AAA – Dynamic Programming

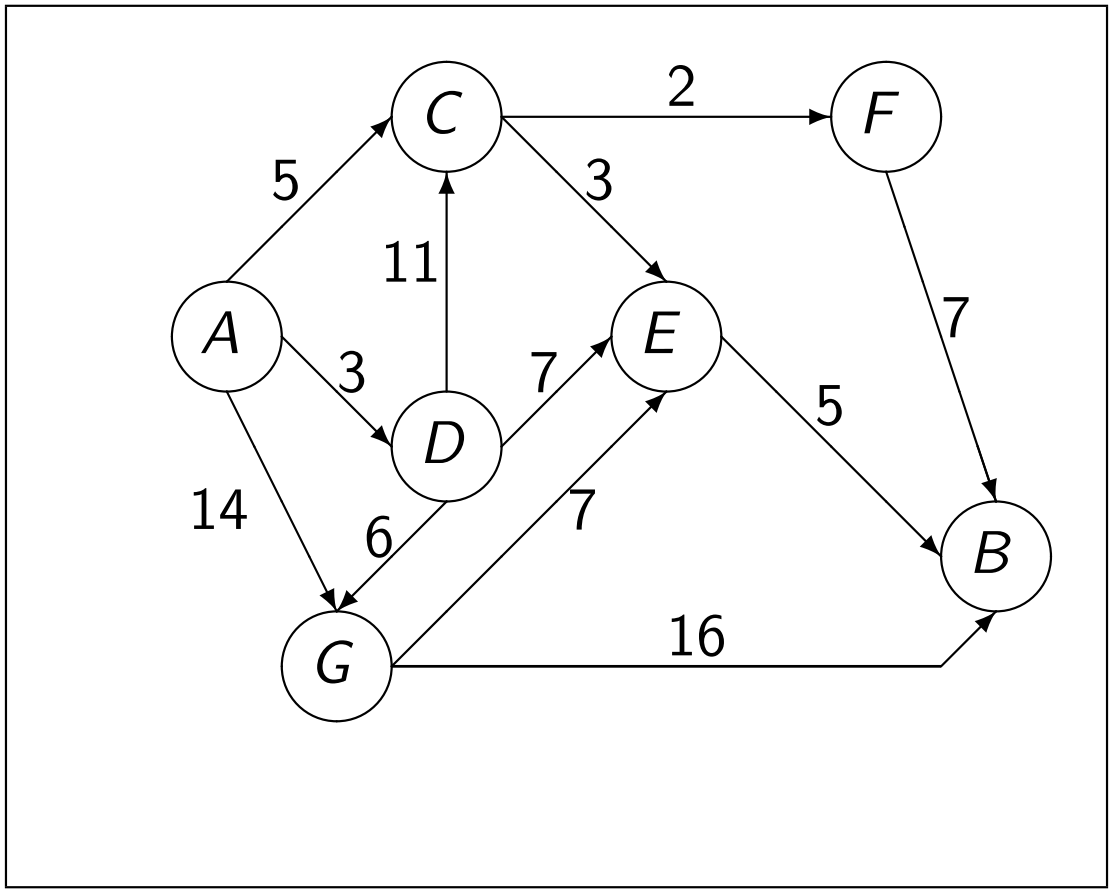
Ian Sanders

First Semester, 2023

Weary Traveller

Problem: Given a weighted directed acyclic graph and a pair of nodes in the graph find a shortest path from the start node to the destination node.

Example In the graph below find the shortest path from A to B

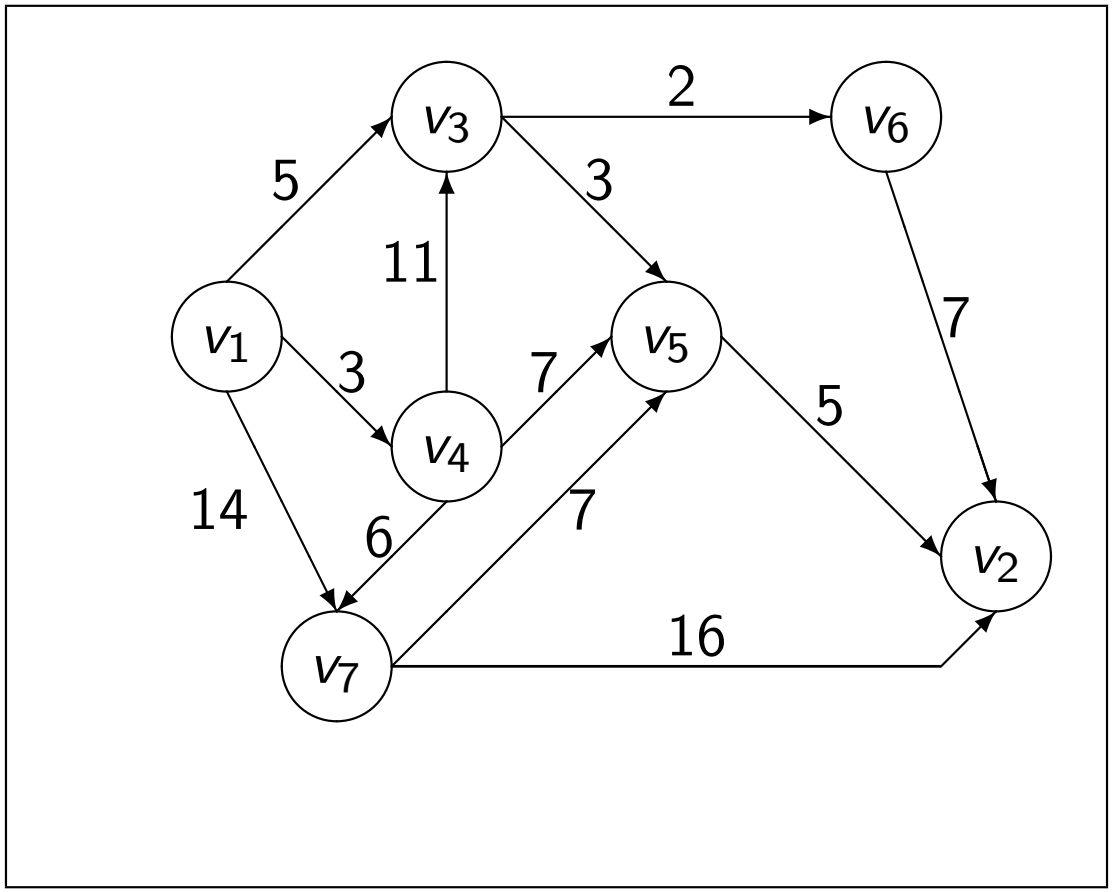


Obvious greedy approach – Repeatedly add the shortest path from the city reached so far to any city which has not yet been visited.
Would give A, D, G, E, B for a distance of 21
Actual shortest path is A, C, E, B for a distance of 13
Greedy approach does not work.

The weary traveller/shortest path problem revisited

Problem: Given a weighted directed acyclic graph and a pair of nodes in the graph find a shortest path from the start node to the destination node.

Example In the graph below find the shortest path from v_1 to v_2



The obvious greedy approach for solving this problem – repeatedly add the shortest path from the city reached so far to any city which has not yet been visited – does not always give the shortest distance.
Notice that

$$D(v_1) = minimum(5 + D(v_3), 3 + D(v_4), 14 + D(v_7))$$

where $D(v_k)$ is used to indicate the shortest distance from city v_k to v_2
So we are solving this problem by first solving three smaller problems of the same type and then "combining" the solutions from those problems to get the solution to our original problem.

This problem exhibits the "optimal subproblem property"
Finding $D(v_3)$, $D(v_4)$ and $D(v_7)$ is done in the same way.
For example,

$$D(v_4) = \mathit{minimum}(11 + D(v_3), 7 + D(v_5), 6 + D(v_7))$$

Note that breaking down the problem like this implies a recursive solution.
However, as we can see in our simple example, such a recursive approach would lead to recalculation of some distances.
Here $D(v_3)$ would be calculated in determining $D(v_4)$ as well as $D(v_1)$.
A better (more efficient) way of dealing with this problem is to use *dynamic programming* and to attack it in a "bottom up" style.

D

Vertex	Dist to v_e
1	
2	
3	
4	
5	
6	
7	

W

	1	2	3	4	5	6	7
1							
2							
3							
4							
5							
6							
7							

P

Vertex	Still possible
1	
2	
3	
4	
5	
6	
7	

T

Vertex	In terminal set
1	
2	
3	
4	
5	
6	
7	

D

Vertex	to v_2
1	∞
2	0
3	∞
4	∞
5	∞
6	∞
7	∞

W

	1	2	3	4	5	6	7
1	∞	∞	5	3	∞	∞	14
2	∞	∞	∞	∞	∞	∞	∞
3	∞	∞	∞	∞	3	2	∞
4	∞	∞	11	∞	7	∞	6
5	∞	5	∞	∞	∞	∞	∞
6	∞	7	∞	∞	∞	∞	∞
7	∞	16	∞	∞	7	∞	∞

P

Vertex	Still possible
1	1
2	0
3	1
4	1
5	1
6	1
7	1

T

Vertex	In terminal set
1	0
2	1
3	0
4	0
5	0
6	0
7	0

D

Vertex	to v_2
1	∞
2	0
3	∞
4	∞
5	5
6	7
7	12

W

	1	2	3	4	5	6	7
1	∞	∞	5	3	∞	∞	14
2	∞	∞	∞	∞	∞	∞	∞
3	∞	∞	∞	∞	3	2	∞
4	∞	∞	11	∞	7	∞	6
5	∞	5	∞	∞	∞	∞	∞
6	∞	7	∞	∞	∞	∞	∞
7	∞	16	∞	∞	7	∞	∞

P

Vertex	Still possible
1	1
2	0
3	1
4	1
5	0
6	0
7	0

T

Vertex	In terminal set
1	0
2	1
3	0
4	0
5	1
6	1
7	1

Complexity

Initialising the arrays D, P and T takes $O(n)$ time and initialising the matrix W takes $O(n^2)$ time.
So overall initialisation is $O(n^2)$ time.
At most $n-1$ passes until $D(v_s)$ is not ∞.
On each pass we consider every vertex but actually only do work for those vertices which are not in the terminal set on that pass.
First determine if the vertex is on the "fringe" and then
if it is on the fringe determine the distance from it to the destination vertex using the distances that have already been calculated in earlier passes.
Both of these tasks are $O(n)$.
So $O(n^2)$ work on each pass.
The worst case this algorithm takes $O(n^3)$ time overall.

Correctness

Essentially by induction...

Comment

This algorithm has a number of features which are common in dynamic programming.

- The approach used to tackle the problem seems to be a recursive solution but one where we might repeatedly calculate the solutions to some problems. To avoid doing this we work bottom up – building the solution to the original problem up from solutions to smaller subproblems.
- We use various "tables" to store the information about the subproblems for use in solving the original problem.

Making change

Recall that the greedy approach to making change of repeatedly choosing the coin with the biggest denomination less than the change required did not always produce a solution with the minimum number of coins.

Whether the greedy approach worked or not depended on the coin set being used.

Here we consider a dynamic programming solution which will *always* find the minimum number of coins.

Suppose that we are using a currency that has n coins of denominations d_i, $1 \leq i \leq n$ and such that $0 < d_1 < d_2 < \ldots < d_n$.

The problem then is to give the minimum number of coins to make up some given value, say C, of change.

Consider the coin set $1, 2, 5, 10, 20, 25, 50, 100$

The greedy approach would not work to make change for 42.

We notice that the optimal solution could be arrived at in one of two ways using coins in the set $1, 2, 5, 10, 20, 25$
It could include a 25 or it could be made up without a 25
If it includes a 25 then the solution would be one coin plus the number of coins required to make up 17 $(42 - 25)$
If it does not include a 25 then the solution would be the same as the solution of making 42 from the set $1, 2, 5, 10, 20$
The solution would then be whichever of these two solutions requires the fewest coins.

In general, if $c[i, j]$ is used to denote the number of coins required to make change for j using the denominations of coins from 1 up to i (i meaning the ith coin in the set not the denomination i) then we can write this choice out as

$$c[i, j] = minimum(c[i-1, j], 1 + c[i, j - d_i])$$

Again it seems that this is inherently a recursive solution but once again we note that if we solved the problem recursively then we would be recalculating some results.

As before to set up a dynamic programming solution we will create a table to store the partial solutions from which we will make up final solution.

Here we create a table $c[1..n, 0..C]$ with one row for each available denomination (from smallest at the top to largest at the bottom) and one column for each value of change from 0 to C. We all set $c[i, 0] = 0$ for all i.

Suppose we are trying to make up 9 from $1, 2, 5$

Amount	0	1	2	3	4	5	6	7	8	9
$d_1 = 1$	0									
$d_2 = 2$	0									
$d_3 = 5$	0									

Amount	0	1	2	3	4	5	6	7	8	9
$d_1 = 1$	0	1	2	3	4	5	6	7	8	9
$d_2 = 2$	0	1	1	2	2	3	3	4	4	5
$d_3 = 5$	0	1	1	2	2	1	2	2	3	3

Here to fill $c[3, 9]$ (making change for 9 using coins from d_1, d_2 and d_3) we would use the equation

$$c[3, 9] = \mathit{minimum}(c[3 - 1, 9], 1 + c[3, 9 - d_3])$$

$$c[3, 9] = \mathit{minimum}(c[2, 9], 1 + c[3, 9 - 5]) = \mathit{minimum}(5, 3)$$

Efficiency

First we set $c[i, 0]$ to be 0 for all i and then we fill $n \times C$ other positions in the array.
Filling each position in the array takes constant time so the algorithm is $O(nC)$ overall.

Correctness

Induction and follows directly from the observation that

$$c[i,j] = minimum(c[i-1,j], 1+c[i,j-d_i]), \forall i = 1,\ldots,n, j = 0,\ldots,C$$

Efficient matrix multiplication

Problem: Given a chain of 2-dimensional matrices $M_1, M_2, \ldots, M_n$ (where M_i has dimensions $d_{i-1} \times d_i$ for all $1 \leq i \leq n$) to be multiplied together, determine the order in which the muliplications must be made in order to do the least amount of work.

As an example of our problem consider multiplying together the three matrices M_1, M_2 and M_3 where M_1 is 3×4, M_2 is 4×2 and M_3 is 2×8.

$((M_1 M_2)M_3)$ which involves $3 \times 4 \times 2 + 3 \times 2 \times 8 = 24 + 48 = 72$ multiplications and

$(M_1(M_2 M_3))$ which involves

$4 \times 2 \times 8 + 3 \times 4 \times 8 = 64 + 96 = 160$ multiplications.

Again a recursive solution seems like the right approach.

Let us begin by defining $m(i,j)$ to be the minimum number of multiplications to compute $M_i \times \ldots \times M_j, 1 \leq i < j \leq n$.
$m(1,n)$ would be the minimum number of multiplications to calculate $M_1 \times M_2 \times \ldots \times M_n$.
To calculate $m(i,j)$ for any i and j we would have to consider all of the possible parenthesizations of this subproblem.

$$m(i,j) = \underset{i \leq k < j}{minimum}(m(i,k) + m(k+1,j) + d_{i-1} d_k d_j), 1 \leq i < j \leq n$$

Note that $m(h,h) = 0$ for $1 \leq h \leq n$.
Again some subproblems are solved repeatedly,
Use dynamic programming and build up the solution from the "bottom".

Create a “table” of solutions to subproblems.
Actually an upper triangular matrix which we will fill from the main diagonal where all the entries are 0 and working towards the top right corner which is where our final solution will be found.

For example, consider $M_1 M_2 M_3 M_4 M_5$.
All that is required are the dimensions of the matrices, so let $d_0 = 3$, $d_1 = 2$, $d_2 = 3$, $d_3 = 4$, $d_4 = 5$, $d_5 = 2$

Then to begin:

	1	2	3	4	5
1	0				
2	-	0			
3	-	-	0		
4	-	-	-	0	
5	-	-	-	-	0

First:
$m(1,2) = d_0 d_1 d_2$,
$m(2,3) = d_1 d_2 d_3$
etc.
Then:

$$m(1,3) = minimum(m(1,1)+m(2,3)+d_0 d_1 d_3, m(1,2)+m(3,3)+d_0 d_2 d_3)$$

$$m(1,3) = minimum(0+24+24, 18+0+36) = minimum(48,54)$$

etc.

	1	2	3	4	5
1	0	18	48	94	88
2	-	0	24	64	76
3	-	-	0	60	64
4	-	-	-	0	40
5	-	-	-	-	0

$m(1, n)$ then contains the minimum number of multiplications to multiply the matrices together.

Efficiency

Calculates values for around half the positions in the matrix m or about $n^2/2$ matrix entries.

The work for each entry involves testing all possible positions of k between i and j. There are at most n of these positions.

For each position of k a small number of constant time operations are done.

Finding the best split for each position is thus $O(n)$. Means $O(n^3)$ overall.

Requires $O(n^2)$ space.

Correctness

Similar to other DP algorithms

Note

Algorithm above does not actually return the optimal factorisation (it only gives the minimum number of multiplications in the optimal factorisation),

Some general comments on dynamic programming

Generally trading space for speed of computation

The general approach to developing dynamic progamming solutions

1. start by developing a top-down approach;
2. determine the relationship of optimal solutions to subproblems and the optimal solution to the overall problem;
3. come up with a "formula" relating these solutions to each other;
4. determine which subproblem solutions might be recomputed;
5. determine a good "table" to store these subproblem solutions;
6. determine how to initialise and subsequently fill the table;
7. and finally determine how to find the solution to the overall problem in the table.