Programación 3 - Trabajando con problemas intratables

Instituto de Computación

Alternativas frente a problemas intratables

- En algunos casos, instancias particulares pueden ser tratables. Ejemplo
  - Cobertura de k vértices, con k fijo.
  - Conjunto independiente en un árbol.
- Algoritmos de aproximación: En algunos problemas de optimización podemos encontrar soluciones no necesariamente óptimas, pero a distancia acotada de una óptima.
- Heurísticas: búsqueda de una solución mediante métodos no rigurosos, como por tanteo, reglas empíricas, etc.

Cobertura con k vértices, con k fijo

- Problema: dado un grafo $G = (V, E)$ con n vértices y un natural k, $0 \leq k \leq n$, ¿existe un conjunto S de k vértices tal que toda arista tiene un extremo en S?
- Existen $\binom{n}{k}$ formas de elegir k vértices.
- En general probar todas las combinaciones posibles es inviable. Por ejemplo, para $k = n/2$, usando la fórmula de Stirling obtenemos $\binom{n}{k} = \Theta\left(\frac{2^n}{\sqrt{n}}\right)$.
- Sin embargo, para k fijo (independiente de n), tenemos

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{(n-k+1)(n-k+2)\dots n}{k!} = \Theta(n^k)\,.$$

- Hay una cantidad polinomial en n de combinaciones a explorar.

Algoritmo de backtracking para explorar combinaciones

```
1  Algorithm Explorar combinaciones
2  |  Hacer S = {}
3  |  Explorar (1, k)
4  end
5  Procedure Explorar(i, k)
   |  /* k elementos tomados de {i ... n}                    */
6  |  if k = 0 then
7  |  |  Probar S
8  |  else if k = n − i + 1 then
9  |  |  Probar S ∪ {i ... n}
10 |  else
11 |  |  Agregar i a S y llamar Explorar (i + 1, k − 1)
12 |  |  Retirar i de S y llamar Explorar (i + 1, k)
13 |  end
14 end
```

Análisis

- Hay $\binom{n}{k}$ ejecuciones de un paso base.
- Las ejecuciones restantes hacen 2 llamadas recursivas. Por lo tanto hay $\binom{n}{k} - 1$ ejecuciones que realizan llamdas recursivas (relación entre cantidad de nodos internos y cantidad de hojas en un árbol binario completo (full)).
- Para nuestro problema en particular, probar si S es una cobertura de aristas por vértices requiere tiempo $O(kn)$.
- Como $\binom{n}{k} = O(n^k)$, el tiempo de ejecución total es $O(kn^{k+1})$.

Backtracking con podas

- No hacer llamadas recursivas que podemos detectar de antemano que no van a prosperar.
- En nuestro caso, para $G = (V, E)$ con $|V| = n$ y $|E| = m$:
  1. Si G tiene un VC de tamaño k, entonces $m \leq k(n-1)$.
     Poda: Si la cantidad de aristas que nos quedan por cubrir es mayor que lo que podríamos cubrir en el mejor de los casos, no seguir por este camino.
  2. Sea $(u, v) \in E$. G tiene un VC de tamaño k si y solo si alguno de los grafos $G - \{u\}$ y $G - \{v\}$ tiene un VC de tamaño $k-1$.
     Poda: No explorar combinaciones en las cuales ambos vértices de una arista quedan afuera de S.

Algoritmo para buscar VC de tamaño k

```
1  Algorithm BuscarVC (G = (V, E), k)
2      if E es vacío then devolver {}
3      if |E| ≥ k|V| then reportar que no existe VC de tamaño k
4      En caso contrario, sea (u, v) ∈ E
5      Hacer S1 = BuscarVC(G − {u}, k − 1)
6      if BuscarVC encontró un VC then devolver S1 ∪ {u}
7      else
8          Hacer S2 = BuscarVC(G − {v}, k − 1))
9          if BuscarVC encontró un VC then devolver S2 ∪ {v}
10         else
11             reportar que no existe VC de tamaño k
12         end
13     end
14 end
```

Análisis

- Cada ejecución realiza a lo sumo dos llamadas recursivas, y con cada una se disminuye en 1 el valor de k. Por lo tanto no hay más que 2^{k+1} llamadas.
- Excluyendo las llamadas recursivas, cada ejecución requiere una cantidad de tiempo que es $O(kn)$.
- El tiempo total de ejecución es por lo tanto $O(2^k kn)$.

Conjuntos independientes en árboles

Si G es un *bosque* y v es una *hoja* (vértice con grado 1), entonces existe un conjunto independiente de tamaño máximo para G que contiene a v.

```
1  Algorithm ConstruirISmáx (G)
2      Hacer S = {}
3      while G contiene aristas do
4          Sea v una hoja y u el único vértice adyacente a v
5          Agregar v a S
6          Eliminar u y v de G
7      end
8      Sea A el conjunto de vértices de G
9      Devolver S ∪ A
10 end
```

Problema de balance de carga

- Máquinas $M_1 \ldots M_m$.
- Tareas con tiempos $t_1 \ldots t_n$.
- $A(i)$ tareas asignadas a máquina i, $1 \leq i \leq m$.
- $T_i = \sum_{j \in A(i)} t_j$.
- *makespan* $T = \max\{T_i\}_{1 \leq i \leq m}$.
- Objetivo: asignar tareas a máquinas (es decir, definir $A(i)$, $1 \leq i \leq m$) de forma de minimizar T.

Algoritmo ávido de aproximación

```
1 Algorithm Greedy-Balance
2 |   Hacer T_i = 0 y A(i) = {} para todo i, 1 ≤ i ≤ m
3 |   for j = 1 to n do
4 |   |   Tomar i ∈ arg mín{T_i}_{1≤i≤m}
5 |   |   Agregar j a A(i) y sumar t_j a T_i
6 |   end
7 end
```

Cotas inferiores para el makespan óptimo

Sea $T^\star$ el mínimo valor de makespan entre todas las asignaciones posibles de tareas a máquinas.

Se cumple

- $T^\star \geq \frac{1}{m} \sum_{j=1}^{n} t_j$.
- Se alcanza cuando el trabajo se reparte de forma perfectamente equitativa entre las m máquinas.
- $T^\star \geq \text{máx}\{t_j\}_{1 \leq j \leq n}$.

Cota para el makespan alcanzado por Greedy-Balance

Sea T el makespan de la solución generada por Greedy-Balance. Se cumple $T \leq 2T^{\star}$.

- Probamos que en todo momento se verifica que $T_i \leq 2T^{\star}$, para todo i, $1 \leq i \leq m$
- Consideramos el instante posterior a la asignación de una tarea j a una máquina i, al final de una iteración arbitraria.
- La carga de trabajo para la máquina i antes de esta asignación está dada por $T_i - t_j$.
- Como i era la máquina con menor carga al momento de ser elegida, entonce $T_i - t_j \leq T_k$, para todo k, $1 \leq k \leq m$.
- Por lo tanto, $T_i - t_j$ está acotado por el promedio de los T_k, $T_i - t_j \leq \frac{1}{m} \sum_{k=1}^{m} T_k$.
- Como $\frac{1}{m} \sum_{k=1}^{m} T_k \leq T^{\star}$ y $t_j \leq T^{\star}$, concluimos que $T_i \leq 2T^{\star}$.

Conclusiones

Supongamos que nos enfrentamos a un problema X.

- Si observamos que $X \leq_P Y$ para un problema Y que es $\mathcal{NP}-Completo$, esto no implica necesariamente que X es difícil.

Conclusiones

Supongamos que nos enfrentamos a un problema X.

- Si observamos que $X \leq_P Y$ para un problema Y que es $\mathcal{NP}-Completo$, esto no implica necesariamente que X es difícil.

- Quizás las instancias de Y a las que se reduce X son casos particulares tratables.

Conclusiones

Supongamos que nos enfrentamos a un problema X.

- Si observamos que $X \leq_P Y$ para un problema Y que es $\mathcal{NP}-Completo$, esto no implica necesariamente que X es difícil.

- Quizás las instancias de Y a las que se reduce X son casos particulares tratables.

- Si probamos que $Y \leq_P X$ ...

Conclusiones

Supongamos que nos enfrentamos a un problema X.

- Si observamos que $X \leq_P Y$ para un problema Y que es $\mathcal{NP}-Completo$, esto no implica necesariamente que X es difícil.

- Quizás las instancias de Y a las que se reduce X son casos particulares tratables.

- Si probamos que $Y \leq_P X$ ...

- Encontrar una solución exacta puede ser impracticable, pero a veces podemos obtener soluciones aproximadas con cotas conocidas para su desempeño.