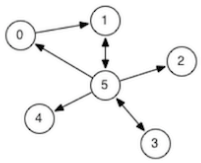
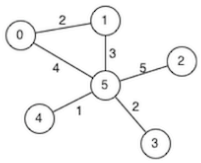
Week 5 Problem Set

Minimum Spanning Trees, Shortest Paths, Maximum Flows

1. (Digraphs)

   a. Consider the following graphs, where bi-directional edges are depicted as two-way arrows rather than having two separate edges going in opposite directions:

   For each of the graphs show the concrete data structures if the graph was implemented via:

   1. adjacency matrix representation (assume full V×V matrix)
   2. adjacency list representation (if non-directional, include both (v,w) and (w,v))

   b. Consider the following map of streets in the Sydney CBD:

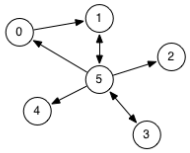
   Represent this as a directed graph, where intersections are vertices and the connecting streets are edges. Ensure that the directions on the edges correctly reflect any one-way streets (this is a driving map, not a walking map). You only need to make a graph which includes the intersections marked with red letters. Some things that don't show on the map: Castlereagh St is one-way heading south, Curtin Pl is a little laneway that you can't drive down and, thanks to the shiny new light rail, George St is now closed for cars between "F" and "K".

   For each of the following pairs of intersections, indicate whether there is a path from the first to the second. Show a simple path if there is one. If there is more than one simple path, show two different paths.

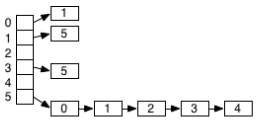
   1. from intersection "D" on Margaret St to insersection "L" on Pitt St
   2. from intersection "J" to the corner of Margaret St and York St (intersection "A")
   3. from intersection "P" on Castlereagh St to the corner of Margaret St and Wynyard Ln ("C")
   4. from the intersection of Castlereagh St and Hunter St ("M") to intersection "H" at Wynyard Park

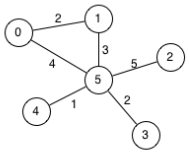
Answer:

a.

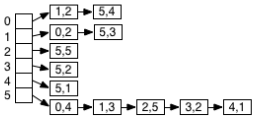


	0	1	2	3	4	5
0	0	1	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	0	0
3	0	0	0	0	0	1
4	0	0	0	0	0	0
5	1	1	1	1	1	0

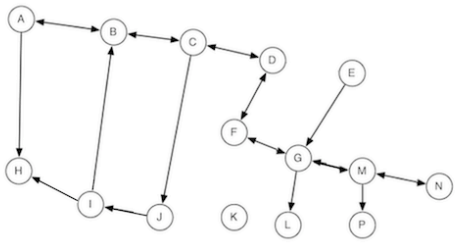




	0	1	2	3	4	5
0	0	2	0	0	0	4
1	2	0	0	0	0	3
2	0	0	0	0	0	5
3	0	0	0	0	0	2
4	0	0	0	0	0	1
5	4	3	5	2	1	0



b. The graph is as follows.



   For the paths:

   1. D → F → G → L and there are no other choices that don't involve loops through F or G.
   2. J → I → B → A.
   3. You can't reach C from P on this graph. Real-life is different, of course.
   4. M → G → F → D → C → B → A → H or M → G → F → D → C → J → I → H.

2. (Warshall's algorithm)

   Apply Warshall's algorithm to compute the transitive closure of your graph from exercise 1b. Choose vertices in alphabetical order. Show the reachability matrix:

   - after the initialisation
   - after the 1st iteration (vertex 'A') of the outermost loop
   - after the 6th iteration (vertex 'F') of the outermost loop
   - at the end.

   Interpret the values in each of these matrices: Which connections between vertices do the different matrices encode?

Answer:

The initial matrix encodes all directed paths of length 1:

tc	[A]	[B]	[C]	[D]	[E]	[F]	[G]	[H]	[I]	[J]	[K]	[L]	[M]	[N]	[P]
[A]	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0
[B]	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
[C]	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0
[D]	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0
[E]	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
[F]	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0
[G]	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0
[H]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[I]	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0
[J]	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
[K]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[L]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[M]	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1
[N]	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
[P]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

After the first iteration:

tc	[A]	[B]	[C]	[D]	[E]	[F]	[G]	[H]	[I]	[J]	[K]	[L]	[M]	[N]	[P]
[A]	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0
[B]	1	1	1	0	0	0	0	1	0	0	0	0	0	0	0
[C]	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0
[D]	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0
[E]	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
[F]	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0
[G]	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0
[H]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[I]	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0
[J]	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
[K]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[L]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[M]	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1
[N]	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
[P]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

This matrix represents the existence of all directed paths:

- of length 1,
- of length >1 that go through vertex A as the only non-endpoint (B→A→B and B→A→H).

After the sixth iteration:

tc	[A]	[B]	[C]	[D]	[E]	[F]	[G]	[H]	[I]	[J]	[K]	[L]	[M]	[N]	[P]
[A]	1	1	1	1	0	1	1	1	0	1	0	0	0	0	0
[B]	1	1	1	1	0	1	1	1	0	1	0	0	0	0	0
[C]	1	1	1	1	0	1	1	1	0	1	0	0	0	0	0
[D]	1	1	1	1	0	1	1	1	0	1	0	0	0	0	0
[E]	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
[F]	1	1	1	1	0	1	1	1	0	1	0	0	0	0	0
[G]	1	1	1	1	0	1	1	1	0	1	0	1	1	0	0
[H]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[I]	1	1	1	1	0	1	1	1	0	1	0	0	0	0	0
[J]	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
[K]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[L]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[M]	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1
[N]	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
[P]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

This matrix represents the existence of all directed paths

- of length 1,
- of length >1 that go through one or more of the vertices A,B,C,D,E,F as non-endpoints.

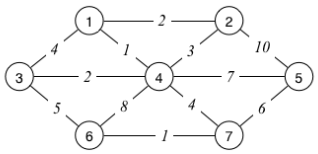
After the last iteration:

tc	[A]	[B]	[C]	[D]	[E]	[F]	[G]	[H]	[I]	[J]	[K]	[L]	[M]	[N]	[P]
[A]	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1
[B]	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1
[C]	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1
[D]	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1
[E]	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1
[F]	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1
[G]	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1
[H]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[I]	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1
[J]	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1
[K]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[L]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[M]	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1
[N]	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1
[P]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The final matrix represents the existence of all directed paths through any of the vertices as non-endpoints, i.e. the transitive closure.

3. (Minimum spanning trees)

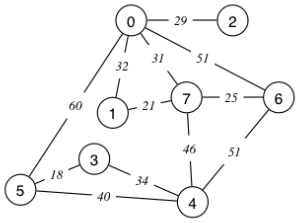
a. Show how Kruskal's algorithm would construct the MST for the following graph:



How many edges do you have to consider?

b. For a graph *G=(V,E)*, what is the least number of edges that might need to be considered by Kruskal's algorithm, and what is the most number of edges? Add one vertex and edge to the above graph to force Kruskal's algorithm to the worst case.

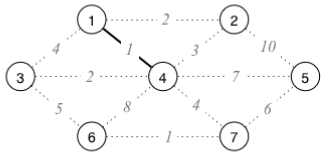
c. Trace the execution of Prim's algorithm to compute a minimum spanning tree on the following graph:



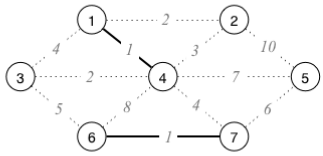
Choose a random vertex to start with. Draw the resulting minimum spanning tree.

Answer:

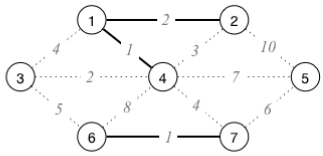
a. In the first iteration of Kruskal's algorithm, we could choose either 1-4 or 6-7, since both edges have weight 1. Assume we choose 1-4. Since its inclusion produces no cycles, we add it to the MST (non-existent edges are indicated by dotted lines):



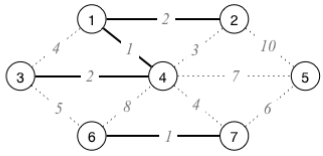
In the next iteration, we choose 6-7. Its inclusion produces no cycles, so we add it to the MST:



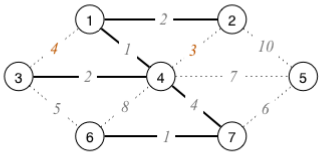
In the next iteration, we could choose either 1-2 or 3-4, since both edges have weight 2. Assume we choose 1-2. Since its inclusion produces no cycles, we add it to the MST:



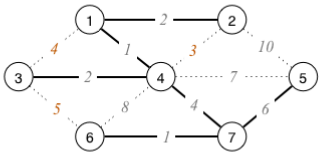
In the next iteration, we choose 3-4. Its inclusion produces no cycles, so we add it to the MST:



In the next iteration, we would first consider the lowest-cost unused edge. This is 2-4, but its inclusion would produce a cycle, so we ignore it. We then consider 1-3 and 4-7 which both have weight 4. If we choose 1-3, that produces a cycle so we ignore that edge. If we add 4-7 to the MST, there is no cycle and so we include it:



Now the lowest-cost unused edge is 3-6, but its inclusion would produce a cycle, so we ignore it. We then consider 5-7. If we add 5-7 to the MST, there is no cycle and so we include it:



At this stage, all vertices are connected and we have a MST.

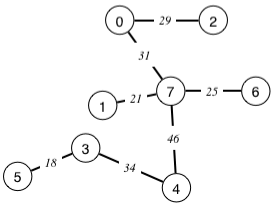
For this graph, we considered 9 of the 12 possible edges in determining the MST.

b. For a graph with *V* vertices and *E* edges, the best case would be when the first *V-1* edges we consider are the lowest cost edges and none of these edges leads to a cycle. The worst case would be when we had to consider all *E* edges. If we added a vertex 8 to the above graph, and connected it to vertex 5 with edge cost 11 (or any cost larger than all the other edge costs in the graph), we would need to consider all edges to construct the MST.

c. If we start at node 5, for example, edges would be found in the following order:

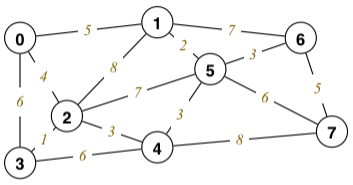
- 5-3
- 3-4
- 4-7
- 1-7
- 6-7
- 0-7
- 0-2

The minimum spanning tree:



4. (Shortest paths)

Consider the following graph:



a. What is the shortest path from vertex 0 to vertex 3? From vertex 0 to vertex 7? How did you determine these?

b. Trace the execution of Dijkstra's algorithm on the graph to compute the minimum distances from source node 0 to all other vertices.

Show the values of `vSet`, `dist[]` and `pred[]` after each iteration.

c. In the given graph, what is the shortest path from vertex 3 to vertex 6? From vertex 6 to vertex 3? How did you determine these?

d. Trace the execution of Floyd's algorithm on the graph to compute the shortests paths between *all* pairs of vertices.

Answer:

a. Shortest path from vertex 0 to vertex 3 has total weight 5. There are really only two plausible choices: either the direct edge from vertex 0 to vertex 3, or two edges via vertex 2. It is easy to see that the total weight on the two-edge path (5) is less than the weight on the single edge (6).

Shortest path from vertex 0 to vertex 7 has total weight 13. Probably (although not necessarily) the best paths are also the most direct ones, which suggests 0-1-6-7, 0-1-5-7 and 0-2-4-7. If we sum the lengths on each of these paths, they are, respectively: 17, 13, 15. So we choose the shortest: 0-1-5-7.

b. Initialisation:

```
vSet = { 0, 1, 2, 3, 4, 5, 6, 7 }
dist = [ 0, ∞, ∞, ∞, ∞, ∞, ∞, ∞ ]
pred = [ -1, -1, -1, -1, -1, -1, -1, -1 ]
```

The vertex in vSet with minimum dist[] is 0. Relaxation along the edges (0,1,5), (0,2,4) and (0,3,6) results in:

```
vSet = { 1, 2, 3, 4, 5, 6, 7 }
dist = [ 0, 5, 4, 6, ∞, ∞, ∞ ]
pred = [ -1, 0, 0, 0, -1, -1, -1 ]
```

Now the vertex in vSet with minimum dist[] is 2. Considering all edges from 2 to nodes still in vSet:

- relaxation along (2,1,8) does not give us a shorter distance to node 1
- relaxation along (2,3,1) yields a smaller value (4+1 = 5) for dist[3], and pred[3] is updated to 2
- relaxation along (2,4,3) yields a smaller value (4+3 = 7) for dist[4], and pred[4] is updated to 2
- relaxation along (2,5,7) yields a smaller value (4+7 = 11) for dist[5], and pred[5] is updated to 2

```
vSet = { 1, 3, 4, 5, 6, 7 }
dist = [ 0, 5, 4, 5, 7, 11, ∞, ∞ ]
pred = [ -1, 0, 0, 2, 2, 2, -1, -1 ]
```

Next, we could choose either 1 or 3, since both vertices have minimum distance 5. Suppose we choose 1. Relaxation along (1,5,2) and (1,6,7) results in new values for nodes 5 and 6:

```
vSet = { 3, 4, 5, 6, 7 }
dist = [ 0, 5, 4, 5, 7, 7, 12, ∞ ]
pred = [ -1, 0, 0, 2, 2, 1, 1, -1 ]
```

Now we consider vertex 3. The only adjacent node still in vSet is 4, but there is no shorter path to 4 through 3. Hence no update to dist[] or pred[]:

```
vSet = { 4, 5, 6, 7 }
dist = [ 0, 5, 4, 5, 7, 7, 12, ∞ ]
pred = [ -1, 0, 0, 2, 2, 1, 1, -1 ]
```

Next we could choose either vertex 4 or 5. Suppose we choose 4. Edge (4,7,8) is the only one that leads to an update:

```
vSet = { 5, 6, 7 }
dist = [ 0, 5, 4, 5, 7, 7, 12, 15 ]
pred = [ -1, 0, 0, 2, 2, 1, 1, 4 ]
```

Vertex 5 is next. Relaxation along edges (5,6,3) and (5,7,6) results in:

```
vSet = { 6, 7 }
dist = [ 0, 5, 4, 5, 7, 7, 10, 13 ]
pred = [ -1, 0, 0, 2, 2, 1, 5, 5 ]
```

Of the two vertices left in vSet, 6 has the shorter distance. Edge (6,7,5) does not update the values for node 7 since dist[7]=13<dist[6]+5=15. Hence:

```
vSet = { 7 }
dist = [ 0, 5, 4, 5, 7, 7, 10, 13 ]
pred = [ -1, 0, 0, 2, 2, 1, 5, 5 ]
```

Processing the last remaining vertex in vSet will obviously not change anything. The values in pred[] determine shortest paths to all nodes as follows:

```
0: distance = 0, shortest path: 0
1: distance = 5, shortest path: 0-1
2: distance = 4, shortest path: 0-2
3: distance = 5, shortest path: 0-2-3
4: distance = 7, shortest path: 0-2-4
5: distance = 7, shortest path: 0-1-5
6: distance = 10, shortest path: 0-1-5-6
7: distance = 13, shortest path: 0-1-5-7
```

c. Shortest path from vertex 3 to vertex 6 has total weight 10. The most direct paths use three edges, e.g. 3-0-1-6, 3-2-1-6 or 3-2-5-6. If we sum the lengths on each of these paths, the shortest one is 3-2-5-6 with total weight 11. However, there is a path from vertex 2 to vertex 5 that is shorter than the direct edge, via vertex 4. This reduces the total weight of the path from 3 to 6 (now via 2, 4 and 5) to 1+3+3+3=10.

Since all weighted edges in the graph are bidirectional, the shortest path from vertex 6 to vertex 3 is the reverse of the shortest path we found from 3 to 6: 6-5-4-2-3. Its total weight is the same (10).

d. Initialisation:

dist	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[0]	0	5	4	6	∞	∞	∞	∞
[1]	5	0	8	∞	∞	2	7	∞
[2]	4	8	0	1	3	7	∞	∞
[3]	6	∞	1	0	6	∞	∞	∞
[4]	∞	∞	3	6	0	3	∞	8
[5]	∞	2	7	∞	3	0	3	6
[6]	∞	7	∞	∞	∞	3	0	5
[7]	∞	∞	∞	∞	8	6	5	0

path	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[0]	–	1	2	3	–	–	–	–
[1]	0	–	2	–	–	5	6	–
[2]	0	1	–	3	4	5	–	–
[3]	0	–	2	–	4	–	–	–
[4]	–	–	2	3	–	5	–	7
[5]	–	1	2	–	4	–	6	7
[6]	–	1	–	–	–	5	–	7
[7]	–	–	–	–	4	5	6	–

After 1st iteration `i=0`:

dist	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[0]	0	5	4	6	∞	∞	∞	∞
[1]	5	0	8	11	∞	2	7	∞
[2]	4	8	0	1	3	7	∞	∞
[3]	6	11	1	0	6	∞	∞	∞
[4]	∞	∞	3	6	0	3	∞	8
[5]	∞	2	7	∞	3	0	3	6
[6]	∞	7	∞	∞	∞	3	0	5
[7]	∞	∞	∞	∞	8	6	5	0

path	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[0]	–	1	2	3	–	–	–	–
[1]	0	–	2	0	–	5	6	–
[2]	0	1	–	3	4	5	–	–
[3]	0	0	2	–	4	–	–	–
[4]	–	–	2	3	–	5	–	7
[5]	–	1	2	–	4	–	6	7
[6]	–	1	–	–	–	5	–	7
[7]	–	–	–	–	4	5	6	–

After 2nd iteration `i=1`:

dist	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[0]	0	5	4	6	∞	7	12	∞
[1]	5	0	8	11	∞	2	7	∞
[2]	4	8	0	1	3	7	15	∞
[3]	6	11	1	0	6	13	18	∞
[4]	∞	∞	3	6	0	3	∞	8
[5]	7	2	7	13	3	0	3	6
[6]	12	7	15	18	∞	3	0	5
[7]	∞	∞	∞	∞	8	6	5	0

path	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[0]	–	1	2	3	–	1	1	–
[1]	0	–	2	0	–	5	6	–
[2]	0	1	–	3	4	5	1	–
[3]	0	0	2	–	4	0	0	–
[4]	–	–	2	3	–	5	–	7
[5]	1	1	2	1	4	–	6	7
[6]	1	1	1	1	–	5	–	7
[7]	–	–	–	–	4	5	6	–

After 3rd iteration `i=2`:

After 3rd iteration `i=2`:

dist	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[0]	0	5	4	5	7	7	12	∞
[1]	5	0	8	9	11	2	7	∞
[2]	4	8	0	1	3	7	15	∞
[3]	5	9	1	0	4	8	16	∞
[4]	7	11	3	4	0	3	18	8
[5]	7	2	7	8	3	0	3	6
[6]	12	7	15	16	18	3	0	5
[7]	∞	∞	∞	∞	8	6	5	0

path	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[0]	–	1	2	2	2	1	1	–
[1]	0	–	2	2	2	5	6	–
[2]	0	1	–	3	4	5	1	–
[3]	2	2	2	–	2	2	2	–
[4]	2	2	2	2	–	5	2	7
[5]	1	1	2	2	4	–	6	7
[6]	1	1	1	1	1	5	–	7
[7]	–	–	–	–	4	5	6	–

After 4th iteration `i=3`: unchanged

After 5th iteration `i=4`:

dist	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[0]	0	5	4	5	7	7	12	15
[1]	5	0	8	9	11	2	7	19
[2]	4	8	0	1	3	6	15	11
[3]	5	9	1	0	4	7	16	12
[4]	7	11	3	4	0	3	18	8
[5]	7	2	6	7	3	0	3	6
[6]	12	7	15	16	18	3	0	5
[7]	15	19	11	12	8	6	5	0

path	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[0]	–	1	2	2	2	1	1	2
[1]	0	–	2	2	2	5	6	2
[2]	0	1	–	3	4	4	1	4
[3]	2	2	2	–	2	2	2	2
[4]	2	2	2	2	–	5	2	7
[5]	1	1	4	4	4	–	6	7
[6]	1	1	1	1	1	5	–	7
[7]	4	4	4	4	4	5	6	–

After 6th iteration `i=5`:

dist	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[0]	0	5	4	5	7	7	10	13
[1]	5	0	8	9	5	2	5	8
[2]	4	8	0	1	3	6	9	11
[3]	5	9	1	0	4	7	10	12
[4]	7	5	3	4	0	3	6	8
[5]	7	2	6	7	3	0	3	6
[6]	10	5	9	10	6	3	0	5
[7]	13	8	11	12	8	6	5	0

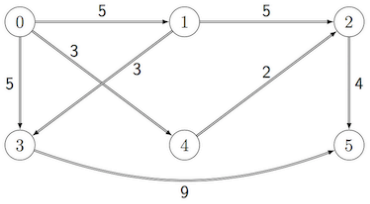
path	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[0]	–	1	2	2	2	1	1	1
[1]	0	–	2	2	5	5	5	5
[2]	0	1	–	3	4	4	4	4
[3]	2	2	2	–	2	2	2	2
[4]	2	5	2	2	–	5	5	7
[5]	1	1	4	4	4	–	6	7
[6]	5	5	5	5	5	5	–	7
[7]	5	5	4	4	4	5	6	–

The last two iterations (`i=6,7`) do not effect any more changes.

Observe that the final distances and shortest paths between 0 and all other vertices are the same as in the solution to exercise 4b, as it should be.

5. (Maximum flow)

a. Identify a maximum flow from source=0 to sink=5 in the following network (without applying the algorithm from the lecture):



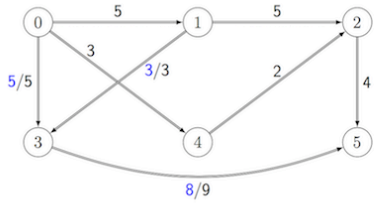
What approach did you use in finding the maxflow?

b. Show how Edmonds and Karp's algorithm would construct a maximum flow from 0 to 5 for the network above. How many augmenting paths do you have to consider?

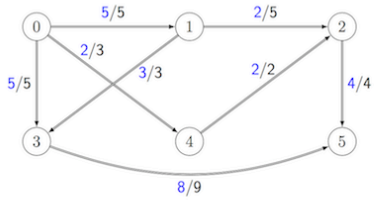
Answer:

a. I suspect (no proof) that most people would use a strategy like:

1. Choose an edge from a vertex into the sink and attempt to see if and how the maximum flow through this edge can be achieved, for example edge (3,5) with capacity 9.
2. Attempt to maximise the flow into this vertex 3 to check if the required capacity 9 can be reached.
3. Realise that the maximum flow into 3 is actually 8, which would get you to a partial solution like:



4. Inspect the other edge into the sink – (2,5) with capacity 4 – and attempt to max out its capacity.
5. Realise that the maximum flow that can be sent into node 2 is indeed 4, which results in the maximum flow shown below.



b. The shortest augmenting path found by BFS is 0-3-5, along which we can send `df`=5. The resulting `flow[][]` matrix is:

b. The shortest augmenting path found by BFS is 0-3-5, along which we can send `df`=5. The resulting `flow[][]` matrix is:

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	0	0	0	5	0	0
[1]	0	0	0	0	0	0
[2]	0	0	0	0	0	0
[3]	-5	0	0	0	0	5
[4]	0	0	0	0	0	0
[5]	0	0	0	-5	0	0

The negative values encode the "reversed" edges in the residual graph.

If we choose neighbours in ascending order, then the next augmenting path found by BFS is 0-1-2-5 with `df`=4. We update the `flow[][]` matrix to:

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	0	4	0	5	0	0
[1]	-4	0	4	0	0	0
[2]	0	-4	0	0	0	4
[3]	-5	0	0	0	0	5
[4]	0	0	0	0	0	0
[5]	0	0	-4	-5	0	0

Now the shortest augmenting path is 0-1-3-5 with `df`=1, so we update the `flow[][]` matrix as follows:

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	0	5	0	5	0	0
[1]	-5	0	4	1	0	0
[2]	0	-4	0	0	0	4
[3]	-5	-1	0	0	0	6
[4]	0	0	0	0	0	0
[5]	0	0	-4	-6	0	0

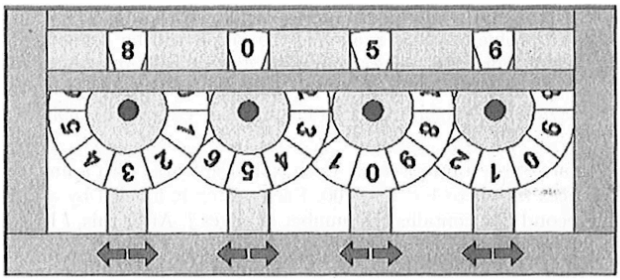
The residual graph now admits one more augmenting path: 0-4-2-1-3-5 with `df`=2, by which some of the current flow from 1 to 2 is "redirected" to 3 and replaced by the same amount from 4 to 2. This results in the `flow[][]` matrix:

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	0	5	0	5	2	0
[1]	-5	0	2	3	0	0
[2]	0	-2	0	0	-2	4
[3]	-5	-3	0	0	0	8
[4]	-2	0	2	0	0	0
[5]	0	0	-4	-8	0	0

No more augmenting path exists. The maximum flow is 5+4+1+2 = 12, the sum of what we could maximally send along the 4 augmenting paths that we computed.

6. **Challenge Exercise**

Consider a machine with four wheels. Each wheel has the digits 0…9 printed clockwise on it. The current *state* of the machine is given by the four topmost digits `abcd`, e.g. 8056 in the picture below.



Each wheel can be controlled by two buttons: Pressing the button labelled with "←" turns the corresponding wheel clockwise one digit ahead, whereas presssing "→" turns it anticlockwise one digit back.

Write a C-program to determine the *minimum* number of button presses required to transform

- a given initial state, `abcd`
- to a given goal state, `efgh`
- without passing through any of *n≥0* given "forbidden" states, `forbidden[]={`$w_1x_1y_1z_1,\ldots,w_nx_ny_nz_n$`}`.

For example, the state 8056 as depicted can be transformed into 0056 in 2 steps if `forbidden[]={}`, whereas a minimum of 4 steps is needed for the same task if `forbidden[]={9056}`. (Why?)

Use your program to compute the least number of button presses required to transform 8056 to 7012 if

a. there are no forbidden states;
b. you are not permitted to pass through any state 7055–8055 (i.e., 7055, 7056, …, 8055 all are forbidden);
c. you are not permitted to pass through any state 0000–0999 or 7055–8055

Answer:

The problem can be solved by a breadth-first search on an undirected graph:

- nodes 0…9999 correspond to the possible configurations 0000 – 9999 of the machine;
- edges connect nodes v and w if, and only if, one button press takes the machine from v to w and vice versa.

The following code implements BFS on this graph with the help of the integer queue ADT from the lecture (`queue.h`, `queue.c`). Note that the graph is built *implicitly* : nodes are generated as they are encountered during the search.

```
#include "queue.h"

int shortestPath(int source, int target, int forbidden[], int n) {
   int visited[10000];

   int i;
   for (i = 0; i <= 9999; i++)
      visited[i] = -1;              // mark all nodes as unvisited
   for (i = 0; i < n; i++)
      visited[forbidden[i]] = -2;  // mark forbidden nodes as visited => they won't be selected
   visited[source] = source;

   queue Q = newQueue();
   QueueEnqueue(Q, source);
   bool found = (target == source);
   while (!found && !QueueIsEmpty(Q)) {
      int v = QueueDequeue(Q);
      if (v == target) {
         found = true;
      } else {
         int wheel, turn;
         for (wheel = 10; wheel <= 10000; wheel *= 10) { // fancy way of generating the
            for (turn = 1; turn <= 9; turn += 8) {       // eight neighbour configurations of v
               int w = wheel * (v / wheel) + (v % wheel + (wheel/10) * turn) % wheel;
               if (visited[w] == -1) {
                  visited[w] = v;
                  QueueEnqueue(Q, w);
               }
            }
         }
      }
   }
   dropQueue(Q);
   if (found) {                     // unwind path to determine length
      int length = 0;
      while (target != source) {
         target = visited[target];  // move to predecessor on path
         length++;
      }
      return length;
   } else {
      return -1;                    // no solution
   }
}
```

a. 9
b. 17
c. ∞ (unreachable)