CS342: Bioinformatics
Lecture 6

The Motif Finding Problem

- Goal: Given a set of DNA sequences, find a set of ℓ-mers, one from each sequence, that maximizes the consensus score

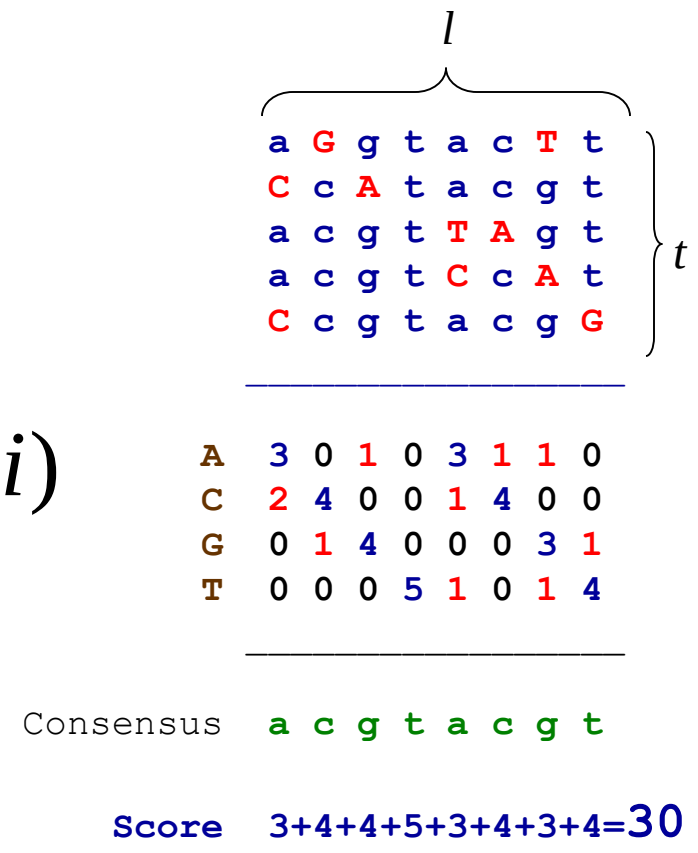
- Input: A ***t*** x ***n*** matrix of ***DNA***, and ℓ, the length of the pattern to find

- Output: An array of ***t*** starting positions
  s = (s_1, s_2, … s_t) maximizing *Score*(**s**,***DNA***)

Scoring Motifs

- Given **s** = $(s_1, \ldots s_t)$ and ***DNA***:

$$Score(\mathbf{s}, \mathbf{DNA}) = \sum_{i=1}^{l} \max_{k \in \{A,C,G,T\}} count(k, i)$$



Brute Force Solution

- Compute the scores for all possible combinations of starting positions **s**
- The best score determines the best profile and the consensus pattern in ***DNA***
- The goal is to maximize *Score*(**s**,***DNA,l***) by varying the starting positions s_i, where:

$$s_i = [1, \ldots, n\text{-}l+1]$$
$$i = [1, \ldots, t]$$

Brute Force Pseudocode

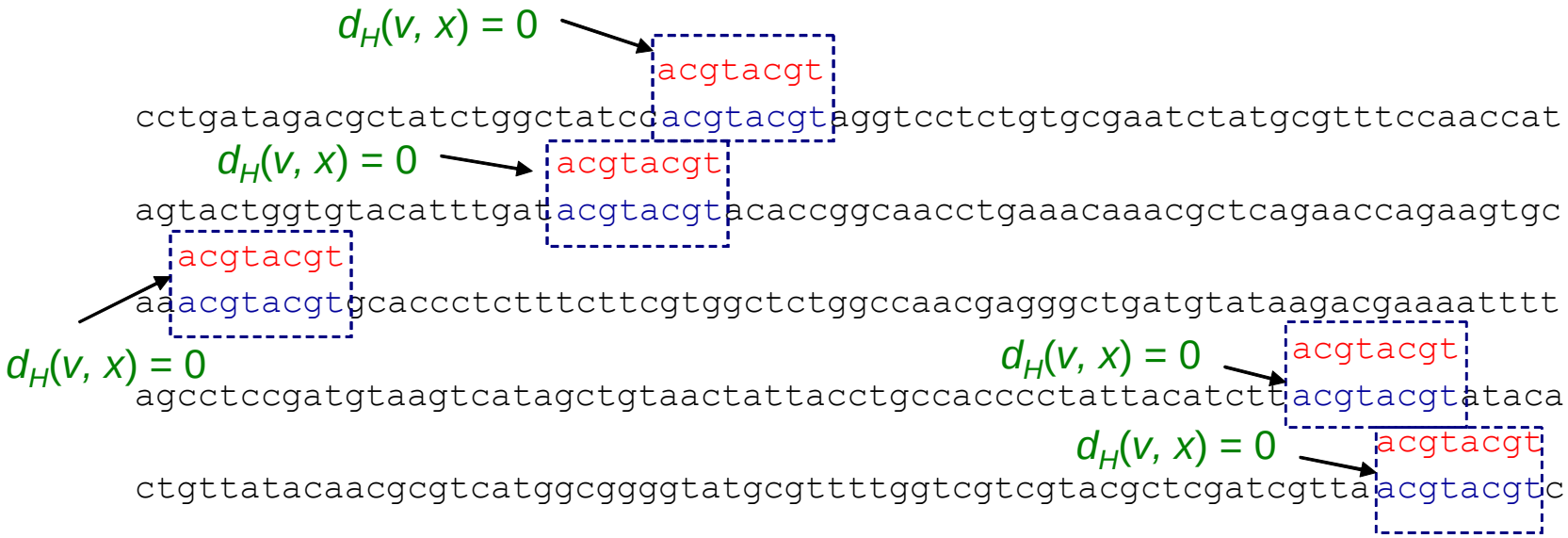
1. BruteForceMotifSearch(DNA, t, n, *l*)
2. bestScore ← 0
3. for each s = (s_1, s_2, …, s_t) from (1, 1, …, 1)
   to (n- *l*+1, n- *l*+1, …, n- *l*+1)
4. if score(s, DNA, *l*) > bestScore
5. bestScore ← score(s, DNA, *l*)
6. bestMotif ← (s_1, s_2, …, s_t)
7. return bestMotif

The Median String Problem

- Given a set of ***t*** DNA sequences find a pattern that appears in all ***t*** sequences with the minimum number of mutations
- This pattern will be the motif
- Rather than finding the maximal consensus string, this approach attempts to find the minimal distance string

Total Distance: An Example

- Given ***v*** = "acgtacgt" and **s**

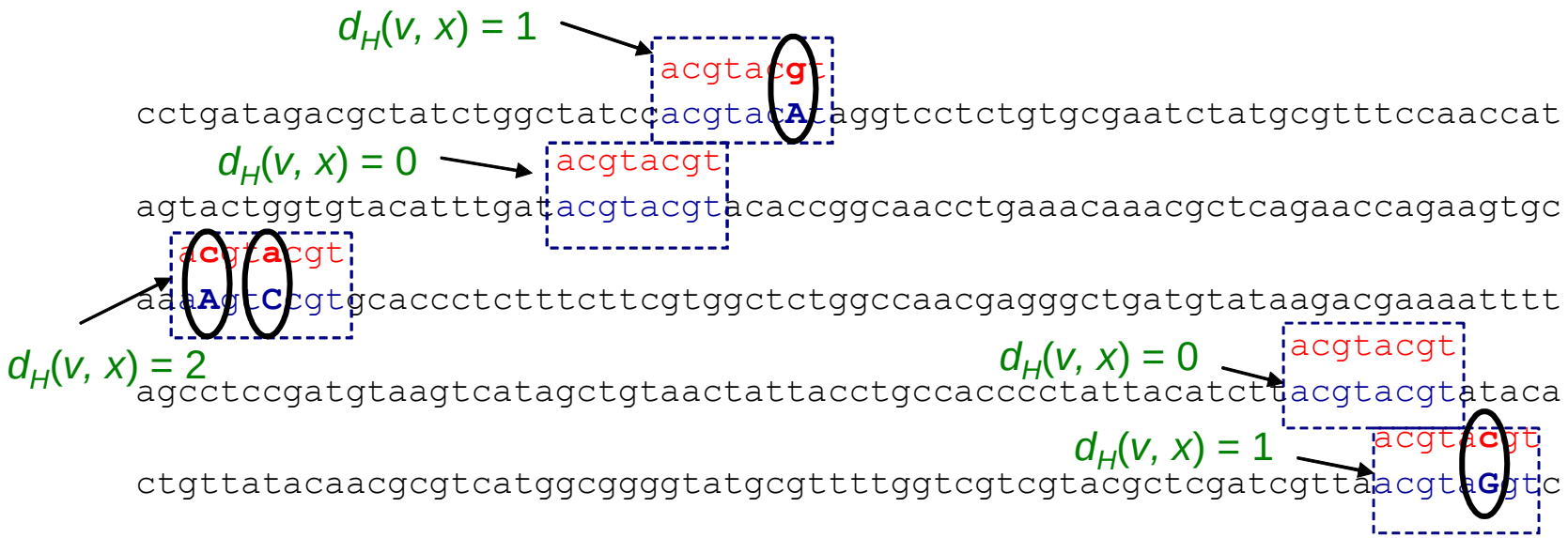


v is the sequence in red, *x* is the sequence in blue

- $TotalDistance(\mathbf{v}, \mathbf{DNA}) = 0$

Total Distance: An Example

- Given ***v*** = "acgtacgt" and **s**



v is the sequence in red, *x* is the sequence in blue

- $TotalDistance(\mathbf{v},\mathbf{DNA}) = 1+0+2+0+1 = 4$

Total Distance: Definition

- For each DNA sequence ***i***, compute all $d_H(\boldsymbol{v}, \boldsymbol{x})$, where ***x*** is an $\boldsymbol{\ell}$-mer with starting position s_i

  $(1 \leq s_i \leq \boldsymbol{n} - \boldsymbol{\ell} + 1)$

- Find minimum of $d_H(\boldsymbol{v}, \boldsymbol{x})$ among all $\boldsymbol{\ell}$-mers in sequence ***i***
- *TotalDistance(**v,DNA**)* is the sum of the minimum Hamming distances for each DNA sequence ***i***
- $TotalDistance(\boldsymbol{v},\boldsymbol{DNA}) = min_{\mathbf{s}}\, d_H(\boldsymbol{v}, \boldsymbol{s})$, where ***s*** is the set of starting positions $s_1, s_2, \ldots s_t$

The Median String Problem

- Goal: Given a set of DNA sequences, find a median string
- Input: A ***t*** x ***n*** matrix DNA, and ℓ, the length of the pattern to find
- Output: A string ***v*** of ℓ nucleotides that minimizes *TotalDistance(**v**,**DNA**)* over all strings of that length

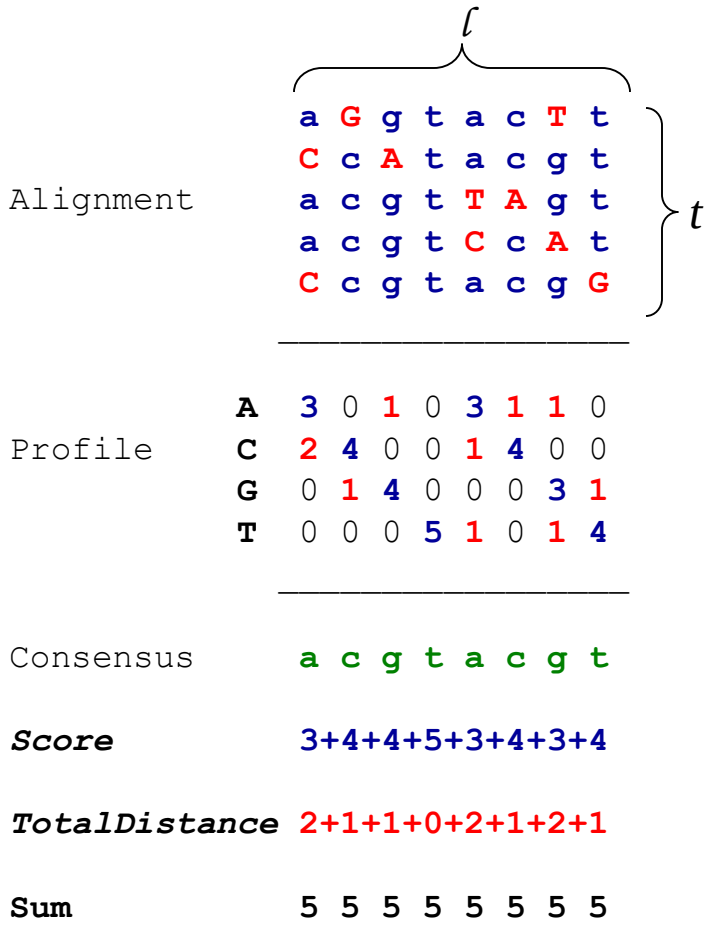
Median String Search Algorithm

1. MedianStringSearch(DNA, t, n, *l*)
2. bestMotif ← “”
3. bestDistance ← t × *l*
4. for each *l* –mer, s, from “aaa…a” to “ttt…t”
5. if TotalDistance(s, DNA) < bestDistance
6. bestDistance ← TotalDistance(s, DNA)
7. bestMotif ← s
8. return bestMotif

Equivalent Problems

- Motif Finding Problem ≡ Median String Problem
- The *Motif Finding* is a maximization problem while *Median String* is a minimization problem
- However, the *Motif Finding* problem and *Median String* problem are computationally equivalent
  (they give the same output for a common input)
- Need to show that minimizing *TotalDistance* is equivalent to maximizing *Score*

We're looking for the same thing



- At any column ***i***
  $Score_i + TotalDistance_i = \boldsymbol{t}$
- Because there are ℓ columns
  $Score + TotalDistance = \ell * \boldsymbol{t}$
- Rearranging:
  $Score = \ell * \boldsymbol{t} - TotalDistance$
- $\ell * \boldsymbol{t}$ is constant: the minimization of the right side is equivalent to the maximization of the left side

Why Bother?

- What is the point of reformulating the Motif Finding problem as the Median String problem?

Improving Motif Finding

1. BruteForceMotifSearch(DNA, t, n, *l*)
2. bestScore ← 0
3. for each s = (s_1, s_2, …, s_t) from (1, 1, …, 1)
   to (n- *l*+1, n- *l*+1, …, n- *l*+1)
4. if score(s, DNA, l) > bestScore
5. bestScore ← score(s, DNA, *l*)
6. bestMotif ← (s_1, s_2, …, s_t)
7. return bestMotif

How to Structure the Search?

- How can we perform the line

for each **s**=(s_1,s_2 , . . ., s_t) from (1,1 . . . 1) to (n–ℓ+1, . . ., n–ℓ+1) ?

- We need a method to more efficiently examine the many possible motifs locations
- This is not very different than exploring all "*t*-digit base *(n-l+1)*" numbers

Improving Median String

1. MedianStringSearch(DNA, t, n, *l*)
2. bestMotif ← “”
3. bestDistance ← t × *l*
4. for each *l* –mer, v, from “aaa…a” to “ttt…t”
5. if TotalDistance(v, DNA) < bestDistance
6. bestDistance ← TotalDistance(v, DNA)
7. bestMotif ← v
8. return bestMotif

How to Best Explore Permutations?

- For the Median String Problem we need to consider all 4^{ℓ} possible ℓ-mers:

aa… aa
aa… ac
aa… ag
aa… at
aa… ca
.
.
tt… tt

How to organize this search?

Simple Code

```
def NextLeaf(a, L, k):
    # generates L^k permutations
    for i in reversed(range(L)):
        if (a[i] < k):
            a[i] += 1
            break
        else:
            a[i] = 1
    return a
```

- Each call generates a new permutation

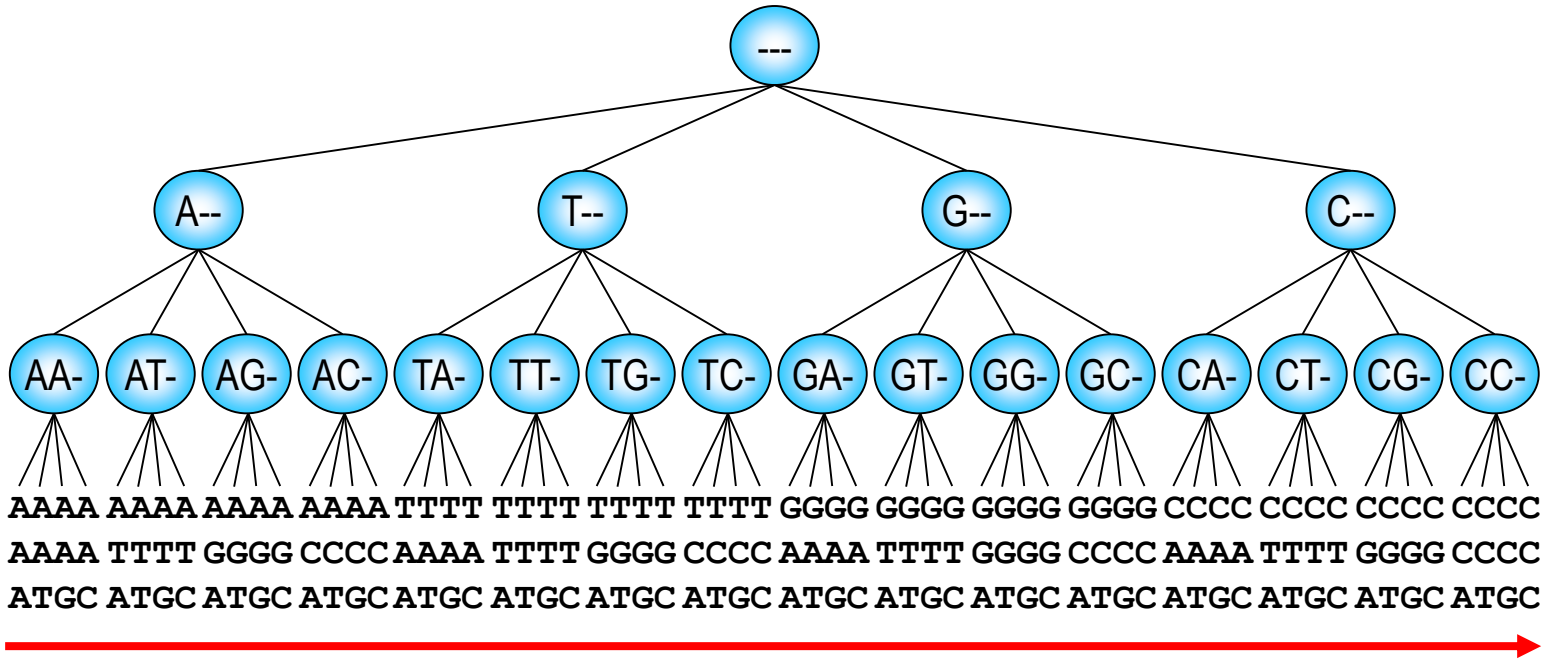
NextLeaf Usage

```
def AllLeaves(L, k):
    a = [1 for i in xrange(L)]
    while True:
        print a
        a = NextLeaf(a, L, k)
        if (sum(a) == L)
            return
```

- Is there another way to search permutations?

Search Tree

- Our standard method for enumerating permutations just traverses the leaf nodes
- Suppose after checking the first or second letter we already know the solution could not be the one we are looking for?

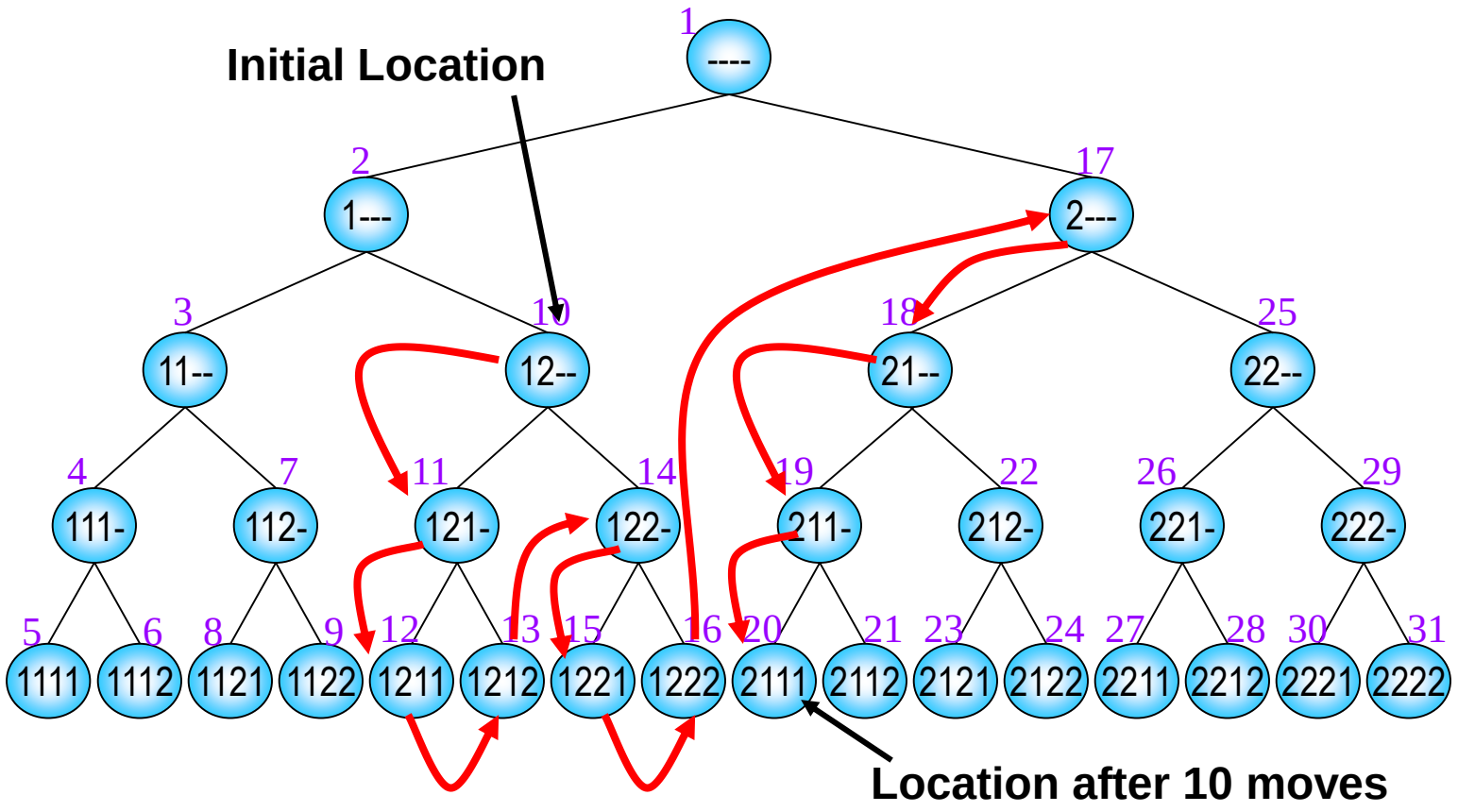


Analyzing Search Trees

- Characteristics of the search trees:
  - The unique permutations reside at leaves
  - A parent node is a common prefix of its children
- How can we traverse the tree?
- Things we'd like to do:
  - Visit all the nodes (interior and leaves)
  - Visit the next node (in an ordered way)
  - Bypass the children of a node

Depth First Search

- Start from the root and explore down to the bottom one path at a time



Visiting the Next Vertex

- Uses 0s to encode unspecified part of interior nodes (the dashes in our figure)

```
def NextVertex(a, i, L, k):
    if (i < L):
        a[i] = 1
        return (a, i+1)
    else:
        for j in reversed(range(L)):
            if (a[j] < k):
                a[j] += 1
                return (a, j+1)
            a[j] = 0
    return (a, 0)
```

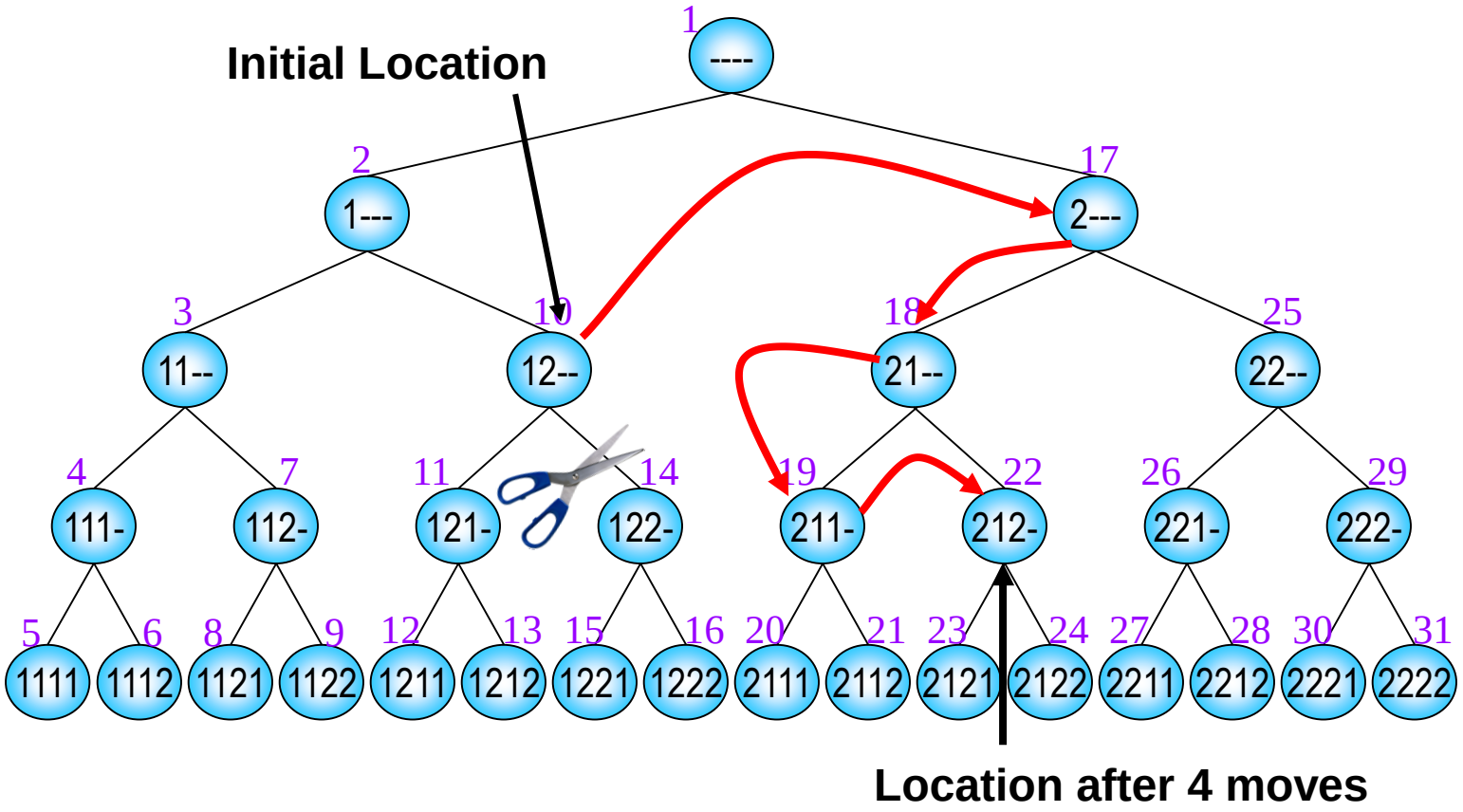
Bypass Nodes

- Given a prefix (internal vertex), find next vertex after skipping all of the current vertex's children

```
def Bypass(a, i, L, k):
    for j in reversed(range(i)):
        if (a[j] < k):
            a[j] += 1
            return (a, j+1)
        a[j] = 0
    return (a, 0)
```

Bypass Example

- Bypassing descendents of nodes “12—” and “211-”



Revisiting Brute Force Search

- Now that we have method for navigating the tree, let's convert our pseudocode version of BruteForceMotifSearch to real code

```
def BruteForceMotifSearchAgain(DNA,t,n,l):
    s = [1 for i in range(t)]
    bestScore = Score(s, DNA)
    while (True):
        s = NextLeaf(s,t,n-l+1)
        if (Score(s, DNA) > bestScore):
            bestScore = Score(s, DNA)
            bestMotif = [x for x in s]
        if (sum(s) == t):
            break
    return bestMotif
```

Can We Do Better?

- Sets of **s=**(s_1, s_2, …,s_t) may have a weak profile for the first *i* positions (s_1, s_2, …,s_i)
- Every row of alignment may add at most ℓ to Score
- Optimism: if all subsequent *(t-i)* positions (s_{i+1}, …s_t) add

$$(t - i) * \ell \text{ to Score}(\mathbf{s}, i, DNA)$$

- If Score(**s**,*i*,***DNA***) + $(t - i) * \ell$ < **BestScore**, it makes no sense to search subtrees of the current vertex
  - Use **ByPass()**

Rewrite Using Tree Traversal

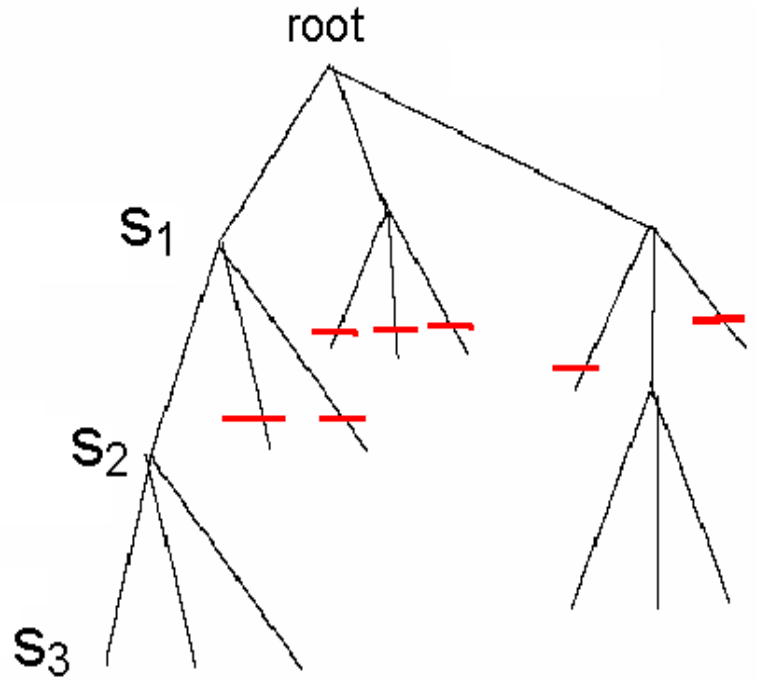
- Before we apply a branch-and-bound strategy let's rewrite the brute-force algorithm using a search tree

```
def SimpleMotifSearch(DNA,t,n,l):
    s = [0 for i in range(t)]
    bestScore = 0
    i = 0
    while (True):
        if (i < t):
            s, i = NextVertex(s,i,t,n-l+1)
        else:
            if (Score(s, DNA, l) > bestScore):
                bestScore = Score(s, DNA, l)
                bestMotif = [x for x in s]
            s, i = NextVertex(s,i,t,n-l+1)
            if (sum(s) == 0):
                break
    return bestMotif
```

Branch and Bound Motif Search

- Since each level of the tree goes deeper into search, discarding a prefix discards all following branches

- This saves us from looking at $(n - \ell + 1)^{t-i}$ leaves
  - Use **NextVertex()** and **ByPass()** to navigate the tree



Branch-and-Bound Motif Code

```
def BranchAndBoundMotifSearch(DNA,t,n,l):
    s = [0 for i in xrange(t)]
    bestScore = 0
    i = 0
    while (True):
        if (i < t):
            optimisticScore = Score(s, DNA, l) + (t-i)*l
            if (optimisticScore < bestScore):
                s, i = Bypass(s,i,t,n-l+1)
            else:
                s, i = NextVertex(s,i,t,n-l+1)
        else:
            score = Score(s, DNA, l)
            if (score > bestScore):
                bestScore = score
                bestMotif = [x for x in s]
            s, i = NextVertex(s,i,t,n-l+1)
        if (sum(s) == 0):
            break
    return bestMotif
```

Improving Median Search

- Recall the computational differences between motif search and median string search
  - The Motif Finding Problem needs to examine all $(\boldsymbol{n} - \boldsymbol{l} + 1)^t$ combinations for **s**.
  - The Median String Problem needs to examine 4^{ℓ} combinations of $\boldsymbol{v}$. This number is relatively small
- We want to use median string algorithm with the Branch and Bound trick!

Insight for Improving Median Search

- Note that if, at any point, the total distance for a prefix is greater than that for the best word so far:

  TotalDistance (*prefix*, *DNA*) > *BestDistance*

  there is no use exploring the remaining part of the word

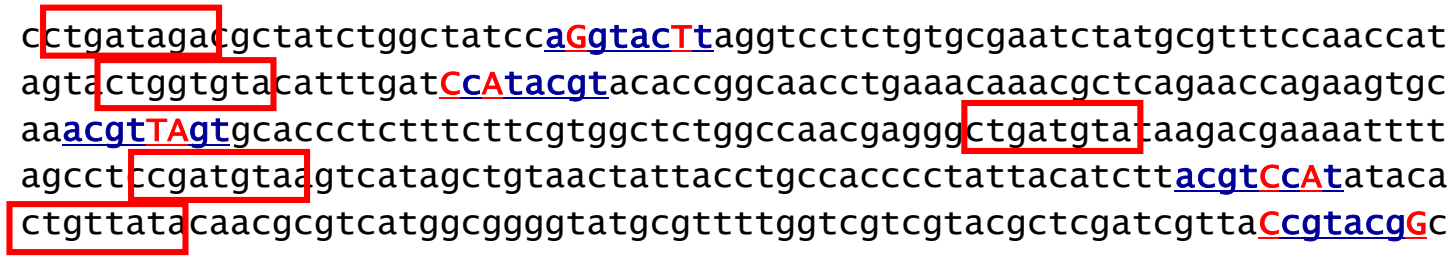
- We can eliminate that branch and BYPASS exploring that branch further

Bounded Median String Search

```
def BranchAndBoundMedianSearch(DNA,t,n,l):
    s = [1 for i in xrange(t)]
    bestDistance, bestWord = l*t, ''
    i = 1
    while (i > 0):
        if (i < l):
            prefix = NucleotideString(s, i)
            optimisticDistance = TotalDistance(prefix, DNA)
            if (optimisticDistance > bestDistance):
                s, i = Bypass(s,i,l,t)
            else:
                s, i = NextVertex(s,i,l,t)
        else:
            word = NucleotideString(s, l)
            if (TotalDistance(word, DNA) < bestDistance):
                bestDistance = TotalDistance(word, DNA)
                bestWord = word
            s, i = NextVertex(s,i,l,t)
    return bestWord
```

Today's Bad Example

- The example used in today's lecture was the best motif until we allowed the mutations!



- The target motif has a consensus score of 30
- But [2, 5, 46, 4, 1] = 31 and [2, 5, 46, 6, 1] = 34
- >30 solutions with consensus of 30 or better
- Which is the real Motif?

Greedy Algorithms

- **Def:** Algorithms that make locally optimal choices using a metric with the hope of finding a globally optimal solution.

- **Example**: Making change with US coins.

- **Optimization Problem**: Given an input, compute a solution, subject to various constraints, that either minimizes cost or maximizes profit.

Coin-Changing: Greedy Algorithm

Cashier's algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.

```
Sort coins denominations by value: c_1 < c_2 < … < c_n.

   ↙ coins selected

S ← ϕ
while (x ≠ 0) {
   let k be largest integer such that c_k ≤ x
   if (k = 0)
      return "no solution found"
   x ← x - c_k
   S ← S ∪ {k}
}
return S
```