北京工業大學

算法设计与分析

课程报告

组员学号 ________ 姓名 __________

组员学号 ________ 姓名 __________

组员学号 ________ 姓名 __________

组员学号 ________ 姓名 __________

组员学号 ________ 姓名 __________

组员学号 ________ 姓名 __________

报告完成日期 ________2019.12.10________

教师评语:	
成绩:	指导教师签字:
	评阅日期: 年 月 日

目录

一.广义背包

1.问题重述

广义背包问题的描述如下：给定载重量为 M 的背包和 n 种物品，每种物品有一定的重量和价值，现在需要设计算法，在不超过背包载重量的前提下，巧妙选择物品，使得装入背包的物品的总价值最大化。规则是，每种物品均可装入背包多次或不装入（但不能仅装入物品的一部分）。

2.符号说明

$\boldsymbol{w_i}$	**第 i 件物品的重量(为正整数)**
$\boldsymbol{v_i}$	第 i 件物品的价值(为自然数)
$\boldsymbol{x}$	解向量
$\boldsymbol{x_i}$	第 i 件物品转入背包的数量
M	背包最大承载重量
n	物品种类数量
$\boldsymbol{V_{max}}$	最大价值
$\boldsymbol{d(i,j)}$	背包现有价值
$\boldsymbol{x_i}$	第 i 件物品转入背包的数量 $0 \le x_i \le \left\lfloor \frac{M}{\boldsymbol{w_i}} \right\rfloor$
i	选择的第 i 种物品 $i = [1,n]$
j	背包剩余承载量
k	指示变量. $k \in [0,M]$

3.问题描述以及递推公式

问题描述

$$\begin{cases} Vmax = \max \sum_{i=1}^{n} v_i x_i \\ \sum_{i=1}^{n} w_i x_i \le M \\ x_i \in [0, \left\lfloor \frac{M}{\boldsymbol{w_i}} \right\rfloor] \\ i \in [1,n] \end{cases}$$

递推公式

$$d(i,j) = \begin{cases} max\,\{\, d(i-1,j), d(i-1,j-x_i w_i) + x_i v_i \} & ,j \ge x_i w_i \\ d(i-1,j) & ,0 \le j < w_i \end{cases} \quad ①$$

$$d(0,k) = 0\ (k \in [0,M])$$

$$Vmax = d(n,M)$$

4.求出解向量

通过例子说明:

$M = 10$

$w = [1,6,4,3]$

$v = [1,3,2,6]$

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9	10
0	1	2	6	7	8	12	13	14	18	19

$$Vmax = 19$$

$$x = [1,0,0,3]$$

5.优化

$$d(i,j) = \begin{cases} max\{\, d(\, i-1,j\,), d(\, i\,, j-w_i\,) + v_i \} & ,j \geq w_i > 0 \\ d(\, i-1, j\,) & ,0 \leq j < w_i \end{cases} \quad ②$$

与 0-1 背包的对比

	0-1 背包				**广义背包**			
质量	w_1	w_2	…	w_n	w_1	w_2	…	w_n
价值	v_1	v_2	…	v_n	v_1	v_2	…	v_n
装入数	y_1	y_2	…	y_n	x_1	x_2	…	x_n

$$y_i \in \{0,1\}$$

$$x_i \in [0, \left\lfloor \frac{M}{\boldsymbol{w_i}} \right\rfloor]$$

转换: 广义背包中第 i 件物品,选了x_i次 → 0-1 背包的x_i种物品

即:

	广义背包								
质量	w_1	w_1	…	w_1	w_2	w_2	…	w_2	…
价值	v_1	v_1	…	v_1	v_2	v_2	…	v_2	…
装 入 数	a_{11}	a_{12}	…	$a_{1\left\lfloor \frac{M}{w_1} \right\rfloor}$	a_{21}	a_{22}	…	$a_{2\left\lfloor \frac{M}{w_2} \right\rfloor}$	…

$$a_{ic} = \{0,1\} \;\; (i \in [1,n], \qquad c \in [1, \left\lfloor \frac{M}{\boldsymbol{w_i}} \right\rfloor])$$

转换后的公式

$\boldsymbol{d(i,j)}$	**背包现有价值**

$\boldsymbol{n}$	物品的种类
$\boldsymbol{n'}$	转换前的物品的种类
$\boldsymbol{w_i}$	第 i 件物品的质量
$\boldsymbol{x_i}$	第 i 件物品转入背包的数量
i	选择的第 i 种物品
j	背包剩余承载量
k	下标变量
$\boldsymbol{Vmax}$	最优解

$$\begin{cases} Vmax = \max \sum_{i=1}^{n} v_i x_i \\ \sum_{i=1}^{n} w_i x_i \ \le M \\ x_i \in \{0,1\} \end{cases} \quad ③$$

递推公式:

$$d(i,j) = \begin{cases} max\,\{\,d(i-1,j), d(i-1,j-w_i)+v_i\} & ,j \ge w_i > 0 \\ d(i-1,j) & ,0 \le \ j < w_i \end{cases}$$

$$\boldsymbol{n} = \sum_{k=1}^{\boldsymbol{n'}} \left\lfloor \frac{M}{\boldsymbol{w_k}} \right\rfloor, \qquad k \in [1, \boldsymbol{n'}]$$

$$i \in [1, \boldsymbol{n}]$$

6.证明最优子结构(反证法)

通过转化,则证明广义背包的最优子结构与 0-1 背包类似

证明:

假设$(x_1, x_2, \ldots, x_n)$是公式③所对应的最优解,

则$(x_1, x_2, \ldots, x_{n-1})$是下列子问题的最优解。

$$\begin{cases} \max \sum_{i=1}^{n-1} v_i x_i \\ \sum_{i=1}^{n-1} w_i x_i \ \le M - w_n x_n \\ x_i \in \{0,1\} \\ i \in [1, n-1] \end{cases}$$

反证法:

证明:

假设$(x_1, x_2, \ldots, x_{n-1})$不是上述子问题的最优解，而$(y_1, y_2, \ldots, y_{n-1})$是上述子问题的最优解.

则

$$\sum_{i=1}^{n-1} w_i x_i < \sum_{i=1}^{n-1} w_i y_i$$

$$w_n x_n + \sum_{i=1}^{n-1} w_i y_i \le M$$

因此

$$v_n x_n + \sum_{i=1}^{n-1} v_i y_i \ \ge \ \sum_{i=1}^{n} v_i x_i$$

$$w_n x_n + \sum_{i=1}^{n-1} w_i y_i \le M$$

故$(y_1, y_2, \dots, y_{n-1}, x_n)$为该问题的最优解,与假设中$(x_1, x_2, \dots, x_n)$为该问题的最优解相矛盾.

证毕.

7.优化后的算法以及时空复杂度

```
/*
*递推公式
```

$$d(i,j) = \begin{cases} max\,\{\, d(\,i-1,j\,), d(\,i\,,j-w_i\,)+v_i\,\} & ,j \ge w_i > 0 \\ d(\,i-1,j\,) & ,0 \le \ j < w_i \end{cases}$$

```
*时空复杂度
        M:背包最大承重量
        n:物品种类个数
```

时间复杂度	$\boldsymbol{O(nM)}$
空间复杂度	$O(M)$

```
*输入
    m:物品质量数组
    M:背包最大承重量
   n:物品种类个数
*输出
   最大价值与选择物品数量
*/
function GKP(M, m, v) {
    var n = m.length
    var i, j;
    var x = {} //解向量
    for (i = 0; i <= n; i++) {
        x[i] = 0
    }
    var path = new Map()
    var f = new Array(M + 1).fill(0)
    for (i = 1; i <= n; i++) {
```

```
        for (j = m[i - 1]; j <= M; j++) {
            var tmp = f[j - m[i - 1]] + v[i - 1]
            if (f[j] < tmp) {
                f[j] = tmp
                path[[i, j]] = 1//用 hash 表一次储存 (i,j)
            }
        }
    }
    //解路径
    j = M
    i = n
    while (i > 0 && j > 0) {
        if (path[[i, j]] == 1) {
            x[i - 1]++
            j = j - m[i - 1]
        } else {
            i = i - 1
        }
    }

    return [f[M], x]
}
//广义背包测试数据
var M = 10
var m = [1, 6, 4, 3]
var v = [1, 3, 2, 6]
console.log(GKP(M, m, v))

M = 230
m = [20, 25, 40, 12, 31]
v = [1, 2, 3, 1, 5]
console.log(GKP(M, m, v))
```

8.测试用例

组号	测试输入	算法输出

1	`var M = 10`
	`var m = [1, 6, 4, 3]`
	`var v = [1, 3, 2, 6]`
		最少花费 = 19
		解向量 = [1,0,0,3]

2	`M = 230`
	`m = [20, 25, 40, 12, 31]`
	`v = [1, 2, 3, 1, 5]`
		最少花费 = 36
		解向量 = [0,0,0,1,7]

二.TSP

1.问题描述

所谓 TSP 问题是指旅行商要去 n 个城市推销商品，其中每个城市到达且仅到达一次，并且要求所走的路程最短（该问题又称货郎担问题、邮递员问题、售货员问题等）。

2.符号说明以及递推公式

城市→点

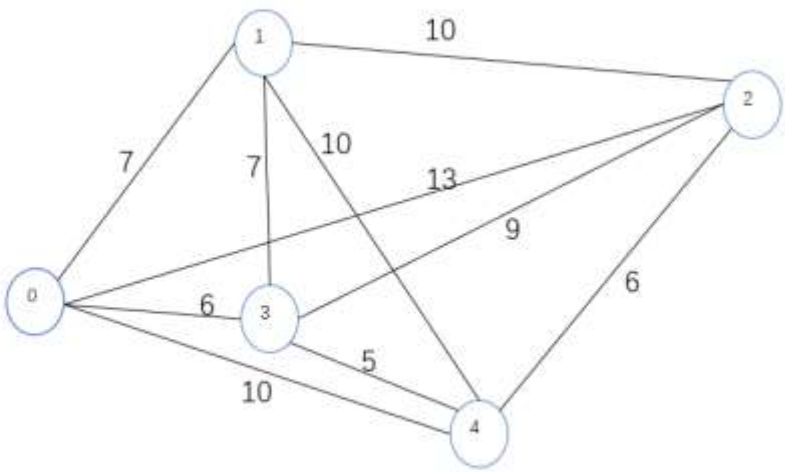
s	起点
V	所有点的集合
V'	从点 i 到点 s 包含所有点的集合
c_{ij}	从点 i 到点 j 的花费
$d(i,V')$	当前点 i,经过V'所有点一次且仅一次到点 s 的最小花费

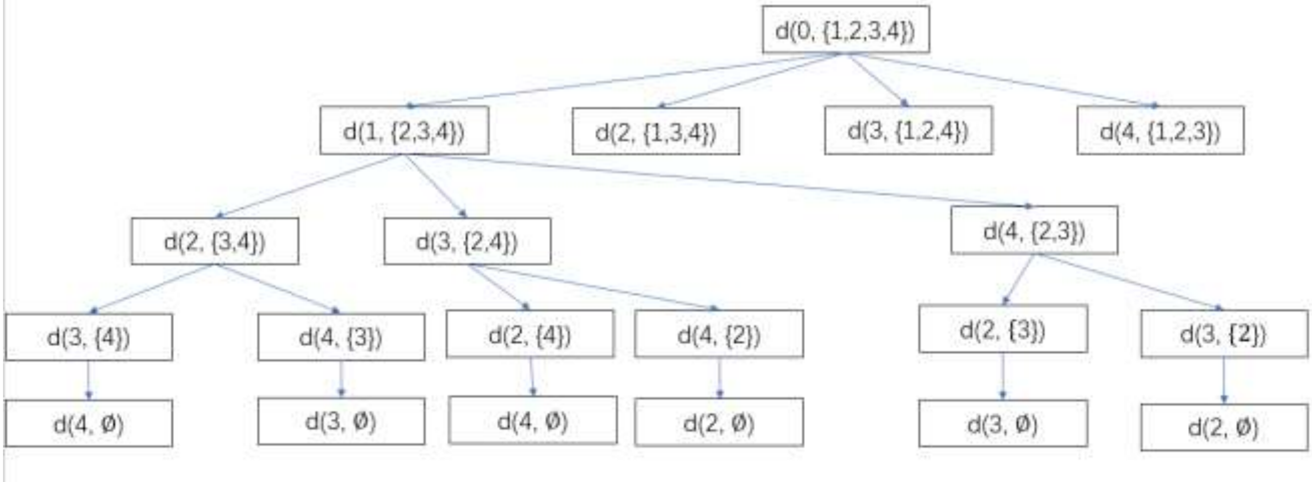
则$d(i,V')$的递推关系

$$d(i,V')=\begin{cases} c_{is}\quad, V'=\emptyset, i\neq s \\ \min\limits_{k\in V'}\{c_{ik}+d(k,V'-\{k\})\}, V'\neq\emptyset \end{cases}$$

$$d_{min}=d(s,V-\{s\})$$

3.例子





4.状态压缩

以集合 S={1,2,3}为例

集合	状态压缩后二进制
Ø	000B
{1}	001B
{2}	010B
{3}	100B
{1,2}	011B
{1,3}	101B
{2,3}	110B
{1,2,3}	111B

5.算法实现以及算法时空复杂度

/*
*算法依据的递推公式

$$d(i,V') = \begin{cases} c_{is} \quad , \ V' = \emptyset, i \neq s \\ \min\limits_{k \in V'}\{c_{ik} + d(k, V' - \{k\})\}, V' \neq \emptyset \end{cases}$$

*时空复杂度

*n 为点的个数

时间复杂度	$\boldsymbol{O(n^2 2^n)}$
空间复杂度	$O(\boldsymbol{n}2^n)$

*说明

*只处理无向图

```
    *起点为 0
*输入
    *D 为二维数组(nxn 矩阵)
*输出
    * 最短路径,最短花费
*代码实现
    *JavaScript 代码
*/
function TSP(D) {
    const INF = 65535 //定义的最大值
    var n= D.length // n 的个数
    var i, j, k, min, tmp;
    var b = 1 << (n - 1); // 点集状态总数
    var dp = {}  // 记录状态(状态压缩)
    /*
    状态压缩
    将一个集合压缩为二进制位数
    */
    var bridge = {} //记录中间节点
    for (i = 0; i < n; i++) { //初始化 dp 与 bridge
        dp[i] = {}
        bridge[i] = {}
        for (j = 0; j < b; j++)
        {
            dp[i][j] = 0;
            bridge[i][j] = -1;
        }
    }
    for (i = 0; i < n; i++) { //初始化 dp 的第 0 列
        dp[i][0] = D[i][0]
    }
    //初始化动态规划表结束

    //遍历 dp
    for (i = 1; i < b - 1; i++) {
        for (j = 1; j < n; j++) {
            if ((1 << (j - 1) & i) == 0) {
                //点 j 未访问
```

$$//\min_{k\in V'}\{c_{ik}+d(k,V'-\{k\})\},V'\neq\emptyset$$

```
                min = INF
                for (k = 1; k < n; k++) { //遍历点集
                    if (1 << (k - 1) & i) {
                        //点 k 在集合中
```

```
                        // 松弛操作
                        tmp = D[j][k] + dp[k][i - (1 << (k - 1))]
//{c_ik + d(k,V' - {k})}
                        if (tmp < min) {
                            min = tmp
                            dp[j][i] = min
                            bridge[j][i] = k
                        }
                    }
                }
            }
        }
    }
    //处理最后一列
    //c_is  , V' = ∅,i ≠ s
    min = INF
    for(k=1;k<n;k++) {

        tmp = D[0][k] + dp[k][b-1-(1<<(k-1))]
        if( tmp < min) {
            min = tmp
            dp[0][b-1] = min
            bridge[0][b-1] = k
        }
    }
    var Vmax = dp[0][b-1]
    //构造路径
     var path = [0]
    for(i=b-1,j=0;i>0;) {
        j = bridge[j][i] //  下一个节点
        i = i-( 1<<(j-1))
        path.push(j)
    }
    path.push(0)
    return [path, Vmax]
}


//测试数据

var m = [

    [0, 7, 6, 10, 13],
    [7, 0, 7, 10, 10],
    [6, 7, 0, 5, 9],
```

```
    [10, 10, 5, 0, 6],
    [13, 10, 9, 6, 0]
]

var res = TSP(m)
console.log(res)
```

6.数据测试

<table>
<tr><th>组号</th><th>输入数据</th><th>算法输出</th></tr>
<tr><td>1</td><td><pre>var m = [
    [0, 7, 6, 10, 13],
    [7, 0, 7, 10, 10],
    [6, 7, 0, 5, 9],
    [10, 10, 5, 0, 6],
    [13, 10, 9, 6, 0]
] //输入的矩阵</pre></td><td><pre>最短路径 : [0, 1, 4, 3, 2, 0]
最少花费 : 34</pre></td></tr>
<tr><td>2</td><td><pre>var n = 22
var m = new Array(n)
for (var i = 0; i < n; i++) {
    m[i] = new Array(n).fill(0
)
}
var s1 = new Date().getTime()
res = TSP(m)
var s2 = new Date().getTime()
console.log("花费时
间:",s2 - s1)</pre></td><td><pre>花费时间(单位:ms): 6873</pre></td></tr>
<tr><td>3</td><td><pre>var n = 23
var m = new Array(n)
for (var i = 0; i < n; i++) {
    m[i] = new Array(n).fill(0
)
}</pre></td><td><pre>FATAL ERROR:
 JavaScript heap out of memory</pre>详见附录
此时
$$22^2 2^{22} = 2.030043136 \times 10^9$$
$$22 \times 2^{22} = 9.2274688 \times 10^7$$</td></tr>
</table>

三.附录

1.参考资料

1.用动态规划方法求解广义背包问题 豆丁

https://www.docin.com/p-1571618425.html

2. 动态规划求解 TSP(旅行商)问题 https://blog.csdn.net/masibuaa/article/details/8236074

2.TSP 运行错误信息

<--- Last few GCs --->

[9124:00000200ED644110] 6900 ms: Mark-sweep 1386.2 (1396.5) -> 1354.9 (1361.7) MB, 143.6 / 0.0 ms (+ 1.9 ms in 1 steps since start of marking, biggest step 1.9 ms, walltime since start of marking 151 ms) (average mu = 0.241, current mu = 0.142) alloc[9124:00000200ED644110] 7063 ms: Mark-sweep 1387.0 (1394.0) -> 1374.7 (1381.5) MB, 124.8 / 0.0
ms (+ 2.0 ms in 2 steps since start of marking, biggest step 2.0 ms, walltime since start of marking 146 ms) (average mu = 0.232, current mu = 0.223) alloc

<--- JS stacktrace --->

==== JS stack trace ===

0: ExitFrame [pc: 000001391CBDC5C1]
Security context: 0x01413de17991 <JSObject>
1: TSP [000003625F904939] [C:\Users\13298\Documents\Python Scripts\ds_git\TSP_DP.js:~2] [pc=000001391CC71FA5](this=0x01eb3b58d481 <JSGlobal Object>,D=0x03625f904979 <JSArray[23]>)
2: /* anonymous */ [000003625F904A89] [C:\Users\13298\Documents\Python Scripts\ds_git\TSP_DP.js:87] [bytecode=000002FCD16D6DC1 offset=174](this=0x03625f904bb9 <Object map...
FATAL ERROR: Ineffective mark-compacts near heap limit Allocation failed - JavaScript heap out of memory
1: 00007FF7FA7EDD8A v8::internal::GCIdleTimeHandler::GCIdleTimeHandler+4506
2: 00007FF7FA7C8886 node::MakeCallback+4534
3: 00007FF7FA7C9200 node_module_register+2032
4: 00007FF7FAAE30DE v8::internal::FatalProcessOutOfMemory+846
5: 00007FF7FAAE300F v8::internal::FatalProcessOutOfMemory+639

```
 6: 00007FF7FACC9804 v8::internal::Heap::MaxHeapGrowingFactor+9620
 7: 00007FF7FACC07E6 v8::internal::ScavengeJob::operator=+24550
 8: 00007FF7FACBEE3C v8::internal::ScavengeJob::operator=+17980
 9: 00007FF7FACC7B87 v8::internal::Heap::MaxHeapGrowingFactor+2327
10: 00007FF7FACC7C06 v8::internal::Heap::MaxHeapGrowingFactor+2454
11: 00007FF7FADF1CD8 v8::internal::Factory::AllocateRawArray+56
12: 00007FF7FADF2652 v8::internal::Factory::NewFixedArrayWithFiller+66
13: 00007FF7FAE2748C v8::internal::Factory::NewCallHandlerInfo+157532
14: 00007FF7FAE27124 v8::internal::Factory::NewCallHandlerInfo+156660
15: 00007FF7FAE28275 v8::internal::Factory::NewCallHandlerInfo+161093
16: 00007FF7FAB649A8 v8::internal::SharedFunctionInfo::SetScript+23528
17: 00007FF7FAB47953 v8::internal::JSReceiver::class_name+20595
18: 00007FF7FADE2F59 v8::internal::wasm::WasmCodeManager::LookupCode+15273
19: 00007FF7FADE6014 v8::internal::wasm::WasmCodeManager::LookupCode+27748
20: 000001391CBDC5C1
```

3.运行环境

nodejs 版本

```
PS C:\Users\13298\Documents\Python Scripts\ds_git> node  -v
v10.16.3
```

电脑环境

Windows 版本

Windows 10 家庭中文版

系统

处理器:	Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz 2.50 GHz
已安装的内存(RAM):	16.0 GB (15.9 GB 可用)
系统类型:	64 位操作系统，基于 x64 的处理器
笔和触控:	没有可用于此显示器的笔或触控输入

4.结论

动态规划仅能解决复杂度低的的 TSP 问题. 对于更复杂的问题, 应选用其他算法(如蚁群算法)