# Algorithms and Data Structures - Coursework 2

s1725018

November 2020

## 1 Question 1

Consider the problem of taking a set of n items with sizes $s_1, ..., s_n$ , and values $v_1, ..., v_n$ respectively. We assume $s_i, v_i \in$ for all $1 \leq i \leq n$. Suppose we are also given a capacity $C \in N$.

The packing problem is the problem of finding a subset $S \subseteq \{1, ..., n\}$ such that $\sum_{i \in S} s_i \leq C$ and such that $\sum_{i \in S} v_i$ is maximized subject to the first constraint.

We write $P_{n,C}$ to denote the value $\sum_{i \in S} v_i$ of the maximum-value packing on the set of all items. For any $k \leq n$, and any $\widehat{C} \leq C, \widehat{C} \in N$, we can consider the same problem on the first $k$ items in regard to capacity $\widehat{C}$. We denote the maximum-value packing for such a sub-problem by $P_{n,\widehat{C}}$.

The goal is to develop a $\Theta(nC)$ dynamic programming algorithm to compute the optimal packing with respect to the original $n$ items and capacity $C$.

### 1.1 Question 1a

Prove a suitable recurrence for $P_{k,\widehat{C}}$ that holds for all $k \leq n$ and $\widehat{C} \leq C$.

### 1.2 Answer 1a

I will begin by proof by showing the different cases on the $k$ item. We know that $k$ is the number of the first $k$ items that we add to bag given a capacity of $\widehat{C}$ and that $P_{k,\widehat{C}}$ is the maximum value of these first $k$ items. We also note that $\widehat{C}$ is the amount of capacity left when we take the combination of the first $k$ items $S_k \subseteq \{1, ..., k\}$ that maximise the value $\sum_{i \in S_k} v_i$.

Now that we have defined the problem we can now separate it into different cases and prove it on a case by case analysis. Since we are taking a dynamic programming approach to the problem, we solve this problem in a bottom up fashion. Our first (1.) case we can identify is when $k = 0$, that is what is the value of $P_{k,\widehat{C}}$ when there are no items. The second case (2.) would be when $k \neq 0$ but the size of the $k$th item $s_k > \widehat{C}$ and finally the last case (3.) would be when $k \neq 0$ but the size of the $k$th item $s_k \leq \widehat{C}$. These cases are exhaustive and cover all the possible outcomes for $k$ and $\widehat{C}$ this is because it boils down to if we can take item $k$ or not given the reaming capacity $\widehat{C}$ for the $k-1$th items and if it is indeed better to take $k$ or not.

1. $k = 0$ then $P_{k,\widehat{C}} = 0$
    - Proof: By definition we know that $P_{k,\widehat{C}} = \sum_{i \in S_k} v_i$ maximum-value packing on the set of $k$ items where $S_k \subseteq \{1, ..., k\}$ is the subset of items these $k$ items.

      However since $k = 0$ then this subset $S_k = \varnothing$ since we have no items we can take. Note that this does not depend on the capacity $\widehat{C}$ because for any capacity up to $C$ since we have not taken any items.

Hence since $S_k = \varnothing$ when we evaluate the maximum-value packing on the set of $k$ items $P_{k,\widehat{C}} = \sum_{i \in S_k} v_i = \sum_{i \in \varnothing} v_i = 0$ since there are no items so no values to add. Therefore when $k = 0$ then $P_{k,\widehat{C}} = 0$. ■

2. $k \neq 0$ and $s_k > \widehat{C}$ then $P_{k,\widehat{C}} = P_{k-1,\widehat{C}}$

   - Proof: We know that we are now looking at the subset $S_k \subseteq \{1, ..., k\}$ which is not empty since $k \neq 0$. That means we are trying to add the $k$th item, however the size $s_k > \widehat{C}$ for the current $k-1$ items this means that we cannot add the $k$th item because we do not have enough capacity for it and as a result since we have not added the item. So $P_{k,\widehat{C}}$ has the same value as the $P_{k-1,\widehat{C}}$. Since now we look down the recurrence for a smaller subset, in particular one with $k-1$ items. Therefore $P_{k,\widehat{C}} = P_{k-1,\widehat{C}}$. ■

3. $k \neq 0$ and $s_k \leq \widehat{C}$ then $P_{k,\widehat{C}} = max\{P_{k-1,\widehat{C}}, P_{k-1,\widehat{C}-s_k} + v_k\}$

   - Proof: Now we assume we can take the $k$th item since its capacity is less than the capacity that remains. However we want to maximise the values we are taking so we ask ourselves, what is better, taking the $k$th item and reducing the capacity so we have less capacity to add more items but we have the value of the $k$th item or is it better to leave the $k$th item behind and have more capacity to take more of the $k-1$ items that can result in a higher value if we decide not to take it.

     As a result we can check is the value of $P_{k-1,\widehat{C}} > P_{k-1,\widehat{C}-s_k} + v_k$ this means that the value of the $k-1$ items with the remaining capacity $\widehat{C}$ is greater than taking the $k$th item, losing $s_k$ capacity and gaining $v_k$ value. If this is the case we would prefer not to take the $k$th item.

     Likewise if $P_{k-1,\widehat{C}} \leq P_{k-1,\widehat{C}-s_k} + v_k$ the contrary is true we would prefer to take item $k$ since when we take it the value for the remaining $k-1$ items including $k$ that can fit in the remaining capacity $\widehat{C} - s_k$ will still be greater despite loosing some capacity. So we would take item $k$.

     Since our problem is to maximise the value of the items we take subject to the constraint in this case the capacity $C$, at every iteration we would like to find the maximum of the two cases from above. That is we would like to always get the highest value in every case when we are deciding to take or not to take the $k$th item. And hence this is why we check for $max$.

     So the two cases illustrated above tells us how to update $P_{k,\widehat{C}}$ according to if we take the $k$th item or not. If we take the item then we have to subtract $s_k$ from $\widehat{C}$ and add the value of $k$ namely, $v_k$ and then look at the $k-1$ remaining items. As a result we get $P_{k,\widehat{C}} = P_{k-1,\widehat{C}-s_k} + v_k$. Likewise if we don't take the $k$th item $P_{k,\widehat{C}} = P_{k-1,\widehat{C}}$ since we dont add any value and we look again at the reaming $k-1$ items.

     Therefore, putting it all together we get an expression for $P_{k,\widehat{C}}$ when $k \neq 0$ and $s_k \leq \widehat{C}$. That is $P_{k,\widehat{C}} = max\{P_{k-1,\widehat{C}}, P_{k-1,\widehat{C}-s_k} + v_k\}$ since we want to maximise the value of the items we take.■

As a result by 1, 2 and 3 we have found an expression for the recurrence for our problem equal to

$$P_{k,\widehat{C}} = \begin{cases} 0, & \text{if } k = 0 \\ P_{k-1,\widehat{C}}, & \text{if } k \neq 0 \text{ and } s_k > \widehat{C} \\ max\{P_{k-1,\widehat{C}}, P_{k-1,\widehat{C}-s_k} + v_k\} & \text{otherwise} \end{cases}$$

## 1.3 Question 1b

Use your recurrence above to develop a $\Theta(nC)$ dynamic programming algorithm to compute the optimal packing with respect to the original $n$ items and capacity $C$. Formally justify the $\theta(nC)$ run-time of your algorithm.

## 1.4 Answer 1b

I have designed an algorithm based on the recurrence that was discovered in question 1a. The output of the algorithm is optimal packing value which in our case will be the maximum value of the $n$ items you can take given the capacity $C$. The **Inputs** of the algorithm are as described below but follow the same convention of that of the question.

- $n$ is the list of items.
- $C$ is the capacity, an integer.
- $s$ is the list of item of sizes $\{s_1, ..., s_n\}$.
- $v$ is the list of item of values $\{v_1, ..., v_n\}$

**Algorithm 1:** Optimal Packing DP

```
   Input: $n, C, s, v$
   Output: optimal packing value
1  $n_{length} = n.length$;
2  initialize a new 2-D array $P[0, ..., n_{length}][0, ..., C]$;
3  for $i = 0$ to $C$ do
4      $P[0][i] = 0$;
5  for $i = 1$ to $n_{length}$ do
6      for $\widehat{c} = 0$ to $C$ do
7          if $s[i] \leq \widehat{c}$ then
8              if $P[i-1][\widehat{c}] > P[i-1][\widehat{c} - s[i]] + v[i]$ then
9                  $P[i][\widehat{c}] = P[i-1][\widehat{c}]$;
10             else
11                 $P[i][\widehat{c}] = P[i-1][\widehat{c} - s[i]] + v[i]$;
12         else
13             $P[i][\widehat{c}] = P[i-1][\widehat{c}]$;
14 return $P[n_{length}][C]$
```

A couple of things to note. Firstly, we have constructed the 2-D array $P$ in the algorithm in such away that it holds the values of $P_{k,\widehat{C}}$ $\forall k, 0 \leq k \leq n$ and $\forall \widehat{C}, 0 \leq \widehat{C} \leq C$. Therefore, the output of the algorithm will be the value of the 2-D array at $P[n_{length}][C]$ this is because this is the entry which takes into account all the items and the capacity $C$ given which is equal to the value $P_{n,C}$.

Secondly is that I am using the notation "*.length*" to get the value of the length of an array. I use it to get the length of the array $n$.In this case "*n.length*" will be equal to the total number of items in the problem.

Firstly I will explain how the algorithm works and then I will prove the complexity of it. We first look at the first case when $k = 0$ of the recurrence defined above, this is done in the loop in lines **3-4** which populates the 2-D array $P$. This is the case when there are no items and so it is the same as indexing the $P[0][i]$ where $i$ are equal to the capacities up to $C$ and by the first case of the recurrence we know that $P[0][i] = 0$ $\forall i, 0 \leq i \leq C$. Now we have an outer and inner loop this is done to iterate over all the remaining entries in our 2-D array $P$ (in the algorithm), we populate in a bottom up fashion starting from $i = 1$ up to $n_{length}$ to index all the items in the list and for the capacities we start from $\widehat{c} = 0$ to $C$ to cover all the capacities. Line **7** is the first important check which checks if the capacity of item $i$ is less than the current capacity $\widehat{c}$ if this is not the case then we we jump to line **13** which corresponds to the case in the recurrence where $k \neq 0$ and $s_k > \widehat{C}$ and as a result we let $P[i][\widehat{c}] = P[i-1][\widehat{c}]$ just like in the recurrence. Now the lines **8-11** are used to find the "*max*" between $P_{k-1,\widehat{C}}$ and $P_{k-1,\widehat{C}-s_k} + v_k$. That is what the **if** statement in line **8** is used for to decide which value is greater and then in line **9-12** it updates the 2-D array accordingly. So if $P_{k-1,\widehat{C}}$ is greater then we go to line **9** and let $P[i][\widehat{c}] = P[i-1][\widehat{c}]$. Likewise if $P_{k-1,\widehat{C}-s_k} + v_k$ is greater than $P_{k-1,\widehat{C}}$ then we go to line **11** and let $P[i][\widehat{c}] = P[i-1][\widehat{c} - s[i]] + v[i]$. Finally line **14** will return the value $P[n_{length}][C]$ which explained before will be $P_{n,C}$ due to the construction of the 2-D array as it will hold the value of the maximum value over all the items $n$ and as a result we have found the maximal value for the given problem.

I will now formally justify the run-time of the **Algorithm 1**. First I will start by showing that the upper-bound of the run time of this algorithm is $O(nC)$.

Firstly we see that lines **1-2** take time $O(1)$ to run since "*.length*" takes $O(1)$ to get the length of the given array and to initialize a 2-D array requires $O(1)$ time since we are not yet populating it. The first **for** loop in line **3** runs a total of $C$ times and the contents of the for loop run in constant time since we are just setting the values of $P[0][i]$ to 0. Hence the run time for the first **for** loop is $O(C)O(1) = O(C)$.

Now we get to line **5** which is the outer for loop that runs all from line **5-13**. This loop runs a total of $n.length - 1$ times which is equal to $n-1$ times if we let $n$ be the number of items in the list. As a result the loop runs in $O(n)$ time since $n-1$ is still $O(n)$.

The inner for loop in line **6** runs from 0 up to $C$. This means that it is executed a total of $C$ times, as a result the run time for this loop is $O(C)$. We can see that the run times for lines **8-13** are all $O(1)$ this is because they preform operation such as comparisons or storing a value in the 2-D array $P$ which all require constant time to do. As a result the run time of the inner loop is $O(C)O(1) = O(C)$.

We can now derive the run time for the outer loop. Since we know it runs in $O(n)$ and the contents of this loop run in $O(C)$ time (lines **6-13**). As a result the run time of the **for** loop on line **5** is $O(C)O(n) = O(nC)$.

The final line, line **14** takes $O(1)$ time to run since we are just returning the value of the optimal packing.

So if we put all this together we get: lines **1-2, 14** run time is $O(1)$. The **for** loop on lines **3-4** runs in $O(n)$ and the bigger **for** loop from lines **5-13** runs in $O(nC)$. Hence the total run time for our algorithm is $3 * O(1) + O(n) + O(nC) = O(1) + O(n) + O(nC) = O(n) + O(nC) = O(nC)$. Therefore, the upper bound on the run time of **Algorithm 1** is $O(nC)$.

Now I will show that the lower-bound of the run time of this algorithm is $\Omega(nC)$. All the constant time lines (**1-2, 4, 7-14**) will all take $\Omega(1)$. As a result we can see that for any sized inputs $n$ and $C$ the **for** loop on lines **3-4** will still run in $\Omega(n)$. Likewise the **for** loop on lines **5-13** will also run in $\Omega(nC)$. This is because of any sized inputs the **for** loop in line **5** will always run $n-1$ times and as a result is $\Omega(n)$ similarly the loop in line **6** will always run $C$ times so $\Omega(C)$. As a result we get that the lower bound on the run time is $\Omega(nC)$ since when we put all of this together $\Omega(nC)$ will be the dominating term.

As a result, We can see that since the run time of the algorithm is both $O(nC)$ and $\Omega(nC)$ it follows that the run time of **Algorithm 1** is $\Theta(nC)$. ■

What is important to note is that **Algorithm 1** only gives us the maximum value of the packing, but it does not tell us which items have picked to maximise the value of the packing.

We can now change **Algorithm 1** so that it takes into account which items it has taken in. We do this by creating a new 2-D array $In$. This 2-D array is of the same dimensions as $P$. This 2-D array $In$ keeps track of which items we have decide to take, it takes binary values 0 or 1. Therefore if we decide to take item $i$ with when we have capacity $\widehat{c}$ we let $In[i][\widehat{c}] = 1$ (line **15** of **Algorithm 2**) otherwise we let it equal to 0 this is because in all the other cases we do not take item $i$. **Algorithm 2** is the implementation of this change. This algorithm takes in the same parameters as **Algorithm 1** but returns the additional 2-D array $In$ which is then used to print out the index of the items used to solve the problem in **Algorithm 3**. What is most important is that by adding this new 2-D array $In$ and populating, this does not increase the run time of the algorithm. This is because we populate the 2-D array $In$ exactly at the same points when we populate the 2-D array $P$ in **Algorithm 1**. Since all these computations only require $\Theta(1)$ this does not change the the run time of the loop form lines **7-18** and so as a result the run time of this loop will still be $\Theta(nC)$ and so we get that the run time of **Algorithm 2** will be $\Theta(nC)$.

**Algorithm 2:** Optimal Packing DP with Item track

```
Input: $n, C, s, v$
Output: optimal packing value, $In$
1  $n_{length} = n.length$;
2  initialize a new 2-D array $P[0, ..., n_{length}][0, ..., C]$;
3  initialize a new 2-D array $In[0, ..., n_{length}][0, ..., C]$;
4  for $i = 0$ to $C$ do
5      $P[0][i] = 0$;
6      $In[0][i] = 0$;
7  for $i = 0$ to $n_{length}$ do
8      for $\widehat{c} = 0$ to $C$ do
9          if $s[i] \leq \widehat{c}$ then
10             if $P[i-1][\widehat{c}] > P[i-1][\widehat{c} - s[i]] + v[i]$ then
11                 $P[i][\widehat{c}] = P[i-1][\widehat{c}]$;
12                 $In[i][\widehat{c}] = 0$;
13             else
14                 $P[i][\widehat{c}] = P[i-1][\widehat{c} - s[i]] + v[i]$;
15                 $In[i][\widehat{c}] = 1$;
16         else
17             $P[i][\widehat{c}] = P[i-1][\widehat{c}]$;
18             $In[i][\widehat{c}] = 0$;
19 return $P[n_{length}][C]$ and $In$
```

Once we have the 2-D array $In$ we can navigate through it to print out the items that where added to the optimal packing returned by **Algorithm 2** by using **Algorithm 3**. The logic behind the algorithm is below.

1. If $In[i][C] = 1$
   - We know that we have added item $i$ so we now move to $In[i-1][C - s[i]]$, so we look at the remaining $i - 1$ items with capacity $C - s[i]$, which is what is left over after we have taken into account the size of the item.
2. $In[i][C] = 0$
   - Then we know we didn't select the item so as a result we just look at $In[i-1][C]$ since item $i$ has not been added, so we just look at the remaining $i - 1$ items with capacity $C$ .

**Algorithm 3:** Print the items used for Optimal Packing, when given the 2-D array $In$

```
Input: $n, C, s, In$
Output: $null$
1  $n_{length} = n.length$;
2  for $i = n_{length}$ down to $0$ do
3      if $In[i][C] = 1$ then
4          $print(\text{“}i\text{”})$;
5          $C = C - s[i]$;
```

# 2 Question 2

King Arthur has problems to administer his realm. His court contains $n$ knights and he rules over $m$ counties. The knights differ in their abilities and local popularity: Each knight $i$ can oversee at most $q_i$ counties, and each county $j$ will revolt unless it is overseen by some knight in a given subset $S_j \subseteq \{1, ..., n\}$ of the knights. Only 1 knight can oversee a county, to prevent conflicts between the knights. The king discusses the problem with his court magician Merlin.

## 2.1 Question 2a

Show how Merlin can use the Max-Flow algorithm to efficiently compute an assignment of the counties to the knights that prevents a revolt, provided that one exists. How can he determine whether the algorithm was successful?

Prove the complexity of this algorithm, in terms of some function $F(v, e)$, where $F(v, e)$ denotes the running time of a Max-Flow algorithm on a graph with $v$ vertices and $e$ edges.

## 2.2 Answer 2a

To show how Merlin can use the Max-Flow algorithm to efficiently one first has to transform the problem of knights and counties to a network flow graph so he can run a Max-Flow algorithm. This problem is a variation of the maximum bipartite matching problem. The sets of knights and counties can be illustrated a bipartite graph $G$. A bipartite is a graph whose vertices can be divided into two independent subsets $L$ and $R$ such that every edge $(u, v)$ connects a vertice from $L$ to one in $R$. For every edge $(u, v)$ we have that $u \subseteq L$ and $v \subseteq R$. In our case we let the set $L = \{c_1, ..., c_m\}$ be the set of the $m$ counties and let the set $R = \{k_1, ..., k_n\}$ be the set of the $k$ knights. We have defined the vertices of the bipartite graph $G$, now I will explain what the edges that connect the two sets of vertices are. Since we know that each county $j$ will revolt unless it is overseen by some knight in a given subset $S_j \subseteq \{1, ..., n\}$ of the knights, we can create an edge that connects each county vertice $c_j$ in $L$ to each knight vertice in $R$ that is in the given subset $S_j$.

Therefore, the bipartite graph $G$ is defined below in terms of the set of vertices $V$ and edges $E$:

$$V = \{\{(c_j) : 1 \leq j \leq m\} \cup \{(k_i) : 1 \leq i \leq n\}\} \tag{1}$$

$$E = \{(c_j, k_i) : 1 \leq j \leq m, k_i \subseteq S_j\} \tag{2}$$

Where $m$ is the number of counties, $n$ is the number of knights and $S_j$ is the subset of knights that can oversee county $c_j$. As a result we get that the number of vertices in $G$ is $|V| = m + n$ this is because we create a vertice for every knight and county in the problem. Likewise the number of edges in graph $G$ is $|E| = \sum_{j \in m} |S_j|$ this is because for each vertex county $c_j$ in $L$ we connect it to exactly $|S_j|$ elements in $R$, so we just sum over all the counties and get the expression.

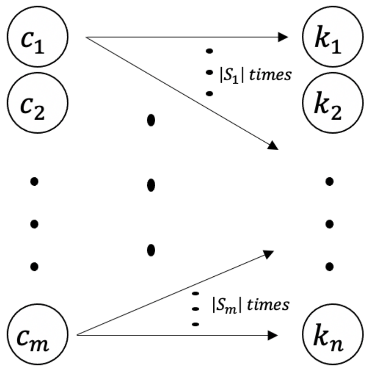

Figure 1: This shows the construction for the bipartite graph $G$

However we are still not done we need to augment graph $G$ by adding new vertices and edges and also setting the capacities on each of the edges in this new graph $G'$ so Merlin can run a Max-Flow algorithm through it. We start by adding two new vertices, namely '$s$' and '$t$' these will correspond to the "*source*" and "*target*" vertices respectively. We connect the '$s$' with every vertice in the set $L$ corresponding to the set of counties $c_j$. Likewise we connect each vertice in the set $R$ with the newly created vertex '$t$'. So our new set of vertices $V'$ and edges $E'$ in $G'$ are defined below.

$$V' = \{\{(c_j) : 1 \leq j \leq m\} \cup \{(k_i) : 1 \leq i \leq n\} \ \cup \{s\} \cup \{t\}\} \tag{3}$$

$$E' = \{\{(c_j, k_i) : 1 \leq j \leq m \text{ and } k_i \subseteq S_j\} \cup \{(s, c_j) : 1 \leq j \leq m\} \cup \{(k_i, t) : 1 \leq i \leq n\}\} \tag{4}$$

Like before $m$ is the number of counties, $n$ is the number of knights and $S_j$ is the subset of knights that can oversee county $c_j$. As a result we get that the number of vertices in $G'$ is $|V'| = m + n + 2$ this is because we now include the two new vertices namely, '$s$' and '$t$'. Likewise the number of edges in graph $G'$ is $|E'| = \sum_{j \in m} |S_j| + m + n$ this is because in addition the original $\sum_{j \in m} |S_j|$ edges we had we have now created $m$ new edges from the source $s$ to each county $c_j$, likewise we have created $n$ new edges from each knight $k_i$ to the target vertice $t$.

I will now define the capacity function $c$ of $G'$ this is because we need to know the capacity of each edge in the graph in order to be able to run a max-flow algorithm through it.

$$c(u, v) = \begin{cases} 1, & \text{for edge } (s, c_j) : \forall j \text{ where } 1 \leq j \leq m \\ 1, & \text{for edge } (c_j, k_i) : 1 \leq j \leq m \text{ and } i \subseteq S_j \\ q_i, & \text{for edge } (k_i, t) : \forall i \text{ where } 1 \leq j \leq n \\ 0 & \text{otherwise} \end{cases}$$

Again $m$ is the number of counties, $n$ is the number of knights and $S_j$ is the subset of knights that can oversee county $c_j$. I will now explain the reason behind each of the capacities. We note that all the capacities of all the edges $(s, c_j) = 1$ this is because we can only have one knight allocated to one county this means that the flow coming in to county $c_j$ must be equal to 1. Likewise the capacity for all the edges $(c_j, k_i)$ will be equal to 1, this is because we can picture this as the matching from the set $L$ to the set $R$ and as a result each edge will have capacity 1 which will so the maximum flow through the graph will correspond to the cardinality of the maximum matching of counties to knights which I will prove shortly. The most interesting capacities are the ones for the edges $(k_i, t)$ they all have capacity $q_i$ which is the maximum number of counties that knight $k_i$ can oversee. We set the capacity of these edges to this value so the knight $k_i$ can oversee at most $q_i$ due to Flow Conservation.

This is the subtle difference in our problem, with respect to the original maximal bipartite matching, is that is that an edge from the set $L$ to $R$ can share a vertex in $R$. This is because a knight $k_i$ can oversee at most $q_i$ counties, this means that me can allocate various counties $c_i$ in $L$ the set of counties to one knight in $R$ that has capacity $q_i$.

What is also important to note is that $c(c_j, k_i) = 0$ for all the edges in which the county $c_j$ cannot be over seen by the knight $k_i$, which means that the knight $k_i$ is not in the set $S_j$.

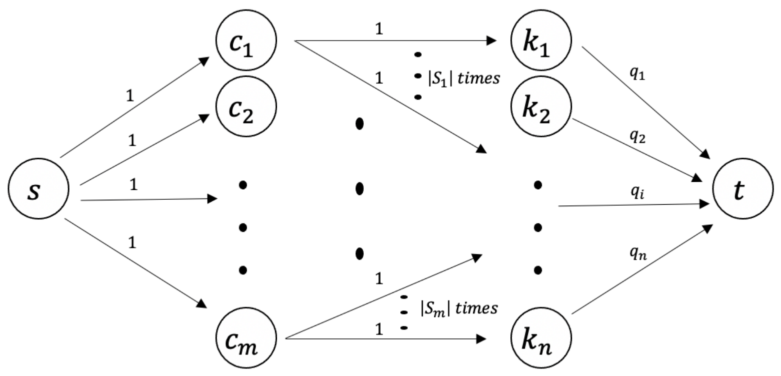

Figure 2: This shows the construction for the network graph $G'$ with the new vertices added ($s$ and $t$) and the capacities of each edge using the capacity function $c$.

Now that we have defined the graph $G'$ and the capacity function $c$ we can show the following claims: that each county $c_j$ will be allocated at most one knight that is in the subset $S_j$ of knights that can oversee county $c_j$ (1), that each knight $k_i$ will be allocated at most $q_i$ counties (2) by the construction of the capacity function $c$ and graph $G'$. Finally I will show that the maximum flow of our graph $G'$ is equal to the maximal matching of the original graph $G = m$ where $m$ is the number of counties (3). This means that if Merlin can use a max-flow algorithm to find a max flow of the graph then he has subsequently found the allocation, that Arthur can use, of counties $m$ to knights $n$.

1. Each county $c_j$ will be allocated at most one knight that is in the subset $S_j$ of knights that can oversee county $c_j$.

   - Proof: To see that each county will only be allocated to at most one knight, we can use properties of network flows. In particular Capacity Constraint and Flow Conservation.

     By Capacity Constraint we know that for all $u, v \in V'$ the flow of each edge will be:

     $$0 \leq f(u,v) \leq c(u,v) \tag{5}$$

     This means that the flow of any edge will be at most the capacity of that edge.

     Now we focus on the flow passing through the county $c_i$. we know by Flow Conservation that:

     $$\sum_{v \in V'} f(v, c_i) = \sum_{v \in V'} f(c_i, v) \tag{6}$$

     However since we know by construction of the graph $G'$ that $c(v, c_i) = 0$, $\forall v \in V'/\{s\}$ therefore $f(v, c_i) = 0$, $\forall v \in V'/\{s\}$ otherwise $c(v, c_i) = 1$ and by the Capacity Constraint we know that $f(s, c_i) \leq c(s, c_i) = 1$ so if we put this together with Flow Conservation we get that:

     $$\sum_{v \in V'} f(v, c_i) = f(s, c_i) \leq c(s, c_i) = 1 \tag{7}$$

     Which means that:

     $$\sum_{v \in V'} f(v, c_i) \leq 1 \tag{8}$$

     Now we look at the r.h.s of the equation, $\sum_{v \in V'} f(c_i, v)$, and we see that $c(c_i, v) = 0$, $\forall v \notin S_i$ where $S_i$ is the set of knights that $c_i$ can be allocated this is by the way we created the edges that leave vertice $c_i$ and as a result we know that:

     $$\sum_{v \in V'} f(c_i, v) = \sum_{v \in S_j} f(c_i, v) \tag{9}$$

Where $S_j$ is the set of knights that $c_j$ can be allocated to. Putting (8) and (9) together and using the definition of Flow Conservation we get that:

$$\sum_{v \in V'} f(c_i, v) = \sum_{v \in V'} f(c_i, v) \iff \sum_{v \in S_j} f(c_i, v) \leq 1 \tag{10}$$

Which means that the sum of the flow out of the vertice $c_i$ is at most 1 and it exactly goes to the knights that are in the set $S_i$ which correspond to the knights that can be allocated to county $c_i$. So we get that each county is allocated to at most one knight that can oversee it. ■

2. Each knight $k_i$ will be allocated at most $q_i$ counties.

   - Proof: We now want to show that each knight is allocated at most $q_i$ counties, we can do this in a similar fashion to 1. by using Capacity Constraint and Flow Conservation. Again we know that for any knight $k_i$ by Flow Conservation that:

   $$\sum_{v \in V'} f(v, k_i) = \sum_{v \in V'} f(k_i, v) \tag{11}$$

   First by looking at the r.h.s of equation (11) we know that $c(k_i, v) = 0$, $\forall v \in V'/\{t\}$ therefore $f(k_i, v) = 0$, $\forall v \in V'/\{t\}$ otherwise $c(k_i, t) = q_i$ and by the Capacity Constraint we know that $f(k_i, t) \leq c(k_i, t) = q_i$ so if we put this together with Flow Conservation we get that:

   $$\sum_{v \in V'} f(k_i, v) = f(k_i, t) \leq c(k_i, t) = q_i \tag{12}$$

   $$\sum_{v \in V'} f(k_i, v) \leq q_i \tag{13}$$

   Now observing the l.h.s, we know that $c(c_i, k_j) = 1$, $\forall j \in S_i$ where $S_i$ is the set of knights that an over see $c_i$, otherwise $c(c_i, k_j) = 0$ if county $c_j$ cannot be allocated to knight $k_j$. Therefore we know that:

   $$\sum_{v \in V'} f(v, k_i) = \sum_{c_i \in CK_i} f(c_i, k_i) \tag{14}$$

   Where $CK_i$ is the set of counties that can be allocated to knight $k_i$. So now if we put equations (13) and (14) together with the definition of Flow Conservation we get that:

   $$\sum_{v \in V'} f(v, k_i) = \sum_{v \in V'} f(k_i, v) \iff \sum_{c_i \in CK_i} f(c_i, k_i) \leq q_i \tag{15}$$

   Which means that the maximum flow entering vertice $k_i$ is at most $q_i$ and it exactly comes from a set $CK_i$ of counties that can be allocated to $k_i$. ■

3. The maximum flow of our graph $G'$ is equal to the maximal matching of the original graph $G$ where the cardinalty of the maximum matching is equal to $m$ where it is the number of counties in our case.

   - Proof: To prove this we need to prove the following: (A) **Integrity theorem**: If each edge in a flow network has integral capacity, then there exists an integral maximal flow. (A) Since we are using the Ford-Fulkerson algorithm we know that we can use a proof by induction on the $n$th iterations of the while loop of the algorithm. After the first iteration since $c$ only takes on integer values for all the edges in our graph and the flow $f$ is set to 0, we have that $c_p$ only takes integer values. Hence for the loop in the Ford-Fulkerson algorithm we assume that for the $n$th iteration $c_p$ has only taken integer values since the capacity only contain integers, so they can only ever be integers. Now we show it is true for $n+1$th iteration. Since the algorithm only increases the value of the flow by 1 the flow at the $n+1$th iteration will also be integral value and so as a result the flow will be integral. ■

Now I will show that if M is a matching in our graph $G$ then there is an integral-valued flow $f$ in $G'$ with the value that $|f| = |M|$. Conversely, if $f$ is an integral-valued flow in $G'$, then there is a matching $M$ in $G$ with cardinality $|M| = |f|$.

We will first show that if M is a matching in our graph $G$ then there is an integral-valued flow $f$ in $G'$ with the value that $|f| = |M|$. We know that each edge in the original bipartite graph in which we have a match for corresponds to 1 unit of flow between in $G'$ for the path that goes through the vertices $s \to c_i \to k_i \to t$. As a result we can make a cut $(L \cup \{s\}, R \cup \{t\})$ with value being equal to $|M|$ because it is the number of vertices from $L$ that go to on in $R$, in our case counties to knights which will correspond to $|f|$ since this is the definition of the net flow through a cut and we know that the flow will always go to the set $R$ by construction of $G'$. As a result $|f| = |M|$.

Now we will show that if we have a flow in which is integral valued in $G'$, then there is a matching $M$ in $G$ with cardinality $|M| = |f|$. We can construct a set of points $M$ for which we know that there is a flow through it. Let $M = \{(c_j, k_i) : \text{where } f(c_j, k_i) > 0\}$ this means that there is a flow going form vertice $c_j$ to vertice $k_i$. Note that for each vertice $c_j$ it has exactly one unit of flow entering it since the capacity $c(s, c_j) = 1$ we know by Flow Conservation it must leave, but by construction we know that all $c(c_j, k_j) = 1$ therefore $c_j$ will go to exactly one vertice $k_j$. However what is important to note is that different counties can be allocated to the same knight because $c(k_i, t) = q_i$. As a result this means that $M$ corresponds to a matching of counties to knights. And this will correspond to the flow $f$ since if we cut the graph at $(L \cup \{s\}, R \cup \{t\})$ we get that $|f| = \sum_{c_j, k_i \in V'} f(c_j, k_i)$ which is equal to the matching in $M$. Therefore $|M| = |f|$. ■

We are now left to show that if $f$ is a maximum flow in graph $G'$ then it corresponds to the cardinality of maximum matching in the bipartite graph $G$.

Suppose that $m$ is the maximum matching in $G$ and that the corresponding flow $f$ in $G'$ is not maximum. Then there exists a maximum flow $|f'| > |f|$. Since the capacities of the graph are integer-valued then we know that $f'$ is also integer valued by **Integrity theorem**. So we know that $f'$ corresponds to a matching in $m'$ in $G$ with cardinality $|m'| = |f'| > |f| = |m|$, which contradicts that $m$ is a maximum matching.

Likewise if we assume that there is a maximum flow $f$ in $G'$ with not a maximum matching $m$ we know that, we can find a matching where $|m| < |m'|$, and as a result we know that we can find a flow such that $|f'| = |m'|$ by the previous proof. Therefore we have found a new flow $f'$ which is greater than $f$ which means that we have found a contradiction. As a result we have found that the cardinality of the maximum flow $m$ corresponds to the value of the maximum flow $f$ in $G'$. ■

As a result we have shown in 3. that if Merlin finds a maximum flow on graph $G'$ it will correspond to the maximum matching of cardinality $m$ which is the number of counties in the original bipartite graph $G$. Therefore we have found an allocation of counties to knights that Arthur can use in order to prevent a revolt. To answer the question: How can he determine whether the algorithm was successful? Merlin knows that the algorithm is successful if the value of the max flow in graph $G'$ is equal to the number of counties $m$ this is because we are trying to match all the counties to a set of knights and hence it will only be successful if all the counties are matched to a knight which means that there is a flow going through each vertice $c_i$ in graph $G'$ which corresponds to the maximum matching in the original bipartite graph problem.

What is left to see how does Merlin know which county $c_i$ is ruled by which knight $k_i$? This can be done by using the network flow graph $G'$ after it has found the maximum flow $f$. Since this flow is integral, we can pick one edge from $s$ to $c_j$ for each county which has flow of 1. Since we know that by Flow Conservation that the flow going into $c_j$ must be equal to the flow going out we know that there will be exactly one edge $(c_j, k_i)$ that must have flow one, so by 1. we know that it will go exactly to one knight $k_i$ which is in the subset $S_j$. Therefore if there is an edge $f(c_j, k_i) = 1$ then county $c_i$ and will be ruled by knight $k_i$. We can do this analysis for all counties and see which knights they are ruled by, and so we have found the allocation of counties to knights.

The second part of the question asks us to prove the complexity of in terms of some function $F(v, e)$ where $F(v, e)$ denotes the running time of a Max-Flow algorithm on a graph with $v$ vertices and $e$ edges. We can focus on the graph $G'$ that we have constructed. We know that the number of vertices are defined in equation (3) where $|V'| = m + n + 2$ this is because we now include the two new vertices namely $s$ and $t$ and we know that the original graph $G$ had $m$ counties and $n$ knights and as a result we get the number of vertices in graph $G'$. Likewise we can find the number of edges in $G'$. We have equation (4) we know they are equal $|E'| = \sum_{j\in m} |S_j| + m + n$ this is because in addition the original $\sum_{j\in m} |S_j|$ edges in graph $G$ where each county has exactly one edge to each knight that it can be allocated $S_j$, we have now created $m$ new edges from the source $s$ to each county $c_j$, likewise we have created $n$ new edges from each knight $k_i$ to the target vertice $t$.

As a result we can write the complexity of in terms of some function $F(v, e)$ given the number of edges and vertices in the graph $G'$ calculated above.

$$F(v, e) = F(|V'|, |E'|) = F(m + n + 2, \sum_{j\in m} |S_j| + m + n) \tag{16}$$

However we can attempt to find the complexity of the max-flow algorithm terms of some upper-bound function $O$. I will assume that in this case Merlin uses as the Max-Flow algorithm the algorithm Ford-Fulkerson. We know that the complexity of this algorithm on a network flow graph $G$ is $O(E|f^*|)$ where $E$ are the number of edges in $G$ and $f^*$ is the denotes the maximum flow in the network. As a result we know that the maximum flow in our graph $G'$ is $|f| = |M|$ where $M$ is the maximum matching by our proof 3. Therefore the upper-bound would be when we get a complete matching and it would be exactly when $|M| = m$, where $m$ are the number of counties in in our original problem. As a result we get an expression for the upper-bound of the algorithm for our graph $G'$.

$$O(E|f^*|) = O(E'|f^*|) = O(E'm) = O((\sum_{j\in m} |S_j| + m + n)m) \tag{17}$$

## 2.3 Question 2b

Suppose that Merlin runs the Max-Flow algorithm on the encoded instance, but it does not produce a solution that fits the constraints and prevents all counties from revolting. King Arthur demands an explanation. While he believes that the algorithm/encoding (and proof) is correct, he doubts that Merlin has executed the algorithm correctly on the given large instance. Merlin needs to convince the king that no suitable assignment is possible under the given constraints. Re-running the algorithm step-by-step in front of the king is not an option, because the king does not have that much time. How can Merlin quickly convince the king that there is no solution, based on the structure of the Max-Flow problem? (Note that the argument must work for every instance where there is no solution, not just a particular instance.)

## 2.4 Answer 2b

Suppose that Merlin runs the Max-Flow algorithm but didn't find a solution this means that we have that our max flow is less than the total number of counties $m$. This means that some of our counties are not matched to a knight. We need to find a way to show Arthur why this is the case.

Firstly we note that Merlin has executed the Max-Flow algorithm and so has the max flow $f$ and the network graph with the flow. We know that by the **Max-flow Min-cut Theorem** there must be a cut in this graph where it is equal to the flow $f$ which is not maximal and we know that $|f| < m$. To find such a cut we can begin by constructing the residual network of graph $G'$ which we will call $G'_f$.

To find such a cut I will prove that: The residual network $G'_f$ contains no augmenting paths $\implies$ $|f| = c(S,T)$ for some cut $c(S,T)$ of $G'$. Proof:

- First we know that there are no augmenting paths in $G'_f$.
- Let $S$ be the set of all vertices reachable from $s$ in $G'_f$.
- Let $T$ be the set $V - S$ which are all the vertices not reachable from $s$.
- By construction of $S$ we know that $s \in S$ and since there are no augmenting paths in $G'_f$ we know that $t \in T$.
- We know that $f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u)$.
- What is important to note is that the all the flows through edge $(v,u) = 0$ where $v \in T$ and $u \in S$ this is because if otherwise $c_f(u,v) = f(v,u)$ would be positive and so we would have $(u,v)$ in the edges of the flow, so $v$ would be in $S$.
- Similarly we must have that $f(u,v) = c(u,v)$ where $v \in T$ and $u \in S$ this is because since we cannot transverse it, it must have full capacity and as a result flow is going through it.
- As a result we get that: $f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) = \sum_{u \in S} \sum_{v \in T} c(u,v) - \sum_{u \in S} \sum_{v \in T} 0 = \sum_{u \in S} \sum_{v \in T} c(u,v) = C(S,T)$
- So we get that $f(S,T) = C(S,T)$.
- We know by Lemma 26.4 that $|f| = F(S,T)$ therefore we get that $|f| = F(S,T) = C(S,T)$. ■

What is important about this proof is the way we have partitioned the graph $G'$ which ensures we have found the cut that corresponds to the maximal flow $f$. We can partition $G'$ by separating it into two sets $S$ and $T$. We do this like in the proof where $S$ be the set of all vertices reachable from $s$ in $G'_f$ and $T$ be the set $V - S$ which are all the vertices not reachable from $s$.

By construction of the graph $G'$ what we can also note since we are adding to $S$ all the vertices which are reachable from $s$ in $G'_f$ we can see that $c_f(s, c_i) = c(s, c_i) - f(s, c_i) = c(s, c_i) - 0 = c(s, c_i)$ which means we are adding to $S$ the counties $c_i$ which do not have any flow through them in $G'$, which means that they have no knight allocation. The set $S$ will also include the vertice that can be reached form $c_i$ since if $f(s, c_i) = 0$ in $G'$ and $f(c_i, k_j) = 0$ then the residual capacity of $c_f(c_i, k_j) = 1$ so $k_j$ is reachable via $s \to c_i \to k_j$ and so will be added to $S$.

This means that we will also be adding to the set $S$ all the vertices $c_i$ which have an edge that links to $k_j$ if $k_j$ is in $S$. This is because we know that this $k_j$ is in $S$ which means that all the vertices that go into $k_j$ will also be in $S$ by construction of the set $S$.

Now that we have partitioned the set into $S$ and $T$. We can extract information from them. We know that all the county vertices in the set $S$ will either not have an knight allocation or if the knight $k_i$ is also in $S$ then all the counties $c_j$ that have an edge connecting to this knight will also be in it.

Before continuing lets define a new function $N$ which takes in a vertice $a$ and returns the set of neighbours $B$ of the vertice $a$. That is all the vertices that can be reached from $a$. So for example $N(c_i) = S_i$ this means $N$ will return the set of knights $S_i$ that can oversee $c_i$.

Let $S_c$ be the set of all counties in $S$. Now for each $c_i \in S_c$ we can evaluate $N(c_i)$ which will give us the the set of knights that can oversee $c_i$. If we define $K_{union}$ the union of all sets $N(c_i)$ for all $c_i \in S_c$ we can get the set of all knights that can be allocated to the counties that are in $S_c$. That is:

$$K_{union} = \bigcup N(c_i) \text{ for all } c_i \in S_c \tag{18}$$

Now we have created two sets $K_{union}$ and $S_c$. Just to recapitulate $S_c$ is the set of counties in $S$ where $S$ is the set of vertices reachable in the residual network $G'_f$. $K_{union}$ is the set of all the knights that are neighbouring all the counties in the set $S_c$.

No we can look at the structure of each of them. First we look at the set $K_{union}$ we can evaluate the number of counties that these knights can oversee. We know that a knight $k_i$ can oversee at most $q_i$ counties. Therefore the $K_{union}$ can over see at most $\sum_{k_i \in K_{union}} q_i$ which is simply the sum of $q_i$ for all the knights in $K_{union}$.

Since we know that some of the vertices in $S_c$ do not have an allocation this means that $|S_c| > \sum_{k_i \in K_{union}} q_i$, the number of counties in $S_c$ is greater than the sum of all the capacities that each knight can hold. I will show this is true by proof by contradiction.

Assume that $|S_c| \leq \sum_{k_i \in K_{union}} q_i$ then that means that there are less counties than their are quotas of the knights that these counties can go to. This means that the knights in $K_{union}$ can oversee all the counties in $S_c$. This means that now in the original network flow $G'$ we can add flow through these counties and knights. So now for all $c_i \in S_c$ and $k_j \in K_{union}$ we have $f(c_i, k_j) = 1$. Which in turn by Flow Conservation we know that $f(s, c_i) = 1$. However now we need to update the residual network $G'_f$ capacities. We know that $c_f(s, c_i) = c(s, c_i) - f(s, c_i) = 1 - 1 = 0$ therefore $c_f(s, c_i) = 0$ for all $c_i \in S_c$. But now $c_i \notin S_c$ since they are not reachable in $G'_f$ but we know that $c_i \in S_c$, thus we have a contradiction. ■

This must be the case since if we could allocate all of these county to all of knights we would have no counties in $S_c$ since we would have flow through all of them and so $S_c$ would be empty. This means that there is a subset of counties $S_c$ such that the quotas of the knights $\sum_{k_i \in K_{union}} q_i$ that they are willing to be ruled by is smaller than the number of counties in the subset. And so there cannot be an allocation for these counties in $S_c$.

As a result Merlin can present the sets $K_{union}$ and $S_c$ to Arthur and explain why there cannot be an allocation of knights for the subset $S_c$ of counties with the above explanation.

Below I have created a simple example of how to use this.

This is a simple example with three counties and two knights and how using the procedure explained above Merlin can show Arthur quickly that there cannot be an allocation of counties to knights for this problem. Merlin knows that the max flow is 2 but there are 3 counties.

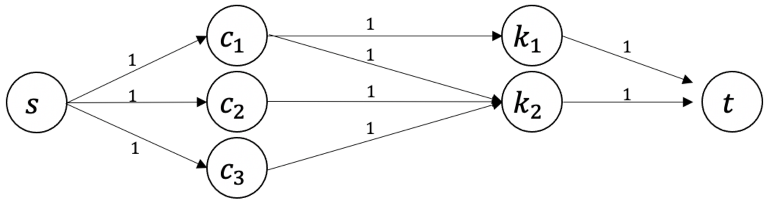

Figure 3: This shows the construction for the network graph $G'$ with the capacities of each edge using the function $c$. In this case we have three counties $c_1, c_2, c_3$ and two knights $k_1, k_2$. which have quotas $q_1 = 1$ and $q_2 = 1$.

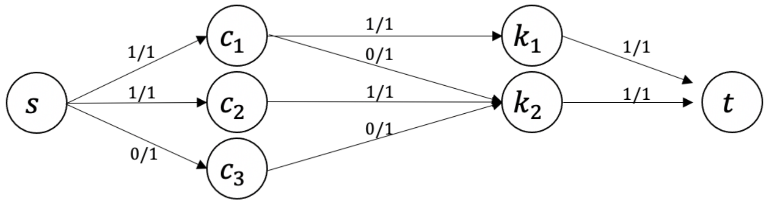

Figure 4: This shows the flow and the capacity of each edge where $(f/c)$ is used as notation. This figure shows the algorithm after it has found the maximum flow of this graph.

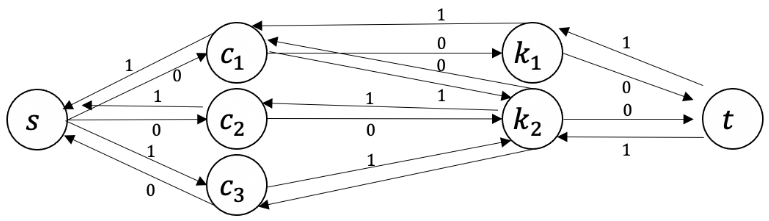

Figure 5: This shows the residual network graph $G'_f$ which we will use to find the min cut and partition the graph into the two sets $S$ and $T$.

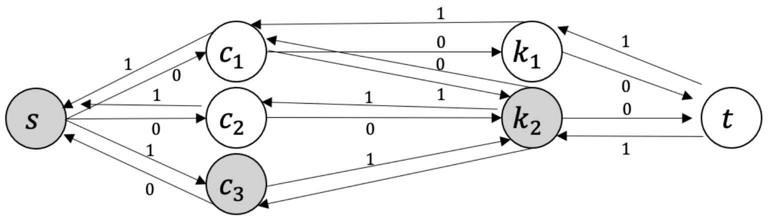

Figure 6: The vertices that are shaded show the reachable vertices from $s$. This means that by construction we have that the set $S = \{s, c_1, c_2, c_3, k_2\}$ this is because we also include all the counties that have an edge connecting them to a knight in $S$. We can see that $T = \{k_1, t\}$.

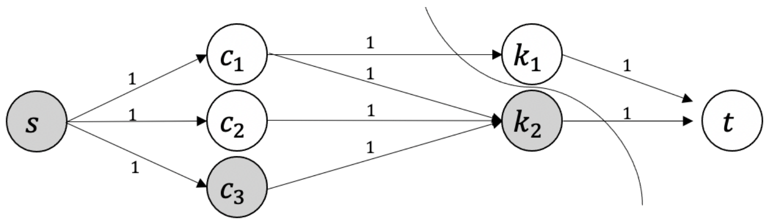

Figure 7: This shows the cut that is created by using the sets $S$ and $T$. We can see that the value of this cut is indeed 2 which is equal to the value of the max flow of the graph $G'$ that Merlin knew after he executed the max flow algorithm.

What is left to show are the sets $S_c$ and $K_{union}$. In this case $S_c = \{c_1, c_2, c_3\}$ and $K_{union} = \{k_1, k_2\}$ this is because it is the set of reachable knights from the counties in $S_c$ in this case it is equal to all the knights. We can see that the size of $|S_c| = 3$ and the number of available quotas in $K_{union}$ is equal to $1 + 1 = 2$. Since $|S_c| > 2$ then there cannot be an allocation of all three counties to the two knights in our question. This is what Merlin can show Arthur and convince him that there is no solution.