Aesthetics and Ordering in Stacked Area Charts
Supplementary Material: Algorithmic Details

Steffen Strunge Mathiesen and Hans-Jörg Schulz
Aarhus University, Denmark

# 1 The iterative optimisation procedure `UpwardsOpt`

The fundamental idea of `UpwardsOpt`, as stated at the very beginning of Sec. 4.1 in the paper, is to iteratively perform the following steps for all layers of a predefined stack of layers:

1. Each layer $i$ is first removed from the stack.
2. The function $FindBestPosition$ then computes layer $i$'s optimal position in the remaining stack.
3. Layer $i$ is then reinserted at that newfound position.

This 3-step procedure is shown in lines 13-15 in Algorithm 1 below, which takes as inputs an initial stack of layers $init$ and a minimum improvement threshold $minImpr$.

The remainder of the algorithm is for bookkeeping, so that we do not visit layers more than once despite them being shuffled around. This bookkeeping builds on the observation that moving a layer $i$ to position $newIdx$ only affects the layers between (and including) positions $i$ and $newIdx$ (lines 18 and 21). If the layer $i$ was moved upwards (line 18), all layers between $i$ and $newIdx$ which are already in the list of done layers get their index decremented by 1, as $i$ has pushed them down one layer (line 19). Yet if the layer $i$ was moved downwards, all layers between $newIdx$ and $i$ which are already in the list of done layers get their index

**Algorithm 1** UpwardsOpt

```
1:  procedure UpwardsOpt(init, minImpr)
2:      order ← init
3:      repeat
4:          oldCost ← cost_chart(order)
5:          i ← 0
6:          done ← []
7:          while i < length(order) do
8:              if i ∈ done then
9:                  i ++
10:                 continue                                          ▷ goes back to line 7
11:             end if
12:
13:             f_i ← pop(order, i)
14:             newIdx ← FindBestPosition(order, f_i)
15:             insert(order, newIdx, f_i)
16:
17:             for d = 0 to length(done) − 1 do
18:                 if (newIdx > i) && (done[d] ∈ [i . . . newIdx]) then
19:                     done[d] - -                                   ▷ layer i moved up
20:                 end if
21:                 if (newIdx < i) && (done[d] ∈ [newIdx . . . i]) then
22:                     done[d] + +                                   ▷ layer i moved down
23:                 end if
24:             end for
25:             done.add(newIdx)
26:         end while
27:     until cost_chart(order) ≥ oldCost · (100% − minImpr)
28: end procedure
```

incremented by 1, as $i$ has pushed them up one layer (line 22). If the layer was not moved at all, nothing happens. After considering all layers (while loop in lines 7 through 26), we compare if the overall cost of the new ordering is better than it was before. As long as the cost improves by at least $minImpr$ percent per iteration, we repeat the above procedure (repeat loop in lines 3 through 27). This overall approach guarantees that no improvement $\geq minImpr$ can be made by moving any individual layer to another position.

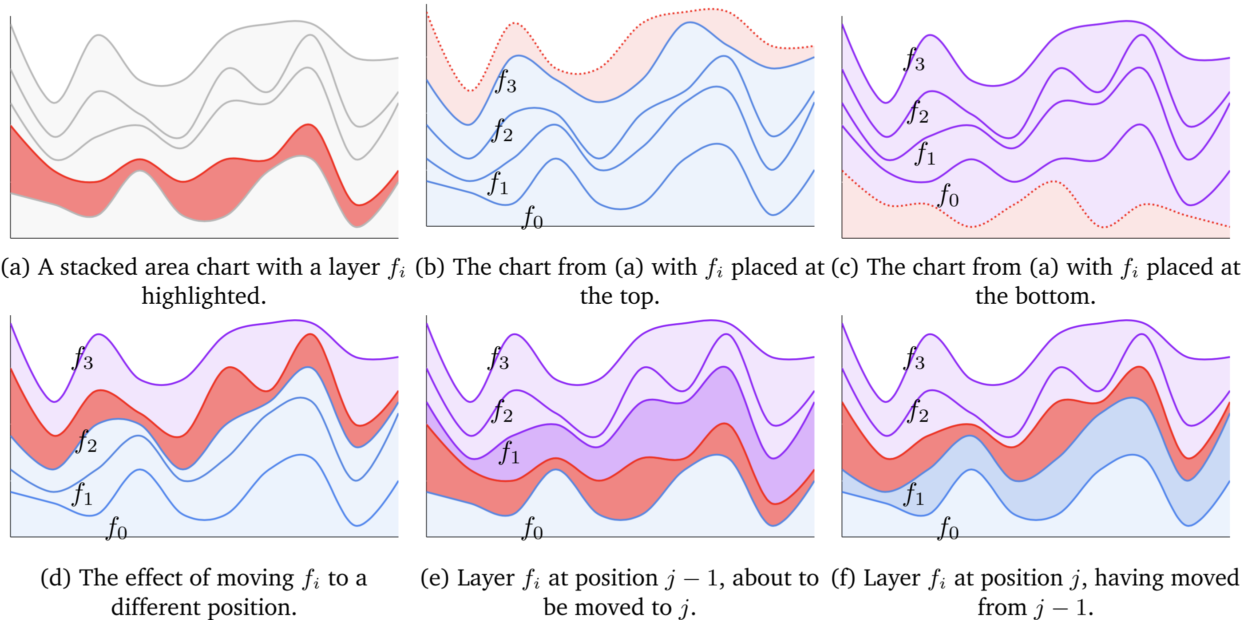
## 2 Finding optimal layer positions with `FindBestPosition`

Finding the best position to reinsert a layer into the stack is done by the `FindBestPosition` algorithm stated in Sec. 4.2 of the paper. This algorithm consists of two stages: a **preprocessing stage** of certain layer costs and the actual **testing stage** using the preprocessed costs to determine layer $i$'s best position in the given stack.

During the preprocessing stage, the following three costs are computed for our layer $i$ and the remaining stack with layer $i$ removed:

- $costBelow$ stores an array of costs for all layers, if a layer lies below layer $i$ in the stack. This means at index $pos$, $costBelow[pos]$ contains the cost of layer $pos$ sitting on the stack of layers 0 through $pos - 1$, excluding layer $i$. This is shown schematically in Fig. 1b with the red layer $i$ is removed from its original position in Fig. 1a and being placed at the very top. The array $costBelow$ then captures the cost values for all layers shown in blue.
- $costAbove$ stores an array of costs for all layers, if a layer lies above layer $i$. This means at index $pos$, $costAbove[pos]$ contains the cost of layer $pos$ sitting on the stack of layers 0 through $pos - 1$ and layer $i$. This is shown schematically in Fig. 1c with the red layer $i$ is removed from its original position in Fig. 1a and being placed at the bottom of the stack. The array $costAbove$ then captures the cost values for all layers shown in purple.
- $costLayer$ stores an array of costs for layer $i$ being positioned at position $pos$. Hence, $costLayer[pos]$ holds the cost of layer $i$ placed on top of layers $0 \dots pos - 1$.

Together, these arrays allow us to specify the cost of a stack with layer $i$ at position $pos$ by summing up $costBelow[0 \dots pos - 1]$, $costLayer[pos]$, and $costAbove[pos \dots n - 2]$ with $n$ being the numbers of layers including $i$. The details are specified in Eqn.10 in the paper.




(a) A stacked area chart with a layer $f_i$ highlighted.

(b) The chart from (a) with $f_i$ placed at the top.

(c) The chart from (a) with $f_i$ placed at the bottom.

(d) The effect of moving $f_i$ to a different position.

(e) Layer $f_i$ at position $j - 1$, about to be moved to $j$.

(f) Layer $f_i$ at position $j$, having moved from $j - 1$.

Figure 1: The principle idea behind `FindBestPosition`: The top row shows the preprocessing stage in which for a layer $f_i$ (a) stacks are precomputed under the assumption that the layers are below $f_i$ (b) or above (c). The bottom row shows the testing stage and how we can piece together any stack from the precomputed stacks (d). In case we only move up the layer $f_i$ by one position, we only have to adjust a few layers from one stack to the next (e+f).

During the testing stage, we then try new positions for layer $i$ by moving it upwards in the stack one layer at a time. By doing so, we can make use of the preprocessed costs to modify the cost value of the previously tested position, as described in Eqn.11 in the paper. This is also illustrated in the transition from Fig. 1e to Fig. 1f: When moving the red layer $i$ up one layer to position $pos$, we need to

- add $costBelow[pos-1]$ (line 19, also shown as the dark blue layer in Fig. 1f)
- subtract $costAbove[pos-1]$ (line 20, also shown as the dark purple layer in Fig. 1e)
- add $costLayer[pos]$ (line 21, also shown as the red layer in Fig. 1f)
- subtract $costLayer[pos-1]$ (line 22, also shown as the red layer in Fig. 1e)

## 3 Discussion of runtime complexities

The time complexities of `BestFirst` and `TwoOpt` are already stated in their respective source publications. However, the implementation used for benchmarking in the paper is a bit different, since it is made for regular stacked area charts instead of streamgraphs. For `BestFirst`, the time complexity is $\mathcal{O}(n^2m)$, where $n$ is the number of layers and $m$ is the number of time points per layer. The algorithm iteratively adds one layer at a time to the stack by checking all layers each time for the best one to add. Testing a layer requires a call to the cost function, which contains a sum over all time points and thus runs in $\mathcal{O}(m)$. Adding the $n$ layers in the correct order by checking $\mathcal{O}(n)$ layers each time adds the $n^2$ part. `TwoOpt` then runs in $\mathcal{O}(rnm)$ time, where $r$ is the number of repeats. As the algorithm runs until it stops improving, it can be difficult to say exactly how many repeats it needs to complete. The number depends on a number of things, including the number of layers, as well as the shape and span of the layers. The $nm$ part originates from calculating the cost for each pair of layers like in `BestFirst` and doing this for every pair of adjacent layers in the stack.

For `UpwardsOpt`, all of the layers are considered one by one for the best position. Running through each layer requires the loop to run $\mathcal{O}(n)$ times. For each layer, all positions in the stack are considered, which is done by a single call to `FindBestPosition` (see line 14 in Algorithm 1. In `FindBestPosition`, the three arrays of length $n$ are initialised, each being filled with results from computing the cost function. The cost function sums over all time points, resulting in a time complexity of $\mathcal{O}(m)$. Doing this $n$ times means that the preprocessing stage of `FindBestPosition` runs in $\mathcal{O}(nm)$. The testing stage contains a loop running from $0$ to $n$, containing merely additions and subtractions. Hence the time complexity of the testing stage is $\mathcal{O}(n)$, and $\mathcal{O}(nm)$ becomes the dominant part for `FindBestPosition`. Combining this with the $\mathcal{O}(n)$ calls to `FindBestPosition` gives us a time complexity of $\mathcal{O}(n^2m)$ for the inner loop of `UpwardsOpt` (i.e., the while loop in Algorithm 1). But since this loop is repeated until it stops improving over $minImpr$, the time complexity gets an added $r$ number of repeats, resulting in $\mathcal{O}(rn^2m)$. Like for `TwoOpt`, it can be difficult to say exactly how large $r$ is. It is of course dependent on $minImpr$, which – as stated in the paper – results in $2 \leq r \leq 4$ for $minImpr = 1\%$.