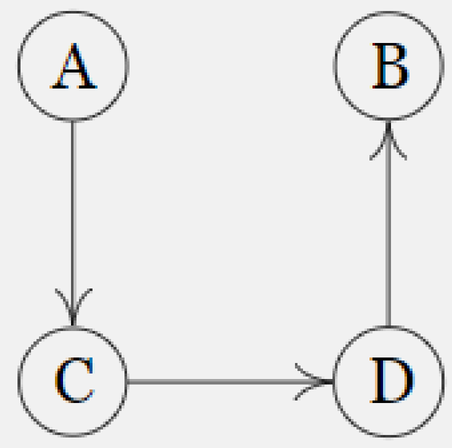
Tutorial

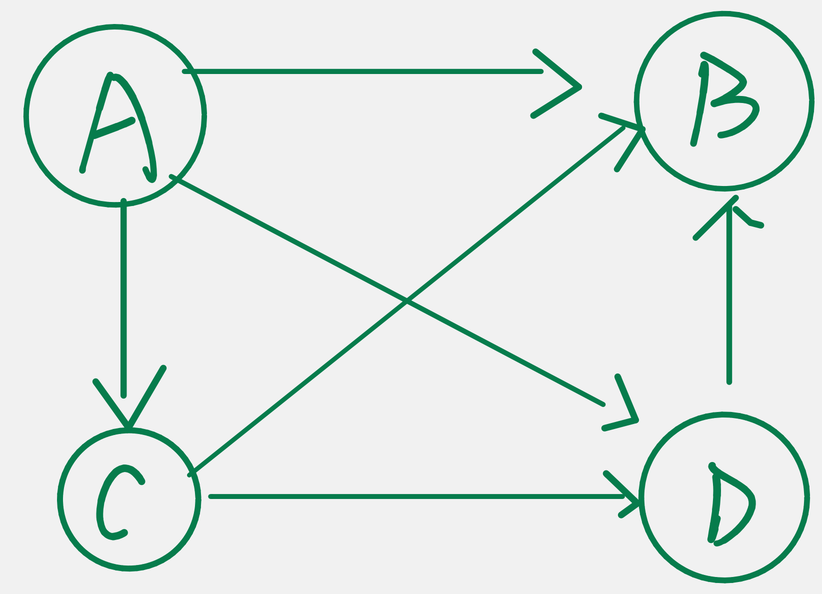
1. Transitive Closure

The *transitive closure* of a directed graph G is a new graph H which contains an edge between vertices u and v if and only if there is a path from u to v using one or more edges in the original graph G.

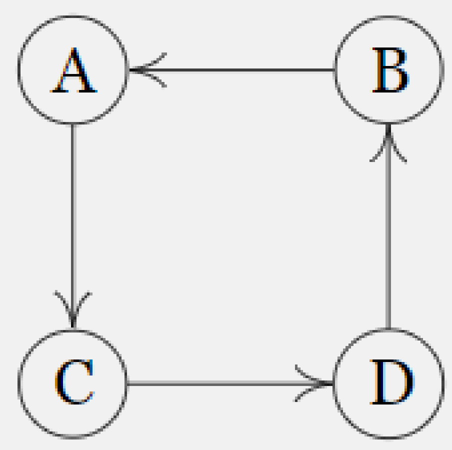
Draw the transitive closure of the following two graphs:

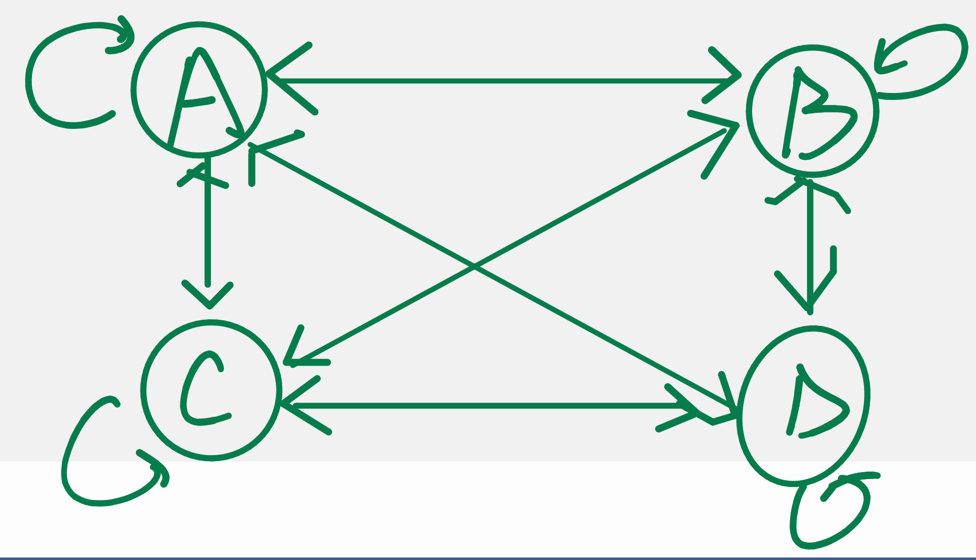
a.





b.





2. Warshall's Algorithm

Warshall's algorithm is a *dynamic programming* algorithm which computes the transitive closure of a graph defined by an adjacency matrix.

First, we'll assume that the n nodes are labelled $1, 2, ..., n$.

We take the problem of *is there a path in G from i to j* (for each pair of nodes $\{i,j\} \subseteq \{1, ..., n\}$) and break divide into the sub-problems (for each $k \in \{1, ..., n\}$):

Is there a path from i to j in G using only nodes in $\{1, ..., k\}$ as intermediate nodes?

Once we have computed this for $k = n$ we have the answer to the original question for each $\{i,j\}$ and can hence compute the transitive closure.

For each of these sub-problems we will let $R_{ij}{}^k$ be defined like so:

$$R_{ij}^k := \begin{cases} 1 & \text{if there is a path from } i \text{ to } j \text{ in } G \text{ using only nodes in } \{1, \ldots, k\} \text{ as intermediate nodes} \\ 0 & \text{otherwise} \end{cases}$$

If we have A, the graph G's adjacency matrix we have the following base case and recursive definition of R^k:

$$R_{ij}{}^0 := A_{ij}, \qquad R_{ij}{}^k := R_{ij}{}^{k-1} \text{ or } (R_{ik}{}^{k-1} \text{ and } R_{kj}{}^{k-1})$$

The adjacency matrix of the transitive closure, B, is then defined by $B_{ij} := R_{ij}{}^n$.

To run Warshall's algorithm, we should compute each of the matrices $R^0, R^1, ..., R^n$.

Use Warshall's algorithm to compute the transitive closure of a graph G defined by the following adjacency matrix,

$$A = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

3. Floyd's Algorithm

So we've seen Warshall's algorithm which computes whether or not there is a path between a pair of vertices $\{i,j\}$ by computing the adjacency matrix of the transitive closure, given the original graph's adjacency matrix.

Floyd's algorithm builds on Warshall's algorithm to solve the *all pairs shortest path* problem. That is, Floyd's algorithm computes the length of the shortest path from each pair of vertices in a graph.

Rather than an adjacency matrix, we will require a weights matrix W, where W_{ij} indicates the weight of the edge from i to j (if there is no edge from i to j then then $W_{ij} = \infty$). We will ultimately find a distance matrix D in which D_{ij} indicates the cost of the shortest path from i to j.

The sub-problems in this case will be answering the following question:

What's the shortest path from i to j using only nodes in $\{1, \dots, k\}$ as intermediate nodes?

To perform the algorithm we find D^k for each $k \in \{0, \dots, n\}$ and set $D := D^n$. The update rule becomes the following:

$$D_{ij}{}^0 := W_{ij}, \qquad D_{ij}{}^k := \min\{D_{ij}{}^{k-1}, D_{ik}{}^{k-1} + D_{kj}{}^{k-1}\}$$

Perform Floyd's algorithm on the graph given by the following weights matrix:

$$W = \begin{bmatrix} 0 & 3 & \infty & 4 \\ \infty & 0 & 5 & \infty \\ 2 & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{bmatrix}.$$

4. Baked Beans Bundles

We have bought n cans of baked beans wholesale and are planning to sell bundles of cans at the University of Melbourne's farmers' market.

Our business-savvy friends have done some market research and found out how much students are willing to pay for a bundle of k cans of baked beans, for each $k \in \{1, \dots, n\}$.

We are tasked with writing a *dynamic programming algorithm* to determine how we should split up our n cans into bundles to maximise the total price we will receive.

a. Write the pseudocode for such an algorithm.

b. Using your algorithm determine how to best split up 8 cans of baked beans, if the prices you can sell each bundle for are as follows:

Bundle Size k	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Price	1	5	8	9	10	17	17	20

c. What's the runtime of your algorithm? What are the space requirements?

Question 2

$$R^0 = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$R^1 = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$R^2 = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$R^3 = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$B := R^4 = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Question 3

$$D^0 = \begin{bmatrix} 0 & 3 & \infty & 4 \\ \infty & 0 & 5 & \infty \\ 2 & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{bmatrix}$$

$$D^1 = \begin{bmatrix} 0 & 3 & \infty & 4 \\ \infty & 0 & 5 & \infty \\ 2 & 5 & 0 & 6 \\ \infty & \infty & 1 & 0 \end{bmatrix}$$

$$D^2 = \begin{bmatrix} 0 & 3 & 8 & 4 \\ \infty & 0 & 5 & \infty \\ 2 & 5 & 0 & 6 \\ \infty & \infty & 1 & 0 \end{bmatrix}$$

$$D^3 = \begin{bmatrix} 0 & 3 & 8 & 4 \\ 7 & 0 & 5 & 11 \\ 2 & 5 & 0 & 6 \\ 3 & 6 & 1 & 0 \end{bmatrix}$$

$$D^4 = \begin{bmatrix} 0 & 3 & 5 & 4 \\ 7 & 0 & 5 & 11 \\ 2 & 5 & 0 & 6 \\ 3 & 6 & 1 & 0 \end{bmatrix}$$

Question 4

(a)

Why not use the following rules

$$dp[m] = max(dp[m], dp[m-n] + val[n])(n \in [1, m])$$

we can have the following codes:

```
import math
import numpy as np



def DynamicProgrammingAlgorithm(n, prices):
    # Create an array to store the maximum total price for each subproblem
    dp = np.zeros(n + 1)

    for i in range(n + 1):
        max_price = 0
        for k in range(i + 1):
            max_price = max(max_price, prices[k] + dp[i - k])
        dp[i] = max_price

    return dp[n]
```

(b)

Runing the following codes, we can have

```
n = 8
prices = [0, 1, 5, 8, 9, 10, 17, 17, 20]
result = DynamicProgrammingAlgorithm(n, prices)
print("Maximum total price:", result)
```

the maximum total price is 22,which is $2, 6$

(c)

the runtime of the algorithm is $O(n^2)$ and space requirements are $O(n)$