INF 4130: Execises to Matchings and Flow

October 2 and 4, 2018

NB: 4/10 is last time for the Thursday exercise class (“Gruppe 2”)

The words "vertex" and "node" has the same meaning below.

With answers

Exercise 1

Solve Exercise 14.4 in the textbook (B&P) (and sketch a data structure for Exercise 14.5).

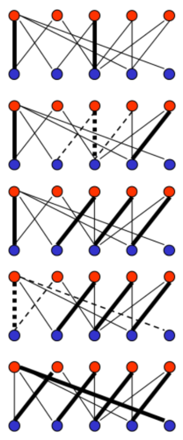
The exercise is to show that the Hungarian Algorithm can be implemented in time $O(n^3)$ for a bipartite graph $G = (X, Y, E)$, with $|X| = |Y| = n$.

As indicated in the exercise 14.4, the full algorithm consists of an outer loop where we repeatedly find and apply augmenting paths. Applying an augmenting path increases the size of our matching by one edge. The step of the outer loop can therefore be executed at most n times.

In the step of the outer loop we first find an unmatched node r and build a tree with this node as a root. During this tree building we may have to go through all edges out of red nodes, which amounts to $O(n^2)$ edges. Thus, if we can find a data-structure that bound the work for each such arc to $O(1)$, we are done. The most costly operation for such an arc occurs if it leads to a matched node outside the current tree. In this case we should include both end-nodes of the found matching edge into the tree, and include the red one of these into a set of *unexplored red nodes*. This set may be stored as a queue of some type, so that adding one and taking (a random one) out both take time $O(1)$. Also other necessary updating during this step takes time $O(1)$ if we have the following data in each node (in addition to its list of neighbours).

```
class Node
{
   Boolean isInTree = FALSE; // True if included in the list
   Node matchedWith = null; // If matched, a pointer to the node it is matched to
   Boolean proc isMatched { return matchedWith != null; }
   Node next;               // A pointer to implement the queue as a simple list
   Node parentInTree        // Pointer backwards toward the root
   < List of neighbours >;
}
```

Many of these variables will have to be reset in each node before each step in the outermost loop, so one should have an easy way to access all nodes. Then this resetting will not affect the worst case time of the implementation. The same is true for “using” an augumenting path as soon as we have found one. We may notice that the running time of this implementation can also be given as $O(n*m)$, where m is the number of edges in the graph and $m \geq n$. This may be considerably smaller than $O(n^3)$



Exercise 2

Solve Exercise 14.6 in the textbook.

We start with the graph given in the exercise, at the top of the figure to the right. We number the vertices from left to right $x_1, x_2, \ldots, x_5$ and $y_1, y_2, \ldots, y_5$. We choose to start by growing a tree from x_5, and immediately get an augmenting path (of *one* edge) if we first look at the edge (x_5, y_4). Remember that an edge with unmatched vertices at both ends is a (simplest possible) augmenting path.

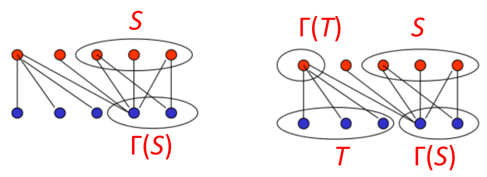
After this augmentation has been done, we can start building a tree from for instance x_4, and one possibility is then that we find the augmenting path x_4-y_3-x_3-y_2 (dotted lines in graph number two).

Applying this augmenting path we get the third graph, and if we then build a tree from x_2, we sooner or later find the augmenting path indicated by the dotted lines in graph number four. Applying *that* augmenting path results in the perfect matching of graph number five.

Exercise 3

Assume $|X| = |Y|$. Then show that, if we have found a subset S of X with $|\Gamma(S)| < |S|$, we can also easily find a subset of T of Y with $|\Gamma(T)| < |T|$.

This is actually very easy. Assume we have a subset S of X such that $\Gamma(S)$ has fewer vertices than S, as shown in the figure below. By the definition of $\Gamma(S)$ no edge can go between S and $T = Y - \Gamma(S)$. Therefore $\Gamma(T)$ must be a subset (not necessarily proper) of $X - S$, and thereby be smaller than T.



Exercise 4

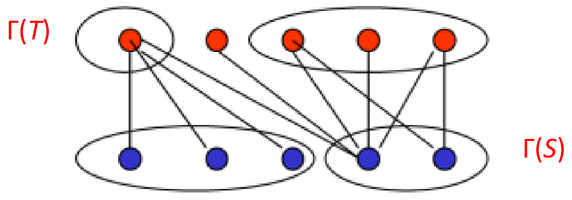
Question 4.a

Show that, for general graphs, any "node cover" will never have fewer nodes than there are edges in any matching. A "node cover" is a subset NC of nodes that "covers" all the edges. That is: all edges have at least one of its end nodes in NC.

Assume that a graph G has a matching M, and a node cover NC. Then each edge in M must have at least one of its end nodes in NC (otherwise NC did not cover all edges). The end nodes of an edge in M must be separate from the end nodes of any other edge in M. Thus the number of nodes in NC must be at least as large as the number of edges in M.

Question 4.b

Look at some examples with bipartite graphs, and observe that in such graphs you can always find a matching and a node cover of the same size. (It is in fact not difficult to prove this by looking at the situation when the Hungarian algorithm stops after having built alternating trees from all unmatched nodes in X, and no augumenting path has been found. The above fact can be used to prove that a certain match is as large as possible, also for cases with $|X| \neq |Y|$.

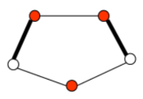


As an example, we can look at the graph from Exercise 3. A cover could be x_1, y_4 and y_5, which is the union of $\Gamma(T)$ and $\Gamma(S)$. This also indicates how a node cover can be found when the matching algorithm stops. A matching with three edges is easy to find, and we then know that this has as many edges as possible.

Question 4.c

Find an example showing that, in *general* graphs, one cannot always find a node cover and a matching of the same size.

Finding a graph where the maximum matching and the minimum vertex cover are of different size is easy. The canonical example of a non-bipartite graph, the odd loop, does the trick. In a C_5, for instance, the largest matching has two edges, while the smallest vertex cover has three nodes.

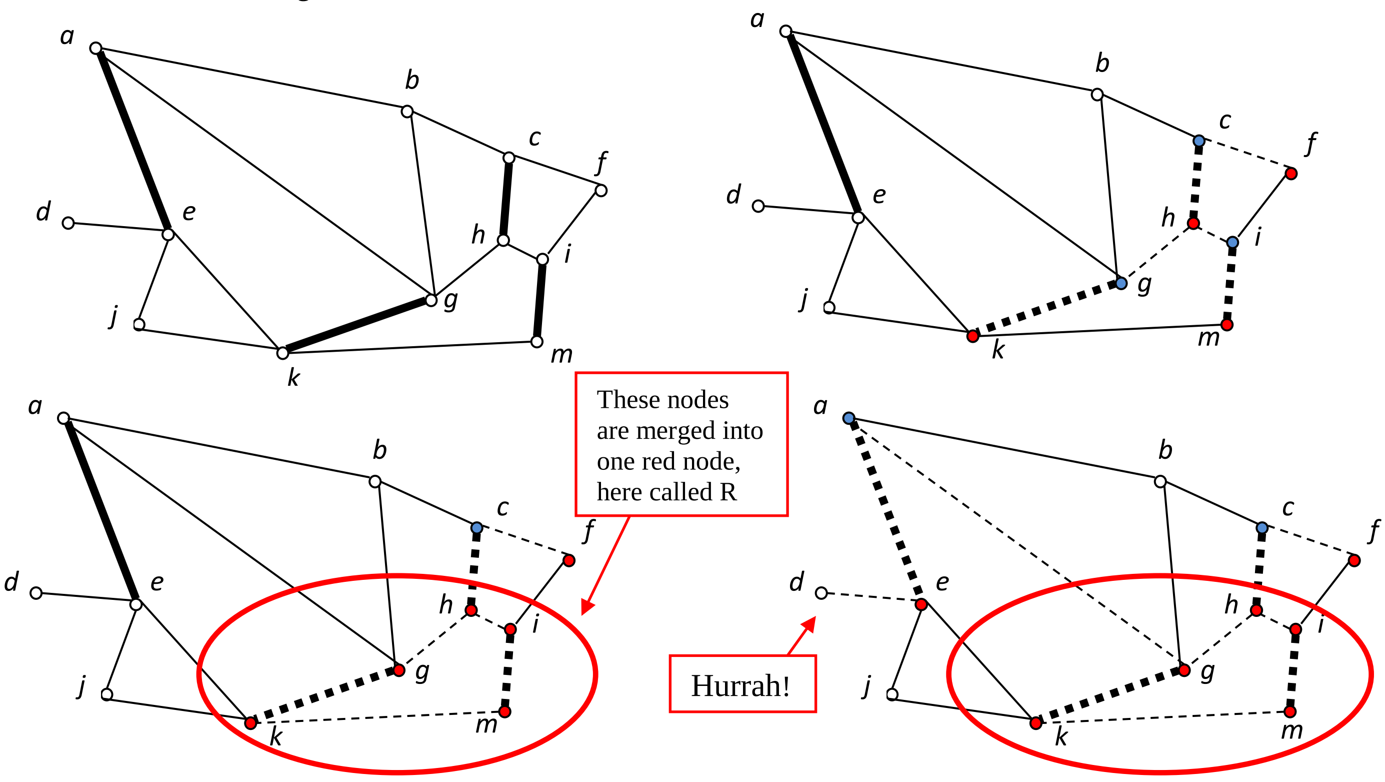


Exercise 5

We are given the following graph *G*, and given matching M. You shall use the maximum matching algorithm for general graphs to find a maximum matching for G, by starting with M. Start at node *f* as the root, then look at the edge *f*-*c* getting also *c*-*h* into the tree. Then look at edges *h*-*g* and *h*-*i*, which will both increase the tree by two nodes each.

Which nodes are now red and blue (assuming that the root *f* is red)?

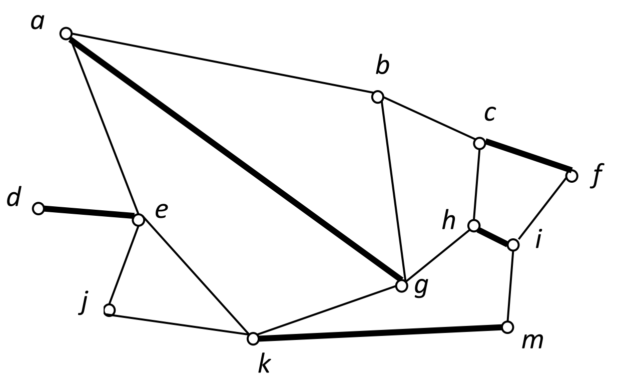
Then look at the unmatched edge out of *m*. What will happen then? Proceed with choices so that you end up finding an augumenting path between *d* and *f* (even though one between *b* and *f* or *j* and *f* is closer by). Show the resulting matching after you have "used" this augumenting path. Finally, decide whether this matching can be increased further.



When "using" the found augumenting path (above), the large red node R (made from merging the nodes *h*, *g*, *k*, *i*, *m* in an earlier step), must now be unwrapped, so that we can see that the correct alternating path through it (from the *d* back to *f*) is:

g, *k*, *m*, *i*, *h*.

The resulting matching will be:



Thus, the size of the matching is increased by one. To decide whether it can be increased even further, we must repeat the tree-building process from all unmatched nodes, that is, from *j* and from *b* (and here the proposed answer from 2017 was in fact wrong, as it said that no augumenting path exists.) There is in fact at least one augumenting path as follows: b, a, g, h, i, m, k, j, and using this will indeed lead to "complete" matching. If we were lucky enough to look at new edges in exactly that order we would need no "collaps" of odd loops along the way. However, if we started by looking at the edge to c from b this would extend the tree with (b, c) and (c, f), and then proceeded by looking at edge (f,.i) and thus also include (i, h) in the tree, and then include (h, g) and thus also (g, a), and finally look at (a, b) we would have an odd loop that we would have to collaps.

Extra exercise: Finalize this process until it finds a proper augumenting path, and thus a complete matching.

Exercise 6

To study the max flow algorithm, go through the example in Figure 14.9 in detail (B&P). See introduction at the bottom of page 439. Note that there are many typos in these graphs in early editions of the book, but most of them should now be corrected. Also note that the first graph in the left column is N (and not N*f*), and that in the right column step 6 has the final flow, while the he last graph is N itself with the (original) capacities , and where the cut is displayed with dotted edges.

Known typos in early editions of the textbook are:

Step 1:	Edge 4-7 in N_f should be dotted.
Step 2—7:	Edge 4-7 shoud be reversed in all N_fs.
Step 2:	Inner edges in the flow graph should be removed.
Step 2:	Edge 0-3 in N_f should not be dotted.
Step 7:	Vertex 5 in *N* should have a double circle, and an edge 2-5 with flow 1 should be added to the flow graph.
Step 7:	The sets should be X = {0,1,2,3,5}, and Y = {4,6,7}.

The figure with typos corrected is included at the end of this document.

Left to the class

Exercise 7 (Question 7.c – 7.f can be left to the students)

Study figure 14.10 on page 444 of the text book (B&P). (Note that there are typos in at least some editions of the book: The edge ($x1$, $y2$) in the upper graph should be removed.) We now look at the duality between finding a maximum matching in the upper graph, and finding a maximum flow in the lower network (graph).

Question 7.a

Look at the following lemma, and explain why it is correct (Hint: This has also been commented on in the lectures, and it relies on the way the algorithm works):

> **Lemma** *In a network with integer capacities one can always find a flow that is both maximum and integer, and the Ford-Fulkerson-algorithm will always find such a flow.*

In other words: If the capacities are integer, we never have to split a flow so that for instance ½ goes down one edge and ½ down another to achieve a maximum flow. This means that if all capacities are 1, we get a maximum flow for the network with either full (1) or no (0) flow in each edge. Such a flow induces a subset of the edges: those with full flow.

FordFulkerson never splits an integer flow into non-integer flows, *and* proves optimality by showing a minimum cut with the same capacity as the flow.

Question 7.b

Use the lemma to explain that finding a maximum matching in the upper graph in Figure 14.10 is the same as finding a maximum flow in the lower network.

With capacity 1, flow is either 0 or 1. A flow of 1 corresponds to the edge being part of the matching.

Question 7.c

Assume that you in Figure 14.10 have the matching {($x2$, $y1$), ($x4$, $y3$), ($x5$, $y5$)}, and show what flow f this corresponds to in the lower network.

Left to the students or the class

Question 7.d

Draw $N(f)$ (the f-derived network) for the flow from 6.c and check that looking for an f-augmenting path from s to t in this graph corresponds to looking for a (matching) augmenting path in the upper graph, with the given matching.

Left to the students or the class

Question 7.e

Use an f-augmenting path found (for instance ($x1$, $y1$, $x2$, $y4$) in the graph and (s, $x1$, $y1$, $x2$, $y4$, t) in the network) to augment the matching/flow, and check that these operations are duals of each other. Verify that you end up in the situation shown in the lower network in figure 14.10 (where flows are indicated).

Left to the students or the class.

Question 7.f

Draw $N(f)$ for this new flow, and show that the flow is a maximum flow by showing a cut with this capacity (4). Then use the method from Exercise 4 above to find a vertex cover of four vertices covering all edges in the upper graph, thereby showing that the matching is a maximum matching. Finally show how the cut and this vertex cover are related.

Left to the students or the class.

Exercise 8 (if you have time)

Show that the following three conditions on undirected graphs are equivalent:

- The graph is bipartite
- The graph is two-colourable
- The graph has no odd cycles

Left to the students or the class.

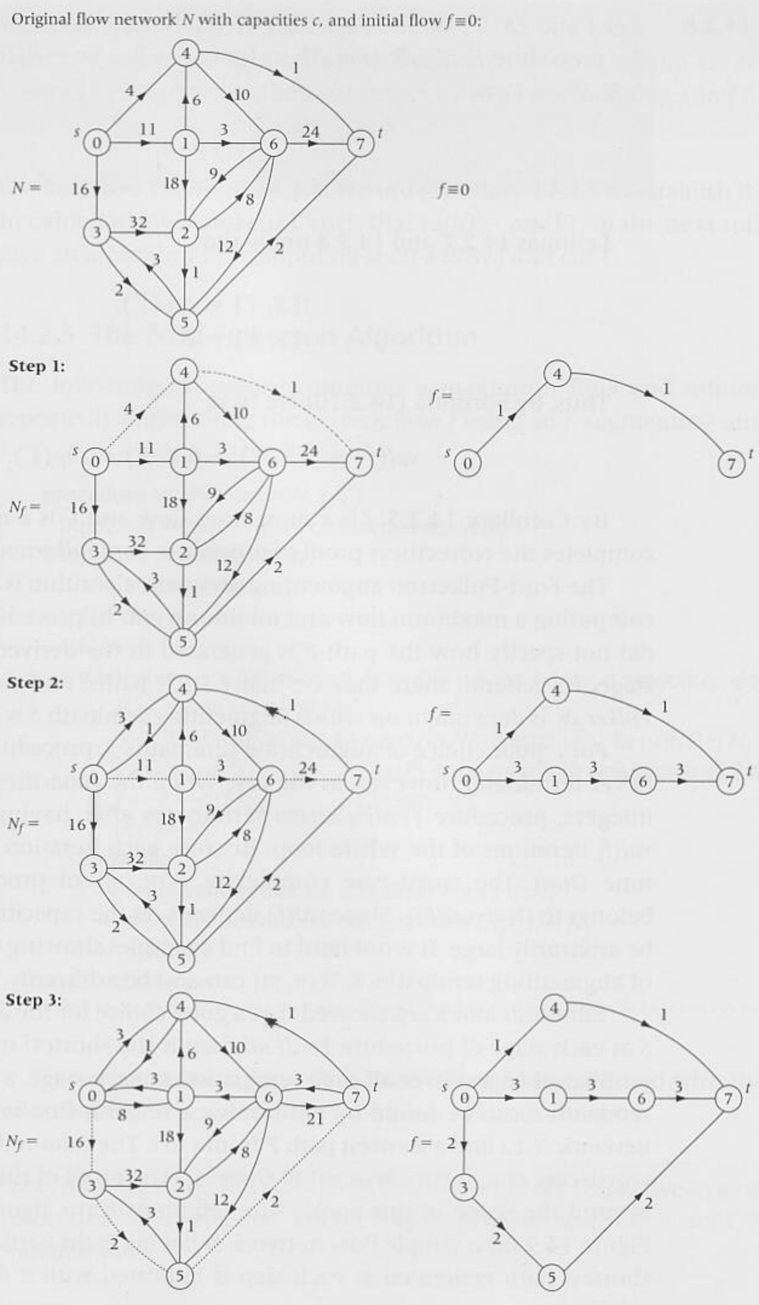
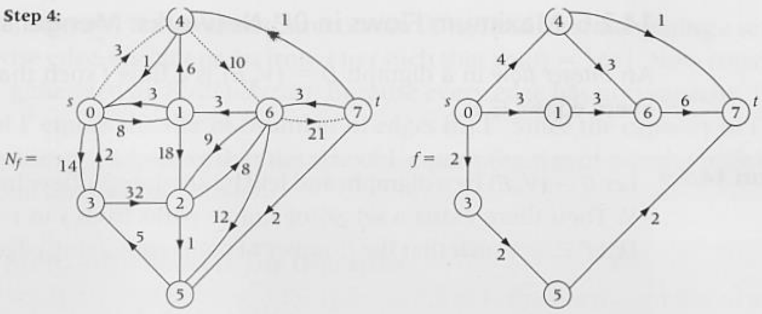

Original flow network N with capacities c, and initial flow $f \equiv 0$:
$N =$
$f \equiv 0$
Step 1:
$N_f =$
$f =$
Step 2:
$N_f =$
$f =$
Step 3:
$N_f =$
$f =$
s
t

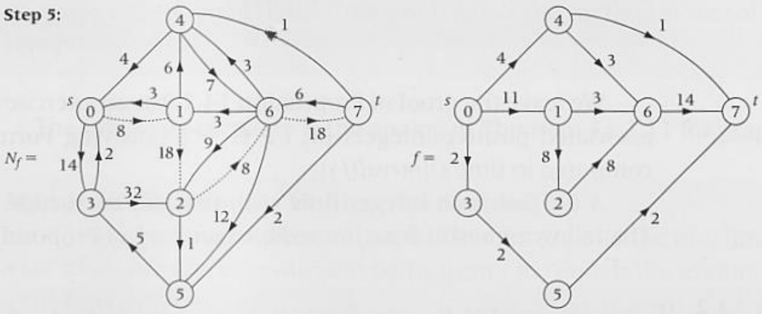
FIGURE 14.9 Action of the Edmonds-Karp algorithm for a sample capacitated network *N*.

FIGURE 14.9 Continued

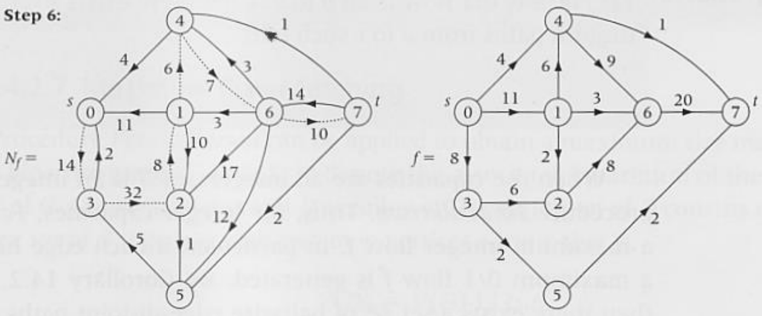
Step 4:



Step 5:

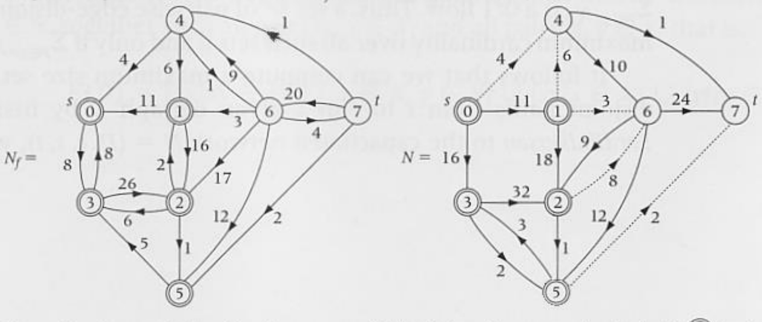


Step 6:



There are no more augmenting semipaths. The final flow f has value 23.

Step 7: Compute the f-derived network N_f and minimum cut $cut(X,Y)$.



The set $X = \{0,1,2,3,5\}$ of vertices that are accessible in N_f from the source s (marked with ◯) and the set $Y = \{4,6,7\}$ of vertices that are not accessible from s determine a cut $\Gamma = cut(X,Y)$ of capacity $c(X,Y) = 4 + 6 + 3 + 8 + 2 = 23$. Hence, we have $val(f) = 23 = cap(\Gamma)$, so that f is a maximum