

Hard problems

Generalizing pairwise alignment

- Alignment of 2 sequences is represented as a 2-row matrix
- In a similar way, we represent alignment of 3 sequences as a 3-row matrix

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

- Score: more conserved columns, better alignment

Sum of pairs (SP) score

- Consider pairwise alignment of sequences

$$a_i \text{ and } a_j$$

imposed by a multiple alignment of k sequences

- Denote the score of this suboptimal (not necessarily optimal) pairwise alignment as

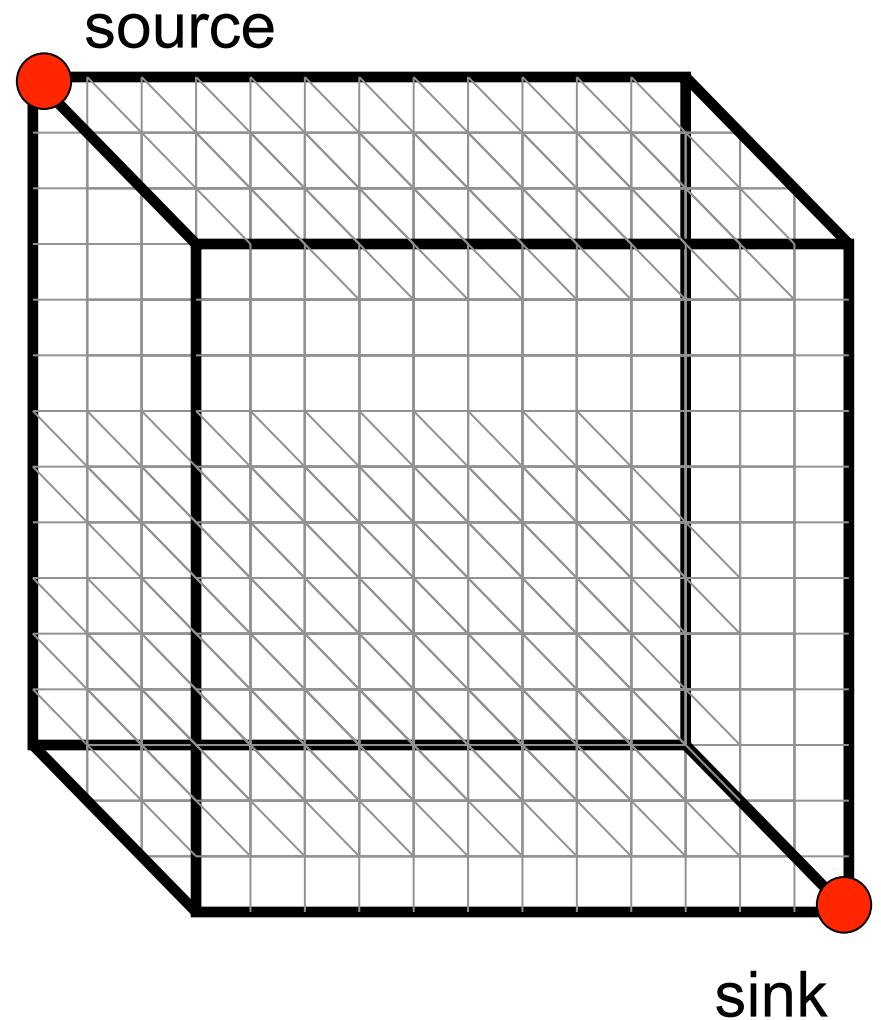
$$s^*(a_i, a_j)$$

- Sum up the pairwise scores for a multiple alignment:

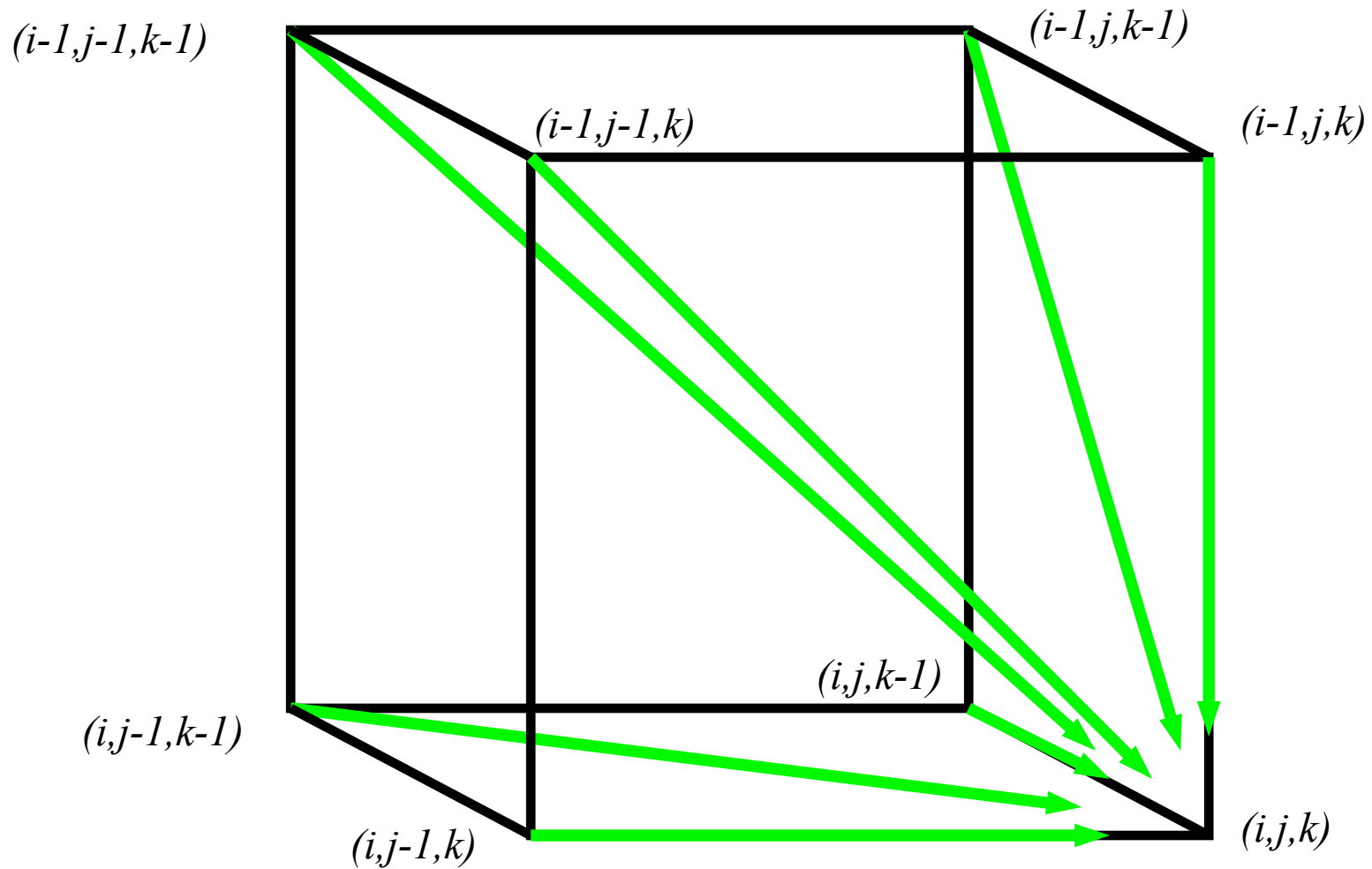
$$s(a_1, \dots, a_k) = \sum_{i,j} s^*(a_i, a_j)$$

Aligning three sequences

- Same strategy as aligning two sequences
- Use a 3-D alignment graph, with each axis representing a sequence to align



Architecture of 3-D Alignment Cell



Multiple alignment: dynamic programming

- $s_{i,j,k} = \max \left\{ \begin{array}{ll} s_{i-1,j-1,k-1} + \delta(v_i, w_j, u_k) & \text{cube diagonal:} \\ s_{i-1,j-1,k} + \delta(v_i, w_j, -) & \text{no indels} \\ s_{i-1,j,k-1} + \delta(v_i, -, u_k) & \text{face diagonal:} \\ s_{i,j-1,k-1} + \delta(-, w_j, u_k) & \text{one indel} \\ s_{i-1,j,k} + \delta(v_i, -, -) & \text{edge diagonal:} \\ s_{i,j-1,k} + \delta(-, w_j, -) & \text{two indels} \\ s_{i,j,k-1} + \delta(-, -, u_k) & \end{array} \right.$
- $\delta(x, y, z)$ is an entry in the 3-D scoring matrix

Multiple alignment: complexity

- For 3 sequences of length n , the run time is $7n^3$; $O(n^3)$
- For k sequences, build a k -dimensional Manhattan, with run time $(2^k-1)(n^k)$; $O(2^k n^k)$
- Conclusion: dynamic programming approach for alignment between two sequences is easily extended to k sequences but it is exponential to k .
- Classification of hard problems.

Decision problems vs. optimization problems

- *Decision problems*: determine whether a given statement is true or false
- *Optimization problems*: find the solution with the best possible score according to some scoring scheme
 - Minimization problems
 - Maximization problems

Hamiltonian cycles

Given a directed graph, we want to decide whether or not there is a *Hamiltonian cycle* in this graph. This is a decision problem.

Hamiltonian cycle: a circular path that visits each node once and exactly once.

TSP - Traveling Salesman Problem

Given a complete graph and an assignment of weights to the edges, find a Hamiltonian cycle of minimum weight.

This is the *optimization version* of the problem. In the *decision version*, we are given a weighted complete graph and a real number c , and we want to know whether or not there exists a Hamiltonian cycle whose combined weight of edges does not exceed c .

Minimum spanning tree problem

Given a connected graph and an assignment of weights to the edges, find a spanning tree of minimum weight.

Minimum spanning tree: a subgraph which is a tree and connects all the vertices together.

This is the *optimization version* of the problem. In the *decision version*, we are given a weighted connected graph and a real number c , and we want to know whether or not there exists a spanning tree whose combined weight of edges does not exceed c .

Important observation

- Each optimization problem has a corresponding decision problem.

The class P

A decision problem D is *solvable in polynomial time* (or *in the class P*), if there exists an algorithm A such that

- A takes instances of D as inputs.
- A always outputs the correct answer “Yes” or “No”.
- There exists a polynomial p such that the execution of A on inputs of size n always terminates in $p(n)$ or fewer steps.

The class P

EXAMPLE: The Minimum Spanning Tree Problem is in the class P . \rightarrow The greedy algorithm is a correct algorithm.

The class P is often considered as synonymous with the class of computationally feasible problems, although in practice some polynomial algorithms are unrealistic.

Witnesses for decision problems

Witness: the instance to a decision problem is “yes”, e.g., in the Hamiltonian cycle problem, any permutation v_1, v_2, \dots, v_n of the vertices of the input graph is a *potential witness*. This potential witness is a *true witness* if v_1 is adjacent to v_2 , \dots and v_n is adjacent to v_1 .

The class NP

A decision problem is *non-deterministically polynomial-time solvable* or *in the class NP* if there exists an algorithm A such that

- A takes as inputs potential witnesses for “yes” answers to problem D .
- A correctly distinguishes true witnesses from false witnesses.
- There exists a polynomial p such that for each potential witnesses of each instance of size n of D , the execution of the algorithm A takes at most $p(n)$ steps.

The class NP

Note that if a problem is in the class NP , then we are able to verify “yes”-answers in polynomial time, if we are provided a true witness.

The $P=NP$ Problem

Are the classes P and NP identical? This is an open problem. It is not hard to show that every problem in P is also in NP , but it is unclear whether every problem in NP is also in P .

The $P=NP$ Problem

Thousands of computer scientists have been unsuccessful for decades to design polynomial-time algorithms for some problems in the class NP .

This constitutes overwhelming *empirical* evidence that the classes P and NP are indeed distinct, but no formal mathematical proof of this fact is known.

Polynomial-time reducibility

Let E and D be two decision problems. We say that D is *polynomial-time reducible to E* if there exists an algorithm A such that

- A takes instances of D as inputs and always outputs the correct answer “Yes” or “No” for each instance of D .
- A uses as a subroutine a hypothetical algorithm B for solving E .
- There exists a polynomial p such that for every instance of D of size n the algorithm A terminates in at most $p(n)$ steps *if each call of the subroutine B is counted as only m steps, where m is the size of the actual input of B .*

An example of polynomial-time reducibility

Theorem: The Hamiltonian cycle problem is polynomial-time reducible to the decision version of TSP.

Proof: Given an instance G with vertices v_1, \dots, v_n of the Hamiltonian cycle problem, let H be the weighted complete graph on v_1, \dots, v_n such that the weight of an edge $\{v_i, v_j\}$ in H is 1 if $\{v_i, v_j\}$ is an edge in G , and is 2 otherwise. Now the correct answer for the instance G of the Hamiltonian cycle problem can be obtained by running an algorithm on the instance $(H, n+1)$ of the TSP.

NP-complete problems

A decision problem E is *NP-complete* if *every* problem in the class NP is polynomial-time reducible to E . The Hamiltonian cycle problem, the decision versions of the TSP and literally hundreds of other problems are known to be NP-complete.

To show a decision problem E is NP-complete, you need to show E is in class NP , and one problem known to be NP-complete is polynomial-time reducible to E .

NP-hard problems

Optimization problems whose decision versions are NP-complete are called *NP-hard*.

Theorem: If there exists a polynomial-time algorithm for finding the optimum in any *NP*-hard problem, then $P = NP$.

NP-hard problems

Proof: Let E be an NP -hard optimization (let us say minimization) problem, and let A be a polynomial-time algorithm for solving it. Now an instance J of the corresponding decision problem D is of the form (I, c) , where I is an instance of E , and c is a number. Then the answer to D for instance J can be obtained by running A on I and checking whether the cost of the optimal solution exceeds c . Thus there exists a polynomial-time algorithm for D , and NP -completeness of the latter implies $P = NP$.

Duality

- For many optimization problems, there is a ***dual*** problem associated with the original (***primal***) problem.
- The relationship between the primal and dual problems are useful in a variety of ways, e.g.
 - helps to find efficient algorithms for solving the primal problem;
 - provides arguments for proving the optimality (or close-to-optimality) of a primal solution.

Consequences for bioinformatics

In view of the overwhelming empirical evidence against the equality $P = NP$ it seems that no NP -hard optimization problem is solvable by an algorithm that is *guaranteed* to:

- run in polynomial time *and*
- *always* produce a optimal solution.

Unfortunately, many, perhaps most, of the important optimization problems in bioinformatics are NP -hard. To make matters worse, the instances of interest in bioinformatics are typically of large size.

What can we do about these problems?

Hard problems in bioinformatics

- Multiple sequence alignment problem
- Protein threading / design problem
- Map / sequence assembly problem
- Many others...

Alternative performance measures?

So far we have been talking about algorithms that

- run in polynomial time on *all* instances
- always find the solution with the *best* score/cost.

Worst case vs. average performance

For many practical purposes, it may be sufficient to have an algorithm whose *average* running time for instances of size n is bounded by a polynomial. Such an algorithm may still be unacceptably slow for some particularly bad instances, but such bad instances will necessarily be very rare and may be of little practical relevance.

Approximation algorithms

While optimal solutions to optimization problems are clearly best, “reasonably” good solutions are also of value.

Let us say that an algorithm for a minimization problem D has a *performance guarantee of $1 + \varepsilon$* if for each instance I of the problem it finds a solution whose cost is at most $(1 + \varepsilon)$ times the cost of the optimal solution for instance I . While D may be *NP*-hard, it may still be possible to find, for some $\varepsilon > 0$, polynomial-time algorithms for D with performance guarantee $1 + \varepsilon$. Such algorithms are called *approximation algorithms*.

Polynomial-time approximation schemes

- A minimization problem D has a *polynomial-time approximation scheme (PTAS)* if for every $\varepsilon > 0$ there exists a polynomial-time algorithm for D with performance guarantee $1 + \varepsilon$.
- *Even if D may be NP-hard, it may still have a PTAS.*

Heuristic algorithms

- reasonably fast on the average instance
- most of the time find solutions within $(1 + \varepsilon)$ of optimum for reasonably small ε
- It is not always easy or possible to mathematically analyze the performance of a heuristic algorithm