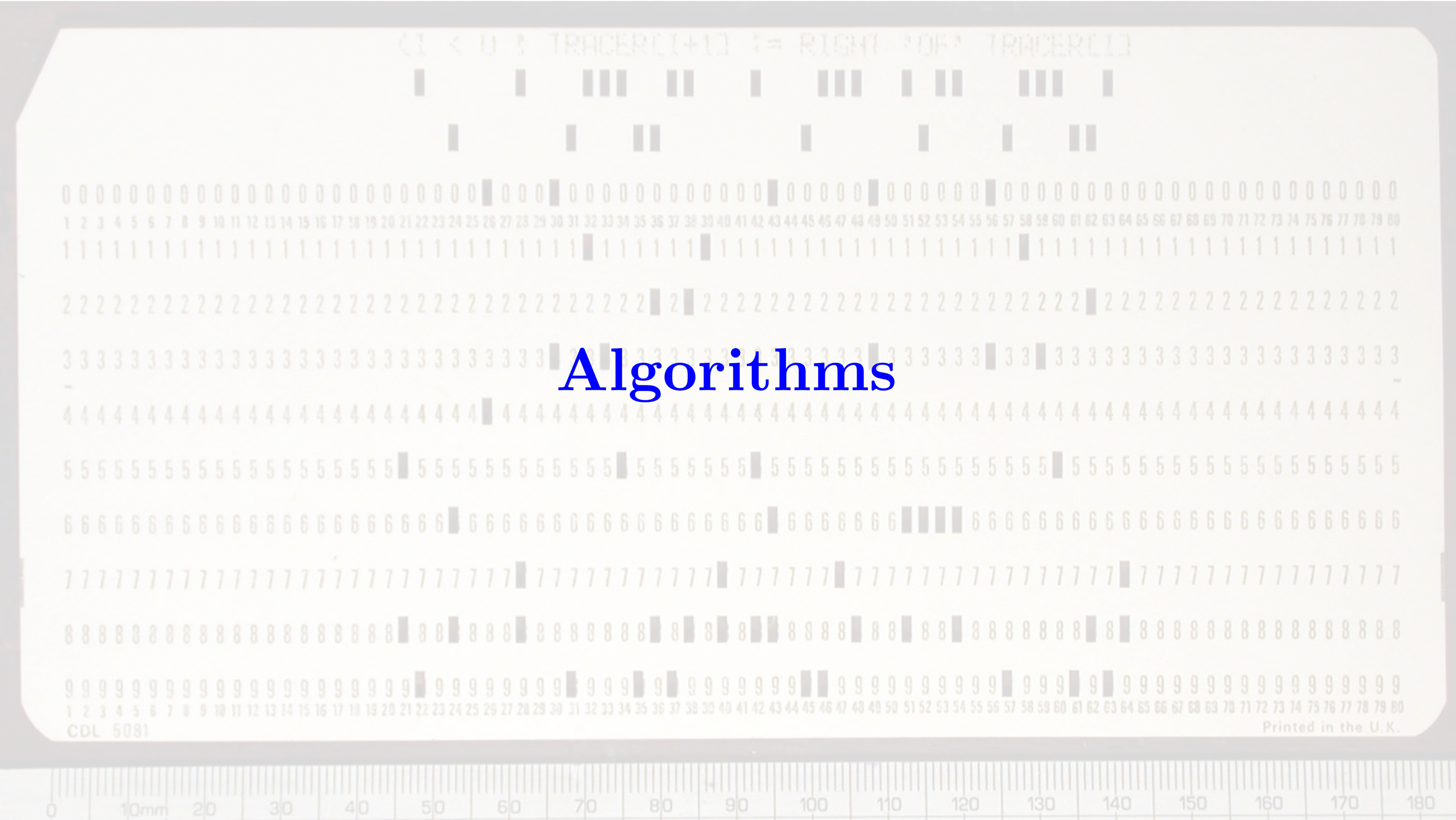
Algorithms



Network Flows

```
Max-Flow
    Initially f(e) = 0 for all e in G
    While there is an s-t path in the residual graph G_f
        Let P be a simple s-t path in G_f
        f' = augment(f, P)
        Update f to f'
        Update the residual grapgh G_f to be G_f'
    EndWhile
    Return f
```

Loop over Paths (called augmenting paths)

```
augment(f, P):
    Let b = bottleneck(P, f)
    For each edge (u,v) P
        If e = (u, v) is a forward edge then
            increase f(e) in G by b
        Else ((u, v) is a backward edge, and let e = (~~u, v~~ (v,u)))
            decrease f(e) in G by b
        Endif
    Endfor
    Return(f)
```

**Here we are constructing the flow. So for backward edges u→v, we need to decrease the flow of v→u by b.

Network Flows

Ford Fulkerson Algorithm.

Repeatedly find a path from s to t in which all edges have some leftover capacity and send flow equal to the capacity of the *bottleneck edge*.

bottleneck edge*: edge with minimum capacity in the augmenting path*

- Requires that all capacities are integers.
- Running time: $O((m+n) \cdot f)$, where f is the value of the flow returned.

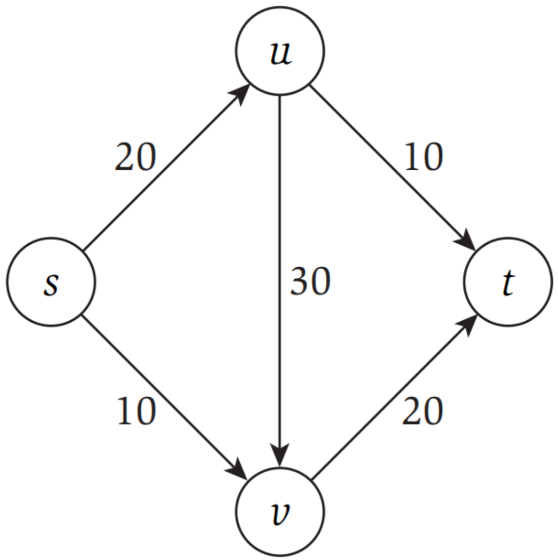
Shortest Path Augmentation

Edmonds-Karp Algorithm.

Always use a shortest $s \rightsquigarrow t$ path (i.e. a path with the minimum no. of edges) in which all edges have non-zero residual capacity as the augmenting path.

The amount of flow sent along the augmenting path is equal to the capacity of a *bottleneck* edge i.e. an edge with the minimum capacity in the path.

How many augmentations will the algorithm do? (total number of loops)



Shortest Path Augmentation

Edmonds-Karp Algorithm.

Always use a shortest $s \rightsquigarrow t$ path (i.e. a path with the minimum no. of edges) in which all edges have non-zero residual capacity as the augmenting path.

The amount of flow sent along the augmenting path is equal to the capacity of a *bottleneck* edge i.e. an edge with the minimum capacity in the path.

Claim. There are at most mn augmentations. *$n = \#$vertices, $m = \#$ edges*

The augmenting path at any stage can be found in $O(m+n)$ time using BFS.

Thus, the overall running time is: $O((m+n) \cdot mn)$.

Definition.

For any vertex v, let $d(v)$ denote the length of the shortest $s \rightsquigarrow v$ path in the *current* residual graph in which each edge has non-zero residual capacity.

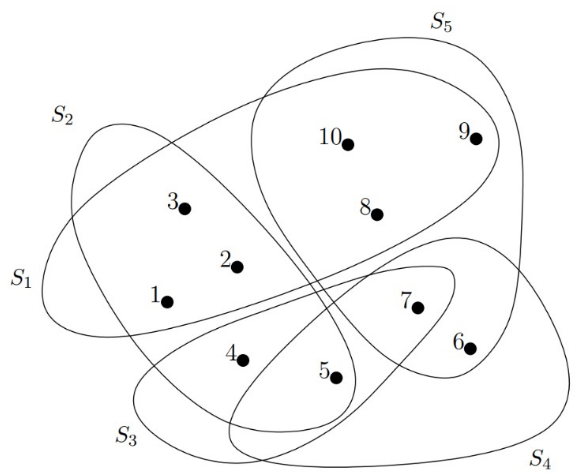
Set Cover

Input. A set X with n elements called the *ground set.*

Subsets: $S_1, S_2, \cdots, S_m \subseteq X$.

Goal. Select the smallest number of sets in $\{S_1, \cdots, S_m\}$ whose union is X.

Example.



$X = \{1, 2, \cdots, 10\}$.

Sets : $S_1, \cdots, S_5$

The smallest collection of these sets that "covers" all elements is $\{S_2, S_5\}$.

Set Cover

Input. A set X with n elements called the *ground set.*

Subsets: $S_1, S_2, \cdots, S_m \subseteq X$.

Goal. Select the smallest number of sets in $\{S_1, \cdots, S_m\}$ whose union is X.

How complex is a brute force approach?

$O(2^m)$

Try a greedy approach to solve the problem

Does your appraoch achieve an optimal solution?

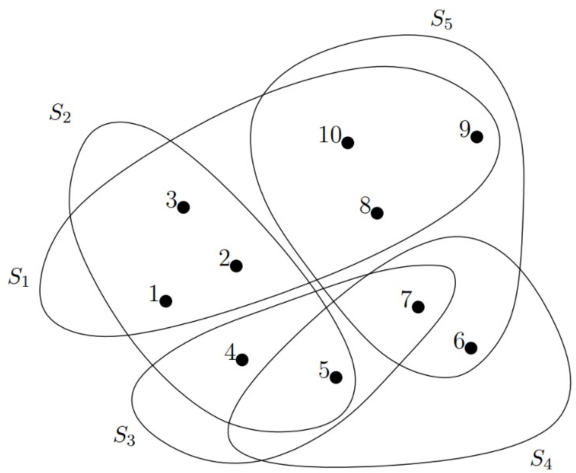
Set Cover

Greedy Algorithm.

We select sets "greedily" one by one.

We always pick the set that covers the maximum number of new elements (i.e., elements not covered by previously picked sets).

What answer do we get for the following input?



1^{st} set: S_1 (6 new elements)

2^{nd} set: S_3 (3 new elements)

3^{rd} set: S_4 (1 new element)

So this algorithm does not give an optimal answer!

Set Cover

Greedy Algorithm.

We select sets "greedily" one by one.

We always pick the set that covers the maximum number of new elements (i.e., elements not covered by previously picked sets).

The set cover problem is an **NP-hard problem**

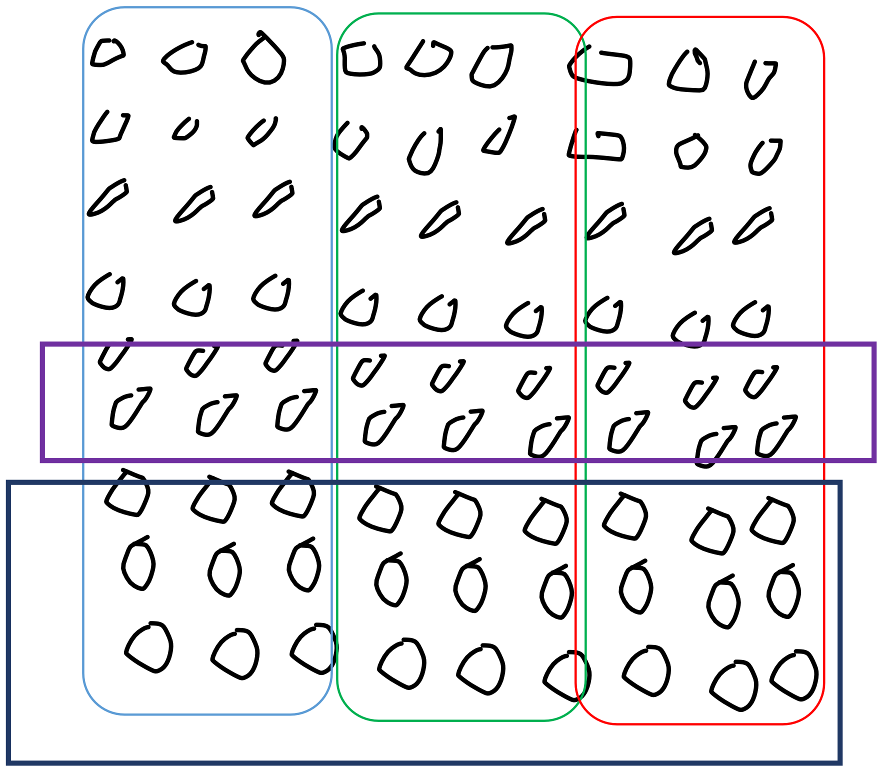
So we have to compromise on something if we want to solve it in **polynomial time**

We this approach we are compromising on correctness

But how good is the solution?

Set Cover

What could the greedy algorithm output?
(use tie breaking for similar cases)



Set Cover

Theorem. The greedy algorithm always returns an answer whose size is at most $\ln n$ times that of the optimal solution. $n = |X|$

Proof. Suppose that the optimal solution has k sets.

Let n_t be the number of uncovered elements in the t^{th} iteration.

Since all elements are uncovered initially $\overset{n_1}{\cancel{n_0}} = n$.

Can be proven by induction on n_t

Since the uncovered elements are covered by the k optimal sets, there is at least one set that covers $\geq n_t/k$ of the these.

Thus, $n_{t+1} \leq n_t - \dfrac{n_t}{k} = n_t\left(1 - \dfrac{1}{k}\right) < n_t \cdot e^{-1/k}$

($\textit{since } 1 - x < e^{-x} \textit{ for } x \neq 0$)

This implies that: $n_t < n \cdot e^{-t/k}$.

For $t = k \ln n$, $n_t < 1$ i.e., $n_t = 0$. $\square$

Under widely believed assumptions, no algorithm has a better guarantee!