CS711008Z Algorithm Design and Analysis

Lecture 7. Basic algorithm design technique: Greedy

Dongbo Bu

Institute of Computing Technology
Chinese Academy of Sciences, Beijing, China

Outline

- Greedy is usually used to solve an optimisation problem with a solution form $X = [x_1, x_2, ..., x_n]$, $(x_i \in S)$.
- Connection with dynamic programming: ShortestPath problem and IntervalScheduling problem.
- Elements of greedy technique.
- Other examples: Huffman Code, Spanning Tree.
- Theoretical foundation of greedy technique: Matroid and submodular functions. 拟阵
- Introduction to important data structures: Binomial Heap, Fibonacci Heap, Union-Find.

- Greedy is usually used to solve an optimization problem whose solving process can be described as a multiple-stage decision process, e.g., solution has the form $X = [x_1, x_2, ..., x_n]$, $x_i \in S$.

  多步决策.

- For this type of problems, we can construct **a tree to enumerate all possible decisions**.

  可枚举所有决策.

- Greedy technique can be treated as finding **a set of paths** from root (null solution) to a leaf node (complete solution). At each intermediate node, greedy rule is applied to select one of its children nodes.

  贪心就是找路的过程 (高效减枝)

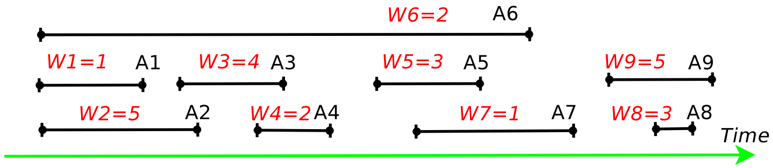
The first example: Two versions of IntervalScheduling problem

IntervalScheduling problem

- Practical problem:
  - a class room is requested by several courses;
  - the i-th course A_i starts from S_i and ends at F_i.
- Objective: to meet as many students as possible.

An instance

Example:



$$\texttt{Solutions:} \quad S_1 = \{A_1, A_3, A_5, A_8\} \quad | \quad S_2 = \{A_6, A_9\}$$

$$\texttt{Benefits:} \quad B(S_1) = 1 + 4 + 3 + 3 = 11 \quad | \quad B(S_2) = 2 + 5 = 7$$

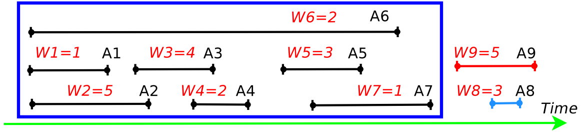
- Formulation:

INPUT:
n activities $A = \{A_1, A_2, ..., A_n\}$ that wish to use a resource. Each activity A_i uses the resource during interval $[S_i, F_i)$. The selection of activity A_i yields a benefit of W_i.
OUTPUT:
To select a collection of **compatible** activities to **maximize benefits**.

- Here, A_i and A_j are **compatible** if there is no overlap between the corresponding intervals $[S_i, F_i)$ and $[S_j, F_j)$, i.e. the resource cannot be used by more than one activities at a time.
- It is assumed that the activities have been sorted according to the finishing time, i.e. $F_i \leq F_j$ for any $i < j$.

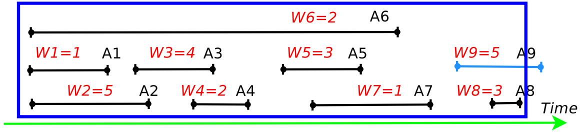
Key observation I

- It is not easy to solve a problem with n activities directly. Let's see whether it can be reduced into smaller sub-problems.
- Solution: a subset of activities. Imagine the solving process as a series of decisions; at each decision step, we choose an activity to use the resource.
- Suppose we have already worked out the optimal solution. Consider **the first decision** in the optimal solution, i.e. whether A_n is selected or not. There are 2 options:
  1. Select activity A_n: the selection leads to a **smaller subproblem**, namely selecting from the activities ending before S_n.



Key observation II

2. Abandon activity A_n: then it suffices to solve another **smaller subproblem**: to select activities from $A_1, A_2, ..., A_{n-1}$.



Key observation cont'd

- Summarizing the two cases, we can design the general form of subproblems as:
  selecting a collection of activities from $A_1, A_2, ..., A_i$ to maximize benefits.
- Denote the optimal solution value as $OPT(i)$.
- Optimal substructure property: ("cut-and-paste" argument)

$$OPT(i) = \max \begin{cases} OPT(pre(i)) + W_i \\ OPT(i-1) \end{cases}$$

Here, $pre(i)$ denotes the largest index of the activities ending before S_i.

Dynamic programming algorithm

$\textsc{Recursive\_DP}(i)$

Require: All A_i have been sorted in the increasing order of F_i.

```
if  i ≤ 0  then
    return  0;
end if
if  i == 1  then
    return  W_1;
end if
Determine the largest index of the activities ending before S_i, denoted as
pre(i).
```

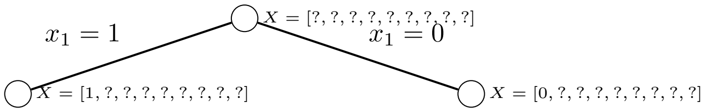
$$m = \max \begin{cases} \textsc{Recursive\_DP}(pre(i)) + W_i \\ \textsc{Recursive\_DP}(i-1) \end{cases}$$

```
return  m;
```

Note:

- The original problem can be solved by calling $\textsc{Recursive\_DP}(n)$.
- It needs $O(n \log n)$ to sort the activities and determine $pre(.)$, and the dynamic programming needs $O(n)$ time.
- Thus, time complexity: $O(n \log n)$

Multiple-stage decision process

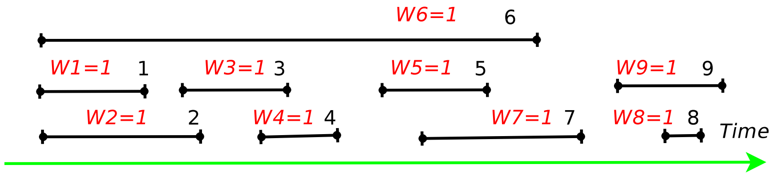


- We have to enumerate the two options $x_1 = 0$ and $x_1 = 1$, as we have no idea which one is optimal.

INTERVALSCHEDULING problem: version 2

Let's investigate a special case

每门课1,有1个同学. ⇒安排的课越多越好.



A special case of IntervalScheduling problem with **all weights** $w_i = 1$.

Formulation:

INPUT:
n activities $A = \{A_1, A_2, ..., A_n\}$ that wish to use a resource. Each activity A_i uses the resource during interval $[S_i, F_i)$.
OUTPUT:
To select as many **compatible activities** as possible.

Greedy selection property

Another key observation: Greedy selection I

- Since this is just a special case, the **optimal substructure property** still holds.
- Besides the optimal substructure property, the special weight setting leads to **"greedy selection" property**.

贪心选择

Theorem

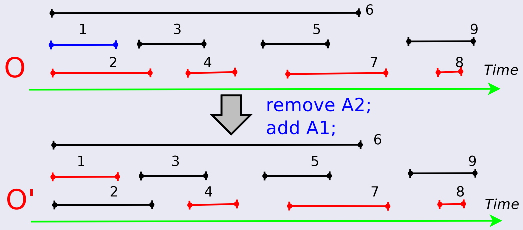
Suppose A_1 is the activity with the earliest ending time. A_1 **is used in an optimal solution.**

Proof.

(exchange argument)

- Suppose we have an optimal solution $O = \{A_{i1}, A_{i2}, ..., A_{iT}\}$ but $A_{i1} \neq A_m$.
- A_1 ends earlier than A_{i1}.
- A_1 is compatible with $A_{i2}, ..., A_{iT}$. (Why?)
- Construct a new subset $O' = O - \{A_{i1}\} \cup \{A_1\}$
- O' is also an optimal solution since $|O'| = |O|$.

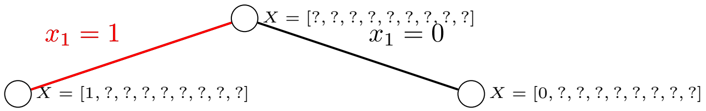
去除A2
换成A1



O' is compatiable since S4 > F2 > F1

两个都是最优解.

Multiple-stage decision process



- Greedy selection rule: $x_1 = 1$ is the optimal option. Thus it is unnecessary to enumerate the two options $x_1 = 0$ and $x_1 = 1$,.

Simplifying the DP algorithm into a greedy algorithm

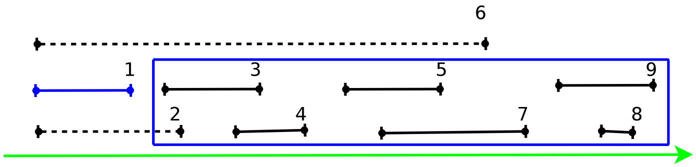
INTERVAL_SCHEDULING_GREEDY(n)

Require: All A_i have been sorted in the increasing order of F_i.

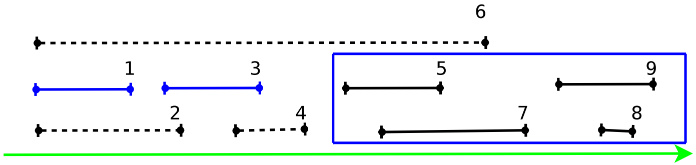
$previous\_finish\_time = -\infty$;
for $i = 1$ to n **do**
    if $S_i \geq previous\_finish\_time$ **then**
        Select activity A_i;
        $previous\_finish\_time = F_i$;
    end if
end for

Time complexity: $O(n \log n)$ (sorting activities in the increasing order of finish time).

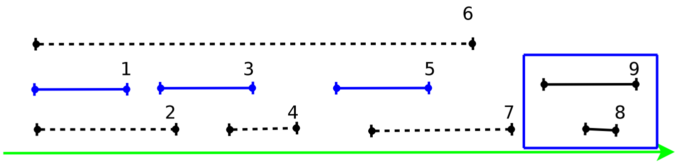
Step 1:



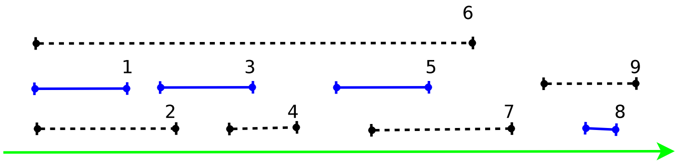
Step 2:



Step 3:



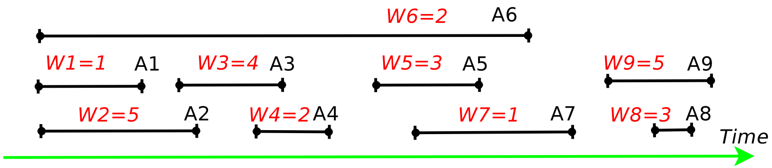
Step 4:



(see another demo)

Question: does the greedy algorithm work in general cases?

Why greedy strategy doesn't work for the general IntervalScheduling problem?



`Solutions:`	$Greedy: \{A_1, A_3, A_5, A_8\}$	|	$OPT: \{A_2, A_4, A_5, A_9\}$
`Benefits:`	$1+4+3+3=11$	|	$5+2+3+5=15$

- Reason: Greedy choice property doesn't hold.
- Note: although the problem is the same, a slight change of weights leads to significant affects on algorithm design.

Elements of greedy algorithm

- In general, greedy algorithms have five components:
  1. A candidate set, from which a solution is created
  2. A selection function, which chooses the best candidate to be added to the solution
  3. A feasibility function, that is used to determine if a candidate can be used to contribute to a solution
  4. An objective function, which assigns a value to a solution, or a partial solution, and
  5. A solution function, which will indicate when we have discovered a complete solution

DP versus Greedy

贪心和动规的关系。

Similarities:

都用来解优化问题。

1. Both dynamic programming and greedy techniques are typically applied to **optimization** problems.
2. **Optimal substructure**: Both dynamic programming and greedy techniques exploit the optimal substructure property.
3. **Beneath every greedy algorithm, there is almost always a more cumbersome dynamic programming solution — CRLS**

笨重。

Differences: 动规要枚举所有的可能解 backtrace

1. A dynamic programming method typically **enumerate all possible options at a decision step**, and the decision cannot be determined before subproblems were solved.
2. In contrast, greedy algorithm does not need to enumerate all possible options—it simply **make a locally optimal (greedy) decision** without considering results of subproblems.

Note:

- Here, "local" means that we have already acquired part of an optimal solution, and the partial knowledge of optimal solution is sufficient to help us make a wise decision.
- Sometimes a rigorous proof is unavailable, thus extensive experimental results are needed to show the efficiency of the greedy technique.

How to design greedy method?

1. 产生10000个instance
2. 使用DP产生最优解
3. 使用NN，输入为Sample和最优解，输出每一步选择.

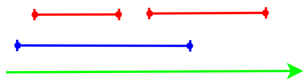
Two strategies:

1. Simplifying a dynamic programming method through greedy selection;
2. Trial-and-error: Imagining the solution-generating process as making a sequence of choices, and trying different greedy selection rules.

Trying other greedy rules

Incorrect trial 1: **earlist start** rule

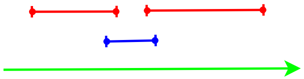
- Intuition: the earlier start time, the better.
- Incorrect. A negative example:



- Greedy solution: blue one. Solution value: 1.
- Optimal solution: red ones. Solution value: 2.

Incorrect trial 2: trying **minimal duration** rule

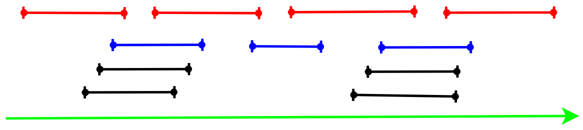
- Intuition: the shorter duration, the better.
- Incorrect. A negative example:



- Greedy solution: blue one. Solution value: 1.
- Optimal solution: red ones. Solution value: 2.

Incorrect trial 3: trying **minimal conflicts** rule

- Intuition: the less conflict activities, the better.
- Incorrect. A negative example:



- Greedy solution: blue ones. Solution value: 3.
- Optimal solution: red ones. Solution value: 4.

Revisiting SHORTESTPATH problem

INPUT:
A directed graph $G =< V, E >$. Each edge $e =< i, j >$ has a distance $d_{i,j}$. A single source node s, and a destination node t;
OUTPUT:
The shortest path from s to t (Or the shortest paths from s to each node $v \in V$, or the shortest paths from each node $v \in V$ to t).

Two versions of SHORTESTPATH problem:

1. No negative cycle: Bellman-Ford dynamic programming algorithm;
2. No negative edge: Dijkstra greedy algorithm.

Optimal sub-structure property in version 1

- Solution: a path from s to t with at most $(n-1)$ edges. Imagine the solving process as making a series of decisions; at each decision step, we decide the subsequent node.
- Suppose we have already obtained an optimal solution O. Consider the final decision (i.e. from which we reach node t) within O. There are several possibilities for the decision:
  - node v such that $< v, t > \in E$: then it suffices to solve a smaller subproblem, i.e. "starting from s to node v via at most $(n-2)$ edges".
- Thus we can design the general form of sub-problems as **"starting from s to a node v via at most k edges"**. Denote the optimal solution value as $OPT(v,k)$.
- Optimal substructure:

从S到v最多k步：

$$OPT(v,k) = \min \begin{cases} OPT(v, k-1) \\ \min_{<u,v>\in E}\{OPT(u,k-1) + d_{u,v}\} \end{cases}$$

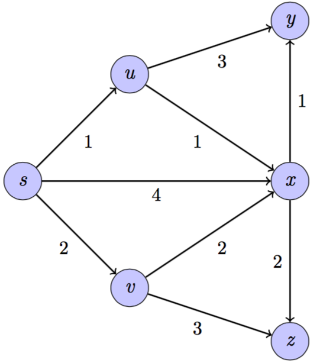
- Note: the first item $OPT(v,k-1)$ is introduced here to describe **"at most"**.
- Time complexity: $O(mn)$

Bellman-Ford algorithm 1956

BELLMAN_FORD(G, s, t)

for $i = 0$ to n **do**
    $OPT[s, i] = 0;$
end for
for any node $v \in V$ **do**
    $OPT[v, 0] = \infty;$
end for
for $k = 1$ to $n - 1$ **do**
    for all node v (in an arbitrary order) **do**
        $OPT[v, k] = \min \begin{cases} OPT[v, k-1], \\ \min_{<u,v>\in E}\{OPT[u, k-1] + d(u, v)\} \end{cases}$
    end for
end for
return $OPT[t, n - 1];$

An example



	k=0	1	2	3	4	5
S	0	0	0	0	0	0
U	–	1	1	1	1	1
V	–	2	2	2	2	2
X	–	4	2	2	2	2
Y	–	–	4	3	3	3
Z	–	–	5	4	4	4

Greedy-selection property in version 2

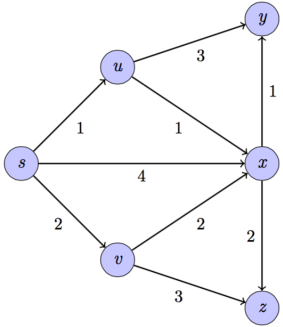
Greedy-selection property

- At the k-th step, let's consider a special node v^*, the nearest node from s via at most $k-1$ edges, i.e. $OPT(v^*, k-1) = min_v OPT(v, k-1)$.

- Consider the optimal substructure property for v^*, i.e.
$$OPT(v^*, k) = \min \begin{cases} OPT(v^*, k-1) \\ \min_{<u,v^*>\in E}\{OPT(u, k-1) + d_{u,v^*}\} \end{cases}$$

- The above equality can be further simplified as:
$OPT(v^*, k) = OPT(v^*, k-1)$
(Why? $OPT(u, k-1) \geq OPT(v^*, k-1)$ and $d_{u,v^*} \geq 0$.)

The meaning of $OPT(v^*, k) = OPT(v^*, k-1)$



	k=0	1	2	3	4	5
S	0	0	0	0	0	0
U	–	1	1	1	1	1
V	–	2	2	2	2	2
X	–	4	2	2	2	2
Y	–	–	4	3	3	3
Z	–	–	5	4	4	4

1. Intuitively v^* (in red circles) can be treated as **has already been explored using at most $(k-1)$ edges**, and the distance will not change afterwards.
2. Thus, the calculations of $OPT(v^*, k)$ (in green rectangles) are in fact redundant.
3. In other words, it suffices to calculate $OPT(v, k) = \min_{<u,v> \in E}\{OPT(u, k-1) + d_{u,v}\}$ for the **unexplored nodes** $v \neq v^*$.

But how to calculate $OPT(v, k)$ for the **unexplored nodes** $v \notin S$?
Let's see a greedy selection rule.

只多走一步到达该点

Theorem

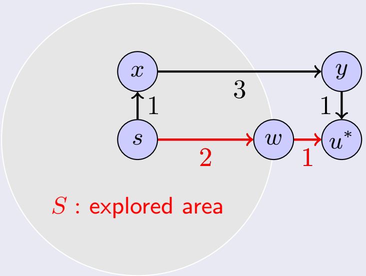
Let S denote the **explored** *nodes.* **Consider the nearest unexplored node u^***, *i.e., u^* is the node u ($u \notin S$) that minimizes $d'(u) = \min_{w \in S}\{d(w) + d(w,u)\}$. Then the path $P = s \to \dots \to w \to u^*$ is one of the shortest paths from s to u^* with distance $d'(u^*)$.*

Proof.

- Suppose there is another path P' from s to u^* shorter than P.
- Without loss of generality, we denote $P' = s \to \dots \to x \to y \to \dots \to u^*$. Here, y denotes the first node in P' leaving out of S.
- But $|P'| \geq d(s,x) + d(x,y) \geq d'(u^*)$. A contradiction.

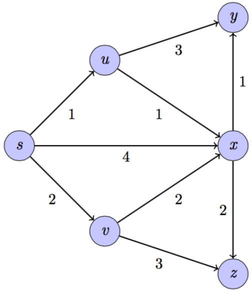
y: $s \to x \to y$: 4

u^*: $s \to w \to u^*$: 3



Key observations

1. Let v^* denote the nearest node from s using at most $k-1$ edges. The shortest distance $d(v^*)$ will not change afterwards.



	k=0	1	2	3	4	5
S	0	0	0	0	0	0
U	–	1	1	1	1	1
V	–	2	2	2	2	2
X	–	4	2	2	2	2
Y	–	–	4	3	3	3
Z	–	–	5	4	4	4

→ 从U出发，只走一步就到达的点，路径最小的.

S→U

V : 2

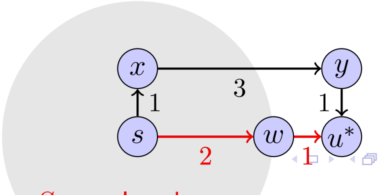
X : 2

Y : 4

Z : ∞

explored

2. Let's u^* denote the nearest unexplored node. The shortest distance can be determined.



Dijkstra's algorithm [1959]

DIJKSTRA(G, s) 加权的BFS

1: $S = \{s\}$; //S denotes the set of explored nodes,
2: $d(s) = 0$; //$d(u)$ stores an upper bound of the shortest-path weight from s to u;
3: **for all** node $v \neq s$ **do**
4: $\quad d(v) = +\infty$;
5: **end for**
6: **while** $S \neq V$ **do**
7: $\quad$ **for all** node $v \notin S$ **do**
8: $\quad\quad d(v) = \min_{u \in S}\{d(u) + d(u, v)\}$;
9: $\quad$ **end for**
10: $\quad$ Select the node v^* ($v^* \notin S$) that minimizes $d(v)$;
11: $\quad S = S \cup \{v^*\}$;
12: **end while**

- Line $(8 - 10)$ is called **"relaxing"**. That is, we test whether the shortest-path to v found so far can be improved by going through u, and if so, update $d(v)$.
- In the case that $d_{u,v} = 1$ for any u, v pair, Dijkstra's algorithm reduces to BFS. Thus, Dijkstra's algorithm can be

BFS : $O(n+m)$

Implementing Dijkstra algorithm using priority queue

DIJKSTRA(G, s)

$key(s) = 0$; $//key(u)$ stores an upper bound of the shortest-path weight from s to u;
PQ. INSERT (s);
$S = \{s\}$; // Let S be the set of explored nodes;
for all node $v \neq s$ **do**
    $key(v) = +\infty$
    PQ. INSERT (v) // n times 优先队列
end for
while $S \neq V$ **do**
    $v = PQ$. EXTRACTMIN(); // n times
    $S = S \cup \{v\}$;
    for each $w \notin S$ and $< v, w > \in E$ **do**
        if $key(v) + d(v, w) < key(w)$ **then**
            PQ.DECREASEKEY$(w, key(v) + d(v, w))$; // m times
        end if
    end for
end while

Here PQ denotes a min-priority queue. (see a demo)

Contributions by Edsger W. Dijkstra

- The semaphore construct for coordinating multiple processors and programs.
- The concept of self-stabilization 090009 an alternative way to ensure the reliability of the system
- "A Case against the GO TO Statement", regarded as a major step towards the widespread deprecation of the GOTO statement and its effective replacement by structured control constructs, such as the while loop.
- ...

ShortestPath: Bellman-Ford algorithm vs. Dijkstra algorithm

A slight change of edge weights leads to a significant change of algorithm design.

1. No negative cycle: an optimal path from s to v has at most $n-1$ edges; thus the optimal solution is $OPT(v, n-1)$. To calculate $OPT(v, n-1)$, we appeal to the following recursion:

$$OPT[v,k] = \min \begin{cases} OPT[v, k-1], \\ \min_{<u,v> \in E}\{OPT[u, k-1] + d(u,v)\} \end{cases}$$

2. No negative edge: This stronger constraint on edge weights implies greedy choice property. In particular, it is not necessary to calculate $OPT(v, i)$ for any explored node $v \in S$, and for the nearest unexplored node, its shortest distance from s is determined.

Time complexity analysis

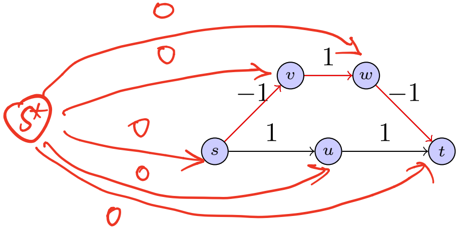
Time complexity of DIJKSTRA algorithm

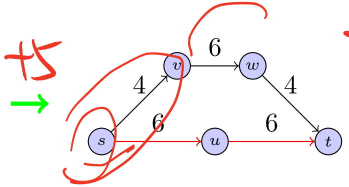
Operation	Linked list	Binary heap	Binomial heap	Fibonacci heap
MAKEHEAP	1	1	1	1
INSERT	1	$\log n$	$\log n$	1
EXTRACTMIN	n	$\log n$	$\log n$	$\log n$
DECREASEKEY	1	$\log n$	$\log n$	1
DELETE	n	$\log n$	$\log n$	$\log n$
UNION	1	n	$\log n$	1
FINDMIN	n	1	$\log n$	1
DIJKSTRA	$O(n^2)$	$O(m \log n)$	$O(m \log n)$	$O(m + n \log n)$

DIJKSTRA algorithm: n INSERT, n EXTRACTMIN, and m DECREASEKEY.

Extension: can we reweigh the edges to make all weight positive?

Trial 1: increasing all edge weights by the same amount

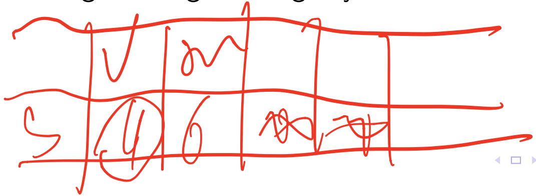




每条边加相同的数

S 0
V -1
W 0
u 1
t -1

- Increasing all the weight by 5 changes the shortest path from s to t.
- Reason: different paths might change by different amount although all edges change by the same mount.



	v	u	w	t
S	4	6	∞	∞

	v	u	w	t
S	4	6	10	∞

	v	u	w	t
S	4	6	10	12

w→t : 4

S→W→t : 14

14>12

不松弛

	v	u	w	t
S	4	6	10	12

END

	v	u	w	t
S	S	S	-	-

	v	u	w	t
S	S	S	v	-

	v	u	w	t
S	S	S	v	u

	v	u	w	t
S	S	S	u	u

Trial 2: increasing an edge weight according to its two ends

- Suppose each node v is associated with a number $c(v)$. We reweigh an edge (u, v) as follows.
  $d'(u, v) = d(u, v) + c(u) - c(v)$ 起点 - 终点
- Note that for any path $u \rightsquigarrow v$, we have
  $d'(u \rightsquigarrow v) = d(u \rightsquigarrow v) + c(u) - c(v)$
- Advantage: the shortest path from u to v with the new weighting function is exact the same to that with the original weighting function.
- But how to define $c(v)$ to make all edge weight positive?

Reweighting schema

- Adding a new node S, and connect S to each node v with an edge weight $d(S, v) = 0$, $d(v, S) = \infty$
- Set $c(v)$ as $dist(S, v)$, the shortest distance from S to v.
- We can prove that for any node pair u and v, $d'(u, v) = d(u, v) + dist(u) - dist(v) \geq 0$.

Johnson algorithm for all pairs shortest path [1977]

JOHNSON(G, d)

Create a new node s^*;
for all node $v \neq s^*$ **do**
    $d(s^*, v) = 0$
end for
Run Bellman-Ford to calculate the shortest distance from s^* to all nodes;
Reweighting: $d'(u, v) = d(u, v) + dist(s^*, u) - dist(s^*, v)$
for all node $u \neq s^*$ **do**
    Run Dijkstra's algorithm with the new weight d' to calculate the shortest paths from u;
    for all node $v \neq s^*$ **do**
        $dist(u, v) = dist(u, v) - dist(s^*, u) + dist(s^*, v)$;
    end for
end for

Time complexity: $O(mn + n^2 \log n)$.

Extension: data structures designed to speed up the Dijkstra's algorithm

Figure 1: Robert W. Floyd, Jean Vuillenmin, Robert Tarjan

(See extra slides for binary heap, binomial heap and Fibonacci heap)

Huffman Code

- Practical problem: how to compact a file when you have the knowledge of frequency of letters?
- Example:

SYMBOL	A	B	C	D	E	
Frequency	24	12	10	8	8	
Fixed Length Code	000	001	010	011	100	$E(L) = 186$
Variable Length Code	00	01	10	110	111	$E(L) = 140$

Formulation

INPUT:
a set of symbols $S = \{s_1, s_2, ..., s_n\}$ with its appearance frequency $P = \{p_1, p_2, ..., p_n\}$;
OUTPUT:
assign each symbol with a binary code C_i to minimize the length expectation $\sum_i p_i |C_i|$.

Requirement: prefix code I

- To avoid the potential ambiguity in decoding, we require the coding to be **prefix code**.

Definition (Prefix coding)

A prefix coding for a symbol set S is a coding such that for any symbols $x, y \in S$, the code $C(x)$ is not prefix of the code $C(y)$.

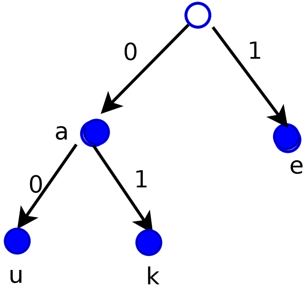
- Intuition: A prefix code can be represented as a binary tree, where a leaf represents a symbol, and the path to a leaf represents the code.
- Our objective: to design an optimal tree T to minimize expected length $E(T)$ (the size of the compressed file).

Requirement: prefix code II

Ex1:

C(a) = 0
C(u) = 00
C(k) = 01
C(e) = 1

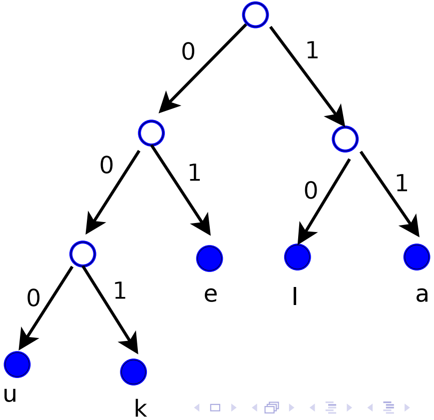
What is 0101?
aeae
k k
Ambiguity!



Ex2:

C(a) = 11
C(e) = 01
C(k) = 001
C(l) = 10
C(u) = 000

What is 1001000001?
l e u



Full binary tree

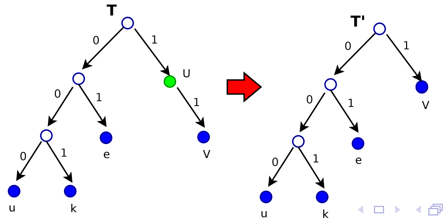
Theorem

An optimal binary tree should be a full tree.

Proof.

- Suppose T is an optimal tree but is not full;
- There is a node u with only one child v;
- Construct a new tree T', where u is replaced with v;
- $E(T') \leq E(T)$ since any child of v has a shorter code.

□



But how to construct the optimal tree?
Let's describe the solving process as a multiple-stage decision process.

- There a total of 2^n options, which makes the dynamic programming infeasible.

Shannon-Fano coding [1949]

Top-down method :

1: Sorting S in the decreasing order of frequency.
2: Splitting S into two sets S_1 and S_2 with almost equal frequencies.
3: Recursively building trees for S_1 and S_2.

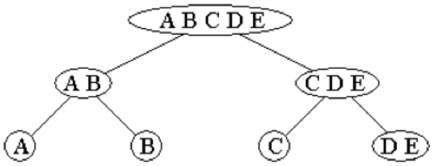
Figure 2: Claude Shannon and Robert Fano

An example: Step 1

Symbol	Freq-quency	1. Step Sum	1. Step Kode	2. Step Sum	2. Step Kode	3. Step Sum	3. Step Kode
A	24	24	0	24	00		
B	12	36	0	12	01		
C	10	26	1	10	10		
D	8	16	1	16		16	110
E	8	8	1	8		8	111

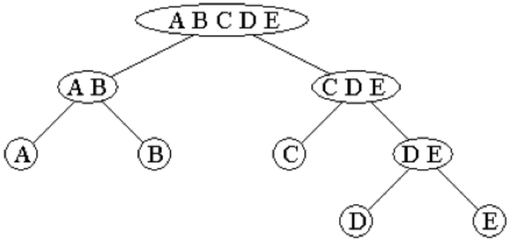
An example: Step 2

Symbol	Freq- quency	1. Step Sum	1. Step Kode	2. Step Sum	2. Step Kode	3. Step Sum	3. Step Kode
A	24	24	0	24	00		
B	12	36	0	12	01		
C	10	26	1	10	10		
D	8	16	1	16		16	110
E	8	8	1	8		8	111



An example: Step 3

Symbol	Freq-quency	1. Step Sum	1. Step Kode	2. Step Sum	2. Step Kode	3. Step Sum	3. Step Kode
A	24	24	0	24	00		
B	12	36	0	12	01		
C	10	26	1	10	10		
D	8	16	1	16		16	110
E	8	8	1	8		8	111



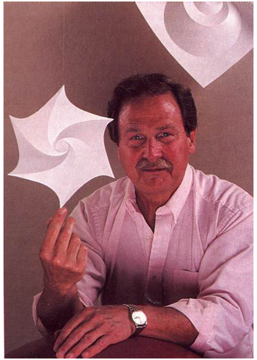
A bottom-up multiple-decision process

- There a total of $\binom{n}{2}$ options.

Huffman code: bottom-up manner [1952]

Bottom-up method:

1: **repeat**
2: Merging the two lowest-frequency letters y and z into a new meta-letter yz,
3: Setting $P_{yz} = P_y + P_z$.
4: **until** only one label is left



Huffman code: bottom-up manner [1952]

Key Observations:

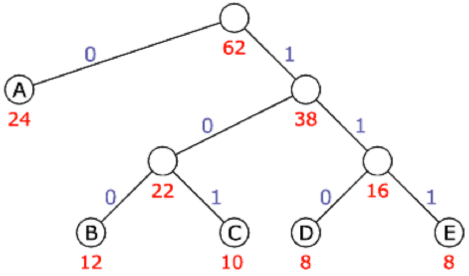
1. In an optimal tree, $depth(u) \geq depth(v)$ iff $P_u \leq P_v$. (Exchange argument)
2. There is an optimal tree, where the lowest-frequency letters Y and Z are siblings. (Why?)
    - Consider a deepest node v.
    - v's parent, denoted as u, should has another child, say w.
    - w should also be a deepest node.
    - v and w have the lowest frequency.

Huffman code algorithm 1952

HUFFMAN(S, P)

if $|S| == 2$ **then**
  return a tree with a root and two leaves;
end if
Extract the two lowest-frequency letters Y and Z from S;
Set $P_{YZ} = P_Y + P_Z$;
$S = S - \{Y, Z\} \cup \{YZ\}$;
$T' =$HUFFMAN(S, P);
$T =$ add two children Y and Z to node YZ in T';
return T;

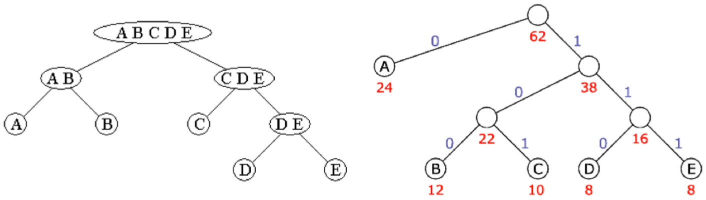
Example



Symbol	Frequency	Code	Code Length	total Length
A	24	0	1	24
B	12	100	3	36
C	10	101	3	30
D	8	110	3	24
E	8	111	3	24

```
---------------------------------------
      ges. 186 bit           tot. 138 bit
      (3 bit code)
```

Shannon-Fano vs. Huffman



		Shannon-Fano			Huffman		
Sym.	Freq.	code	len.	tot.	code	len.	tot.
A	24	00	2	48	0	1	24
B	12	01	2	24	100	3	36
C	10	10	2	20	101	3	30
D	8	110	3	24	110	3	24
E	8	111	3	24	111	3	24
total	186			140			138

(linear 3 bit code)

Huffman algorithm: correctness

Lemma

$E(T') = E(T) - P_{YZ}$

Proof.

$$
\begin{aligned}
E(T) &= \sum_{x \in S} P_x D(x,T) \\
&= P_Y D(Y,T) + P_Z D(Z,T) + \sum_{x \neq Y, x \neq Z} P_x D(x,T) \\
&= P_Y(1 + D(YZ,T')) + P_Z(1 + D(YZ,T')) + \sum_{x \neq Y, x \neq Z} P_x D(x,T) \\
&= P_{YZ} + P_Y D(YZ,T') + P_Z D(YZ,T') + \sum_{x \neq Y, x \neq Z} P_x D(x,T') \\
&= P_{YZ} + E(T')
\end{aligned}
$$

□

Note: $D(x,T)$ denotes the depth of leaf x in tree T.

Huffman algorithm: correctness cont'd

Theorem

Huffman algorithm output an optimal code.

Proof.

(Induction)

- Suppose there is another tree t with smaller expected length;
- In the tree t, let's merge the lowest frequency letters Y and Z into a meta-letter YZ; converting t into a new tree t' with of size $n-1$;
- t' is better than T'. Contradiction.

$\square$

Time complexity:

- $T(n) = T(n-1) + O(n) = O(n^2)$.
- $T(n) = T(n-1) + O(\log n) = O(n \log n)$ if use priority queue.

Note: Huffman code is a bit different example of greedy technique—the problem is shrinked at each step; in addition, the problem is changed a little (the frequency of a new meta letter is the sum frequency of its members).

Application

- In practical operation Shannon-Fano coding is not of larger importance. This is especially caused by the lower code efficiency in comparison to Huffman coding.
- Huffman codes are part of several data formats as ZIP, GZIP and JPEG. Normally the coding is preceded by procedures adapted to the particular contents. For example the wide-spread DEFLATE algorithm as used in GZIP or ZIP previously processes the dictionary based LZ77 compression.

See http://www.binaryessence.com/dct/en000003.htm for details.

Theoretical foundation of greedy strategy: Matroid and submodular functions

Theoretical foundation of greedy strategy

- Consider the following optimization problem: given a finite set of objects N, the objective is to find a subset $S \in \mathcal{F}$ such that a **set function** $f(S)$ is maximized, i.e.,

如果f是线性的，且F是有约束的。那么贪心能找到最优解。

$$\begin{aligned} \max \quad & f(S) \\ s.t. \quad & S \in \mathcal{F} \end{aligned}$$

  Here, $\mathcal{F} \subset 2^N$ represents certain constraints over S.

- In general cases, the problem is clearly intractable — you would better check all possible subsets in $\mathcal{F}$ to avoid missing the optimal solution. However, in certain special cases, greedy strategy applies and generates optimal solution or good approximation solutions.

When greedy strategy is perfect or good enough?

- So what conditions on either $\mathcal{F}$ or $f(S)$ or both does greedy strategy needs?
  - Matroid: Greedy strategy generates optimal solution when **$f(S)$ is a linear function**, and $\mathcal{F}$ can be characterized as **independent subsets**. F的子集线性无关
  - Submodular functions: Greedy strategy might generate provably good approximation when **$f(S)$ is a submodular function**.

When greedy strategy is perfect: Maximizing/minimizing a linear function under matroid constraint

Revisiting MAXIMAL LINEARLY INDEPENDENT SET problem

- Question: Given a set of vectors, to determine the maximal linearly independent set.
- Example:

$$\begin{array}{rl}
V_1 = & [\begin{array}{rrrrr} 1 & 2 & 3 & 4 & 5 \end{array}] \\
V_2 = & [\begin{array}{rrrrr} 1 & 4 & 9 & 16 & 25 \end{array}] \\
V_3 = & [\begin{array}{rrrrr} 1 & 8 & 27 & 64 & 125 \end{array}] \\
V_4 = & [\begin{array}{rrrrr} 1 & 16 & 81 & 256 & 625 \end{array}] \\
V_5 = & [\begin{array}{rrrrr} 2 & 6 & 12 & 20 & 30 \end{array}]
\end{array}$$

- Independent vector set: $\{V_1, V_2, V_3, V_4\}$

Calculating maximal number of independent vectors

INDEPENDENTSET(M)

$I = \{\}$;
for all row vector v **do**
    if $I \cup \{v\}$ is still independent **then**
        $I = I \cup \{v\}$;
    end if
end for
return I;

Correctness: Properties of linear independence vector set

Let's consider the **linear independence** for vectors.

1. **Hereditary property:** if B is an **independent vector set** and $A \subset B$, then A is also an **independent vector set**.
2. **Augmentation property:** if both A and B are **independent vector sets**, and $|A| < |B|$, then there is a vector $v \in B - A$ such that $A \cup \{v\}$ is still an independent vector set.

Example:

$$
\begin{array}{l}
V_1 = [\begin{array}{rrrrr} 1 & 2 & 3 & 4 & 5 \end{array}] \\
V_2 = [\begin{array}{rrrrr} 1 & 4 & 9 & 16 & 25 \end{array}] \\
V_3 = [\begin{array}{rrrrr} 1 & 8 & 27 & 64 & 125 \end{array}] \\
V_4 = [\begin{array}{rrrrr} 1 & 16 & 81 & 256 & 625 \end{array}] \\
V_5 = [\begin{array}{rrrrr} 2 & 6 & 12 & 20 & 30 \end{array}]
\end{array}
$$

- Independent vector sets: $A = \{V_1, V_3, V_5\}$, $B = \{V_1, V_2, V_3, V_4\}$, and $|A| < |B|$.
- Augmentation of A: $A \cup \{V_4\}$ is also independent.

A weighted version

- Question: Given a matrix, **where each row vector is associated with a weight**, to determine a set of linearly independent vectors to maximize the sum of weight.
- Example:

$$\begin{array}{ll}
V_1 = [\ 1 \quad 2 \quad 3 \quad 4 \quad 5\ ] & W_1 = 9 \\
V_2 = [\ 1 \quad 4 \quad 9 \quad 16 \quad 25\ ] & W_2 = 7 \\
V_3 = [\ 1 \quad 8 \quad 27 \quad 64 \quad 125\ ] & W_3 = 5 \\
V_4 = [\ 1 \quad 16 \quad 81 \quad 256 \quad 625\ ] & W_4 = 3 \\
V_5 = [\ 2 \quad 6 \quad 12 \quad 20 \quad 30\ ] & W_5 = 1
\end{array}$$

A general greedy algorithm (by Jack Edmonds [1970])

Matroid_Greedy(M, W)

$I = \{\}$;
Sort row vectors in the decreasing order of their weights;
for all row vector v **do**
    if $I \cup \{v\}$ is still independent **then**
        $I = I \cup \{v\}$;
    end if
end for
return I;

Time complexity: $O(n \log n + nC(n))$, where $C(n)$ is the time needed to check independence.

Matroid greedy algorithm: correctness

Theorem

[Greedy-choice property] Let v be the vector with the largest weight and $\{v\}$ is independent, then there is an optimal vector set A of M and A contains v.

Proof.

- Assume there is an optimal subset B but $v \notin B$.
- Then we can construct A from B as follows:
  1. Initially: $A = \{v\}$;
  2. Until $|A| = |B|$, repeatedly find a new element of B that can be added to A while preserving the independence of A (by augmentation property);
- Finally we have $A = B - \{v'\} \cup \{v\}$.
- We have $W(A) \geq W(B)$ since $W(v) \geq W(v')$ for any $v' \in B$. A contradiction.

□

Theorem

[Optimal substructure property] Let v be the vector with the largest weight and $\{v\}$ is itself independent. The remaining problem reduces to finding an optimal subset in M', where $M' = \{v' \in S, \text{and } v, v' \text{are independent}\}$

Proof.

- Suppose A' is an optimal independent set of M'.
- Define $A = A' \cup \{v\}$.
- Then A is also an independent set of M.
- And A has the maximum weight $W(A) = W(A') + W(v)$.

$\square$

An extension of **linear independence for vectors**: matroid

Matroid [Haussler Whitney, 1935]

- Matroid was proposed to capture the concept of **linear independence** in matrix theory, and generalize the concept in other field, say **graph theory**.
- In fact, in the paper *On the abstract properties of linear independence*, Haussler Whitney said:
  This paper has a close connection with a paper by the author on linear graphs; we say a subgraph of a graph is **independent** *if it contains no circuit.*

Origin 1 of matroid: linear independence for vectors

Let's consider the **linear independence** for vectors.

1. **Hereditary property:** if B is an **independent vector set** and $A \subset B$, then A is also an **independent vector set**
2. **Augmentation property:** if both A and B are **independent vector sets**, and $|A| < |B|$, then there is a vector $v \in B - A$ such that $A \cup \{v\}$ is still an **independent vector set**

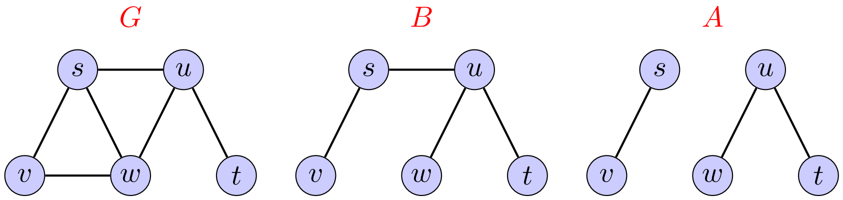
Example:

$$
\begin{array}{l}
V_1 = [\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array}] \\
V_2 = [\begin{array}{ccccc} 1 & 4 & 9 & 16 & 25 \end{array}] \\
V_3 = [\begin{array}{ccccc} 1 & 8 & 27 & 64 & 125 \end{array}] \\
V_4 = [\begin{array}{ccccc} 1 & 16 & 81 & 256 & 625 \end{array}] \\
V_5 = [\begin{array}{ccccc} 2 & 6 & 12 & 20 & 30 \end{array}]
\end{array}
$$

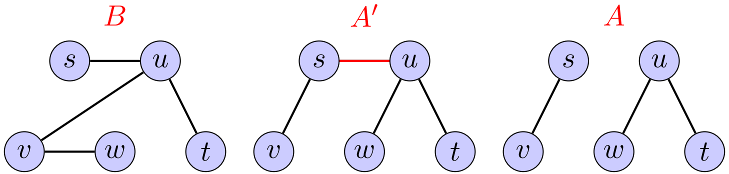
- Independent vector sets: $A = \{V_1, V_3, V_5\}$, $B = \{V_1, V_2, V_3, V_4\}$, and $|A| < |B|$.
- Augmentation of A: $A \cup \{V_4\}$ is also independent.

Origin 2 of matroid: acyclic subgraph [H. Whitney, 1932]

- Given a graph $G =< V, E >$, let's consider the **acyclic property**.
  - **Hereditary property:** if an edge set B is an **acyclic forest** and $A \subset B$, then A is also an **acyclic forest**



- **Augmentation property:** if both A and B are **acyclic forests**, and $|A| < |B|$, then there is an edge $e \in B - A$ such that $A \cup \{e\}$ is still an **acyclic forest**
  - Suppose forest B has more edges than forest A;
  - A has more trees than B. (Why? $\#Tree = |V| - |E|$)
  - B has a tree connecting two trees of A. Denote the connecting edge as (u, v).
  - Adding (u, v) to A will not form a cycle. (Why? it connects two different trees.)



Abstraction: the formal definition of matroid

- A matroid is a pair $M = (N, \mathcal{I})$, where N is a finite nonempty set (called **ground set**), and $\mathcal{I} \subseteq 2^N$ is a family of **independent subsets** of N satisfying the following conditions:
  1. **Hereditary property:** if $B \in \mathcal{I}$ and $A \subset B$, then $A \in \mathcal{I}$;
  2. **Augmentation property:** if $A \in \mathcal{I}$, $B \in \mathcal{I}$, and $|A| < |B|$, then there is some element $x \in B - A$ such that $A \cup \{x\} \in \mathcal{I}$.

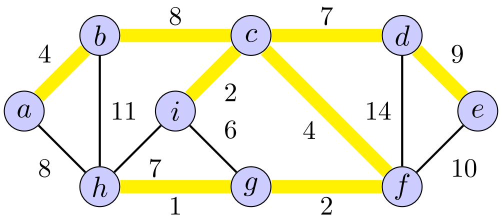
- **Bases and rank**: Maximal independent sets of a matroid M are called **bases**. It follows from the augmentation property that all bases have the same cardinality, which is denoted as **rank**.
- **Bijection basis exchange**: If B_1 and B_2 are two bases of a matroid $M = (N, \mathcal{I})$, then there exists a bijection $\phi : B_1 - B_2 \to B_2 - B_1$ such that:

$$\forall x \in B_1 - B_2,\ B_1 - x + \phi(x) \in \mathcal{I}$$

SPANNING TREE: an application of matroid

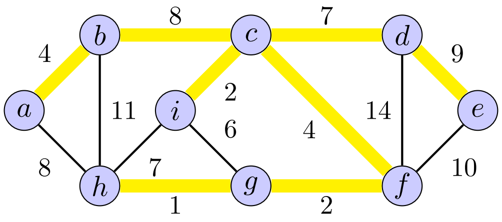
Practical problem:

- In the design of electronic circuitry, it is often necessary to make the pins of several components electrically equivalent by wiring them together.
- To interconnect a set of n pins, we can use $n-1$ wires, each connecting two pins;
- Among all interconnecting arrangements, the one that uses the least amount of wire is usually the most desirable.



Formulation:

Input: A graph G, and each edge $e =< u, v >$ is associated with a weight $W(u, v)$;
Output: a spanning tree with the minimum sum of weights.
Here, a spanning tree refers to a set of $n-1$ edges connecting all nodes.



拟阵

LINEARLY INDEPENDENT SET	⟺	ACYCLIC FOREST	等价关系
MAXIMAL LINEARLY INDEPENDENT SET	⟺	SPANNING TREE	支撑树
WEIGHTED MAXIMAL LINEARLY INDEPENDENT SET	⟺	MINIMUM SPANNING TREE	

- Objective: to find a spanning tree for graph G;
- Basic idea: analogue to Maximal Linearly Independent Set calculation;

GenericSpanningTree(G)

1: $F = \{\}$;
2: **while** F does not form a spanning tree **do**
3: find an edge (u, v) that is **safe** for F;
4: $F = F \cup \{(u, v)\}$;
5: **end while**

Here F denotes an acyclic forest, and F is still acyclic if added by a **safe** edge.

Examples of safe edge and unsafe edge

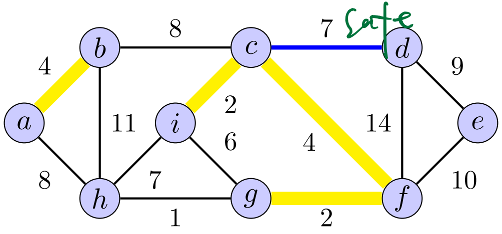


Figure 3: Safe edge

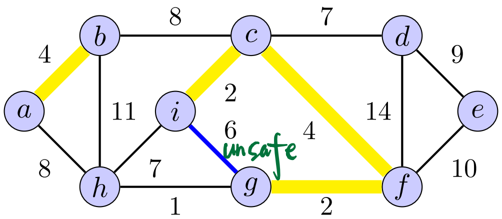


Figure 4: Unsafe edge

Minimum Spanning Tree algorithms

Kruskal's algorithm [1956]

- Basic idea: during the execution, F is always an **acyclic forest**, and the **safe edge** added to F is always a least-weight edge connecting two distinct components.

Figure 5: Joseph Kruskal

Kruskal's algorithm [1956]

MST-KRUSKAL(G, W)

$F = \{\}$;
for all vertex $v \in V$ **do**
    MAKESET(v);
end for
sort the edges of E into nondecreasing order by weight W;
for each edge $(u, v) \in E$ in the order **do**
    if FINDSET$(u) \neq$ FINDSET(v) **then**
        $F = F \cup \{(u, v)\}$;
        UNION (u, v);
    end if
end for

Here, UNION-FIND structure is used to detect whether a set of edges form a cycle.
(See extra slides for UNION-FIND data structure, and a demo of Kruskal algorithm)

Time complexity

- Running time:
  1. Sorting: $O(m \log m)$
  2. Initializing: n MakeSet operations;
  3. Detecting cycle: $2m$ FindSet operations;
  4. Adding edge: $n-1$ Union operations.

- Thus, the total time is $O(m \log n)$ when using Union-Find data structures.
- Provided that the edges are already sorted or can be sorted in $O(n)$ time using radix sort or counting sort, the total time is $O((m+n)\alpha(n))$, where $\alpha(n)$ is a very slowly growing function.

Prim's algorithm

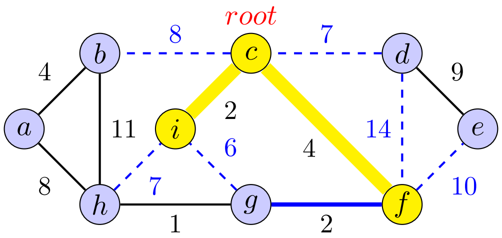
Prim's algorithm [1957]

- Basic idea: the final minimum spanning tree is grown step by step. Let's describe the solving process as a multiple-stage decision process. At each step, the least-weight edge connect the sub-tree to a node not in the tree is chosen.
- Note: One advantage of Prim's algorithm is that no special check to make sure that a cycle is not formed is required.

Figure 6: Robert C. Prim

Theorem

[Greedy selection property] Suppose T is a sub-tree of the final minimum spanning tree, and $e = (u, v)$ is the least-weight edge connect one node in T and another node not in T. Then e is in the final minimum spanning tree.



PRIM algorithm for MINIMUM SPANNING TREE [1957]

MST-PRIM$(G, W, root)$

for all node $v \in V$ and $v \neq root$ **do**
    $key[v] = \infty$;
    $\Pi[v] =$ NULL; $//\Pi(v)$ denotes the predecessor node of v
    PQ.INSERT(v); // n times
end for
$key[root] = 0$;
PQ.INSERT$(root)$;
while $PQ \neq$ NULL **do**
    $u = PQ$.EXTRACTMIN(); // n times
    for all v adjacent with u **do**
        if $W(u, v) < key(v)$ **then**
            $\Pi(v) = u$;
            PQ.DECREASEKEY$(W(u, v))$; // m times
        end if
    end for
end while

Here, PQ denotes a min-priority queue. The chain of predecessor nodes originating from v runs backwards along a shortest path from s to v.

(See a demo)

Time complexity of Prim algorithm

Operation	Linked list	Binary heap	Binomial heap	Fibonacci heap
MakeHeap	1	1	1	1
Insert	1	$\log n$	$\log n$	1
ExtractMin	n	$\log n$	$\log n$	$\log n$
DecreaseKey	1	$\log n$	$\log n$	1
Delete	n	$\log n$	$\log n$	$\log n$
Union	1	n	$\log n$	1
FindMin	n	1	$\log n$	1
Prim	$O(n^2)$	$O(m \log n)$	$O(m \log n)$	$O(m + n \log n)$

Prim algorithm: n Insert, n ExtractMin, and m DecreaseKey.

Note:

1. Matroid is useful when determining whether greedy technique yields optimal solutions.
2. It covers many cases of practical interests (Some exceptions: Huffman code, Interval Scheduling problems).

When greedy strategy is good enough: Maximizing a submodular function

Optimizing a set function

- Most combinatorial optimization problems, e.g., MinCut, MaxCut, VertexCover, SetCover, Minimum Spanning Tree, MaxCoverage, aim to maximize/minimize a set function.
- These problems have the following form:

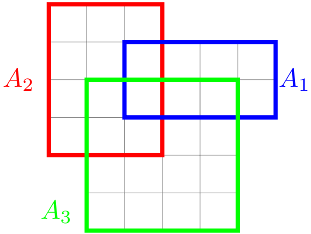
$$\begin{array}{ll} \max / \min & f(S) \\ s.t. & S \in \mathcal{F} \end{array}$$

Here, $S \subseteq N$ represents a subset of a ground set N, $\mathcal{F} \subseteq 2^N$ represents certain constraints over these subsets, and $f(S)$ denotes a set function.

Let's start from the MaxCoverage problem

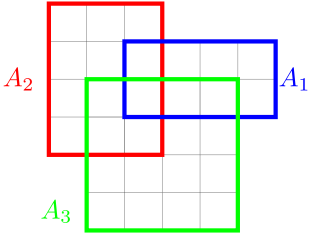
- Consider a set of n elements $N = \{1, 2, ..., n\}$, and m subsets $A_1, A_2, ..., A_m \subseteq N$. The goal of MaxCoverage problem is to select k subsets such that the cardinality of their union is maximized.

$$\begin{array}{ll} \max & f(S) \\ s.t. & |S| \leq k \end{array}$$



Set function

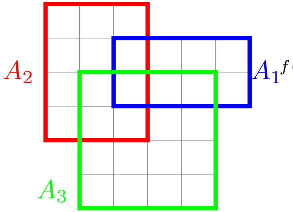
- The objective function in the MAX COVERAGE problem $f(S) = |\bigcup_{A_i \in S}|$ is a **set function** defined over subsets.

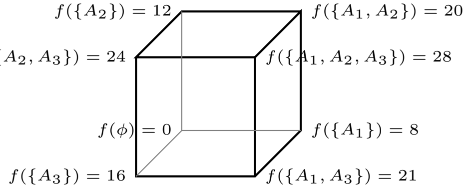


S	$f(S)$
ϕ	0
$\{A_3\}$	16
$\{A_2\}$	12
$\{A_2, A_3\}$	24
$\{A_1\}$	8
$\{A_1, A_3\}$	21
$\{A_1, A_2\}$	20
$\{A_1, A_2, A_3\}$	28

Set function: another viewpoint

- A set function $f : \{0,1\}^m \to \mathbb{R}$ defines value for nodes of a cube.

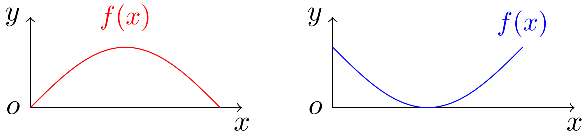




S	A_1	A_2	A_3	f
ϕ	0	0	0	0
$\{A_3\}$	0	0	1	16
$\{A_2\}$	0	1	0	12
$\{A_2, A_3\}$	0	1	1	24
$\{A_1\}$	1	0	0	8
$\{A_1, A_3\}$	1	0	1	21
$\{A_1, A_2\}$	1	1	0	20
$\{A_1, A_2, A_3\}$	1	1	1	28

Revisiting the continuous optimization

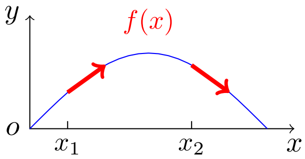
- But how to maximize a set function? Let's revisit the continuous maximization first.

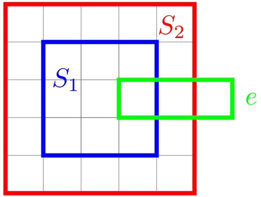


- A continuous function $f : \mathbb{R} \to \mathbb{R}$ can be efficiently maximized if it is **convex**, and can be efficiently maximized if it is **concave**.
- Question: are there **discrete analogy** to convexity or concavity for set functions?

Submodularity: discrete analogy to concavity

- Concavity: $f(x)$ is concave if the derivative $f'(x)$ is non-increasing in x, i.e., when Δx is sufficiently small, $f(x_1 + \Delta x) - f(x_1) \geq f(x_2 + \Delta x) - f(x_2)$ if $x_1 \leq x_2$.





- Submodularity: $f(S)$ is submodular if for any element e, the **marginal contribution** (discrete analogy to derivative) $f(S \cup \{e\}) - f(S)$ is non-increasing in S, i.e., $f(S_1 \cup \{e\}) - f(S_1) \geq f(S_2 \cup \{e\}) - f(S_2)$ if $S_1 \subseteq S_2$.

Submodular functions: decreasing marginal contribution

- Let's consider a set function $f(S)$ defined over subsets $S \subseteq N$, where N is a finite ground set.

Definition (Marginal contribution to a subset S)

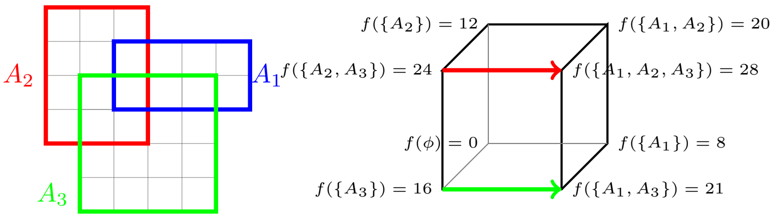
The *marginal contribution of a subset* $T \subseteq X$ *to* S is defined as $f_S(T) = f(S \cup T) - f(S)$.

Definition (Submodular function $f(S)$)

A set function $f : 2^N \rightarrow \mathbb{R}$ is *submodular* iff $\forall S_1 \subseteq S_2 \subseteq N$, $\forall e \in N - S_2,\ \ f_{S_2}(\{e\}) \leq f_{S_1}(\{e\})$. $f(S)$ is supermodular if $-f(S)$ is submodular, and modular if both sub- and supermodular.

- Intuition: marginal contribution is discrete analogy to derivative of a continuous function, while "decreasing marginal contribution" (or "diminishing returns") definition of $f(S)$ is discrete analogy to concave functions.

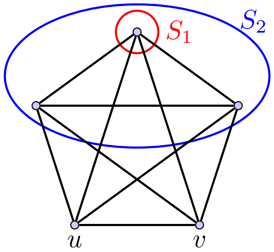
- The objective function in the MAX COVERAGE problem $f(S) = |\bigcup_{A_i \in S}|$ is submodular.



S	A_1	A_2	A_3	f
ϕ	0	0	0	0
$\{A_3\}$	0	0	1	16
$\{A_2\}$	0	1	0	12
$\{A_2, A_3\}$	0	1	1	24
$\{A_1\}$	1	0	0	8
$\{A_1, A_3\}$	1	0	1	21
$\{A_1, A_2\}$	1	1	0	20
$\{A_1, A_2, A_3\}$	1	1	1	28

Submodular function: example 2

- Given a graph $G =< V, E >$. Let $f(S)$ be the number of edges $e = (u, v)$ such that $u \in S$ and $v \in V - S$.



- $f(S)$ is submodular. For example, in the above figure, $f(S_1 \cup \{u\}) - f(S_1) = 8 - 4 = 4$, while $f(S_2 \cup \{u\}) - f(S_2) = 4 - 6 = -2$.

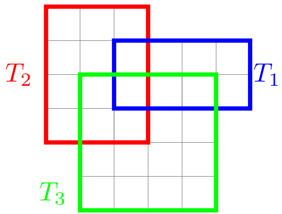
An equivalent definition: subadditive

Definition (Submodular function)

A function $f : 2^N \to \mathbb{R}$ is submodular iff $\forall A, B \subseteq N$, $f(A) + f(B) \geq f(A \cup B) + f(A \cap B)$.

- Taking the submodular function $f(A) = |\bigcup_{i \in A} T_i|$ as an example. Let $A = \{1, 2\}$, $B = \{1, 3\}$, we have

$$\begin{aligned} f(A \cup B) + f(A \cap B) &= |T_1 \cup T_2 \cup T_3| + |T_1| && (1)\\ &\leq |T_1 \cup T_2| + |T_1 \cup T_3| && (2)\\ &= f(A) + f(B) && (3)\end{aligned}$$



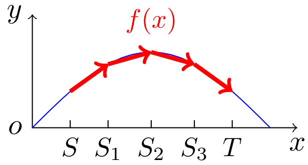
Equivalence of the two definitions

- Subadditivity essentially implies "decreasing marginal contribution", i.e., by setting $S_1 = A \cap B$, $S_2 = A$, and $T = B - A$, the inequality $f(A \cup B) - f(A) \leq f(B) - f(A \cap B)$ can be rewritten as $f(S_2 \cup T) - f(S_2) \leq f(S_1 \cup T) - f(S_1)$.

A useful property of submodular functions: upper bound

Lemma

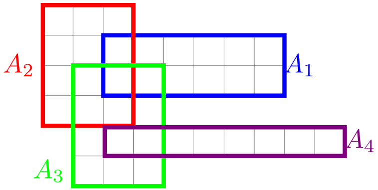
If f is submodular, then $\forall S \subseteq T \subseteq N$, $f(T) \leq f(S) + \sum_{e \in T \setminus S} f_S(e)$. Furthermore, if f is monotone submodular, S need not be a subset of T: $\forall S \subseteq N, T \subseteq N$, $f(T) \leq f(T \cup S) \leq f(S) + \sum_{e \in T \setminus S} f_S(e)$.



- By integrating marginal contributions, it is easy to prove the upper bound for submodular functions.
- Note that this is a discrete analogy to the property for a concave continuous function $f(x)$: $f(b) \leq f(a) + (b-a)f'(a)$.

MaxCoverage problem with cardinality constraint

- Now let's consider the MaxCoverage problem with **cardinality constraint** first, i.e., select k subsets such that the cardinality of their union is maximized, e.g., select 3 subsets in the following example.



- Here we adopt the **value query model**, i.e., an algorithm can query a black-box oracle for the value $f(S)$. An algorithm making polynomial queries is considered to have polynomial running time.
- We also assume that f is normalized, i.e., $f(\phi) = 0$.

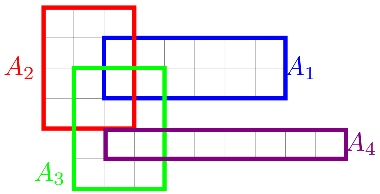
Greedy algorithm (cardinality constraint)

$\textsc{GreedyCardinalityConstraint}(k, N)$

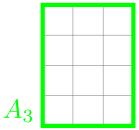
$S = \phi;$
while $|S| < k$ **do**
    $x^* = argmax_{x \in N} f_S(x);$
    $S = S \cup \{x^*\};$
    $N = N - \{x^*\};$
end while
return $S;$

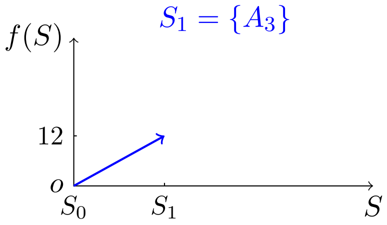
- Greedy rule: at each step, the item with the largest marginal contribution will be selected.

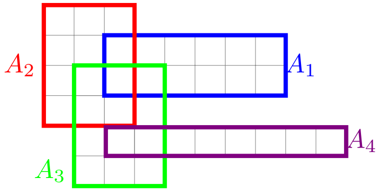
An example: Step 1



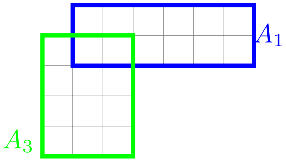
- Let $S_i = \{x_1^*, x_2^*, ..., x_i^*\}$ be the value of S after the i-th execution of the `while` loop. Initially A_3 was selected as $f_\phi(A_3) = 12$.

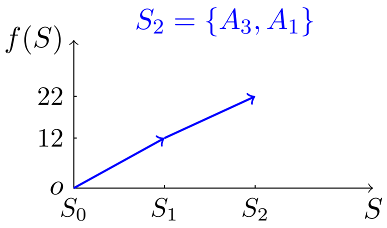




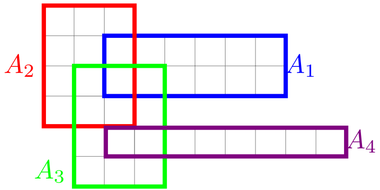


- A_1 was selected with $f_{S_1}(A_1) = 10$, which is larger than $f_{S_1}(A_2) = 8$, and $f_{S_1}(A_4) = 6$.

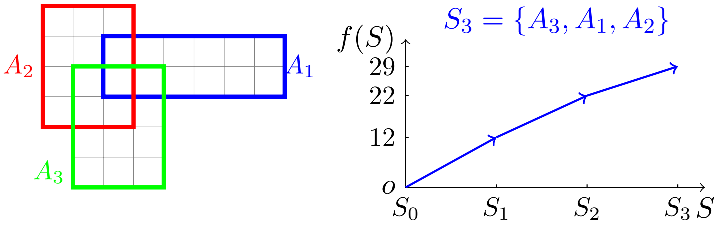




Step 3

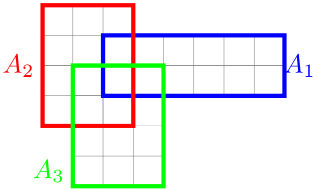


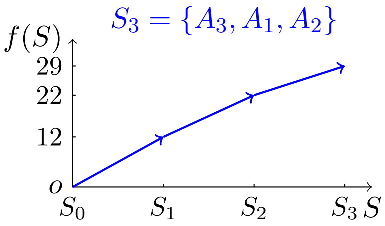
- A_2 was selected as $f_{S_2}(A_2) = 7 > f_{S_2}(A_4) = 6$. Done.



Theorem

Let $S_k = \{x_1^*, x_2^*, ..., x_k^*\}$ *be the set returned by* GreedyCardinalityConstraint, *then* $f(S_k) \geq (1 - \frac{1}{e}) f(S^*)$.





Proof.

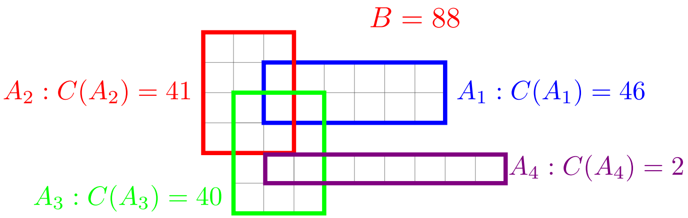
$$
\begin{aligned}
f(S^*) &\leq f(S_{i-1}) + \sum_{e \in S^* \setminus S_{i-1}} f_{S_{i-1}}(e) & (4)\\
&\leq f(S_{i-1}) + \sum_{e \in S^* \setminus S_{i-1}} f_{S_{i-1}}(x_i^*) & (5)\\
&\leq f(S_{i-1}) + \sum_{e \in S^* \setminus S_{i-1}} f_{S_{i-1}}(x_i^*) & (6)\\
&= f(S_{i-1}) + \sum_{e \in S^* \setminus S_{i-1}} (f(S_i) - f(S_{i-1})) & (7)\\
&\leq f(S_{i-1}) + k(f(S_i) - f(S_{i-1})) & (8)
\end{aligned}
$$

- Substracting $kf(S*)$ on both sides, we have: $f(S_i) - f(S^*) \geq (1 - \frac{1}{k})(f(S_{i-1}) - f(S^*))$
- By induction we further have: $f(S_k) \geq (1 - (1 - \frac{1}{k})^k) f(S^*)$.

□

- Now let's further consider the MaxCoverage problem with **knapsack constraint**, i.e., each element $e \in N$ is associated with a cost $C(e)$, and we have a budget B. We aims to select subsets such that the total cost is no more than B and the cardinality of their union is maximized.

$$\begin{array}{ll} \max & f(S) \\ s.t. & \sum_{e\in S} C(e) \leq B \end{array}$$



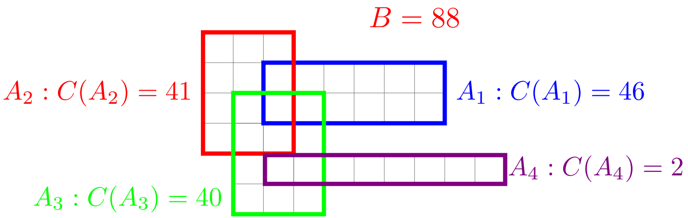
Greedy algorithm (knapsack constraint)

$\textsc{GreedyKnapsackConstraint}(N, \mathbf{C}, B)$

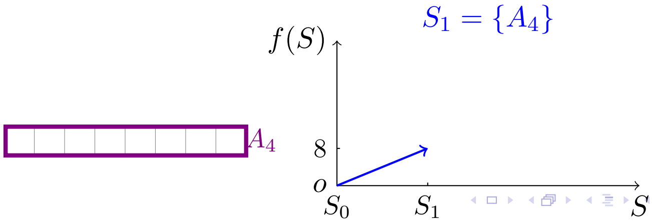
$S = \phi$;
while $\sum_{e \in S} C(e) < B$ **do**
  $x^* = argmax_{x \in N} \frac{f_S(x)}{C(x)}$;
  if $\sum_{e \in S} C(e) + C(x^*) < B$ **then**
    $S = S \cup \{x^*\}$;
  end if
  $N = N - \{x^*\}$;
end while
return S;

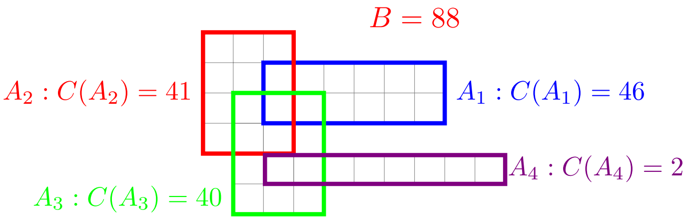
- Greedy rule: instead of maximizing marginal contribution at each step, the algorithm selects the cheapest item.

An example: Step 1

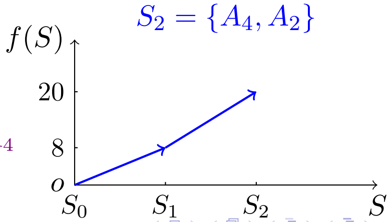


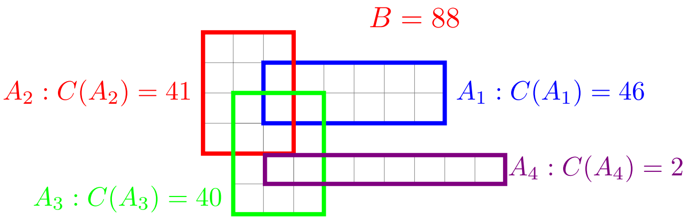
- Let $S_i = \{x_1^*, x_2^*, ..., x_i^*\}$ be the value of S after the i-th execution of the `while` loop. Initially A_4 was selected with $\frac{f_\phi(A_4)}{C(A_4)} = \frac{8}{2}$, which is larger than $A_1(\frac{12}{46})$, $A_2(\frac{12}{41})$ and $A_3(\frac{12}{40})$.



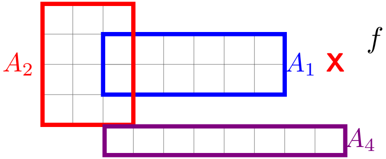


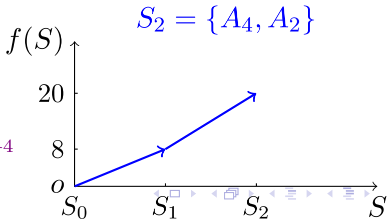
- A_2 was selected with $\frac{f_{S_1}(A_2)}{C(A_2)} = \frac{12}{41}$ as it is larger than $\frac{f_{S_1}(A_1)}{C(A_1)} = \frac{12}{46}$ and $\frac{f_{S_1}(A_3)}{C(A_3)} = \frac{10}{40}$.

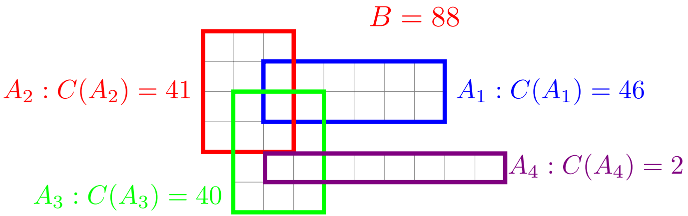




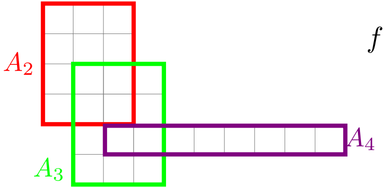
- A_1 was selected with $\frac{f_{S_2}(A_1)}{C(A_1)} = \frac{10}{46}$ as it is larger than $\frac{f_{S_2}(A_3)}{C(A_3)} = \frac{6}{40}$. However we cannot add A_1 to S_2 as $C(A_1) + C(A_2) + C(A_4) = 89 > 88$.

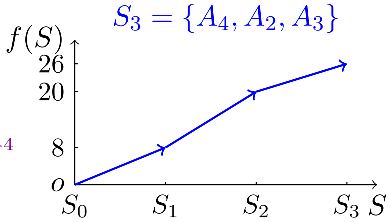






- A_3 was selected. The process ended as no subset can be added without incurrence of violation of knapsack constraint.





Analysis

-

Accelerated greedy algorithm