UCSF

BMI-203: Biocomputing Algorithms

Introduction

Instructor: Ajay N. Jain, PhD

Email: ajain@cc.ucsf.edu

Copyright © 2004, Ajay N. Jain

All Rights Reserved

Important stuff to know

Lectures: Tuesdays 3-5pm in GH-S201 (1 exception: April 27th in GH-261)

Lab: Thursdays 3-5pm in GH-S201 (1 exception: April 1st in GH-261)

Course Web Site: www.cgl.ucsf.edu/Outreach/bmi203

- Schedule
- Lectures
- Homeworks
- Reference material

Textbooks

Introduction to Algorithms: Cormen, Leiserson, and Rivest

Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids: Durbin et al.

Numerical Recipes in C (www.nr.com)

You should probably buy the first book. You will be looking at it 20 years from now if you continue research in bioinformatics and algorithms.

($80 new at Amazon, $30 used)

Biocomputing Algorithms

Computational issues and methods in bioinformatics and computational biology

- Analytical thinking
- Problem decomposition
- Algorithm understanding, design, and implementation

This course is **not** about

- Expert use of existing tools
- Learning how to program (if you can't program in C, Python, or a similar language, you should take the course after you have become somewhat proficient)

Programming Languages

We are agnostic about programming languages

However, for some assignments, it will be much easier to use Python, since you will be provided some code

Languages that are OK: Python, C, C++, Fortran, Java

Perl is probably not OK, but you can try :-)

Lab sessions

TA: Tuan Pham (anhtuan@cc.ucsf.edu)

These sessions are for you to interact if you are having difficulties with the lectures, homeworks, or final project

Homework is 75% of your grade, with 25% for the final project

We expect you to have access to your own computer

You can download Python: www.python.org

You can download Cygwin (gnu c): www.cygwin.com

You can run gcc on the Mac (OS X ships with developer tools, but you have to install them)

For people with special needs, please contact Ajay

UCSF

BMI-203: Biocomputing Algorithms

Lecture 1: Complexity and Sorting

Instructor: Ajay N. Jain, PhD

Email: ajain@cc.ucsf.edu

Copyright © 2004, Ajay N. Jain

All Rights Reserved

UCSF

Outline

Complexity Theory (see Cormen, Chapters 1 and 2)

- Every computer algorithm has execution time and space and/or bandwidth requirements dependent on the size or complexity of its input
- Design of useful algorithms is largely dominated by complexity considerations
- We will cover very basic notational conventions (no proofs)

Sorting (see Cormen, Part II, particularly Chapter 8)

- Sorting is the classic algorithms problem space in which complexity issues are taught
- Bubble sort
- Quicksort

Homework: two sorting algorithms + analysis of run-time perf.

Reference: Introduction to Algorithms, Second Edition by Thomas H. Cormen (Editor), Charles E. Leiserson, Ronald L. Rivest

Computational Complexity Theory

What is an algorithm?

- Given a precise problem description
  - Sort a list of N real numbers from lowest to highest
- An algorithm is a precise method for accomplishing the result

O notation: informally

- Want to capture how fast or how much space an algorithm requires
- We ignore constant factors (even if very large)
- O(N) indicates that an algorithm is linear in the size of its input
  - Example: sum of N numbers is O(N)

Notes on Notation

We will define the complexity of algorithms based on describing a function that provides a boundary on time or space

Formally, we will describe complexity in terms of the membership of the function in a larger set of functions

Notation

- N = {0,1,2,3, ... }
- N+ = {1,2,3,4, ... }
- R = Set of Reals
- R+ = Set of Positive Reals
- R* = R+ U {0}

Comparing $f(n)$ and $g(n)$

Let **f** be a function from **N** to **R**.

O(f) (Big **O** of **f**) is the set of all functions **g** from **N** to **R** such that:

1. There exists a real number **c**>0
2. AND there exists an $\mathbf{n_0}$ in **N**

Such that: $\mathbf{g(n) \leq cf(n)}$ whenever $\mathbf{n \geq n_0}$

Notation and Pronunciation

Proper Notation: $\mathbf{g} \in \mathbf{O(f)}$

“g is oh of f”

Also Seen: $\mathbf{g} = \mathbf{O(f)}$

Big Omega

Let **f** be a function from **N** to **R**.

Ω(f) (Big Ω of **f**) is the set of all functions **g** from **N** to **R** such that:

1. There exists a real number **c**>0
2. AND there exists an $\mathbf{n_0}$ in **N**

Such that: $\mathbf{g(n)} \geq \mathbf{cf(n)}$ whenever $\mathbf{n} \geq \mathbf{n_0}$

Big Theta

$\Theta(f) = O(f) \cap \Omega(f)$

“g is of Order f”

$g \in \Theta(f)$

“g is Order f”

English Interpretations

$O(f)$ - Functions that grow no faster than f

$\Omega(f)$ - Functions that grow no slower than f

$\Theta(f)$ - Functions that grow at the same rate as f

Properties

Constant factors may be ignored

- For all k > 0, kf is O(f)

Higher powers of n grow faster than lower powers

The growth rate of a sum of terms is the growth rate of its fastest term

- So, if you have a linear element of an algorithm and a element that is n^2, then the algorithm will be $O(n^2)$

If f grows faster than g which grows faster than h, then f grows faster than h

Exponential functions grow faster than powers

Logarithms grow more slowly than powers (and all logarithms grow at the same rate)

UCSF

When complexity gets bad

Polynomial time algorithms

- All algorithms such that there exists an integer d where the algorithms is $O(n^d)$

Intractable algorithms

- The class of problems that cannot be solved in polynomial time

Particularly interesting class of intractable problems:

- NP-complete

When this happens, we often care about approximate solutions

- Traveling Salesman Problem: Given N cities, find the route that goes to each city exactly once that minimizes the total distance traveled
- N! ways of ordering N cities
- NP-complete: if you can solve this problem in polynomial time, you can solve all NP-complete problems in polynomial time

ε-approximate solutions abound

- You can find a city ordering such that for any ε, your solution is within ε of the optimal solution
- You can do this in polynomial time

Analyzing an algorithm

Simple sequence of statements, executed once: O(1)

Simple loops

- For (i=0;i<n;++i) { s; }
- O(n)

Nested loops

- For (i=0;i<n;++i) { s; }
  - For (j = 0; j < n; ++j) { s; }
- $O(n^2)$

Multiplicative index jumps can yield O(log(n))

- h = 1; while (h < n) { s; h = 2*h; }

Sorting algorithms

Input: a sequence of n numbers $(a_1, a_2, \ldots, a_n)$

Output: a permutation $(a_1, a_2, \ldots, a_n)$

Such that $a_1 \leq a_2 \leq a_3 \ldots \leq a_n$

Example: Insertion Sort (order a hand of cards)

- Create an empty array of size n
- Find the smallest value in input array
  - Put it in the new array in the last unfilled position
  - Mark the input array value as done
- Repeat until the new array has n values

UCSF

Bubble sort: In English

Go through your list of n numbers, checking for misordered adjacent pairs

Swap any adjacent pairs where the second value is smaller than the first

Repeat this procedure a total of n times

Your final list will be sorted low to high

Bubble sort: In C

```
/* Bubble sort for integers */
void bubble( int a[], int n )
/* Pre-condition: a contains n items to be sorted */
{
   int i, j, t;
   /* Make n passes through the array */
   for(i=0;i < n; i++) {
     /* From the first element to
        the end of the unsorted section */
     for(j=1;j<(n-i);j++) {
     /* If adjacent items are out of order, swap them */
     if( a[j-1]>a[j] ) {
       t = a[j];
       a[j] = a[j-1];
       a[j-1] = t;
      }
    }
  }
}
```

One conditional

Three assignments

Bubble sort complexity: Worst case

UCSF

We make (n-1) passes through the data

- When i = (n-1), (n - i) is (n-(n-1)) = 1
- So, on the last outer loop pass, we don't do the inner loop

How many operations do we do in each pass (at worst)?

- On the last pass, we do one conditional and three assignments
- On the second to last pass, we do 2 and 6
- Etc…

So

- (1*(1+2+ … + (n-1))) compares
- (3*(1+2+ … + (n-1))) assignments
- Recall that sum (1…k) is k(k+1)/2
- We have n(n-1)/2 compares and 3n(n-1)/2 assignments

Since we don't care about constant factors and higher-order polynomials dominate, BubbleSort is $O(n^2)$

QuickSort: In English

Quicksort is a divide and conquer algorithm

It was invented by C. A. R. Hoare

Divide: The array A[p…r] is partitioned into two nonempty subarrays A[p…q] and A[q+1…r] (q is pivot element)

Conquer: The two subarrays A[p…q-1] and A[q+1…r] are themselves subjected to Quicksort (by recurrence)

Combine: The results of the recursion don't need combining, since the subarrays are sorted in place

The final A[p…r] is now sorted

Quicksort: Complexity

The average case for Quicksort is O(n log(n)) with smallish constant factors for good implementations

- The partitioning algorithm requires O(n) time to rearrange the array (it examines every element once)
- The partitioning is done around a **pivot**, which is chosen with no knowledge (in the simplest case); elements are partitioned to be less than or greater than the pivot
- We expect that randomly chosen pivots will tend to partition an array into roughly two halves
- So, we end up doing O(log(n)) partitions, and O(n log(n)) overall

In practice, this is one of the fastest sorting methods known

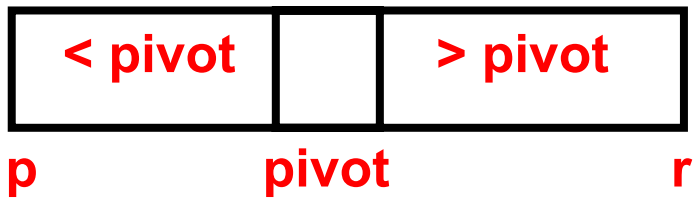
However, its worst case behavior is **$O(n^2)$**: poor luck with the pivot choices can lead to n partitions!

UCSF

Quicksort: In C

```
/* We would call quicksort(a, 0, n-1) */

quicksort( void *a, int p, int r )
  {
  int pivot;
  /* Termination condition! */
  if ( r > p )
    {
    pivot = partition( a, p, r );
    quicksort( a, p, pivot-1 );
    quicksort( a, pivot+1, r );
    }
  }
```



The partition function does all of the work

It selects the pivot element

It partitions the subarray

The quicksort function just does bookkeeping

Note: in C, arrays are passed by reference, so the operations are occurring on the **same** array

Quicksort partitioning

There is a straightforward way to partition

- Pick any element as the pivot (say the first)
- Create a new array of the same size as input
- For each element in the old array, put it at the beginning if it is less than the pivot element
- Else, put it at the end
- [Keep track of the “beginning” and “end”, which move]
- Copy the new array back into the original one
- Return the value of the pivot index

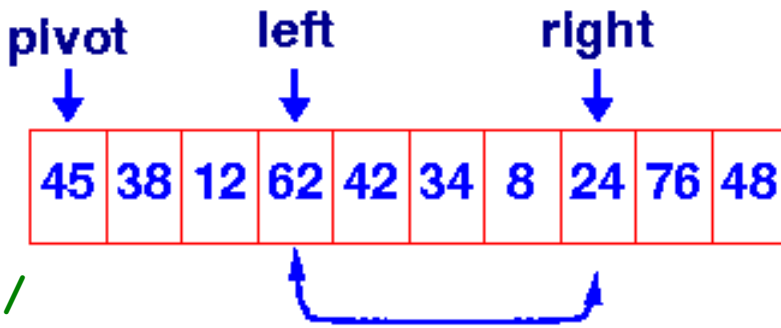
Problem: requires additional space (allocate and free) and an additional n assignments in the end

UCSF

Quicksort: Partition in place

```
int partition( void *a, int p, int r )
  {
  int left, right;
  void *pivot_item;

  pivot_item = a[p];
  pivot = left = p;
  right = r;
  while ( left < right ) {
    /* Move left while item < pivot */
    while( a[left] <= pivot_item ) left++;
    /* Move right while item > pivot */
    while( a[right] > pivot_item ) right--;
    if ( left < right ) SWAP(a,left,right);
    }
  /* right is final position for the pivot */
  a[p] = a[right];
  a[right] = pivot_item;
  return right;
  }
```



Note: this this code does not check that left does not exceed the array bound.

Homework 1: due 4-6-04 (email answers to ajain@cc.ucsf.edu)

Implement BubbleSort and QuickSort for integers

- Instrument your code
  - Count number of assignments
  - Count number of conditionals

Test the time complexity of your algorithms as follows

- For sizes of 100, 200, 300, … 1000
- Generate 100 random arrays
- Sort them using your code

You can use C, Python, Fortran, Lisp, Perl

Using the count data generated, illustrate the following:

- BubbleSort is $O(n^2)$ on average
- QuickSort is O(n log(n)) on average

What to turn in: a single PDF or Word or RTF file

- Readable listing of your code
- Input and output of both procedures on one example of size 100
- Graphical depiction of counts for assignments and conditionals for both functions
- Argument (graphical or textual) that the algorithms' average case performance is as expected

Email enclosure to: ajain@cc.ucsf.edu

Complexity complexities

We care about time complexity (see previous)

We may specifically care about best, worst, or average case complexity (usually average)

Very frequently there is a trade-off between time and space complexity

- Using a huge amount of memory can buy you time
- Example: finding a small gene sequence within a HUGE gene sequence

White board interlude 1

Computational complexity in the real world: Molecular similarity (2D versus 3D)

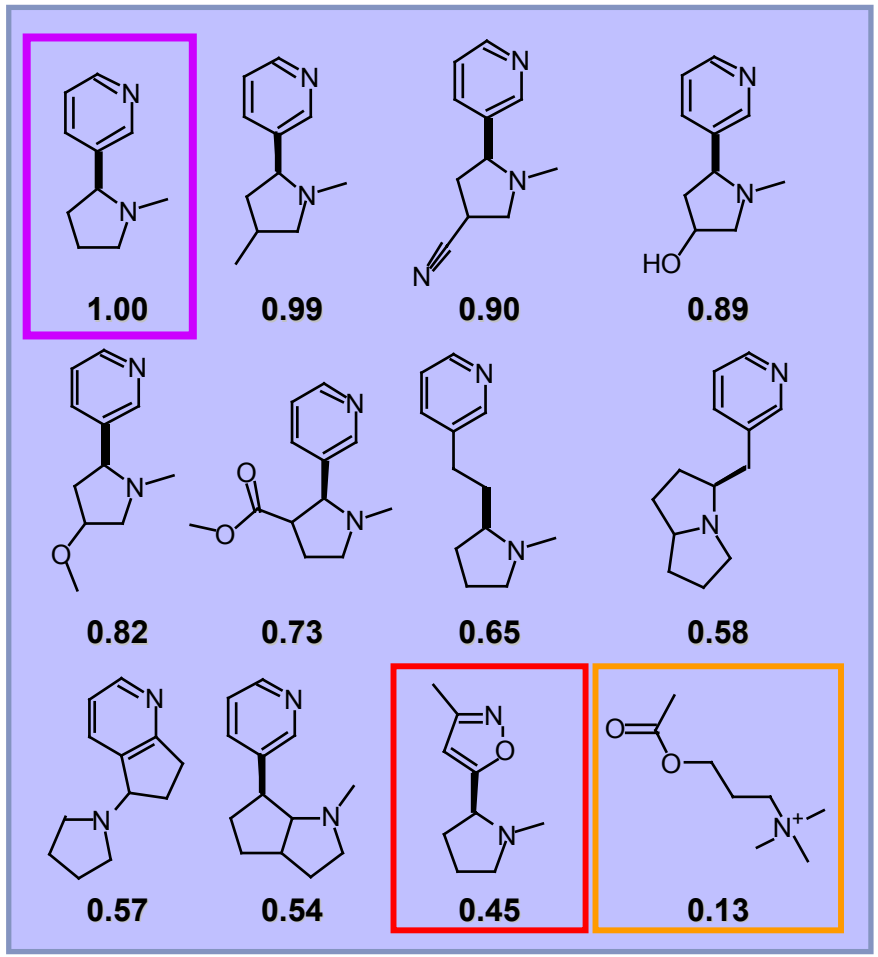
Nicotine example

- Nicotine
- Abbott molecule: competitive agonist
- Natural ligand (acetylcholine)
- Pyridine derivatives

2D similarity

- Graph-based approach to comparing organic structures
- Very efficient algorithm
- Can search 100,000 compounds in seconds

Ranked list versus nicotine places competitive ligands last



Molecular similarity: 2D versus 3D

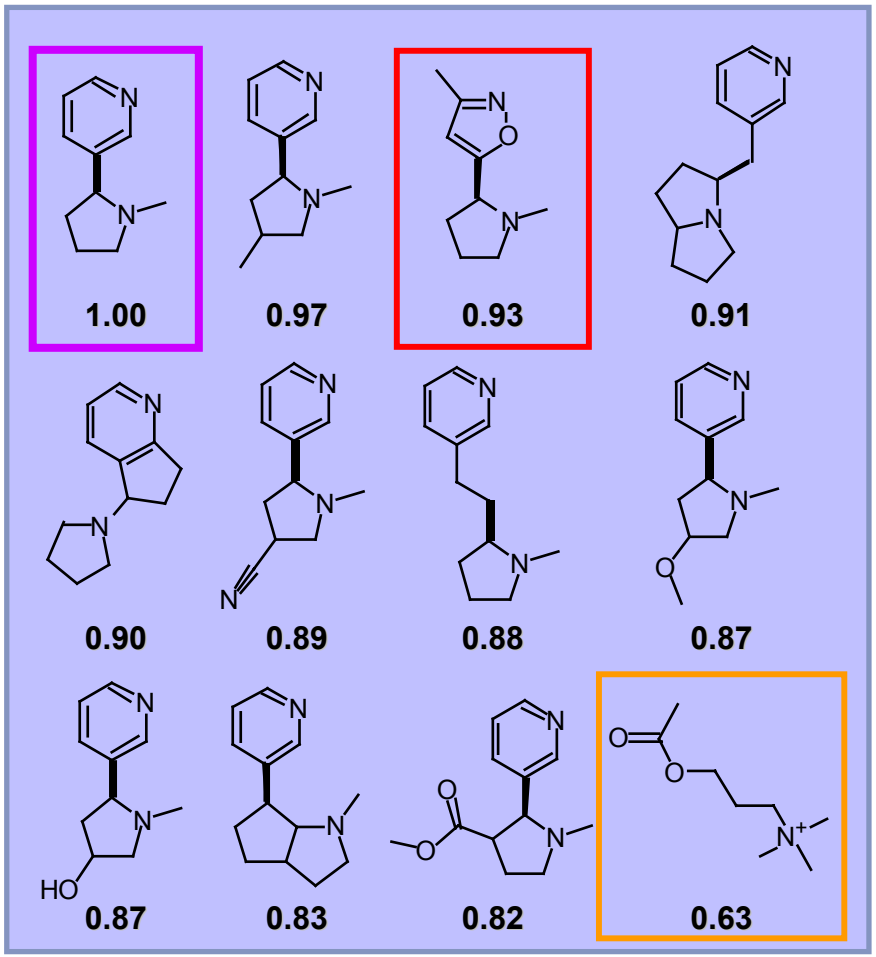
Nicotine example

- Nicotine
- Abbott molecule: competitive agonist
- Natural ligand (acetylcholine)
- Pyridine derivatives

3D similarity

- Surface-based comparison approach
- Requires dealing with molecular flexibility and alignment
- Much slower, but fast enough for practical use

Ranked list places the Abbot ligand near the top, and acetylcholine has a “high” score



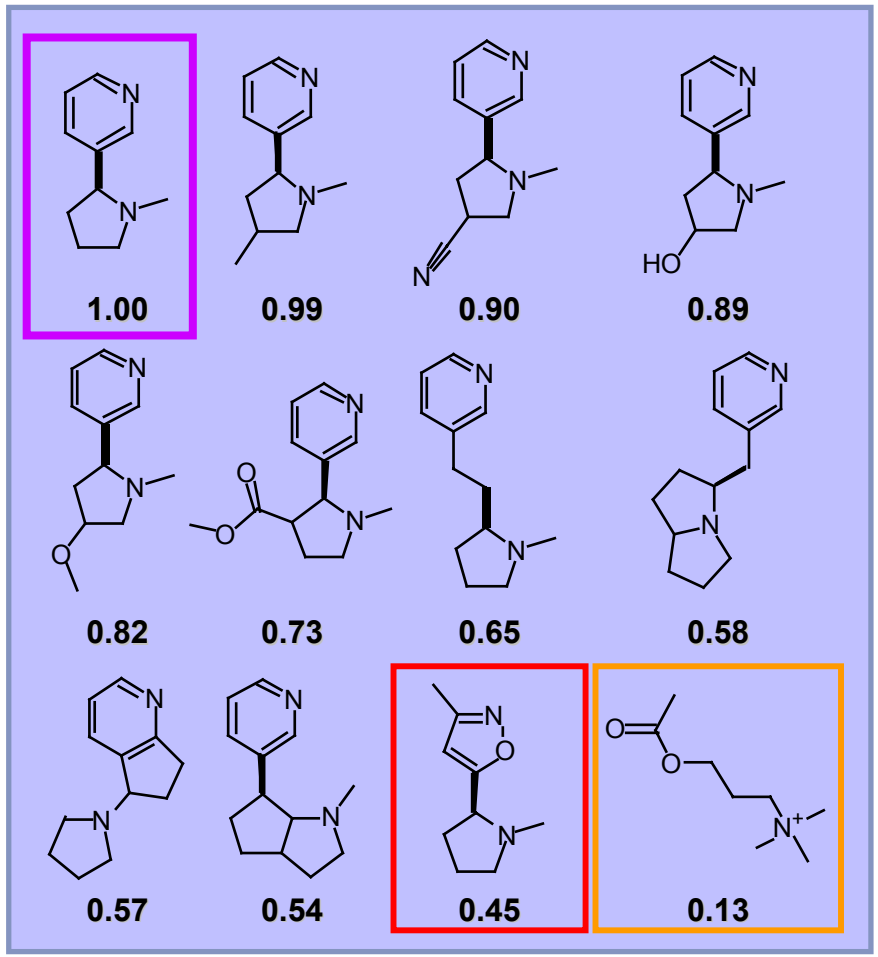
Molecular similarity: 2D

2D similarity

- Graph-based approach to comparing organic structures
- Very efficient algorithm
- Can search 100,000 compounds in seconds

What is the algorithm?

- We compute all atomic paths of length K in a molecule of size N atoms
- We mark a bit in a long bitstring if the corresponding path exists
- We fold the bitstring in half many times, performing an OR, thus yielding a short bitstring
- Given bitstrings A and B, we compute the number of bits in common divided by the total number of bits in either



Complexity: Computing the bitstring is O(N); computing S(A,B) is essentially constant time (small constant!)

UCSF

Molecular similarity: 3D

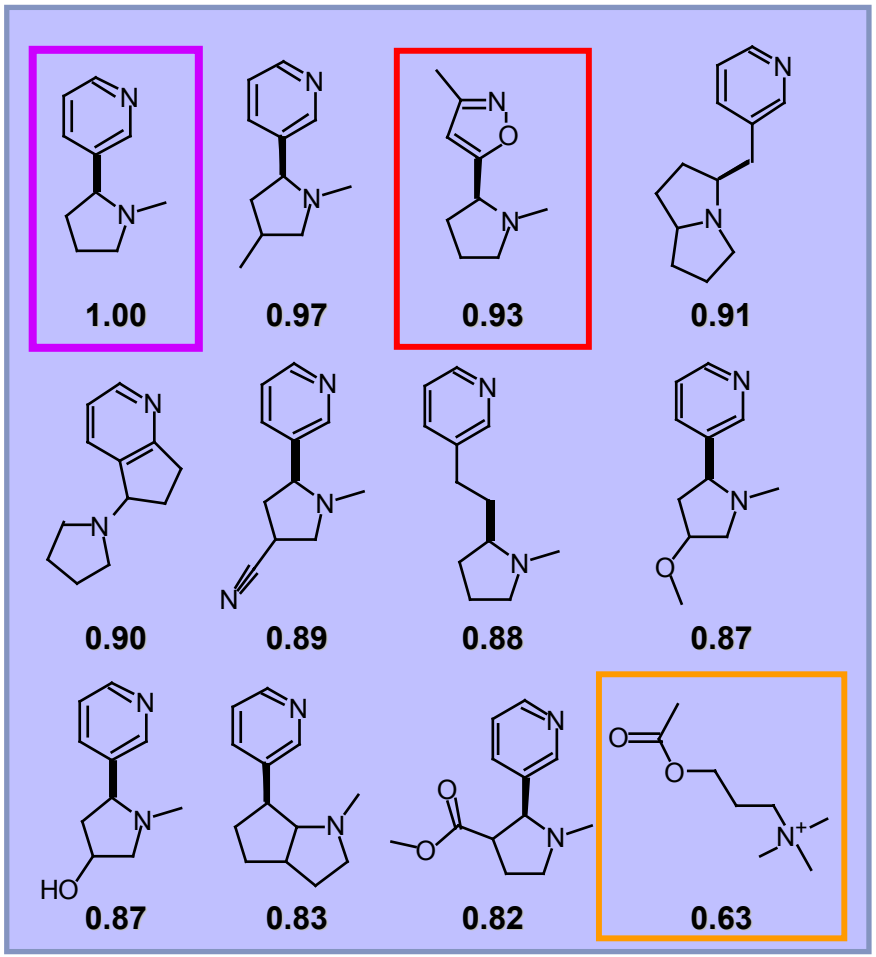
3D similarity

- Surface-based comparison approach
- Requires dealing with molecular flexibility and alignment
- Much slower, but fast enough for practical use

What is the algorithm?

- Take a sampling of the conformations of molecules A and B
- For each conformation, optimize the conformation and alignment of the other molecule to maximize S
- Report the average S for all optimizations

Key issues: not number of atoms. Number of rotatable bonds, alignment



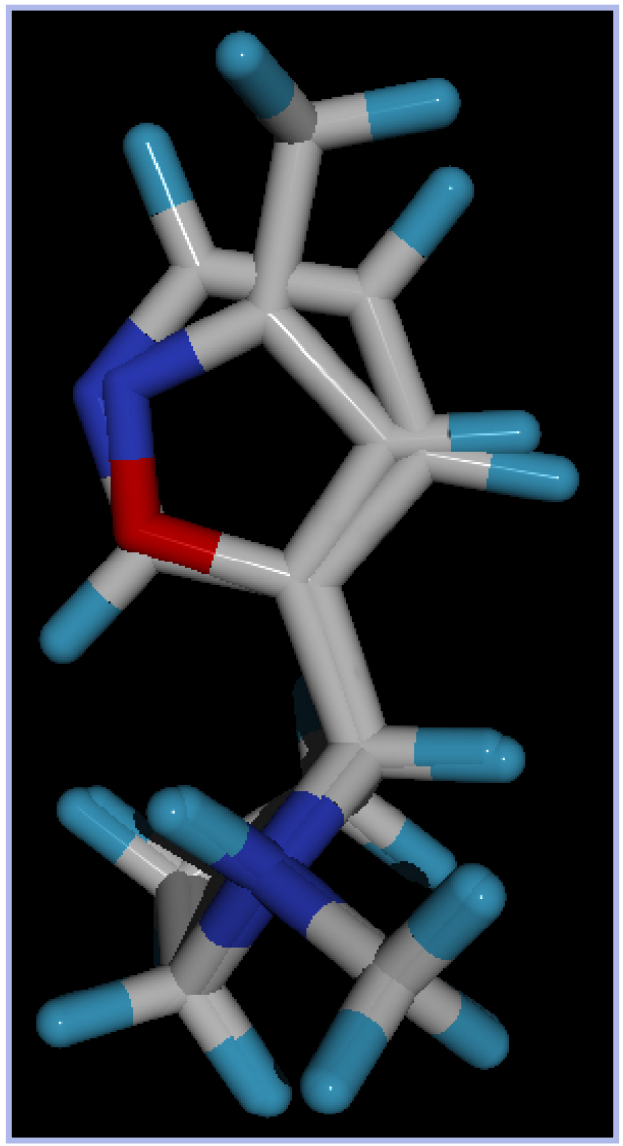
Similarity function: focus on what a protein sees

Formalize our intuition about molecules' non-covalent protein-ligand interactions

- Molecular surface interactions
- Hydrophobic interactions
- Hydrogen bonds
- Salt-bridges

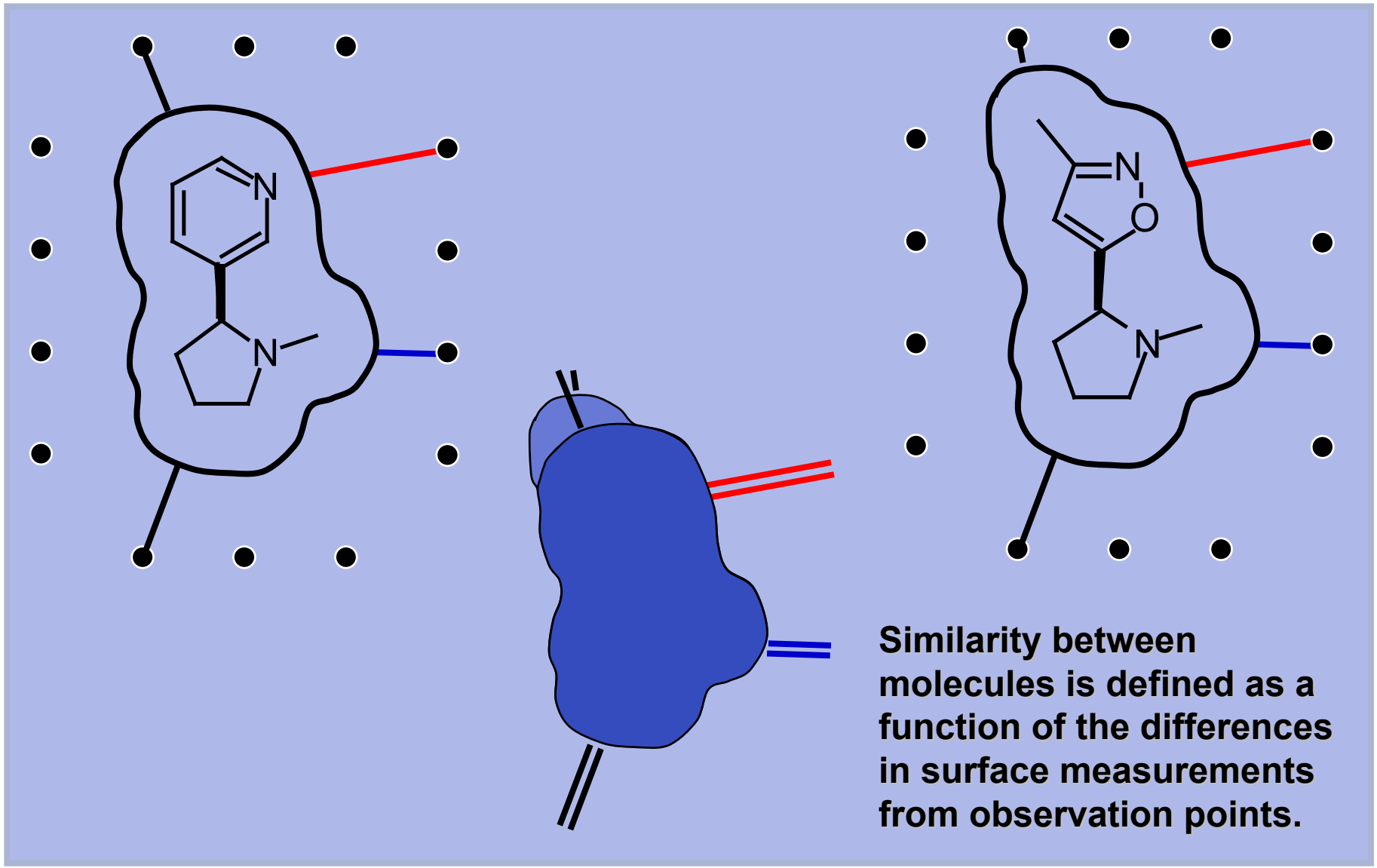
We don't care about

- Graph identity
- Whether we've got a "pyridine" or an "oxazole"
- Whether it's an N or an O that are making available an H-bonding partner

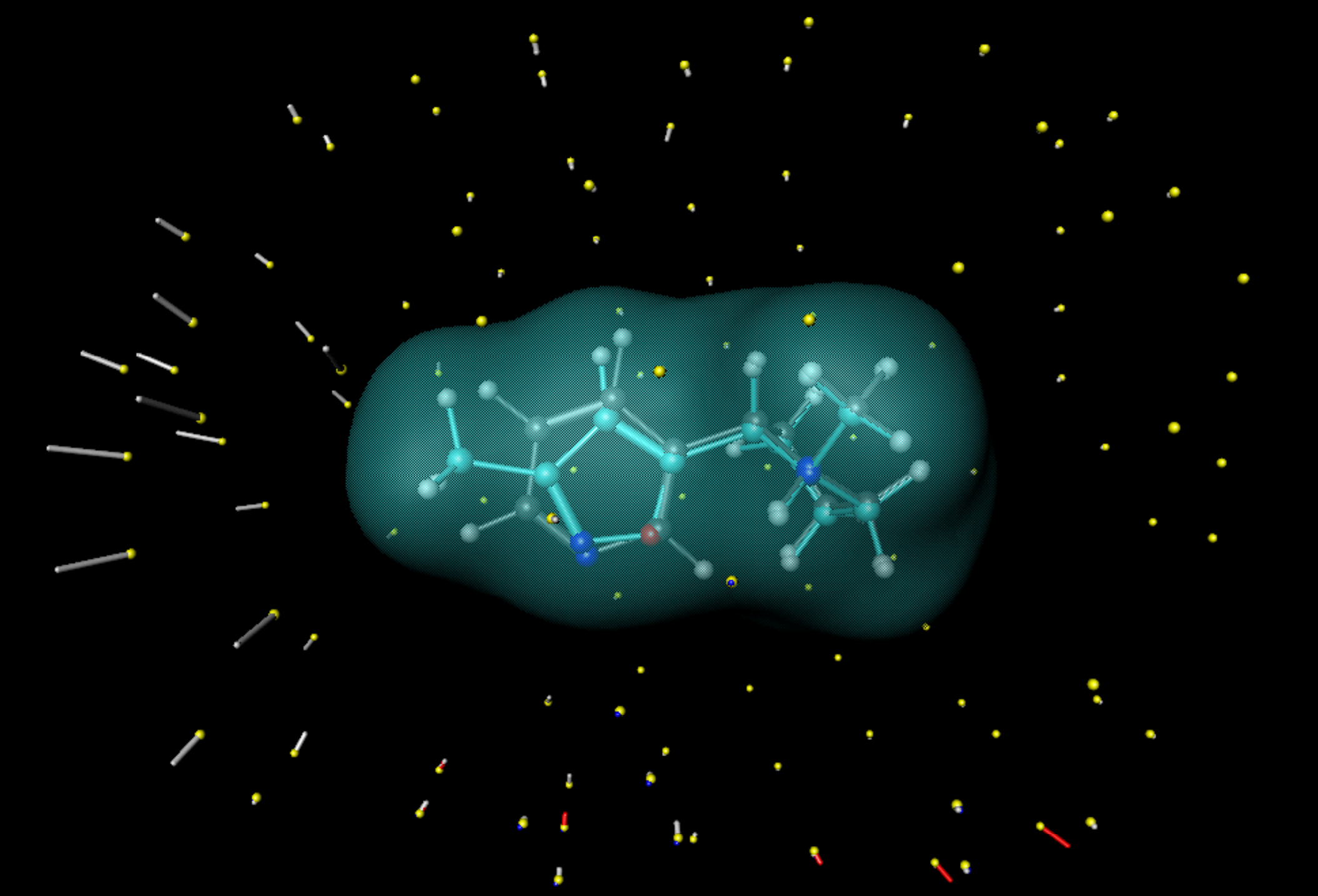


UCSF

Morphological similarity: Measure the molecules from the outside



Similarity between molecules is defined as a function of the differences in surface measurements from observation points.



UCSF

Complexity: bad news

Similarity function

- Given two molecules, each in a particular **pose**
  - Alignment parameters: $(X,Y,Z,\Theta_1, \Theta_2, \Theta_3)$
  - Conformation parameters $(\Phi_1,\Phi_2,\Phi_3,\ldots)$
- The similarity function has a weak dependence on the number of atoms
- We'll call it O(N) in number of atoms
- It's pretty fast, but not blindingly so

Now the bad news: optimization

- We have to find a pose of B that maximizes S(B,A)
- Alignment
  - Assume optimal alignment will not deviate more than 10A from mutual center of mass
  - Sample X,Y,Z at 0.25A
  - 40^3 = 64,000 translations
  - Rotation: sample at 5^0
  - $(360/5)^3$ = 373,248 rotations
  - Total: $24e^9$ ("constant time" but constant is BIG)
- Conformation: K rotatable bonds, sampled 6x each: 6^K

UCSF

Discussion

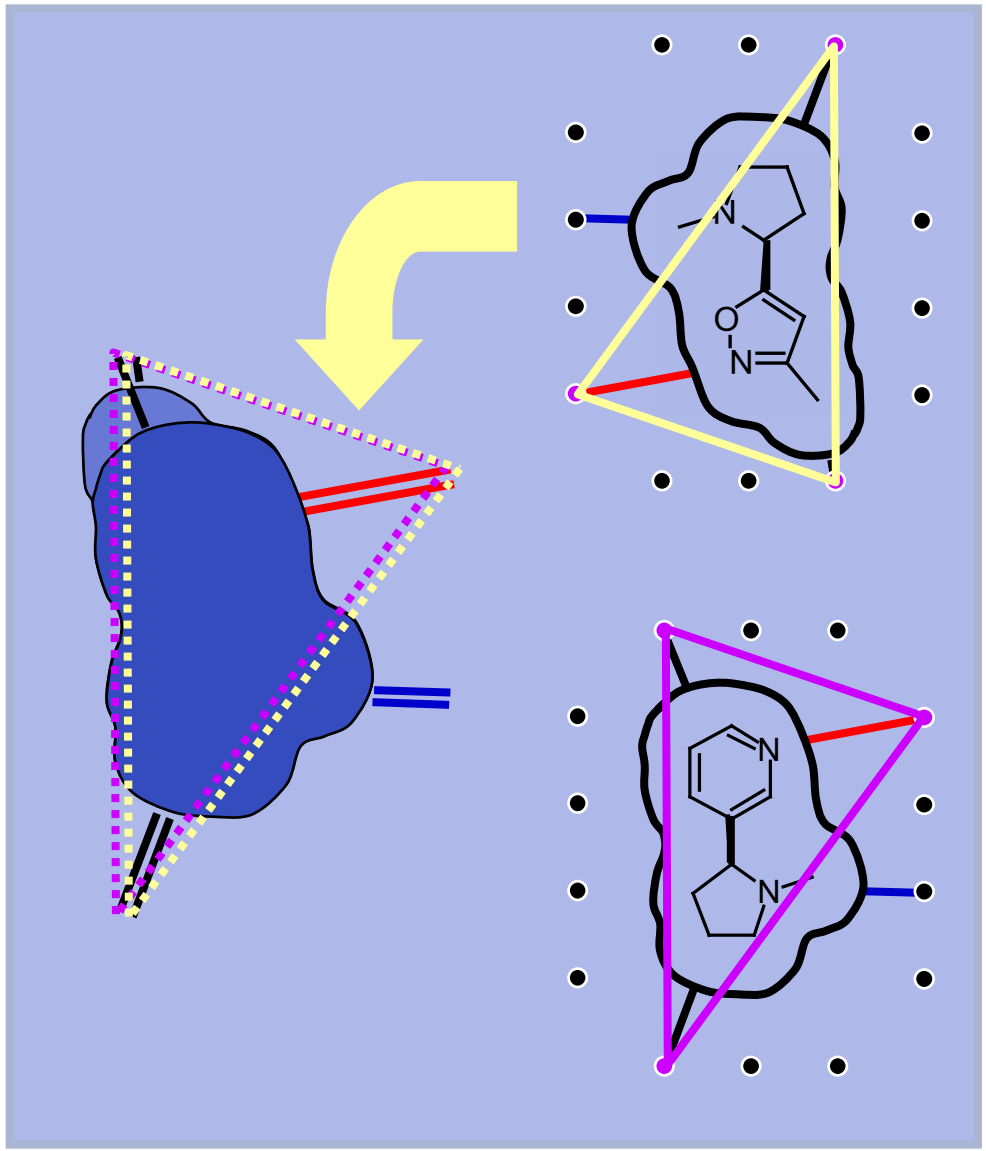
Alignment: approximate solution

Don’t compute similarity for all alignment samplings!

Find a set of alignments likely to yield high scores

- Find observation point triplet correspondences that satisfy:
  - Local similarity is high
  - Internal distances are similar
  - Approximation to full similarity is high
- Evaluate top transformations from above
- Return the best alignment

Perform gradient-based optimization and return final alignment



Conformation: approximate solution

Break molecule into fragments

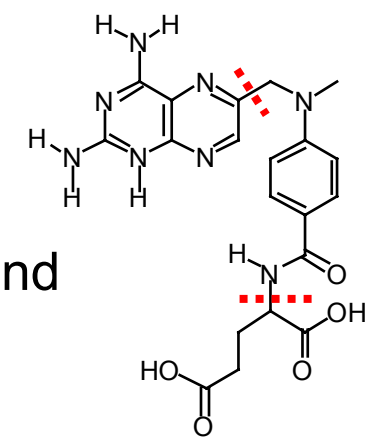
- 10 rotatable bonds: 6^{10} = 60 million conformations
- Assume that we can find good similarity based on a piece of a molecule
- Sample each fragment into a maximum number of conformations

Align each fragment, keep the best

Perform directed alignment of missing pieces, using their sampled conformations

Perform gradient-based optimization of conformation and alignment

Report the best pose



Complexity

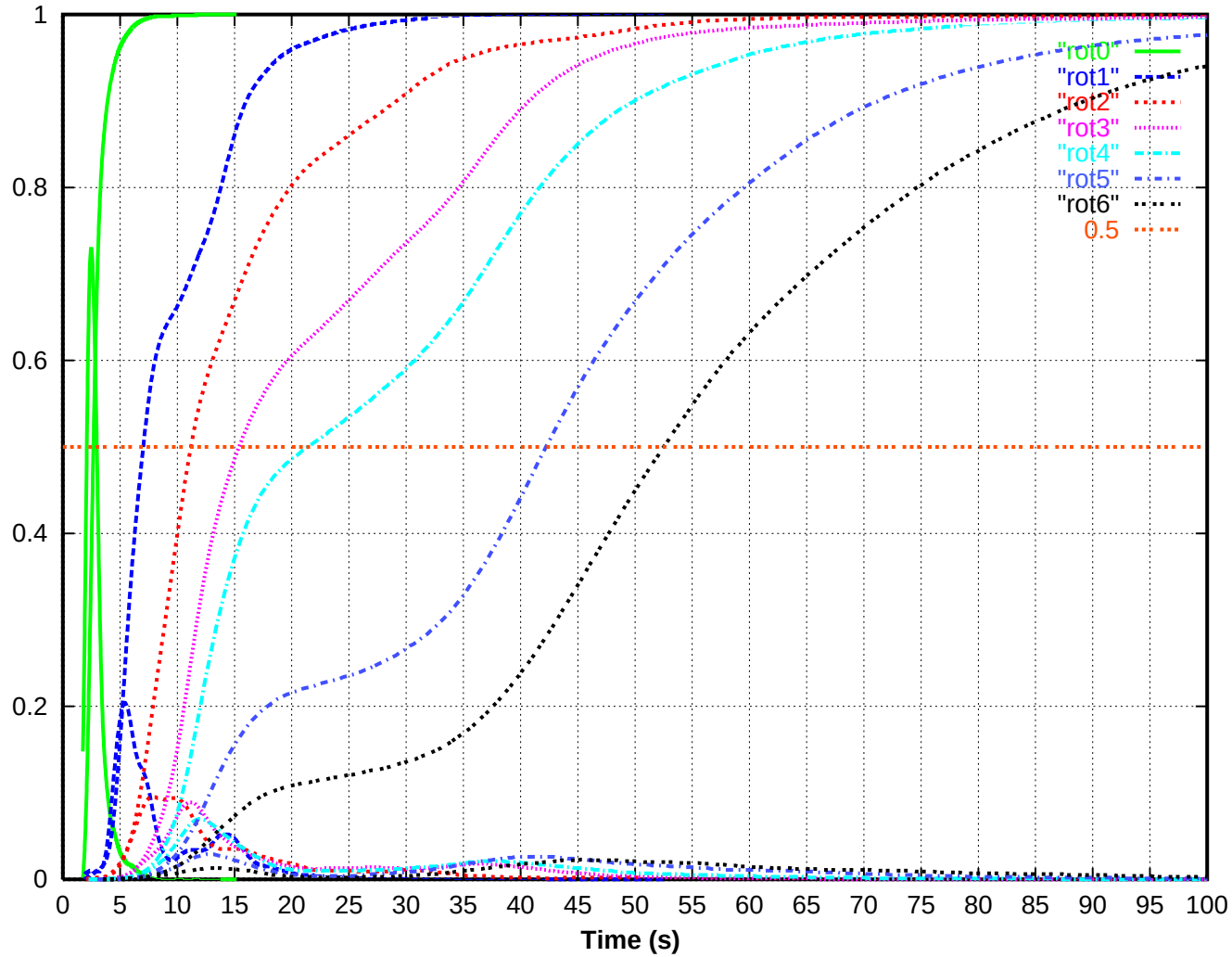
- Number of sampled conformations now **linear** in number of rotatable bonds (only hundreds, not millions)
- Number of alignment optimizations **linear** in number of samples
- Number of fragment merge operations **linear** in number of rotatable bonds
- Expect O(N) for overall algorithm

UCSF

Discussion

Complexity is essentially linear in number of rotatable bonds

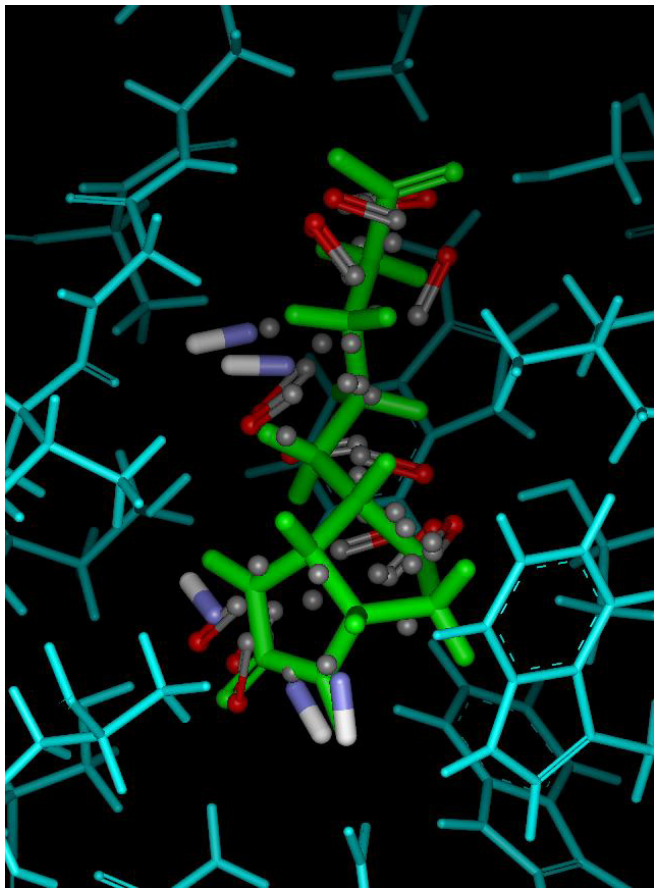
Speed is Bi-Modal: **Fragmented and non-fragmented fall into mixed normal distributions**

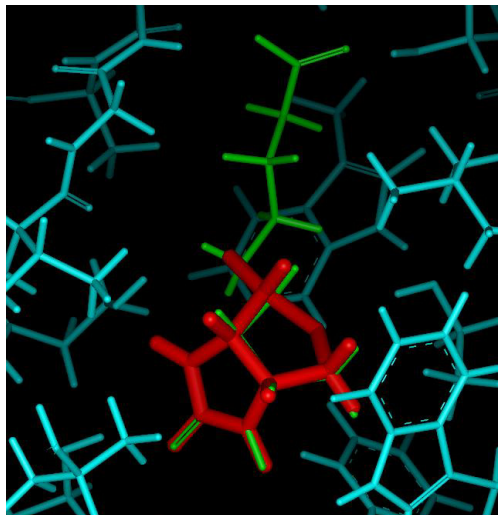


Time of conformation and alignment optimization CDFs for over 100,000 molecules flexibly optimized to maximize similarity to other molecules (chosen to be diverse).

Molecular docking follows the same argument

Divide and conquer strategy essentially linearizes the complexity of docking flexible molecules to protein binding sites



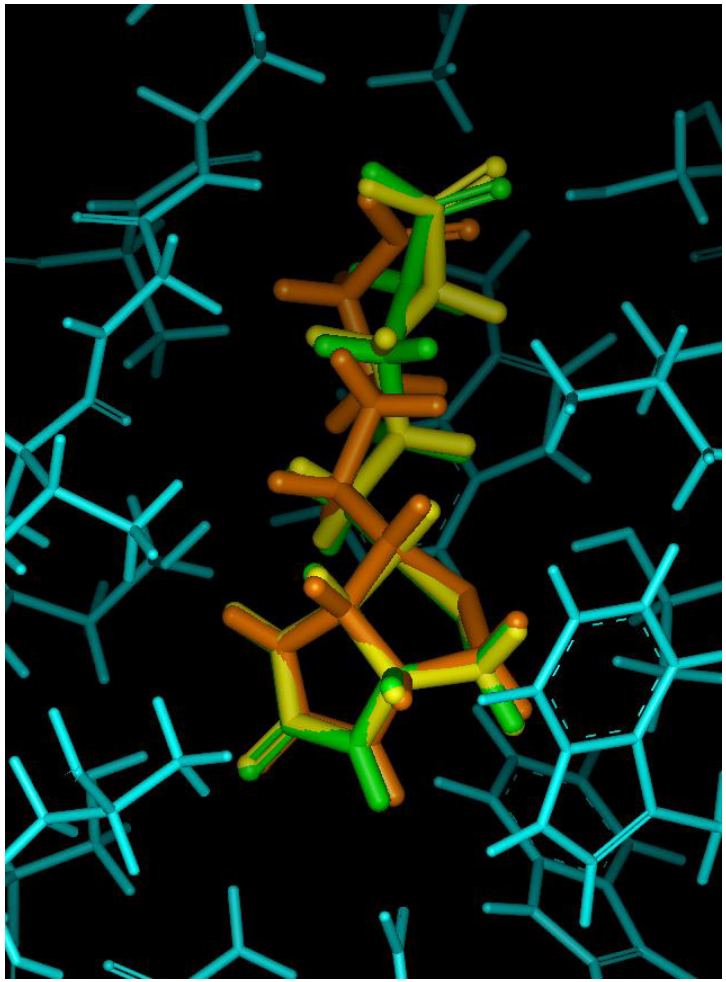


Small probes are complementary to protein

We use these to generate alignments

We dock the fragments of biotin

We chain off of the best ones



Biotin docked to streptavidin from random initial conformation. Best 2 scoring poses shown relative the to crystallographic conformation (green). This takes about a minute on a PIII 400MHz Wintel machine.

Conclusions

Algorithm complexity analysis can be a useful tool in understanding performance issues

Complexity theory allows us to formalize the notions of "how fast" or "how big" depending on the complexity of input

Even though "constant factors" mostly don't matter, when the constants are **HUGE**, they really really do

Many problems of interest are formally intractable, but clever non-exhaustive approaches can often come close enough that such problems can have useful and practical solutions