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， $X=\cup_{S\in F} S$
- 输出:$C\subseteq F$，满足 $X=\cup_{S\in F} S$ 且 C 是满足 $X=\cup_{S\in F} S$ 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*sizeof(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*sizeof(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。

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

1: U←X
2: C←θ
3: **While** U≠θ **Do**
4: 贪心选择能覆盖最多 U 元素的子集 S
5: U←U-S
6: C←C∪{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， $X=\cup_{S\in F} S$，|X|=|F|

输出 : C⊆F，满足 $X=\cup_{S\in F} S$ 且 C 是满足 $X=\cup_{S\in F} S$ 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∈F **Do**

4: If $x_s \geq 1/f$ then C←C∪{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。

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

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:

$$\text{minimize} \sum_{S \in F} C(S) x_s$$
$$\text{s.t.} \sum_{S:e \in S} x_s \geqslant 1, \ e \in X$$
$$x_s = 0 \text{ or } 1, \ S \in F$$

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。

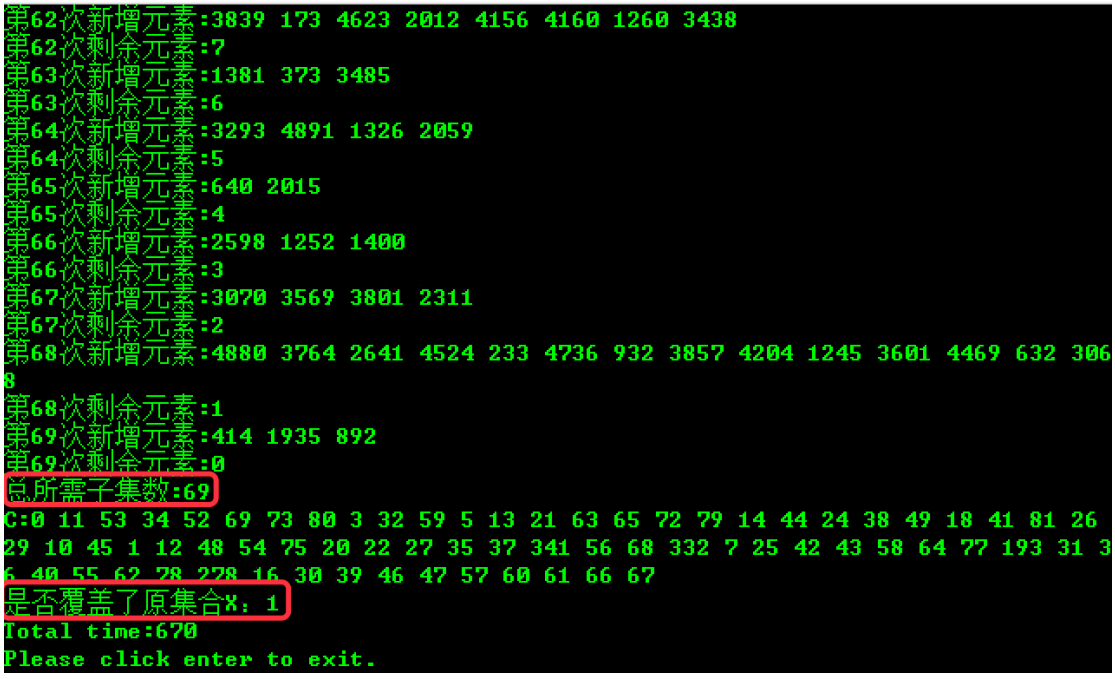

图 2 贪心求解结果展示

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。

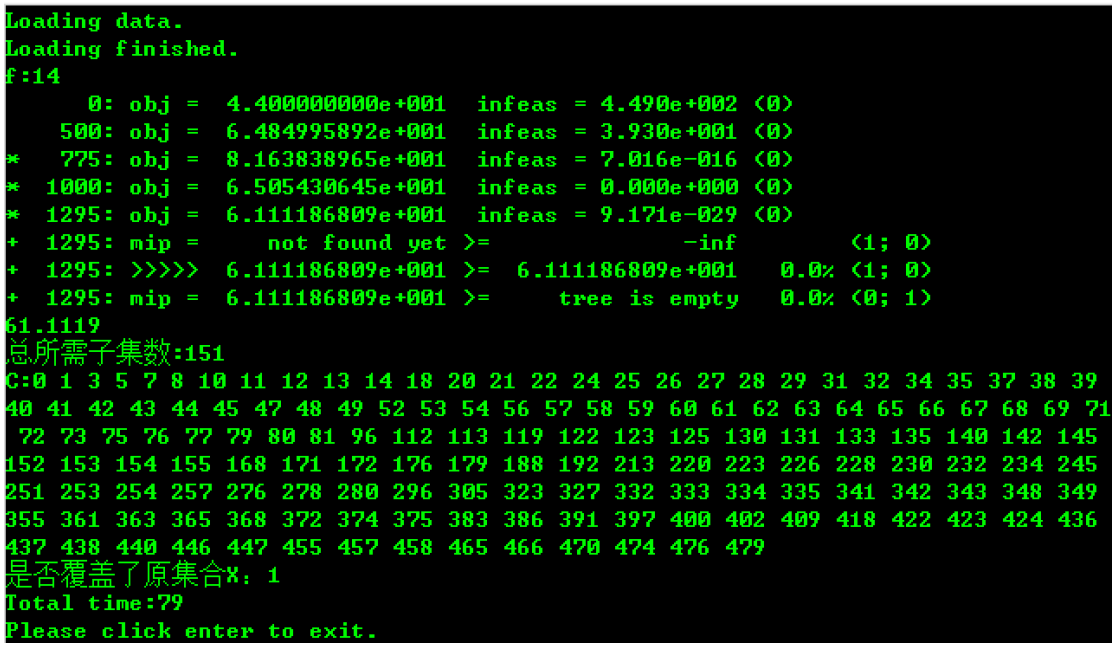

图 3 舍入法结果展示

最后是对精确解的求解展示，如图 4 所示，精确解为 64 个。

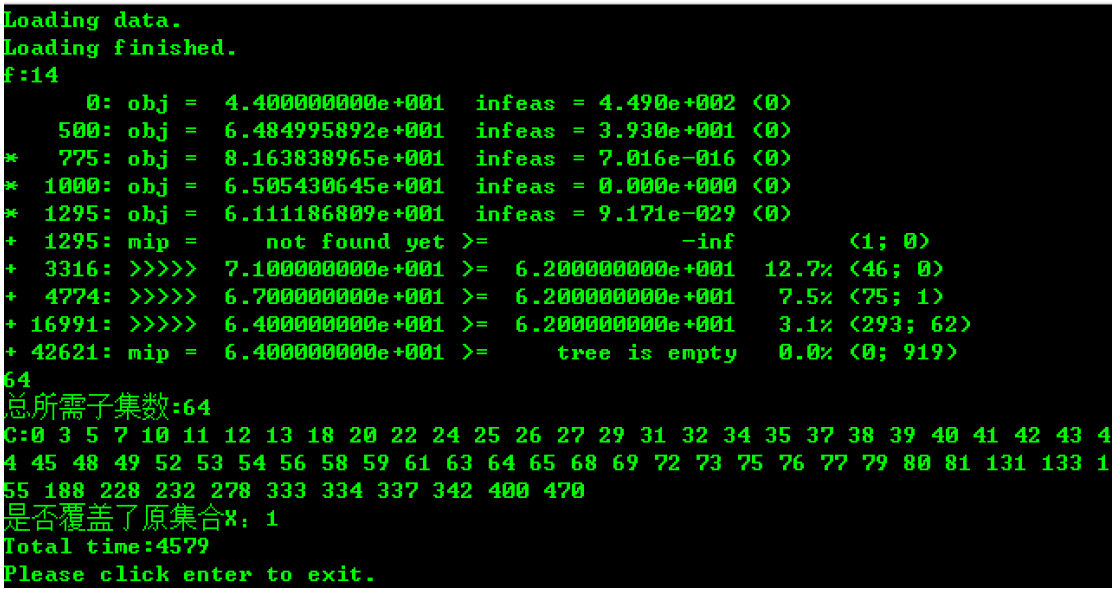

图 4 精确结果展示

(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。

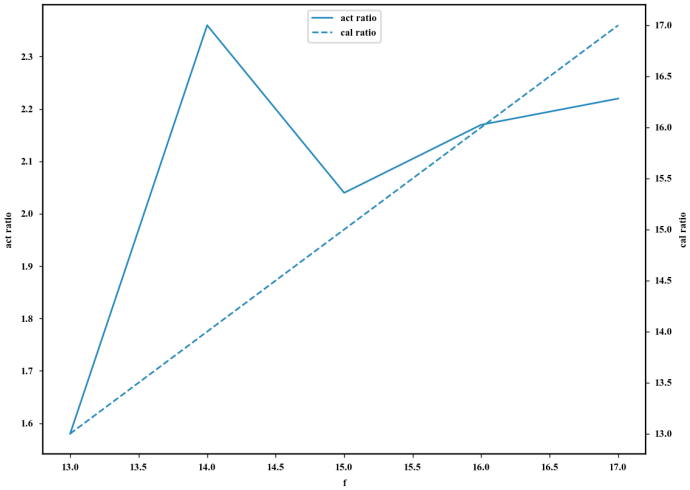

图 5 舍入法近似比和理论近似比随 f 的关系

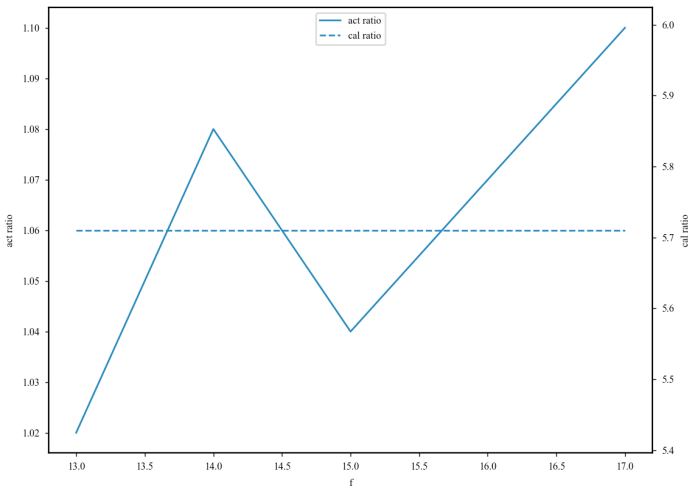

图 6 贪心法近似比和理论近似比随 f 的关系(元素大小固定为 300)

(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。