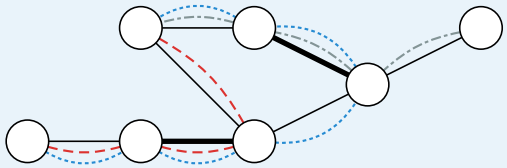
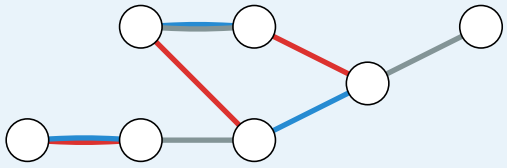
Task 1

a) The matching is maximal, as there is no way to add another edge to the matching without first removing one or more of the edges in the matching. However, it is not maximum, as there exists a matching in the graph with three edges.

b) There exist three different augmenting paths in G with regards to M, resulting in three different matchings M′. The figure below shows the three possible augmenting paths.



By find the symmetric difference between M and the augmenting path P, we get a new matching M′. The three possible matchings are highlighted in the figure below.



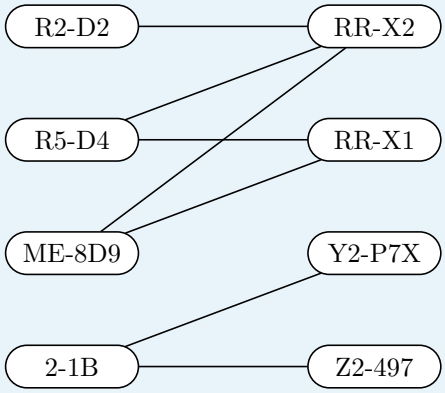
There exists several additional matchings of size 3. None of them can be a result of finding an augmenting path with regards to M *and finding* M′ *based on the symmetric difference of* M *and the augmenting path.*

c) The matching M′, no matter which augmenting path was used, is maximal and maximum. The matching is maximal, as there is no augmenting path in the graph. Thus, no larger matching can exist. A maximum matching is always maximal.

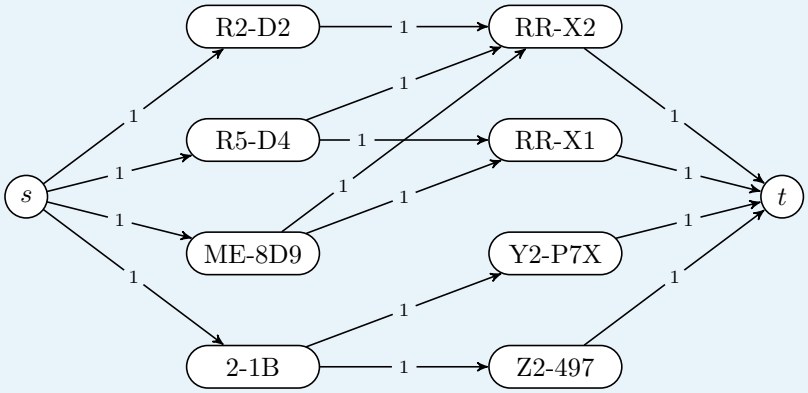
d) There cannot exist such G′, as V consists of an odd number of vertices ($|V| = 7$). Thus, it is impossible for all vertices to be matched, no matter which pairs of vertices are connected by edges.

Task 2

a) Since the droids and their compatible replacement circuits form a bipartite graph (see the figure below), we can use Hall's Marriage Theorem to determine if a perfect matching exists. We can see that there exists a subset of droids ($\{\text{R2-D2}, \text{ME-8D9}, \text{R5-D4}\}$) such that the set of compatible replacement circuits ($\{\text{RR-X1}, \text{RR-X2}\}$) is of lower cardinality than the set of droids. Thus, no perfect matching can exist by Hall's Marriage Theorem.



b) In order to find a maximal matching using flow, we modify the bipartite graph from earlier by adding a source and sink. In addition, each edge is assigned a capacity of 1.



Then, if a maximum integer flow is found in the flow network, the edges used between the robots and the circuits constitute a maximal matching. There exists several maximal matchings (of size 3):

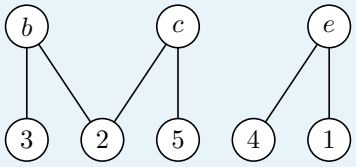
- R2-D2 – RR-X2, R5-D4 – RR-X1 and 2-1B – Y2-P7X
- R2-D2 – RR-X2, R5-D4 – RR-X1 and 2-1B – Z2-497
- R2-D2 – RR-X2, ME-8D9 – RR-X1 and 2-1B – Y2-P7X
- R2-D2 – RR-X2, ME-8D9 – RR-X1 and 2-1B – Z2-497
- R5-D4 – RR-X2, ME-8D9 – RR-X1 and 2-1B – Y2-P7X
- R5-D4 – RR-X2, ME-8D9 – RR-X1 and 2-1B – Z2-497
- ME-8D9 – RR-X2, R5-D4 – RR-X1 and 2-1B – Y2-P7X
- ME-8D9 – RR-X2, R5-D4 – RR-X1 and 2-1B – Z2-497

Task 3

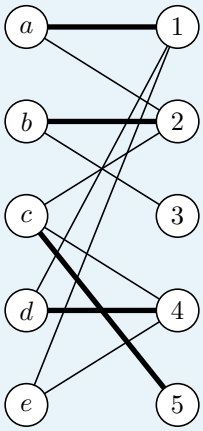
Depending on the order in which the modified DFS algorithm is ran from the different vertices, there are several different ways the algorithm can proceed. However, for any one of these ways, we end up having to perform two more iterations. In the first iteration, b and c are matched to a subset

of $\{2, 3, 5\}$, while e remains unmatched. In the second iteration, all vertices become matched. One possible outcome is the following.

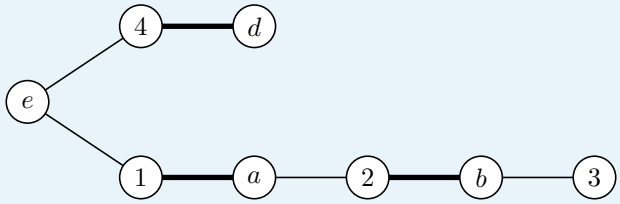
Iteration 1 Running the modified BFS algorithm we get the following trees:



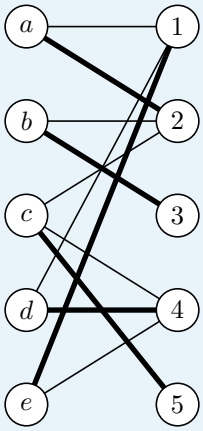
If we run the modified DFS algorithm from the vertices in $\{2, 3, 5\}$ in order, we may end up with finding augmenting paths $2 - b$ and $5 - c$. The new matching then becomes



Iteration 2 Running the modified BFS algorithm we get the following tree:



Running the modified DFS algorithm from 3, we end up finding the augmenting path $3 - b - 2 - a - 1 - e$. The final, maximum matching becomes:



Task 4

a) The algorithm will also work for arbitary graphs, as it only adds safe edges—edges without a neighbouring edge in the matching—to the matching. Hence, a valid matching is found.

The algorithm will also have the same, 1/2, approximation factor. To see why, assume that the algorithm finds the matching M and there is an optimal solution M^*. Both M and M^* are by design maximum matchings. We consequently have either $M = M^*$ or there exists non-empty subsets $E' = M \setminus M^*$ and $E^* = M^* \setminus M$. Let $w(\hat{E}) = \sum_{e \in \hat{E}} w_e$ for any $\hat{E} \subseteq E$. We wish to show that $w(E') \geqslant w(E^*)/2$, which would imply that $w(M) \geqslant w(M^*)/2$.

For each edge $e \in E^*$, there exists one or two edges in E' that share a vertex with e. If there was none, then M would not be maximum—e could be added to M. At least one of the edges in E' that conflict with e must have a weight of at least w_e. Otherwise, e would have been chosen by the algorithm before the conflicting edges. Since E^* is a subset of a matching (E^* is also a matching), each edge $e' \in E'$ conflicts with at most two edges in E^* and can thus count as the neighbouring edge with at least the same weight for at most two edges in E^*. Consequently, each edge in E' must offset the weight of at most two edges of lower or equal weight in E^*. We have that $w(E^*) \leqslant 2w(E')$, which is equivalent to $w(E') \geqslant w(E^*)/2$.

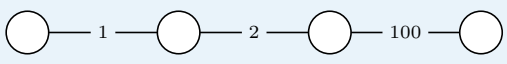
b) By modifying the proof from a), we can see that if two edges in E^* share the same neigbouring edge $e \in E'$ with weight w_e, then the weight of these two edges cannot exceed $cw_e + c^2 w_e$. Thus, $w(E') \geqslant w(E^*)/(c + c^2)$, which provides the approximation factor (for $c = 1$, as earlier, this is equivalent to 1/2). For $c = 3/4$, we get an approximation factor of 16/21 and for $c = 1/2$ we get 4/3. In other words, the algorithm finds an optimal solution—a maximal weight matching—if $c = 1/2$.

c) From b) we know that $1/(c + c^2)$ decides the relationship between $w(E')$ and $w(E^*)$. As long as this relationship is equal to or greater than 1, we are guaranteed that algorithm always finds an optimal solution. If we let $c + c^2 = 1$, we get that $c = (\sqrt{5} - 1)/2 = \Phi \approx 0.618$ (Φ is the golden ratio conjugate).

d) Assume that Min-Weight-Maximal-Matching has returned a non-maximal matching M. Then there exists an edge $e \in E$ that can be included in the matching. In other words, M does not contain any of e's neighbouring edges. Since the algorithm only removes edges from E' if they are adjacent to an edge in M, e will not have been removed from E' and the algorithm could not have stopped. Therefore, there can be no such $e \in E$, and the matching found is maximal.

e) The algorithm works in the same way as the one Erciyes introduced, except for the inverse ordering of E. This change in ordering does not change the runtime of the algorithm, which remains $O(E \lg E)$.

f) In this case, we cannot guarantee a constant approximation factor. This is a result of requiring a maximal matching. If the first few edges are selected in an unfortunate manner, the algorithm can be forced to select an extremely heavy edge. The graph below is an example of this situation. Here, the optimal solution is to select the edge with weight 2. However, the algorithm first selects the edge with weight 1. Then, it is forced to select the edge with a weight of 100. Thus, the algorithm finds a 50.5-approximate solution. By increasing the weight of the rightmost edge, we can force the algorithm to find arbitrarily bad solutions.



Task 5

a) To formulate a linear program in standard form, we need all the inequalities to be written in the same form. That is, we need them to be of the form:

$$a_{i1}x_1 + a_{i2}x_2 + a_{i3}x_3 + a_{i4}x_4 \leqslant b_i \ , \tag{1}$$

where a_{ij} is the element in A on row i and in column j. Two of the inequalites are already on this form, while the other two are not. If we rewrite them, we get

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

From this, we can find both A and **b**.

$$\mathrm{A} = \begin{bmatrix} 1 & -2 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & -1 & -2 \\ 0 & 1 & 0 & 0.5 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 1 \\ 7 \\ 9 \\ 13 \end{bmatrix} \tag{3}$$

($x_1 - 2x_2 \leqslant 1$)

Further, we can find **c** from the objective function.

$$\mathbf{c} = \begin{bmatrix} 1.5 \\ 2 \\ 2 \\ 1 \end{bmatrix} \tag{4}$$

With A, **b** and **c**, we can construct the linear program in standard form.

(Objective function: $\mathbf{c}^T\mathbf{x}$; Inequalities: $\mathrm{A}\mathbf{x} \leqslant \mathbf{b}$)

$$\text{maximize} \quad \mathbf{c}^T\mathbf{x} \tag{5}$$

$$\text{when} \quad \mathrm{A}\mathbf{x} \leqslant \mathbf{b} \tag{6}$$

$$\mathbf{x} \geqslant \mathbf{0} \tag{7}$$

b) We can easily see that there exists solutions to the linear program, and it can therefore not be infeasible. One possible solution is:

$$\mathbf{x} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{8}$$

We can also see that the second inequality limits both x_1 and x_3 to be maximally 7. In a similar fashion, the fourth inequality limits both x_2 and x_4 to be maximally 13 and 26,

respectively. Since all the x_is are limited in value, the linear program cannot be unbounded and must have a finite optimum. If we solve the linear program, using, e.g., a solver, we find that the optimum is 40. A possible optimal solution is:

$$\mathbf{x} = \begin{bmatrix} 0 \\ 10 \\ 7 \\ 16 \end{bmatrix} \tag{9}$$

(As long as $x_1 = 0, x_3 = 7$, every solution with $0 \leqslant x_2 \leqslant 13$ and $x_4 = 26 - x_2$ is optimal.)

c) Since we have the linear program in standard form, it is simple to find the dual. The dual uses both A, $\mathbf{b}$ and $\mathbf{c}$, and is:

$$\begin{aligned} \text{minimize} \quad & \mathbf{b}^T\mathbf{y} && (10) \\ \text{when} \quad & \mathrm{A}^T\mathbf{y} \geqslant \mathbf{c} && (11) \\ & \mathbf{y} \geqslant \mathbf{0} && (12) \end{aligned}$$

Where

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} \tag{13}$$

d) It follows from duality that the dual of the earlier linear program must have a finite optimum, as the linear program had a finite optimum.

e) The duality theorem describes the relationship between a linear program and its dual. Either one of them is infeasible and the other is unbounded, both are infeasible or both have a finite optimum. In the last case, we know that both the linear program and its dual has the same finite dual. Thus, without solving the dual, we know that it has an optimum of 40.

Task 6

a) Let f_{uv} be a variable that indicates the flow over the directed edge from u to v and assume that $c(u, v) = 0$ when $(u, v) \notin \mathrm{E}$. We wish to make sure that the flow over an edge is non-negative and does not exceed the capacity of the edge. We therefore introduce the following restrictions:

$$\begin{aligned} f_{uv} &\leqslant c(u, v) \quad \forall u, v \in \mathrm{V} && (14) \\ f_{uv} &\geqslant 0 \quad \forall u, v \in \mathrm{V} && (15) \end{aligned}$$

Additionally, for each vertex that is not the source nor the sink, the amount of flow into the vertex must be equal to the flow out of the vertex. Thus we must add the following restrictions:[a]

$$\sum_{v \in \mathrm{V}} f_{uv} - \sum_{v \in \mathrm{V}} f_{vu} = 0 \quad \forall u \in \mathrm{V} \setminus \{s, t\} \tag{16}$$

In the max-flow problem we wish to maximize the flow from the source to the sink. We can express this as an objective function by maximizing either the net flow out of the source or into the sink. That is, we wish to

$$\text{maximize} \quad \sum_{v \in \mathrm{V}} f_{sv} - \sum_{v \in \mathrm{V}} f_{vs} \tag{17}$$

Combined, we get a linear program for maximum flow.

b) There exist two variables for each pair of vertices, f_{uv} and f_{vu}, and there are therefore $\Theta(\mathrm{V}^2)$ number of variables. The number of restrictions (inequalities) is also $\Theta(\mathrm{V}^2)$, as there are two per variable, in addition to one per vertex (except s and t).

c) Our linear program from **a)** had two variables for each pair of vertices. When there is no edge from a vertex u to v, it is unnecessary to have a variable to indicate the flow from u to v. The flow is always 0. Thus, we wish to limit variables to situations where they are necessary.[b] We can then express our linear program in the following way:

$$\text{maximize} \quad \sum_{v \in \mathrm{V} \mid (s,v) \in \mathrm{E}} f_{sv} - \sum_{v \in \mathrm{V} \mid (v,s) \in \mathrm{E}} f_{vs} \tag{18}$$

$$\text{when} \quad \sum_{v \in \mathrm{V} \mid (u,v) \in \mathrm{E}} f_{uv} - \sum_{v \in \mathrm{V} \mid (v,u) \in \mathrm{E}} f_{vu} = 0 \qquad \forall u \in \mathrm{V} \setminus \{s,t\} \tag{19}$$

$$f_{uv} \leqslant c(u,v) \quad \forall (u,v) \in \mathrm{E} \tag{20}$$

$$f_{uv} \geqslant 0 \quad \forall (u,v) \in \mathrm{E} \tag{21}$$

Since we have one variable per edge, the number of variables is $\Theta(\mathrm{E}) = \mathrm{O}(\mathrm{V} + \mathrm{E})$. We have one restriction per variable (except s and t) and two per edge. Thus, we have $\Theta(\mathrm{V} + \mathrm{E})$ restrictions.

[a] We can express an equality as two inequalities, where one says the left side should be no smaller than the right side and the other says that the left side should be no larger than the right side.

[b] This reduces the complexity of our linear program.

Task 7

a) Let m_e be a binary variable which is 1 if and only if e is part of the matching.

$$\text{maximize} \quad \sum_{e \in \mathrm{E}} m_e \tag{22}$$

$$\text{when} \quad \sum_{e \in \delta(v)} m_e \leqslant 1 \quad v \in \mathrm{V} \tag{23}$$

Edges connected to v

$$m_e \in \{0,1\} \quad e \in \mathrm{E} \tag{24}$$

b) We need only change the integer program from **a)** slightly, adding to the objective function the weight of each edge.

$$\text{maximize} \quad \sum_{e \in \mathrm{E}} w_e \cdot m_e \tag{25}$$

$$\text{when} \quad \sum_{e \in \delta(v)} m_e \leqslant 1 \quad v \in \mathrm{V} \tag{26}$$

$$m_e \in \{0, 1\} \quad e \in \mathrm{E} \tag{27}$$

Edges connected to v

Change from ILP to LP

c) The linear program becomes

$$\text{maximize} \quad \sum_{e \in \mathrm{E}} w_e \cdot m_e \tag{28}$$

$$\text{when} \quad \sum_{e \in \delta(v)} m_e \leqslant 1 \quad v \in \mathrm{V} \tag{29}$$

$$m_e \geqslant 0 \quad e \in \mathrm{E} \tag{30}$$

We must now check that all the requirements for total unimodularity are fulfilled. Notice that each row corresponds to a vertex, while each column corresponds to an edge. Thus, it follows trivially that all elements in A are 0 or 1. Further, each m_e appears only once per restriction, and in a total of two restrictions (one for each of the two vertices connected by e). Thus, we have exactly two ones per column and the remainder of the column is zeros. Since each column has two elements with an equal sign, we need to be able to place the two involved rows into distinct sets. Since the graph is bipartite, it follows from the properties of bipartite graphs, that the vertices (rows) can be partitioned into two distinct sets with edges only going between the two sets. Thus, we fulfill all three requirements for total unimodularity.

d) Let e_l and e_r denote, respectively, the vertex that belongs to the left and right subset of vertices for a given edge e. Then the dual can be formulate as

$$\text{minimize} \quad \sum_{v \in \mathrm{V}} y_v \tag{31}$$

$$\text{when} \quad y_{e_l} + y_{e_r} \geqslant w_e \quad e \in \mathrm{E} \tag{32}$$

$$y_v \geqslant 0 \quad v \in \mathrm{V} \tag{33}$$

e) A possible solution is

$$y_v = \max_{e \in \delta(v)} w_e \quad v \in \mathrm{L} \tag{34}$$

$$y_v = 0 \quad v \in \mathrm{R} \tag{35}$$

As a result of weak duality, we have that $\sum_{v \in \mathrm{V}} y_v \geqslant \sum_{e \in \mathrm{E}} w_e \cdot m'_e$, where m'_e is an optimal solution for the primal.

Task 8

a) We wish to express that an item is either selected or not. Let the column vector $\mathbf{x}$ consist of $x_1, x_2, \ldots, x_n$, where x_i indicates if item i is placed in the knapsack ($x_i = 1$) or not ($x_i = 0$). We have the following restrictions to make sure that only whole items are used:

$$x_i \in \{0, 1\} \quad \forall i \in \{1, 2, \ldots, n\} \tag{36}$$

We introduce the following restriction to make sure that the weight in the knapsack does not exceed W:

$$\overbrace{\sum_{i=1}^{n} w_i x_i}^{\mathbf{w}^T\mathbf{x}} \leqslant \mathrm{W} \tag{37}$$

In the binary knapsack problem, we wish to select items that maximize the value in the knapsack. That is, we get the following objective function:

$$\text{maximize} \quad \overbrace{\sum_{i=1}^{n} v_i x_i}^{\mathbf{v}^T\mathbf{x}} \tag{38}$$

b) An LP-relaxation is very similar to the original integer program. The only difference is that we can no longer restrict variables to be integers. That is, we must replace $x_i \in \{0, 1\}$:

$$\begin{aligned} \text{maximize} \quad & \mathbf{v}^T\mathbf{x} && (39)\\ \text{when} \quad & \mathbf{w}^T\mathbf{x} \leqslant \mathrm{W} && (40)\\ & \mathbf{x} \leqslant \mathbf{1} && (41)\\ & \mathbf{x} \geqslant \mathbf{0} && (42) \end{aligned}$$

Since this is an LP-relaxation, a solution to this linear program will not necessarily work as a solution to the original integer program.

Note: this linear program solves the fractional knapsack problem, where we can place arbitrary fractions of items into the knapsack.

c) The optimal solution of an integer program is always a solution to the LP-relaxation. Therefore, the optimal solution of the integer program has no larger value for the objective function than the optimal solution of the LP-relaxation. Further, due to the duality theorem, the solution to a linear program is always equal (by the objective function) to the solution of its dual, as long as a solution exists and is not unbounded. Thus, we have that

$$\mathrm{OPT} \leqslant \mathrm{PRIM} = \mathrm{DUAL} \tag{43}$$

d) We have seen that we can formulate an integer program for the binary knapsack problem in polynomial time. Since we know that the binary knapsack problem is NP-hard, integer programming must also be NP-hard.

Integer programming is not a decision problem, and can therefore not be in NP. However, we can construct a decision problem based on integer programming, by asking if there exists a solution that provides a value greater than or equal to y. This decision problem is in NP, as we can verify a solution in polynomial time. Furthermore, since the decision problem is NP-hard, it is also NP-complete.

Task 9

We are given the following weight matrix

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

After performing the first step of the algorithm, i.e., the reduction of the rows, we end up with the following matrix

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

After performing the second step of the algorithm, i.e., the reduction of the columns, we end up with the following matrix

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

Zeroes can optimally be covered by using four lines, covering rows 1 and 2, and columns a and d (there are other ways to use four lines to cover the zeros). We thus perform step five and get the following matrix

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

Going back to step one, we see that neither step one nor step two results in any changes, as each row and column contains a zero. However, we can still (optimally) use four lines to cover the zeroes. Row 2 is covered, along with columns a, d and e (we could also cover rows 1, 2 and 4, along with

column a). Performing step five again, we get the following matrix

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 2 & 0 \\ 7 & 0 & 0 & 21 & 3 \\ 0 & 6 & 1 & 5 & 4 \\ 11 & 6 & 0 & 0 & 0 \\ 0 & 3 & 6 & 3 & 8 \end{pmatrix} \end{array}$$

Again, step one and two do not result in any changes, while we can cover all zeros (optimally) with four lines. That is, we cover column a along with rows 1, 2 and 4 (we could also use columns a and e and rows 2 and 4). Then, after performing step five we get the following matrix

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{pmatrix} 2 & 1 & 1 & 2 & 0 \\ 8 & 0 & 0 & 21 & 3 \\ 0 & 5 & 0 & 4 & 3 \\ 12 & 6 & 0 & 0 & 0 \\ 0 & 2 & 5 & 2 & 7 \end{pmatrix} \end{array}$$

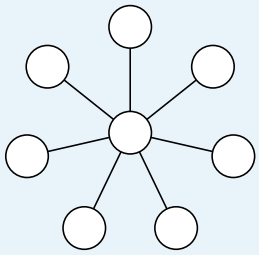
Again, step one and two do not result in any changes. We now need five lines to cover all the zeros. To check, start by covering the zero in the top right corner. Since there are no other zeros in row 1, it is optimal to place the line covering the top right corner as a vertical line covering column e. Similarly, we get the same argument for the bottom left corner. Now, the zero in the middle (intersection of column c and row 3) is the only uncovered zero on row 3 and it is optimal to cover it by placing a line on column c. The same argument goes for the two remaining uncovered zeros in rows 2 and 4.

We can now perform step 6 to find an assignment. We get the assignment: $1 - e$, $2 - b$, $3 - c$, $4 - d$ and $5 - a$.

Task 10

This problem is equivalent to the vertex cover problem, if we model the planets as vertices and direct travel routes as edges.

a) The algorithm can end up picking all but one of the planets before having covered all the direct travel routes, even in situations where a single planet would suffice. Consider the situation in which the planets are arranged in a star pattern (equivalent to the star graph, S_k), with one planet connected to all the others by direct travel routes. There are no other travel routes.



The central planet forms an optimal solution. However, the algorithm can end up picking all

the outer planets. Then our solution will consist of 7 planets, a seven-fold increase from the optimal solution. Moreover, for any k, the layout S_k results in a possible k-fold increase from the optimal solution. Thus, the algorithm does not have a constant-factor approximation guarantee.

b) The suggest change does not fix the problem described prior. Thus, it does not result in the algorithm gaining a constant-factor approximation guarantee.

c) The proposed algorithm will find a 2-approximate solution. For each direct travel route, an optimal solution contains at least one of the two planets connected by the route. Thus, for each uncovered direct travel route, we add to the set two planets, of which at least one needs to be in the solution. That is, the number of unnecessary planets added is no larger than the number of needed planets. Therefore, the solution found does not consist of more than double the number of planets of an optimal solution.

Task 11

This problem is equivalent to the uncapacitated facility location problem.

a) We will show that there exists a constant $c > 0$, such that unless P = NP, there cannot exist a $(c \ln |D|)$-approximation algorithm. Our approach is to show that if the statement is false, it implies that Theorem 1.14 (K114 in the compendium) is false. As Theorem 1.14 has been proven true, we can then conclude that our statement holds true. To show the similarity of the statements, we need to perform the following steps:

(i) Find a reduction from the set cover problem.

(ii) Show that an optimal solution of the reduction is also optimal for the instance of the set cover problem, with equal cost/score.

(iii) Show that any solution to the reduced instance can be turned into a valid solution for the instance of the set cover problem, without increasing the cost/score.

If we can perform all three steps, we know that if we can find an approximation algorithm for our problem, then there exists an approximation algorithm for the set cover problem that is at least as good. Theorem 1.14 states that there exists a constant $c > 0$ such that unless P = NP, there does not exist a $(c \ln n)$-approximation for the set cover problem. Thus, we cannot (if the three steps are performed) have a $(c \ln |D|)$-approximation algorithm for our problem.[a]

(i) An instance of the set cover problem consists of a set of elements $E = \{1, 2, \ldots, n\}$ and a collection of sets $S = \{S_1, S_2, \ldots, S_m\}$, such that $S_i \subseteq E$ and $\cup_{S_i \in S} S_i = E$. Associated with each $S_i \in S$, there is a cost w_i. We wish to find a subset $S' \subseteq S$ such that $\cup_{S_i \in S'} S_i = E$, while $\sum_{S_i \in S'} w_i$ is minimized.

From an instance of the set cover problem, we can construct an instance of our problem by creating hideouts $F = \{1, 2, \ldots, m\}$, one for each set in S. The operation score of hideout i is set to the cost of set i, $f_i = w_i$. That is, using hideout i is equivalent to $S_i \in S'$. Further, we let the set of spaceships $D = \{1, 2, \ldots, n\}$ consist of one spaceship for each element in E. Thus, we are interested in assigning each spaceship $j \in D$ to a hideout $i \in F$ such that $j \in S_i$. In order to achieve such an assignment, we will set the cost of assigning $j \in D$ to $i \in F$ in a way so that if $j \notin S_i$, the cost is higher than using hideout i' where $j \in S_{i'}$. Let $c_{\max} = \max_{i \in F} f_i$. Then, we let the discovery scores of $j \in D$ be

$$c_{ij} = \begin{cases} 0 & \text{if } j \in \mathrm{S}_i \\ c_{\max} + 1 & \text{otherwise} \end{cases}$$

Always better to open a new hideout

Then we have a way of reducing an instance of the set cover problem to one for our problem.

(ii) To show that an optimal solution to the reduction is an optimal solution to the set cover instance, we note that an optimal solution $\mathrm{F}^* \subseteq \mathrm{F}$ must for each $j \in \mathrm{D}$ contain an $i \in \mathrm{F}^*$ such that $j \in \mathrm{S}_i$. Otherwise, we can select any $i \in (\mathrm{F} \setminus \mathrm{F}^*)$, with $j \in \mathrm{S}_i$. Adding i to F^* reduces the score, as the discovery score of j is $c_{\max} + 1$ prior to adding i. After adding i, the discovery score of j is 0, while the operation score of i is $f_i < c_{\max} + 1$. Thus, an optimal solution to the reduction is a subset $\mathrm{S}' \subseteq \mathrm{S}$, such that $\cup_{\mathrm{S}_i \in \mathrm{S}'} \mathrm{S}_i = \mathrm{E}$. Further, since the discovery score of each $j \in \mathrm{D}$ is 0, the score of F^* is equivalent to the cost of S'. If there exists a subset $\mathrm{S}^* \subseteq \mathrm{S}$ ($\cup_{\mathrm{S}_i \in \mathrm{S}^*} \mathrm{S}_i = \mathrm{E}$) with lower cost than S', it follows that F^* is not an optimal solution. Thus, S' is an optimal solution to the instance of the set cover problem and the cost of S' is equal to the score of F^*.

(iii) Let $\mathrm{F}' \subseteq \mathrm{F}$ be a solution to the reduced instance. Then, we need to show that we can use F' to construct a solution S' to the instance of the set cover problem, such that the cost of S' is no higher than the score of F'. Note that we cannot use F' directly as a solution, since there can exist $j \in \mathrm{D}$ with a discovery score different from 0. If this is the case, then there would be no set $\mathrm{S}_i \in \mathrm{S}'$ with $j \in \mathrm{S}_i$. However, if the discovery score of j is not 0, it is $c_{\max} + 1$. Thus, we can improve F' by finding $i \in \mathrm{F}$ such that $j \in \mathrm{S}_i$ ($f_i < c_{\max} + 1$). We can therefore in polynomial time construct F'' such that for $j \in \mathrm{D}$, there exists $i \in \mathrm{F}''$ where $j \in \mathrm{S}_i$. Furthermore, the score of F'' is no higher than the score of F'. Let $\mathrm{S}' = \{\mathrm{S}_i : i \in \mathrm{F}''\}$. It holds that $\cup_{\mathrm{S}_i \in \mathrm{S}'} \mathrm{S}_i = \mathrm{E}$ and the cost of S' is no higher than the score of F'.

b) We will use the greedy ($\mathrm{H}_n = \mathrm{O}(\ln n)$)-approximation algorithm for the set cover problem, by constructing a reduction from our problem to the set cover problem.

We can construct a reduction by letting each spaceship be a separate element, i.e., $\mathrm{E} = \mathrm{D}$. Then, for each hideout $i \in \mathrm{F}$, we add to S one set $\mathrm{S}_i^{\mathrm{T}} = \mathrm{T}$ for each non-empty subset $\mathrm{T} \subseteq \mathrm{E}$. The cost w_i^{T} of $\mathrm{S}_i^{\mathrm{T}}$ is set to

$$w_i^{\mathrm{T}} = f_i + \sum_{j \in \mathrm{T}} c_{ij} \ .$$

That is, the score of using hideout i and assigning all spaceships $j \in \mathrm{T}$ to hideout i.

For any solution $\mathrm{S}' \subseteq \mathrm{S}$ of the reduced instance, we can construct a solution $\mathrm{F}' \subseteq \mathrm{F}$ to the original instance, with a score no higher than the cost of S'. Simply let $\mathrm{F}' = \{i : \mathrm{S}_i^{\mathrm{T}} \in \mathrm{S}'\}$. Similarly, we can from any solution $\mathrm{F}' \subseteq \mathrm{F}$ to the original instance, create a solution $\mathrm{S}' \subseteq \mathrm{S}$ to the reduced instance by letting $\mathrm{S}' = \{\mathrm{S}_i^{\mathrm{T}_i} : i \in \mathrm{F}'\}$, such that T_i is the set of spaceships assigned to hideout i. The cost of S' is no higher than the score of F'. Consequently, the two instances have the same optimal solution, with the same cost/score.

Unfortunately, our reduction is not polynomial, due to the way that S is constructed. We must therefore use a trick in order to use the suggested ($\mathrm{H}_n = \mathrm{O}(\lg n)$)-approximation algorithm. Notice that when the algorithm greedily selects a set to add to S, it selects the set S_i which minimizes $w_i / |\hat{\mathrm{S}}_i|$, where $\hat{\mathrm{S}}_i$ are the non-covered elements in S_i. We can exploit this property, by noticing that for a given $i \in \mathrm{F}$, at any point when the set of remaining, uncovered spaceships is D', then the sets $\mathrm{S}_i^{\mathrm{T}}$ that can minimize $w_i / |\hat{\mathrm{S}}_i^{\mathrm{T}}|$ will for each $r \in \{1, 2 \ldots, |\mathrm{D}'|\}$ be a single set $\mathrm{S}_i^{\mathrm{T}_r}$ such that $\mathrm{T}_r \subseteq \mathrm{D}'$, $|\mathrm{T}_r| = r$ and T_r consists of the r remaining spaceships $j \in \mathrm{D}'$ with the smallest discovery score c_{ij} for hideout i. Thus, for each of the $|\mathrm{D}|$ or fewer iteration

of the algorithm, we can in polynomial time ($O(FD \lg D)$ time) construct the $O(FD)$ sets S_i^T that need to be considered. Thus, we need only construct a polynomial number of sets and we can use the approximation algorithm from the set cover problem.

[a] Given that $|D| = n$ in the reduction.