CS 341: ALGORITHMS (F19) — LECTURE 7
GREEDY ALGORITHMS II
ERIC BLAIS

We saw in the last lecture how the greedy algorithm method can be used to solve the interval scheduling and minimizing lateness problems. In this lecture, we will examine a few other problems that can be solved by greedy algorithms and see how a common exchange method can be used to establish the correctness of these algorithms.

1. Interval colouring

There are some situations where we can prove the correctness of the greedy algorithm directly (or with a structural proof) instead of having to use either the always-ahead or the exchange argument. One such case is with the interval colouring problem.

**Definition 7.1 (Interval colouring problem).** Given a set of $n$ pairs of start and finish times $(s_1, f_1), \ldots, (s_n, f_n)$ where each pair $(s_i, f_i)$ satisfies $s_i < f_i$, find a colouring of the intervals with as few colours as possible such that no two intervals that overlap share the same colour.

For example, here is a colouring of a set of 9 intervals that satisfies the condition that no two overlapping intervals share the same colour:

Note that in this case the colouring is not optimal because there is another colouring that uses only 3 colours.

There is a simple greedy algorithm that solves the interval colouring problem:

- Sort the intervals by start time.
- At each interval $i$, if there is a colour $c$ that was already used previously and is not assigned to any interval overlapping $i$, assign that colour to $i$; otherwise, assign a new colour to $i$.

By construction, this algorithm guarantees that the overlapping intervals will always be assigned distinct colours. But how can we argue that it requires the minimum number of colours? This is where a direct argument can be used:

**Theorem 7.2.** Let $d$ be the maximum number of intervals that cover any given point on the line. Then any legal colouring of the intervals requires at least $d$ colours and that is exactly the number of colours used by the greedy algorithm.
Proof. Let $p$ denote a point that is covered by $d$ intervals. Since all $d$ intervals that cover $p$ overlap each other, any colouring of the intervals that assigns distinct colours to overlapping intervals must use at least $d$ colours.

And since at most $d$ intervals cover any point, when we reach the start point of an interval, at most $d - 1$ other intervals cover that point, so if $d$ colours have already been used previously, at least 1 of them is available to colour the current interval. \[\square\]

2. Fractional knapsack

The fractional knapsack problem aims to find the maximum value of items that we can fit in a backpack, when we can subdivide items into any fractional parts.

Definition 7.3 (Fractional knapsack). Given a set of $n$ items that have positive weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$, as well as a maximum weight capacity $W$ of the knapsack, find a set of amounts $x_1, \ldots, x_n$ of each item that you put in your backpack that maximizes the total value

$$\sum_{i=1}^{n} \frac{x_i}{w_i} v_i$$

among all possible sets of amounts where for each item $i$ we have $0 \leq x_i \leq w_i$ and the total weight taken is $\sum_{i=1}^{n} x_i \leq W$.

For example, an instance may have three items with weights and values

<table>
<thead>
<tr>
<th>$w_i$</th>
<th>$v_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

and a total knapsack capacity weight $W = 6$.

There is a natural greedy algorithm for this problem: sort the items by decreasing relative value $v_i/w_i$, then consider each element in turn and put as much of it in the knapsack as can fit.

\begin{algorithm}
Order the items by decreasing value of $v_i/w_i$;
\begin{for} [i = 1, \ldots, n] do
\begin{set} $x_i \leftarrow \min\{W, w_i\}$;
\begin{set} $W \leftarrow W - x_i$;
\end{set}
\end{set}
\end{for}
\return $x_1, \ldots, x_n$;
\end{algorithm}

Theorem 7.4. The GreedyKnapsack algorithm solves the fractional knapsack problem.

Proof. Let $y_1, \ldots, y_n$ be a valid solution to the fractional knapsack problem, using the ordering defined by the algorithm where $v_i/w_i$ is decreasing. And let $x_1, \ldots, x_n$ be the solution returned by the GreedyKnapsack algorithm. We prove that $x_1, \ldots, x_n$ is a valid solution by induction on the number of indices $i \leq n$ for which $x_i \neq y_i$.

In the base case, when $x_i = y_i$ for each $i = 1, 2, \ldots, n$, then $x_1, \ldots, x_n$ is the same solution as $y_1, \ldots, y_n$ so it is a valid solution.

For the induction step, let $m = |\{i \leq n : x_i \neq y_i\}| \geq 1$ be the number of differences in the solution. By the induction hypothesis, if there is a valid solution $y'_1, \ldots, y'_n$ with
Let $k \leq n$ be the smallest index where $x_k \neq y_k$. Then it must be that $x_k > y_k$ since GreedyKnapsack maximizes the value of $x_k$. And since $\sum x_i = \sum y_i$, there must be an index $\ell > k$ for which $y_\ell > x_\ell$. Let’s exchange a $\delta$ amount of weight of element $\ell$ for element $k$ to obtain the solution $y'$ where

$$y'_k = y_k + \delta \quad \text{and} \quad y'_\ell = y_\ell - \delta.$$  

Choose $\delta = \min\{x_k - y_k, y_\ell - x_\ell\}$ so that $y'_k = x_k$ or $y'_\ell = x_\ell$ and, therefore, $|\{i \leq n : x_i \neq y'_i\}| < m$. To complete the proof, we must show that $y'_1, \ldots, y'_n$ is a valid solution. The difference in the total value of the solutions $y'$ and $y$ satisfies

$$\delta(v_k/w_k) - \delta(v_\ell/w_\ell) = \delta(v_k/w_k - v_\ell/w_\ell).$$

The ordering of the elements guarantees that $v_k/w_k \geq v_\ell/w_\ell$ so that the total value of $y'$ is at least as large as that of $y$ and, therefore, $y'$ is a valid solution. □

What if we now consider the variant of the problem where we are only allowed to choose $x_i \in \{0, w_i\}$? (I.e., where the items are indivisible.) Does the greedy algorithm above still work? You should convince yourself that it is no longer correct—and you should be able to see how the exchange argument we used above fails in this situation! We will revisit this version of the knapsack problem in the next section, when we see how the dynamic programming algorithm design technique can be used to solve it efficiently when $W$ is not too large. (And we will see the problem again in the NP-completeness section, when we will see why we shouldn’t expect to find an efficient algorithm for this problem in general.)

### 3. Bonus: Offline caching

**Note:** You are not responsible for the material in this section. It was not covered in class and is included here just for interest.

A cache in a computer system is a small but very fast block of memory that enables significant speedup on the data that we store on it. But how do we decide what data to keep in the cache? To define this problem, we introduce a bit of notation. The units of data in memory are called *pages*. A cache can contain up to $k$ pages; we consider the setting where the total number of pages is much larger than $k$. A sequence of page requests is a list of page identifiers. A page request incurs a cost of 0 if the page is currently in the cache; otherwise it incurs a cost of 1 as the page needs to be brought into memory—this event is known as a page fault. If the cache already contains $k$ pages at the moment of a page fault, one of the $k$ current pages must be evicted—removed from the cache—to make room for the requested page.

In the (idealized) setting where we know ahead of time exactly the entire sequence of page requests, the problem of minimizing the cost of serving all the page requests is known as offline caching.

**Definition 7.5** (Offline caching problem). Given a cache of size $k$ and a sequence of $k$ page requests, determine which page should be evicted (if any) at each page fault to minimize the cost of serving all the page requests in the sequence.

As an example, if we label the pages $a, b, c$ and consider the setting where the cache has size $k = 2$, the page request sequence

$$a \ b \ c \ b \ c \ b \ a \ c$$
can be served with cost 4 (i.e., with 4 page faults in total). We can see this by drawing the contents of the cache after each request (with element that is newly inserted in red).

Requests: \[ a \ b \ c \ b \ c \ b \ a \ c \]

Cache: \[ a \ a \ c \ c \ c \ c \ c \ c \ b \ b \ b \ b \ b \ a \ a \]

In this example, 4 page faults is optimal: there is no alternative way that we could have chosen to keep other elements in the cache that would result in fewer faults. But can we design an algorithm that will always guarantee that it serves the page requests with minimum cost? There are in fact many reasonable greedy algorithms for the problem that we can examine:

- **LRU**: (Least-recently-used) Evict the page that has been accessed least recently.
- **FIFO**: (First-in-first-out) Evict the page that was added to the cache least recently.
- **LFU**: (Least-frequently-used) Evict the page that has been accessed least frequently.
- **LFD**: (Longest-forward-distance) Evict the page that will be accessed the furthest in the future.

The first three of these options are popular because they are online strategies: they do not require us to know the entire sequence of page requests that will be coming in the future. But, as you can check by constructing counter-examples, none of those three can guarantee that they will always minimize the number of page faults. The LFD algorithm, however, does offer this guarantee and solve the offline caching problem.

**Theorem 7.6.** The greedy LFD algorithm always minimizes the number of page faults.

**Proof sketch.** We again want to use an exchange argument.

Let \( C_1, \ldots, C_n \) be the contents of the cache of any sequence of page request services that minimizes the total number of page faults. Let \( G_1, \ldots, G_n \) be the contents of the cache when we run the LFD algorithm. The notion of “closeness” between the two sequences we use is the minimum index \( j \) where \( C_j \neq G_j \). In this proof, we want to show that given \( C \) and \( G \), we can always construct a new optimal algorithm whose cache contents \( C'_1, \ldots, C'_n \) will satisfy \( C'_j = G_1, \ldots, C'_j = G_j \).

Let \( a = C_j \setminus G_j \) be the element kept in the cache of the optimal algorithm but expelled from the greedy algorithm’s cache at request \( j \), and let \( b = G_j \setminus C_j \) be the element that the greedy algorithm kept in the cache instead. We build \( C' \) by copying the greedy algorithm until step \( j \), then after that we continue expelling the same elements as the optimal algorithm \( C \) does for the requests \( j + 1, \ldots, k - 1 \) where \( k \) is the first request in which either

- \( C \) evicts \( a \); or
- \( C \) brings \( b \) back into the cache by evicting some other page \( c \).

The key observation is that the first step where we can’t mimic the optimal algorithm must be of one of these two forms; in general it could also be possible that we get a page request for \( a \) before either of the above two types of requests occur, but by definition of our greedy algorithm, this can only happen after there was a page request for \( b \) (the second case above).

Now if at time \( k \) the optimal algorithm evicted \( a \), we evict \( b \) from \( C' \). And if instead at request \( k \) the optimal algorithm evicts \( c \) to bring \( b \) back into the cache, we evict \( c \) and bring back \( a \) into the cache \( C' \). In both cases, we then obtain \( C'_k = C_k \) and we can continue following the optimal algorithm to handle the requests \( k + 1, \ldots, n \). \( \square \)
Note that the above argument is not a complete proof, for a slightly technical reason: the algorithm we constructed to have cache $C'$ doesn’t exactly follow the rules of cache algorithms we defined earlier. In the case where at time $k$, the optimal algorithm evicted $c$ and brought $b$ into the cache, we made $C'$ also replace an element from its cache as well—but in fact for this request $C'$ does not get a page fault (because it already has $b$ in cache) so it does not get to update its cache content.

**Exercise 7.1** (For interested students). Complete the proof of correctness of the LFD algorithm by either (i) modifying the definition of cache algorithms to allow updates even on non-page faults, or (ii) showing how we can turn $C'$ into a “proper” cache algorithm that updates its cache only on page faults without changing the contents of $C'_1, \ldots, C'_j$. 