Introduction to Algorithms and Data Structures

Lecture 17: Introduction to Greedy Algorithms

Mary Cryan

School of Informatics
University of Edinburgh

Optimization

Sem1: data structures, searching, sorting, graphs and graph algorithms.

Sem2: harder computational questions; "optimization" not just yes/no, applications to grammar problems.

In optimization, we don't just want to ask yes/no questions ... we want to find the "best" (in some way) solution. These are optimization problems.

For a specific optimization problem, we aim to design an efficient algorithm that always computes the optimal solution on every possible input instance.

- "Efficient" usually means "polynomial-time" (in the size of the input instance).

One optimization problem we have seen so far is shortest paths.

Concept of "polynomial-time"

From Maths, we know the concept of a polynomial:

> *A polynomial is an algebraic expression composed of algebraic terms on variables and (constant) co-efficients, using $\times, +, -$.*
>
> *(In algebra, often have a single-variable polynomial over x)*

> *An algorithm is said to be polynomial-time if its running time is bounded above by a polynomial in its input size.*
>
> *(input size is usually written as n, but for some inputs may have extra size variables(s), eg, for graphs, we will have $m = |E|$ as well as $n = |V|$)*

Examples of polynomial-time running-times are $O(m + n)$ (BFS and DFS), $O(n \cdot \log(n))$ (Merge-Sort) and $O((m + n) \log(n))$ (Heap-based Dijkstra).

- We can allow the $\log(n)$ terms within "polynomial" as for upper-bounding, we could substitute with an extra n term.
- The exponents within a polynomial always need to be constant.

Algorithmic Paradigms

Divide and Conquer

Idea: Divide problem instance into smaller sub-instances of the same problem, solve these recursively, and then put solutions together to a solution of the given instance.

Examples: MergeSort, Quicksort.

Greedy Algorithms

Idea: Find solution by always making the choice that looks optimal at the moment — don't look ahead, never go back.

Example: Dijkstra's Algorithm.

Dynamic Programming . . . coming in Lecture 18 on Tuesday

None of these approaches is a "silver bullet" . . . will work for some optimization problems, but be sub-optimal on others.

Greedy Example 1: the coin changing" problem

In the UK coins have denominations 1p, 2p, 5p, 10p, 20p, 50p, £1 and £2.

A frequently-executed task in the retail sector involves taking an input value (say 88p) and calculating a collection of coins (may include duplicates) which will sum to that value.

We assume an unlimited supply of coins of each value.

The coin-changing problem

The coin changing problem is the problem, given an input value v ($v \in \mathbb{N}_0$) of calculating a collection of coins (of minimum cardinality) that will sum to v.

This is an *optimization* problem, as we want a solution with as few individual coins as possible (we want to *minimize* the number of coins handed back).

Want to do this for arbitrary systems of coin denominations (not just UK).

The coin-changing problem

Given: A value $v \in \mathbb{N}_0$, plus a sequence of coin values $c_1, c_2, \ldots, c_k \in \mathbb{N}_0$ (these representing the denominations of the relevant system).

Output: A multiset S of coins whose values sum to v, whose cardinality (size) of S is the minimum possible for v in this coin system.

Return solution as an array S of length k with $S[i]$ being the number of coins of value c_{i+1} for this optimal solution, for each $0 \leq i \leq k-1$.

- Both the value needed in change (v) and the system of coin denominations ($c_1, c_2, \ldots, c_k$) are part of the input.

  We want to solve the general case.

The Greedy approach for "coin changing"

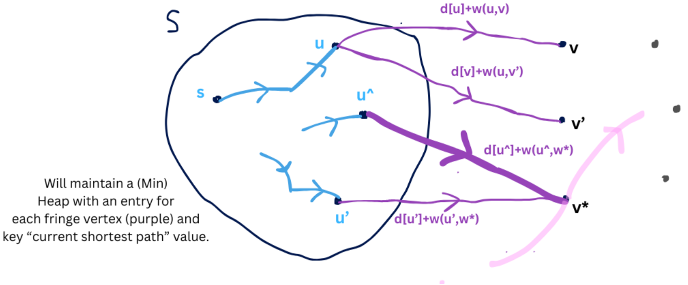
"largest coin first"

Iterate with current "leftover" value $\widehat{v}$:

> *Identify the largest coin value c_j such that $c_j \leq \widehat{v}$,*
> *add a c_i coin to the set, and re-iterate with $\widehat{v} \leftarrow \widehat{v} - c_j$.*

- The greedy algorithm always chooses the coin of max-value ($\leq$ than remaining value)
- A very natural heuristic which will work on many coin systems (eg UK)
- This is *not* guaranteed to be optimal for all systems - try the system with coin values $1, 5, 7$ for the value $v = 18$
- So Greedy does not give an optimal solution for the general case of coin changing.

Greedy Example 2: Dijkstra's Algorithm



Our iterative ("greedy") step for Dijkstra was to update $S \leftarrow S \cup \{v^*\}$ for the v^* with a fridge edge (u, v^*) which minimizes $d[u] + w(u, v^*)$ (over all current fringe edges).

We have yet to prove correctness!

Dijkstra's Algorithm: proof of correctness

A Greedy Heuristic is often easy/natural to define ... but often the proof of correctness takes some creativity (and rigorous argument).

The iterative nature (add one, add another, ...) of a Greedy algorithm means that often we will want to do an inductive proof.

Need to get the claim/invariant right, before attempting the proof.

For Dijkstra, we will prove the following invariant by induction:

> Invariant: *Before each iteration of the Dijkstra loop*[1]*, for every $u \in S$, $d[u]$ contains the value $d_G(s, u)$ of the shortest path value possible in G, and $\pi[u]$ is the predecessor vertex to u along that/a shortest path.*
>
> *(“$d[u]$ and $\pi[u]$ are correct for every u already added to S”)*

[1] Line 3 of DijkstraSimple, line 4 of Heap Dijkstra

Dijkstra's Algorithm: proof of correctness

proof: (by induction)

Base case: After the initialisation in 1. and 2, we have $S = \{s\}$ and $d[s] = 0$, as it should be. $\pi[s] = \text{NIL}$ is correct for the source s.

Induction step: Assume the invariant holds for S (Induction Hypothesis (IH)).

The easy case is if there are are no fringe edges from S. In this case the vertices in $V \setminus S$ are unreachable from s, and we are done. QED.

The key case to consider is when S does have fringe edges. Let (u^*, v^*) be the fringe edge with the current minimum $d[u^*] + w(u^*, v^*)$ value.

- We know that v^* is the fringe vertex that Dijkstra will add to S next.
- So we need to justify that this addition gives correct values for $d[v^*], \pi[v^*]$.

Suppose that in fact the assigned $d[v^*] \leftarrow d[u^*] + w(u^*, v^*)$ was NOT the optimal value of a shortest path from s to v^*.

Let p be a path of optimal distance $d_G(s, v^*)$ from s to v^* in G (we are supposing that $d_G(s, v^*) < d[u^*] + w(u^*, v^*)$).
We are going to derive a contradiction

Dijkstra's Algorithm: proof of correctness

Think about how the supposedly better path p moves from s to v^* in G.

- It has to travel from inside S (s) to outside S (v^*)
- So if we draw the connected path p from s to v^* it has to "break out" of S on some fringe edge.
- We focus on the first fringe edge along the path p, suppose it is $(\widehat{u}, x)$.

Now think about the prefix path $p_{|s \to x}$ of p starting at s, with final edge $(\hat{u}, x)$.

- This prefix path has distance *at most* $d_G(s, v^*)$ (no negative weights in G).
- Hence $(\widehat{u}, x)$ is a fringe edge for S with candidate path value $d[\widehat{u}] + w(\widehat{u}, x)$ which is $\leq d_G(s, v^*)$ and strictly less than $d[u^*] + w(u^*, v^*)$.

This is a CONTRADICTION to our choice of (u^*, v^*)!!
If this had been the case , we would have added x (not v^*) to S.

Hence $d[u^*] + w(u^*, v^*) = d_G(s, v^*)$ must have been true (and $\pi[v^*] \leftarrow u^*$ is also valid). QED

proving correctness for Greedy Algorithms

The step of proving correctness for Greedy Algorithms is essential.

- Greedy Heuristics tend to "feel right" and are often the first approach you will try when trying to come up with a solution.
- However, they don't always give optimal results.

Example 1: The Greedy Heuristic doesn't give optimal results for coin changing.

Example 2: Even for Single-Source Shortest Paths (SSSP), the correctness depended on us defining the right measure to choose between fringe vertices.

(an alternative "greedy" choice might have been to just choose the fringe edge with minimum $w(u, v^*)$, for example. This version of Greedy does not give an optimal solution to SSSP)

Reading

[Roughgarden] "Algorithms Illuminated":

- 13.1 discusses the general concept of a Greedy Algorithm.
- 9.3 gives a proof of correctness for Dijkstra.
- Section 6.2 of Kleinberg+Tardos has a similar discussion of general Dynamic Programming principles.

[CLRS] (ed 4):

- 15.1 general content on Greedy Algorithms
- 22.4 proof of Dijkstra