Speeding up exact MSA using forward dynamic programming
Sum-of-pairs score

Idea (from a computational viewpoint): define cost and optimality such that ideas from pairwise alignment can be reused ...

Cost of alignment = “sum of the cost of each column”

Problem: Given a set of sequences $S_1, S_2, S_3, \ldots$, a score matrix and a gap cost, find a multiple alignment of optimal sum-of-pairs cost
Sum-of-pairs score

Idea (from a computational viewpoint): define cost and optimality such that ideas from pairwise alignment can be reused ...

Cost of alignment = “sum of the cost of each column”

Possible solution: Extension of dynamic programming method, idea is to fill out table of size proportional to product of sequence lengths, i.e. $|S_1| \cdot |S_2| \cdot |S_3| \cdots$, or $n^k$ for $k$ sequences of length $n$ ...
Computing an exact MSA using SP-score

* Let \( D(i_1, \ldots, i_K) \) be the cost of an optimal mult. align of \( S_1[i_1 \ldots i_2], \ldots, S_K[i_1 \ldots i_K] \)

* Compute \( D(i_1, \ldots, i_K) \) by optimizing over all possible last columns of such an alignment.

\[
\begin{bmatrix}
S_2[i_1 i_2] \\
S_2[i_1 i_2] \\
\vdots \\
S_K[i_1 i_K]
\end{bmatrix}, \begin{bmatrix}
S_2[i_1 i_2] \\
\vdots \\
S_K[i_1 i_K]
\end{bmatrix}, \ldots, \begin{bmatrix}
S_2[i_1 i_2] \\
\vdots \\
S_K[i_1 i_K]
\end{bmatrix}, \ldots
\]

* possible last columns \( = 2^K - 1 \)

\[
D(i_1, \ldots, i_K) = \min_{\text{last.col}} \left[ D(i_1, \ldots, i_K) + \text{cost( last.col) } \right]
\]
Computing an exact MSA using SP-score

* Let $D(i_1, \ldots, i_k)$ be the cost of an optimal mult-align of $S_1[1\ldots i_2], \ldots, S_k[1\ldots i_k]$

* Compute $D(i_1, \ldots, i_k)$ by optimizing over all
  
  **Time:** $O(n^k \cdot 2^k \cdot \text{“time to compute cost of last column”})$

  **Space:** $O(n^k)$

\[
\begin{bmatrix}
S_2[i_2] \\
\vdots \\
S_k[i_k], \\
\end{bmatrix}, \quad \begin{bmatrix}
S_2[i_3] \\
\vdots \\
S_k[i_k], \\
\end{bmatrix}, \ldots, \begin{bmatrix}
S_2[i_n] \\
\vdots \\
S_k[i_n], \\
\end{bmatrix}, \ldots
\]

* possible last columns = $2^k - 1$

\[
D(i_1, \ldots, i_k) = \min_{\text{last.col}} \left[ D(i_1, \ldots, i_k) + \text{cost(last.col)} \right]
\]
Speeding up the computation

**Idea:** Avoid filling the entire dynamic programming table

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>c</th>
<th>t</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Speeding up the computation

Idea: Avoid filling the entire dynamic programming table

We fill out only parts of the table close to the diagonal, i.e. we do not search the entire space of possible alignments. The time and space consumption is proportional to the part of table we fill, but we are not guaranteed to find an optimal alignment.
Speeding up the computation

Idea: Avoid filling the entire dynamic programming table

Can we do better, i.e. fill only a small fraction of the table in practice and be guaranteed to find an optimal alignment?

We fill out only parts of the table close to the diagonal, i.e. we do not search the entire space of possible alignments. The time and space consumption is proportional to the part of table we fill, but we are not guaranteed to find an optimal alignment.
**Forward dynamic programming**

**Backward dynamic programming:** We compute the value of a cell \((i_1, ..., i_k)\) by looking at all its predecessors in the alignment graph.

\[
D(i_1, ..., i_k) = \min_{\text{last.col}} \left[ D(i_1', ..., i_k') + \text{cost(last.col)} \right]
\]

**Forward dynamic programming:** A cell \((i_1, ..., i_k)\) sends forward its value to all its successors (in the alignment graph) when its value is known. A cell knows its value when it has received values from all its predecessors.

“Ready Cells” = \[
\{(0, ..., 0), 0\}
\]

for each cell \(c\) in “Ready Cells”:

for each cell \(s\) in Succ\((c)\):

\[s\.val = \min(s\.val, c\.val + \text{cost of edge \(c\) from \(s\)})\]

if “\(s\) has received values from all its predecessors”:

“Ready Cells” = “Ready Cells” + \{\(s\), s\.val\}

return \((n_1, ..., n_k)\.val\)
Can “Forward dynamic programming” be implemented efficiently?

**From graph theory:** a topological ordering of a directed acyclic graph is a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges.

What is a topological ordering of this graph?
**Forward dynamic programming**

Can “Forward dynamic programming” be implemented efficiently?

From graph theory: a **topological ordering** of a directed acyclic graph is a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges.

What is a topological ordering of this graph? E.g. a row-, column-, or diagonal-wise numbering of the cells/nodes.
Forward dynamic programming

Can “Forward dynamic programming” be implemented efficiently?

From graph theory: a topological ordering of a directed acyclic graph is a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges.

In general: The ordering of the nodes in an alignment graph according to the lexicographical ordering of their coordinates \((i_1, \ldots, i_k)\) is a topological ordering.
Forward dynamic programming

“Ready Cells” = [ {(0,...0),0} ]
for each cell c in “Ready Cells”:
    for each cell s in Succ(c):
        s.val = min(s.val, c.val + “cost of edge c from s”)
        if “s has received values from all its predecessors”:
            “Ready Cells” = “Ready Cells” + {s, s.val}
return (n₁,...nₖ).val

Can be simplified to:

for each cell c in “cells in topological order”:
    for each cell s in Succ(c):
        s.val = min(s.val, c.val + “cost of edge c from s”)
return (n₁,...nₖ).val

Can be implemented with same time/space consumption as “normal” backward dynamic programming.
Forward dynamic programming

“Ready Cells” = [ {(0,...0),0} ]
for each cell c in “Ready Cells”:
    for each cell s in Succ(c):
        s.val = min(s.val, c.val + “cost of edge c from s”)
        if “s has received values from all its predecessors”:
            “Ready Cells” = “Ready Cells” + {s, s.val}
return (n_1,...n_k).val

Can be simplified to:

for each cell c in “cells in topological order”:
    for each cell s in Succ(c):
        s.val = min(s.val, c.val + “cost of edge c from s”)
return (n_1,...n_k).val

Can be implemented with same time/space consumption as “normal” backward dynamic programming. So what is the point?
Forward dynamic programming

Idea: Only send forward values if it makes sense, i.e. avoid to send forward a value to a cell/node which cannot be on an optimal path.

```
  for each cell s in Succ(c):
    s.val = min(s.val, c.val + "cost of edge c from s")
    if "s has received values from all its predecessors":
      "Ready Cells" = "Ready Cells" + {s, s.val}
  return (n_1,...n_k).val
```

Can be simplified to:

```
  for each cell c in "cells in topological order":
    for each cell s in Succ(c):
      s.val = min(s.val, c.val + "cost of edge c from s")
  return (n_1,...n_k).val
```

Can be implemented with same time/space consumption as "normal" backward dynamic programming. So what is the point?
Forward dynamic programming

**Idea:** Only send forward values if it makes sense, i.e. avoid to send forward a value to a cell/node which cannot be on an optimal path.

For each cell $s$ in $\text{Succ}(c)$:
- Update $s$'s value as $\min(s\text{.val}, c\text{.val} + \text{"cost of edge c from s"})$
- Update $s$'s value after receiving values from all its predecessors: $s\text{.val} = \text{\"Ready Cells\" + \{s, s\text{.val}\}}$

Can be simplified to:

- In topological order: $\text{cc}(c)$:
  - $s\text{.val} = \min(s\text{.val}, c\text{.val} + \text{\"cost of edge c from s\"})$

Can be implemented with the same time/space consumption as "normal" backward dynamic programming. So what is the point?
Forward dynamic programming

Idea: Only send forward values if it makes sense, i.e. avoid to send forward a value to a cell/node which cannot be on an optimal path.

for each cell s in Succ(c):
  If the path "start → c → s → end" cannot be optimal, then do not send value to s, i.e do not send value to s if
  \[ c.\text{val} + \text{cost}(c \to s) + \text{cost}(s \to \text{end}) > \text{OPT} \]
  Can be simplified to:
  \[ \text{cc}(c) = \min \{ c.\text{val} + \text{"cost of edge c from s"}, \ldots \} \]

Can be implemented with same time/space consumption as "normal" backward dynamic programming. So what is the point?
Forward dynamic programming

**Idea:** Only send forward values if it makes sense, i.e. avoid to send forward a value to a cell/node which cannot be on an optimal path.

```
for each cell s in Succ(c):
    if the path “start → c → s → end” cannot be optimal, then do not send value to s, i.e do not send value to s if
    c.val + cost(c → s) + cost(s → end) > OPT
```

**Problem:** cost(s → end) and OPT are unknown. Relax the restriction such that we do not send value to s if

```
c.val + cost(c → s) + LowerBound(cost(s → end)) > UpperBound(OPT)
```

Can be implemented with same time/space consumption as “normal” backward dynamic programming. So what is the point?
An upper bound of OPT

How do we get an upper bound on the optimal cost of an MSA using column based SP score?

Sample a number of possible alignments and keep the best score

Use the 2-approximation algorithm

...
An upper bound of OPT

How do we get an upper bound on the optimal cost of an MSA using column based SP score?

Sample a number of possible alignments and keep the best score

Use the 2-approximation algorithm

How do we get a lower bound on cost(s → end) for cell $s = (i_1, \ldots, i_k)$?

i.e. a lower bound the cost of an optimal SP MSA of

$s_1[i_1 \ldots n_1], \ldots, s_k[i_k \ldots n_k]$
A lower bound of cost(s → end)

How do we get a lower bound on cost(s → end) for cell s = (i_1,...i_k)?

i.e. a lower bound the cost of an optimal SP MSA of

\[ s_1[i_1...n_1],...,s_k[i_k...n_k] \]

Recall the definition of SP-score and the obvious fact that the cost of an induced pairwise alignment of row u and v is at least the cost of an optimal pairwise alignment of string u and v.

\[ \text{OPT}(s_1[i_1...n_1],...,s_k[i_k...n_k]) > \sum_{1 \leq u < v \leq k} D(s_u[i_u...n_u], s_v[i_v...n_v]) \]

How do we compute this efficiently?
A lower bound of cost(s → end)

Assume that we have table $T^u,v$ such that $T^u,v_{[i,j]} = D(s^u[i..n^u], s^v[j..n^v])$, then

$$\text{OPT}(s^u[i_1..n^u], \ldots, s^v[k_i..n^v]) > \sum_{1 \leq u < v < k} D(s^u[i_u..n_u], s^v[i_v..n_v])$$

We can compute and store all tables $T^u,v$ in time $O(k^2 n^2)$ by aligning the reversed $S^u$ against the reversed $S^v$ (figure out the details), i.e. we can compute and store $T^u,v$ in time and space $O(n)$. We have

$$\text{OPT}(s^u[i_1..n^u], \ldots, s^v[k_i..n^v])$$

Assume that we have table $T^u,v$ such that $T^u,v_{[i,j]} = D(s^u[i..n^u], s^v[j..n^v])$, then

$$\text{OPT}(s^u[i_1..n^u], \ldots, s^v[k_i..n^v]) > \sum_{1 \leq u < v < k} D(s^u[i_u..n_u], s^v[i_v..n_v])$$
A fast exact MSA algorithm

for each cell c in “cells in topological order”:
    for each cell s in Succ(c):
        if c.val + cost(c → s) + LB(cost(s → end)) ≤ UP(OPT):
            s.val = min(s.val, c.val + cost(c → s))

return (n₁,...nₖ).val

Any problems?
A fast exact MSA algorithm

for each cell c in “cells in topological order”:
  for each cell s in Succ(c):
    if c.val + cost(c → s) + LB(cost(s → end)) ≤ UP(OPT):
      s.val = min(s.val, c.val + cost(c → s))

return (n₁,...,nₖ).val

Any problems?

We still iterate over all cells c, so there is no speed-up. We should only consider cells c which have received at least one value from a predecessor.
A fast exact MSA algorithm

Q = “empty priority queue”
Q.insert ( {cell=(0,..,0), val=0} )
while true:
    c = Q.deleteMin()
    if c == (n₁,...nₖ):
        return c.val
    else:
        for each cell s in Succ(c):
            if c.val + cost(c → s) + LB(cost(s → end)) ≤ UP(OPT):
                if s in Q:
                    Q.updateElm( {s, min(s.val, c.val + cost(c → s))} )
                else:
                    Q.insert( {s, c.val + cost(c → s)} )

Q is a priority queue where the priority of an element (i.e. cell) is the coordinates of the cell in the alignment graph.
A fast exact MSA algorithm

Q is a priority queue where the priority of an element (i.e. cell) is the coordinates of the cell in the alignment graph.
Anything else?

Is it a fast algorithm?
Anything else?

Is it a fast algorithm?

Time and space consumption depends on the quality of the upper bound on OPT and the lower bounds on alignments of suffixes!

How do we get an optimal alignment?
Anything else?

Is it a fast algorithm?

Time an space consumption depends on the quality of the upper bound on OPT and the lower bounds on alignments of suffixes!

How do we get an optimal alignment?

Backtrack through the part of the alignment graph visited. Keep all visited cells in memory, i.e. when the algorithm ends you have the visited part of the alignment graph in memory.
Anything else?

Is it a fast algorithm?

Time an space consumption depends on the quality of the upper bound on OPT and the lower bounds on alignments of suffixes!

How do we get an optimal alignment?

Backtrack through the part of the alignment graph visited. Keep all visited cells in memory, i.e. when the algorithm ends you have the visited part of the alignment graph in memory.

In order to reduce space consumption, you might want to first run the algorithm where you do not store the graph. This give you the optimal score OPT. Now rerun the algorithm with the optimal score OPT as upper bound, this should minimize the part of the alignment graph you visit, i.e. reduce space consumption.