

Learning Outcomes

COMP202
Complexity of Algorithms
Fundamental Solution Techniques
Dynamic Programming
[See Chapter 5 in Goodrich and Tamassia.]

At the conclusion of this set of lecture notes, you should:

1. Understand the general ideas behind the method of dynamic programming.
2. See the importance of the method of dynamic programming for solving problems having suitable structure, and be able to recognize when it can be applied.
3. Be familiar with some standard problems for which dynamic programming is a successful solution technique such as the $\{0, 1\}$ Knapsack Problem and the Weighted Interval Scheduling Problem.

Dynamic Programming

The *Dynamic Programming* technique is somewhat similar to Divide-and-Conquer algorithms.

The main difference is that (possibly) repetitive recursive calls are replaced by a reference to already computed values stored in a *special table*.

Dynamic Programming (cont.)

Dynamic programming is primarily used for optimization problems (maximizing or minimizing some function).

It is often applied where a *brute force* search for optimal value is *infeasible*.

However, dynamic programming is efficient only if the problem has a certain amount of structure that can be exploited.

Dynamic Programming (cont.)

An effective dynamic programming solution depends on the following factors:

- ▶ *Simple sub-problems*: There must be a way of breaking the whole problem into smaller sub-problems which have a similar structure.
- ▶ *Sub-problem optimality*: An optimal solution to the global problem must be composed of optimal solutions to a sub-problem, or collection of sub-problems.
- ▶ *Sub-problem overlap*: Optimal solutions to some sub-problems can themselves contain sub-problems in common (and often do).

{0, 1} Knapsack Problem

The {0, 1} *Knapsack Problem* is the knapsack problem where *taking fractions of items is forbidden*.

Recall we have a set, S , of n items where item i has benefit b_i and an integer weight w_i .

We have the following objective:

Find a subset $T \subseteq S$ that

$$\begin{aligned} &\text{maximizes } \sum_{i \in T} b_i \\ &\text{subject to } \sum_{i \in T} w_i \leq W_{\max}, \end{aligned}$$

where W_{\max} is the maximum total allowed weight in the knapsack.

Dynamic Programming (cont.)

A typical approach to solve a problem using Dynamic Programming is to first find a *recursive* solution to the problem.

Once this recursive solution is found, we then try to determine a way to compute solutions to the subproblems “from the bottom up”, so that we *avoid recomputing solutions* many times.

{0, 1} Knapsack Problem (cont.)

Unlike the Fractional Knapsack Problem, the {0, 1} version does not have a natural greedy solution (that works all the time).

Exponential solution: We can “easily” solve the {0, 1} knapsack problem in $O(2^n)$ time by *testing all possible* subsets of the n items.

Unfortunately, *exponential complexity* is unacceptable for large n , so we then want to focus on nice characterizations for sub-problems in order to use dynamic programming.

{0, 1} Knapsack problem (cont.)

We said before that we're looking for a recursive solution, so consider an optimal set of items $I \subseteq S$, ignoring for the moment that we don't know what exactly is in I .

The obvious (but important) thing to notice is that either I contains item n , or it doesn't.

If I *does not* use item n , then the optimal solution for S is the same as the optimal solution for the set $\{1, 2, \dots, n-1\}$.

If I *does* use item n , then the optimal solution consists of item n , together with an optimal solution for the set $\{1, 2, \dots, n-1\}$, but with a new maximum allowed weight of $W_{\max} - w_n$.

{0, 1} Knapsack problem (cont.)

To make our solution as easy as possible to find, we will compute these values "from the bottom up", i.e. first we compute $B[1, w]$ for each $w \leq W_{\max}$.

Then we use those values to find $B[2, w]$ for all values of w , and so forth.

{0, 1} Knapsack problem (cont.)

To more precisely define this recursive solution, let $S_k = \{1, 2, \dots, k\}$ denote the set containing the first k items, and define $S_0 = \emptyset$.

Let $B[k, w]$ be the *maximum total benefit* obtained using a subset of S_k , and having total weight *at most* w .

Then we define $B[0, w] = 0$, for each $w \leq W_{\max}$, and

$$B[k, w] = \begin{cases} B[k-1, w] & \text{if } w < w_k \\ \max\{B[k-1, w], b_k + B[k-1, w - w_k]\} & \text{otherwise.} \end{cases}$$

Our desired solution is then $B[n, W_{\max}]$.

{0, 1} Knapsack Problem - Algorithm

01KNAPSACK(S, W)

▷ Input: Set S of n items with weights w_i , benefits b_i , and a total weight W .

▷ Output: A subset T , of S with weight at most W .

```
1 for w ← 0 to W
2   do
3     B[0, w] ← 0
4 for k ← 1 to n
5   do
6     for w ← W downto wk
7       do
8         if B[k-1, w-wk] + bk > B[k-1, w] then
9           B[k, w] ← B[k-1, w-wk] + bk
10        else
11          B[k, w] ← B[k-1, w]
```

{0, 1} Knapsack problem

The running time of 01KNAPSACK is dominated by the two nested “for” loops, where the outer loop iterates n times, and the inner loop W times.

Theorem: The 01KNAPSACK algorithm finds a *highest benefit subset* of S with total weight at most W in $O(nW)$ time.

[Note that the algorithm can be easily modified to also return the *set* of items that gives the maximum benefit. How?]

Weighted Interval Scheduling

Recall the Interval Scheduling Problem. This is the problem where we are given a collection of n intervals (or jobs/time periods) that we wish to schedule on one machine (or in one room).

Our goal was to *maximize* the number of intervals that we could schedule on this single machine.

This problem could be easily solved with a greedy algorithm.

Weighted Interval Scheduling (cont.)

A related problem we consider is *Weighted Interval Scheduling*.

Here we still have a collection of n intervals (or tasks). Each interval has a start time s_i and a finish time f_i .

Furthermore, each interval has a value v_i .

Goal: In this case, the aim is to schedule a subset of non-conflicting intervals having *maximum total value*.

Weighted Interval Scheduling (cont.)

Despite the similarities to the Interval Scheduling Problem without weights (or where each interval has value 1), there is no natural greedy algorithm to solve the Weighted Interval Scheduling Problem.

So we need some other approach.

First, similar to the original problem, suppose we have sorted the intervals in order of their *finishing time*, so that

$$f_1 \leq f_2 \leq \dots \leq f_n.$$

To help in what follows, define for an interval j , define $p(j)$ to be the largest index $i < j$ such that intervals i and j are disjoint (i.e. the largest value of i such that $f_i \leq s_j$).

We define $p(j) = 0$ if there is no disjoint interval $i < j$.

Weighted Interval Scheduling (cont.)

Like most dynamic programming problems, we first look for a recursive solution.

So, consider some optimal schedule S , again ignoring for the moment that we don't actually know what S is.

- ▶ Either S schedules the interval n , or it doesn't.

We note that if interval n is in the solution, then no interval with index larger than $p(n)$ can be in the solution, because of the definition of $p(n)$.

So if S contains interval n , then it also must contain an optimal solution to the sub-problem that consists of intervals $\{1, 2, \dots, p(n)\}$.

If interval n isn't in the optimal solution S , then S consists of an optimal solution of the subproblem with intervals $\{1, 2, \dots, n-1\}$.

Weighted Interval Scheduling (cont.)

With these observations, we have that

$$Opt(j) = \max\{v_j + Opt(p(j)), Opt(j-1)\}.$$

We also see that (depending upon the value of $Opt(j)$ above) either

$$S_j = S_{p(j)} \cup \{j\} \text{ or } S_j = S_{j-1}.$$

Weighted Interval Scheduling (cont.)

With the last reasoning in mind, we define S_j to be an optimal solution to the subproblem consisting of the intervals $\{1, \dots, j\}$, and $Opt(j)$ to be the value of S_j .
(Also define $S_0 = \emptyset$ and $Opt(0) = 0$.)

For any interval j , either $j \in S_j$ or $j \notin S_j$.

The important thing to notice is that if $j \in S_j$ then

$$Opt(j) = v_j + Opt(p(j)).$$

If $j \notin S_j$ then

$$Opt(j) = Opt(j-1).$$

Weighted Interval Scheduling (cont.)

This recursive solution can be used to find $Opt(n)$ (and the set S_n , containing the set of tasks that make up an optimal schedule).

But in order to *avoid* an exponential time solution, we instead compute the values in order $Opt(1), Opt(2), \dots, Opt(n)$.

Doing so avoids repeating calculations that would otherwise be performed in the recursive calls.

Theorem: Finding $Opt(n)$ can be performed in time $O(n \log n)$.