Optimisation and Operations Research

Lecture 15: The Greedy Heuristic

Matthew Roughan
<matthew.roughan@adelaide.edu.au>
http://www.maths.adelaide.edu.au/matthew.roughan/notes/OORII/

School of Mathematical Sciences,
University of Adelaide

September 16, 2019

Section 1

Heuristics

Algorithms and heuristics

So far in this course, we have used *algorithms*:

- *e.g.*, Simplex

An algorithm precisely specifies a recipe for a computation.

A *heuristic* is a rule of thumb or an educated guess

- in our context they are rules that might lead to *good* solutions
- often based on simple intuition
- sometimes easier to code up
- often used when there isn't a fast-enough algorithm known

Algorithms and heuristics

A heuristic usually leads to an "algorithm"

In optimisation we often make the distinction that

- an *algorithm* is guaranteed to find the optimal solution
- a *heuristic* makes no guarantees
  - though we hope it will find a good solution
  - and it may find the optimal solution

We might even talk about a *meta-heuristic*, which is a general idea that can be converted into a heuristic for a particular problem, which leads to an "algorithm", sometimes an exact one, and other times not.

- *greedy* meta-heuristic
  $\Rightarrow$ Dijskstra's algorithm on shortest paths problem

Section 2

The Greedy Heuristic

The Greedy Heuristic

Iterate

- Create a set of feasible candidates/choices
  - local "move" from current solution
  - partial solutions (don't need to know all of the variables at once)
- Rate candidates by value (in terms of the objective)
- Choose the best

Stop when you run out of choices

The Greedy Heuristic

- Intuition
  - often an optimal solution has a few important pieces, and the rest are "noise"
  - greedy gets the important bits first
  - sometimes this is even guaranteed to find the optimal solution
- Bad bits
  - locally good decisions can be globally bad
  - method is short-sighted
  - go down a dead end and there isn't any way to go back

Examples

- Knapsack problem
- Coin Changing
- TSP
- Huffman coding
- Shortest paths

Knapsack problem [KV00]

Example (Knapsack problem)

A hiker can choose from the following items when packing a knapsack:

Item	1 chocolate	2 raisins	3 camera	4 jumper	5 drink
w_i (kg)	0.5	0.4	0.8	1.6	0.6
v_i (value)	2.75	2.5	1	5	3.0
v_i/w_i	5.5	6.25	1.25	3.125	5

However, the hiker cannot carry more than 2.5 kg all together.

Objective: choose the number of each item to pack in order to maximise the total value of the goods packed, without violating the mass constraint.

Knapsack problem in general

Integral knapsack problem

- we have a knapsack (backpack) which can take weight W
- we want to fit as much useful stuff into it as possible
  - maximize the value of the items contained in the knapsack
- each item i
  - has a weight w_i
  - has a value v_i (we want to maximise total value)
- we have one *indicator* variable, z_i, for each item
  - if we include the item, we say $z_i = 1$
  - otherwise $z_i = 0$
- summarizing

$$\max \left\{ \sum_i v_i z_i \middle| \sum_i w_i z_i \leq W, z_i = 0 \text{ or } 1 \right\}$$

Knapsack problem computational complexity

- The knapsack decision problem is NP-complete
  - the decision problem is:

    "Can we find an allocation with value at least V and weight less than W?"

- The knapsack optimisation problem (described above) is NP-hard
  - it is at least as hard as the decision problem
  - there are no known polynomial-time checks for optimality

Greedy knapsack heuristic (due to Dantzig)

1. Calculate the value to weight ratio v_i / w_i
2. Sort the items in decreasing order
3. For $i = 1..n$
    1. if there is room for item i, add it

Sorting is $O(n \log n)$, so this component dominates performance.

Knapsack problem variants

Very common (in different forms)

- fractional (allows fractions of items)
- unbounded (multi-items, *i.e.*, $z_i \in \mathbb{Z}^+$)
- multiple constraints: *e.g.*, volume and weight
- multiple knapsacks $\Rightarrow$ Bin-packing problem

Coin Changing Problem

Problem: given possible coins and banknotes pay an amount \$$z$ using the smallest number of coins and banknotes.

Example

Australian currency:

banknotes \$100, \$50, \$20, \$10, \$5;

coins \$2, \$1, 50c, 20c, 10c, 5c.

So \$105.50 can be paid (minimally) using $\$100 + \$5 + 50c$

General problem: given coins and banknotes of value c_i for $i = 1, \ldots, n$, then solve

$$\min \left\{ \sum_{i=1}^{n} x_i \;\middle|\; \sum_{i=1}^{n} x_i c_i = z, \; x_i \in \mathbb{Z}^+ \right\}$$

Where, x_i is the number of value c_i coins/banknotes.

Coin Changing Greedy Solution

Input: z, and (decreasing) coin values

$\mathbf{c} = (100, 50, 20, 10, 5, 2, 1, 0.5, 0.2, 0.1, 0.05)$

Output: $\mathbf{x}^* \in \mathbb{Z}^n$

```
i ← 1
x_i ← 0
while z > 0 do
    if c_i ≤ z then
        x_i ← x_i + 1
        z ← z − c_i
    else
        i ← i + 1
        x_i ← 0
    end
end
```

Algorithm 1: Greedy Coin Change

Coin Changing Greedy Solution

Example

Given currency $\mathbf{c} = (4, 3, 1)$ and $z = 6$

1. $i = 1,\ c_i = 4$
   1. $x_1 = 1,\ z = 2$
2. $i = 2,\ c_i = 3$
   1. $x_2 = 0,\ z = 2$
3. $i = 3,\ c_i = 1$
   1. $x_3 = 1,\ z = 1$
   2. $x_3 = 2,\ z = 0$

So greedy gives $\mathbf{x} = (1, 0, 2)$
Actual optimal solution is $\mathbf{x} = (0, 2, 0)$

Coin Changing Greedy Solution

- There are smarter ways to do this
  - add all of a particular coin you can in one go
    - complexity is $O(n)$, where n is number of coins
  - but I like the recursive nature of the above
- For *canonical* coins systems, greedy is optimal

Definition (Canonical Coin System)

A coin system is canonical if the greedy solution is always optimal.

  - US coins are canonical
  - Conditions to check if a system is canonical are involved
  - We could treat design of coin system as an optimisation in itself
- Frobenius coin problem is find the largest amount that *cannot* be obtained using only specified coins.
  - see also postage stamp problem and McNugget problem

Travelling salesperson problem (TSP)

Given a set of towns, $i = 1, \ldots, n$, and distances between the towns

$$D = [d_{ij}] = \begin{array}{c} \\ 1 \\ 2 \\ \vdots \\ i \\ \vdots \\ n \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & \cdots & j & \cdots & n \end{array} \\ \left[\begin{array}{cccccc} & & & \vdots & & \\ & & & \vdots & & \\ & & & \vdots & & \\ \cdots & \cdots & \cdots & d_{ij} & & \\ & & & & & \\ & & & & & \end{array} \right] \end{array}$$

Objective: construct a directed cycle of minimum total distance going through each town exactly once.

TSP Formulation

The decision is, basically, which links do we choose to use in the tour.

Letting $x_{ij} = \begin{cases} 1 & \text{if link } (i,j) \text{ is chosen} \\ 0 & \text{if link } (i,j) \text{ is not chosen} \end{cases}$, then we have

$$
\begin{array}{llcll}
(ILP) & \min d & = & \sum_{i=1}^{n} \sum_{j=1}^{n} d_{ij} x_{ij} & \\
\text{s.t.} & \sum_{j=1}^{n} x_{ij} & = & 1, \quad \forall i = 1, \ldots, n & \text{(only } \textit{one} \text{ link from i)} \\
& \sum_{i=1}^{n} x_{ij} & = & 1, \quad \forall j = 1, \ldots, n & \text{(only } \textit{one} \text{ link to j)} \\
& \sum_{i \in S} \left(\sum_{j \in S^c} x_{ij} \right) & \geq & 1, \quad \forall S \subset N & \text{(connectedness)} \\
& & & x_{ij} = 0 \text{ or } 1 \quad \text{for all } \quad i, j &
\end{array}
$$

This is an example of a classic 0–1 (Binary) Integer Linear Program.
The equations are linear, but the variables are integer (here binary).

TSP Alt Formulation

- The ILP describes links by n^2 binary variables x_{ij}
- We can write the same information much more concisely by describing the *tour* as a *permutation* of the integers $1, \ldots, n$.
  - a permutation is just lists the cities in some order
  - it's hard to write this formulation in ILP, but it can be easier to work with when programming
  - we often see this

TSP Greedy Heuristic

- Start at an arbitrary node (usually city 1)
- Choose the nearest town i_1 to city 1 as the second town
- Choose the nearest town i_2 to city i_1 as the third town
- And so on ...

Greedy doesn't work very well for the TSP, but it can provide an initial solution, which we can then improve.

Coding

- We have a "text" made up of a series of messages, or symbols

$$a, b, c, d$$

- We know the PMF (prob. mass function) of the messages

$$P(a), P(b), P(c), P(d)$$

- We want to have a binary code for each symbol, *e.g.*,

$$\begin{aligned} a &\leftrightarrow 00 \\ b &\leftrightarrow 01 \\ c &\leftrightarrow 10 \\ d &\leftrightarrow 11 \end{aligned}$$

- We want to minimise the average number of bits
  - in the example, the average is 2
  - can we do better?

Coding

- Imagine

$$
\begin{aligned}
P(a) &= 1/2 \\
P(b) &= 1/4 \\
P(c) &= 1/8 \\
P(d) &= 1/8
\end{aligned}
$$

- And we use the code

$$
\begin{aligned}
a &\leftrightarrow 0 \\
b &\leftrightarrow 10 \\
c &\leftrightarrow 110 \\
d &\leftrightarrow 111
\end{aligned}
$$

Average message length

$$\text{bits per word } = 1\frac{1}{2} + 2\frac{1}{4} + 3\frac{1}{8} + 3\frac{1}{8} = \frac{7}{4} < 2$$

How should we minimise code length in general?

Formalised coding problem

Objective: minimise the average code length

$$L = E\left[\ell\right] = \sum_{k=1}^{m} \ell_k p_k$$

where

$$\begin{aligned} \ell_k &= \text{length of } k\text{th code word} \\ p_k &= \text{probability of } k\text{th code work} \end{aligned}$$

Subject to the Kraft inequality (won't go into this here, but it's needed to make it possible to decode)

Huffman coding

1. We are building a tree
2. Start with each symbol in Ω as a leaf of the tree.
3. Repeat the following rule
    1. merge the two current nodes with the lowest probabilities to get a new node of the tree
4. The root is when we get a probability 1.

Huffman coding example 1

X	Probability
a	0.25
b	0.25
c	0.2
d	0.15
e	0.15

Huffman coding example 1

X	Probability		
a	0.25	→	0.25
b	0.25	→	0.25
c	0.2	→	0.2
d	0.15	→	0.3
e	0.15	↗ (to 0.3)	

Huffman coding example 1

X	Probability		
a	0.25 →	0.25 →	0.25
b	0.25 →	0.25 →	0.45
c	0.2 →	0.2 ↗	
d	0.15 →	0.3 →	0.3
e	0.15 ↗		

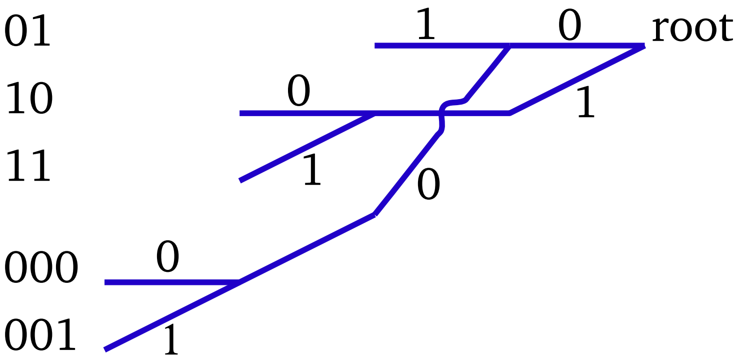
Huffman coding example 1

X	Probability			
a	0.25	0.25	0.25	0.55
b	0.25	0.25	0.45	0.45
c	0.2	0.2		
d	0.15	0.3	0.3	
e	0.15			

Huffman coding example 1

X	Probability				
a	0.25	0.25	0.25	0.55	1.0
b	0.25	0.25	0.45	0.45	
c	0.2	0.2			
d	0.15	0.3	0.3		
e	0.15				

Huffman coding example 1



- Read the codes from the root to the end point.
- Assign 0 to the branch with higher probability at each node.
  - this choice is arbitrary, but will mean we get consistent results

Huffman coding example 1

X	Probability	Codeword
a	0.25	01
b	0.25	10
c	0.2	11
d	0.15	000
e	0.15	001

Coding and Information Theory

Theorem

Huffman coding is optimal (in the sense that the expected length of its codewords is at least as good as any other code).

- Huffman coding is not so obviously greedy
  - we group the two smallest probabilities
  - roughly it is trying to grab as much *entropy* as it can each step
- There's a lot more to this topic
  - information theory
  - unique decodability
- But it's another example of a *good* greedy algorithm
  - and it's a real example (Huffman like codes are really used in many, many places)

Takeaways

- Heuristics are used to construct algorithms to attack difficult problems
  - not guaranteed to find optimal solution
  - but can often find good solutions to hard problems
- Greedy heuristic is one of the most common
  - very simple and easy to implement
  - works well for some problems
    - ★ when optimal solutions are sparse
    - ★ when optimal solutions are built up from optimal solutions to subproblems
  - works badly for others, but still might be used to construct an initial solution that we can build on

Further reading I

Bernhard Korte and Jens Vygen, *Combinatorial optimization*, Springer, 2000.