Lecture 21: Greedy Algorithms

COSC242: Algorithms and Data Structures

Brendan McCane

Department of Computer Science, University of Otago

Greedy Algorithms

Dijkstra's algorithm and Prim's algorithm are both examples of greedy algorithms.

A greedy algorithm tries to solve an optimisation problem by making a sequence of choices. At each decision point, the alternative that seems best at that moment is chosen.

This is such a simple approach that it is what one usually tries first.

Priority queues are essential data structures for many greedy algorithms.

In both Dijkstra's and Prim's algorithms, a priority queue is used to extract the next node to visit.

Knapsack problems

Consider now a totally different kind of optimisation problem.

Suppose we are given a set $S = \{s_1, s_2, ..., s_n\}$ where each item s_i has a positive benefit (or value) v_i and has a weight (or cost) w_i. Take v_i and w_i to be integers.

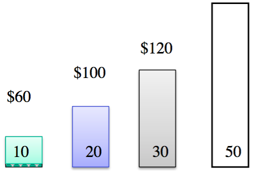
Suppose we want to choose a maximum-benefit subset that doesn't exceed a given weight $w_{\max}$ (like a thief who wants to load up his knapsack with valuables but can't carry more than W kilograms, and wants to know which items to pack in).

If we are restricted to entirely accepting or rejecting each item, we have the 0-1 Knapsack Problem. (Think of the thief loading up gold bars of various weights.)

If we are allowed to take fractions of items, we have the Fractional Knapsack Problem. (Think of bags of gold dust instead of solid bars of gold.)

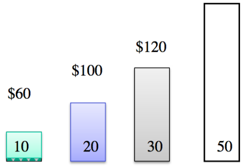
Fractional knapsack greedy algorithm

```
1: procedure FRACKNAP($S, V, W, w_{\max}$)
2:     Initiallise priority queue $Q$
3:     for each $s_i \in S$ do
4:         $p_i = \frac{v_i}{w_i}$                ▷ $p_i$ is normalised value
5:         $Q$.enqueue($s_i$) using $p_i$ as priority.
6:     current_weight $\leftarrow 0$
7:     set knapsack to empty
8:     while current_weight $< w_{\max}$ do
9:         $s_k \leftarrow$ Q.dequeue()
10:        $x_k \leftarrow \min(w_k, w_{\max} -$ current_weight$)$
11:        current_weight $\leftarrow$ current_weight $+ x_k$
12:        add $\frac{x_k}{w_k}$ of $s_k$ to knapsack
```



0-1 Knapsack problem

Let's try the greedy algorithm on the 0-1 knapsack problem.



In this version we have to put either all of an item, or none of an item in the knapsack. What is the greedy solution for the above problem? What is the optimal solution?

What about this problem: $W = [20, 30, 10, 5, 15, 25, 3, 17, 22, 31]$, $V = [100, 120, 60, 40, 20, 45, 23, 72, 102, 31]$, $W_{\text{max}} = 50$.

Optimal value is 265. Optimal items are (one-indexed) 1, 4, 7, 9.

Solving the 0-1 knapsack problem

Given: a set $S = \{s_1, s_2, ..., s_n\}$ where each item s_i has a positive benefit (or value) v_i and has a weight (or cost) w_i. Take v_i and w_i to be integers. A maximum total weight, $w_{\max}$.

Required: to choose a subset of S such that the total weight does not exceed $w_{\max}$ and the sum of the values v_i is maximal.

Let V be a 2D array storing the maximum value possible considering the first k items in S (call those k items, S_k), and maximum total weight w.

If $k \in S_k$ and we remove k from S_k (call it S_{k-1}), then the resulting set must be the optimum for the problem with a maximum total weight of $w - w_k$. Why?

Recursive non-greedy solution

We are left with the following observations:

- If there are no items in our set (S_0), then the maximum value is 0.
- If there is no space in our knapsack, then the maximum value is 0
- If the k^{th} item can't fit in the knapsack, then the maximum is the same as the maximum for $k-1$ items.
- Otherwise, the maximum is either:
  - the maximum without the k^{th} item in the optimal set, in which case we have a new problem with $k-1$ items and maximum weight w.
  - the maximum with the k^{th} item in the optimal set, in which case we have a new problem with $k-1$ items and maximum weight $w - w_k$.

Recursive non-greedy solution

So we can define our optimum $V[k, w]$ recursively as:

$$\begin{aligned} V[0, w] &= 0 \\ V[k, 0] &= 0 \\ V[k, w] &= V[k-1, w] \text{ if } w_k > w \\ V[k, w] &= \max(V[k-1, w], v_k + V[k-1, w-w_k]) \end{aligned}$$

The Algorithm

```
1: function RecursiveKnapsack(k, W, V, wmax)
2:     if k==0 or wmax $\leq$ 0 then return $0, \phi$
3:     if W[k]>wmax then                                  ▷ Can't fit k into knapsack
4:         return RecursiveKnapsack(k-1,W,V,wmax)
5:                                          ▷ Check the maximum value without k
6:     v1, items_not $\leftarrow$ RecursiveKnapsack(k-1,W,V,wmax)
7:                                             ▷ Check the maximum value with k
8:     v2, items_do $\leftarrow$ RecursiveKnapsack(k-1,W,V,wmax-W[k])
9:     v2 $\leftarrow$ v2 + V[k]                                ▷ add the value of item k
10:    items_do.add(k)                                     ▷ add item k to the list
11:    if v2>v1 then return v2, items_do                               ▷ do use k
12:    else return v1, items_not                                      ▷ don't use k
```

What is the complexity of this function?

Can we be more efficient?

In RecursiveKnapsack, W and V don't change, so the only things that change in different recursive calls are k and wmax. k can be any integer from 1 to n and wmax can be any integer from 1 to $w_{\max}$. So we should be able to produce a solution in $O(nw_{\max})$.

The reason why RecursiveKnapsack is so expensive is because it recomputes the same values over and over again. If we could store those values when they're computed and then retrieve them when needed, we could save ourselves a lot of computation.

This technique is called *memoisation*.

Alternatively, we could write an iterative version of the algorithm.

Memoised Version

```
1: initiallise global memo[n,$w_{\text{max}}$] as 2D array, set all to -1
2: function KnapMemo(k, W, V, wmax)
3:     if k==0 or wmax $\leq$ 0 then return $0, \phi$
4:     if memo[k,wmax]$\neq -1$ then return memo[k,wmax]
5:     if W[k]>wmax then memo[k,wmax] $\leftarrow$ KnapMemo(k-1,W,V,wmax)
6:     else
7:         v1, items_not $\leftarrow$ KnapMemo(k-1,W,V,wmax)
8:         v2, items_do $\leftarrow$ KnapMemo(k-1,W,V,wmax-W[k])
9:         v2 $\leftarrow$ v2 + V[k]; items_do.add(k)        $\triangleright$ add item k to the list
10:        if v2>v1 then memo[k,wmax] $\leftarrow$ v2, items_do    $\triangleright$ do use k
11:        else memo[k,wmax] $\leftarrow$ v1, items_not           $\triangleright$ don't use k
12:    return memo[k,wmax]
```

Iterative Version

```
function KnapIter(n, W, V, wmax)
    initiallise bestVal[n,wmax] as 2D array, set all to 0
    initiallise bestSet[n,wmax] as 2D array, set all to $\phi$
    for k $\leftarrow$ 1 to n do
        for w $\leftarrow$ 1 to wmax do
            if W[k]$>$w then
                bestVal[k,w] $\leftarrow$ bestVal[k-1,w]; bestSet[k,w] $\leftarrow$ bestSet[k-1,w]
            else
                withKVal $\leftarrow$ V[k] $+$ bestVal[k-1,max(w-W[k],0)]
                if bestVal[k-1,w] $>$ withKVal then
                    bestVal[k,w] $\leftarrow$ bestVal[k-1,w]; bestSet[k,w] $\leftarrow$ bestSet[k-1,w]
                else
                    bestVal[k,w] $\leftarrow$ withKVal
                    bestSet[k,w] $\leftarrow$ bestSet[k-1,max(w-W[k],0)]$+$k
    return bestVal, bestSet
```

Relevant parts of the textbook

Greedy algorithms are discussed in Chapter 16, but there isn't much on the knapsack problem although it is discussed at the end of Section 16.2.

The non-greedy solutions to the 0-1 knapsack problem are examples of dynamic programming algorithms. Dynamic programming is discussed in Chapter 15 and we will look at dynamic programming in more depth in the next two lectures.