CSE 421

RNA Secondary Structure, Sequence Alignment, Longest Path in a DAG

Shayan Oveis Gharan
Knapsack Problem: Running Time

Running time: $\Theta(n \cdot W)$

- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete.

Knapsack approximation algorithm:

There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum in time $\text{Poly}(n, \log W)$. 

1000000 items $W = 2^{1000}$. 

$w_1 \leq \cdots \leq w_n \leq W$ 
$w_1 \leq \cdots \leq w_n \leq V$ 
$W$ 

$n \cdot \log W$ bits 
$n \cdot \log V$ bits 
$\log W$ bits
DP Ideas so far

- You may have to define an ordering to decrease #subproblems

- You may have to strengthen DP, equivalently the induction, i.e., you have may have to carry more information to find the Optimum.

- This means that sometimes we may have to use two dimensional or three dimensional induction

- To think about DP try to characterize OPT.

- To write down your solution explicitly and carefully write OPT(i,..,i)
RNA Secondary Structure
RNA Secondary Structure

**RNA**: A String $B = b_1b_2\ldots b_n$ over alphabet $\{A, C, G, U\}$.

**Secondary structure.** RNA is single-stranded so it tends to loop back and form base pairs with itself. This structure is essential for understanding behavior of molecule.

**Ex:** GUCGAAUGAGGCAAUGUAACACGUGGCUACGGCGAGA

complementary base pairs: $A-U, C-G$
RNA Secondary Structure (Formal)

Secondary structure. A set of pairs $S = \{ (b_i, b_j) \}$ that satisfy:

[Watson-Crick.]

- $S$ is a matching and
- each pair in $S$ is a Watson-Crick pair: A-U, U-A, C-G, or G-C.

[No sharp turns.]: The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_j) \in S$, then $i < j - 4$.

[Non-crossing.]: If $(b_i, b_j)$ and $(b_k, b_l)$ are two pairs in $S$, then we cannot have $i < k < j < l$.

Free energy: Usual hypothesis is that an RNA molecule will maximize total free energy.

Goal: Given an RNA molecule $B = b_1b_2\ldots b_n$, find a secondary structure $S$ that maximizes the number of base pairs.

approximate by number of base pairs
Secondary Structure (Examples)

```
C-------------G
      |       |
      |       |
A-------------U
      |       |
U-------------A
```

base pair

```
G-------------G
      |       |
      |       |
C-------------C
```

```
G-------------G
      |       |
      |       |
A-------------G
```

```
C-------------G
      |       |
      |       |
A-------------U
      |       |
U-------------A
```

crossing

```
A U G U G G C C A U
```

```
A U G G G G C A U
```

```
A U G G G C C A U
```

```
A U G U G G C C A U
```

sharp turn

```
A U G U G G C C A U
```

```
A U G G G C C A U
```

```
A U G U G G C C A U
```

ok

```
A U G U G G C C A U
```

<4
First attempt. Let $OPT(n) = \text{maximum number of base pairs in a secondary structure of the substring } b_1b_2...b_n$.

Suppose $b_n$ is matched with $b_t$ in $OPT(n)$. What IH should we use?

Difficulty: This naturally reduces to two subproblems

- Finding secondary structure in $b_1, ..., b_{t-1}$, i.e., OPT(t-1)
- Finding secondary structure in $b_{t+1}, ..., b_{n-1}$
**Definition:** \( OPT(i, j) \) = maximum number of base pairs in a secondary structure of the substring \( b_i, b_{i+1}, ..., b_j \)

The most important part of a correct DP; It fixes IH

**Case 1:** If \( j - i \leq 4 \).
- \( OPT(i, j) = 0 \) by no-sharp turns condition.

**Case 2:** Base \( b_j \) is not involved in a pair.
- \( OPT(i, j) = OPT(i, j-1) \)

**Case 3:** Base \( b_j \) pairs with \( b_t \) for some \( i \leq t < j - 4 \)
- non-crossing constraint **decouples** resulting sub-problems
- \( OPT(i, j) = \max_{t: \text{pairs with } b_t} \{ 1 + OPT(i, t - 1) + OPT(t + 1, j - 1) \} \)
Let $M[i,j] = \text{empty}$ for all $i,j$.

Compute-OPT$(i,j)$

1. If $(j-i \leq 4)$
   
   return $0$;

2. If $(M[i,j] \text{ is empty})$
   
   $M[i,j] = \text{Compute-OPT}(i,j-1)$

   for $t = i$ to $j-5$
   
   if $(b_t,b_j \text{ is in \{} A-U, U-A, C-G, G-C \text{\}})$
     
     $M[i,j] = \max(M[i,j], 1 + \text{Compute-OPT}(i,t-1) + \text{Compute-OPT}(t+1,j-1))$

   return $M[j]$

Does this code terminate?
What are we inducting on?
Formal Induction

Let \( OPT(i, j) = \) maximum number of base pairs in a secondary structure of the substring \( b_i, b_{i+1}, ..., b_j \)

We induct on \(|i-j|\).

Base Case: \( OPT(i, j) = 0 \) for all \( i, j \) where \(|j - i| \leq 4\).

IH: For some \( \ell \geq 4 \), Suppose we have computed \( OPT(i, j) \) for all \( i, j \) where \(|i - j| \leq \ell\).

IS: Goal: We find \( OPT(i, j) \) for all \( i, j \) where \(|i - j| = \ell + 1\). Fix \( i, j \) such that \(|i - j| = \ell + 1\).

Case 1: Base \( b_j \) is not involved in a pair.

- \( OPT(i, j) = OPT(i, j-1) \) [this we know by IH since \(|i - (j - 1)| = \ell\)]

Case 2: Base \( b_j \) pairs with \( b_t \) for some \( i \leq t < j - 4 \)

- \( OPT(i, j) = \max_{t:b_i \text{ pairs with } b_t} \left\{ 1 + OPT(i, t - 1) + OPT(t + 1, j - 1) \right\} \)

We know by IH since difference \( \leq \ell\)
Bottom-up DP

\[
\begin{align*}
\text{for } k &= 1, 2, \ldots, n-1 \\
\text{for } i &= 1, 2, \ldots, n-1 \\
\text{\quad } j &= i + k \\
\text{\quad if } (j-i \leq 4) \\
\text{\quad \quad } M[i,j] &= 0; \\
\text{\quad else} \\
\text{\quad \quad } M[i,j] &= M[i,j-1] \\
\text{\quad \quad for } t = i \text{ to } j-5 \text{ do} \\
\text{\quad \quad \quad if } (b_t, b_j \text{ is in } \{A-U, U-A, C-G, G-C\}) \\
\text{\quad \quad \quad \quad } M[i,j] &= \max(M[i,j], 1 + M[i,t-1] + M[t+1,j-1])
\end{align*}
\]

return $M[1, n]$

Running Time: $O(n^3)$
Lesson

We may not always induct on $i$ or $w$ to get to smaller subproblems.

We may have to induct on $|i - j|$ or $i + j$ when we are dealing with more complex problems, e.g., intervals.
Sequence Alignment
Word Alignment

How similar are two strings?

ocurrance
occurrence

ococcurrence -

5 mismatches, 1 gap

oc - urrence
occurrence

1 mismatch, 1 gap

oc - urrence - ance
occurrence

0 mismatches, 3 gaps
Edit Distance

Cost = # of gaps + #mismatches.

Applications.
• Basis for Unix diff and Word correct in editors.
• Speech recognition.
• Computational biology.

Cost: 5

Cost: 3
Sequence Alignment

Given two strings \(x_1, \ldots, x_m\) and \(y_1, \ldots, y_n\) find an alignment with minimum number of mismatch and gaps.

An alignment is a set of ordered pairs \((x_{i_1}, y_{j_1}), (x_{i_2}, y_{j_2}), \ldots\) such that \(i_1 < i_2 < \ldots\) and \(j_1 < j_2 < \ldots\)

**Example:** CTACCG vs. TACATG.

Sol: We aligned \(x_2\)-\(y_1\), \(x_3\)-\(y_2\), \(x_4\)-\(y_3\), \(x_5\)-\(y_4\), \(x_6\)-\(y_6\).

So, the cost is 3.
DP for Sequence Alignment

Let $OPT(i, j)$ be min cost of aligning $x_1, ..., x_i$ and $y_1, ..., y_j$

Case 1: OPT matches $x_i, y_j$
- Then, pay mis-match cost if $x_i \neq y_j + \min \text{cost of aligning } x_1, ..., x_{i-1} \text{ and } y_1, ..., y_{j-1}$ i.e., $OPT(i - 1, j - 1)$

Case 2: OPT leaves $x_i$ unmatched
- Then, pay gap cost for $x_i + OPT(i - 1, j)$

Case 3: OPT leaves $y_j$ unmatched
- Then, pay gap cost for $y_j + OPT(i, j - 1)$
**Sequence-Alignment**($m, n, x_1 x_2 \ldots x_m, y_1 y_2 \ldots y_n$) {
  
  for $i = 0$ to $m$
  
  $M[0, i] = i$

  for $j = 0$ to $n$
  
  $M[j, 0] = j$

  for $i = 1$ to $m$
  
  for $j = 1$ to $n$
  
  $M[i, j] = \min(\ (x_i=y_j \ ? \ 0:1) + M[i-1, j-1],$
  
  $\ 1 + M[i-1, j],$
  
  $\ 1 + M[i, j-1])$

  return $M[m, n]$
}

**Analysis:** $\Theta(mn)$ time and space.

**English words or sentences:** $m, n \leq 10, \ldots, 20$.

**Computational biology:** $m = n = 100,000$. 10 billions ops OK, but 40GB array?
If we are not using strong induction in the DP, we just need to use the last (row) of computed values.

```plaintext
Sequence-Alignment(m, n, x1x2...x_m, y1y2...y_n) {
    for i = 0 to m
        M[0, i] = i
    for j = 0 to n
        M[j, 0] = j
    for i = 1 to m
        for j = 1 to n
            M[i, j] = min( (x_i=y_j ? 0:1) + M[i-1, j-1],
                            1 + M[i-1, j],
                            1 + M[i, j-1])
    return M[m, n]
}
```

Just need \( i - 1 \) rows to compute \( M[i,j] \)
DP with $O(m + n)$ memory

- Keep an Old array containing values of the last row
- Fill out the new values in a New array
- Copy new to old at the end of the loop

```c
Sequence-Alignment(m, n, x_1x_2...x_m, y_1y_2...y_n) {
    for i = 0 to m
        O[i] = i
    for i = 1 to m
        N[0]=i
        for j = 1 to n
            N[j] = min( (x_i=y_j ? 0:1) + O[j-1],
                        1 + O[j],
                        1 + N[j-1])
    for j = 1 to n
        O[j]=N[j]
    return N[n]
}
```
Lesson

Advantage of a bottom-up DP:

It is much easier to optimize the space.
Longest Path in a DAG
Longest Path in a DAG

**Goal:** Given a DAG G, find the longest path.

**Recall:** A directed graph G is a DAG if it has no cycle.

This problem is NP-hard for general directed graphs:
- It has the Hamiltonian Path as a special case.
Q: What is the right ordering?
Remember, we have to use that G is a DAG, ideally in defining the ordering.

We saw that every DAG has a topological sorting
So, let’s use that as an ordering.
Suppose we have labelled the vertices such that \((i,j)\) is a directed edge only if \(i < j\).

Let \(OPT(j)\) = length of the longest path ending at \(j\)

Suppose in the longest path ending at \(j\), last edge is \((i, j)\). Then, none of the \(i + 1, \ldots, j - 1\) are in this path since topological ordering. So,

\[ OPT(j) = OPT(i) + 1. \]
Suppose we have labelled the vertices such that \((i, j)\) is a directed edge only if \(i < j\).

Let \(OPT(j)\) = length of the longest path ending at \(j\)

\[
OPT(j) = \begin{cases} 
0 & \text{If } j \text{ is a source} \\
1 + \max_{i : (i, j) \text{ an edge}} OPT(i) & \text{o.w.}
\end{cases}
\]
Let G be a DAG given with a topological sorting: For all edges \((i,j)\) we have \(i < j\).

\[
\text{Compute-OPT}(j)\{
    \text{if (in-degree}(j)==0) return 0
    \text{if (M}[j]==\text{empty})
        \text{M}[j]=0;
    \text{for all edges } (i,j)
        \text{M}[j] = \text{max}(\text{M}[j],1+\text{Compute-OPT}(i))
    \text{return } \text{M}[j]
\}
\]

Output \(\text{max}(\text{M}[1],\ldots,\text{M}[n])\)

**Running Time**: \(O(n + m)\)

**Memory**: \(O(n)\)

Can we output the longest path?
Let G be a DAG given with a topological sorting: For all edges (i,j) we have i<j.

Initialize Parent[j]=-1 for all j.

Compute-OPT(j){
  if (in-degree(j)==0)
    return 0
  if (M[j]==empty)
    M[j]=0;
  for all edges (i,j)
    if (M[j] < 1+Compute-OPT(i))
      M[j]=1+Compute-OPT(i)
      Parent[j]=i
  return M[j]
}

Let M[k] be the maximum of M[1],...,M[n]

While (Parent[k]!=-1)
  Print k
  k=Parent[k]
DP Techniques Summary

Recipe:
• Follow the natural induction proof.
• Find out additional assumptions/variables/subproblems that you need to do the induction
• Strengthen the hypothesis and define w.r.t. new subproblems

Dynamic programming techniques.
• Whenever a problem is a special case of an NP-hard problem an ordering is important:
  • Adding a new variable: knapsack.
  • Dynamic programming over intervals: RNA secondary structure.

Top-down vs. bottom-up:
• Different people have different intuitions
• Bottom-up is useful to optimize the memory