así que no entraremos en mucho más detalle.

Cálculo de la eficiencia teórica

La eficiencia teórica del algoritmo que hemos propuesto coincide con la que se menciona en los apuntes de la parte teórica de la asignatura, es decir, O(n^2*2^n). Vamos a justificar esta eficiencia:

- El cálculo de las distancias (j, vacío) requiere n-1 consultas de la matriz.
- El cálculo de todas las distancias (j, conjunto de ciudades) de manera que $1 \leq |\text{ciudades}| = k \leq n-2$
- El cálculo de las distancias (1, ciudades\{j}): n-1 sumas.

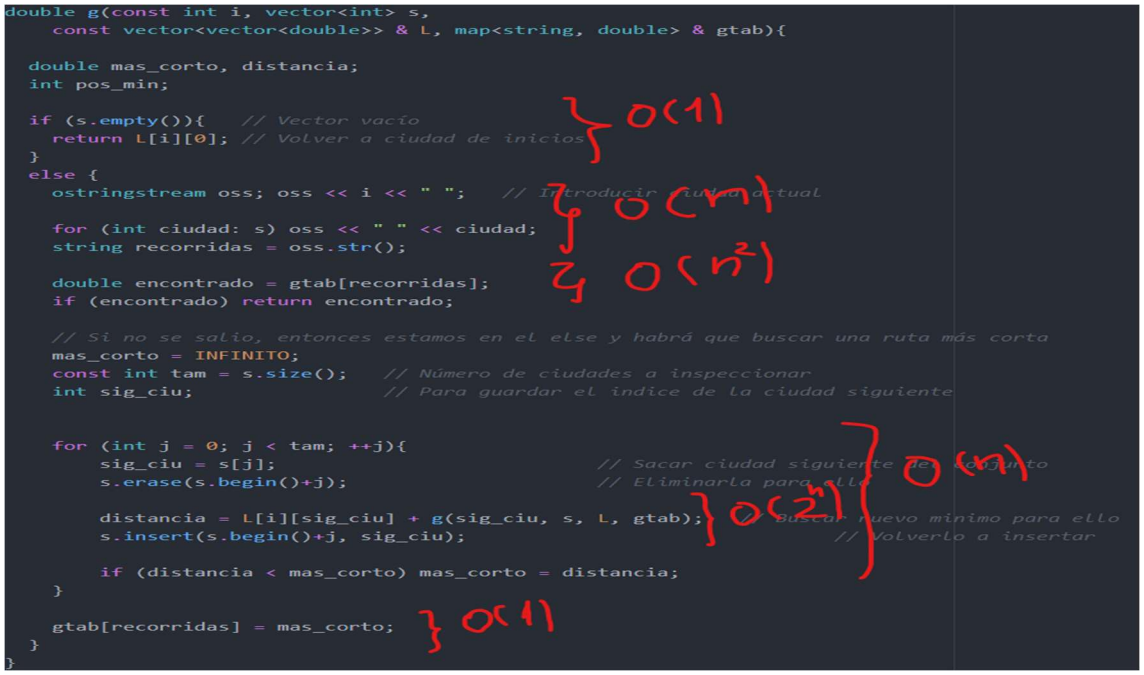
Con todo esto tenemos que, el tiempo de cálculo en el peor caso es

$$\Theta\left(2(n-1)+\sum_{k=1}^{n-2}(n-1)k\binom{n-2}{k}\right)=\Theta\left(n^2 2^n\right)$$

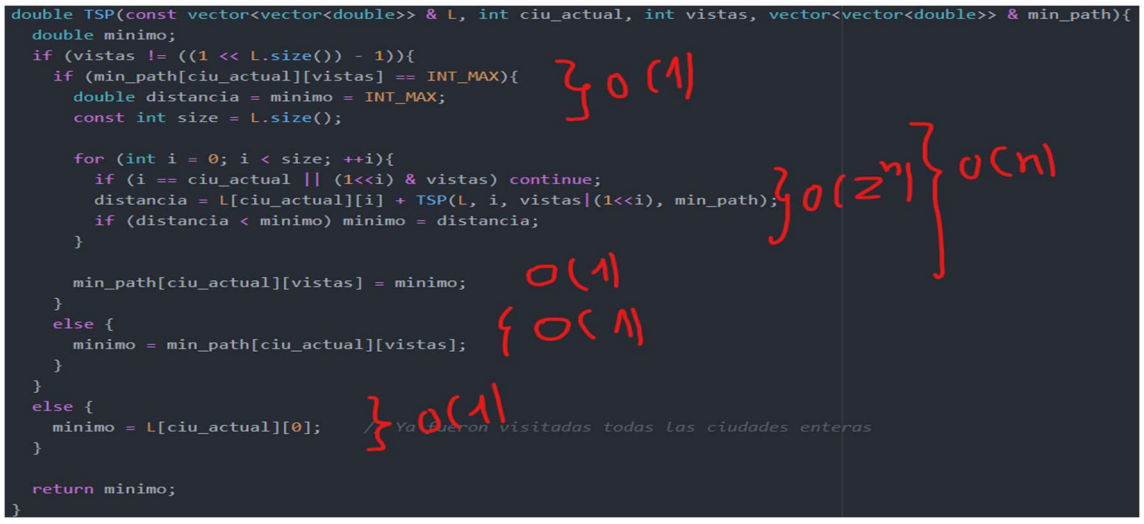
Esta eficiencia teórica no siempre se puede llegar a implementar y en nuestro caso nos encontramos con un problema relacionado con la gestión de las estructuras de datos, pues un conjunto no se puede comparar de manera directa con otro ni hacer la búsqueda, inserción de elementos O(1).

Así pues, se han llevado a cabo dos implementaciones, una en la que se usan estructuras de datos complejos (versión ineficiente) y otra en la que se lleva a cabo una codificación del conjunto para hacer las operaciones descritas anteriormente O(1) (versión eficiente). La codificación que se ha llevado a cabo fue encontrada en un artículo del cual dejaremos el link al final del guion.

La versión ineficiente tiene un orden de complejidad de O(n^3*2^n)



La versión eficiente tiene un orden de complejidad de O(n^2*2^n)

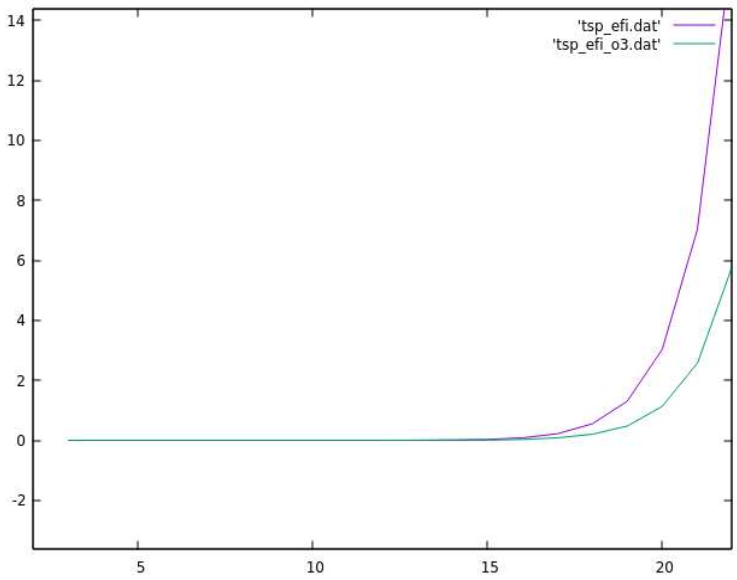


Cálculo de la eficiencia empírica

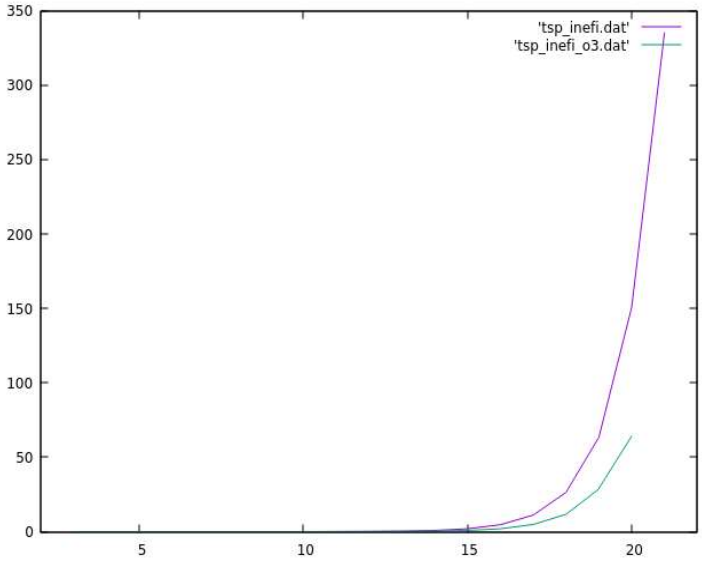
Para llevar a cabo estos cálculos vamos a llevar a cabo varias ejecuciones para diferentes tamaños, partiendo de 3 y llegando hasta 22. Usando para esto el fichero ulysses22.dat.

Tamaño	Tsp_efi (seg)	Tsp_efi_o3(seg)	Tsp_inefi(seg)	Tsp_inefi_ o3(seg)
3	1.7e-06	1.6e-06	7.79e-05	7.21e-05
4	2.3e-06	1.9e-06	9.49e-05	4.62e-05
5	4.9e-06	3.2e-06	0.000144	6.44e-05
6	1.01e-05	5.5e-06	0.0003068	0.0001704
7	2.33e-05	1.13e-05	0.0008113	0.0003957
8	0.0001172	5.03e-05	0.0028611	0.0009518
9	0.0001429	5.99e-05	0.0067731	0.0029404
10	0.0004014	0.0001311	0.0179408	0.0070537
11	0.0008452	0.0003039	0.0475733	0.0184985
12	0.0020924	0.0009809	0.125775	0.0504506
13	0.0050979	0.0018118	0.337929	0.129913
14	0.0126424	0.0040761	0.785438	0.331708
15	0.0308974	0.0103001	1.93137	0.828955
16	0.0830521	0.0290099	4.70658	1.96987
17	0.218223	0.0833021	11.144	4.81945
18	0.551045	0.204286	26.407	11.6401
19	1.30288	0.476054	63.401	28.6161
20	3.03347	1.1356	150.93	63.9061
21	7.01205	2.56695	335.021	142.84
22	16.5317	5.81844	697.51	323.97

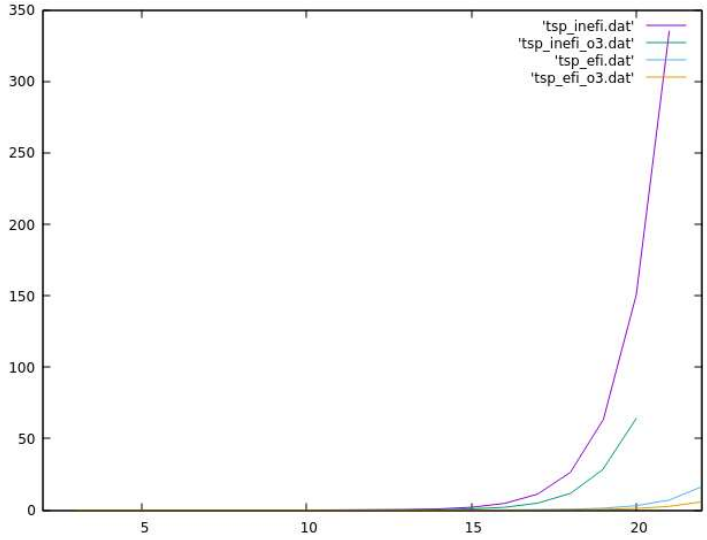
Viendo los resultados de las mediciones es evidente ver que la versión eficiente es, valga la redundancia, muchos más eficiente que la ineficiente. Es tal la diferencia que la versión más optimizada de la ineficiente, no llega a alcanzar ni de lejos a la versión sin optimizar de la eficiente. Para ver esto de manera más clara, vamos a representar y comparar estos datos gráficamente.



Podemos observar que una optimización de la versión eficiente nos da una mejora del 284%, es decir, casi 3 veces más rápido, que es una mejora realmente interesante.



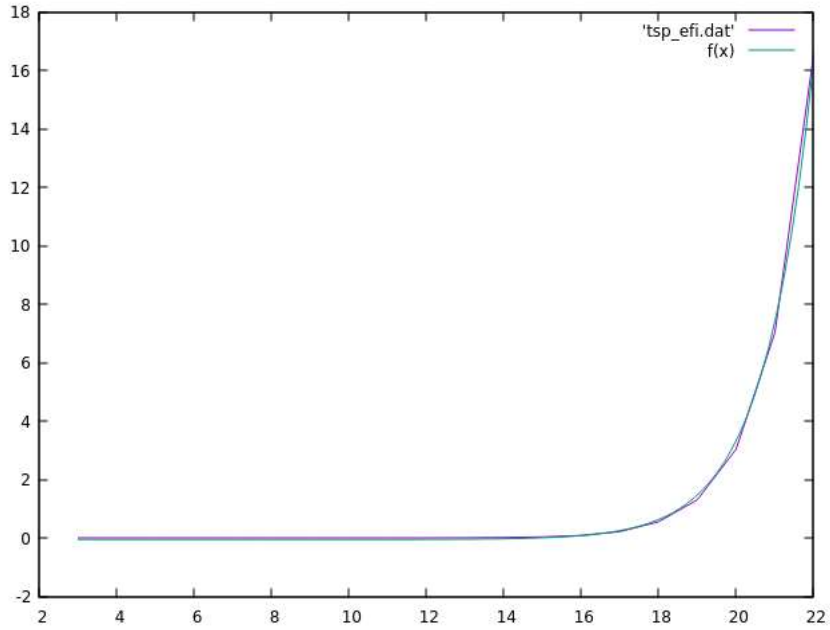
Para la versión ineficiente la mejora es menor que en el caso anterior, es de un 236%. Esta mejora no es nada despreciable, se traduce en una reducción del tiempo de ejecución de más de la mitad.



En la última gráfica podemos ver una comparación global y como dijimos antes la versión ineficiente optimizada no alcanza a la peor versión eficiente, esto es debido a que la diferencia de complejidades no reside en una constante que se pueda sumar, sino en el grado del polinomio que multiplica a la exponencial.

Cálculo de la eficiencia híbrida

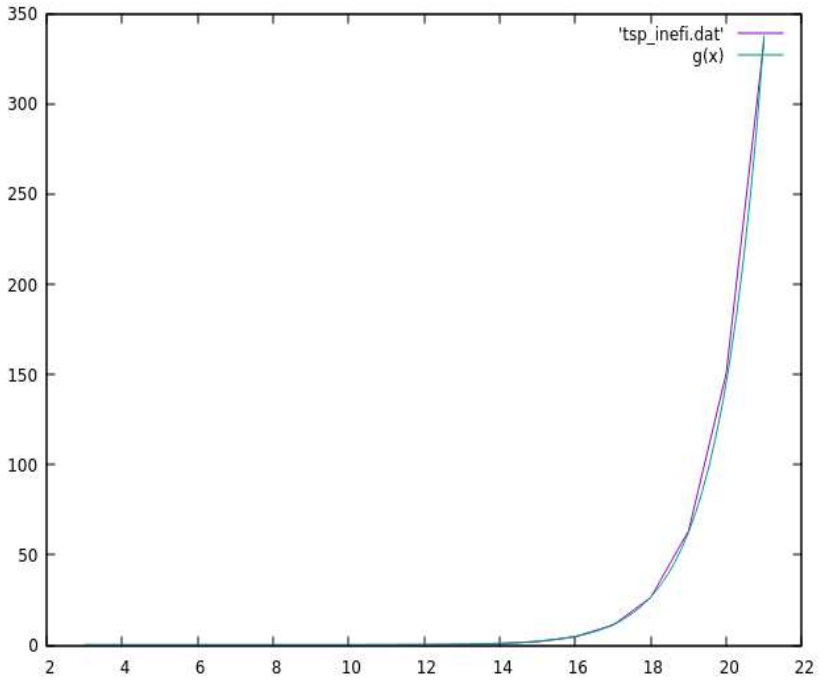
Partiendo del cálculo de las eficiencias anteriores y usando las herramientas que nos proporciona la asignatura como es ente caso Gnuplot, hallamos con un comando simple la eficiencia hibrida, además de representarla gráficamente. Para ello usamos una función f(x) = a0*x*x*2**x + a1. La eficiencia es:



Para la versión eficiente. Siendo la fórmula de la eficiencia híbrida la siguiente:

f(n) = 8.04861e-09 * (x^2) * (2^n) -0.0562579

La curva ajustada se adapta perfectamente a la curva obtenida.



Para la versión ineficiente. Siendo la fórmula de la eficiencia híbrida la siguiente:

f(n) = 1.73574e-08 * (x^3) * (2^n) + 0.218465

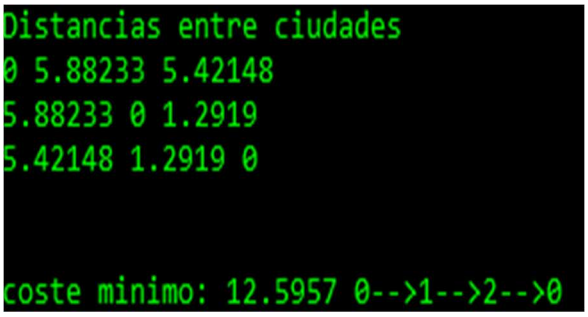
En ese caso también tenemos un ajuste bastante bueno, además del crecimiento exponencial cúbico tal y como habíamos calculado en la eficiencia teórica.

Caso de ejecución

Para poder ver un caso de ejecución de manera obvia y clara, vamos a ejecutar para tamaños mínimos, en nuestro caso 3 y 5. Para verificar el correcto funcionamiento del algoritmo, también vamos a ejecutarlo para uno de los ficheros tsp (ulysses16.tsp) y comparar así el resultado.

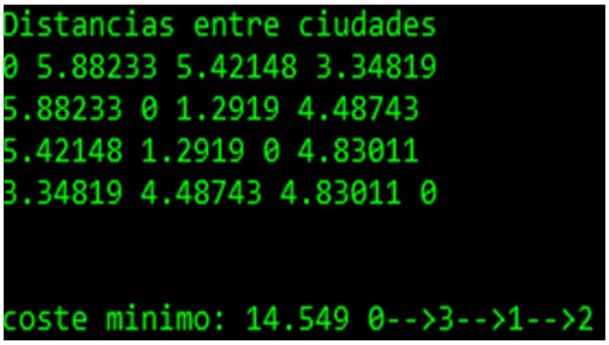
Para 3 ciudades:

Mostramos la matriz de distancias entre ciudades y vemos que el recorrido mínimo es {0,1,2} y que el costo es 12.597. Este recorrido era trivial pues solo había una alternativa ({0,2,1}). Comprobemos el costo total. [(0,1) = 5.88 + (1,2) = 1.29 + (2,0) = 5.42] el resultado es 12.59, tal y como nos dice el algoritmo.



Para 4 ciudades:

En este caso tenemos más recorridos posibles. El recorrido que obtenemos es el siguiente {0,3,1,2} con costo 14.549. Esto se obtiene de [(0,3) + (3,1) + (1,2) + (2,0)] = [3.348+4.487+1.29+5.42] = 14.549.



Otros recorridos son {0,1,2,3,0} con coste 15.34 o bien {0,1,3,2,0} con coste 20.61. En definitiva, este es el recorrido mínimo y por lo tanto óptimo.

Para finalizar vamos a buscar el camino mínimo para ulysses16.tour. La matriz de distancias es bastante grande (tiene n*n elementos, en este caso 2^8), es bastante complicado hacer cálculos a mano con estas magnitudes. El resultado dado es:

El coste coincide con el que se nos da en el opt tour con menos de 3 décimas de diferencia. Y la secuencia también (hemos sumado 1 para hacer coincidir la enumeración), como vemos a continuación.

TOUR_SECTION

1 14 13 12 7 6 15 5 11 9 10 16 3 2 4 8

-1

4. Conclusiones.

Tras haber hecho el estudio del primer ejercicio, que consistía en la búsqueda de la subsecuencia de mayor longitud (LCS), y el segundo ejercicio, que consistía en dar una solución al problema del viajante de comercio, podemos afirmar varias cosas.

Las técnicas de programación dinámica no se pueden aplicar, en cualquier caso, solo cuando se cumplen unos requisitos específicos. Cuando estos se cumplen, la programación dinámica nos garantiza la obtención de unos resultados óptimos, sin embargo, no nos dice nada sobre la eficiencia ni los tiempos de ejecución. La complejidad de la solución puede verse mejorada o empeorada En el caso del TSP, comparándolo con fuerza bruta, la eficiencia mejora muchísimo, pero sigue siendo un orden de complejidad exponencial multiplicado por un polinomio cuadrado y además de que tenemos restricciones en cuanto a la memoria (para TSP consumo de n*2^n).

Observamos además que un algoritmo no siempre puede ser implementado como sea desea, esto dependerá bastante de las herramientas de programación tal y como hemos visto en nuestras diferentes implementaciones del TSP, que, tras investigar un poco, hemos conseguido adaptar un tipo de dato para implementar el algoritmo de la manera más parecida a la ideada.

En definitiva, la programación dinámica nos da resultados óptimos y es capaz de reducir el tiempo de ejecución si se usa con el problema adecuado. Es muy útil para resolver problemas combinacionales en los que se repitan unas operaciones determinadas más de una vez, pero el precio a pagar para ello es el alto consumo de memoria, que limitará el tamaño del problema a resolver.

FIN

Recuerden lavarse las manos, evitar el contacto social, cuidarse y cuidar a los suyos

#YoMeQuedoEnCasa