CSC373

Week 4: Dynamic Programming

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?

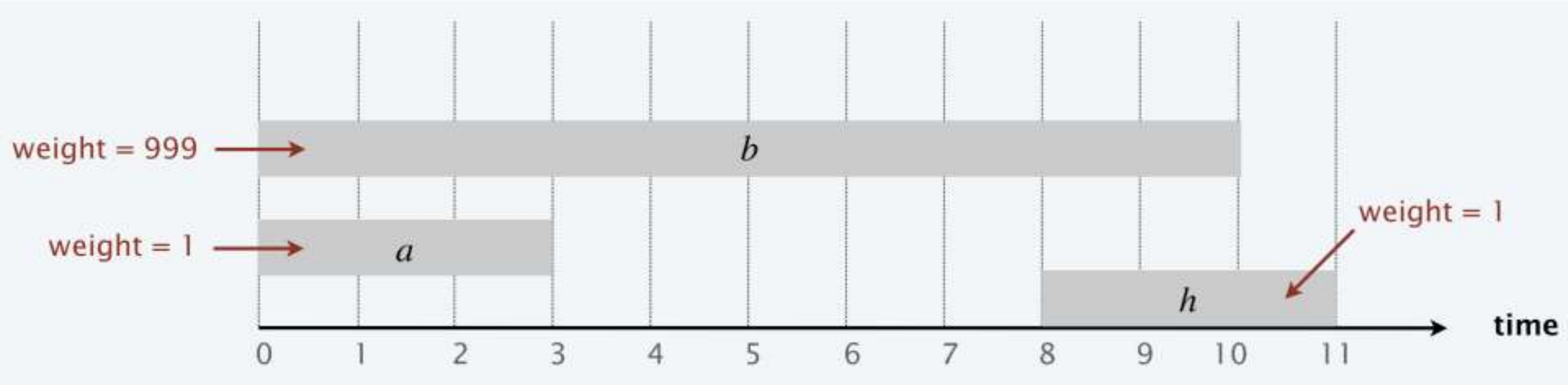
Weighted Interval Scheduling

- 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!

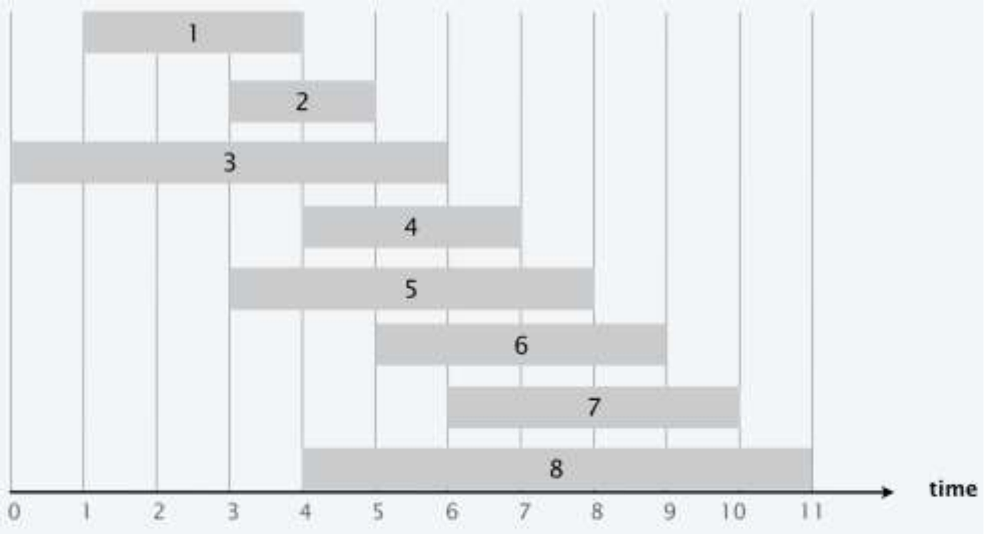


Weighted Interval Scheduling

- 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: $f_1 \le f_2 \le \cdots \le f_n$
  - $p[j]$ = largest $i < j$ such that job i is compatible with job j (i.e., $f_i \le s_j$)
    - Jobs $1, \ldots, i$ are compatible with j, but jobs $i+1, \ldots, j-1$ 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
  - Two options regarding job n:
    - Option 1: Job n is in OPT
      - Can't use incompatible jobs $\{p[n]+1, \ldots, n-1\}$
      - Must select optimal subset of jobs from $\{1, \ldots, p[n]\}$
    - Option 2: Job n is not in OPT
      - Must select optimal subset of jobs from $\{1, \ldots, n-1\}$
  - OPT is best of both options
  - Notice that in both options, we need to solve the problem on a prefix of our ordering

Weighted Interval Scheduling

- The DP approach
  - $OPT(j)$ = max total weight of compatible jobs from $\{1, \ldots, j\}$
  - 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 $(n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n)$

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

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

RETURN COMPUTE-OPT(n).

COMPUTE-OPT(j)

IF $(j = 0)$

RETURN 0.

ELSE

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

Brute Force Solution

$\text{COMPUTE-OPT}(j)$

$\text{IF } (j = 0)$

  $\text{RETURN } 0.$

ELSE

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

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

Brute Force Solution

- Brute force running time
  - 2^n
    - It is possible that $p(j) = j - 1$ for each j
    - Each call to Compute-OPT(j) calls Compute-OPT($j - 1$) twice
  - Slight optimization:
    - If $p[j] = j - 1$, just use $w_j + \text{Compute–Opt}(j - 1)$, otherwise make two calls and take the best
    - New worst case: $p(j) = j - 2$ for each j
      - Running time: $T(n) = T(n-1) + T(n-2)$
      - Fibonacci, golden ratio, … ☺
      - $T(n) = O(\varphi^n)$, where $\varphi \approx 1.618$

Dynamic Programming

- Why is the runtime high?
  - Some solutions are being computed many, many times
    - E.g., if $p[5] = 3$, then Compute-OPT(5) calls Compute-OPT(4) and Compute-OPT(3)
    - But Compute-OPT(4) in turn calls 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($n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n$)

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

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

$M[0] \leftarrow 0.$ ← global array

RETURN M-COMPUTE-OPT(n).

M-COMPUTE-OPT(j)

IF ($M[j]$ is uninitialized)

$M[j] \leftarrow \max \{$ M-COMPUTE-OPT $(j-1)$, w_j + 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
    - Each call takes $O(1)$ time, not considering the time spent in the 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$(n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n)$

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

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

$M[0] \leftarrow 0.$

FOR $j = 1$ TO n

previously computed values

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

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

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.

Optimal Solution

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

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

- Save space by storing only one bit of information for each j: which option yielded the maximum weight
- To reconstruct the optimal solution, start with $j = n$
  - If $S(j) = L$, update $j \leftarrow j - 1$
  - If $S(j) = R$, add j to the solution and update $j \leftarrow p[j]$
  - If $S(j) = \perp$, 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

Creative Commons Attribution-Share Alike 2.5
by Dake

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 item 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 $1, \ldots, i$ 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 $OPT(i, w)$
  - $i \in \{1, \ldots, n\}$
  - $w \in \{1, \ldots, W\}$ (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: $W, v_1, \ldots, v_n, w_1, \ldots, w_n$
  - Time should be polynomial in $\log W + \sum_{i=1}^{n} (\log v_i + \log w_i)$

What if…?

- If we were told that $W = poly(n)$ …
  - 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 $1, \ldots, i$
  - 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 $OPT(i-1, v)$

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

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 $\max\{v : OPT(n, v) \le W\}$
  - This approach has running time $O(n \cdot V)$, where $V = v_1 + \cdots + v_n$
  - 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).

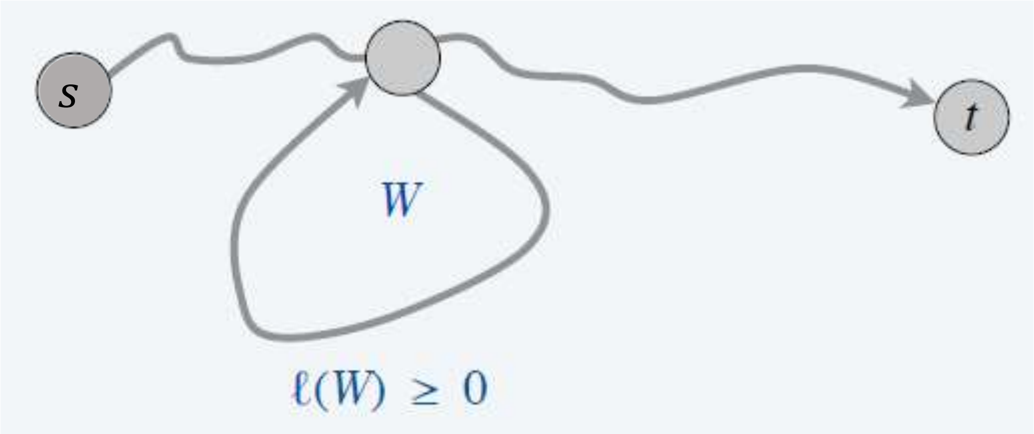
Single-Source Shortest Paths

- 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



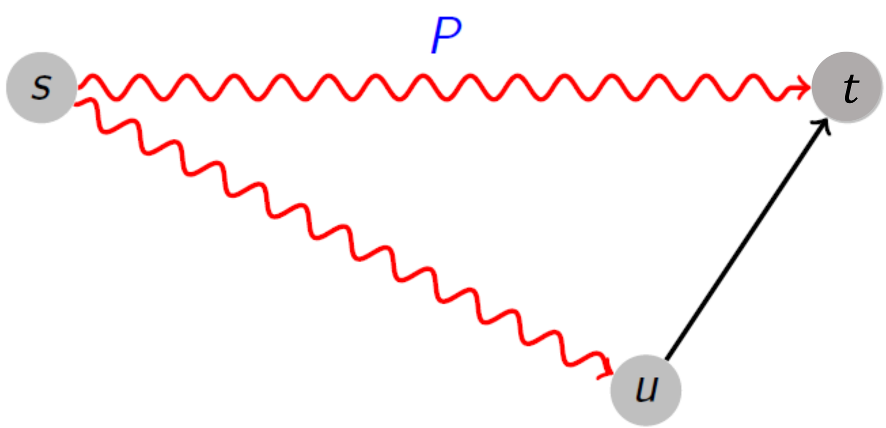
Single-Source Shortest Paths

- 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

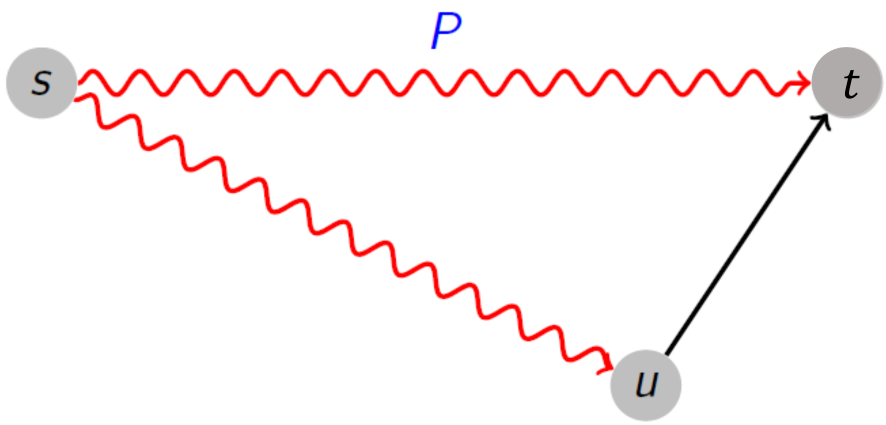
Optimal Substructure Property

- 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	$O(n^{2.38}\, W)$	Sankowsi, 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 $M_1, \ldots, M_n$ where the dimension of M_i is $d_{i-1} \times d_i$
  - Goal: Compute $M_1 \cdot M_2 \cdot \ldots \cdot M_n$

- 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 $d_0, d_1, \ldots, d_n$ (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 $M_i \cdot \ldots \cdot M_j$
  - Here, $1 \leq i \leq j \leq n$
  - Q: Why do we not just care about prefixes and suffices?
    - $\big(M_1 \cdot (M_2 \cdot M_3 \cdot M_4)\big) \cdot M_5 \Rightarrow$ need to know optimal solution for $M_2 \cdot M_3 \cdot M_4$

$$OPT(i,j) = \begin{cases} 0 & i = j \\ \min\limits_{\text{k:i≤k<j}} OPT(i,k) + OPT(k+1,j) + d_{i-1}d_k d_j & \text{if } i < j \end{cases}$$

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