Summary so far

- We have seen how to establish whether "my problem" is tractable or not:
  - If I can think of a polynomial algorithm then it is tractable.
    - If "my algorithm" is fast enough, then I am happy.
    - It <u>may be still</u> that its complexity is too high with "my input size" and for the short time allowed for "my application". In this case I am in trouble.
  - If I can reduce it from an untractable problem then it is untractable.
    - Its complexity is <u>almost certainly</u> too high with "my input size" and for the short time allowed for "my application". In this case I am in trouble.
- What to do when "I am in trouble"?

Tractability in Bioinformatics/1

- Assume an algorithm A in bioinformatics with running time $O(n^2)$ for which the input is the whole data set in Genbank.
- Let us make the conservative estimate that the size of this input doubles every nine months.
- Moore's Law says that computer speed doubles every 18 months.
- Algorithm A today takes one hour to run on the fastest available computer, and it will take eight hours to run on the fastest available computer 18 months from now.

Tractability in Bioinformatics/2

On the other hand:

- There are problems in which the input size is small: examples in phylogeny, genome rearrangement. Here time complexity is not a issue in practice.
- There are problems whose solutions deserve long running time: examples in fragment assembly for genome sequencing.
- Average case behavior may be far from worst case.

Approximation algorithms

- I want “my solution” to “my optimization problem” to:
  - Find an optimal solution.
  - In polynomial time.
  - For any instance.
- Drop condition 2: Exponential algorithm; hardly feasible.
- Drop condition 3 and/or 1: Heuristics.
- Drop condition 1: Approximations.

Approximations and Heuristics

- Approximation algorithm: I cannot find the optimal solution in short time, so I find a “good” solution in short time.
- Heuristics: I do “reasonable” assumptions that “most probably”, or “almost always”, lead me to a “decent” – “maybe” optimal – solution.
- Actually approximations are a special case of heuristics.

Approximation algorithms

- Approximation algorithms are thought for hard optimization problems.
- Optimization problem (I,Sol,f,{max/min}):
  - Assume that finding the solution that {max/min}imizes f is too time consuming.
  - Maybe I can find in reasonable time a solution in Sol that is not too far from the optimal:
    - A constant multiplicative factor.
    - A delta away from it where delta appears in the complexity.

Approximation algorithms

- Optimization problem (I,Sol,f,{max/min}). what is a solution in Sol ? Examples:
  - TSP: any path visiting all cities (not necessarily of minimum length).
  - COMPATIBILITY: a set of k’≤ k compatible characters.
  - MINIMUM VERTEX COVER: a vertex cover (not necessarily of minimum size).
  - An alignment (not necessarily that of minimum cost).

Why Approximation algorithms?

- Why studying approximation algorithms:
  - To design solutions to NP-hard problems.
  - They are heuristics with a mathematically rigorous model behind.
  - They open a new world of complexity classes that
    - Show how hard problems are, and
    - It can help to solve open theoretical problems.

r-Approximation algorithms

- An algorithm is an r-approximation for an optimization problem P if:
  - It runs in polynomial time.
  - Il always produces a solution ‘sol’ in Sol which is within a factor r of the value ‘opt’ of the optimal solution.
    - In case “max f”, I have r<1 and I produce a solution sol such that $r\ opt \leq sol \leq opt$
    - In case “min f”, I have r>1 and I produce a solution sol such that $opt \leq sol \leq r\ opt$.

Approximation of knapsack

- MAXIMUM KNAPSACK:
  - INPUT: n items with profits p1,...,pn and sizes a1,...,an, and integer b (capacity).
  - OUTPUT: a subset of the items having total size not greater than the capacity, and maximum total profit.
- Algorithm A:
  - Sort the items in non-decreasing order of pi/ai.
  - Take them in that order as long as they fit in the knapsack.

2-approximation of knapsack

- Algorithm A:
  - Sort the items in non-decreasing order of pi/ai.
  - Take them in that order as long as they fit in the knapsack.
  - At the end name ptot the total sum of profits pi of selected items.
  - Output max{ptot,pmax} where pmax is the highest profit.
- It is a *greedy* algorithm.
- It is a 2-approximation: prove it as an exercise (hint: check when ptot is bad).

Class APX

- APX is the class of all problems in NPO for which there exist polynomial time r-approximation algorithms with $r \geq 1$.

- MAX-KNAPSACK is in APX, but also MAX-SAT, MIN-VERTEX COVER, and also some problems in bioinformatics...

Reversal Distance is in APX

- Two genomic sequences G1 and G2 given as two permutations of the set of labels {1,..,n}.
- The Reversal Distance between G1 and G2 is the minum number of reversals that transform G1 in G2.
- Ex. 1254$\underline{763}$ → 12$\underline{543}$67 → 1234567; RD=2.
- Computing the reversal distance is NP-hard.
- Considered relevant in genome rearrangments, also knows as inversion distance.
- It is a metric.
- Tractability of signed version.

Breakpoint Distance

- G1=Π(G2) and G2=1 2 ... n.
- The Breakpoint Distance between G1 and G2 is the number of i's in {0,..,n+1} such that |G1[i]-G1[i+1]|≠1, assuming G1[0]=0 and G1[n+1]=n+1.
- The breakpoint distance can be computed in linear time.
- The breakpoint distance is a 2-approximation of the reversal distance.

Syntenic Distance is in APX

- A genome is seen as m sets (chromosomes) of elements over a set of n objects (genes).
- Ex. G1={1,2,3},{2,5,6},{4} and G2={1,2,5},{3,6},{4}.
- The Syntenic Distance between G1 and G2 is the minimum number of translocations, fusions and fissions that transform G1 into G2.
- It is a metric.
- Canonical version: m sets to be transformed into {1}{2}...{n}.
- The trivial m-1 fusions + n-1 fissions is a 2-approximations.
- Practically uninteresting compared to the worst case exponential exact branch and bound solution.

On approximability

- There are problems that can be approximated more or better than others.
- And some that cannot be approximated at all...
- MIN-TSP, MAX-CLIQUE are in NPO but provably not in APX (unless P=NP)...

Not approximability in bioinformatics

- The multiple sequence alignment problem is NP-hard with respect to the number k of sequences *k*.
- Does the problem become tractable under reasonable biological assumptions, such as using a different (biologically significant) scoring schemes, limiting the number of gaps that can be inserted?

Computational complexity of multiple sequence alignment

- [recent result] For every scoring scheme “used by biologists”, the multiple sequence alignment problem is NP-hard. This remains true even if the number and size of gaps that can be inserted into each sequence is restricted in “the most severe” way possible.

- The multiple alignment problem cannot be approximated, even if the number and size of gaps that can be inserted into each sequence is most severely restricted.

A few words about the proof

- These negative results were proved by reducing the MAX-CUT problem for graphs to the multiple sequence alignment problem.

- The idea: given a simple graph G, a multiple sequence alignment problem is constructed in such a way that from a (nearly) optimal solution of the sequence alignment problem a cut in the graph G of (nearly) maximal size can be reconstructed in polynomial time.

The practice of multiple alignment

- The most frequently used multiple sequence alignment algorithm used in practice is CLUSTAL.
- This is a heuristic algorithm for which no performance guarantee is known.
- There are more accurate heuristics solutions that are slower.

Approximation and heuristics strategies

- Greedy algorithms (knapsack).
- Dynamic Programming (alignments).
- (Integer) Linear Programming (SNPs).
- Computing lower and upper bounds (distances).
- ... Intuition!

Heuristics: the idea

- Il non determinismo consente di “*controllare*” un numero esponenziale di possibilità in tempo polinomiale.

- La simulazione di questo potente meccanismo in tempo deterministico polinomiale significherebbe che:
  - tutti i problemi verificabili efficientemente possiedono insospettate proprietà che si prestano ad essere sfruttate anche per la loro risoluzione;
  - per un'ampia classe di problemi, la ricerca esaustiva può essere sostituita da procedure efficienti. Non esaustive, ma quasi...

Branch and bound

- Branch and bound for looking for a k-clique in a graph (problem useful for some motifs finding strategies).

- In the graph remove nodes that have less than k adjacent nodes.

- It can result very fast in practice.

SOLUZIONI

What follows are solutions to exercises and ideas of exercises.

Proof of 2-approx of knapsack/1

- A runs in O(*n log n*) time.
- Let j be the first item not selected by A; ptot is the sum of the pi's of the first j-1 elements (sorted by pi/ai) that has atot < b occupancy.
- We have that opt < ptot + pj because:
  - Exchanging any subset of selected items with any of the unselected s.t. occupancy ≤ atot, does not incresae profit → opt < ptot + max possible profit filling the free (b-atot) space.
  - atot+aj > b → opt ≤ ptot+pj/aj(b -atot) < ptot + pj.

Proof of 2-approx of knapsack/2

Hence opt < ptot + pj

- If pj ≤ ptot then opt < 2 ptot ≤ 2 *max*{ptot,pmax}
- If pj > ptot then pmax ≥ pj > ptot and then

opt < ptot+pj ≤ ptot+pmax < 2pmax = 2max{ptot,pmax}

In both cases *max*{ptot,pmax} is a 2-approximation of opt because opt/2 ≤ max{ptot,pmax} ≤ opt