CS711008Z Algorithm Design and Analysis

Lecture 7. Basic algorithm design technique: Greedy [1]

Dongbo Bu

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

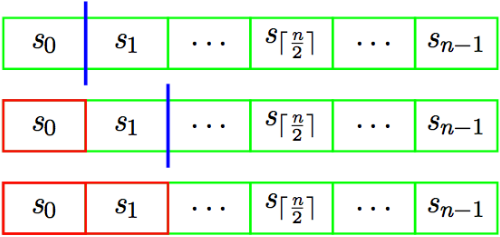
[1] The slides were made based on Chapter 15, 16 of Introduction to algorithms, Chapter 6, 4 of Algorithm design.

Outline

- Connection with dynamic programming: ShortestPath problem and IntervalScheduling problem;
- Elements of greedy technique;
- Other examples: Huffman Code, Spanning Tree;
- Theoretical foundation of greedy technique: Matroid.
- Introduction to important data structures: Binomial Heap, Fibonacci Heap, Union-Find;

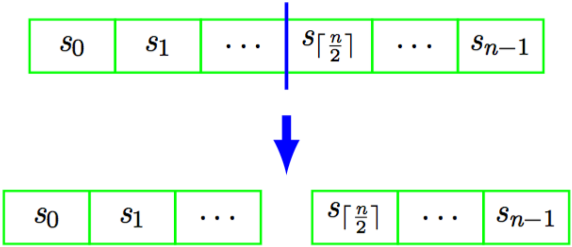
If a problem can be reduced into smaller sub-problems I

- There are two possible solving strategies:
  1. **Incremental**: to solve the original problem, it suffices to solve a smaller sub-problem; thus the problem is shrunk step-by-step. In other words, a feasible solution can be constructed step-by-step.
     For example, in Gale-Shapley algorithm, the final complete solution is constructed step by step, and a **stable, partial** matching is maintained during the construction process.



If a problem can be reduced into smaller sub-problems II

2. **divide-and-conquer**: the original problem is decomposed into several independent sub-problems; thus, a feasible solution to the original problem can be constructed by assembling the solutions to independent sub-problems.



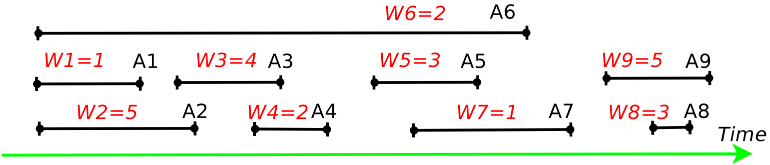
The first example: Two versions of INTERVALSCHEDULING problem

IntervalScheduling problem

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

An instance

Example:



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

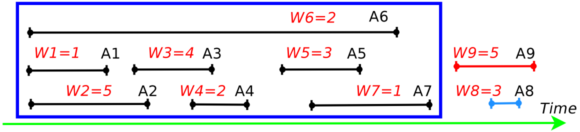
- Formulation:

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

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

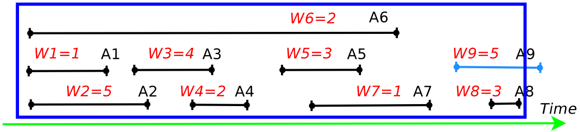
Key observation I

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



Key observation II

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



Key observation cont'd

- Summarizing the two cases, we can design the general form of subproblems as:
  selecting a collection of activities from $A_1, A_2, ..., A_i$ to maximize benefits.
- Denote the optimal solution value as $OPT(i)$.
- Optimal substructure property: ("cut-and-paste" argument)
  $$OPT(i) = \max \begin{cases} OPT(pre(i)) + W_i \\ OPT(i-1) \end{cases}$$
  Here, $pre(i)$ denotes the largest index of the activities ending before S_i.

Dynamic programming algorithm

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

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

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

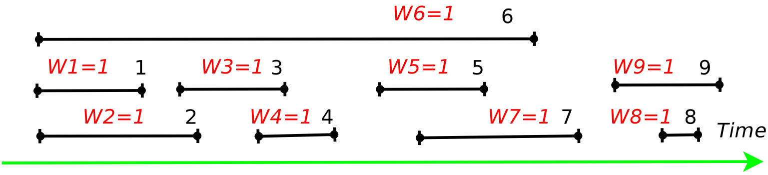
$$m = \max \begin{cases} \textsc{Recursive\_DP}(pre(i)) + W_i \\ \textsc{Recursive\_DP}(i-1) \end{cases}$$

Note:

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

IntervalScheduling problem: version 2

Let's investigate a special case



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

IntervalScheduling problem: version 2

Formulation:

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

Greedy selection property

Another key observation: Greedy selection I

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

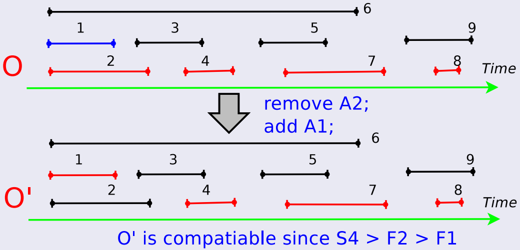
Theorem

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

Proof.

(exchange argument)

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



Simplifying the DP algorithm into a greedy algorithm

INTERVAL_SCHEDULING_GREEDY(n)

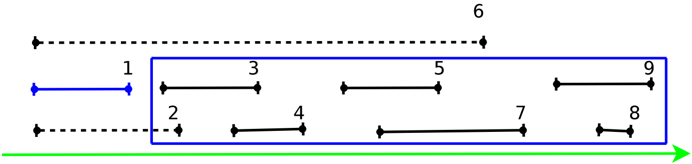
Require: All A_i have been sorted in the increasing order of F_i.

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

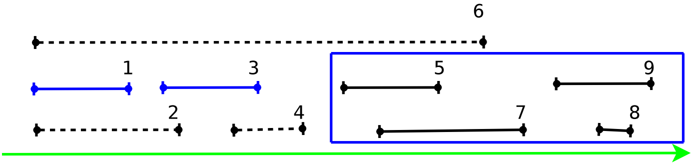
Time complexity: $O(n \log n)$ (sorting activities in the increasing order of finish time).

An example I

Step 1:

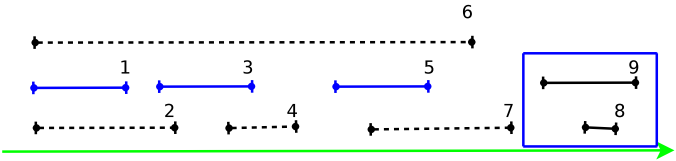


Step 2:

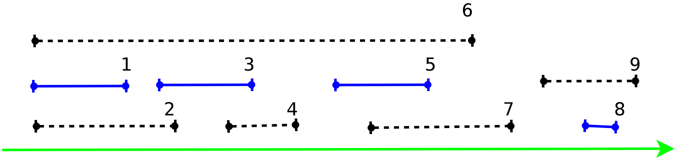


An example II

Step 3:



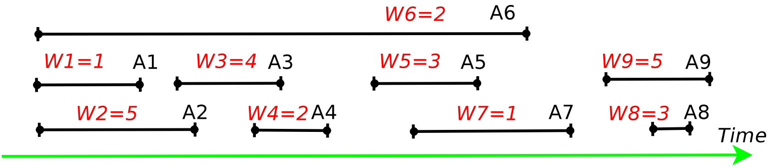
Step 4:



(see another demo)

Question: does the greedy algorithm work in general cases?

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



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

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

Elements of greedy algorithm

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

DP versus Greedy

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?

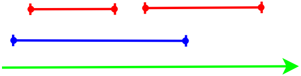
Two strategies:

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

Trying other greedy rules

Incorrect trial 1: **earlist start** rule

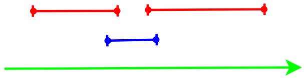
- Intuition: the earlier start time, the better.
- Incorrect. A negative example:



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

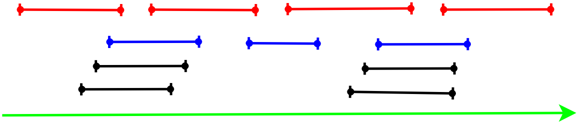
Incorrect trial 2: trying **minimal duration** rule

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



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

- 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 $G =< V, E >$. Each edge $e =< i, j >$ has a distance $d_{i,j}$. A single source node s, and a destination node t;
OUTPUT:
The shortest path from s to t.

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

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

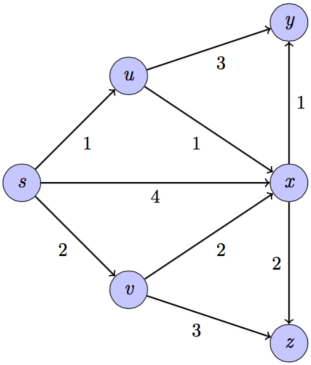
- Note: the first item $OPT(v, k-1)$ is introduced here to describe **"at most"**.
- Time complexity: $O(mn)$

Bellman-Ford algorithm 1956

BELLMAN_FORD(G, s, t)

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

An example



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

Greedy-selection property in version 2

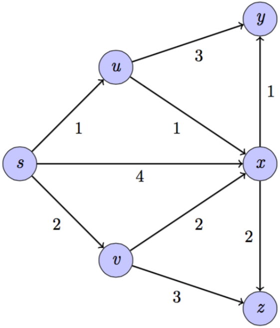
Greedy-selection property

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

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

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

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



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

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

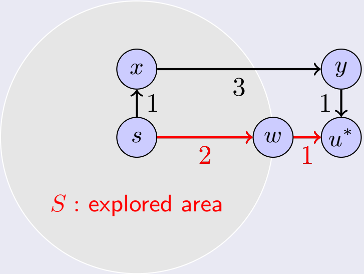
But how to calculate $OPT(v, k)$ for the **unexplored nodes** $v \notin S$?
Let's see a greedy selection rule.

Theorem

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

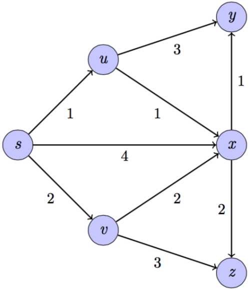
Proof.

- Suppose there is another path P' from s to u^* shorter than P.
- Without loss of generality, we denote $P' = s \rightarrow ... \rightarrow x \rightarrow y \rightarrow ... \rightarrow u^*$. 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.



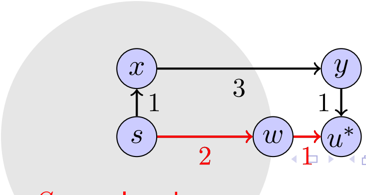
Key observations

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



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

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



Dijkstra's algorithm [1959]

DIJKSTRA(G, s)

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

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

Implementing Dijkstra algorithm using priority queue

$\textsc{Dijkstra}(G, s)$

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

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

Contributions by Edsger W. Dijkstra

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

ShortestPath: Bellman-Ford algorithm vs. Dijkstra algorithm

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

1. No negative cycle: an optimal path from s to v has at most $n-1$ edges; thus the optimal solution is $OPT(v, n-1)$. To calculate $OPT(v, n-1)$, we appeal to the following recursion:
$$OPT[v,k] = \min \begin{cases} OPT[v, k-1], \\ \min_{<u,v>\in E}\{OPT[u, k-1] + d(u, v)\} \end{cases}$$
2. No negative edge: This stronger constraint on edge weights implies greedy choice property. In particular, it is not necessary to calculate $OPT(v, i)$ for any explored node $v \in S$, and for the nearest unexplored node, its shortest distance from s is determined.

Time complexity analysis

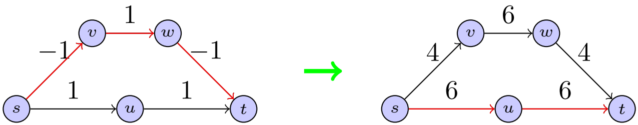
Time complexity of DIJKSTRA algorithm

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

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

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

Trial 1: increasing all edge weights by the same amount



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

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 $dist(S, v)$, the shortest distance from S to v.
- We can prove that for any node pair u and v, $d'(u, v) = d(u, v) + dist(u) - dist(v) \geq 0$.

Johnson algorithm for all pairs shortest path [1977]

$\textsc{Johnson}(G, d)$

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

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

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

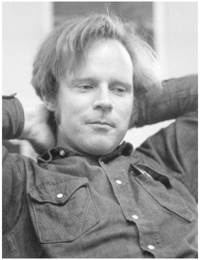


Figure 1: Robert W. Floyd, Jean Vuillenmin, Robert Tarjan

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

Huffman Code

Compressing files

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

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

Formulation

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

Requirement: prefix code I

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

Definition (Prefix coding)

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

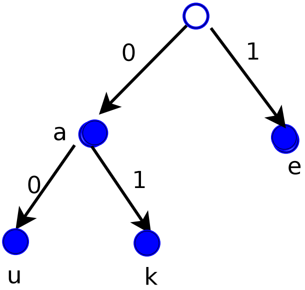
- Intuition: A prefix code can be represented as a binary tree, where a leaf represents a symbol, and the path to a leaf represents the code.
- Our objective: to design an optimal tree T to minimize expected length $E(T)$ (the size of the compressed file).

Requirement: prefix code II

Ex1:

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

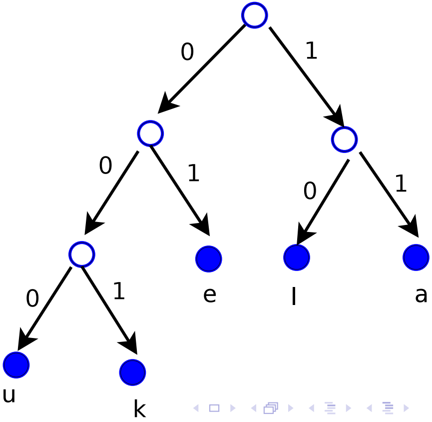
What is 0101?
aeae
k k
Ambiguity!



Ex2:

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

What is 1001000001?
l e u



Full binary tree

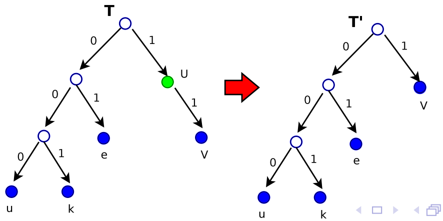
Theorem

An optimal binary tree should be a full tree.

Proof.

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

$\square$



Top-down manner: a false start

Shannon-Fano coding [1949]

Top-down method :

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

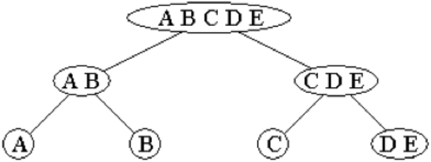
Figure 2: Claude Shannon and Robert Fano

An example: Step 1

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

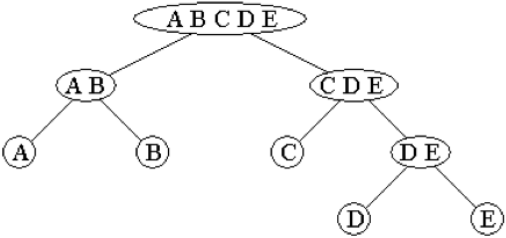
An example: Step 2

Symbol	Frequency	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

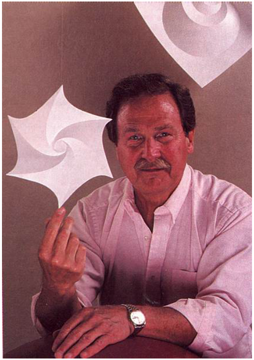
Symbol	Frequency	1. Step Sum	1. Step Kode	2. Step Sum	2. Step Kode	3. Step Sum	3. Step Kode
A	24	24	0	24	00		
B	12	36	0	12	01		
C	10	26	1	10	10		
D	8	16	1	16		16	110
E	8	8	1	8		8	111



Bottom-up manner

Bottom-up method:

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



Huffman code: bottom-up manner [1952]

Key Observations:

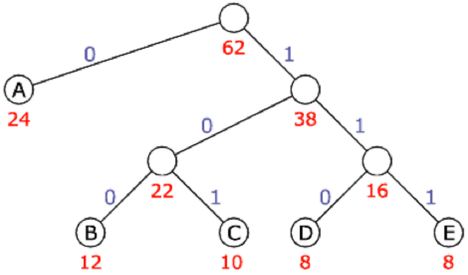
1. In an optimal tree, $depth(u) \geq depth(v)$ iff $P_u \leq P_v$. (Exchange argument)
2. There is an optimal tree, where the lowest-frequency letters Y and Z are siblings. (Why?)
   - Consider a deepest node v.
   - v's parent, denoted as u, should has another child, say w.
   - w should also be a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

Huffman code algorithm 1952

$\textsc{Huffman}(S, P)$

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

Example

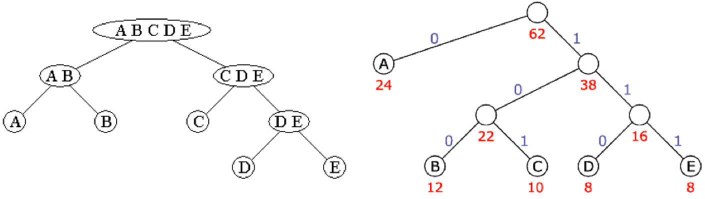


Symbol	Frequency	Code	Code Length	total Length
A	24	0	1	24
B	12	100	3	36
C	10	101	3	30
D	8	110	3	24
E	8	111	3	24

----

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

Shannon-Fano vs. Huffman



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

(linear 3 bit code)

Huffman algorithm: correctness

Lemma

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

Proof.

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

□

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

Huffman algorithm: correctness cont'd

Theorem

Huffman algorithm output an optimal code.

Proof.

(Induction)

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

□

Analysis

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.

Matroid: theoretical foundation of greedy strategy

Revisiting MAXIMAL LINEARLY INDEPENDENT SET problem

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

$$\begin{array}{rl} A_1 = & [\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array}] \\ A_2 = & [\begin{array}{ccccc} 1 & 4 & 9 & 16 & 25 \end{array}] \\ A_3 = & [\begin{array}{ccccc} 1 & 8 & 27 & 64 & 125 \end{array}] \\ A_4 = & [\begin{array}{ccccc} 1 & 16 & 81 & 256 & 625 \end{array}] \\ A_5 = & [\begin{array}{ccccc} 2 & 6 & 12 & 20 & 30 \end{array}] \end{array}$$

- Independent vector set: $\{A_1, A_2, A_3, A_4\}$

Calculating maximal number of independent vectors

INDEPENDENTSET(M)

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

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

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

Example:

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

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

A weighted version

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

$$\begin{array}{lcl}
A_1 = [\; 1 \quad 2 \quad 3 \quad 4 \quad 5 \;] & & W_1 = 9 \\
A_2 = [\; 1 \quad 4 \quad 9 \quad 16 \quad 25 \;] & & W_2 = 7 \\
A_3 = [\; 1 \quad 8 \quad 27 \quad 64 \quad 125 \;] & & W_3 = 5 \\
A_4 = [\; 1 \quad 16 \quad 81 \quad 256 \quad 625 \;] & & W_4 = 3 \\
A_5 = [\; 2 \quad 6 \quad 12 \quad 20 \quad 30 \;] & & W_5 = 1
\end{array}$$

A general greedy algorithm

$\textsc{Matroid\_Greedy}(M, W)$

```
A = {};
Sort row vectors in the decreasing order of their weights;
for all row vector v do
    if A ∪ {v} is still independent then
        A = A ∪ {v};
    end if
end for
return A;
```

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

Matroid greedy algorithm: correctness

Theorem

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

Proof.

- Assume there is an optimal subset B but $v \notin B$;
- We have
- 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.

Matroid greedy algorithm: correctness cont'd

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 $M' = \{v' \in S,$ and v, v' are independent$\}$

Proof.

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

$\square$

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

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

Origin 1 of matroid: linear independence for vectors

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

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

Example:

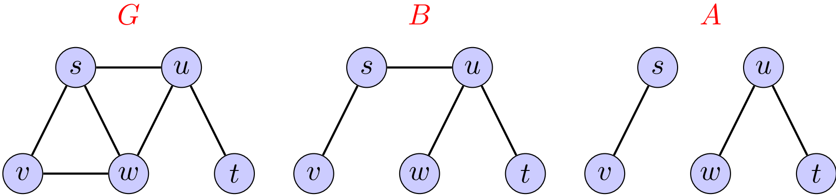
$$
\begin{array}{l}
V_1 = [\;1 \quad 2 \quad 3 \quad 4 \quad 5\;] \\
V_2 = [\;1 \quad 4 \quad 9 \quad 16 \quad 25\;] \\
V_3 = [\;1 \quad 8 \quad 27 \quad 64 \quad 125\;] \\
V_4 = [\;1 \quad 16 \quad 81 \quad 256 \quad 625\;] \\
V_5 = [\;2 \quad 6 \quad 12 \quad 20 \quad 30\;]
\end{array}
$$

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

Origin 2 of matroid: acyclic sub-graph (H. Whitney, 1932) I

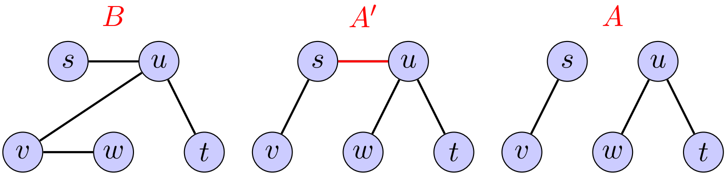
Given a graph $G =< V, E >$, let's consider the **acyclic property**.

1. **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 sub-graph (H. Whitney, 1932) II

2. **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.)

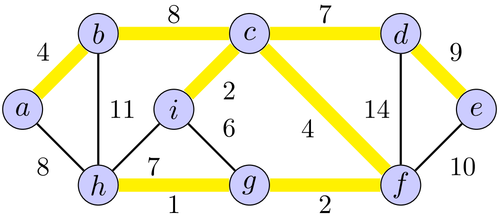
A matroid is a pair $M = (S, \mathcal{L})$ satisfying the following conditions:

1. S is a finite nonempty set (called **ground set**), and $\mathcal{L}$ is a family of INDEPENDENT SUBSETS of S.
2. **Hereditary property:** if $B \in \mathcal{L}$ and $A \subset B$, then $A \in \mathcal{L}$;
3. **Augmentation property:** if $A \in \mathcal{L}$, $B \in \mathcal{L}$, and $|A| < |B|$, then there is some element $x \in B - A$ such that $A \cup \{x\} \in \mathcal{L}$.

SPANNING TREE: an application of matroid

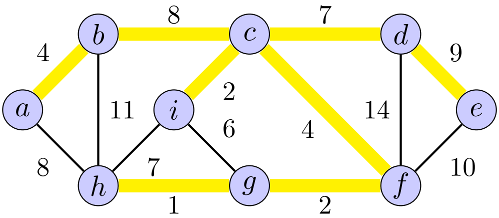
Practical problem:

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

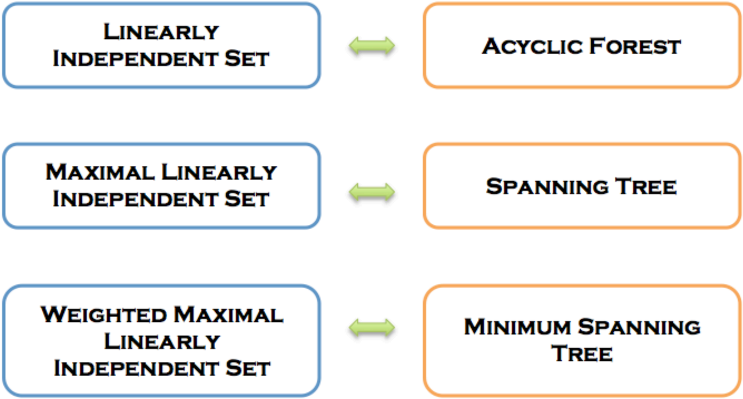


Formulation:

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



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)

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

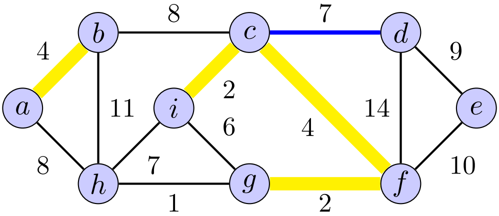


Figure 3: Safe edge

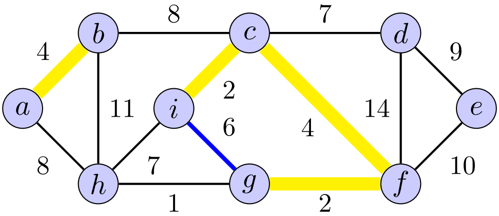


Figure 4: Unsafe edge

MINIMUM SPANNING TREE algorithms

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

Figure 5: Joseph Kruskal

Kruskal's algorithm [1956]

MST-KRUSKAL(G, W)

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

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

Time complexity

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

- Thus, the total time is $O((m+n)\alpha(n))$, where $\alpha(n)$ is a very slowly growing function.
- Since $\alpha(n) = O(\lg n)$, the total running time is $O(m \lg n)$.

Prim’s algorithm

Prim's algorithm [1957]

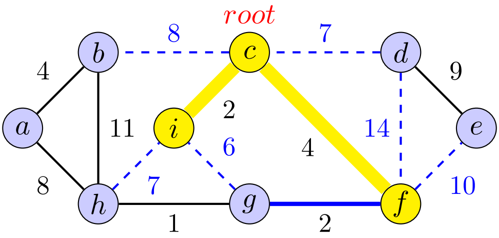
- Basic idea: the final minimum spanning tree is grown step by step. At each step, the least-weight edge connect the sub-tree to a node not in the tree is chosen.
- Note: One advantage of Prim's algorithm is that no special check to make sure that a cycle is not formed is required.

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$(G, W, root)$

for all node $v \in V$ and $v \neq root$ **do**
    $key[v] = \infty$;
    $\Pi[v] = \textsc{Null}$; $//\Pi(v)$ denotes the predecessor node of v
    $PQ.\textsc{Insert}(v)$; // n times
end for
$key[root] = 0$;
$PQ.\textsc{Insert}(root)$;
while $PQ \neq \textsc{Null}$ **do**
    $u = PQ.\textsc{ExtractMin}()$; // n times
    for all v adjacent with u **do**
        if $W(u, v) < key(v)$ **then**
            $\Pi(v) = u$;
            $PQ.\textsc{DecreaseKey}(W(u, v))$; // m times
        end if
    end for
end while

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

Time complexity of PRIM algorithm

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

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

Note:

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