CS 122/222 Week 6

Discussion 1A

Rosemary He
Slides adapted from Nathan LaPierre

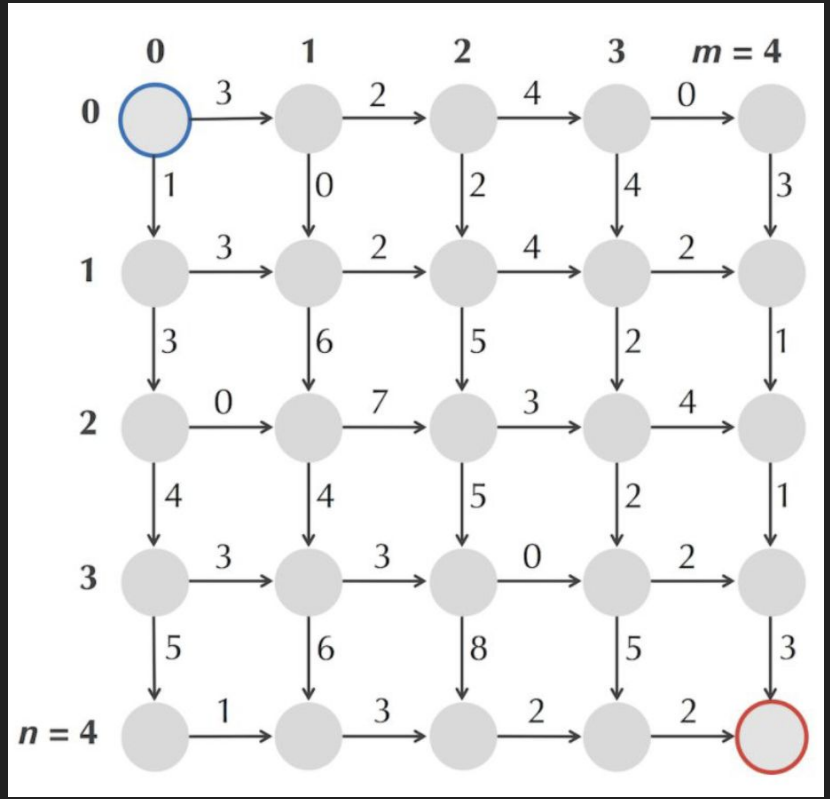
We will wait until 12:03

Announcements

- Stepik chap 3 due May 8 Saturday
- Project 2 due May 18 Tuesday

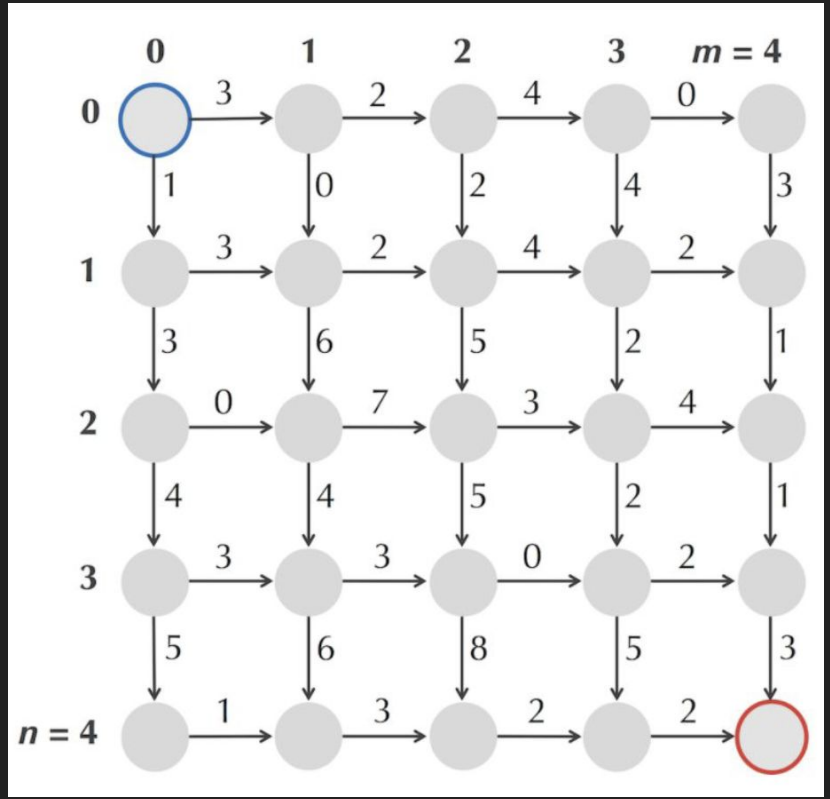
Manhattan Tourist Problem

Goal is to move from top-left of grid (“source”) to bottom-right (“sink”) while covering path with maximum sum of edge weights (e.g. “sights seen in Manhattan”)



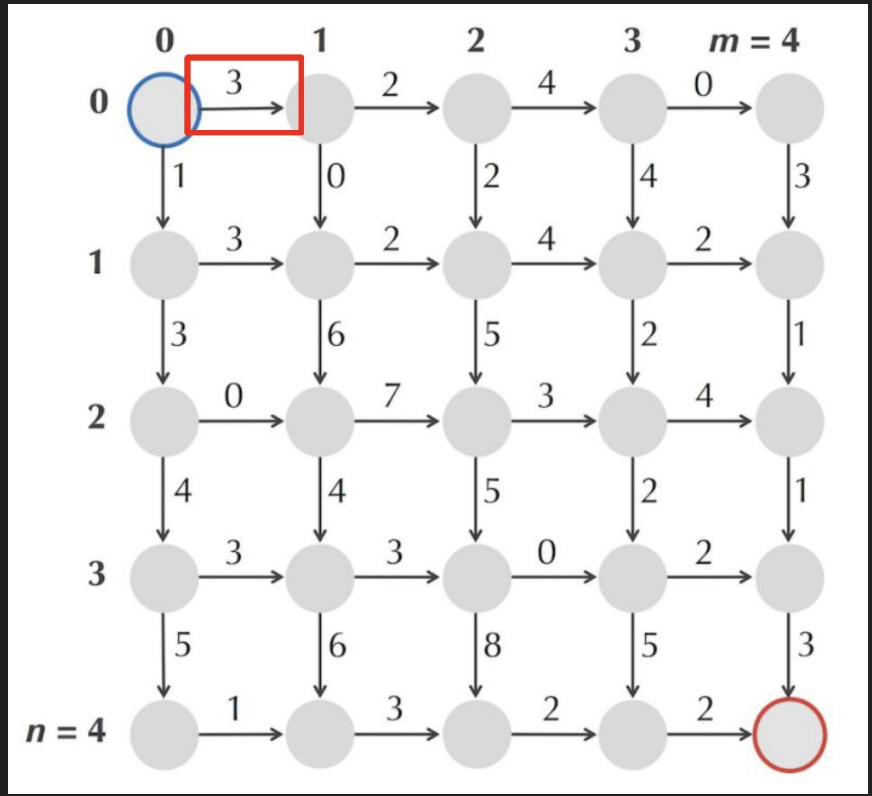
Greedy Algorithm

A “greedy algorithm” takes the optimal (highest-weight) choice at each possible step. Starting from the (blue) source node, what is the first edge taken by a greedy algorithm?



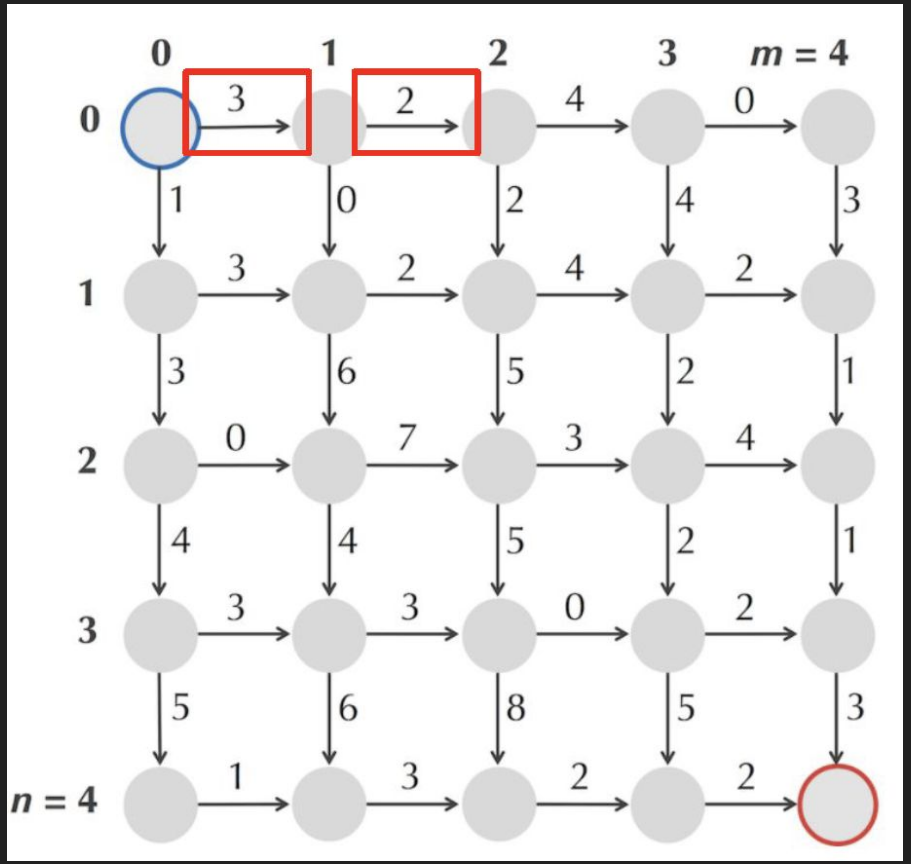
Greedy Algorithm

A “greedy algorithm” takes the optimal (highest-weight) choice at each possible step.



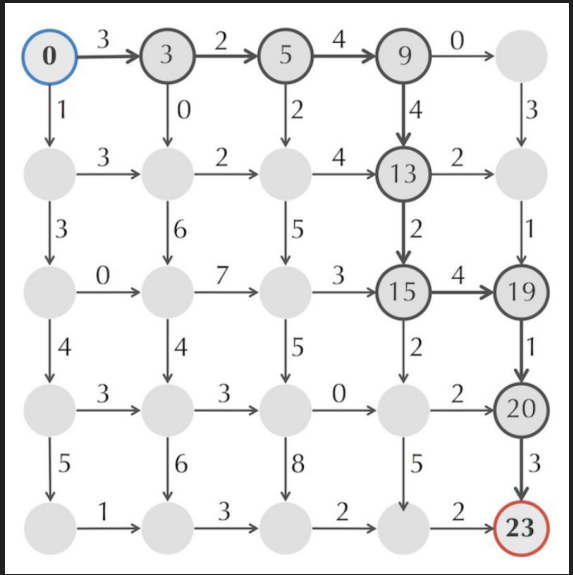
Greedy Algorithm

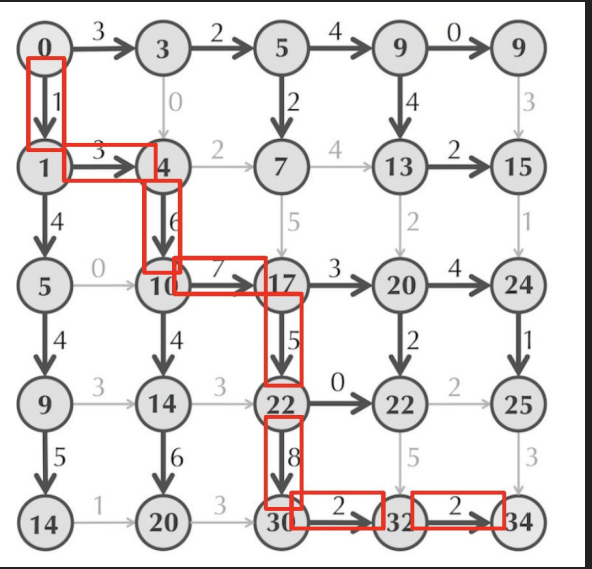
A “greedy algorithm” takes the optimal (highest-weight) choice at each possible step.



Greedy Algorithm is not optimal

A “greedy algorithm” (left) takes the optimal choice at each possible step. However, looking at the optimal solution on the right, we can see that the greedy algorithm does NOT always find the optimal solution.





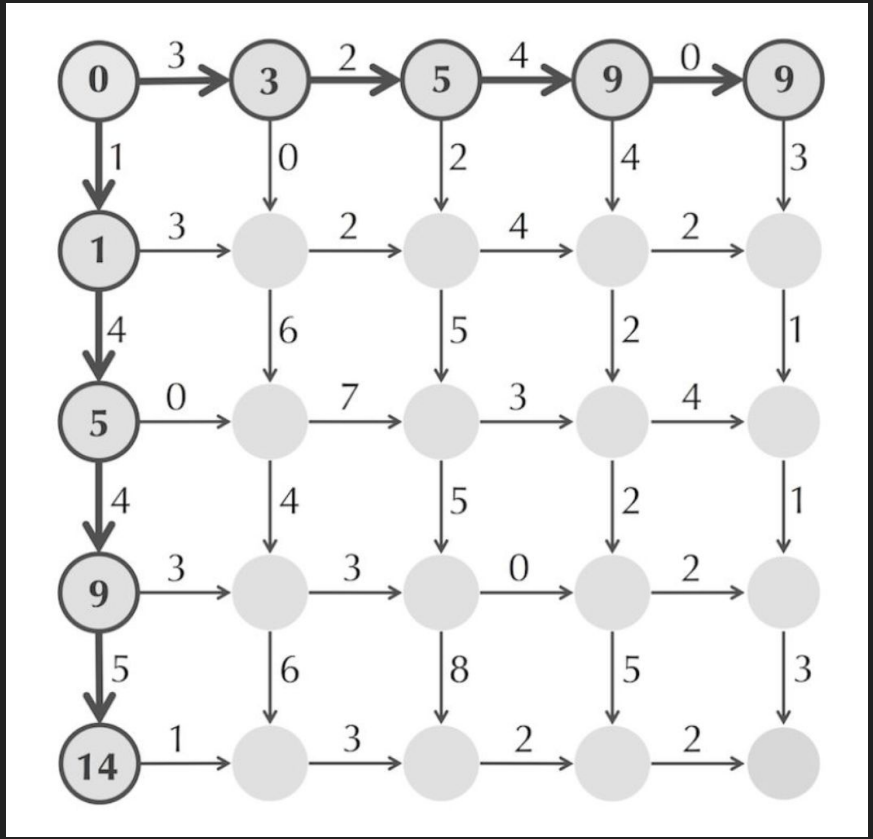
Dynamic Programming

One way to find the optimal path efficiently is to build up the solution from the bottom. We want to find the optimal path to each node X. We can do this by finding the optimal path to each of the nodes that could lead to X, and then comparing those values and picking the path into X that has the highest value.

This strategy of optimizing a problem by iteratively solving subproblems and saving their solutions is called dynamic programming and is very popular in all fields of computer science.

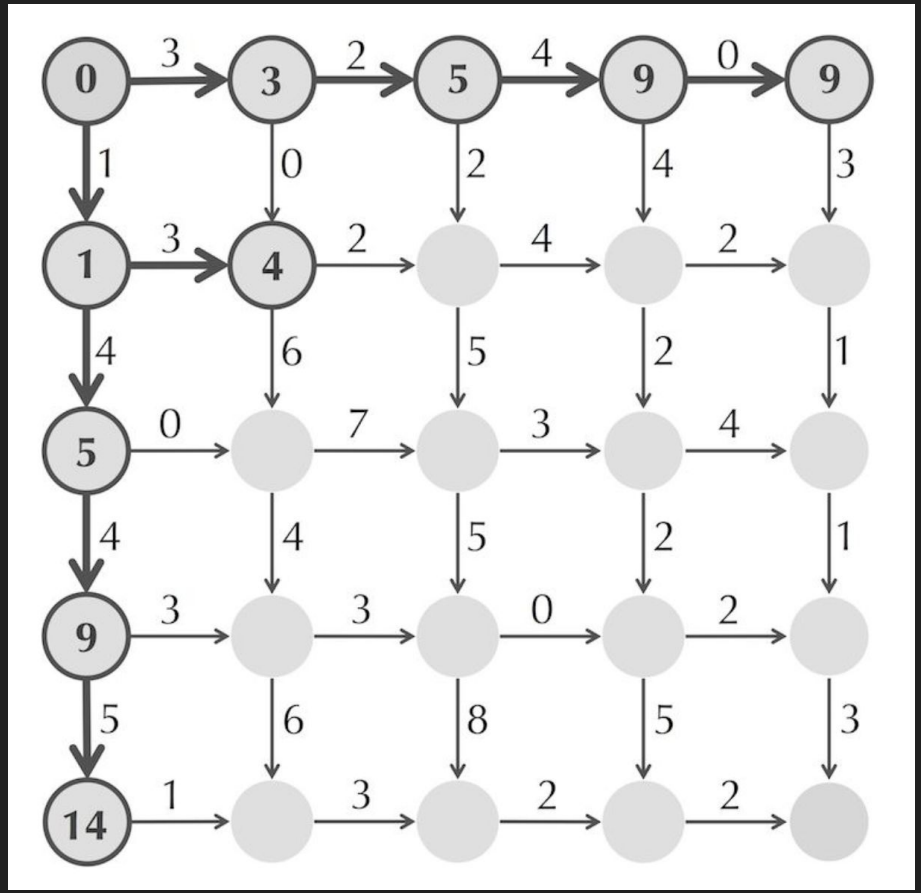
Dynamic Programming

The optimal way to get to any of the nodes on the top row or leftmost column is to travel in a straight line. Since we can’t ever travel up or left, so we can fill those paths in right away.



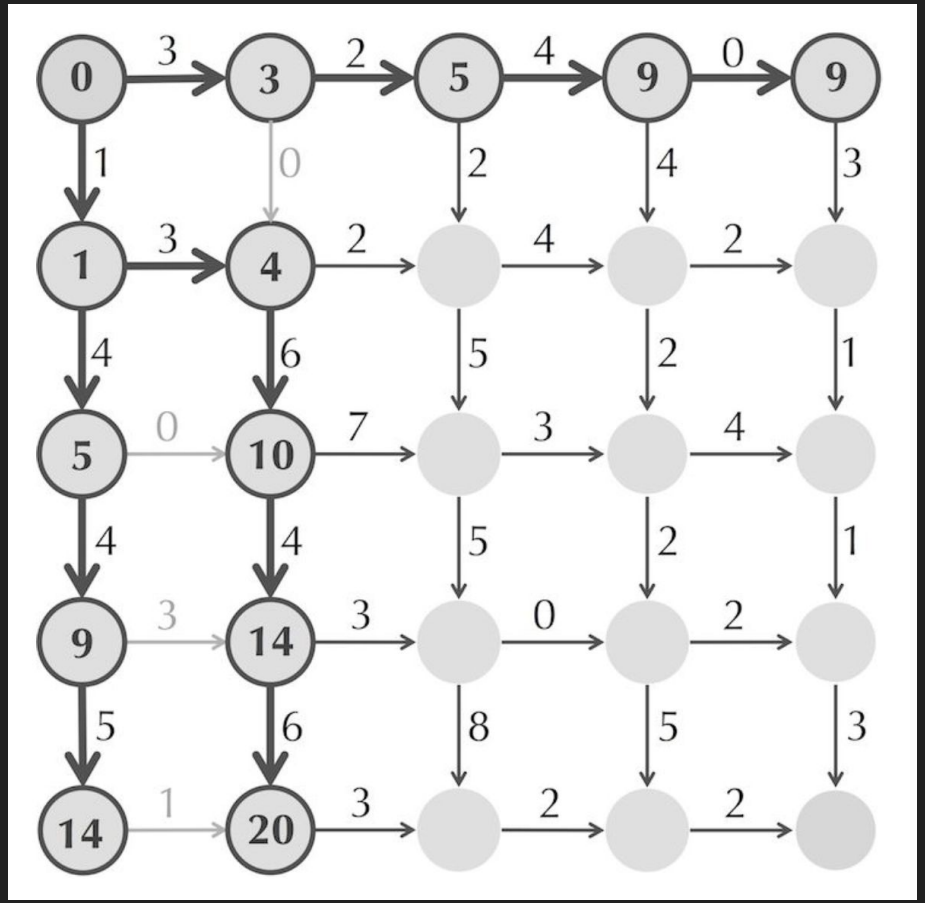
Dynamic Programming

We can get to node (1,1) either by starting from node (0, 1) with value 3 and adding the edge weight 0, or starting from node (1, 0) with value 1 and adding the edge weight 3. It turns out the latter is higher value (4), so we choose that path to node (1, 1) and give it a value of 4.



Dynamic Programming

We can get to node (2, 1) from either node (1, 1) with value 4 plus edge weight 6, or node (2, 0) with value 5 plus edge weight 0. Clearly, the path from (1, 1) with value 10 is better. Likewise for all nodes in column 1.

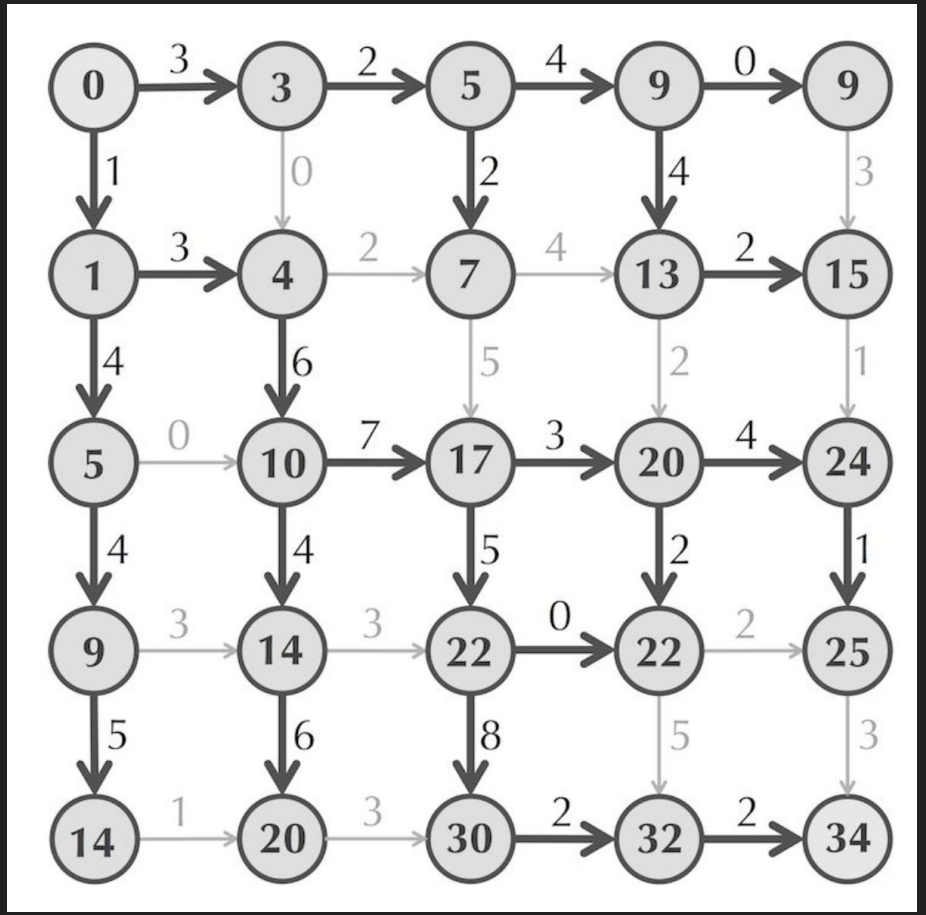


Dynamic Programming

We can fill in all nodes the same way, finding that the optimal sum of weights is 34 at the sink node. Thus, our general algorithm is:

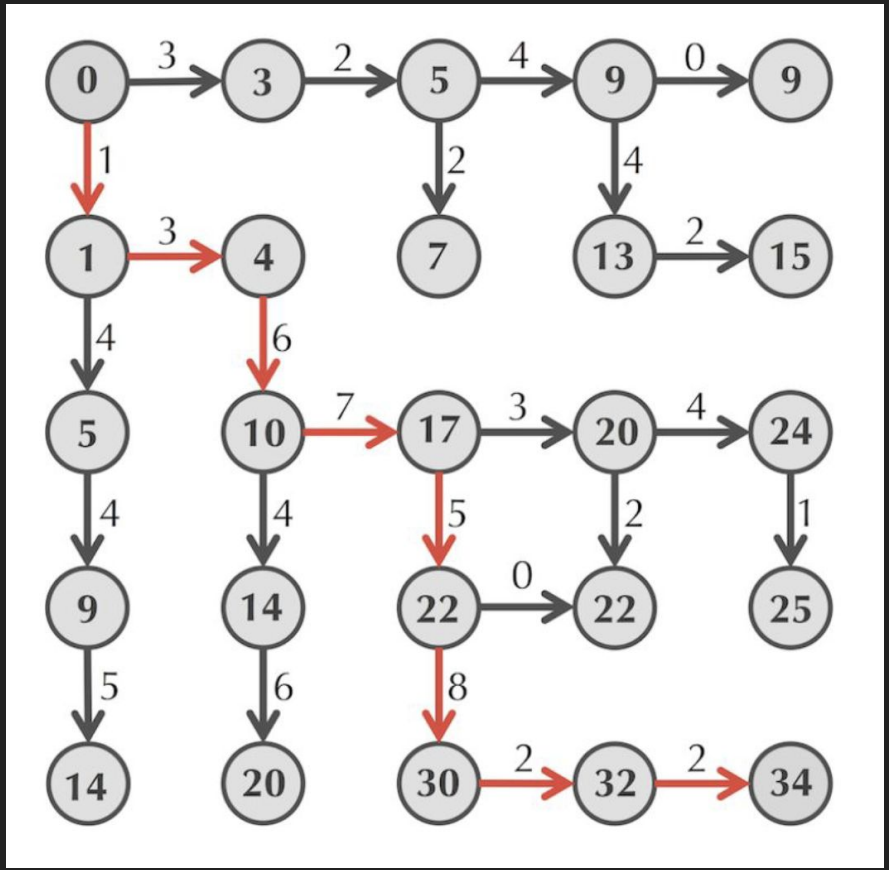
The optimal path to node (i, j) is the maximum of these two choices:

- Value of node (i-1, j) plus edge weight from (i-1, j) to (i, j)
- Value of node (i, j-1) plus edge weight from (i, j-1) to (i, j)



Backtracking to find path

Using DP we were able to maximize the score for the Manhattan Tourist problem. But we also want to know the path traveled. To do that, you need to make another matrix where each node stores the previous node chosen to maximize its value. Then when you get to the sink node, you can “backtrack” backwards through the path of nodes back to the source node to retrace the optimal path.



Sequence Alignment

We want to align two sequences, ideally matching as many bases as possible. Imagine we have a reference genome segment (top row) and a read we want to align to it (bottom row):

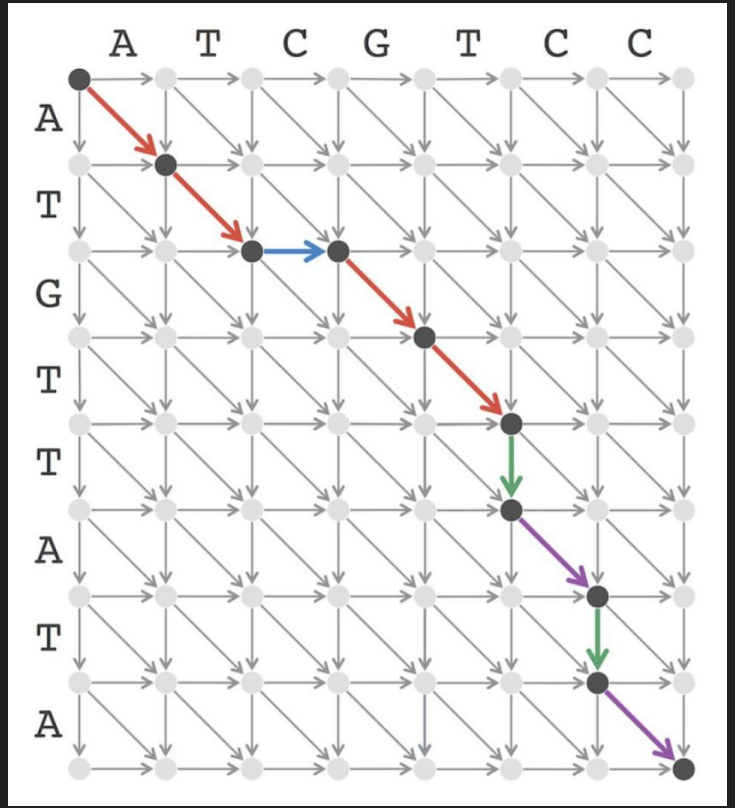
Using the solution on the left, we can see that there are no matches! But clearly, the solution on the right is more optimal. Here, the read has a “deletion” of the first letter of the reference, and an “insertion” relative to the reference in terms of the last letter, and that all other letters match.

Sequence Alignment

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

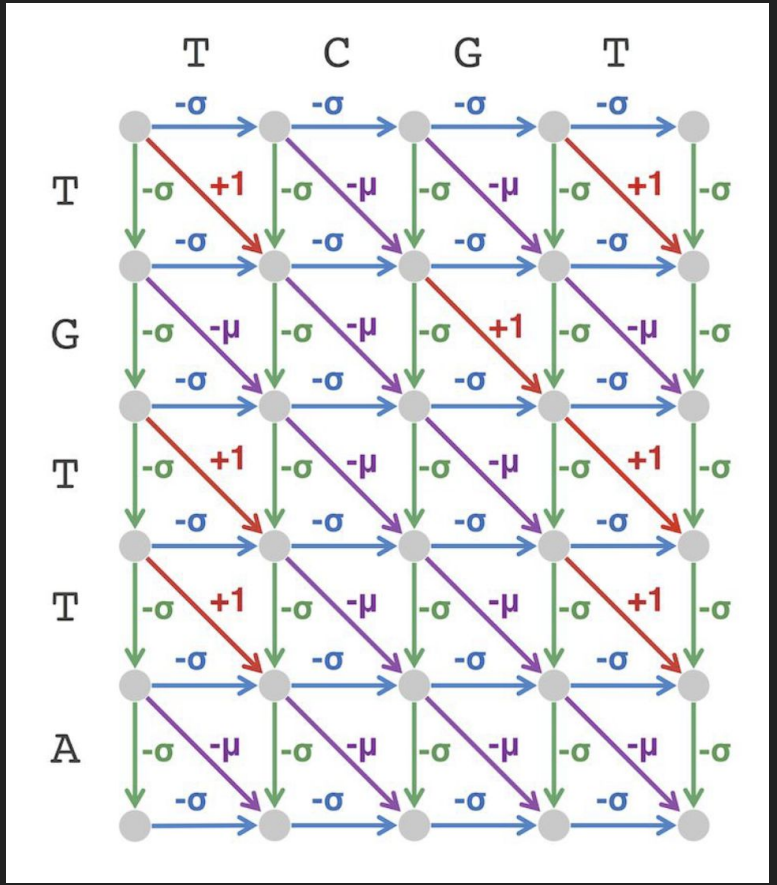
We can clearly visualize the optimal alignment problem as a graph problem similarly to the Manhattan Tourist problem, except now we have three options:

- An insertion uses a letter of the read but not the reference, so it goes right
- A deletion uses a letter of the reference but not the read, so it goes down
- A match/mismatch, uses a letter of both the read and the reference, so it goes diagonally



Scoring Alignment

In the simplest form of scoring, we assign a negative score for an inserted or deleted base (-σ here), a negative or zero score for a mismatch (-μ here), and a positive score (+1 here) for a match.
Note that we have -μ in the diagonal where letters are different, and +1 where they are the same.

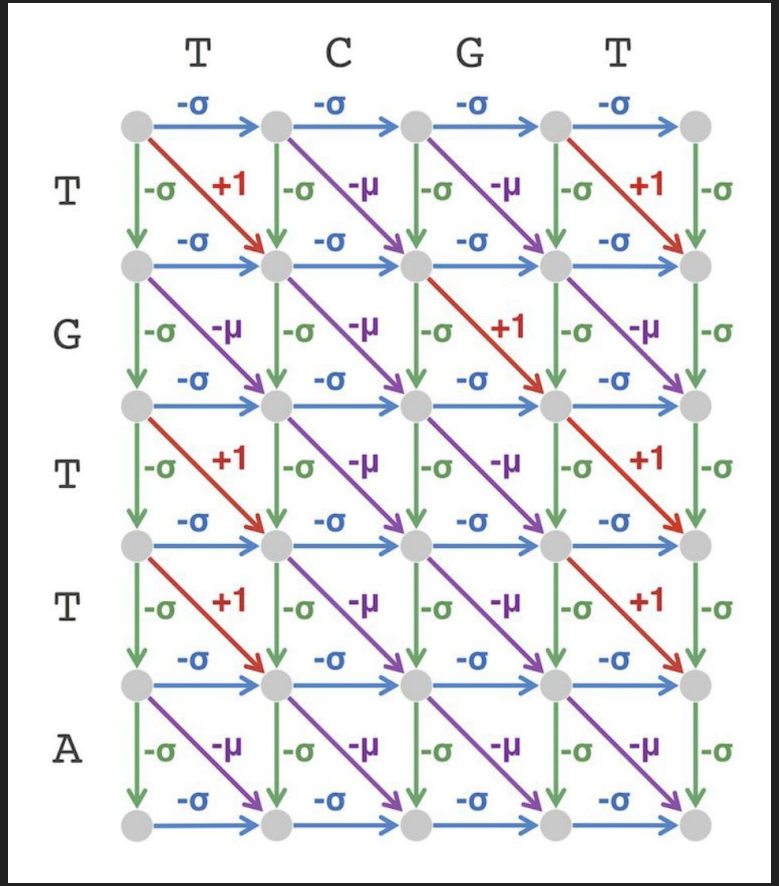


Global Alignment

We want to find the best alignment between two entire sequences.
We can use the DP solution from before, except now each node has three possible entry nodes:

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

The DP solution to the global alignment problem is called the Needleman-Wunsch algorithm.



Practice

Use the Needleman-Wunsch Algorithm to find the global alignment of sequences “ATGGCGT” and “ATGAGT”. (Scoring scheme: Match = 3, mismatch = -1, gap = -2)

Credit: Tommer Schwarz

Use the Needleman-Wunsch Algorithm to find the global alignment of sequences “ATGGCGT” and “ATGAGT”. (Scoring scheme: Match = 3, mismatch = -1, gap = -2)

		A	T	G	A	G	T
	0						
A							
T							
G							
G							
C							
G							
T							

Use the Needleman-Wunsch Algorithm to find the global alignment of sequences “ATGGCGT” and “ATGAGT”. (Scoring scheme: Match = 3, mismatch = -1, gap = -2)

		A	T	G	A	G	T
	0	-2	-4	-6	-8	-10	-12
A	-2						
T	-4						
G	-6						
G	-8						
C	-10						
G	-12						
T	-14						

Use the Needleman-Wunsch Algorithm to find the global alignment of sequences “ATGGCGT” and “ATGAGT”. (Scoring scheme: Match = 3, mismatch = -1, gap = -2)

		A	T	G	A	G	T
	0	-2	-4	-6	-8	-10	-12
A	-2	3	1	-1	-3	-5	-7
T	-4						
G	-6						
G	-8						
C	-10						
G	-12						
T	-14						

Use the Needleman-Wunsch Algorithm to find the global alignment of sequences “ATGGCGT” and “ATGAGT”. (Scoring scheme: Match = 3, mismatch = -1, gap = -2)

		A	T	G	A	G	T
	0	-2	-4	-6	-8	-10	-12
A	-2	3	1	-1	-3	-5	-7
T	-4	1					
G	-6	-1					
G	-8	-3					
C	-10	-5					
G	-12	-7					
T	-14	-9					

Use the Needleman-Wunsch Algorithm to find the global alignment of sequences “ATGGCGT” and “ATGAGT”. (Scoring scheme: Match = 3, mismatch = -1, gap = -2)

		A	**T**	**G**	**A**	**G**	**T**
	0	-2	-4	-6	-8	-10	-12
A	-2	**3**	1	-1	-3	-5	-7
T	-4	1	**6**	4	2	0	-2
G	-6	-1	4	**9**	7	5	3
G	-8	-3	2	**7**	**8**	10	8
C	-10	-5	0	5	**6**	8	9
G	-12	-7	-2	3	4	**9**	7
T	-14	-9	-4	1	2	7	

Use the Needleman-Wunsch Algorithm to find the global alignment of sequences “ATGGCGT” and “ATGAGT”. (Scoring scheme: Match = 3, mismatch = -1, gap = -2)

		A	**T**	**G**	**A**	**G**	**T**
	0	-2	-4	-6	-8	-10	-12
A	-2	**3**	1	-1	-3	-5	-7
T	-4	1	**6**	4	2	0	-2
G	-6	-1	4	**9**	7	5	3
G	-8	-3	2	**7**	**8**	10	8
C	-10	-5	0	5	**6**	8	9
G	-12	-7	-2	3	4	**9**	7
T	-14	-9	-4	1	2	7	**12**

Now, what are the optimal alignment(s), using the bolded numbers?

		A	T	G	A	G	T
	0	-2	-4	-6	-8	-10	-12
A	-2	**3**	1	-1	-3	-5	-7
T	-4	1	**6**	4	2	0	-2
G	-6	-1	4	**9**	7	5	3
G	-8	-3	2	**7**	**8**	10	8
C	-10	-5	0	5	**6**	8	9
G	-12	-7	-2	3	4	**9**	7
T	-14	-9	-4	1	2	7	**12**

		A	T	G	A	G	T
	0	-2	-4	-6	-8	-10	-12
A	-2	**3**	1	-1	-3	-5	-7
T	-4	1	**6**	4	2	0	-2
G	-6	-1	4	**9**	7	5	3
G	-8	-3	2	**7**	**8**	10	8
C	-10	-5	0	5	**6**	8	9
G	-12	-7	-2	3	4	**9**	7
T	-14	-9	-4	1	2	7	**12**

ATGA-GT
ATGGCGT

ATG-AGT
ATGGCGT

Local Alignment

Now suppose we want to find the sub-sequence of two strings with the highest alignment score. For instance, consider this global alignment solution:

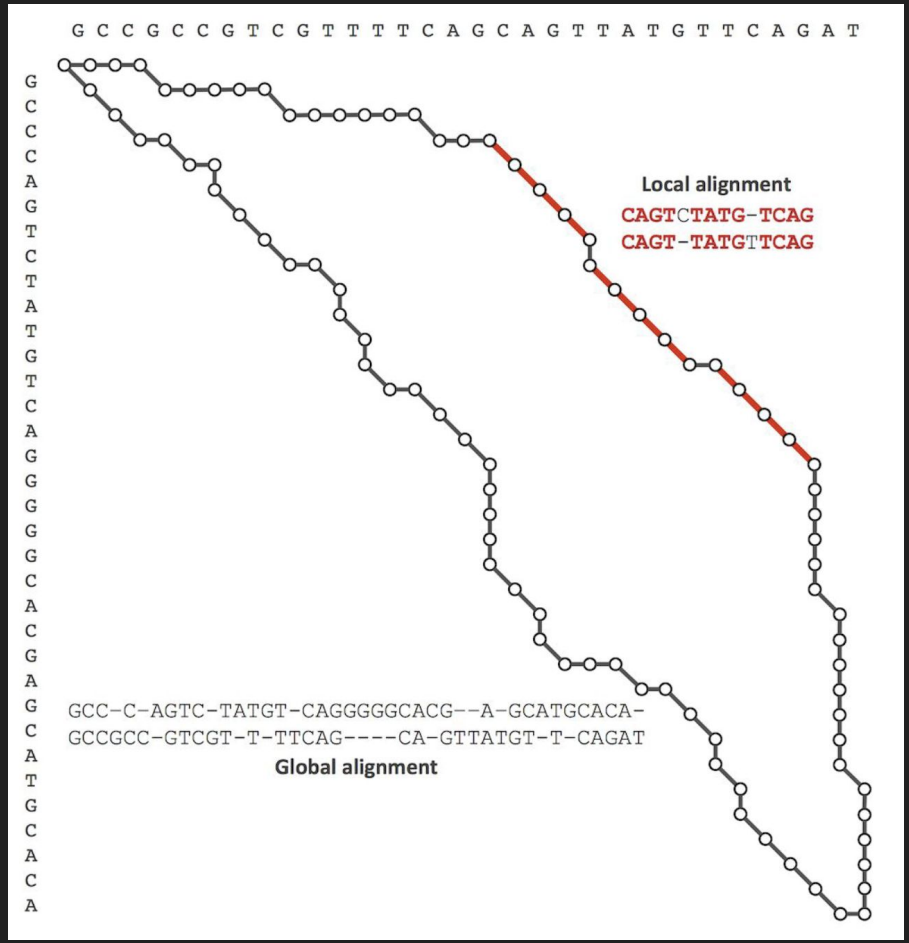
```
GCC-C-AGTC-TATGT-CAGGGGGCACG--A-GCATGCACA-
GCCGCC-GTCGT-T-TTCAG----CA-GTTATGT-T-CAGAT
```

But suppose we only want to match one part of these strings really well, and we don’t care about the rest. We may instead prefer this local alignment solution:

```
---G----C-----C--CAGTCTATG-TCAGGGGGCACGAGCATGCACA
GCCGCCGTCGTTTTCAGCAGT-TATGTTCAG-----A------T-----
```

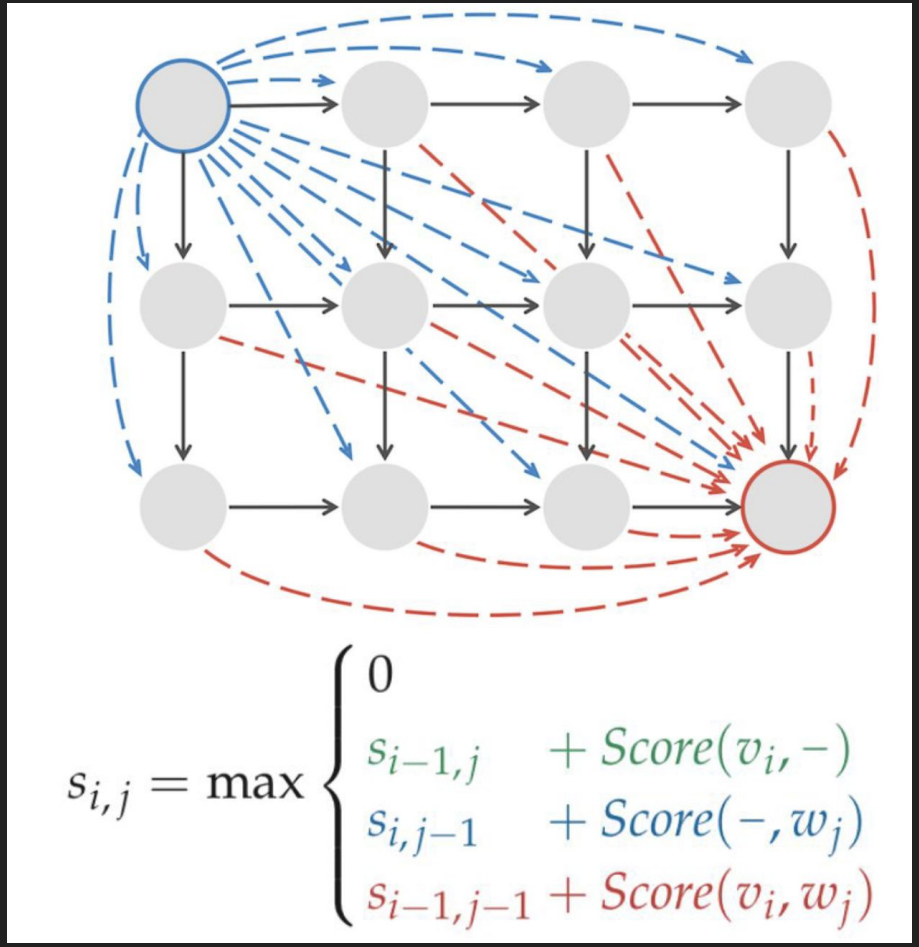
Local Alignment

The path through the alignment graph is totally different.
What we want is a “free ride” (with 0 penalty for indels/mismatches) to the part of the graph where the optimal local alignment takes place.



Local Alignment

We connect the source node and sink node to every other edge in the graph with a “zero” edge weight and then add an option to our DP algorithm to take the “free ride” from the source node or to the sink node with zero penalty!
This is the “Smith-Waterman” algorithm, and it is the local alignment analogue of the Needleman-Wunsch algorithm.



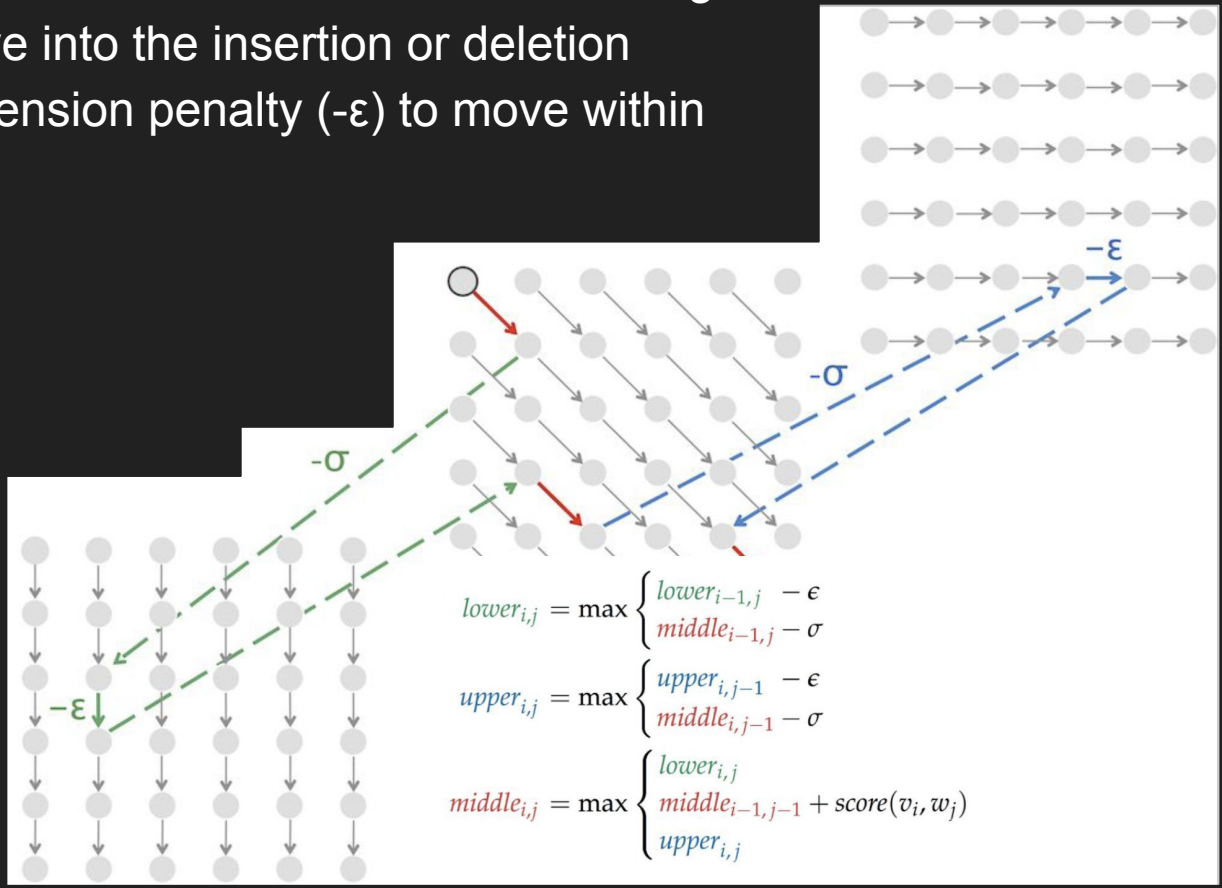
Affine gap penalty

A problem to consider: one single multi-base indel (right) is more likely to occur than separate single-base indels (left).

To deal with this, we introduce an “affine gap” penalty, where there is a larger penalty for “opening” a gap (inserting or deleting a base) than there is for “extending” a gap (e.g. adding another base to the indel).

In practice, we can have three separate graphs for matches/mismatches, insertions, and deletions. We take a large gap open penalty (-σ) to move into the insertion or deletion graph, and a smaller gap extension penalty (-ε) to move within them.

Since there is no penalty to close a gap, there is a zero-weight edge from graphs to the correspond node in the match/misma graph.



Tips

- See the end of the chapter for more space-efficient implementation of these algorithms.
- Needleman-Wunsch and Smith-Waterman are time-consuming algorithms. Use them only sparingly.
  - Namely, you don't need to use them if your read matches to the reference perfectly or clearly just has one or two SNPs or errors. Only use them if you may have an indel.
- How can you tell if you have an indel?
  - Say (for example) you're using the hashing algorithm. The different segments of your read may each line up to the reference, but slightly offset from one another.
  - If you try putting the segments right next to each other, it will generate many mismatches (since the part of the read after the indel will be offset from the reference by the length of the indel).
- Optionally, you can first solve SNPs, then separately find indels, and combine your two files later.
- You may want to just compute your hash index or BWT + FM index once, then store it and load it later for subsequent runs. There's no need to recompute these every time you run your program.
- Limit the length of indels you're willing to accept. Don't try to run NW or SW to align your read to the reference when the read segments map thousands of bases apart.
- Look at the answer files for the example datasets to give an idea of roughly how frequently you should expect to see indels and their length distribution. That way you'll have an idea of whether you're calling way too many or way too few.

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/14_dp_in_less_time_space_v2.pdf