MARKO MEKJAVIČ

S1813308

ALGORITHMS AND DATA STRUCTURES

COURSEWORK 2

Scanned by Easy Scanner

1a)

S_n – set of all elements : $\{1, \dots, n\}$
$n.size$ – size of n-th element
$n.size = s_n$

$n.value$ – value of n-th element
$n.value = v_n$

Proposed Recurrence:

$$P_{n,c} = \begin{cases} 0 & \text{if } n = 0 \\ P_{n-1,c} & \text{if } n.size > c \\ \max(P_{n-1,c},\ P_{n-1,c-n.size} + n.value) & \text{otherwise} \end{cases}$$

optimal solution for maximum-value packing for k and $\hat{c}$

Claim: $P_{k,\hat{c}}$ is a suitable recurrence that holds for all $k \leq n$ and any arbitrary $\hat{c}$.

Proof: In order to prove the above claim I will first show that the proposed recurrence follows the optimal substructure concept

Optimal Substrure Lemma

Proof by Induction:

- Let's define S_n as a set of n elements $\{1, \ldots, n\}$ and let $\mathcal{A}_c(S_n)$ be the set containing the optimal solution - best valued items from S_n satisfying capacity C.

- Hence, sum of elements in the optimal solution $\mathcal{A}_c(S_n)$ is a maximum-value packing for S_n under C:

$$\sum_{i \in \mathcal{A}_c(S_n)} i.value = P_{n,c}$$

- Now we know that $\mathcal{A}_c(S_n)$ either includes the last element n or not. We will go over both cases:

(a) $n \in \mathcal{A}_c(S_n)$
(b) $n \notin \mathcal{A}_c(S_n)$

(a) $n \in \mathcal{A}_c(S_n)$

Let's firs suppose n is in optimal solution. Then we also claim that a set without n-th element: $\mathcal{A}_{c-n.size}(S_n) \setminus \{n\}$ is an optimal solution for a set $\{1, \ldots, n-1\}$ under capacity constraint $C - n.size$.

Contradiction:

- Suppose $\mathcal{C}_{c\text{-}n.size}(S_n) \setminus \{n\}$ is not an optimal solution. Then there exists another set $\mathcal{C}_{c\text{-}n.size}(S_{n-1})^*$ such that its maximum-value packing is bigger than that of $\mathcal{C}_{c\text{-}n.size}(S_n) \setminus \{n\}$ on the set $\{1, \dots, n-1\}$:

$$\sum_{i \in \mathcal{C}_{c\text{-}n.size}(S_{n-1})^*} i.value > \sum_{i \in \mathcal{C}_{c\text{-}n.size}(S_n) \setminus \{n\}} i.value$$

under capacity constraints of c-n.size.

- From this we can derive that by adding n to $\mathcal{C}_{c\text{-}n.size}(S_{n-1})^*$ we get a new solution:

$$\mathcal{C}_c(S_n)^* = \mathcal{C}_{c\text{-}n.size}(S_{n-1})^* \cup \{n\}$$

whose value packing is then:

$$\sum_{i \in \mathcal{C}_c(S_n)^*} i.value = \sum_{i \in \mathcal{C}_{c\text{-}n.size}(S_{n-1})^*} i.value + n.value >$$

$$> \sum_{i \in \mathcal{C}_{c\text{-}n.size}(S_n) \setminus \{n\}} i.value + n.value = \sum_{i \in \mathcal{C}_c(S_n)} i.value$$

which is a contradiction.

(b) $n \notin \mathcal{A}_c(S_n)$

Now let's consider that the optimal solution doesn't contain n. Then the optimal solution for set $\{1, \dots, n\}$ is nothing else but the optimal solution for when we consider only $n-1$ elements, under same C.
Suppose $\mathcal{A}_c(S_{n-1})$ is the optimal solution for $\{1, \dots, n-1\}$ then:

$$\mathcal{A}_c(S_n) = \mathcal{A}_c(S_{n-1}) \iff \sum_{i \in \mathcal{A}_c(S_n)} i.value = \sum_{i \in \mathcal{A}_c(S_{n-1})} i.value$$

Contradiction:

Suppose $\mathcal{A}_c(S_{n-1})$ is not an optimal solution for $\{1, \dots, n-1\}$. Then $\mathcal{A}_c(S_n)$ is not optimal solution for $\{1, \dots, n\}$ which clearly is a contradiction.

• Now let's prove that the recurrence $P_{k,\hat{c}}$ is a maximum-value packing for all $k \leq n$ and arbitrary $\hat{c}$.

Base Case: $k = 0$

So since we have no elements to choose from we can't get any value and hence $P_{k,\hat{c}} = 0$, which indeed is the maximum-value packing.

Holds for base case.

Inductive Hypothesis:

Now let's assume the recurrence holds for all k up to n and arbitrary $\hat{c}$.

Induction:

Let's prove the recurrence holds for $n = k$.

If $k = n$ then we know that the optimal solution either includes n or not.

1. $n \in$ optimal solution
2. $n \notin$ optimal solution

~~(2) Let's take a look at the example when we take~~
~~in~~

(2) Let's first look at the scenario when the size of n-th element is simply bigger than the size of the knapsack. In that case we clearly cannot take n and hence our maximum-value packing must be exactly the one for n-1 elements.
Therefore:

$$P_{n,c} = P_{n-1,c}$$

And since we have already proven in Optimal Substructure lemma that the recurrence relies on the optimal substructure of optimal solution to subproblems and because we know that $P_{n-1,c}$ is a maximum-value packing from the Induction Hypothesis, we also know that $P_{n,c}$ must be the optimal solution to maximum-value packing for $\{1,\dots,n\}$ since n is too big.
Hence, we have proven for case n.size > C that the recurrence holds for all $k \le n$ and arbitrary $\hat{C}$.

(1) Now let's consider the scenario when $n.size < C$. Here we can take n or not.

(a) Let's first consider the example of not taking n. Then the maximum-value packing must be no other than $P_{n-1,c}$, which we have already proven to be maximum-value packing for S_{n-1}.

(b) Let's now take a look at the example of when we take n. Then our optimal solution for maximum-value packing must be:

$$\sum_{i \in \mathcal{A}_{c-n.size}(S_n) \setminus \{n\}} i.value + n.value$$

where $\mathcal{A}_{c-n.size}(S_n) \setminus \{n\}$ is the optimal solution to the subproblem for $\{1, \ldots, n-1\}$ under capacity constraints of $c-n.size$, which according to optimal substructure lemma we know holds.

Now in order to conclude whether to take n or not when given the chances, we need to compare (a) and (b):

$$\max\left(P_{n-1,c}, \sum_{i \in \mathcal{A}_{c-n.size}(S_n) \setminus \{n\}} i.value + n.value\right)$$

which we know is: $\max\left(P_{n-1,c}, P_{n-1,c-n.size} + n.value\right)$

and this is exactly our recurrence relation for the case of when $n.size < c$.
Hence, since both possible solutions rely on the optimal substructure lemma, $P_{n,c}$ in that case must as well be the optimal solution to the maximum-~~profit~~ value packing.

Hence, we have proven for both cases that the recurrence holds for all $k \leq n$ and therefore $P_{k,c}$ is a maximum-value packing.

1b

Dynamic Programming Algorithm using Bottom-Up Approach

Input: n – list of elements
V – two-dimensional array dimensions $(n+1) \times (C+1)$ for storing values
C – knapsack's capacity
K – two-dimensional array dimensions $(n) \times (C)$ for storing whether we put the element into our knapsack or not

```
0  KnapSack(n, V, C, K)
1
2  for ĉ to C:
3      V[0, ĉ] = 0
4  for i=1 to n:
5      for ĉ=0 to C:
6          if (i.size > ĉ) and (i.value + V[i-1, ĉ-i.size] >
7                                                > V[i-1, ĉ]):
8              V[i, ĉ] = i.value + V[i-1, ĉ-i.size]
9              K[i, ĉ] = 1
10
11         else:
12             V[i, ĉ] = V[i-1, ĉ]
13             K[i, ĉ] = 0
```

```
14    set_of_taken = []
15    S = C
16    for i = n down to 1:
17        if K[i, S] == 1:
18            set_of_taken.append(i)
19            S = S - i.size
20
21    return set_of_taken, V[n, C]
```

Scanned by Easy Scanner

Description of the Algorithm

I will firstly describe how the algorithm works and analyze it's time complexity along the way, and then explain how it incorporates the derived recurrence from 1a.

- The algorithm takes four variables as an input:
  - n - list of elements to choose from
  - V - 2-D array for storing values of size $(n+1) \times (C+1)$
  - C - capacity of the knapsack
  - K - 2-D array for keeping track of which elements to take of size $n \times C$

- The idea behind this algorithm is to compute each value, that is, optimal value for any combination of C and n, exactly once and store it in a table V. Then when needed to compute such problem again we can just look for it's value in the table rather than computing it again.
Throughout this process we ~~store~~ keep track whether the optimal solution for ~~combin~~ problem i and C contains i by indicating 1 in K[i, C] if it contains and 0 if not.

- First step is to cover the base case when we have no elements to choose from. In that case the optimal solution for any capacity remains 0. This we take care of in lines 2-3 where we initialize table $V[0, C] = 0$ for all C.
This takes exactly $\Theta(C)$.

• Second step is to compute all other values in the table, which happens in lines 4 – 13. For this I used two for loops to loop over all n elements, that is from 1 to n, and the second for loop allows me to loop over all capacities $\hat{c}$ from 0 to C.
Now for any i and $\hat{c}$ the optimal solution will either be the optimal solution for $i-1$ at same capacity $\hat{c}$ or a composition of optimal solution for $i-1$ at capacity $\hat{c} - i.size$ and value of i.
Now due to functionality of the for loop and because we have taken care of the base case we know that we have already computed the optimal solution for the two subproblems and stored them in the table. Hence, we don't need to compute them again but just look them up in the table.

• Algorithm then compares the two possible results and if the addition of the i-th element brings bigger value it stores this decision in table K by setting $K[i, \hat{c}] = 1$. Otherwise it sets it to 0.

• Now let us look at time complexity of this procedure. We know that looking up the value in the table takes $\Theta(1)$ constant time and so do addition and comparison of two values. Hence, we can drop these time complexities. Regarding the iteration over the whole table of V we know that for every iteration of the outter for loop which goes from 1 to n, we need $C+1$ iterations of inner for loop going from 0 to $C+1$. Hence, the time complexity

of this procedure is $\Theta(n(C+1))$.

- Now let's consider the procedure of tracing back which elements we took described in lines 14-21. I used one for loop from n to 1 to check whether or not the optimal solution consists of the last element and then on whether the sub-problem consists of $(i-1)$-th element etc.
All operations in this procedure; look up in the table, subtraction and appending take constant time and can be dropped. So since we need to iterate n times the time complexity of this problem is $\Theta(n)$.

- To sum up time complexities of the 3 procedures:
$$\Theta(C) + \Theta(n(C+1)) + \Theta(n) =$$

$= \Theta(nC + 2n + C)$ and as n goes to ∞ the only relevant term remains nC.

Hence, time complexity of KnapSack algorithm is ~~$\Theta(nC+2n+C)$~~ $\Theta(nC)$.

- Now let me explain how KnapSack incorporates the recurrence from 1b. By taking care of the base case, we found the optimal solution for $i=0$ and all $\hat{C}$.
Now for any $i > 0$ and any $\hat{C}$ we see from lines 6-7 that the optimal solution for i and $\hat{C}$ is ~~either the optimal~~ based on the

optimal substructure, which means it can either be the optimal solution for $i-1$ at same capacity $\hat{C}$ or summation of the last element and the optimal solution for $i-1$ at capacity $\hat{C} - i.size$, whichever is bigger. And this is nothing else than the ~~above~~ recurrence defined in 1a:

$$P_{n,c} = \begin{cases} 0 & \text{if } n = 0 \\ P_{n-1,c} & \text{if } n.size > C \\ \max(P_{n-1,c}, P_{n-1,c-n.size}) & \text{otherwise} \end{cases}$$

Hence, we have proven that KnapSack incorporates the recurrence from 1a.

2a

In order to show that Merlin can use MaxFlow algorithm to efficiently compute an assignment of counties to the knights under explained constraints (EC), I will firstly construct a graph for the given problem and then prove the following 3 claims:

Claim 1: Constructed graph is a Flow Network

Claim 2: MaxFlow can at most be the number of counties $\Rightarrow |f| = m$

Claim 3: MaxFlow equals number of counties - $-|f| = |c| = m$ - if and only if there exists a perfect assignment of counties to knights under (EC)

Explained Constraints from the question (EC) (from now on referred as EC)

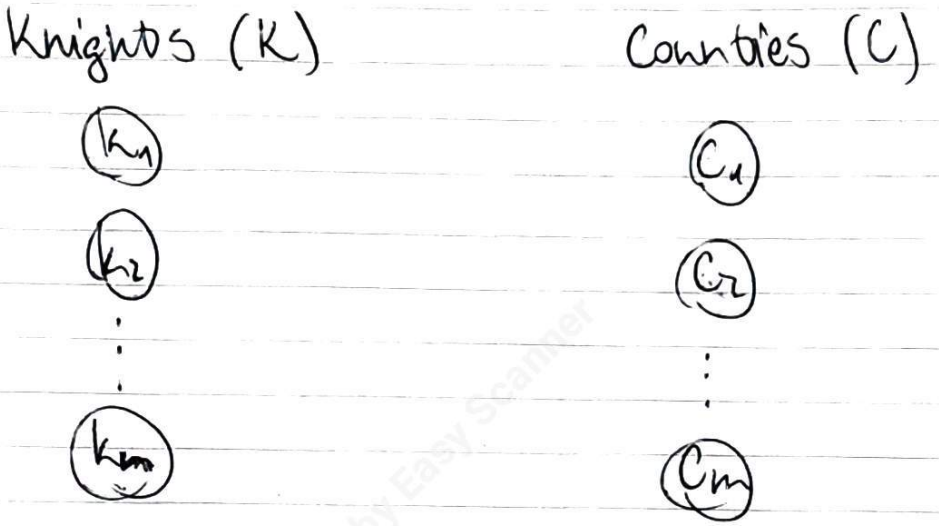
(1) each knight i (or k_i) can oversee at most q_i counties

(2) each county j (or c_j) will revolt if its not overseen by some knight k_i in a given subset $S_j \subseteq \{1, \dots, n\}$ of preferred knights.

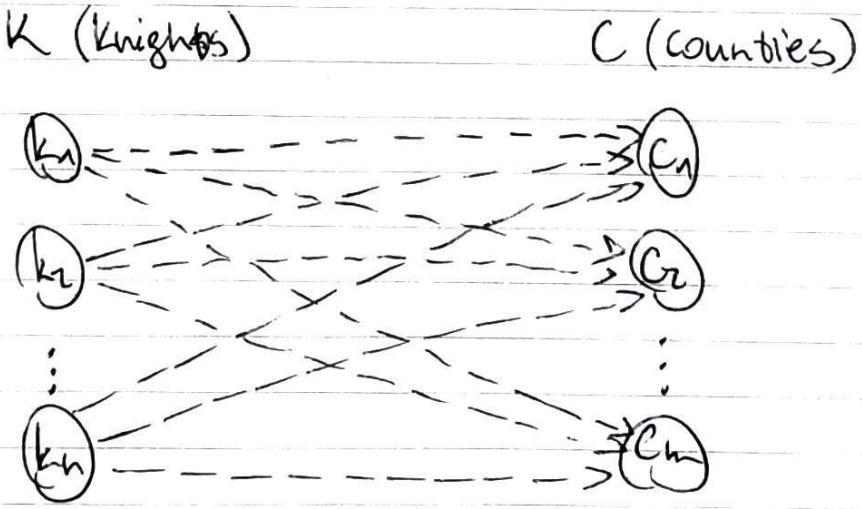
(3) only one knight can oversee a county

Construction of Graph

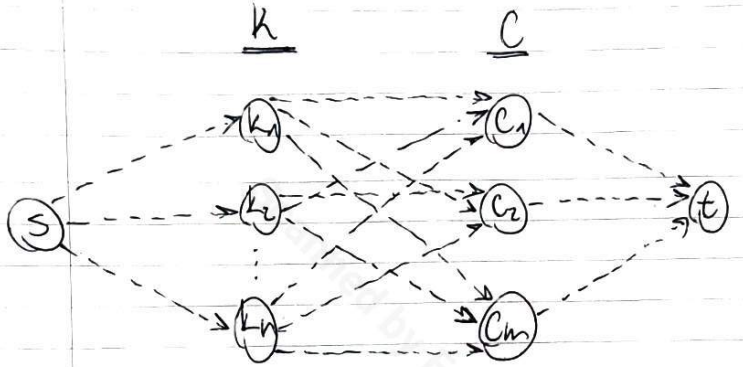
- So since we have n knights and m counties, we can represent members of each set (K - knights an C - counties) as vertices in its own layer:



- Now let's connect every $k_i \in K$ with every $c_j \in C$ with a directed edge from K to C:



Now let's add two vertices; source s on the left of K, and sink t to the right of C. Let's connect every $k_i \in K$ to the source s with directed edge and likewise every $c_j \in C$ to sink t:

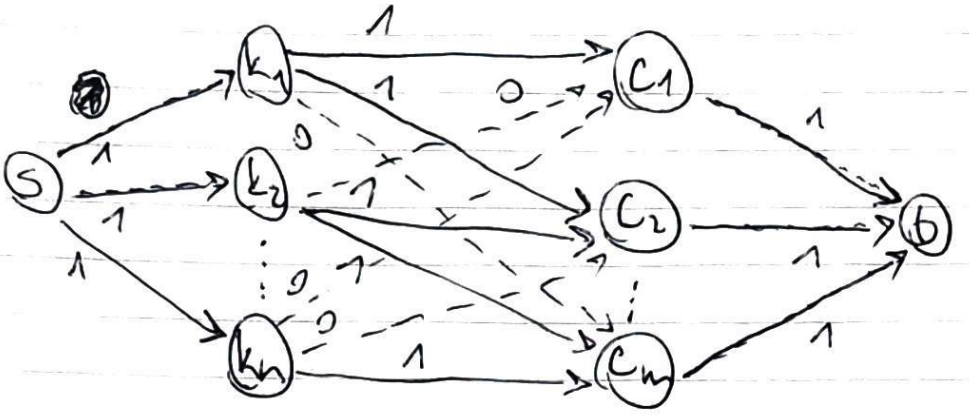


Now let's ~~count~~ define a capacity ~~constraint~~ function $c(u,v)$ that assigns capacities to every directed edge.

$$c(u,v) = \begin{cases} q_i & \text{if } u = s \text{ and } v = k_i \in K \\ 1 & \text{if } u = c_j \in C \text{ and } v = t \\ 1 & \text{if } u = k_i \in K \text{ such that } k_i \in S_j \\ & \quad \text{and } v = c_j \in C \\ 0 & \text{otherwise} \end{cases}$$

so applying $c(u,v)$ to our graph (let's refere to it as G') we would get something similar to this:

Knights (K) Countries (C)



dotted

I decided to use ~~both~~ lines for those edges that were assigned 0 by c(u,v) for ~~better~~ distinguishing the difference.

Hence, we have constructed a graph G' under (EC) restrictions. ~~mainly~~ I would especially like to point out that only edges between $k_i \in K$ and $c_j \in C$ ~~s~~ that satisfy $k_i \in S_j$, are assigned 1 by c(u,v). ~~so that~~

So now that we have constructed a graph G' let's prove that ~~it~~ it indeed is a Flow Network.

Claim 1: G' is a Flow Network

Proof: I will prove that G' satisfies ~~both~~ both constraints to be a Flow Network.

i) Flow Network is a four-tuple (G, c, s, t) such that $G = (V, E)$ is a directed graph with two distinguished vertices, source s and sink t.

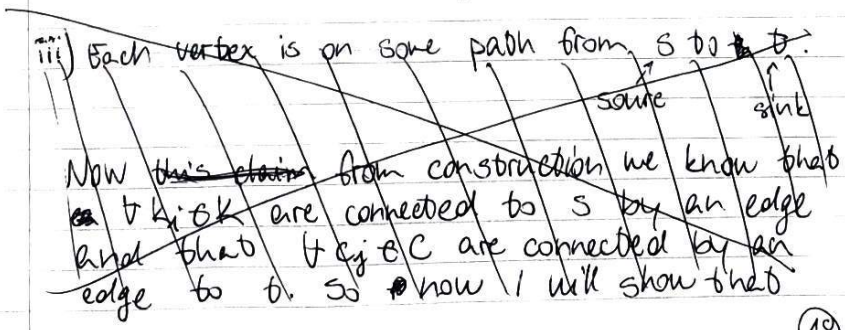
As it can be observed from construction, graph G (before adding s and t) is a directed graph and two distinguished vertices were attached, increasing G into G' – still connected graph.

ii) c is some capacity function:

$c: V \times V \to \mathbb{R}$ such that:
- $\forall (u,v) \in E;\ c(u,v) \geq 0$
- $\forall (u,v) \notin E;\ c(u,v) = 0$

As it can be seen from construction, weights of edges in G' were set to 0 if the edge is not in the set that satisfies (EC) and set to $c(u,v) > 0$ otherwise.
Hence, the proposed capacity function in construction satisfies required constraints.

iii) Each vertex is on some path from s to t:



Now from construction we know that $\forall k_i \in K$ are connected to s by an edge and that $\forall c_j \in C$ are connected by an edge to t. So now I will show that

~~there is at least one edge between [redacted] and~~ $\forall c_j \in C$. Since we know from description that ~~[redacted]~~ every country c_j has a set of favourable knights ~~[redacted]~~

So since we have proven that G' satisfies both constraints, we have proven that G' is indeed a Flow Network.

Now let's prove <u>Claim 2</u> stating:

$$\text{MaxFlow is at most } m \Rightarrow |f| = |C| = m$$

(m ↓ number of countries)

<u>Proof</u>: It was already proven under Corrolary 4 that for a Flow Network $\mathcal{N} = (G=(V,E), c, s, t)$ and Network Flow f in $\mathcal{N}$ that:

$$|f| = f(V,t)$$

(network)

~~that is the proof~~ which intuitively means that the flow is equal to flow coming into sink t from all vertices in V.
Since we know from construction that the only vertices connected to ~~(with capacity $c(v,t) > 0$) are~~ the sink t are only countries. So hence for our concrete example:

$$|f| = f(s,V) = f(V,t) = f(C,t)$$

(set of countries)

Now since we know that the flow f satisfies the capacity constraints, and since we know from construction that capacity of every edge from C to t ~~and that there are exactly $|C| = m$ edges~~ is exactly 1 ($c(u,v) = 1$) and also that there are exactly $|C| = m$ edges from C to t we can extend the previous mathematical notation:

$$\underbrace{|f| = f(V,t) = f(C,t) \leq c(C,t) = |C| = m}$$

$$|f| \leq |C| = m$$

So hence we have proven our second claim, that the MaxFlow is at most m – number of counties.

Now let's move onto proving the third claim. I will prove both directions of the claim.

Claim 3:

MaxFlow equals number of counties ($|f| = |C| = m$) $\iff$ there exists a perfect assignment of counties to knights under ($\exists C$)

(=>)

Claim: There is an optimal assignment of counties to knights under (EC)
$\Rightarrow$ there exists a MaxFlow such that $|f| = m$

So in order to prove this claim stated above I will follow the following steps:

1. Go over the optimal assignment
2. Define a flow f in G'
3. Show that defined flow fits all constraints
4. Show that then $|f| = m$

So let's start with 1:
We know that there is an optimal assignment of counties to knights under (EC). So we know that each county $c_j \in C$ is overseen by some knight $k_i \in K$ such that $k_i \in S_j$ and that all knights don't oversee more than q_i counties. Let's rewrite the last constraint the following way:
Let's denote Z_i for every knight k_i as a $Z_i \subseteq C$, so a set of all assigned counties.
Then we know that $|Z_i| \leq q_i$ for every knight k_i.

② Now let's define a flow $f : V \times V \to \mathbb{R}$ in G':

$$f(u,v) = \begin{cases} 1 & \text{if } v = t \text{ and } u = c_i \; ; \; c_i \in C \\ 1 & \text{if } u = k_i \in K \text{ and } v = c_j \in C \\ & \text{and } c_j \in z_i \text{ for } \forall k_i \in K \\ |z_i| & \text{if } u = s \text{ and } v = k_i \in K \\ 0 & \text{otherwise} \end{cases}$$

Now let's further define f as such that? for every $f(u,v) \geq 0$ then there must be $f(v,u) = -f(u,v)$.

③ Now let's show that f fits all the constraints and hence indeed is a flow:

(a) Capacity Constraint: $\forall (u,v) \in V \times V \; ; \; f(u,v) \leq c(u,v)$

To prove that f satisfies the above constraint let me prove by cases:

(i) case $u = c_i$ and $v = t \Rightarrow f(u,v) = 1$
We know from the construction of the Flow Network that the capacity $c(u,v)$ for edges from c to t is 1 ($c(u,v) = 1$). And since our proposed flow f sets all those edges to 1 then we fit the criteria:

$$f(u,v) \leq c(u,v)$$

(ii) Case $u = k_i$ and $v = c_j$ $\Rightarrow$ $f(u,v) = 1$
and $c_j \subseteq Z_i$ $\forall k_i \in K$

So we know from the construction of G' that capacity ($c(u,v) = 1$) for all edges that exist from K to C under the explained constraints ($\mathcal{C}$). Since this case for f is defined under same constraints for edges between K to C and sets them to 1, we see that it holds true under:

$$f(u,v) \leq c(u,v)$$

(iii) Case $u = S$ and $v = k_i \in K$ $\Rightarrow$ $f(u,v) = |Z_i|$

So we know from construction of G' that all capacities for all edges between S and K are limited to ℓ_i ($c(u,v) = \ell_i$) for every $k_i \in K$. Since we have defined $|Z_i| \leq \ell_i$ we know that we here fit the constraints:

$$f(u,v) \leq c(u,v)$$

(iv) otherwise $f(u,v) = 0$

So from construction we defined $c(u,v) = 0$ for $\forall (u,v) \notin E$, and since we've covered all other cases for $f(u,v)$ we know that the remaining cases will either

$$f(u,v) \neq c(u,v) = 0 \text{ for } \forall (u,v) \notin E$$

or $f(u,v) < c(u,v)$ for remaining edges that exist, hence satisfying: $f(u,v) \leq c(u,v)$.

So we have shown it holds for capacity constraints.

(b) Skew Symmetry: $\forall (u,v) \in V \times V,\ f(u,v) = -f(u,v)$

So ~~we have also~~ if we look at the definition of f we specified that for all $f(u,v) \geq 0$ there must be an edge from v to u with a same flow in the other direction:

$$f(v,u) = -f(u,v)$$

So hence we see that G' holds for skew-symmetry.

(c) Flow Conservation: $\forall u \in (V - \{s,t\}),\ \sum_{v \in V} f(u,v) = 0$

In order to prove correctness of f under flow conservation I will split the flow into 3 phases:

1. $s \longrightarrow k$
2. $k \longrightarrow c$
3. $c \longrightarrow t$

and explain how the flow conserves given we have the optimal assignment under (BC).

(1) So we know that the flow going from s is in total $\sum_{i \in k} |z_i|\ \forall k_i \in k$ since the flow from s to every $k_i \in k$ is exactly $|z_i|$ which is nothing else but the number of all assigned counties.

② So looking at second phase the flow from every $k_i \in K$ to C must be exactly $|T_i|$.
Now we know from (EC) that every county $c_j \in C$ is assigned only one knights and that knight must be $k_i \in S_j$.
So hence there is no chance two knights be assigned same county and since every edge ~~has $f(u,v)$ =~~ between K and C has $f(u,v) = 1$ then given that the flow $s \to k_i$ is defined based on assigned counties, we see that there are then exactly $|T_i|$ edges connecting every $k_i \in K$ and C_j. Hence, we conserve the flow in this step.

↓
Adding to exactly $|T_i|$ in out-flow from k_i.

③ Now let's consider the last step, from C to t.
We know that every county has only one edge to K which has a flow of 1 in order to fit the (EC) to be assigned exactly one preferred knights. ~~And since we know~~ So therefore, the net flow going into C is $|C|$. Now from definition of $f(u,v)$ we see that every edge from C to t is assigned 1 and from construction of G' we know that there are exactly $|C|$ edges since all $c_j \in C$ are connected to t with exactly one edge we can conclude that the flow going from C to t is also $|C|$, which is m.
Hence, we have proven flow conservation for the last step.

Since we have proven flow conservation for all steps we showed that:

$$cf(s,V) = cf(V,t)$$

and hence proposed flow $cf(u,v)$ indeed is a flow!

(4) Now to build on the previous proof of flow conservation we have already proven that if there exists a perfect assignment of candies to knights under (EC), then: $cf(s,V) = cf(V,t) = |C|$ which is nothing else then m!
And since we have show that $|cf| = cf(V,t) = cf(C,t)$ we have proven that if there is a perfect assignment, then:

$$|cf| = m!$$

(<=)

Claim: There exists a MaxFlow such that $|f| = m$

$\Rightarrow$ There is an optimal assignment of countles to knights under (3C)

So in order to prove that there ~~MaxFlow~~ really is an optimal assignment I will go through the MaxFlow from t to s and argue based on construction of Flow Network G' and properties of Network Flow $f(u,v)$.

~~• So since we know that $|f| = f(V,t) = |C| = m$ only the flow is coming on to only~~

• So we know that $|f| = f(V,t) = |C| = m$ but we also know from construction that $f(V,t) = f(C,t)$ since those are the only vertices connected to t. But what is more, we know that capacity $c(u,v) = 1$ for when $u = c_j \in C$ and $v = t$. And since we know that flow $f(u,v)$ must satisfy the capacity constraints then the only way for $f(C,t) = |C| = m$ is if every edge ~~in the~~ $(C \to t)$ is assigned 1 by flow $f(u,v)$.
~~And~~ And due to flow conservation we know that as much flow goes out of $c_j \in C$ must also come in. In our example this means that exactly one edge between K and $\forall c_j \in C$ can there be because we know from construction of G that $c(u,v) = 1$ for all edges between K and C. This means that each county $c_j \in C$ is overseen by exactly one knight $k_i \in K$ which satisfies (3C) (3). But what is more ~~that~~ is that we know $c(u,v) = 1$ only if $u = k_i \in K$ ~~and~~ $k_i \in S_j$ such that

and $v = c_j \in C$, which we defined in construction of G'. This is important because it means that only edges the flow can choose from to preserve the flow, are edges to the knights preferred by each county. So hence, we also satisfy (EC) (2) which states that every county $c_j \in C$ can only be overseen by a favored knight k_i such that $k_i \in S_j$.
Now we need to make sure that (EC) (2) is also satisfied, which states that every knight k_i can oversee at most q_i counties.
So we know from construction that capacity $c(u,v)$ for $u = s$ and $v = k_i \in K$ is restricted by q_i - limit for every knights $\Rightarrow c(u,v) = q_i$.
Hence, since flow satisfies capacity constraints, each knight k_i won't recive larger flow than q_i.

Now since flow also satisfies flow conservation we can be certain that therefore every knight k_i is assigned at most q_i counties since every edge between K and C can at most be 1. Therefore, we have also satisfied (EC) (2).

Hence, since we have shown that flow $f(u,v)$ satisfies all (EC) we have proven that:

MaxFlow $|f| = p$ exists $\Rightarrow$ There's an optimal assignment of counties to knights under (EC)

- So to answer the question on how can Merlin determine whether the algorithm was successful, he can simply check ~~whether~~ if the MaxFlow computed by the algorithm is equal to the numer of counties: $|ef| = |C| = m$.
And this we have already proven a few pages before, that if $|ef| = m$, then there is an assignment under (EC).

- Below I built the algorithm and I broke it down into 3 steps:

  ① Generate a Flow Network

  ② Compute the Max-Flow

  ③ Reconstructed the assignment from the Max-Flow

① Generate a Flow Network

So algorithm generateNetwork(C, K, S, s, t, Q) takes six inputs:

C - set of counties
K - set of knights
S - array of sets S_j
s - source s
t - sink t
Q - array of q_i

The algorithm closely implements the construction of ~~Network~~ Flow Network ~~from~~ explained before.

```
0  generateNetwork(C, K, S, s, t, Q):
1      ~~graph~~ network G';
2
3      for knight in knights:
4          G'.make_edge(s, knight, Q)
5          for county in counties:
6              if knight in S.county:
7                  G'.make_edge(knight, county, Q)
8              if edge(county, t) not in G':
9                  G'.make_edge(county, t, Q)
10
11     return G'

0  make_edge(u, v, Q):
1      if u=s and v=knights:
2          edge = Q.knight
3      if u=county and v=t:
4          edge = 1
5      if u=knight and v=county:
6          edge = 1
7
8      return edge
```

So the point behind generateGraph() is to loop over every S_j for every knight to check if that knight can oversee county C_j ($=j$). If so it will then add the edge to network G' but the assigned capacity is determined by a helper function make_edge() which literally implements the capacity method from construction of Flow Network, with an exception of ~~set~~ not adding edges that ~~are not favour~~ do not fit the constraints ($c(u,v)$ sets them to 0!)

So suppose all operations of calling make_edge() and adding an edge to the network G', all take constant time, then we know that the complexity of generateNetwork() algorithm is:

$$O(|K| \times \sum_{1}^{} |S_j|) \leq O(|V||E|)$$

so although we are not actually looking at every edge and every vertex, it is a very good estimation, hence:

$$O(|V||E|)$$

This time complexity is such because for every knights (line 3) we need to consider S_j for every $j \in C$ (C - Counties), which is seen in lines 5-6.

(2) Compute MaxFlow

Once I generated one network I will use the Ford-Fulkerson's Algorithm as the MaxFlow Algorithm

```
0  Ford-Fulkerson(G')
1      f ← flow of value 0
2      while there exists an augmented path P in G'_f do
3          f ← f + f_P
4
5      return f
```

Since there are more different implementations of Ford-Fulkerson's algorithm I decided to use the Edmonds-Karp's implementation for which time complexity is:

$$O(|V|^2|E|)$$

Once I compute the MaxFlow I will check whether or not it equals the $|C| = m$. If the MaxFlow equals number of counties, then I will use Breadth First Search algorithm (BFS) to loop over the network and add the edges from K to C whose flow is 1.
Otherwise, if MaxFlow is not equal to number of counties, I will return an empty set, since there is no ~~valid~~ possible assignment.

③ Generate Assignment

```
0   BFS(G', s):
1       let Q be a queue
2       let A be the assignment sets
3
4       label s as discovered
5       Q.enqueue(s)
6
7       while Q is not empty do:
            ~~v = Q.edge~~
8           v := Q.dequeue()
9           if v == t then
10              return A
11
12          for all edges from v to w in G'.adjacentEdge(v) do
13              if w is not discovered:
14                  label w as discovered
15                  Q.enqueue(w)
16
17              if edge(v,w) == 1
18                  and v = knight and w = county
19                      A ← edge(v,w)
```

So we know that ~~B~~ BFS's time complexity is $O(|V|+|E|)$ in worst.

So now I will summarize the time complexity of the described algorithm presented in 3 parts.

$$O(|V||E|) + O(|V|^2|E|) + O(|V| + |E|) =$$

↑ Generate Network — ↑ Ford-Fulkerson — ↑ BFS

$= O(|V|^2|E| + |V||E| + |V| + |E|)$ and as the number $|V|$ and $|E|$ increase and goes to ∞, we see that the first term ~~dominates~~ starts to dominate time complexity, so hence:

$= O(|V|^2|E|)$ which is the complexity of ~~Ford~~-Fulkerson's algorithm.

And since $F(V, e)$ denotes running time of a MaxFlow algorithm - in our case Ford-Fulkerson's - then we know that:

$$O(|V|^2|E|) = \overset{O}{(}F(V, e)) = \overset{O}{(}F(n+m+2, n+m+\sum_j |s_j|))$$

since our ~~g~~ network consists of exactly $n+m+2$ vertices and $n+m+\sum_j |s_j|$ edges.

Hence, time complexity of ~~Merlin~~ the algorithm I'd give to Merlin is:

$$O(F(n+m+2, n+m+\sum_j |s_j|))$$

(2b)

Since the king doesn't believe that Merlin executed the algorithm correctly I would suggest Merlin to prove the following ~~constraints~~ claims:

(1) Produced flow by MaxFlow algorithm is indeed a network flow

~~(2) Residual networks of residuals~~
(2) Produced flow is maximal

(3) $|f| < m$

(1) So firstly I would recommend Merlin to prove that the produced flow is indeed a network flow, meaning it satisfies the below constraints:

(a) Skew symmetry: $\forall (u,v) \in V \times V \; ; \; f(u,v) = -f(v,u)$

(b) Capacity constraints: $\forall (u,v) \in V \times V \; ; \; f(u,v) \leq c(u,v)$

(c) Flow conservation: $\forall u \in (V - \{s,t\}), \sum_{v \in V} f(u,v) = 0$

Merlin can ~~perform this~~ check whether or not the flow satisfies the above constraints relatively fast, in time of $O(n^2)$ by looping over all vertices for every vertex - i.e 2 nested for loops.

(2) Once he has proven that the MaxFlow algorithm has produced a valid network flow on the given network, I would advise him to prove that the produced network is maximal. This Merlin can do by constructing or obtaining a residual network

G_f and show that there are no more augmented paths. This Merlin can do by running the BFS (Breadth First Search) algorithm to traverse through the residual network and check along the way if there exists another augmented path. If it does not, then by the Max-Flow min-cut Theorem f is a maximum flow in G_f.

(3) Lastly, I'd recommend him to prove that if $|f| < m$ this means there is no valid assignment. Merlin can do this by using proof from 2a where we have proven that if $|f| \neq m$, then there is no valid assignment available.

If Merlin manages to prove all these claims, which I think he can, then the King will understand and agree that there truly isn't a possible assignment available.