Dynamic Programming

Carlos Cotta

Departamento de Lenguajes y Ciencias de la Computación
Universidad de Málaga

http://www.lcc.uma.es/∼ccottap

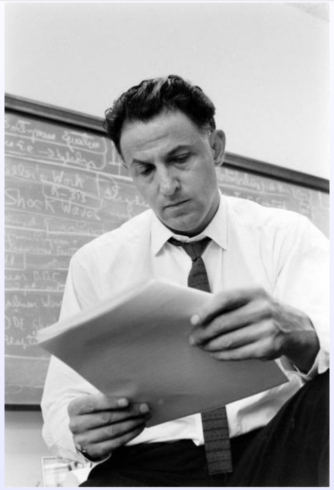
Comput Eng, Softw Eng, Comput Sci & Math – 2023-2024

UNIVERSIDAD DE MÁLAGA

Dynamic Programming

1. Introduction
   - General Ideas
   - Example: Binomial Coefficients
2. Dynamic Programming
   - Basic Elements
   - Example: Coin Change Problem
3. Functions with Memory
   - Top-Down Approach
   - Example: Coin Change Revisited
4. Examples
   - The 0-1 Knapsack Problem
   - All Shortest Paths

Historical Background



Richard E. Bellman
(1920-1984)

Dynamic programming was invented by American mathematician Richard Bellman back in the 1950's, for solving multi-stage decision problems.

"Programming" refers to "planning" in 50's jargon, i.e., using a table-based solving method.

It is a general design approach.

Applicability

Dynamic programming is often (but not always), used to solve combinatorial optimization problems.

It is useful when problems can be divided in overlapping subproblems.

We will commonly use tables to store intermediate results (subproblems) to build the final solution.

Computing Binomial Coefficients

Although not an optimization problem, it can be approached by dynamic programming.

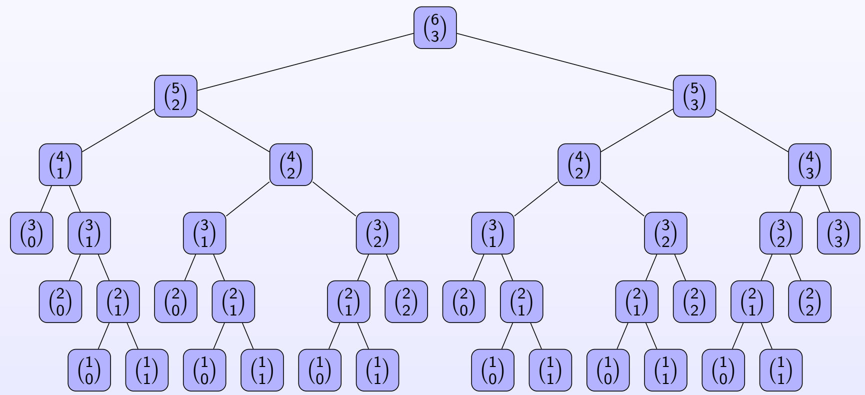
Binomial coefficient $C(n,k)$ can be expressed as:

$$C(n,k) = \begin{cases} 1 & (k=0) \vee (n=k) \\ C(n-1,k-1) + C(n-1,k) & n > k > 0 \end{cases}$$

Note the occurrence of overlapped subproblems:

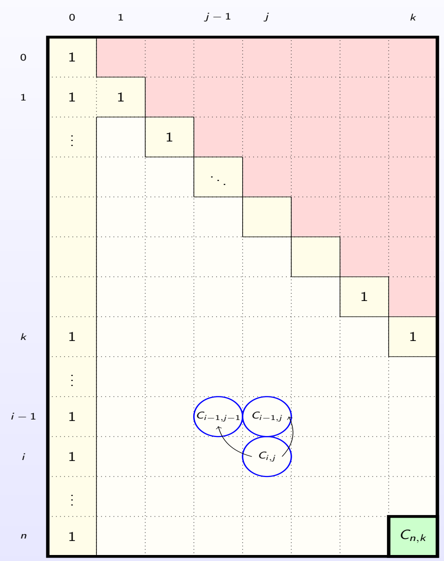
$$\begin{aligned} C(6,3) &= C(5,2) + C(5,3) = \\ &= (C(4,1) + C(4,2)) + (C(4,2) + C(4,3)) = \cdots \end{aligned}$$

Repeated Subproblems



Store Intermediate Results

To compute overlapped (i.e., repeated) terms efficiently, a table is built:



The Algorithm

Binomial Coefficients

```
func BinomialCoefficient (↓n, k: ℕ):ℕ
variables
    i, j:ℕ
    C:ARRAY [0..n][0..k] OF ℕ
begin
    for i ← 0 to n do
        for j ← 0 to min(i,k) do
            if (j= 0)∨(j=i) then C[i,j] ← 1
            else C[i,j] ← C[i−1,j−1]+C[i−1,j]
        endfor
    endfor
    return C[n,k]
end
```

Complexity

Let the sum be the basic operation. Each table cell requires computing 1 sum (except column $j = 0$ and diagonal $j = i$ that require none). Thus:

$$
\begin{aligned}
t(n,k) &= \sum_{i=1}^{k}\sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^{n}\sum_{j=1}^{k} 1 = \\
&= \sum_{i=1}^{k}(i-1) + \sum_{i=k+1}^{n} k = \\
&= \frac{k(k-1)}{2} + k(n-k) \in \Theta(k^2 + nk)
\end{aligned}
$$

Since $n \geqslant k$, $\Theta(k^2 + nk) = \Theta(\max(k^2, nk)) = \Theta(nk)$.
The additional storage used is also $\Theta(nk)$.

DP Step by Step

1. Characterize the structure of the optimal solution
   1. Determine what is the solution.
   2. Determine how to construct the solution.
   3. Determine how to evaluate the solution.
   4. Establish the optimal substructure property.
   5. Determine the subproblems to be solved.
2. Define recursively the value of the optimal solution
3. Compute the optimal value in an bottom-up fashion.
   1. Design an adequate data structure to store values/decisions.
   2. Determine dependencies within the data structure.
   3. Find an appropriate traversal order.
   4. Devise the bottom-up algorithm.
4. Reconstruct the optimal solution using this information.

Optimality Principle

Optimal substructure

A problem has the optimal substructure property if its optimal solution comprises the optimal solution for smaller subproblems.

Bellman's principle of optimality

An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

Optimality Principle

In other words, let $d_1, \cdots, d_n$ be the optimal sequence of decisions in a multi-stage problem. If s_i is the state (partial solution) after taking decision d_i, then sequence $d_{i+1}, \cdots, d_n$ is optimal for completing the solution, regardless of decisions $d_1, \cdots, d_i$.

Bellman's principle of optimality is a necessary condition for applying dynamic programming.

It may also hint that an optimal greedy approach may be possible (this will be tackled in next unit).

Optimality Principle

Example: shortest path in a graph

Let us consider the problem of finding the shortest path (in terms of edges traversed) between nodes u, v in graph $G(V, E)$.

1. If $u = v$ or $(u, v) \in E$ the problem is trivial.
2. Otherwise, let p be an optimal path from u to v. There will be some intermediate node w, and hence

$$u \overset{p}{\rightsquigarrow} v = u \overset{p_1}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$$

The path p_1 from u to w must be optimal. If it were not, there would be a shorter path – call it p_1'. Then

$$u \overset{p'}{\rightsquigarrow} v = u \overset{p_1'}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$$

would be shorter than p, but this is impossible because p was optimal by the initial assumption. Hence, p_1 is the optimal path from u to w (and the same holds for p_2).

Overlapped Subproblems

The optimal substructure principle is exploited in a bottom-up fashion: small subproblems are optimally solved, and their solutions combined to form the whole global solution.

We have to make a choice regarding which subproblems have to be used for building the optimal solution. Its cost is the cost of the subproblem(s) solution(s) plus the cost of making the choice.

The total number of subproblems must be low (i.e., polynomial in the size of the input).

Overlapped Subproblems

Recursion and subproblem overlapping imply the same subproblems will be found over and over again.

A table can be built to store the solutions to subproblems, thus avoiding recomputations.

Storing decisions made along with solution costs allows reconstructing the optimal solution.

The Coin Change Problem

Coin Change

We have an unlimited supply of coins of n different denominations $d_1, \cdots, d_n$. We have to pay M cents using the minimal number of coins.

The Coin Change Problem

Solutions

Solutions

Solutions are multisets $S = [\![m_1, \ldots, m_k]\!]$, where each $m_i \in \{1, \ldots, n\}$ is one of the coins used.

Valid solutions are those that add up to M cents, i.e.,

$$v(S) = \sum_{j=1}^{n} [S]_j \cdot v_j = M$$

where $[S]_j$ is the cardinality of j in S (that is, the number of coins of type j in S).

The Coin Change Problem

Objective function

Objective function

The goal is minimizing the number of coins used, that is, the cardinality of the solution:

$$\min f(S) = |S| = \sum_{j=1}^{n} [S]_j \ .$$

The Coin Change Problem

Decisions

Decisions

Let S be a partial solution. S can be extended in different ways:

1. Binary decisions: given a certain coin type, we decide whether we use it or not.
2. Multi-way decisions:
   1. given a certain coin type, we decide how many coins of that type we shall use.
   2. among all coin types, we decide which one we are going to use.

The Coin Change Problem

Optimal substructure – Binary decisions

Let $n, n-1, n-2, \ldots, 2, 1$ be the order in which we consider the coin types when taking decisions.

Assume we know the optimal solution S^*. Assume that out first decision was:

- Not using a coin of type n: trivially, S^* is the optimal solution using coins of types $n-1, n-2, \ldots, 2, 1$.
- Using a coin of type n: let $S' = S^* \setminus \{n\}$. Then

$$v(S') = v(S^*) - d_n = M - d_n$$

$$f(S') = |S'| = |S^*| - 1 = f(S^*) - 1$$

Assume S' is not optimal for returning $M - d_n$ cents.

The Coin Change Problem
Optimal substructure – Binary decisions

If S' is not optimal for $M - d_n$ cents, there is a better solution S'':

$$v(S'') = M - d_n$$

$$f(S'') = |S''| < f(S')$$

If S'' exists, we can construct $S''' = S'' \cup \{n\}$. Then,

$$v(S''') = v(S'') + d_n = M$$

$$f(S''') = |S'''| = |S''| + 1 < f(S') + 1 = f(S^*)$$

Thus, S''' would be better than S^* for the original problem, which is impossible since we were assuming S^* to be the optimal solution.

The Coin Change Problem
Subproblems

If we use a coin, we decrease the target amount of money. If we do not use a coin type, we decrease the coin types allowed. Hence, the subproblems are thus characterized by the target amount of money j and the type of coin i on which the current decision is being taken.

Subproblems considered

$C_{i,j}$ = minimum number of coins needed to add up j cents using only coins of values $d_1, \cdots, d_i$.

The initial problem to be solved is therefore $C_{n,M}$.

The Coin Change Problem
Bellman equation

Base cases:

1. $C_{i,0} = 0$ for $1 \leqslant i \leqslant n$.
2. $C_{1,j} = \infty$ for $j < d_1$.

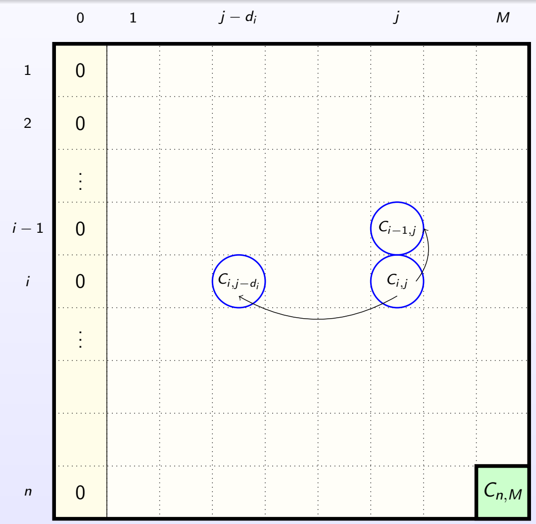
General case: we have two options:

1. not using any coin d_i. Then $C_{i,j} = C_{i-1,j}$.
2. use at least one coin d_i. Then $C_{i,j} = 1 + C_{i,j-d_i}$.

We are minimizing, and hence

$$C_{i,j} = \min\left(C_{i-1,j}, 1 + C_{i,j-d_i}\right)$$

The Coin Change Problem
Data structure required



We fill the table from top to bottom, from left to right.

The Coin Change Problem
Example

Assume hypothetical coins of 1, 4 and 6 cents. We have to pay $M = 8$ cents.

	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3	1	2	3	4	2
$d_3 = 6$	0	1	2	3	1	2	1	2	2

The optimal solution is using 2 coins of 4 cents.

The Coin Change Problem
The algorithm

Coin Change

func CoinChange ($\downarrow d$: ARRAY $[1..n]$ OF $\mathbb{N}$, $\downarrow M$: $\mathbb{N}$):$\mathbb{N}$
variables i, j:$\mathbb{N}$; C:ARRAY $[1..n][0..M]$ OF $\mathbb{N}$
begin
  for $i \leftarrow 1$ **to** n **do** $C[i,0] \leftarrow 0$ **endfor**
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** M **do**
      if $(i=1) \wedge (j<d[i])$ **then** $C[i,j] \leftarrow \infty$
      else if $i=1$ **then** $C[i,j] \leftarrow 1 + C[i, j-d[1]]$
      else if $j<d[i]$ **then** $C[i,j] \leftarrow C[i-1,j]$
      else $C[i,j] \leftarrow \min(C[i-1,j], 1+C[i, j-d[i]])$
    endfor
  endfor
  return $C[n, M]$
end

The Coin Change Problem
Solution reconstruction

Coin Change

func CoinChangeReconstruct ($\downarrow C$: ARRAY $[1..n, 0..M]$ OF $\mathbb{N}$, $\downarrow M$: $\mathbb{N}$):$[\![\mathbb{N}]\!]$
variables i, j:$\mathbb{N}$; B:$[\![\mathbb{N}]\!]$
begin
  $B \leftarrow \emptyset; i \leftarrow n; j \leftarrow M$
  while $j > 0$ **do**
    if $(i = 1) \vee (C[i,j] \neq C[i-1,j])$ **then**
      $B \leftarrow B \cup \{i\}$
      $j \leftarrow j - d_i$
    else
      $i \leftarrow i - 1$
    endif
  endwhile
  return B
end

The Best of Both Worlds

Due to overlapped problems, a recursive solution (top-down) can provide a simple yet very inefficient algorithm.

A standard bottom-up dynamic programming approach avoids recomputation by storing in a table the solution of all possible subproblems, maybe including problems that are not required at all for a given problem instance.

By using functions with memory we aim to combine the simplicity of the top-down approach and the efficiency of the bottom-up approach.

Memoization (not a typo)

The goal is solving just the required subproblems, and doing it just once.

A memoized algorithm uses a look-up table to store partial results:

1. Each cell stores the solution to a subproblem.
2. All cells are initially marked as "empty".
3. Cells are filled whenever the subproblem is solved by the first time.
4. Further attempts on the same subproblem are solved by checking the table.

The Coin Change Problem

Memoized algorithm

Coin Change

func CoinChange ($\downarrow d$: ARRAY $[1..n]$ OF $\mathbb{N}$, $\downarrow M$: $\mathbb{N}$):$\mathbb{N}$
variables i, j:$\mathbb{N}$; C:ARRAY $[1..n][0..M]$ OF $\mathbb{Z}$
begin
  // Initializes the table
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** M **do** $C[i, j] \leftarrow -1$ **endfor**
    $C[i, 0] \leftarrow 0$
  endfor
  return CoinChangeRec(n, d, M, C)
end

The Coin Change Problem

Memoized algorithm

Coin Change

func CoinChangeRec ($\downarrow i$: $\mathbb{N}$, $\downarrow d$: ARRAY $[1..n]$ OF $\mathbb{N}$,
    $\downarrow M$: $\mathbb{N}$, $\downarrow\uparrow C[][]$: $\mathbb{Z}$):$\mathbb{N}$
variables $n1, n2$:$\mathbb{N}$
begin
  if $C[i, M] = -1$ **then**
    if $(i = 1) \wedge (M < d[i])$ **then** $C[i, M] \leftarrow \infty$
    else if $i = 1$ **then**
      $C[i, M] \leftarrow 1 +$CoinChangeRec($1, d, M - d[1], C$)
    else if $M < d[i]$ **then**
      $C[i, M] \leftarrow$ CoinChangeRec($i - 1, d, M, C$)
  // continues...

The Coin Change Problem

Memoized algorithm

Coin Change

  // continued...
    else
      $n1 \leftarrow$ CoinChangeRec($i - 1, d, M, C$)
      $n2 \leftarrow 1+$CoinChangeRec($i, d, M - d[i], C$)
      $C[i, M] \leftarrow \min(n1, n2)$
    endif
  endif
  return $C[i, M]$
end

The Coin Change Problem

Memoized algorithm

Considering the same problem instance as before:

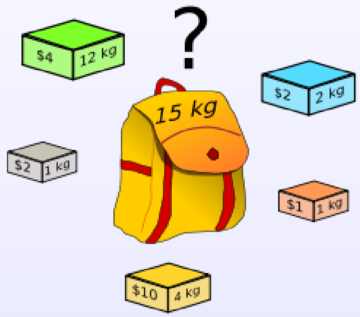
	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	-	2	-	1	-	-	-	2
$d_3 = 6$	-	-	2	-	-	-	-	-	2

Problem Statement

0-1 KNAPSACK

Given a collection $B = \{o_1, \cdots, o_n\}$ of n objects with weights $w_1, \cdots, w_n$ and values $v_1, \cdots, v_n$, find the subset $S \subseteq B$ with the highest value, such that its total weight does not exceed W.

We will denote knapsack instances as $\langle B, W \rangle$.



Solutions

Solutions

Solutions are sets $S = \{o_{i_1}, \ldots, o_{i_k}\} \subseteq B$.

Valid solutions are those that do not weigh more than W, i.e.,

$$w(S) = \sum_{o_j \in S} w_j = M .$$

Objective function

Objective function

The goal is maximizing the value of objects in the solution:

$$\max f(S) = \sum_{o_j \in S} v_j .$$

Decisions

Decisions

Let S be a partial solution. S can be extended in different ways:

1. Binary decisions: given a certain object, we decide whether we use it or not.
2. Multi-way decisions: among all available objects, we decide which one we are going to use.

Optimal substructure – Binary decisions

Let $o_n, o_{n-1}, \ldots, o_1$ be the order in which we consider the objects when taking decisions.

Assume we know the optimal solution S^*. Assume that out first decision was:

- Not using object o_n: trivially, S^* is the optimal solution for the problem instance $\langle B \setminus \{o_n\}, W\rangle$.
- Using object o_n: let $S' = S \setminus \{o_n\}$. Then

$$w(S') = w(S^*) - w_n \leqslant W - w_n$$

$$f(S') = f(S^*) - v_n$$

Assume S' is not optimal for the problem instance $\langle B \setminus \{o_n\}, W - w_n\rangle$.

Optimal substructure – Binary decisions

If S' is not optimal for $\langle B \setminus \{o_n\}, W - w_n\rangle$, there is a better solution S'':

$$w(S'') \leqslant W - w_n$$

$$f(S'') > f(S')$$

If S'' exists, we can construct $S''' = S'' \cup \{o_n\}$. Then,

$$w(S''') = w(S'') + w_n \leqslant M$$

$$f(S''') = f(S'') + v_n > f(S') + v_n = f(S^*)$$

Thus, S''' would be better than S^* for $\langle B, W\rangle$, which is impossible since we were assuming S^* to be the optimal solution.

Subproblems

The state of the problem can then be characterized by the remaining knapsack capacity and the objects whose inclusion in the knapsack is still undecided.

Subproblems considered

$V_{i,j}$ = highest value of a subset of elements from $\{o_1, \cdots, o_i\}$ that fits in a knapsack of capacity j.

The goal is finding $V_{n,W}$.

Bellman Equation

There are two possibilities for $V_{i,j}$:

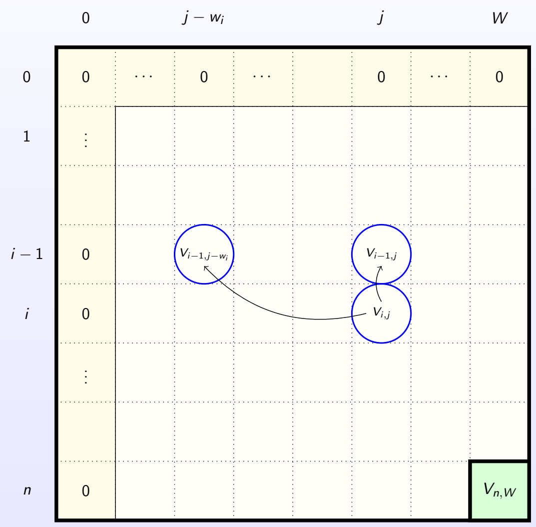
1. not using o_i; then $V_{i,j} = V_{i-1,j}$.
2. using o_i; then $V_{i,j} = v_i + V_{i-1,j-w_i}$.

The problem is trivial if $i = 0$ (no objects) or $j = 0$ (no available space). In both cases the optimal value is 0.

$$V_{i,j} = \begin{cases} 0 & (i=0) \vee (j=0) \\ V_{i-1,j} & (i>0) \wedge (j>0) \wedge (p_i > j) \\ \max(V_{i-1,j}, v_i + V_{i-1,j-w_i}) & (i>0) \wedge (j>0) \wedge (p_i \leqslant j) \end{cases}$$

We use a table $V[i,j]$ $(0 \leqslant i \leqslant n,\ 0 \leqslant j \leqslant W)$ to store partial results.

Data Structure Used



We fill the table from top to bottom (left to right or vice versa, since there are no dependencies within a row).

Algorithm

0-1 Knapsack

```
func Knapsack (↓v, p: ARRAY [1..n] OF ℕ, ↓W: ℕ):ℕ
variables i, j:ℕ; V:ARRAY [0..n][0..W] OF ℕ
begin
    for j ← 0 to W do V[0,j] ← 0 endfor
    for i ← 0 to n do V[i,0] ← 0 endfor
    for i ← 1 to n do
        for j ← 1 to W do
            if (j<p[i]) then V[i,j] ← V[i−1,j]
            else V[i,j] ← max(V[i−1,j], v[i]+V[i−1,j−p[i]])
            endif
        endfor
    endfor
    return V[n,W]
end
```


Example

Consider the instance:

object	weight	value
1	2	12€
2	1	10€
3	3	20€
4	2	15€

W=5

We obtain:

	capacity					
i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

Reconstructing the Optimal Solution

	i	capacity 0	1	2	3	4	5
	0	0	0	0	0	0	0
$p_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$p_2 = 1, v_2 = 10$	2	0	10	12	22	22	22
$p_3 = 3, v_3 = 30$	3	0	10	12	22	30	32
$p_4 = 2, v_4 = 15$	4	0	10	15	25	30	37

The optimal value is $V[4,5] = 37$€.

- $V[4,5] \neq V[3,5] \Rightarrow o_4$ is in the optimal solution. We substract 2 from the capacity.
- $V[3,3] = V[2,3] \Rightarrow o_3$ is not in the optimal solution.
- $V[2,3] \neq V[1,3] \Rightarrow o_2$ is in the optimal solution. We substract 1 from the capacity.
- $V[1,2] \neq V[0,2] \Rightarrow o_1$ is in the solution.

Problem Statement

All Shortest Paths

Given a connected undirected graph $G(V, E)$ in which each edge $e \in E$ has a weight w_e, find the shortest path between each pair of nodes $u, v \in V$.

We can assume $V = \{1, \ldots, n\}$. Then, the graph can be represented by a matrix M such that $M_{uv} = w_{(u,v)}$ is the weight of edge (u, v) ($M_{uv} = \infty$ if $(u, v) \notin E$).

Solutions

We want to determine for each pair of vertices $u, v \in V$ the shortest path $u \overset{p}{\leadsto} v$.

Any path $u \overset{p}{\leadsto} v$ can be represented as a sequence of vertices $\langle w_0, w_1, w_2, \ldots, w_{m_p} \rangle$, where m_p is the number of vertices in the path, $w_0 = u$ and $w_{m_p} = v$.

Decisions

We can construct a path by specifying which will be the vertices $w_1, \ldots, w_{m_p - 1}$ that we visit between u and v. These need not be decided in any particular order.

Objective Function

The quality of a path $u \overset{p}{\rightsquigarrow} v = \langle w_0, w_1, w_2, \ldots, w_{m_p} \rangle$ is given by

$$f(\langle w_0, w_1, w_2, \ldots, w_{m_p} \rangle) = \sum_{i=1}^{m_p} M_{w_{i-1}w_i} \ .$$

Optimal Substructure Property

As we saw, the shortest path among two nodes u, v exhibits optimal substructure: if the path p between them is not trivial and goes through an intermediate node w,

$$u \overset{p}{\rightsquigarrow} v = u \overset{p_1}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$$

then p_1 and p_2 are the optimal paths from u to w and from w to v respectively.

Subproblems

We have to take decisions on intermediate vertices in the path. Without loss of generality, let this decision be taken on the vertex with highest index in the path.

Subproblems considered

$C_{i,j,k}$ = length of the shortest path between nodes i and j, assuming this path can only go through nodes in $\{1, \cdots, k\}$.

We aim to compute $C_{i,j,n}$ for each $i, j \in V$

Bellman Equation

Base case ($k = 0$, i.e., no intermediate nodes in any path):

- $C_{i,j,k} = M_{i,j}$

General case ($k > 0$). There are two options:

- Not using node k. Then, $C_{i,j,k} = C_{i,j,k-1}$
- Using node k. Then, $C_{i,j,k} = C_{i,k,k-1} + C_{k,j,k-1}$

Therefore:

$$C_{i,j,k} = \begin{cases} M_{i,j} & k = 0 \\ \min\left(C_{i,j,k-1}, C_{i,k,k-1} + C_{k,j,k-1}\right) & k > 0 \end{cases}$$

Data Structure Used

C is a 3-dimensional matrix but we only need to keep a single 2-dimensional slice at any given moment (all dependencies are between the $(k-1)$-th slice and the k-th slice).

We thus use a matrix $S_{1\ldots n,1\ldots n}$ that represents the k-th slice of $C_{1\ldots n,1\ldots n,1\ldots n}$.

We update the matrix n times, one for each slice across the third dimension of the original matrix. In the k-th iteration:

$$S_{i,j} = \min\left(S_{i,j}, S_{i,k} + S_{k,j}\right)$$

Algorithm

Floyd-Warshall Algorithm

func AllShortestPaths ($\downarrow M$: ARRAY $[1..n,1..n]$ OF $\mathbb{R}$):ARRAY $[1..n,1..n]$ OF $\mathbb{R}$
variables i,j,k:$\mathbb{N}$; S:ARRAY $[0..n][0..n]$ OF $\mathbb{N}$
begin
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
      $S[i,j] \leftarrow M[i,j]$
    endfor
  endfor
  for $k \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$ **to** n **do**
      for $j \leftarrow 1$ **to** n **do**
        $S[i,j] \leftarrow \min(S[i,j], S[i,k]+S[k,j])$
      endfor
    endfor
  endfor
  return C
end

Path Reconstruction

To reconstruct the optimal paths we need a trace matrix T in which we store the decisions taken to update matrix S. Let $T_{i,j}^{(k)}$ be the value of $T_{i,j}$ in the k-th iteration:

$$T_{i,j}^{(k)} = \begin{cases} j & k = 0 \\ T_{i,j}^{(k-1)} & C_{i,j,k} = C_{i,j,k-1} \\ k & C_{i,j,k} \neq C_{i,j,k-1} \end{cases}$$

If $T_{i,j}^{(n)} = j$, then the optimal path from i to j is trivial (direct connection).

If $T_{i,j}^{(n)} = k \neq j$, then the optimal path from i to j is the concatenation of the optimal path from i to k and from k to j.

Algorithm Complexity

Since:

- computing $S_{i,j}$ is $\Theta(1)$,
- S has size $\Theta(n^2)$,
- k goes from 0 to n,

the overall time complexity is $\Theta(n^3)$.

As for the memory, the space complexity is just $\Theta(n^2)$.

Specific Bibliography

C. Cotta
Programación Dinámica. Introducción y Ejercicios Resueltos,
UMA Editorial, 2018
http://www.lcc.uma.es/~ccottap/PD

Image Credits

- Picture of Richard Bellman: by Alfred Eisenstaedt, 1962, Life Magazine, © Time Inc.
- Euro coins: public domain
- Knapsack cartoon: Dake, CC-BY-SA