

学号：19S103256

姓名：文荟俨

实验三：近似算法

1. 实验目的

- (1) 掌握近似算法的基本设计思想与方法
- (2) 掌握集合覆盖问题近似算法的设计思想与方法
- (3) 熟练使用高级编程语言实现近似算法
- (4) 利用实现测试给出的不同近似算法的性能以理解其优缺点

2. 实验问题

集合覆盖问题。

- 输入:有限集 X , X 的子集合族 F , $X = \bigcup_{S \in F} S$
- 输出: $C \subseteq F$, 满足 $X = \bigcup_{S \in C} S$ 且 C 是满足 $X = \bigcup_{S \in C} S$ 的最小集族, 即 $|C|$ 最小。

3. 实验步骤

(1) 数据生成

该部分算法完全遵从实验指导书部分。实验数据存储采用自定的.tf 二进制文件存储, 具体格式如图 1 所示。起始的 1 个 int 存储 X 集合的元素个数 m , 接下来的 $m * \text{sizeof}(\text{int})$ 个字节存储 X 的具体内容。这之后的 1 个 int 存储 $|S|$, 接着用 1 个 int 存储 S_1 集合的元素个数 s_1 , 再往后的 $s_1 * \text{sizeof}(\text{int})$ 个字节存储 S_1 的具体内容。如此往复, 直到存储完所有的 S 。

(int) X元素个数n
(int) X元素
....
(int) S元素个数m
(int) S1元素个数s1
(int) S1元素
....
....
(int) Sm元素个数sm
(int) Sm元素
....

图 1 数据集 tf 二进制文件格式说明

(2) 贪心算法

算法 **Greedy_cover**(X, F)

输入: 有限集 X , X 的子集合族 F , $X = \bigcup_{S \in F} S$, $|X| = |F|$

输出: $C \subseteq F$, 满足 $X = \bigcup_{S \in C} S$ 且 C 是满足 $X = \bigcup_{S \in C} S$ 的最小集族, 即 $|C|$ 最小

1: $U \leftarrow X$
2: $C \leftarrow \emptyset$
3: While $U \neq \emptyset$ Do
4: 贪心选择能覆盖最多 U 元素的子集 S
5: $U \leftarrow U - S$
6: $C \leftarrow C \cup \{S\}$
7: Return C

算法第 3-6 步的循环次数至多为 $\min(|X|, |F|)$ ，第 4 步需要时间 $O(|F||X|)$ ，因此总的复杂度为二者相乘，为 $O(|F||X|\min(|X|, |F|))$ ，在 $|X|=|F|$ 的前提下，是一个 $O(n^2)$ 的近似算法。

该部分核心代码如清单 1 所示。

清单 1 贪心算法核心代码

```
while (real_size(U)>0) {
    print_temp++;
    //chose S
    max_cover_num = 0;
    max_cover = -1;
    for (int i = 0; i < S_flag.size(); i++) {
        //找 S 中 cover U 的最大的集合
        if (S_flag[i] != -1) {
            cover_temp = cover_num(U, S, S_num[i], sum_num(S_num,
i));
            if(cover_temp>max_cover_num) {
                max_cover_num = cover_temp;
                max_cover = i;
            }
        }
    }
    S_flag[max_cover] = -1;
    C.push_back(max_cover);
    for (int i = 0; i < S_num[max_cover]; i++) {
        for (int j = 0; j < U.size(); j++) {
            if (U[j] == -1) continue;
            else if (S[sum_num(S_num, max_cover) + i] == U[j]) {
                U[j] = -1;
            }
        }
    }
}
if (whether_show_process) {
    cout << "第" << print_temp << "次新增元素:";
    for (int i = 0; i < S_num[max_cover]; i++) {
        cout << S[sum_num(S_num, max_cover) + i] << " ";
    }
    cout << endl;
}
cout << "第" << print_temp << "次剩余元素:";
```

```

    cout << real_size(U) << endl;
}

```

(3) 线性规划舍入算法

算法 **Appox_Cover_VC**(X, F)

输入：有限集 X, X 的子集合族 F, $X = \bigcup_{S \in F} S$, $|X| = |F|$

输出： $C \subseteq F$, 满足 $X = \bigcup_{S \in F} S$ 且 C 是满足 $X = \bigcup_{S \in F} S$ 的最小集族, 即 $|C|$ 最小

1: 根据 X 和 F 的关系, 构建线性规划表达式

2: 调用 glpk 库求解 LP 松弛问题的最优解 x

3: For $S \in F$ **Do**

4: If $x_s \geq 1/f$ then $C \leftarrow C \cup \{S\}$

5: Return C

其中, f 是 X 中元素的最大频率。所谓频率是指 F 中包含某个元素的集合个数。原问题需要表示成如下线性规划, 实验中我们取 $C(S)=1$ 。

$$\begin{aligned}
 & \text{minimize } \sum_{S \in F} C(S)x_S \\
 & \text{s.t. } \sum_{S: e \in S} x_S \geq 1, e \in X \\
 & \quad x_s \in [0, 1], S \in F
 \end{aligned}$$

该部分的核心代码如清单 2 所示。

清单 2 线性规划求子集覆盖问题的核心代码

```

double *es = (double *)malloc((size * size + 1) * sizeof(double));
for (int i = 0; i < size; i++)
    for (int j = 0; j < size; j++)
        es[i*size + j + 1] = 0;
int max_num = 0;
int max_temp;
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        for (int k = 0; k < S_num[j]; k++) {
            if (X[i] == S[sum_num(S_num, j) + k]) {
                es[i*size + j + 1] = 1;
            }
        }
    }
}

for (int i = 0; i < size; i++) {
    max_temp = 0;
    for (int j = 0; j < size; j++) {
        max_temp += es[i*size + j + 1];
    }
    if (max_temp > max_num) max_num = max_temp;
}
cout << "f:" << max_num << endl;

clock_t begin, end;
begin = clock();
glp_prob *lp;

```

```

lp = glp_create_prob();
glp_set_prob_name(lp, "sample");
glp_set_obj_dir(lp, GLP_MIN); // maximization

glp_add_rows(lp, size);
for (int i = 1; i <= size; i++) {
    glp_set_row_bnds(lp, i, GLP_LO, 1.0, 0.0);
}

glp_add_cols(lp, size);
for (int i = 1; i <= size; i++) {
    if (whether_precise)
        glp_set_col_kind(lp, i, GLP_IV);
    glp_set_col_bnds(lp, i, GLP_UP, 0.0, 1.0);
    glp_set_col_bnds(lp, i, GLP_LO, 0.0, 0.0);
    glp_set_obj_coef(lp, i, 1.0);
}

int *ia = (int *)malloc((size * size + 1) * sizeof(int));
int *ja = (int *)malloc((size * size + 1) * sizeof(int));
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        ia[i*size + j + 1] = i + 1;
        ja[i*size + j + 1] = j + 1;
    }
}

glp_load_matrix(lp, size*size, ia, ja, es); // 将约束系数矩阵导入

glp_iocp param;
glp_init_iocp(&param);
param.presolve = GLP_ON;
param.msg_lev = GLP_MSG_ON;
glp_intopt(lp, &param);

vector<int> C;
cout << glp_mip_obj_val(lp) << endl;
for (int i = 1; i <= size; i++) {
    if (glp_mip_col_val(lp, i) != 0) {
        if (glp_mip_col_val(lp, i) >= (1.0 / max_num))
            C.push_back(i-1);
    }
}

end = clock();
glp_delete_prob(lp);

```

(4) 精确算法

在第 3.3 节线性规划问题的基础上，将其改造为整数规划问题，限定 x_s 的值为 0 或 1，可得到精确解，如下所示：

$$\begin{aligned} & \text{minimize } \sum_{S \in F} C(S)x_S \\ & \text{s.t. } \sum_{S: e \in S} x_S \geq 1, e \in X \\ & \quad x_S = 0 \text{ or } 1, S \in F \end{aligned}$$

对于 glpk 库来说，只需在参数设定中添加“GLP_IV”的限定即可，将参数限定为 int 型，完整代码为 `glp_set_col_kind(lp, i, GLP_IV)`。

(5) 算法准确性验证

为了验证算法准确性，我们将最后求得的子集作并运算，检测其是否等于 X ，该部分代码如清单 3 所示。

清单 3 算法准确性检测代码

```
bool check_right(vector<int> C, vector<int> X, vector<int> S_num,
vector<int> S) {
    int *flag = (int *)malloc(X.size() * sizeof(int));
    for (int i = 0; i < X.size(); i++) flag[i] = 0;
    for (int i = 0; i < C.size(); i++) {
        for (int j = 0; j < S_num[C[i]]; j++) {
            for (int k = 0; k < X.size(); k++) {
                if (X[k] == S[sum_num(S_num, C[i]) + j])
                    flag[k] = 1;
            }
        }
    }
    for (int i = 0; i < X.size(); i++) {
        if (flag[i] == 0) {
            free(flag);
            return false;
        }
    }
    return true;
}
```

4. 实验结果与分析

(1) 实验参数配置

本实验将 exe 程序和 config 配置文件抽离开来，能够实现较好的交互，对配置文件的说明如清单 4 所示。可以通过 method 选择生成数据集、贪心求解和线性规划求解三种方法。我们可以在 [2] 的功能模块下处改变 whether_precies 选择是否求得精确解。其他具体的参数也可以在功能区下面配置。

清单 4 config.ini

```
[method]
; 0: generate node
; 1: greedy search
; 2: linear solution
method = 0
```

```

[0]
;生成随机点
size=500
data_path = L:/why_workspace/cpl/txt/500.tf
max_node = 20

[1]
size=300
data_path = L:/why_workspace/cpl/txt/300.tf
whether_show_process = 1

[2]
size=300
data_path = L:/why_workspace/cpl/txt/300.tf
whether_precise = 1

```

(2) 实验过程展示

在数据集元素个数等于 500 时，我们采用贪心求解方法，得到了图 2 所示结果。可以看到，每一步贪心搜索所新增的元素都有打印。最终需要的子集个数为 69 个，“C:”处输出的是 S 中的第多少个子集。最后我们进行了正确性检验，覆盖了原集合 X，算法正确。

```

第62次新增元素:3839 173 4623 2012 4156 4160 1260 3438
第62次剩余元素:7
第63次新增元素:1381 373 3485
第63次剩余元素:6
第64次新增元素:3293 4891 1326 2059
第64次剩余元素:5
第65次新增元素:640 2015
第65次剩余元素:4
第66次新增元素:2598 1252 1400
第66次剩余元素:3
第67次新增元素:3070 3569 3801 2311
第67次剩余元素:2
第68次新增元素:4880 3764 2641 4524 233 4736 932 3857 4204 1245 3601 4469 632 306
8
第68次剩余元素:1
第69次新增元素:414 1935 892
第69次剩余元素:0
总所需子集数:69
C:0 11 53 34 52 69 73 80 3 32 59 5 13 21 63 65 72 79 14 44 24 38 49 18 41 81 26
29 10 45 1 12 48 54 75 20 22 27 35 37 341 56 68 332 7 25 42 43 58 64 77 193 31 3
6 40 55 62 78 278 16 30 39 46 47 57 60 61 66 67
是否覆盖了原集合X: 1
Total time:670
Please click enter to exit.

```

图 2 贪心求解结果展示

采用舍入法近似求解的结果如图 3 所示，所需子集个数为 151 个，大于贪心法的数量。

```

Loading data.
Loading finished.
f:14
    0: obj = 4.400000000e+001 infeas = 4.490e+002 <0>
    500: obj = 6.484995892e+001 infeas = 3.930e+001 <0>
*   775: obj = 8.163838965e+001 infeas = 7.016e-016 <0>
*  1000: obj = 6.505430645e+001 infeas = 0.000e+000 <0>
*  1295: obj = 6.11186809e+001 infeas = 9.171e-029 <0>
+  1295: mip = not found yet >= -inf <1; 0>
+  1295: >>>> 6.11186809e+001 >= 6.11186809e+001 0.0% <1; 0>
+  1295: mip = 6.11186809e+001 >= tree is empty 0.0% <0; 1>
61.1119
总所需子集数:151
C:0 1 3 5 7 8 10 11 12 13 14 18 20 21 22 24 25 26 27 28 29 31 32 34 35 37 38 39
40 41 42 43 44 45 47 48 49 52 53 54 56 57 58 59 60 61 62 63 64 65 66 67 68 69 71
72 73 75 76 77 79 80 81 96 112 113 119 122 123 125 130 131 133 135 140 142 145
152 153 154 155 168 171 172 176 179 188 192 213 220 223 226 228 230 232 234 245
251 253 254 257 276 278 280 296 305 323 327 332 333 334 335 341 342 343 348 349
355 361 363 365 368 372 374 375 383 386 391 397 400 402 409 418 422 423 424 436
437 438 440 446 447 455 457 458 465 466 470 474 476 479
是否覆盖了原集合X: 1
Total time:79
Please click enter to exit.

```

图3 舍入法结果展示

最后是对精确解的求解展示，如图4所示，精确解为64个。

```

Loading data.
Loading finished.
f:14
    0: obj = 4.400000000e+001 infeas = 4.490e+002 <0>
    500: obj = 6.484995892e+001 infeas = 3.930e+001 <0>
*   775: obj = 8.163838965e+001 infeas = 7.016e-016 <0>
*  1000: obj = 6.505430645e+001 infeas = 0.000e+000 <0>
*  1295: obj = 6.11186809e+001 infeas = 9.171e-029 <0>
+  1295: mip = not found yet >= -inf <1; 0>
+  3316: >>>> 7.100000000e+001 >= 6.200000000e+001 12.7% <46; 0>
+  4774: >>>> 6.700000000e+001 >= 6.200000000e+001 7.5% <75; 1>
+ 16991: >>>> 6.400000000e+001 >= 6.200000000e+001 3.1% <293; 62>
+ 42621: mip = 6.400000000e+001 >= tree is empty 0.0% <0; 919>
64
总所需子集数:64
C:0 3 5 7 10 11 12 13 18 20 22 24 25 26 27 29 31 32 34 35 37 38 39 40 41 42 43 4
4 45 48 49 52 53 54 56 58 59 61 63 64 65 68 69 72 73 75 76 77 79 80 81 131 133 1
55 188 228 232 278 333 334 337 342 400 470
是否覆盖了原集合X: 1
Total time:4579
Please click enter to exit.

```

图4 精确结果展示

(3) 性能对比

我们对数据集进行了重新生成以加速精确解的求解，采用的方式是限定S中元素的数量，即f。我们进行了3次重复实验，其中一次的实验结果如表1所示。

表 1 近似比分析对比

100	f	time(ms)	子集个数	理论近似比	实际近似比
贪心	20	4	14	4.62	1.08
舍入	13	4	20	13	1.54
精确解		5	13		
200	f	time(ms)	子集个数	理论近似比	实际近似比
贪心	20	17	23	5.3	1
舍入	13	14	37	13	1.61
精确解		62	23		
300	f	time(ms)	子集个数	理论近似比	实际近似比
贪心	20	55	44	5.71	1.07
舍入	16	26	89	16	2.17
精确解		192	41		
400	f	time(ms)	子集个数	理论近似比	实际近似比
贪心	20	109	57	5.99	1.08
舍入	15	19	108	15	2.04
精确解		112	53		
500	f	time(ms)	子集个数	理论近似比	实际近似比
贪心	20	208	69	6.22	1.08
舍入	14	77	151	14	2.36
精确解		4556	64		
1000	f	time(ms)	子集个数	理论近似比	实际近似比
贪心	20	1465	136	6.91	1.02
舍入	17	385	297	17	2.22
精确解		1024911	134		

我们分别对贪心法和舍入法 f 值进行了变换，得到图 5、图 6 所示结果。

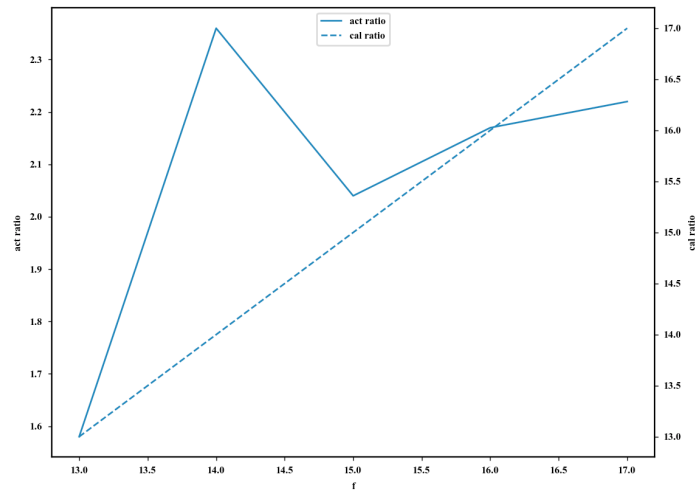


图 5 舍入法近似比和理论近似比随 f 的关系

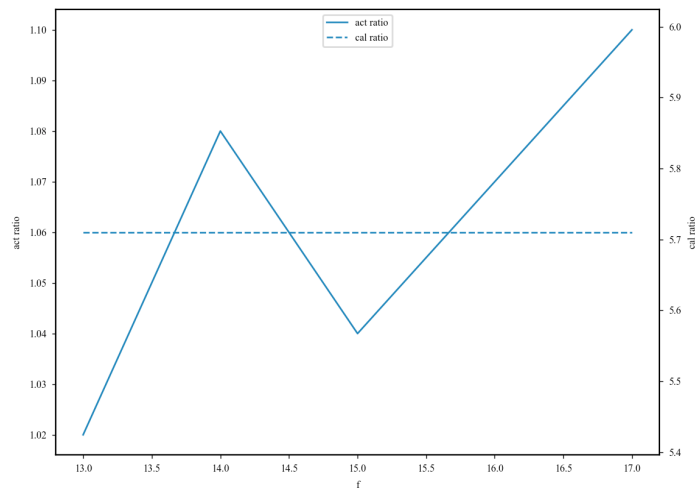


图 6 贪心法近似比和理论近似比随 f 的关系(元素大小固定为 300)

(4) 总结

贪心算法的理论近似比为 $\ln(|X|+1)$ ，舍入法的理论近似比为 f 。随着 f 的增加，舍入法和精确解的差距越来越大，这一点从图 5 和图 6 可以看出：当 f 从 13 增加到 17 时，贪心法的实际近似比几乎未变，舍入法从 1.6 增加到了 2.3 左右。综上，分析认为贪心法比舍入法更加适合那些 f 较大的数据集。

5. 实验心得

这次实验与实验一和实验二侧重点不同，之前两个实验注重算法复现，这次更加注重算法实践。对于舍入法，老师并没有要求从底层解线性方程，而是将问题转化为 glpk 库支持的格式，借助第三方 API 解决问题。因此这次实验主要的时间是在配置环境和学习 glpk 的使用。实践过程中，自己想到了一个比较巧妙求解精确解的方法，就是将松弛线性规划限定为 0-1 整数规划问题，这样就省力不少。