COMP20003 Workshop Week 11

1	Floyd-Warshall Algorithm
2	Graph Search
3	A* Search
4	IDA* Search
5	Assignment 2

Floyd-Warshall Algorithm

Purpose= ?

Similarity to Dijkstra = ?

Complexity = ?

Floyd-Warshall Algorithm - APSP

Given a weighted DAG G=(V,E,w(E))

Find shortest path (path with min weight) between all pairs of vertices.

Idea= ?

Floyd-Warshall Algorithm

Find shortest path between all pairs (s,t) of vertices. That means minimizing `dist(s,t)`.

At the start `dist` is the adjacent matrix:

`if s==t: dist(s,s) = 0`, otherwise

`dist(s,t) = w(s,t)` OR `dist(s,t) = ∞`

Main algorithm:

```
for each vertex i do
   for each pair (s,t) do
      if  s->i->t is better than s->t
         update dist(s,t)
```

Conditions=?

Data structures= ?

Floyd-Warshall Algorithm

Main algorithm:

```
for each vertex i do
   for each pair (s,t) do
      if dist(s,i)+dist(i,t) < dist(s,t):
         update dist(s,t)
```

Conditions= ?

Data structures / Graph representation = ?

Complexity =

Floyd-Warshall Algorithm: write C code

Supposing: A[][] is the adjacent matrix, V is number of vertices. Write C code for the algorithm:

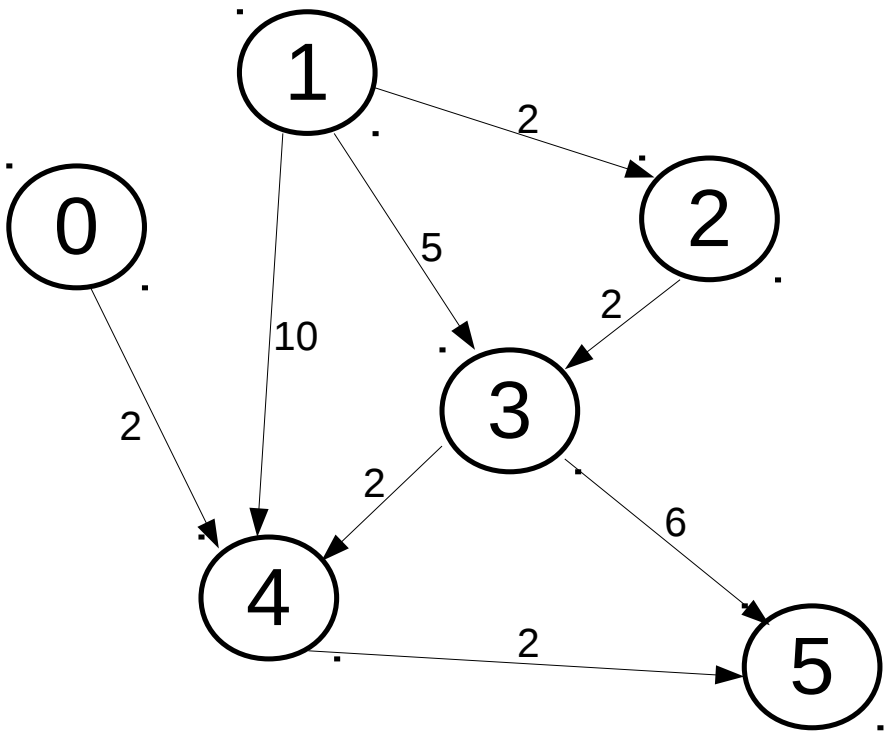
How to retrieve path from i→j ?

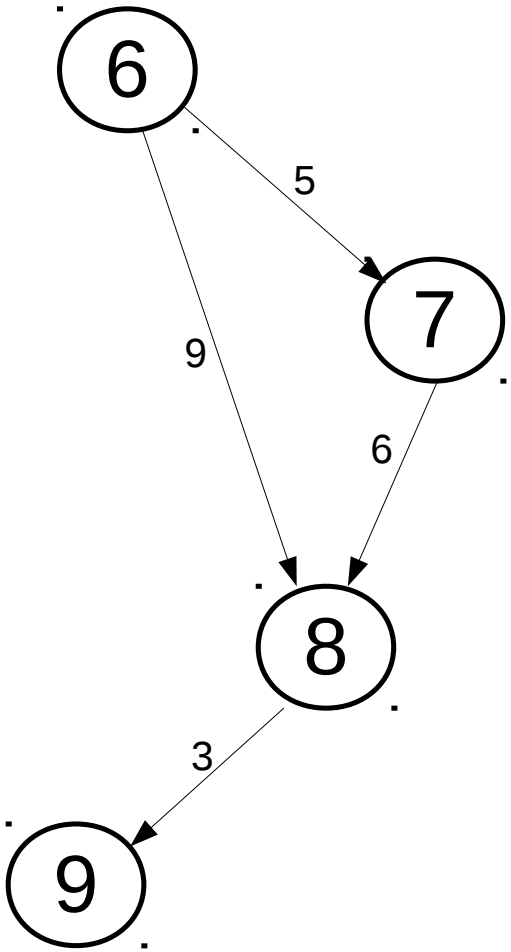
Floyd-Warshall Algorithm

Floyd-Warshall algorithm (weights, **A[i][i]**= 0, no path=∞)

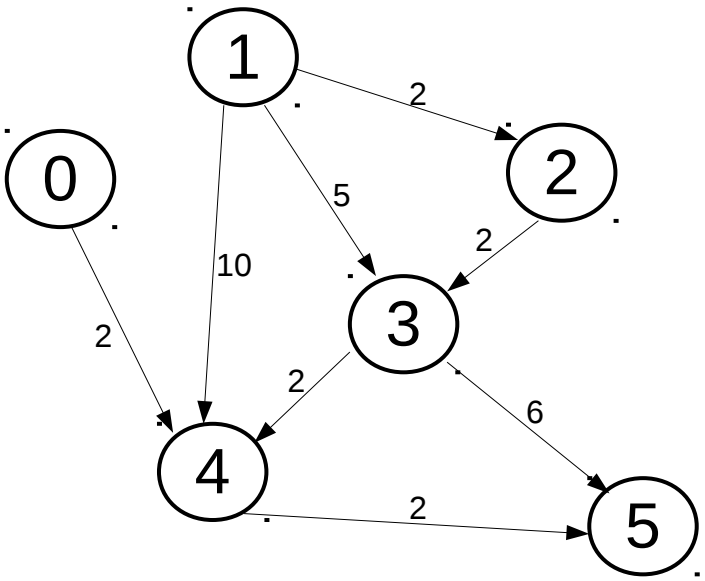
```
for(i=0;i<V;i++)
  for(s=0;s<V;s++)
    for(t=0;t<V;t++)
       if(A[s][i]+A[i][t] < A[s][t])
           A[s][t] = (A[s][i]+A[i][t]);
```

Q11.1





Q 11.1 [simplified]



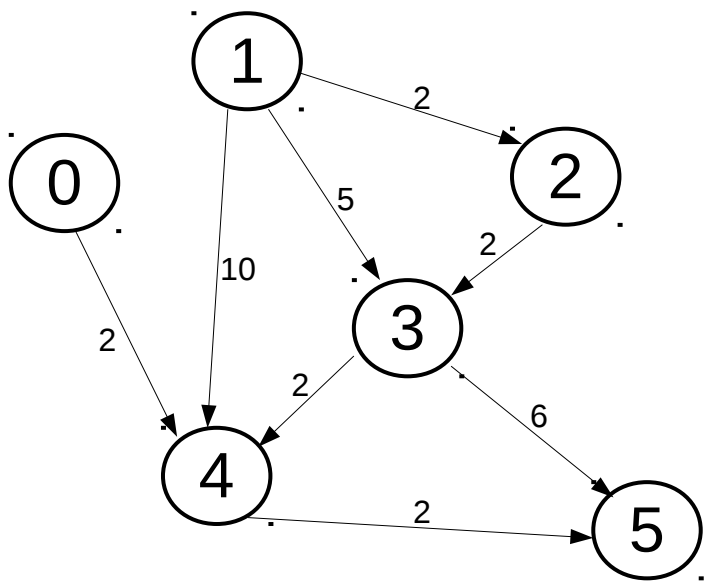
Draw the matrix representation. Trace the Floyd-Warshall algorithm.

TO

FROM

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

Q 11.1 [simplified]



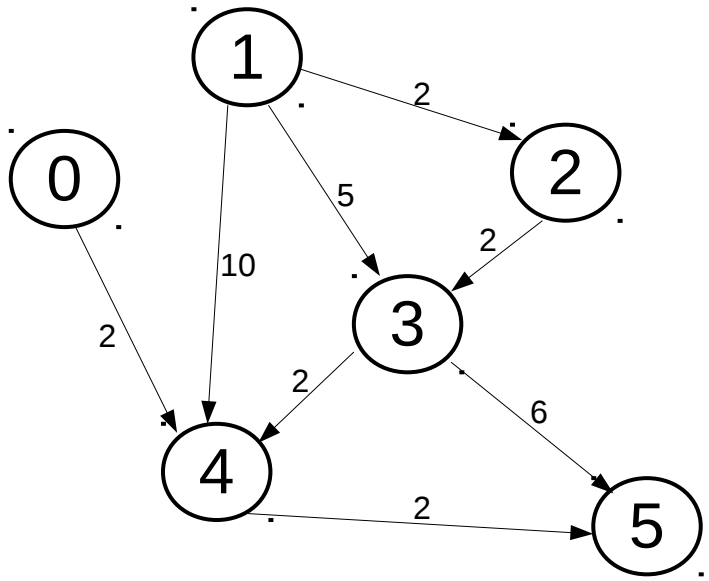
Trace the Floyd-Warshall algorithm.
Step i= 0, 1, 2, 3, 4, 5
TO

FROM

	0	1	2	3	4	5
0					2	
1			2	5	10	
2				2		
3					2	6
4						2
5						

empty cell for ∞
(note A[i][i] could be zero if we want)

Q 11.1 [simplified]



Trace the Floyd-Warshall algorithm (empty means ∞).
Step i= 0, 1, 2, 3, 4, 5

TO

FROM

	0	1	2	3	4	5
0	0				2	
1		0	2	5	10	
2			0	2		
3				0	2	6
4					0	2
5						0

Q11.1

1. Does the Floyd-Warshall algorithm work on graphs where there are negative weights? justify your answer.

Q11.1

2. Given a sparse graph represented as an adjacency list, how would you approach the all pairs shortest paths problem? Does this differ from the approach you would take for a dense graph represented as a matrix? Compare the computational complexity of the two approaches.

Q11.1: APSP – comparing Dijkstra & Floyd-Warshall

Big-O complexity
(supposing to apply Dijkstra for the APSP task)

	Dijkstra	Floyd-Warshall
General		
Sparse		
Dense		

Q11.1: APSP – comparing Dijkstra & Floyd-Warshall

Big-O complexity
(supposing to apply Dijkstra for the APSP task)

	Dijkstra	Floyd-Warshall
Sparse General	$V\ (V+E)\ \log V$	V^3
Dense	$V^2\ \log V$	V^3
	$V^3\ \log V$	V^3

5 min break: qoct

Do quality of casual teaching for **both** 1) your tutor, and 2) your demonstrator.

To navigate to the website, just google **qoct**

Graph Search

The Task: Path Finding

finding a path P from node S to node G, so that P satisfy some condition:

P is $S = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_k = G$

condition: for example minimize/maximize the sum of weight, minimize the max edge weight.

Example: weighted graph with vertices are cities, an edge represent the road between the 2 cities, and weight is just road length. We want to find the shortest part from MEL to SYD.

Algorithm? We can change Dijkstra algorithm to serve this purpose.

The Task: Path Finding

Example: find the shortest part from MEL to SYD.

We can use Dijkstra Algorithm:

- DA find shortest path from one sourse (MEL) to any other node, but
- We can stop the algorithm right after the shortest path to SYD has been found.

Is that algorithm practically efficient? Before finding path to SYD we find any other path that is shorter than to SYD, including, say, path to Geelong, Ballarat, or Adelaide!

The Task: Path Finding

Algorithm like Dijkstra is called blind search – it just relies on the information on the graph.

If at each city we have some additional information, estimating how close the city is to SYD (for example, straight-line distance between that city and SYD), we can employ that information to improve the search efficiency by pruning the impossible path.

- Such information is a heuristic.
- Search algorithm that employs a heuristic is called heuristic search.

A* Search

A* Search employs a heuristics to estimate the cost of paths. The main idea:

- Avoid expanding the nodes that are already expensive
- Focus on path that show promise

A* search

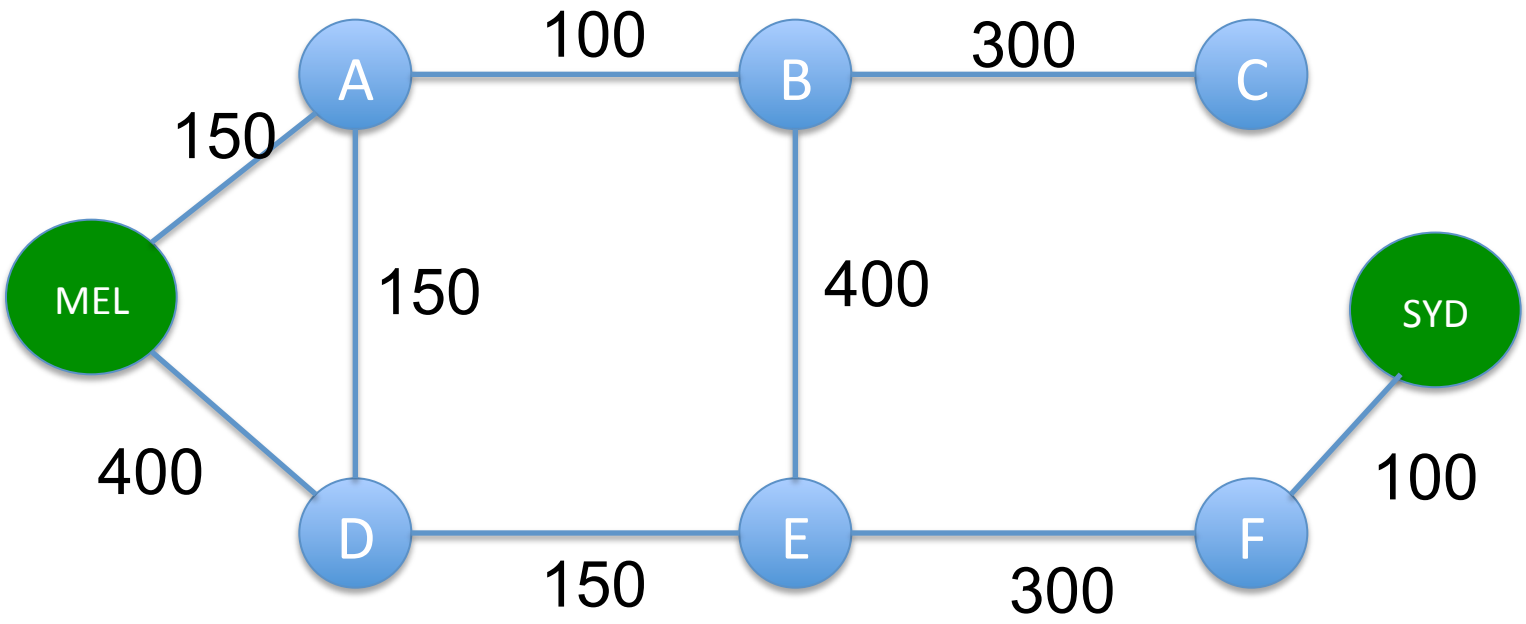
Finding shortest path, which `h(s)` = straight-line distance to SYD (goal).

Terminologies: as in ass2. In particular:

- a vertex like MEL is called a state
- a node `n` of the search includes:
  - a state `n.s`
  - cost so far to reach `n` (ie reaching state `n.s`) : `n.g`
  - heuristic value at `n`: `n.h`
  - `n.f` : the estimated cost for the current path:

```
n.f = n.g + n.h
```

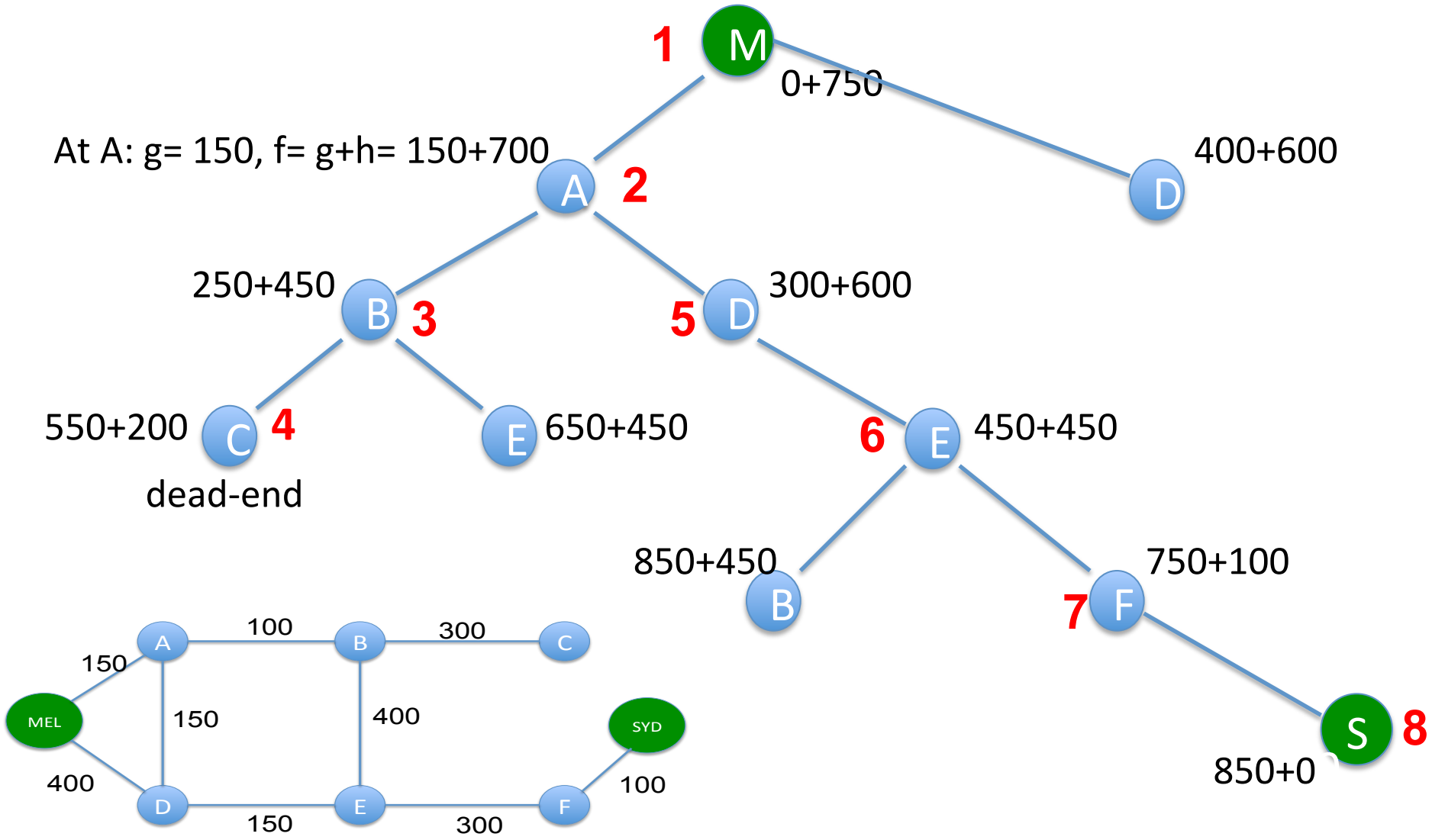
A* search (A-D are imaginable cities ☺)



MEL	A	B	C	D	E	F	SYD
750	700	450	200	600	450	100	0

Straight-line distances

A* Example



MEL	A	B	C	D	E	F	SYD
750	700	450	200	600	450	100	0

Red number: node expanded, in order

IDA* Search

Is an advanced version of A*.

1. First, use $h(s_0)$ as an estimated threshold B. B is actually the best possible result any algorithm can get.

   Then:

2. Find path from s_0 to the end:
   - During this process, keep track of min estimated cost B'
   - If found a path with cost <= B, it is a result.
   - If there is no path with estimated cost <=B, terminates
3. If path not found, update B with B' and repeat step 2.

Note 1: In your ass2 spec, Step 2 is the `IDA()` function.

Note 2: In `IDA()`, B is used to decide if we should avoid expanding a node

Ass2: Q&A