**Search**

- Motivations
  - Play tic-tac-toe
  - Play chess
  - Play with the Web
  - Play Darwin*

*Except in Kansas ...

---

**The Human Genome Project**

- human DNA is a string of ~3 billion letters (A, T, G, C), making up about 20,000-25,000 genes

---

**“Genetics 101”**

Cells are the fundamental working units of every living system. All the instructions needed to direct their activities are contained within the chemical DNA (deoxyribonucleic acid).

DNA from all organisms is made up of the same chemical and physical components. The DNA sequence is the particular side-by-side arrangement of bases along the DNA strand (e.g., ATCGGA). This order spells out the exact instructions required to create a particular organism with its unique traits.

The genome is an organism's complete set of DNA. Genomes vary widely in size: the smallest known genome for a free-living organism (a bacterium) contains about 600,000 DNA base pairs, while human and mouse genomes have some 3 billion.

DNA in the human genome is arranged into 24 distinct chromosomes—physically separate molecules that range in length from about 50 million to 250 million base pairs. A few types of major chromosomal abnormalities, including missing or extra copies or gross breaks and rejoinings (translocations), can be detected by microscopic examination. Most changes in DNA, however, are more subtle and require a closer analysis of the DNA molecule to find perhaps single-base differences.

Each chromosome contains many genes, the basic physical and functional units of heredity. Genes are specific sequences of bases that encode instructions on how to make proteins. Genes comprise only about 2% of the human genome; the remaining consists of noncoding regions, whose functions may include providing chromosomal structural integrity and regulating where, when, and in what quantity proteins are made. The human genome is estimated to contain 20,000-25,000 genes.

---

**The Human Genome Project**

- Good news: truckloads of data
- Bad news: what does it mean?
- Figure it out (in part) by matching
  - match unknown sequence against sequences of known functionality
  - the hope: similarity of structure suggests similarity of function

---

**Central Dogma of Modern Biology**

- DNA encodes genes and is inherited
- DNA is transcribed under control of proteins into RNA
- RNA is translated into proteins by ribosomes
- Proteins run the cell, and thus organisms
Genetics

- Proteins are made up of amino acids
- DNA represents each amino acid by a triple of letters in the "alphabet" of 4 nucleotides: adenine, thymine, guanine, cytosine.
- Hence
  - two similar sequences of DNA letters ➔
  - two similar sequences of amino acids ➔
  - two similar structures in proteins ➔
  - similar biochemical behavior of the proteins

Searching for similarity

- Same idea holds in other domains
  - Medical diagnosis and treatment
    - Find examples in database of similar cases and lookup treatment and prognosis
  - Marketing analytics
    - Look for patterns in custom usage and relate to customer behavior
    - Anti-terrorism analysis
      - Look for patterns in communication traffic or actual physical movement patterns and relate to behaviors of groups

Matching in genetics case

unk:   a t c g c c t a t t g t c g a c c
known: a t a g c a g c t c a t c g a c g

The Biology Behind Matching

- Evolution happens.
- Changes to the genome during replication:
  - Point mutations: change a letter, e.g., C ➔ A
  - Omissions: drop a letter
  - Insertions: insert a letter
- Similarity of sequence useful to discover
  - Similarity of function
  - Evolutionary history

More Complex Example

\[
\begin{align*}
\text{a a t c t g c} & \text{c a t t g} & \text{t c g a c g c} \\
\text{m m m m} & \text{m m m} & \text{m m} \\
\text{a a t c a g c} & \text{a g c t c a t} & \text{c g a c g g} \\
\text{m m m m m} & \text{m m m m m} & \text{m m m m m}
\end{align*}
\]

\[
\begin{align*}
\text{a a t c a g c} & \text{a g c t c a t} & \text{c g a c g g} \\
\text{a a t c a g c} & \text{a g c t c a t} & \text{c g a c g g}
\end{align*}
\]

Matching

- Every differing position has 3 possible explanations:
  - mutation
  - insertion
  - deletion
Matching As Tree Search

Every path through the tree is an hypothesis about matching sequences.
Want to evaluate likelihood of path to a leaf of the tree, and compare to other
paths to leaves – this lets us decide how similar two sequences are, and
causes of differences in sequences.

Depth first search

Breadth first search

If it's 6.001

• It's gotta have code:
  • Represent a tree by:
    • a root node (the start of the tree)
    • Plus a set of “children” nodes for each node, unless the node is a leaf (has no children)
  • Represent search by:
    • a queue of nodes to visit

If it's 6.001

• It's gotta have code:

```scheme
(define (dfsearch start-state)
  (define (search1 queue)
    (cond ((null? queue)
            (display "done"))
          (else
            (display "visiting ")
            (display (car queue))
            (search1 (append (children (car queue))
                           (cdr queue))))))
  (search1 (list start-state)))
```

If it's 6.001

• It's gotta have code:

```scheme
(define (bfsearch start-state)
  (define (search1 queue)
    (cond ((null? queue)
            (display "done"))
          (else
            (display "visiting ")
            (display (car queue))
            (search1 (append (cdr queue) 
                            (children (car queue))))))))
  (search1 (list start-state)))
```
Matching
Now that we can search a tree, how do we decide which paths are more interesting?

\[
\begin{align*}
\text{atcagctatttgctgacc} \\
\text{atatgctatttgctgacc} \\
\text{ataxgctatttgctgacc} \\
\text{atagctatttgctgacc}
\end{align*}
\]

Define a Distance Metric
- Given two sequences, s1 & s2,
  - Distance is 0 if they are identical
  - Penalty for each point mutation
    - Different for different mutations
  - Penalty for insertion/deletion of nucleotides
  - “Distance” is sum of penalties
- Now we can get the best explanation.

Representing Mutation Penalty

\[
\begin{array}{cccc}
 & A & C & G & T \\
A & 0 & .3 & .4 & .3 \\
C & .4 & 0 & .2 & .3 \\
G & .1 & .3 & 0 & .2 \\
T & .3 & .4 & 1 & 0
\end{array}
\]

2-D Table

\[
\begin{align*}
\text{(define point-mutations (make-table2))} \\
\text{(table2-set! point-mutations 'A 'A 0)} \\
\text{(table2-set! point-mutations 'A 'C .3)} \\
\text{(table2-set! point-mutations 'A 'G .4)} \\
\ldots
\text{(table2-get point-mutations 'A 'C)} \\
\text{=> .3} \\
\text{(table2-get point-mutations 'A 'X)} \\
\text{=> #f}
\end{align*}
\]

- But how to implement a 2-D table?

A Table Abstraction using alists

\[
\begin{align*}
\text{(define (find-assoc-binding key alist)} \\
\text{(cond ((null? alist) #f)} \\
\text{((equal? key (caar alist)) (car alist)} \\
\text{((else (find-assoc-binding key (cdr alist)))))}
\end{align*}
\]

\[
\begin{align*}
\text{(define (find-assoc key alist)} \\
\text{(let ((binding (find-assoc-binding key alist)} \\
\text{(if binding (cadr binding) #f)}))}
\end{align*}
\]

\[
\begin{align*}
\text{(define (add-assoc key val alist)} \\
\text{(cons (list key val) alist))}
\end{align*}
\]

Non-Abstract but Compact!

\[
\begin{align*}
\text{(define mutation-penalties)} \\
\text{('((a (c .3) (g .4) (t .3))} \\
\text{ ((c (a .4) (g .2) (t .3))} \\
\text{ ((g (a .1) (c .3) (t .2))} \\
\text{ (t (a .3) (c .4) (g .1))})}
\end{align*}
\]

\[
\begin{align*}
\text{(define (mutation to from)} \\
\text{(if (eq? from to) 0)} \\
\text{((let ((row (find-assoc-binding to mutation-penalties)))))} \\
\text{((if row (find-assoc from (cdr row)) #f))}
\end{align*}
\]
A Table ADT

(define table1-tag 'table1)
(define (make-table) (value table1-tag '()))
(define (table-get tbl key)
  (find-assoc key (cdr tbl))
  (set-cdr! tbl (add-assoc! key value (cdr tbl))))

• Note: we mutate structure, unlike before.

Table2 is a table of Table1’s

(define table2-tag 'table2)
(define (make-table) (cons table2-tag (cons table1-tag (make-table)))
(define (table-get tbl row key-col)
  (let ((row (table-get (cdr tbl) row)))
    (if row
        (table-get row key-col) #f))
(define (table-set! tbl row key-col val)
  (let ((row (table-get (cdr tbl) row)))
    (if row
        (table-set! row key-col val) #f))
    (let ((new-row (make-table)))
      (table-set! new-row key-col val)
      (table-set! (cdr tbl) row new-row))))

We have the Penalties

point-mutations
  | {table2}
  | {table1}
  | (t (table1 (t 0) (g 0.1) (c 0.4) (a 0.3)))
  | (g (table1 (t 0.2) (g 0.3) (c 0.1))
  | (c (table1 (t 0.3) (g 0.2) (c 0.0) (a 0.4)))
  | (a (table1 (t 0.3) (g 0.4) (c 0.3) (a 0.0)))
  | (define omit-penalty .5)
  | (define insert-penalty .7)

Defining Mutations More Abstractly

(table2-set! point-mutations 'a 'a 0)
(table2-set! point-mutations 'a 'c 0.3) ;; e.g., from c to a
(table2-set! point-mutations 'a 'g 0.4)
(table2-set! point-mutations 'a 't 0.3)
(table2-set! point-mutations 'c 't 0.3)
(table2-set! point-mutations 'g 't 0.2)
(table2-set! point-mutations 'c 't 0.3)
(table2-set! point-mutations 'g 't 0.1)
(table2-set! point-mutations 'c 'g 0.2)
(table2-set! point-mutations 'c 'g 0.1)
(table2-set! point-mutations 'c 'g 0.2)
(table2-set! point-mutations 'c 'g 0.1)
(table2-set! point-mutations 'c 'g 0.2)
(table2-set! point-mutations 'c 'g 0.1)
(table2-set! point-mutations 'c 'g 0.2)
(table2-set! point-mutations 'c 'g 0.1)

Simplest Matcher

(define (match0 one two)
  (define (helper x y score)
    (cond ((and (null? x) (null? y)) score)
      ((null? x) (helper x (cdr y) (+ score omit-penalty)))
      ((null? y) (helper (cdr x) y (+ score insert-penalty)))
      ((eq? (car x) (car y)) (helper (cdr x) (cdr y) score))
      (else (let ((mutated (helper (cdr x) (cdr y) (+ score (mutation (car x) (car y))))
                    (omitted (helper (cdr x) y (+ score omit-penalty)))
                    (inserted (helper (car x) y (+ score insert-penalty)))
                    (min mutated omitted inserted))
                    (helper x one two))))
  (helper one two 0.0))

… sloooooooooooooow!!!
Matching As Tree Search

\[
\begin{array}{c}
\text{a} & \text{t} & \text{c} & \text{a} & \text{g} & \text{c} & \text{g} \\
\text{a} & \text{g} & \text{t} & \text{c} & \text{a} & \text{g} & \text{c} & \text{g} & \text{a} & \text{c} & \text{g} & \text{g}
\end{array}
\]

Time complexity? \( T(n) = \Theta(3^n) \)

Observation

\[
\begin{array}{c}
\text{a} & \text{t} & \text{c} & \text{a} & \text{g} & \text{c} & \text{g} & \text{a} & \text{c} & \text{g} & \text{g} \\
\text{a} & \text{g} & \text{t} & \text{c} & \text{a} & \text{g} & \text{c} & \text{g} & \text{a} & \text{c} & \text{g} & \text{g}
\end{array}
\]

Memory to the Rescue

- "Memoization"
- Store the results of computing sub-paths and substitute lookup for computation
- How to store the results?
- (Still, \(-n^2\))

Remember Fibonacci

\[
\begin{align*}
&\text{(define (fib n)} \\
&\quad \text{(cond ((= n 0) 0) \nl &\quad \text{((= n 1) 1) \nl &\quad \text{(else (+ (fib (- n 1)) (fib (- n 2)))))})}
\end{align*}
\]

\( T(n) = \Theta(\phi^n) \)

\[
\begin{align*}
&\text{(define (fibmemo n)} \\
&\quad \text{(let ((old-val (table1-get old-vals n)))} \\
&\quad \text{(else (let ((new-val \}
\end{align*}
\]

\( T(n) = \Theta(n) \)

Better Memoized Matching

\[
\begin{align*}
&\text{(define (match1 one two)} \\
&\quad \text{(let ((past (make-table2)))} \\
&\quad \text{(define (helper x y score)} \\
&\quad \text{(let ((old (table2-get past x y)))} \\
&\quad \text{(if old (score))} \\
&\quad \text{(let (new \}
\end{align*}
\]

\[
\begin{align*}
&\quad \text{(table2-set! past x y (- new score))} \\
&\quad \text{(helper one two 0.0))})}
\end{align*}
\]

\( T(n) = \Theta(n^2) \)

- We store best score from here \((x,y)\) to end.
- Still too slow for long sequences!
- Can we not consider some of the worst partial matches?

Can We Be Smarter Still?

- Cut off bad paths:
  - Estimate an upper bound on matches of interest
  - Declare any match worse than this to be infinitely bad (and stop pursuing it)
- Advantages?
- Disadvantage?
Idea: Pursue “Best” Matches

Best First Search

• Extend only the best sequence

(class (bestsearch start-state)
  (define (search1 queue)
    (cond ((done? (car queue))
      (display "done")
      (car queue))
      (else
        (display "visiting ")
        (display (car queue))
        (search1 (merge (sort (children (car queue)))
                     (cdr queue))))))))

Best First Search

Beam Search

• Beam: like best-first, but keep only n best children of a node

Varieties of Search

• depth first
  (append (children (car queue))(cdr queue))

• breadth first
  (append (cdr queue)(children (car queue)))

• best first
  (merge (sort (children (car queue)))
         (cdr queue))

• beam search
  (merge (list-head n (sort (children (car queue))))
         (cdr queue))

General Search Framework

Return of the Biologists

• Short queries, large databases...

• Some large subsequences are common (clichés)

• Good matches will contain large identical subsequences

• Pre-compute table of all occurrences of specific patterns

• Extend match outward (both directions) from these exact matches
BLAST: Find common, extend

Generalize

- DNA
  - Nucleotides: A, C, T, G
  - Mutation rates
  - Insertion/omission penalties

- Proteins
  - Amino Acids: val, leu, ile, met, phe, asn, glu, gln, ...
  - Mutation rates
  - Insertion/omission penalties

Let's Play Games...