Mat 3770
Bin Packing
or
The Knapsack Problem

Spring 2014

1

Dynamic Programming

- Used when a problem can be partitioned into non–independent sub–problems
- Solve each sub–problem once; solution is saved for use in other sub–problems
- Combine solutions of sub–problems into a solution for the original problem
- Effective when a given sub–problem may arise from more than one partial set of choices

2

Approach

- The structure of an optimal solution is **recursively** defined.
- The value of an optimal solution is computed in a **bottom–up** fashion.
- An example: Fibonacci numbers

$$F_1 = 1,\ F_2 = 1,\ F_n = F_{n-2} + F_{n-1}$$

3

Problem Definition

Generally: Given a knapsack with weight capacity K and n objects of weights $w_1, w_2, \ldots, w_n$, is it possible to find a collection of objects such that their weights add up to K, i.e.

$$\textit{find } w_{i_1}, w_{i_2}, \ldots, w_{i_m}, \textit{ such that } \ K = \sum_{j=1}^{m} w_{i_j}?$$

For example, given a list: { `12, 10, 40, 3, 11, 26, 37, 28, 9, 18` }, does any subset add up to 72?

yes: { `12, 3, 22, 37, 9` }

4

This Problem Has Many Variations

- Allow same weights to be used multiple times
- Ask: how close to full (without going over) can we get?
  I.e., if $W = \{w_1, w_2, \ldots, w_n\}$, we want to minimize

  $$K - \sum_{w \in W'} w \textit{ over all } W' \subseteq W$$

  subject to the constraint that

  $$K - \sum_{w \in W'} w \geq 0$$

- Assume there's a corresponding *value* v_i for each w_i (e.g., gold's value & its weight). Maximize the total *value* given that the weight must be at most K:

  $$\textit{Maximize over } I \subseteq \{1, \ldots, n\}, \sum_{i \in I} v_i, \textit{ given } \sum_{i \in I} w_i \leq K$$

5

The Basic Problem

- Is there $\mathtt{I} \subseteq \{1, \ldots, n\}$ *such that* $\sum_{i \in \mathtt{I}} w_i = K$
- Let predicate

  $$P[i,k] = \begin{cases} \texttt{true} & : \textit{ if } \exists\, \mathtt{I} \subseteq \{1, \ldots, i\} \ni \sum_{j \in I} w_j = k. \\ \texttt{false} & : \textit{ otherwise} \end{cases}$$

  and we want the value of $P[n,K]$

6

Observation

- $P[i,k] = \begin{cases} P[i-1,k] & : \text{ if } w_i \text{ is not used} \\ P[i-1,k-w_i] & : \text{ if } w_i \text{ is used} \end{cases}$
- Furthermore:
  - $P[i,k]$ is false if $k < 0$
  - $P[i,k]$ is true for $k = 0$
  - $P[1,k]$ is true IFF $k = w_1$
- So, we can build a table to solve this type of problem.

Bin Packing Algorithm

```
// initialize first row
P[0, 0] = true
for currentWeight := 1 to K
    P[0, currentWeight] = false

// calculate rest of rows 1 through n
for i := 1 to n
    for currentWeight := 0 to K
        P[i, currentWeight] = false
        if P[i - 1, currentWeight] then
            P[i, currentWeight] = true
        else
            if currentWeight - wi >= 0 then
                if P[i - 1, currentWeight - wi] then
                    P[i, currentWeight] = true
```


Worksheet

Time Complexity

- The idea behind dynamic programming is to build a table iteratively, where we solve a problem by storing solutions to subproblems efficiently.
- What is our complexity for finding $P[n,K]$?
- We have to fill in the table out to the K^{th} column for $n-1$ rows, then consider the n^{th} row at column K if $P[n-1,K] \neq true$.
- Thus: $O((n-1)K+1) = O(nk)$
  which could be bad if $K = 2^n$.

Follow–up

- To make it easier to know which objects were used, we could mark $P[i, currentWeight]$ if w_i was used (i.e., if $P[i-1,k-w_i] = true$)
- The we can trace back through the table to find which w_i were used.
- Note that recapturing the subset would take $O(n)$ time.

A Variation On The Problem

- Given a collection of items, $T = \{t_1, \ldots, t_n\}$ where t_i has (integer) size s_i and a set B of bins each with a fixed (integer) capacity b.
- A subset of the items can be packed in one bin as long as the sum of the sizes of the items is less than or equal to b.
- The goal is to pack all items, minimizing the number of bins used.

An Example

Suppose there are 7 items
with sizes 1, 4, 2, 1, 2, 3, 5, and
the bin capacity $b = 6$.

One (optimal) solution:

Bin 1 :	1 and 5	= 6
Bin 2 :	4 and 2	= 6
Bin 3 :	1, 2, and 3	= 6

Solution Methods

- Exhaustive Search
- Greedy Approach
- Dynamic Programming
- Hierarchical or Divide–and–Conquer
- Mathematical Programming

Exhaustive Search

Many Possibilities — One is:

- Find all partitions of the items — from all in a single set, to all items in separate sets.
- Determine the feasible partitions, and
- choose one of the feasible solutions that uses a minimum number of bins.

Greedy Approach

- Optionally, can sort items in increasing order by size:
  1, 1, 2, 2, 3, 4, 5
- Pack a b in with the items given in order until the bin is full
- get another (empty bin) and keep packing until all items are in bins.
- Results:
  Bin 1 : 1, 1, 2, 2 Bin 2 : 3 Bin 3 : 4 Bin 4 : 5
  Suboptimal since it uses 4 bins instead of 3

Dynamic Programming

Strategy: solve the problem for the first k items, then consider the $(k+1)^{st}$ iteration and determine the best way to place the $(k+1)^{st}$ item in previous (or a new) bins.

- Items: 1, 4, 2, 1, 2, 3, 5 Bin capacity: 6
- Step 1 : place 1 $\boxed{1}$
- Step 2 : place 4 $\boxed{1, 4}$
- Step 3 : place 2 $\boxed{1, 4}$ $\boxed{2}$
- Step 4 : place 1 $\boxed{1, 1, 4}$ $\boxed{2}$
- Step 5 : place 2 $\boxed{1, 1, 4}$ $\boxed{2, 2}$
- Step 6 : place 3 $\boxed{1, 1, 4}$ $\boxed{2, 2}$ $\boxed{3}$
- Step 7 : place 5 $\boxed{1, 1, 4}$ $\boxed{2, 2}$ $\boxed{3}$ $\boxed{5}$

- Note: order makes a difference in this algorithm.
- If they are sorted in descending order, 5, 4, 3, 2, 2, 1, 1, we happen to get an optimal solution, but this isn't guaranteed in all cases.
- $\boxed{5,\ 1}$ $\boxed{4,\ 2}$ $\boxed{3,\ 2,\ 1}$

Divide and Conquer

- Partition the problem into 2 subproblems
  For example:
  $\boxed{1,\ 4,\ 2,\ 1,\ 2,\ 3,\ 5} \Rightarrow \boxed{1,\ 4,\ 2,\ 1}$ and $\boxed{2,\ 3,\ 5}$
- Recursively partition each subproblem until each subset can fit into a bin.
  - $\boxed{1,\ 4,\ 2,\ 1} \Rightarrow \boxed{1,\ 4}$ and $\boxed{2,\ 1}$ bins 1 & 2
  - $\boxed{2,\ 3,\ 5} \Rightarrow \boxed{2,\ 3}$ and $\boxed{5}$ bins 3 & 4
  - Uses four bins – suboptimal

Linear Programming

A linear programming problem may be stated as follows:

Given real numbers $b_1, \dots, b_m$, $c_1, \dots, c_n$, and a_{ij} (for $1 \leq i \leq m$ and $1 \leq j \leq n$), minimize (or maximize) the function:

$$Z(X_1, \dots, X_n) = c_1 X_1 + \cdots + c_n X_n$$

subject to the conditions:

$$\begin{array}{lll} a_{11}X_1 + a_{12}X_2 + \cdots + a_{1n}X_n & \{\leq, =, \geq\} & b_1 \\ a_{21}X_1 + a_{22}X_2 + \cdots + a_{2n}X_n & \{\leq, =, \geq\} & b_2 \\ \dots & \dots & \dots \\ a_{m1}X_1 + a_{m2}X_2 + \cdots + a_{mn}X_n & \{\leq, =, \geq\} & b_m \end{array}$$

Vocabulary

- The X_i are called **decision variables**
- Z is the **objective function**
- The conditions are called **constraints**

Vocabulary

- A **solution** is any specification of values for the decision variable
- A **feasible solution** is one in which all the constraints are satisfied
- An **infeasible solution** leaves one or more of the constraints unsatisfied
- An **optimal solution** is a feasible solution which minimizes (maximizes) Z
- The **solution (search) space** is the set of all possible configurations of the decision variables.

Mathematical Programming

We can formulate the decision problem in general form as follows:

- Let U be the set of items, $U = \{u_1, u_2, \dots, u_n\}$
- Let B be the set of bins, $B = \{b_1, b_2, \dots, b_k\}$
  For our particular problem, all our $b_i = 6$
- Form a complete bipartite graph $G = (U, B, E)$, with the goal of assigning an item to one and only one bin
- The weight w_{ij} (the i^{th} item in the j^{th} bin) of an edge connecting one vertex u_i in U and one vertex b_j in B, is set to $s(u_i)$, the size of item u_i.

The χ Function

$$\chi_{ij} = \begin{cases} 0 & : \textit{if } i^{th} \textit{ item not in } j^{th} \textit{ bin} \\ 1 & : \textit{otherwise} \end{cases}$$

The objective is to *maximize* $\sum_{(i,\ j) \in E} (w_{ij} \times \chi_{ij})$

Subject to the following four constraints:

1. `[0, 1]` **Constraint**: cannot have part of an item in a bin
$$\chi_{ij} \in \{0, 1\}$$
2. **Capacity Constraint**: the sum of the sizes of the items in each bin cannot exceed its capacity
$$\sum (w_{ij} \times \chi_{ij}) \ \ \forall i \in U \ \leq \ b_j \ \ \forall j \in B$$
3. **Assignment Constraint**: each item can only be assigned to one bin
$$\sum (\chi_{ij}) \ \forall j \in B \ = \ 1 \ \forall i \in U$$
4. **Completeness Constraint**: the total number of χ_{ij} whose value is 1 is equal to n — every item gets put in some bin.
$$\sum_{(i,\ j)} \chi_{ij} = n$$

A Possible Solution

- Step 1: Set the number of bins $|B|$ to a lower bound found by dividing the sum of the item sizes by the bin capacity.
- Step 2: Formulate the linear programming problem and use it to find a feasible solution satisfying the set of constraints.
- Step 3: If a solution exists for $|B|$ bins, we're done!
- Step 4: Otherwise, set $|B|$ to $|B| + 1$ and repeat steps 1 – 4.