Important Example: Gene Sequence Matching

- “Century of Biology”
- Two views of computer science’s relationship to biology:
  - Bioinformatics: computational methods to help discover new biology from lots of data
  - Engineering on biological substrates: using biology instead of electronics

Corrigendum

(define (filter! f lst)
  (if (null? lst)
      '()
      (let ((root (cons '() lst)))
        (define (iter l)
          (cond ((null? (cdr l)))
            ((f (cadr l)) (iter (cdr l)))
            (else
              (set-cdr! l (cddr l))
              (iter l)))))
      (iter root)
      (cdr root))))

Central Dogma of Modern Biology

- DNA encodes genes & 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 assembled from a sequence of amino acids
- DNA (and RNA) represent each amino acid by a triple of codes in a 4-letter “alphabet” of nucleotides: guanine, cytosine, adenine and thymine (uracil in RNA), which we denote by G, C, A and T
- How many amino acids could we encode?

\[ 4^3 = 64 \]

How Nucleotides code for Amino Acids

Biological Matching Heuristic

- Evolution makes incremental changes in genome
  - Point mutations: e.g., G \rightarrow\ A
  - Dropping or inserting a nucleotide during copying
  - Gene duplication, then
    - Separate drift
    - Separate function
- Similarity of sequence useful to discover
  - Similarity of function
  - Evolutionary history
- Given a gene sequence with unknown function, match it against ones with known function to get clues
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
  - May be asymmetric, so not true distance
- Corresponds to log(probability), assuming independence of each mutation

Matching Paradigm

- Given a new section of DNA of interest, match it against all potentially useful segments of known-function DNA
- Best match(es) may lead to insight

Details of matching

```
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>.3</td>
<td>.4</td>
<td>.3</td>
</tr>
<tr>
<td>C</td>
<td>.4</td>
<td>0</td>
<td>.2</td>
<td>.3</td>
</tr>
<tr>
<td>G</td>
<td>.1</td>
<td>.3</td>
<td>0</td>
<td>.2</td>
</tr>
<tr>
<td>T</td>
<td>.3</td>
<td>.4</td>
<td>.1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Representing Mutation Penalty

2-D Table

```scheme
(define point-mutations (make-table2))
(table2-set! point-mutations 'A 'A 0)
(table2-set! point-mutations 'A 'C .3)
(table2-set! point-mutations 'A 'G .4)
...```

Remember Table Abstractions

```scheme
(define (find-assoc-binding key alist)
  (cond ((null? alist) #f)
        ((equal? key (caar alist)) (car alist))
        (else (find-assoc-binding key (cdr alist))))

(define (find-assoc key alist)
  (let ((binding (find-assoc-binding key alist)))
    (if binding
      (cadr binding)
      #f)))
```

Note Scheme’s assoc assv assq
Non-Abstract but Compact!

```
(define mutation-penalties
  '((a 0.4) (c 0.2) (g 0.1) (t 0.3))
  (c 0.2) (g 0.1) (t 0.3))
  (g 0.1) (c 0.3) (t 0.25))
  (t 0.3) (c 0.4) (g 0.2))
)
```

```
(define (mutation to from)
  (if (eq? from to)
      0
      (let ((row (find-assoc-binding ; == assoc
          to mutation-penalties)))
        (if row
            (find-assoc from (cdr row)); == cadr
              #f)))))
```

Remember Table ADT

```
(define table1-tag 'table1)
(define (make-table1)
  (cons table1-tag '()))
(define (table1-get tbl key)
  (find-assoc key (cdr tbl)))
(define (table1-set! tbl key val)
  (set-cdr! tbl
    (add-assoc! key val (cdr tbl))))

• Note: we mutate structure, unlike before.
```

Mutating Version of add-assoc

```
(define (add-assoc! key val alist)
  (let ((binding (find-assoc-binding key alist)))
    (cond (binding
          (set-car! (cdr binding) val)
          alist)
          (else
            (add-assoc key val alist))))
```

Table2 is a table of Table1’s

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

Defining Mutations More Abstractly

```
(table2-set! point-mutations 'a 'a 0)
(table2-set! point-mutations 'a 'g 0.4)
(table2-set! point-mutations 'g 'a 0.1)
(table2-set! point-mutations 'g 'g 0.2)
(table2-set! point-mutations 't 'a 0.3)
(table2-set! point-mutations 't 't 0.0)
```

We have the Penalties

```
(point-mutations
  (table2
    (t (table1 [t 0] [g 0.1] [c 0.4] [a 0.3])
        [g (table1 [t 0.2] [g 0] [c 0.3] [a 0.1])]
        [c (table1 [t 0.3] [g 0.2] [c 0] [a 0.1])]
        [a (table1 [t 0.3] [g 0.4] [c 0.3] [a 0.1]))])
  (define omit-penalty .5)
  (define insert-penalty 0.7))
```
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 x (cdr y) (+ score omit-penalty))))
            (inserted
             (helper (cdr x) y (+ score insert-penalty))))
            (min mutated omitted inserted)))))
  (helper one two 0.0))

... sloooooooooooow!!!

Remember Fibonacci

(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1)) (fib (- n 2)))))
  (define old-vals (make-table))
  (define (fibmemo n)
    (let ((old-val (table1-get old-vals n)))
      (cond (old-val old-val)
            (else
              (let ((new-val
                     (cond ((= n 0) 0)
                           ((= n 1) 1)
                           (else (+ (fibmemo (- n 1))
                                     (fibmemo (- n 2)))))))
                  (table1-set! old-vals n new-val)
              new-val))))

Better Memoized Matching

(define (match1 one two)
  (let ((past (make-table2)))
    (define (helper x y score)
      (let ((old (table2-get past x y)))
        (cond (old (+ old score))
              (else (let ((new (match0 one two 0.0)))
                        (table2-set! past x y (- new score))
              new))))))
  (helper one two 0.0))

• 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?

Cutting 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)

Performance with Cutoff

<table>
<thead>
<tr>
<th>penalty (#past)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>a</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>a</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>a</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>a</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>a</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
</tr>
</tbody>
</table>

• Dramatic savings
• May miss answer if we guess cutoff badly
• Better: just pursue the “top k” paths
  – Reformulate as a search problem

Idea: Pursue “best” matches

x c a g...

[Diagram of alignments and scores]

- 0.3
- 0.5
- 0.7
- 0.6
- 0.8
- 1.0
Generalized Search

- **Depth-first**: pursue first alternative from last "move"
  - E.g., first consider string of all mutations
- **Breadth-first**: consider all "moves" of length 1, then 2, then ...
  - Must keep track of all possible move sequences
- **Best-first**: consider extending only the best sequence
  - Still must keep track of all
- **Beam**: like best-first, but remember only top $n$ sequences

Search State

```scheme
(define (make-search-state score x y history)
  (list score x y history))
(define ss-score car)
(define ss-x cadr)
(define ss-y caddr)
(define ss-history cadddr)
```

Framework for Search

```scheme
(define (search start-state done? succ-fn merge-fn)
  (define (inner queue)
    (if (null? queue)
      #f
      (let ((current (car queue)))
        (if (done? current)
          current
          (inner (merge-fn (succ-fn current) (cdr queue)))))))
  (inner (list start-state)))
```

Depth-first

- Finds best answer, but with maximum work
- For $n$ mismatches, BFS will generate $3^n$ states on the queue
- Won't find any solution until all partial matches have been put on queue

Breadth-first

```scheme
(define (match-dfs x y)
  (search (make-search-state 0.0 x y '())
    (lambda (new old)
      (append old new))))
```

Best-first

```
```
Best-first examples

(define (match-best-fs x y)
  (search (make-search-state
           0.0 x y '(())
           completed? successors priority-merge))

(define s1 '(a a t c g a t t g c a g c c t a t t g t c g a c g c))
(define s2 '(a a t c a g c a g c a g g))
(define s3 '(a g a t c a g c a c t c a t c g a c g g))

> (match-best-fs s1 s2)
(1.8 () ()
 ((m c g) (= g g) (= c c) (= a a)
  (= g g) (= c c) (= t t) (m g a)
  (m t c) (= t t) (m a c) (m t g)
  (m c a) (= c c) (= g g) (m t a)
  (= c c) (= t t) (= a a) (= a a)))

> (match-best-fs s2 s3)
(1.2 () ()
 ((= g g) (= g g) (= c c) (= a a)
  (= g g) (= c c) (= t t))
  (i g)
  (= a a) (= c c) (= g g) (= a a)
  (= c c) (= t t) (= a a)
  (o g)
  (= a a)))

But, We Forgot Something

• Dynamic Programming Principle:
  – If two partial solutions lead to the same
    search state, keep only the better one
  – We used memoization earlier
  – Current implementation fails to do this!

When merging queues

1. Drop from new any state for which
   – There is already a memoized state matching the
     same tails, and
   – The memoized state has a better score
2. Memoize the remaining new states
3. Drop from old any state for which
   – The state now memoized has a better score
   (Could only happen if a new state is better.)

Code for new merging rule
(define (my-merge new old)
  (let* ((new-useful ...
          (old-useful ...))
        (priority-merge new-useful old-useful)))

Final match-best-fs

(define (match-best-fs x y)
  (let* ((new-useful ...
          (old-useful ...))
        (priority-merge new-useful old-useful)))
  (search start completed? successors my-merge))

Beam Search
(define (match-best-fs x y width)
  (let* ((new-useful ...
          (old-useful ...))
        (priority-merge new-useful old-useful)))
  (search start completed? successors my-merge))
Performance

<table>
<thead>
<tr>
<th></th>
<th>Best-First-Search</th>
<th>Beam Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>penalty max queue</td>
<td></td>
<td></td>
</tr>
<tr>
<td># past</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s1,s2</td>
<td>1.8</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>34</td>
</tr>
<tr>
<td>s1,s3</td>
<td>2.8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>112</td>
</tr>
<tr>
<td>s2,s3</td>
<td>1.2</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>41</td>
</tr>
<tr>
<td>s1,s1,s2</td>
<td>3.6</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>57</td>
<td>68</td>
</tr>
<tr>
<td>s1,s1,s3</td>
<td>5.6</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>63</td>
<td>472</td>
</tr>
<tr>
<td>s1,s1,s1,s1</td>
<td>7.2</td>
<td>1</td>
</tr>
<tr>
<td>s1,s2,s2</td>
<td>119</td>
<td>136</td>
</tr>
<tr>
<td>s1,s1,s1,s1</td>
<td>11.2</td>
<td>106</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>1922</td>
</tr>
</tbody>
</table>

Greatest Improvements

- 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