

CSC373

Weeks 4,5: Dynamic Programming

Nisarg Shah

Recap

- Greedy Algorithms
 - Interval scheduling
 - Interval partitioning
 - Minimizing lateness
 - Huffman encoding
 - ...

5.4 Warning: Greed is Stupid

If we're very very very very lucky, we can bypass all the recurrences and tables and so forth, and solve the problem using a *greedy* algorithm. The general greedy strategy is look for the best first step, take it, and then continue. While this approach seems very natural, it almost never works; optimization problems that can be solved correctly by a greedy algorithm are *very* rare. Nevertheless, for many problems that should be solved by dynamic programming, many students' first intuition is to apply a greedy strategy.

For example, a greedy algorithm for the edit distance problem might look for the longest common substring of the two strings, match up those substrings (since those substitutions don't cost anything), and then recursively look for the edit distances between the left halves and right halves of the strings. If there is no common substring—that is, if the two strings have no characters in common—the edit distance is clearly the length of the larger string. If this sounds like a stupid hack to you, pat yourself on the back. It isn't even *close* to the correct solution.

Everyone should tattoo the following sentence on the back of their hands, right under all the rules about logarithms and big-Oh notation:

**Greedy algorithms never work!
Use dynamic programming instead!**

What, never?

No, never!

What, *never*?

Well... hardly ever.⁶

Jeff Erickson on greedy algorithms...

The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was secretary of Defense, and he actually had a pathological fear and hatred of the word 'research'. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term 'research' in his presence. You can imagine how he felt, then, about the term 'mathematical'. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose?

— Richard Bellman, on the origin of his term 'dynamic programming' (1984)



Richard Bellman's quote from Jeff Erickson's book

Dynamic Programming

- **Outline**

- Breaking the problem down into simpler subproblems, solve each subproblem just once, and store their solutions.
- The next time the same subproblem occurs, instead of recomputing its solution, simply look up its previously computed solution.
- Hopefully, we save a lot of computation at the expense of modest increase in storage space.
- Also called “**memoization**”

- **How is this different from divide & conquer?**

Weighted Interval Scheduling

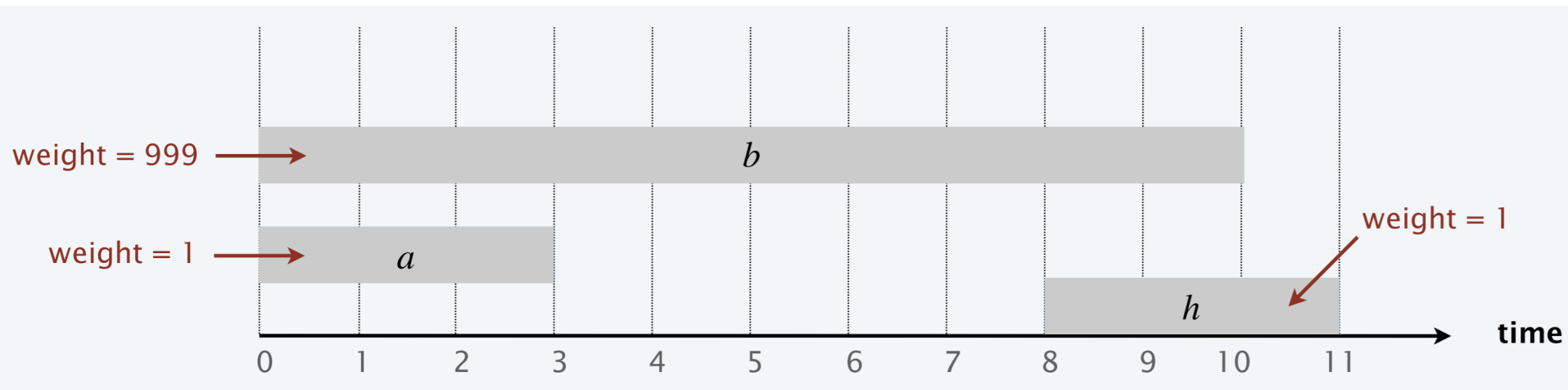
- **Problem**

- Job j starts at time s_j and finishes at time f_j
- Each job j has a weight w_j
- Two jobs are compatible if they don't overlap
- **Goal:** find a set S of mutually compatible jobs with highest total weight $\sum_{j \in S} w_j$

- Recall: If all $w_j = 1$, then this is simply the interval scheduling problem from last week
 - Greedy algorithm based on earliest finish time ordering was optimal for this case

Recall: Interval Scheduling

- What if we simply try to use it again?
 - Fails spectacularly!



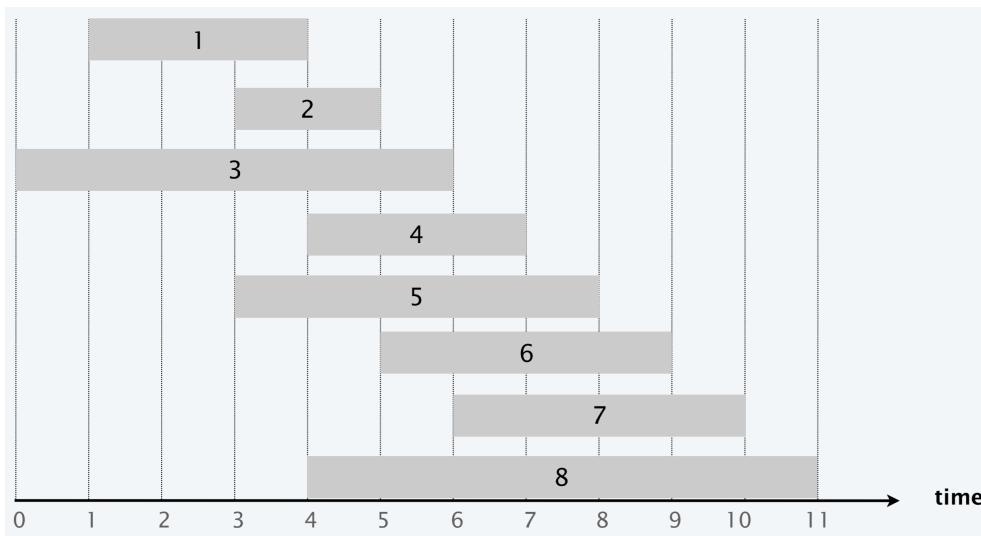
Weighted Interval Scheduling

- What if we use other orderings?
 - By weight: choose jobs with highest w_j first
 - Maximum weight per time: choose jobs with highest $w_j/(f_j - s_j)$ first
 - ...
- None of them work!
 - They're arbitrarily worse than the optimal solution
 - In fact, under a certain formalization, “no greedy algorithm” can produce any “decent approximation” in the worst case (beyond this course!)

Weighted Interval Scheduling

- **Convention**

- Jobs are sorted by finish time: $f_1 \leq f_2 \leq \dots \leq f_n$
- $p[j] =$ largest $i < j$ such that job i is compatible with job j (i.e., $f_i \leq s_j$)
 - Jobs $1, \dots, i$ are compatible with j , but jobs $i + 1, \dots, j - 1$ aren't
 - $p[j]$ can be computed via binary search



E.g.,

$$p[8] = 1,$$

$$p[7] = 3,$$

$$p[2] = 0,$$

...

Weighted Interval Scheduling

- The DP approach

- Let OPT be an optimal solution
- Take cases based on the “last decision”
- Two options for job n :
 - Option 1: Job n is in OPT
 - OPT *can't* use incompatible jobs $\{p[n] + 1, \dots, n - 1\}$
 - OPT *must* use an optimal subset of jobs from $\{1, \dots, p[n]\}$
 - Option 2: Job n is not in OPT
 - OPT *must* use an optimal subset of jobs from $\{1, \dots, n - 1\}$
- OPT *must* be the best of the two options
- Solve both subproblems, take the best

Weighted Interval Scheduling

- The DP approach

- $OPT(j)$ = max total weight of compatible jobs from $\{1, \dots, j\}$

- Base case: $OPT(0) = 0$

- Two cases regarding job j :

- Job j is selected: optimal weight is $w_j + OPT(p[j])$

- Job j is not selected: optimal weight is $OPT(j - 1)$

- Bellman equation:

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{OPT(j - 1), w_j + OPT(p[j])\} & \text{if } j > 0 \end{cases}$$

Brute Force Solution

BRUTE-FORCE ($n, s_1, \dots, s_n, f_1, \dots, f_n, w_1, \dots, w_n$)

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \dots \leq f_n$.

Compute $p[1], p[2], \dots, p[n]$ via binary search.

RETURN COMPUTE-OPT(n).

COMPUTE-OPT(j)

IF ($j = 0$)

 RETURN 0.

ELSE

 RETURN $\max \{ \text{COMPUTE-OPT}(j-1), w_j + \text{COMPUTE-OPT}(p[j]) \}$.

Brute Force Solution

COMPUTE-OPT(j)

IF ($j = 0$)

 RETURN 0.

ELSE

 RETURN $\max \{ \text{COMPUTE-OPT}(j-1), w_j + \text{COMPUTE-OPT}(p[j]) \}$.

- Q: Worst-case running time of COMPUTE-OPT(n)?
 - a) $\Theta(n)$
 - b) $\Theta(n \log n)$
 - c) $\Theta(1.618^n)$
 - d) $\Theta(2^n)$

Dynamic Programming

- **Why is the runtime high?**
 - Only n possible inputs to the algorithm
 - But some inputs are being solved again and again!
 - E.g., if $p[5] = 3$, then $\text{Compute-OPT}(5)$ calls $\text{Compute-OPT}(4)$ and $\text{Compute-OPT}(3)$
 - But $\text{Compute-OPT}(4)$ in turn calls $\text{Compute-OPT}(3)$ again
- **Memoization** trick
 - Simply remember what you've already computed, and re-use the answer if needed in future

Dynamic Program: Top-Down

- Let's store COMPUTE-OPT(j) in $M[j]$

TOP-DOWN($n, s_1, \dots, s_n, f_1, \dots, f_n, w_1, \dots, w_n$)

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \dots \leq f_n$.

Compute $p[1], p[2], \dots, p[n]$ via binary search.

$M[0] \leftarrow 0$.  global array

RETURN M-COMPUTE-OPT(n).

M-COMPUTE-OPT(j)

IF ($M[j]$ is uninitialized)

$M[j] \leftarrow \max \{ \text{M-COMPUTE-OPT}(j-1), w_j + \text{M-COMPUTE-OPT}(p[j]) \}$.

RETURN $M[j]$.

Dynamic Program: Top-Down

- **Claim:** This memoized version takes $O(n \log n)$ time
 - Sorting by finish time: $O(n \log n)$
 - Computing $p[j]$ -s: $O(n \log n)$
 - For each j , at most one of the calls to M-Compute-OPT(j) will make two recursive calls to M-Compute-OPT
 - At most $O(n)$ total calls to M-Compute-OPT
 - Every call to M-Compute-OPT takes $O(1)$ time, not considering the time spent within its recursive calls
 - Hence, the initial call, M-Compute-OPT(n), finishes in $O(n)$ time
 - Overall time is $O(n \log n)$

Dynamic Program: Bottom-Up

- Find an order in which to call the functions so that the sub-solutions are ready when needed

BOTTOM-UP($n, s_1, \dots, s_n, f_1, \dots, f_n, w_1, \dots, w_n$)

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \dots \leq f_n$.

Compute $p[1], p[2], \dots, p[n]$.

$M[0] \leftarrow 0$.

previously computed values

FOR $j = 1$ **TO** n

$M[j] \leftarrow \max \{ M[j-1], w_j + M[p[j]] \}$.

Top-Down vs Bottom-Up

- **Top-Down may be preferred...**
 - ...when not all sub-solutions need to be computed on some inputs
 - ...because one does not need to think of the “right order” in which to compute sub-solutions
- **Bottom-Up may be preferred...**
 - ...when all sub-solutions will anyway need to be computed
 - ...because it is faster as it prevents recursive call overheads and unnecessary random memory accesses
 - ...because sometimes we can free-up memory early
 - ...makes the runtime analysis a bit easier to think about

Optimal Solution

- This approach gave us the optimal value
- **What about the actual solution (subset of jobs)?**
 - Idea: Maintain the optimal value *and an optimal solution*
 - So, we compute two quantities:

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{OPT(j-1), w_j + OPT(p[j])\} & \text{if } j > 0 \end{cases}$$

$$S(j) = \begin{cases} \emptyset & \text{if } j = 0 \\ S(j-1) & \text{if } j > 0 \wedge OPT(j-1) \geq w_j + OPT(p[j]) \\ \{j\} \cup S(p[j]) & \text{if } j > 0 \wedge OPT(j-1) < w_j + OPT(p[j]) \end{cases}$$

Optimal Solution

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{OPT(j-1), w_j + OPT(p[j])\} & \text{if } j > 0 \end{cases}$$

$$S(j) = \begin{cases} \emptyset & \text{if } j = 0 \\ S(j-1) & \text{if } j > 0 \wedge OPT(j-1) \geq w_j + OPT(p[j]) \\ \{j\} \cup S(p[j]) & \text{if } j > 0 \wedge OPT(j-1) < w_j + OPT(p[j]) \end{cases}$$

This works with both top-down and bottom-up implementations.

We can compute OPT and S simultaneously, or compute OPT first and then compute S .

Question: What is the overall running time and space complexity?

Optimal Solution

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{OPT(j-1), w_j + OPT(p[j])\} & \text{if } j > 0 \end{cases}$$

$$S(j) = \begin{cases} \perp & \text{if } j = 0 \\ N & \text{if } j > 0 \wedge OPT(j-1) \geq w_j + OPT(p[j]) \\ Y & \text{if } j > 0 \wedge OPT(j-1) < w_j + OPT(p[j]) \end{cases}$$

- Save space by storing only one bit per j = which option was used
- To reconstruct the optimal solution, start with $j = n$
 - If $S(j) = N$, update $j \leftarrow j - 1$
 - If $S(j) = Y$, add j to the solution and update $j \leftarrow p[j]$
 - If $j = 0$, stop

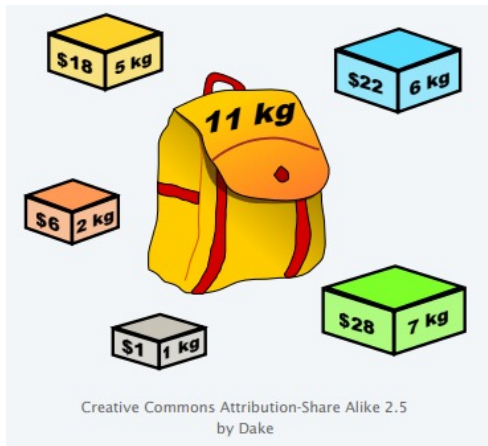
Optimal Substructure Property

- Dynamic programming applies well to problems that have **optimal substructure property**
 - Optimal solution to a problem can be computed easily given optimal solution to subproblems
- **Recall: divide-and-conquer also uses this property**
 - Divide-and-conquer is a special case in which the subproblems don't "overlap"
 - So, there's no need for memoization
 - In dynamic programming, two of the subproblems may in turn require access to solution to the same subproblem

Knapsack Problem

- **Problem**

- n items: item i provides value $v_i > 0$ and has weight $w_i > 0$
- Knapsack has weight capacity W
- Assumption: W , v_i -s, and w_i -s are all integers
- **Goal:** pack the knapsack with a subset of items with highest total value subject to their total weight being at most W



| i | v_i | w_i |
|-----|-------|-------|
| 1 | \$1 | 1 kg |
| 2 | \$6 | 2 kg |
| 3 | \$18 | 5 kg |
| 4 | \$22 | 6 kg |
| 5 | \$28 | 7 kg |

knapsack instance
(weight limit $W = 11$)

A First Attempt

- Let $OPT(w)$ = maximum value we can pack with a knapsack of capacity w
 - **Goal:** Compute $OPT(W)$
 - **Claim:** $OPT(w)$ must use at least one job j with weight $\leq w$ and then optimally pack the remaining capacity of $w - w_j$
 - Let $w^* = \min_j w_j$
 - $OPT(w) = \begin{cases} 0 & \text{if } w < w^* \\ \max_{j:w_j \leq w} v_j + OPT(w - w_j) & \text{if } w \geq w^* \end{cases}$
- **This is wrong!**
 - It might use an item more than once!

A Refined Attempt

- $OPT(i, w)$ = maximum value we can pack using only items $1, \dots, i$ in a knapsack of capacity w
 - **Goal:** Compute $OPT(n, W)$
- Consider item i
 - If $w_i > w$, then we can't choose i . Use $OPT(i - 1, w)$
 - If $w_i \leq w$, there are two cases:
 - If we choose i , the best is $v_i + OPT(i - 1, w - w_i)$
 - If we don't choose i , the best is $OPT(i - 1, w)$

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i - 1, w) & \text{if } w_i > w \\ \max \{ OPT(i - 1, w), v_i + OPT(i - 1, w - w_i) \} & \text{otherwise} \end{cases}$$

Running Time

- Consider possible evaluations $OPT(i, w)$
 - $i \in \{1, \dots, n\}$
 - $w \in \{1, \dots, W\}$ (recall weights and capacity are integers)
 - There are $O(n \cdot W)$ possible evaluations of OPT
 - Each is evaluated at most once (memoization / bottom-up)
 - Each takes $O(1)$ time to evaluate
 - The total running time is $O(n \cdot W)$
- **Q: Is this polynomial in the input size?**
 - A: No! But it's **pseudo-polynomial**.
 - Recall the inputs: $W, v_1, \dots, v_n, w_1, \dots, w_n$
 - Time should be polynomial in $\log W + \sum_{i=1}^n (\log v_i + \log w_i)$

What if...?

- If we were told that $W = \text{poly}(n)$...
 - That is, the value of W , and not its number of bits, is polynomially bounded in the input length
 - Then, this algorithm would run in polynomial time
- Q: What if, instead of the weights being small integers, we are told that the values are small integers?
 - Then we can use a different dynamic programming approach!

A Different DP

- $OPT(i, v)$ = minimum capacity needed to pack a total value of at least v using items $1, \dots, i$
 - **Goal:** Compute $\max\{v : OPT(n, v) \leq W\}$
- Consider item i
 - If we choose i , we need capacity $w_i + OPT(i - 1, v - v_i)$
 - If we don't choose i , we need capacity $OPT(i - 1, v)$

$$OPT(i, v) = \begin{cases} 0 & \text{if } v \leq 0 \\ \infty & \text{if } v > 0, i = 0 \\ \min \left\{ \begin{array}{l} w_i + OPT(i - 1, v - v_i) \\ OPT(i - 1, v) \end{array} \right\} & \text{if } v > 0, i > 0 \end{cases}$$

A Different DP

- $OPT(i, v)$ = minimum capacity needed to pack a total value of at least v using items $1, \dots, i$
 - **Goal:** Compute $\max\{v : OPT(n, v) \leq W\}$
 - This approach has running time $O(n \cdot V)$, where $V = v_1 + \dots + v_n$
 - So, we can get $O(n \cdot W)$ or $O(n \cdot V)$, whichever is smaller
- Can we remove the dependence on both V and W ?
 - Not likely.
 - Knapsack problem is NP-complete (we'll see later).

FPTAS

- While we cannot hope to solve the problem exactly in time $O(\text{poly}(n, \log W, \log V))$...
 - For any $\epsilon > 0$, we can get a value that is within $1 + \epsilon$ multiplicative factor of the optimal value in time $O\left(\text{poly}\left(n, \log W, \log V, \frac{1}{\epsilon}\right)\right)$
 - Such algorithms are known as fully polynomial-time approximation scheme (FPTAS)
 - **Core idea behind FPTAS for knapsack:**
 - Approximate all weights and values up to the desired precision
 - Solve knapsack on approximate input using DP

Single-Source Shortest Paths

- **Problem**

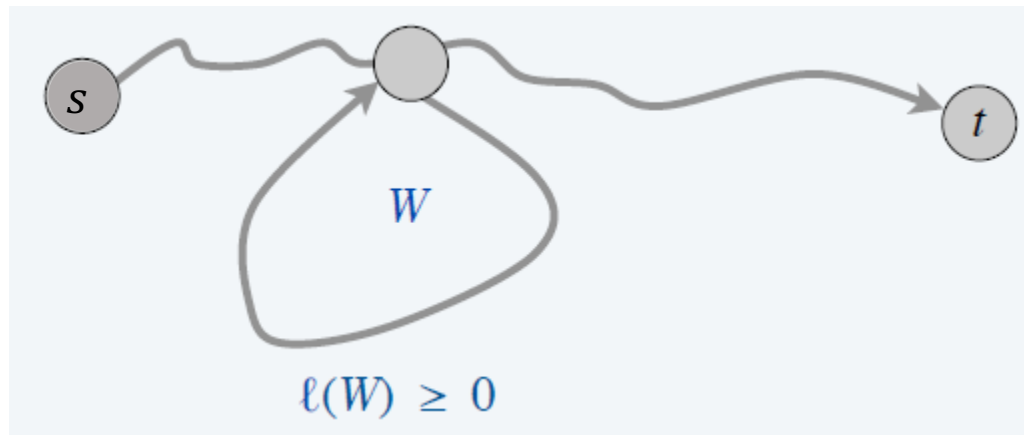
- **Input:** A directed graph $G = (V, E)$ with edge lengths ℓ_{vw} on each edge (v, w) , and a source vertex s
- **Goal:** Compute length of the shortest path from s to every vertex t

- When $\ell_{vw} \geq 0$ for each (v, w) ...

- Dijkstra's algorithm can be used for this purpose
- But it fails when some edge lengths can be negative
- What do we do in this case?

Single-Source Shortest Paths

- Cycle length = sum of lengths of edges in the cycle
- If there is a negative length cycle, shortest paths are not even well defined...
 - You can traverse the cycle arbitrarily many times to get arbitrarily “short” paths

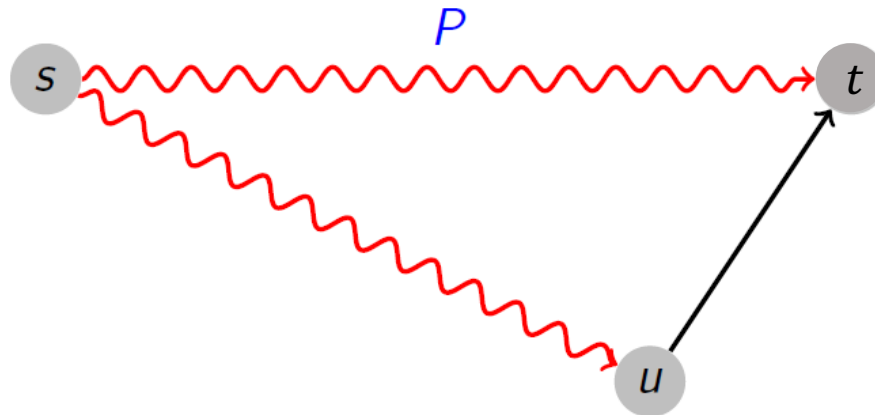


Single-Source Shortest Paths

- But if there are no negative cycles...
 - Shortest paths are well-defined even when some of the edge lengths may be negative
- **Claim:** With no negative cycles, there is always a shortest path from any vertex to any other vertex that is **simple**
 - Consider the shortest $s \rightsquigarrow t$ path with the fewest edges among all shortest $s \rightsquigarrow t$ paths
 - If it has a cycle, removing the cycle creates a path with fewer edges that is no longer than the original path

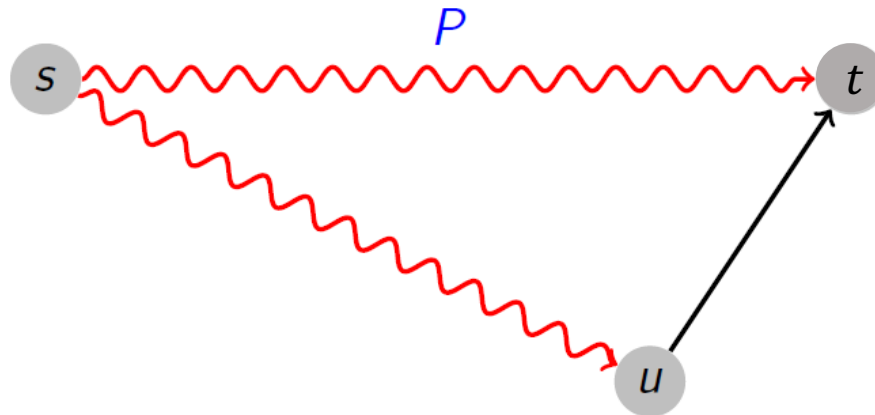
Optimal Substructure Property

- Consider a simple shortest $s \rightsquigarrow t$ path P
 - It could be just a single edge
 - But if P has more than one edges, consider u which immediately precedes t in the path
 - If $s \rightsquigarrow t$ is shortest, $s \rightsquigarrow u$ must be shortest as well and it must use one fewer edge than the $s \rightsquigarrow t$ path



Optimal Substructure Property

- $OPT(t, i)$ = length of the shortest path from s to t using at most i edges
- **Then:**
 - Either this path uses at most $i - 1$ edges $\Rightarrow OPT(t, i - 1)$
 - Or it uses i edges $\Rightarrow \min_u OPT(u, i - 1) + \ell_{ut}$



Optimal Substructure Property

- $OPT(t, i)$ = shortest path from s to t using at most i edges

- **Then:**

- Either this path uses at most $i - 1$ edges $\Rightarrow OPT(t, i - 1)$

- Or it uses exactly i edges $\Rightarrow \min_u OPT(u, i - 1) + \ell_{ut}$

$$OPT(t, i) = \begin{cases} 0 & i = 0 \vee t = s \\ \infty & i = 0 \wedge t \neq s \\ \min \left\{ OPT(t, i - 1), \min_u OPT(u, i - 1) + \ell_{ut} \right\} & \text{otherwise} \end{cases}$$

- **Running time:** $O(n^2)$ calls, each takes $O(n)$ time $\Rightarrow O(n^3)$

- **Q:** What do you need to store to also get the actual paths?

Side Notes

- **Bellman-Ford-Moore algorithm**
 - Improvement over this DP
 - Running time $O(mn)$ for n vertices and m edges
 - Space complexity reduces to $O(m + n)$

| year | worst case | discovered by |
|------|------------------------------|-----------------------------|
| 1955 | $O(n^4)$ | Shimbel |
| 1956 | $O(m n^2 W)$ | Ford |
| 1958 | $O(m n)$ | Bellman, Moore |
| 1983 | $O(n^{3/4} m \log W)$ | Gabow |
| 1989 | $O(m n^{1/2} \log(nW))$ | Gabow-Tarjan |
| 1993 | $O(m n^{1/2} \log W)$ | Goldberg |
| 2005 | $O(n^{2.38} W)$ | Sankowski, Yuster-Zwick |
| 2016 | $\tilde{O}(n^{10/7} \log W)$ | Cohen-Mądry-Sankowski-Vladu |
| 20xx | ??? | |

single-source shortest paths with weights between $-W$ and W

Maximum Length Paths?

- Can we use a similar DP to compute maximum length paths from s to all other vertices?
- This is well defined when there are no positive cycles, in which case, yes.
- What if there are positive cycles, but we want maximum length *simple* paths?

Maximum Length Paths?

- **What goes wrong?**
 - Our DP doesn't work because its path from s to t might use a path from s to u and edge from u to t
 - But path from s to u might in turn go through t
 - The path may no longer remain simple
- In fact, maximum length simple path is **NP-hard**
 - Hamiltonian path problem (i.e., “is there a path of length $n - 1$ in a given undirected graph?”) is a special case

All-Pairs Shortest Paths

- **Problem**

- **Input:** A directed graph $G = (V, E)$ with edge lengths ℓ_{vw} on each edge (v, w) and no negative cycles
- **Goal:** Compute the length of the shortest path from all vertices s to all other vertices t

- Simple idea:

- Run single-source shortest paths from each source s
- Running time is $O(n^4)$
- Actually, we can do this in $O(n^3)$ as well

All-Pairs Shortest Paths

- **Problem**

- **Input:** A directed graph $G = (V, E)$ with edge lengths ℓ_{vw} on each edge (v, w) and no negative cycles
- **Goal:** Compute the length of the shortest path from all vertices s to all other vertices t

- $OPT(u, v, k)$ = length of shortest simple path from u to v in which intermediate nodes are from $\{1, \dots, k\}$
- **Exercise:** Write down the Bellman equation for OPT such that given all subsolutions, it requires $O(1)$ time to compute
- **Running time:** $O(n^3)$ calls, $O(1)$ per call $\Rightarrow O(n^3)$

Chain Matrix Product

- **Problem**

- **Input:** Matrices M_1, \dots, M_n where the dimension of M_i is $d_{i-1} \times d_i$
- **Goal:** Compute $M_1 \cdot M_2 \cdot \dots \cdot M_n$

- But matrix multiplication is **associative**

- $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
- So, isn't the optimal solution going to call the algorithm for multiplying two matrices exactly $n - 1$ times?
- **Insight:** the time it takes to multiply two matrices depends on their dimensions

Chain Matrix Product

- Assume
 - We use the brute force approach for matrix multiplication
 - So, multiplying $p \times q$ and $q \times r$ matrices requires $p \cdot q \cdot r$ operations
- Example: compute $M_1 \cdot M_2 \cdot M_3$
 - M_1 is 5 X 10
 - M_2 is 10 X 100
 - M_3 is 100 X 50
 - $(M_1 \cdot M_2) \cdot M_3 \rightarrow 5 \cdot 10 \cdot 100 + 5 \cdot 100 \cdot 50 = 30000$ ops
 - $M_1 \cdot (M_2 \cdot M_3) \rightarrow 10 \cdot 100 \cdot 50 + 5 \cdot 10 \cdot 50 = 52500$ ops

Chain Matrix Product

- **Note**

- Our input is simply the dimensions d_0, d_1, \dots, d_n (such that each M_i is $d_{i-1} \times d_i$) and not the actual matrices

- **Why is DP right for this problem?**

- Optimal substructure property
- Think of the final product computed, say $A \cdot B$
- A is the product of some prefix, B is the product of the remaining suffix
- For the overall optimal computation, each of A and B should be computed optimally

Chain Matrix Product

- $OPT(i, j) = \min$ ops required to compute $M_i \cdot \dots \cdot M_j$
 - Here, $1 \leq i \leq j \leq n$
 - **Q:** Why do we not just care about prefixes and suffices?
 - $(M_1 \cdot (M_2 \cdot M_3 \cdot M_4)) \cdot M_5 \Rightarrow$ need to know optimal solution for $M_2 \cdot M_3 \cdot M_4$

$$OPT(i, j) = \begin{cases} 0 & i = j \\ \min_{k:i \leq k < j} OPT(i, k) + OPT(k + 1, j) + d_{i-1}d_kd_j & \text{if } i < j \end{cases}$$

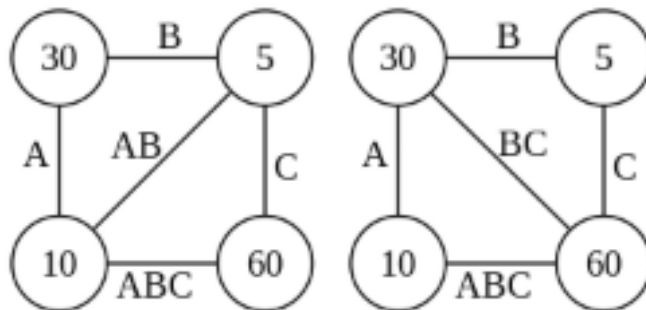
- **Running time:** $O(n^2)$ calls, $O(n)$ time per call $\Rightarrow O(n^3)$

Chain Matrix Product

NOT IN SYLLABUS

- Can we do better?

- Surprisingly, yes. But not by a DP algorithm (that I know of)
- Hu & Shing (1981) developed $O(n \log n)$ time algorithm by reducing chain matrix product to the problem of “optimally” triangulating a regular polygon



Source: Wikipedia

Example

- A is 10×30 , B is 30×5 , C is 5×60
- The cost of each triangle is the product of its vertices
- Want to minimize total cost of all triangles

Edit Distance

- Edit distance (aka sequence alignment) problem
 - How similar are strings $X = x_1, \dots, x_m$ and $Y = y_1, \dots, y_n$?
- Suppose we can delete or replace symbols
 - We can do these operations on any symbol in either string
 - How many deletions & replacements does it take to match the two strings?

Edit Distance

- **Example:** occurrence vs occurrence

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| o | c | u | r | r | a | n | c | e | - |
| | | | | | | | | | |
| o | c | c | u | r | r | e | n | c | e |

6 replacements, 1 deletion

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| o | c | - | u | r | r | a | n | c | e |
| | | | | | | | | | |
| o | c | c | u | r | r | e | n | c | e |

1 replacement, 1 deletion

Edit Distance

- Edit distance problem

- Input

- Strings $X = x_1, \dots, x_m$ and $Y = y_1, \dots, y_n$
- Cost $d(a)$ of deleting symbol a
- Cost $r(a, b)$ of replacing symbol a with b
 - Assume $r(a, b) = r(b, a)$ and $r(a, a) = 0$, for all a, b

- Goal

- Compute the minimum total cost for matching the two strings

- Optimal substructure?

- Want to delete/replace at one end and recurse

Edit Distance

- **Optimal substructure**

- **Goal:** match x_1, \dots, x_m and y_1, \dots, y_n

- Consider the last symbols x_m and y_n

- **Three options:**

- Delete x_m , and optimally match x_1, \dots, x_{m-1} and y_1, \dots, y_n

- Delete y_n , and optimally match x_1, \dots, x_m and y_1, \dots, y_{n-1}

- Match x_m and y_n , and optimally match x_1, \dots, x_{m-1} and y_1, \dots, y_{n-1}

- We incur cost $r(x_m, y_n)$

- Recall: $r(a, a) = 0$, so no cost if x_m and y_n already match

- Hence in the DP, we need to compute the optimal solutions for matching x_1, \dots, x_i with y_1, \dots, y_j for all (i, j)

Edit Distance

- $E[i, j]$ = edit distance between x_1, \dots, x_i and y_1, \dots, y_j
- Bellman equation

$$E[i, j] = \begin{cases} 0 & \text{if } i = j = 0 \\ B & \text{if } i = 0 \wedge j > 0 \\ A & \text{if } i > 0 \wedge j = 0 \\ \min\{A, B, C\} & \text{otherwise} \end{cases}$$

where

$$A = d(x_i) + E[i - 1, j], B = d(y_j) + E[i, j - 1]$$

$$C = r(x_i, y_j) + E[i - 1, j - 1]$$

- $O(n \cdot m)$ time, $O(n \cdot m)$ space

Edit Distance

$$E[i, j] = \begin{cases} 0 & \text{if } i = j = 0 \\ d(y_j) + E[i, j - 1] & \text{if } i = 0 \wedge j > 0 \\ d(x_i) + E[i - 1, j] & \text{if } i > 0 \wedge j = 0 \\ \min\{A, B, C\} & \text{otherwise} \end{cases}$$

where

$$A = d(x_i) + E[i - 1, j], B = d(y_j) + E[i, j - 1]$$

$$C = r(x_i, y_j) + E[i - 1, j - 1]$$

- **Space complexity can be reduced in bottom-up approach**
 - While computing $E[\cdot, j]$, we only need to store $E[\cdot, j]$ and $E[\cdot, j - 1]$,
 - So, the additional space required is $O(m)$
 - By storing two rows at a time instead, we can make it $O(n)$
 - Usually, we include the storage of inputs, so both are $O(n + m)$
 - But this is not enough if we want to compute the actual solution

Hirschberg's Algorithm

NOT IN SYLLABUS

- The optimal solution can be computed in $O(n \cdot m)$ time and $O(n + m)$ space too!

Programming
Techniques

G. Manacher
Editor

A Linear Space Algorithm for Computing Maximal Common Subsequences

D.S. Hirschberg
Princeton University

The problem of finding a longest common subsequence of two strings has been solved in quadratic time and space. An algorithm is presented which will solve this problem in quadratic time and in linear space.

Key Words and Phrases: subsequence, longest common subsequence, string correction, editing

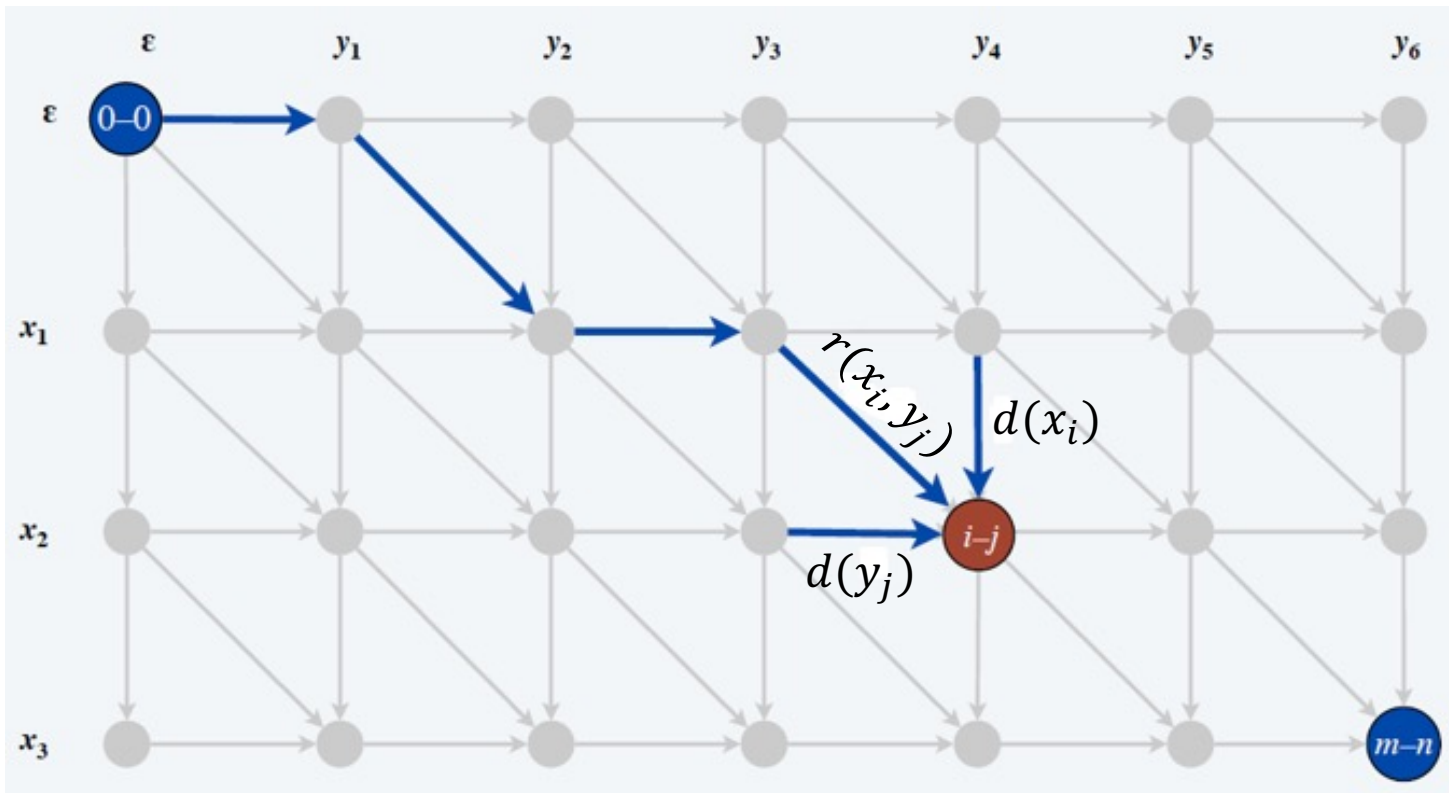
CR Categories: 3.63, 3.73, 3.79, 4.22, 5.25



Hirschberg's Algorithm

NOT IN SYLLABUS

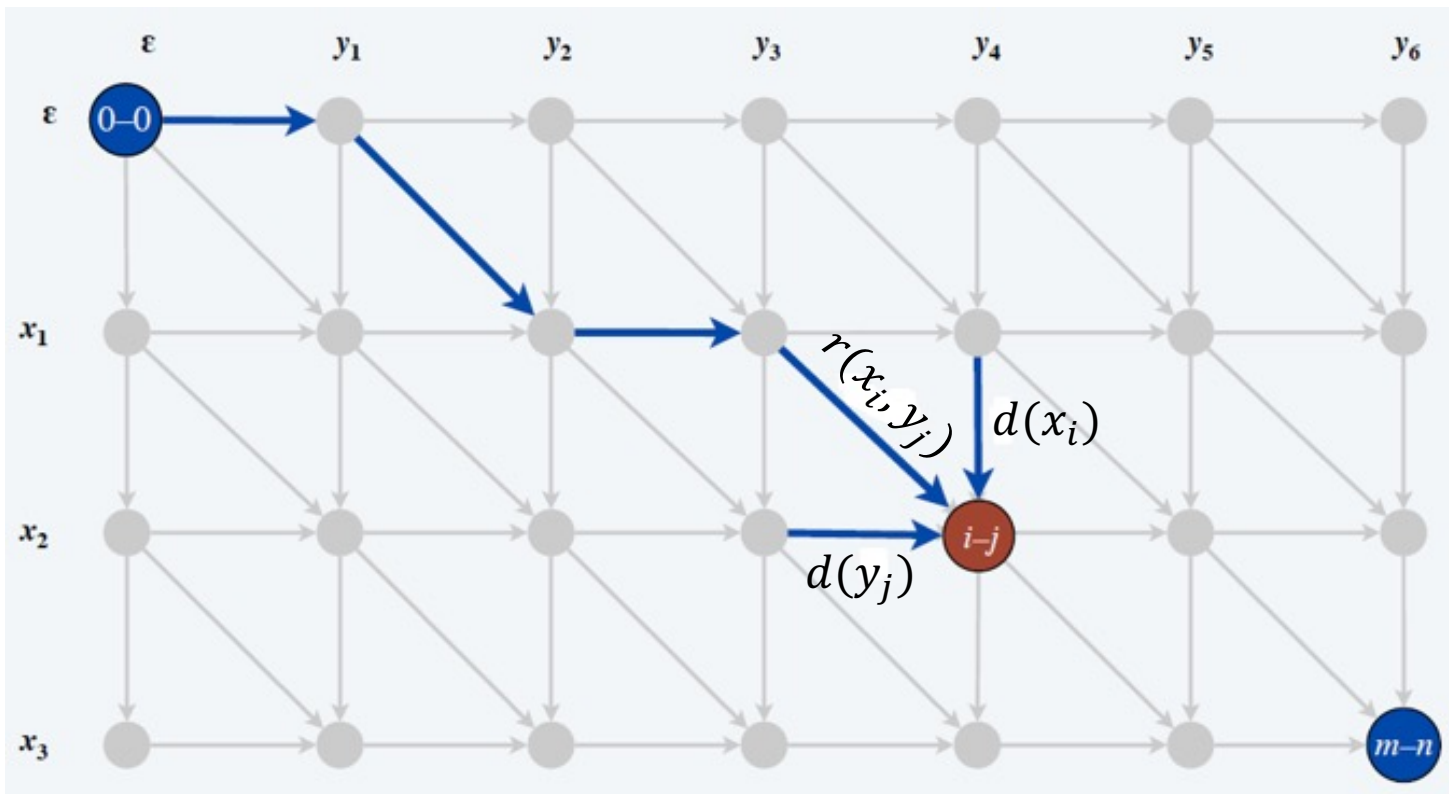
- Key idea nicely combines divide & conquer with DP
- Edit distance graph



Hirschberg's Algorithm

NOT IN SYLLABUS

- Observation (can be proved by induction)
 - $E[i, j]$ = length of shortest path from $(0, 0)$ to (i, j)

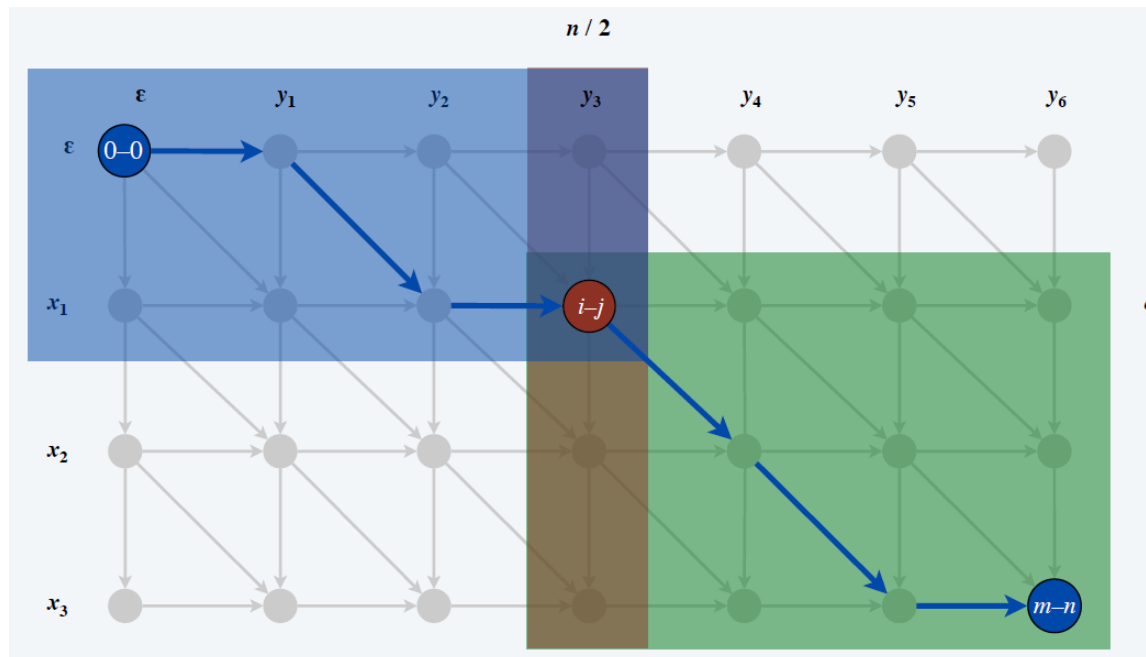


Hirschberg's Algorithm

NOT IN SYLLABUS

- Lemma

- Shortest path from $(0,0)$ to (m,n) passes through $(q, n/2)$ where q minimizes length of shortest path from $(0,0)$ to $(q, n/2)$ + length of shortest path from $(q, n/2)$ to (m,n)

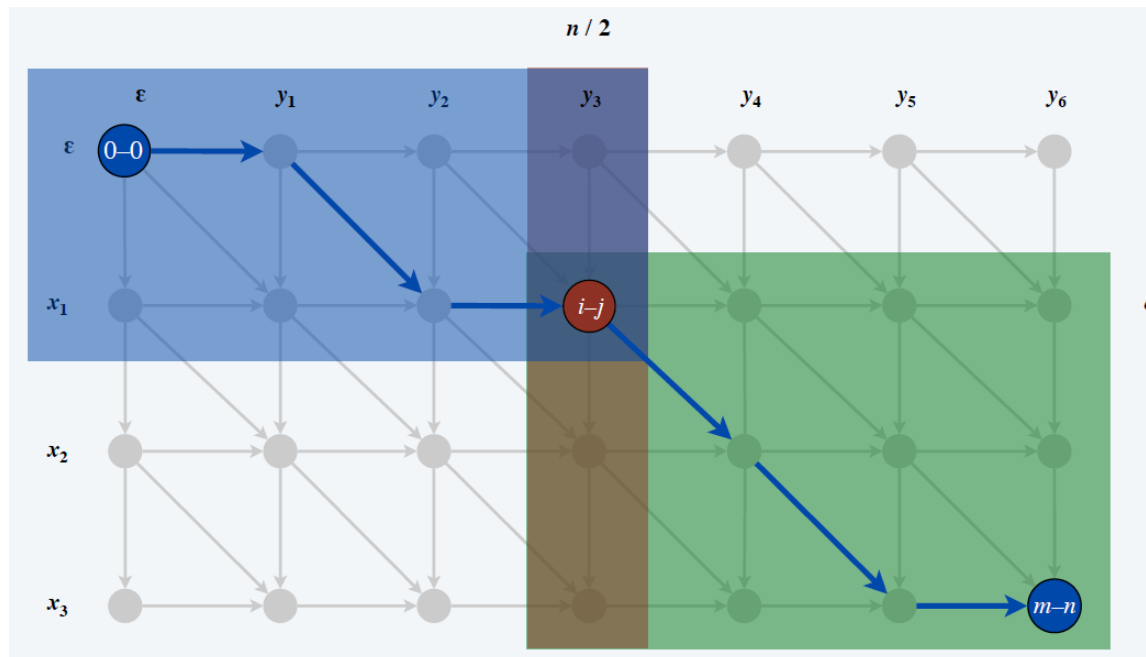


Hirschberg's Algorithm

NOT IN SYLLABUS

- Idea

- Find q using divide-and-conquer
- Find shortest paths from $(0,0)$ to $(q, n/2)$ and $(q, n/2)$ to (m, n) using DP



Application: Protein Matching

| | A | R | N | D | C | Q | E | G | H | I | L | K | M | F | P | S | T | W | Y | V |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| A | 7 | -3 | -3 | -3 | -1 | -2 | -2 | 0 | -3 | -3 | -3 | -1 | -2 | -4 | -1 | 2 | 0 | -5 | -4 | -1 |
| R | -3 | 9 | -1 | -3 | -6 | 1 | -1 | -4 | 0 | -5 | -4 | 3 | -3 | -5 | -3 | -2 | -2 | -5 | -4 | -4 |
| N | -3 | -1 | 9 | 2 | -5 | 0 | -1 | -1 | 1 | -6 | -6 | 0 | -4 | -6 | -4 | 1 | 0 | -7 | -4 | -5 |
| D | -3 | -3 | 2 | 10 | -7 | -1 | 2 | -3 | -2 | -7 | -7 | -2 | -6 | -6 | -3 | -1 | -2 | -8 | -6 | -6 |
| C | -1 | -6 | -5 | -7 | 13 | -5 | -7 | -6 | -7 | -2 | -3 | -6 | -3 | -4 | -6 | -2 | -2 | -5 | -5 | -2 |
| Q | -2 | 1 | 0 | -1 | -5 | 9 | 3 | -4 | 1 | -5 | -4 | 2 | -1 | -5 | -3 | -1 | -1 | -4 | -3 | -4 |
| E | -2 | -1 | -1 | 2 | -7 | 3 | 8 | -4 | 0 | -6 | -6 | 1 | -4 | -6 | -2 | -1 | -2 | -6 | -5 | -4 |
| G | 0 | -4 | -1 | -3 | -6 | -4 | -4 | 9 | -4 | -7 | -7 | -3 | -5 | -6 | -5 | -1 | -3 | -6 | -6 | -6 |
| H | -3 | 0 | 1 | -2 | -7 | 1 | 0 | -4 | 12 | -6 | -5 | -1 | -4 | -2 | -4 | -2 | -3 | -4 | 3 | -5 |
| I | -3 | -5 | -6 | -7 | -2 | -5 | -6 | -7 | -6 | 7 | 2 | -5 | 2 | -1 | -5 | -4 | -2 | -5 | -3 | 4 |
| L | -3 | -4 | -6 | -7 | -3 | -4 | -6 | -7 | -5 | 2 | 6 | -4 | 3 | 0 | -5 | -4 | -3 | -4 | -2 | 1 |
| K | -1 | 3 | 0 | -2 | -6 | 2 | 1 | -3 | -1 | -5 | -4 | 8 | -3 | -5 | -2 | -1 | -1 | -6 | -4 | -4 |
| M | -2 | -3 | -4 | -6 | -3 | -1 | -4 | -5 | -4 | 2 | 3 | -3 | 9 | 0 | -4 | -3 | -1 | -3 | -3 | 1 |
| F | -4 | -5 | -6 | -6 | -4 | -5 | -6 | -6 | -2 | -1 | 0 | -5 | 0 | 10 | -6 | -4 | -4 | 0 | 4 | -2 |
| P | -1 | -3 | -4 | -3 | -6 | -3 | -2 | -5 | -4 | -5 | -5 | -2 | -4 | -6 | 12 | -2 | -3 | -7 | -6 | -4 |
| S | 2 | -2 | 1 | -1 | -2 | -1 | -1 | -1 | -2 | -4 | -4 | -1 | -3 | -4 | -2 | 7 | 2 | -6 | -3 | -3 |
| T | 0 | -2 | 0 | -2 | -2 | -1 | -2 | -3 | -3 | -2 | -3 | -1 | -1 | -4 | -3 | 2 | 8 | -5 | -3 | 0 |
| W | -5 | -5 | -7 | -8 | -5 | -4 | -6 | -6 | -4 | -5 | -4 | -6 | -3 | 0 | -7 | -6 | -5 | 16 | 3 | -5 |
| Y | -4 | -4 | -4 | -6 | -5 | -3 | -5 | -6 | 3 | -3 | -2 | -4 | -3 | 4 | -6 | -3 | -3 | 3 | 11 | -3 |
| V | -1 | -4 | -5 | -6 | -2 | -4 | -4 | -6 | -5 | 4 | 1 | -4 | 1 | -2 | -4 | -3 | 0 | -5 | -3 | 7 |

Traveling Salesman

- **Input**

- Complete directed graph $G = (V, E)$
- $d_{i,j}$ = distance from node i to node j

- **Output**

- Minimum distance which needs to be traveled to start from some node v , visit every other node exactly once, and come back to v
 - That is, the minimum cost of a Hamiltonian cycle

Traveling Salesman

- Approach

- Let's start at node $v_1 = 1$
 - It's a cycle, so the starting point does not matter
- Want to visit the other nodes in some order, say v_2, \dots, v_n
- Total distance is $d_{1,v_2} + d_{v_2,v_3} + \dots + d_{v_{n-1},v_n} + d_{v_n,1}$
 - Want to minimize this distance

- Naïve solution

- Check all possible orderings
- $(n - 1)! = \Theta\left(\sqrt{n} \cdot \left(\frac{n}{e}\right)^n\right)$ (Stirling's approximation)

Traveling Salesman

- DP Approach

- Consider v_n (the last node before returning to $v_1 = 1$)
 - If $v_n = c$
 - Find the optimal order of visiting nodes $\{2, \dots, n\}$ that ends at c
 - Need to keep track of the subset of nodes to be visited and the end node
- $OPT[S, c]$ = minimum total travel distance when starting at 1, visiting each node in S exactly once, and ending at $c \in S$
- Answer to the original problem:
 - $\min_{c \in S} OPT[S, c] + d_{c,1}$, where $S = \{2, \dots, n\}$

Traveling Salesman

- DP Approach

- To compute $OPT[S, c]$, we can condition over the vertex visited right before c in the optimal trip

- Bellman equation

$$OPT[S, c] = \min_{m \in S \setminus \{c\}} (OPT[S \setminus \{c\}, m] + d_{m,c})$$

$$\text{Final solution} = \min_{c \in \{2, \dots, n\}} (OPT[\{2, \dots, n\}, c] + d_{c,1})$$

- Time: $O(n \cdot 2^n)$ calls, $O(n)$ time per call $\Rightarrow O(n^2 \cdot 2^n)$
 - Much better than the naïve solution which has $(n/e)^n$

Traveling Salesman

- Bellman equation

$$OPT[S, c] = \min_{m \in S \setminus \{c\}} (OPT[S \setminus \{c\}, m] + d_{m,c})$$

$$\text{Final solution} = \min_{c \in \{2, \dots, n\}} OPT[\{2, \dots, n\}, c] + d_{c,1}$$

- Space complexity: $O(n \cdot 2^n)$
 - But computing the optimal solution with $|S| = k$ only requires storing the optimal solutions with $|S| = k - 1$
- Question:
 - Using this observation, how much can we reduce the space complexity?

DP Concluding Remarks

- High-level steps in designing a DP algorithm
 - Focus on a single decision in optimal solution
 - Typically, the first/last decision
 - For each possible way of making that decision...
 - [Optimal substructure] Write the optimal solution of the problem in terms of the optimal solutions to subproblems
 - Generalize the problem...
 - ...by looking at the type of subproblems needed
 - E.g., in the edit distance problem, we realize that we need to solve the problem for prefixes (x_1, \dots, x_i) and (y_1, \dots, y_j) for all (i, j)
 - Write the Bellman equation, cover your base cases
 - Think about optimizing the running time/space using tricks
 - Often easier in the bottom-up implementation