贪心算法

深圳大学计算机与软件学院
卢亚辉

本章大纲

- 贪心法
- **Huffman** 编码
- 最小生成树
- 活动选择问题
- 硬币找零
- 背包问题

最优化问题

- 最优化问题：
  - 给出一个问题的实例，一组约束条件和目标函数，找到一个可行的解决方案，对于给定的实例为目标函数的最优值。
- 可行的解决方案满足问题的约束条件。
  - 解决方案中要详细说明约束条件中的限制因素。
  - 例：在背包问题中，我们要求背包中所有物件的质量总和不能超过所能承受的最大重量。

基本思想

- 从问题的某一个初始解出发，通过一系列的贪心选择——当前状态下的局部最优选择，逐步逼近给定的目标，尽可能快地求得更好的解。
- 在贪心算法（greedy method）中也采用逐步构造最优解的方法。在每个阶段，都作出一个按某个评价函数最优的决策，该评价函数最优称为贪心准则（greedy criterion）。
- 贪心算法的正确性，就是要证明按贪心准则求得的解是全局最优解。
- 贪心算法不能对所有问题都得到全局最优解。但对许多问题它能产生全局最优解，如单源最短路经问题，最小生成树问题等。

适合求解问题的特征

- 贪心选择性质：可通过局部最优（贪心）选择达到全局最优解；
  - 通常以自顶向下的方式进行，每次选择后将问题转化为规模更小的子问题；
  - 该性质是贪心法使用成功的保障，否则得到的是近优解；
- 最优子结构性质：问题的最优解包含它的子问题的最优解；
  - 并不是所有具有最优子结构性质的问题都可以采用贪心策略；
  - 往往可以利用最优子结构性质来证明贪心选择性质；

HuffMan编码

- 编码
- Huffman编码
- 应用

定长编码：ASCII

假设我们必须为文本中的每一个字符赋予一串称为**代码字**的比特位，对 n 个不同字符组成的文本进行编码，并且这些字符都来自于某张字母表。例如，我们可以使用一种**定长编码**，对每个字符赋予一个长度同为 m （$m \geq \log_2 n$）的比特串。这就是标准 7 位 ASCII 码

Ctrl	Dec	Hex	Char	Code
^@	0	00		NUL
^A	1	01		SOH
^B	2	02		STX
^C	3	03		ETX
^D	4	04		EOT
^E	5	05		ENQ
^F	6	06		ACK
^G	7	07		BEL
^H	8	08		BS
^I	9	09		HT
^J	10	0A		LF
^K	11	0B		VT
^L	12	0C		FF
^M	13	0D		CR
^N	14	0E		SO
^O	15	0F		SI
^P	16	10		DLE
^Q	17	11		DC1
^R	18	12		DC2
^S	19	13		DC3
^T	20	14		DC4
^U	21	15		NAK
^V	22	16		SYN
^W	23	17		ETB
^X	24	18		CAN
^Y	25	19		EM
^Z	26	1A		SUB
^[	27	1B		ESC
^\	28	1C		FS
^]	29	1D		GS
^^	30	1E	▲	RS
^-	31	1F	▼	US

Dec	Hex	Char
32	20	
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	⌂*

128	Ç	144	É	161	í	177	▒	193	┴	209	╤	225	ß	241	±
129	ü	145	æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
130	é	146	Æ	163	ú	179	|	195	├	211	╙	227	π	243	≤
131	â	147	ô	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
132	ä	148	ö	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
133	à	149	ò	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
134	å	150	û	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
135	ç	151	ù	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
136	ê	152	_	169	_	185	╣	201	╔	217	┘	233	⊕	249	·
137	ë	153	Ö	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
138	è	154	Ü	171	½	187	╗	203	╦	219	▉	235	δ	251	√
139	ï	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	_
140	î	157	¥	173	¡	189	╜	205	═	221	▌	237	ϕ	253	²
141	ì	158	_	174	«	190	╛	206	╬	222	▐	238	ε	254	■
142	Ä	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	
143	Å	160	á	176	▒	192	└	208	╨	224	α	240	≡		

Source: www.LookupTables.com

* ASCII code 127 has the code DEL. Under MS-DOS, this code has the same effect as ASCII 8 (BS). The DEL code can be generated by the CTRL + BKSP key.

定长编码：汉字（GB2312标准）

-
-

啊： B0A1

兵：B1F8

16区

B0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
A		啊	阿	埃	挨	哎	唉	哀	皑	癌	蔼	矮	艾	碍	爱	隘
B	鞍	氨	安	俺	按	暗	岸	胺	案	肮	昂	盎	凹	敖	熬	翱
C	袄	傲	奥	懊	坝	澳	芭	扒	叭	吧	笆	八	疤	巴	拔	跋
D	靶	把	耙	坝	霸	罢	爸	白	柏	百	摆	佰	败	拜	稗	斑
E	班	搬	扳	般	颁	板	版	扮	拌	伴	瓣	半	办	绊	邦	帮
F	梆	榜	膀	绑	棒	磅	蚌	镑	傍	谤	苞	胞	包	褒	剥	

17区

B1	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
A		薄	雹	保	堡	饱	宝	抱	报	暴	豹	鲍	爆	杯	碑	悲
B	卑	北	辈	背	贝	钡	倍	狈	备	惫	焙	被	奔	苯	本	笨
C	崩	绷	甭	泵	蹦	进	逼	鼻	比	鄙	笔	彼	碧	蓖	蔽	毕
D	毙	毖	币	庇	痹	闭	敝	弊	必	辟	壁	臂	避	陛	鞭	边
E	编	贬	扁	便	变	卞	辨	辩	辫	遍	标	彪	膘	表	鳖	憋
F	别	瘪	彬	斌	濒	滨	宾	摈	兵	冰	柄	丙	秉	饼	炳	

变长编码

的原理。为了产生平均长度最短的比特串，有一种基于古老思想的编码生成方案，它把较短的代码字分配给更常用的字符，把较长的代码字分配给较不常用的字符。（具体来说，19 世纪中叶 Samuel Morse 发明的电报码中就应用了这个思想。在 Morse 码中，像 e(·)和 a(·–)这样的常用字符被赋予了点划组成的短序列，而像 q(––·–)和 z(––··)这样的不常用字符则赋予了长序列。）

如果使用**变长编码**，则会遇到定长编码所不曾有的一种问题，它要求对不同的字符赋予长度不同的代码字。也就是说，如何能知道编码文本中用了多少位来代表第一个字符（或者更一般地来说，是第 i 个字符）呢？为了防止问题复杂化，我们只讨论**自由前缀码**（或者简称**前缀码**）。在前缀码中，所有的代码字都不是另一个字符代码字的前缀。因此，经过这样的编码，可以简单地扫描一个比特串，直到得到一组等于某个字符代码字的比特位，用该字符替换这些比特位，然后重复上述操作，直到达到比特串的末尾。

Huffman编码的思想

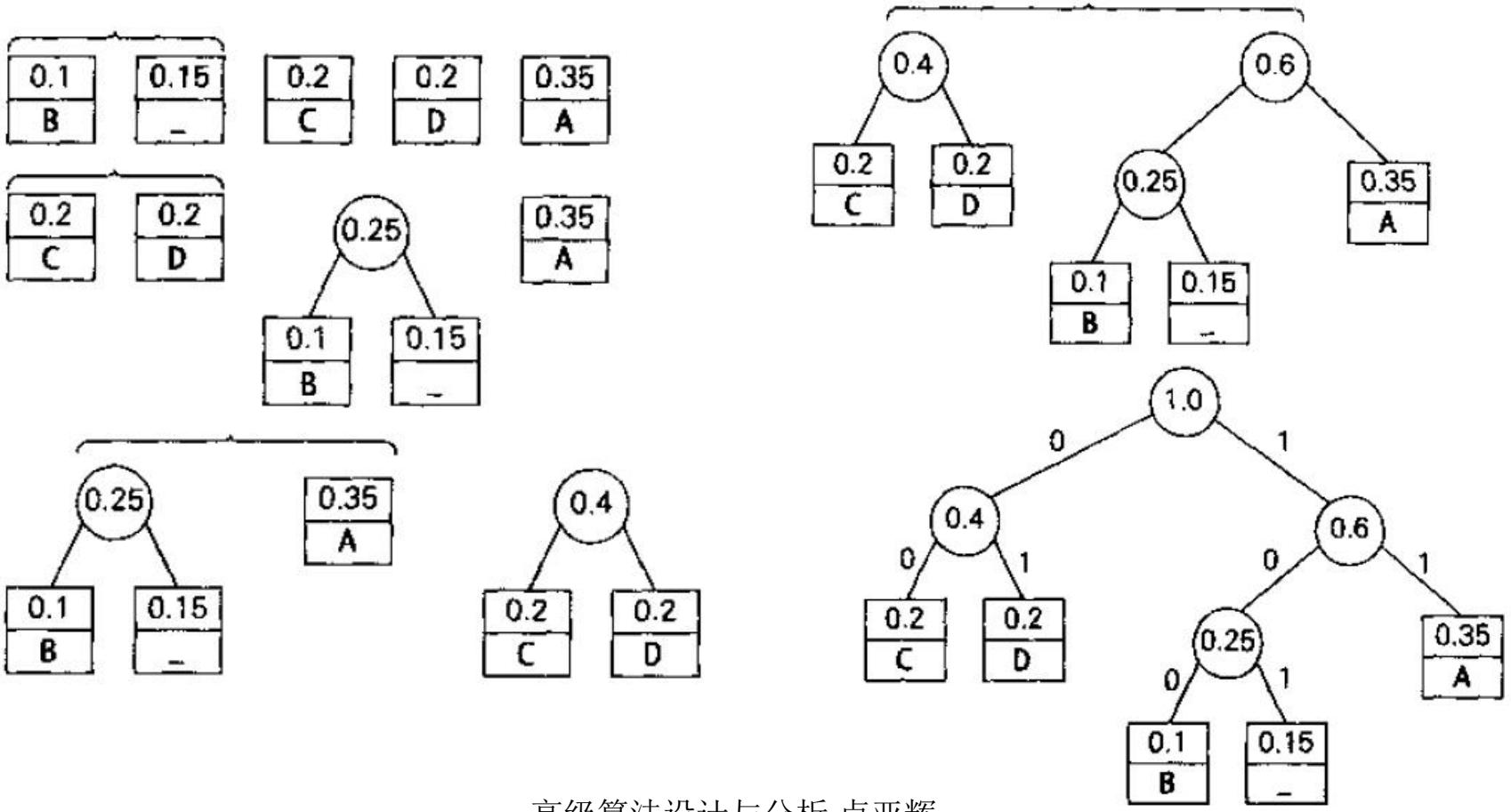
如果已知字符的出现概率，可以按照这种方式构造许多棵代表给定字母表的树，但如何构造一棵将较短比特串分配给高频字符、将较长比特串分配给低频字符的树呢？我们可以根据下面的贪婪算法来构造，当戴维·哈夫曼还是麻省理工学院的学生时，作为课后作业的一部分，他发明了这个算法[Huf52]。

哈夫曼编码是广泛地用于数据文件压缩的十分有效的编码方法。其压缩率通常在20%～90%之间。哈夫曼编码算法用字符在文件中出现的频率表来建立一个用0，1串表示各字符的最优表示方式。

给出现频率高的字符较短的编码，出现频率较低的字符以较长的编码，可以大大缩短总码长。

例　考虑一个包含 5 个字符的字符表，它们的出现概率如下：

字符	A	B	C	D	-
出现概率	0.35	0.1	0.2	0.2	0.15



DAD 被编码为 011101，而 100110111011101 解码以后就是 BAD_AD。

根据给定的出现概率和求得的代码字的长度，在这套编码中，每个字符的期望位长是

$$2\times0.35+3\times0.1+2\times0.2+2\times0.2+3\times0.15=2.25$$

如果我们用定长编码表示相同的字母表，对于每一个字符至少要用 3 个比特来表示。因此，对于这个简单的例子，哈夫曼编码实现的**压缩率**（这是压缩算法效率的一种标准度量）是[(3-2.25)/3]×100%=25%。换句话说，我们可以指望一个文本的哈夫曼编码要比其定长编码少占用 25%的存储空间（哈夫曼编码的大量实验告诉我们，这种方法的压缩率一般为 20%～80%，这依赖于所压缩文本中的字符）。■

哈夫曼编码是一种最重要的文件压缩方法。除了它的简单性和通用性外，它生成的还是一种最优编码，也就是最小长度编码（条件是，字符出现的概率是独立的，也是事先知道的）。实际上，最简单的哈夫曼压缩要求事先扫描给定的文本，来对文本中字符的出现次数计数。然后就用这些出现次数构造哈夫曼编码树，并按照前面描述的方式对文本进行编码。然而，这种方案要求我们必须把编码树的信息包含在编码文本中，以保证成功解码。通过所谓的**动态哈夫曼编码**可以克服这个缺点，这种方法在每次从源文本中读入一个字符时就更新编码树（参见[Say00]）。

哈夫曼算法

第一步： 初始化 n 个单节点的树，并为它们标上字母表中的字符。把每个字符的概率记在树的根中，用来指出树的**权重**（更一般地来说，树的权重等于树中所有叶子的概率之和）。

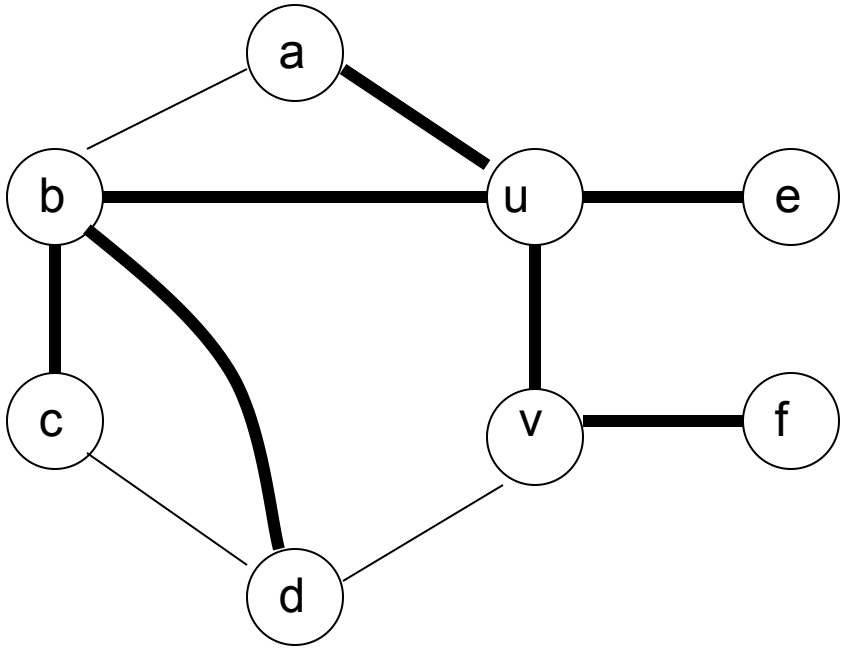
第二步： 重复下面的步骤，直到只剩一棵单独的树。找到两棵权重最小的树（次序无关紧要，但请参见习题中的第 2 题）。把它们作为新树中的左右子树，并把其权重之和作为新的权重记录在新树的根中。

必须知道，哈夫曼算法的应用并不仅限于数据压缩。假设有 n 个正数 $w_1, w_2, \ldots, w_n$，要把它们分配给一棵二叉树的 n 个叶子，每个叶子一个数。如果把加权路径长度定义为 $\sum_{i=1}^{n} l_i w_i$，其中 l_i 是从根到第 i 个叶子的简单路径的长度，如何构造一棵具有最小加权路径长度的二叉树呢？这正是哈夫曼算法所要解决的一个更一般性的问题。（对于编码问题来说，l_i 和 w_i 分别是代码字的长度和第 i 个字符的出现概率。）这种问题出现在包括决策在内的许多场合。作为一个例子，考虑从 n 个可能目标中猜测一个选定目标的游戏（比如说猜测一个 $1 \sim n$ 的整数），玩家可以问一些能够回答是或者否的问题。由于玩这个游戏的策略不同，可以建立不同的**决策树**[7]模型，就像图 9.12 给出的 $n = 4$ 的树。

10. 设计一种策略，使在下面的游戏中，期望提问的次数达到最小（[Gar94]，#52）。我们有一副纸牌，是由一张 A，两张 2，三张 3，一直到九张 9 组成的，一共包含 45 张牌。有人从这副洗过的牌中抽出一张牌，我们必须问一连串可以回答是或否的问题来确定这张牌的点数。

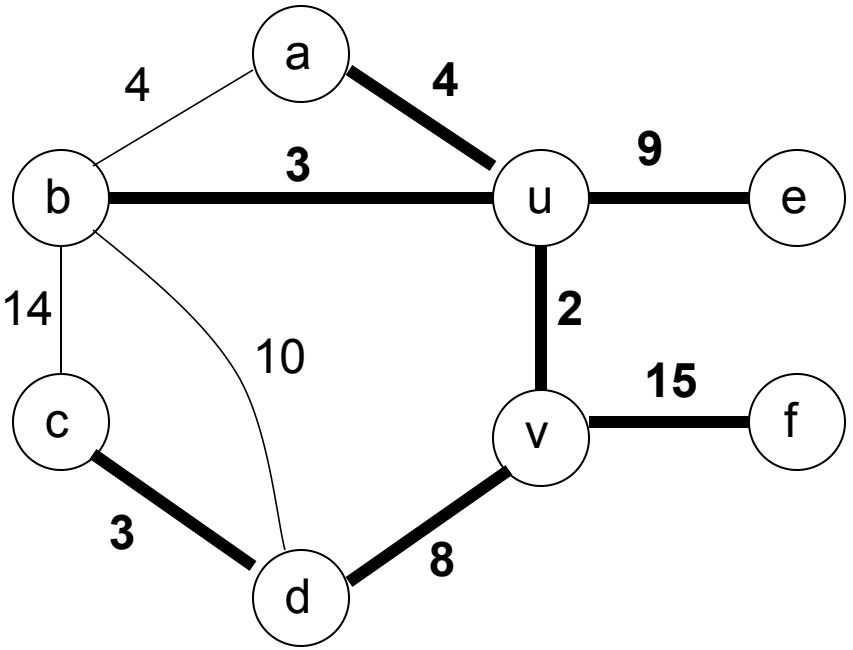
生成树

- 对于一个连通的无向图$G = (V, E)$，它的*生成树*是指*包含图G中所有节点的树*
- 思考：一个图可以有多颗生成树吗？



最小生成树(MST)

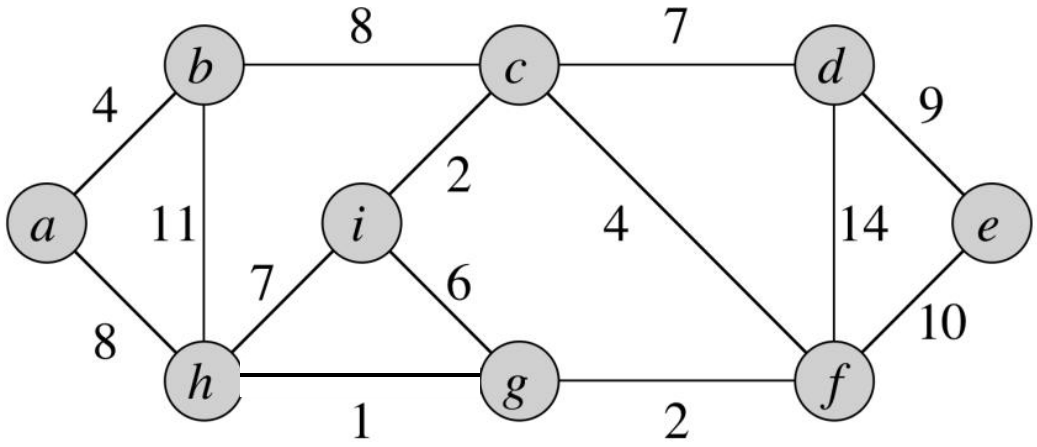
- *图的权值*指的是该图中所有边的权值之和
- 对于一个连通无向的加权*G*，它的*最小生成树*指的是权值最小的生成树a
- 思考：对于一个图，最小生成树可能存在多颗吗？



$$\text{MST } T:\ w(T) = \sum_{(u,v)\in T} w(u,v) \quad \text{最小化}$$

Kruskal算法: 举例(1)

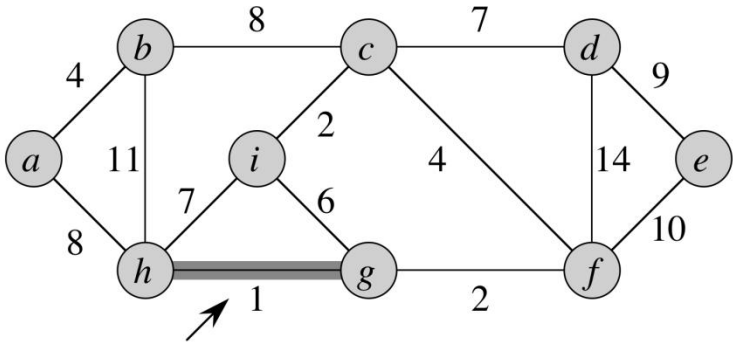
考虑以下图:



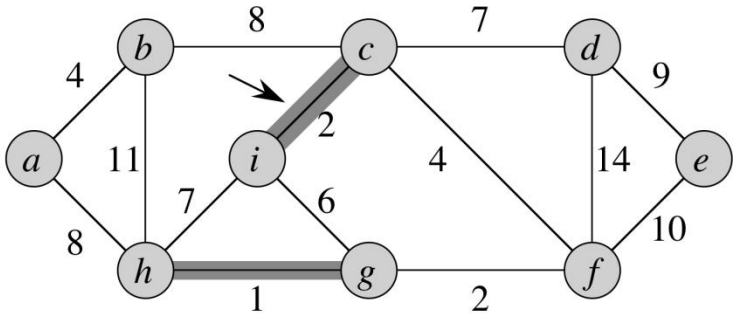
- 求一个MST
- 将边按照权值非降的方式排序:
  $(h, g), (i, c), (g, f), (a, b), (c, f), (i, g), (c, d), (h, i), (a, h),$
  $(b, c), (d, e), (f, e), (b, h), (d, f)$

Kruskal算法: 举例(2)

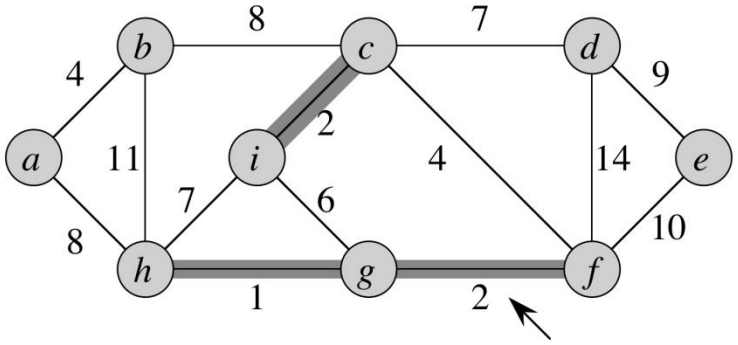
(a)



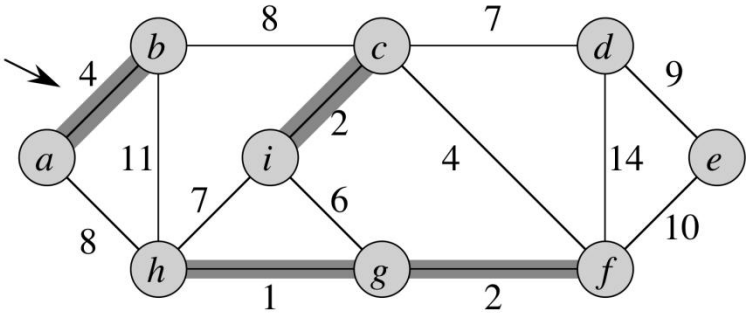
(b)



(c)

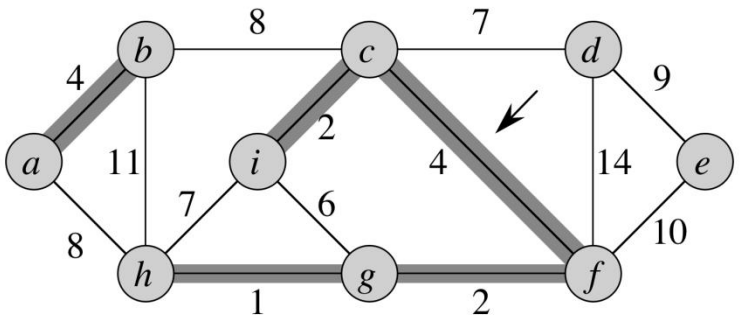


(d)

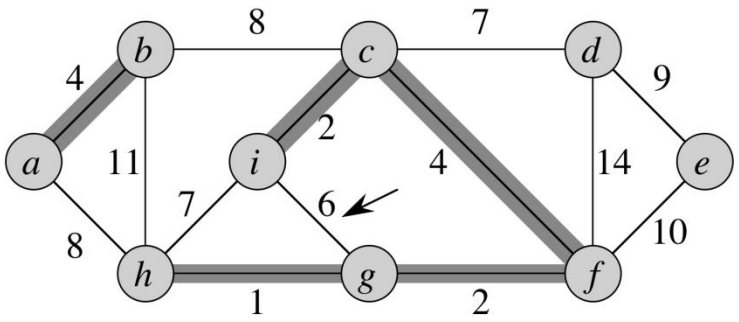


Kruskal算法: 举例(3)

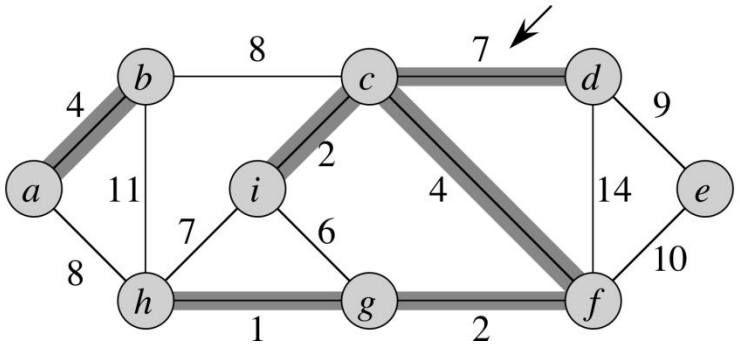
(e)



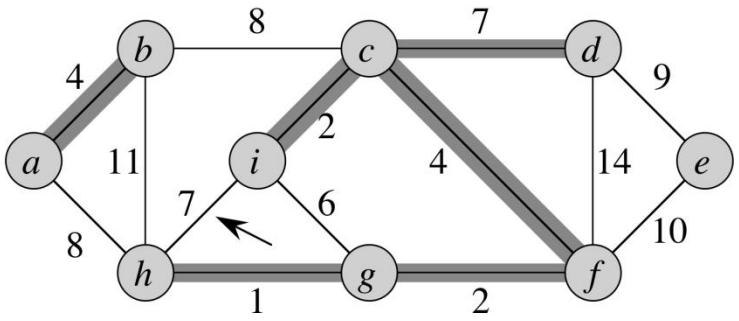
(f)



(g)

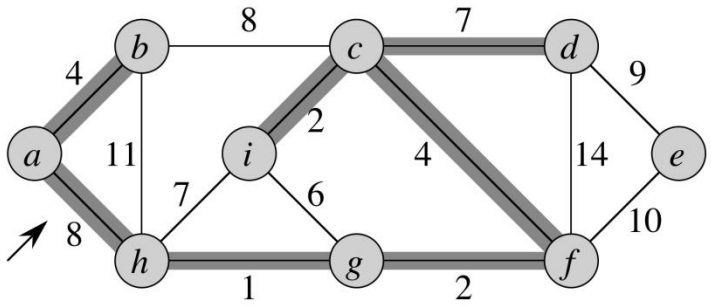


(h)

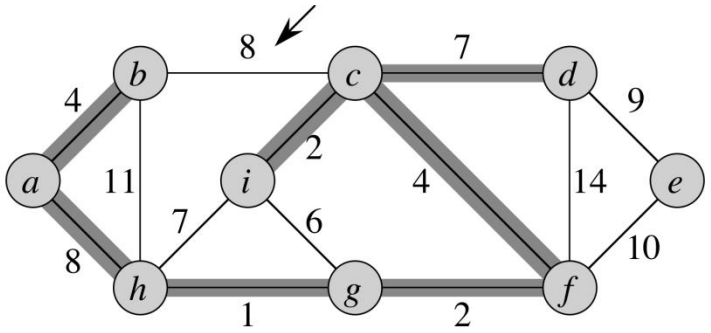


Kruskal算法: 举例(4)

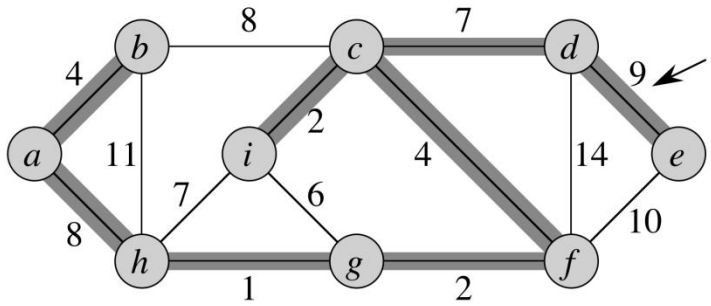
(i)



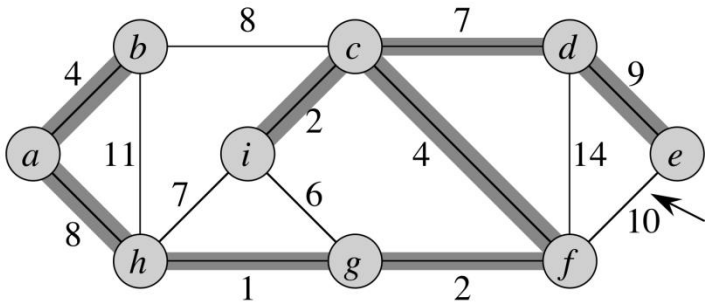
(j)



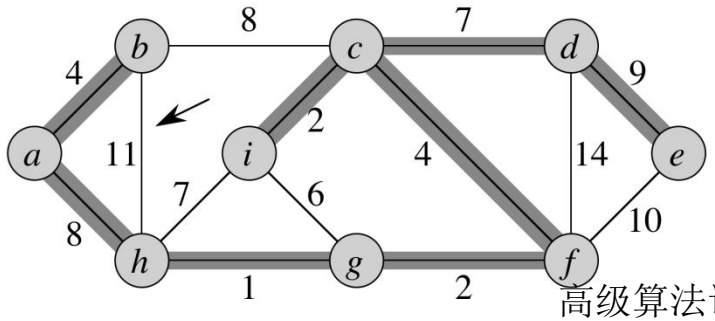
(k)



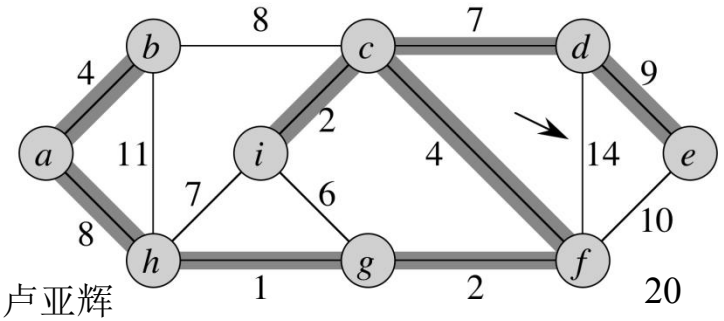
(l)



(m)



(n)



Kruskal算法分析

```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

初始化A: $O(1)$

第一个FOR循环: 有$|V|$个Make-Sets操作, 总时间代价为$O(|V|)$

边排序的代价: $O(|E| \lg |E|)$

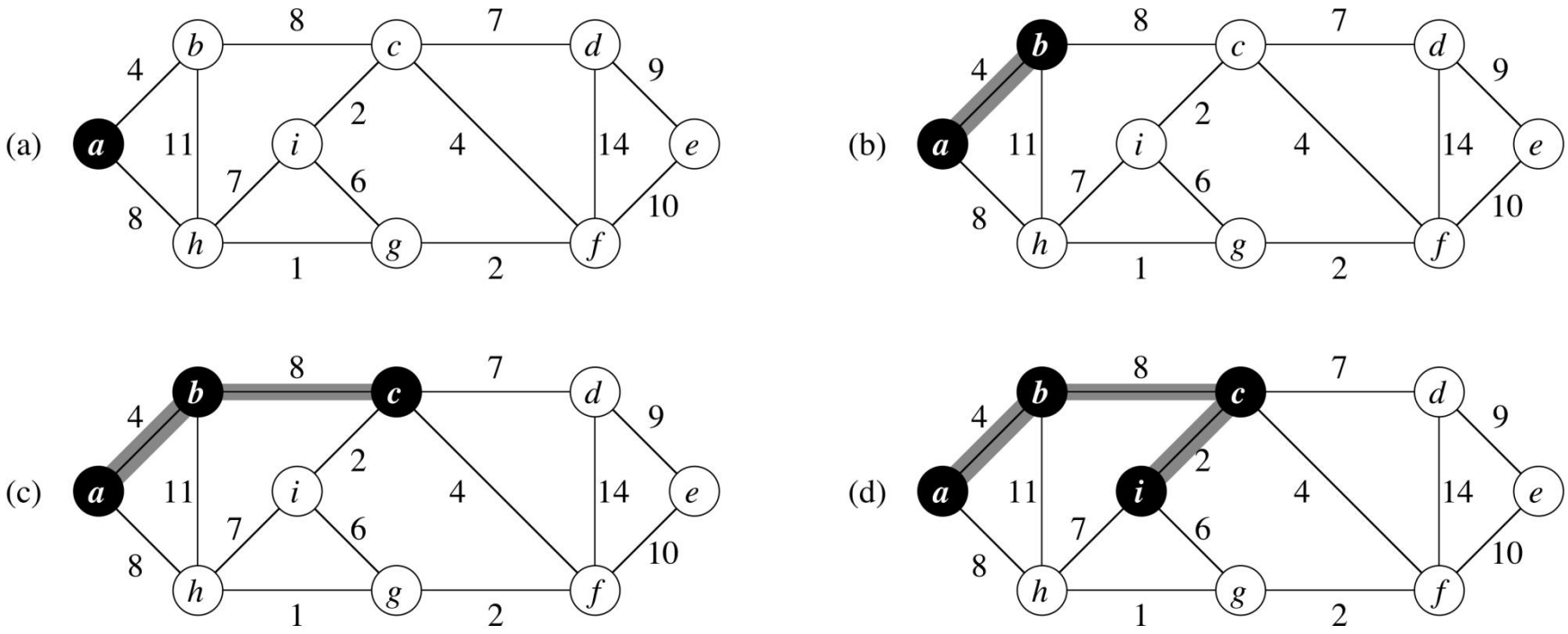
第二个FOR循环: $O(|E|)$ 个Find-Sets操作: 每个的代价为O(1)

$O(|E|)$ 个Unions操作: 每个的代价可以做到$O(\lg |V|)$，这里需要更高级的实现，参加课本第21章

总的时间复杂度: $O(|E| \lg |E|)$ 或者$O(|E| \lg |V|)$

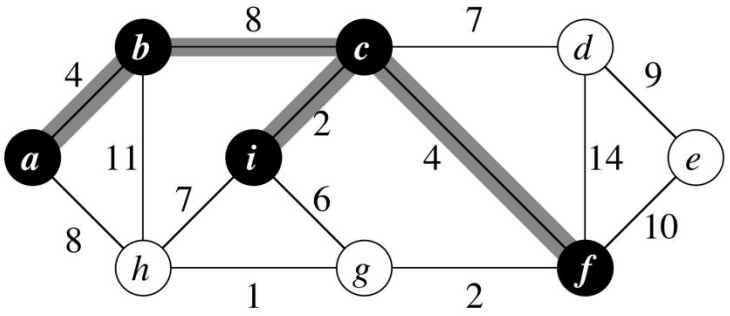
- 注意: $|E| \le |V|^2$ ➔ $\lg |E| = O(\lg |V|)$

Prim算法: 举例1 (1)

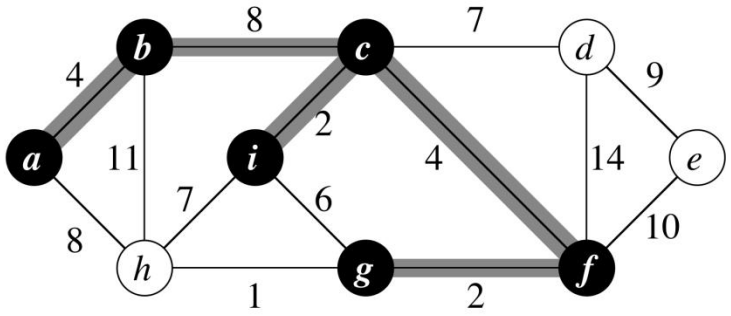


Prim算法: 举例1(2)

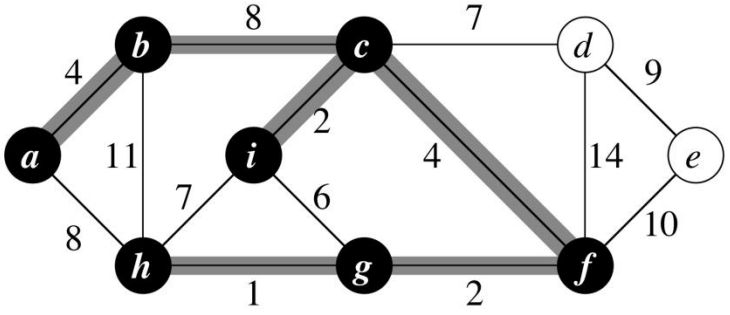
(e)



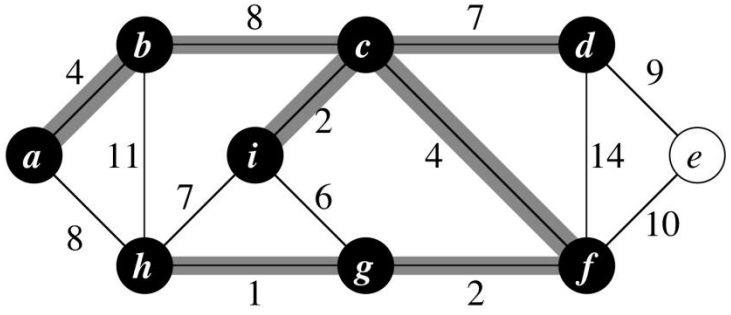
(f)



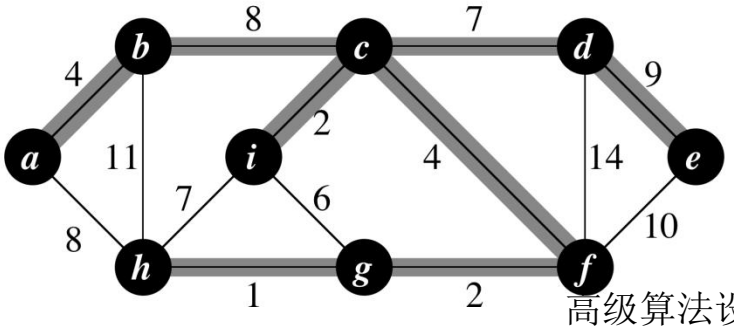
(g)



(h)



(i)



Prim算法分析

```
PRIM(G, w, r)
  Q = ∅
  for each u ∈ G.V
      u.key = ∞
      u.π = NIL
      INSERT(Q, u)
  DECREASE-KEY(Q, r, 0)        // r.key = 0
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      for each v ∈ G.Adj[u]
          if v ∈ Q and w(u, v) < v.key
              v.π = u
              DECREASE-KEY(Q, v, w(u, v))
```

- Q 为一个最小堆.
- 初始化 Q 和第一个for循环的代价: $O(|V|)$
- while 循环: $|V|$ 次迭代
  - Extract-Min: 每个操作$O(\lg |V|)$
  - for loop: 一共$O(|E|)$次
    - $v \in Q$ 可以在$O(1)$ 时间完成
    - 最后一行: $O(\lg |V|)$ 时间
    - 其它操作都可以$O(1)$时间完成
- 整个while循环: $O(|V| \lg |V|) + O(|E| \lg |V|) = O(|E| \lg |V|)$
- 总共的时间代价: $O(|E| \lg |V|)$

更多关于贪心算法

- 一个最优化问题能找到最佳的贪心算法时，他通常在其他的解决方案中有一些优点（例如动态规划和回溯）：
  - 在寻找局部最优解选择时通常更有效率。
  - 通常易于实施。

活动选择问题：一个活动实例

- 假设你在迪士尼主题乐园，你买了特殊的快速通道票，使得等待游玩项目时间最短。（两个娱乐设施之间的快速通道）
  - 有很多搭乘车次，每一车次的开始和到达时间都不同。
  - 假设我们忽略搭乘时步行和车等待你上车的时间，也就是说在两趟车次之间赶车的时间忽略不计。
- *问题*：如何让你尽可能的玩到更多的项目。
- 这就关于活动选择问题。

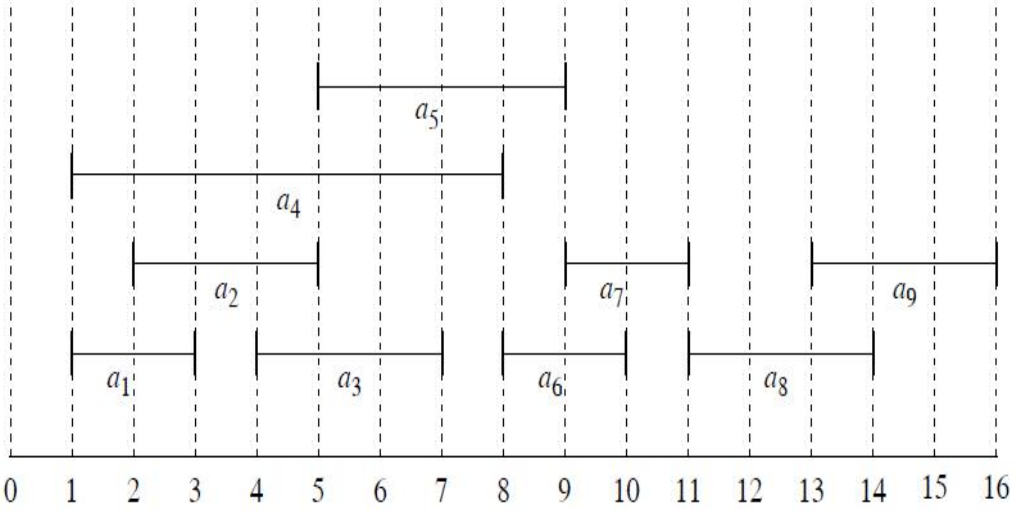
动态选择问题：定义

- *问题*: 给定一个 n个元素的活动集合$S = \{a_1, \ldots, a_n\}$，其中a_i 的时间间隔$[s_i, f_i)$, s_i表示开始时间， f_i时间表示结束时间，找到一个最大的兼容子集。
  - 活动之间的时间没有重叠表示活动之间是兼容的。
  - 不失一般性，我们假设：$f_1 \le f_2 \le \ldots \le f_n$

活动选择问题：实例

有9个活动的集合：

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



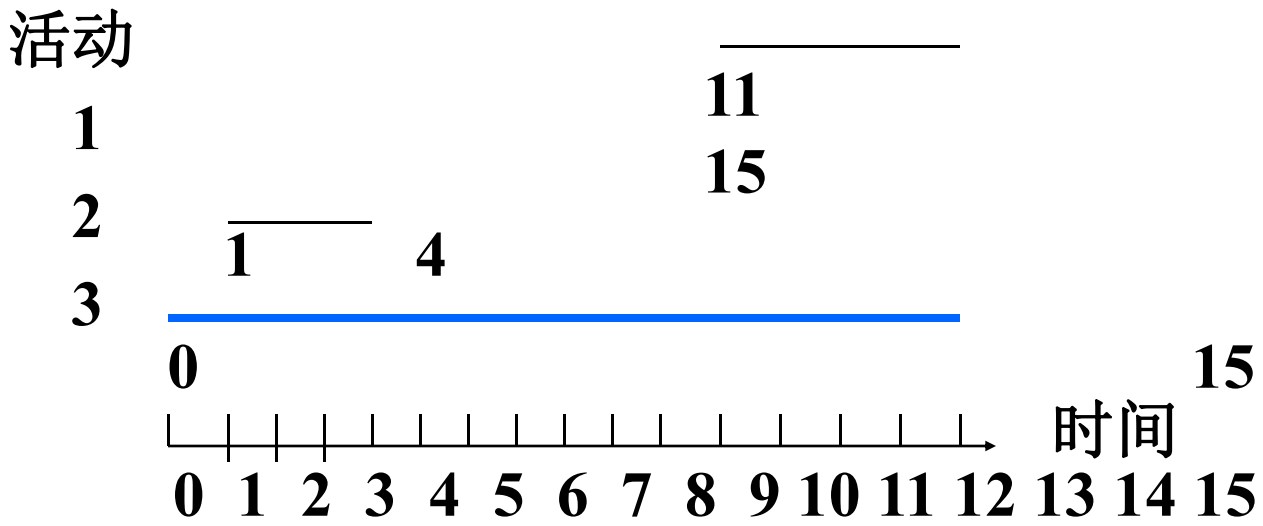
很多实施方案：$\{a_1, a_3, a_6, a_8\}$, $\{a_1, a_3, a_7, a_9\}$, $\{a_1, a_3, a_6, a_9\}$, $\{a_2, a_5, a_7, a_9\}$, $\{a_1, a_5, a_7, a_8\}$, ……

活动选择:贪心选择

- 有几个直观合理的贪心选择值得考虑:
  - *最早开始时间优先*: 选择一个最早开始时间的可兼容活动
  - *最小持续时间优先*: 选择一个最小时间间隔的可兼容活动。
  - *最早完成时间优先*: 选择一个最早结束时间的可兼容活动
- ***Question***: 哪一个会有效?

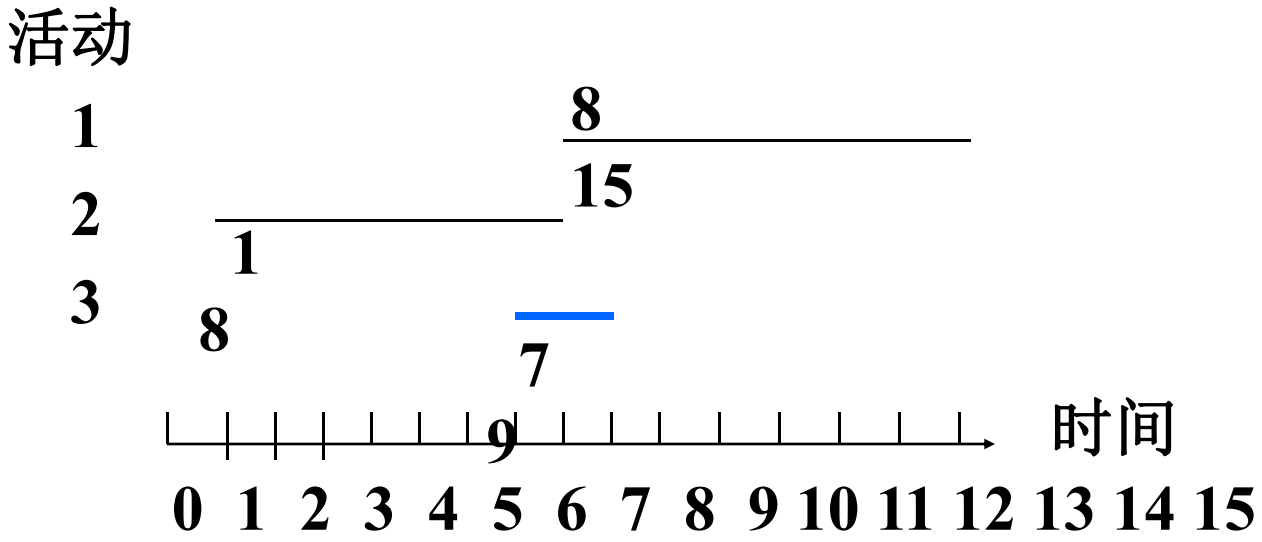
反例1

- 贪心选择准则：最早开始时间优先



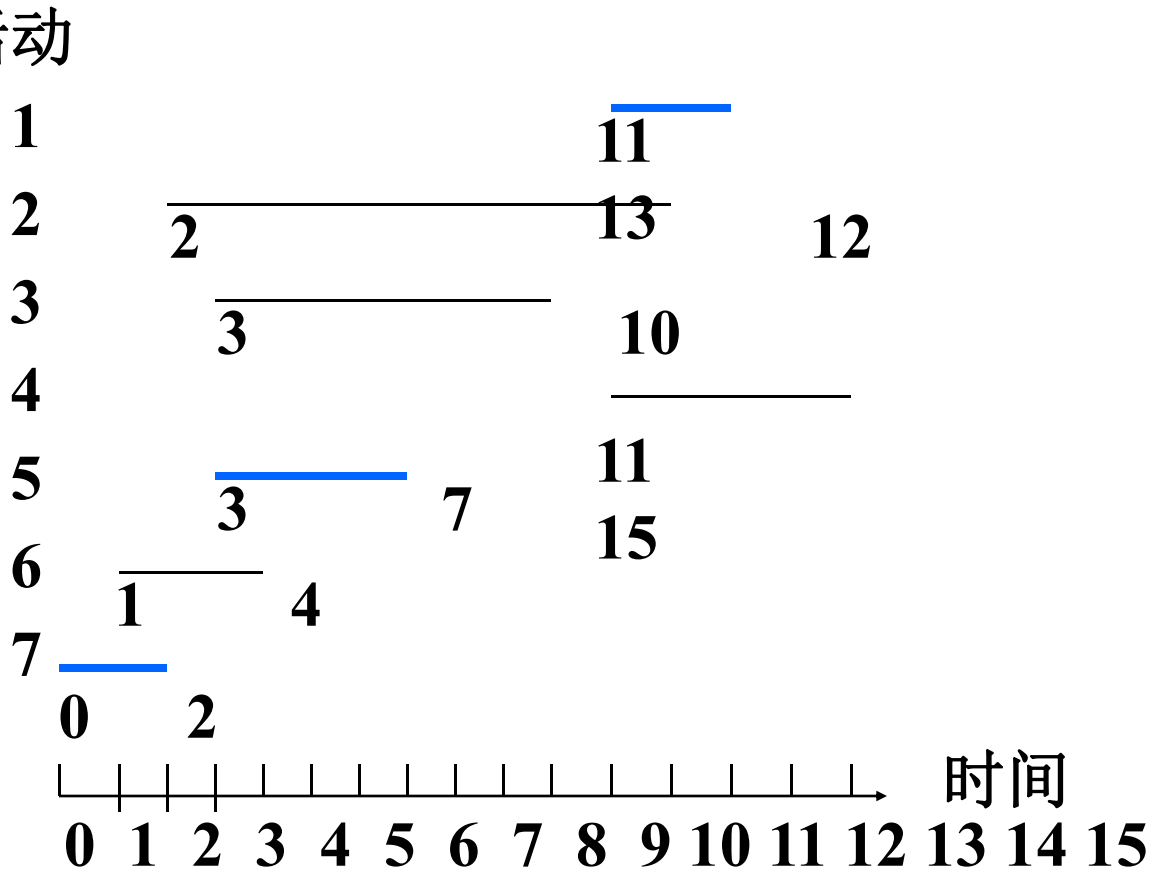
反例2

- 贪心选择准则：最小时间间隔优先



实例3

- 贪心选择准则: 最早结束时间优先

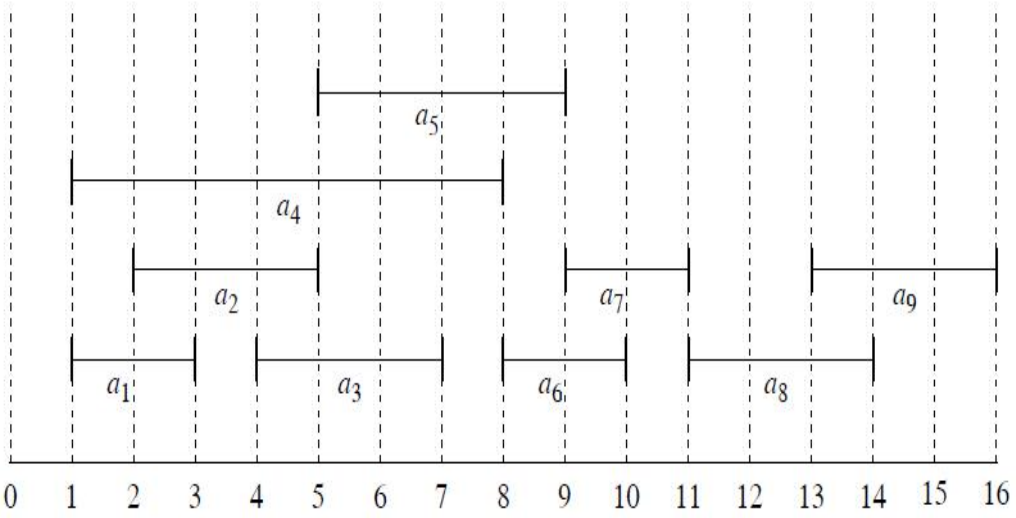


- 你能找到一个反例吗?

实例4

- 贪心选择准则：最早结束时间优先
- 此准则对这个例子也使用。

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



- 需要证明这个贪心算法的正确性。

活动选择：一个贪心算法

- 首先我们更多的以公式的形式表示这个算法（最早结束时间基准），然后证明它的正确性。
  - 我们假设: $f_1 \le f_2 \le \ldots \le f_n$

贪心活动选择(s, f) // $s = (s_1, \ldots, s_n)$， $f = (f_1, \ldots, f_n)$

$n = s.length$ // 活动的数量

$A = \{a_1\}$ // A 存储一个解决方案, 让 $a_1 = (s_1, f_1)$ 为第一个

$j = 1$ // a_j 表示上一个被添加的活动

for $i = 2$ **to** n //选择下一个活动

  if $s_i \ge f_j$ // a_i 是兼容的

    $A = A \cup \{a_i\}$

    $j = i$ // 保存上一个被添加的活动

return A

运行时间: $\Theta(n)$

- 当包括分类时间时为 $\Theta(n \log n)$

证明贪心活动选择的最优性 (1)

论点: a_1 在所有的活动中有最早结束时间, 则先选择 a_1是一个最佳的方案。

证明:

- A为最优方案。让活动 a_1 成为贪心选择 (i即为最早选择的一个). 如果 $a_1 \in A$, 证明完成。
- 如果 $a_1 \notin A$, 我们需要证明 $A^* = A - \{a\} + \{a_1\}$是另一个最优方案包括$a_1$,而a是在A中某个有最早结束时间的活动.
- 在算法中，活动根据结束时间进行分类, $f(a_1) \leq f(a)$. 如果$f(a_1) \leq s(a)$ 我们可以添加a_1 到 A, 表明A不是最优的. 如果 $s(a) < f(a_1)$, 则a_1和 a*重叠* . $f(a_1) \leq f(a)$, 如果我们移除a 并且添加a_1, 我们得到另一个最优方案A^* ，它包括a_1， A^* 是最优的因为 $|A^*| = |A|$.

证明贪心活动选择的最优性(2)

法则: 贪心活动选择是最优的, 也就是说, 对于每一个活动选择问题都能得到一个最优解决方案。

证明:

- 让算法选择活动 a_1 。
- S^* 为活动的子集且不与a_1重叠。

$$S^* = \{a_i \mid i = 2, \ldots, n \ ,\ \ s_i \geq f(a_1)\}.$$

- *让B为S**的一个最优解决方案.
- *根据S**的定义, $A^* = \{a_1\} \cup B$ 是兼容的, 而且是原始问题的一个解决方案.

证明贪心活动选择的最优性(3)

证明法则(续):

- 我们可以得出A^*是一个最优解决方案是矛盾的.
- 假设 A^* 不是原始问题的最优方案。
- 让A是一个包含a_1的最优解决方案。

  因此$|A^*| < |A|$, $|A - \{a_1\}| > |A^* - \{a_1\}| = |B|$.
- 但是 $A - \{a_1\}$ 也是 S^*这个问题的一个方案,和B是S^*一个最优方案的假设相矛盾。
- 这就表明A^* 必须是原始问题的一个最优方案.

活动选择: 最优子结构

- *假设*a_1是最佳方案A中的活动，并且有最早的完成时间，则 $A - \{a_1\}$ 是另一个最佳解决方案对于问题 $S^* = \{a_i \in S \mid s_i \geq f_1\}$.
  - 换句话说:一旦第一个活动被选择,该问题就可转变为：为活动选择找到一个最优的解决方案，这个活动在S中且与第一个选定的活动兼容。

贪心法：基本原理

- 贪心法是设计算法中另一种常用的策略，就像分治法、回溯法和动态规划算法一样。
- 经典贪心算法基本思想：
  - 遵循某些贪心准则，在当前状态下做出局部最优选择。这被称为贪心选择。
  - 我们希望能够从局部最优解中推导出全局最优解。
  - *贪心选择属性*：局部最优解导出全局最优解。
- 在设计好的贪心算法的过程中，找到一个合适的贪心选择准则是很关键的。
  - 不同的贪心准则会导致不同的结果。

贪心法：不足

- 尽管贪心算法能够得出可行的解决方案，但它得出的可能不总是最优解。
- 因此需要证明对于任何有效的输入，贪心算法总能找到最优解。
- 为了反驳贪心算法不能得出最优解这种观点，我们需要反例。

硬币找零

- 100元，给23，找77，如何找？
- 为什么要这么找？
- 找钱问题：用最少的货币数找出钱A

(1)货币数量和种类不限制情形：具有贪心选择性质，可以使用贪心法：按货币单位从高往低给付，总能得到最优解。

设n个货币 $P=\{p_1,p_2,\ldots,p_n\}$，d_i和x_i分别是p_i的货币单位和选择的数量，问题的形式描述为：

$$\min\left\{\sum_{i=1}^{n} x_i\right\}$$

$$\text{s.t.}\quad \begin{cases} \sum_{i=1}^{n} d_i x_i = A \\ x_i = 0,1 \end{cases}$$

硬币找零

- (2)货币数量和种类有限制情形：贪心法并不总能得到最优解

  — 穷举法

  解空间$X=\{(x_1,x_2,\ldots,x_n)|x_i\in\{0,1\},i=1,\ldots,n\}$ ∴$|X|=2^n$

  对任意$x\in X$, 判断x是否是可行解（即满足约束条件）的时间为$O(n)$。

  因此，穷举法的时间复杂度$T(n)=O(n2^n)$

  — 贪心法

  如问题：8个硬币(5个1分,2个1角,1个1角5分)，数出2角

  按货币单位降序数钱（贪心策略）：1×15+5×1=2角,

  显然不是最优解。

0/1 背包问题

- 假定
  - *背包能够承受的重量* $C > 0$,
  - *N个物件分别有重量为* $w_i > 0$ ， 价值分别为 $p_i > 0$ for $i = 1, \ldots n$,
- 指出一个子集 A $\{1, 2, \ldots, n\}$ 满足以下公式:

$$\max \sum_{i \in A} p_i \text{ , subject to } \sum_{i \in A} w_i \leq C$$

- 这个问题已有的解决方案
  - 回溯法
  - 动态规划
  - 贪心法

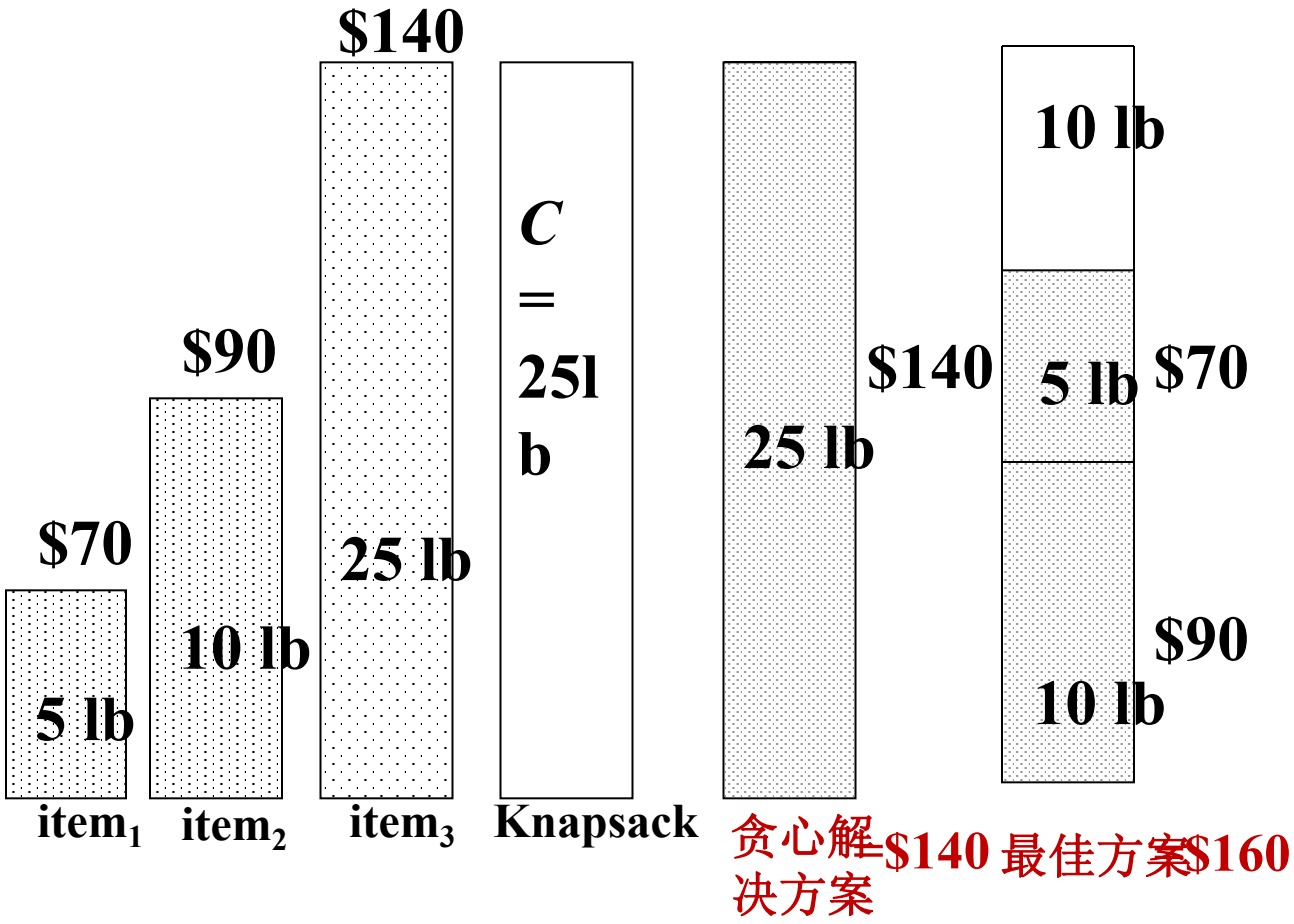
用全新的视角看待贪心解决方案

0/1 背包问题：贪心法解决方案

- 得到局部最优选择有一些可能的贪心选择准则：
  - *最大价值优先*——选择可用价值最高的物件放入背包中。
  - *最小重量优先*——选择可用重量最小的物件放入背包中。
  - *最大重量优先*——选择可用重量最大的物件放入背包中。
  - *最大性价比优先*——选择可用的、价值重量比最高的物件放入背包中。
- 不尽人意的是，以上方法没有一种能保证是最优解决方案——我们能够找到每一种方案的反例。

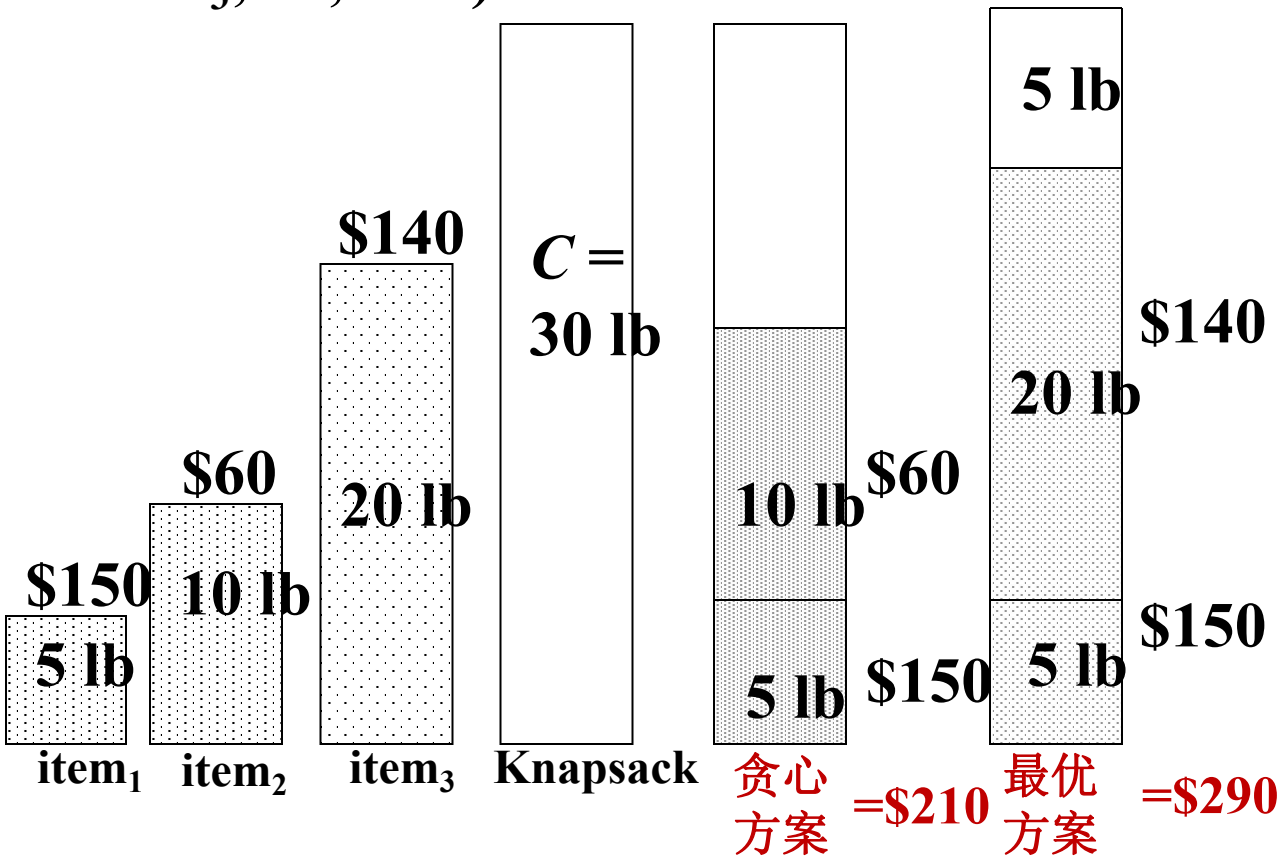
贪心法 1：选择准则：最大价值优先——反例

- 有三个物件，背包的可承受重量为 25.

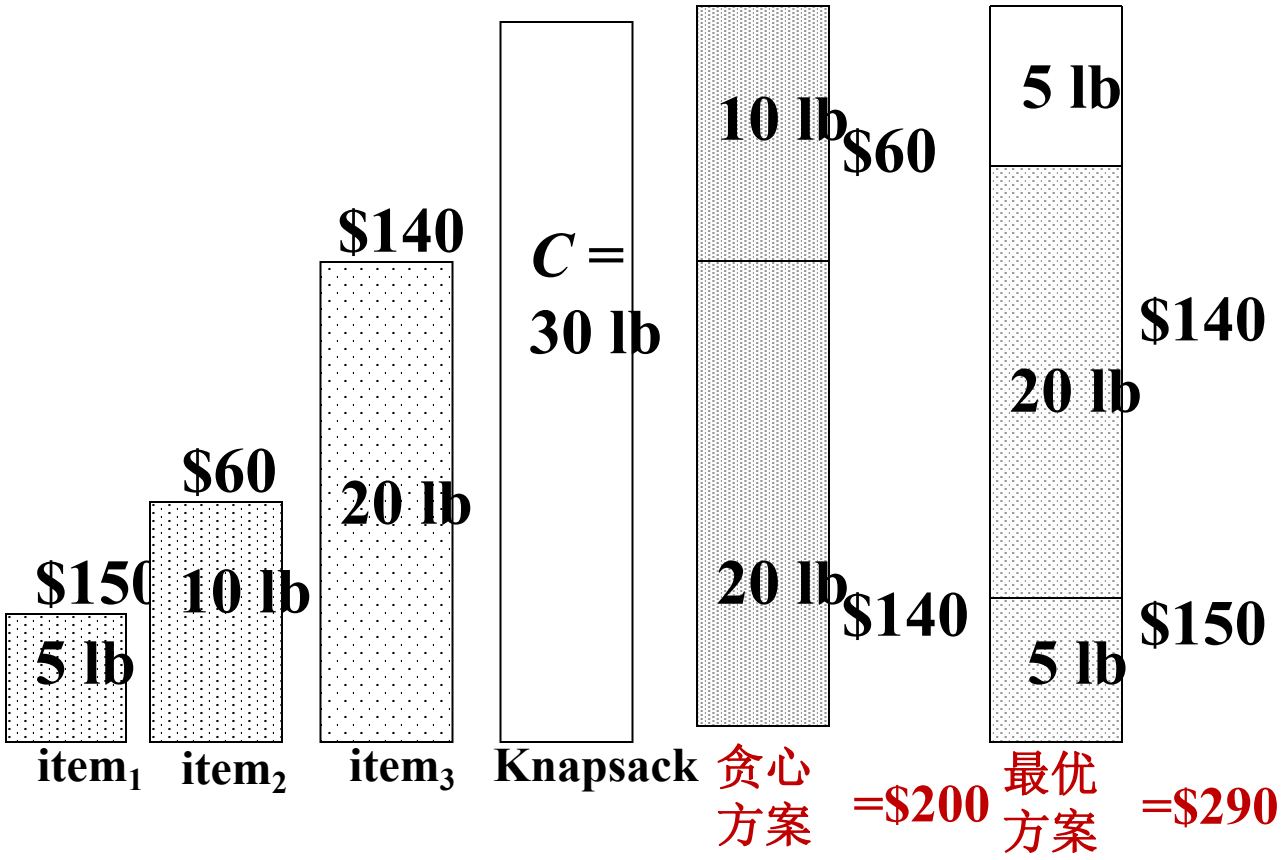


贪心法 2：选择准则：最小重量优先——反例

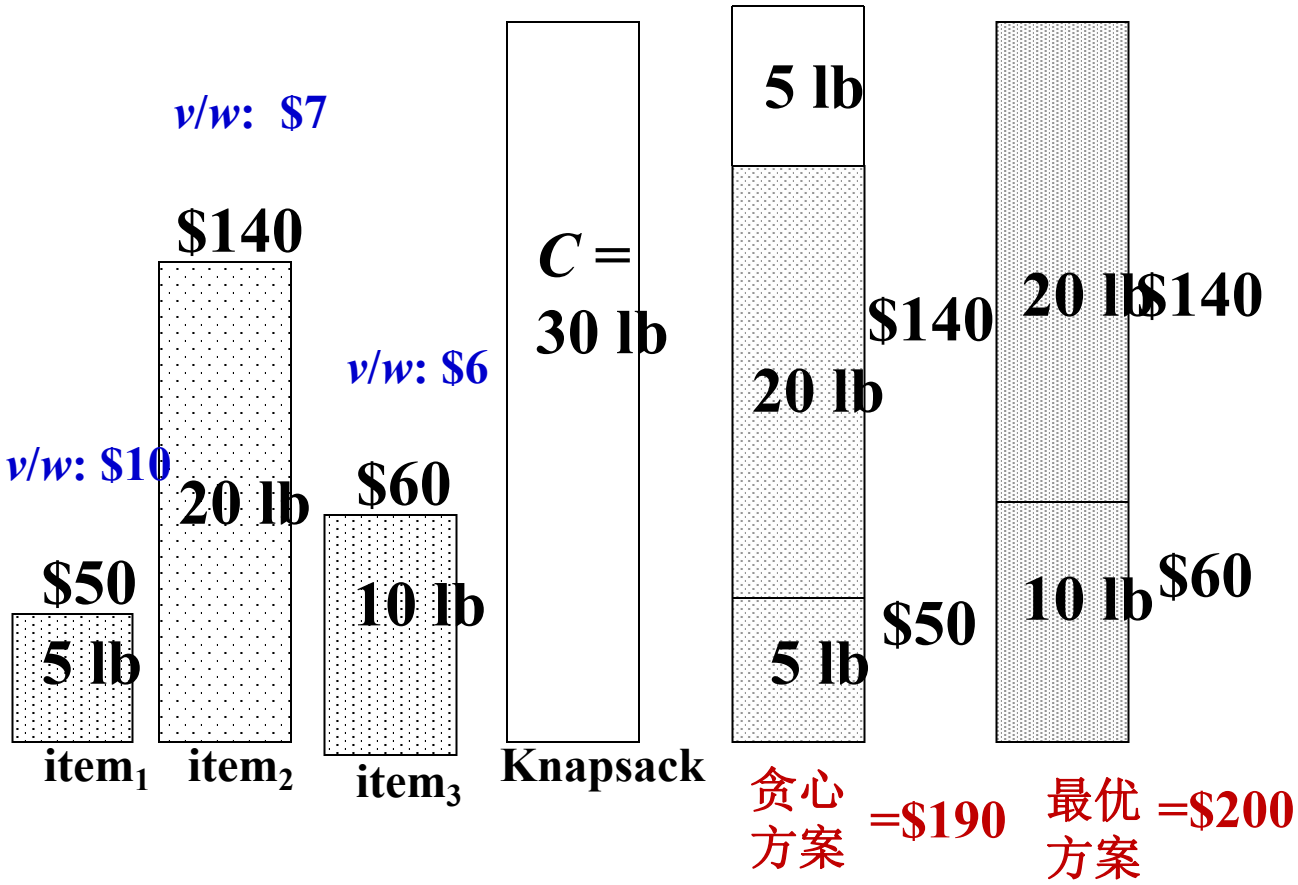
- 3 items: ($item_1$, 5, 150), ($item_2$,10, 60), ($item_3$, 20, 140)



贪心法 3：选择准则：最大重量优先——反例



贪心法 4：选择准则：最高性价比优先——反例



背包问题的贪心算法

- 对于0/1背包问题没有最好的贪心算法。
- 但是对于部分背包问题有最优的贪心算法，就是以最大价值重量比优先为基础的选择准则。
- 这种贪心算法的原理如下（更多细节详读回溯法章节）：
  - 根据价值/重量比降序排列所有物件。
  - 根据顺序依次将这些物件添加到背包中直到没有更多的物件或者下一个物件添加后会超出背包的承受范围。
  - 如果背包还是没有超出承受重量，用未选择的部分物件填满它。