CSE 421: Algorithms

Winter 2014

Lecture 16: Sequence alignment and Bellman-Ford

Reading:
Sections 6.6-6.10

sequence alignment: edit distance

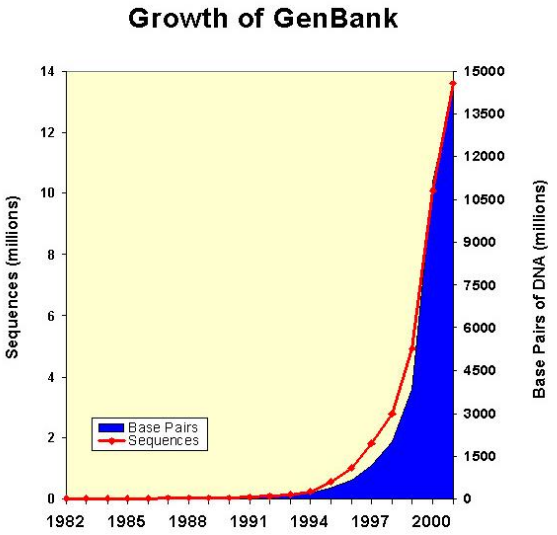
- Given:
  - Two strings of characters $A=a_1\ a_2 \ldots a_n$ and $B=b_1\ b_2 \ldots b_m$
- Find:
  - The minimum number of edit steps needed to transform A into B where an edit can be:
  - insert a single character
  - delete a single character
  - substitute one character by another

sequence alignment vs edit distance

- Sequence Alignment
  - Insert corresponds to aligning with a "–" in the first string
    Cost δ (in our case 1)
  - Delete corresponds to aligning with a "–" in the second string
    Cost δ (in our case 1)
  - Replacement of an a by a b corresponds to a mismatch
    Cost α_{ab} (in our case 1 if a≠b and 0 if a=b)
- In Computational Biology this alignment algorithm is attributed to Smith & Waterman

applications

- "diff" utility – where do two files differ
- Version control & patch distribution – save/send only changes
- Molecular biology
  - Similar sequences often have similar origin and function
  - Similarity often recognizable despite millions or billions of years of evolutionary divergence



recursive solutions

- **Sub-problems:** Edit distance problems for **all prefixes** of **A** and **B** that don't include all of both **A** and **B**
- Let **D(i,j)** be the number of edits required to transform $a_1 a_2 \ldots a_i$ into $b_1 b_2 \ldots b_j$
- Clearly **D(0,0)=0**

computing D(n,m)

- Imagine how best sequence handles the last characters a_n and b_m
- If best sequence of operations
  - deletes a_n then **D(n,m)=**
  - inserts b_m then **D(n,m)=**
  - replaces a_n by b_m then **D(n,m)=**
  - matches a_n and b_m then **D(n,m)=**

computing D(n,m)

- Imagine how best sequence handles the last characters a_n and b_m
- If best sequence of operations
  - deletes a_n then **D(n,m)=D(n-1,m)+1**
  - inserts b_m then **D(n,m)=D(n,m-1)+1**
  - replaces a_n by b_m then **D(n,m)=D(n-1,m-1)+1**
  - matches a_n and b_m then **D(n,m)=D(n-1,m-1)**

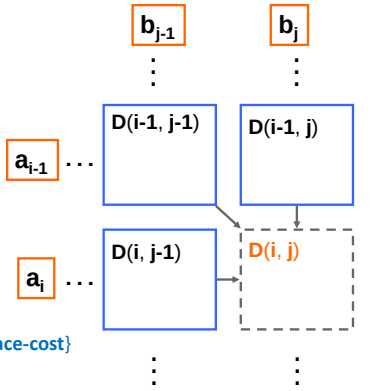
recursive algorithm D(n,m)

```
if n=0 then
    return (m)
else if m=0 then
    return(n)
else
    if a_n=b_m then
        replace-cost ← 0
    else
        replace-cost ← 1
    endif
    return(min{ D(n-1, m) + 1,
                D(n, m-1) + 1,
                D(n-1, m-1) + replace-cost } )
```

cost of substitution of a_n by b_m (if used)

dynamic programming

```
for j = 0 to m;  D(0,j) ← j; endfor
for i = 1 to n;  D(i,0) ← i; endfor
for i = 1 to n
    for j = 1 to m
        if a_i=b_j then
            replace-cost ← 0
        else
            replace-cost ← 1
        endif
        D(i,j) ← min { D(i-1, j) + 1,
                       D(i, j-1) + 1,
                       D(i-1, j-1) + replace-cost}
    endfor
endfor
```



example run with AGACATTG and GAGTTA

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
0									
G	1								
A	2								
G	3								
T	4								
T	5								
A	6								

example run with AGACATTG and GAGTTA

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2								
G	3								
T	4								
T	5								
A	6								

example run with AGACATTG and GAGTTA

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2	1	2	1					
G	3								
T	4								
T	5								
A	6								

example run with AGACATTG and GAGTTA

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2	1	2	1	2	3	4	5	6
G	3	2	1	2	2	3	4	5	5
T	4								
T	5								
A	6								

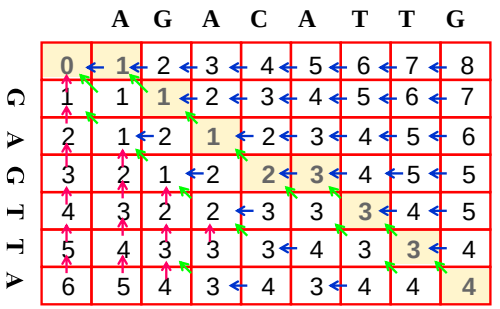
example run with AGACATTG and GAGTTA

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2	1	2	1	2	3	4	5	6
G	3	2	1	2	2	3	4	5	5
T	4	3	2	2	3	3	3	4	5
T	5	4	3	3	3	4	3	3	4
A	6	5	4	3	4	3	4	4	4

example run with AGACATTG and GAGTTA

		A	G	A	C	A	T	T	G
	0	← 1	← 2	← 3	← 4	← 5	← 6	← 7	← 8
G	↑ 1	↖ 1	← 1	← 2	← 3	← 4	← 5	← 6	← 7
A	↑ 2	↑ 1	↖ 2	← 1	← 2	← 3	← 4	← 5	← 6
G	↑ 3	↑ 2	↑ 1	↖ 2	← 2	← 3	← 4	← 5	← 5
T	↑ 4	↑ 3	↑ 2	↑ 2	← 3	↖ 3	← 3	← 4	← 5
T	↑ 5	↑ 4	↑ 3	↑ 3	↑ 3	← 4	↖ 3	← 3	← 4
A	↑ 6	↑ 5	↑ 4	↖ 3	← 4	↑ 3	← 4	↖ 4	← 4

example run with AGACATTG and GAGTTA



rreading off the operations

- Follow the sequence and use each color of arrow to tell you what operation was performed.
- From the operations can derive an optimal alignment

A G A C A T T G
_ G A G _ T T A

saving space

- To compute the distance values we only need the last two rows (or columns)
  - O(min(m,n)) space
- To compute the alignment/sequence of operations
  - seem to need to store all O(mn) pointers/arrow colors
- Nifty divide and conquer variant that allows one to do this in O(min(m,n)) space and retain O(mn) time
  - In practice the algorithm is usually run on smaller chunks of a large string, e.g. m and n are lengths of genes so a few thousand characters

    Researchers want all alignments that are close to optimal

    Basic algorithm is run since the whole table of pointers (2 bits each) will fit in RAM
  - Ideas are neat, though

saving space

- Alignment corresponds to a path through the table from lower right to upper left
  - Must pass through the middle column
- Recursively compute the entries for the middle column from the left
  - If we knew the cost of completing each then we could figure out where the path crossed
  - Problem

    There are n possible strings to start from.
  - Solution

    Recursively calculate the right half costs for each entry in this column using alignments starting at the other ends of the two input strings!
  - Can reuse the storage on the left when solving the right hand problem

shortest paths with negative edge weights

- Dijsktra's algorithm failed with negative-cost edges
  - What can we do in this case?
  - Negative-cost cycles could result in shortest paths with length $-\infty$
- Suppose no negative-cost cycles in G
  - Shortest path from **s** to **t** has at most **n-1** edges

    If not, there would be a repeated vertex which would create a cycle that could be removed since cycle can't have negative cost

shortest paths with negative edge weights

- We want to grow paths from **s** to **t** based on the # of edges in the path
- Let **Cost(s,t,i)**=cost of minimum-length path from **s** to **t** using up to **i** hops.
  - $Cost(v,t,0) = \begin{cases} 0 \text{ if } v=t \\ \infty \text{ otherwise} \end{cases}$
  - $Cost(v,t,i) =$

shortest paths with negative edge weights

- We want to grow paths from **s** to **t** based on the # of edges in the path
- Let **Cost(s,t,i)**=cost of minimum-length path from **s** to **t** using up to **i** hops.
  - $Cost(v,t,0) = \begin{cases} 0 \text{ if } v=t \\ \infty \text{ otherwise} \end{cases}$
  - $Cost(v,t,i) = \min\{ Cost(v,t,i-1), \min_{(v,w)\in E}(c_{vw}+Cost(w,t,i-1)) \}$

Bellman-Ford

- Observe that the recursion for **Cost(s,t,i)** doesn't change **t**
  - Only store an entry for each **v** and **i**

    Termed OPT(**v**,**i**) in the text
- Also observe that to compute **OPT(*,i)** we only need **OPT(*,i-1)**
  - Can store a current and previous copy in **O(n)** space.

Bellman-Ford

```
ShortestPath(G,s,t)
    for all v∈V
        OPT[v]←∞
    OPT[t]←0
    for i=1 to n-1 do
        for all v∈V do
            OPT'[v]←min_{(v,w)∈E} (c_vw+OPT[w])
        for all v∈V do
            OPT[v]←min(OPT'[v],OPT[v])

    return OPT[s]
```

O(mn) time

negative cycles

- Claim: There is a negative-cost cycle that can reach t iff for some vertex v∈V, Cost(v,t,n)<Cost(v,t,n-1)
- Proof:

negative cycles

- Claim: There is a negative-cost cycle that can reach t iff for some vertex v∈V, Cost(v,t,n)<Cost(v,t,n-1)
- Proof:
  - We already know that if there aren't any then we only need paths of length up to n-1
  - For the other direction

    The recurrence computes Cost(v,t,i) correctly for **any** number of hops i

    The recurrence reaches a fixed point if for every v∈V, Cost(v,t,i)=Cost(v,t,i-1)

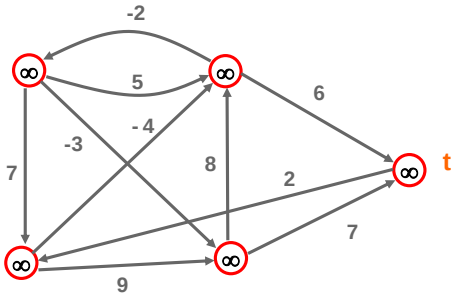
    A negative-cost cycle means that eventually some Cost(v,t,i) gets smaller than any given bound

    Can't have a negative cost cycle if for every v∈V, Cost(v,t,n)=Cost(v,t,n-1)

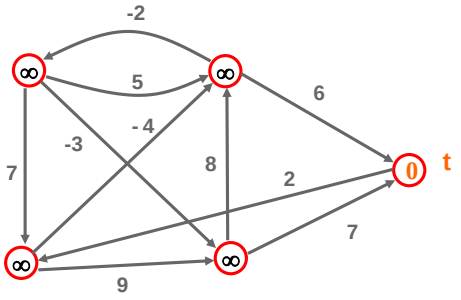
last details

- Can run algorithm and stop early if the OPT and OPT' arrays are ever equal
  - Even better, one can update only neighbors v of vertices w with OPT'[w]≠OPT[w]
- Can store a successor pointer when we compute OPT
  - Homework assignment
- By running for step n we can find some vertex v on a negative cycle and use the successor pointers to find the cycle

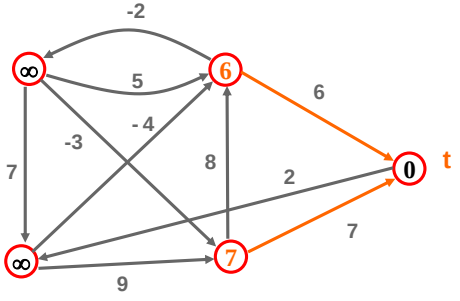
Bellman-Ford



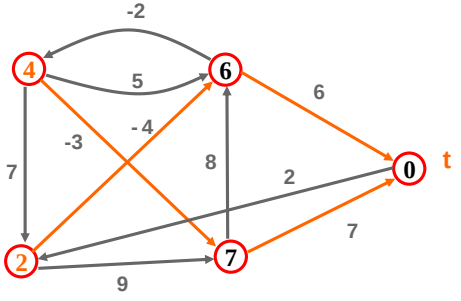
Bellman-Ford



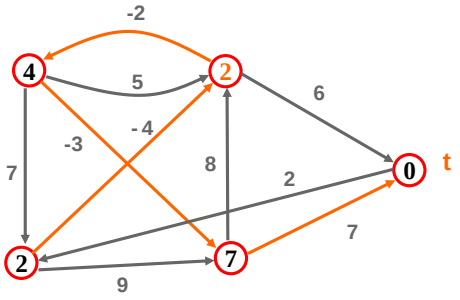
Bellman-Ford



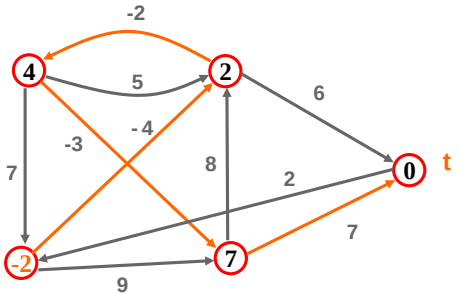
Bellman-Ford



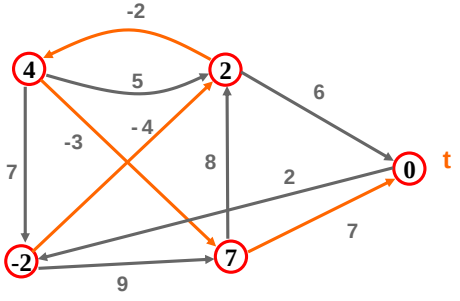
Bellman-Ford



Bellman-Ford



Bellman-Ford



Bellman-Ford with a DAG

Edges only go from lower to higher-numbered vertices
- Update distances in reverse order of topological sort
- Only one pass through vertices required
- O(**n**+**m**) time