Greedy algorithms

CMPSC 465 - Yana Safonova

Cover set

The cover set problem

Problem (Set Cover)

Input:

- *a set* B
- *subsets* $S_1, \ldots, S_m \subseteq B$

Output: *a collection of subsets* $S_{i_1}, \ldots, S_{i_k}$ *s.t.* $\bigcup_{j=1}^{k} S_{i_j} = B$

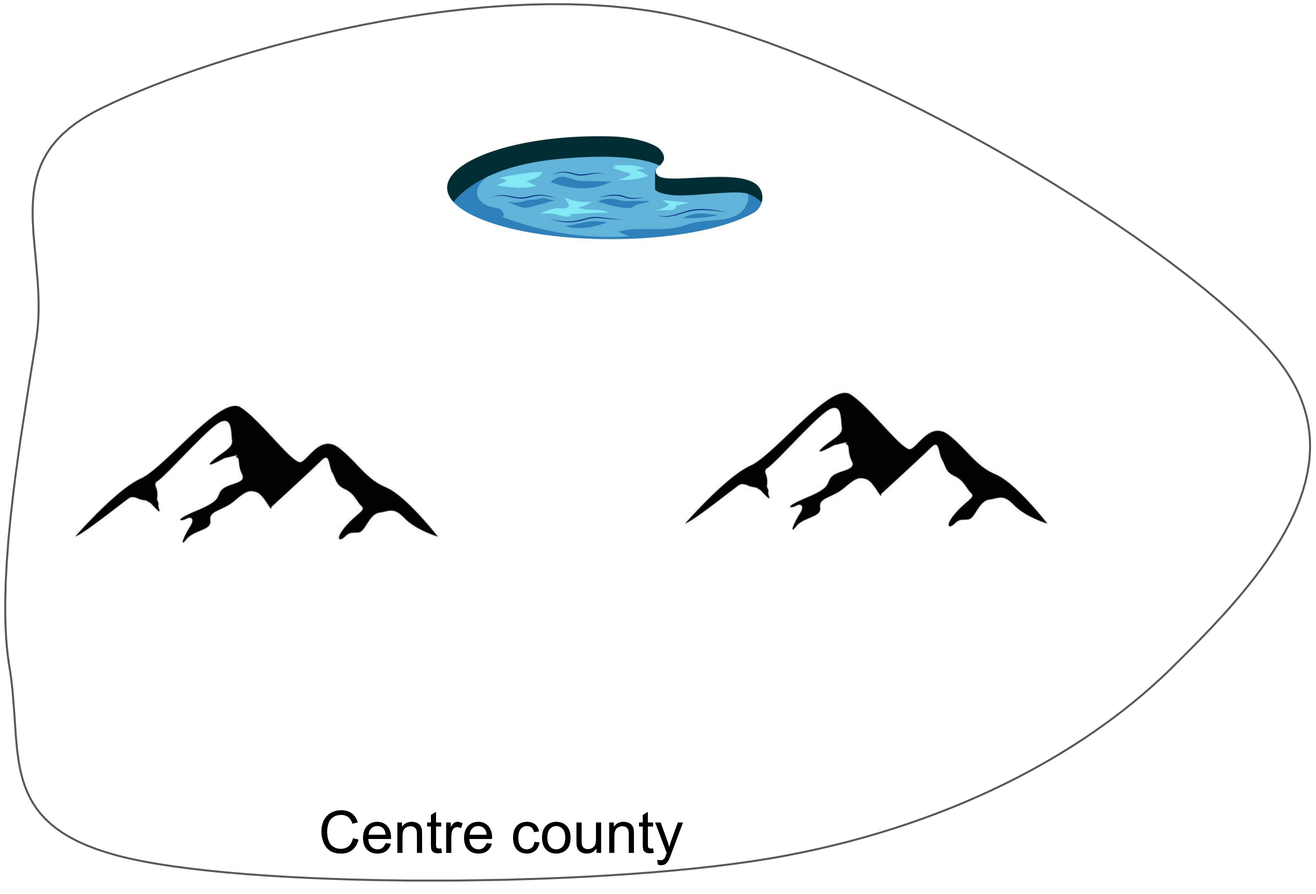
Goal: *minimize the number of selected subsets*

Set cover: example

Each post office can serve 30 miles. How to build the minimum number of posts to serve the Centre county?

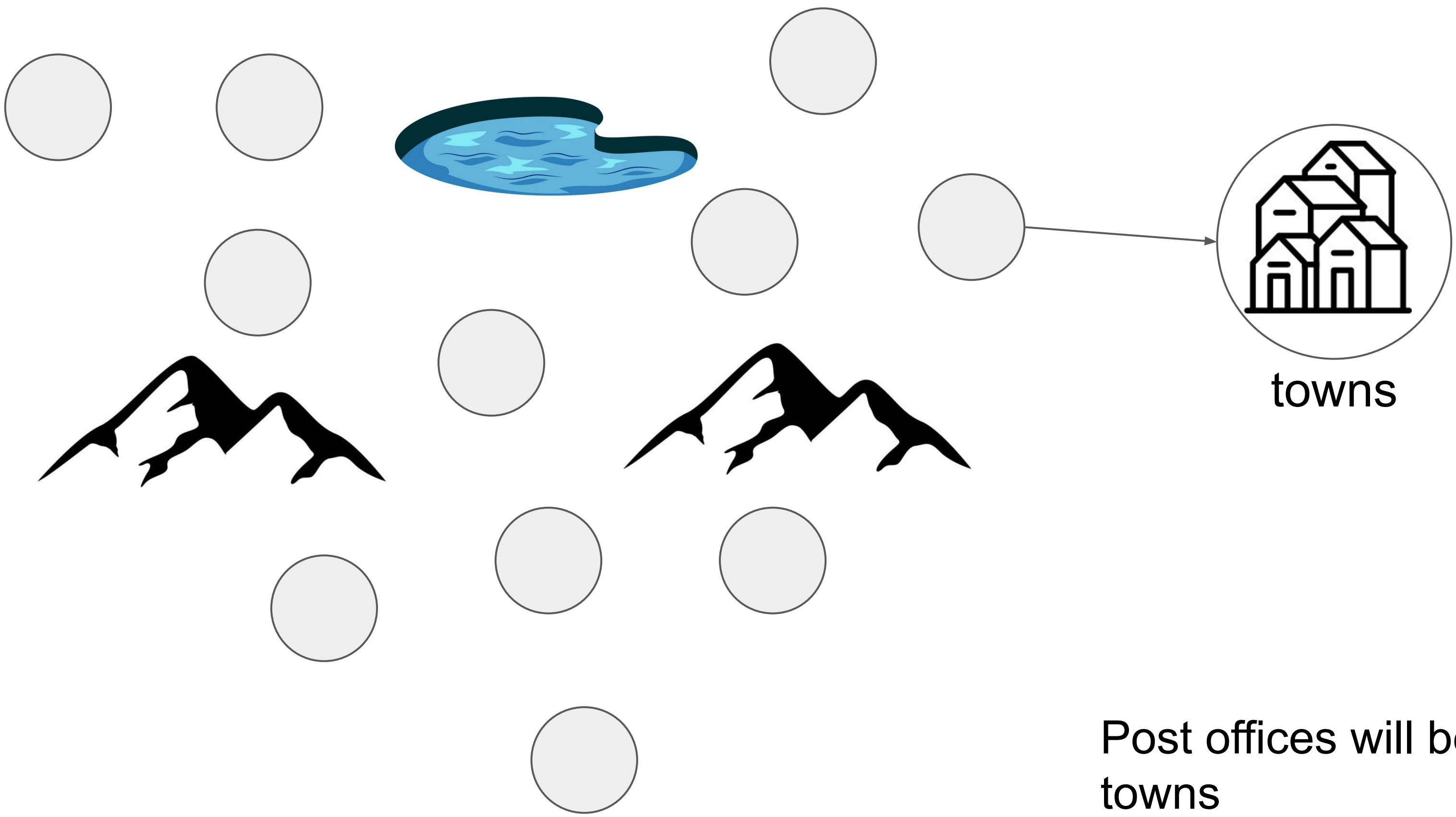
Set cover: example

Each post office can serve 30 miles. How to build the minimum number of posts to serve the Centre county?



Set cover: example

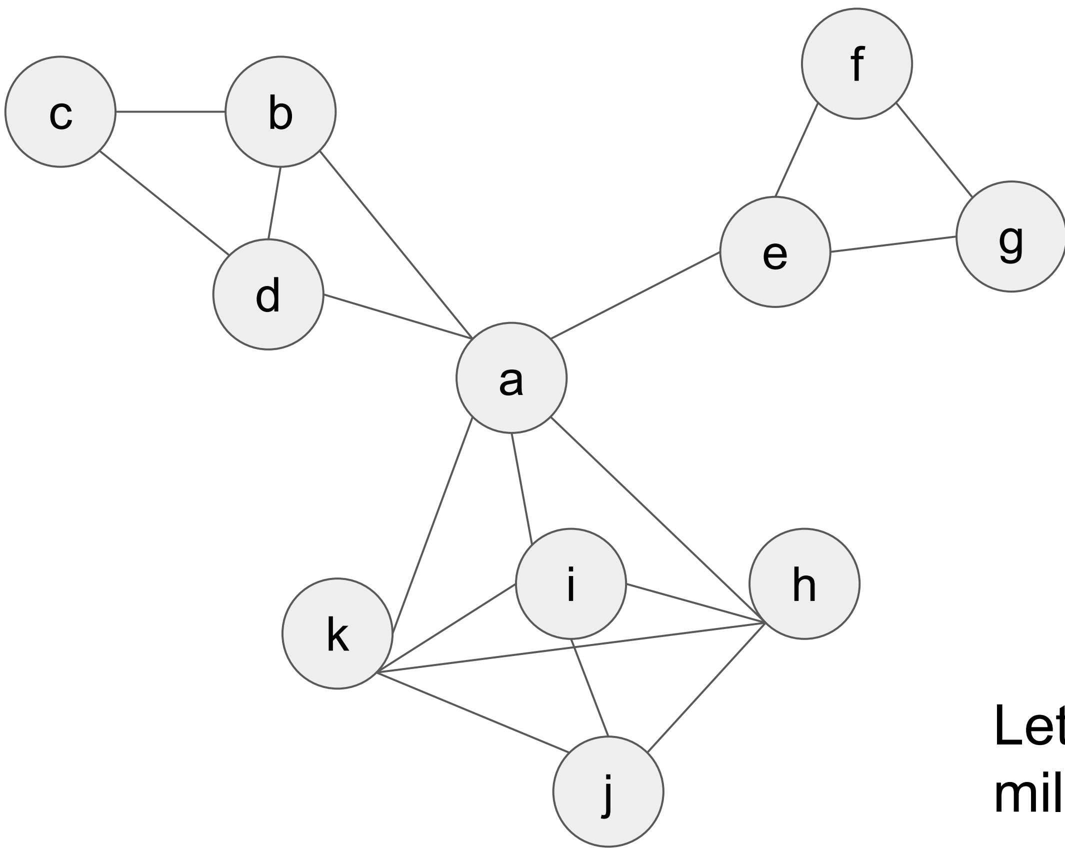
Each post office can serve 30 miles. How to build the minimum number of posts to serve the Centre county?



Post offices will be built in towns

Set cover: example

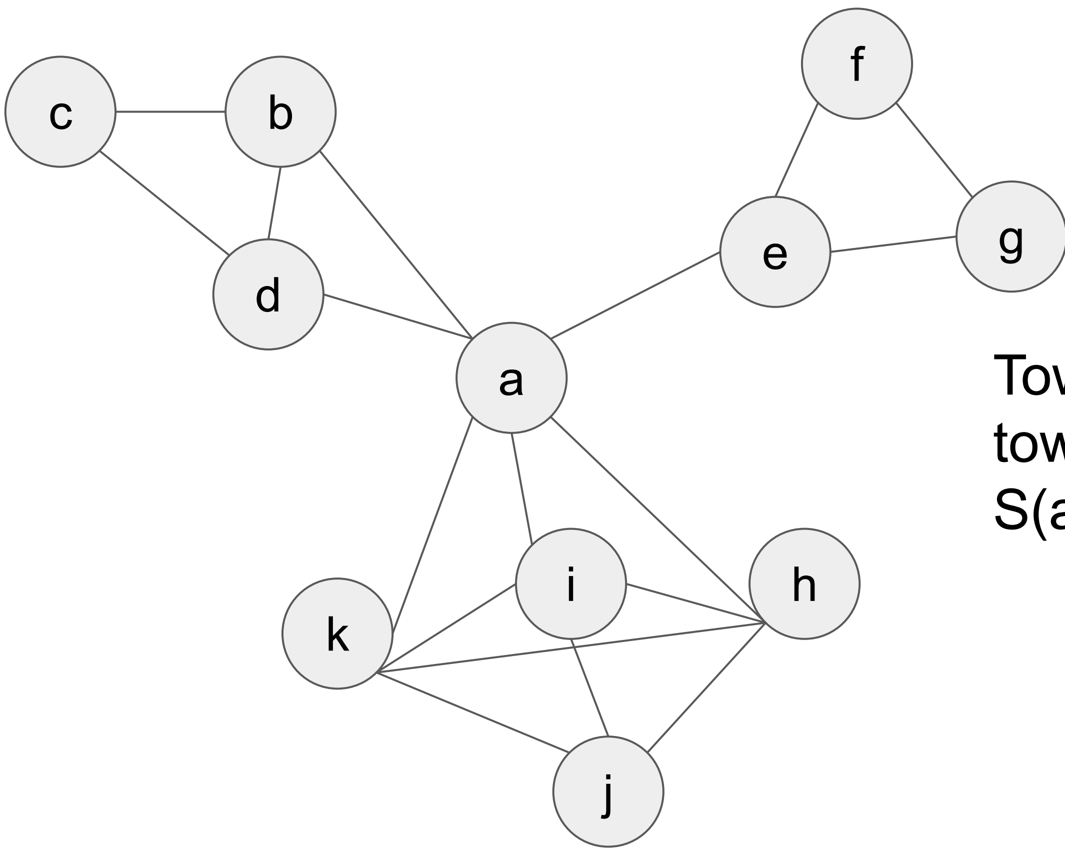
Each post office can serve 30 miles. How to build the minimum number of posts to serve the Centre county?



Let’s connect towns with ≤30 mile distance

Set cover: example

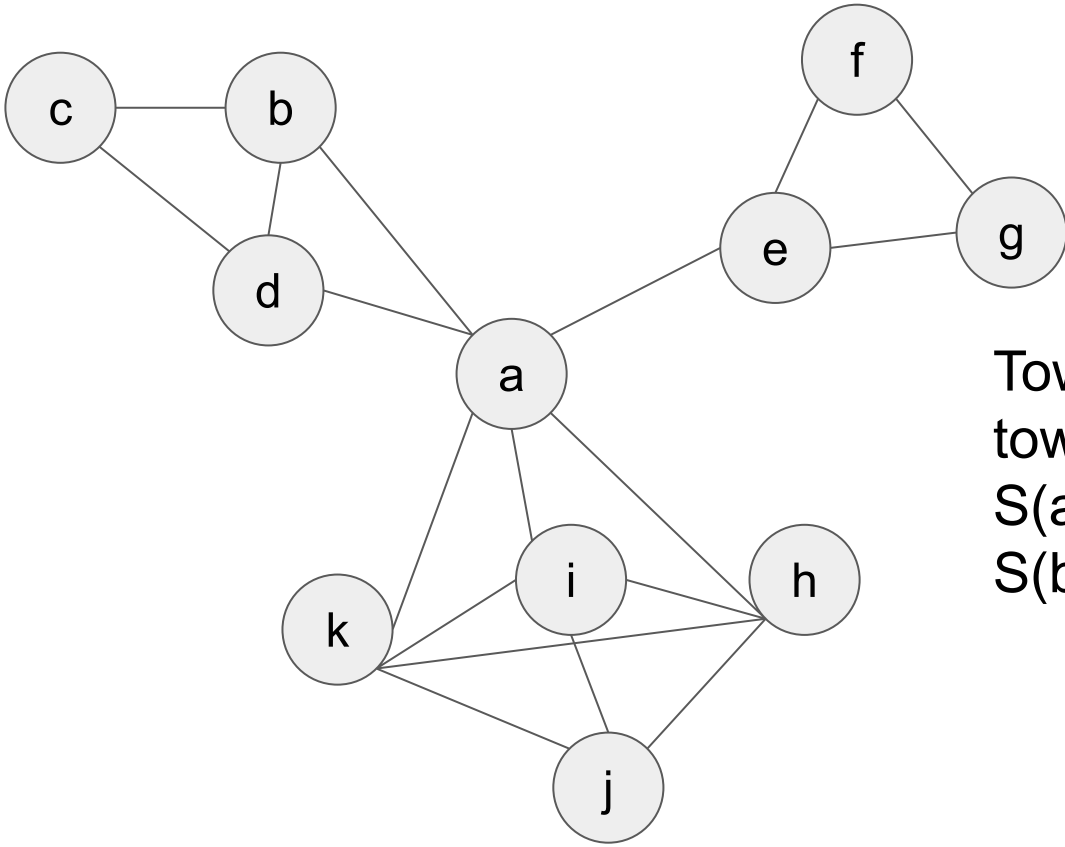
Each post office can serve 30 miles. How to build the minimum number of posts to serve the Centre county?



Towns reachable from each town:
S(a) = {a, b, d, e, k, i, h}

Set cover: example

Each post office can serve 30 miles. How to build the minimum number of posts to serve the Centre county?



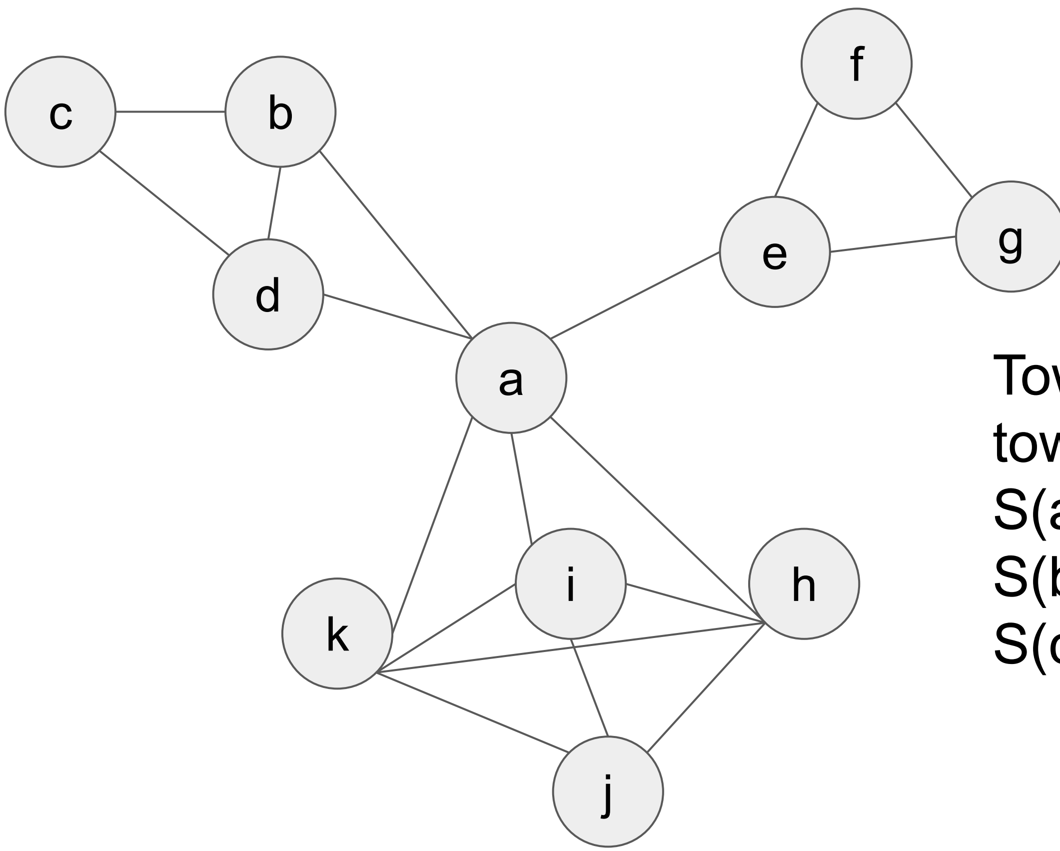
Towns reachable from each town:

S(a) = {a, b, d, e, k, i, h}

S(b) = {a, b, c, d}

Set cover: example

Each post office can serve 30 miles. How to build the minimum number of posts to serve the Centre county?



Towns reachable from each town:

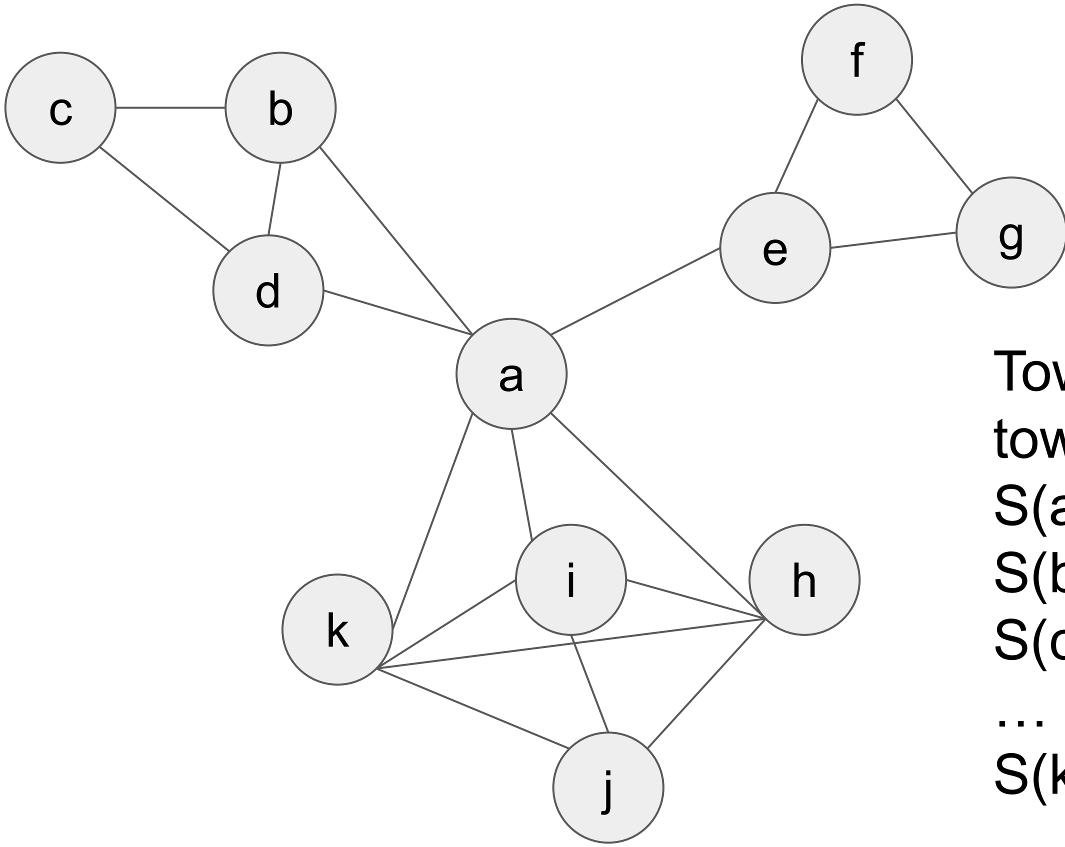
S(a) = {a, b, d, e, k, i, h}

S(b) = {a, b, c, d}

S(c) = {b, c, d}

Set cover: example

Each post office can serve 30 miles. How to build the minimum number of posts to serve the Centre county?

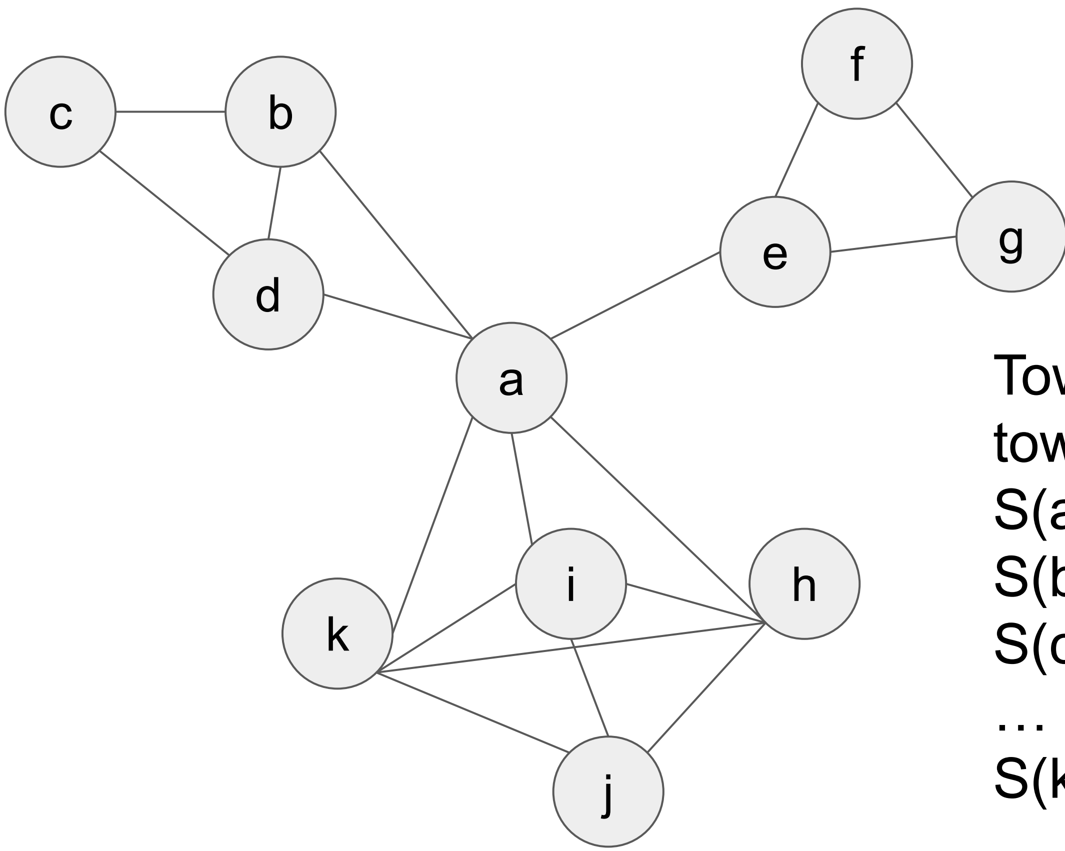


Towns reachable from each town:

S(a) = {a, b, d, e, k, i, h}
S(b) = {a, b, c, d}
S(c) = {b, c, d}
…
S(k) = {a, h, i, j, k}

Set cover: example

Finding the cover set for S(a), …, S(k) will solve the post office problem

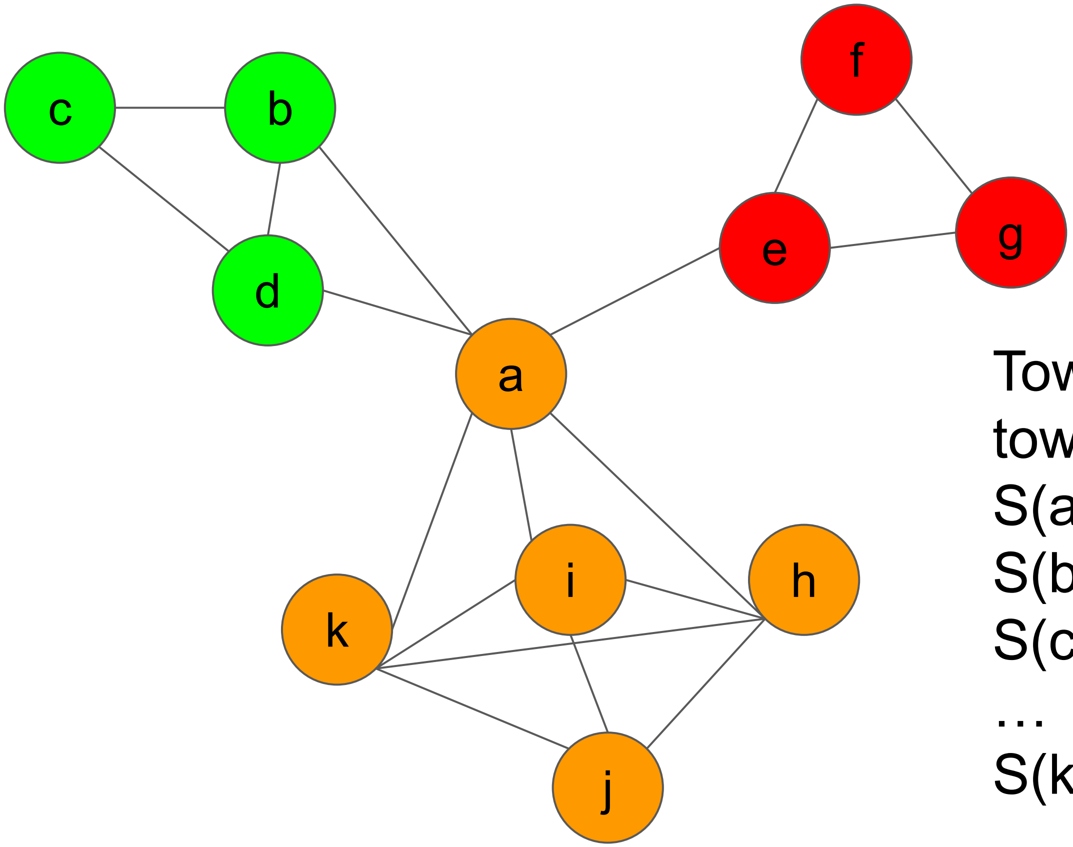


Towns reachable from each town:

S(a) = {a, b, d, e, k, i, h}
S(b) = {a, b, c, d}
S(c) = {b, c, d}
…
S(k) = {a, h, i, j, k}

Set cover: example

The optimal solution:



Towns reachable from each town:

S(a) = {a, b, d, e, k, i, h}

S(b) = {a, b, c, d}

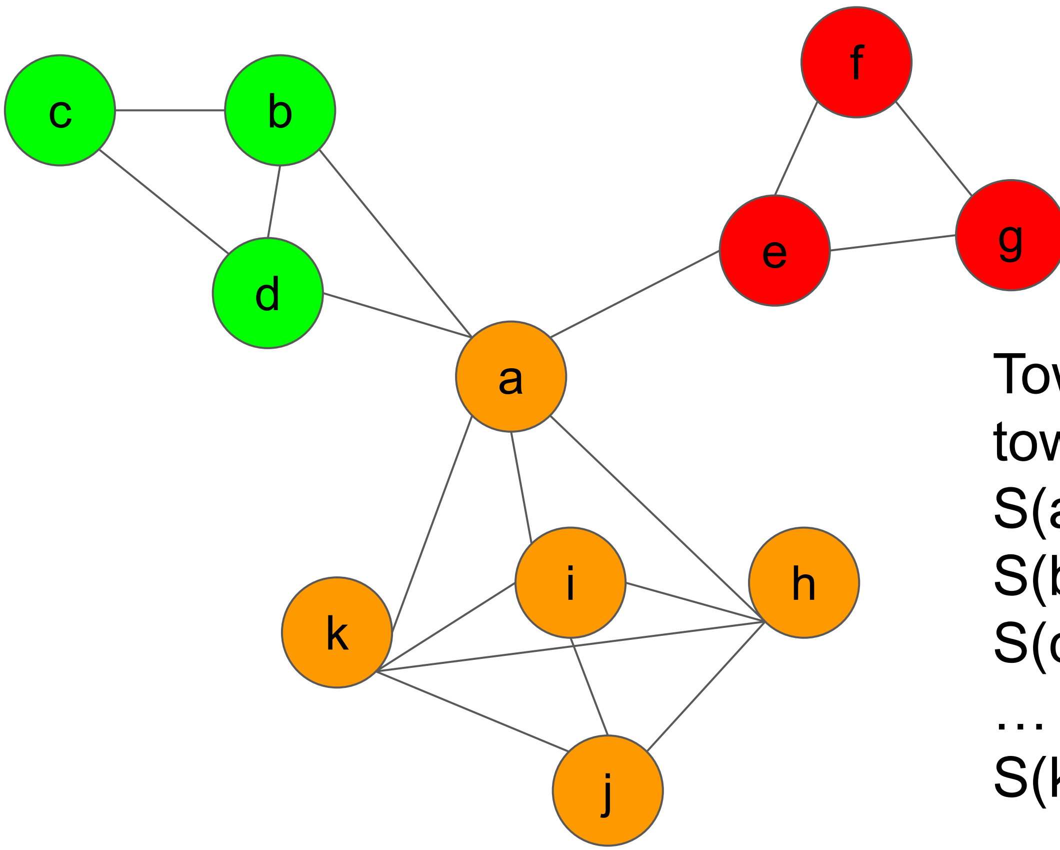
S(c) = {b, c, d}

…

S(k) = {a, h, i, j, k}

Set cover: example

The optimal solution: S(b), S(f), S(i)



Towns reachable from each town:

S(a) = {a, b, d, e, k, i, h}

S(b) = {a, b, c, d}

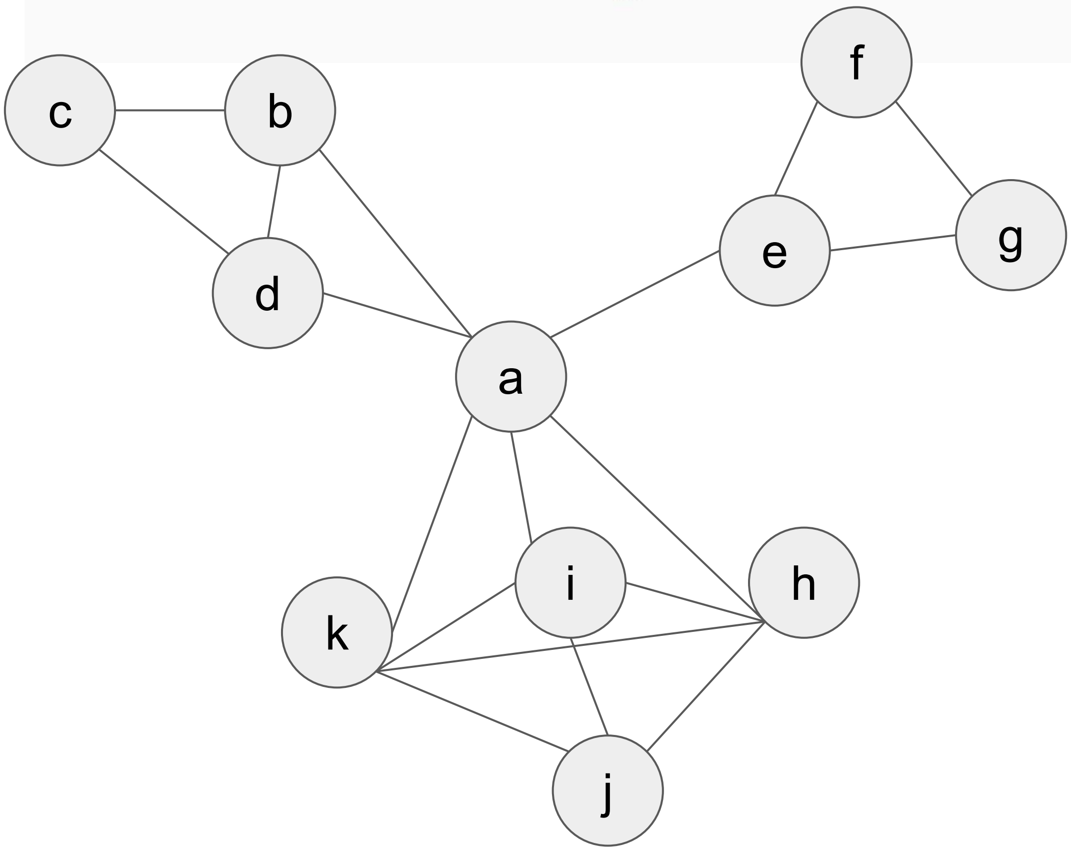
S(c) = {b, c, d}

…

S(k) = {a, h, i, j, k}

The greedy algorithm

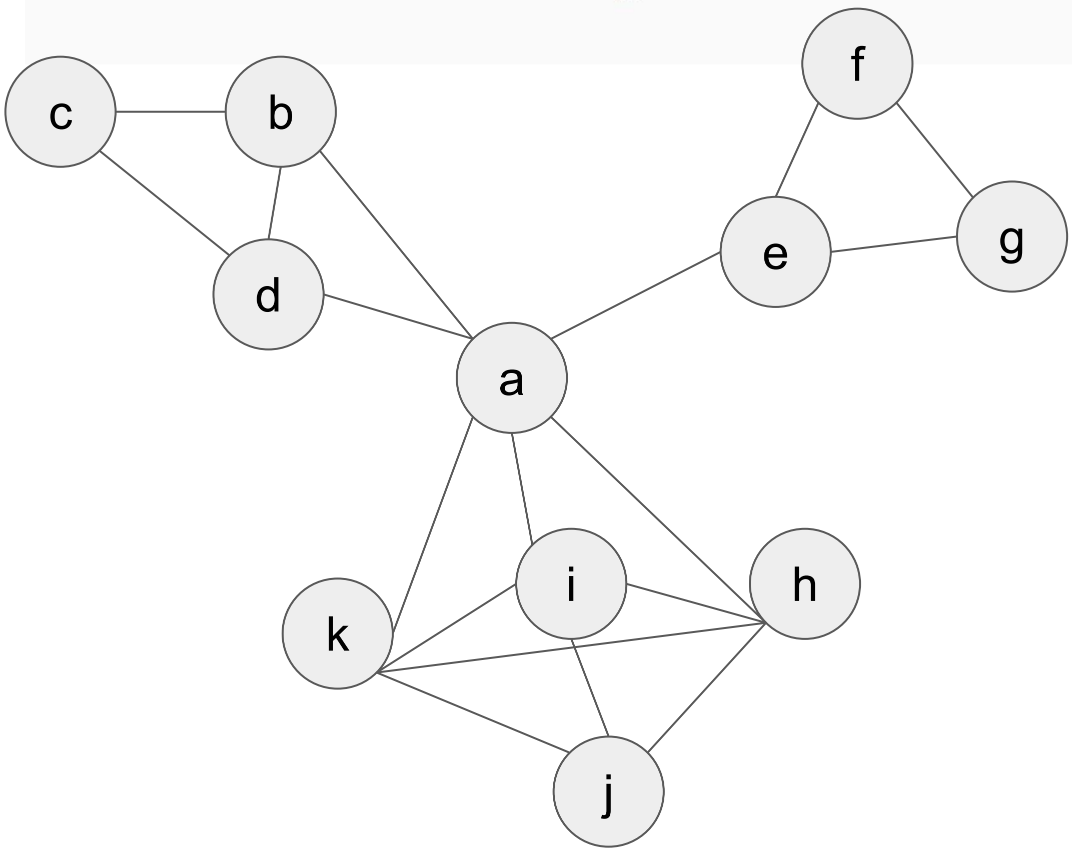
Greedy heuristic: choose the next subset with the most number of uncovered items, until B gets covered



S(a) = {a, b, d, e, k, i, h}
S(b) = {a, b, c, d}
S(c) = {b, c, d}
S(d) = {d, b, c, a}
S(e) = {e, a, f, g}
S(f) = {f, e, g}
S(g) = {g, e, ,f}
S(i) = {i, a, h, j, k}
S(j) = {j, h, i, k}
S(h) = {h, a, i, j, k}
S(k) = {a, h, i, j, k}

The greedy algorithm

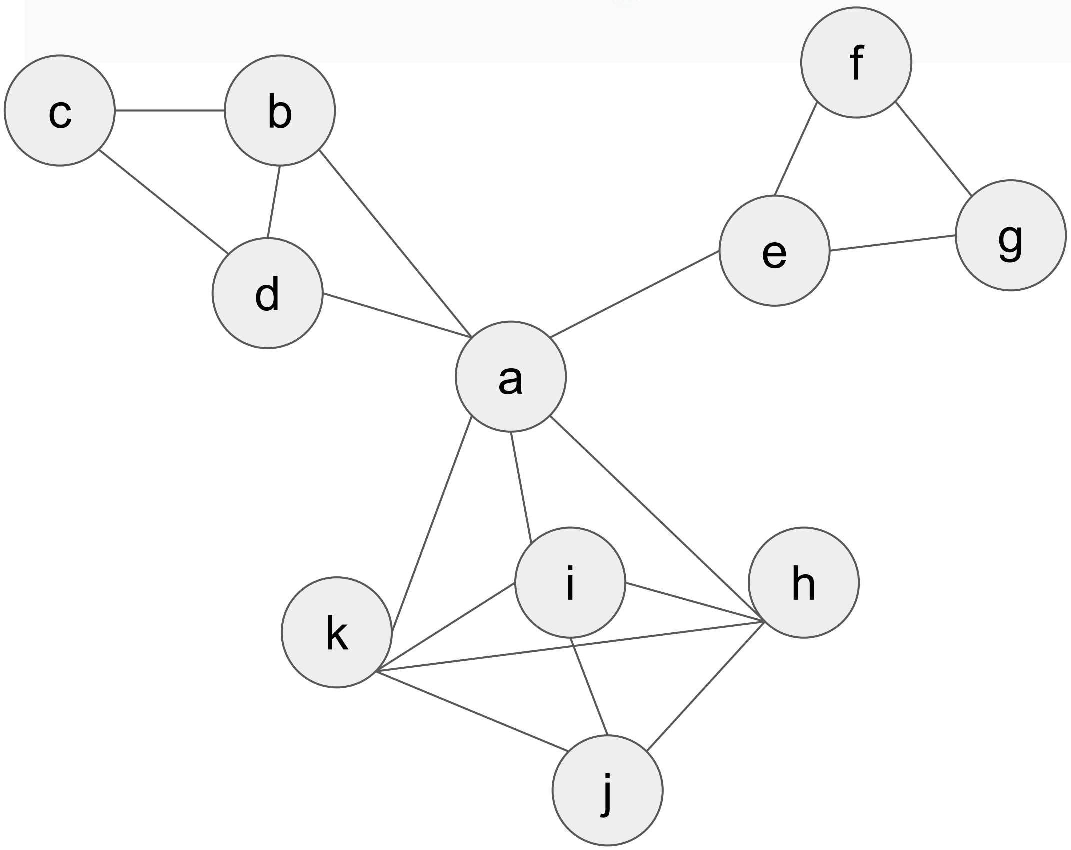
Greedy heuristic: choose the next subset with the most number of uncovered items, until B gets covered



S(a) = {a, b, d, e, k, i, h}
S(b) = {a, b, c, d}
S(c) = {b, c, d}
S(d) = {d, b, c, a}
S(e) = {e, a, f, g}
S(f) = {f, e, g}
S(g) = {g, e, ,f}
S(i) = {i, a, h, j, k}
S(j) = {j, h, i, k}
S(h) = {h, a, i, j, k}
S(k) = {a, h, i, j, k}

The greedy algorithm

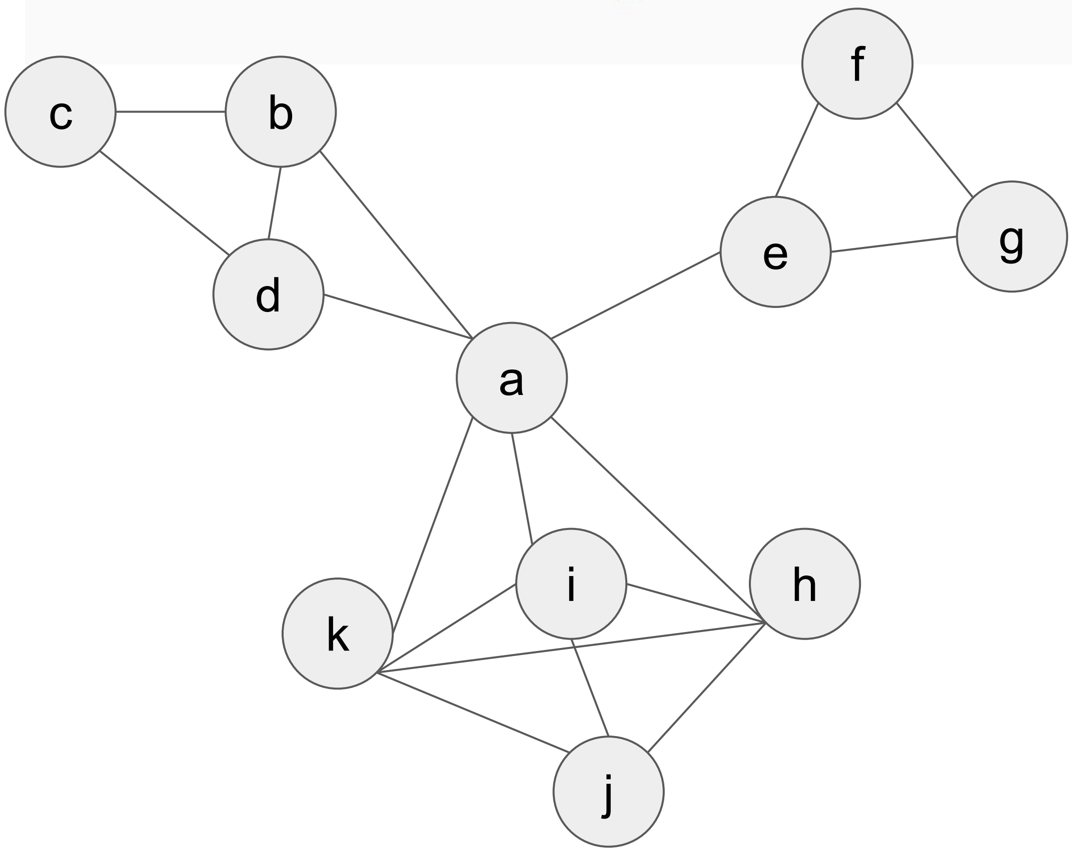
Greedy heuristic: choose the next subset with the most number of uncovered items, until B gets covered



S(a) = {a, b, d, e, k, i, h}
S(b) = {a, b, c, d}
S(c) = {b, c, d}
S(d) = {d, b, c, a}
S(e) = {e, a, f, g}
S(f) = {f, e, g}
S(g) = {g, e, ,f}
S(i) = {i, a, h, j, k}
S(j) = {j, h, i, k}
S(h) = {h, a, i, j, k}
S(k) = {a, h, i, j, k}

The greedy algorithm

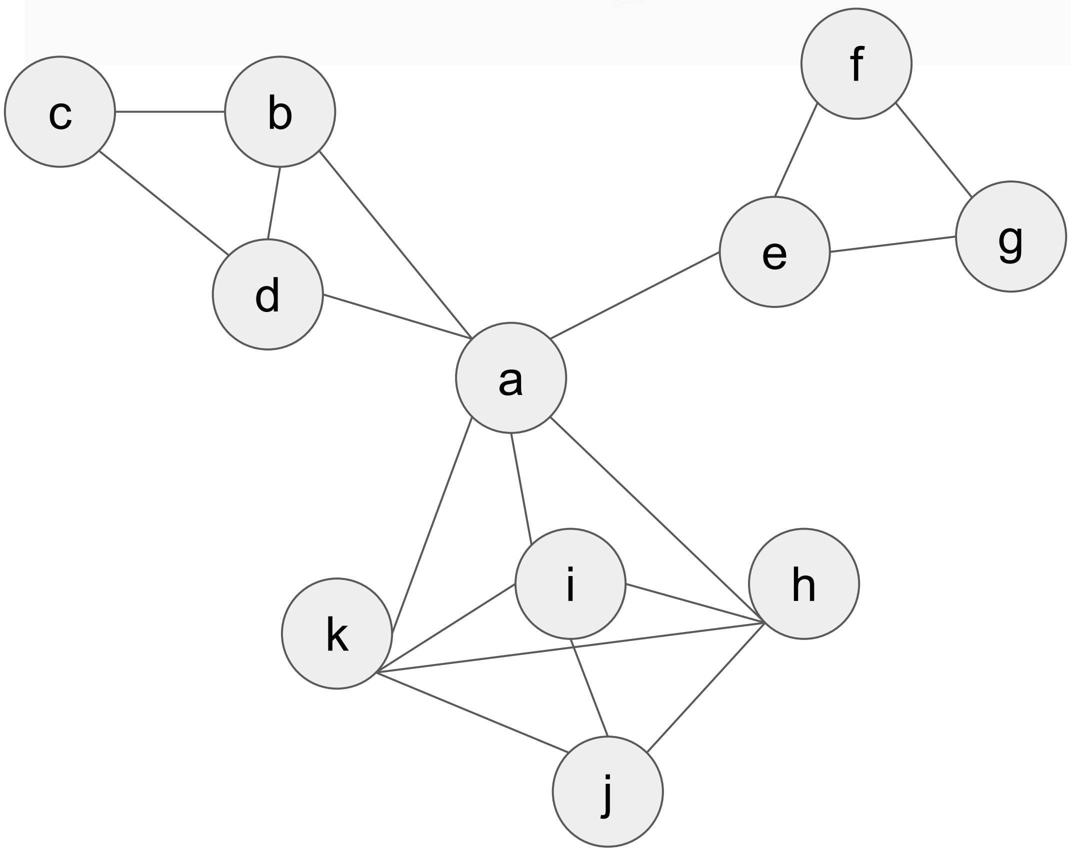
Greedy heuristic: choose the next subset with the most number of uncovered items, until B gets covered



S(a) = {a, b, d, e, k, i, h}
S(b) = {a, b, c, d}
S(c) = {b, c, d}
S(d) = {d, b, c, a}
S(e) = {e, a, f, g}
S(f) = {f, e, g}
S(g) = {g, e, ,f}
S(i) = {i, a, h, j, k}
S(j) = {j, h, i, k}
S(h) = {h, a, i, j, k}
S(k) = {a, h, i, j, k}

The greedy algorithm

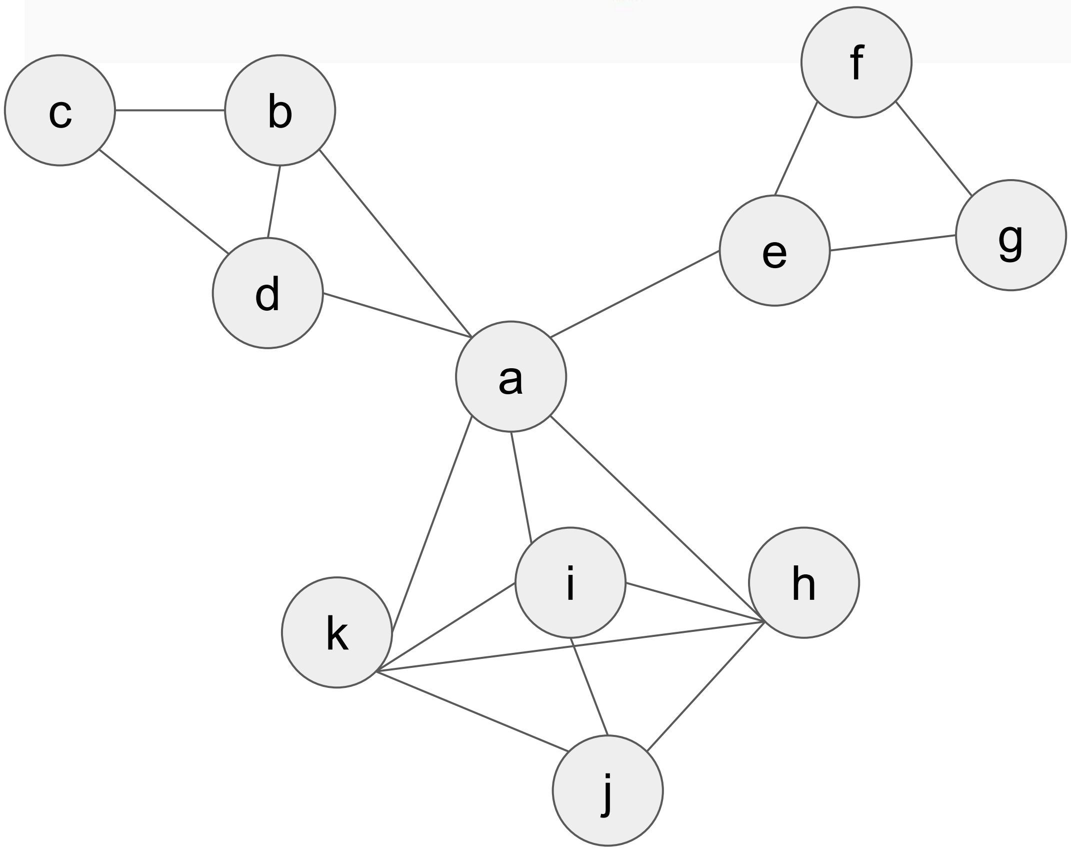
Greedy heuristic: choose the next subset with the most number of uncovered items, until B gets covered



S(a) = {a, b, d, e, k, i, h}
S(b) = {a, b, c, d}
S(c) = {b, c, d}
S(d) = {d, b, c, a}
S(e) = {e, a, f, g}
S(f) = {f, e, g}
S(g) = {g, e, ,f}
S(i) = {i, a, h, j, k}
S(j) = {j, h, i, k}
S(h) = {h, a, i, j, k}
S(k) = {a, h, i, j, k}

The greedy algorithm

Greedy heuristic: choose the next subset with the most number of uncovered items, until B gets covered



S(a) = {a, b, d, e, k, i, h}

S(b) = {a, b, c, d}

S(c) = {b, c, d}

S(d) = {d, b, c, a}

S(e) = {e, a, f, g}

S(f) = {f, e, g}

S(g) = {g, e, ,f}

S(i) = {i, a, h, j, k}

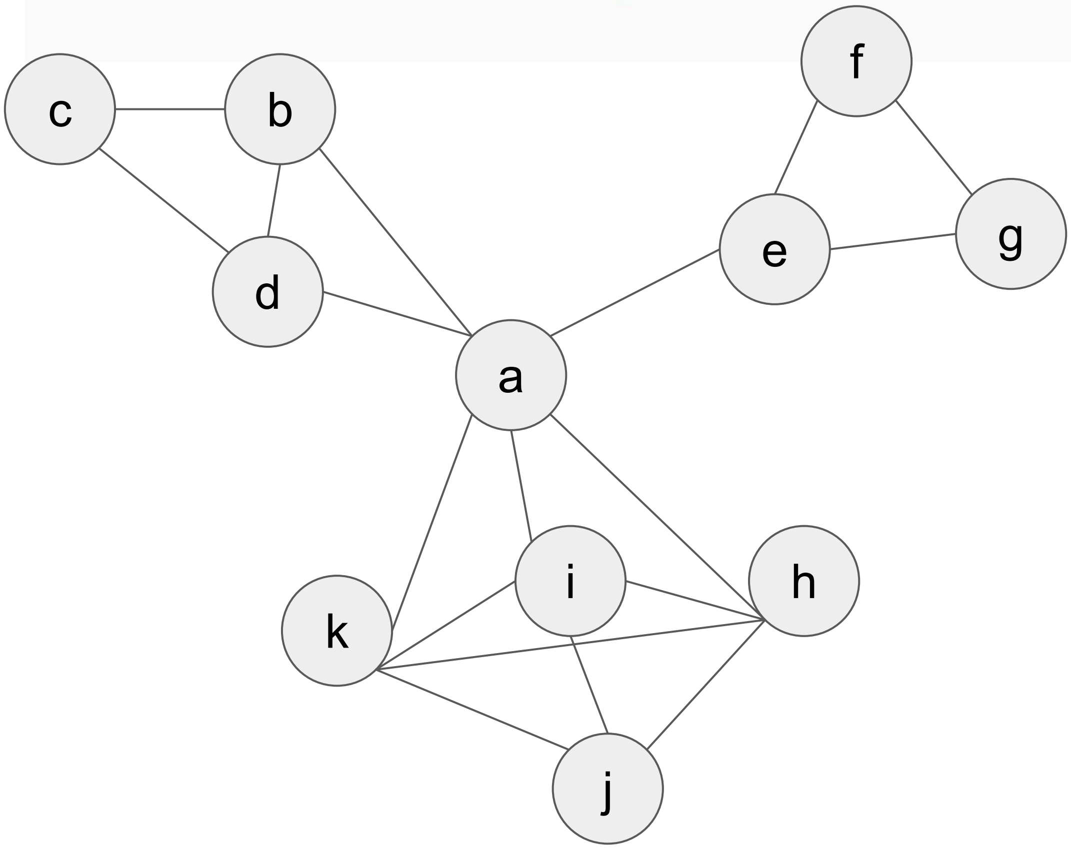
S(j) = {j, h, i, k}

S(h) = {h, a, i, j, k}

S(k) = {a, h, i, j, k}

The greedy algorithm

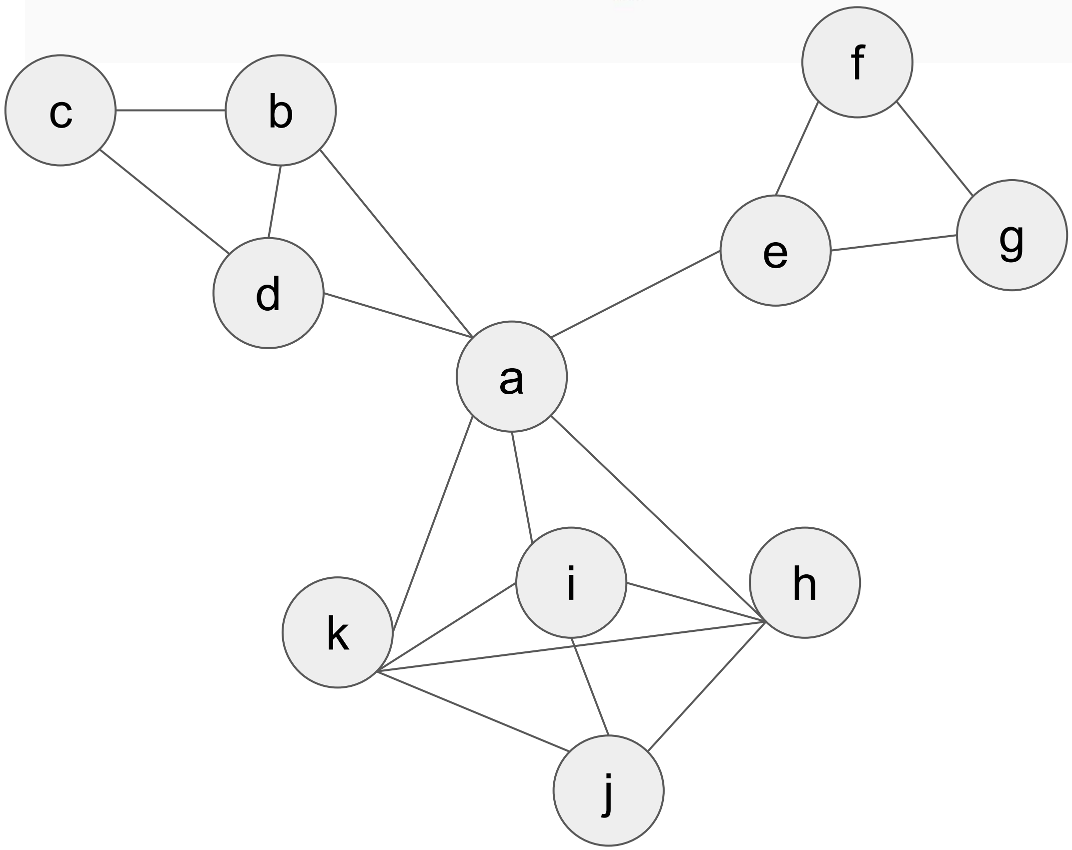
Greedy heuristic: choose the next subset with the most number of uncovered items, until B gets covered



S(a) = {a, b, d, e, k, i, h}
S(b) = {a, b, **c,** d**}**
S(c) = {b, c, d}
S(d) = {d, b, c, a}
S(e) = {e, a, **f, g}**
S(f) = {f, e, g}
S(g) = {g, e, ,f}
S(i) = {i, a, h, j, k}
S(j) = {j, h, i, k}
S(h) = {h, a, i, j, k}
S(k) = {a, h, i, j, k}

The greedy algorithm

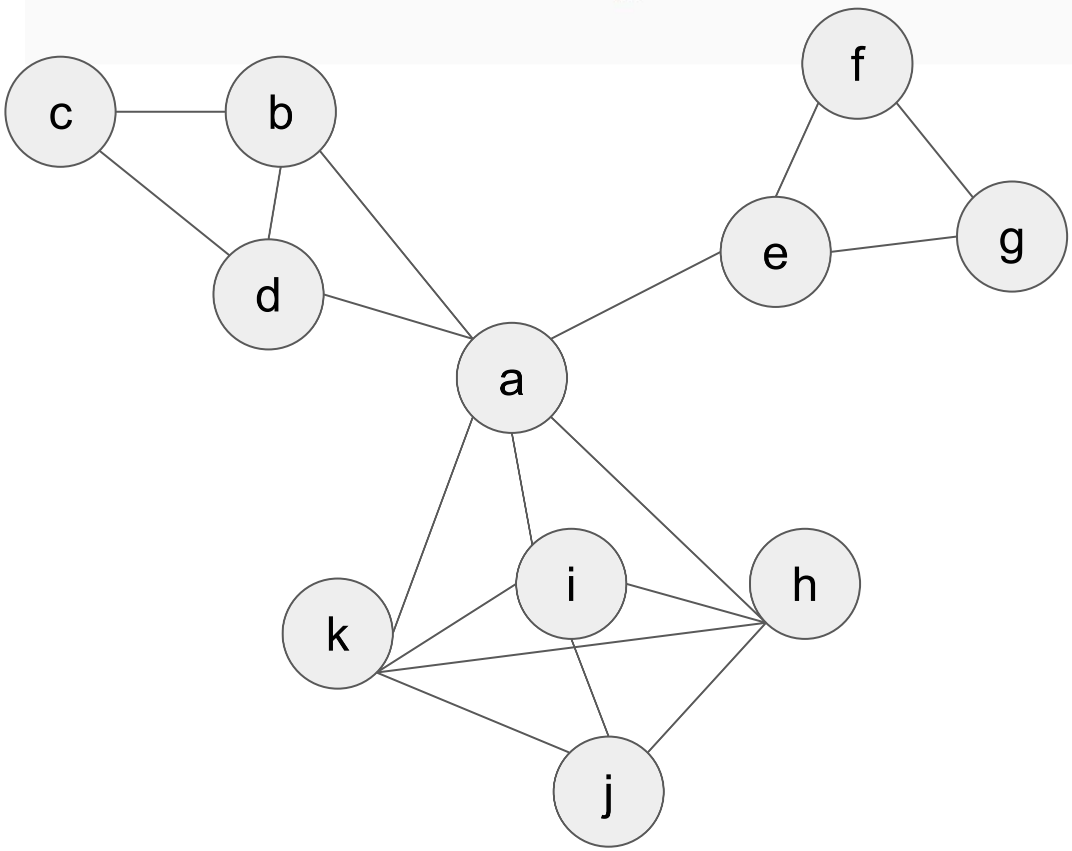
Greedy heuristic: choose the next subset with the most number of uncovered items, until B gets covered



S(a) = {a, b, d, e, k, i, h}
S(b) = {a, b, c, d}
S(c) = {b, c, d}
S(d) = {d, b, c, a}
S(e) = {e, a, f, g}
S(f) = {f, e, g}
S(g) = {g, e, ,f}
S(i) = {i, a, h, j, k}
S(j) = {j, h, i, k}
S(h) = {h, a, i, j, k}
S(k) = {a, h, i, j, k}

The greedy algorithm

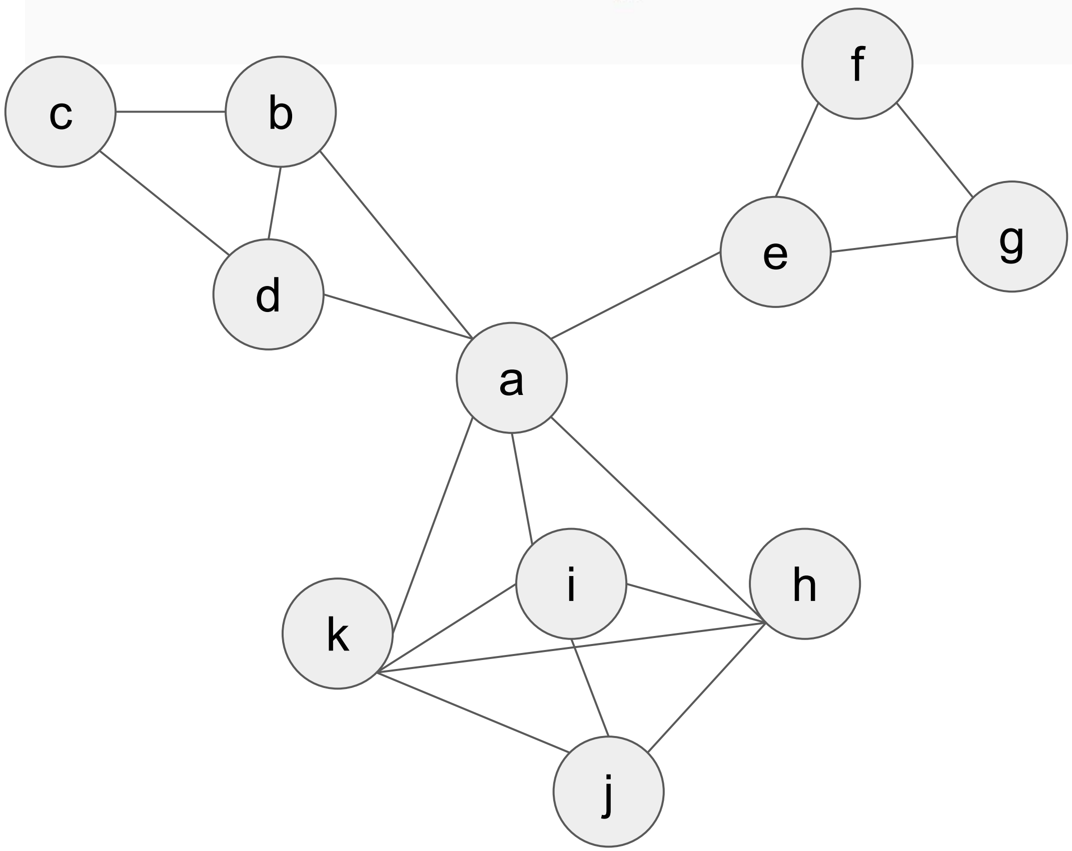
Greedy heuristic: choose the next subset with the most number of uncovered items, until B gets covered



S(a) = {a, b, d, e, k, i, h}
S(b) = {a, b, c, d}
S(c) = {b, c, d}
S(d) = {d, b, c, a}
S(e) = {e, a, f, g}
S(f) = {f, e, g}
S(g) = {g, e, ,f}
S(i) = {i, a, h, j, k}
S(j) = {j, h, i, k}
S(h) = {h, a, i, j, k}
S(k) = {a, h, i, j, k}

The greedy algorithm

Greedy heuristic: choose the next subset with the most number of uncovered items, until B gets covered



S(a) = {a, b, d, e, k, i, h}
S(b) = {a, b, c, d}

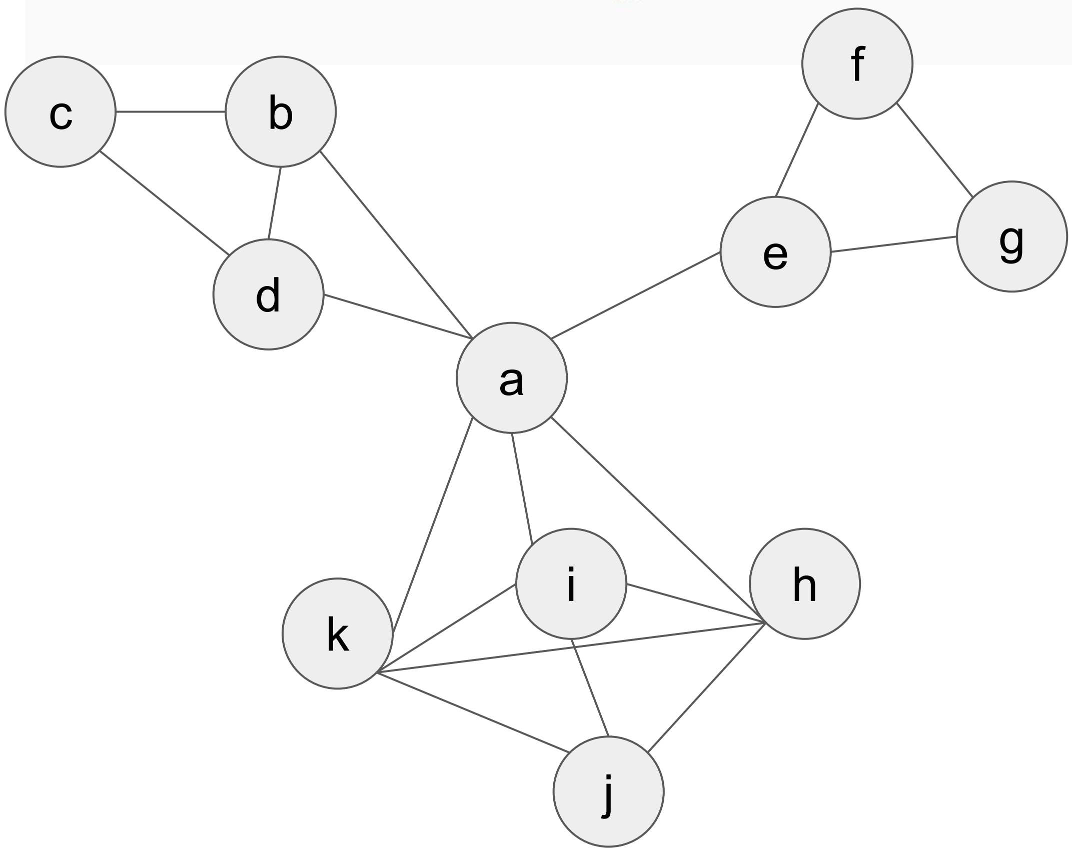
S(e) = {e, a, f, g}

S(i) = {i, a, h, j, k}

$k_{GREEDY} = 4$

The greedy algorithm

Greedy heuristic: choose the next subset with the most number of uncovered items, until B gets covered



S(a) = {a, b, d, e, k, i, h}
S(b) = {a, b, c, d}

S(e) = {e, a, f, g}

S(i) = {i, a, h, j, k}

$k_{GREEDY} = 4$
$k_{OPT} = 3$

The optimal algorithm

S1	**S2**	**S3**	**S4**	**Cover all elements?**	**# sets**
0	0	0	0	No	0
0	0	0	1	No	1
0	0	1	0	No	1
0	0	1	1	Yes	2
0	1	0	0	Yes	1
0	1	0	1	Yes	2
…	…	…	…	…	…
1	1	1	1	Yes	4

The optimal algorithm

S1	S2	S3	S4	Cover all elements?	# sets
0	0	0	0	No	0
0	0	0	1	No	1
0	0	1	0	No	1
0	0	1	1	Yes	2
0	1	0	0	Yes	1 (!!!)
0	1	0	1	Yes	2
…	…	…	…	…	…
1	1	1	1	Yes	4

$O(2^N)$ = exponential running time

Greedy vs optimal algorithms for the cover set problem

Although the greedy solution is not optimal, but it's not off by much

Theorem

Assume $|B| = n$ and the optimal solution uses k subsets. Then the greedy algorithm uses at most $k \ln(n)$ subsets

Proof

Proof: Let n_t be the number of elements not covered by the greedy algorithm after t iterations.

Proof

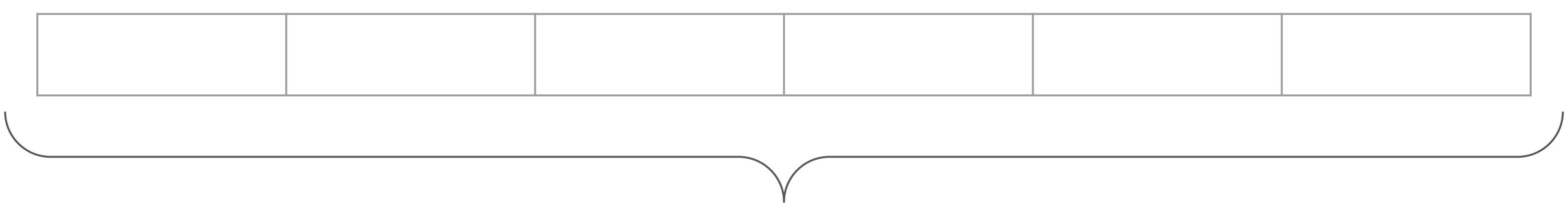
Proof: Let n_t be the number of elements not covered by the greedy algorithm after t iterations. These remaining n_t elements are covered by the optimal k subsets.

Proof

Proof: Let n_t be the number of elements not covered by the greedy algorithm after t iterations. These remaining n_t elements are covered by the optimal k subsets. So some subsets has $\geq \frac{n_t}{k}$ of these uncovered elements, and the greedy algorithm will pick a set of size at least $\frac{n_t}{k}$.

Proof

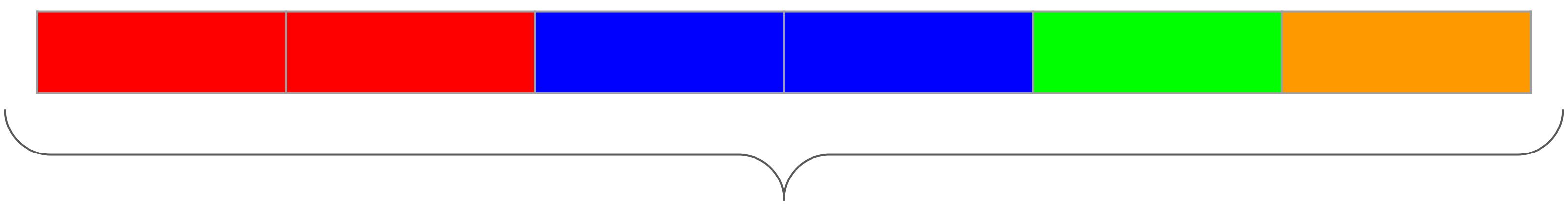
Proof: Let n_t be the number of elements not covered by the greedy algorithm after t iterations. These remaining n_t elements are covered by the optimal k subsets. So some subsets has $\geq \frac{n_t}{k}$ of these uncovered elements, and the greedy algorithm will pick a set of size at least $\frac{n_t}{k}$.



$n_t = 6, \quad k = 4$

Proof

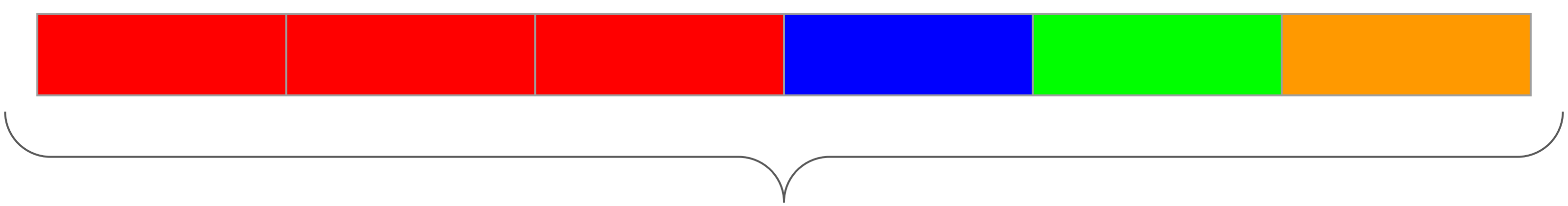
Proof: Let n_t be the number of elements not covered by the greedy algorithm after t iterations. These remaining n_t elements are covered by the optimal k subsets. So some subsets has $\geq \frac{n_t}{k}$ of these uncovered elements, and the greedy algorithm will pick a set of size at least $\frac{n_t}{k}$.



$n_t = 6, \quad k = 4$

Proof

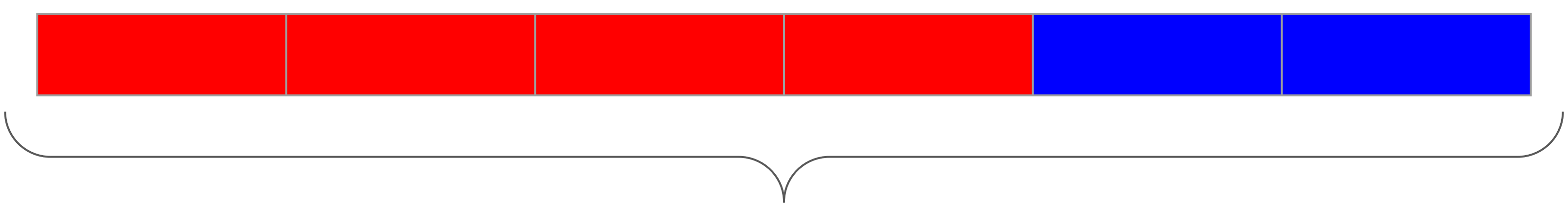
Proof: Let n_t be the number of elements not covered by the greedy algorithm after t iterations. These remaining n_t elements are covered by the optimal k subsets. So some subsets has $\geq \frac{n_t}{k}$ of these uncovered elements, and the greedy algorithm will pick a set of size at least $\frac{n_t}{k}$.



$n_t = 6, \quad k = 4$

Proof

Proof: Let n_t be the number of elements not covered by the greedy algorithm after t iterations. These remaining n_t elements are covered by the optimal k subsets. So some subsets has $\geq \frac{n_t}{k}$ of these uncovered elements, and the greedy algorithm will pick a set of size at least $\frac{n_t}{k}$.



$n_t = 6,\quad k = 4$

Proof

Proof: Let n_t be the number of elements not covered by the greedy algorithm after t iterations. These remaining n_t elements are covered by the optimal k subsets. So some subsets has $\geq \frac{n_t}{k}$ of these uncovered elements, and the greedy algorithm will pick a set of size at least $\frac{n_t}{k}$.
So, $n_{t+1} \leq n_t - \frac{n_t}{k} = n_t \left(1 - \frac{1}{k}\right)$

Proof

Proof: Let n_t be the number of elements not covered by the greedy algorithm after t iterations. These remaining n_t elements are covered by the optimal k subsets. So some subsets has $\geq \frac{n_t}{k}$ of these uncovered elements, and the greedy algorithm will pick a set of size at least $\frac{n_t}{k}$.
So, $n_{t+1} \leq n_t - \frac{n_t}{k} = n_t \left(1 - \frac{1}{k}\right)$

Repeatedly applying this:

$$n_t \leq n_{t-1}\left(1 - \frac{1}{k}\right) \leq n_{t-2}\left(1 - \frac{1}{k}\right)^2 \leq \cdots \leq n_0\left(1 - \frac{1}{k}\right)^t = n\left(1 - \frac{1}{k}\right)^t$$

Proof

Proof: Let n_t be the number of elements not covered by the greedy algorithm after t iterations. These remaining n_t elements are covered by the optimal k subsets. So some subsets has $\geq \frac{n_t}{k}$ of these uncovered elements, and the greedy algorithm will pick a set of size at least $\frac{n_t}{k}$.
So, $n_{t+1} \leq n_t - \frac{n_t}{k} = n_t \left(1 - \frac{1}{k}\right)$

Repeatedly applying this:

$$n_t \leq n_{t-1}\left(1 - \frac{1}{k}\right) \leq n_{t-2}\left(1 - \frac{1}{k}\right)^2 \leq \cdots \leq n_0\left(1 - \frac{1}{k}\right)^t = n\left(1 - \frac{1}{k}\right)^t$$

Using the fact: $1 - x \leq e^{-x}$ (equality when $x = 0$)

$$n_t \leq n\left(1 - \frac{1}{k}\right)^t \leq ne^{-t/k}$$

Proof

Proof: Let n_t be the number of elements not covered by the greedy algorithm after t iterations. These remaining n_t elements are covered by the optimal k subsets. So some subsets has $\geq \frac{n_t}{k}$ of these uncovered elements, and the greedy algorithm will pick a set of size at least $\frac{n_t}{k}$.
So, $n_{t+1} \leq n_t - \frac{n_t}{k} = n_t \left(1 - \frac{1}{k}\right)$

Repeatedly applying this:

$$n_t \leq n_{t-1}\left(1 - \frac{1}{k}\right) \leq n_{t-2}\left(1 - \frac{1}{k}\right)^2 \leq \cdots \leq n_0\left(1 - \frac{1}{k}\right)^t = n\left(1 - \frac{1}{k}\right)^t$$

Using the fact: $1 - x \leq e^{-x}$ (equality when $x = 0$)

$$n_t \leq n\left(1 - \frac{1}{k}\right)^t \leq ne^{-t/k}$$

Greedy algorithm terminates when $n_t < 1$. Let's find out what t makes $n_t < 1$

Proof

Since $n_t < ne^{-t/k}$, it suffices to make $ne^{-t/k} \leq 1$

Proof

Since $n_t < ne^{-t/k}$, it suffices to make $ne^{-t/k} \leq 1$

Solving $ne^{-t/k} \leq 1$

$\iff e^{-t/k} \leq \frac{1}{n} \iff -\frac{t}{k} \leq \ln(\frac{1}{n}) \iff t \geq -k\ln(\frac{1}{n}) = k\ln(n)$

Proof

Since $n_t < ne^{-t/k}$, it suffices to make $ne^{-t/k} \leq 1$

Solving $ne^{-t/k} \leq 1$
$\iff e^{-t/k} \leq \frac{1}{n} \iff -\frac{t}{k} \leq \ln(\frac{1}{n}) \iff t \geq -k\ln(\frac{1}{n}) = k\ln(n)$

At $t = k\ln(n)$, $n_t < 1$. Everything is covered $\square$

Proof

Since $n_t < ne^{-t/k}$, it suffices to make $ne^{-t/k} \leq 1$

Solving $ne^{-t/k} \leq 1$

$\iff e^{-t/k} \leq \frac{1}{n} \iff -\frac{t}{k} \leq \ln(\frac{1}{n}) \iff t \geq -k\ln(\frac{1}{n}) = k\ln(n)$

At $t = k\ln(n)$, $n_t < 1$. Everything is covered $\square$

Proof of the fact $1 - x \leq e^{-x}$ (equality when $x = 0$):

Proof

Since $n_t < ne^{-t/k}$, it suffices to make $ne^{-t/k} \leq 1$

Solving $ne^{-t/k} \leq 1$

$\iff e^{-t/k} \leq \frac{1}{n} \iff -\frac{t}{k} \leq \ln(\frac{1}{n}) \iff t \geq -k\ln(\frac{1}{n}) = k\ln(n)$

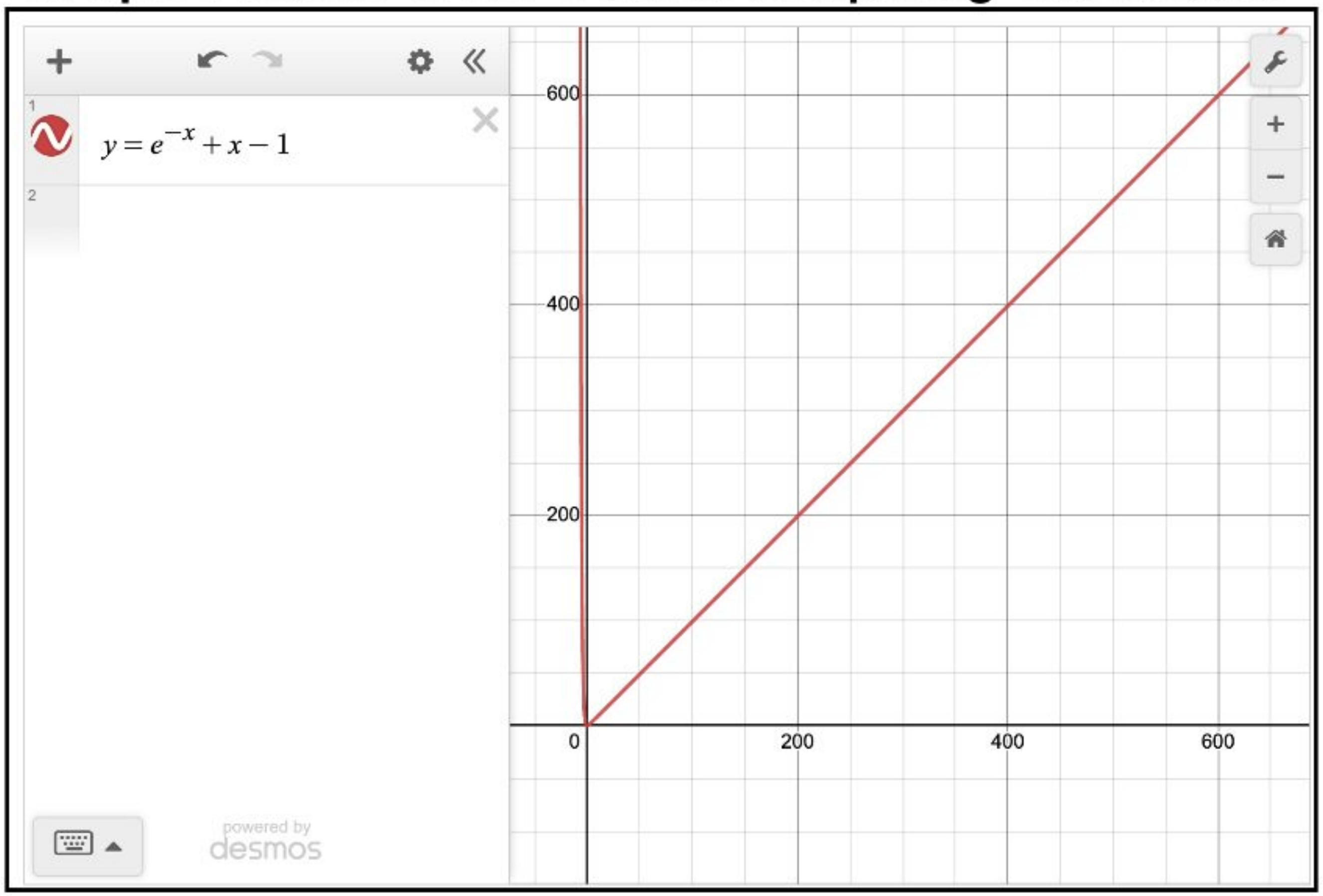
At $t = k\ln(n)$, $n_t < 1$. Everything is covered $\square$

Proof of the fact $1 - x \leq e^{-x}$ (equality when $x = 0$):

Consider $f(x) = e^{-x} - (1 - x) \geq 0$

$f(x) = e^{-x} - (1 - x)$

Graph Plotter :: An Online Graphing Calculator



Proof

Since $n_t < ne^{-t/k}$, it suffices to make $ne^{-t/k} \leq 1$

Solving $ne^{-t/k} \leq 1$

$\iff e^{-t/k} \leq \frac{1}{n} \iff -\frac{t}{k} \leq \ln(\frac{1}{n}) \iff t \geq -k\ln(\frac{1}{n}) = k\ln(n)$

At $t = k\ln(n)$, $n_t < 1$. Everything is covered $\square$

Proof of the fact $1 - x \leq e^{-x}$ (equality when $x = 0$):

Consider $f(x) = e^{-x} - (1 - x) \geq 0$

$f'(x) = -e^{-x} + 1$. Critical point at $x = 0$, achieving minimum $\square$

Implementation

Input:
U - set of elements,
F - family of sets: $\cup_{S \in F} S = U$

Output:
E - a family of sets; $E \subseteq F$: $\cup_{S \in E} S = U$

Pseudocode:
E = {}

```
while U is not empty do:
    choose S from F that maximizes the cover of elements in U
    add S to E
    subtract S's elements from U

return E
```

Implementation

```
from collections import defaultdict

# F is a list of sets

# First prepare a list of all sets where each element appears
D = defaultdict(list)
for S_idx, S in enumerate(F):
    for element in S:
        D[element].append(S_idx)

L = defaultdict(set)
# Place sets into an array that tells us which sets have
corresponding size
for S_idx, S in enumerate(F):
    L[len(S)].add(S_idx)
```

Implementation

F = the input list of sets

D = map from elements to the list of sets containing it

L = map of set lengths to indices of sets in F

```
E = [] # Keep track of selected sets

# Now loop over each set size
for set_size in range(max(len(S) for S in F), 0, -1):
    if set_size in L:
        P = L[set_size] # set of all sets with size = set_size
        while len(P) > 0:
            S_idx = P.pop()
            E.append(S_idx)
            for a in F[S_idx]: # all elements in the current set
                for y in D[a]: # all sets containing the element a
                    if y != S_idx: # not the current set
                        # removing a from y
                        S2 = F[y]
                        L[len(S2)].remove(y)
                        S2.remove(a)
                        L[len(S2)].add(y)

print E
```

Implementation

F = the input list of sets

D = map from elements to the list of sets containing it

L = map of set lengths to indices of sets in F

```
E = [] # Keep track of selected sets

# Now loop over each set size
for set_size in range(max(len(S) for S in F), 0, -1):
    if set_size in L:
        P = L[set_size] # set of all sets with size = set_size
        while len(P) > 0:
            S_idx = P.pop()
            E.append(S_idx)
            for a in F[S_idx]: # all elements in the current set
                for y in D[a]: # all sets containing the element a
                    if y != S_idx: # not the current set
                        # removing a from y
                        S2 = F[y]
                        L[len(S2)].remove(y)
                        S2.remove(a)
                        L[len(S2)].add(y)

print E
```

Cannot be executed more times than the number of elements

Hashsets access and edit elements using Θ (1)

Total:
O(N)
N = # elements