Universidad de Buenos Aires

Facultad de Ingeniería

75.29 Teoría de Algoritmos

Trabajo Práctico 3

Integrantes

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

Segundo cuatrimestre de 2016

Índice

1. Clases de Complejidad

1.1. Objetivo

El objetivo de esta sección del trabajo es analizar 4 problemas, sus posibles soluciones y su clasificación según clase de complejidad.

1.2. Introducción Teórica

La clasificación de clases de complejidad de los problemas surge de la necesidad de comparar problemas para los cuales no se conoce una solución eficiente (polinomial), pero no se ha demostrado que no la tienen.

1.2.1. Reducciones

Para comparar la dificultad de dos problemas X e Y tomamos como criterio la posibilidad de usar a uno para resolver al otro. Si se tiene un algoritmo que resuelve X, y en base a ese resultado podemos obtener una solución para el problema Y en tiempo polinómico, entonces decimos que:

- "Y es polinomialmente reducible a X" ($Y \leqslant_p X$)
- "X es tan difícil como Y", ya que resolver X asegura que podemos resolver Y, pero no se cumple la inversa. No nos referimos estrictamente al tiempo de ejecución.

Se dice por otro lado que dos problemas son *polinómicamente equivalente* ($Y \equiv_p X$) cuando cualquiera puede usarse para resolver al otro: $Y \leqslant_p X$ y $X \leqslant_p Y$.

1.2.2. Problemas de decisión

Identificamos a un problema de decisión por las siguientes componentes:

- Entrada: string $s \in S$ (siendo S el espacio de strings posibles).
- Salida: *si* o *no* (string aceptada o rechazada).

Definimos entonces a un problema X como el conjunto de strings que acepta.

Definimos a su vez a un algoritmo A que resuelve a un problema X como una función que toma inputs y devuelve si el problema acepta o no a esa string. Formalmente:

$$A(s) = si \iff s \in X \tag{1}$$

REVISAR

Nos son particularmente interesantes estos problemas porque la reducción de un problema de decisión X a otro Y consiste en transformar en tiempo polinómico la entrada de X en una entrada de Y de forma tal que la salida de Y sea la esperada por X.

1.2.3. Certificación

Llamamos *certificado* a una cadena t que aporta información sobre si una cadena de input s será aceptada para un problema X. Por ejemplo, para un problema de decisión como "verificar si en un grafo hay ciclos", un posible certificado es un conjunto de vértices que supuestamente forman un ciclo. Es algo similar a una solución propuesta.

Dado un problema X, una entrada s y un certificado t, llamamos *certificador* a una función B que toma certificados y evidencia con ellos que $s \in X$. Para el ejemplo anterior, un certificador tomaría el conjunto de vértices t y la cadena s (que serían los vértices del grafo) y verificaría que t es efectivamente un ciclo. Formalmente:

$$B(s,t) = si \iff t \text{ demuestra que } s \in X \tag{2}$$

Es importante observar que el certificador no analiza la cadena en sí, sino que evalúa el certificado. Por lo tanto, su función es en realidad comprobar soluciones propuestas al problema, lo cual no necesariamente resuelve el problema en sí mismo. Por este motivo, muchas veces es más sencilla y rápida la acción de un certificador que la de un algoritmo de resolución de un problema.

Llamamos *certificador eficiente* a uno que certifica en tiempo polinómico.

1.2.4. P y NP

Son de nuestro interés dos clases de complejidad:

- P: clase que contiene a todos los problemas para los cuales existen algoritmos que los resuelven en tiempo polinómico. Si X es un problema, A un algoritmo y p es una función polinómica:

$$P = \{X / \exists A \in O(p(|s|)) \text{ que lo resuelve}\} \tag{3}$$

- NP: clase que contiene a todos los problemas certificables en tiempo polinómico. Si X es un problema y B un certificador:

$$NP = \{X / \exists B \in O(p(|s|)) \text{ que lo certifica}\} \tag{4}$$

Inmediatamente podemos observar que $P \subseteq NP$, ya que si un problema puede resolverse en tiempo polinómico, entonces para certificar soluciones propuestas puede usarse el mismo algoritmo polinómico de resolución, ignorando el certificado.

Adicionalmente, de la definición de reducciones podemos observar que:

- Si $Y \leqslant_p X$ y X es resoluble en tiempo polinómico, entonces Y también. $X \in P \Rightarrow Y \in P$.
- Vale la recíproca: $Y \leqslant_p X \wedge Y \notin P \Rightarrow X \notin P$.

1.2.5. NP-Completo

Además, existe un subconjunto de NP que cumplen con la característica de ser más difíciles que cualquier otro prolema en NP. A esta subclase la denominamos NP-completo. Visto de otro modo, que un problema pertenezca a aquella clase asegura que si se tiene un algoritmo para resolverlo, entonces es posible utilizarlo para transformarlo en tiempo polinómico a cualquier problema de NP.

Entonces un problema X perteneciente a esta clase cumple con dos características:

1. $X \in NP$
2. $\forall Y \in NP : Y \leqslant_p X$

Es importante que se cumplan ambas, ya que los problemas que cumplen la segunda condición son llamados NP-hard independientemente de si son o no NP.

Quizás se puede plantear la "situación de hoy en día"

1.3. Determinar la clase de complejidad de un problema.

Para muchas aplicaciones es de interés determinar si un problema pertenece a alguna de las tres clases anteriores. En cada caso, el método utilizado será:

- P: encontrar una solución polinomial.
- NP: encontrar un certificador polinomial.
- NP-completo: reducir un problema NP-completo conocido al problema en cuestión. Además habrá que probar independientemente que es NP.

A continuación se presentan algunos problemas conocidos que serán de utilidad.

1.3.1. SAT y 3-SAT

El problema SAT (satisfacibilidad booleana) consiste en los siguientes elementos:

- Un conjunto $X = \{x_1, x_2, , x_n\}$ de variables booleanas.
- k condiciones C_i a satisfacer, cada una con l términos unidos por disyunciones ($\vee$).
- Cada término t_i en una condición es una de las variables booleanas de X o su negación.

El problema se trata de decidir si existe una asignación de las variables $\nu : X \to \{0, 1\}$ (mapeo concreto de cada variable a 0 o 1) que satisfaga simultáneamente todas las condiciones. Es decir, ν tal que $\bigwedge_{i=1}^{k} C_i = 1$.

El 3-SAT es un caso particular de este, donde $|C_i| = 3 \quad \forall i$. Es demostrable (**CITAR**) que es polinómicamente equivalente a SAT.

La importancia de este problema reside en que el Teorema de Cook (**CITAR?**) demuestra que es NP-Completo, modelando con una tabla la computación completa de cualquier máquina de Turing y expresándo los valores de la tabla y sus relaciones con variables booleanas. (**MEJORAR**). De este modo se reduce cualquier Máquina de Turing a un problema SAT.

Mostrar que SAT y 3-SAT son NP, por otro lado es sencillo. Un certificado es una asignación propuesta. Para verificarlo, solo será necesario ver para cada condición que esa asignación la hace verdadera. La certificación, por lo tanto, es lineal en el tamaño de la entrada, que es la cantidad total de términos ($\sum_{i=1}^{k} |C_i|$).

1.3.2. Independent Set (IS)

Un conjunto de vértices de un grafo $G = (V, E)$ es *independiente* si no contiene ningún par de vértices adyascentes.

Mientras que encontrar un IS de tamaño pequeño es trivial (cualquier vértice es un IS de tamaño 1), encontrar instancias grandes es costoso. Por este motivo, el problema de decisión asociado consiste en decidir si hay un IS de tamaño al menos k para un k dado.

1.3.3. Vertex Cover (VC)

Se dice que un conjunto de vértices *cubre* al grafo cuando todas las aristas tienen al menos uno de sus extremos en el conjunto.

Al revés del problema anterior, encontrar un VC de tamaño grande es trivial, ya que por ejemplo el conjunto V completo es un VC. Sin embargo, encontrar instancias pequeñas no es sencillo. Por lo tanto, el problema de decisión asociado consiste en decidir si existe un VC de tamaño a lo sumo k para un k dado.

1.3.4. Relación entre los problemas

Teorema: VC $\equiv_p$ IS

Demostración: queremos probar que si S es un IS en G, entonces $V - S$ es un VC en G.

1. S es IS $\Rightarrow V - S$ es VC:

   Como dijimos antes, V es un VC. Si quitamos uno a uno elementos de S a V, sabemos que en ningún caso se quitarán ambos extremos de una arista. Si así fuera, dos de los elementos quitados estarían unidos y S no sería un IS.

2. $V - S$ es VC $\Rightarrow S$ es IS:

   Por contradicción: Si S no fuera un IS, entonces podríamos afirmar que existen dos vértices $v_1, v_2 \in S$ tal que v_1 y v_2 están conectados. Entonces, $V - S$ no contendrá la unión entre esos dos vértices y por lo tanto no será un VC, contradiciendo la hipótesis.

Entonces se ve que con resolver IS para tamaño k, se resuelve VC para tamaño $n - k$ y viceversa. Entonces, la única transformación requerida entre un problema y otro es la cuenta $n - k$, lo cual es $O(1)$, lo cual demuestra que son polinomialmente equivalentes.

Teorema: 3-SAT $\leqslant_p$ IS

Demostración: queremos probar que se puede modelar el problema de 3-SAT como un problema de conjuntos independientes. Creamos el siguiente modelo:

- Para cada una de las k condiciones creamos tres vértices, uno por cada término. Tendremos entonces, $|V| = 3k$
- Para modelar las incompatibilidades entre los términos de distintas condiciones (x_i en una y x_i en otra), unimos con aristas estos casos.

- Como es suficiente satisfacer un término de cada condición para que la asignación satisfaga a todas, entonces unimos los 3 vértices de cada condición entre sí, formando triángulos. Esto no significa que no es posible satisfacer más términos, solo que hacerlo complica las incompatibilidades y no es necesario para demostrar que existe una asignación satisfactoria.

De este modo, si encontramos un conjunto independiente de al menos k vértices, nos aseguramos de que elegimos al menos k términos, uno de cada condición, sin que haya incompatibilidades. Este conjunto da como resultado, justamente, una asignación satisfactoria.

Corolario: 3-SAT $\equiv_p$ SAT $\leqslant_p$ IS $\equiv_p$ VC

Por este motivo, tanto IS como VC son NP-completos y podemos tratar de reducir esos problemas a los que querramos demostrar que también lo son.

1.4. Problema de Ciclos Negativos (CN)

Se tiene un grafo dirigido y pesado G, cuyas aristas tienen pesos que pueden ser negativos. Se pide devolver si el grafo tiene algún ciclo con peso negativo.

El algoritmo de Bellman-Ford de búsqueda de caminos mínimos en grafos tiene la característica de ser óptimo con aristas de pesos negativos (cuando Dijkstra no lo es) y la ventaja de detectar, al final de su ejecución, ciclos negativos.

Aquel consiste, en cada paso, en recorrer todas las aristas $(u, v) \in E$ viendo si la distancia actual al vértice v es mejorable si se llega desde u. Esto puede verse en pseudocódigo como:

```
for e in edges:
    if(distance[e.dst] > distance[e.src] + e.weight):
        distance[e.dst] = distance[e.src] + e.weight
        parent[e.dst] = e.src
```

Si no hay ciclos negativos, en cada uno de estos pasos el algoritmo deja en su valor óptimo la distancia a cada vértice a distancia sin peso más cercano. Es decir, en el primer paso, los vértices adyascentes al origen s quedarán con su menor distancia. En el segundo, los adyascentes a esos (distancia sin peso 2 a s) quedarán minimizados. Por este motivo, como el camino mínimo no puede tener más de $|V| - 1$ aristas (no puede recorrer más de una vez un vértice o no sería óptimo) y por lo tanto al terminarse $|V| - 1$ pasos de los anteriores puede asegurarse que se llegó a destino con el mejor camino posible.

Por otro lado, **en caso de haber ciclos negativos**, estos son mejorables infinitamente (se los puede seguir recorriendo sin fin y seguir bajando su peso total). De este modo, si luego de $|V| - 1$ iteraciones todavía hay una arista mejorable, podemos asegurar que el grafo tiene ciclos negativos, que era el objetivo de este problema.

Entonces, el problema se reduce a aplicar el algoritmo de Bellman-Ford y responder 1 si el algoritmo encontró ciclos negativos.

Esto no significa necesariamente que sea el algoritmo más eficiente para resolver este problema, pero a los fines de este trabajo es suficiente, dado que el algoritmo es claramente polinómico, con complejidad $O(|E||V|)$. Concluimos entonces que CN $\in$ P.

1.5. Problema de Ciclos Nulos (C0)

Se tiene un grafo dirigido y pesado G, cuyas aristas tienen pesos que pueden ser negativos. Se pide devolver si el grafo tiene algún ciclo con peso exactamente igual a cero.

Se puede demostrar que el problema de búsqueda de ciclos negativos es *NP-Completo*.

Primero demostraremos que es *NP*, o sea, que es certificable en tiempo polinomial. Para el problema de ciclos negativos es sencillo: dado un certificado que consiste en un ejemplo de ciclo simplemente tenemos que calcular la suma de sus pesos y ver si es cero, y además verificar que es un ciclo. Esto puede hacerse en tiempo lineal.

Luego reducimos un problema *NP-Completo* conocido, que es el de *subset-sum* al problema de búsqueda de ciclos nulos. Una forma de este problema consiste en n elementos $w_1, \ldots, w_n \in \mathbb{Z}$ y su objetivo es encontrar un que deben sumar 0. Se prueba que este problema Estos son los pasos:

1. Crear un grafo de $2n$ vértices. Por cada elemento w_i creamos un vértice u_i y uno v_i.
2. Unimos cada v_i con su respectivo u_i mediante una arista de peso w_i.
3. Luego unimos cada u_i con todos los v_j (incluyendo el propio), mediante aristas de peso 0.

Esto generará grafos como este:

INSERTAR IMAGEN

Entonces, lo que ocurre es lo siguiente: - Como lo que se encuentra es un ciclo, entonces seguro que ese ciclo incluye al menos a ambos vértices de un elemento. Nunca se elegirá "medio vértice", ya que si así fuera no habría ciclo. Por sabemos que **si el ciclo pasa por un elemento, se suma su valor**. - Como toda arista que no es la propia de cada elemento es nula, entonces un ciclo que tome varios elementos tendrá como peso total la suma de los elementos. - Si un ciclo suma 0, entonces la suma de los elementos por los que pasa es 0. - Si encontramos un ciclo de peso total 0, entonces habremos encontrado un subconjunto de elementos que suman 0 entre sí, lo cual es la solución al problema planteado.

1.6. Problema de decisión de Scheduling con tiempos de ejecución distintos (P3)

Se tiene un conjunto de n tareas, cada una con un tiempo de ejecución $t_i \in R_+$, una fecha límite de finalización $d_i \in R_+$ y una ganancia $v_i \in R_+$ que será otorgada si se finaliza antes que su tiempo límite. Se pide devolver si existe alguna planificación que obtenga una ganancia total, mayor o igual a $k \in R_+$ sabiendo que no se pueden ejecutar dos tareas a la vez

Vamos a demostrar que este problema es NP-Completo. Para ello, primero debemos demostrar que el problema es NP, pudiendo verificar una solución al problema en tiempo polinomial. Luego aplicaremos una reducción al problema de la suma de sub-conjuntos (Subset Sum Problem).

Supongamos que tenemos una planificación de la forma $y = s[1n]$ tal que para cada tarea i, denominamos a $s[i]$ como el tiempo en el que inicia dicha tarea. Para verificar que esta solución corresponde al problema, tenemos que ver que las tareas no se solapen y que la ganancia total obtenida (es decir, la suma de las ganancias de las tareas que terminaron antes de su deadline) sea al menos k.

El certificador propuesto es el siguiente:

Para cada elemento $s[i]$ de nuestra solución propuesta
Para cada elemento $s[j]$ restante $(i \neq j)$
Si $s[i] < s[j]$
$fin[i] = s[i] + t_i$

Si $fin[i] \leq s[j]$
Si $fin[i] <= d_i$
$GananciaTotal+ = v_i$
Fin Si
Sino
return False
Fin Si
Fin Si
Fin Para
Fin Para
Si $GananciaTotal \leq k$
return False
return True

En otras palabras, para cada comienzo de tarea, se debe verificar que dicha tarea, sumado a su tiempo de ejecución, no se solape con el comienzo de ninguna otra tarea que empieza después de la actual. Además, si esto se cumple, y el fin de la tarea ocurre antes de su deadline, se suma a la ganancia total obtenida. Luego resta a saber si la ganancia total obtenida supera al k determinado para esa instancia del problema. Este certificador corre en tiempo polinomial, es $O(n^2)$.

Demostrado esto, ahora realizamos una reducción de un problema conocido NP-Completo a nuestro problema, para formalizar la demostración de que P3 es NP-Completo. Para ello, utilizamos el Subset Sum Problem.

A partir de una instancia del Subset Sum Problem: SSPI = $(sspi[1n], k)$ donde sspi es un set de números enteros positivos y k un número positivo, puedo construirme una instancia de P3, siendo los tiempos de ejecución $t_i = sspi_i$, las ganancias $v_i = sspi_i$, mi límite k equivalente en ambas instancias y mis deadlines $d_i = k \forall i$. Esta transformación es $O(n)$.

Ahora bien, si yo consigo una planificación P que resuelve mi problema $P3$, entonces, tengo un set S de tareas que terminan de ejecutarse antes de su deadline, por ende, $\sum_{i \in S} sspi_i \leq k$ y cuya ganancia es al menos k, esto es $\sum_{i \in S} sspi_i \geq k$. Combinando ambas restricciones, obtenemos $\sum_{i \in S} sspi_i = k$, lo cual es una solución para el Subset Sum Problem.

1.7. Problema de Scheduling con k fijo (P4)

Se tiene un conjunto de n tareas, cada una con un tiempo de ejecución igual a 1, una fecha límite de finalización $d_i \in N$ y una ganancia $v_i \in R$ que será otorgada si se finaliza antes que su tiempo límite. Se pide devolver si existe alguna planificación que obtenga una ganancia total $k \in R$ sabiendo que no se pueden ejecutar dos tareas a la vez.

El problema NP-completo conocido que resulta intuitivo aplicar es otra variante del *subset-sum*, ya que el problema de decisión *P4* es una elección de elementos de un conjunto para que sumen un valor específico. Esta variante consiste en:

- Un conjunto de valores $W = \{w_1, , w_n\} \in \mathbb{N}$.
- Un número K que es nuestra suma objetivo.

Para reducir subset-sum a nuestro problema, podemos transformar la entrada del siguiente modo:

- Cada elemento w_i será una tarea con:

    - Tiempo de ejecución igual a 1. Esto es obligatorio para la entrada de este problema.
    - Deadline n+1 (o infinito). Esto permite que todas las tareas sean elegidas.

- La ganancia deseada K será el mismo número que la suma del *subset-sum* original.

Esta entrada será la de P4, que decidirá si existe una planificación que tenga como ganancia total K, o sea que elegirá un subconjunto de W que sume K, resolviendo el problema de *subset-sum*. Por lo tanto, sabemos que SS $\leqslant_p$ P4 y por lo tanto P4 $\in$ NP-Completo.

2. Algoritmos de Aproximación

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

2.1.1. Solución desarrollada y sus características

En el trabajo práctico anterior se desarrolló un algoritmo que resolvía este problema cuya complejidad era $O(n * W)$. Este tipo de algoritmos es, se dice, $pseudo - polinomial$, ya que no solo depende del parámetro n sino también de W. Esto implica que a valores relativamente bajos de W, se mantenía polinomial, mas no así cuando W crecía mucho.

Ahora bien, podemos desligarnos del valor de la capacidad de la mochila (W) para construir un algoritmo que corra en tiempo polinomial, y devuelva un resultado que se aproxime a la solución óptima. Para esto, volvemos a usar la programación dinámica, construyendo un algoritmo que permitirá pasar de un orden de complejidad de $O(n * W)$ a $O(n^2 * v^*)$ (siendo v^* el máximo valor de entre los asignados a los elementos de entrada). Este orden de complejidad también es $pseudo - polinomial$, pero ya no depende de la capacidad de la mochila, sino del valor asignado a los elementos. En otras palabras, la capacidad puede ser tan grande como se desee, que no afectará al tiempo de ejecución de nuestro algoritmo solución. El algoritmo se definirá en una sección posterior.
A su vez, si se les asigna valores enteros pequeños a los elementos, el problema puede resolverse en tiempo polinomial, y, cuando los v_i son altos, la ventaja que ofrece este algoritmo es que no tenemos que lidiar específicamente con estos v_i altos, sino que podemos alterarlos ligeramente para que se mantengan pequeños (en base a un factor que veremos a continuación) y obtener una solución que se aproxime a la óptima.

Si detectamos que los valores v_i son muy altos, logramos la aproximación deseada *normalizando* dichos valores en base a un factor b determinado y utilizándo estos nuevos valores más pequeños en el desarrollo del algoritmo. A saber, definiendo

$$\tilde{v_i} = \lceil v_i/b \rceil \cdot b$$

, y aprovechando que absolutamente todos los valores dependen ahora del factor b, podremos quedarnos simplemente con el valor escalado

$$\hat{v_i} = \lceil v_i/b \rceil$$

.

Eligiendo un b acorde, sabemos que la resolución del problema tanto con los valores normalizados como con los coeficientes escalados, tienen el mismo set de soluciones óptimas, con los valores óptimos difiriendo solamente por un factor b esta diferencia es lo que llamamos *aproximación*.

Más específicamente, si S es la solución aproximada, y $\hat{S}$ es la solución óptima exacta, obtenemos:

$$(1 + \epsilon) \sum_{i \in S} v_i \geq \sum_{i \in \hat{S}} v_i$$

siendo ϵ un factor de precisión que se utilizará para determinar el factor de escala b, el cual detallamos a continuación.

2.1.1.1. Factor de normalización b

Debemos elegir un b acorde, que permita empequeñecer lo suficiente el valor de los elementos de entrada para que el algoritmo corra en tiempo polinomial. Para ello podemos tomar algo que dependa del máximo de estos v_i y de la cantidad de elementos total de entrada. Además, podemos tener en cuenta la precisión de la aproximación a utilizar, tomando como entrada un parámetro ϵ que defina que tan lejos de la solución óptima nos podemos encontrar. Esto implica, por supuesto, que mientras más pequeño sea ϵ, entonces más precisión debemos lograr, y, por ende, el tiempo de ejecución de nuestro algoritmo será mayor.

Tomando esto en cuenta, definimos $b = (\epsilon/n) \cdot max_i v_i$

2.1.1.2. Nuevo algoritmo

Para este nuevo algoritmo también se utilizará la programación dinámica. Como vimos anteriormente, un problema que dependa de la capacidad de la mochila, siendo ésta muy grande, genera un set de subproblemas enorme, por lo tanto, y sumado al manejo de valores que describimos anteriormente (el hecho de que podemos modificarlos para hacerlos más pequeños y trabajar con ellos), nos da la pauta de que deberíamos manejar un set de subproblemas que dependa de los valores asignados a los elementos y no de la capacidad remanente de la mochila.

Por ende, nuestro valor óptimo ahora dependerá de nuestra cantidad de elementos i, y de un valor V ($\overline{opt}(n, V)$), y dará como resultado la capacidad de mochila W más pequeña que se necesita para poder obtener el valor V. Tendremos subproblemas para toda nuestra cantidad de elementos $i = 0 \dots n$ y todos los valores hasta llegar a la suma total de los mismos $V = 0 \dots \sum_i v_i$. Definiendo el máximo de los valores asignados como v^*, sabemos que $\sum_i v_i \leq nv^*$, por lo que nuestra matriz de subproblemas será, a lo sumo, de $n \times nv^*$, por ende, estamos hablando de un orden de complejidad $O(n^2 v^*)$. Siendo que lo que guardamos en la matriz es el valor de W más pequeño para obtener el V particular, el problema queda solucionado al encontrar el valor de V máximo para el cual el valor óptimo w hallado sea menor o igual a la capacidad de la mochila original del problema W.

Con este orden de complejidad, manteniendo pequeño los valores asignados a los elementos como ya detallamos, podremos lograr un tiempo polinomial.

Los casos que se toman en cuenta para dividir en sub-problemas son los siguientes, siendo S la solución óptima final:

- Si $n \notin S \Rightarrow \overline{opt}(n, V) = \overline{opt}(n-1, V)$
- Si $n \in S \Rightarrow \overline{opt}(n, V) = w_n + \overline{opt}(n-1, max(0, V - v_n))$

2.1.1.3. Algoritmo de aproximación final

Teniendo en cuenta todo lo detallado hasta aquí, el algoritmo de aproximación desarrollado consiste en construir todos los valores $\hat{v}_i = \lceil v_i/b \rceil$ siendo su factor escalado $b = (\epsilon/n) \cdot max_i v_i$, lo cual se logra en tiempo polinomial, y ejecutar el algoritmo de la sección anterior con estos nuevos valores $\hat{v}_i$ de cada elemento.

Ahora bien, como el algoritmo a utilizar es de orden $O(n^2 v^*)$, debemos determinar $v^* = max_i \hat{v}_i$ para obtener el orden de nuestra aproximación. Sabemos que el elemento de valor máximo en la instancia original del problema (v_j) también será el elemento de valor máximo en la instancia del problema escalada por b, por lo que $max_i \hat{v}_i = \hat{v}_j = \lceil v_j/b \rceil = n\epsilon^{-1} = v^*$. Reemplazando correspondientemente, el orden de complejidad de nuestro algoritmo de aproximación será $O(n^3 \epsilon^{-1})$, por lo que es polinomico para todo valor de ϵ fijo mayor a 0.

2.1.2. Tiempos de ejecución

Como vimos en el análisis anterior, el tiempo de ejecución varía tanto con la cantidad de elementos como con la precisión que queremos darle a la aproximación. Podemos ver en el siguiente gráfico la diferencia de tiempos en base a la precisión:

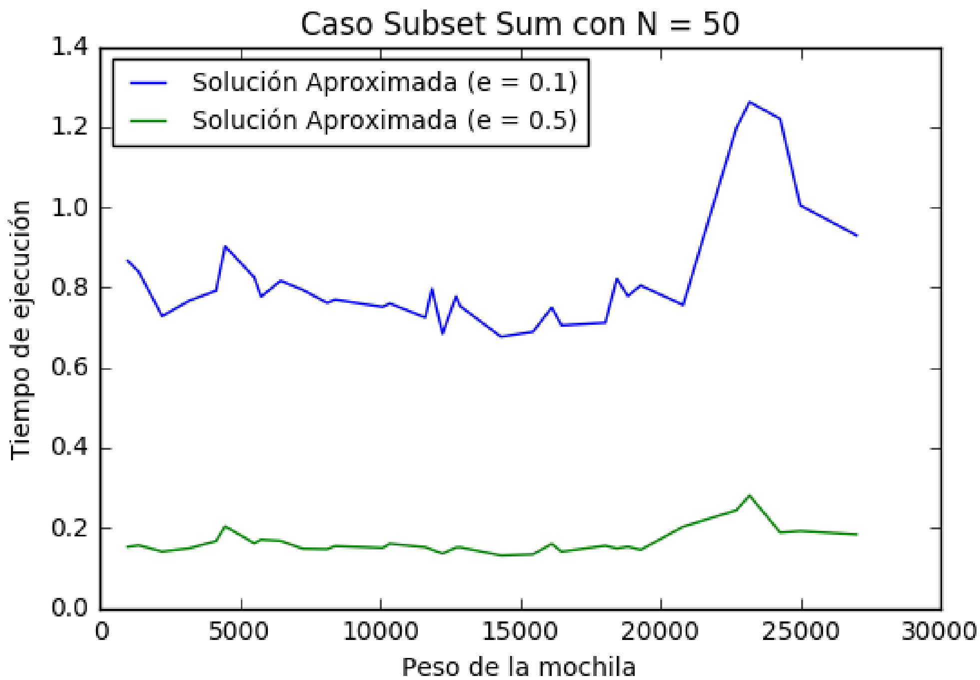

Figura 1: $n = 50$, w_i creciendo

Podemos destacar que el tiempo de ejecución no aumenta en base a la capacidad de la mochila, lo cual es una de las principales diferencias con la solución que se encontró en el trabajo práctico anterior.

Además, vemos una gran diferencia entre los tiempos de ejecución dependiendo de la precisión. En efecto, esta diferencia de precisión trae aparejado una diferencia en los valores óptimos que se encuentran. La idea de trabajar con estos algoritmos será entonces ver cuánto podemos *resignar* de aproximación al valor óptimo para dar con un tiempo de ejecución acorde a lo deseado.

A continuación se presenta la diferencia de valores encontrados con estas dos precisiones y la comparación con el valor óptimo:

Problema	Valor Optimo	Solucion Aproximada con e = 0.1	Solucion Aproximada con e = 0.5
1	8373	8384 (+11)	8414 (+41)
2	5847	5850 (+3)	5882 (+35)
3	5962	5970 (+8)	6017 (+55)
4	4888	4893 (+5)	4925 (+37)
5	4889	4895 (+6)	4930 (+41)
6	8181	8194 (+13)	8233 (+52)
7	6033	6041 (+8)	6085 (+52)
8	6865	6874 (+9)	6911 (+46)
9	7082	7091 (+9)	7136 (+54)
10	7605	7612 (+7)	7641 (+36)
11	9533	9550 (+17)	9613 (+79)
12	7654	7662 (+8)	7717 (+63)
13	9577	9588 (+11)	9642 (+65)
14	11287	11299 (+12)	11377 (+90)

Cuadro 1: Diferencias entre valores optimos dependiendo de la precisión de la aproximación

Es notable la diferencia que existe entre los valores obtenidos por cada una de las soluciones. Es por esto que se acentúa la importancia de decidir entre proximidad al valor óptimo y tiempo de ejecución para este tipo de algoritmos. En base a esto podemos decir que mientras más pequeña sea la precisión elegida, más cerca de la solución exacta estará, por lo que la diferencia con el valor óptimo será menor, el tiempo de ejecución para encontrar estos nuevos valores se disparará (dejará de ser *polinomial*) y nos encontraríamos en un escenario mucho más parecido a la complejidad $pseudo - polinomial$ que trabajamos en el trabajo anterior, debido a la relación del orden del problema con el parámetro ϵ.

2.2. El problema del viajante de comercio

El problema del viajante es de complejidad NP-completo cuya solución tiene un orden temporal de $O(n^2 2^n)$ y un orden espacial de $O(n2^n)$.

Dada la complejidad de algoritmo y las limitaciones físicas de las computadoras, es posible encontrar la solución al problema para un numero reducido de ciudades (alredededor de 20 ciudades).

Para ello se existen métodos que aproximan la solución, aplicando algunas condiciones que permitan tomar decisiones para encontrar el camino mínimo.

2.2.1. Desigualdad triangular

En muchas situaciones prácticas, la manera menos costosa de ir de u a w es ir directamente, sin pasos intermedios. Es decir, cortar una parada intermedia nunca aumenta el costo. Formalmente se dice que la función de costo c satisface la **desigualdad triangular** si para todo vértices u, v, $w \in V$ se cumple

$$c(u, w) \leq c(u, v) + c(v, w)$$

La desigualdad triangular se satisface naturalmente en varias aplicaciones. Por ejemplo, si los vértices del gráfico son puntos en el plano y el coste de viajar entre dos vértices es la distancia euclidiana entre ellos, entonces se satisface la desigualdad triangular. (Cormen et al. 2009)

2.2.2. Algoritmo de aproximación

Aplicando la desigualdad triangular descripta anteriormente, se calcula un árbol recubridor mínimo cuyo peso da un límite inferior del costo de un tour óptimo del viajante de comercio. Luego, se utiliza el árbol recubridor mínimo para crear un recorrido cuyo costo no sea más del doble del peso mínimo del árbol recubridor, siempre y cuando se satisfaga la desigualdad triangular. (Cormen et al. 2009)

Un pseudocódigo para calcular en forma aproximada el ciclo hamiltoniano mínimo es el siguiente:

```
función TSP (G)
  T = árbol recubridor mínimo de G
  raiz = raíz del recorrido (origen)
  camino = visitar los nodos de T comenzando por la raíz
  retornar camino
```

El algoritmo para encontrar el árbol recubridor mínimo puede ser el de Prim o Kruskal. Se implementó el algoritmo de Kruskal.

El algoritmo de Kruskal se puede describir de la siguiente manera

```
función kruskal(G):
  para cada vértice v de G.V:
    crear un conjunto que contenga a v

  E = aristas de G ordenadas por peso creciente

  T = árbol T vacío

  para cada arista e de E:
    si C(e.origen) != C(e.destino):
      agregar la arista e al árbol T
      unir los conjuntos C(e.origen) y C(e.destino)

  retornar T
```

Para operar los conjuntos se utilizó una estructura de conjuntos disjuntos que posee los siguientes métodos:

Buscar Busca el conjunto al que pertenece un vértice dado. Si el vértice no se encuentra en ningún conjunto, crea uno para ese elemento.

Unir Une los conjuntos de dos vértices dados.

Se implementó la estructura que posee dos heurísticas. La primera, *uniones por ranking*, es decir, unir el conjunto pequeño al conjunto más grande. La segunda, *compresión del camino*, consiste en "aplanar" el árbol en una operación de búsqueda, haciendo que los nodos visitados apunten directamente a la raíz del conjunto. Esto hace más eficiente búsquedas futuras, la operación más recurrente. Las operaciones de `Buscar` y `Unir` tienen un costo de $\log(n)$ siendo n la cantidad de vértices almacenados en la estructura. (Cormen et al. 2009, chap. 21)

El algoritmo de Kruskal recorre todas las aristas y para cada una de ellas opera con la estructura de datos realizando 2 búsquedas (una por cada vértice que une la arista) y posiblemente una unión de conjuntos. Por lo tanto, el orden temporal del algoritmo es $O(m \log n)$ siendo m la cantidad de aristas y n la cantidad de vértices.

Una vez obtenido el árbol, se realiza un recorrido en profundidad desde el vértice de origen. Este tipo de recorrido tiene una complejidad temporal de $O(m + n)$ y una complejidad espacial de $O(n)$ donde m es la cantidad de aristas y n la cantidad de vértices del grafo.

Por lo tanto, la complejidad temporal del algoritmo es $O(m \log n)$, lo cual es una gran mejora respecto de la complejidad $O(n^2 2^n)$ de solución óptima. Con respecto a la complejidad espacial, el orden de la aproximación de $O(n)$ también es una gran mejora con respecto a la complejidad $O(n 2^n)$ de la solución óptima.

Sin embargo, la solución aproximada sólo tiene utilidad si los datos de los grafos satisfacen la desigualdad triangular, donde el costo de la ruta no debería superar al doble de la solución óptima.

2.2.3. Tiempos de ejecución y resultados

Los datos utilizados para 15, 17 y 21 ciudades son los recopilados por John Burkardt, de los cuales varios provienen de TSPLIB95. El resto de los datos fueron generados aleatoriamente y se pueden encontrar en el repositorio con el prefijo *ex*. Los datos generados aleatoriamente son los mismos del TP N° 2, pero como el algoritmo de aproximación sólo funciona para grafos simétricos unidireccionales, se toma el triángulo inferior de la matriz.

En la Figura 2 se puede visualizar el tiempo de ejecución para 4 ciudades en adelante. Se muestran los valores hasta el conjunto de 10 ciudades. Para más ciudades, el tiempo insumido por el algoritmo de aproximación es prácticamente depreciable comparado con los del algoritmo Bellman–Held–Karp.

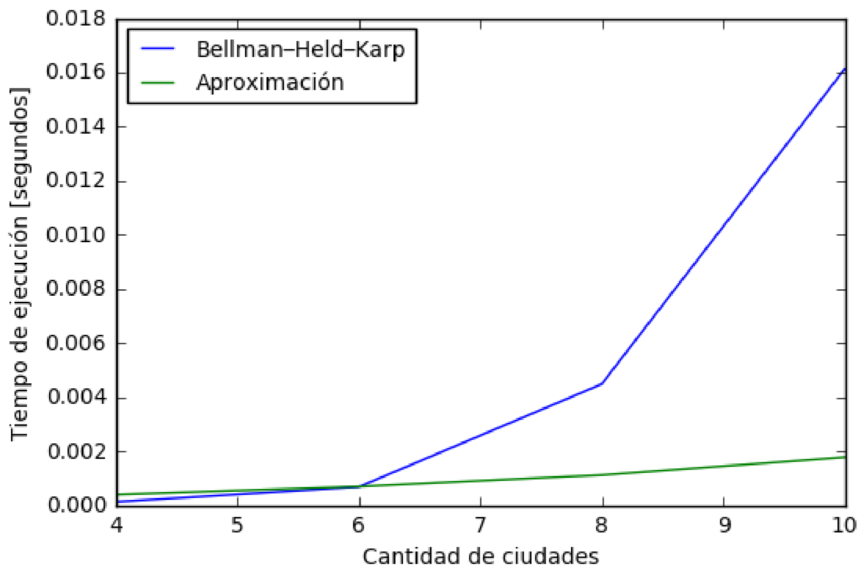

Figura 2: Comparativa del tiempo de ejecución del problema del viajante

En el Cuadro 2 se realiza una comparación del costo del tour del viajante obtenido.

Ciudades	Bellman–Held–Karp	Aproximación	Incremento [%]	Azar
4	26	31	19	Si
6	25	38	52	Si
8	39	39	0	Si
10	144	145	0,69	Si
11	200	325	63	Si
12	118	165	40	Si
13	210	235	12	Si
14	171	429	151	Si
15	291	366	26	No
16	165	271	64	Si
17	2085	2352	13	No
18	371	1144	208	Si
19	819	1945	137	Si
20	694	1817	162	Si
21	2707	3803	40	No

Cuadro 2: Espacio utilizado por el problema del viajante

Utilizando un algoritmo de aproximación el costo se incrementa, en promedio, un 65 %. En 4 casos el valor aproximado supera al doble de la solución óptima. Sin embargo, una consideración a tener en cuenta es que los datos generados aleatoriamente no se corresponden a ninguna distribución de ciudades, por lo que no necesariamente cumple la **desigualdad triangular**.

Los datos que no son generados al azar y que pueden corresponder con ciudades reales son:

15 ciudades Ejemplo de John Burkardt

17 ciudades Conjunto de ciudades de Alemania (Martín Gröetschel)

21 ciudades Conjunto de ciudades de Alemania (Martín Gröetschel)

Para esos 3 casos, el incremento del costo es de 26 %, por lo que el resultado es mejor cuando tienen relación con datos reales.

3. Código

Aclaración: en este documento solo aparece el código que refleja las consignas del trabajo práctico. En nuestro repositorio de código se pueden encontrar tests, ejemplos, benchmarks de algoritmos y más: github.com/ezeperez26/tda-fiuba

3.1. Algoritmos de aproximación

3.1.1. El problema de la mochila

3.1.1.1. Función que setea el ambiente para llamar al algoritmo de aproximación

```
def knapsack_bottom_up_aproximado(items_value, max_value, items_weight, knapsack_weight,
precision_e):

    cant_items = len(items_value)

    # Seteo el factor 'b' y normalizo
    print("Precision: " + str(precision_e) + ". Cantidad Items: " + str(cant_items) +
    ". Max Value: " + str(max_value))
    b = (precision_e / cant_items) * max_value
    print("b: " + str(b))
    rounded_items_value = [ ceil(value/b) for value in items_value ]
    rounded_max_value = ceil(max_value/b)

    # Inicializo mi matriz de resultados
    matrix_value_range = (rounded_max_value * cant_items)
    results = [[float("inf") for x in range(matrix_value_range + 1)] for y in range(cant_items + 1)]
    for i in range(0, cant_items):
        results[i][0] = 0

    # Corro el algoritmo en si, midiendo el tiempo
    knapsack_bottom_up_aproximado_core(rounded_items_value, matrix_value_range,
                                       items_weight, knapsack_weight, results)

    return int(int(knapsack_get_optimum_value_aproximado(results, cant_items,
                   matrix_value_range, knapsack_weight)) * b)
```

3.1.1.2. Implementación del algoritmo de aproximación

```
def knapsack_bottom_up_aproximado_core(items_value, max_value, items_weight,
knapsack_weight, results):

    value_sum = 0
    value_accum = 0
    for i in range(1, len(items_value) + 1):
```

```
        value_accum += items_value[i - 1]
        if i > 1:
            value_sum += items_value[i - 2]

        for v in range(1, value_accum + 1):

            if v > value_sum:
                results[i][v] = items_weight[i - 1] +
                                results[i - 1][max(0, v - items_value[i - 1])]
            else:
                results[i][v] = min(results[i - 1][v], items_weight[i - 1] +
                                results[i - 1][max(0, v - items_value[i - 1])])
```

3.1.1.3. Función que obtiene el valor óptimo aproximado de la matriz solución

```
def knapsack_get_optimum_value_aproximado(results, cant_items, matrix_value_range, knapsack_weight):

    # Recorro la matriz viendo desde el mayor valor V posible
    # Si encuentro que se puede llegar con un W menor a la capacidad de la
    # mochila, ese es mi V optimo
    for v in range(matrix_value_range, 0, -1):

        for i in range(cant_items, -1, -1):

            if results[i][v] <= knapsack_weight:

                return v
```

3.1.1.4. Función que devuelve el set solución aproximado

```
def knapsack_get_solution_set_aproximado(n, V, items_value, items_weight, optimum_values):

    result = []

    if n == 0 or V == 0:
        return None
    else:

        for i in range(cant_items, 0, -1):

            for v in range(V, 0, -1):

                if items_weight[i - 1] + optimum_values[i - 1][v - items_values[i - 1]] == optimum_value
                    result.append(i)
                    v -= items_value[i - 1]

    return list(reversed(result))
```

3.1.2. El problema del viajante de comercio

3.1.2.1. Función que calcula el camino y costo mínimo

```python
def travelling_salesman_aprox_path(graph):
    """
    Recibe el grafo sobre el cual se desea resolver problema del viajante de
    comercio.
    Retorna una lista con el orden de ciudades que minimiza el costo del viaje.
    """

    if not isinstance(graph, Graph):
        print("El parámetro no es de tipo Graph")
        return

    # Obtengo el árbol recubridor mínimo
    tree_graph = graph.minimum_spanning_tree()

    # Busca el camino óptimo
    path = []

    def dfs(start):
        path.append(start)
        for next in tree_graph.adj(start):
            if next not in path:
                dfs(next)
    dfs(0)

    # Calculo el costo del tour
    opt = 0
    current = -1
    for next in path:
        if current < 0:
            current = 0
            continue
        edge = graph.edge(current, next)
        opt += edge.weight
        current = next

    # Agrego el costo al origen
    opt += graph.edge(current, 0).weight
    path.append(0)

    return opt, list(path)
```

3.1.2.2. Método de la clase Graph que genera el árbol recubridor mínimo

```python
    def minimum_spanning_tree(self):
        """
```

```
        Retorna el árbol recubridod mínimo del grafo no dirigido G.
        El árbol retornado es una lista de aristas.
        """
        if not self.is_undirected():
            raise ValueError("MinimumSpanningTree: input is not undirected")
        for u in self:
            for v in self.adj(u):
                if self.edge(u, v).weight != self.edge(v, u).weight:
                    raise ValueError("MinimumSpanningTree: asymmetric weights")

        # Algoritmo de Kruskal: ordena las aristas por peso y las agrega a la
        # estructura de conjuntos disjuntos hasta que no queden conjuntos disjuntos.
        edges = [e for e in self.iter_edges()]
        edges.sort(key=lambda e: e.weight)
        subtrees = UnionFind()
        tree = []
        for e in edges:
            if subtrees[e.src] != subtrees[e.dst]:
                tree.append(e)
                tree.append(Edge(e.dst, e.src, e.weight))
                subtrees.union(e.src, e.dst)

        tree_graph = Graph(self.V())

        for e in tree:
            tree_graph.add_edge(e.src, e.dst, e.weight)

        return tree_graph
```

3.1.2.3. Clase UnionFind

```
class UnionFind(object):
    """Estructura de datos de Union-buscar"""

    def __init__(self):
        """Crea una estructura vacía de Union-buscar."""
        self.weights = {}
        self.parents = {}

    def __getitem__(self, object):
        """Busca y retorna el nombte del conjunto que contiene el objeto."""

        # check for previously unknown object
        if object not in self.parents:
            self.parents[object] = object
            self.weights[object] = 1
            return object

        # find path of objects leading to the root
```

```
        path = [object]
        root = self.parents[object]
        while root != path[-1]:
            path.append(root)
            root = self.parents[root]

        # compress the path and return
        for ancestor in path:
            self.parents[ancestor] = root
        return root

    def __iter__(self):
        """Iterar a través de todos los items encontrados por la estructura."""
        return iter(self.parents)

    def union(self, *objects):
        """Encuentra los conjuntos que contienen los objetos y une."""
        roots = [self[x] for x in objects]
        heaviest = max([(self.weights[r], r) for r in roots])[1]
        for r in roots:
            if r != heaviest:
                self.weights[heaviest] += self.weights[r]
                self.parents[r] = heaviest
```

4. Referencias

Cormen, Thomas, Charles Leiserson, Ronald Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*. third edition. MIT Press.