Scheduling Problem with Greedy Solution

Setting:

- a number of customers, waiting to be served
- only one customer can be served at a given time
- it takes time $T(j)$ to serve customer j

Goal (as all customers equally important):

minimize sum of waiting times

Optimal Strategy:

if $T(j) < T(k)$ *then serve* j *before* k

Proof of optimality:

- consider a schedule S not following that strategy
- in S, there are two jobs j and k such that
  - j comes just before k
  - but $T(k) < T(j)$
- Hence S can be improved, by swapping j and k.
- Thus S is not optimal.

Greedy Algorithms

We saw a Greedy algorithm for the scheduling problem:

- it needs not explore search spaces, or backtrack
- but it knows where to set its foot next.

Many problems, but not all, allow greedy solutions; the proof of correctness often argues

> *a schedule not following strategy can be improved*

but observe that local optima are not necessarily global:

> *a schedule not improvable by "local" changes may still not be optimal*

Example: list the numbers $1..n$ so as to minimize sum of neighbor distances, where the distance between i and j is

$$\min(|i - j|, 5)$$

An optimal listing is $1..n$ (or $n..1$), with cost $n - 1$. But

$$6\ 7\ 8\ 9\ 10\ 1\ 2\ 3\ 4\ 5$$

is not optimal but swapping two numbers makes it worse.

Event Scheduling

Setting:

- one server which can host only one event at a time
- some events, each with fixed start & finish time

Goal (as all events equally important):

maximize the number of events allocated

Attempts at greedy strategy:

1. prioritize the shortest events
2. prioritize the events with fewest overlaps

For each strategy, there is a a counterexample that shows that it is not always optimal

Event Scheduling with Greedy Solution

Recall Setting:

- one server which can host only one event at a time
- some events, each with fixed start & finish time

Goal:

maximize the number of events allocated

Greedy strategy:

prioritize events that finish early

there is no counterexample as correctness can be proven:

- if $E_1..E_n$ is valid schedule
- and $E'_1..E'_m$ follows strategy
- then $m \geq n$ and E'_n will not finish later than E_n.

Another correct greedy strategy is the symmetric:

prioritize events that start late

Thus scheduling order may not equal execution order

Minimum-Cost Spanning Tree

Setting:

- undirected connected graph $G = (V, E)$
- where each edge has a weight > 0

a Spanning Tree is a subset E_0 of E that

- spans: each $v \in V$ is end point of some edge in E_0
- is a tree: (V, E_0) is acyclic and connected

Goal: find spanning tree with minimum total weight

- all spanning trees have $|V| - 1$ edges
- there may be many spanning trees
- and even more than one minimum-cost spanning tree

We shall present two greedy algorithms for finding one minimum-cost spanning tree.

Select an Edge with No Regret

Common Strategy: build minimal spanning tree by

- adding one edge at a time
- such that we will never regret our choice.

With T_0 the current edges, we can inductively assume

T_0 can be extended to a minimum spanning tree.

When can an edge e be added to T_0 with no regret?

- suppose the nodes can be partitioned into 2 or more components with all edges in T_0 intra-component
- let E' be the inter-component edges (observe that $E' \neq \emptyset$ and $T_0 \cap E' = \emptyset$).

Lemma:

- if $e \in E'$ and no other edge $\in E'$ has smaller weight
- then it is safe to add e to T_0, in that $T_0 \cup \{e\}$ can be extended to a minimum-cost spanning tree.

Proof: lecture notes

Kruskal's algorithm

Kruskal's algorithm builds tree T_0 piecemeal:

- we keep track of connectivity components wrt T_0
- thus all edges in T_0 are intra-component
- and it is safe to add an inter-component edge with lowest weight.

Accordingly, Kruskal's algorithm first sorts the edges with lowest weight first, and then for each edge e:

- checks if its end points are in different components
- if so, it adds e to T_0

Running time (with $n = |V|$ and $a = |E|$):

1. sorting of edges: $\Theta(a \lg(a)) = \Theta(a \lg(n))$
2. we must a times check if two nodes are in different components, and if so merge those components; by union-find trees each can be done in time $O(\lg n)$.

Thus the total running time is in $\Theta(a \lg(n))$.

Prim's algorithm

Prim's algorithm lets the tree T_0 grow from a single node

- the nodes in T_0 form one component
- all other nodes form another component

To find an inter-component edge with lowest weight, we thus need for each node w not in T_0 to keep track of

> *the minimum weight of an edge from w to a node in T_0, together with one such edge, called $e(w)$*

To implement that, at least two feasible approaches:

- priority queue of nodes, where w has high priority if $e(w)$ has low weight
- no priority queue; instead rely on linear search.

Prim's algorithm, Running time Analysis

Two operations contribute to the total cost:

1. find & extract w with minimum $e(w)$:
   - with linear search: in $\Theta(n)$
   - with priority queue: in $\Theta(\lg n)$

   number of times this is done: $\Theta(n)$
2. update $e(w)$ when u added to T_0 and there is edge (u, w):
   - with linear search: in $\Theta(1)$
   - with priority queue: in $\Theta(\lg n)$

   number of times this is done: a

Thus the total running time is

- with linear search: $\Theta(n^2 + a) = \Theta(n^2)$
- with priority queue: $\Theta(n \lg(n) + a \lg(n)) = \Theta(a \lg(n))$

Single Source Shortest Paths

Given a directed graph where edges have length, find

the smallest total length of a path from s to u

- for specific source s
- for each node u

This can be accomplished through Dijkstra's algorithm.

- while negative cycles would make no sense
- edges with negative length
  - can be handled by Floyd's algorithm
  - but not by Dijkstra's algorithm

  There are algorithms (such as Bellman-Ford) for single source shortest paths that do allow negative lengths, but they are generally slower than Dijkstra's.

Dijkstra's Algorithm

Dijkstra's algorithm uses

- a table d that maps each node to a number ≥ 0
- a subset S of the nodes

and maintains the invariant:

1. if $v \in S$ then $d[v]$ is the smallest length of a path from the source to v (and there will be a path of that length where all nodes belong to S)
2. if $v \notin S$ then $d[v]$ is the smallest length of a path from the source to v where all nodes but v are in S

Key insight:

- if $d[v]$ is minimum among nodes not in S
- then it is safe to add v to S
  (this relies on lengths not being negative)

Dijkstra's Algorithm, Initialization

Recall the invariant:

1. if $v \in S$ then $d[v]$ is the smallest length of a path from the source to v (and there will be a path of that length where all nodes belong to S)
2. if $v \notin S$ then $d[v]$ is the smallest length of a path from the source to v where all nodes but v are in S

which is established by

$S \leftarrow \emptyset$
foreach node X
  if X is the source
    $d[X] \leftarrow 0$
  else
    $d[X] \leftarrow \infty$

Dijkstra's Algorithm, Main Loop

Recall the invariant:

1. if $v \in S$ then $d[v]$ is the smallest length of a path from the source to v (and there will be a path of that length where all nodes belong to S)
2. if $v \notin S$ then $d[v]$ is the smallest length of a path from the source to v where all nodes but v are in S

which is maintained by

> **while** there is a node $\notin S$
>   let $Y \notin S$ be a node with $d[Y]$ minimal
> $S \leftarrow S \cup \{Y\}$
>   **foreach** edge $Y \xrightarrow{q} Z$ with $Z \notin S$
>   **if** $d[Y] + q < d[Z]$
> $d[Z] \leftarrow d[Y] + q$
>   record: path to Z is thru Y

In 1st iteration, Y will be the source.

Dijkstra's Algorithm, Running Time

The analysis is as for Prim's algorithm:

1. to find & extract Y with minimum $d(Y)$:
   - with linear search: in $\Theta(n)$
   - with priority queue: in $\Theta(\lg n)$

   number of times this is done : $\Theta(n)$
2. update $d(Z)$ when Y with edge $Y \to Z$ added to S:
   - with linear search: in $\Theta(1)$
   - with priority queue: in $\Theta(\lg n)$

   number of times this is done: $\Theta(a)$

Thus the total running time is

- with linear search: $\Theta(n^2 + a) = \Theta(n^2)$
- with priority queue: $\Theta(n \lg(n) + a \lg(n)) = \Theta(a \lg(n))$

	Dijkstra's algorithm	differs from Prim's:
graph is	directed	undirected
computes	shortest paths	minimum spanning tree
maintains	distance from source	distance to current tree

Single/All Shortest Path

Source	Target	Algorithm	Running Time
Single	All	Dijkstra's	$\Theta(a \lg(n))$ or $\Theta(n^2)$
All	Single	Dijkstra's (reverse edges)	$\Theta(a \lg(n))$ or $\Theta(n^2)$
All	All	Floyd's n times Dijkstra's	$\Theta(n^3)$ $\Theta(an \lg(n))$ or $\Theta(n^3)$
Single	Single	Dijkstra's (early exit?)	NOT $\Theta(n)$

Fractional Knapsack

Recall the general knapsack problem:

- n items; each item i has weight w_i and value v_i
- weight limit (capacity) of W

Goal: find $x_1..x_n$ that

- satisfies $\sum_{i=1}^{n} x_i w_i \leq W$
- maximizes $\sum_{i=1}^{n} x_i v_i$

We have already looked at the binary version:

each x_i is either 0 or 1

We shall now consider the fractional version:
each x_i is a real number with $0 \leq x_i \leq 1$.

Which version is easier to solve?

- the binary can be solved by brute force
- the fractional cannot be solved by brute force
- but ...

Fractional Knapsack, Greedy Solution

Consider the example, with $W = 100$:

i	1	2	3	4	5
w	10	20	30	40	50
v	20	30	66	40	60

We may give priority to items with

1. higher value: $x = (0, 0, 1, 0.5, 1)$ giving 146
2. lower weight: $x = (1, 1, 1, 1, 0)$ giving 156
3. higher $\frac{v_i}{w_i}$: $x = (1, 1, 1, 0, 0.8)$ giving 164

Assessment:

- Strategies 1 and 2 are not always optimal
- Strategy 3 is indeed always optimal (proof in notes)

Is strategy 3 optimal also in the binary setting?

no, as showed by simple counterexample
where we are forced to take "too much"

Binary Encodings of Alphabets

We want to transmit sequences of symbols where each symbol α comes with a frequency $F(\alpha)$. Example:

α	a	b	c	d
$F(\alpha)$	0.4	0.1	0.3	0.2

An encoding B maps each symbol into a bit string, allowing us to encode a sequence of symbols:

$$B(\alpha_1..\alpha_n) = B(\alpha_1)..B(\alpha_n)$$

Goal: find B that for random sequence s produces

minimum expected length of $B(s)$

Naive approach: Fixed length encoding, such as

α	a	b	c	d
$B(\alpha)$	00	01	10	11

If $|s| = n$ then $|B(s)| = 2n$

Optimal Encodings of Alphabets

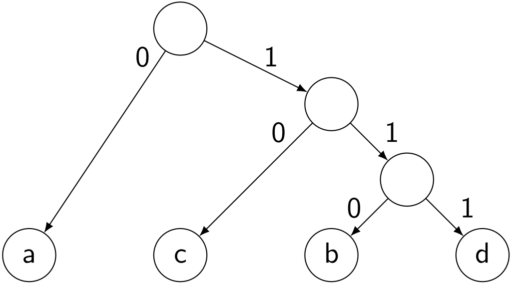
Variable length encoding:

α	a	b	c	d
$F(\alpha)$	0.4	0.1	0.3	0.2
$B(\alpha)$	0	110	10	111

If $|s| = n$ then we may expect $|E(s)|$ to be:

$$1 \cdot 0.4n + 3 \cdot 0.1n + 2 \cdot 0.3n + 3 \cdot 0.2n = 1.9n$$

To avoid ambiguity, no code has another code as prefix and thus B can be represented as binary tree:



Greedy Algorithm for Optimal Encoding

Huffman's algorithm:

1. initially, each symbol α forms a singleton tree, with frequency $F(\alpha)$
2. in each step, find trees t_1 and t_2 with $F(t_1)$, $F(t_2)$ minimal, and form tree t with t_1 and t_2 as children, and with $F(t) = F(t_1) + F(t_2)$

To prove optimality, a key observation is:

- if T is an optimal encoding
- then there exists (another) optimal encoding T' where the two least frequent symbols are siblings.