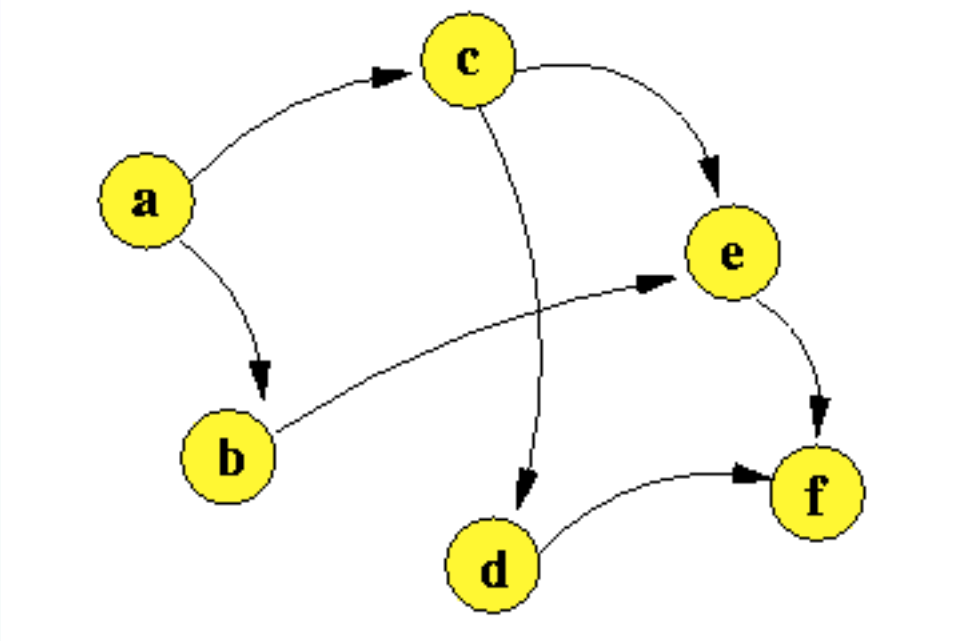
COMP20007 Workshop Week 12

1	**Transitive Closure, problem T1**
2	**Dynamic Programming: Warshall's, Floyd's**
3	**DP: Problem T5 Baked Bean Bundles**
4	**Manual execution of Warshall & Floyd: T3 & T4**
5	**T5: Revision: quicksort & top-down mergesort**
	LAB:
	A2
	baked bean bundles (implementation)

Transitive Closure of digraphs

Transitive Closure



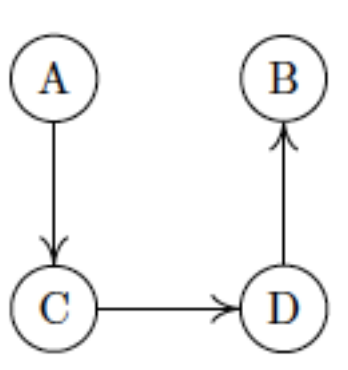
Related Tasks:

- Compute the transitive closure for a digraph
- Find APSP for a weighted graph

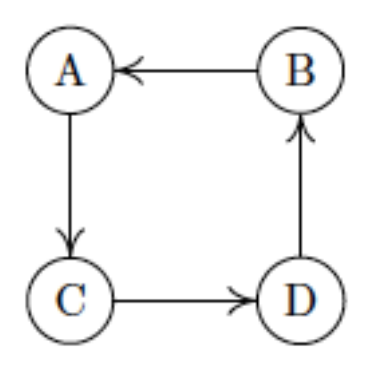
T1: Transitive Closure of digraphs

Draw the transitive closure of the following two graphs:

(a)



(b)



Dynamic Programming – an important topic

Dynamic Programming is just a fancy way to say **'remembering stuffs I've done to save time later'**

Example: fib(n)

Dynamic Programming

To solve a complex problem:

- breaks it down into a collection of simpler sub-problems,
- solves each of those sub-problems just once, and storing their solutions in some data structure (array, hash table etc) for quick lookup (based on input parameters) later
- each time, when a solved subproblem occurs, do lookup and reuse the solution instead of recomputing.

Normally, in DP we solve the small-size sub-problems first, and then progressively solve the bigger size. The most important steps are to work out:

- the problem parameters, and
- the relationship between the solution for larger parameter and that for smaller parameter.

DP: Examples

- Fibonacci: find fib(n)
- Find Transitive Closure of a digraph
- APSP: all pair shortest paths for a weighted graph
- Baked Bean Bundles (problem T4)

Warshall's Algorithm: DP for Transitive Closure

- Input: adjacent matrix A
- Main argument: transitiveness: if there are paths i→k and k→j, then there is path i→j which uses k as an interim stepstone.
- To do DP, we need to decide:
  - what are parameters
  - what is the relationship between solutions for a bigger and a smaller parameter
  - what are the base cases (when solution is ready)

Warshall's Algorithm: DP for Transitive Closure

- adjacent matrix A is T^0
- we can go from T^0 to T^1 by $T^1_{ij} = T^0_{ij} \;||\; (T^0_{i1} \;\&\&\; T^0_{1j})$
- we can go from T^{k-1} to T^k by $T^k_{ij} = T^{k-1}_{ij} \;||\; (T^{k-1}_{ik} \;\&\&\; T^{k-1}_{kj})$
- how would we store T^k_{ij} for easy lookup? Do we really need an array of 2D arrays?
- *Write the algorithm, by first set T= A, and then progressing k from 1 to n.*
- *And so, remember that this DP is characterised by*

$$T0 = A$$

$$T^k_{ij} = T^{k-1}_{ij} \;||\; (T^{k-1}_{ik} \;\&\&\; T^{k-1}_{kj}) \text{ for } k = 1..n$$

DP for APSP: Floyd's Algorithm

- Task: APSP – For a weighted graph G, find shortest path between all pair of vertices.
- Main idea: if we have shortest paths for i→k and for k→j then we can have shortest path for i→j just by joining the formers.
- So, the idea is just similar to the Warshall's. Can we build the DP relationship?

DP: Floyd's Algorithm (APSP)

Floyd's algorithm builds on Warshall's algorithm to solve the all pairs shortest path problem: computing the length of the shortest path between each pair of vertices in a graph.

Rather than an adjacency matrix, we will require a weights matrix W , where Wij indicates the weight of the edge from i to j (if there is no edge from i to j then then Wij = ∞). We will ultimately find a distance matrix D in which Dij 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, . . . , k} as intermediate nodes?

To perform the algorithm we find Dk for each k ∈ {0,...,n} and set D := Dn. The update rule becomes the following:

$$D^0_{ij} := W_{ij}, \qquad D^k_{ij} := \min\left\{D^{k-1}_{ij}, D^{k-1}_{ik} + D^{k-1}_{kj}\right\}$$

T5: 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-savy 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,...,n}.

We are tasked with writing a dynamic programming algorithm to determine how we should split up out 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?

T5: Baked Beans Bundles

(a) Design a DP algorithm, Write the pseudocode for that algorithm.

We have bought n cans of baked beans, n=8.

We have the following things as the start:

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

What is the parameter of the task? What's base case? What's the DP relationship?

T5: Baked Beans Bundles

(b) Run the algorithm manually

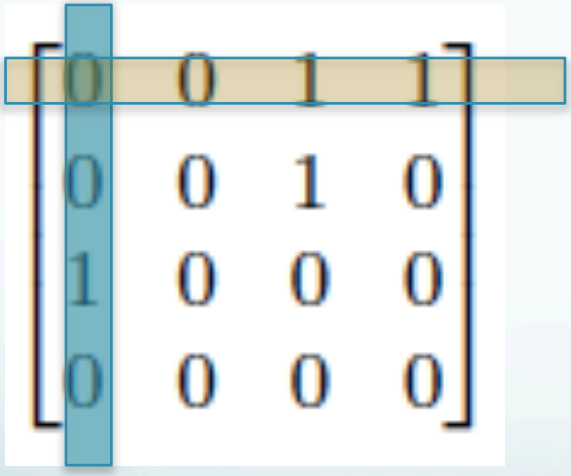
(c) Complexity = ?

k	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
price $	0	1	5	8	9	10	17	17	20
revenue R									
bundle B									

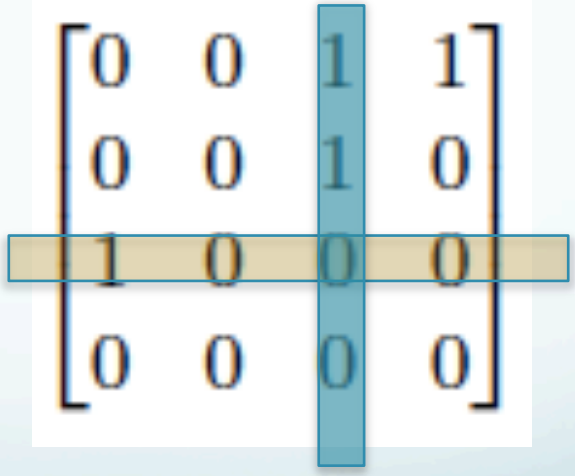
T2: Running Warshall's Algorithm

Set T= A, T is T0
transition from 0 to 1 by:
- looking at for all possible ij
- it can be done by using colum 1 and row 1 as references

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



similarly for any transition from k-1 to k



T3: manual exec of Floyd's

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

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

$$D^0_{ij} := W_{ij}, \qquad D^k_{ij} := \min\left\{D^{k-1}_{ij}, D^{k-1}_{ik} + D^{k-1}_{kj}\right\}$$

Example: $D_{k-1} \rightarrow D_k$ when k=1

$$D^0_{ij} := W_{ij}, \qquad D^k_{ij} := \min\left\{D^{k-1}_{ij}, D^{k-1}_{ik} + D^{k-1}_{kj}\right\}$$

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

row **k**:
$A_{kj} < \infty$ if there is a path **k**→j

column **k**:
$A_{ik} < \infty$ if there is a path i→**k**

T5: Revision for quicksort & mergesort

(a) Perform a single Hoare Partition on the following array, taking the first element as the pivot.
[3, 8, 5, 2, 1, 3, 5, 4, 8]

(b) Perform Quicksort on the array from (a). You may use whatever partitioning strategy you like

(c) Perform Mergesort on the array from (a).