Cosc 201
Algorithms and Data Structures
Lecture 20 (7/5/2025)
Greedy Algorithms

Brendan McCane
brendan.mccane@gmail.com

UNIVERSITY OF OTAGO
Te Whare Wānanga o Otāgo
NEW ZEALAND

Types of algorithms

- So far, we have seen these algorithms:
  - Merge sort (and quicksort)
  - Graph algorithms (breadth-first traversal, depth-first traversal)
  - Graph path algorithms (Dijkstra's algorithm, Prim's algorithm)
  - Tree algorithms (preorder, inorder, postorder, insert, delete, rotate)
- This is just a list of algorithms.
- It would be useful to characterise algorithms and problems into types.
- This might give us some hints about the sort of tools we can bring to bear when solving a problem.

Types of algorithms

Broadly, so far we have:

- Traversal algorithms (almost always $O(n)$). Can be recursive, or use a queue or a stack.
- Divide-and-conquer algorithms:
  - e.g. mergesort, quicksort
  - Often $O(n \log n)$ but not always.
  - Useful if we can split the problem into two equal parts and solve those parts independently, then merge the results.
- Greedy algorithms:
  - characterised by choosing the current best option to explore.
  - Dijkstra's algorithm and Prim's algorithm are two examples.
  - Often use a priority queue.

Compression

- Let's say we have the string "aaaabbcd" which we want to encode into the smallest number of bits.
- There are four different characters, so I might choose $a \to 00$, $b \to 01$ etc.
- There are 8 character in total, so that takes 16 bits.
- Can we do better?
- Hint: use variable length codes.

Huffman coding

- Well, we could choose $a \rightarrow 0$, $b \rightarrow 10$, $c \rightarrow 110$ and $d \rightarrow 111$.
- The string "aaaabbcd" can then be represented as "00001010110111". Which is only 14 bits.
- This is the best we can do for this string and is called a Huffman code.
- Why can't we use $c \rightarrow 11$, $b \rightarrow 110$ instead?
- That would only use 13 bits.
- The coding then would be "0000101011110".
- The resulting coding is ambiguous - consider the last 3 bits. Should that be a "d" or a "ca"?
- For variable-length codes, we must use no-prefix codes where no character encoding is a prefix of any other code.
- Confusingly, these sorts of codes are called prefix codes.

Huffman algorithm

- Huffman codes can be constructed in a greedy way. We need a min-priority queue and a binary tree.
- We also need to know the frequencies of each of the characters in the message.
- The basic algorithm is as follows:
  1. Create a leaf node for each character and its frequency, and insert all the nodes into a min-priority queue.
  2. While there is more than one node in the queue:
     - 2.1 Remove the two nodes of lowest frequency from the queue.
     - 2.2 Create a new internal node with these two nodes as children. The frequency of the new node is the sum of the frequencies of the two nodes.
     - 2.3 Insert the new node back into the queue.
  3. The remaining node is the root of the Huffman tree.
  4. Assign binary codes to each character by traversing the tree. Assign 0 to the left edge and 1 to the right edge.

Huffman example

Let's run the algorithm on with frequencies $(a, 4), (b, 2), (c, 1), (d, 1)$. The priority queue is on the left, and the tree is on the right.

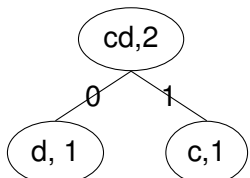
Q: [(d,1), (c,1), (b,2), (a,4)]

Tree empty.

Huffman example

Merge c and d, and insert back into Q

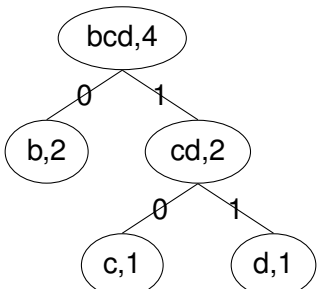
Q: [(cd,2), (b,2), (a,4)]



Huffman example

Merge cd and b, and insert back into Q

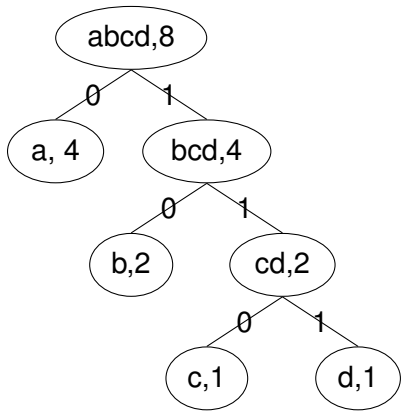
Q: [(cdb,4), (a,4)]



Huffman example

Merge bcd and a, and insert back into Q

Q: [(bcdb,8)]



Huffman coding discussion

- Is quite widely used and is optimal if codes are constrained to an integer number of bits.
- Arithmetic coding assigns codes based on fractions.
- Other codes use the context probabilities to encode symbols (e.g. the most likely next symbol given the current symbol).
- For one-off codes, you have to also transmit the character codes. This is usually only a small addition compared to the size of the message, but is still an extra cost.
- Compression ratios of 1:2 and 1:3 are common.

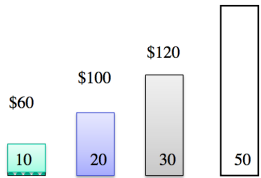
Knapsack problems

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: **end for**
7: current_weight $\leftarrow 0$
8: set knapsack to empty
9: **while** current_weight $< w_{\max}$ **do**
10: $s_k \leftarrow$ Q.dequeue()
11: $x_k \leftarrow \min(w_k, w_{\max} -$ current_weight$)$
12: current_weight $\leftarrow$ current_weight $+ x_k$
13: add $\frac{x_k}{w_k}$ of s_k to knapsack
14: **end while**
15: **end procedure**



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:

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

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 function returning the maximum value possible considering the first k items in S (call those k items, S_k), and available 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 ≤ 0 then return 0, φ
3:     end if
4:     if W[k]>wmax then                                   ▷ Can't fit k into knapsack
5:         return RecursiveKnapsack(k-1,W,V,wmax)
6:     end if                                              ▷ Check the maximum value without k
7:     v1, items_not ← RecursiveKnapsack(k-1,W,V,wmax)
8:                                                        ▷ Check the maximum value with k
9:     v2, items_do ← RecursiveKnapsack(k-1,W,V,wmax-W[k])
10:    v2 ← v2 + V[k]                                      ▷ add the value of item k
11:    items_do.add(k)                                     ▷ add item k to the list
12:    if v2>v1 then return v2, items_do                   ▷ do use k
13:    else return v1, items_not                           ▷ don't use k
14:    end if
15: end function
```

What is the complexity of this function?