Algoritmekonstruksjon TDT4125 - Assignment 1

Ola Kristoffer Hoff

12th February, 2023

Task 1

Consider the graph $G = (V, E)$ below, and let M be the matching consisting of the highlighted edges.

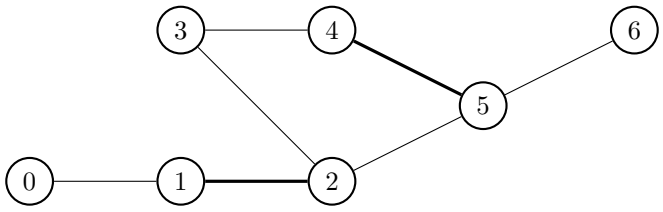


Figure 1: Graph G with labelled vertices. Matcing M is shown.

a Is M maximum, maximal or neither?

The graph cannot be extended by another edge without using a vertex already in use, hence the matching is at least maximal.

By eyeballing it I can see that it is possible to improve the matching, and hence it's not maximum. How will be shown in the next step.

b Find an augmenting path P in G with regards to M and use P to construct a new, larger matching M'.

The path $P = \{3, 4, 5, 6\}$ is an augmenting path. Using this to turn M into M' by: $M' = M \oplus P$

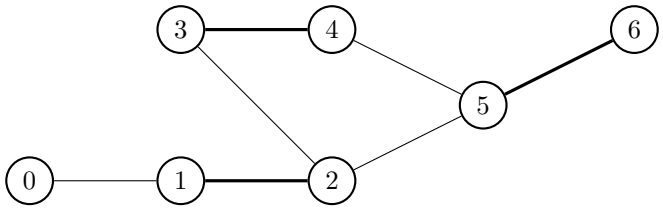


Figure 2: Matching M'.

c Is M' maximum, maximal or neither?

Since it is derived from M it's still at least maximal. Also note that there is an odd number of vertices, seven, and we have saturated six of them. This means there is no improvement to be made in this unweighted graph matching. We can also see that there are no more augmenting paths. This is easy to see given that the definition requires the path to start and end in unsaturated vertices, which we only have one of. Hence, the matching is maximum.

d We say that a matching is perfect if all vertices are matched. There does not exist a perfect matching for G. We wish to construct a new graph $G' = (V, E')$, such that G' has a perfect matching. Does G' exist?

No, as stated above, we have an odd number of vertices, so there will always be one "left out". So the new graph, given the same vertex set will not be able to have a perfect matching either.

Task 2

You are in possession of several droids in need of repair. The droids have all had a malfunction in the logic circuit that controls their movements, either due to water damage, short circuits or blown capacitors. Because of an ongoing rebellion, the available replacement circuits in your planetary system are severely limited. As each droid is only compatible with some types of circuits, you wish find the best matching of your droids and the available circuits, such that you maximise the number of repaired droids. The table below shows a list of your droids (values in the left column), available replacement circuits (values in the top row) and the compatabilities.

	RR-X1 (4)	Y2-P7X (5)	RR-X2 (6)	Z2-497 (7)
R2-D2 (0)			X	
ME-8D9 (1)	X		X	
2-1B (2)		X	X	X
R5-D4 (3)	X		X	

Table 1: Table of droids and compatible circuits, indices noted in parentheses.

a Can Hall's Marriage Theorem be used to determine if there exists a perfect matching of droids and replacement circuits? If yes, does a perfect matching exist?

So Hall's Marriage Theorem states that a perfect matching for a graph: $G = (U, V, E)$, exists if: $|S| \leq |N(S)|, \ \forall S \subseteq U$. If we state this problem as a bipartite graph problem with droids on the left side, U, and circuits on the right, V. Then we can apply Hall's theorem to prove if there exists a perfect matching.

We can see by example that there's not a perfect matching. Since we have the subset: $S = \{R2 - D2, ME - 8D9, R5 - D4\}$. Here the $|S| = 3$ and $N(S) = 2$. Hence, by contradiction there is not a perfect matching in the graph.

b Use the flow-based matching algorithm to find a best possible matching of droids and replacement circuits.

To solve the problem as a flow-based problem we first create a bipartite graph representation of the problem, as seen in figure 3.

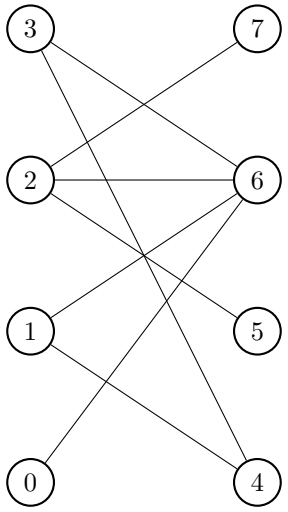


Figure 3: Bipartite graph of table 1.

Then we have to turn it into a flow problem. First we add a source and drain, connect from the source to all left side vertices, and the same for all right side vertices to the drain. All edges are directed and going right, from source towards the drain. All edges also have a capacity of one. We then get figure 4.

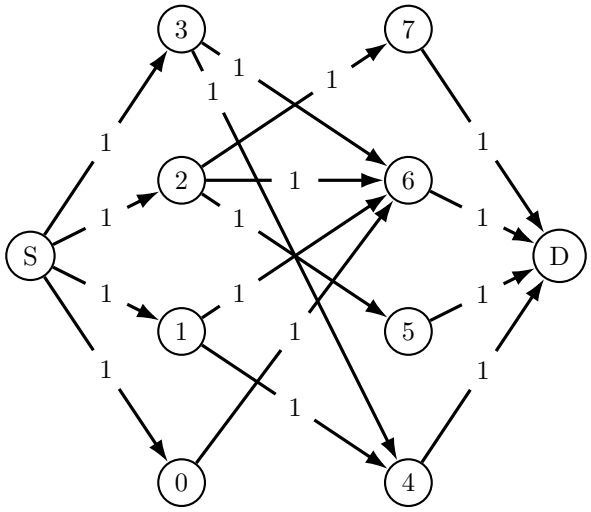


Figure 4: Bipartite graph from figure 3 turned into a flow network.

We can now solve this flow-problem with the Ford-Fulkerson algorithm[1] (algorithm 1). I've tried to illustrate the algorithm step for step with the following figures.

Algorithm 1 Sudo-code for the Ford-Fulkerson algorithm

$G = (V, E)$
$S \in V, S \leftarrow source$
$D \in V, D \leftarrow drain$
Ensure: $\forall e \in E => e_{flow} \leftarrow 0$ ▷ Initialise all flow to zero.
while $P \leftarrow find\_augmenting\_path(G', S, D)$ **do** ▷ Can be found using a BFS [2](page. 18)
$G \leftarrow G + P$ ▷ Add augmenting path to flow.
end while
return $\sum e_{flow}$, $e \leftarrow (u, D), u \in V$ ▷ Return flow into D, maximum flow.

It is worth noting that the augmenting path found by BFS or DFS is any path that can carry flow from S to D. I've used DFS since I assumed it would make for more illustrative visuals.

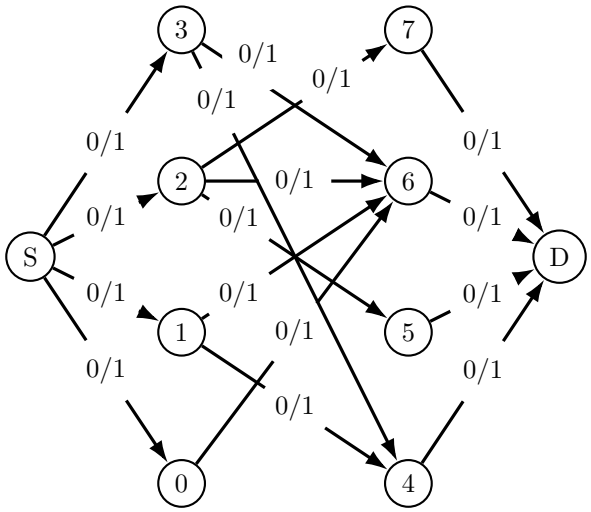


Figure 5: Initial state.

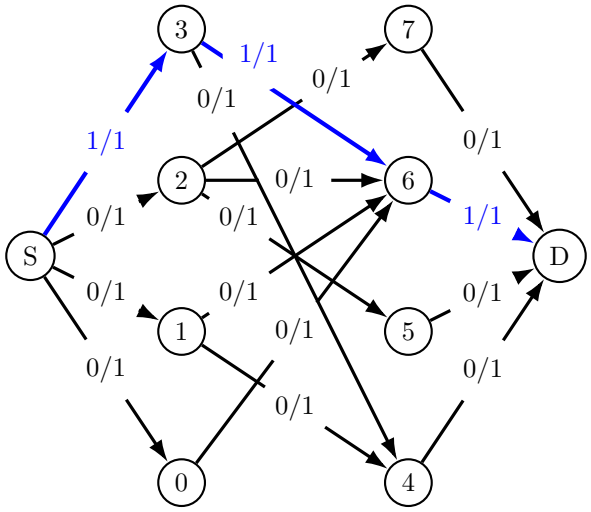


Figure 6: Iteration 1.

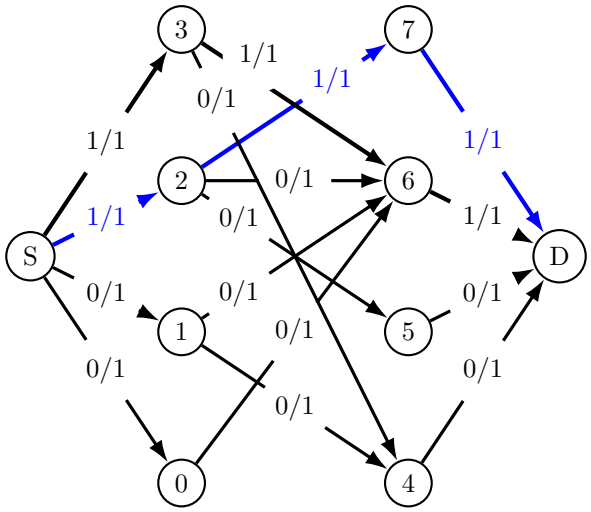


Figure 7: Iteration 2.

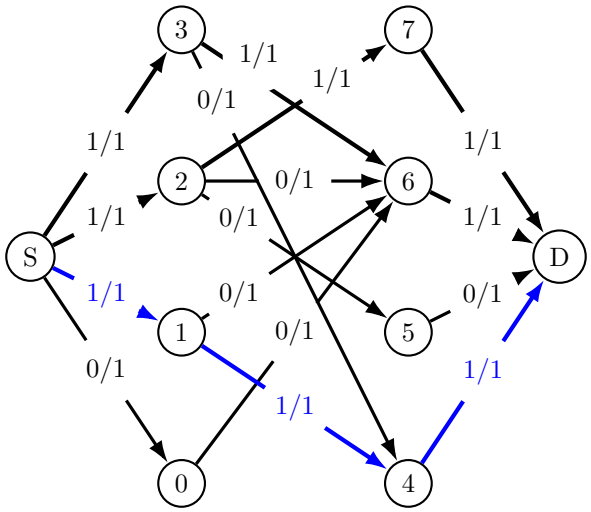


Figure 8: Iteration 3.

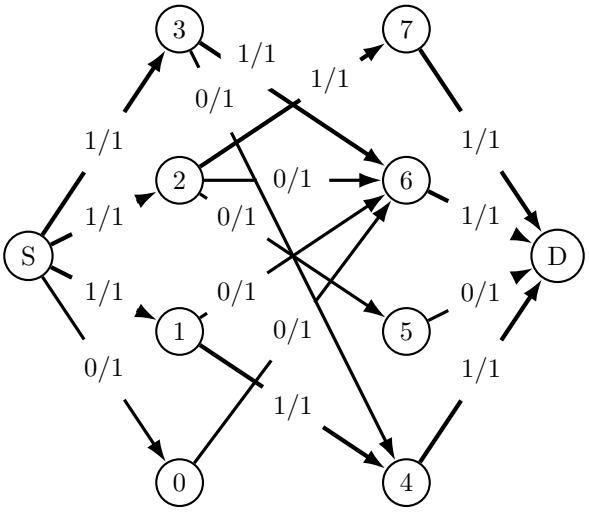


Figure 9: Iteration 4.

Here, in the fourth iteration, the algorithm would stop as there is no more augmenting paths from S to D left. We see the flow is equal to 3. Lastly I have showed the bipartite graph solved for maximum matching in figure 10.

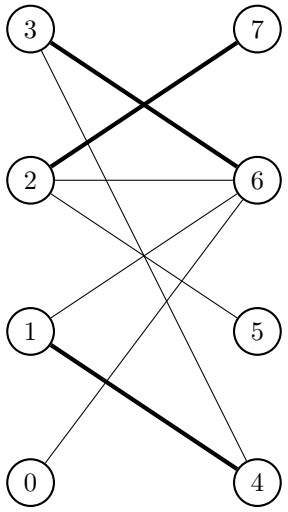
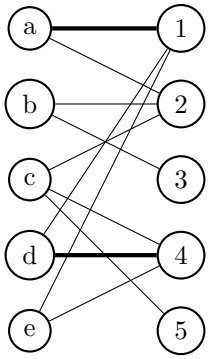


Figure 10: Bipartite graph after solving it as a flow-based problem.

Task 3

Consider the graph $G = (V, E)$ below, and let M be the matching consisting of the highlighted edges. Assume that M is the result of running one iteration of the Hopcroft-Karp algorithm. Perform the remaining iterations of the Hopcroft-Karp algorithm to find a maximum matching in G.



First I would like to note that the graph M, a matching of G, is not the actual result of running the Hopcroft-Karp algorithm, but we have this as an assumption. The algorithm is detailed in the compendium at page 21, I will be basing myself on it. Also I refer to the graph as: $G = (A \cup B, E)$, $A \cup B = V$, this to easily distinguish between the left, A, and right, B, side vertices.

First we need to generate the set F by running the BFS from all unsaturated vertices in A, $\{b, c, e\}$, until they find a match in B. From this we get the BFS trees shown in figure 11.

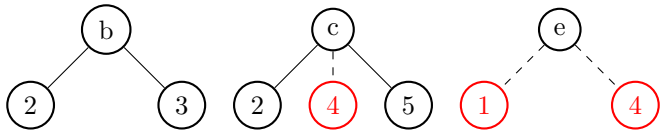


Figure 11: The DFS from the free vertices in A at iteration 2. Paths in the trees that are marked with dashed edges and a red node at the end are not augmenting paths, but paths being constructed when other paths finished.

Now that the BFS trees are constructed we will run DFS on the leaf vertices. Let's start with 2 in b. This gets us a path: $P_1 = \{2, b\}$. We add P_1 to the set P of disjoint augmenting paths. We now remove this path from the DFS trees and all orphan vertices, giving us figure 12

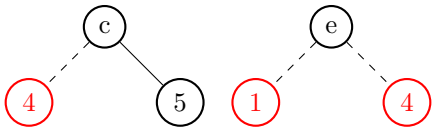


Figure 12: The DFS trees from figure 11, but after removing the first path.

The next and last path is obvious: $P_2 = \{5, c\}$.
We now use the paths in P to update our matching M by: $M = M \oplus p, \ \forall p \in P$.

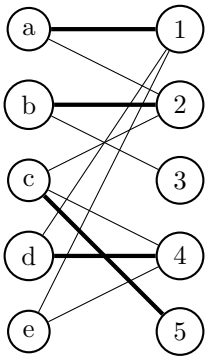


Figure 13: Matching M after second iteration.

The resulting matching after completing the second iteration of the algorithm is shown in figure 13. Now we do the same again for the next iteration:
We get these BFS trees:

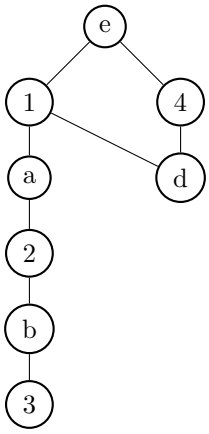


Figure 14: The DFS from the free vertex, e, in A at iteration 3.

Now we run DFS from vertex 3 and find the path: $P_1 = \{3, b, 2, a, 1, e\}$. This is the only path this iteration, and we add it to M, shown in figure 15, to finish the iteration.

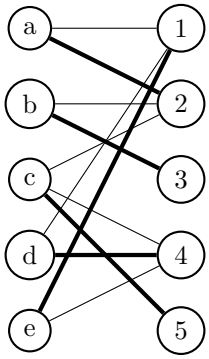


Figure 15: Matching M after third iteration.

As we can clearly see in figure 15 the matching is now perfect, and we can stop the algorithm.

Task 4

On page K26 and K27 of the compendium, Erciyes introduces an algorithm for finding maximal matchings in weighted bipartite graphs. The algorithm guarantees that the maximal matching has a weight of at least 1/2 the weight of any maximal weighted matching in the given graph. We thus say that the algorithm has an approximation factor of 1/2, since it is guaranteed to obtain at least 1/2 of the value of an optimal solution.

a Does the algorithm work for arbitrary graphs? If so, what is the approximation factor of the algorithm in this case?

The algorithm would work on any weighted graph, not just bipartite graphs. It just picks the largest (or smallest) weight that is still available and continues until there are non. This will always result in a maximal matching, since there are no edges to be added. The question of how good the approximation is for an arbitrary graph is harder.

We can think of it like this: if there are no augmenting paths, then it is the maximum matching, this case puts our approximation factor at 1.0.

Then we have the more common case of having augmenting paths. If an augmenting path, P, exists, then we can find $M' = M \oplus P$ giving us: $|M'| = |M| + 1$ in cardinality. However, it's possible that the weights in the new path is smaller than the existing one, leading to the weight of $|M'| \leq |M|$.

Let's look more into this with an example. Here we are trying to construct a graph that the algorithm will perform very badly at. The scenario here is the graphs below. The worst possible outcome for the algorithm is if it chooses the highest value, but then by that choice removes the two next highest values. Treating this in

the realm of integers we see that given that it choose the weight w; the alternative could have been $2(w-1)$. This gives the factor of: $\lim_{w\to\infty} \frac{w}{2(w-1)} = \frac{1}{2}$

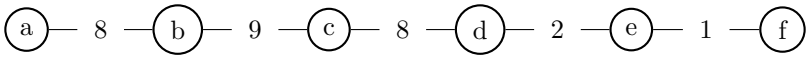


Figure 16: Example graph.

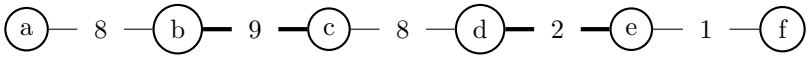


Figure 17: Weight obtained by algorithm: 11.

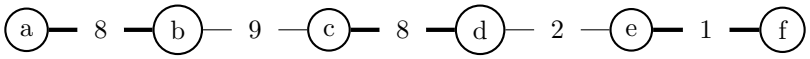


Figure 18: Weight in maximum matching of graph: 17.

Another worst case is the minimisation of the number of edges involved. Below we can see that the algorithm can be unlucky and take up three edges or saturate and isolate three vertices by one choice of edge.

In the best case one edge choice only removes two edges from the graph. We then have the edges from the algorithm, A, and the edges from the maximum matching, M. We then get the factor of edge usage: $\frac{|A|}{|M|} = \frac{\frac{1}{3}a_w}{\frac{1}{2}m_w} = \frac{2a_w}{3m_w},\ a_w \geq m_w \Rightarrow factor\ = \frac{2}{3}$. This factor is even greater than the previous one.

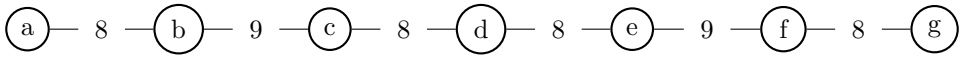


Figure 19: Example graph.

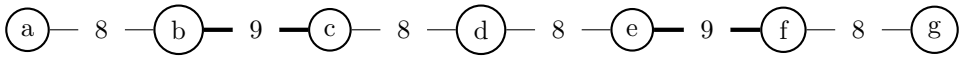


Figure 20: Weight obtained by algorithm: 18.

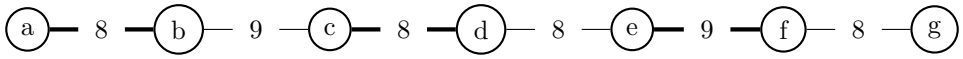


Figure 21: Weight in maximum matching of graph: 25.

In conclusion, from this non-formal proof, we have seen that it probably is the case that the factor is still $\frac{1}{2}$ for any arbitrary weighted graph too.

b Assume now that the weights in the graph are such that if $W = \langle w_1, w_2, ..., w_{|E|} \rangle$ is the weights sorted in increasing order, then $w_i \leq cw_{i+1}$ for all $i \in \{1, 2, ..., |E|-1\}$ $with$ $0 < c \leq 1$.

What is the approximation factor of the algorithm if $c = 3/4$? How about $c = 1/2$?

We can check these c-values with the same test case as in the last subsection. Here the relationship, in weight, between the weights are given. Let's start with $c = \frac{3}{4}$:

$$
\begin{aligned}
w_{|E|-2} &\leq cw_{|E|-1} \leq c^2 w_{|E|} \\
w_{|E|-2} &= cw_{|E|-1} = c^2 w_{|E|} \\
\text{factor} &= \frac{w_{|E|}}{w_{|E|-2} + w_{|E|-1}} \\
\text{factor} &= \frac{w_{|E|}}{c^2 w_{|E|} + cw_{|E|}} \\
\text{factor} &= \frac{w_{|E|}}{cw_{|E|}(c+1)} \\
\text{factor} &= \frac{1}{c(c+1)} \\
\text{factor} &= \frac{4}{3}\frac{4}{3+4} = \frac{16}{21} \approx 0.76
\end{aligned}
$$

We get for $c = \frac{3}{4}$ that we are guaranteed a solution that is at least 0.76 of the maximum. Note that we put the terms equal to each other in line two to maximise the disregard values, to present a worst case for the algorithm.

Doing the exact same for the other value, $c = \frac{1}{2}$:

$$
\begin{aligned}
\text{factor} &= \frac{1}{c(c+1)} \\
\text{factor} &= \frac{2}{1}\frac{2}{1+2} = \frac{4}{3} \approx 1.33
\end{aligned}
$$

Here the factor indicates that we will get a better solution with the greedy algorithm, this is due to the constraint relation, the other values are not big enough to make up for the gap. This makes sense, if both values had been a half of the first one, then they would be equal, but they are half and a quarter, so they only make up 75% of the biggest value, at max.

c What is the largest value c can take if the algorithm is to be guaranteed to find an optimal solution?

Using the results from the previous subsection we can solve for c:

$$
\begin{aligned}
\text{factor} &= \frac{1}{c(c+1)} = 1 \\
c(c+1) &= 1 \\
c^2 + c - 1 &= 0 \\
c &= -\frac{1+\sqrt{5}}{2} \vee \frac{\sqrt{5}-1}{2} \\
c &= \frac{\sqrt{5}-1}{2}, \;\; 0 < c \leq 1 \\
c &\approx 0.62
\end{aligned}
$$

This is very interesting, I notices that c here is the golden ratio minus one. We see here that if the relationship between the weights are less than 0.62 the algorithm is bound to find the optimal solution.

d We are now interested in finding a matching that is both maximal and has the lowest possible weight. Inspired by Erciyes's algorithm, we construct the following greedy algorithm (2).

Show that Min-Weight-Maximal-Matching finds a maximal matching.

Algorithm 2 Min-Weight-Maximal-Matching(G)

```
1: E' = E                                   ▷ ordered by increasing weight
2: M = ∅
3: while E'.length > 0 do
4:     e = the first edge in E'
5:     M = M ∪ {e}
6:     remove edges in E' adjacent to e
7: end while
8: return M
```

This is the exact same algorithm as the maximum only we select edges in increasing and not decreasing order of weight. So the proof is fairly straight forward. If it does not create a maximal matching then it means there is an edge between two unsaturated vertices that is not active, but should be. If such an edge had existed it would have to be in the set E' to start with. Then it would not have been removed, because we only remove active edges and their adjacent edges. This means when an edge is selected, we remove it and all edges from the vertices the edge connects. If we have two unsaturated vertices, that means no edge has connected to them, hence our edge between them could not have been removed, hence it must still be in E', and the algorithm would pick it at some point since it continues until the E' is empty.

e What is the running time of Min-Weight-Maximal-Matching?

First of the sorting of E to E' is bound by what ever sorting algorithm we choose. If the weights had been in a given integer range we could have used a radix sort for linear time, but more general it would probably use something that gives $O(|E|log(|E|))$ (i.e. merge- or heap sort).

The running time of the while loop in the algorithm is bound by $O(|E|)$. The calculation of M can be done in $O(1)$. The removal of edges at first sight seem to be bound by $O(|E|)$, and it is, but we don't get $O(|N|^2)$ for the loop. Since if we had just removed one edge each iteration we would run through all edges, and then have a constant of $O(1)$ on the removal. However, if we remove more than one edge at a time, we "skip" iterations in the loop, since it is also bound by the same variable. Hence, on average the removal is bound to $O(1)$.

So the while loop is bound by $O(|E|)$. This gives us that the whole algorithm is bound by: $O(|E|log(|E|) + |E|) = O(|E|(log(|E|) + 1)) = O(|E|log(|E|))$. Hence it is the sorting of the initial edges that is bounding the algorithm.

f Does the algorithm provide a constant approximation guarantee (if so, what is this constant) or can the ratio between the solution found by the algorithm and the optimal solution grow arbitrarily large?

It is not bound as the other algorithm is. The other algorithm had a constraint on the weights, they had to be positive, this limited the backfiring effect of negative weights. Here it is flipped and we can always choose a greater weight, hypothetically.

Here the worst case is if we by choosing the smallest, have to choose the most, since we have positive weights only, the total which will increase with the number of edges. If we choose weight w_1 and then have to choose w_3 as well, and w_2 is left. we can have the scenario that w_3 is arbitrarily big and w_2 is just one greater than w_1.

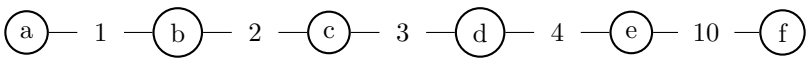


Figure 22: Example graph.

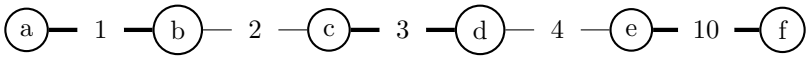


Figure 23: Weight obtained by algorithm: 14.

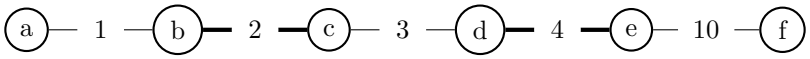


Figure 24: Weight in maximum matching of graph: 6.

As I've tried to illustrate in the figure above, the choice of smallest leads the algorithm to have to choose the greatest weight, to create a maximal matching. This last weight can be as large as we can imagine and hence this algorithm is not bound. The problem could always be transformed into a case where it could be solved by the other algorithm though.

Task 5

Consider the following set of inequalities:

$$\begin{aligned} x_1 &\leq 2x_2 + 1 \\ x_1 + x_3 &\leq 7 \\ 2x_4 + x_3 &\geq x_2 - 9 \\ x_2 + 0.5x_4 &\leq 13 \end{aligned} \tag{1}$$

and the following objective function:

$$\text{maximize } 1.5x_1 + 2x_2 + 2x_3 + x_4 \tag{2}$$

a Formulate a linear program in standard form (using A, b, c and x) using the inequalities (1) as constraints and objective function (2) and where $x \geq 0$.

$$\max c^T x \;\; s.t. \;\; Ax \leq b, \;\; x \geq 0$$

$$c = \begin{bmatrix} 1.5 \\ 2 \\ 2 \\ 1 \end{bmatrix}$$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & -2 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & -1 & -2 \\ 0 & 1 & 0 & 0.5 \end{bmatrix}$$

$$b = \begin{bmatrix} 1 \\ 7 \\ 9 \\ 13 \end{bmatrix}$$

b Is the linear program infeasible, unbounded or does it have a bounded optimum?

First we can check and see that $x = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ is a feasible solution, hence it is not infeasible.

By studying the constraints we can see that it is bounded. Since $x \geq 0$ we know the minimum feasible value. This due to the fact that the objective function grows when either x_i grows. The smallest value is then 0 with the point above.

x_1 and x_3 are strictly bound to be between $[0, 7]$. Same goes for x_2 and x_4 with $[0, 13]$ and $[0, 26]$ respectively. All values of x are bound in a limited interval, hence the problem is bounded.

c Formulate the dual program to the linear program from a).

We have the primal problem now:

$$\max c^T x \;\; s.t. \;\; Ax \leq b, \;\; x \geq 0$$

The dual problem is on the form:

$$\min y^T b \;\; s.t. \;\; y^T A \geq c, \;\; y \geq 0$$

d Is the dual from a) infeasible, unbounded or does it have a bounded optimum?

Since the primal is feasible and bounded that infers that the dual also is. Since they per definition have the same optimal solution.

e Is there any correlation between the solutions to the linear program from a) and its dual from c)?

Yes, as stated the optimal solution in both are the same as shown:

$$\begin{aligned} Ax &\leq b \\ c \leq yA & \\ cx \leq yAx &\leq yb \end{aligned}$$

As we can see, when $Ax = b$ or $yA = c$ we get that the other one must hold true too, both solutions are the same: $cx = yAx = yb$.

The to linear problems approaches the optimal solution from each "end", and will then meet in the "middle", at the solution.

Task 6

Let $G = (V, E)$ be a directed simple graph, where each edge $(u, v) \in E$ has an associated capacity $c(u, v) \geq 0$. Let two distinct nodes $s, t \in V$ be the source and sink, respectively. We are interested in solving the maximum-flow problem in this graph.

a Formulate a linear program that solves the maximum-flow problem in this graph. (*Hint: assume that* $c(u, v) = 0$ *when there does not exist an edge from* u *to* v.)

So we want a LP that solves the maximum-flow from s to t. First we then need to define an objective function. This would be the function to maximise. We have two choices here, we can define it as the flow leaving the source, s, or the flow entering the sink, t. The latter can be written on the form:

$$\sum f(e), \;\; e = (u, t) \in E \mid u, t \in V, \;\; f : flow \; of \; edge$$

This is the same as summing the last column in a neighbourhood matrix of the graph.

Then we need to have some constraints. These are quite easy to express, since we have a defined capacity function. Each edge, $e = (u, v)$, has a constraint of: $c(e) \geq 0$.

This would give us the A matrix:

$$A = \begin{bmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{bmatrix}$$

This would be equal to a diagonal matrix, with just ones, since each edge has their constraint alone.

This is because the x vector is a list of variables corresponding to all the edges. And a row in A will then choose one variable/edge to which we have the constraint in the first row of b.

$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_{|E|} \end{bmatrix}$$

The b vector would then be the actual capacity of the edges.

$$b = \begin{bmatrix} c(e_1) \\ \vdots \\ c(e_{|E|}) \end{bmatrix}$$

The A matrix would be $|E| \times |E|$, and b would be $|E| \times 1$. This would then be the same as: $x_i \leq c(e_i)$.

We are not done adding constraints yet though. We need to ensure that the flow conservation constraint is valid too. This means that the flow entering a vertex must equal the flow out of the vertex. This is not the case for the source and sink.

$$\sum_{i=0}^{|E|} f(i,v) = \sum_{i=0}^{|E|} f(v,i), \;\; v, \forall i \in V \backslash \{s,t\}$$

Adding this constraint to the A and b matrix is simple, we simply add new rows. To maintain the problem on standard form we must convert the $Ax = b$ constraints into: $Ax \leq b \wedge Ax \geq b \Rightarrow Ax \leq b \wedge -Ax \leq -b$. Adding to A will look like this:

$$A = \left[\begin{array}{ccc} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \\ \hline f_1(x_1) & \cdots & f_1(x_{|E|}) \\ -f_1(x_1) & \cdots & -f_1(x_{|E|}) \\ & \vdots & \\ f_{|V|}(x_1) & \cdots & f_{|V|}(x_{|E|}) \\ -f_{|V|}(x_1) & \cdots & -f_{|V|}(x_{|E|}) \end{array}\right]$$

Here we have added two new rows per vertex in the graph. The function $f_i(x_j)$ denotes the flow to vertex i from edge j, negative if the flow is going out of the vertex. Then the second row is negated to turn the equality constraint into an inequality constraint so we can add it to the A matrix in the standard form.

The b vector will then be equal to zero in the rest of the cases. The x vector will be unchanged. We can then write the problem on the standard form:

$$\max_x \; c^T x, \;\; Ax \leq b, \; x \geq 0$$

b Express the number of variables and constraints in your linear program using asymptotic notation, with regards to the number of nodes $|V|$ and edges $|E|$.

The number of variables is equal to the number of edges:

$$|x| = |E|$$

The number of constraints are equal to first the number of edges then the number of vertices times two, as described above:

$$A : (|E| + 2|V| \times |E|) \; matrix$$

c If your linear program from a) does not use $O(V+E)$ variables and constraints, formulate a new linear program for the maximum-flow problem that does so.

As we can see my program does do this.

$$O(|E| + 2|V|) \Rightarrow O(|E| + |V|)$$

For the number of variables I assume that you not expect it to use $O(|V| + |E|)$ and just $O(|E|)$ as I don't see the reason for more variables than edges in the graph.

Task 7

Let $G = (V, E)$ be an undirected graph.

a Construct an integer program that finds a maximum matching in G.

A integer program is a linear program with the added constraint of having a integer solution:

$$\max_x \ c^T x, \ \ s.t. \ Ax \leq b, \ x \geq 0, \ \ x \in \mathbb{Z}^n$$

Here we find the maximum by cardinality since it's not a weighted graph.

Here c would be a vector of length $|E|$ with just ones. Since we want to maximise the number of edges selected in the matching, the cardinality. x should be all the a representation of if an edge is in the matching or not, 1 or 0, hence length of $|E|$.

A should consist of each row corresponding to a vertex, and each column being an edge. If vertex i is connected to edge j then there is a 1, in cell (i, j), else 0. b should be the same as c, since each vertex can at most have one active edge connected to it.

$$\max_x \ c^T x, \ \ s.t. \ Ax \leq b, \ x \geq 0, \ \ x \in \{0, 1\}^n$$

b Now associate each edge e with a weight $w_e \geq 0$.

Construct an integer program that finds a maximum weighted matching in G, with regards to the weights.

Now we need to do a few adjustments to the program from the previous subsection.

First we need to change c so the objective function is correct. This can be done by having the weight of each edge in c instead of just ones. The constraints remain the same, since these just ensure that we end up with a maximal matching.

c In the following assume that G is bipartite.

If the coefficient matrix A of a linear program is total unimodular, there exists an optimal solution where the variables take whole number values, given that the values in b are integer. A matrix A is unimodular if the following is fulfilled for A or A^T:

1. **All elements in A have a value in $\{-1, 0, 1\}$**
2. **Each column contains at most two non-zero elements**
3. **We can divide the rows in the matrix into two sets using the following rules: if a column contains two non-zero elements with the same sign, the two rows belong to distinct sets. If a column contains two non-zero elements with different signs, then the rows belong to the same set.**

Modify your solution to b) such that you obtain a linear program with a total unimodular matrix A, that finds a maximum weighted matching.

My program is already on this form. If we look at how i defined A I stated that each column represents an edge. An, undirected, edge can be described as a set of two vertices. And I stated that if a vertex was

connected to the edge it would have a 1 in the column corresponding to it. Per definition there are exactly two vertices connected to each edge, meaning that there is exactly two ones in each column of A. This fulfils all three conditions for a total unimodular matrix.

d Formulate the dual of your program from c).

The dual is on the form (I have just transposed some values to get it on a different form from earlier):

$$\min_y b^T y \;\; s.t. \;\; A^T y \geq c, \; y \geq 0, \;\; y \in \mathbb{Z}^n$$

All the known variables are the same, the new variable, y, represents a vector with weight for each vertex. The constraints is that each vertex pair, from and edge, multiplied by y (the sum of the weights for the vertices), must be greater than or equal to the weight of that edge. Since b is just a one vector, the solution to minimise is the sum of the elements of y. I interpret this as minimising the "weight" of a vertex, the sum of all active weight of edges connected to it. However, the weight is constraint to be at least the weight of the edge in question. Since the edge with the highest weight connected to a vertex will satisfy all other edges connected to the same vertex, this will be the chosen one. This will then at the optimal solution be the maximum weighted matching.

e Find a valid, but not necessarily optimal, solution to your program from d). What can you say about the value of the objective function for this solution compared with an optimal solution to the primal?

As we don't have a graph I understand the task as to find a theoretical solution. Since y must always yield values greater than the edge weight, we can set all elements in y to the weight with the highest value. This would be a valid solution.

This value will obviously be greater than (or equal to) the optimal value of the primal solution. This is also obvious from the definition of the dual relationship:

$$c^T x \leq y^T A x \leq y^T b$$

Task 8

We are interested in finding a solution to the 0-1 (binary) knapsack problem. An instance of this problem consists of a set of n items, each with an associated weight w_i and a value v_i. We are given a knapsack of capacity W and wish to select a subset of items such that the combined weight of the items does not exceed W and the combined value of the items is maximized.

a Formulate an integer program that solves the binary knapsack problem.

The problem will have the form:

$$\max_x c^T x, \;\; s.t. \;\; Ax \leq b, \; x \geq 0, \;\; x \in \{0,1\}^n$$

Here x will be binary so the objective function will be c containing the values of the items. Then we will have A as a vector actually of $1 \times n$ size, containing the weight of each item. b will be a scalar value equal to W.

b Formulate the LP-relaxation of your integer program.

LP-relaxtation is a quite simple transformation. Here we simply allow x-values to be in the interval of the integer set from the integer program. This would then have the form:

$$\max_x c^T x, \;\; s.t. \;\; Ax \leq b, \; x \geq 0, \;\; x \in [0,1]^n$$

c Given that the integer program has an optimal solution OPT, the LP-relaxation has an optimal solution PRIM and the dual of the LP-relaxation (which we have not formulated) has an optimal solution DUAL. What is the relationship between the value of these optimal solutions?

In the case of a linear program the definition for the relation between the solutions of the primal and dual is:

$$c^T x \leq y^T Ax \leq y^T b$$

This is the same for a integer program, with the exception that they are not per definition equal at their optimum, unless the optimum is the same for the linear program.

This means that we have this relation:

$$OPT \leq PRIM \leq DUAL$$

d What complexity class does integer programming belong to? Explain your argumentation.

Integer programming is NP-hard. We turn the problem of linear programming into a discrete integer problem. This means that we can't easily balance weights and such with algebra, since we are only working with integers. If we have a linear program with constraint: $0 \leq x \leq 1$, this would be expressed as: $x \in \{0, 1\}$, in the integer program version. This is now a binary problem, and it can be shown to model logical formulas, which is the foundations of NP-completeness [2](page. 43).

Task 9

Use the matrix version of the Hungarian method to solve the assignment problem given by the following weight matrix.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{pmatrix} 7 & 11 & 10 & 8 & 9 \\ 5 & 2 & 1 & 19 & 4 \\ 3 & 13 & 7 & 8 & 10 \\ 12 & 11 & 4 & 1 & 4 \\ 0 & 7 & 9 & 3 & 11 \end{pmatrix} \end{array}$$

Step 1: Reduce rows; subtract the least value of each row from all entries in the given row.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{pmatrix} 0 & 4 & 3 & 1 & 2 \\ 4 & 1 & 0 & 18 & 3 \\ 0 & 10 & 4 & 5 & 7 \\ 11 & 10 & 3 & 0 & 3 \\ 0 & 7 & 9 & 3 & 11 \end{pmatrix} \end{array}$$

Step 2: Reduce columns; subtract the least value of each column from all entries in the given column.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{pmatrix} 0 & 3 & 3 & 1 & 0 \\ 4 & 0 & 0 & 18 & 1 \\ 0 & 9 & 4 & 5 & 5 \\ 11 & 9 & 3 & 0 & 1 \\ 0 & 6 & 9 & 3 & 9 \end{pmatrix} \end{array}$$

Step 3: Cover zeros; cover all zero entry rows and columns using the minimum number of lines.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} \text{a} & \text{b} & \text{c} & \text{d} & \text{e} \end{array} \\ \begin{pmatrix} \mathbf{0} & \mathbf{3} & \mathbf{3} & \mathbf{1} & \mathbf{0} \\ \mathbf{4} & \mathbf{0} & \mathbf{0} & \mathbf{18} & \mathbf{1} \\ \mathbf{0} & 9 & 4 & 5 & 5 \\ \mathbf{11} & \mathbf{9} & \mathbf{3} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & 6 & 9 & 3 & 9 \end{pmatrix} \end{array}$$

Step 4: If the number of lines is n (number of vertices on each side of the bipartite graph, 5 in our case) go to step 6. We have 4.

Step 5: Find the smallest uncovered value x. Subtract x from all uncovered elements and add x to elements in intersecting points of covering lines. Go to step 1.

x is 3.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} \text{a} & \text{b} & \text{c} & \text{d} & \text{e} \end{array} \\ \begin{pmatrix} 3 & 3 & 3 & 1 & 0 \\ 7 & 0 & 0 & 18 & 1 \\ 0 & 6 & 1 & 2 & 2 \\ 14 & 9 & 3 & 0 & 1 \\ 0 & 3 & 6 & 0 & 6 \end{pmatrix} \end{array}$$

Step 1:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} \text{a} & \text{b} & \text{c} & \text{d} & \text{e} \end{array} \\ \begin{pmatrix} 3 & 3 & 3 & 1 & 0 \\ 7 & 0 & 0 & 18 & 1 \\ 0 & 6 & 1 & 2 & 2 \\ 14 & 9 & 3 & 0 & 1 \\ 0 & 3 & 6 & 0 & 6 \end{pmatrix} \end{array}$$

Step 2:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} \text{a} & \text{b} & \text{c} & \text{d} & \text{e} \end{array} \\ \begin{pmatrix} 3 & 3 & 3 & 1 & 0 \\ 7 & 0 & 0 & 18 & 1 \\ 0 & 6 & 1 & 2 & 2 \\ 14 & 9 & 3 & 0 & 1 \\ 0 & 3 & 6 & 0 & 6 \end{pmatrix} \end{array}$$

Step 3:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} \text{a} & \text{b} & \text{c} & \text{d} & \text{e} \end{array} \\ \begin{pmatrix} \mathbf{3} & 3 & 3 & \mathbf{1} & \mathbf{0} \\ \mathbf{7} & \mathbf{0} & \mathbf{0} & \mathbf{18} & \mathbf{1} \\ \mathbf{0} & 6 & 1 & \mathbf{2} & \mathbf{2} \\ \mathbf{14} & 9 & 3 & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & 3 & 6 & \mathbf{0} & \mathbf{6} \end{pmatrix} \end{array}$$

Step 4: We have $4 < 5 = n$.

Step 5: $x = 1$

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} \text{a} & \text{b} & \text{c} & \text{d} & \text{e} \end{array} \\ \begin{pmatrix} 3 & 2 & 2 & 1 & 0 \\ 8 & 0 & 0 & 19 & 2 \\ 0 & 5 & 0 & 2 & 2 \\ 14 & 8 & 2 & 0 & 1 \\ 0 & 2 & 5 & 0 & 6 \end{pmatrix} \end{array}$$

We still have zeros in all rows and columns so we skip straight to step 3:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} \text{a} & \text{b} & \text{c} & \text{d} & \text{e} \end{array} \\ \begin{pmatrix} \mathbf{3} & 2 & \mathbf{2} & \mathbf{1} & \mathbf{0} \\ \mathbf{8} & \mathbf{0} & \mathbf{0} & \mathbf{19} & \mathbf{2} \\ \mathbf{0} & 5 & \mathbf{0} & \mathbf{2} & \mathbf{2} \\ \mathbf{14} & 8 & \mathbf{2} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & 2 & \mathbf{5} & \mathbf{0} & \mathbf{6} \end{pmatrix} \end{array}$$

Step 4: We have $5 = 5 = n$. So we go to step 6.

Step 6: Assignment; Select a row or column with only one zero and assign. If not found, select arbitrarily. Select other assignments so that no two tasks are assigned to same person.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} \text{a} & \text{b} & \text{c} & \text{d} & \text{e} \end{array} \\ \begin{pmatrix} 3 & 2 & 2 & 1 & \mathbf{0} \\ 8 & \mathbf{0} & 0 & 19 & 2 \\ 0 & 5 & \mathbf{0} & 2 & 2 \\ 14 & 8 & 2 & \mathbf{0} & 1 \\ \mathbf{0} & 2 & 5 & 0 & 6 \end{pmatrix} \end{array}$$

If we look at the assignments in the original matrix:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} \text{a} & \text{b} & \text{c} & \text{d} & \text{e} \end{array} \\ \begin{pmatrix} 7 & 11 & 10 & 8 & \mathbf{9} \\ 5 & \mathbf{2} & 1 & 19 & 4 \\ 3 & 13 & \mathbf{7} & 8 & 10 \\ 12 & 11 & 4 & \mathbf{1} & 4 \\ \mathbf{0} & 7 & 9 & 3 & 11 \end{pmatrix} \end{array}$$

This is equal to a total cost of: $0 + 2 + 7 + 1 + 9 = 19$. The matching is shown in the bipartite graph below, figure 25.

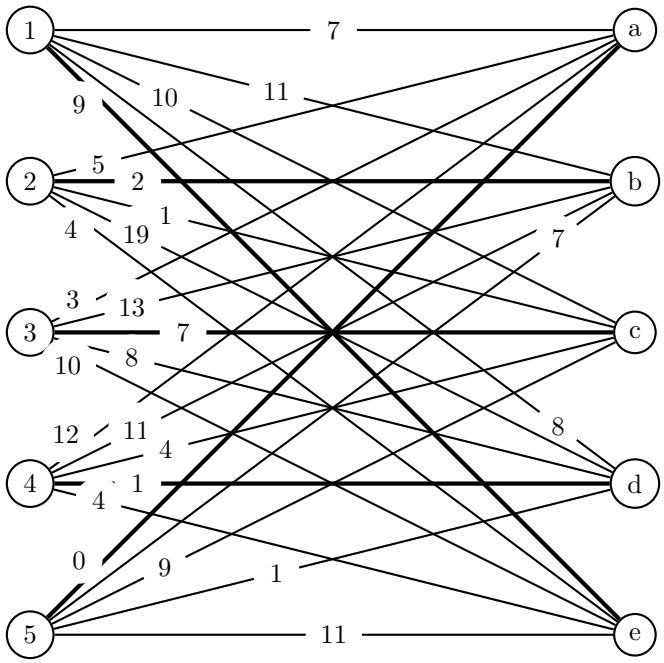


Figure 25: Matching of the weighted matrix.

Task 10

The rebels wish to gather intel on ship movement in the galaxy by infiltrating spaceports throughout the system. The rebels are already in possession of information about which pairs of planets ships are able to move directly between (not all pairs of planets are safe to travel directly between, while some pairs do not

make sense to travel directly between). To minimize the required personell, it has been decided that it is sufficient to for each pair of planets (p_i, p_j) which ships can travel directly between, infiltrate the spaceport of either p_i or p_j. That is, if E is the set of pairs of planets with direct travel between, the rebels wish to find a minimal subset $P' \subseteq P$ of planets such that for each $(p_i, p_j) \in E$ at least one of p_i and p_j is in P'.

Finding a minimal subset P' is NP-hard and the rebels are therefore looking for a good approximation algorithm to solve their problem.

a One possible algorithm is to pick one and one remaining planet at random, until the selected planets cover all possible direct travel routes. Will this algorithm yield an approximation guarantee? If so, what is the approximation factor?

So this problem can be represented as a graph with spaceports as vertices and travel routes as edges. What we then want is to find the vertex cover of the graph, such that all edges are connected to a vertex in the vertex cover set.

The random approach is not a very good method, but it will find a solution, but probably not the optimal one. The algorithm does have an approximation guarantee, but it's rather bad. Let's think of the worst scenario for the algorithm. If we have a center hub vertex connected to all other vertices, and we choose all but the center one. This means that we might have to pick all but one vertex before we find the solution. Hence the solution could have been a vertex set of one, but it became $|V| - 1$ instead.

$$\lim_{|V| \to \infty} \frac{2}{|V| - 1} = 0$$

We could end up with an approximation close to zero percent of the optimal solution, or if we flip it, infinitely larger.

As illustrated in figure 26, an optimal solution would be $P' = \{a\}$, but this algorithm could find $P' = \{b, c, d, e\}$.

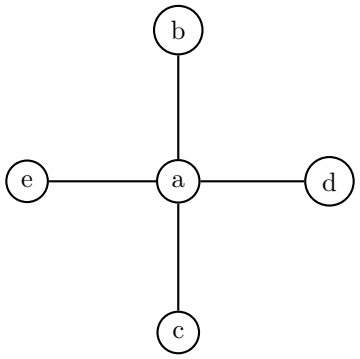


Figure 26: Illustration of a bad case for the random algorithm.

b A possible improvement of the algorithm is to for each planet selected, remove all covered travel routes and planets that are no longer connected to any direct travel route. Does this improve the approximation guarantee of the algorithm?

No, we can still show a fairly bad possibility. Referring the the figure 26, if a now was chosen, all connected vertices would be removed, and we would get the optimal solution. However, if any of the vertices connected to a is chosen instead, we only remove that vertex, since a is still bound in a pair with the rest. Then it's not hard to imagine a case where all vertices connected to a gets chosen in turn, as earlier, then when the last one is chosen, a will also be removed, and we will next choose one of the left vertices. This leads to the same worst case:

$$\lim_{|V| \to \infty} \frac{OPT}{APX} = \lim_{|V| \to \infty} \frac{2}{|V| - 1} = 0$$

c Another possible greedy algorithm is to instead pick uncovered direct travel routes at random. For each selected pair of planets (p_i, p_j), both p_i and p_j are added to the set. This process is repeated until all travel routes are covered. Will this algorithm yield an approximation guarantee? If so, what is the approximation factor?

I would assume that when (p_i, p_j) is selected, all edges connected to both p_i and p_j will be covered as well. Otherwise this would just pick all the edges in turn and all vertices would end up in the final set.

This will break my earlier example since a will immediately be added to the set and a almost optimal solution would be found. However, I have created a new worst case for these restrictions, as shown in figure 27. Imagine the earlier example, but we now add a new vertex to all vertices connected to a.

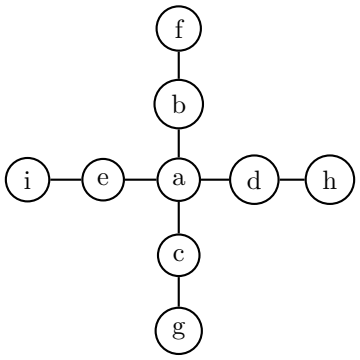


Figure 27: Illustration of a bad case for the random algorithm.

Now we can have the scenario where all outer vertices edges get picked. This would add the two vertices to the set and fulfil the pair constraint with a, but a still has a pair constraint with all the other ones. This would result in all "outer" edges needing to be picked before a would be satisfied.

The optimal solution is to just pick all edges connected to a. In the first case we do $\frac{|V|-1}{2}$ selections, where two new vertices are added to the set each time. In the other case we do the same amount of selection, only that a is added each time, hence we have almost half the set of vertices.

$$\lim_{|V|\to\infty} \frac{OPT}{APX} = \lim_{|V|\to\infty} \frac{\frac{|V|-1}{2}+1}{2\frac{|V|-1}{2}} = \frac{1}{2}$$

This gives us that an approximation guarantee of $\frac{1}{2}$ for this algorithm.

Task 11

The infiltration did not go as planned and the rebels are in retreat, wishing to lie low and avoid observation. In order to reduce their chance of discovery, they plan to store each of their spaceships at one of several hideouts. Rebel planners have created a scoring system that assigns to each of the hideouts $F = 1, 2, \ldots, m$, an *operation score* $f_i \geq 0$ associated with storing spaceships at the given hideout $i \in F$. The higher the score, the more likely discovery is at the given hideout. Likewise, the planners have assigned a discovery score c_{ij} to each pair of hideout $i \in F$ and spaceship $j \in D$, where $D = 1, 2, \ldots, n$. The higher the score, the more likely the spaceship is to be observed when transported from its current location to hideout i. The planners wish to find an assignment of spaceships to hideouts (each hideout can store the entire fleet of spaceships), such that the combined operation scores of the used hideouts, combined with the discovery scores of the spaceship-hideout assignments, is minimized. That is, they wish to find a subset of hideouts $F' \subseteq F$ that minimizes:

$$\sum_{i \in F'} f_i + \sum_{j \in D} \min_{i \in F} c_{ij}$$

The planners have correctly discovered that their problem is an NP-hard problem. However, they still wish to find an approximate solution to the problem, and require your help in doing so.

a Show that there exists a constant $c > 0$, such that unless $P = NP$, a $(cln|D|)$-approximation algorithm can not exist for the problem.
Hint 1: Can you use theorem 1.14 on page K114 in the compendium?
Hint 2: Can you make a reduction from the set cover problem?

"There exists some constant $c > 0$ such that if there exists a $(clnn)$-approximation algorithm for the unweighted set cover problem, then P = NP." [2](page. 114, theorem 1.14). To show that there exists a $c > 0$ such that a $(cln|D|)$-approximation algorithm can't exist, unless $P = NP$, we simply have to show that our problem can be reduced to the form of a unweighted set cover problem.

So the problem can be represented as a set cover problem of a bipartite graph, $G = (D, F, E')$. With the spaceships on the left, D. Each spaceship needs exactly one edge to the right side, the hideouts, F, but multiple spaceships can go to the same hideout. Instead of saying a spaceship needs one edge, we can say that a vertex, in D, is either in the vertex set cover or not. This will lead to that when all vertices in D are added, all edges will be covered and we have a vertex covered graph. This we can reduce to a set cover problem.

First, we observe that the ground set E is equal to all edges, E', in the graph $G = (D, F, E')$. Then we have that all subsets $\{S_1, S_2, \ldots, S_{|D|}\}$ of weights $\{w_1, w_2, \ldots, w_{|D|}\}$ is a set of all edges who are connected to the respective vertex in D. Now we have that for any vertex cover C, there is a set cover $I = C$ of the same weight, and hence we have reduced the problem.

b Find an $O(ln|D|)$-approximation algorithm for the problem.
Hint 1: Can you use a reduction to the set cover problem?
Hint 2: Have a look at the greedy algorithm for set cover on page K110 in the compendium.

Using the greedy algorithm [2](page. 111, Algorithm 1.2) for the set cover problem, and our reduced problem from above, we can solve the problem in $O(ln|D|)$. We simply have to define the $arg\ min$-function. This would be to choose the edge that connects the (i, j), $i \in D \wedge j \in F$ such that $w_{ij} + f_j$ is the smallest. If j already is in the set F', then $f_j = 0$, since it's not adding new cost after it already is in use.

References

[1] GeeksForGeeks. *Ford-Fulkerson Algorithm for Maximum Flow Problem*. URL: `https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/`. (accessed: 07.02.2023).

[2] NTNU. *Kompendium, Algoritmekonstruksjon*. NTNU, 2023.