Universidad de Buenos Aires

Facultad de Ingeniería

75.29 Teoría de Algoritmos

Trabajo Práctico 2

Integrantes

- Arjovsky, Tomás
- Gavrilov, Seva
- Pereira, Fernando
- Pérez Dittler, Ezequiel

Segundo cuatrimestre de 2016

Índice

1. Programación dinámica

1.1. El problema de la mochila (versión 0-1)

En este problema, tenemos una cantidad n de items, cada uno de los cuales posee un peso w y un valor v, ambos no negativos. Con estos items, se requiere llenar una mochila, la cual posee una capacidad máxima determinada.

El problema plantea encontrar los items que se incluirán en la mochila, de tal forma que la suma de todos sus pesos particulares w_i no supere la capacidad máxima de la misma y, además, la suma de todos los valores particulares v_i de los objetos que se incluyen, sea máxima.

Contrario a lo que uno puede intuir, no existe un algoritmo greedy eficiente que lo resuelva, por lo que caemos en la programación dinámica como una nueva técnica para encontrar soluciones óptimas a determinados problemas, como el de la mochila, partiendo el mismo en sub-problemas cada vez más pequeños (los cuales se resolverán mucho más fácilmente), y solapando las soluciones a dichos sub-problemas para llegar a la solución del problema original. Es decir, debemos procurar resolver sub-problemas cada vez más sencillos, los cuales se utilizarán para encontrar la solución al problema mayor.

En este informe se verá un pequeño análisis general del orden de complejidad de la solución encontrada, como así también las diferencias entre los tiempos de ejecución de dos enfoques distintos para implementar la solución (Bottom-up y Top-Down).

1.1.1. Solución

En esta versión del problema de la mochila, podemos entender que un único ítem puede pertenecer a la solución óptima o no. Siendo S nuestro conjunto solución, resumiremos la misma en encontrar el valor máximo $V = \sum v_i$ con $i \in S$ que se puede incluir en la mochila sin superar su capacidad, es decir, restringido a que $\sum w_i \leq W$.

Es trivial ver que, si tenemos un caso hipotético en el que la capacidad de la mochila es $W = 100$ y el peso de un elemento $w_i = 101$, entonces, dicho elemento i queda descartado de la solución óptima. Por otro lado, si el peso del item es menor a la capacidad de la mochila, puede o no pertenecer a la solución óptima. Esto se resuelve comparando entre el valor máximo V que se puede llegar a obtener con una solución óptima *sin* el elemento corriente, y el valor máximo V que se puede obtener con una solución óptima *incluyendo* el elemento corriente. Simplemente, el mayor de estos dos valores, es la solución óptima que se está buscando.

Suponemos que tenemos n elementos $\{1 .. n\}$. Quiero encontrar la solución óptima para n elementos y una capacidad de W. Formalizando lo expresado en los últimos dos párrafos, quiero encontrar el valor máximo V_{max} que puedo obtener cumpliendo las restricciones planteadas ($V_{max} = valor\_optimo(n, W)$).

Comenzando con el elemento n, si $w_n > W \Rightarrow valor\_optimo(n, W) = valor\_optimo(n-1, W)$, ya que no puedo incluir a mi elemento n, por lo que debo encontrar una solución óptima con los $n-1$ elementos restantes de mi conjunto, y el mismo peso máximo como restricción (no se incluyó el elemento, por lo que no se ocupó espacio).

Ahora bien, si $w_n \leq W \Rightarrow valor\_optimo(n, W) = max(valor\_optimo(n-1, W), v_n + valor\_optimo(n-1, W-w_n))$. Como se planteó anteriormente, se debe encontrar el valor máximo entre una solución sin incluir al elemento corriente, y una solución incluyendo al elemento (esto es, encontrar una combinación con los n - 1 elementos restantes, ya habiéndole sumado el valor del elemento n, y habiéndole restado a la capacidad total de la mochila el peso del elemento que incluí). Esta recurrencia es la que se plantea para resolver el problema de la mochila.

1.1.2. Complejidad

Al resolver este problema por programación dinámica, uno de los puntos importantes a destacar es el de la *memoización*. Es decir, debemos utilizar alguna estructura de datos para poder guardar los resultados de cada uno de los sub-problemas, y utilizarlos cuando se los necesite nuevamente. En efecto, nos aseguramos de calcular una solución óptima para un sub-problema en particular solo una vez, y luego tomar ese resultado cuantas veces lo necesitemos en $O(1)$.

Al tomar la recurrencia planteada en la sección anterior, vemos que a nuestros sub-problemas los estamos dividiendo en base a dos parámetros:

- Cantidad de elementos
- Peso restante en la mochila

Es decir, cada cantidad de elementos distinta y peso restante distinto es un sub-problema particular a resolver. Por ende, podemos utilizar una matriz $M[n][W]$, en la cual guardaremos el valor óptimo conseguido para cada sub-problema particular (es decir, para determinada cantidad de elementos $i \in 1 \dots n$ y determinado peso de la mochila $w \in 1 \dots W$).

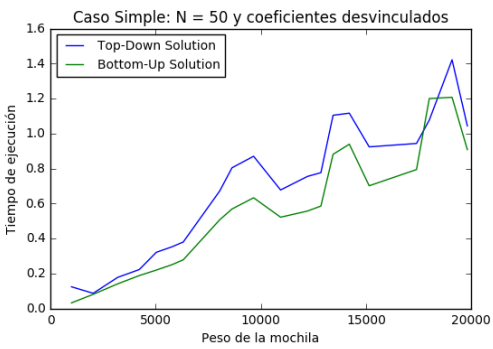
El algoritmo va resolviendo cada sub-problema particular, por lo que irá llenando la matriz planteada con el valor óptimo de cada suproblema. Si necesitamos la solución para un sub-problema ya resuelto, simplemente consultamos la matriz en O(1), por lo que el orden de nuestro algoritmo depende de la cantidad de sub-problemas a resolver, ergo, de la cantidad de elementos que posee la matriz. Por esto, podemos decir que nuestro algoritmo es $O(nW)$.

Ahora bien, es de notar que no es un orden de complejidad polinomial dependiente de n común, ya que también depende de W. Este tipo de algoritmos se los conoce como *pseudo-polinomiales*, los cuales pueden ser eficientes si los valores w_i de los ítems entrada son lo suficientemente pequeños, pero cuya complejidad aumenta muchísimo para valores muy grandes como veremos en los próximos ejemplos.

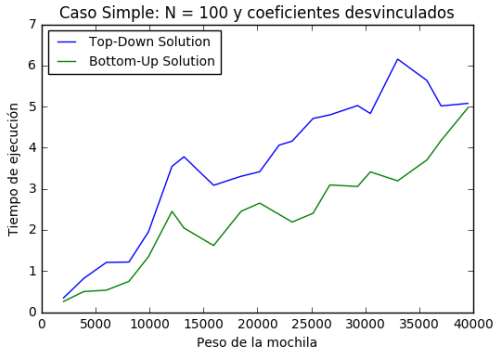
1.1.3. Tiempos de ejecución

En esta sección se encontrarán tiempos de ejecución alcanzados por las soluciones implementadas para el problema de la mochila. Vale aclarar que se implementó tanto una solución Top-Down como una solución Bottom-Up, con sus respectivas ventajas y desventajas, y justamente, uno de los enfoques de esta sección es mostrar la diferencia de tiempos entre cada una en base a las características de los parámetros de entrada (como ser el peso de la mochila o el peso de cada uno de los ítems de entrada).

Primeramente, mostramos un par de casos básicos, con 50 y 100 elementos, pesos de mochila máximo cerca de los 26000 y 52000 respectivamente, y valores y pesos de los elementos distribuidos uniformemente.

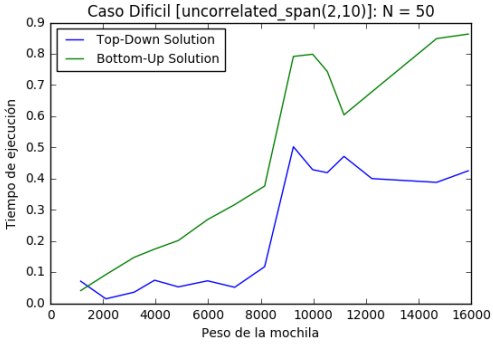


(a) $n = 50$, $W_{max} \approx 20000$

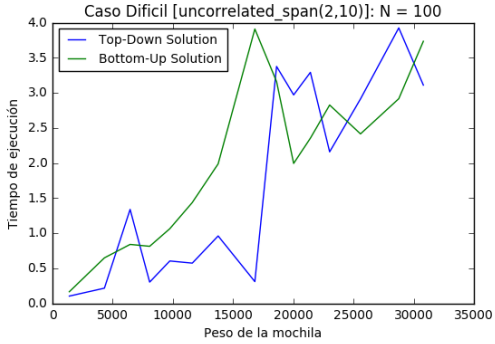


(b) $n = 100$, $W_{max} \approx 40000$

Figura 1: Instancias con pesos y valores desvinculados (uncorrelated)

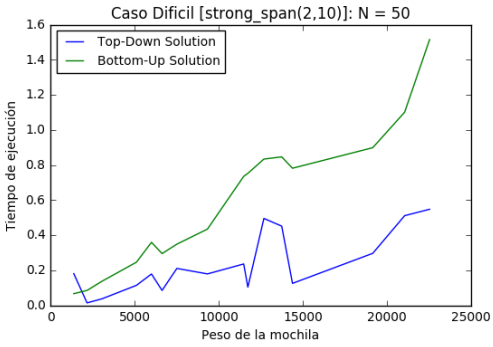


(a) $n = 50$, $W_{max} \approx 16000$

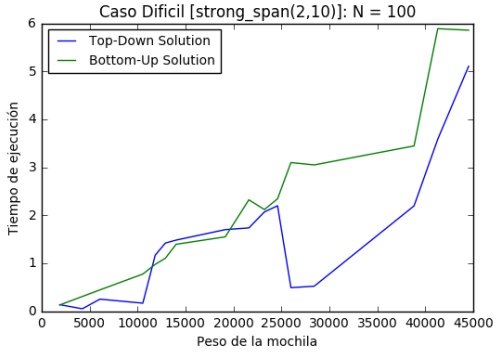


(b) $n = 100$, $W_{max} \approx 31000$

Figura 2: Instancias con pesos y valores desvinculados difícil ($uncorrelated\_span(2, 10)$)



(a) $n = 50$, $W_{max} \approx 22000$

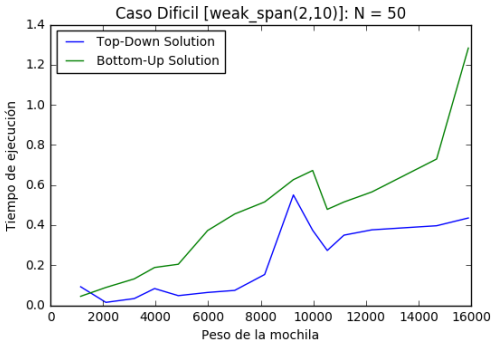


(b) $n = 100$, $W_{max} \approx 44000$

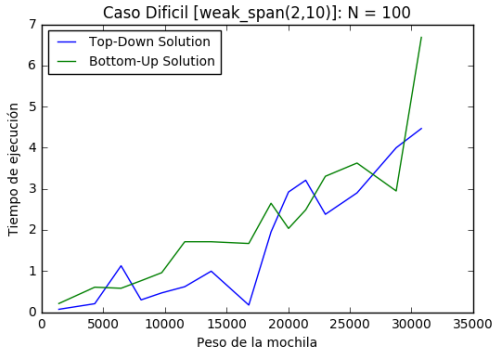
Figura 3: Instancias con pesos y valores debilmente vinculados difícil ($strongly\_correlated\_span(2, 10)$)

Como podemos ver en el caso básico inicial, los tiempos entre la solución Top-Down y Bottom-Up crecen relativamente en forma similar para esta instancia del problema. Lo que hay que notar de estos dos ejemplos es que el tiempo de ejecución del problema con $n = 100$ (para el mismo peso) es el doble del tiempo encontrado para el problema con $n = 50$. Podemos ver, por ejemplo, con un peso $W = 10000$, que en el gráfico con

50 elementos, la solución Bottom-Up tardó aproximadamente 0.5 [s] y la Top-Down 0.8 [s], mientras que para 100 elementos y mismo peso, los tiempos son aproximadamente de 1 [s] y 1.6 [s] respectivamente. Un comportamiento similar (aunque más variable, por ser instancias más difíciles), se puede ver en los tiempos de las demás ejecuciones.

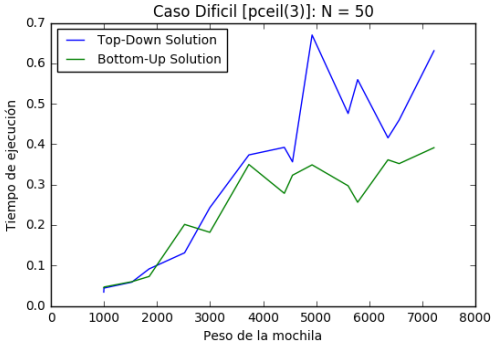


(a) $n = 50$, $W_{max} \approx 16000$

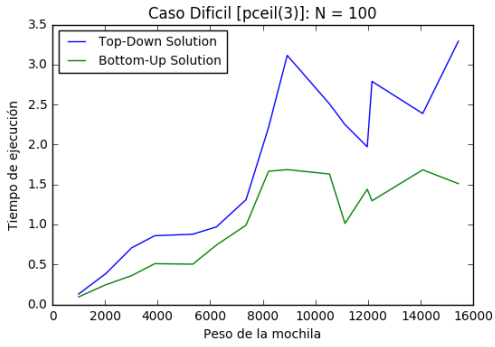


(b) $n = 100$, $W_{max} \approx 31000$

Figura 4: Instancias con pesos y valores fuertemente vinculados difícil ($weakly\_correlated\_span(2, 10)$)

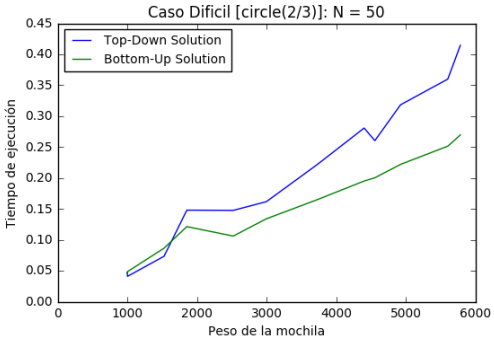


(a) $n = 50$, $W_{max} \approx 7000$

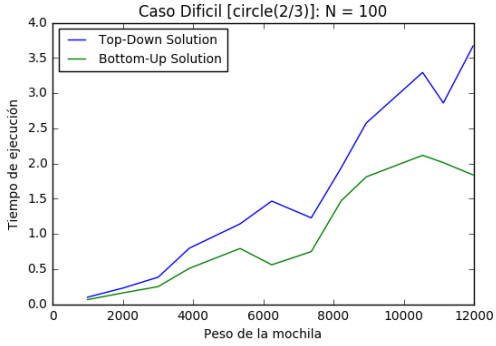


(b) $n = 100$, $W_{max} \approx 15000$

Figura 5: Instancia difícil ($pceil(3)$)



(a) $n = 50$, $W_{max} \approx 6000$



(b) $n = 100$, $W_{max} \approx 12000$

Figura 6: Instancia difícil ($circle(2/3)$)

Ahora pasamos a un caso más particular e interesante. Nos enfocamos en una instancia del problema en la que la capacidad máxima de la mochila es grande y tenemos un set de entrada con pesos w_i similares entre sí y muy altos (tal que solo uno de los elementos entra en la mochila).

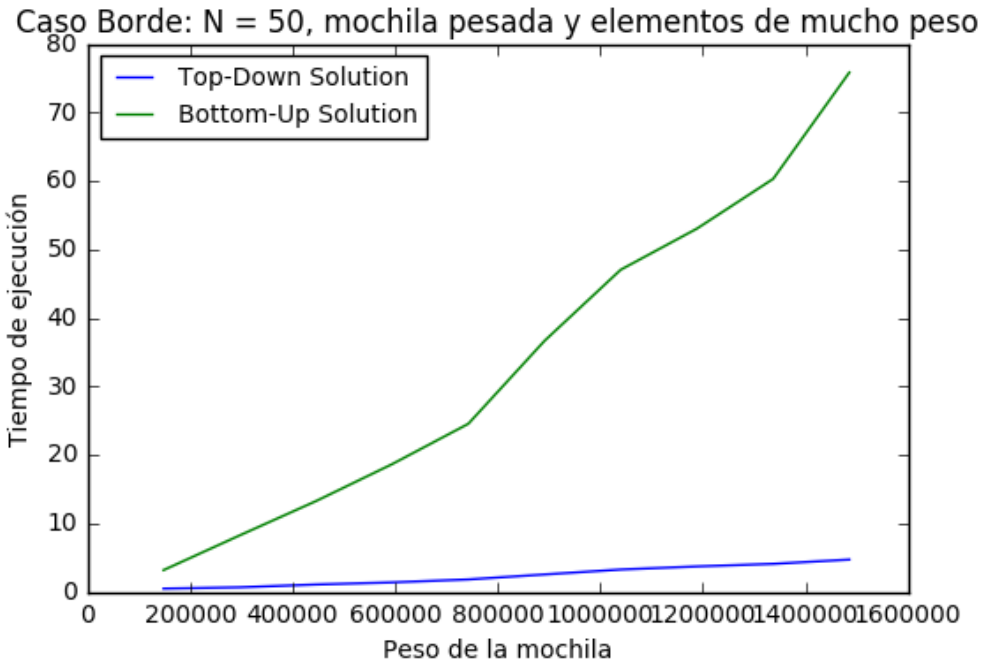


Figura 7: $n = 50$, w_i altos y similares entre sí

Una de las cosas a remarcar es la diferencia entre los tiempos de ejecución de la implementación Top-Down y la Bottom-Up en este tipo de instancias del problema.

La enorme diferencia se debe básicamente a la forma de resolver el problema que tiene cada técnica. La implementación Bottom-Up va resolviendo desde los sub-problemas más pequeños hasta llegar al problema final deseado iterativamente, obteniendo la solución óptima para absolutamente todos los sub-problemas cuyos parámetros de entrada son menores o iguales al problema original. La desventaja de esta técnica, es que está desperdiciando mucho tiempo en resolver sub-problemas que *podrían no utilizarse* para resolver el problema deseado. En otras palabras, la solución Bottom-Up llena completamente la matriz M de resultados óptimos, cuando hay muchos sub-problemas que no son necesarios.

Por otro lado, la solución Top-Down arranca desde el problema con los parámetros originales que queremos resolver, y recursivamente va partiendo el original en sub-problemas y resolviéndolos hasta obtener todos los resultados deseados. Es decir, la solución Top-Down solo se enfoca en resolver los sub-problemas estrictamente necesarios en los que se divide el problema original, sin gastar tiempo de cómputo en sub-problemas cuya solución jamás utilizaríamos. La desventaja de la solución Top-Down es el overhead que puede traer una solución recursiva, y el espacio en el stack que ésta requiere, el cual se reduce de cierta forma utilizando correctamente variables globales.

Informalmente, podemos decir que la solución Bottom-Up se toma su tiempo en resolver absolutamente todo, mientras que la solución Top-Down va al grano y resuelve lo estrictamente necesario. Es por eso que en este tipo de escenarios, la solución Top-Down puede resultar mucho más eficiente que la Bottom-Up.

1.2. El problema del viajante de comercio

El problema del viajante consiste, dado n ciudades a visitar, en encontrar un camino de costo mínimo que recorra una única vez cada ciudad, regresando finalmente al origen. Dirigirse de una ciudad a otra tiene un costo, pudiendo ser simétrico o no.

El problema del viajante de comercio se hizo muy popular en la década del 50 y 60, luego de que se presentara el artículo "Solution of a large-scale traveling-salesman problem" (Dantzig, Fulkerson, and Johnson 1954) donde resuelve el problema para 49 ciudades (una por cada estado de EEUU y Washington). El método propuesto por el artículo usa técnicas de programación lineal, aún cuando siquiera existían los programas informáticos.

En el año 1962 se presentan simultáneamente y en forma independiente los artículos "A dynamic programming approach to sequencing problems" (Held and Karp 1962) y "Dynamic programming treatment of the travelling salesman problem" (Bellman 1962). Ambos proponen el uso de la programación dinámica como técnica para encontrar la solución del problema del viajante.

1.2.1. Solución

Se define la función $D(v, S)$ la distancia mínima desde v hasta la ciudad de origen, S el conjunto de ciudades a visitar. Si el conjunto S se encuentra vacío, $D(v, S) = d_{v0}$. Se define d_{ij} como la distancia desde la ciudad i hasta la ciudad j. Para el resto de los casos, $D(v, S) = \text{mín}_{u \in S}(d_{vu} + D(u, S - \{u\}))$

$$D(v, S) = \begin{cases} c_{vv0} & \text{si } S = \varnothing \\ \text{mín}_{u \in S}[c_{vu} + D(u, S - u)] & \text{otro caso} \end{cases}$$

Un pseudocódigo para calcular la distancia del ciclo hamiltoniano mínimo es el siguiente:

```
function TSP (M, n)
  for k := 2 to n do
    C({1, k}, k) := M[1,k]
  end for

  for s := 3 to n do
    for all S in {1, 2, . . . , n}, |S| = s do
      for all k in S do
        C(S, k) = min [C(S - {k}, m) + M[m,k]]
      end for
    end for
  end for

  opt := min[C({1, 2, 3, . . . , n}, k) + M[k,1]]
  return (opt)
end
```

1.2.2. Complejidad

Al ser el problema del viajante de complejidad NP-completo, no existen algoritmos que permitan tomar decisiones para encontrar la solución. Esto implica que se deben evaluar todas las soluciones posibles y elegir

el valor de menor costo.

El número de ciclos posibles del problema del viajante se puede calcular de la siguiente manera: desde el origen restan $(n-1)$ ciudades para empezar, luego se debe elegir cualquiera de las $(n-2)$ ciudades restantes y así sucesivamente. De esta forma, multiplicando todas las cantidades se obtiene el número total de caminos posibles:

$$(n-1)! = (n-1) \cdot (n-2) \cdot (n-3) \cdots 3 \cdot 2 \cdot 1$$

Por lo tanto, el método directo implica evaluar $(n-1)!$ soluciones posibles, siendo su orden $O((n-1)!)$. En el caso particular de 10 ciudades esto significaría evaluar 362880, es decir, para un problema no excesivamente grande el número de caminos posibles aumenta considerablemente.

La cantidad de operaciones fundamentales empleadas algoritmo de Held–Karp es:

$$\left(\sum_{k=2}^{n-1} k(k-1)\binom{n-1}{k}\right) + (n-1) = (n-1)(n-2)2^{n-3} + (n-1)$$

Por lo tanto, su orden temporal es $O(n^2 2^n)$. Para el caso particular de $n = 10$ el número de soluciones se reduce de 362880 a 102400.

Si se asigna una unidad de espacio al número $D(v, S)$ la cantidad de espacio requerida es:

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

Por lo tanto, su orden espacial es $O(n2^n)$.

1.2.3. Tiempos de ejecución y espacio consumido

El argoritmo implementado es de tipo bottom-up, es decir, se van calculando las distancias desde conjuntos pequeños hasta llegar al conjunto de tamaño $n-1$.

Los datos utilizados para 15, 17 y 21 ciudades son los recopilados por John Burkardt. El resto de los datos fueron generados aleatoriamente y se pueden encontrar en el repositorio con el prefijo *ex*.

En la Figura 8 se puede visualizar el tiempo de ejecución para 4 ciudades en adelante. Se muestran los valores hasta el conjunto de 21 ciudades dado que no sólo demora cada vez más la ejecución del algoritmo si no que también se eleva el consumo de espacio de memoria, como se verá más adelante, alcanzando el valor de memoria disponible.

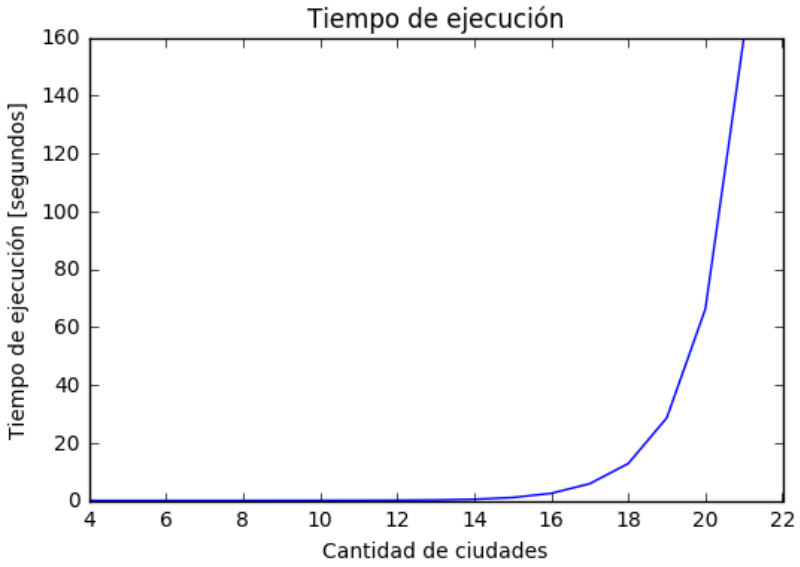


Figura 8: Tiempo de ejecución del problema del viajante

El espacio consumido que se representa en la Figura 9 se refiere a la cantidad de valores almacenados de los costos para ir desde el origen a cada conjunto. Es decir, la cantidad de registros almacenados para cada conjunto. Para obtener la información del espacio ocupado en una unidad en particular, por ejemplo en Megabytes, basta con determinar el espacio que ocupa cada registro en esa unidad y luego multiplicar por los valores dados.

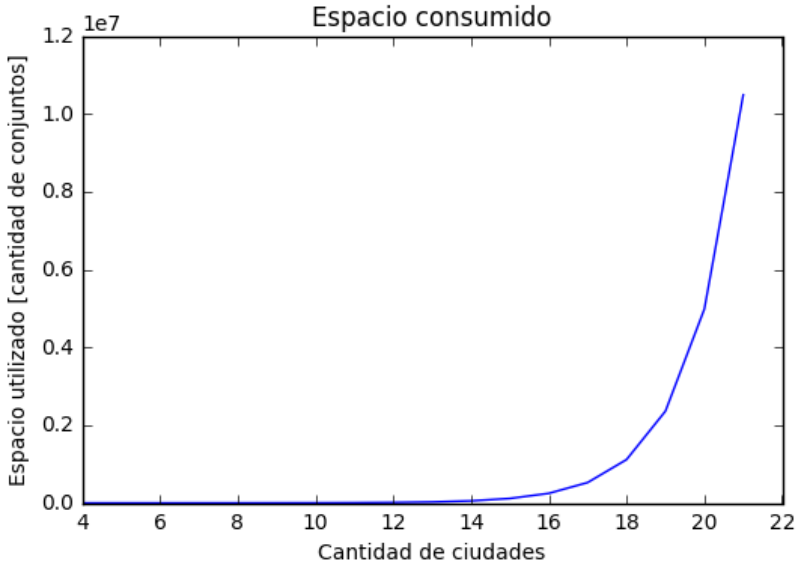


Figura 9: Espacio utilizado por el problema del viajante

En el Cuadro 1 se realiza una comparación del espacio consumido sin utilizar programación dinámica, utilizando programación dinámica y el ahorro logrado.

Ciudades	Sin programación dinámica	Con programación dinámica	Ahorro
4	6	12	-6
5	24	32	-8
6	120	80	40
7	720	192	528
8	5040	448	4592
9	40320	1024	39296
10	362880	2304	360576
11	3628800	5120	3623680
12	39916800	11264	39905536
13	479001600	24576	478977024
14	6227020800	53248	6226967552
15	87178291200	114688	87178176512
16	1307674368000	245760	1307674122240
17	20922789888000	524288	20922789363712
18	355687428096000	1114112	355687426981888
19	6402373705728000	2359296	6402373703368704
20	121645100408832000	4980736	121645100403851000
21	2432902008176640000	10485760	2432902008166150000

Cuadro 1: Espacio utilizado por el problema del viajante

Para los casos de 4 y 5 ciudades no se logra un ahorro, pero en ambos enfoques el orden de magnitud del espacio consumido es el mismo, por lo que no es un problema a tener en cuenta.

A partir de las 7 ciudades en adelante, el orden de magnitud de ambos comienza a distanciarse, resultando un ahorro importante con la programación dinámica.

A efecto ejemplificador del espacio de memoria real ocupado, se toma una unidad aproximada del tamaño de un registro de 256 bytes, sin considerar el espacio adicional que consume la estructura de datos donde se almacenan los registros, pudiendo ser depreciable.

En el Cuadro 2 se detalla el espacio consumido según la cantidad de ciudades del problema.

Ciudades	Espacio ocupado (en Megabytes)
4	0,003
5	0,008
6	0,02
7	0,05
8	0,11
9	0,25
10	0,56
11	1,25
12	2,75
13	6
14	13
15	28
16	60
17	128
18	272
19	576
20	1216
21	2560
22	5376
23	11264
24	23552
25	49152
26	102400

Cuadro 2: Espacio utilizado por el problema del viajante en Megabytes

A partir de las 22 ciudades el consumo de memoria supera la memoria RAM disponible en la PC utilizada para correr el algoritmo.

Una propuesta para superar el problema del límite de memoria RAM es utilizar la memoria en disco. Aunque esto implicaría un aumento en los tiempos de ejecución debido a las operaciones de lectura y escritura de archivos.

2. Flujo de redes

2.1. Objetivo

Esta sección del trabajo tiene como objetivo la resolución del problema de Selección de Proyectos mediante su modelado como una red de flujo y la aplicación del algoritmo de Ford Fulkerson para maximizar la ganancia esperada.

2.2. Introducción Teórica

Los problemas de flujo de redes parten de un grafo G = (V,E) con las siguientes características:

- G es dirigido.
- Hay un nodo s (fuente/origen) que solo tiene aristas salientes.
- Hay un nodo t (sumidero/destino) que solo tiene aristas entrantes.
- Cada arista $e \in E$ tiene una *capacidad* a la que denominaremos c_e.

Por cada arista puede pasar una cantidad determinada de flujo. Esta es una función $f(e)$ aplicable a las aristas, con dos condiciones:

- Capacidad: $f(e) \leq c_e$.
- Conservación: el flujo entrante total (entre todas las aristas) debe ser el mismo que el total saliente: $\sum_{e \text{in} v} f(e) = \sum_{e \text{out} v} f(e)$.

Este tipo de problemas, si bien tiene aplicaciones directas como redes de transporte de vehículos, de flujo, de comunicaciones y muchas otras directas, también puede utilizarse como modelo para problemas complejos que no son intuitivamente comparables. Un problema de este tipo es el tratado en este trabajo, que es el de *selección de proyectos*.

Por otro lado, también es de interés definir un *corte* (A, B) de la red como una partición de G en dos conjuntos tales que $s \in A$ y $t \in B$. La *capacidad de ese corte* es la suma de las capacidades que lo cruzan: $c(A, B) = \sum_{e \text{out} A} c_e$.

En general, este modelo deviene en problemas de optimización, que son el de *maximización de flujo* y el de *búsqueda de mínimo corte*. En el primer caso, se busca maximizar el *valor del flujo* $v(f)$ definido como el flujo total saliente de s. En el segundo caso se busca un corte tal que su capacidad sea la mínima.

2.2.1. Max-Flow, Min-Cut

El teorema de máximo flujo, mínimo corte muestra que el máximo flujo corresponde a la capacidad del mínimo corte. Este teorema se basa en la importante propiedad de que la capacidad de cualquier corte en una red es cota superior del valor del flujo.

Una demostración común, como la encontrada en el capítulo 7 del Kleinberg-Tardos **(CITAR)** acude a mostrar la situación final del algoritmo de Ford Fulkerson para maximización del flujo.

2.2.2. Algoritmo de Ford Fulkerson

Este algoritmo, que será el utilizado en el trabajo, trabaja con el *Grafo Residual* G_f de la red, con las siguientes características:

- Una arista $e \in E(G)$ tal que $f(e) < c_e$ produce una *arista residual hacia adelante* (foreward edge) $e' \in E(G_f)$ tal que $c_{e'} = c_e - f(e)$. Esto puede verse como lo que aún puede aumentarse de flujo en esa arista.
- Una arista $e \in E(G)$ tal que $f(e) > 0$ produce una *arista residual hacia atrás* (backward edge) $e' \in E(G_f)$ tal que $c_{e'} = f(e)$. Esto puede verse como "lo que puede quitarse de flujo para ser asignado en otro lado".

La estrategia del algoritmo para maximizar el flujo se basa en buscar caminos de s a t en el grafo residual para aumentar el flujo en un cuello de botella (bottleneck) b correspondiente a la mínima capacidad de ese camino. Finalizará cuando ya no haya caminos s-t en G_f.

La importancia reside en que en cada paso se aumenta $v(f)$ en b (que es mayor a 0), y sabemos que $v(f)$ está acotado, con lo cual **sabemos que el algoritmo termina**. Además, al terminar, hay un corte natural (A, B) donde A es el conjunto de todos los nodos alcanzables por s y B todo el resto. Como la ejecución termina cuando ya no hay caminos s-t, sabemos que la capacidad residual de las aristas entre A y B es nula. Entonces, la capacidad de ese corte está saturada hacia adelante (si no, habría forward edges) y no hay flujo de B hacia A (si no, habría backward edges de A a B). De este modo, nos aseguramos de que ese corte está saturado, mostrando que se realiza la desigualdad de capacidad y valor de flujo, y **demostrando Max-Flow Min-Cut** y simultáneamente que **el algoritmo efectivamente obtiene el Máximo flujo**.

Una vez finalizada la ejecución de Ford-Fulkerson, lo único necesario para encontrar el corte mínimo es hacer una búsqueda como BFS para encontrar los nodos alcanzables por t.

2.3. Problema de Selección de Proyectos

2.3.1. Problema

El problema en análisis tiene como objetivo maximizar las ganancias del Ing. F.B. Su empresa tiene:

- Un conjunto $P = \{P_1, P_2, \ldots, P_m\}$ de proyectos posibles para tomar. Cada proyecto P_i llevado a cabo provee una ganancia g_i a la empresa.
- Un conjunto $A = \{A_1, A_2, \ldots, A_n\}$ de áreas de investigación posibles a tomar. Cada área A_k investigada conlleva un costo a_k de inversión para la empresa.
- Cada proyecto requiere que se haya investigado un cierto conjunto de áreas $R_i \subseteq A$ para poder ser ejecutado.

El objetivo es elegir los proyectos (y por lo tanto también las áreas de investigación) adecuados para maximizar la ganancia del Ingeniero.

2.3.2. Modelo de Red

Para este modelo creamos la siguiente red de flujo:

- El nodo s simbolizará las ganancias. Por lo tanto, s apuntará a los proyectos. Como en redes de flujo no se utilizan los valores de los nodos, utilizaremos las capacidades de estas aristas, dándoles el valor g_i.
- El nodo t simbolizará las inversiones, con lo cual apuntará a las áreas de investigación. En este caso, las capacidades serán los costos a_k.
- Entre requerimientos y proyectos habrá aristas de dependencia: una arista (i, j) donde i es un proyecto y j es un requerimiento simboliza que i necesita de j para poder llevarse a cabo. Las capacidades de aquellas serán tratadas más adelante.

Esto da grafos como el siguiente: **(DIBUJAR IMAGEN)**

La idea de este modelo es utilizar el algoritmo de Ford-Fulkerson para obtener un corte mínimo de forma tal que el conjunto alcanzable por s sea el conjunto de proyectos a tomar, con sus áreas correspondientes.

2.3.3. Compatibilidad

Si bien este sentido de dependencias es poco intuitivo (uno tendería a decir que (i, j) simboliza que j depende de i), este modelo tiene un beneficio grande, que consiste en la facilidad de implementar la satisfacción de dependencias.

Para asegurarnos de que el corte final (A, B) sea compatible necesitamos que no haya ninguna arista saliente de A, ya que de haberla, significaría que hemos seleccionado un proyecto sin pagar el costo de investigar una de sus dependencias. Con este fin haremos que las aristas de dependencia tengan **capacidad infinita**.

Sabemos que el flujo está acotado aunque sea por la suma de las capacidades salientes de s:

$$v(f) \leq \sum_{i \text{outs}} c_e = C \tag{1}$$

Esto se debe a que $(\{s\}, G - \{s\})$ es un corte válido, y como tal, su capacidad acota al flujo máximo. Como el corte encontrado por el algoritmo es el de mínima capacidad, $c(A, B)$ estará acotada superiormente por C. De este modo entonces nos aseguramos que ninguna de esas aristas de capacidad infinita puede cruzar el corte, ya que si así fuera, la capacidad $c(A, B)$ superaría la cota que hallamos previamente.

2.3.4. Optimalidad

Para comprobar que el córte mínimo es efectivamente el que buscamos, necesitamos ver qué valor tiene la capacidad de un corte. Con este fin resulta útil recordar que queremos maximizar la ganancia $G(A)$, que puede ser escrita de la siguiente manera:

$$G(A) = \sum_{i/P_i \in A} g_i - \sum_{k/A_k \in A} a_k \tag{2}$$

Por otro lado, a la capacidad del corte contribuyen las aristas de las áreas investigadas y las de proyectos no tomados (**IMAGEN**), con lo cual puede ser escrita de la siguiente:

$$c(A, B) = \sum_{k/A_k \in A} a_k + \sum_{i/P_i \notin A} g_i = \sum_{k/A_k \in A} a_k + C - \sum_{i/P_i \in A} g_i = C - G(A) \tag{3}$$

De este modo, ya que C es constante, vemos que minimizar el corte es igual a maximizar la ganancia, con lo cual este modelo resuelve el problema planteado.

2.3.5. Detalles de implementación

En términos generales, la solución fue implementada con una clase ProjectSelection correspondiente al problema, que utiliza a una clase Flow, que modela la red de flujo y es capaz de aplicar el algoritmo de Ford Fulkerson.

Ya que el grafo de la red y el residual comparten una gran cantidad de información, en lugar de tratarlos por separado utilizamos únicamente al grafo principal, añadiendo las backward edges con capacidad 0 al agregar cada arista.

El uso del grafo residual se da sobre todo en la búsqueda de caminos a aumentar y la capacidad residual se calcula en el momento en la función $e_transitable$ para ver si un camino residual puede pasar por esa arista dado el flujo en ese momento:

```
def e_transitable(g, e, flow):
    return (e.is_backwards and e.capacity > 0) \
        or ((not e.is_backwards) and flow[e] < e.capacity)
```

Como fue explicado para arriba, si es una *foreward edge*, la capacidad residual es positiva cuando la capacidad no está saturada y por lo tanto puede pasar más flujo por allí. Para una *backward edge* , la capacidad residual es positiva cuando hay flujo en el sentido contrario que puede ser reducido para usar en otro lado. Estos son los casos en donde un camino *s-t* puede pasar por allí.

La lógica principal del algoritmo de Ford_Fulkerson está implementada del siguiente modo:

```
def get_max_flow(g, source, target):
    flow = g.get_empty_flow()
    path = g.flow_path(source, target, flow)

    while path != None:
        b = g.bottleneck(path, flow)
        for edge in path:
            flow[edge] -= b if edge.is_backwards else -b
        path = g.flow_path(source, target, flow)

    return flow
```

Como las capacidades residuales son calculadas en el momento y en base al flujo, no hace falta "actualizar" al grafo residual después de un aumento.

La **búsqueda de caminos** entre s y t fue implementada con **DFS**, ya que teniendo en cuenta la estructura del grafo, un BFS produciría siempre recorrer todos los vértices: como se recorre por distancias al origen, se recorrerán siempre primero todos los proyecots (d = 1), luego todas las áreas de investigación (d=3) y recién al final se llegará a t (d = 3). Con DFS el peor caso será el de recorrer todos los vértices, mientras que es posible que llegue en visitando un proyecto y un área.

El **Mínimo corte** también fue implementado con un DFS recursivo, pero esto ocurrió por comodidad y no por eficiencia, ya que tanto en ese caso como en BFS buscar el mínimo corte implica recorrer todos los nodos alcanzables por s.

2.3.6. Complejidad

Para calcular la complejidad es necesario acotar la cantidad de iteraciones que tendrá el algoritmo. Como sabemos que el flujo aumentará en una cantidad positiva en cada iteración y sabemos que $v(f) < C$, entonces a lo sumo habrá C iteraciones.

Cada una de las iteraciones, a su vez, consiste principalmente en la búsqueda de un camino, cuyo peor caso recorre todo los nodos en orden $O(|V| + |E|) = O(n + m + 2r)$, siendo $r = \sum_{i=1}^{m} |Ri|$ (la cantidad de aristas). En este orden se ve la cantidad de aristas dos veces por la generación de backward edges, aunque esto no afecta realmente al orden.

Buscar el bottleneck se hace dentro del camino encontrado, que es un camino simple, con lo cual es como máximo $O(n + m)$. Lo mismo para el aumento del camino.

Finalmente, todo el orden total es de $O(C(n + m + r))$. Esto significa que es lineal en el n, m y r siempre y cuando se mantenga C.

Sin embargo, no sería lógico decir aquello, ya que C no es independiente de n o de m. Para obtener esa cota del valor del flujo puede tomarse el corte tanto en $(\{s\}, G - \{s\})$ o en $(G - \{t\}, \{t\})$ y la mínima capacidad entre ambos cortes sería una cota. Por eso podemos expresar la parte de C como $\sum_{i=1}^{m} g_i + \sum_{k=1}^{n} a_k$, que también depende de m y n.

Para simplificar las cuentas (con pérdida de generalidad) suponemos que $g_i = a_k = 1 \forall i \forall k$. En este caso relajado podríamos fijar para el problema $\Omega((m + n)(m + n + r))$, lo cual es **cuadrático en m y n, y lineal en r (manteniendo las otras entradas fijas)**. El O real será más grande y dependerá de las capcacidades (ganancias y pérdidas). Sin embargo, si suponemos que las ganancias están acotadas por g_max y las inversiones por a_max, lo cual es razonable, los cálculos llevan al mismo resultado, con diferencia de constantes, que no afectan al orden.

Esta fuerte dependencia de los costos se debe a que no se implementó un mecanismo inteligente de elección de caminos, permitiendo que el bottleneck pueda ser tan pequeño como 1. Teniendo en cuenta que lo ideal sería aumentar primero los paths de mayor bottleneck, hay algoritmos mejorados como el *Scaling Max Flow*, cuya complejidad resulta de $O((n + m + r)log_2(C))$, lo cual baja el orden de los cuadráticos a $O(xlogx)$ según mostrado en la sección 7.3 del libro (Kleinberg and Tardos 2005).

2.3.7. Algunos casos

En los tests provistos pueden verse algunos casos interesantes.

2.3.7.1. Sin presupuesto.

El *test_no_selection* muestra que cuando se gastaría más dinero en cualquier investigación que la que se obtendría con sus respectivos proyectos, el algoritmo elige no tomar ningún área ni proyecto.

(IMAGEN)

Teóricamente esto es esperable, dado que el corte por s, que tiene las ganancias, será más pequeño que el corte por t.

2.3.7.2. Aumento de m

2.3.7.3. Aumento de n

2.3.7.4. Aumento de r

Referencias

Bellman, Richard. 1962. "Dynamic Programming Treatment of the Travelling Salesman Problem." *Journal of Association for Computing Machinery.*

Dantzig, George, Delbert Fulkerson, and Selmer Johnson. 1954. "Solution of a Large-Scale Traveling Salesman Problem." *Operations Research 2.*

Held, Michael, and Richard M. Karp. 1962. "A Dynamic Programming Approach to Sequencing Problems." *Journal for the Society for Industrial and Applied Mathematics.*

Kleinberg, Jon, and Éva Tardos. 2005. *Algorithm Design.*