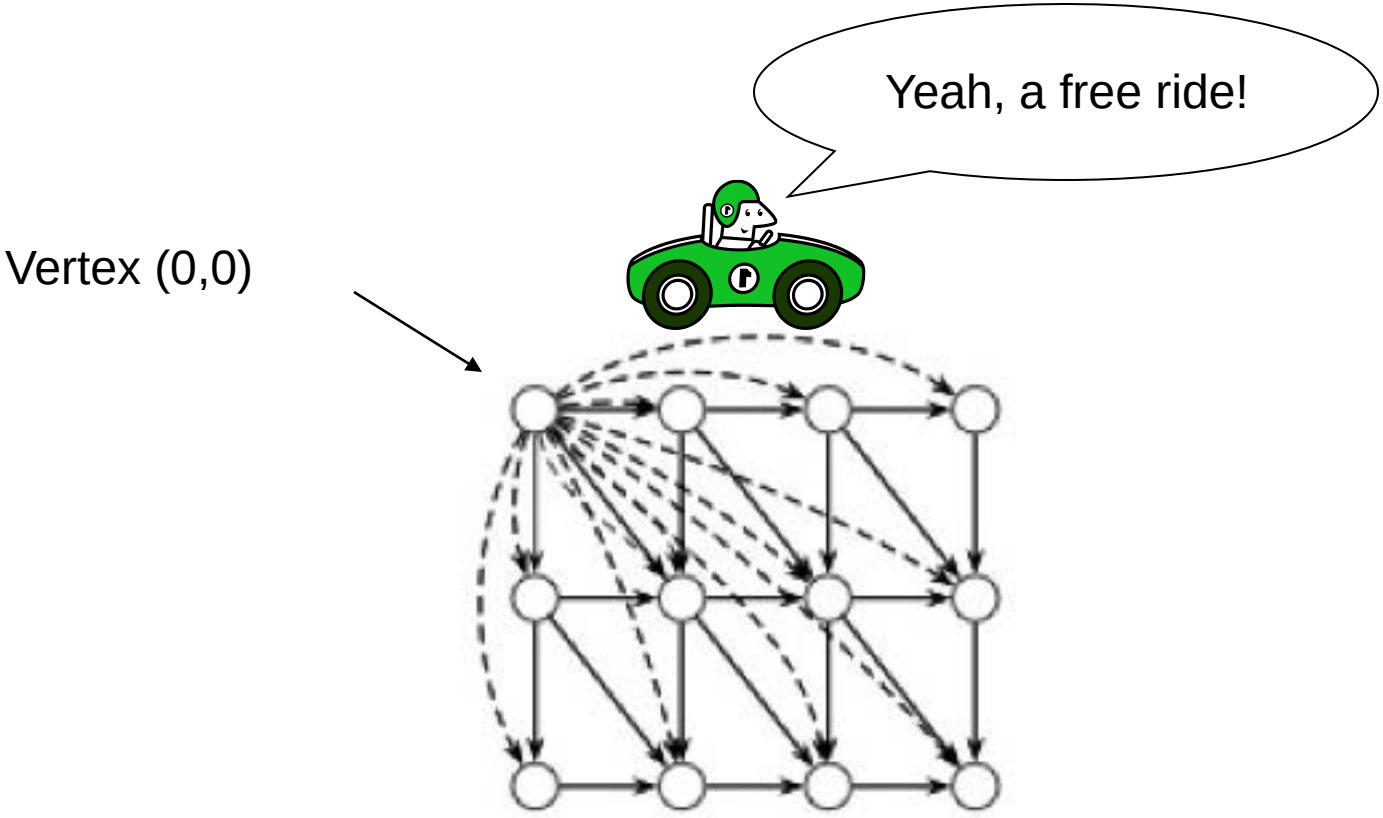
Local alignment: free rides



The dashed edges represent the free rides from (0,0) to every other node.

The local alignment recurrence

- The largest value of $s_{i,j}$ over the whole edit graph is the score of the best local alignment.
- The recurrence:

$$s_{i,j} = max \begin{cases} 0 \\ s_{i-1,j-1} + \delta(v_i, w_j) \\ s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \end{cases}$$

Notice there is only this change from the original recurrence of a Global Alignment

The local alignment recurrence

- The largest value of $s_{i,j}$ over the whole edit graph is the score of the best local alignment.

- The recurrence:

$$s_{i,j} = max \begin{cases} 0 \\ s_{i-1,j-1} + \delta(v_i, w_j) \\ s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \end{cases}$$

Power of ZERO: there is only this change from the original recurrence of a Global Alignment - since there is only one “free ride” edge entering into every vertex

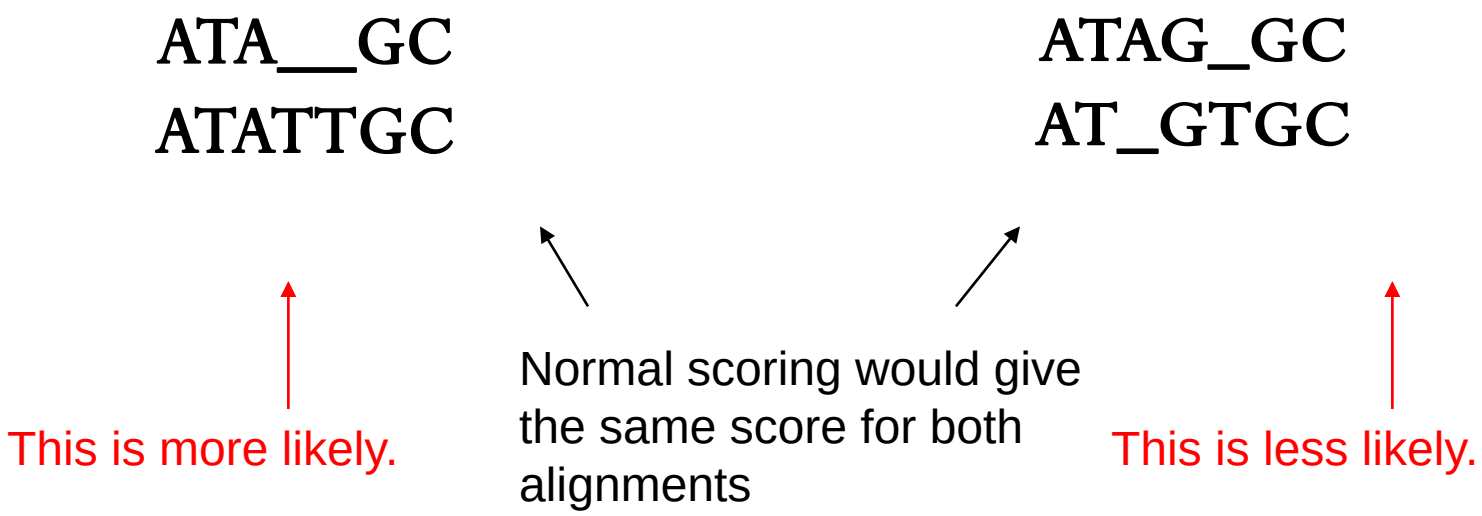
Scoring indels: naïve approach

- A fixed penalty σ is given to every indel:
  - $-\sigma$ for 1 indel,
  - -2σ for 2 consecutive indels
  - -3σ for 3 consecutive indels, etc.

Can be too severe penalty for a series of 100 consecutive indels

Affine gap penalties

- In nature, a series of *k* indels often come as a single event rather than a series of *k* single nucleotide events:



Accounting for gaps

- *Gaps*- contiguous sequence of spaces in one of the rows

- Score for a gap of length x is:

  $-(\rho + \sigma x)$

  where $\rho > 0$ is the penalty for introducing a gap:

  gap opening penalty

  ρ will be large relative to σ:

  gap extension penalty

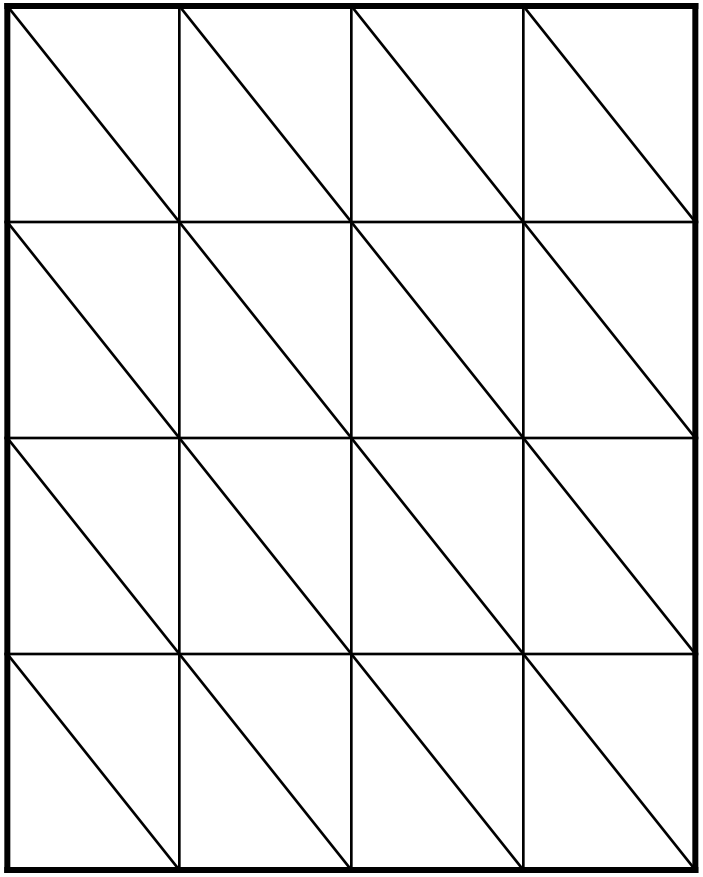
  because you do not want to add too much of a penalty for extending the gap.

Affine gap penalties

- Gap penalties:
  - $-\rho-\sigma$ when there is 1 indel
  - $-\rho-2\sigma$ when there are 2 indels
  - $-\rho-3\sigma$ when there are 3 indels, etc.
  - $-\rho-x\cdot\sigma$ (-gap opening - *x* gap extensions)

- Somehow reduced penalties (as compared to naïve scoring) are given to runs of horizontal and vertical edges

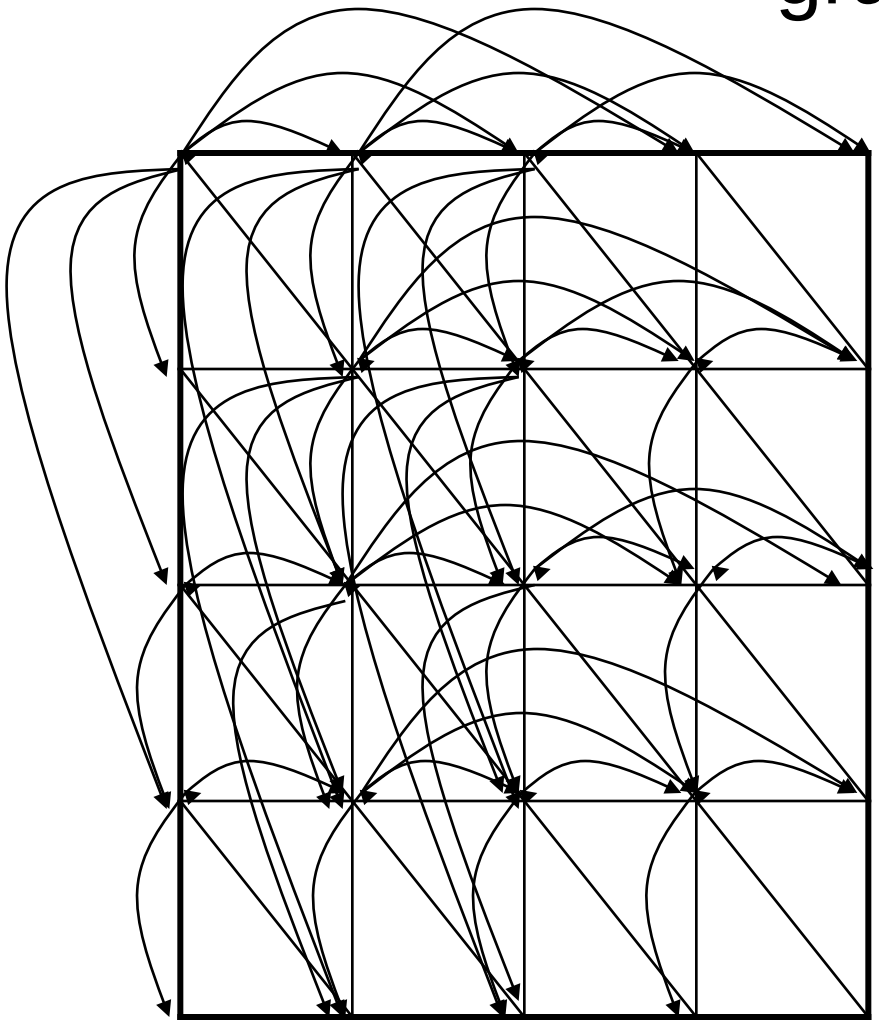
Affine gap penalties and edit graph



To reflect affine gap penalties we have to add “long” horizontal and vertical edges to the edit graph. Each such edge of length *x* should have weight

$$-\rho - x * \sigma$$

Adding “affine penalty” edges to the edit graph

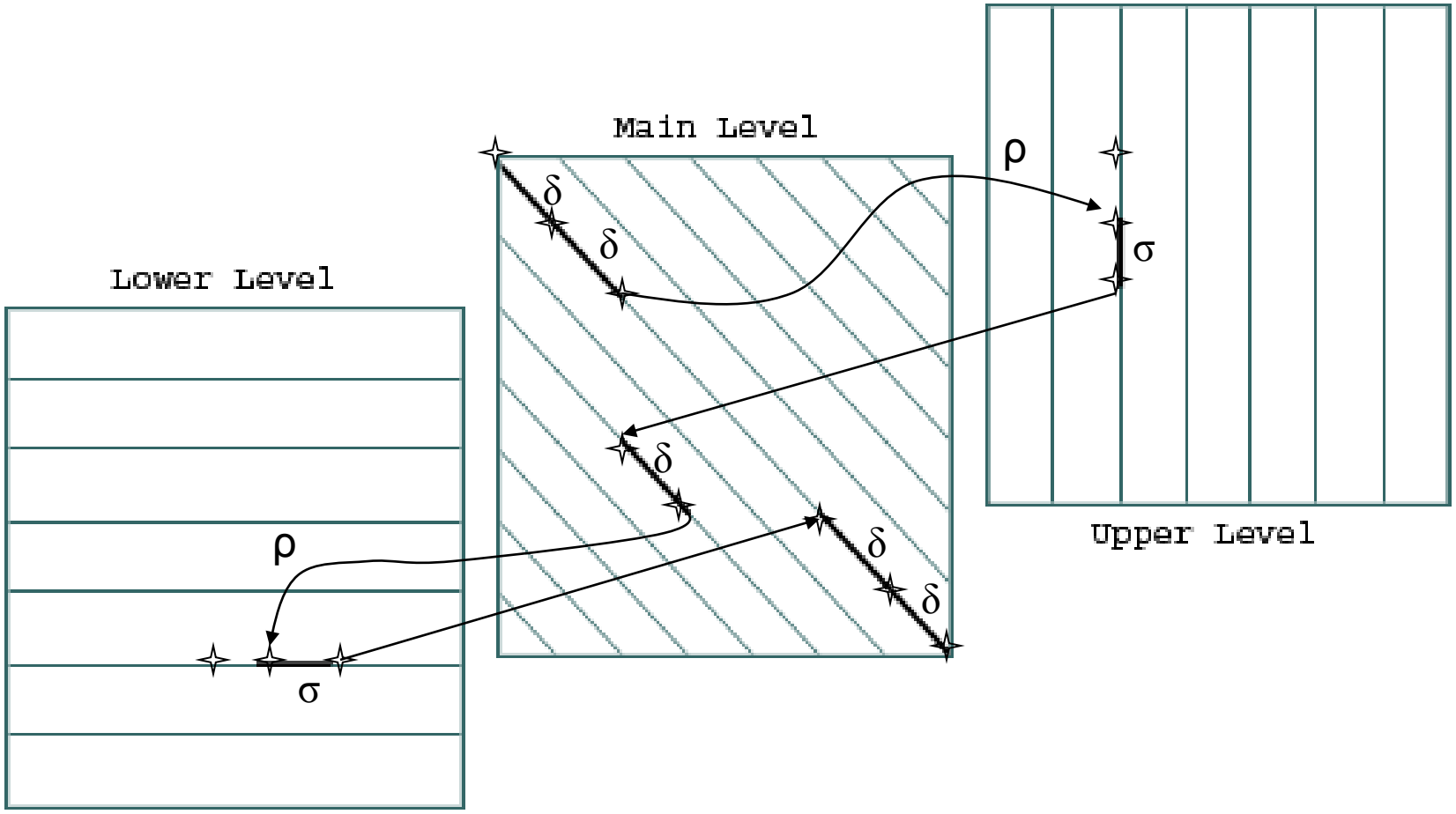


There are many such edges!

Adding them to the graph increases the running time of the alignment algorithm by a factor of ***n*** (where ***n*** is the number of vertices)

So the complexity increases from $O(n^2)$ to $O(n^3)$

Create three layers



Affine gap penalties and 3-layer edit graph

- The three recurrences for the scoring algorithm creates a 3-layered graph.
- The top level creates/extends gaps in the sequence *w.*
- The bottom level creates/extends gaps in sequence *v.*
- The middle level extends matches and mismatches.

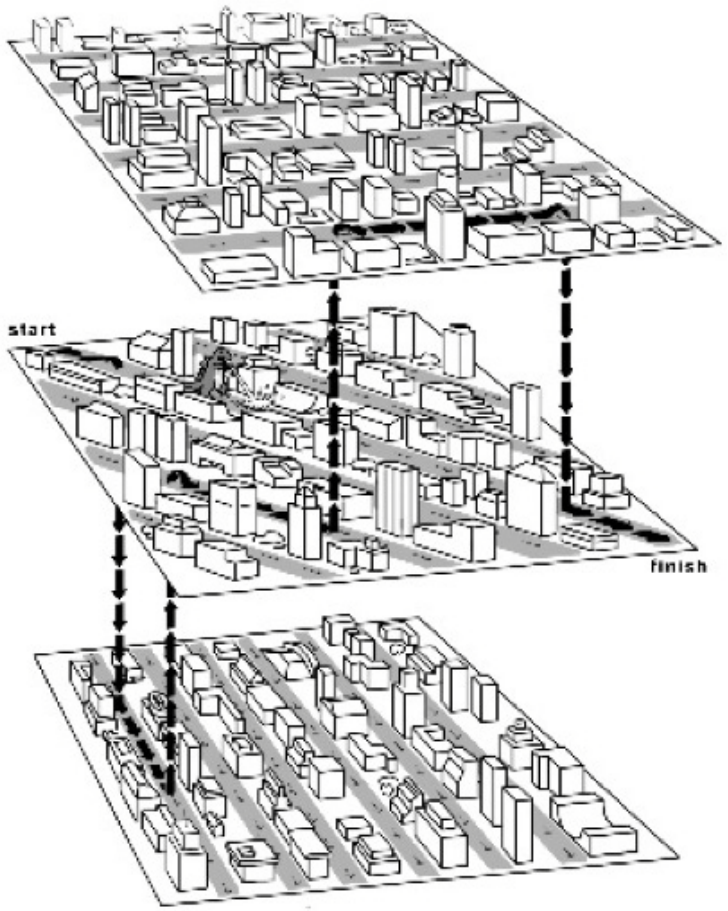
Switching between the three layers

- Levels:
  - The **main level** is for diagonal edges
  - The **lower level** is for horizontal edges
  - The **upper level** is for vertical edges

- A jumping penalty is assigned to moving from the main level to either the upper level or the lower level (-*r*- *s*)

- There is a gap extension penalty for each continuation on a level other than the main level (-*s*)

Three level city grid



Gaps in w

Matches/Mismatches

Gaps in v

Affine gap penalty recurrences

$$\downarrow s_{i,j} = \max \begin{cases} \downarrow s_{i-1,j} - \sigma \\ s_{i-1,j} - (\rho+\sigma) \end{cases}$$

Continue Gap in *w* (deletion)

Start Gap in *w* (deletion): from middle

$$\rightarrow s_{i,j} = \max \begin{cases} \rightarrow s_{i,j-1} - \sigma \\ s_{i,j-1} - (\rho+\sigma) \end{cases}$$

Continue Gap in *v* (insertion)

Start Gap in *v* (insertion):from middle

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

Match or Mismatch

End deletion: from top

End insertion: from bottom

Space efficient dynamic programming

Quadratic space will kill you faster than quadratic time.

$(30,000)^2$ bytes equals one gigabyte, while $(30,000)^2$ operations takes 1000 seconds on a machine doing 1 million steps per second.

The dynamic programming algorithms we have seen only look at neighboring rows/columns to make their decision.

Computing the *highest cell score* in the matrix does not require keeping more than then last column and the best value to date, for a total of $O(n)$ space, where $n \leq m$.

Note that reconstructing the optimal *alignment* does seem to require keeping the entire matrix, however.

But Hirshberg found a clever way to reconstruct the alignment in $O(nm)$ time using only $O(n)$ space, by recomputing the appropriate portions of the matrix.

Space efficient dynamic programming

For each cell, we drag along the row number where the optimal path to in crossed the middle (m/2nd) column.

This requires only $O(n)$ extra memory, one cell per row.

This works because the crossing point k of the $(m/2)$nd column means that the optimal alignment lies in the sub matrices A from (1,1) to $(m/2, k)$, and B from $(m/2, k)$ to (m, n).

Note that the number of cells in A and B totals only half of the the original mn cells.

Further, these dynamic programming algorithms are linear in the number of cells they compute.

Thus the total amount of recomputation done is

$$\sum_{i=0}^{\lg m} mn/2^i = 2mn$$

so the total work remains $O(mn)$