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

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, se dice, es de orden $pseudo - polinomial$, 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 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 y utilizándo estos nuevos valores más pequeños. 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*).

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. En otras palabras, nuestros nuevos valores óptimos diferirán como mucho en un factor $(1 + \epsilon)$ por debajo de la solución óptima original. 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 $O(n^2v^*)$. 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.

La recurrencia para este nuevo problema podemos plantearla como sigue, 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 $O(n^2v^*)$, 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

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:

```
function TSP (G, c)
  r = G.V.first
  T = minimum_spanning_tree(G, c, r)
  path = ordered_vertex_visit(T, r)
  return (path)
end
```

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

2.2.3. Complejidad

2.2.4. Tiempos de ejecución

Referencias

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