6. DYNAMIC PROGRAMMING

Raveen de Silva, r.desilva@unsw.edu.au
office: K17 202
Course Admin: Anahita Namvar, cs3121@cse.unsw.edu.au

School of Computer Science and Engineering
UNSW Sydney

Term 2, 2022

Table of Contents

What is dynamic programming?

The main idea is to solve a large problem recursively by building from (carefully chosen) subproblems of smaller size.

Optimal substructure property

We must choose subproblems in such a way that

optimal solutions to subproblems can be combined into an optimal solution for the full problem.

Why is dynamic programming useful?

- Recently we discussed greedy algorithms, where the problem is viewed as a sequence of stages and we consider only the locally optimal choice at each stage.

- We saw that some greedy algorithms are incorrect, i.e. they fail to construct a globally optimal solution.

- Also, greedy algorithms are unhelpful for certain types of problems, such as enumeration ("count the number of ways to ...").
- Dynamic programming can be used to efficiently consider all the options at each stage.

Why is dynamic programming efficient?

- We have already seen one problem-solving paradigm that used recursion: divide-and-conquer.

- D&C aims to break a large problem into *disjoint* subproblems, solve those subproblems recursively and recombine.

- However, DP is characterised by *overlapping subproblems*.

Why is dynamic programming efficient?

Overlapping subproblems property

We must choose subproblems in such a way that

the same subproblem occurs several times in the recursion tree.

When we solve a subproblem, we *store the result* so that subsequent instances of the same subproblem can be answered by just looking up a value in a table.

The parts of a dynamic programming algorithm

- A dynamic programming algorithm consists of three parts:

  - a definition of the **subproblems**;
  - a **recurrence relation**, which determines how the solutions to smaller subproblems are combined to solve a larger subproblem, and

  - any **base cases**, which are the trivial subproblems - those for which the recurrence is not required.

Putting it all together

- The original problem may be one of our subproblems, or it may be solved by combining results from several subproblems, in which case we must also describe this process.

- Finally, we should be aware of the time complexity of our algorithm, which is usually given by multiplying the *number of subproblems* by the *'average' time taken to solve a subproblem using the recurrence*.

Table of Contents

Longest Increasing Subsequence

Problem

Instance: a sequence of n real numbers $A[1..n]$.

Task: determine a subsequence (not necessarily contiguous) of maximum length, in which the values in the subsequence are strictly increasing.

Longest Increasing Subsequence

- A natural choice for the subproblems is as follows: for each $1 \leq i \leq n$, let $P(i)$ be the problem of determining the length of the longest increasing subsequence of $A[1..i]$.

- However, it is not immediately obvious how to relate these subproblems to each other.

- A more convenient specification involves $Q(i)$, the problem of determining $\mathrm{opt}(i)$, the length of the longest increasing subsequence of $A[1..i]$ *ending at* the last element $A[i]$.
- Note that the overall solution is recovered by $\max\{\mathrm{opt}(i) \mid 1 \leq i \leq n\}$.

Longest Increasing Subsequence

- We will try to solve $Q(i)$ by extending the sequence which solves $Q(j)$ for some $j < i$.
- One example of a greedy algorithm using this framework is as follows.

Attempt 1

Solve $Q(i)$ by extending the solution of $Q(j)$ for j as close to i as possible, i.e. the largest j such that $A[j] < A[i]$.

Exercise

Design an instance of the problem for which this algorithm is not correct.

Longest Increasing Subsequence

- Instead, assume we have solved all the subproblems for $j < i$.

- We now look for all indices $j < i$ such that $A[j] < A[i]$.

- Among those we pick m so that $\text{opt}(m)$ is maximal, and extend that sequence with $A[i]$.

- This forms the basis of our recurrence!

- The recurrence is not necessary if $i = 1$, as there are no previous indices to consider, so this is our base case.

Longest Increasing Subsequence

Solution

Subproblems: for each $1 \leq i \leq n$, let $Q(i)$ be the problem of determining $\text{opt}(i)$, the maximum length of an increasing subsequence of $A[1..i]$ which ends with $A[i]$.

Recurrence: for $i > 1$,

$$\text{opt}(i) = \max\{\text{opt}(j) \mid j < i, A[j] < A[i]\} + 1.$$

Base case: $\text{opt}(1) = 1$.

Longest Increasing Subsequence

i	1	2	3	4	5	6	7	8
$A[i]$	1	5	3	6	2	7	4	8
$\text{opt}(i)$	1	2	2	3	2	4	3	5

$$\text{opt}(i) = \max\{\text{opt}(j) \mid j < i, A[j] < A[i]\} + 1$$

Longest Increasing Subsequence

- Upon computing a value $\text{opt}(i)$, we store it in the i^{th} slot of a table, so that we can look it up in the future.

- The overall longest increasing subsequence is the best of those ending at some element, i.e. $\max\{\text{opt}(i) \mid 1 \le i \le n\}$.

- Each of n subproblems is solved in $O(n)$, and the overall solution is found in $O(n)$. Therefore the time complexity is $O(n^2)$.

Exercise

Design an algorithm which solves this problem and runs in time $O(n \log n)$.

Longest Increasing Subsequence

- Why does this produce optimal solutions to subproblems? We can use a kind of "cut and paste" argument.

- We claim that truncating the optimal solution for $Q(i)$ must produce an optimal solution of the subproblem $Q(m)$.

- Otherwise, if a better solution for $Q(m)$ existed, we could extend that instead to find a better solution for $Q(i)$ as well.

Longest Increasing Subsequence

- What if the problem asked for not only the length, but the entire longest increasing subsequence?
- This is a common extension to such problems, and is easily handled.
- In the i^{th} slot of the table, alongside $\text{opt}(i)$ we also store the index m such that the optimal solution for $Q(i)$ extends the optimal solution for $Q(m)$.
- After all subproblems have been solved, the longest increasing subsequence can be recovered by backtracking through the table.
- This contributes only a constant factor to the time and memory used by the algorithm.

Longest Increasing Subsequence

i	1	2	3	4	5	6	7	8
$A[i]$	1	5	3	6	2	7	4	8
opt(i)	1	2	2	3	2	4	3	5
pred(i)		1	1	2	1	4	3	6

$$\text{opt}(i) = \max\{\text{opt}(j) \mid j < i, A[j] < A[i]\} + 1$$

Activity Selection

Problem

Instance: A list of n activities with starting times s_i and finishing times f_i. No two activities can take place simultaneously.

Task: Find the *maximal total duration* of a subset of compatible activities.

Activity Selection

- Remember, we used the greedy method to solve a somewhat similar problem of finding a subset with the *largest possible number* of compatible activities, but the greedy method *does not* work for the present problem.

- As before, we start by sorting the activities by their finishing time into a non-decreasing sequence, and henceforth we will assume that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Activity Selection

- We can then specify the subproblems: for each $1 \leq i \leq n$, let $P(i)$ be the problem of finding the duration $t(i)$ of a subsequence $\sigma(i)$ of the first i activities which

  1. consists of non-overlapping activities,
  2. ends with activity i, and
  3. is of maximal total duration among all such sequences.

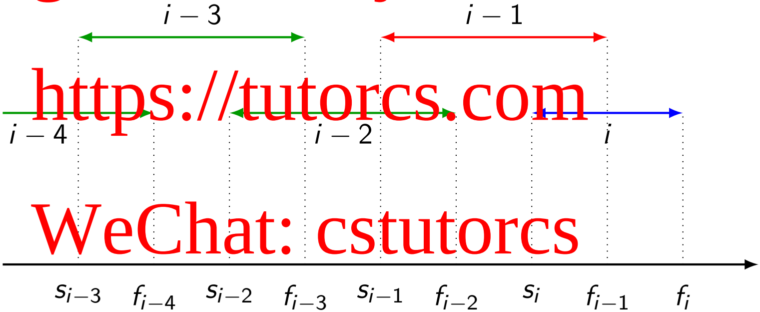
- As in the previous problem, the second condition will simplify the recurrence.

Activity Selection

- We would like to solve $P(i)$ by appending activity i to $o(j)$ for some $j < i$.

- We require that activity i not overlap with activity j, i.e. the latter finishes before the former begins.

- Among all such j, our recurrence will choose that which maximises the duration $t(j)$.
- There is no need to solve $P(1)$ in this way, as there are no preceding activities.


$i-3$
$i-1$
$i-4$
$i-2$
i
s_{i-3}
f_{i-4}
s_{i-2}
f_{i-3}
s_{i-1}
f_{i-2}
s_i
f_{i-1}
f_i

Activity Selection

Solution

Subproblems: for each $1 \leq i \leq n$, let $P(i)$ be the problem of determining $t(i)$, the maximal duration of a non-overlapping subsequence of the first i activities which ends with activity i.

Recurrence: for $i > 1$,

$$t(i) = \max\{t(j) \mid j < i,\ f_j < s_i\} + f_i - s_i.$$

Base Case: $t(1) = f_1 - s_1$.

Activity Selection

- Again, the best overall solution is given by $\max\{t(i) \mid 1 \leq i \leq n\}$

- Sorting the activities took $O(n \log n)$. Each of n subproblems is solved in $O(n)$, and the overall solution is found in $O(n)$. Therefore the time complexity is $O(n^2)$.

- Why does this recurrence produce optimal solutions to subproblems $P(i)$?

- Let the optimal solution of subproblem $P(i)$ be given by the sequence $\sigma = \langle k_1, k_2, \ldots, k_{m-1}, k_m \rangle$, where $k_m = i$.

- We claim: the truncated subsequence $\sigma' = \langle k_1, k_2, \ldots, k_{m-1} \rangle$ gives an optimal solution to subproblem $P(k_{m-1})$.
- Why? We apply the same "cut and paste" argument!

Activity Selection

- Suppose instead that $P(k_{m-1})$ is solved by a sequence τ' of larger total duration than σ'.

- Then let τ be the sequence formed by extending τ' with activity i.

- It is clear that τ has larger total duration than σ. This contradicts the earlier definition of σ as the sequence solving $P(i)$.

- Thus, the optimal sequence for problem $P(i)$ is obtained from the optimal sequence for problem $P(j)$ (for some $j < i$) by extending it with i.

- Suppose we also want to construct the optimal sequence which solves our problem.

- In the i^{th} slot of our table, we should store not only $t(i)$ but the value j such that the optimal solution of $P(i)$ extends the optimal solution of subproblem $P(j)$.

Making Change

Problem

Instance: You are given n types of coin denominations of values $v_1 < v_2 < \ldots < v_n$ (all integers). Assume $v_1 = 1$ (so that you can always make change for any integer amount) and that you have an unlimited supply of coins of each denomination.

Task: make change for a given integer amount C, using as few coins as possible.

Making Change

Attempt 1

Greedily take as many coins of value v_m as possible, then v_{m-1}, and so on.

- This approach is very tempting, and works for almost all real-world currencies.

- However, it doesn't work for all sequences v_i. In general, we will need to use DP.

Exercise

Design a counterexample to the above algorithm.

Making Change

- We will try to find the optimal solution for not only C, but every amount up to C.

- Assume we have found optimal solutions for every amount $j < i$ and now want to find an optimal solution for amount i.

- We consider each coin v_k as part of the solution for amount i, and make up the remaining amount $i - v_k$ with the previously computed optimal solution.

Making Change

- Among all of these optimal solutions, which we find in the table we are constructing recursively, we pick one which uses the fewest number of coins.

- Supposing we choose coin m, we obtain an optimal solution $\mathrm{opt}(i)$ for amount i by adding one coin of denomination v_m to $\mathrm{opt}(i - v_m)$.

- If $C = 0$ the solution is trivial: use no coins.

Making Change

Solution

Subproblems: for each $0 \leq i \leq C$, let $P(i)$ be the problem of determining $\text{opt}(i)$, the fewest coins needed to make change for an amount i.

Recurrence: for $i > 0$,

$$\text{opt}(i) = \min\{\text{opt}(i - v_k) \mid 1 \leq k \leq n, v_k \leq i\} + 1.$$

Base case: $\text{opt}(0) = 0$.

- There is no extra work required to recover the overall solution; it is just opt(C).

- Each of C subproblems is solved in $O(n)$ time, so the time complexity is $O(nC)$.

Note

Our algorithm is *NOT* a polynomial time algorithm in the *length* of the input, because C is represented by $\log C$ bits, while the running time is $O(nC)$. There is no known polynomial time algorithm for this problem!

Making Change

- Why does this produce an optimal solution for each amount $i \leq C$?

- Consider an optimal solution for some amount i, and say this solution includes at least one coin of denomination v_m for some $1 \leq m \leq n$.

- Removing this coin must leave an optimal solution for the amount $i - v_m$, again by our "cut-and-paste" argument.
- By considering all coins of value at most i, we can pick m for which the optimal solution for amount $i - v_m$ uses the fewest coins.

Making Change

- Suppose we were required to also determine the exact number of each coin required to make change for amount C.

- In the i^{th} slot of the table, we would store both $\mathrm{opt}(i)$ and the coin type $k = \mathrm{pred}(i)$ which minimises $\mathrm{opt}(i - v_k)$.
- Then $\mathrm{pred}(C)$ is a coin used in the optimal solution for total C, leaving $C' = C - \mathrm{pred}(C)$ remaining. We then repeat, identifying another coin $\mathrm{pred}(C')$ used in the optimal solution for total C', and so on.

Notation

We denote the k that minimises $\mathrm{opt}(i - v_k)$ by

$$\operatorname*{argmin}_{1 \leq k \leq n} \mathrm{opt}(i - v_k).$$

Integer Knapsack Problem (Duplicate Items Allowed)

Problem

Instance: You have n types of items; all items of kind i are identical and of weight w_i and value v_i. All weights are *integers*. You can take any number of items of each kind. You also have a knapsack of capacity C.

Task: Choose a combination of available items which all fit in the knapsack and whose value is as large as possible.

Integer Knapsack Problem (Duplicate Items Allowed)

- Similarly to the previous problem, we solve for each total weight up to C.

- Assume we have solved the problem for all total weights $j < i$.
- We now consider each type of item, the kth of which has weight w_k. If this item is included, we would fill the remaining weight with the already computed optimal solution for $i - w_k$.

- We choose the m which maximises the total value of the optimal solution for $i - w_m$ plus an item of type m, to obtain a packing of total weight i of the highest possible value.

Integer Knapsack Problem (Duplicate Items Allowed)

Solution

Subproblems: for each $0 \leq i \leq C$, let $P(i)$ be the problem of determining $\text{opt}(i)$, the maximum value that can be achieved using *up to i* units of weight, and $m(i)$, the type of some item in such a collection.

Recurrence: for $i > 0$,

$$m(i) = \operatorname*{argmax}_{1 \leq k \leq n, w_k \leq i} \left(\text{opt}(i - w_k) + v_k\right)$$

$$\text{opt}(i) = \begin{cases} 0 & \text{if } m(i) \text{ is undefined} \\ \text{opt}\left(i - w_{m(i)}\right) + v_{m(i)} & \text{otherwise.} \end{cases}$$

Base case: $\text{opt}(0) = 0$, $m(0)$ undefined.

Integer Knapsack Problem (Duplicate Items Allowed)

- The overall solution is $\mathrm{opt}(C)$, as the optimal knapsack can hold *up to* C units of weight.

- Each of C subproblems is solved in $O(n)$, for a time complexity of $O(nC)$.

- Again, our algorithm is *NOT* polynomial in the *length* of the input.

Integer Knapsack Problem (Duplicate Items NOT Allowed)

Problem

Instance: You have n items, the ith of which has weight w_i and value v_i. All weights are *integers*. You also have a knapsack of capacity C.

Task: Choose a combination of available items which all fit in the knapsack and whose value is as large as possible.

Integer Knapsack Problem (Duplicate Items NOT Allowed)

- Let's use the same subproblems as before, and try to develop a recurrence.

Question

If we know the optimal solution for each total weight $j < i$, can we deduce the optimal solution for weight i?

Answer

No! If we begin our solution for weight i with item k, we have $i - w_k$ remaining weight to fill. However, we did not record whether item k was itself used in the optimal solution for that weight.

Integer Knapsack Problem (Duplicate Items NOT Allowed)

- At this point you may object that we can in fact reconstruct the optimal solution for total weight $i - w_k$ by storing the values $m(i)$ as we did earlier, and then backtrack.

- This unfortunately has two flaws:
  1. the optimal solution for $i - w_k$ is not necessarily unique, so we may have recorded a selection including item k when a selection of equal value without item k also exists, and
  2. if all optimal solutions for $i - w_k$ use item k, it is still possible that the best solution for i combines item k with some suboptimal solution for $i - w_k$.

- The underlying issue is that with this choice of subproblems, this problem does not have the *optimal substructure property*.

Integer Knapsack Problem (Duplicate Items NOT Allowed)

- When we are unable to form a correct recurrence between our chosen subproblems, this usually indicates that the subproblem specification is inadequate.

- What extra parameters are required in our subproblem specification?

- We would like to know the optimal solution for each weight without using item k.
- Directly adding this information to the subproblem specification still doesn't lead to a useful recurrence. How could we capture it less directly?

Integer Knapsack Problem (Duplicate Items NOT Allowed)

- For each total weight i, we will find the optimal solution using only the first k items.

- We can take cases on whether item k is used in the solution:
  - if so, we have $i - w_k$ remaining weight to fill using the first $k - 1$ items, and

  - otherwise, we must fill all i units of weight with the first $k - 1$ items.

Integer Knapsack Problem (Duplicate Items NOT Allowed)

Solution

Subproblems: for $0 \leq i \leq C$ and $0 \leq k \leq n$, let $P(i, k)$ be the problem of determining $\text{opt}(i, k)$, the maximum value that can be achieved using up to i units of weight *and* using only the first k items, and $m(i, k)$, the (largest) index of an item in such a collection.

Recurrence: for $i > 0$ and $1 \leq k \leq n$,

$$\text{opt}(i, k) = \max\left(\text{opt}(i, k-1), \text{opt}(i - w_k, k-1) + v_k\right),$$

with $m(i, k) = m(i, k-1)$ in the first case and k in the second.

Base cases: if $i = 0$ or $k = 0$, then $\text{opt}(i, k) = 0$ and $m(i, k)$ is undefined.

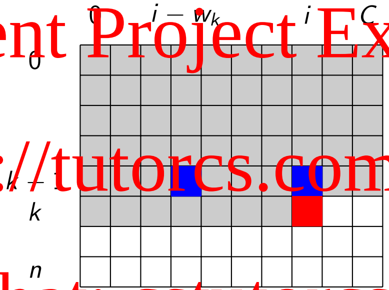
Integer Knapsack Problem (Duplicate Items NOT Allowed)

- We need to be careful about the order in which we solve the subproblems.

- When we get to $P(i, k)$, the recurrence requires us to have already solved $P(i, k-1)$ and $P(i - w_k, k-1)$.

- This is guaranteed if we subproblems $P(i, k)$ in increasing order of k, then increasing order of capacity i.

Integer Knapsack Problem (Duplicate Items NOT Allowed)



- The overall solution is opt(C, n).
- Each of $O(nC)$ subproblems is solved in constant time, for a time complexity of $O(nC)$.

Balanced Partition

Problem

Instance: a set of n positive integers x_i.

Task: partition these integers into two subsets S_1 and S_2 with sums Σ_1 and Σ_2 respectively, so as to minimise $|\Sigma_1 - \Sigma_2|$.

Balanced Partition

- Suppose without loss of generality that $\Sigma_1 \geq \Sigma_2$.
- Let $\Sigma = x_1 + \ldots + x_n$, the sum of all integers in the set.
- Observe that $\Sigma_1 + \Sigma_2 = \Sigma$, which is a constant, and upon rearranging it follows that

$$\Sigma_1 - \Sigma_2 = 2\left(\frac{\Sigma}{2} - \Sigma_2\right).$$

- So, all we have to do is find a subset S_2 of these numbers with total sum as close to $\Sigma/2$ as possible, but not exceeding it.

Balanced Partition

- For each integer x_i in the set, construct an item with both weight and value equal to x_i.

- Consider the knapsack problem (with duplicate items not allowed), with items as specified above and knapsack capacity $\Sigma/2$.

Solution

The best packing of this knapsack produces an optimally balanced partition, with set S_1 given by the items outside the knapsack and set S_2 given by the items in the knapsack.

Matrix chain multiplication

- Let A and B be matrices. The matrix product AB exists if A has as many columns as B has rows: if A is $m \times n$ and B is $n \times p$, then AB is $m \times p$.

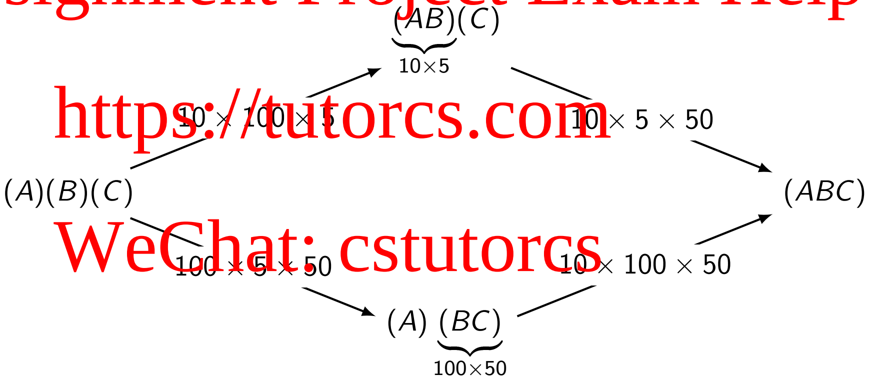
- Each element of AB is the dot product of a row of A with a column of B, both of which have length n. Therefore $m \times n \times p$ multiplications are required to compute AB.

- Matrix multiplication is *associative*, that is, for any three matrices of compatible sizes we have $A(BC) = (AB)C$.

- However, the number of real number multiplications needed to obtain the product can be very different.

Matrix chain multiplication

Suppose A is 10×100, B is 100×5 and C is 5×50.



Evaluating $(AB)C$ involves only 7500 multiplications, but evaluating $A(BC)$ requires 75000 multiplications!

Problem

Instance: a compatible sequence of matrices $A_1A_2...A_n$, where A_i is of dimension $s_{i-1} \times s_i$.

Task: group them in such a way as to minimise the total number of multiplications needed to find the product matrix.

Matrix chain multiplication

- How many groupings are there?

- The total number of different groupings satisfies the following recurrence (why?):

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i),$$

with base case $T(1) = 1$.

- One can show that the solution satisfies $T(n) = \Omega(2^n)$.

- Thus, we cannot efficiently do an exhaustive search for the optimal grouping.

Matrix chain multiplication

Note

The number of groupings $T(n)$ is a very famous sequence: the *Catalan numbers*. This sequence answers many seemingly unrelated combinatorial problems, including:

- the number of balanced bracket sequences of length $2n$;
- the number of full binary trees with $n + 1$ leaves;
- the number of lattice paths from $(0, 0)$ to (n, n) which never go above the diagonal;
- the number of noncrossing partitions of an $n + 2$-sided convex polygon;
- the number of permutations of $\{1, \ldots, n\}$ with no three-term increasing subsequence.

Matrix chain multiplication

- Instead, we try dynamic programming. A first attempt might be to specify subproblems corresponding to prefixes of the matrix chain, that is, find the optimal grouping for $A_1 A_2 \ldots A_i$.

- This is not enough to construct a recurrence; consider for example splitting the chain as

$$(A_1 A_2 \ldots A_j)(A_{j+1} A_{j+2} \ldots A_i).$$

Matrix chain multiplication

- Instead we should specify a subproblem corresponding to each contiguous subsequence $A_{i+1}A_{i+2}\ldots A_j$ of the chain.

- The recurrence will consider all possible ways to place the outermost multiplication, splitting the chain into the product

$$\underbrace{(A_{i+1}\ldots A_k)}_{s_i \times s_k}\underbrace{(A_{k+1}\ldots A_j)}_{s_k \times s_j}.$$

- No recursion is necessary for subsequences of length one.

Matrix chain multiplication

Solution

Subproblems: for all $0 \leq i < j \leq n$, let $P(i,j)$ be the problem of determining $\text{opt}(i,j)$, the fewest multiplications needed to compute the product $A_{i+1}A_{i+2}\dots A_j$.

Recurrence: for all $j - i > 1$,

$$\text{opt}(i,j) = \min\{\text{opt}(i,k) + s_i s_k s_j + \text{opt}(k,j) \mid i < k < j\}.$$

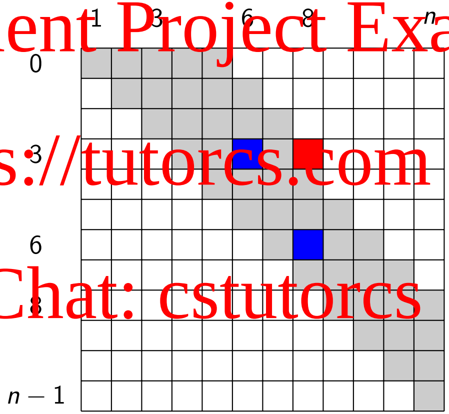
Base cases: for all $0 \leq i \leq n-1$, $\text{opt}(i,i+1) = 0$.

Matrix chain multiplication

- We have choose the order of iteration carefully. To solve a subproblem $P(i, j)$, we must have already solved $P(i, k)$ and $P(k, j)$ for each $i < k < j$.

- The simplest way to ensure this is to solve the subproblems in increasing order of $j - i$, i.e. subsequence length.

Matrix chain multiplication



$$\text{opt}(3,8) = \min\{\text{opt}(3,k) + s_3 s_k s_8 + \text{opt}(k,8) \mid 3 < k < 8\}$$

Matrix chain multiplication

- The last subproblem to be solved is $P(0, n)$, which gives the overall solution.

- To recover the actual bracketing required, we should store alongside each value $\text{opt}(i, j)$ the splitting point k used to obtain it.

Matrix chain multiplication

- Each of $O(n^2)$ subproblems is solved in $O(n)$ time, so the overall time complexity is $O(n^3)$.

Extension

Not all subproblems take the same amount of time to solve. For a subsequence of length l, only $l - 1$ potential splitting points are considered. This raises the question of whether we can prove a tighter bound for the time complexity using amortisation.

Does this algorithm actually run in $o(n^3)$? Justify your answer.

Problem

Instance: two sequences $S = \langle a_1, a_2, \ldots a_n \rangle$ and $S^* = \langle b_1, b_2, \ldots, b_m \rangle$.

Task: find the length of a longest common subsequence of S, S^*.

Longest Common Subsequence

- A sequence s is a *subsequence* of another sequence S if s can be obtained by deleting some of the symbols of S (while preserving the order of the remaining symbols).

- Given two sequences S and S^* a sequence s is a *Longest Common Subsequence* of S, S^* if s is a subsequence of both S and S^* and is of maximal possible length.

- This can be useful as a measurement of the similarity between S and S^*.
- Example: how similar are the genetic codes of two viruses? Is one of them just a genetic mutation of the other?

Longest Common Subsequence

- A natural choice of subproblems considers prefixes of both sequences, say

$$S_i = \langle a_1, a_2, \ldots, a_i \rangle \text{ and } S_j^* = \langle b_1, b_2, \ldots, b_j \rangle.$$

- If a_i and b_j are the same symbol (say c), the longest common subsequence of S_i and S_j^* is formed by appending c to the solution for S_{i-1} and S_{j-1}^*.
- Otherwise, a common subsequence of S_i and S_j^* cannot contain both a_i and b_j, so we consider discarding either of these symbols.
- No recursion is necessary when either S_i or S_j^* are empty.

Longest Common Subsequence

Solution

Subproblems: for all $0 \leq i \leq n$ and all $0 \leq j \leq m$ let $P(i,j)$ be the problem of determining $\text{opt}(i,j)$, the length of the longest common subsequence of the truncated sequences $S_i = \langle a_1, a_2, \ldots, a_i \rangle$ and $S^*_j = \langle b_1, b_2, \ldots, b_j \rangle$.

Recurrence: for all $i, j > 0$,

$$\text{opt}(i,j) = \begin{cases} \text{opt}(i-1, j-1) + 1 & \text{if } a_i = b_j \\ \max(\text{opt}(i-1,j), \text{opt}(i,j-1)) & \text{otherwise.} \end{cases}$$

Base cases: for all $0 \leq i \leq n$, $\text{opt}(i,0) = 0$, and for all $0 \leq j \leq n$, $\text{opt}(0,j) = 0$.

Longest Common Subsequence

- Iterating through the subproblems $P(i,j)$ in lexicographic order (increasing i, then increasing j) guarantees that $P(i-1,j)$, $P(i,j-1)$ and $P(i-1,j-1)$ are solved before $P(i,j)$, so all dependencies are satisfied.

- The overall solution is $\text{opt}(n,m)$.
- Each of $O(nm)$ subproblems is solved in constant time, for an overall time complexity of $O(nm)$.

- To reconstruct the longest common subsequence itself, we can record the direction from which the value $\text{opt}(i,j)$ was obtained in the table, and backtrack.

Longest Common Subsequence


```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
for i ← 1 to m
    do c[i, 0] ← 0
for j ← 0 to n
    do c[0, j] ← 0
for i ← 1 to m
    do for j ← 1 to n
           do if x_i = y_j
                 then c[i, j] ← c[i − 1, j − 1] + 1
                      b[i, j] ← "↖"
                 else if c[i − 1, j] ≥ c[i, j − 1]
                         then c[i, j] ← c[i − 1, j]
                              b[i, j] ← "↑"
                         else c[i, j] ← c[i, j − 1]
                              b[i, j] ← "←"
return c and b
```

j		0	1	2	3	4	5	6
i	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Longest Common Subsequence

- What if we have to find a longest common subsequence of three sequences S, S^*, S^{**}?

Question

Can we do $\mathrm{LCS}\left(\mathrm{LCS}\left(S, S^*\right), S^{**}\right)$?

Answer

Not necessarily!

Longest Common Subsequence

- Let $S = ABCDEGG$, $S^* = ACBEEFG$ and $S^{**} = ACCEDGF$. Then

$$\mathrm{LCS}\left(S, S^*, S^{**}\right) = ACEG.$$

- However,

$$\begin{aligned}
&\mathrm{LCS}\left(\mathrm{LCS}\left(S, S^*\right), S^{**}\right)\\
&= \mathrm{LCS}\left(\mathrm{LCS}\left(ABCDEGG, ACBEEFG\right), S^{**}\right)\\
&= \mathrm{LCS}\left(ABEG, ACCEDGF\right)\\
&= AEG.
\end{aligned}$$

Exercise

Confirm that $\mathrm{LCS}\left(\mathrm{LCS}\left(S^*, S^{**}\right), S\right)$ and $\mathrm{LCS}\left(\mathrm{LCS}\left(S, S^{**}\right), S^*\right)$ also give wrong answers.

Longest Common Subsequence

Problem

Instance: three sequences $S = \langle a_1, a_2, \ldots, a_n \rangle$, $S^* = \langle b_1, b_2, \ldots, b_m \rangle$ and $S^{**} = \langle c_1, c_2, \ldots, c_l \rangle$.

Task: find the length of a longest common subsequence of S, S^* and S^{**}.

Longest Common Subsequence

Solution

Subproblems: for all $0 \le i \le n$, all $0 \le j \le m$ and all $0 \le k \le l$, let $P(i, j, k)$ be the problem of determining $\text{opt}(i, j, k)$, the length of the longest common subsequence of the truncated sequences $S_i = \langle a_1, a_2, \ldots, a_i \rangle$, $S^*_j = \langle b_1, b_2, \ldots, b_j \rangle$ and $S^{**}_k = \langle c_1, c_2, \ldots, c_k \rangle$.

Recurrence: for all $i, j, k > 0$,

$$\text{opt}(i,j,k) = \begin{cases} \text{opt}(i-1, j-1, k-1) + 1 & \text{if } a_i = b_j = c_k \\ \max \begin{pmatrix} \text{opt}(i-1,j,k), \\ \text{opt}(i,j-1,k), \\ \text{opt}(i,j,k-1) \end{pmatrix} & \text{otherwise.} \end{cases}$$

Base cases: if $i = 0$, $j = 0$ or $k = 0$, $\text{opt}(i, j, k) = 0$.

Longest Common Subsequence

- Iterating through the subproblems $P(i, j, k)$ in lexicographic order (increasing i, then increasing j, then increasing k) guarantees that $P(i-1, j, k)$, $P(i, j-1, k)$, $P(i, j, k-1)$ and $P(i-1, j-1, k-1)$ are solved before $P(i, j, k)$, so all dependencies are satisfied.

- The overall solution is $\text{opt}(n, m, l)$.

- Each of $O(nml)$ subproblems is solved in constant time, for an overall time complexity of $O(nml)$.
- To reconstruct the longest common subsequence itself, we can record the direction from which the value $\text{opt}(i, j, k)$ was obtained in the table, and backtrack.

Shortest Common Supersequence

Problem

Instance: two sequences $s = \langle a_1, a_2, \ldots, a_n \rangle$ and $s^* = \langle b_1, b_2, \ldots, b_m \rangle$.

Task: find a shortest common supersequence S of s, s^*, i.e., a shortest possible sequence S such that both s and s^* are subsequences of S.

Shortest Common Supersequence

Solution

Find a longest common subsequence $LCS(s, s^*)$ of s and s^*, then add back the differing elements of the two sequences in the right places, in any compatible order.

Example

If

$$s = abacada \text{ and } s^* = xbycazd$$

then

$$\text{LCS}(s, s^*) = bcad$$

and therefore

$$\text{SCS}(s, s^*) = axbyacazda.$$

Edit Distance

Problem

Instance: Given two text strings A of length n and B of length m, you want to transform A into B. You are allowed to insert a character, delete a character and to replace a character with another one. An insertion costs c_I, a deletion costs c_D and a replacement costs c_R.

Task: find the lowest total cost transformation of A into B.

Edit Distance

- Edit distance is another measure of the similarity of pairs of strings.

- Note: if all operations have a unit cost, then you are looking for the minimal number of such operations required to transform A into B; this number is called the *Levenshtein distance* between A and B.

- If the sequences are sequences of DNA bases and the costs reflect the probabilities of the corresponding mutations, then the minimal cost represents how closely related the two sequences are.

Edit Distance

- Again we consider prefixes of both strings, say $A[1..i]$ and $B[1..j]$.

- We have the following options to transform $A[1..i]$ into $B[1..j]$:
  1. delete $A[i]$ and then transform $A[1..i-1]$ into $B[1..j]$;
  2. transform $A[1..i]$ to $B[1..j-1]$ and then append $B[j]$;
  3. transform $A[1..i-1]$ to $B[1..j-1]$ and if necessary replace $A[i]$ by $B[j]$.

- If $i = 0$ or $j = 0$, we only insert or delete respectively.

Edit Distance

Solution

Subproblems: for all $0 \leq i \leq n$ and $0 \leq j \leq m$, let $P(i,j)$ be the problem of determining $\text{opt}(i,j)$, the minimum cost of transforming the sequence $A[1..i]$ into the sequence $B[1..j]$.

Recurrence: for $i, j \geq 1$,

$$\text{opt}(i,j) = \min \begin{cases} \text{opt}(i-1,j) + c_D \\ \text{opt}(i,j-1) + c_I \\ \begin{cases} \text{opt}(i-1,j-1) & \text{if } A[i] = B[j] \\ \text{opt}(i-1,j-1) + c_R & \text{if } A[i] \neq B[j]. \end{cases} \end{cases}$$

Base cases: $\text{opt}(i,0) = ic_D$ and $\text{opt}(0,j) = jc_I$.

Edit Distance

- The overall solution is $\mathrm{opt}(n, m)$.

- Each of $O(nm)$ subproblems is solved in constant time, for a total time complexity of $O(nm)$.

Maximising an expression

Problem

Instance: a sequence of numbers with operations $+, -, \times$ in between, for example

$$1 + 2 - 3 \times 6 - 1 - 2 \times 3 - 5 \times 7 + 2 - 8 \times 9.$$

Task: place brackets in a way that the resulting expression has the largest possible value.

Maximising an expression

- What will be the subproblems?

- Similar to the matrix chain multiplication problem earlier, it's not enough to just solve for prefixes $A[1..i]$.

- Maybe for a subsequence of numbers $A[i+1..j]$ place the brackets so that the resulting expression is maximised?

Maximising an expression

- How about the recurrence?

- It is natural to break down $A[i+1..j]$ into $A[i+1..k] \odot A[k+1..j]$, with cases $\odot = +, -, \times$.

- In the case $\odot = +$, we want to maximise the values over both $A[i+1..k]$ and $A[k+1..j]$.

- This doesn't work for the other two operations!

- We should look for placements of brackets not only for the maximal value but also for the minimal value!

Maximising an expression

Exercise

Write a complete solution for this problem. Your solution should include the subproblem specification, recurrence and base cases. You should also describe how the overall solution is to be obtained, and analyse the time complexity of the algorithm.

Turtle Tower

Problem

Instance: You are given n turtles, and for each turtle you are given its weight and its strength. The strength of a turtle is the maximal weight you can put on it without cracking its shell.

Task: find the largest possible number of turtles which you can stack one on top of the other, without cracking any turtle.

Hint

Surprisingly difficult! Order turtles in increasing order of the sum of their weight and their strength, and proceed by recursion.

Integer Partitions

Problem

Instance: a positive integer n.

Task: compute the number of partitions of n, i.e., the number of distinct *multisets* of positive integers $\{n_1, \ldots, n_k\}$ such that $n_1 + \ldots + n_k = n$.

Integer Partitions

Hint

It's not obvious how to construct a recurrence between the number of partitions of different values of n. Instead consider restricted partitions!

Let $\text{nump}(i, j)$ denote the number of partitions of j in which no part exceeds i, so that the answer is $\text{nump}(n, n)$.

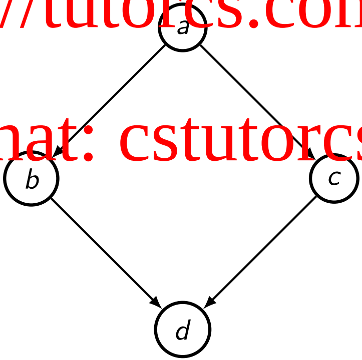
The recursion is based on relaxation of the allowed size i of the parts of j for all j up to n. It distinguishes those partitions where all parts are $\leq i - 1$ and those where at least one part is exactly i.

Table of Contents

Directed acyclic graphs

Definition

Recall that a *directed acyclic graph* (DAG) is exactly that: a directed graph without (directed) cycles.



Topological ordering

Definition

Recall that in a directed graph, a *topological ordering* of the vertices is one in which all edges point "left to right".

- A directed graph admits a topological ordering if and only if it is acyclic.

- There may be more than one valid topological ordering for a particular DAG.
- A topological ordering can be found in linear time, i.e. $O(|V| + |E|)$.

Shortest path in a directed acyclic graph

Problem

Instance: a directed acyclic graph $G = (V, E)$ in which each edge $e \in E$ has a corresponding weight $w(e)$ (which may be negative), and a designated vertex $s \in V$.

Task: find the shortest path from s to each vertex $t \in V$.

Notation

Let $n = |V|$ and $m = |E|$.

Shortest path in a directed acyclic graph

- If all edge weights are positive, the single source shortest path problem is solved by Dijkstra's algorithm in $O(m \log n)$.

- Later in this lecture, we'll see how to solve the general single source shortest path problem in $O(nm)$ using the Bellman-Ford algorithm.

- However, in the special case of directed acyclic graphs, a simple DP solves this problem in $O(n + m)$, i.e. linear time.

Shortest path in a directed acyclic graph

- The natural subproblems are the shortest path to each vertex.

- Each vertex v with an edge to t is a candidate for the penultimate vertex in an $s - t$ path.

- The recurrence considers the path to each such v, plus the weight of the last edge, and selects the minimum of these options.

- The base case is s itself, where the shortest path is obviously zero.

Shortest path in a directed acyclic graph

Solution

Subproblems: for all $t \in V$, let $P(t)$ be the problem of determining $\text{opt}(t)$, the length of a shortest path from s to t.

Recurrence: for all $t \neq s$,

$$\text{opt}(t) = \min\{\text{opt}(v) + w(v,t) \mid (v,t) \in E\}.$$

Base case: $\text{opt}(s) = 0$.

Shortest path in a directed acyclic graph

- The solution is the entire list of values opt(t).

- At first it appears that each of n subproblems is solved in $O(n)$ time, giving a time complexity of $O(n^2)$.

- However, each edge is only considered once (at its endpoint), so we can use the tighter bound $O(m)$.

Shortest path in a directed acyclic graph

Question

In what order should we solve the subproblems?

- In any DP algorithm, the recurrence introduces certain dependencies, and it is crucial that these dependencies are respected.

- Here, $\text{opt}(t)$ depends on all the $\text{opt}(v)$ values for vertices v with outgoing edges to t, so we need to solve $P(v)$ for each such v before solving $P(t)$.

- We can achieve this by solving the vertices in topological order, from left to right. All edges point from left to right, so any vertex with an outgoing edge to t is solved before t is.

DP on a directed acyclic graph

- Many problems on directed acyclic graphs can be solved in the same way: first use topological sort, then DP over the vertices in that order.

- If we replace the min in the earlier recurrence by max, we have an algorithm to find the longest path from s to each t. This problem is much harder on general graphs; indeed, there is no known algorithm to solve it in polynomial time.

- Often a graph will be specified in a way that makes it obviously acyclic, with a natural topological order.

Assembly line scheduling

Problem

Instance: You are given two assembly lines, each consisting of n workstations. The k^{th} workstation on each assembly line performs the k^{th} of n jobs.

- To bring a new product to the start of assembly line i takes s_i units of time.

- To retrieve a finished product from the end of assembly line i takes f_i units of time.

Assembly line scheduling

Problem (continued)

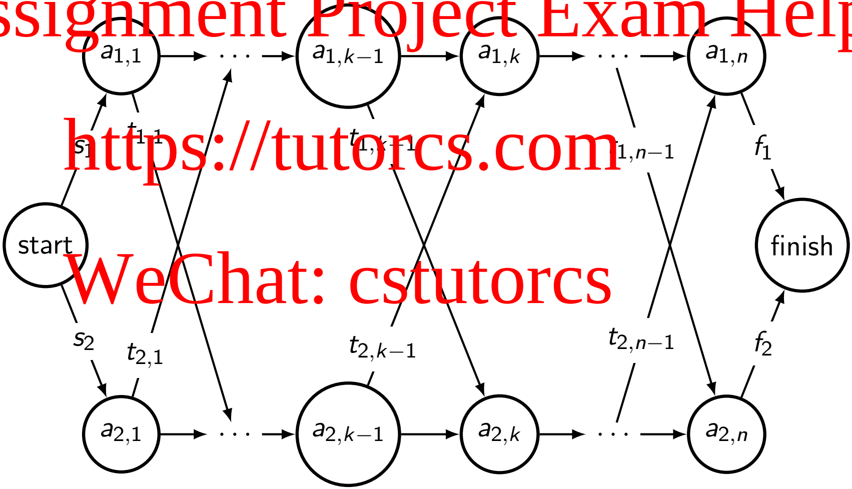
- On assembly line i, the k^{th} job takes $a_{i,k}$ units of time to complete.

- To move the product from station k on assembly line i to station $k+1$ on the other line takes $t_{i,k}$ units of time.

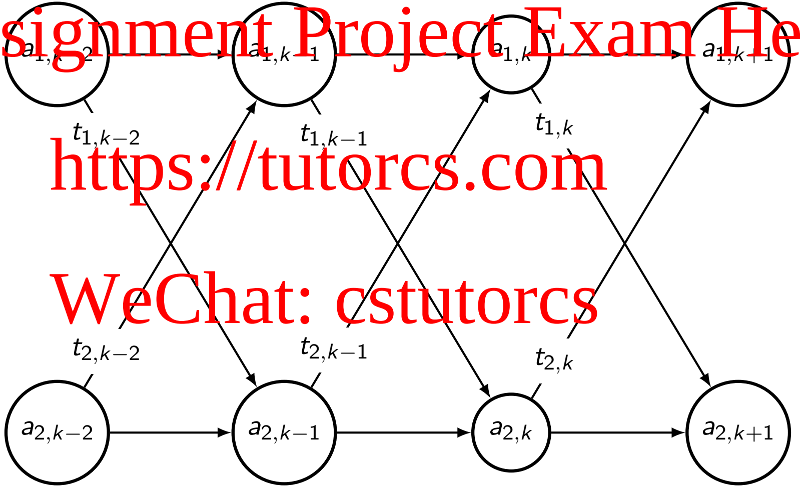
- There is no time required to continue from station k to station $k+1$ on the same line.

Task: Find a *fastest way* to assemble a product using both lines as necessary.

Assembly line scheduling



Assembly line scheduling



Assembly line scheduling

- We will denote internal vertices using the form (i, k) to represent workstation k on assembly line i.
- The problem requires us to find the shortest path from the start node to the finish node, where unlabelled edges have zero weight.
- This is clearly a directed acyclic graph, and moreover the topological ordering is obvious:

$$\text{start}, (1,1), (2,1), (1,2), (2,2), \ldots, (1,n), (2,n), \text{finish}.$$

  So we can use DP!
- There are $2n+2$ vertices and $4n$ edges, so the DP should take $O(n)$ time, whereas Dijkstra's algorithm would take $O(n \log n)$.

Assembly line scheduling

- We'll solve for the shortest path from s to each vertex (i, k).

- To form a recurrence, we should consider the ways of getting to workstation k on assembly line i.

- We could have come from workstation $k-1$ on either line, after completing the previous job.
- The exception is the first workstation, which leads to the base case.

Assembly line scheduling

Solution

Subproblems: for $i \in \{1, 2\}$ and $1 \leq k \leq n$, let $P(i, k)$ be the problem of determining $\text{opt}(i, k)$, the minimal time taken to complete the first k jobs, with the k^{th} job performed on assembly line i.

Recurrence: for $k > 1$,

$$\text{opt}(1, k) = \min\left(\text{opt}(1, k-1), \text{opt}(2, k-1) + t_{2,k-1}\right) + a_{1,k}$$
$$\text{opt}(2, k) = \min\left(\text{opt}(2, k-1), \text{opt}(1, k-1) + t_{1,k-1}\right) + a_{2,k}.$$

Base cases: $\text{opt}(1, 1) = s_1 + a_{1,1}$ and $\text{opt}(2, 1) = s_2 + a_{2,1}$.

Assembly line scheduling

- As the recurrence uses values from both assembly lines, we have to solve the subproblems in order of increasing k, solving both $P(1, k)$ and $P(2, k)$ at each stage.

- Finally, after obtaining $\text{opt}(1, n)$ and $\text{opt}(2, n)$, the overall solution is given by

$$\min\left(\text{opt}(1, n) + f_1, \text{opt}(2, n) + f_2\right).$$

- Each of $2n$ subproblems is solved in constant time, and the final two subproblems are combined as above in constant time also. Therefore the overall time complexity is $O(n)$.

Remark

This problem is important because it has the same design logic as the Viterbi algorithm, an extremely important algorithm for many fields such as speech recognition, decoding convolutional codes in telecommunications etc. This will be covered in COMP4121 Advanced Algorithms.

Single Source Shortest Paths

Problem

Instance: a directed weighted graph $G = (V, E)$ with edge weights $w(e)$ which can be negative, but without cycles of negative total weight, and a designated vertex $s \in V$.

Task: find the weight of the shortest path from vertex s to every other vertex t.

Notation

Let $n = |V|$ and $m = |E|$.

Single Source Shortest Paths

- How does this problem differ to the one solved by Dijkstra's algorithm?
- In this problem, we allow negative edge weights, so the greedy strategy no longer works.
- Note that we disallow cycles of negative total weight. This is only because with such a cycle, there is no shortest path - you can take as many laps around a negative cycle as you like.
- This problem was first solved by Shimbel in 1955, and was one of the earliest uses of Dynamic Programming.

Bellman-Ford algorithm

Observation

For any vertex t, there is a shortest $s-t$ path without cycles.

Proof Outline

Suppose the opposite. Let p be a shortest $s-t$ path, so it must contain a cycle. Since there are no negative weight cycles, removing this cycle produces an $s-t$ path of no greater length.

Observation

It follows that every shortest $s-t$ path contains any vertex v at most once, and therefore has at most $n-1$ edges.

Bellman-Ford algorithm

- For every vertex t, let's find the weight of a shortest $s - t$ path consisting of at most i edges, for each i up to $n - 1$.
- Suppose the path in question is

$$p = \underbrace{s \to \ldots \to v}_{p'} \to t,$$

with the final edge going from v to t.

- Then p' must be itself the shortest path from s to v of at most $i - 1$ edges, which is another subproblem!
- No such recursion is necessary if $t = s$, or if $i = 0$.

Bellman-Ford algorithm

Solution

Subproblems: for all $0 \leq i \leq n-1$ and all $t \in V$, let $P(i,t)$ be the problem of determining $\text{opt}(i,t)$, the length of a shortest path from s to t which contains at most i edges.

Recurrence: for all $i > 0$ and $t \neq s$,

$$\text{opt}(i,t) = \min\{\text{opt}(i-1,v) + w(v,t) \mid (v,t) \in E\}.$$

Base cases: $\text{opt}(i,s) = 0$, and for $t \neq s$, $\text{opt}(0,t) = \infty$.

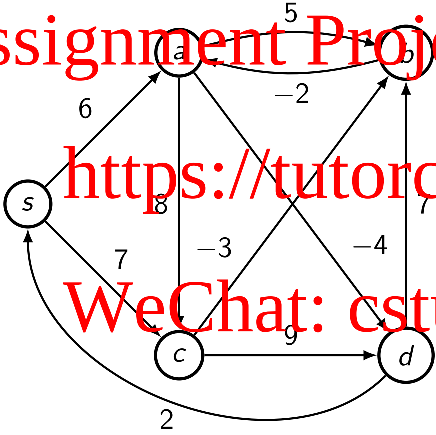
Bellman-Ford algorithm

- The overall solutions are given by $\text{opt}(n-1, t)$.

- We proceed in n rounds ($i = 0, 1, \ldots, n-1$). In each round, each edge of the graph is considered only once.

- Therefore the time complexity is $O(nm)$.

Bellman-Ford algorithm



$i \backslash t$	s	a	b	c	d
0	0	∞	∞	∞	∞
1	0	6	∞	7	∞
2	0	6	4	7	2
3	0	2	4	7	2
4	0	2	4	7	-2

$$\mathrm{opt}(i, t) = \min\{\mathrm{opt}(i-1, v) + w(v, t) \mid (v, t) \in E\}.$$

Bellman-Ford algorithm

- How do we reconstruct an actual shortest $s - t$ path?

- As usual, we'll store one step at a time and backtrack. Let $\text{pred}(i, t)$ be the immediate predecessor of vertex t on a shortest $s - t$ path of at most i edges.

- The additional recurrence required is

$$\text{pred}(i, t) = \underset{v \in V}{\text{argmin}}\{\text{opt}(i-1, v) + w(v, t) \mid (v, t) \in E\}.$$

Bellman-Ford algorithm

- There are several small improvements that can be made to this algorithm.

- As stated, we build a table of size $O(n^2)$, with a new row for each 'round'.

- It is possible to reduce this to $O(n)$. Including opt$(i-1,t)$ as a candidate for opt(i,t), doesn't change the recurrence, so we can instead maintain a table with only one row, and overwrite it at each round.

Bellman-Ford algorithm

- The SPFA (Shortest Paths Faster Algorithm) speeds up the later rounds by ignoring some edges. This optimisation and others (e.g. early exit) do not change the worst case time complexity from $O(nm)$, but they can be a significant speed-up in practical applications.

- The Bellman-Ford algorithm can also be augmented to detect cycles of negative weight.

All Pairs Shortest Paths

Problem

Instance: a directed weighted graph $G = (V, E)$ with edge weights $w(e)$ which can be negative, but without cycles of negative total weight.

Task: find the weight of the shortest path from every vertex s to every other vertex t.

Notation

Let $n = |V|$ and $m = |E|$.

Floyd-Warshall algorithm

- We can use a similar idea, this time in terms of the intermediate vertices allowed on an $s - t$ path.

- Label the vertices of V as $v_1, v_2, \ldots, v_n$.

- Let S be the set of vertices allowed as intermediate vertices. Initially S is empty, and we add vertices $v_1, v_2, \ldots, v_n$ one at a time.

Floyd-Warshall algorithm

Question

When is the shortest path from s to t using the first k vertices as intermediates actually an improvement on the value from the previous round?

Answer

When there is a shorter path of the form

$$s \to \underbrace{\ldots}_{v_1,\ldots,v_{k-1}} \to v_k \to \underbrace{\ldots}_{v_1,\ldots,v_{k-1}} \to t.$$

Floyd-Warshall algorithm

Solution

Subproblems: for all $1 \leq i, j \leq n$ and $0 \leq k \leq n$, let $P(i, j, k)$ be the problem of determining $\text{opt}(i, j, k)$, the weight of a shortest path from v_i to v_j using only $v_1, \ldots, v_k$ as intermediate vertices.

Recurrence: for all $1 \leq i, j, k \leq n$,

$$\text{opt}(i, j, k) = \min(\text{opt}(i, j, k-1), \text{opt}(i, k, k-1) + \text{opt}(k, j, k-1)).$$

Base cases:

$$\text{opt}(i, j, 0) = \begin{cases} 0 & \text{if } i = j \\ w(i, j) & \text{if } (v_i, v_j) \in E \\ \infty & \text{otherwise.} \end{cases}.$$

Floyd-Warshall algorithm

- Since $P(i, j, k)$ depends on $P(i, j, k-1)$, $P(i, k, k-1)$ and $P(k, j, k-1)$, we solve subproblems in increasing order of k.

- The overall solutions are given by $\text{opt}(i, j, n)$, where all vertices are allowed as intermediates.

- Each of $O(n^3)$ subproblems is solved in constant time, so the time complexity is $O(n^3)$.

- The space complexity can again be improved by overwriting the table every round.

Table of Contents

Puzzle

You have 2 lengths of fuse that are guaranteed to burn for precisely 1 hour each. Other than that fact, you know nothing; they may burn at different (indeed, at variable) rates, they may be of different lengths, thicknesses, materials, etc.

How can you use these two fuses to time a 45 minute interval?

That's All, Folks!!