CSE 421: Introduction to Algorithms

Dynamic Programming

Paul Beame
Dynamic Programming

- Dynamic Programming
  - Give a solution of a problem using smaller sub-problems where the parameters of all the possible sub-problems are determined in advance
  - Useful when the same sub-problems show up again and again in the solution
A simple case: Computing Fibonacci Numbers

- Recall $F_n = F_{n-1} + F_{n-2}$ and $F_0 = 0$, $F_1 = 1$

- Recursive algorithm:
  
  Fibo(n)
  
  if $n=0$ then return(0)
  
  else if $n=1$ then return(1)
  
  else return(Fibo(n-1) + Fibo(n-2))
Call tree - start

```
F (6)
  F (5)
    F (4)
      F (3)
        F (2)
          F (1)
            1 0
```

```
F (6)
  F (5)
    F (4)
      F (3)
        F (2)
          F (1)
            1 0
```
Memoization (Caching)

- Remember all values from previous recursive calls
- Before recursive call, test to see if value has already been computed

Dynamic Programming
- Convert memoized algorithm from a recursive one to an iterative one
Fibonacci Dynamic Programming Version

- FiboDP(n):
  - F[0] ← 0
  - F[1] ← 1
  - for i=2 to n do
    - F[i] ← F[i-1] + F[i-2]
  - endfor
  - return(F[n])
Fibonacci: Space-Saving Dynamic Programming

FiboDP(n):
    prev ← 0
    curr ← 1
    for i = 2 to n do
        temp ← curr
        curr ← curr + prev
        prev ← temp
    endfor
return(curr)
Dynamic Programming

- Useful when
  - same recursive sub-problems occur repeatedly
  - Can anticipate the parameters of these recursive calls
  - The solution to whole problem can be figured out with knowing the internal details of how the sub-problems are solved
  
  - principle of optimality
    “Optimal solutions to the sub-problems suffice for optimal solution to the whole problem”
Three Steps to Dynamic Programming

- Formulate the answer as a recurrence relation or recursive algorithm

- Show that the number of different values of parameters in the recursive calls is “small”
  - e.g., bounded by a low-degree polynomial
  - Can use memoization

- Specify an order of evaluation for the recurrence so that you already have the partial results ready when you need them.
Weighted Interval Scheduling

- Same problem as interval scheduling except that each request $i$ also has an associated value or weight $w_i$
  - $w_i$ might be
    - amount of money we get from renting out the resource for that time period
    - amount of time the resource is being used $w_i = f_i - s_i$
  - Goal: Find compatible subset $S$ of requests with maximum total weight
Greedy Algorithms for Weighted Interval Scheduling?

- No criterion seems to work
  - Earliest start time $s_i$
    - Doesn’t work
  - Shortest request time $f_i - s_i$
    - Doesn’t work
  - Fewest conflicts
    - Doesn’t work
  - Earliest finish time $f_i$
    - Doesn’t work
  - Largest weight $w_i$
    - Doesn’t work
Towards Dynamic Programming: Step 1 – A Recursive Algorithm

- Suppose that like ordinary interval scheduling we have first sorted the requests by finish time $f_i$ so $f_1 \leq f_2 \leq \ldots \leq f_n$
- Say request $i$ comes before request $j$ if $i < j$
- For any request $j$ let $p(j)$ be
  - the largest-numbered request before $j$ that is compatible with $j$
  - or 0 if no such request exists
- Therefore $\{1, \ldots, p(j)\}$ is precisely the set of requests before $j$ that are compatible with $j$
Towards Dynamic Programming: Step 1 – A Recursive Algorithm

- Two cases depending on whether an optimal solution $O$ includes request $n$
  - If it **does** include request $n$ then all other requests in $O$ must be contained in $\{1, \ldots, p(n)\}$
    - Not only that!
      - Any set of requests in $\{1, \ldots, p(n)\}$ will be compatible with request $n$
      - So in this case the optimal solution $O$ must contain an optimal solution for $\{1, \ldots, p(n)\}$
    - “Principle of Optimality”
Towards Dynamic Programming: Step 1 – A Recursive Algorithm

- Two cases depending on whether an optimal solution $O$ includes request $n$
  - If it does not include request $n$ then all requests in $O$ must be contained in $\{1, \ldots, n-1\}$
  - Not only that!
    - The optimal solution $O$ must contain an optimal solution for $\{1, \ldots, n-1\}$
    - “Principle of Optimality”
Towards Dynamic Programming: Step 1 – A Recursive Algorithm

- All subproblems involve requests \{1, \ldots, i\} for some \(i\)

- For \(i=1, \ldots, n\) let \(\text{OPT}(i)\) be the weight of the optimal solution to the problem \{1, \ldots, i\}

- The two cases give
  \[
  \text{OPT}(n) = \max [w_n + \text{OPT}(p(n)), \text{OPT}(n-1)]
  \]

- Also
  - \(n \in O\) iff \(w_n + \text{OPT}(p(n)) > \text{OPT}(n-1)\)
Towards Dynamic Programming: Step 1 – A Recursive Algorithm

- Sort requests and compute array $p[i]$ for each $i=1,...,n$

ComputeOpt($n$)
  
  if $n=0$ then return(0)

else
  
  $u \leftarrow$ ComputeOpt($p[n]$)
  
  $v \leftarrow$ ComputeOpt($n-1$)
  
  if $w_n + u > v$ then return($w_n + u$)
  
  else return($v$)

endif
Towards Dynamic Programming: Step 2 – Small # of parameters

- \text{ComputeOpt}(n)\ can\ take\ exponential\ time\ in\ the\ worst\ case
  - \(2^n\)\ calls\ if\ \(p(i) = i-1\)\ for\ every\ \(i\)

- There are only \(n\)\ possible\ parameters\ to\ \text{ComputeOpt}

- Store these answers in an array \(\text{OPT}[n]\)\ and only recompute when necessary
  - Memoization

- Initialize \(\text{OPT}[i] = 0\)\ for\ \(i = 1, \ldots, n\)
Dynamic Programming: Step 2 – Memoization

ComputeOpt(n)
if n=0 then return(0)
else
   u ← MComputeOpt(p[n])
   v ← MComputeOpt(n-1)
   if w_n + u > v then
      return(w_n + u)
   else return(v)
endif

MComputeOpt(n)
if OPT[n] = 0 then
   v ← ComputeOpt(n)
   OPT[n] ← v
return(v)
else
   return(OPT[n])
endif
Dynamic Programming Step 3: Iterative Solution

- The recursive calls for parameter $n$ have parameter values $i$ that are $< n$

IterativeComputeOpt($n$)

array $OPT[0..n]$  
$OPT[0] \leftarrow 0$

for $i=1$ to $n$

  if $w_i + OPT[p[i]] > OPT[i-1]$ then
    $OPT[i] \leftarrow w_i + OPT[p[i]]$
  else
    $OPT[i] \leftarrow OPT[i-1]$
  endif

endfor
Producing the Solution

IterativeComputeOptSolution(n)
array OPT[0..n], Used[1..n]
OPT[0] ← 0
for i = 1 to n
    if \( w_i + OPT[p[i]] > OPT[i-1] \) then
        OPT[i] ← \( w_i + OPT[p[i]] \)
        Used[i] ← 1
    else
        OPT[i] ← OPT[i-1]
        Used[i] ← 0
    endif
endfor

i ← n
S ← \( \emptyset \)
while i > 0 do
    if Used[i] = 1 then
        S ← S \( \cup \) \{i\}
        i ← p[i]
    else
        i ← i - 1
    endif
endwhile
**Example**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td>15</td>
<td>11</td>
<td>12</td>
<td>18</td>
</tr>
<tr>
<td>$f_i$</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>13</td>
<td>14</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>$w_i$</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>

**p[i]**

**OPT[i]**

**Used[i]**
## Example

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td>15</td>
<td>11</td>
<td>12</td>
<td>18</td>
</tr>
<tr>
<td>$f_i$</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>13</td>
<td>14</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>$w_i$</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>$p[i]$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>OPT[i]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Used[i]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>s_i</strong></td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td>15</td>
<td>11</td>
<td>12</td>
<td>18</td>
</tr>
<tr>
<td><strong>f_i</strong></td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>13</td>
<td>14</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td><strong>w_i</strong></td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td><strong>p[i]</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td><strong>OPT[i]</strong></td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>14</td>
<td>16</td>
</tr>
<tr>
<td><strong>Used[i]</strong></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td>15</td>
<td>11</td>
<td>12</td>
<td>18</td>
</tr>
<tr>
<td>$f_i$</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>13</td>
<td>14</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>$w_i$</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>$p[i]$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>OPT[i]</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>14</td>
<td>16</td>
</tr>
<tr>
<td>Used[i]</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$S=\{9,7,2\}$
Segmented Least Squares

Least Squares

Given a set \( P \) of \( n \) points in the plane \( p_1=(x_1,y_1),...,p_n=(x_n,y_n) \) with \( x_1<...<x_n \) determine a line \( L \) given by \( y=ax+b \) that optimizes the total ‘squared error’

\[
\text{Error}(L,P)=\sum_i(y_i-ax_i-b)^2
\]

A classic problem in statistics

Optimal solution is known (see text)

Call this line(\( P \)) and its error \( \text{error}(P) \)
Least Squares
Segmented Least Squares

What if data seems to follow a piece-wise linear model?
Segmented Least Squares
Segmented Least Squares
Segmented Least Squares

- What if data seems to follow a piece-wise linear model?
- Number of pieces to choose is not obvious
- If we chose $n-1$ pieces we could fit with 0 error
  - Not fair
- Add a penalty of $C$ times the number of pieces to the error to get a total penalty
- How do we compute a solution with the smallest possible total penalty?
Recursive idea

If we knew the point \( p_j \) where the last line segment began then we could solve the problem optimally for points \( p_1, \ldots, p_j \) and combine that with the last segment to get a global optimal solution.

- Let \( \text{OPT}(i) \) be the optimal penalty for points \( \{p_1, \ldots, p_i\} \).
- Total penalty for this solution would be
  \[
  \text{Error}(\{p_j, \ldots, p_n\}) + C + \text{OPT}(j-1)
  \]
Segmented Least Squares
Segmented Least Squares

- Recursive idea
  - We don’t know which point is $p_j$
  - But we do know that $1 \leq j \leq n$
  - The optimal choice will simply be the best among these possibilities
- Therefore

$$OPT(n) = \min_{1 \leq j \leq n} \{ \text{Error}(\{p_j, \ldots, p_n\}) + C + OPT(j-1) \}$$
Dynamic Programming Solution

SegmentedLeastSquares(n)
array $OPT[0..n]$, $Begin[1..n]$
$OPT[0] ← 0$
for $i = 1$ to $n$
  $OPT[i] ← Error\{(p_1,\ldots,p_i)\} + C$
  $Begin[i] ← 1$
  for $j = 2$ to $i-1$
    $e ← Error\{(p_j,\ldots,p_i)\} + C + OPT[j-1]$
    if $e < OPT[i]$ then
      $OPT[i] ← e$
      $Begin[i] ← j$
    endif
  endfor
endfor
return($OPT[n]$)

FindSegments
$i ← n$
$S ← \emptyset$
while $i > 1$ do
  compute Line($\{p_{Begin[i]},\ldots,p_i\}$)
  output $(p_{Begin[i]}, p_i)$, Line
  $i ← Begin[i]$
endwhile
Knapsack (Subset-Sum) Problem

- **Given:**
  - integer $W$ (knapsack size)
  - $n$ object sizes $x_1, x_2, \ldots, x_n$

- **Find:**
  - Subset $S$ of $\{1, \ldots, n\}$ such that $\sum_{i \in S} x_i \leq W$
    - but $\sum_{i \in S} x_i$ is as large as possible
Recursive Algorithm

- Let $K(n, W)$ denote the problem to solve for $W$ and $x_1, x_2, \ldots, x_n$
- For $n > 0$,
  - The optimal solution for $K(n, W)$ is the better of the optimal solution for either $K(n-1, W)$ or $x_n + K(n-1, W-x_n)$
- For $n = 0$
  - $K(0, W)$ has a trivial solution of an empty set $S$ with weight 0
Recursive calls

- Recursive calls on list ..., 3, 4, 7
Common Sub-problems

- Only sub-problems are $K(i,w)$ for
  - $i = 0,1,..., n$
  - $w = 0,1,..., W$

- Dynamic programming solution
  - Table entry for each $K(i,w)$
    - **OPT** - value of optimal soln for first $i$ objects and weight $w$
    - **belong** flag - is $x_i$ a part of this solution?
  - Initialize $OPT[0,w]$ for $w=0,...,W$
  - Compute all $OPT[i,\ast]$ from $OPT[i-1,\ast]$ for $i>0$
Dynamic Knapsack Algorithm

for $w = 0$ to $W$; $OPT[0, w] \leftarrow 0$; end for
for $i = 1$ to $n$ do
  for $w = 0$ to $W$ do
    $OPT[i, w] \leftarrow OPT[i-1, w]$
    $belong[i, w] \leftarrow 0$
    if $w \geq x_i$ then
      $val \leftarrow x_i + OPT[i-1, w-x_i]$
      if $val > OPT[i, w]$ then
        $OPT[i, w] \leftarrow val$
        $belong[i, w] \leftarrow 1$
    end if
  end for
end for
return ($OPT[n, W]$)

Time $O(nW)$
Sample execution on 2, 3, 4, 7 with K=15
Saving Space

- To compute the value $OPT$ of the solution only need to keep the last two rows of $OPT$ at each step.

- What about determining the set $S$?
  - Follow the $\text{belong}$ flags $O(n)$ time.
  - What about space?
Three Steps to Dynamic Programming

- Formulate the answer as a recurrence relation or recursive algorithm

- Show that the number of different values of parameters in the recursive algorithm is “small”
  - e.g., bounded by a low-degree polynomial

- Specify an order of evaluation for the recurrence so that you already have the partial results ready when you need them.
RNA Secondary Structure: Dynamic Programming on Intervals

- RNA: sequence of bases
  - String over alphabet \{A, C, G, U\}

- RNA folds and sticks to itself like a zipper
  - A bonds to U
  - C bonds to G
  - Bends can’t be sharp
  - No twisting or criss-crossing

- How the bonds line up is called the RNA secondary structure
ACGAUACUGCAACACUCUCUGUGACGAACCCAGCCAGGUGUAA
Another view of RNA Secondary Structure

A---C---A---U---C---U---G---U---G---A---C---G---A---U---G---U---A

No crossing
RNA Secondary Structure

- **Input:** String $x_1 \ldots x_n \in \{A,C,G,U\}^*$
- **Output:** Maximum size set $S$ of pairs $(i,j)$ such that
  - $\{x_i, x_j\} = \{A, U\}$ or $\{x_i, x_j\} = \{C, G\}$
  - The pairs in $S$ form a matching
  - $i < j - 4$ (no sharp bends)
  - No crossing pairs
    - If $(i,j)$ and $(k,l)$ are in $S$ then it is not the case that they cross as in $i < k < j < l$
Recursion Solution

- Try all possible matches for the last base

\[
\text{OPT}(1..j) = \text{MAX}(\text{OPT}(1..j-1), 1 + \text{MAX}_{k=1..j-5} (\text{OPT}(1..k-1) + \text{OPT}(k+1..j-1)))
\]

General form:

\[
\text{OPT}(i..j) = \text{MAX}(\text{OPT}(i..j-1), 1 + \text{MAX}_{k=i..j-5} (\text{OPT}(i..k-1) + \text{OPT}(k+1..j-1)))
\]
RNA Secondary Structure

- 2D Array $\text{OPT}(i,j)$ for $i \leq j$ represents optimal # of matches entirely for segment $i..j$
- For $j-i \leq 4$ set $\text{OPT}(i,j)=0$ (no sharp bends)
- Then compute $\text{OPT}(i,j)$ values when $j-i=5,6,...,n-1$ in turn using recurrence.
- Return $\text{OPT}(1,n)$
- Total of $O(n^3)$ time
- Can also record matches along the way to produce $S$
  - Similar polynomial-time algorithm for other problems
    - Context-Free Language recognition
    - Optimal matrix products, etc.
  - All use dynamic programming over intervals
Sequence Alignment: Edit Distance

- **Given:**
  - Two strings of characters $A= a_1, a_2, ..., a_n$ and $B= b_1, b_2, ..., b_m$

- **Find:**
  - The minimum number of edit steps needed to transform $A$ into $B$ where an edit can be:
    - insert a single character
    - delete a single character
    - substitute one character by another
Sequence Alignment vs Edit Distance

- **Sequence Alignment**
  - Insert corresponds to aligning with a “–” in the first string
    - Cost $\delta$ (in our case 1)
  - Delete corresponds to aligning with a “–” in the second string
    - Cost $\delta$ (in our case 1)
  - Replacement of an $a$ by a $b$ corresponds to a mismatch
    - Cost $\alpha_{ab}$ (in our case 1 if $a\neq b$ and 0 if $a=b$)

- In Computational Biology this alignment algorithm is attributed to Smith & Waterman
Applications

- "diff" utility – where do two files differ
- Version control & patch distribution – save/send only changes
- Molecular biology
  - Similar sequences often have similar origin and function
  - Similarity often recognizable despite millions or billions of years of evolutionary divergence
Growth of GenBank

- Sequences (millions)
- Base Pairs of DNA (millions)

Years: 1982 to 2000

Legend:
- Blue: Base Pairs
- Red: Sequences
Recursive Solution

- **Sub-problems**: Edit distance problems for all prefixes of A and B that don’t include all of both A and B

- Let $D(i,j)$ be the number of edits required to transform $a_1 a_2 \ldots a_i$ into $b_1 b_2 \ldots b_j$

- Clearly $D(0,0)=0$
Computing $D(n,m)$

- Imagine how best sequence handles the last characters $a_n$ and $b_m$
- If best sequence of operations
  - deletes $a_n$ then $D(n,m) = D(n-1,m) + 1$
  - inserts $b_m$ then $D(n,m) = D(n,m-1) + 1$
  - replaces $a_n$ by $b_m$ then
    $$D(n,m) = D(n-1,m-1) + 1$$
  - matches $a_n$ and $b_m$ then
    $$D(n,m) = D(n-1,m-1)$$
Recursive algorithm $D(n,m)$

if $n=0$ then
    return $(m)$
elseif $m=0$ then
    return $(n)$
else
    if $a_n=b_m$ then
        replace-cost $\leftarrow 0$
    else
        replace-cost $\leftarrow 1$
    endif
    return $(\min\{ D(n-1, m) + 1, D(n, m-1) +1, D(n-1, m-1) + \text{replace-cost} \})$
for $j = 0$ to $m$; $D(0,j) \leftarrow j$; endfor
for $i = 1$ to $n$; $D(i,0) \leftarrow i$; endfor
for $i = 1$ to $n$
for $j = 1$ to $m$
    if $a_i = b_j$ then
        replace-cost $\leftarrow 0$
    else
        replace-cost $\leftarrow 1$
    endif
endfor
$D(i,j) \leftarrow \min \{ D(i-1, j) + 1, D(i, j-1) + 1, D(i-1, j-1) + \text{replace-cost} \}$
endfor
endfor
Example run with AGACATTG and GAGTTA
Example run with AGACATTG and GAGTTA

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>C</th>
<th>A</th>
<th>T</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example run with AGACATTG and GAGTTA
Example run with AGACATTG and GAGTTA

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>C</th>
<th>A</th>
<th>T</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example run with AGACATTG and GAGTTA

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>C</th>
<th>A</th>
<th>T</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Example run with AGACATTG and GAGTTA

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>C</th>
<th>A</th>
<th>T</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Example run with AGACATTG and GAGTTA

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>C</th>
<th>A</th>
<th>T</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Reading off the operations

- Follow the sequence and use each color of arrow to tell you what operation was performed.
- From the operations can derive an optimal alignment

A G A C A T T G
_ G A G _ T T A
Saving Space

- To compute the distance values we only need the last two rows (or columns)
  - $O(\min(m,n))$ space
- To compute the alignment/sequence of operations
  - seem to need to store all $O(mn)$ pointers/arrow colors
- Nifty divide and conquer variant that allows one to do this in $O(\min(m,n))$ space and retain $O(mn)$ time
  - In practice the algorithm is usually run on smaller chunks of a large string, e.g. $m$ and $n$ are lengths of genes so a few thousand characters
    - Researchers want all alignments that are close to optimal
    - Basic algorithm is run since the whole table of pointers (2 bits each) will fit in RAM
  - Ideas are neat, though
Saving space

- Alignment corresponds to a path through the table from lower right to upper left
  - Must pass through the middle column
- Recursively compute the entries for the middle column from the left
  - If we knew the cost of completing each then we could figure out where the path crossed
- Problem
  - There are $n$ possible strings to start from.
- Solution
  - Recursively calculate the right half costs for each entry in this column using alignments starting at the other ends of the two input strings!
  - Can reuse the storage on the left when solving the right hand problem
Shortest paths with negative cost edges (Bellman-Ford)

- Dijsktra’s algorithm failed with negative-cost edges
  - What can we do in this case?
  - Negative-cost cycles could result in shortest paths with length $-\infty$

- Suppose no negative-cost cycles in G
  - Shortest path from $s$ to $t$ has at most $n-1$ edges
    - If not, there would be a repeated vertex which would create a cycle that could be removed since cycle can’t have –ve cost

68
Shortest paths with negative cost edges (Bellman-Ford)

- We want to grow paths from $s$ to $t$ based on the # of edges in the path.
- Let $\text{Cost}(s,t,i)=$cost of minimum-length path from $s$ to $t$ using up to $i$ hops.
  - $\text{Cost}(v,t,0)=\begin{cases}0 \text{ if } v=t \\ \infty \text{ otherwise} \end{cases}$
  - $\text{Cost}(v,t,i)=\min\{\text{Cost}(v,t,i-1), \min_{(v,w) \in E}(c_{vw}+\text{Cost}(w,t,i-1))\}$
Bellman-Ford

- Observe that the recursion for $\text{Cost}(s,t,i)$ doesn’t change $t$
  - Only store an entry for each $v$ and $i$
  - Termed $\text{OPT}(v,i)$ in the text
- Also observe that to compute $\text{OPT}(*,i)$ we only need $\text{OPT}(*,i-1)$
  - Can store a current and previous copy in $O(n)$ space.
Bellman-Ford

ShortestPath(G,s,t)

for all \( v \in V \)

\( OPT[v] \leftarrow \infty \)

\( OPT[t] \leftarrow 0 \)

for \( i=1 \) to \( n-1 \) do

for all \( v \in V \) do

\( OPT'[v] \leftarrow \min_{(v,w) \in E} (c_{vw} + OPT[w]) \)

for all \( v \in V \) do

\( OPT[v] \leftarrow \min(OPT'[v], OPT[v]) \)

return \( OPT[s] \)

\( O(mn) \) time
Negative cycles

- **Claim:** There is a negative-cost cycle that can reach \( t \) iff for some vertex \( v \in V \), \( \text{Cost}(v,t,n) < \text{Cost}(v,t,n-1) \)

- **Proof:**
  - We already know that if there aren’t any then we only need paths of length up to \( n-1 \)
  - For the other direction
    - The recurrence computes \( \text{Cost}(v,t,i) \) correctly for any number of hops \( i \)
    - The recurrence reaches a fixed point if for every \( v \in V \), \( \text{Cost}(v,t,i) = \text{Cost}(v,t,i-1) \)
    - A negative-cost cycle means that eventually some \( \text{Cost}(v,t,i) \) gets smaller than any given bound
      - Can’t have a –ve cost cycle if for every \( v \in V \), \( \text{Cost}(v,t,n) = \text{Cost}(v,t,n-1) \)
Last details

- Can run algorithm and stop early if the $\text{OPT}$ and $\text{OPT}'$ arrays are ever equal
  - Even better, one can update only neighbors $v$ of vertices $w$ with $\text{OPT}'[w] \neq \text{OPT}[w]$.

- Can store a successor pointer when we compute $\text{OPT}$
  - Homework assignment

- By running for step $n$ we can find some vertex $v$ on a negative cycle and use the successor pointers to find the cycle.
Bellman-Ford
Bellman-Ford

Diagram of a graph with weighted edges.
Bellman-Ford
Bellman-Ford
Bellman-Ford

![Graph Diagram]

- Nodes: 0, 1, 2, 3
- Edges with weights:
  - (0, 1) = 5
  - (1, 2) = -2
  - (2, 3) = 7
  - (3, 0) = 9
- Additional edges with weights:
  - (0, 2) = 2
  - (1, 3) = -3
  - (2, 0) = 6
  - (3, 1) = 8

The diagram illustrates the Bellman-Ford algorithm's process for finding the shortest paths in a graph with negative edge weights.
Bellman-Ford
Bellman-Ford with a DAG

Edges only go from lower to higher-numbered vertices
• Update distances in reverse order of topological sort
• Only one pass through vertices required
• $O(n+m)$ time