UNIVERSITAT JAUME I

EI1022, Algoritmia

Problemas 5: Ramificación y acotación

El problema de la mochila sin fraccionamiento

- Disponemos de n objetos, cada uno con un valor v_i y un peso w_i. Además disponemos de una mochila con capacidad de carga C.
- Queremos cargar la mochila, sin sobrepasar su capacidad de carga, de forma que el valor de lo que contenga sea máximo.
- El conjunto de soluciones factibles será:

$$X = \left\{ (x_0, \ldots, x_{n-1}) \in \{0,1\}^n \,\middle|\, \sum_{0 \le i < n} x_i w_i \le C \right\}$$

Implementación

- Crea el programa `knapsack_bab.py`
- Podrás reutilizar parte del código de `knapsack.py` (el programa de la práctica de backtracking):
  - El formato de entrada es el mismo que el de `knapsack.py`, no habrá que cambiar `read_data`
  - `show_results` tampoco cambia

Implementación (2)

- Reescribe la función `process`:
  - Básate en la función `knapsack_bab_solve` de las trasparencias de teoría:
    - Cota pesimista: Utiliza un algoritmo voraz para encontrar una solución.
    - Cota optimista: Utiliza un algoritmo voraz para encontrar la solución óptima al problema de la mochila continua.
  - Para facilitar la implementación y no tener que trabajar con un nivel adicional de índices, asumiremos que los objetos están ordenados de mayor a menor ratio valor/peso.

Pruebas

- En el aula virtual tienes un fichero comprimido donde las pruebas del problema de la mochila están en el directorio `knapsacks_bab`
- Hay tres ficheros con problemas: `small.kps` (5 objetos), `medium.kps` (30 objetos) y `large.kps` (60 objetos)
- Los ficheros `small.sol`, `medium.sol` y `large.sol` tienen las soluciones correspondientes
- Usa `small.kps` para depurar tu programa y `medium.kps` y `large.kps` para ver los tiempos de ejecución

El problema del empaquetado (*bin packing*)

- Tenemos N objetos que queremos guardar en el mínimo número de contenedores
- Supondremos que todos los contenedores son iguales y que cada uno de ellos admite cualquier número de objetos siempre que su peso total sea menor o igual que su capacidad de carga, C
- También supondremos que los pesos de los objetos son mayores que cero y que ningún objeto tiene un peso mayor que la carga del contenedor

Formalización

- Identificaremos los objetos con sus pesos, de modo que podemos representar los objetos como la secuencia $w = (w_0, w_1, \ldots, w_{n-1})$, donde $w_i > 0$ es el peso del objeto i
- Una solución será una tupla de n números de contenedor $x = (x_0, x_1, \ldots, x_{n-1})$, donde x_i es el número del contenedor en el que se almacenará el objeto i
- Si asumimos que los contenedores se numeran desde cero en adelante y que en nuestra solución no hay ninguno vacío, podemos escribir el número de contenedores de la solución como

$$\mathrm{NC}((x_0, x_1, \ldots, x_{n-1})) = 1 + \max_{0 \le i < n} x_i$$

Formalización (2)

- Por lo tanto, el conjunto de soluciones es

$$X = \left\{ x \in \mathbb{N}^n \;\middle|\; \forall\, 0 \leq c < \mathrm{NC}(x) : 0 < \sum_{\substack{i<n \\ x_i = c}} w_i \leq C \right\}$$

- Y nuestra función objetivo será:

$$f(x) = \mathrm{NC}(x)$$

- Queremos encontrar la solución $\hat{x}$ que minimiza f:

$$\hat{x} = \arg\min_{x \in X} f(x)$$

Ejemplo

- Disponemos de infinitos contenedores de capacidad 10 y seis objetos de pesos $(1, 2, 8, 7, 8, 3)$
- La solución $(1, 0, 0, 2, 1, 2)$ representa el siguiente reparto de objetos en tres contenedores:
  - Contenedor 0: objetos 1 y 2. Peso: $2 + 8 = 10$
  - Contenedor 1: objetos 0 y 4. Peso: $1 + 8 = 9$
  - Contenedor 2: objetos 3 y 5. Peso: $7 + 3 = 10$

Cota optimista

- Trata de rellenar los huecos en los contenedores ya usados como si el problema fuera continuo (considera que los objetos restantes se pueden fraccionar) pero con dos restricciones:
  - No consideres fraccionables aquellos objetos mayores que el mayor hueco de los contenedores
  - No consideres los huecos en los que no quepa ni siquiera el objeto más pequeño.
- Divide el peso que no has utilizado entre la capacidad para estimar el número de contenedores adicionales.

Cota optimista: Algoritmo

- Obtén el peso de los objetos sobre los que todavía no has decidido y que quepan en alguno de los contenedores.
- Supón que esos objetos se van a poder distribuir en los huecos de aquellos contenedores ya utilizados cuyo espacio libre sea mayor o igual que el menor de los objetos pendientes.
- Estima el número de contenedores necesarios para los objetos mayores que el mayor hueco dividiendo su peso total por la capacidad de los contenedores

Cota pesimista

- Podemos utilizar un algoritmo voraz para encontrar una cota pesimista.
- En el tema de voraces vimos dos algoritmos de aproximación:
  - "En el primero en que quepa". Garantiza que el número de contenedores que utiliza es menor o igual que $\frac{17}{10}f(\hat{x})$
  - "En el primero en que quepa ordenado". Garantiza que el número de contenedores que utiliza es menor o igual que $\frac{6}{9} + \frac{11}{9}f(\hat{x})$

Implementación

- Crea el programa `binpacking_bab.py`
- Podrás reutilizar parte del código de `binpacking.py` (el programa de la práctica de voraces):
  - El formato de entrada es el mismo que el de `binpacking.py`, no habrá que cambiar `read_data`
  - `show_results` tampoco cambia

Implementación (2)

- Reescribe el método `process`:
  - Contendrá la clase `BinpackingDS`, que hereda de `BoundedDecisionSequence`
  - Para la cota optimista, utiliza el algoritmo explicado antes
  - Para la cota pesimista, aprovecha el código de `binpacking_pqq.py`, pero teniendo en cuenta que:
    - La lista de espacio libre/ocupado se debe inicializar teniendo en cuenta los items ya colocados (seguramente, la tendrás en tu `Extra`)
    - Hay que recorrer los objetos no colocados
  - No hace falta usar `binpacking_pqqo.py` porque asumiremos que los datos de entrada están ordenados de mayor a menor peso

Pruebas

- Dentro del fichero auxiliar del aula virtual tienes el directorio `binpacking_bab` con ficheros de prueba y el programa `bpack_sol_viewer.py` para ver las soluciones en un formato más legible
- Hay tres ficheros con problemas: `small.bpk` (6 objetos), `medium.bpk` (200 objetos) y `large.bpk` (500 objetos)
- Los ficheros `small.sol`, `medium.sol` y `large.sol` tienen las soluciones correspondientes
- Usa `small.bpk` para depurar tu programa y `medium.bpk` y `large.bpk` para ver los tiempos de ejecución

Pruebas (2)

- Para ver una solución con `bpack_sol_viewer.py` debes pasarle como parámetro de la línea de órdenes el problema y por la entrada estándar la solución.
- Por ejemplo (editado para que quepa):
```
python3 bpack_sol_viewer.py binpacking_bab/small.bpk
    < binpacking_bab/small.sol
```
- Puedes encadenar la salida de tu programa:
```
python3 binpacking_bab.py < binpacking_bab/small.bpk
    | python3 bpack_sol_viewer.py binpacking_bab/small.bpk
```