《计算复杂性理论》
第10讲 回溯法

山东师范大学信息科学与工程学院
段会川
2014年12月

目录

0-1背包问题

- 定义
- 状态空间树(解空间树)
- 可行解数量(解空间大小)

0-1背包问题

编号	1	2	3	4	5
重量	2	2	6	5	4
价值	6	3	5	4	6

背包容量：10　　王秋芬，P101

解：1,1,0,0,1
重量：2+2+4=8
价值：6+3+6=15

背包问题的一个例子：应该选择哪些盒子，才能使价格尽可能地大，而保持重量小于或等于15 kg？

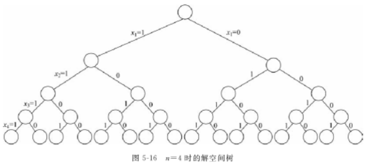


图 5-16　n=4 时的解空间树

0-1背包问题的状态空间树—子集树

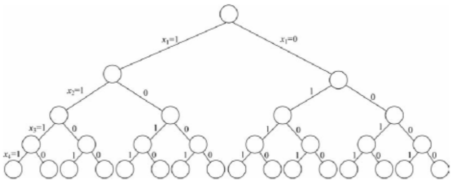


图 5-16　n=4 时的解空间树

i	1	2	3	4
w_i	5	4	6	3
v_i	10	40	30	50

$W = 10$
$X^* = (0, 1, 0, 1)$
$W^* = 4 + 3 = 7$
$V^* = 40 + 50 = 90$

http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf

---

TSP问题

- 定义
- 状态空间树(解空间树)
- 可行解数量(解空间大小)

TSP问题

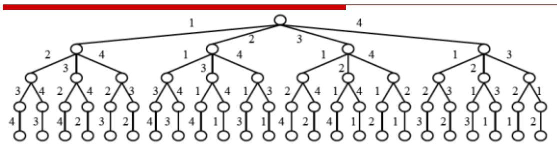
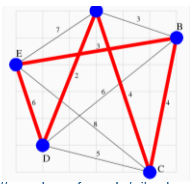


图 7.1　n=4 时货郎担问题的状态空间树



	A	B	C	D	E
A	0	3	4	2	7
B		0	4	6	3
C			0	5	8
D				0	6
E					0

http://people.sc.fsu.edu/~jburkardt/latex/genetic_2013_fsu/genetic_2013_fsu.html

目录

5.1 穷举搜索—基本思想(P123)

假设问题的初始状态、目标状态和算符的定义都是确定的，那么问题的解空间就是确定的。对问题求解就是指如何有效地搜索这个确定的解空间，从中找出问题的真正解。搜索方法有很多，如穷举搜索、深度优先搜索、广度优先搜索等。

穷举搜索是一种最基本的搜索方法，其搜索思想是：针对问题的可能解是有限种的情况，逐一检查所有可能的情况，从中找到问题真正的解。可以想象，当问题的可能解较多时，穷举搜索的效率会比较低，它是最耗时的一种解决方法。因此，这种搜索方法一般在问题求解没有更好的算法被选用时，才被使用。

5.1 穷举搜索—0-1背包示例

i	1	2	3
w_i	5	20	10
v_i	50	140	60

$W = 30$

$X^* = (0, 1, 1)$

$W^* = 20 + 10 = 30$

$V^* = 140 + 60 = 200$

http://www.radford.edu/~nokie/classes/360/greedy.html

5.1 穷举搜索—0-1背包示例

编号	1	2	3	4	5
重量	2	2	6	5	4
价值	6	3	5	4	6

背包容量：10　　王秋芬，P101

解：1,1,0,0,1
重量：2+2+4=8
价值：6+3+6=15

5.1 穷举搜索—TSP示例(P123)

【例 5-1】 给定一个有向带权图 $G=(V,E)$，权非负，如图 5-1 所示。找出顶点 1→5 的最短路径及其长度。

问题分析：该问题所有可能的路径只有三条，分别是 1→2→4→5，1→3→4→5，1→4→5，采用穷举搜索的方法逐一检查这三条路径的长度，得出的最短路径为 1→2→4→5，其长度为 3。

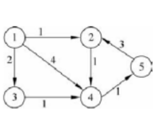
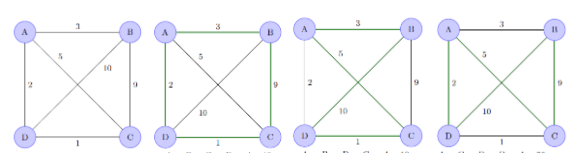
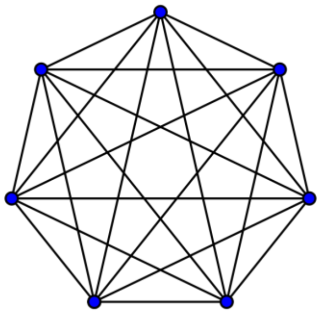


图 5-1　有向带权图



$A \to B \to C \to D \to A = 15$　　$A \to B \to D \to C \to A = 19$　　$A \to C \to B \to D \to A = 26$

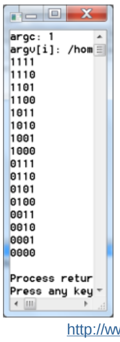
5.1 穷举搜索—TSP示例



0-1背包问题的穷举算法—C++代码

```
vector<int> KSv;
void Knapsack(int n) {
    if (n==0) {
        for (auto x:KSv)
            cout << x;
        cout << endl;
        return; }
    KSv.push_back(0);
    Knapsack(n-1);
    KSv.pop_back();
    KSv.push_back(1);
    Knapsack(n-1);
    KSv.pop_back();
}
```


0-1背包问题的穷举算法实质—深度优先搜索



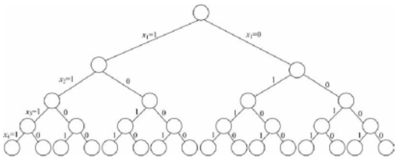


图 5-16 $n=4$ 时的解空间树

i	1	2	3	4
w_i	5	4	6	3
v_i	10	40	30	50

$W = 10$

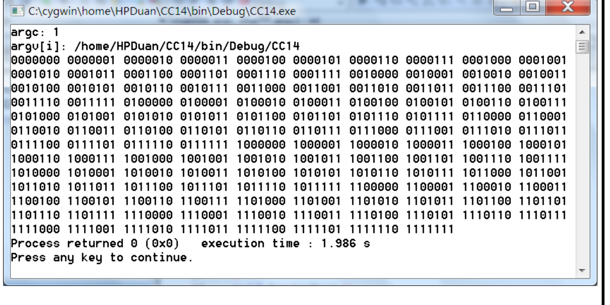
$X^* = (0, 1, 0, 1)$

$W^* = 4 + 3 = 7$

$V^* = 40 + 50 = 90$

http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf

0-1背包问题的穷举算法—运行示例

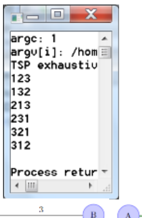


TSP问题的穷举算法—C++代码

```
vector<int> TSPv;
void TSP(int i, int n) {
    if (i==n-1) {
        for (auto x:TSPv)
            cout << x+1;
        cout << endl;
        return;
    }
    for (int j=i; j<n; ++j) {
        swap(TSPv[i],TSPv[j]);
        TSP(i+1, n);
        swap(TSPv[i],TSPv[j]);
    }
}
```

```
void TSPt(int n)
{
    for (int i=0; i<n; ++i)
        TSPv.push_back(i);
    TSP(0,n);
}
```


TSP问题的穷举算法实质—深度优先搜索



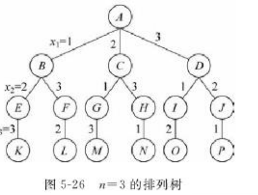
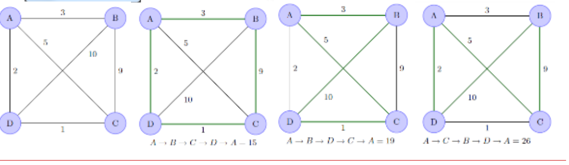
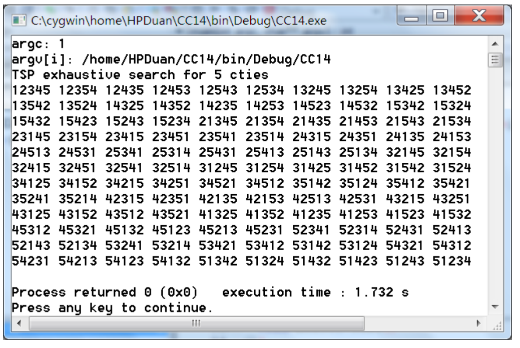


图 5-26 $n=3$ 的排列树



TSP问题的穷举算法—运行示例




```
argc: 1
argv[i]: /home/HPDuan/CC14/bin/Debug/CC14
TSP exhaustive search for 5 cties
12345 12354 12435 12453 12543 12534 13245 13254 13425 13452
13542 13524 14325 14352 14235 14253 14523 14532 15342 15324
15432 15423 15243 15234 21345 21354 21435 21453 21543 21534
23145 23154 23415 23451 23541 23514 24315 24351 24135 24153
24513 24531 25341 25314 25431 25413 25143 25134 32145 32154
32415 32451 32541 32514 31245 31254 31425 31452 31542 31524
34125 34152 34215 34251 34521 34512 35142 35124 35412 35421
35241 35214 42315 42351 42135 42153 42513 42531 43215 43251
43125 43152 43512 43521 41325 41352 41235 41253 41523 41532
45312 45321 45132 45123 45213 45231 52341 52314 52431 52413
52143 52134 53241 53214 53421 53412 53142 53124 54321 54312
54231 54213 54123 54132 51342 51324 51432 51423 51243 51234

Process returned 0 (0x0)   execution time : 1.732 s
Press any key to continue.
```


目录

- 两个基本问题
- 穷举搜索
- 深度优先搜索
- 回溯法
- 作业

- 基本思想
- 伪代码
- 一般示例
- 0-1背包问题示例
- TSP问题示例

5.2 深度优先搜索—基本思想(P124)

给定图 $G=(V,E)$。深度优先搜索的思想为：初始时，所有顶点均未被访问过，任选一个顶点 v 作为源点。该方法先访问源点 v，并将其标记为已访问过；然后从 v 出发，选择 v 的下一个邻接点(子结点) w，如果 w 已访问过，则选择 v 的另外一个邻接点；如果 w 未被访问过，则标记 w 为已访问过，并以 w 为新的出发点，继续进行深度优先搜索；如果 w 及其子结点均已搜索完毕，则返回到 v，再选择它的另外一个未曾访问过的邻接点继续搜索，直到图中所有和源点有路径相通的顶点均已访问过为止；若此时图 G 中仍然存在未被访问过的顶点，则另选一个尚未访问过的顶点作为新的源点重复上述过程，直到图中所有顶点均被访问过为止。

5.2 深度优先搜索—伪代码

- 输入：图G(V,E)，起始顶点s
- 输出：访问次序

```
//初始化
for each v in V
    visited[v] = false
prev_i = 0
post_i = 0
DFS(G,s)
```

```
DFS(G,v)
visited[v] = true
prev[prev_i++] = v
for each (v,u) in E
    if not visited[u]
        DFS(G,u)
post[post_i++] = v
```


5.2 深度优先搜索—示例(P124-5, 例5-3)

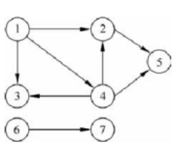


图 5-2 有向图

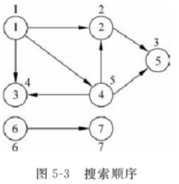


图 5-3 搜索顺序

5.2 深度优先搜索—示例(P125, 例5-4)

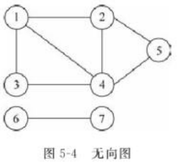


图 5-4 无向图

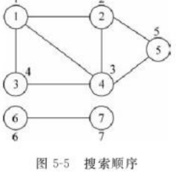
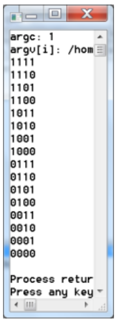


图 5-5 搜索顺序

深度优先搜索—0-1背包示例



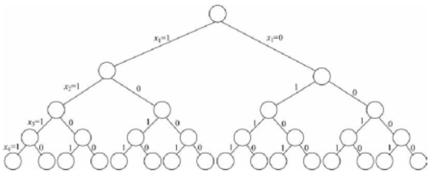
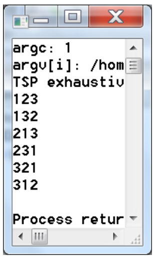


图 5-16 $n=4$ 时的解空间树

深度优先搜索—TSP示例



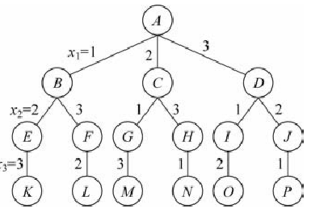


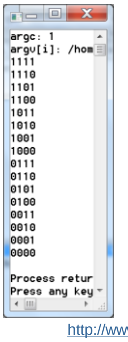
图 5-26 $n=3$ 的排列树

目录

- ☐ 两个基本问题
- ☐ 穷举搜索
- ☐ 深度优先搜索
- ☐ 回溯法
- ☐ 作业

- ☐ 穷举(深度优先搜索)的可能改进
- ☐ 基本思想
- ☐ 求解步骤
- ☐ 基本算法
- ☐ 0-1背包问题的回溯示例
- ☐ 0-1背包问题的回溯算法
- ☐ 0-1背包问题的回溯复杂度
- ☐ TSP问题的回溯示例
- ☐ TSP问题的回溯算法
- ☐ TSP问题的回溯复杂度

0-1背包问题穷举法(深度优先搜索)的可能改进



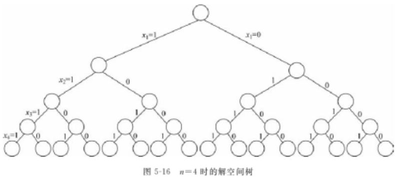


图 5-16 $n=4$ 时的解空间树

i	1	2	3	4
w_i	5	4	6	3
v_i	10	40	30	50

$W = 10$

$X^* = (0, 1, 0, 1)$

$W^* = 4 + 3 = 7$

$V^* = 40 + 50 = 90$

http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf

TSP问题穷举法(深度优先搜索)的可能改进

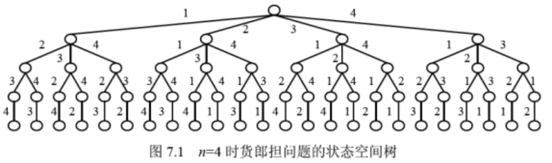
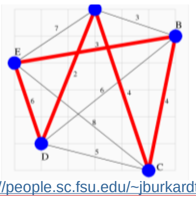


图 7.1 n=4 时货郎担问题的状态空间树



	A	B	C	D	E
A	0	3	4	2	7
B		0	4	6	3
C			0	5	8
D				0	6
E					0

http://people.sc.fsu.edu/~jburkardt/latex/genetic_2013_fsu/genetic_2013_fsu.html

5.3 回溯法—基本思想(P126)

回溯法是一种搜索方法。用回溯法解决问题时，首先应明确搜索范围，即问题所有可能解组成的范围。这个范围越小越好，且至少包含问题的一个(最优)解。为了定义搜索范围，需要明确以下几个方面：

(1) 问题解的形式：回溯法希望问题的解能够表示成一个 n 元组($x_1, x_2, \cdots, x_n$)的形式。

(2) 显约束：对分量 x_i($i=1,2,\cdots,n$)的取值范围限定。

(3) 隐约束：为满足问题的解而对不同分量之间施加的约束。

(4) 解空间：对于问题的一个实例，解向量满足显约束的所有 n 元组构成了该实例的一个解空间。

注意：同一个问题的显约束可能有多种，相应解空间的大小就会不同，通常情况下，解空间越小，算法的搜索效率越高。

5.3 回溯法—基本算法(P131-2)

```
void Backtrack(int t)
if t>n then
  output(x)
else
  for i=s(n,t) to e(n,t)
    x[t] = d[i] //d为分支上的数据
    if constraint(t) and bound(t)
      Backtrack(t+1)
```


5.3 回溯法—基本算法

Algorithm Backtrack(x):
  Input: A problem instance x for a hard problem
  Output: A solution for x or "no solution" if none exists
  $F \leftarrow \{(x,\emptyset)\}$.   {$F$ is the "frontier" set of subproblem configurations}
  while $F \neq \emptyset$ **do**
    select from F the most "promising" configuration (x,y)
    expand (x,y) by making a small set of additional choices
    let $(x_1,y_1),(x_2,y_2),\ldots,(x_k,y_k)$ be the set of new configurations.
    for each new configuration (x_i,y_i) **do**
      perform a simple consistency check on (x_i,y_i)
      if the check returns "solution found" **then**
        return the solution derived from (x_i,y_i)
      if the check returns "dead end" **then**
        discard the configuration (x_i,y_i) {Backtrack}
      else
        $F \leftarrow F \cup \{(x_i,y_i)\}$ {(x_i,y_i) starts a promising search path}
  return "no solution"

http://ww3.algorithmdesign.net/
Algorithm Design -- Foundations, Analysis, and Internet Examples
Michael T. Goodrich and Roberto Tamassia
Ch13. NP-Completeness, P628

5.3 回溯法—求解步骤

- 问题描述
- 问题分析
- 算法描述
- 步骤1：定义问题的解空间
- 步骤2：确定解空间的组织机构
- 步骤3：搜索解空间
  - 步骤3-1：设置约束条件
  - 步骤3-2：设置限界条件
  - 步骤3-3：执行搜索

5.3 回溯法—0-1背包问题(P135)

【例 5-7】 0-1 背包问题。

（1）问题描述：给定 n 种物品和一背包。物品 i 的重量是 w_i，其价值为 v_i，背包的容量为 W。一种物品要么全部装入背包，要么全部不装入背包，不允许部分装入。装入背包的物品的总重量不超过背包的容量。问应如何选择装入背包的物品，使得装入背包中的物品总价值最大？

$$\max_{x_1,x_2,\cdots,x_n} V(x_1,x_2,\cdots,x_n)=\sum_{i=1}^{n}x_i v_i,$$

$$s.t.\quad \sum_{i=1}^{n}x_i w_i \le W,$$

$$x_i \in \{0,1\}, i=1,2,\cdots,n.$$

5.3 回溯法—0-1背包问题(P135)

（2）问题分析：根据问题描述可知，0-1 背包问题要求找出 n 种物品集合$\{1,2,3,\cdots,n\}$中的一部分物品，将这部分物品装入背包。装进去的物品总重量不超过背包的容量且价值之和最大。即：找到 n 种物品集合$\{1,2,3,\cdots,n\}$的一个子集，这个子集中的物品总重量不超过背包的容量，且总价值是集合$\{1,2,3,\cdots,n\}$的所有不超过背包容量的子集中物品总价值最大的。

按照回溯法的算法框架，首先需要定义问题的解空间，然后确定解空间的组织结构，最后进行搜索。搜索前要解决两个关键问题，一是确定问题是否需要约束条件（用于判断是否有可能产生可行解），如果需要，如何设置？二是确定问题是否需要限界条件（用于判断是否有可能产生最优解），如果需要，如何设置？

5.3 回溯法—0-1背包问题求解步骤(P135)

步骤 1：定义问题的解空间。

0-1 背包问题是要将物品装入背包，并且物品有且只有两种状态。第 $i(i=1,2,\cdots,n)$ 种物品是装入背包能够达到目标要求，还是不装入背包能够达到目标要求呢？很显然，目前还不确定。因此，可以用变量 x_i 表示第 i 种物品是否被装入背包的行为，如果用“0”表示不被装入背包，用“1”表示装入背包，则 x_i 的取值为 0 或 1。该问题解的形式是一个 n 元组，且每个分量的取值为 0 或 1。由此可得，问题的解空间为：$(x_1,x_2,\cdots,x_n)$，其中 $x_i=0$ 或 1，$(i=1,2,\cdots,n)$。

5.3 回溯法—0-1背包问题求解步骤(P135)

步骤 2：确定解空间的组织结构。

问题的解空间描述了 2^n 种可能的解，也可以说是 n 个元素组成的集合的所有子集个数。可见，问题的解空间树为子集树。采用一棵满二叉树将解空间有效地组织起来，解空间树的深度为问题的规模 n。图 5-16 描述了 $n=4$ 时的解空间树。

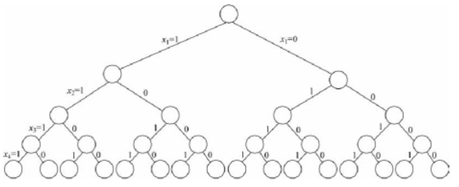


图 5-16　$n=4$ 时的解空间树

5.3 回溯法—0-1背包问题求解步骤(P135)

步骤 3：搜索解空间。

步骤 3-1：是否需要约束条件？如果需要，如何设置？

0-1 背包问题的解空间包含 2^n 个可能的解，是不是每一个可能的解描述的装入背包的物品的总重量都不超过背包的容量呢？显然不是，这个问题存在某种或某些物品无法装入背包的情况。因此，需要设置约束条件来判断所有可能的解描述的装入背包的物品总重量是否超出背包的容量，如果超出，为不可行解；否则为可行解。搜索过程将不再搜索那些导致不可行解的结点及其孩子结点。约束条件的形式化描述为：

$$\sum_{i=1}^{n} w_i x_i \leqslant W \qquad (5\text{-}1)$$

5.3 回溯法—0-1背包问题求解步骤(P135)

步骤 3-2：是否需要限界条件？如果需要，如何设置？

0-1 背包问题的可行解可能不止一个，问题的目标是找一个所描述的装入背包的物品总价值最大的可行解，即最优解。因此，需要设置限界条件来加速找出该最优解的速度。

如何设置限界条件呢？根据解空间的组织结构可知，任何一个中间结点 z（中间状态）均表示从根结点到该中间结点的分支所代表的行为已经确定，从 z 到其子孙结点的分支的行为是不确定的。也就是说，如果 z 在解空间树中所处的层次是 t，从第 1 种物品到第 $t-1$ 种物品的状态已经确定，接下来要确定第 t 种物品的状态。无论沿着 z 的哪一个分支进行扩展，第 t 种物品的状态就确定了。那么，从第 $t+1$ 种物品到第 n 种物品的状态还不确定。这样，可以根据前 t 种物品的状态确定当前已装入背包的物品的总价值，用 cp 表示。第 $t+1$ 种物品到第 n 种物品的总价值用 rp 表示，则 cp+rp 是所有从根出发的路径中经过中间结点 z 的可行解的价值上界。如果价值上界小于或等于当前搜索到的最优解描述的装入背包的物品总价值（用 bestp 表示，初始值为 0），则说明从中间结点 z 继续向子孙结点搜索不可能得到一个比当前更优的可行解，没有继续搜索的必要；反之，则继续向 z 的子孙结点搜索。因此，限界条件可描述为：

$$\mathrm{cp} + \mathrm{rp} > \mathrm{bestp} \qquad (5\text{-}2)$$

5.3 回溯法—0-1背包问题求解步骤(P136)

步骤 3-3：搜索过程。从根结点开始，以深度优先的方式进行搜索。根结点首先成为活结点，也是当前的扩展结点。由于子集树中约定左分支上的值为“1”，因此沿着扩展结点的左分支扩展，则代表装入物品，此时，需要判断是否能够装入该物品，即判断约束条件成立与否，如果成立，即进入左孩子结点，左孩子结点成为活结点，并且是当前的扩展结点，继续向纵深结点扩展；如果不成立，则剪掉扩展结点的左分支，沿着其右分支扩展。右分支代表物品不装入背包，肯定有可能导致可行解。但是沿着右分支扩展有没有可能得到最优解呢？这一点需要由限界条件来判断。如果限界条件满足，说明有可能导致最优解，即进入右分支，右孩子结点成为活结点，并成为当前的扩展结点，继续向纵深结点扩展；如果不满足限界条件，则剪掉扩展结点的右分支，开始向最近的活结点回溯。搜索过程直到所有活结点变成死结点结束。

5.3 回溯法—0-1背包问题求解示例(P136-9)

i	1	2	3	4
w_i	3	5	2	1
v_i	9	10	7	4

$W = 7$

$X^* = (1, 0, 1, 1)$

$W^* = 3+2+1 = 6$

$V^* = 9+7+4 = 20$

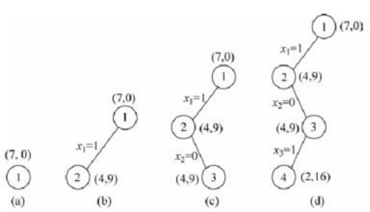


图 5-17　搜索过程 1

5.3 回溯法—0-1背包问题求解示例(P136-9)

i	1	2	3	4
w_i	3	5	2	1
v_i	9	10	7	4

$W = 7$

$X^* = (1, 0, 1, 1)$

$W^* = 3+2+1 = 6$

$V^* = 9+7+4 = 20$

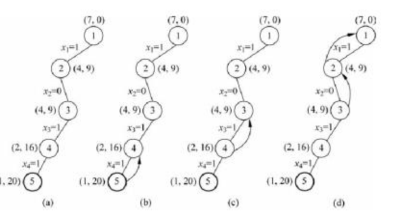


图 5-18　搜索过程 2

5.3 回溯法—0-1背包问题求解示例(P136-9)

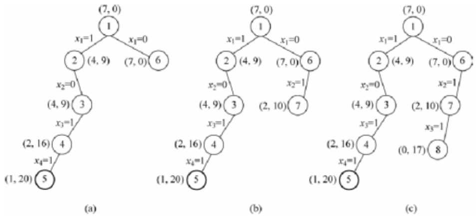


图 5-19 搜索过程 3

i	1	2	3	4
w_i	3	5	2	1
v_i	9	10	7	4

$W = 7$

$X^* = (1, 0, 1, 1)$
$W^* = 3 + 2 + 1 = 6$
$V^* = 9 + 7 + 4 = 20$

5.3 回溯法—0-1背包问题求解示例(P136-9)

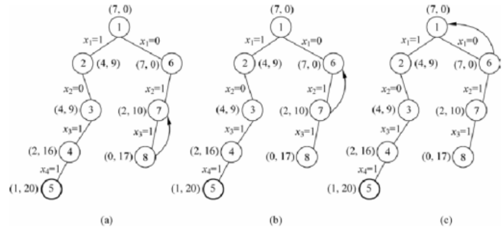


图 5-20 搜索过程 4

i	1	2	3	4
w_i	3	5	2	1
v_i	9	10	7	4

$W = 7$

$X^* = (1, 0, 1, 1)$
$W^* = 3 + 2 + 1 = 6$
$V^* = 9 + 7 + 4 = 20$

5.3 回溯法—0-1背包问题求解示例(P136-9)

i	1	2	3	4
w_i	3	5	2	1
v_i	9	10	7	4

$W = 7$

$X^* = (1, 0, 1, 1)$
$W^* = 3 + 2 + 1 = 6$
$V^* = 9 + 7 + 4 = 20$

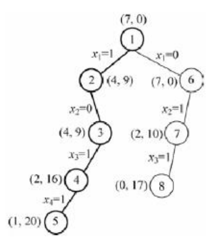


图 5-21 搜索过程 5

5.3 回溯法—0-1背包问题求解改进(P139-40)

- 将物品按价值重量比的倒序排列
  - 则剩余物品$l+1, \cdots, n$装入背包的最大可能价值是贪心地从$l+1$号物品装起，直至装满整个背包，其中最后一个可装物品允许部分装入
  - 由此可得装入背包物品的价值上限
  - 如果该价值上限与当前已装入物品价值的和小于已经获得的最优解，则停止向前并回溯
- 算法如右

```
VB = 0, WO = WR, i = 1
while W>0
    if w[i]<W
        VB += v[i]
        WO -= w[i]
    else
        break
    i++
end while
VB += v[i]*WO/w[i]
```


5.3 回溯法—0-1背包问题求解改进(P139-40)

i	1	2	3	4
w_i	3	5	2	1
v_i	9	10	7	4

$W = 7$

$X^* = (1, 0, 1, 1)$
$W^* = 3 + 2 + 1 = 6$
$V^* = 9 + 7 + 4 = 20$

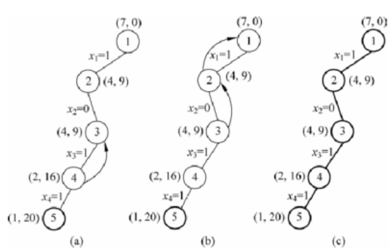


图 5-22 搜索过程

5.3 回溯法—0-1背包问题求解算法

Algorithm 1.1: Knapsack1(ℓ)
global $X, OptX, OptP$
if $\ell = n + 1$

$$\text{then } \begin{cases} \text{if } \sum_{i=1}^{n} w_i x_i \le M \\ \quad \text{then } \begin{cases} CurP \leftarrow \sum_{i=1}^{n} p_i x_i \\ \text{if } CurP > OptP \\ \quad \text{then } \begin{cases} OptP \leftarrow CurP \\ OptX \leftarrow [x_1, \ldots, x_n] \end{cases} \end{cases} \end{cases}$$

$$\text{else } \begin{cases} x_\ell \leftarrow 1 \\ \text{Knapsack1}(\ell + 1) \\ x_\ell \leftarrow 0 \\ \text{Knapsack1}(\ell + 1) \end{cases}$$

参考 王秋芬P140-3

http://lib.hunre.edu.vn/Gg-7095-ggdx-06Knapsackbacktrack.pdf

5.3 回溯法—0-1背包问题求解算法

Algorithm 2.1: Knapsack2(ℓ, $CurW$)

global $X, OptX, OptP$

if $\ell = n+1$

$$
\textbf{then}\ \begin{cases} \text{if } \sum_{i=1}^{n} p_i x_i > OptP \\ \quad \textbf{then}\ \begin{cases} OptP \leftarrow \sum_{i=1}^{n} p_i x_i \\ OptX \leftarrow [x_1, \ldots, x_n] \end{cases} \end{cases}
$$

$$
\textbf{else}\ \begin{cases} \text{if } CurW + w_\ell \le M \\ \quad \textbf{then}\ \begin{cases} x_\ell \leftarrow 1 \\ \mathsf{Knapsack2}(\ell+1, CurW + w_\ell) \\ x_\ell \leftarrow 0 \\ \mathsf{Knapsack2}(\ell+1, CurW) \end{cases} \\ \quad \textbf{else}\ \begin{cases} x_\ell \leftarrow 0 \\ \mathsf{Knapsack2}(\ell+1, CurW) \end{cases} \end{cases}
$$

$$CurW = \sum_{i=1}^{\ell-1} w_i x_i.$$

参考 王秋芬P140-3

http://lib.hunre.edu.vn/Gg-7095-ggdx-06Knapsackbacktrack.pdf

5.3 回溯法—0-1背包问题求解算法

Algorithm 3.1: Knapsack3(ℓ, $CurW$)

external GreedyRKnap()

global $X, OptX, OptP$

if $\ell = n+1$

$$
\textbf{then}\ \begin{cases} \text{if } \sum_{i=1}^{n} p_i x_i > OptP \\ \quad \textbf{then}\ \begin{cases} OptP \leftarrow \sum_{i} p_i x_i \\ OptX \leftarrow [x_1, \ldots, x_n] \end{cases} \end{cases}
$$

$$
\textbf{else}\ \begin{cases} B \leftarrow \sum_{i=1}^{\ell-1} p_i x_i + \mathsf{GreedyRKnap}(n-\ell+1; p_\ell, \ldots, p_n, w_\ell, \ldots, w_n, M - CurW) \\ \text{if } B \le OptP \ \textbf{then return} \\ \text{if } CurW + w_\ell \le M \\ \quad \textbf{then}\ \begin{cases} x_\ell \leftarrow 1 \\ \mathsf{Knapsack3}(\ell+1, CurW + w_\ell) \end{cases} \\ \text{if } B \le OptP \ \textbf{then return} \\ x_\ell \leftarrow 0 \\ \mathsf{Knapsack3}(\ell+1, CurW) \end{cases}
$$

$$\frac{p_1}{w_1} \ge \cdots \ge \frac{p_n}{w_n}.$$

算法复杂度：$O(n2^n)$

参考 王秋芬P140-3

http://lib.hunre.edu.vn/Gg-7095-ggdx-06Knapsackbacktrack.pdf

5.3 回溯法—0-1背包问题算法复杂度

（7）算法分析。

判断约束函数需 $O(1)$，在最坏情况下有 2^n-1 个左孩子，约束函数耗时最坏为 $O(2^n)$。计算上界限界函数需要 $O(n)$时间，在最坏情况下有 2^n-1 个右孩子需要计算上界，限界函数耗时最坏为 $O(n2^n)$。0-1 背包问题的回溯算法所需的计算时间为 $O(2^n)+O(n2^n)=O(n2^n)$。

5.3 回溯法—TSP问题描述(P152)

【例 5-10】 旅行商问题。

（1）问题描述。

设有 n 个城市组成的交通图，一个售货员从住地城市出发，到其他城市各一次去推销货物，最后回到住地城市。假定任意两个城市 i,j 之间的距离 d_{ij} ($d_{ij}=d_{ji}$)是已知的，问应该怎样选择一条最短的路线？

5.3 回溯法—TSP问题分析(P152)

（2）问题分析。

旅行商问题给定 n 个城市组成的无向带权图 $G=(V,E)$，顶点代表城市，权值代表城市之间的路径长度。要求找出以住地城市开始的一个排列，按照这个排列的顺序推销货物，所经路径长度是最短的。问题的解空间是一棵排列树。显然，对于任意给定的一个无向带权图，存在某两个城市（顶点）之间没有直接路径（边）的情况。也就是说，并不是任何一个以住地城市开始的排列都是一条可行路径（问题的可行解），因此需要设置约束条件，判断排列中相邻两个城市之间是否有边相连，有边相连则能走通；反之，不是可行路径。另外，在所有可行路径中，要找一条最短的路线，因此需要设置限界条件。

5.3 回溯法—TSP问题求解步骤(P153)

步骤 1：定义问题的解空间。

旅行商问题的解空间形式为 n 元组($x_1,x_2,\cdots,x_n$)，分量 x_i($i=1,2,\cdots,n$)表示第 i 个去推销货物的城市号。假设住地城市编号为城市 1，其他城市顺次编号为 2，3，…，n。n 个城市组成的集合为 $S=\{1,2,\cdots,n\}$。由于住地城市是确定的，因此 x_1 的取值只能是住地城市，即 $x_1=1$，$x_i \in S-\{x_1,x_2,\cdots,x_{i-1}\}$，$i=2,\cdots,n$。

步骤 2：确定解空间的组织结构。

该问题的解空间是一棵排列树，树的深度为 n。$n=4$ 的旅行商问题的解空间树如图 5-34 所示。

步骤 3：搜索解空间。

步骤 3-1：设置约束条件。

用二维数组 $g[\,][\,]$存储无向带权图的邻接矩阵，如果 $g[i][j] \neq \infty$表示城市 i 和城市 j 有边相连，能走通。

5.3 回溯法—TSP问题求解步骤(P153)

步骤 3：搜索解空间。

步骤 3-1：设置约束条件。

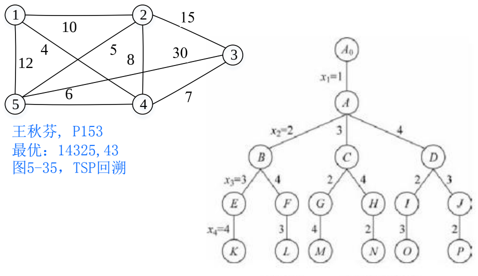
用二维数组 $g[\,][\,]$ 存储无向带权图的邻接矩阵，如果 $g[i][j]\neq\infty$ 表示城市 i 和城市 j 有边相连，能走通。

步骤 3-2：设置限界条件。

用 cl 表示当前已走过的城市所用的路径长度，用 bestl 表示当前找到的最短路径的路径长度。显然，继续向纵深处搜索时，cl 不会减少，只会增加。因此当 cl≥bestl 时，没有继续向纵深处搜索的必要。限界条件可描述为：cl<bestl，cl 的初始值为 0，bestf 的初始值为 $+\infty$。

步骤 3-3：搜索过程。扩展结点沿着某个分支扩展时需要判断约束条件和限界条件，如果两者都满足，则进入深一层继续搜索。反之，剪掉扩展生成的结点。搜索到叶子结点时，找到当前最优解。搜索过程直到全部活结点变成死结点。

5.3 回溯法—TSP问题求解示例(P153-6)



王秋芬，P153
最优：14325, 43
图5-35，TSP回溯

图 5-34 $n=4$ 的解空间树

5.3 回溯法—TSP问题求解示例(P153-6)

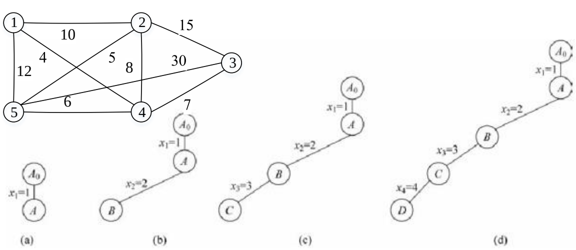
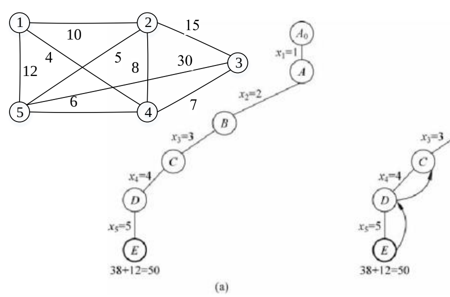


图 5-36 搜索过程 1

王秋芬，P153
最优：14325, 43
图5-35，TSP回溯

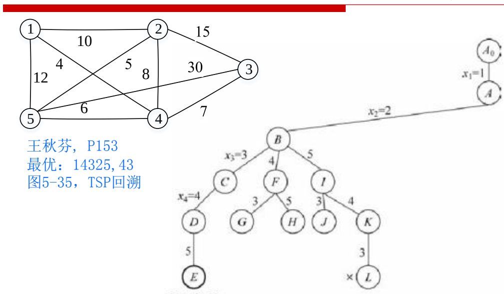
5.3 回溯法—TSP问题求解示例(P153-6)



王秋芬，P153
最优：14325, 43
图5-35，TSP回溯

图 5-37 搜索过程 2

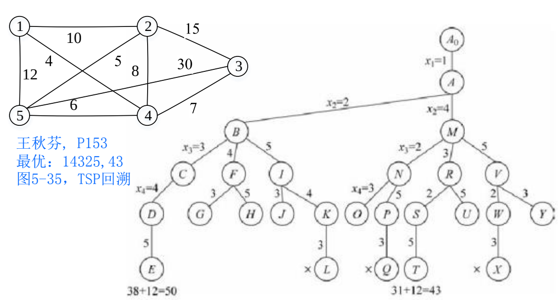
5.3 回溯法—TSP问题求解示例(P153-6)



王秋芬，P153
最优：14325, 43
图5-35，TSP回溯

图 5-38 搜索过程 3

5.3 回溯法—TSP问题求解示例(P153-6)



王秋芬，P153
最优：14325, 43
图5-35，TSP回溯

图 5-39 搜索过程 4

5.3 回溯法—TSP问题求解示例(P153-6)

王秋芬，P153
最优：14325，43
图5-35，TSP回溯

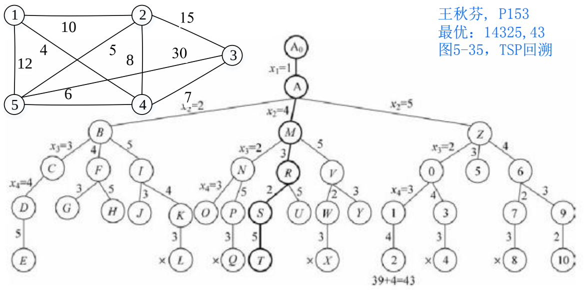


图 5-40 搜索过程 5

5.3 回溯法—TSP问题求解算法(P156)

```
void Traveling(t)
  if t>n
    if W[x[n],1]!=∞ && L+W[x[n],1]<bestL
      bestX[] <- x[]
      bestL += L+W[x[n],1]
  else
    for j=t to n
      if W[x[t-1],x[j]]!=∞ && L+W[x[n],1]<bestL
        swap(x[t], x[j])
        L += W[x[n],1]
        Traveling(t+1)
        L -= W[x[n],1]
        swap(x[t], x[j])
```


5.3 回溯法—TSP问题求解算法

```
Algorithm TSP_Backtrack(A, ℓ, lengthSoFar, minCost)
n ← length[A]    // number of elements in the array A
if ℓ = n
   then minCost ← min(minCost, lengthSoFar + distance[A[n], A[1]])
   else for i ← ℓ + 1 to n
             do Swap A[ℓ + 1] and A[i]    // select A[i] as the next city
                newLength ← lengthSoFar + distance[A[ℓ], A[ℓ + 1]]
                if newLength ⩾ minCost    // this will never be a better solution
                   then skip    // prune
                   else minCost ←
                           min(minCost, TSP_Backtrack(A, ℓ + 1, newLength, minCost))
                Swap A[ℓ + 1] and A[i]    // undo the selection
 return minCost
```

http://www.win.tue.nl/~kbuchin/teaching/2IL15/backtracking.pdf
The Eindhoven University of Technology

5.3 回溯法—TSP问题求解算法

（6）算法分析。

判断限界函数需要 $O(1)$ 时间，在最坏情况下有 $1+(n-1)+[(n-1)(n-2)]+\cdots+[(n-1)(n-2)\cdots 2]\leqslant n(n-1)!$ 个结点需要判断限界函数，故耗时 $O(n!)$；在叶子结点处记录当前最优解需要耗时 $O(n)$，在最坏情况下会搜索到每一个叶子结点，叶子结点有 $(n-1)!$ 个，故耗时为 $O(n!)$。因此，旅行售货员问题的回溯算法所需的计算时间为 $O(n!)+O(n!)=O(n!)$。

目录

- 两个基本问题
- 穷举搜索
- 深度优先搜索
- 回溯法
- 作业

1. 给出0-1背包问题的穷举算法伪代码，并说明其复杂度
2. 给出TSP问题的回溯算法伪代码，并说明其复杂度
3. 给出回溯法求解问题的基本步骤
4. 给出0-1背包问题的回溯算法伪代码，并说明其复杂度
5. 给出TSP问题的回溯算法伪代码，并说明其复杂度