

CS711008Z Algorithm Design and Analysis

Lecture 7. Basic algorithm design technique: Greedy

Dongbo Bu

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

- Greedy is usually used to maximize or minimize a set function $f(S)$, where S is a subset of a ground set.
- Two examples to exhibit the connection with dynamic programming: `SINGLESOURCESHORTESTPATH` problem and `INTERVALSCHEDULING` problem.
- Elements of greedy technique.
- Other examples: `HUFFMAN CODE`, `SPANNING TREE`.
- Theoretical foundation of greedy technique: Matroid and submodular set functions.
- Important data structures: `BINOMIAL HEAP`, `FIBONACCI HEAP`, `UNION-FIND`.

Greedy technique

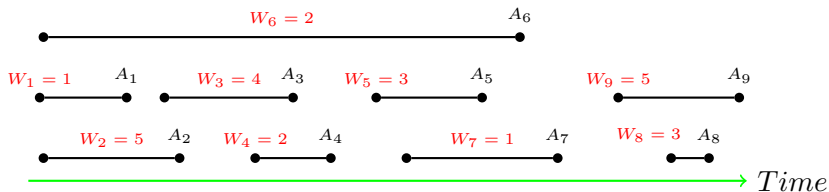
- Greedy technique typically applies to the **optimization problems** if:
 - ① The original problem can be divided into smaller subproblems.
 - ② The recursion among sub-problems can be represented as **optimal-substructure property**: the optimal solution to the original problem can be calculated through **combining the optimal solutions to subproblems**.
 - ③ We can design a **greedy-selection rule** to select a certain sub-problem at a stage.
- In particular, 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 $X = [x_1, x_2, \dots, x_n]$, $x_i = 0/1$.
- For this type of problems, we can construct **a tree to enumerate all possible decisions**, and 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.

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



Solutions: $S_1 = \{A_1, A_3, A_5, A_8\}$ | $S_2 = \{A_6, A_9\}$
Benefits: $B(S_1) = 1 + 4 + 3 + 3 = 11$ | $B(S_2) = 2 + 5 = 7$

INPUT:

n activities $A = \{A_1, A_2, \dots, A_n\}$ 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 .

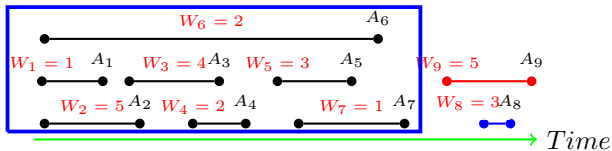
OUTPUT:

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 $i < j$.

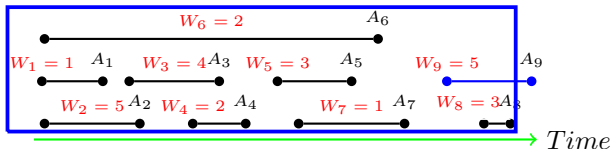
Defining general form of subproblems I

- 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. Let's describe the solving process as a series of decisions: at each decision step, an activity was chosen 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 .



Defining general form of subproblems II

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



Optimal sub-structure property

- Summarizing the two cases, we can design the general form of subproblems as: **selecting a collection of activities from A_1, A_2, \dots, A_i to maximize benefits**. Let's 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

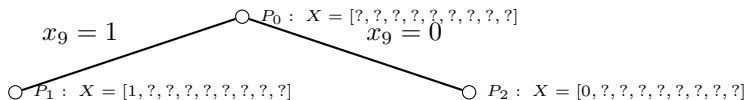
RECURSIVE_DP(i)

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

```
1: if  $i \leq 0$  then  
2:   return 0;  
3: end if  
4: if  $i == 1$  then  
5:   return  $W_1$ ;  
6: end if  
7: Let  $pre(i)$  denotes the largest index of the activities ending before  $S_i$   
8:  $m = \max \begin{cases} \text{RECURSIVE\_DP}(pre(i)) + W_i \\ \text{RECURSIVE\_DP}(i - 1) \end{cases}$   
9: return  $m$ ;
```

- The original problem can be solved by calling RECURSIVE_DP(n).
- It needs $O(n \log n)$ to sort the activities and determine $pre(\cdot)$, and the dynamic programming needs $O(n)$ time. Thus, the total running time is $O(n \log n)$

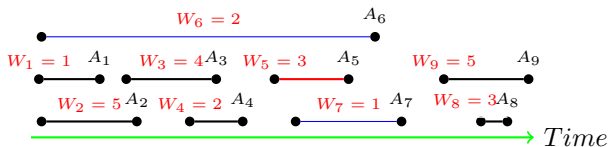
Multiple-stage decision process



- Here we represent a solution as $X = [x_1, x_2, \dots, x_9]$, where $x_i = 1$ denotes the selection of activity A_i and abandon otherwise.
- At the first decision step, we have to enumerate the two options $x_9 = 0$ and $x_9 = 1$, as we have no idea which one is optimal.

A more cumbersome dynamic programming algorithm

- 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. Let's describe the solving process as a series of decisions: at each decision step, an activity is chosen to use the resource.
- Suppose we have already worked out the optimal solution. Consider **the first decision** in the optimal solution, i.e. a certain activity A_i is selected. There are at most n options:
 - Select an activity A_i : the selection leads to a **smaller subproblem**, namely, selecting from the activity set with A_i and the activities conflicting with A_i removed.



Optimal sub-structure property

- Summarizing these cases, we can design the general form of subproblems as: **selecting a collection of activities from a subset S ($S \subseteq \{A_1, A_2, \dots, A_n\}$) to maximize benefits.**
Let's denote the optimal solution value as $OPT(S)$.
- Optimal substructure property: (“cut-and-paste” argument)

$$OPT(S) = \max_{A_i \in S} \{OPT(S') + W_i\}$$

Here, S' represents the subset with A_i and the activities conflicting with A_i removed from S .

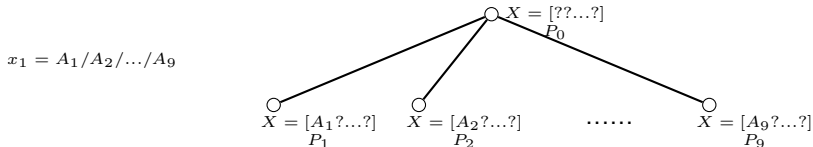
Dynamic programming algorithm

RECURSIVE_DP(S)

```
1: if  $S$  is empty then
2:   return 0;
3: end if
4:  $m = 0$ ;
5: for all activity  $A_i \in S$  do
6:   Set  $S'$  as the subset with  $A_i$  and the activities conflicting with  $A_i$ 
   removed from  $S$ ;
7:   if  $m < \text{RECURSIVE\_DP}(S') + W_i$  then
8:      $m = \text{Recursive\_DP}(S') + W_i$ ;
9:   end if
10: end for
11: return  $m$ ;
```

- The original problem can be solved by calling $\text{RECURSIVE_DP}(\{A_1, A_2, \dots, A_n\})$.
- The total running time is $O(2^n)$ as the number of subproblems is exponential.

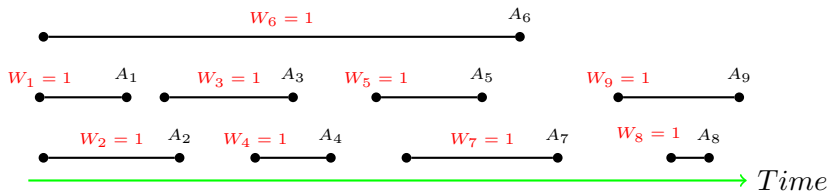
Multiple-stage decision process



- Here we represent a solution as $X = [x_1, x_2, \dots]$, where $x_i \in \{A_1, A_2, \dots, A_9\}$ denotes the activity selected at the i -th decision step.
- At the first decision step, we have to enumerate 9 options $x_1 = A_1, x_1 = A_2, \dots, x_1 = A_9$ as we have no idea which one is optimal.

INTERVALSCHEDULING problem: version 2

Let's investigate a special case



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

INPUT:

n activities $A = \{A_1, A_2, \dots, A_n\}$ 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 property: 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**, i.e., to select as many courses as possible, we first select the course with the earliest ending time.

Theorem

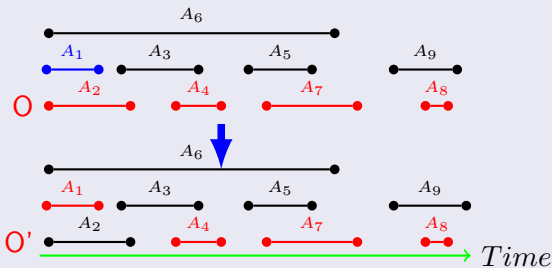
Suppose A_1 is the activity with the earliest ending time. A_1 is used in an optimal solution.

Another property: greedy-selection II

Proof.

(exchange argument)

- Suppose we have an optimal solution $O = \{A_{i1}, A_{i2}, \dots, A_{iT}\}$ but $A_{i1} \neq A_1$.
- A_1 is compatible with A_{i2}, \dots, A_{iT} since A_1 ends earlier than A_{i1} .
- Let's construct a new subset $O' = O - \{A_{i1}\} \cup \{A_1\}$. It is clear that O' is also an optimal solution since $|O'| = |O|$.



Simplifying the DP algorithm into a greedy algorithm

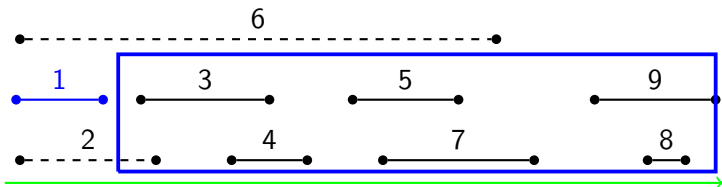
INTERVAL_SCHEDULING_GREEDY(n)

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

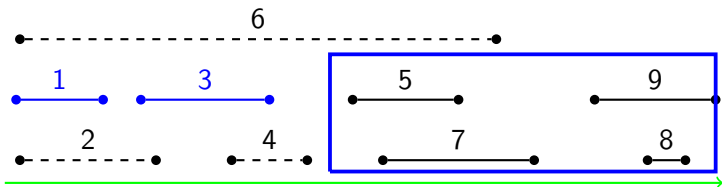
```
1: previous_finish_time =  $-\infty$ ;  
2: for  $i = 1$  to  $n$  do  
3:   if  $S_i \geq \textit{previous\_finish\_time}$  then  
4:     Select activity  $A_i$ ;  
5:      $\textit{previous\_finish\_time} = F_i$ ;  
6:   end if  
7: end for
```

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

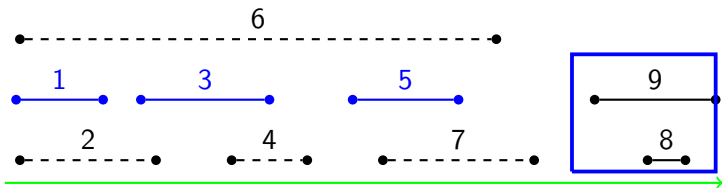
An example: Step 1



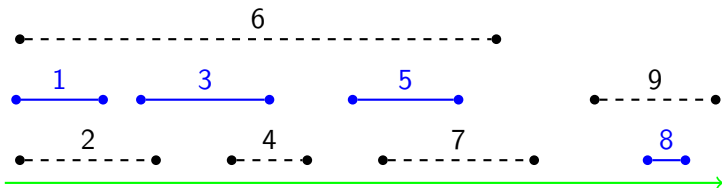
Step 2



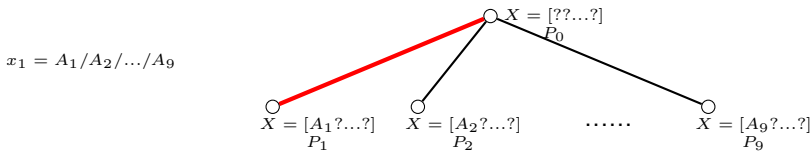
Step 3



Step 4



Greedy-selection rule in multiple-stage decision process



- Here we represent a solution as $X = [x_1, x_2, \dots]$, where $x_i \in \{A_1, A_2, \dots, A_9\}$ denotes the activity selected at the i -th decision step.
- At the first decision step, there are a total of 9 options $x_1 = A_1, x_1 = A_2, \dots, x_1 = A_9$.
- The dynamic programming technique has to enumerate 9 options $x_1 = A_1, x_1 = A_2, \dots, x_1 = A_9$ as it is unknown which one is optimal. In contrast, the greedy algorithm selects A_1 directly according to the greedy-selection property.

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

Similarities:

- 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

DP versus Greedy cont'd

Differences:

- 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?

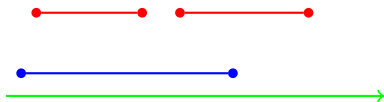
Two strategies:

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

Trying other greedy rules

Incorrect trial 1: **earliest start** rule

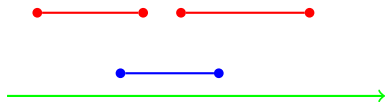
- Intuition: the earlier start time, the better.
- Incorrect. A negative example:



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

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

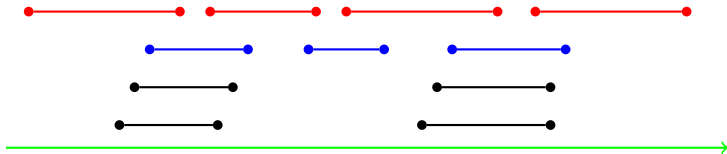
- Intuition: the shorter duration, the better.
- Incorrect. A negative example:



- 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

Revisiting SINGLE SOURCE SHORTEST PATHS problem

INPUT:

A directed graph $G = \langle V, E \rangle$. Each edge $e = \langle i, j \rangle$ 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

Optimal sub-structure property

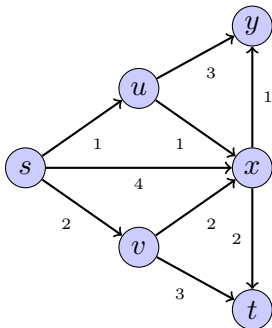
- Solution: a path from s to t with at most $(n - 1)$ edges.
Describing the solving process as making a series of decisions; at each decision step, we decide the subsequent node.
- Suppose we have already obtained an optimal solution O . Consider the final decision (i.e. from which we reach node t) within O . There are several possibilities for the decision:
 - node v such that $\langle v, t \rangle \in E$: 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:
$$OPT(v, k) = \min \begin{cases} OPT(v, k - 1) \\ \min_{\langle u, v \rangle \in E} \{OPT(u, k - 1) + d_{u,v}\} \end{cases}$$
- 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)

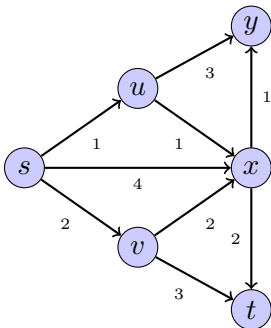
```
1: for  $i = 0$  to  $n$  do
2:    $OPT[s, i] = 0$ ;
3: end for
4: for all node  $v \in V$  do
5:    $OPT[v, 0] = \infty$ ;
6: end for
7: for  $k = 1$  to  $n - 1$  do
8:   for all node  $v$  (in an arbitrary order) do
9:      $OPT[v, k] = \min \begin{cases} OPT[v, k - 1], \\ \min_{\langle u, v \rangle \in E} \{OPT[u, k - 1] + d(u, v)\} \end{cases}$ 
10:   end for
11: end for
12: return  $OPT[t, n - 1]$ ;
```

An example: Step 1



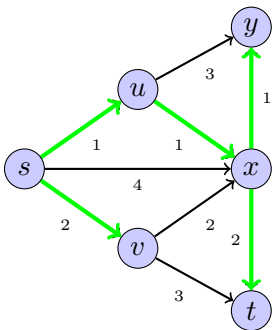
	k=0	1	2	3	4	5
s	0	0	0	0	0	0
u	—	1				
v	—	2				
x	—	4				
y	—	—				
t	—	—				

Step 2



	k=0	1	2	3	4	5
<i>s</i>	0	0	0	0	0	0
<i>u</i>	–	1	1			
<i>v</i>	–	2	2			
<i>x</i>	–	4	2			
<i>y</i>	–	–	4			
<i>t</i>	–	–	5			

Final step

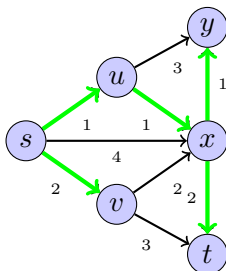


	k=0	1	2	3	4	5
<i>s</i>	0	0	0	0	0	0
<i>u</i>	–	1	1	1	1	1
<i>v</i>	–	2	2	2	2	2
<i>x</i>	–	4	2	2	2	2
<i>y</i>	–	–	4	3	3	3
<i>t</i>	–	–	5	4	4	4

- Recall that the collection of the shortest paths from s to all nodes form a shortest path tree.

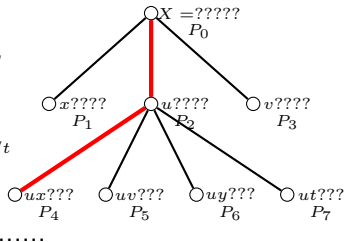
Greedy-selection property in version 2

Greedy-selection rule in multiple-stage decision process



$$x_1 = x/u/v$$

$$x_2 = x/v/y/t$$



- The construction of the shortest path tree rooted at s is a multiple-stage decision process: The tree is described as $X = [x_1, x_2, \dots, x_5]$, where $x_i \in V$ represents the node selected at the i -th stage. The Dijkstra's algorithm selects the nearest node adjacent to “explored” nodes at each stage.
- This greedy-selection rule works due to the following two observations of `BELLMAN_FORD` algorithm, namely, some operations are redundant, and the nearest node adjacent to “explored” nodes has its shortest distance determined.

Redundant calculations in `BELLMAN_FORD` algorithm

- 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.

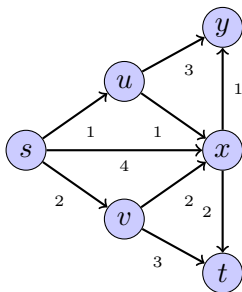
$$OPT(v^*, k) = \min \begin{cases} OPT(v^*, k - 1) \\ \min_{\langle u, v^* \rangle \in E} \{ OPT(u, k - 1) + d_{u, v^*} \} \end{cases}$$

- 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 $OPT(v^*, k) = OPT(v^*, k - 1)$ for $k = 2$



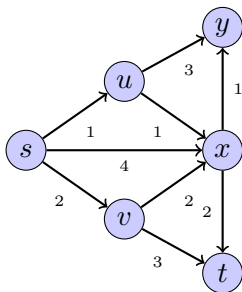
	k=0	1	2	3	4	5
s	0	0	0	0	0	0
u	-	1	1	1	1	1
v	-	2				
x	-	4				
y	-	-				
t	-	-				

- 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. Thus, the calculations of $OPT(v^*, k)$ (in green rectangles) are in fact redundant.
- In other words, it suffices to calculate

$$OPT[v, k] = \min \begin{cases} OPT[v, k - 1], \\ \min_{\langle u, v \rangle \in E} \{OPT[u, k - 1] + d(u, v)\} \end{cases} \quad \text{for}$$

the **unexplored nodes** $v \neq v^*$.

The meaning of $OPT(v^*, k) = OPT(v^*, k - 1)$ for $k = 3$



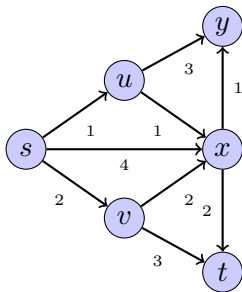
	k=0	1	2	3	4	5
s	0	0	0	0	0	0
u	—	1	1	1	1	1
v	—	2	2	2	2	2
x	—	4	2	2	2	2
y	—	—	4			
t	—	—	5			

- 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. Thus, the calculations of $OPT(v^*, k)$ (in green rectangles) are in fact redundant.
- In other words, it suffices to calculate

$$OPT[v, k] = \min \begin{cases} OPT[v, k - 1], \\ \min_{\langle u, v \rangle \in E} \{OPT[u, k - 1] + d(u, v)\} \end{cases} \quad \text{for}$$

the **unexplored nodes** $v \neq v^*$.

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
t	—	—	5	4	4	4

- 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. Thus, the calculations of $OPT(v^*, k)$ (in green rectangles) are in fact redundant.
- In other words, it suffices to calculate

$$OPT[v, k] = \min \begin{cases} OPT[v, k - 1], \\ \min_{\langle u, v \rangle \in E} \{OPT[u, k - 1] + d(u, v)\} \end{cases} \quad \text{for}$$

the **unexplored nodes** $v \neq v^*$.

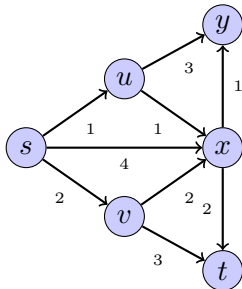
A faster implementation of BELLMAN_FORD

FAST_BELLMAN_FORD(G, s, t)

```
1:  $S = \{s\}$ ; //  $S$  denotes the set of explored nodes,
2: for  $i = 0$  to  $n$  do
3:    $OPT[s, i] = 0$ ;
4: end for
5: for all node  $v \in V$  do
6:    $OPT[v, 0] = \infty$ ;
7: end for
8: for  $k = 1$  to  $n - 1$  do
9:   for all node  $v \notin S$  (in an arbitrary order) do
10:     $OPT[v, k] = \min \begin{cases} OPT[v, k - 1], \\ \min_{\langle u, v \rangle \in E} \{OPT[u, k - 1] + d(u, v)\} \end{cases}$ 
11:   end for
12:   Add the nodes with minimum  $OPT(v, k)$  to  $S$ ;
13: end for
14: return  $OPT[t, n - 1]$ ;
```

Greedy-selection rule

- Now the question is how to efficiently calculate $OPT(v, k)$ for the **unexplored nodes** $v \notin S$. Take the example shown below. The unexplored nodes include v , x , y and t .



	k=0	1	2	3	4	5
s	0	0	0	0	0	0
u	-	1	1	1	1	1
v	-	2				
x	-	4				
y	-	-				
t	-	-				

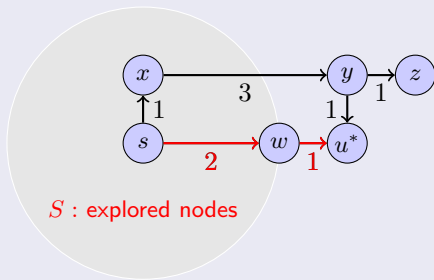
- Note that it is unnecessary to consider all unexplored nodes; instead, we can consider **only the unexplored nodes adjacent to an explored node**, i.e., nodes v , x , y . Furthermore, among these nodes, the nearest ones (v and x) have its shortest distance determined. Thus we can iteratively select such nodes until reaching node t .

Theorem

Let S denote the **explored** nodes, and for each explored node v , let $d(v)$ denote the shortest distance from s to v . **Consider the nearest unexplored node u^* adjacent to an explored node**, 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 $P = s \rightarrow \dots \rightarrow w \rightarrow u^*$ is one of the shortest paths from s to u^* with distance $d'(u^*)$.

Proof.

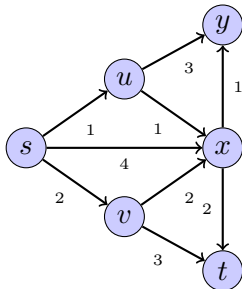
- Suppose there is another path P' from s to u^* that is shorter than P .
- Without loss of generality, we denote $P' = s \rightarrow \dots \rightarrow x \rightarrow y \rightarrow \dots \rightarrow u^*$. Here, y denotes the first node in P' leaving out of S .
- But $|P'| \geq d(s, x) + d(x, y) \geq d'(u^*)$.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



S : explored nodes

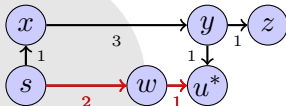
BELLMAN_FORD vs DIJKSTRA: two differences

- 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				
x	-	4				
y	-	-				
t	-	-				

- Let's u^* denote the nearest unexplored node adjacent to an explored node. The shortest path from s to u^* is determined.



S : explored nodes

Dijkstra's algorithm [1959]

DIJKSTRA(G, s, t)

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

- Lines (9 – 11) are 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 $d(v)$.
- When $d_{u,v} = 1$ for any edge $\langle u, v \rangle$, the Dijkstra's algorithm reduces to BFS. Thus, the Dijkstra's algorithm can be treated as

weighted BFS

Implementing Dijkstra's algorithm using priority queue

DIJKSTRA(G, s, t)

- 1: $key(s) = 0$; $//key(u)$ stores an upper bound of the shortest distance from s to u ;
- 2: $PQ.$ INSERT (s);
- 3: **for all** node $v \neq s$ **do**
- 4: $key(v) = +\infty$
- 5: $PQ.$ INSERT (v) $//n$ times
- 6: **end for**
- 7: $S = \{\}$; $//$ Let S be the set of explored nodes;
- 8: **while** $S \neq V$ **do**
- 9: $v^* = PQ.$ EXTRACTMIN(); $//n$ times
- 10: $S = S \cup \{v^*\}$;
- 11: **for all** $v \notin S$ and $\langle v^*, v \rangle \in E$ **do**
- 12: **if** $key(v) + d(v^*, v) < key(v)$ **then**
- 13: $PQ.$ DECREASEKEY($v, key(v^*) + d(v^*, v)$); $//m$ times
- 14: **end if**
- 15: **end for**
- 16: **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 — 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:

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

- ② No negative edge: This stronger constraint on edge weights implies greedy-selection property. In particular, it is unnecessary to calculate $OPT(v, i)$ for any explored node $v \in S$, and for the nearest unexplored node adjacent to an explored node, its shortest distance is determined.

Time complexity analysis

Time complexity of The Dijkstra's 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	$O(n^2)$	$O(m \log n)$	$O(m \log n)$	$O(m + n \log n)$

The Dijkstra's 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



- 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 amount.

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.

$$d'(u, v) = d(u, v) + c(u) - c(v)$$

- 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 $\text{dist}(s^*, v)$, the shortest distance from s^* to v .
- We can prove that for any node pair u and v ,

$$d'(u, v) = d(u, v) + \text{dist}(u) - \text{dist}(v) \geq 0$$

Johnson algorithm for all pairs shortest path [1977]

JOHNSON(G)

- 1: Create a new node s^* ;
- 2: **for all** node $v \neq s^*$ **do**
- 3: $d(s^*, v) = 0$
- 4: **end for**
- 5: Run BELLMAN_FORD algorithm to calculate the shortest distance from s^* to each node v (denoted as $dist(s^*, v)$);
- 6: Reweighting: $d'(u, v) = d(u, v) + dist(s^*, u) - dist(s^*, v)$
- 7: **for all** node $u \neq s^*$ **do**
- 8: Run Dijkstra's algorithm with the new weight d' to calculate the shortest paths from u to any node v (denoted as $d^*(u, v)$);
- 9: **for all** node $v \neq s^*$ **do**
- 10: $d^*(u, v) = d^*(u, v) - dist(s^*, u) + dist(s^*, v)$;
- 11: **end for**
- 12: **end for**

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

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

Binary heap, Binomial heap, and Fibonacci heap

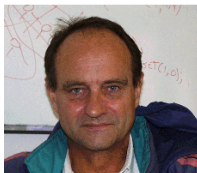
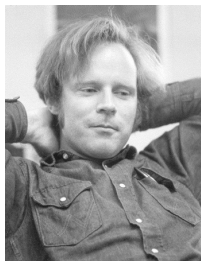


Figure 1: Robert W. Floyd, Jean Vuillienmin, Robert Tarjan

(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	$E(L) = 140$

INPUT:

a set of symbols $S = \{s_1, s_2, \dots, s_n\}$ with its appearance frequency $P = \{p_1, p_2, \dots, p_n\}$;

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

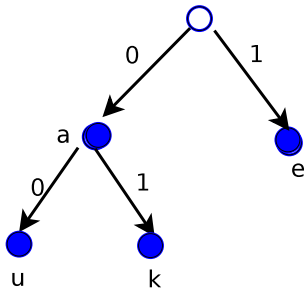
Ex1:

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

What is 0101?

aeae
k k

Ambiguity!

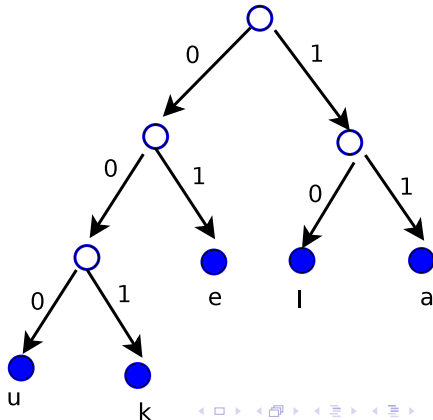


Ex2:

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

What is 1001000001?

l e u



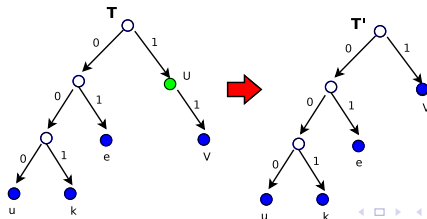
Full binary tree

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.

A top-down multiple-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

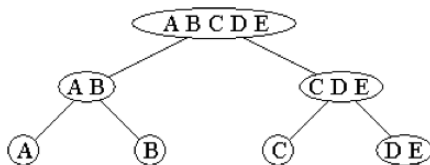
An example: Step 1

Symbol	Freq- quency	1. Step		2. Step		3. Step	
		Sum	Kode	Sum	Kode	Sum	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



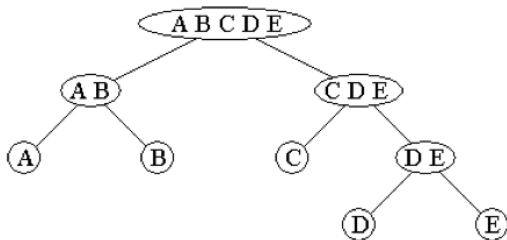
An example: Step 2

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



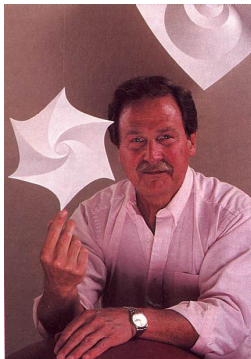
A bottom-up multiple-decision process

- 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:

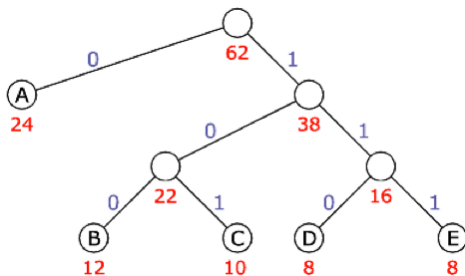
- ① In an optimal tree, $depth(u) \geq depth(v)$ iff $P_u \leq P_v$.
(Exchange argument)
- ② There is an optimal tree, where the lowest-frequency letters Y and Z are siblings. (Why?)
 - Consider a deepest node v .
 - v 's parent, denoted as u , should have another child, say w .
 - w should also be a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

Huffman code algorithm 1952

HUFFMAN(S, P)

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

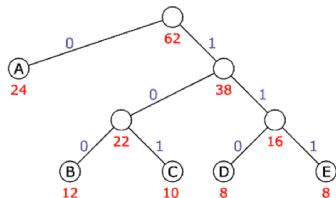
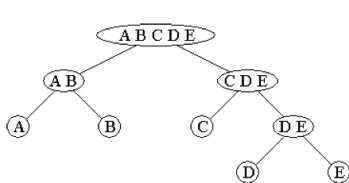
Example



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 (3 bit code)	tot. 138 bit
------------------------------	--------------

Shannon-Fano vs. Huffman



Sym.	Freq.	Shannon-Fano			Huffman		
		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.



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).

- 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.,

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

Here, $\mathcal{F} \subseteq 2^N$ 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.
 - 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:

$$V_1 = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$

$$V_2 = \begin{bmatrix} 1 & 4 & 9 & 16 & 25 \end{bmatrix}$$

$$V_3 = \begin{bmatrix} 1 & 8 & 27 & 64 & 125 \end{bmatrix}$$

$$V_4 = \begin{bmatrix} 1 & 16 & 81 & 256 & 625 \end{bmatrix}$$

$$V_5 = \begin{bmatrix} 2 & 6 & 12 & 20 & 30 \end{bmatrix}$$

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

Calculating maximal number of independent vectors

INDEPENDENTSET(M)

```
1:  $S = \{\}$ ;  
2: for all row vector  $v$  do  
3:   if  $S \cup \{v\}$  is still independent then  
4:      $S = S \cup \{v\}$ ;  
5:   end if  
6: end for  
7: return  $S$ ;
```

- Here we adopt the INDEPENDENCE ORACLE model for \mathcal{M} :
given $S \subseteq N$, the oracle returns whether S is independent or not.

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:

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

- 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.

An extension to weighted vectors

- 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:

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

A general greedy algorithm (by Jack Edmonds [1971])

MATROIDGREEDY(M, W)

```
1:  $S = \{\}$ ;  
2: Sort row vectors in the decreasing order of their weights;  
3: for all row vector  $v$  do  
4:   if  $S \cup \{v\}$  is still independent then  
5:      $S = S \cup \{v\}$ ;  
6:   end if  
7: end for  
8: return  $S$ ;
```

- Time complexity: $O(n \log n + nC(n))$, where $C(n)$ is the time needed to query the INDEPENDENCE ORACLE.

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 $W(A) = W(A') + W(v)$.



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



- 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*, Hassler 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

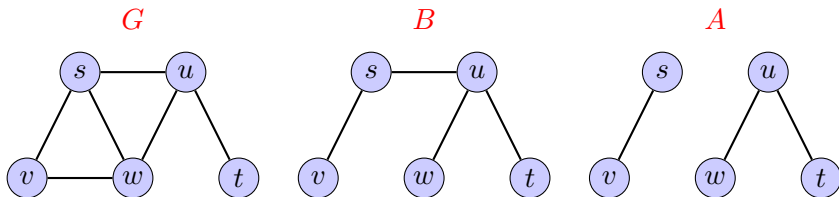
- Let's consider the **linear independence** for vectors.
 - ① **Hereditary property:** if B is an **independent vector set** and $A \subset B$, then A is also an **independent vector set**
 - ② **Augmentation property:** if both A and B are **independent vector sets**, and $|A| < |B|$, then there is a vector $v \in B - A$ such that $A \cup \{v\}$ is still an **independent vector set**
- Example:

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

- We have two independent vector sets: $A = \{V_1, V_3, V_5\}$, $B = \{V_1, V_2, V_3, V_4\}$, and $|A| < |B|$. The augmentation of A , $A \cup \{V_4\}$, is also independent.

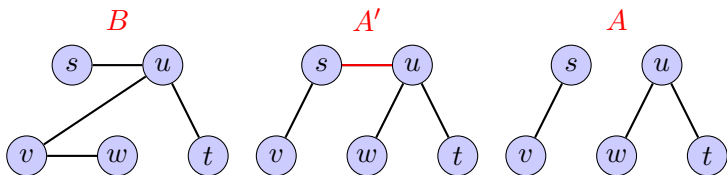
Origin 2 of matroid: acyclic subgraph [H. Whitney, 1932]

- Given a graph $G = \langle V, E \rangle$, 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**



Origin 2 of matroid: acyclic subgraph

- **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.)
 - This can also be proved through examining the **incidence matrix** of G : a linear dependence among columns (edges) corresponds to a cycle.



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**. The augmentation property is equivalent to the fact that all bases have the same cardinality, which is denoted as **rank**.
- **Circuit:** The minimal dependent sets are denoted as circuits, which are completely dual to the maximal independent sets. In fact, matroids can also be characterized in terms of circuits.
- **Bijection basis exchange:** If B_1 and B_2 are two bases of a matroid $M = (N, \mathcal{I})$, then there exists a bijection $\phi : B_1 \setminus B_2 \rightarrow B_2 \setminus B_1$ such that:

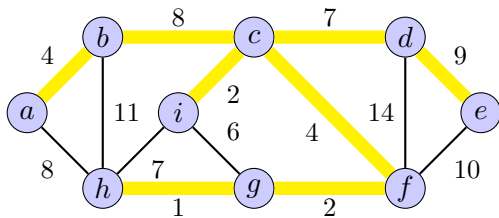
$$\forall x \in B_1 \setminus B_2, B_1 - x + \phi(x) \in \mathcal{I}$$

SPANNING TREE: an application of matroid

MINIMUM SPANNING TREE problem

- 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.

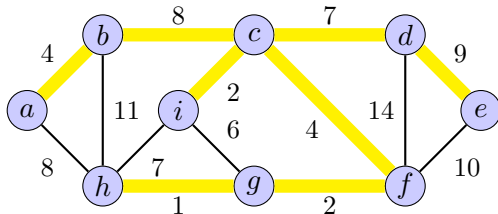


MINIMUM SPANNING TREE problem

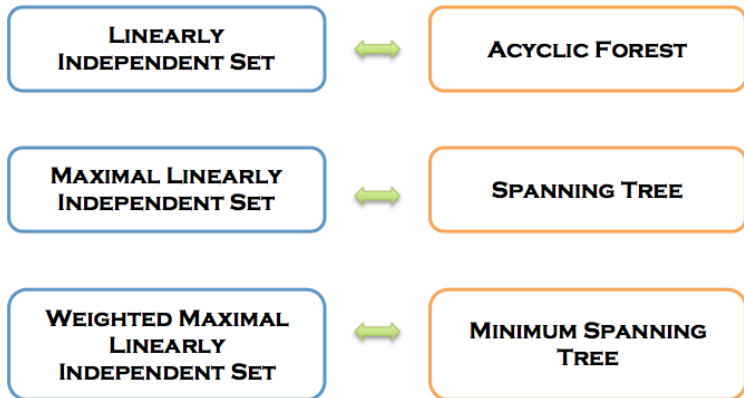
Input: A graph G , and each edge $e = \langle u, v \rangle$ is associated with a weight $W(u, v)$;

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.



INDEPENDENT VECTOR SET versus ACYCLIC FOREST



GENERIC SPANNING TREE algorithm

- 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

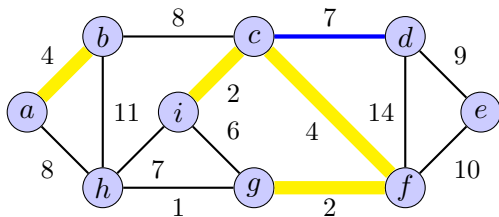


Figure 3: Safe edge

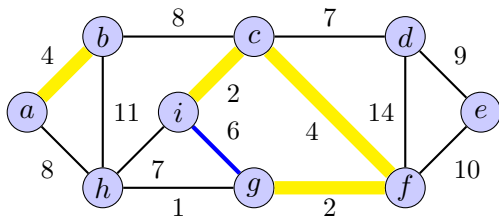


Figure 4: Unsafe edge

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)

```
1:  $F = \{\}$ ;
2: for all vertex  $v \in V$  do
3:   MAKESET( $v$ );
4: end for
5: Sort the edges of  $E$  in an nondecreasing order by weight  $W$ ;
6: for each edge  $(u, v) \in E$  in the order do
7:   if FINDSET( $u$ )  $\neq$  FINDSET( $v$ ) then
8:      $F = F \cup \{(u, v)\}$ ;
9:     UNION ( $u, v$ );
10:  end if
11: end for
```

Here, UNION-FIND structure is used to detect whether a set of edges form a cycle.

(See slides on UNION-FIND data structure, and a demo of Kruskal algorithm)

- 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.

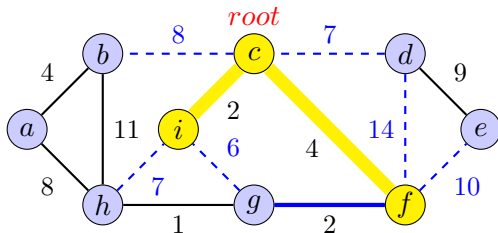


Figure 6: Robert C. Prim

Greedy-selection property

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($G, W, root$)

```
1: for all node  $v \in V$  and  $v \neq root$  do
2:    $key[v] = \infty$ ;
3:    $\Pi[v] = \text{NULL}$ ; //  $\Pi(v)$  denotes the predecessor node of  $v$ 
4:    $PQ.\text{INSERT}(v)$ ; // n times
5: end for
6:  $key[root] = 0$ ;
7:  $PQ.\text{INSERT}(root)$ ;
8: while  $PQ \neq \text{NULL}$  do
9:    $u = PQ.\text{EXTRACTMIN}()$ ; // n times
10:  for all  $v$  adjacent with  $u$  do
11:    if  $W(u, v) < key(v)$  then
12:       $\Pi(v) = u$ ;
13:       $PQ.\text{DECREASEKEY}(W(u, v))$ ; // m times
14:    end if
15:  end for
16: 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

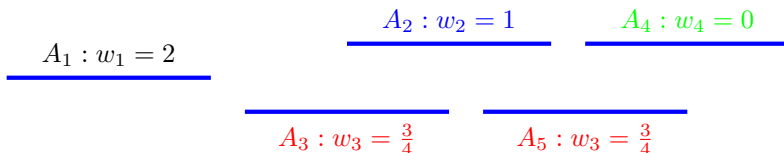
Operation	Linked list	Binary heap	Binomial heap	Fibonacci heap
MAKEHEAP	1	1	1	1
INSERT	1	$\log n$	$\log n$	1
EXTRACTMIN	n	$\log n$	$\log n$	$\log n$
DECREASEKEY	1	$\log n$	$\log n$	1
DELETE	n	$\log n$	$\log n$	$\log n$
UNION	1	n	$\log n$	1
FINDMIN	n	1	$\log n$	1
PRIM	$O(n^2)$	$O(m \log n)$	$O(m \log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

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

Why does the greedy algorithm fail for the weighted
INTERVALSCHEDULING problem?

Greedy algorithm fails for the weighted INTERVALSCHEDULING problem

- Matroid covers many cases of practical interests, and it is useful when determining whether greedy technique yields optimal solutions. However, greedy algorithm fails for the weighted INTERVALSCHEDULING problem.



Solutions:	Greedy : $\{A_1, A_2, A_4\}$		OPT : $\{A_1, A_3, A_5\}$
Benefits:	$2 + 1 + 0 = 3$		$2 + \frac{3}{4} + \frac{3}{4} = 3.5$

Independence in INTERVALSCHEDULING problem

$$\underline{A_1 : w_1 = 2}$$

$$\underline{A_2 : w_2 = 1}$$

$$\underline{A_4 : w_4 = 0}$$

$$\underline{A_3 : w_3 = \frac{3}{4}}$$

$$\underline{A_5 : w_3 = \frac{3}{4}}$$

- Let's think of a set of intervals as “independent” if they don't conflict each other. Let N denote the interval set, and \mathcal{I} represent the family of all independent interval sets.
- Let's examine the following properties:
 - **Hereditary property:** Any subset of an independent interval set is still independent.
 - **Augmentation property:** Consider $A = \{A_1, A_3, A_5\}$ and $B = \{A_1, A_2\}$. Although $|A| > |B|$, the augmentation $B \cup \{x\}$ with any interval $x \in A \setminus B$ is not independent.
- Thus $M = (N, \mathcal{I})$ doesn't form a matroid.
- We claim that if $M = (N, \mathcal{I})$ doesn't form a matroid, there definitely exists a weighting schema that causes the greedy algorithm to fail.

Let's examine whether the greedy algorithm works perfectly all the time

Theorem

*Suppose that $M = (N, \mathcal{I})$ is an independence system, i.e., \mathcal{I} has the **hereditary property**. Then M is a matroid iff for **any nonnegative weighting schema** over N , the greedy algorithm returns a basis of the maximum weight.*

- Here each element $x \in N$ is associated with a nonnegative weight $w(x)$, and the weight of a subset $S \subseteq N$ is defined as the total weights of the elements in S .
- We consider the greedy algorithm that iteratively adds the heaviest element that maintain independence.

- Suppose we have an independent system $M = (N, \mathcal{I})$ but it doesn't satisfy the augmentation property. We prove the theorem by constructing a weighting schema that causes the greedy algorithm to fail.
- Let A, B be independent sets with $|A| = |B| + 1$, but the addition of any element $x \in A \setminus B$ to B never gives an independent set, say $A = \{A_1, A_3, A_5\}$ and $B = \{A_1, A_2\}$ in the following example.

$$\begin{array}{ccc}
 \underline{A_1 : w_1 = 2} & \underline{A_2 : w_2 = 1} & \underline{A_4 : w_4 = 0} \\
 & \underline{A_3 : w_3 = \frac{3}{4}} & \underline{A_5 : w_5 = \frac{3}{4}}
 \end{array}$$

- We construct the following weighting schema:

$$w(x) = \begin{cases} w_1 & \text{if } x \in A \cap B \\ w_2 & \text{if } x \in B \setminus A \\ w_3 & \text{if } x \in A \setminus B \\ w_4 & \text{if } x \in \bar{A} \cap \bar{B} \end{cases}$$

- The weights w_1, w_2, w_3, w_4 were designed as below:
 - $w_1 > w_2 > w_3 > w_4 = 0$: Thus the greedy algorithm will choose elements in $A \cap B$ first, then $B \setminus A$, and finally $\bar{A} \cap \bar{B}$. Note that the elements in $A \setminus B$ will not be selected since the addition of such element to B never gives an independent interval set.
 - $w_1|A \cap B| + w_2|B \setminus A| < w_1|A \cap B| + w_3|A \setminus B|$: The first term represents the benefits returned by the greedy algorithm, while the second one the benefits returned by augmenting based on A . Thus the inequality implies the failure of the greedy algorithm.
- To achieve these two objectives simultaneously, we set w_1, w_2, w_3, w_4 as:

$$\begin{aligned}
 w_1 &= 2 \\
 w_2 &= \frac{1}{|B \setminus A|} \\
 w_3 &= \frac{1+\epsilon}{|A \setminus B|} \\
 w_4 &= 0
 \end{aligned}$$

where $0 < \epsilon < \frac{1}{|B \setminus A|}$. We use $\epsilon = \frac{1}{2}$ in the above example.

Tight connection between matroid and greedy algorithm

- On one side, if you can prove that the problem of interest is a matroid, then you have powerful algorithm automatically.
- On the other side, if the greedy algorithm works perfectly all the time, then the problem might be a matroid.

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:

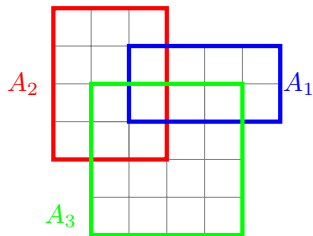
$$\begin{array}{ll} \max / \min & f(S) \\ \text{s.t.} & S \in \mathcal{F} \end{array}$$

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

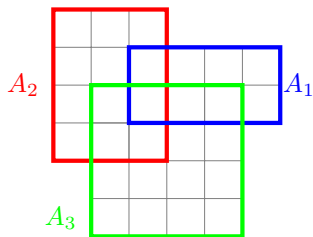
- Consider a set of n elements $N = \{1, 2, \dots, n\}$, and m subsets $A_1, A_2, \dots, A_m \subseteq N$. The goal of MAXCOVERAGE problem is to select k subsets such that the cardinality of their union is maximized.

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



Set function

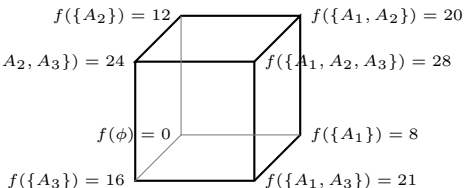
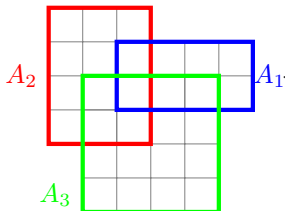
- The objective function in the MAX COVERAGE problem $f(S) = |\bigcup_{A_i \in S}|$ is a **set function** defined over subsets.



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 from cube

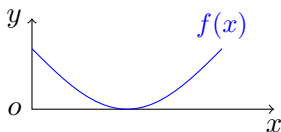
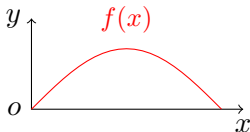
- A set function $f : \{0, 1\}^m \rightarrow \mathbb{R}$ defines value for nodes of the cube.



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.

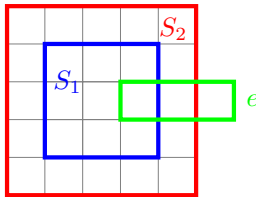
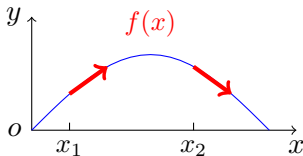


- A continuous function $f : \mathbb{R} \rightarrow \mathbb{R}$ can be efficiently maximized if it is **convex**, and can be efficiently maximized if it is **concave**.

Question: are there **discrete analogue** to convexity or concavity for set functions?

Submodularity: discrete analogue 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 gain** (discrete analogy to derivative) $f(S + e) - f(S)$ is non-increasing in S , i.e., if $S_1 \subseteq S_2$, $f(S_1 + e) - f(S_1) \geq f(S_2 + e) - f(S_2)$. (For simplicity, we use $f(S + e)$ to represent $f(S \cup \{e\})$ and use $f(e)$ to represent $f(\{e\})$.)

Submodular functions: decreasing marginal gain

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

Definition (marginal gain to a subset S)

The *marginal gain* 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 gain is discrete analogy to derivative of a continuous function, while “decreasing marginal gain” (or “diminishing returns”) definition of $f(S)$ is discrete analogy to concave functions.

An equivalent definition: subadditive

Definition (Submodular function)

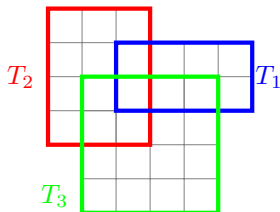
A function $f : 2^N \rightarrow \mathbb{R}$ is submodular iff $\forall A, B \subseteq N$,
 $f(A) + f(B) \geq f(A \cup B) + f(A \cap B)$.

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

$$f(A \cup B) + f(A \cap B) = |T_1 \cup T_2 \cup T_3| + |T_1| \quad (1)$$

$$\leq |T_1 \cup T_2| + |T_1 \cup T_3| \quad (2)$$

$$= f(A) + f(B) \quad (3)$$



- Subadditivity essentially implies “decreasing marginal gain”, 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 $f(S_2 \cup T) - f(S_2) \leq f(S_1 \cup T) - f(S_1)$.

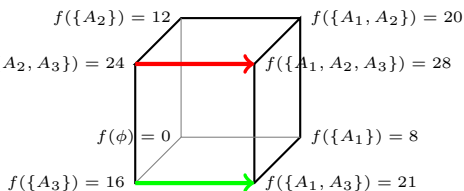
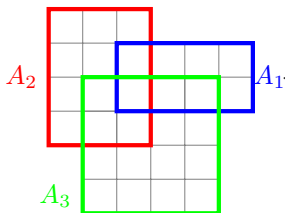
Examples of submodular functions

Example 1: Linear function and budget-additive functions

- A set function $f : 2^N \rightarrow \mathbb{R}$ is *linear* (also known as *additive*, *modular* if $f(S) = \sum_{i \in S} w_i$, where w_i denotes weight of element $i \in N$.
- Posing an upper-bound limitation on a modular function, we will have a monotone submodular function,
 $f(S) = \min\{\sum_{i \in S} w_i, B\}$ for any $w_i > 0, B > 0$.

Example 2: Set systems and coverage

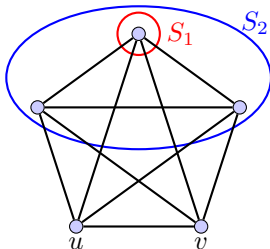
- Given a ground set N and several subset $A_1, A_2, \dots, A_n \subset N$, the coverage function $f(S) = |\bigcup_{i \in S} A_i|$ is submodular. This naturally extends to the weighted version:
 $f(S) = w(\bigcup_{i \in S} A_i)$, where $w : N \rightarrow \mathbb{R}_+$.



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

Example 3: Cut of graph

- Given a graph $G = \langle V, E \rangle$. 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,
 $f(S_1 + u) - f(S_1) = 8 - 4 = 4$, while
 $f(S_2 + u) - f(S_2) = 4 - 6 = -2$.

Example 4: Rank functions of matroid

- Given a matroid $M = (N, \mathcal{I})$, the rank function $r(S) = \max\{|A| : A \subseteq S, A \in \mathcal{I}\}$ is monotone submodular.
- This function can also extend to the weighted version, i.e., $r(S) = \max\{w(A) : A \subseteq S, A \in \mathcal{I}\}$, where $w : N \rightarrow \mathbb{R}_+$ represents a non-negative weighting function.
- For example, $r(\{V_1, V_2, V_3, V_4, V_5\}) = 4$.

$V_1 = [$	1	2	3	4	5]	$W_1 = 9$
$V_2 = [$	1	4	9	16	25]	$W_2 = 7$
$V_3 = [$	1	8	27	64	125]	$W_3 = 5$
$V_4 = [$	1	16	81	256	625]	$W_4 = 3$
$V_5 = [$	2	6	12	20	30]	$W_5 = 1$

Example 5: Valuation functions and welfare functions

- Sometimes we assume a function is submodular just because in some settings, it is natural to expect a decreasing marginal benefits.
- An example is the welfare function $f : 2^N \rightarrow \mathbb{R}_+$ on subsets of items. This might have a specific form
$$f(S) = \min\{\sum_{i \in S} w_i, B\}.$$

Example 6: Entropy of joint probability distribution

- Given a joint probability distribution $P(\mathbf{X})$ over discrete-valued random variables $\mathbf{X} = [x_1, x_2, \dots, x_n]$, the function $f(S) = H(\mathbf{X}_S)$ is monotone submodular, where H is the Shannon entropy, i.e.,

$$H(\mathbf{X}_S) = - \sum_{i \in S} P(x_i) \log P(x_i)$$

- If the random variables are real-valued, then $H(\mathbf{X}_S)$ is also submodular but not generally monotone.

Example 7: Facility location

- Given a set of locations $N = \{1, 2, \dots, n\}$ and m customers. If we open up a facility at location j , then it serves customer i with value $V_{i,j} \geq 0$. Each customer chooses the facility with the highest value. Thus, suppose we select a set of locations $S \subseteq N$ to open up facilities, the total value provided for these customers is:

$$f(S) = \sum_{i=1}^m \max_{j \in S} V_{i,j}$$

- $f(S)$ is monotone submodular.

Other examples of submodular functions

- Influence of advertisement over social networks: Assume you have a product and you want to advertise it in a social network. The problem is how to select nodes in the social network to maximize the influence.
- Word alignment: Consider a sentence in Chinese and its translation in English. We want to know the correspondence of words in the two sentences. The correspondence can be described using a bi-partite graph and measured using a submodular function.
- Documents summarization: Consider a set of related documents and we want to summarize them. The summarization can be measured using a submodular function.

Properties of submodular functions

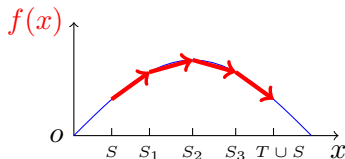
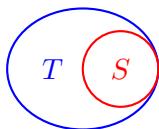
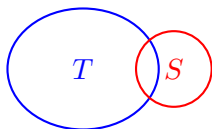
Several properties of submodular functions

- ① Non-negative linear combinations of submodular functions are still submodular, i.e., if f_1, f_2, \dots, f_n are submodular on the same ground set N , and $\omega_1, \omega_2, \dots, \omega_n$ are non-negative reals, then $\omega_1 f_1 + \omega_2 f_2 + \dots + \omega_n f_n$ is also submodular. This is important as when we are designing objective functions to maximize, we can first design some simple submodular pieces, and then combine them.
- ② Truncation of monotone submodular functions are still submodular, i.e., $\min(f(S), C)$ is submodular when $f(S)$ is monotone submodular and C is a constant.
- ③ If $f : 2^N \rightarrow \mathbb{R}_+$ is submodular, then the function g defined as $g(S) = \phi(f(S))$, where ϕ is concave, is also submodular.
- ④ f is submodular iff for any $S \subset N$, f_S is submodular.

A useful property of submodular functions: upper bound

Lemma

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

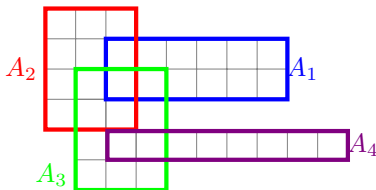


- This lemma can be easily proved by integrating marginal gains. Note that this is a discrete analogy to the property for a concave continuous function $f(x)$:

$$f(b) \leq f(a) + (b - a)f'(a) \text{ for } a < b.$$

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., $f(\phi) = 0$.

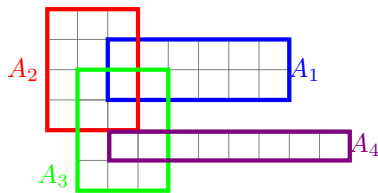
Greedy algorithm (cardinality constraint)

- Basic idea: at each step, the item with **the largest marginal gain** will be selected.

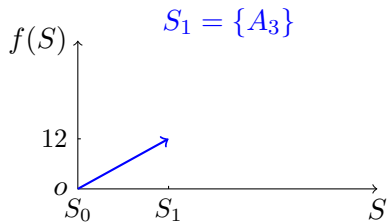
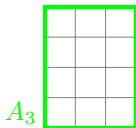
GREEDYCARDINALITYCONSTRAINT(k, N)

```
1:  $S = \phi$ ;  
2: while  $|S| < k$  do  
3:    $\hat{x} = \operatorname{argmax}_{x \in N} f_S(x)$ ;  
4:    $S = S \cup \{\hat{x}\}$ ;  
5:    $N = N - \{\hat{x}\}$ ;  
6: end while  
7: return  $S$ ;
```

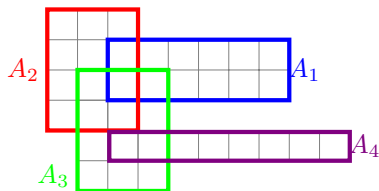
An example: Step 1



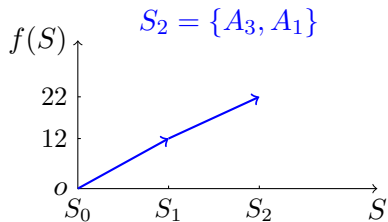
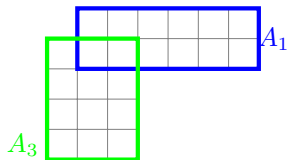
- Let $S_i = \{\hat{x}_1, \hat{x}_2, \dots, \hat{x}_i\}$ be the value of S after the i -th execution of the while loop. Initially A_3 was selected as $f_\phi(A_3) = 12$.



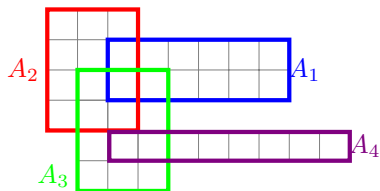
Step 2



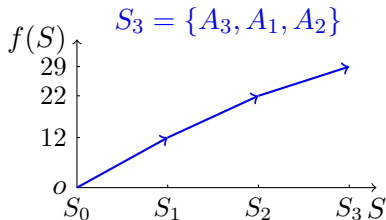
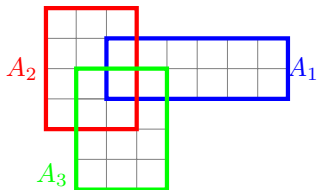
- A_1 was selected with $f_{S_1}(A_1) = 10$, which is larger than $f_{S_1}(A_2) = 8$, and $f_{S_1}(A_4) = 6$.



Step 3

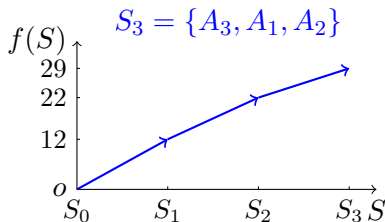
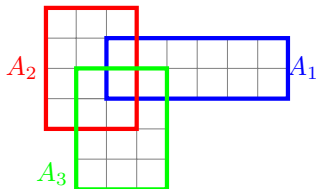


- A_2 was selected as $f_{S_2}(A_2) = 7 > f_{S_2}(A_4) = 6$. Done.



Theorem

Let $S_k = \{\hat{x}_1, \hat{x}_2, \dots, \hat{x}_k\}$ be the set returned by GREEDYCARDINALITYCONSTRAINT, then $f(S_k) \geq (1 - \frac{1}{e})f(S^*)$.



- Intuition: At each iteration, the gap $f(S^*) - f(S_i)$ was narrowed down. Let's consider the boundary case, i.e., after selecting k elements. In this case, the gap was reduced to be at most $\frac{1}{k}f(S^*)$.

- Let's consider the gap $g_{i-1} = f(S_{i-1}) - f(S^*)$.

$$f(S^*) \leq f(S_{i-1}) + \sum_{e \in S^* \setminus S_{i-1}} f_{S_{i-1}}(e) \quad (4)$$

$$\leq f(S_{i-1}) + \sum_{e \in S^* \setminus S_{i-1}} f_{S_{i-1}}(\hat{x}_i) \quad (5)$$

$$\leq f(S_{i-1}) + \sum_{e \in S^* \setminus S_{i-1}} f_{S_{i-1}}(\hat{x}_i) \quad (6)$$

$$= f(S_{i-1}) + \sum_{e \in S^* \setminus S_{i-1}} (f(S_i) - f(S_{i-1})) \quad (7)$$

$$\leq f(S_{i-1}) + k(f(S_i) - f(S_{i-1})) \quad (8)$$

- By subtracting $kf(S^*)$ on both sides, we can obtain the induction relationship between g_{i-1} and g_i as follows:

$$f(S_i) - f(S^*) \geq (1 - \frac{1}{k})(f(S_{i-1}) - f(S^*))$$

- By induction we further have $f(S_i) \geq (1 - (1 - \frac{1}{k})^i)f(S^*)$, and thus after k iterations, $f(S_k) \geq (1 - (1 - \frac{1}{k})^k)f(S^*) \geq (1 - \frac{1}{e})f(S^*)$.

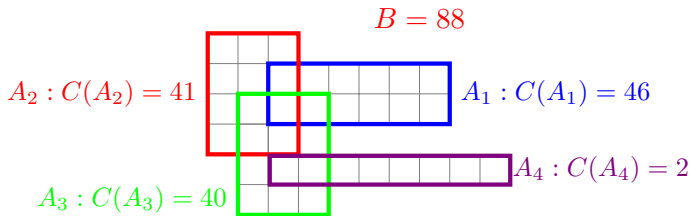
- In 1998, U. Feige proved the lower bound of $(1 - \frac{1}{e})$ for the approximation ratio of a polynomial time algorithm.
- But note that if it is allowed to sacrifice the cardinality constraints, the greedy algorithm can give much stronger guarantee. For example, running the greedy algorithm to select $2k$ elements gives an approximation $(1 - \frac{1}{k})^{2k} \approx 0.86$ approximation ratio. This is important because that in most practical cases, the constraints are rarely in stone.

MAXCOVERAGE problem under **knapsack constraint**

MAXCOVERAGE problem under **knapsack constraint**

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

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



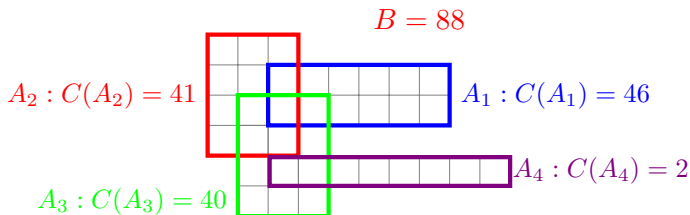
Greedy algorithm (knapsack constraint)

- Basic idea: at each step, the algorithm selects the item with **the highest benefit-cost ratio** rather than the item with **the largest marginal gain**.

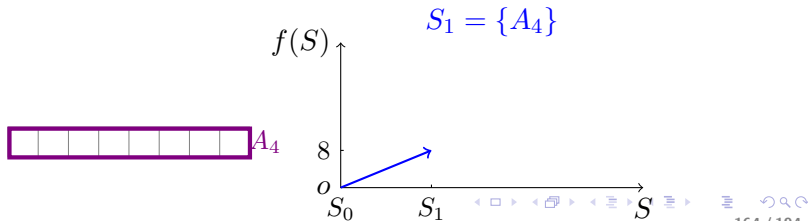
GREEDYKNAPSACKCONSTRAINT(N, \mathbf{C}, B)

```
1:  $S = \phi$ ;  
2: while  $N \neq \text{NULL}$  do  
3:    $\hat{x} = \operatorname{argmax}_{x \in N} \frac{f_S(x)}{C(x)}$ ;  
4:   if  $\sum_{e \in S} C(e) + C(\hat{x}) \leq B$  then  
5:      $S = S \cup \{\hat{x}\}$ ;  
6:   end if  
7:    $N = N - \{\hat{x}\}$ ;  
8: end while  
9: return  $S$ ;
```

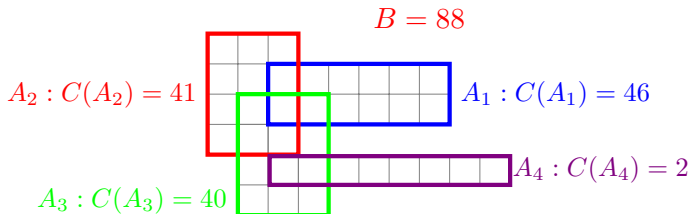
An example: Step 1



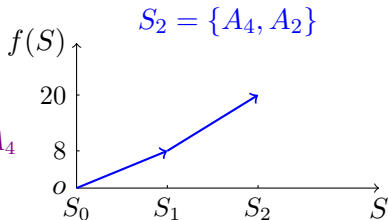
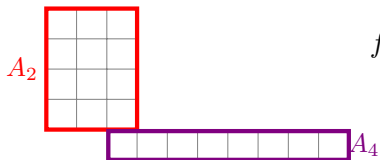
- Let $S_i = \{\hat{x}_1, \hat{x}_2, \dots, \hat{x}_i\}$ be the value of S after the i -th execution of the while loop. Initially A_4 was selected with $\frac{f_\phi(A_4)}{C(A_4)} = \frac{8}{2}$, which is larger than $A_1(\frac{12}{46})$, $A_2(\frac{12}{41})$ and $A_3(\frac{12}{40})$.



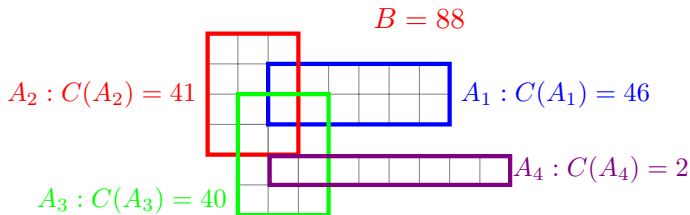
Step 2



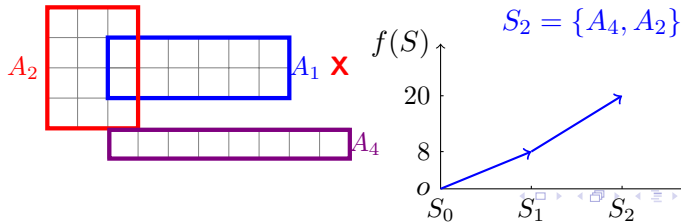
- A_2 was selected with $\frac{f_{S_1}(A_2)}{C(A_2)} = \frac{12}{41}$ as it is larger than $\frac{f_{S_1}(A_1)}{C(A_1)} = \frac{12}{46}$ and $\frac{f_{S_1}(A_3)}{C(A_3)} = \frac{10}{40}$.



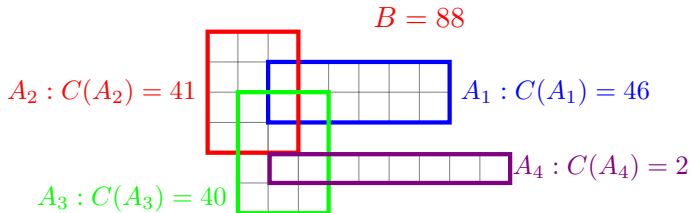
Step 3



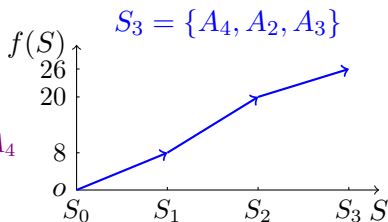
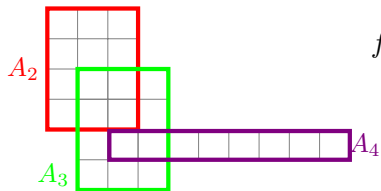
- A_1 was selected with $\frac{f_{S_2}(A_1)}{C(A_1)} = \frac{10}{46}$ as it is larger than $\frac{f_{S_2}(A_3)}{C(A_3)} = \frac{6}{40}$. However we cannot add A_1 to S_2 as $C(A_1) + C(A_2) + C(A_4) = 89 > 88$.



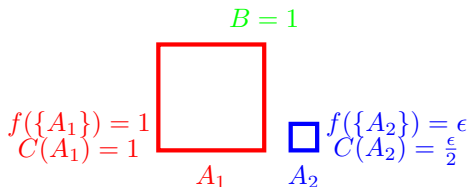
Step 4



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



- Unfortunately, the GREEDYKANPSACKCONSTRAINT algorithm has an unbounded approximation ratio.
- Consider the following instance: $N = \{e_1, e_2\}$, and $C(e_1) = 1$, $C(e_2) = \frac{\epsilon}{2}$ ($1 > \epsilon > 0$), and $B = 1$. The set function is $f(\{e_1\}) = 1$, and $f(\{e_2\}) = \epsilon$.

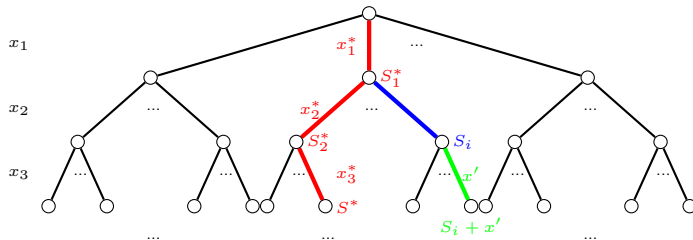


- The optimal solution is $S^* = \{e_1\}$ with $f(S^*) = 1$. In contrast, the GREEDYKANPSACKCONSTRAINT algorithm returns $S^* = \{e_2\}$ with $f(S^*) = \epsilon$. As ϵ approaches to 0, the approximation ratio becomes arbitrarily large.
- Reason: the algorithm selects elements according to **benefit-cost ratio** but totally ignores **the value of elements**.

Let consider the boundary case of knapsack constraint

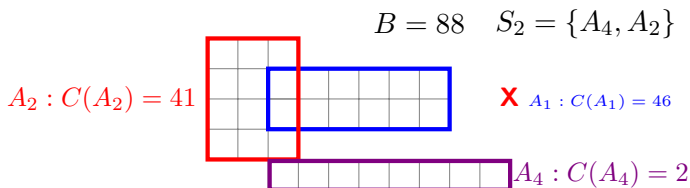
Theorem

Let S_i be the value of S at the i -th iteration. Suppose the addition of \hat{x} to S_i incurs the violation of the knapsack constraint, i.e., $C(S_i) + C(\hat{x}) > B$, then $f(S_i + \hat{x}) \geq (1 - \frac{1}{e})f(S^*)$.



- Intuition: at each iteration, the gap $f(S^*) - f(S_i)$ was narrowed down. The occurrence of the violation of knapsack constraint implies that the algorithm reaches a “boundary”. In this case, if adding one more element is permitted, we will obtain an upper bound for $f(S^*)$.

An example



- In the above example, A_1 was selected as it has the highest benefit-cost ratio $\frac{f_{S_2}(\{A_1\})}{C(A_1)} = \frac{10}{46}$. We cannot add A_1 to $S_2 = \{A_4, A_2\}$ as $C(S_2) + C(A_1) = 89 > 88$; however, the addition of A_1 provides an upper bound for $f(S^*)$, i.e., $f(\{A_4, A_2, A_1\}) = 30 \geq (1 - \frac{1}{e})f(S^*)$.

- Let's denote $S_i = \{\hat{x}_1, \hat{x}_2, \dots, \hat{x}_i\}$ and examine the gap $f(S_{i-1}) - f(S^*)$.

$$f(S^*) \leq f(S_{i-1}) + \sum_{x \in S^* \setminus S_{i-1}} f_{S_{i-1}}(x) \quad (9)$$

$$= f(S_{i-1}) + \sum_{x \in S^* \setminus S_{i-1}} f_{S_{i-1}}(x) \frac{C(x)}{C(x)} \quad (10)$$

$$\leq f(S_{i-1}) + \sum_{x \in S^* \setminus S_{i-1}} C(x) \frac{f_{S_{i-1}}(\hat{x}_i)}{C(\hat{x}_i)} \quad (11)$$

$$\leq f(S_{i-1}) + B \frac{f_{S_{i-1}}(\hat{x}_i)}{C(\hat{x}_i)} \quad (12)$$

$$= f(S_{i-1}) + B \frac{f(S_i) - f(S_{i-1})}{C(\hat{x}_i)} \quad (13)$$

- Next we can obtain the following recursion of gaps:

$$f(S_i) - f(S^*) \geq (1 - \frac{C(\hat{x}_i)}{B})(f(S_{i-1}) - f(S^*))$$

- Solving the recursion, we obtain the following bound:

$$f(S_i) \geq \left(1 - \prod_{k=1}^i \left(1 - \frac{C(\hat{x}_k)}{B}\right)\right) f(S^*) \quad (14)$$

$$\geq \left(1 - e^{-\frac{C(S_i)}{B}}\right) f(S^*) \quad (15)$$

- Now consider the boundary case of the increase in $f(S_i)$, i.e., $C(S_i) + C(\hat{x}) > B$, then if adding one more element was permitted:

$$f(S_i \cup \{\hat{x}\}) \geq \left(1 - e^{-\frac{C(S_i) + C(\hat{x})}{B}}\right) f(S^*) \quad (16)$$

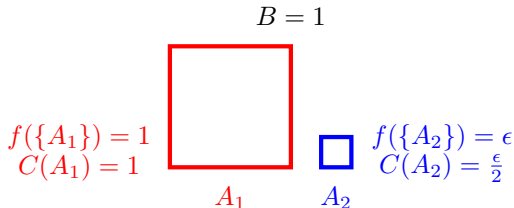
$$\geq \left(1 - \frac{1}{e}\right) f(S^*) \quad (17)$$

Improvement of GREEDYKNAPSACKCONSTRAINT: considering the value of elements [Leskovec, 2007]

- Basic idea: By **considering the value of elements**, the algorithm partly circumvents the shortcoming of GREEDYKNAPSACKCONSTRAINT. In the above example, the algorithm returns $\{A_1\}$ rather than $\{A_2\}$.

GREEDYKNAPSACKCONSTRAINT2(N, \mathbf{C}, B)

- 1: $e^* = \operatorname{argmax}_{e \in N, C(e) \leq B} f(\{e\})$;
- 2: $S_{G_1} = \text{GREEDYKNAPSACKCONSTRAINT}(N, \mathbf{C}, B)$;
- 3: **return** $\max\{f(S_{G_1}), f(\{e^*\})\}$;



Theorem

$f(S_{G_2}) \geq \frac{1}{2}(1 - \frac{1}{e})f(S^*)$, where S_{G_2} denotes the set returned by GREEDYKNAPSACKCONSTRAINT2.

Proof.

- After the last element of S_{G_1} was added, the while loop in GREEDYKNAPSACKCONSTRAINT1 should be executed at least once (an auxiliary element can be added to guarantee this).
- At these iterations, line 5 was evaluated to FALSE, i.e., $C(S_{G_1}) + C(\hat{x}) > B$. We consider the first iteration:

$$2f(S_{G_2}) \geq f(S_{G_1}) + f(e^*) \quad (18)$$

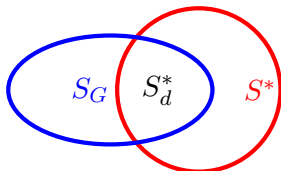
$$\geq f(S_{G_1}) + f(\hat{x}) \quad (19)$$

$$\geq f(S_{G_1} + \hat{x}) \quad (20)$$

$$\geq (1 - \frac{1}{e})f(S^*) \quad (21)$$

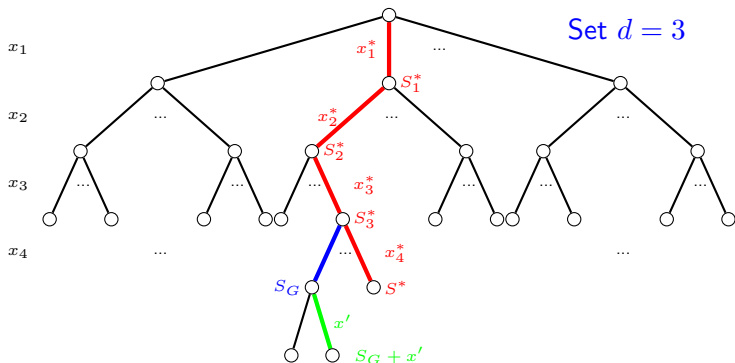
Improvement of GREEDYKNAPSACKCONSTRAINT:

Starting from **a good initial set**



- Basic idea: The algorithm circumvents the shortcoming of GREEDYKNAPSACKCONSTRAINT by considering value of items. Specifically, we divide S^* into two parts, namely, S_d^* and $S^* - S_d^*$, where S_d^* contains **the top d items with large contribution to $f(S^*)$** . Thus, if we run greedy algorithm starting from **the good initial set S_d^*** , the marginal gain of any element will not be higher than the average of $f(S_d^*)$.
- Question: How to obtain a good initial set?

Finding a good initial set using **partial enumeration**



- Suppose S^* has been sorted **in an order of decreasing contribution**. By enumerating all S of size $|S| = d$, we definitely know the first d items in S^* , denoted as S_d^* , in their correct order. These d items are important since if running greedy starting from S_d^* , the marginal gain at any step, say $f_{S_G}(x')$, will not be larger than the average of the first d items.

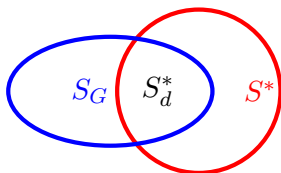
Greedy with partial enumeration [Khuller 1999, Sviridenko 2004]

GREEDYKNAPSACKCONSTRAINT3(N, \mathbf{C}, B)

- 1: $S' = \operatorname{argmax}_{C(S) \leq B, |S| < d} f(S)$;
- 2: **for all** S such that $C(S) \leq B, |S| = d$ **do**
- 3: $N' = N - S$;
- 4: S_{G_1} = Running GREEDYKNAPSACKCONSTRAINT on N' with S
 as initial set;
- 5: **if** $f(S_{G_1}) > f(S')$ **then**
- 6: $S' = S_{G_1}$;
- 7: **end if**
- 8: **end for**
- 9: **return** S' ;

Theorem

Let S^* denote the optimal solution, and S_G denote the set returned by GREEDYKNAPSACKCONSTRAINT3. By setting $d = 3$, $f(S_G) \geq (1 - \frac{1}{e})f(S^*)$.



Proof.

- Let's write S^* as $S^* = \{x_1^*, x_2^*, \dots, x_k^*\}$, and denote the first i items as $S_i^* = \{x_1^*, x_2^*, \dots, x_i^*\}$. Here these items are sorted **in a decreasing order of marginal gain**, i.e.,
$$x_i^* = \operatorname{argmax}_{x \in S^* \setminus S_{i-1}^*} f_{S_{i-1}^*}(x).$$
- First, we claim that after d iterations, the marginal gain of any element is upper-bounded. In particular,
$$\frac{1}{d}f(S_d^*) \geq f_{S_G}(x'),$$
 where $S_d^* \subseteq S_G$, and adding x' to S_G leads to the violation of knapsack constraint.
- Second, running GREEDYKNAPSACKCONSTRAINT using $S = S_d^*$ as initial set is equivalent to running with initial set $S = \phi$, $N' = N \setminus S_d^*$, and set function $f_{S_d^*}$. As $f_{S_d^*}$ is also sub-modular, $f(S_G + x') - f(S_d^*) \geq (1 - \frac{1}{e})(f(S^*) - f(S_d^*))$.

$$f(S_G) = f(S_G + x') - f_{S_G}(x') \quad (22)$$

$$\geq f(S_G + x') - \frac{1}{d}f(S_d^*) \quad (23)$$

$$\geq (1 - \frac{1}{e})f(S^*) + (\frac{1}{e} - \frac{1}{d})f(S_d^*) \quad (24)$$

$$\geq (1 - \frac{1}{e})f(S^*) \quad (25)$$

S_d^* provides upper bound for marginal gain in further steps

Lemma

After d iterations, the marginal gain of any element is upper-bounded. In particular, $\frac{1}{d}f(S_d^) \geq f_{S_G}(x')$, where $S_d^* \subseteq S_G$, and adding x' to S_G leads to the violation of knapsack constraint.*

Proof.

- Note that the items in S^* are sorted in a decreasing order of marginal gain, i.e., $x_i^* = \operatorname{argmax}_{x \in S^* \setminus S_{i-1}} f_{S_{i-1}^*}(x)$.
- Thus for $1 \leq k \leq d$,

$$f_{S_G}(x') \leq f_{S_{k-1}^*}(x') \quad (26)$$

$$\leq f(S_{k-1}^* + x_k^*) - f(S_{k-1}^*) \quad (27)$$

$$\leq f(S_k^*) - f(S_{k-1}^*) \quad (28)$$

- Summing for $1 \leq k \leq d$, we have: $f_{S_G}(x') \leq \frac{1}{d}f(S_d^*)$.



MAXCOVERAGE problem under **matroid constraint**

MAXCOVERAGE problem under **matroid constraint**

- Now let's further consider the MAXCOVERAGE problem with **matroid constraint**, i.e., given a matroid $\mathcal{M} = (N, \mathcal{I})$, where \mathcal{I} represents independent subsets of N .
- We aim to select **an independent subset** S from \mathcal{I} such that $f(S)$ is maximized.

$$\begin{array}{ll} \max & f(S) \\ \text{s.t.} & S \in \mathcal{I} \end{array}$$

An example: Welfare maximization

- Welfare maximization problem: Consider m items $I = \{I_1, I_2, \dots, I_m\}$ and n people. Each person i is associated with a submodular valuation function $f_i : 2^M \rightarrow \mathbb{R}_+$. The objective is to partition M into $M = S_1 \cup S_2 \cup \dots \cup S_n$ such that the total valuation $\sum_{i=1}^n f_i(S_i)$ is maximized.
- Construct a partition matroid: We first create n clones of each item, denoted as $N = \{I_{11}, I_{12}, \dots, I_{mn}\}$, where item I_{ij} has the color j . Then we think of a set of items as independent if no pair of items have identical color. Thus we construct a partition matroid $M = (N, \mathcal{I})$, where $\mathcal{I} = \{S : |S \cap I_i| \leq 1\}$.
- Thus, the welfare maximization problem is equivalent to $\max f(S) \text{ s.t. } S \in \mathcal{I}$.

GREEDYMATROIDCONSTRAINT algorithm [Nemhauser, 1978]

- Basic idea: at each iteration, the element x' with the largest marginal gain was added to S if $S \cup \{x'\}$ is independent. Note that the set returned is a base of M .

GREEDYMATROIDCONSTRAINT(N, \mathcal{I})

```
1:  $S = \phi$ ;  
2: while  $N \neq \{\}$  do  
3:    $x' = \operatorname{argmax}_{x \in N} f_S(x)$ ;  
4:   if  $S \cup \{x'\} \in \mathcal{I}$  then  
5:      $S = S \cup \{x'\}$ ;  
6:   end if  
7:    $N = N - x'$ ;  
8: end while  
9: return  $S$ ;
```


Theorem

Let S be the set returned by GREEDYMATROIDCONSTRAINT, then $f(S) \geq \frac{1}{2}f(S^*)$.

- Let's write $S^* = \{x_1^*, \dots, x_r^*\}$, where r denotes the rank of M .
- We order S as $S = \{x'_1, \dots, x'_r\}$, where $x'_i = \phi(x_i^*)$, and $\phi()$ represents the bijective basis exchange function between S^* and S . Then,

$$f(S^*) \leq f(S) + \sum_{x_i^* \in S^* \setminus S} f_S(x_i^*) \quad (29)$$

$$\leq f(S) + \sum_{x_i^* \in S^*} f_S(x_i^*) \quad (30)$$

$$\leq f(S) + \sum_{i=1}^r f_{S_{i-1}}(x_i^*) \quad (31)$$

$$\leq f(S) + \sum_{i=1}^r f_{S_{i-1}}(x'_i) \quad (32)$$

$$\leq f(S) + \sum_{i=1}^r (f(S_i) - f(S_{i-1})) \quad (33)$$

$$\leq 2f(S) \quad (34)$$


Speeding up greedy algorithm through **avoiding redundant evaluations of $f(S)$**

Reducing the number of evaluations

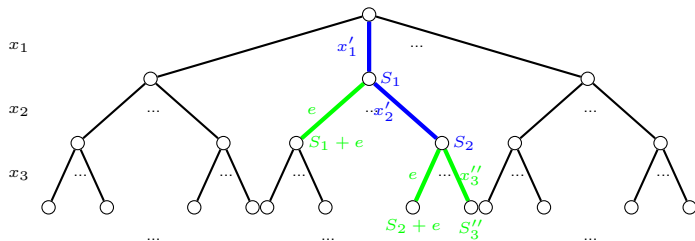
- Motivation: In some applications, the evaluation of the set function $f(S)$ might be expensive. For example, evaluating the influence of advertisement in social networks requires computationally expensive simulations.
- The standard greedy algorithm commonly requires a large number of evaluations of $f(S)$ due to **the *argmax* operation at each iteration**. For example, there are $O(kn)$ evaluations in GREEDYCARDINALITYCONSTRAINT algorithm.

GREEDYCARDINALITYCONSTRAINT(k, N)

```
1:  $S = \phi$ ;  
2: while  $|S| < k$  do  
3:    $\hat{x} = \operatorname{argmax}_{x \in N} f_S(x)$ ;  
4:    $S = S \cup \{\hat{x}\}$ ;  
5:    $N = N - \{\hat{x}\}$ ;  
6: end while  
7: return  $S$ ;
```

- Question: could we **identify redundant evaluations** and thus reduce the number of the evaluation of $f(S)$? 

Identifying redundant evaluations of $f(S)$



- Basic idea: Take the iteration at S_2 as an example. The **marginal gain** of all elements except x'_1 and x'_2 will be evaluated in the $\operatorname{argmax}_{f_{S_2}(x)}$ operation. Note that we have already obtained **the upper bound of the marginal gain** of these elements, say $f_{S_2}(e) \geq f_{S_1}(e)$. If an element e has its upper bound smaller than the calculated marginal gain of another element, then e could be neglected at this iteration safely.

ACCELERATEDGREEDY algorithm [Minoux, 1978]

ACCELERATEDGREEDY(k, N)

```
1:  $S = \phi$ ;  
2: Set  $U(x) = f(x)$  for all  $x \in N$ ;  
3: while  $|S| < k$  do  
4:   while TRUE do  
5:      $\hat{x} = \operatorname{argmax}_{x \in N} U(x)$ ;  
6:     if  $\hat{x}$  has already been selected once then  
7:       break;  
8:     end if  
9:     Calculate  $f_S(\hat{x})$  and update  $U(\hat{x}) = f_S(\hat{x})$ ;  
10:    if  $f_S(\hat{x}) > \max_{x \in N, x \neq \hat{x}} U(x)$  then  
11:      break;  
12:    end if  
13:  end while  
14:   $S = S \cup \{\hat{x}\}$ ;  
15:   $N = N - \{\hat{x}\}$ ;  
16: end while  
17: return  $S$ ;
```

- When function f is submodular, the accelerated greedy algorithm produces a greedy solution; furthermore, it produces identical solution to the corresponding standard greedy algorithm if the greedy solution is unique.
- When applied to the optimal network problem, the accelerated greedy algorithm requires on average only 2-3 calculations of f at each iteration, leading to a significant speed-up of nearly $\frac{n}{6}$.
- For example, on graphs with about 180 nodes and 500 edges, the accelerated greedy algorithm is 50-100 times faster than the standard greedy algorithm.

A tightly-related problem: MINCOST COVERAGE

- In contrast to maximizing a monotone submodular function under cardinality constraint, the MINCOST COVERAGE problem aims to find the minimum sets that achieves a given amount of objective function value.

$$\begin{array}{ll}\min & |S| \\ \text{s.t.} & f(S) \geq C\end{array}$$

- For example, the SETCOVER problem aims to find the minimum number of sets that covers the ground set.

Greedy algorithm for MINCOST COVERAGE [Wolsey, 1982]

Theorem

Consider a monotone, submodular, and integer-valued set function $f : 2^N \rightarrow \mathbb{N}$. Let S_0, S_1, \dots be the sets returned by the GREEDYCARDINALITYCONSTRAINT algorithm, and let l be the smallest index such that $f(S_l) \geq C$. Then

$$l \leq (1 + \ln \max_{x \in N} f(x)) OPT$$

where OPT denotes the optimum.