

Introduction to Algorithms and Data Structures

Lecture 18: “Seam Carving” and Edit distance (via Dynamic Programming)

Mary Cryan

School of Informatics
University of Edinburgh

Seam Carving

We need to fit images into new dimensions (relevant for tablet/mobile layouts).

Naïve approaches to adapting to varying dimensions include cropping and scaling. Both have their flaws - see video by Shai Avidan and Ariel Shamir: https://www.youtube.com/watch?v=6NcIJXTlugc&feature=emb_logo

A better, more flexible, approach is to search for **seams** in the image, a **seam** being a connected sequence of pixels running from top-to-bottom (*vertical*) or from left-to-right (*horizontal*).

- ▶ More general than deleting a column.
- ▶ Seams can be deleted (or duplicated) and the rows and columns of the altered image will have uniform lengths 1 less (or more) than before.
- ▶ For re-sizing images, we will want to find seams of low-energy (where there is little difference between the seam pixels and their surrounding pixels).

Seam Carving

We are given an image $\mathbf{I} : [m] \times [n]$ where each pixel is a colour (maybe RGB). Our dimensions are $m \times n$ but we want to fit to different dimensions $m' \times n'$.

Definition

In a image \mathbf{I} of dimensions $m \times n$, we define a **vertical seam** to be any sequence

$$\mathbf{s} = j_1, \dots, j_m \in [n]$$

such that for every $i \in [m] \setminus \{1\}$, we have $|j_i - j_{i-1}| \leq 1$ and $2 \leq j_i \leq n - 1$ (don't touch left/right sides).

A **horizontal seam** is any sequence

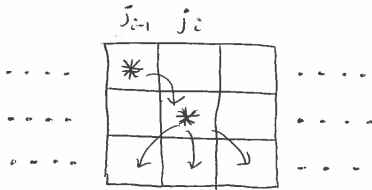
$$\mathbf{s} = i_1, \dots, i_n \in [m]$$

such that for every $j \in [n] \setminus \{1\}$, we have $|i_j - i_{j-1}| \leq 1$ and $2 \leq i_j \leq m - 1$.

(recall $[k]$ is the set of values $\{1, \dots, k\}$)

Seams and Energy

In building a (vertical) seam we are allowed to move (straight) down, 1-pixel left, or 1-pixel right.



"8-connected window for constructing a seam"

We need to evaluate the **energy** of a pixel, as we prefer low-energy seams.

We assume some energy function $e = e_I$ applied to pixels of that image.

Then

$$e(s) =_{\text{def}} \begin{cases} \sum_{i=1}^m e_I(i, j_i) & \text{s is a vertical seam} \\ \sum_{j=1}^n e_I(i_j, j) & \text{s is a horizontal seam} \end{cases}$$

Energy functions

Many options for (pixel) energy function, often a (local) **gradient** score.

- ▶ L_1 gradient scoring (for pixel (i,j)) can be written as

$$e_I(i,j) =_{\text{def}} \left| \frac{\partial}{\partial x} I \right|_{i,j} + \left| \frac{\partial}{\partial y} I \right|_{i,j}.$$

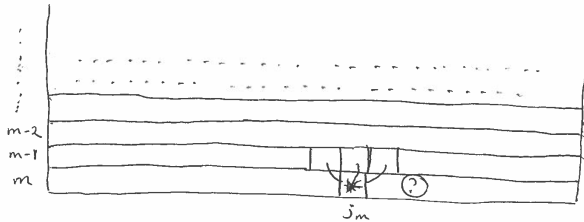
- ▶ $\frac{\partial}{\partial x}, \frac{\partial}{\partial y}$ are defined in the image processing context:
 - ▶ For example the **Sobel operators** can be used to calculate $\frac{\partial}{\partial x}$ and $\frac{\partial}{\partial y}$:

$$\begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \qquad \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

- ▶ The first (vertical) would calculate $\frac{\partial}{\partial x} |_{i,j}$ as $(I_{i-1,j+1} - I_{i-1,j-1}) + 2(I_{i,j+1} - I_{i,j-1}) + (I_{i+1,j+1} - I_{i+1,j-1})$.
- ▶ A colour image will have 3 color channels, so 3 $\frac{\partial}{\partial x}$ and 3 $\frac{\partial}{\partial y}$ scores to sum.

Computing an optimal seam

We assume we are looking for a vertical seam (without loss of generality).
Take a recursive view and understand the optimal seam in terms of (slightly) shorter seam



Suppose an optimal vertical seam ends at pixel (m, j_m) .

- then we know that j_{m-1} was either j_m , j_{m-1} , or j_m+1 .
- depends on which of $(m-1, j_m)$, $(m-1, j_{m-1})$ and $(m-1, j_m+1)$ is the bottom endpoint of the best $(m-1)$ -length vertical seam.
- three subproblems of length $(m-1)$.

Recurrence for optimal (vertical) seam

Assume that we have precomputed $e_I(i, j)$ for every pixel $1 \leq i \leq m$, every $1 \leq j \leq n$ ($\Theta(1)$ time for each (i, j)).

($e_I(i, j)$ slightly different for top/bottom rows)

Definition

For every i, j , $1 \leq i \leq m$, $1 \leq j \leq n$, we define $opt_I(i, j)$ to be the cost of the minimum-cost vertical seam from (somewhere in) row 1 to pixel (i, j) , where cost is scored as at the end of slide 4.

We have the following recurrence:

$$opt_I(i, j) = e_I(i, j) + \begin{cases} 0 & \text{if } i = 1 \\ \min\{opt_I(i-1, j-1), \\ opt_I(i-1, j), \\ opt_I(i-1, j+1)\} & \text{if } i > 1 \end{cases}$$

(we will set $e_I(i, j) \leftarrow \infty$ if $j = 1$ or n)

Dynamic programming implementation

We will need a table/array of size $m \cdot n$, let the table be opt .
(I assume indexing starts at 1 for this algorithm)

- ▶ Entry $opt[i, j]$ will store the value of $opt_I(i, j)$ (when we have computed it).
- ▶ We will need to have the local $e_I(i, j)$ energy values pre-computed (for each $1 \leq i \leq m, 1 \leq j \leq n$) and stored in a table e (of dimensions $m \times n$). We will set $e[i, 1] \leftarrow \infty, e[i, n] \leftarrow \infty$ for all $i \in [m]$ to make sure the seam avoids the sides.
- ▶ For image processing applications we definitely need to know the pixel sequence for the *actual* seam of optimal score.
Define another table/array p of dimensions $m \times n$ to hold $-1, 0, 1$ values (indicating whether j_i was $j_{i-1} - 1, j_{i-1}, j_{i-1} + 1$ for the good seam).

Dynamic programming implementation

Algorithm Vertical-Seam(I, m, n)

1. **for** $j \leftarrow 1$ **to** n
2. **for** $i \leftarrow 1$ **to** m
3. $e[i, j] \leftarrow$ "compute $e_I(i, j)$ " $// \Theta(1)$ time
4. $opt[1, j] \leftarrow e[1, j], p[1, j] \leftarrow 0$ $//$ Base case
5. **for** $i \leftarrow 1$ **to** m
6. **for** $j \leftarrow 1$ **to** n
7. $opt[i, j] \leftarrow opt[i - 1, j], p[i, j] \leftarrow 0$ $//$ default case
8. **if** $opt[i - 1, j - 1] < opt[i, j]$ **then**
9. $opt[i, j] \leftarrow opt[i - 1, j - 1], p[i, j] \leftarrow -1$
10. **if** $opt[i - 1, j + 1] < opt[i, j]$ **then**
11. $opt[i, j] \leftarrow opt[i - 1, j + 1], p[i, j] \leftarrow +1$
12. $opt[i, j] \leftarrow opt[i, j] + e[i, j]$ $//$ Always add $e[i, j]$
13. $j^* \leftarrow 2$
14. **for** $j \leftarrow 1$ **to** n
15. **if** $opt[m, j] < opt[m, j^*]$ **then** $j^* \leftarrow j$
16. Print("Best vertical seam ends at cell (m, j^*) ").

Wrapping up

- ▶ After the algorithm has terminated, we find the optimal vertical seam by searching row m for the minimum $opt[m, j]$ value (one final loop).
- ▶ The double-loop between lines 5-12 does the main work, computing the opt values using the recurrence. There are $m \cdot n$ iterations of lines 7-12, and we can check that for a specific (i, j) , lines 7-12 take $O(1)$ time. So the algorithm has worst-case running-time $O(mn)$.
(ok, should also say lines 1-4 take $O(mn)$ and lines 13-16 take $O(n)$)
- ▶ The values in the p array make it very easy to reconstruct the actual sequence of pixels forming the seam (even easier than edit distance, etc).
- ▶ We specify how to compute $e[i, j]$ as the specific energy function can vary (see Avidan-Shamir paper). These are functions of local (to (i, j)) pixels and hence will always be computable in $O(1)$ -time (per (i, j) entry).

As discussed in the video, the seam will either be deleted (if we are aiming to **reduce the width**) or alternatively duplicated (if aiming to **increase width**).

There is a similar algorithm for Horizontal-Seam.

Edit distance

Our setting is strings over some input alphabet. We want to measure the **edit distance** between two given strings s, t over that alphabet.

We have three operations on strings - **insertion**, **deletion**, and **substitution**.

Examples:

- ▶ DNA or RNA strings over their 4-character alphabet: for example, "AATCCGCTAG" versus "AAACCCTTAG".
- ▶ Words from a natural language - for example, "kitten" versus "sitting".

k	i	t	t	e	n	-
s	i	t	t	i	n	g

(3 operations: 2 "substitutions" and 1 "insertion")

Sequence Alignment

We often talk about possible **alignments** of two (or more) sequences. For example, here are two competing alignments for a given pair of DNA sequences:

A	C	C	G	G	T	A	T	C	C	T	A	G	G	A	C
A	C	C	T	A	T	C	T	-	-	T	A	G	G	A	C

A	C	C	G	G	T	A	T	C	C	T	A	G	G	A	C
A	C	C	-	-	T	A	T	C	T	T	A	G	G	A	C

An alignment of two sequences $s \in \Sigma^m, t \in \Sigma^n$ is any padding (with some $-$ insertions) s' of s , and t' of t such that

$$|s'| = |t'|$$
$$(s'_i \neq -) \vee (t'_i \neq -) \quad \text{for all } 1 \leq i \leq |s'|$$

The score of an alignent is the total number of **insertions** ($s'_i \in \Sigma$ with $t'_i = -$), **deletions** ($s'_i = -$ with $t'_i \in \Sigma$) and **substitutions** ($s'_i \neq t'_i, s'_i \in \Sigma, t'_i \in \Sigma$).

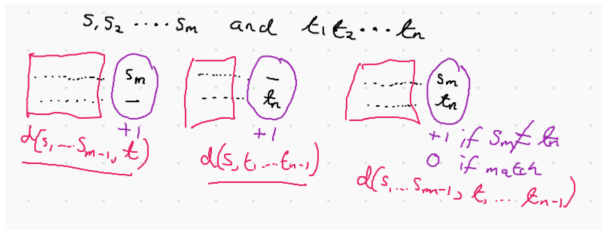
Edit distance

The **edit distance** $d(s, t)$ between two strings $s, t \in \Sigma^*$ is the minimum number of operations possible for an alignment of those strings.

We start with strings over the alphabet Σ .

How to align these? We don't know.

But we do know there are only 3 ways the final column can be arranged!



And the “best possible” for each of these 3 possibilities is another “edit distance” problem for an input that is **slightly** smaller.

Edit distance

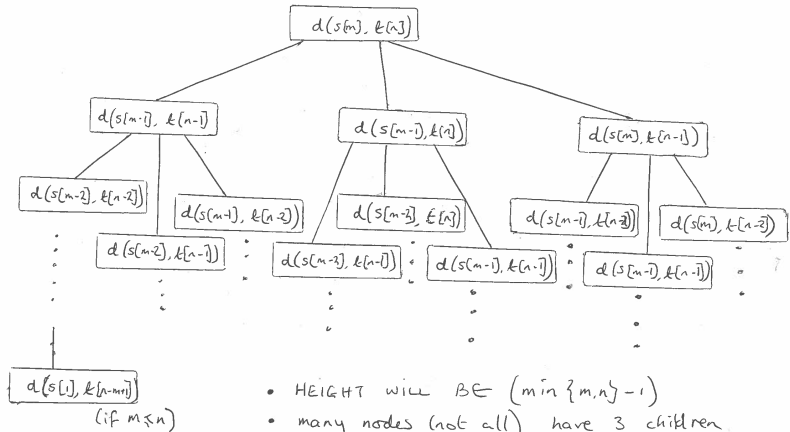
We get a natural recurrence for the edit distance for $s = s[1 \dots m], t = t[1 \dots n]$:

$$d(s[1 \dots m], t[1 \dots n]) = \begin{cases} m & \text{if } n = 0 \\ n & \text{if } m = 0 \\ d(s[1 \dots m-1], t[1 \dots n-1]) & \text{if } s_m = t_n \\ 1 + \min\{d(s[1 \dots m-1], t[1 \dots n-1]), \\ d(s[1 \dots m-1], t[1 \dots n]), \\ d(s[1 \dots m], t[1 \dots n-1])\} & \text{if } s_m \neq t_n \end{cases}$$

Justification?

Whatever the best alignment is, its right column must *either* be a substitution, or a deletion, or an insertion.

A recursive implementation?



- HEIGHT will BE $(\min\{m, n\} - 1)$
- many nodes (not all) have 3 children
- #-of-leaves grows similar to $3^{\min\{m, n\} - 1}$

Recursion tree is exponential in size ... however there are at most $m \cdot n$ sub-problems that can arise! So we are in a situation where DP can be exploited

Dynamic programming implementation

We will need a table/array of size $(m + 1) \cdot (n + 1)$, let the table be d .

- ▶ Entry $d[i, j]$ is intended to store the value of $d(s[1 \dots i], t[1 \dots j])$ (when we have computed it).
- ▶ We need to fill the table in a careful order - need to be sure that $d[i - 1, j - 1]$, $d[i - 1, j]$ and $d[i, j - 1]$ have *already* been computed before we exploit the recurrence to compute $d[i, j]$.

We will also keep a table/array called a which will store values 0, 1, 2, 3 to mark whether the optimum for $s[1 \dots i], t[1 \dots j]$ ended in a **match** (0), a **substitution** (1), an **insertion** (2) or a **deletion** (3).

The a table will help us reconstruct the actual (best) alignment that achieves the edit distance.

(the 0/1/2/3 are just quaternary “flags” and their values are not significant)

Dynamic programming implementation

Algorithm Edit-Distance($s[1 \dots m], t[1 \dots n]$)

1. **for** $i \leftarrow 0$ **to** m
2. $d[i, 0] \leftarrow i, a[i, 0] \leftarrow 3$
3. **for** $j \leftarrow 0$ **to** n
4. $d[0, j] \leftarrow j, a[0, j] \leftarrow 2$
5. **for** $i \leftarrow 1$ **to** m **do**
6. **for** $j \leftarrow 1$ **to** n **do**
7. **if** $s_i = t_j$ **then**
8. $d[i, j] \leftarrow d[i - 1, j - 1]$
9. $a[i, j] \leftarrow 0$
10. **else**
11. $d[i, j] \leftarrow 1 + \min\{d[i, j - 1], d[i - 1, j], d[i - 1, j - 1]\}$
12. **if** $d[i, j] = d[i - 1, j - 1] + 1$ **then** $a[i, j] \leftarrow 1$
13. **else if** $d[i, j] = d[i, j - 1] + 1$ **then** $a[i, j] \leftarrow 2$
14. **else** $a[i, j] \leftarrow 3$

Reconstructing the best alignment

We use the information in the a table to fill two arrays b, c .

- ▶ b will hold the padded version of s (the s'), in reverse
- ▶ c will hold the padded version of t (the t'), in reverse
- ▶ we will build b, c by “working-back” through the table a (having started at $a[m, n]$ ($i \leftarrow m, j \leftarrow n$)).
- ▶ At each step, we will check whether $a[i, j]$ is either
 - 0/1 In this case we insert character s_i into b , and character t_j into c , then decrement both i and j
 - 2 In this case we insert character ‘—’ into b , and character t_j into c , then decrement j (but not i)
 - 3 In this case we insert character s_i into b , and character ‘—’ into c , then decrement i (but not j)
- ▶ At some point either i or j will hit 0, then we need to “finish off” b and c with a “run of insertions” or a “run of deletions”.
- ▶ This results with the exact alignment stored in b and c , in reverse order. We then can print out in reverse.

Running time

It is not too hard to show that the running time for Edit-Distance is the same as the space of its primary tables, ie, $\Theta(mn)$.

Reading Materials

Seam Carving:

- ▶ “Seam-Carving for Content-Aware Image Resizing” (clickable). I have used slightly different notation.
- ▶ There are many Seam-Carving implementations in Python available on the Internet, worth Googling them and taking a look. They are usually using `numpy` for file i/o and other functionality, plus some image processing resource too.



Edit Distance:

- ▶ [CLRS] does not cover the same Dynamic Programming problems as us, but section 15.3 is related to our discussion of “Principles of Dynamic Programming” (though mentions problems we haven’t studied).
- ▶ [CLRS] presents *longest common subsequence*, which is related to edit distance, in Section 15.4. In fact, LCS can be solved as an easy application of edit distance.