Algorithms and Data Structures in Biology

Dynamic Programming Algorithms

Ugo Dal Lago

University of Bologna, Academic Year 2018/2019

Dynamic Programming

- Up to now, we have analysed two different design strategies for algorithms:
  - **Exhaustive Search**
    - *Correctness* holds in a perfect sense, i.e., without any possibility of errors.
    - *Complexity*, at least in the worst case, can be very high, although branch-and-bound can be of help.
  - **Greedy**
    - *Correctness* only holds in an approximate sense, while bounds on the approximate ratio can sometime be given.
    - *Complexity* is lower than in exhaustive search, although in general polynomial in the size of the input.
- **A natural question**: Is there a way to tame certain problems so as to remain low in complexity without losing correctness?

Dynamic Programming

- Up to now, we have analysed two different design strategies for algorithms:
  - **Exhaustive Search**
    - *Correctness* holds in a perfect sense, i.e., without any possibility of errors.
    - *Complexity*, at least in the worst case, can be very high, although branch-and-bound can be of help.
  - **Greedy**
    - *Correctness* only holds in an approximate sense, while bounds on the approximate ratio can sometime be given.
    - *Complexity* is lower than in exhaustive search, although in general polynomial in the size of the input.
- **A natural question**: Is there a way to tame certain problems so as to remain low in complexity without losing correctness?

Dynamic Programming

- Up to now, we have analysed two different design strategies for algorithms:
  - **Exhaustive Search**
    - *Correctness* holds in a perfect sense, i.e., without any possibility of errors.
    - *Complexity*, at least in the worst case, can be very high, although branch-and-bound can be of help.
  - **Greedy**
    - *Correctness* only holds in an approximate sense, while bounds on the approximate ratio can sometime be given.
    - *Complexity* is lower than in exhaustive search, although in general polynomial in the size of the input.
- **A natural question**: Is there a way to tame certain problems so as to remain low in complexity without losing correctness?

The Change Problem, Again!

Change Problem:
Convert some amount of money M into given denominations, using the smallest possible number of coins.

Input: An amount of money M, and an array of d denominations $\mathbf{c} = (c_1, c_2, \ldots, c_d)$, in decreasing order of value ($c_1 > c_2 > \cdots > c_d$).

Output: A list of d integers $i_1, i_2, \ldots, i_d$ such that $c_1 i_1 + c_2 i_2 + \cdots + c_d i_d = M$, and $i_1 + i_2 + \cdots + i_d$ is as small as possible.

- We introduced an exhaustive search algorithm for this problem which, however, was very inefficient.
- The greedy algorithm we also introduced at the beginning of the course was however imprecise, at least in some cases.

The Change Problem, Again!

Change Problem:
Convert some amount of money M into given denominations, using the smallest possible number of coins.

Input: An amount of money M, and an array of d denominations $\mathbf{c} = (c_1, c_2, \ldots, c_d)$, in decreasing order of value ($c_1 > c_2 > \cdots > c_d$).

Output: A list of d integers $i_1, i_2, \ldots, i_d$ such that $c_1 i_1 + c_2 i_2 + \cdots + c_d i_d = M$, and $i_1 + i_2 + \cdots + i_d$ is as small as possible.

- We introduced an exhaustive search algorithm for this problem which, however, was very inefficient.
- The greedy algorithm we also introduced at the beginning of the course was however imprecise, at least in some cases.

The Structure of Optimal Solutions

- The key observation for understanding dynamic programming is the following: in a given problem, **optimal solutions are recursively optimal**.
- As an example, consider the Change Problem, with $\mathbf{c} = (1, 3, 7)$ and an optimal solution $\mathbf{i} = (i_1, i_2, i_3)$ for $M = 77$.
  - If $i_1 > 1$, then $(i_1 - 1, i_2, i_3)$ will be optimal for $M - 1 = 76$. Otherwise, we could find a triple (j_1, j_2, j_3) for 76 such that $j_1 + j_2 + j_3 < i_1 - 1 + i_2 + i_3$ and $(j_1 + 1, j_2, j_3)$ would sum to something less than $i_1 + i_2 + i_3$, contradicting the optimality of $\mathbf{i}$.
  - Similarly if $i_2 > 1$ or $i_3 > 1$.
- In all these cases, the search for the optimal solution can be performed by **looking at all possible sub-problems**, then choosing the best solution.
  - In the example above, when asked to look for an optimal solution for $M = 77$, we could look for optimal solutions for 76, 74 or 70, and take "the best one".

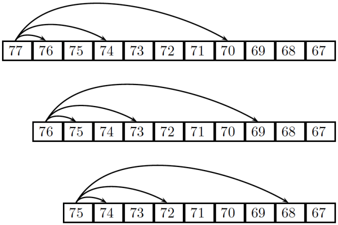
The Structure of Optimal Solutions

- The key observation for understanding dynamic programming is the following: in a given problem, **optimal solutions are recursively optimal**.
- As an example, consider the Change Problem, with $\mathbf{c} = (1, 3, 7)$ and an optimal solution $\mathbf{i} = (i_1, i_2, i_3)$ for $M = 77$.
  - If $i_1 > 1$, then $(i_1 - 1, i_2, i_3)$ will be optimal for $M - 1 = 76$. Otherwise, we could find a triple (j_1, j_2, j_3) for 76 such that $j_1 + j_2 + j_3 < i_1 - 1 + i_2 + i_3$ and $(j_1 + 1, j_2, j_3)$ would sum to something less than $i_1 + i_2 + i_3$, contradicting the optimality of $\mathbf{i}$.
  - Similarly if $i_2 > 1$ or $i_3 > 1$.
- In all these cases, the search for the optimal solution can be performed by **looking at all possible sub-problems**, then choosing the best solution.
  - In the example above, when asked to look for an optimal solution for $M = 77$, we could look for optimal solutions for 76, 74 or 70, and take "the best one".

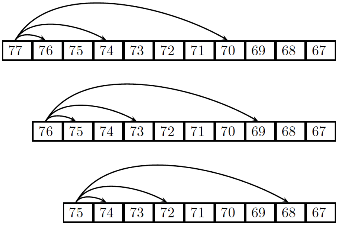
The Structure of Optimal Solutions

- The key observation for understanding dynamic programming is the following: in a given problem, **optimal solutions are recursively optimal**.
- As an example, consider the Change Problem, with $\mathbf{c} = (1, 3, 7)$ and an optimal solution $\mathbf{i} = (i_1, i_2, i_3)$ for $M = 77$.
  - If $i_1 > 1$, then $(i_1 - 1, i_2, i_3)$ will be optimal for $M - 1 = 76$. Otherwise, we could find a triple (j_1, j_2, j_3) for 76 such that $j_1 + j_2 + j_3 < i_1 - 1 + i_2 + i_3$ and $(j_1 + 1, j_2, j_3)$ would sum to something less than $i_1 + i_2 + i_3$, contradicting the optimality of $\mathbf{i}$.
  - Similarly if $i_2 > 1$ or $i_3 > 1$.
- In all these cases, the search for the optimal solution can be performed by **looking at all possible sub-problems**, then choosing the best solution.
  - In the example above, when asked to look for an optimal solution for $M = 77$, we could look for optimal solutions for 76, 74 or 70, and take "the best one".

A Recursive Algorithm for the Change Problem



A Recursive Algorithm for the Change Problem



RecursiveChange(M, $\mathbf{c}$, d)

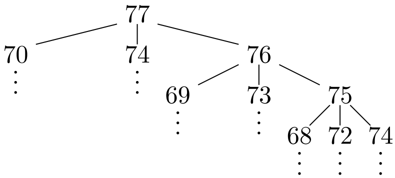
```
if M = 0
    return 0
bestNumCoins ← ∞
for i ← 1 to d
    if M ≥ c_i
        numCoins ← RecursiveChange(M − c_i, c, d)
        if numCoins + 1 < bestNumCoins
            bestNumCoins ← numCoins + 1
return bestNumCoins
```

Strange *Dejà Vu*?

- The complexity of RECURSIVECHANGE can be easily seen to be exponential.
- Indeed, a call to the algorithm with first parameter equal to M would produce a pattern similar to the following one:
- This is not too different than the situation we got when we analysed the first variation on FIBONACCI. Is it possible to do apply the same trick?

Strange *Dejà Vu*?

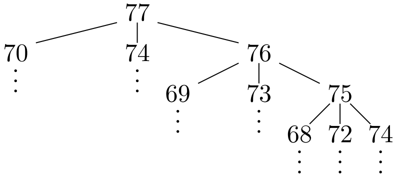
- The complexity of RecursiveChange can be easily seen to be exponential.
- Indeed, a call to the algorithm with first parameter equal to M would produce a pattern similar to the following one:



- This is not too different than the situation we got when we analysed the first variation on Fibonacci. Is it possible to do apply the same trick?

Strange *Dejà Vu*?

- The complexity of RecursiveChange can be easily seen to be exponential.
- Indeed, a call to the algorithm with first parameter equal to M would produce a pattern similar to the following one:



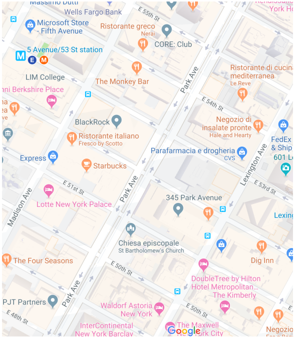
- This is not too different than the situation we got when we analysed the first variation on Fibonacci. Is it possible to do apply the same trick?

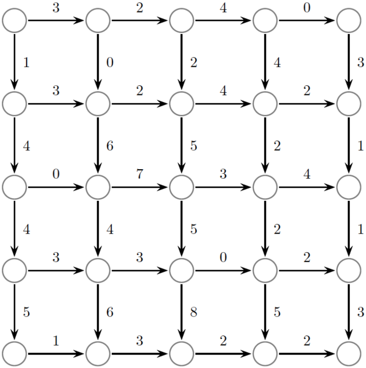
A Dynamic Programming Algorithm for the Change Problem

DPCHANGE(M, **c**, d)

$bestNumCoins_0 \leftarrow 0$
for $m \leftarrow 1$ **to** M
    $bestNumCoins_m \leftarrow \infty$
    for $i \leftarrow 1$ **to** d
        if $m \geq c_i$
            if $bestNumCoins_{m-c_i} + 1 < bestNumCoins_m$
                $bestNumCoins_m \leftarrow bestNumCoins_{m-c_i} + 1$
return $bestNumCoins_M$

The Manhattan Tourist Problem





Representing Grids

- A rectangle-shaped portion of Manhattan's map, together with the number of attractions on each boulevard's segment, can be seen as a *graph*.
  - For the moment, we do not know what a graph is, formally.
- Concretely, such a graph can be seen as a pair of $n \times m$ matrices:
  - A matrix $\overrightarrow{w}$ which gives the number of attractions to the east-bound street from each coordinate.
  - A matrix $\overset{\downarrow}{w}$ which gives the number of attractions to the south-bound street from each coordinate.
- The **source** is the coordinate $(0, 0)$, while the **target** is the coordinate (n, m).
- A **path** is a sequence of moves in $\{S, E\}$ of length $n + m$, which encodes the route taken by the tourist.

The Manhattan Tourist Problem

Manhattan Tourist Problem:
Find a longest path in a weighted grid.

> **Input:** A weighted grid G with two distinguished vertices: a *source* and a *sink*.
>
> **Output:** A longest path in G from *source* to *sink*.

- **Exhaustive Search**
  - Enumerate *all possible* paths from the source to the sink
  - The number of such paths become too large, even for moderately large graphs.
- **Greedy**
  - Instead, we could build paths by reasoning locally, based on the weight of the outgoing edges.
  - The approximation ratio of the obtained algorithm is very bad.

The Manhattan Tourist Problem

Manhattan Tourist Problem:
Find a longest path in a weighted grid.

Input: A weighted grid G with two distinguished vertices: a *source* and a *sink*.

Output: A longest path in G from *source* to *sink*.

- **Exhaustive Search**
  - Enumerate *all possible* paths from the source to the sink
  - The number of such paths become too large, even for moderately large graphs.
- **Greedy**
  - Instead, we could build paths by reasoning locally, based on the weight of the outgoing edges.
  - The approximation ratio of the obtained algorithm is very bad.

The Manhattan Tourist Problem

Manhattan Tourist Problem:
Find a longest path in a weighted grid.

Input: A weighted grid G with two distinguished vertices: a *source* and a *sink*.

Output: A longest path in G from *source* to *sink*.

- **Exhaustive Search**
  - Enumerate *all possible* paths from the source to the sink
  - The number of such paths become too large, even for moderately large graphs.
- **Greedy**
  - Instead, we could build paths by reasoning locally, based on the weight of the outgoing edges.
  - The approximation ratio of the obtained algorithm is very bad.

Dynamic Programming to the Rescue

$\textsc{ManhattanTourist}(\overset{\downarrow}{\mathbf{w}}, \overrightarrow{\mathbf{w}}, n, m)$

$s_{0,0} \leftarrow 0$
for $i \leftarrow 1$ **to** n
    $s_{i,0} \leftarrow s_{i-1,0} + \overset{\downarrow}{w}_{i,0}$
for $j \leftarrow 1$ **to** m
    $s_{0,j} \leftarrow s_{0,j-1} + \underline{\overrightarrow{w}}_{0,j}$
for $i \leftarrow 1$ **to** n
    for $j \leftarrow 1$ **to** m
        $s_{i,j} \leftarrow \max \begin{cases} s_{i-1,j} + \overset{\downarrow}{w}_{i,j} \\ s_{i,j-1} + \overrightarrow{w}_{i,j} \end{cases}$
return $s_{n,m}$

Dynamic Programming to the Rescue

- The complexity of ManhattanTourist is polynomial in n and m.
  - More specifically, it is $O(nm)$: for every pair of coordinates, we do a constant amount of work to find the optimal value of it.
- The correctness of the algorithm can be proved by giving an appropriate invariant:

$$\left(\forall i'.Longest(s_{i',0})\right) \wedge \left(\forall j'.Longest(s_{0,j'})\right)$$
$$\left(\forall 1 < i' < i.\forall j.Longest(s_{i',j})\right) \wedge \left(\forall 1 < j' \leq j.Longest(s_{i,j'})\right)$$

where $Longest(s_{k,h})$ means that $s_{k,h}$ contains the length of the longest path from $s_{k,h}$ to 0.

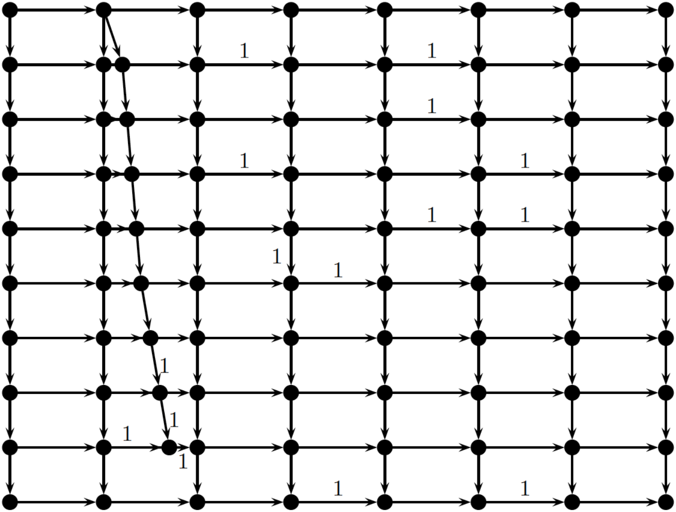
Dynamic Programming to the Rescue

- The complexity of MANHATTANTOURIST is polynomial in n and m.
  - More specifically, it is $O(nm)$: for every pair of coordinates, we do a constant amount of work to find the optimal value of it.
- The correctness of the algorithm can be proved by giving an appropriate invariant:

$$\big(\forall i'.Longest(s_{i',0})\big) \wedge \big(\forall j'.Longest(s_{0,j'})\big)$$
$$\big(\forall 1 < i' < i.\forall j.Longest(s_{i',j})\big) \wedge \big(\forall 1 < j' \leq j.Longest(s_{i,j'})\big)$$

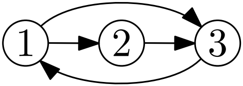
where $Longest(s_{k,h})$ means that $s_{k,h}$ contains the length of the longest path from $s_{k,h}$ to 0.

But. . .



Directed Acyclic Graphs

- A **directed graph** is a pair $G = (V, E)$ such that V is a finite set and $E \subseteq G \times G$ is the set of edges.
- **Example**: the pair $(\{1, 2, 3\}, \{(1, 2), (2, 3), (1, 3), (3, 1)\})$ is a graph, which represented graphically as follows:

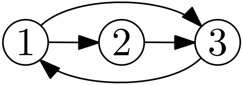


  When (and if) the node identity is not important, we just omit the numbers:

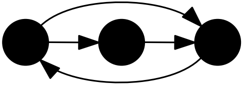
- A **path** in a graph $G = (V, E)$ is a sequence of *consecutive* edges (namely edges such that the target of the first is the source of the second).
- A directed graph is **acyclic** (or a DAG) when none of its paths is *cyclic*, namely none of its path starts and ends at the same node.

Directed Acyclic Graphs

- A **directed graph** is a pair $G = (V, E)$ such that V is a finite set and $E \subseteq G \times G$ is the set of edges.
- **Example**: the pair $(\{1, 2, 3\}, \{(1, 2), (2, 3), (1, 3), (3, 1)\})$ is a graph, which represented graphically as follows:

  When (and if) the node identity is not important, we just omit the numbers:

- A **path** in a graph $G = (V, E)$ is a sequence of *consecutive* edges (namely edges such that the target of the first is the source of the second).
- A directed graph is **acyclic** (or a DAG) when none of its paths is *cyclic*, namely none of its path starts and ends at the same node.

Weighted DAGs

- A DAG $G = (V, E)$ is said to be *weighted* if every edge in $e \in E$ comes equipped with a nonnegative number w_e.
- To any path in a weighted DAG one can naturally associate its weight, namely the sum of the weights of all its edges.

- How could we solve this problem?
- Can we adapt the dynamic programming algorithm for the Manhattan Tourist problem to this new problem?

Weighted DAGs

- A DAG $G = (V, E)$ is said to be *weighted* if every edge in $e \in E$ comes equipped with a nonnegative number w_e.
- To any path in a weighted DAG one can naturally associate its weight, namely the sum of the weights of all its edges.

Longest Path in a DAG Problem:
Find a longest path between two vertices in a weighted DAG.

Input: A weighted DAG G with *source* and *sink* vertices.

Output: A longest path in G from *source* to *sink*.

- How could we solve this problem?
- Can we adapt the dynamic programming algorithm for the Manhattan Tourist problem to this new problem?

Weighted DAGs

- A DAG $G = (V, E)$ is said to be *weighted* if every edge in $e \in E$ comes equipped with a nonnegative number w_e.
- To any path in a weighted DAG one can naturally associate its weight, namely the sum of the weights of all its edges.

Longest Path in a DAG Problem:
Find a longest path between two vertices in a weighted DAG.

Input: A weighted DAG G with *source* and *sink* vertices.

Output: A longest path in G from *source* to *sink*.

- How could we solve this problem?
- Can we adapt the dynamic programming algorithm for the Manhattan Tourist problem to this new problem?

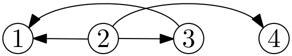
Dynamic Programming on DAGs

- Given a DAG $G = (V, E)$ and a vertex $v \in V$, the *predecessors* of v are those vertexes w for which $(w, v) \in E$. The set of all predecessors of v is indicated as $Predecessors(v)$.
- We could then solve the Longest Path Problem by computing the length of the path from the source to **any** vertex v using this equation:

  $$s_v = \max_{w \in Predecessors(v)} (s_w + w_{w,v})$$

  where $s_v = \infty$ if the $Predecessors(v) = \emptyset$.
- But then the question is: **in which order** should we compute the s_v?
  - Any *topological sort* of the graph would be fine, where a topological sort of a graph $G = (V, E)$ is any linear ordering of V which is compatible with E: if $(v, w) \in E$, then $v < w$.

Topological Sort on an Example Graph



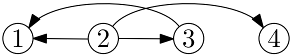
- One possible topological sort of the graph above is

$$2 > 4 > 3 > 1$$

- But are two more:

$$2 > 3 > 1 > 4 \qquad 2 > 3 > 4 > 1$$

Topological Sort on an Example Graph



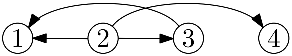
- One possible topological sort of the graph above is

$$2 > 4 > 3 > 1$$

- But are two more:

$$2 > 3 > 1 > 4 \qquad 2 > 3 > 4 > 1$$

Topological Sort on an Example Graph



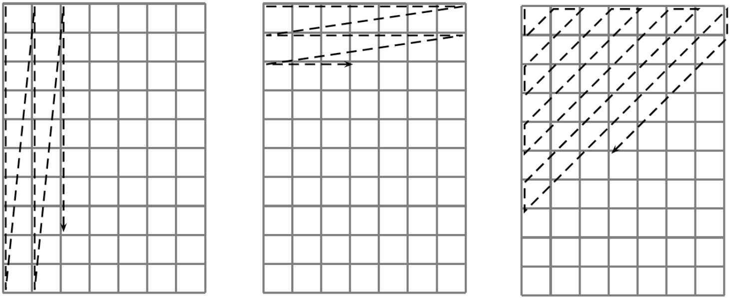
- One possible topological sort of the graph above is

$$2 > 4 > 3 > 1$$

- But are two more:

$$2 > 3 > 1 > 4 \qquad\qquad 2 > 3 > 4 > 1$$

The Topological Sort on the Manhattan Tourist Problem



The Edit Distance

TGCATAT

↓ delete last T

TGCATA

↓ delete last A

TGCAT

↓ insert A at the front

ATGCAT

↓ substitute C for G in the third position

ATCCAT

↓ insert a G before the last A

ATCCGAT

The Edit Distance

TGCATAT

↓ insert A at the front

ATGCATAT

↓ delete T in the sixth position

ATGCAAT

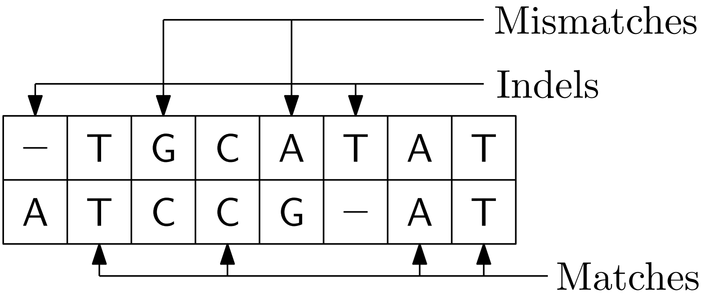
↓ substitute G for A in the fifth position

ATGCGAT

↓ substitute C for G in the third position

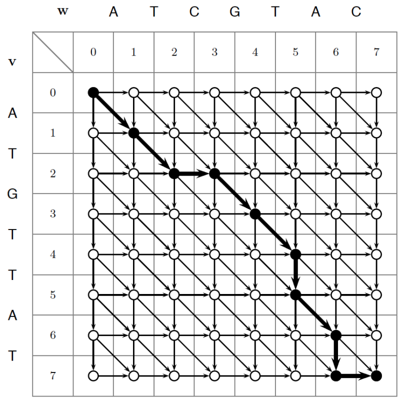
ATCCGAT

Alignment Matrices



The Edit Graph

		0	1	2	2	3	4	5	6	7	7
v	=		A	T	-	G	T	T	A	T	-
			|	|		|	|		|		
w	=		A	T	C	G	T	-	A	-	C
		0	1	2	3	4	5	5	6	6	7



The Edit Graph

- Could we apply *the dynamic programming scheme* we already know to the edit graph?
- The key question, however, is how to defined the weights of this graph, namely how to turn the graph into a *weighted DAG*.
- Please observe that:
  - Vertical and horizontal edges correspond to insertions and deletions.
  - Slanting edges correspond to matches and mismatches, depending on the characters involved.

Longest Common Subsequence

- Given two strings

$$\mathbf{v} = v_1 \cdots v_n \qquad \mathbf{w} = w_1 \cdots w_m$$

  a **common subsequence** of $\mathbf{v}$ and $\mathbf{w}$ is a pair of sequences of positions

$$1 \leq i_1 < i_2 < \ldots < i_k \leq n \qquad 1 \leq j_1 < j_2 < \ldots < j_k \leq m$$

  such that $v_{i_t} = w_{j_t}$ for every $1 \leq t \leq k$.

Longest Common Subsequence

- Given two strings

$$\mathbf{v} = v_1 \cdots v_n \qquad \mathbf{w} = w_1 \cdots w_m$$

a **common subsequence** of $\mathbf{v}$ and $\mathbf{w}$ is a pair of sequences of positions

$$1 \le i_1 < i_2 < \ldots < i_k \le n \qquad 1 \le j_1 < j_2 < \ldots < j_k \le m$$

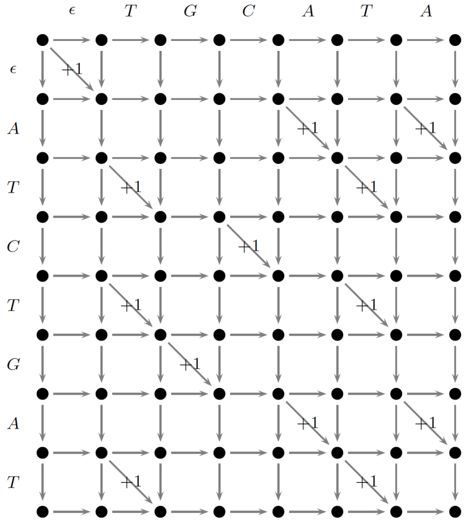
such that $v_{i_t} = w_{j_t}$ for every $1 \le t \le k$.

Longest Common Subsequence Problem:
Find the longest subsequence common to two strings.

Input: Two strings, $\mathbf{v}$ and $\mathbf{w}$.

Output: The longest common subsequence of $\mathbf{v}$ and $\mathbf{w}$.

How to Weight the Edit Graphs with LCS in Mind



A Dynamic Programming Algorithm for LCS

LCS($\mathbf{v}, \mathbf{w}$)

for $i \leftarrow 0$ **to** n

  $s_{i,0} \leftarrow 0$

for $j \leftarrow 1$ **to** m

  $s_{0,j} \leftarrow 0$

for $i \leftarrow 1$ **to** n

  for $j \leftarrow 1$ **to** m

$$s_{i,j} \leftarrow \max \begin{cases} s_{i-1,j} \\ s_{i,j-1} \\ s_{i-1,j-1} + 1, & \text{if } v_i = w_j \end{cases}$$

$$b_{i,j} \leftarrow \begin{cases} \text{“}\uparrow\text{''} & \text{if } s_{i,j} = s_{i-1,j} \\ \text{“}\leftarrow\text{''} & \text{if } s_{i,j} = s_{i,j-1} \\ \text{“}\nwarrow\text{''}, & \text{if } s_{i,j} = s_{i-1,j-1} + 1 \end{cases}$$

return $(s_{n,m}, \mathbf{b})$

A Dynamic Programming Algorithm for LCS

```
PRINTLCS(b, v, i, j)
  if i = 0 or j = 0
      return
  if b_{i,j} = "↖"
      PRINTLCS(b, v, i − 1, j − 1)
      print v_i
  else
      if b_{i,j} = "↑"
          PRINTLCS(b, v, i − 1, j)
      else
          PRINTLCS(b, v, i, j − 1)
```

Back to the Edit Distance

- How should we **weight** the edit graph while trying to compute the edit distance between two strings?
- Clearly:
  - Indels should cost 1.
  - Mismatches should cost 1.
  - Matches should cost 0.
- But this implies that computing the edit distance is a *minimization* rather than a *maximization* problem.
- The crucial recurrence is the following one:

$$s_{i,j} = \min \begin{cases} s_{i-1,j} + 1 & \\ s_{i,j-1} + 1 & \\ s_{i-1,j-1} & \text{if } v_i = w_j \\ s_{i-1,j-1} + 1 & \text{if } v_i \neq w_j \end{cases}$$

Back to the Edit Distance

- How should we **weight** the edit graph while trying to compute the edit distance between two strings?
- Clearly:
  - Indels should cost 1.
  - Mismatches should cost 1.
  - Matches should cost 0.
- But this implies that computing the edit distance is a *minimization* rather than a *maximization* problem.
- The crucial recurrence is the following one:

$$s_{i,j} = \min \begin{cases} s_{i-1,j} + 1 & \\ s_{i,j-1} + 1 & \\ s_{i-1,j-1} & \text{if } v_i = w_j \\ s_{i-1,j-1} + 1 & \text{if } v_i \neq w_j \end{cases}$$

Back to the Edit Distance

- How should we **weight** the edit graph while trying to compute the edit distance between two strings?
- Clearly:
  - Indels should cost 1.
  - Mismatches should cost 1.
  - Matches should cost 0.
- But this implies that computing the edit distance is a *minimization* rather than a *maximization* problem.
- The crucial recurrence is the following one:

$$s_{i,j} = \min \begin{cases} s_{i-1,j} + 1 & \\ s_{i,j-1} + 1 & \\ s_{i-1,j-1} & \text{if } v_i = w_j \\ s_{i-1,j-1} + 1 & \text{if } v_i \neq w_j \end{cases}$$

Global Sequence Alignment

- Sometimes, it makes a lot of sense to stipulate that certain edit operations have a different score than others.
- This can be modeled by a function

  $$\delta : \Sigma \cup \{-\} \times \Sigma \cup \{-\} \to \mathbb{R}_{\geq 0}$$

  which gives the score of any column in the alignment matrix.

Global Sequence Alignment

- Sometimes, it makes a lot of sense to stipulate that certain edit operations have a different score than others.
- This can be modeled by a function

$$\delta : \Sigma \cup \{-\} \times \Sigma \cup \{-\} \to \mathbb{R}_{\geq 0}$$

which gives the score of any column in the alignment matrix.

Global Alignment Problem:
Find the best alignment between two strings under a given scoring matrix.

Input: Strings $\mathbf{v}$, $\mathbf{w}$ and a scoring matrix δ.

Output: An alignment of $\mathbf{v}$ and $\mathbf{w}$ whose score (as defined by the matrix δ) is maximal among all possible alignments of $\mathbf{v}$ and $\mathbf{w}$.

Global Sequence Alignment

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \\ s_{i-1,j-1} + \delta(v_i, w_j) \end{cases}$$

Other Forms of Alignment

- There are at least three forms of alignment other than the global one.
  1. **Local Alignment Problem**
     - You are not looking for an alignment of the two string, but of segments of those.
  2. **Alignment with Gap Penalties**
     - Sometimes, there can be huge gaps between strings, and having a (negative) score which is linear in the length of the gap is an overkill.
  3. **Multiple Alignment**
     - Alignmentd between not two but many strings could possibly be looked for.
- In all these cases, the dynamic programming recipe can be applied, although the underlying edit graph needs to be adapted.

Other Forms of Alignment

- There are at least three forms of alignment other than the global one.
  1. **Local Alignment Problem**
     - You are not looking for an alignment of the two string, but of segments of those.
  2. **Alignment with Gap Penalties**
     - Sometimes, there can be huge gaps between strings, and having a (negative) score which is linear in the length of the gap is an overkill.
  3. **Multiple Alignment**
     - Alignmentd between not two but many strings could possibly be looked for.
- In all these cases, the dynamic programming recipe can be applied, although the underlying edit graph needs to be adapted.

Thank You!

Questions?