Dynamic Programming

T. M. Murali

February 25, 27, March 17, 19 2008

Algorithm Design Techniques

1. Goal: design efficient (polynomial-time) algorithms.

Algorithm Design Techniques

1. Goal: design efficient (polynomial-time) algorithms.
2. Greedy
   - Pro: natural approach to algorithm design.
   - Con: many greedy approaches a problem. Only some may work.
   - Con: many problems for which *no* greedy approach is known.

Algorithm Design Techniques

1. Goal: design efficient (polynomial-time) algorithms.
2. Greedy
   - Pro: natural approach to algorithm design.
   - Con: many greedy approaches a problem. Only some may work.
   - Con: many problems for which *no* greedy approach is known.
3. Divide and conquer
   - Pro: simple to develop algorithm skeleton.
   - Con: conquer step can be very hard to implement efficiently.
   - Con: usually reduces time for a problem known to be solvable in polynomial time.

Algorithm Design Techniques

1. Goal: design efficient (polynomial-time) algorithms.
2. Greedy
   - Pro: natural approach to algorithm design.
   - Con: many greedy approaches a problem. Only some may work.
   - Con: many problems for which *no* greedy approach is known.
3. Divide and conquer
   - Pro: simple to develop algorithm skeleton.
   - Con: conquer step can be very hard to implement efficiently.
   - Con: usually reduces time for a problem known to be solvable in polynomial time.
4. Dynamic programming
   - More powerful than greedy and divide-and-conquer strategies.
   - *Implicitly* explore space of all possible solutions.
   - Solve multiple sub-problems and build up correct solutions to larger and larger sub-problems.
   - Careful analysis needed to ensure number of sub-problems solved is polynomial in the size of the input.

History of Dynamic Programming

- Bellman pioneered the systematic study of dynamic programming in the 1950s.

History of Dynamic Programming

- Bellman pioneered the systematic study of dynamic programming in the 1950s.
- Dynamic programming = "planning over time."
- The Secretary of Defense at that time was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.
  - "it's impossible to use dynamic in a pejorative sense"
  - "something not even a Congressman could object to" Reference:
  - Bellman, R. E., *Eye of the Hurricane, An Autobiography*.

Applications of Dynamic Programming

- Computational biology: Smith-Waterman algorithm for sequence alignment.
- Operations research: Bellman-Ford algorithm for shortest path routing in networks.
- Control theory: Viterbi algorithm for hidden Markov models.
- Computer science (theory, graphics, AI, ...): Unix `diff` command for comparing two files.

Review: Interval Scheduling

INTERVAL SCHEDULING

INSTANCE: Nonempty set $\{(s_i, f_i), 1 \leq i \leq n\}$ of start and finish times of n jobs.

SOLUTION: The largest subset of mutually compatible jobs.

- Two jobs are *compatible* if they do not overlap.

Review: Interval Scheduling

Interval Scheduling

INSTANCE: Nonempty set $\{(s_i, f_i), 1 \leq i \leq n\}$ of start and finish times of n jobs.

SOLUTION: The largest subset of mutually compatible jobs.

- Two jobs are *compatible* if they do not overlap.
- Greedy algorithm: sort jobs in increasing order of finish times. Add next job to current subset only if it is compatible with previously-selected jobs.

Weighted Interval Scheduling

WEIGHTED INTERVAL SCHEDULING

INSTANCE: Nonempty set $\{(s_i, f_i), 1 \leq i \leq n\}$ of start and finish times of n jobs and a weight $v_i \geq 0$ associated with each job.

SOLUTION: A set S of mutually compatible jobs such that $\sum_{i \in S} v_i$ is maximised.

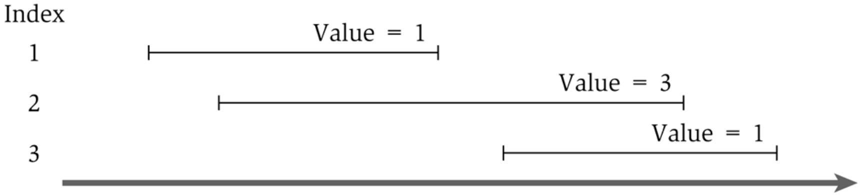


Figure 6.1 A simple instance of weighted interval scheduling.

Weighted Interval Scheduling

WEIGHTED INTERVAL SCHEDULING

INSTANCE: Nonempty set $\{(s_i, f_i), 1 \leq i \leq n\}$ of start and finish times of n jobs and a weight $v_i \geq 0$ associated with each job.

SOLUTION: A set S of mutually compatible jobs such that $\sum_{i \in S} v_i$ is maximised.

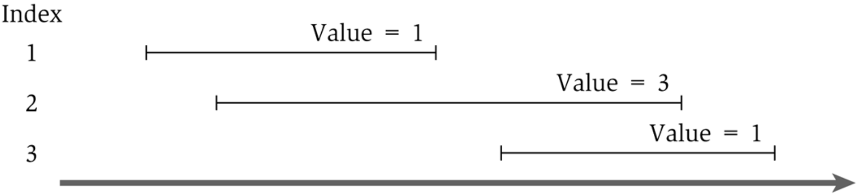


Figure 6.1 A simple instance of weighted interval scheduling.

- Greedy algorithm can produce arbitrarily bad results for this problem.

Approach

- Sort jobs in increasing order of finish time and relabel: $f_1 \leq f_2 \leq \ldots \leq f_n$.
- Request i comes before request j if $i < j$.
- $p(j)$ is the largest index $i < j$ such that job i is compatible with job j. $p(j) = 0$ if there is no such job i.

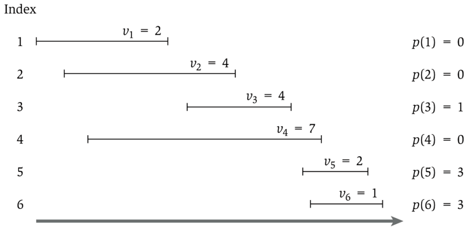
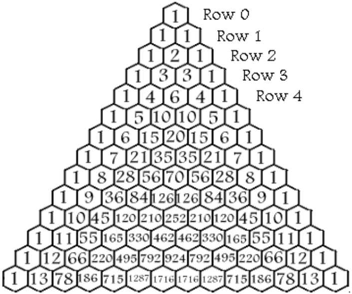


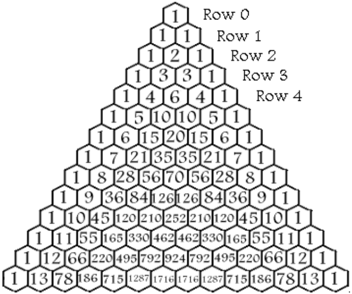
Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

- We will develop optimal algorithm from very obvious statements about the problem.

Detour: a Binomial Identity

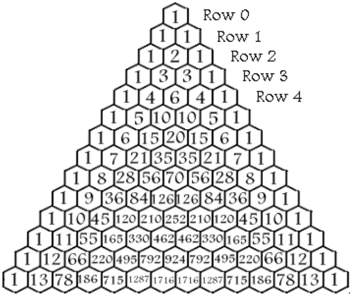


Detour: a Binomial Identity



- Pascal's triangle:
  - Each element is a binomial co-efficient.
  - Each element is the sum of the two elements above it.

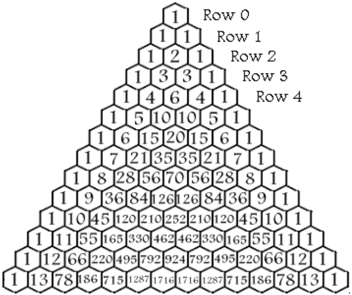
Detour: a Binomial Identity



- Pascal's triangle:
  - Each element is a binomial co-efficient.
  - Each element is the sum of the two elements above it.

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}$$

Detour: a Binomial Identity



- Pascal's triangle:
  - Each element is a binomial co-efficient.
  - Each element is the sum of the two elements above it.

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}$$

- Proof: either we select the nth element or not ...

Sub-problems

- Let $\mathcal{O}$ be the optimal solution. Two cases to consider.
  - Case 1 job n is not in $\mathcal{O}$.
  - Case 2 job n is in $\mathcal{O}$.

Sub-problems

- Let $\mathcal{O}$ be the optimal solution. Two cases to consider.
  - Case 1: job n is not in $\mathcal{O}$. $\mathcal{O}$ must be the optimal solution for jobs $\{1, 2, \ldots, n-1\}$.
  - Case 2: job n is in $\mathcal{O}$.

Sub-problems

- Let $\mathcal{O}$ be the optimal solution. Two cases to consider.

  Case 1 job n is not in $\mathcal{O}$. $\mathcal{O}$ must be the optimal solution for jobs $\{1, 2, \ldots, n-1\}$.

  Case 2 job n is in $\mathcal{O}$.

    - $\mathcal{O}$ cannot use incompatible jobs $\{p(n)+1, p(n)+2, \ldots, n-1\}$.
    - Remaining jobs in $\mathcal{O}$ must be the optimal solution for jobs $\{1, 2, \ldots, p(n)\}$.

Sub-problems

- Let $\mathcal{O}$ be the optimal solution. Two cases to consider.
  - Case 1: job n is not in $\mathcal{O}$. $\mathcal{O}$ must be the optimal solution for jobs $\{1, 2, \ldots, n-1\}$.
  - Case 2: job n is in $\mathcal{O}$.
    - $\mathcal{O}$ cannot use incompatible jobs $\{p(n)+1, p(n)+2, \ldots, n-1\}$.
    - Remaining jobs in $\mathcal{O}$ must be the optimal solution for jobs $\{1, 2, \ldots, p(n)\}$.
- $\mathcal{O}$ must be the best of these two choices!

Sub-problems

- Let $\mathcal{O}$ be the optimal solution. Two cases to consider.
  - Case 1: job n is not in $\mathcal{O}$. $\mathcal{O}$ must be the optimal solution for jobs $\{1, 2, \ldots, n-1\}$.
  - Case 2: job n is in $\mathcal{O}$.
    - $\mathcal{O}$ cannot use incompatible jobs $\{p(n)+1, p(n)+2, \ldots, n-1\}$.
    - Remaining jobs in $\mathcal{O}$ must be the optimal solution for jobs $\{1, 2, \ldots, p(n)\}$.
- $\mathcal{O}$ must be the best of these two choices!
- Suggests finding optimal solution for sub-problems consisting of jobs $\{1, 2, \ldots, j-1, j\}$, for all values of j.

Recursion

- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($\mathrm{OPT}(0) = 0$).

Recursion

- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($\mathrm{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\mathrm{OPT}(n)$.

Recursion

- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution (OPT(0) = 0).
- We are seeking $\mathcal{O}_n$ with a value of OPT(n).
- To compute OPT(j):

  Case 1 $j \notin \mathcal{O}_j$:

Recursion

- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($\text{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\text{OPT}(n)$.
- To compute $\text{OPT}(j)$:

  Case 1 $j \notin \mathcal{O}_j$: $\text{OPT}(j) = \text{OPT}(j-1)$.

Recursion

- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($\text{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\text{OPT}(n)$.
- To compute $\text{OPT}(j)$:

  Case 1 $j \notin \mathcal{O}_j$: $\text{OPT}(j) = \text{OPT}(j-1)$.

  Case 2 $j \in \mathcal{O}_j$:

Recursion

- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution (OPT(0) = 0).
- We are seeking $\mathcal{O}_n$ with a value of OPT(n).
- To compute OPT(j):

  Case 1 $j \notin \mathcal{O}_j$: $\text{OPT}(j) = \text{OPT}(j-1)$.
  Case 2 $j \in \mathcal{O}_j$: $\text{OPT}(j) = v_j + \text{OPT}(p(j))$

Recursion

- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($\text{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\text{OPT}(n)$.
- To compute $\text{OPT}(j)$:

  Case 1 $j \notin \mathcal{O}_j$: $\text{OPT}(j) = \text{OPT}(j-1)$.
  Case 2 $j \in \mathcal{O}_j$: $\text{OPT}(j) = v_j + \text{OPT}(p(j))$

$$\text{OPT}(j) = \max(v_j + \text{OPT}(p(j)), \text{OPT}(j-1))$$

Recursion

- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($\text{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\text{OPT}(n)$.
- To compute $\text{OPT}(j)$:

  Case 1 $j \notin \mathcal{O}_j$: $\text{OPT}(j) = \text{OPT}(j-1)$.
  Case 2 $j \in \mathcal{O}_j$: $\text{OPT}(j) = v_j + \text{OPT}(p(j))$

$$\text{OPT}(j) = \max(v_j + \text{OPT}(p(j)), \text{OPT}(j-1))$$

- When does request j belong to $\mathcal{O}_j$?

Recursion

- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($\text{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\text{OPT}(n)$.
- To compute $\text{OPT}(j)$:

  Case 1 $j \notin \mathcal{O}_j$: $\text{OPT}(j) = \text{OPT}(j-1)$.
  Case 2 $j \in \mathcal{O}_j$: $\text{OPT}(j) = v_j + \text{OPT}(p(j))$

$$\text{OPT}(j) = \max(v_j + \text{OPT}(p(j)), \text{OPT}(j-1))$$

- When does request j belong to $\mathcal{O}_j$? If and only if $v_j + \text{OPT}(p(j)) \geq \text{OPT}(j-1)$.

Recursive Algorithm

```
Compute-Opt(j)
  If j = 0 then
    Return 0
  Else
    Return max(v_j + Compute-Opt(p(j)), Compute-Opt(j − 1))
  Endif
```

Recursive Algorithm

```
Compute-Opt(j)
  If j = 0 then
    Return 0
  Else
    Return max(v_j + Compute-Opt(p(j)), Compute-Opt(j − 1))
  Endif
```

- Correctness of algorithm follows by induction.
- What is the running time of the algorithm?

Recursive Algorithm

```
Compute-Opt(j)
  If j = 0 then
    Return 0
  Else
    Return max(v_j+Compute-Opt(p(j)), Compute-Opt(j − 1))
  Endif
```

- Correctness of algorithm follows by induction.
- What is the running time of the algorithm? Can be exponential in n.

Recursive Algorithm

```
Compute-Opt(j)
  If j=0 then
    Return 0
  Else
    Return max(v_j+Compute-Opt(p(j)), Compute-Opt(j-1))
  Endif
```

- Correctness of algorithm follows by induction.
- What is the running time of the algorithm? Can be exponential in n.
- When $p(j) = j - 2$, for all $j \geq 2$: recursive calls are for $j - 1$ and $j - 2$.

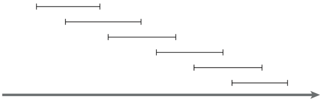


Figure 6.4 An instance of weighted interval scheduling on which the simple Compute-Opt recursion will take exponential time. The values of all intervals in this instance are 1.

Memoisation

- Store OPT(j) values in a cache and reuse them rather than recompute them.

Memoisation

- Store OPT(j) values in a cache and reuse them rather than recompute them.

```
M-Compute-Opt(j)
  If j=0 then
    Return 0
  Else if M[j] is not empty then
    Return M[j]
  Else
   Define M[j] = max(v_j+M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
    Return M[j]
  Endif
```

Running Time of Memoisation

```
M-Compute-Opt(j)
  If j = 0 then
    Return 0
  Else if M[j] is not empty then
    Return M[j]
  Else
    Define M[j] = max(v_j+M-Compute-Opt(p(j)), M-Compute-Opt(j − 1))
    Return M[j]
  Endif
```

- Claim: running time of this algorithm is $O(n)$ (after sorting).

Running Time of Memoisation

```
M-Compute-Opt(j)
  If j = 0 then
    Return 0
  Else if M[j] is not empty then
    Return M[j]
  Else
   Define M[j] = max(v_j+M-Compute-Opt(p(j)), M-Compute-Opt(j − 1))
    Return M[j]
  Endif
```

- Claim: running time of this algorithm is $O(n)$ (after sorting).
- Time spent in a single call to `M-Compute-Opt` is $O(1)$ apart from time spent in recursive calls.
- Total time spent is the order of the number of recursive calls to `M-Compute-Opt`.
- How many such recursive calls are there in total?

Running Time of Memoisation

```
M-Compute-Opt(j)
  If j = 0 then
    Return 0
  Else if M[j] is not empty then
    Return M[j]
  Else
   Define M[j] = max(v_j+M-Compute-Opt(p(j)), M-Compute-Opt(j − 1))
    Return M[j]
  Endif
```

- Claim: running time of this algorithm is $O(n)$ (after sorting).
- Time spent in a single call to `M-Compute-Opt` is $O(1)$ apart from time spent in recursive calls.
- Total time spent is the order of the number of recursive calls to `M-Compute-Opt`.
- How many such recursive calls are there in total?
- Use number of filled entries in M as a measure of progress.
- Each time `M-Compute-Opt` issues two recursive calls, it fills in a new entry in M.

Computing $\mathcal{O}$ in Addition to OPT(n)

Computing $\mathcal{O}$ in Addition to OPT(n)

- Explicitly store $\mathcal{O}_j$ in addition to OPT(j).

Computing $\mathcal{O}$ in Addition to OPT(n)

- Explicitly store $\mathcal{O}_j$ in addition to OPT(j). Running time becomes $O(n^2)$.

Computing $\mathcal{O}$ in Addition to OPT(n)

- Explicitly store $\mathcal{O}_j$ in addition to OPT(j). Running time becomes $O(n^2)$.
- Recall: request j belong to $\mathcal{O}_j$ if and only if $v_j + \text{OPT}(p(j)) \geq \text{OPT}(j-1)$.
- Can recover $\mathcal{O}_j$ from values of the optimal solutions in $O(j)$ time.

Computing $\mathcal{O}$ in Addition to OPT(n)

- Explicitly store $\mathcal{O}_j$ in addition to OPT(j). Running time becomes $O(n^2)$.
- Recall: request j belong to $\mathcal{O}_j$ if and only if $v_j + \text{OPT}(p(j)) \geq \text{OPT}(j-1)$.
- Can recover $\mathcal{O}_j$ from values of the optimal solutions in $O(j)$ time.

```
Find-Solution(j)
  If j=0 then
    Output nothing
  Else
    If v_j + M[p(j)] ≥ M[j-1] then
      Output j together with the result of Find-Solution(p(j))
    Else
      Output the result of Find-Solution(j-1)
    Endif
  Endif
```

From Recursion to Iteration

- Unwind the recursion and convert it into iteration.
- Can compute values in M iteratively in $O(n)$ time.
- `Find-Solution` works as before.

```
Iterative-Compute-Opt
  M[0] = 0
  For j = 1, 2, ..., n
    M[j] = max(v_j + M[p(j)], M[j − 1])
  Endfor
```

Basic Outline of Dynamic Programming

- To solve a problem, we need a collection of sub-problems that satisfy a few properties:
  1. There are a polynomial number of sub-problems.
  2. The solution to the problem can be computed easily from the solutions to the sub-problems.
  3. There is a natural ordering of the sub-problems from "smallest" to "largest".
  4. There is an easy-to-compute recurrence that allows us to compute the solution to a sub-problem from the solutions to some smaller sub-problems.

Basic Outline of Dynamic Programming

- To solve a problem, we need a collection of sub-problems that satisfy a few properties:
  1. There are a polynomial number of sub-problems.
  2. The solution to the problem can be computed easily from the solutions to the sub-problems.
  3. There is a natural ordering of the sub-problems from "smallest" to "largest".
  4. There is an easy-to-compute recurrence that allows us to compute the solution to a sub-problem from the solutions to some smaller sub-problems.
- Difficulties in designing dynamic programming algorithms:
  1. Which sub-problems to define?
  2. How can we tie up sub-problems using a recurrence?
  3. How do we order the sub-problems (to allow iterative computation of optimal solutions to sub-problems)?

Least Squares Problem

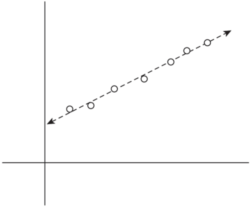


Figure 6.6 A "line of best fit."

- Given scientific or statistical data plotted on two axes.
- Find the "best" line that "passes" through these points.

Least Squares Problem

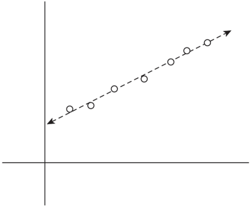


Figure 6.6 A "line of best fit."

- Given scientific or statistical data plotted on two axes.
- Find the "best" line that "passes" through these points.
- How do we formalise the problem?

Least Squares Problem

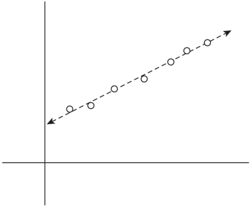


Figure 6.6 A "line of best fit."

- Given scientific or statistical data plotted on two axes.
- Find the "best" line that "passes" through these points.
- How do we formalise the problem?

LEAST SQUARES
INSTANCE: Set $P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ of n points.
SOLUTION: Line $L : y = ax + b$ that minimises

$$\text{Error}(L, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2.$$

Least Squares Problem

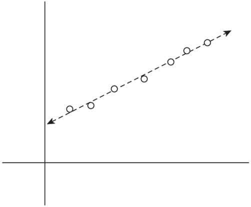


Figure 6.6 A "line of best fit."

- Given scientific or statistical data plotted on two axes.
- Find the "best" line that "passes" through these points.
- How do we formalise the problem?

LEAST SQUARES
INSTANCE: Set $P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ of n points.
SOLUTION: Line $L : y = ax + b$ that minimises

$$\text{Error}(L, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2.$$

- Solution is achieved by

$$a = \frac{n \sum_i x_i y_i - \left(\sum_i x_i\right)\left(\sum_i y_i\right)}{n \sum_i x_i^2 - \left(\sum_i x_i\right)^2} \text{ and } b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

Segmented Least Squares

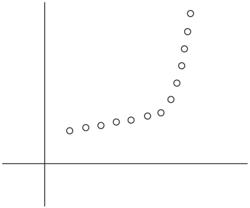


Figure 6.7 A set of points that lie approximately on two lines.

Segmented Least Squares

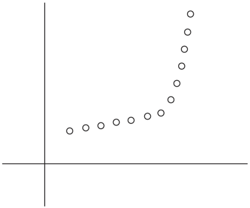


Figure 6.7 A set of points that lie approximately on two lines.

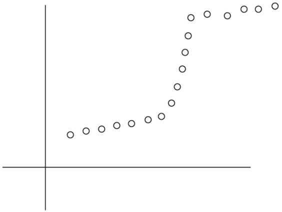


Figure 6.8 A set of points that lie approximately on three lines.

Segmented Least Squares

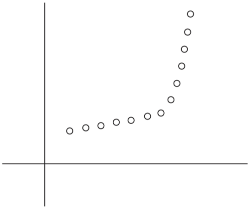


Figure 6.7 A set of points that lie approximately on two lines.

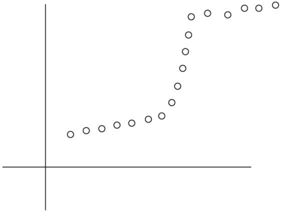


Figure 6.8 A set of points that lie approximately on three lines.

- Want to fit multiple lines through P.
- Each line must fit contiguous set of x-coordinates.
- Lines must minimise total error.

Segmented Least Squares

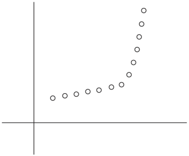


Figure 6.7 A set of points that lie approximately on two lines.

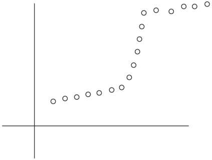


Figure 6.8 A set of points that lie approximately on three lines.

Segmented Least Squares

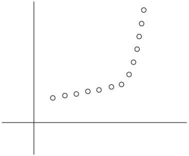


Figure 6.7 A set of points that lie approximately on two lines.

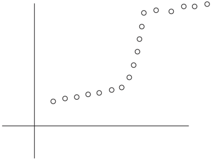


Figure 6.8 A set of points that lie approximately on three lines.

SEGMENTED LEAST SQUARES

INSTANCE: Set $P = \{p_i = (x_i, y_i), 1 \leq i \leq n\}$ of n points, $x_1 < x_2 < \cdots < x_n$.

SOLUTION: A integer k, a partition of P into k segments $\{P_1, P_2, \ldots, P_k\}$, k lines $L_j : y = a_j x + b_j, 1 \leq j \leq k$ that minimise

$$\sum_{j=1}^{k} \text{Error}(L_j, P_j)$$

- A subset P' of P is a *segment* if $1 \leq i < j \leq n$ exist such that $P' = \{(x_i, y_i), (x_{i+1}, y_{i+1}), \ldots, (x_{j-1}, y_{j-1}), (x_j, y_j)\}$.

Segmented Least Squares

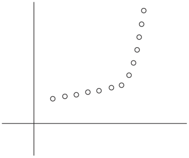


Figure 6.7 A set of points that lie approximately on two lines.

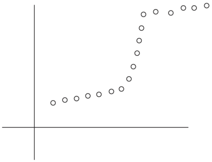


Figure 6.8 A set of points that lie approximately on three lines.

SEGMENTED LEAST SQUARES

INSTANCE: Set $P = \{p_i = (x_i, y_i), 1 \leq i \leq n\}$ of n points, $x_1 < x_2 < \cdots < x_n$ and a parameter $C > 0$.

SOLUTION: A integer k, a partition of P into k segments $\{P_1, P_2, \ldots, P_k\}$, k lines $L_j : y = a_j x + b_j, 1 \leq j \leq k$ that minimise

$$\sum_{j=1}^{k} \text{Error}(L_j, P_j) + Ck.$$

- A subset P' of P is a *segment* if $1 \leq i < j \leq n$ exist such that $P' = \{(x_i, y_i), (x_{i+1}, y_{i+1}), \ldots, (x_{j-1}, y_{j-1}), (x_j, y_j)\}$.

Formulating the Recursion: I

- Observation: p_n is part of some segment in the optimal solution. This segment starts at some point p_i.
- Let $OPT(i)$ be the optimal value for the points $\{p_1, p_2, \ldots, p_i\}$.
- Let $e_{i,j}$ denote the minimum error of any line that fits $\{p_i, p_2, \ldots, p_j\}$.
- We want to compute $\mathrm{OPT}(n)$.

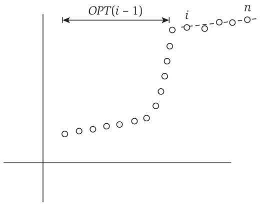


Figure 6.9 A possible solution: a single line segment fits points $p_i, p_{i+1}, \ldots, p_n$, and then an optimal solution is found for the remaining points $p_1, p_2, \ldots, p_{i-1}$.

- If the last segment in the optimal partition is $\{p_i, p_{i+1}, \ldots, p_n\}$, then

$$\mathrm{OPT}(n) = e_{i,n} + C + \mathrm{OPT}(i-1)$$

Formulating the Recursion: II

- Consider the sub-problem on the points $\{p_1, p_2, \ldots p_j\}$
- To obtain $\text{OPT}(j)$, if the last segment in the optimal partition is $\{p_i, p_{i+1}, \ldots, p_j\}$, then

$$\text{OPT}(j) = e_{i,j} + C + \text{OPT}(i-1)$$

Formulating the Recursion: II

- Consider the sub-problem on the points $\{p_1, p_2, \ldots p_j\}$
- To obtain $\text{OPT}(j)$, if the last segment in the optimal partition is $\{p_i, p_{i+1}, \ldots, p_j\}$, then

$$\text{OPT}(j) = e_{i,j} + C + \text{OPT}(i-1)$$

- Since i can take only j distinct values,

$$\text{OPT}(j) = \min_{1 \leq i \leq j} \left(e_{i,j} + C + \text{OPT}(i-1)\right)$$

- Segment $\{p_i, p_{i+1}, \ldots p_j\}$ is part of the optimal solution for this sub-problem if and only if the minimum value of $\text{OPT}(j)$ is obtained using index i. solution

Dynamic Programming Algorithm

$$\text{OPT}(j) = \min_{1 \leq i \leq j} (e_{i,j} + C + \text{OPT}(i-1))$$

```
Segmented-Least-Squares(n)
  Array M[0 ... n]
  Set M[0] = 0
  For all pairs i ≤ j
    Compute the least squares error e_{i,j} for the segment p_i, ..., p_j
  Endfor
  For j = 1, 2, ..., n
    Use the recurrence (6.7) to compute M[j]
  Endfor
  Return M[n]
```

Dynamic Programming Algorithm

$$\mathrm{OPT}(j) = \min_{1 \le i \le j} \left(e_{i,j} + C + \mathrm{OPT}(i-1)\right)$$

```
Segmented-Least-Squares(n)
  Array M[0...n]
  Set M[0] = 0
  For all pairs i ≤ j
    Compute the least squares error e_{i,j} for the segment p_i, ..., p_j
  Endfor
  For j = 1, 2, ..., n
    Use the recurrence (6.7) to compute M[j]
  Endfor
  Return M[n]
```

- Running time is $O(n^3)$, can be improved to $O(n^2)$.
- We can find the segments in the optimal solution by backtracking

RNA Molecules

- RNA is a basic biological molecule. It is single stranded.
- RNA molecules fold into complex "secondary structures."
- Secondary structure often governs the behaviour of an RNA molecule.
- Various rules govern secondary structure formation:

RNA Molecules

- RNA is a basic biological molecule. It is single stranded.
- RNA molecules fold into complex "secondary structures."
- Secondary structure often governs the behaviour of an RNA molecule.
- Various rules govern secondary structure formation:

1. Pairs of bases match up; each base matches with ≤ 1 other base.
2. Adenine always matches with Uracil.
3. Cytosine always matches with Guanine.
4. There are no kinks in the folded molecule.
5. Structures are "knot-free".

RNA Molecules

- RNA is a basic biological molecule. It is single stranded.
- RNA molecules fold into complex "secondary structures."
- Secondary structure often governs the behaviour of an RNA molecule.
- Various rules govern secondary structure formation:

1. Pairs of bases match up; each base matches with ≤ 1 other base.
2. Adenine always matches with Uracil.
3. Cytosine always matches with Guanine.
4. There are no kinks in the folded molecule.
5. Structures are "knot-free".

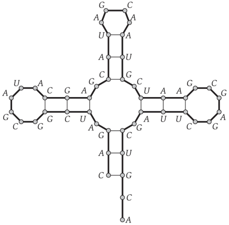


Figure 6.13 An RNA secondary structure. Thick lines connect adjacent elements of the sequence; thin lines indicate pairs of elements that are matched.

RNA Molecules

- RNA is a basic biological molecule. It is single stranded.
- RNA molecules fold into complex "secondary structures."
- Secondary structure often governs the behaviour of an RNA molecule.
- Various rules govern secondary structure formation:

1. Pairs of bases match up; each base matches with ≤ 1 other base.
2. Adenine always matches with Uracil.
3. Cytosine always matches with Guanine.
4. There are no kinks in the folded molecule.
5. Structures are "knot-free".

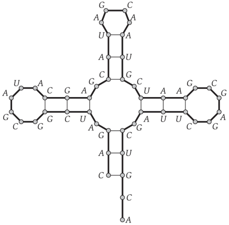


Figure 6.13 An RNA secondary structure. Thick lines connect adjacent elements of the sequence; thin lines indicate pairs of elements that are matched.

- Problem: given an RNA molecule, predict its secondary structure.

RNA Molecules

- RNA is a basic biological molecule. It is single stranded.
- RNA molecules fold into complex "secondary structures."
- Secondary structure often governs the behaviour of an RNA molecule.
- Various rules govern secondary structure formation:

1. Pairs of bases match up; each base matches with ≤ 1 other base.
2. Adenine always matches with Uracil.
3. Cytosine always matches with Guanine.
4. There are no kinks in the folded molecule.
5. Structures are "knot-free".

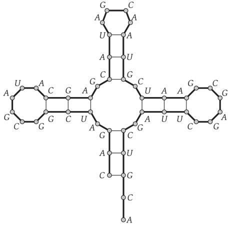


Figure 6.13 An RNA secondary structure. Thick lines connect adjacent elements of the sequence; thin lines indicate pairs of elements that are matched.

- Problem: given an RNA molecule, predict its secondary structure.
- Hypothesis: In the cell, RNA molecules form the secondary structure with the lowest total free energy.

Formulating the Problem

- An *RNA molecule* is a string $B = b_1 b_2 \dots b_n$; each $b_i \in \{A, C, G, U\}$.
- A *secondary structure on B* is a set of pairs $S = \{(i, j)\}$, where $1 \leq i, j \leq n$ and

Formulating the Problem

- An *RNA molecule* is a string $B = b_1 b_2 \dots b_n$; each $b_i \in \{A, C, G, U\}$.
- A *secondary structure on B* is a set of pairs $S = \{(i, j)\}$, where $1 \leq i, j \leq n$ and
  1. *(No kinks.)* If $(i, j) \in S$, then $i < j - 4$.
  2. *(Watson-Crick)* The elements in each pair in S consist of either $\{A, U\}$ or $\{C, G\}$ (in either order).
  3. S is a *matching*: no index appears in more than one pair.
  4. *(No knots)* If (i, j) and (k, l) are two pairs in S, then we cannot have $i < k < j < l$.

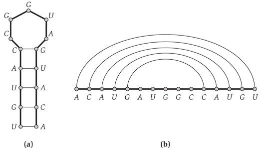


Figure 6.14 Two views of an RNA secondary structure. In the second view, (b), the string has been "stretched" lengthwise, and edges connecting matched pairs appear as noncrossing "bubbles" over the string.

- The *energy* of a secondary structure is proportional to the number of base pairs in it.

Dynamic Programming Approach

- $OPT(j)$ is the maximum number of base pairs in a secondary structure for $b_1 b_2 \dots b_j$.

Dynamic Programming Approach

- *OPT*(j) is the maximum number of base pairs in a secondary structure for $b_1 b_2 \ldots b_j$. $\text{OPT}(j) = 0$, if $j \leq 5$.

Dynamic Programming Approach

- $OPT(j)$ is the maximum number of base pairs in a secondary structure for $b_1 b_2 \ldots b_j$. $\mathrm{OPT}(j) = 0$, if $j \leq 5$.
- In the optimal secondary structure on $b_1 b_2 \ldots b_j$

Dynamic Programming Approach

- $OPT(j)$ is the maximum number of base pairs in a secondary structure for $b_1 b_2 \ldots b_j$. $\text{OPT}(j) = 0$, if $j \leq 5$.
- In the optimal secondary structure on $b_1 b_2 \ldots b_j$
  1. if j is not a member of any pair, use $\text{OPT}(j-1)$.

Dynamic Programming Approach

- $OPT(j)$ is the maximum number of base pairs in a secondary structure for $b_1 b_2 \dots b_j$. $\text{OPT}(j) = 0$, if $j \leq 5$.
- In the optimal secondary structure on $b_1 b_2 \dots b_j$
  1. if j is not a member of any pair, use $\text{OPT}(j-1)$.
  2. if j pairs with some $t < j - 4$,

Dynamic Programming Approach

- $OPT(j)$ is the maximum number of base pairs in a secondary structure for $b_1 b_2 \ldots b_j$. $\text{OPT}(j) = 0$, if $j \leq 5$.
- In the optimal secondary structure on $b_1 b_2 \ldots b_j$
  1. if j is not a member of any pair, use $\text{OPT}(j-1)$.
  2. if j pairs with some $t < j - 4$, knot condition yields two independent sub-problems!

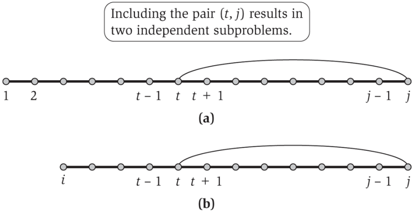


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

Dynamic Programming Approach

- $OPT(j)$ is the maximum number of base pairs in a secondary structure for $b_1 b_2 \ldots b_j$. $\mathrm{OPT}(j) = 0$, if $j \leq 5$.
- In the optimal secondary structure on $b_1 b_2 \ldots b_j$
  1. if j is not a member of any pair, use $\mathrm{OPT}(j-1)$.
  2. if j pairs with some $t < j - 4$, knot condition yields two independent sub-problems! $\mathrm{OPT}(t-1)$ and ???

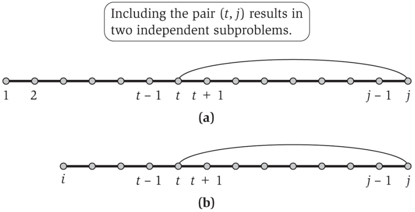


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

Dynamic Programming Approach

- $OPT(j)$ is the maximum number of base pairs in a secondary structure for $b_1 b_2 \ldots b_j$. $\text{OPT}(j) = 0$, if $j \leq 5$.
- In the optimal secondary structure on $b_1 b_2 \ldots b_j$
  1. if j is not a member of any pair, use $\text{OPT}(j-1)$.
  2. if j pairs with some $t < j - 4$, knot condition yields two independent sub-problems! $\text{OPT}(t-1)$ and ???
- Insight: need sub-problems indexed both by start and by end.

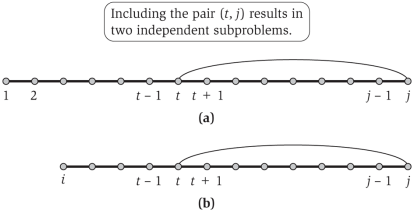


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

Correct Dynamic Programming Approach

- $OPT(i, j)$ is the maximum number of base pairs in a secondary structure for $b_i b_2 \ldots b_j$.

$$\text{OPT}(i, j) = \max \left(\text{OPT}(i, j{-}1), \qquad \right)$$

Correct Dynamic Programming Approach

- *OPT(i, j)* is the maximum number of base pairs in a secondary structure for $b_i b_2 \dots b_j$. $\text{OPT}(i,j) = 0$, if $i \geq j - 4$.

$$\text{OPT}(i,j) = \max\left(\text{OPT}(i, j{-}1), \qquad \qquad \qquad \qquad \right)$$

Correct Dynamic Programming Approach

- $OPT(i, j)$ is the maximum number of base pairs in a secondary structure for $b_i b_2 \ldots b_j$. $\text{OPT}(i, j) = 0$, if $i \geq j - 4$.
- In the optimal secondary structure on $b_i b_2 \ldots b_j$

$$\text{OPT}(i, j) = \max \Big(\text{OPT}(i, j{-}1), \qquad \Big)$$

Correct Dynamic Programming Approach

- *OPT(i, j)* is the maximum number of base pairs in a secondary structure for $b_i b_2 \ldots b_j$. $\text{OPT}(i, j) = 0$, if $i \geq j - 4$.
- In the optimal secondary structure on $b_i b_2 \ldots b_j$
    1. if j is not a member of any pair, compute $\text{OPT}(i, j-1)$.

$$\text{OPT}(i, j) = \max \left(\text{OPT}(i, j-1), \qquad \qquad \qquad \qquad \right)$$

Correct Dynamic Programming Approach

- *$OPT(i,j)$* is the maximum number of base pairs in a secondary structure for $b_i b_2 \dots b_j$. $\text{OPT}(i,j) = 0$, if $i \geq j - 4$.
- In the optimal secondary structure on $b_i b_2 \dots b_j$
  1. if j is not a member of any pair, compute $\text{OPT}(i, j-1)$.
  2. if j pairs with some $t < j - 4$, compute $\text{OPT}(i, t-1)$ and $\text{OPT}(t+1, j-1)$.

$$\text{OPT}(i,j) = \max\left(\text{OPT}(i, j-1), \qquad\qquad\qquad\qquad\qquad\qquad\right)$$

Correct Dynamic Programming Approach

- $OPT(i,j)$ is the maximum number of base pairs in a secondary structure for $b_i b_2 \ldots b_j$. $\mathrm{OPT}(i,j) = 0$, if $i \geq j - 4$.
- In the optimal secondary structure on $b_i b_2 \ldots b_j$
  1. if j is not a member of any pair, compute $\mathrm{OPT}(i, j-1)$.
  2. if j pairs with some $t < j - 4$, compute $\mathrm{OPT}(i, t-1)$ and $\mathrm{OPT}(t+1, j-1)$.
- Since t can range from i to $j-1$,

$$\mathrm{OPT}(i,j) = \max\left(\mathrm{OPT}(i, j{-}1), \qquad \right)$$

Correct Dynamic Programming Approach

- $OPT(i, j)$ is the maximum number of base pairs in a secondary structure for $b_i b_2 \ldots b_j$. $\text{OPT}(i, j) = 0$, if $i \geq j - 4$.
- In the optimal secondary structure on $b_i b_2 \ldots b_j$
  1. if j is not a member of any pair, compute $\text{OPT}(i, j - 1)$.
  2. if j pairs with some $t < j - 4$, compute $\text{OPT}(i, t - 1)$ and $\text{OPT}(t + 1, j - 1)$.
- Since t can range from i to $j - 1$,

$$\text{OPT}(i, j) = \max \left(\text{OPT}(i, j{-}1), \max_t \left(1{+}\text{OPT}(i, t{-}1){+}\text{OPT}(t{+}1, j{-}1) \right) \right)$$

Correct Dynamic Programming Approach

- *OPT*(i, j) is the maximum number of base pairs in a secondary structure for $b_i b_2 \ldots b_j$. $\text{OPT}(i, j) = 0$, if $i \geq j - 4$.
- In the optimal secondary structure on $b_i b_2 \ldots b_j$
  1. if j is not a member of any pair, compute $\text{OPT}(i, j - 1)$.
  2. if j pairs with some $t < j - 4$, compute $\text{OPT}(i, t - 1)$ and $\text{OPT}(t + 1, j - 1)$.
- Since t can range from i to $j - 1$,

$$\text{OPT}(i, j) = \max\left(\text{OPT}(i, j-1), \max_t \left(1+\text{OPT}(i, t-1)+\text{OPT}(t+1, j-1)\right)\right)$$

- In the "inner" maximisation, t runs over all indices between i and $j - 1$ that are allowed to pair with j.

Dynamic Programming Algorithm

$$\text{OPT}(i,j) = \max\left(\text{OPT}(i,j-1), \max_t \left(1+\text{OPT}(i,t-1)+\text{OPT}(t+1,j-1)\right)\right)$$

- There are $O(n^2)$ sub-problems.
- How do we order them from "smallest" to "largest"?

Dynamic Programming Algorithm

$$\text{OPT}(i,j) = \max\left(\text{OPT}(i,j-1), \max_t\left(1+\text{OPT}(i,t-1)+\text{OPT}(t+1,j-1)\right)\right)$$

- There are $O(n^2)$ sub-problems.
- How do we order them from "smallest" to "largest"?
- Note that computing $\text{OPT}(i,j)$ involves sub-problems $\text{OPT}(k,l)$ where $l-k < j-i$.

Dynamic Programming Algorithm

$$\mathrm{OPT}(i,j) = \max\left(\mathrm{OPT}(i,j-1), \max_t \left(1+\mathrm{OPT}(i,t-1)+\mathrm{OPT}(t+1,j-1)\right)\right)$$

- There are $O(n^2)$ sub-problems.
- How do we order them from “smallest” to “largest”?
- Note that computing $\mathrm{OPT}(i,j)$ involves sub-problems $\mathrm{OPT}(k,l)$ where $l-k < j-i$.

```
Initialize OPT(i, j) = 0 whenever i ≥ j − 4
For k = 5, 6, . . . , n − 1
  For i = 1, 2, . . . n − k
    Set j = i + k
    Compute OPT(i, j) using the recurrence in (6.13)
  Endfor
Endfor
Return OPT(1, n)
```

Dynamic Programming Algorithm

$$\text{OPT}(i,j) = \max\left(\text{OPT}(i,j-1), \max_t \left(1+\text{OPT}(i,t-1)+\text{OPT}(t+1,j-1)\right)\right)$$

- There are $O(n^2)$ sub-problems.
- How do we order them from "smallest" to "largest"?
- Note that computing $\text{OPT}(i,j)$ involves sub-problems $\text{OPT}(k,l)$ where $l - k < j - i$.

```
Initialize OPT(i, j) = 0 whenever i ≥ j − 4
For k = 5, 6, . . . , n − 1
  For i = 1, 2, . . . n − k
    Set j = i + k
    Compute OPT(i, j) using the recurrence in (6.13)
  Endfor
Endfor
Return OPT(1, n)
```

- Running time of the algorithm is $O(n^3)$.

Example of Algorithm

RNA sequence *ACCGGUAGU*

	$j = 6$	7	8	9
4	0	0	0	
3	0	0		
2	0			
$i = 1$				

Initial values

	$j = 6$	7	8	9
4	0	0	0	0
3	0	0	1	
2	0	0		
$i = 1$	1			

Filling in the values for $k = 5$

	$j = 6$	7	8	9
4	0	0	0	0
3	0	0	1	1
2	0	0	1	
$i = 1$	1	1		

Filling in the values for $k = 6$

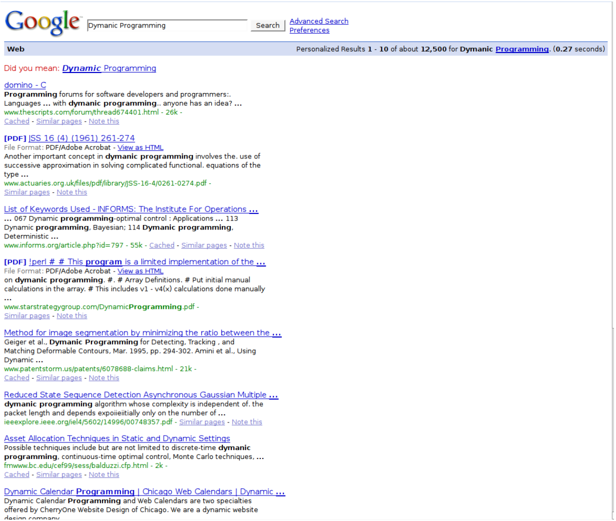
	$j = 6$	7	8	9
4	0	0	0	0
3	0	0	1	1
2	0	0	1	1
$i = 1$	1	1	1	

Filling in the values for $k = 7$

	$j = 6$	7	8	9
4	0	0	0	0
3	0	0	1	1
2	0	0	1	1
$i = 1$	1	1	1	2

Filling in the values for $k = 8$

Google Search for "Dymanic Programming"



- How do they know "Dynamic" and "Dymanic" are similar?

Sequence Similarity

- Given two strings, measure how similar they are.
- Given a database of strings and a query string, compute the string most similar to query in the database.
- Applications:
  - Online searches (Web, dictionary).
  - Spell-checkers.
  - Computational biology
  - Speech recognition.
  - Basis for Unix `diff`.

Defining Sequence Similarity

```
o-currance
occurrence
```

```
o-curr-ance
occurre-nce
```

```
abbbaa--bbbbaab
ababaaabbbba-b
```

Defining Sequence Similarity

```
o-currance
occurrence
```

```
o-curr-ance
occurre-nce
```

```
abbbaa--bbbbaab
ababaaabbbbba-b
```

- *Edit distance* model: how many changes must you to make to one string to transform it into another?
- Changes allowed are deleting a letter, adding a letter, changing a letter.

Edit Distance

```
abbbaa--bbbbaab
ababaaabbbbba-b
```

- Proposed by Needleman and Wunsch in the early 1970s.
- Input: two string $x = x_1 x_2 x_3 \ldots x_m$ and $y = y_1 y_2 \ldots y_n$.
- Sets $\{1, 2, \ldots, m\}$ and $\{1, 2, \ldots, n\}$ represent positions in x and y.

Edit Distance

```
abbbaa--bbbbaab
ababaaabbbbba-b
```

- Proposed by Needleman and Wunsch in the early 1970s.
- Input: two string $x = x_1x_2x_3 \ldots x_m$ and $y = y_1y_2 \ldots y_n$.
- Sets $\{1, 2, \ldots, m\}$ and $\{1, 2, \ldots, n\}$ represent positions in x and y.
- A *matching* of these sets is a set M of ordered pairs such that
  1. in each pair (i, j), $1 \leq i \leq m$ and $1 \leq j \leq m$ and
  2. no index from x (respectively, from y) appears as the first (respectively, second) element in more than one ordered pair.

Edit Distance

```
abbbaa--bbbbaab
ababaaabbbbba-b
```

- Proposed by Needleman and Wunsch in the early 1970s.
- Input: two string $x = x_1 x_2 x_3 \dots x_m$ and $y = y_1 y_2 \dots y_n$.
- Sets $\{1, 2, \dots, m\}$ and $\{1, 2, \dots, n\}$ represent positions in x and y.
- A *matching* of these sets is a set M of ordered pairs such that
  1. in each pair (i, j), $1 \leq i \leq m$ and $1 \leq j \leq m$ and
  2. no index from x (respectively, from y) appears as the first (respectively, second) element in more than one ordered pair.
- A matching M is an *alignment* if there are no "crossing pairs" in M: if $(i, j) \in M$ and $(i', j') \in M$ and $i < i'$ then $j < j'$.

Edit Distance

```
abbbaa--bbbbaab
ababaaabbbbba-b
```

- Proposed by Needleman and Wunsch in the early 1970s.
- Input: two string $x = x_1x_2x_3\dots x_m$ and $y = y_1y_2\dots y_n$.
- Sets $\{1, 2, \dots, m\}$ and $\{1, 2, \dots, n\}$ represent positions in x and y.
- A *matching* of these sets is a set M of ordered pairs such that
  1. in each pair (i, j), $1 \leq i \leq m$ and $1 \leq j \leq m$ and
  2. no index from x (respectively, from y) appears as the first (respectively, second) element in more than one ordered pair.
- A matching M is an *alignment* if there are no "crossing pairs" in M: if $(i, j) \in M$ and $(i', j') \in M$ and $i < i'$ then $j < j'$.
- Cost of an alignment is the sum of gap and mismatch penalties:
  Gap penalty Penalty $\delta > 0$ for every unmatched index.
  Mismatch penalty Penalty $\alpha_{x_iy_j} > 0$ if $(i, j) \in M$.

Edit Distance

```
abbbaa--bbbbaab
ababaaabbbbba-b
```

- Proposed by Needleman and Wunsch in the early 1970s.
- Input: two string $x = x_1 x_2 x_3 \ldots x_m$ and $y = y_1 y_2 \ldots y_n$.
- Sets $\{1, 2, \ldots, m\}$ and $\{1, 2, \ldots, n\}$ represent positions in x and y.
- A *matching* of these sets is a set M of ordered pairs such that
  1. in each pair (i, j), $1 \leq i \leq m$ and $1 \leq j \leq m$ and
  2. no index from x (respectively, from y) appears as the first (respectively, second) element in more than one ordered pair.
- A matching M is an *alignment* if there are no "crossing pairs" in M: if $(i, j) \in M$ and $(i', j') \in M$ and $i < i'$ then $j < j'$.
- Cost of an alignment is the sum of gap and mismatch penalties:
  Gap penalty Penalty $\delta > 0$ for every unmatched index.
  Mismatch penalty Penalty $\alpha_{x_i y_j} > 0$ if $(i, j) \in M$.
- Output: compute an alignment of minimal cost.

Dynamic Programming Approach

- Consider index $m \in x$ and index $n \in y$. Is $(m, n) \in M$?

Dynamic Programming Approach

- Consider index $m \in x$ and index $n \in y$. Is $(m, n) \in M$?
- Claim: $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.

Dynamic Programming Approach

- Consider index $m \in x$ and index $n \in y$. Is $(m, n) \in M$?
- Claim: $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.
- $OPT(i, j)$: cost of optimal alignment between $x = x_1 x_2 x_3 \dots x_i$ and $y = y_1 y_2 \dots y_j$.
  - $(i, j) \in M$:

Dynamic Programming Approach

- Consider index $m \in x$ and index $n \in y$. Is $(m, n) \in M$?
- Claim: $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.
- $OPT(i, j)$: cost of optimal alignment between $x = x_1 x_2 x_3 \ldots x_i$ and $y = y_1 y_2 \ldots y_j$.
  - $(i, j) \in M$: $\text{OPT}(i, j) = \alpha_{x_i y_j} + \text{OPT}(i - 1, j - 1)$.

Dynamic Programming Approach

- Consider index $m \in x$ and index $n \in y$. Is $(m, n) \in M$?
- Claim: $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.
- $OPT(i, j)$: cost of optimal alignment between $x = x_1 x_2 x_3 \ldots x_i$ and $y = y_1 y_2 \ldots y_j$.
  - $(i, j) \in M$: $\text{OPT}(i, j) = \alpha_{x_i y_j} + \text{OPT}(i - 1, j - 1)$.
  - i not matched:

Dynamic Programming Approach

- Consider index $m \in x$ and index $n \in y$. Is $(m, n) \in M$?
- Claim: $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.
- $OPT(i, j)$: cost of optimal alignment between $x = x_1x_2x_3 \ldots x_i$ and $y = y_1y_2 \ldots y_j$.
  - $(i, j) \in M$: $\text{OPT}(i, j) = \alpha_{x_iy_j} + \text{OPT}(i-1, j-1)$.
  - i not matched: $\text{OPT}(i, j) = \delta + \text{OPT}(i-1, j)$.

Dynamic Programming Approach

- Consider index $m \in x$ and index $n \in y$. Is $(m, n) \in M$?
- Claim: $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.
- $OPT(i, j)$: cost of optimal alignment between $x = x_1 x_2 x_3 \ldots x_i$ and $y = y_1 y_2 \ldots y_j$.
  - $(i, j) \in M$: $\text{OPT}(i, j) = \alpha_{x_i y_j} + \text{OPT}(i - 1, j - 1)$.
  - i not matched: $\text{OPT}(i, j) = \delta + \text{OPT}(i - 1, j)$.
  - j not matched: $\text{OPT}(i, j) = \delta + \text{OPT}(i, j - 1)$.

Dynamic Programming Approach

- Consider index $m \in x$ and index $n \in y$. Is $(m, n) \in M$?
- Claim: $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.
- $OPT(i, j)$: cost of optimal alignment between $x = x_1 x_2 x_3 \ldots x_i$ and $y = y_1 y_2 \ldots y_j$.
  - $(i, j) \in M$: $\text{OPT}(i, j) = \alpha_{x_i y_j} + \text{OPT}(i-1, j-1)$.
  - i not matched: $\text{OPT}(i, j) = \delta + \text{OPT}(i-1, j)$.
  - j not matched: $\text{OPT}(i, j) = \delta + \text{OPT}(i, j-1)$.

$$\text{OPT}(i, j) = \min\left(\alpha_{x_i y_j} + \text{OPT}(i-1, j-1), \delta + \text{OPT}(i-1, j), \delta + \text{OPT}(i, j-1)\right)$$

  - $(i, j) \in M$ if and only if minimum is achieved by the first term.
- What are the base cases?

Dynamic Programming Approach

- Consider index $m \in x$ and index $n \in y$. Is $(m, n) \in M$?
- Claim: $(m, n) \not\in M \Rightarrow m \in x$ not matched or $n \in y$ not matched.
- $OPT(i,j)$: cost of optimal alignment between $x = x_1 x_2 x_3 \ldots x_i$ and $y = y_1 y_2 \ldots y_j$.
  - $(i,j) \in M$: $\text{OPT}(i,j) = \alpha_{x_i y_j} + \text{OPT}(i-1, j-1)$.
  - i not matched: $\text{OPT}(i,j) = \delta + \text{OPT}(i-1, j)$.
  - j not matched: $\text{OPT}(i,j) = \delta + \text{OPT}(i, j-1)$.

$$\text{OPT}(i,j) = \min\left(\alpha_{x_i y_j} + \text{OPT}(i-1, j-1), \delta + \text{OPT}(i-1, j), \delta + \text{OPT}(i, j-1)\right)$$

  - $(i,j) \in M$ if and only if minimum is achieved by the first term.
- What are the base cases? $\text{OPT}(i, 0) = \text{OPT}(0, i) = i\delta$.

Dynamic Programming Algorithm

$$\text{OPT}(i,j) = \min\left(\alpha_{x_i y_j} + \text{OPT}(i-1,j-1), \delta + \text{OPT}(i-1,j), \delta + \text{OPT}(i,j-1)\right)$$

```
Alignment(X,Y)
  Array A[0...m, 0...n]
  Initialize A[i,0]=iδ for each i
  Initialize A[0,j]=jδ for each j
  For j=1,...,n
     For i=1,...,m
         Use the recurrence (6.16) to compute A[i,j]
     Endfor
  Endfor
  Return A[m,n]
```

Dynamic Programming Algorithm

$$\mathrm{OPT}(i,j) = \min\left(\alpha_{x_i y_j} + \mathrm{OPT}(i-1,j-1), \delta + \mathrm{OPT}(i-1,j), \delta + \mathrm{OPT}(i,j-1)\right)$$

```
Alignment(X,Y)
  Array A[0...m,0...n]
  Initialize A[i,0]=iδ for each i
  Initialize A[0,j]=jδ for each j
  For j=1,...,n
     For i=1,...,m
         Use the recurrence (6.16) to compute A[i,j]
     Endfor
  Endfor
  Return A[m,n]
```

- Running time is $O(mn)$. Space used in $O(mn)$.

Dynamic Programming Algorithm

$$\text{OPT}(i,j) = \min\left(\alpha_{x_i y_j} + \text{OPT}(i-1,j-1), \delta + \text{OPT}(i-1,j), \delta + \text{OPT}(i,j-1)\right)$$

```
Alignment(X,Y)
  Array A[0...m,0...n]
  Initialize A[i,0]=iδ for each i
  Initialize A[0,j]=jδ for each j
  For j=1,...,n
     For i=1,...,m
         Use the recurrence (6.16) to compute A[i,j]
     Endfor
  Endfor
  Return A[m,n]
```

- Running time is $O(mn)$. Space used in $O(mn)$.
- Can compute $\text{OPT}(m,n)$ in $O(mn)$ time and $O(m+n)$ space (Hirschberg 1975, Chapter 6.7).

Dynamic Programming Algorithm

$$\text{OPT}(i,j) = \min\left(\alpha_{x_i y_j} + \text{OPT}(i-1,j-1), \delta + \text{OPT}(i-1,j), \delta + \text{OPT}(i,j-1)\right)$$

```
Alignment(X,Y)
  Array A[0...m, 0...n]
  Initialize A[i,0] = iδ for each i
  Initialize A[0,j] = jδ for each j
  For j = 1,...,n
     For i = 1,...,m
         Use the recurrence (6.16) to compute A[i,j]
     Endfor
  Endfor
  Return A[m,n]
```

- Running time is $O(mn)$. Space used in $O(mn)$.
- Can compute $\text{OPT}(m,n)$ in $O(mn)$ time and $O(m+n)$ space (Hirschberg 1975, Chapter 6.7).
- Can compute *alignment* in the same bounds by combining dynamic programming with divide and conquer.

Graph-theoretic View of Sequence Alignment

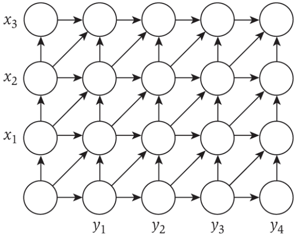


Figure 6.17 A graph-based picture of sequence alignment.

- Grid graph G_{xy}:
  - Rows labelled by symbols in x and columns labelled by symbols in y.
  - Edges from node (i, j) to $(i, j + 1)$), to $(i + 1, j)$, and to $(i + 1, j + 1)$.
  - Edges directed upward and to the right have cost δ.
  - Edge directed from (i, j) to $(i + 1, j + 1)$ has cost $alpha_{x_{i+1}y_{i+1}}$.

Graph-theoretic View of Sequence Alignment

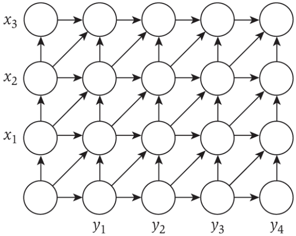


Figure 6.17 A graph-based picture of sequence alignment.

- Grid graph G_{xy}:
  - Rows labelled by symbols in x and columns labelled by symbols in y.
  - Edges from node (i, j) to $(i, j+1)$), to $(i+1, j)$, and to $(i+1, j+1)$.
  - Edges directed upward and to the right have cost δ.
  - Edge directed from (i, j) to $(i+1, j+1)$ has cost $alpha_{x_{i+1}y_{i+1}}$.
- *f(i, j)*: minimum cost of a path in G_{XY} from $(0, 0)$ to (i, j).
- Claim: $f(i, j) = \text{OPT}(i, j)$ and diagonal edges in the shortest path are the matched pairs in the alignment.

Motivation

- Computational finance:
  - Each node is a financial agent.
  - The cost c_{uv} of an edge (u, v) is the cost of a transaction in which we buy from agent u and sell to agent v.
  - Negative cost corresponds to a profit.
- Internet routing protocols
  - Dijkstra's algorithm needs knowledge of the entire network.
  - Routers only know which other routers they are connected to.
  - Algorithm for shortest paths with negative edges is decentralised.
  - We will not study this algorithm in the class. See Chapter 6.9.

Problem Statement

- Input: a directed graph $G = (V, E)$ with a cost function $c : E \rightarrow \mathbb{R}$, i.e., c_{uv} is the cost of the edge $(u, v) \in E$.
- A *negative cycle* is a directed cycle whose edges have a total cost that is negative.
- Two related problems:
  1. If G has no negative cycles, find the *shortest s-t path*: a path of from source s to destination t with minimum total cost.
  2. Does G have a *negative cycle*?

Problem Statement

- Input: a directed graph $G = (V, E)$ with a cost function $c : E \to \mathbb{R}$, i.e., c_{uv} is the cost of the edge $(u, v) \in E$.
- A *negative cycle* is a directed cycle whose edges have a total cost that is negative.
- Two related problems:
  1. If G has no negative cycles, find the *shortest s-t path*: a path of from source s to destination t with minimum total cost.
  2. Does G have a *negative cycle*?

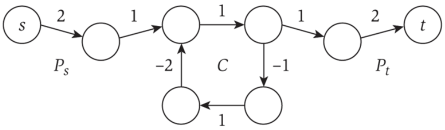


Figure 6.20 In this graph, one can find s-t paths of arbitrarily negative cost (by going around the cycle C many times).

Approaches for Shortest Path Algorithm

1. Dijsktra's algorithm.
2. Add some large constant to each edge.

Approaches for Shortest Path Algorithm

1. Dijsktra's algorithm. Computes incorrect answers because it is greedy.
2. Add some large constant to each edge. Computes incorrect answers because the minimum cost path changes.

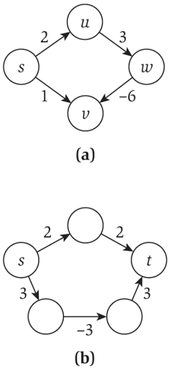


Figure 6.21 (a) With negative edge costs, Dijkstra's Algorithm can give the wrong answer for the Shortest-Path Problem. (b) Adding 3 to the cost of each edge will make all edges nonnegative, but it will change the identity of the shortest *s*-*t* path.

Dynamic Programming Approach

- Assume G has no negative cycles.
- Claim: There is a shortest path from s to t that is *simple* (does not repeat a node)

Dynamic Programming Approach

- Assume G has no negative cycles.
- Claim: There is a shortest path from s to t that is *simple* (does not repeat a node) and hence has at most $n - 1$ edges.

Dynamic Programming Approach

- Assume G has no negative cycles.
- Claim: There is a shortest path from s to t that is *simple* (does not repeat a node) and hence has at most $n - 1$ edges.
- How do we define sub-problems?

Dynamic Programming Approach

- Assume G has no negative cycles.
- Claim: There is a shortest path from s to t that is *simple* (does not repeat a node) and hence has at most $n-1$ edges.
- How do we define sub-problems?
  - Since the shortest s-t path has $\leq n-1$ edges, let us consider how we can reach t using i edges, for different values of i.
  - Since we do not know which nodes will be in the shortest s-t path, let us consider how we can reach t from each node in V.

Dynamic Programming Approach

- Assume G has no negative cycles.
- Claim: There is a shortest path from s to t that is *simple* (does not repeat a node) and hence has at most $n-1$ edges.
- How do we define sub-problems?
  - Since the shortest s-t path has $\leq n-1$ edges, let us consider how we can reach t using i edges, for different values of i.
  - Since we do not know which nodes will be in the shortest s-t path, let us consider how we can reach t from each node in V.
- Sub-problems defined by varying the number of edges in the shortest path and by varying the starting node in the shortest path.

Dynamic Programming Sub-problems

- $OPT(i, v)$: minimum cost of a v-t path that uses at most i edges.
- t is not explicitly mentioned in the sub-problems.
- Goal is to compute $\text{OPT}(n-1, s)$.

Dynamic Programming Sub-problems

- $OPT(i, v)$: minimum cost of a v-t path that uses at most i edges.
- t is not explicitly mentioned in the sub-problems.
- Goal is to compute $\text{OPT}(n-1, s)$.

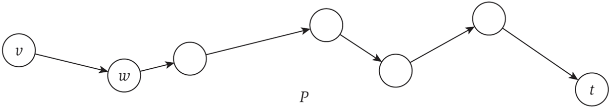


Figure 6.22 The minimum-cost path P from v to t using at most i edges.

- Let P be the optimal path whose cost is $\text{OPT}(i, v)$.

Dynamic Programming Sub-problems

- $OPT(i, v)$: minimum cost of a v-t path that uses at most i edges.
- t is not explicitly mentioned in the sub-problems.
- Goal is to compute $\text{OPT}(n-1, s)$.

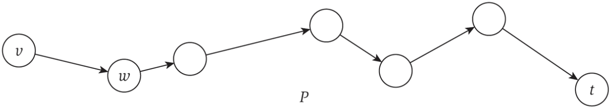


Figure 6.22 The minimum-cost path P from v to t using at most i edges.

- Let P be the optimal path whose cost is $\text{OPT}(i, v)$.
  1. If P actually uses $i-1$ edges, then $\text{OPT}(i, v) = \text{OPT}(i-1, v)$.
  2. If first node on P is w, then $\text{OPT}(i, v) = c_{vw} + \text{OPT}(i-1, w)$.

Dynamic Programming Sub-problems

- $OPT(i, v)$: minimum cost of a v-t path that uses at most i edges.
- t is not explicitly mentioned in the sub-problems.
- Goal is to compute $\text{OPT}(n-1, s)$.

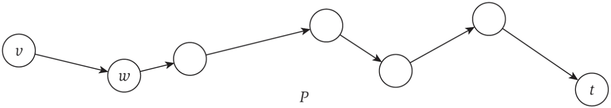


Figure 6.22 The minimum-cost path P from v to t using at most i edges.

- Let P be the optimal path whose cost is $\text{OPT}(i, v)$.
  1. If P actually uses $i-1$ edges, then $\text{OPT}(i, v) = \text{OPT}(i-1, v)$.
  2. If first node on P is w, then $\text{OPT}(i, v) = c_{vw} + \text{OPT}(i-1, w)$.

$$\text{OPT}(i, v) = \min\left(\text{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \text{OPT}(i-1, w)\right)\right)$$

Alternate Dynamic Programming Formulation

- $OPT(i, v)$: minimum cost of a v-t path that uses exactly i edges. Goal is to compute

Alternate Dynamic Programming Formulation

- $OPT(i, v)$: minimum cost of a v-t path that uses exactly i edges. Goal is to compute

$$\min_{i=1}^{n-1} \text{OPT}(i, s).$$

Alternate Dynamic Programming Formulation

- $OPT(i, v)$: minimum cost of a v-t path that uses exactly i edges. Goal is to compute

$$\min_{i=1}^{n-1} \mathrm{OPT}(i, s).$$

- Let P be the optimal path whose cost is $\mathrm{OPT}(i, v)$.

Alternate Dynamic Programming Formulation

- $OPT(i, v)$: minimum cost of a v-t path that uses exactly i edges. Goal is to compute

$$\min_{i=1}^{n-1} \text{OPT}(i, s).$$

- Let P be the optimal path whose cost is $\text{OPT}(i, v)$.
  - If first node on P is w, then $\text{OPT}(i, v) = c_{vw} + \text{OPT}(i-1, w)$.

Alternate Dynamic Programming Formulation

- $OPT(i, v)$: minimum cost of a v-t path that uses exactly i edges. Goal is to compute

$$\min_{i=1}^{n-1} \text{OPT}(i, s).$$

- Let P be the optimal path whose cost is $\text{OPT}(i, v)$.
  - If first node on P is w, then $\text{OPT}(i, v) = c_{vw} + \text{OPT}(i-1, w)$.

$$\text{OPT}(i, v) = \min_{w \in V} \left(c_{vw} + \text{OPT}(i-1, w)\right)$$

Alternate Dynamic Programming Formulation

- *OPT*(i, v): minimum cost of a v-t path that uses exactly i edges. Goal is to compute

$$\min_{i=1}^{n-1} \mathrm{OPT}(i, s).$$

- Let P be the optimal path whose cost is $\mathrm{OPT}(i, v)$.
  - If first node on P is w, then $\mathrm{OPT}(i, v) = c_{vw} + \mathrm{OPT}(i-1, w)$.

$$\mathrm{OPT}(i, v) = \min_{w \in V} \left(c_{vw} + \mathrm{OPT}(i-1, w)\right)$$

- Compare the recurrence above to the previous recurrence:

$$\mathrm{OPT}(i, v) = \min\left(\mathrm{OPT}(i-1, v), \min_{w \in V} \left(c_{vw} + \mathrm{OPT}(i-1, w)\right)\right)$$

Bellman-Ford Algorithm

$$\text{OPT}(i, v) = \min\left(\text{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \text{OPT}(i-1, w)\right)\right)$$

```
Shortest-Path(G, s, t)
  n = number of nodes in G
  Array M[0...n-1, V]
  Define M[0, t] = 0 and M[0, v] = ∞ for all other v ∈ V
  For i = 1, ..., n-1
    For v ∈ V in any order
      Compute M[i, v] using the recurrence (6.23)
    Endfor
  Endfor
  Return M[n-1, s]
```

Bellman-Ford Algorithm

$$\text{OPT}(i, v) = \min\left(\text{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \text{OPT}(i-1, w)\right)\right)$$

```
Shortest-Path(G, s, t)
  n = number of nodes in G
  Array M[0...n-1, V]
  Define M[0,t]=0 and M[0,v]=∞ for all other v∈V
  For i=1,...,n-1
    For v∈V in any order
      Compute M[i,v] using the recurrence (6.23)
    Endfor
  Endfor
  Return M[n-1,s]
```

- Space used is $O(n^2)$. Running time is $O(n^3)$.
- If shortest path uses k edges, we can recover it in $O(kn)$ time by tracing back through smaller sub-problems.

An Improved Bound on the Running Time

- Suppose G has n nodes and $m \ll \binom{n}{2}$ edges. Can we demonstrate a better upper bound on the running time?

An Improved Bound on the Running Time

- Suppose G has n nodes and $m \ll \binom{n}{2}$ edges. Can we demonstrate a better upper bound on the running time?

$$M[i, v] = \min \left(M[i-1, v], \min_{w \in V} \left(c_{vw} + M[i-1, w] \right) \right)$$

An Improved Bound on the Running Time

- Suppose G has n nodes and $m \ll \binom{n}{2}$ edges. Can we demonstrate a better upper bound on the running time?

$$M[i, v] = \min\left(M[i-1, v], \min_{w \in V}\left(c_{vw} + M[i-1, w]\right)\right)$$

- w only needs to range over neighbours of v.
- If n_v is the number of neighbours of v, then in each round, we spend time equal to

$$\sum_{v \in V} n_v =$$

An Improved Bound on the Running Time

- Suppose G has n nodes and $m \ll \binom{n}{2}$ edges. Can we demonstrate a better upper bound on the running time?

$$M[i, v] = \min\left(M[i-1, v], \min_{w \in V}\left(c_{vw} + M[i-1, w]\right)\right)$$

- w only needs to range over neighbours of v.
- If n_v is the number of neighbours of v, then in each round, we spend time equal to

$$\sum_{v \in V} n_v = m.$$

- The total running time is $O(mn)$.

Improving the Memory Requirements

$$M[i, v] = \min\left(M[i-1, v], \min_{w \in V}\left(c_{vw} + M[i-1, w]\right)\right)$$

- The algorithm uses $O(n^2)$ space to store the array M.

Improving the Memory Requirements

$$M[i, v] = \min \left(M[i-1, v], \min_{w \in V} \left(c_{vw} + M[i-1, w] \right) \right)$$

- The algorithm uses $O(n^2)$ space to store the array M.
- Observe that $M[i, v]$ depends only on $M[i-1, *]$ and no other indices.

Improving the Memory Requirements

$$M[i, v] = \min\left(M[i-1, v], \min_{w \in V}\left(c_{vw} + M[i-1, w]\right)\right)$$

- The algorithm uses $O(n^2)$ space to store the array M.
- Observe that $M[i, v]$ depends only on $M[i-1, *]$ and no other indices.
- Modified algorithm:
  1. Maintain two arrays M and N indexed over V.
  2. At the beginning of each iteration, copy M into N.
  3. To update M, use

$$M[v] = \min\left(N[v], \min_{w \in V}\left(c_{vw} + N[w]\right)\right)$$

Improving the Memory Requirements

$$M[i, v] = \min\left(M[i-1, v], \min_{w \in V}\left(c_{vw} + M[i-1, w]\right)\right)$$

- The algorithm uses $O(n^2)$ space to store the array M.
- Observe that $M[i, v]$ depends only on $M[i-1, *]$ and no other indices.
- Modified algorithm:
  1. Maintain two arrays M and N indexed over V.
  2. At the beginning of each iteration, copy M into N.
  3. To update M, use

$$M[v] = \min\left(N[v], \min_{w \in V}\left(c_{vw} + N[w]\right)\right)$$

- Claim: at the beginning of iteration i, M stores values of $\text{OPT}(i-1, v)$ for all nodes $v \in V$.
- Space used is $O(n)$.

Computing the Shortest Path: Algorithm

$$M[v] = \min\left(N[v], \min_{w \in V}\left(c_{vw} + N[w]\right)\right)$$

- How can we recover the shortest path that has cost $M[v]$?

Computing the Shortest Path: Algorithm

$$M[v] = \min\left(N[v], \min_{w \in V}\left(c_{vw} + N[w]\right)\right)$$

- How can we recover the shortest path that has cost $M[v]$?
- For each node v, maintain $f(v)$, the first node after v in the current shortest path from v to t.
- To maintain $f(v)$, if we ever set $M[v]$ to $\min_{w \in V}\left(c_{vw} + N[w]\right)$, set $f(v)$ to be the node w that attains this minimum.
- At the end, follow $f(v)$ pointers from s to t.

Computing the Shortest Path: Correctness

- *Pointer graph* $P(V, F)$: each edge in F is $(v, f(v))$.
  - Can P have cycles?
  - Is there a path from s to t in P?
  - Can there be multiple paths s to t in P?
  - Which of these is the shortest path?

Computing the Shortest Path: Cycles in P

$$M[v] = \min\left(N[v], \min_{w \in V}\left(c_{vw} + N[w]\right)\right)$$

- Claim: If P has a cycle C, then C has negative cost.

Computing the Shortest Path: Cycles in P

$$M[v] = \min\left(N[v], \min_{w \in V}\left(c_{vw} + N[w]\right)\right)$$

- Claim: If P has a cycle C, then C has negative cost.
  - If we set $f(v) = w$ at any time, then $M[v] \geq c_{vw} + M[w]$ after that time.

Computing the Shortest Path: Cycles in P

$$M[v] = \min\left(N[v], \min_{w \in V}\left(c_{vw} + N[w]\right)\right)$$

- Claim: If P has a cycle C, then C has negative cost.
  - If we set $f(v) = w$ at any time, then $M[v] \geq c_{vw} + M[w]$ after that time.
  - Let $v_1, v_2, \ldots v_k$ be the nodes in C and assume that (v_k, v_1) is the last edge to have been added.
  - What is the situation just before this addition?

Computing the Shortest Path: Cycles in P

$$M[v] = \min\left(N[v], \min_{w \in V}\left(c_{vw} + N[w]\right)\right)$$

- Claim: If P has a cycle C, then C has negative cost.
  - If we set $f(v) = w$ at any time, then $M[v] \geq c_{vw} + M[w]$ after that time.
  - Let $v_1, v_2, \ldots v_k$ be the nodes in C and assume that (v_k, v_1) is the last edge to have been added.
  - What is the situation just before this addition?
  - $M[v_i] \geq c_{v_i v_{i+1}} + M[v_{i+1}]$, for all $1 \leq i < k - 1$.
  - $M[v_k] > c_{v_k v_1} + M[v_1]$.

Computing the Shortest Path: Cycles in P

$$M[v] = \min\left(N[v], \min_{w \in V}\left(c_{vw} + N[w]\right)\right)$$

- Claim: If P has a cycle C, then C has negative cost.
  - If we set $f(v) = w$ at any time, then $M[v] \geq c_{vw} + M[w]$ after that time.
  - Let $v_1, v_2, \ldots v_k$ be the nodes in C and assume that (v_k, v_1) is the last edge to have been added.
  - What is the situation just before this addition?
  - $M[v_i] \geq c_{v_i v_{i+1}} + M[v_{i+1}]$, for all $1 \leq i < k - 1$.
  - $M[v_k] > c_{v_k v_1} + M[v_1]$.
  - Adding all these inequalities, $0 > \sum_{i=1}^{k-1} c_{v_i v_{i+1}} + c_{v_k v_1}$.

Computing the Shortest Path: Cycles in P

$$M[v] = \min\left(N[v], \min_{w \in V}\left(c_{vw} + N[w]\right)\right)$$

- Claim: If P has a cycle C, then C has negative cost.
  - If we set $f(v) = w$ at any time, then $M[v] \geq c_{vw} + M[w]$ after that time.
  - Let $v_1, v_2, \ldots v_k$ be the nodes in C and assume that (v_k, v_1) is the last edge to have been added.
  - What is the situation just before this addition?
  - $M[v_i] \geq c_{v_i v_{i+1}} + M[v_{i+1}]$, for all $1 \leq i < k - 1$.
  - $M[v_k] > c_{v_k v_1} + M[v_1]$.
  - Adding all these inequalities, $0 > \sum_{i=1}^{k-1} c_{v_i v_{i+1}} + c_{v_k v_1}$.
- Corollary: if G has no negative cycles that P does not either.

Computing the Shortest Path: Paths in P

- Let P be the pointer graph upon termination of the algorithm.
- Consider the path P_v in P obtained by following the pointers from v to $f(v) = v_1$, to $f(v_1) = v_2$, and so on.

Computing the Shortest Path: Paths in P

- Let P be the pointer graph upon termination of the algorithm.
- Consider the path P_v in P obtained by following the pointers from v to $f(v) = v_1$, to $f(v_1) = v_2$, and so on.
- Claim: P_v terminates at t.

Computing the Shortest Path: Paths in P

- Let P be the pointer graph upon termination of the algorithm.
- Consider the path P_v in P obtained by following the pointers from v to $f(v) = v_1$, to $f(v_1) = v_2$, and so on.
- Claim: P_v terminates at t.
- Claim: P_v is the shortest path in G from v to t.

Bellman-Ford Algorithm: Early Termination

$$M[v] = \min\left(N[v], \min_{w \in V}\left(c_{vw} + N[w]\right)\right)$$

- In general, after i iterations, the path whose length is $M[v]$ may have many more than i edges.

Bellman-Ford Algorithm: Early Termination

$$M[v] = \min\left(N[v], \min_{w \in V}\left(c_{vw} + N[w]\right)\right)$$

- In general, after i iterations, the path whose length is $M[v]$ may have many more than i edges.
- Early termination: If M equals N after processing all the nodes, we have computed all the shortest paths to t.