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. 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.

1.1.1. 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.

1.1.2. 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

1.1.3. 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:

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

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.

1.1.4. 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.

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

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. Determinar la clase de complejidad de un problema.

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 paramétro 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:

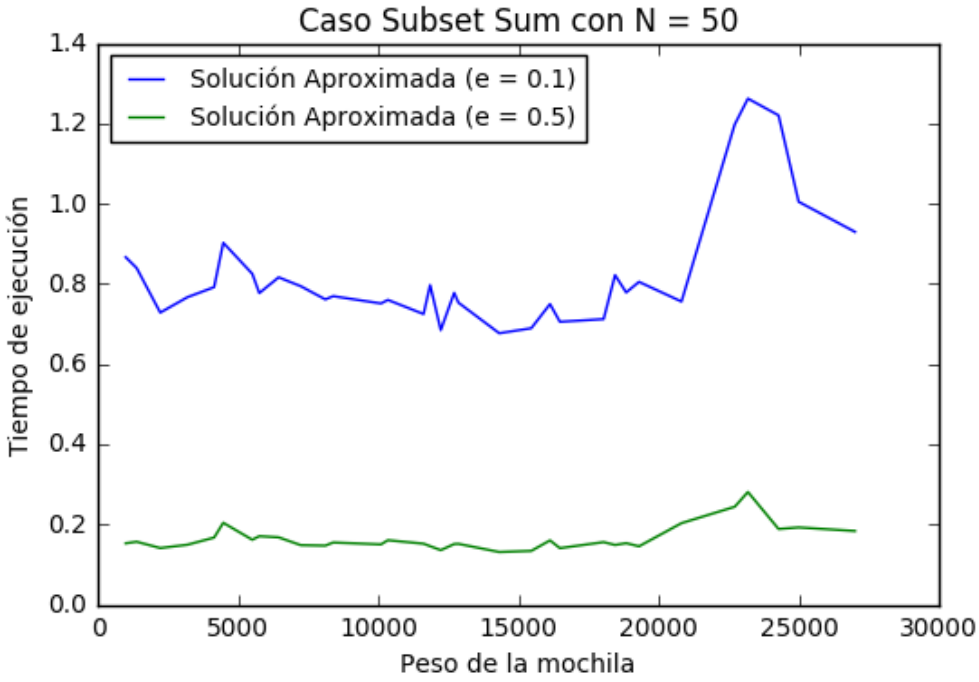

Figura 1: $n = 50$, w_i creciendo

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 (alrededor 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 $O(n2^n)$ 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.

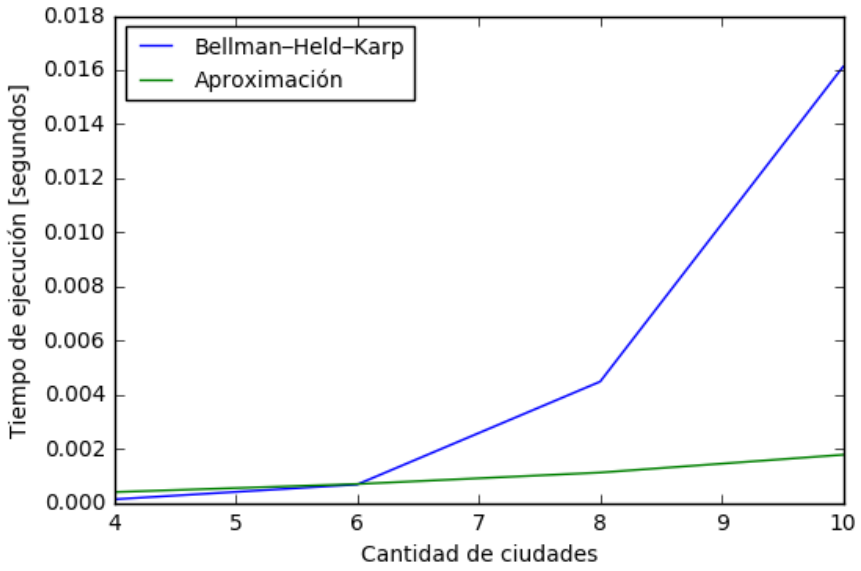

Figura 2: Comparativa del tiempo de ejecución del problema del viajante

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
```

3.1.2. El problema del viajante de comercio

3.1.2.1. Función que calcula el camino y costo mínimo

```
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

```
    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
```

Referencias

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