中国科学院大学
University of Chinese Academy of Sciences

计算机算法设计与分析

083500M01001H

Chap 9&10&11 课程作业

2022 年 12 月 1 号

Professor: 刘玉贵

学生: 周胤昌
学号: 202228018670052
学院: 网络空间安全学院
所属专业: 网络空间安全
方向: 安全协议理论与技术

Problem 1

设集合 $S=\{x_1,x_2,\cdots,x_n\}$, x_i 出现的概率为 $0<p_i<1$, $\sum_{i=1}^{n} p_i=1$. 试设计一个算法, 按照 S 的概率分布对 S 的元素进行随机选择 (一个).

Solution: 可以设计出以下的随机算法, 代码有 C++ 版本、也有 python 版本 (对应的输出结果紧跟代码之后):

```cpp
#include <bits/stdc++.h> //万能头文件
using namespace std;

int random_pick(vector<pair<int, double>> nums_probs) {
    int res;
    random_device rd; //定义一个非确定性的真随机生成器
    uniform_real_distribution<double> U(0, 1); //随机数分布对象
    double x = U(rd); //在 [0,1) 内随机选取实数作为 x
    double cumu_prob = 0.0; //初始化累积概率
    for(auto num_prob : nums_probs) {
        cumu_prob += num_prob.second;
        if(x < cumu_prob) {
            res = num_prob.first;
            break;
        }
    }
    return res;
}

int main() {
    vector<pair<int, double>> nums_probs = {
        {1, 0.2}, {2, 0.1}, {3, 0.6}, {4, 0.1}
    };
    cout << " 自写算法的抽取结果: " << random_pick(nums_probs) << endl;
    random_device rd;
    discrete_distribution<> d({0.2, 0.1, 0.6, 0.1});
    cout << " 调用 C++11 中算法的抽取结果: " << d(rd) + 1 << endl;
}
```

上述 C++ 代码对应的输出结果为:

```
开始运行...
自写算法的抽取结果: 3
调用 C++11中算法的抽取结果: 3
运行结束.
```

```python
import random
import numpy as np

def random_pick(some_list, probabilities):
    x = random.uniform(0, 1)
    cumulative_probability = 0.0
    for item, item_probability in zip(some_list, probabilities):
        cumulative_probability += item_probability
        if x < cumulative_probability: break
    return item

if __name__ == '__main__':
    some_list = [1, 2, 3, 4]
    probabilities = [0.2, 0.1, 0.6, 0.1]
    print(" 自写算法的抽取结果: ", random_pick(some_list, probabilities))
    print(" 调用 numpy 中算法的抽样结果: ", np.random.choice(some_list, 1, probabilities))
```

上述 python 代码对应的输出结果为:

```
开始运行...
自写算法的抽取结果: 3
调用 numpy 中算法的抽样结果: [3]
运行结束.
```

显然该随机算法的复杂度主要取决于 for 循环, 因此上述算法的时间复杂度为 $O(n)$, 其中 n 为集合 S 中的元素个数. 而空间复杂度显然为 $O(1)$.

Problem 2

设计概率算法, 求解 $365!/\left(340!\cdot 365^{25}\right)$.

Solution: 该问题的数是概率论中著名的生日问题的解答. 在 k 个人中, 至少 2 个人有相同生日的概率为

$$1-\frac{365!}{(365-k)!\cdot 365^k}$$

根据斯特林公式以及近似公式:

$$n!\sim\sqrt{2\pi n}\left(\frac{n}{e}\right)^n (n\to\infty),\quad \ln(1+x)\sim x-\frac{x^2}{2}(x\to 0)$$

因此有:

$$\begin{aligned}
\frac{n!}{(n-k)!\cdot n^k} &\sim \frac{\sqrt{2\pi n}}{\sqrt{2\pi(n-k)}}\cdot\frac{\left(\frac{n}{e}\right)^n}{\left(\frac{n-k}{e}\right)^{n-k}\cdot n^k}\\
&\sim\left(\frac{n}{n-k}\right)^{n-k}\cdot e^{-k}=\exp\left[(n-k)\ln\left(1+\frac{k}{n-k}\right)-k\right]\\
&\sim\exp\left\{(n-k)\left[\frac{k}{n-k}-\frac{\frac{k^2}{(n-k)^2}}{2}\right]-k\right\}\\
&\sim\exp\left\{-\frac{k^2}{2(n-k)}\right\}\sim\exp\left\{-\frac{k^2}{2n}\right\}
\end{aligned}$$

因此可得

$$\left.\frac{n!}{(n-k)!\cdot n^k}\right|_{n=365}^{k=25} = \frac{365!}{340!\cdot 365^{25}} \approx \exp\left\{-\frac{25^2}{730}\right\} = 0.424788$$

Problem 3

设计一个 Las Vegas 随机算法, 求解电路板布线问题. 将该算法与分支限界算法结合, 观察求解效率.

Solution: 该算法的设计核心是: **采用随机放置位置策略并结合分支限界算法**. 算法的 C++ 代码如下所示:

```
#include <bits/stdc++.h> //万能头文件, 刷题时可以用, 大型项目千万不能用
using namespace std;

//表示方格位置上的结构体
struct position{
    int row;
    int col;
};

//分支限界算法
bool FindPath(
    position start, position finish, int &PathLen,
    position *&path, int **grid, int m, int n
) { //找到最短布线路径, 若找得到则返回 true, 否则返回 false
    //起点终点重合则不用布线
    if((start.row == finish.row) && (start.col == finish.col)) {
        PathLen = 0;
        return true;
    }

    //设置方向移动坐标值: 东南西北
    position offset[4];
    offset[0].row = 0;
    offset[0].col = 1; //右移
    offset[1].row = 1;
    offset[1].col = 0; //下移
    offset[2].row = 0;
    offset[2].col = -1; //左移
    offset[3].row = -1;
    offset[3].col = 0; //上移

    int NumNeighBlo = 4; //相邻的方格数
    position here, nbr;
    here.row = start.row; //设置当前方格, 即搜索单位
    here.col = start.col;
    //由于 0 和 1 用于表示方格的开放和封闭, 故距离: 2-0 3-1
    grid[start.row][start.col] = 0; //-2 表示强, -1 表示可行, -3 表示不能当作路线
    //队列式搜索, 标记可达相邻方格
    queue<position> q_FindPath;
```

```
    do {
        int num = 0; //方格未标记个数
        position selectPosition[5]; //保存选择位置
        for(int i = 0; i < NumNeighBlo; i++) {
            //到达四个方向
            nbr.row = here.row + offset[i].row;
            nbr.col = here.col + offset[i].col;
            if(grid[nbr.row][nbr.col] == -1) { //该方格未标记
                grid[nbr.row][nbr.col] = grid[here.row][here.col] + 1;
                if((nbr.row == finish.row) && (nbr.col == finish.col)) {
                    break;
                }
                selectPosition[num].row = nbr.row;
                selectPosition[num].col = nbr.col;
                num++;
            }
        }
        if(num > 0) { //如果标记, 则在这么多个未标记个数中随机选择一个位置 (本算法核心)
            q_FindPath.push(selectPosition[rand()%(num)]); //随机选一个入队
        }
        if((nbr.row == finish.row) && (nbr.col == finish.col)) {
            break; //是否到达目标位置 finish
        }
        //判断活结点队列是否为空
        if(q_FindPath.empty() == true) return false; // 无解
        //访问队首元素出队
        here = q_FindPath.front();
        q_FindPath.pop();
    } while (true);

    //构造最短布线路径
    PathLen = grid[finish.row][finish.col];
    path = new position[PathLen]; //路径
    //从目标位置 finish 开始向起始位置回溯
    here = finish;
    for(int j = PathLen - 1; j >= 0; j--) {
        path[j] = here;
        //找前驱位置
        for(int i = 0; i <= NumNeighBlo; i++) {
            nbr.row = here.row + offset[i].row;
            nbr.col = here.col + offset[i].col;
            if(grid[nbr.row][nbr.col] == j) {//距离 +2 正好是前驱位置
                break;
            }
        }
        here = nbr;
    }
    return true;
}
```

```
int main() {
    cout << "---------分支限界算法之布线问题---------" << endl;
    int path_len, path_len1, m, n;
    position *path, *path1, start, finish, start1, finish1;
    cout << " 在一个 m*n 的棋盘上, 请分别输入 m 和 n, 代表行数和列数, 然后输入回车" << endl;
    cin >> m >> n;
    //创建棋盘格
    int **grid = new int * [m + 2], **grid1 = new int * [m + 2];
    for(int i = 0; i < m + 2; i++) {
        grid[i] = new int[n + 2];
        grid1[i] = new int[n + 2];
    }
    //初始化棋盘
    for(int i = 1; i <= m; i++) {
        for(int j = 1; j <= n; j++) {
            grid[i][j] = -1;
        }
    }
    //设置方格阵列的围墙
    for(int i = 0; i <= n + 1; i++) {
        grid[0][i] = grid[m + 1][i] = -2; //上下的围墙
    }
    for(int i = 0; i <= m + 1; i++) {
        grid[i][0] = grid[i][n + 1] = -2; //左右的围墙
    }
    cout << " 初始化棋盘格和加围墙" << endl;
    cout << "------------------" << endl;
    for(int i = 0; i < m + 2; i++) {
        for(int j = 0; j < n + 2; j++) {
            cout << grid[i][j] << " ";
        }
        cout << endl;
    }
    cout << "------------------" << endl;
    cout << " 请输入已经占据的位置 (行坐标, 列坐标), 代表此位置不能布线" << endl;
    cout << " 例如输入 2 2(然后输入回车), 表示坐标 (2,2) 不能布线;" <<
    " 当输入坐标为 0 0(然后输入回车) 表示结束输入" << endl;
    //添加已经布线的棋盘格
    while(true) {
        int ci, cj;
        cin >> ci >> cj;
        if(ci > m || cj > n) {
            cout << " 输入非法!";
            cout << " 行坐标 <" << m << ", 列坐标 <" << n << " 当输入的坐标为 0,0 时结束输入"
↪   << endl;
            continue;
        } else if (ci == 0 || cj == 0) {
            break;
        } else {
            grid[ci][cj] = -3;
        }
    }
```

```cpp
//布线前的棋盘格
cout << " 布线前的棋盘格" << endl;
cout << "------------------" << endl;
for(int i = 0; i < m + 2; i++) {
    for(int j = 0; j < n + 2; j++) {
        cout << grid[i][j] << " ";
    }
    cout << endl;
}
cout << "------------------" << endl;
cout << " 请输入起点位置坐标" << endl;
cin >> start.row >> start.col;
cout << " 请输入终点位置坐标" << endl;
cin >> finish.row >> finish.col;
clock_t starttime, endtime;
starttime = clock(); //程序开始时间
srand((unsigned) time (NULL)); //初始化时间种子, 是随机选择的关键
int time = 0; //为假设运行次数
start1 = start, finish1 = finish, path_len1 = path_len, path1 = NULL; //初始值拷贝
for(int i = 0; i < m + 2; i++) {
    for(int j = 0; j < n + 2; j++) {
        grid1[i][j] = grid[i][j];
    }
}
bool result = FindPath(start1, finish1, path_len1, path1, grid1, m, n);
while(result == 0 && time < 1000) { //尝试次数最多为 1000 次
    //初始值拷贝
    start1 = start, finish1 = finish, path_len1 = path_len, path1 = NULL;
    for(int i = 0; i < m + 2; i++) {
        for(int j = 0; j < n + 2; j++) {
            grid1[i][j] = grid[i][j];
        }
    }
    time++;
    cout << endl;
    cout << " 没有找到路线, 第" << time << " 次尝试" << endl;
    result = FindPath(start1, finish1, path_len1, path1, grid1, m, n);
}
endtime = clock(); //程序结束时间

if(result == 1) {
    cout << "------------------" << endl;
    cout << "$ 代表围墙" << endl;
    cout << "# 代表已经占据的点" << endl;
    cout << "* 代表布线路线" << endl;
    cout << "= 代表还没有布线的点" << endl;
    cout << "------------------" << endl;
    for(int i = 0; i <= m + 1; i++) {
        for(int j = 0; j <= n + 1; j++) {
            if(grid1[i][j] == -2) cout << "$ ";
            else if(grid1[i][j] == -3) cout << "# ";
```

```
                else {
                    int r;
                    for(r = 0; r < path_len1; r++) {
                        if(i == path1[r].row && j == path1[r].col) {
                            cout << "* ";
                            break;
                        }
                        if(i == start1.row && j == start1.col) {
                            cout << "* ";
                            break;
                        }
                    }
                    if(r == path_len1) cout << "= ";
                }
            }
            cout << endl;
        }
        cout << "-------------------" << endl;
        cout << " 路径坐标和长度" << endl;
        cout << endl;
        cout << "(" << start1.row << "," << start1.col << ")" << " ";
        for(int i = 0; i < path_len1; i++) {
            cout << "(" << path1[i].row << "," << path1[i].col << ")" << " ";
        }
        cout << endl;
        cout << endl;
        cout << " 路径长度: " << path_len1 + 1 << endl;
        cout << endl;
        time++;
        cout << " 布线完毕, 共查找" << time << " 次" << endl;
        cout << " 算法运行时间为: " << (endtime - starttime) << "ms" << endl;
    } else {
        cout << endl;
        cout << " 经过多次尝试, 依然没有找到路线" << endl;
    }
    return 0;
}
```

上述代码的关键之处在于:

- P5 页的第 13 行代码, 这里是当前点相邻四个点是否可以放置, 如果可以放置用 selectPostion 保存下来, 并用 num 记录有多少个位置可以放置;
- P5 页的第 19 行代码, 这里利用上面保存的可以放置的点, 然后**随机选取其中一个加入队列**, 这就是 Las Vegas 算法的精髓;
- P7 页的第 17 行代码, 作用是初始化时间种子, 是伪随机生成器的关键, 即是随机选择的关键.

结果分析:

- 测试样例 1: 3×3 棋盘, 代码的交互输出过程如下:

```
开始运行...
---------分支限界算法之布线问题---------
在一个 m*n 的棋盘上，请分别输入 m 和 n，代表行数和列数，然后输入回车
3 3
初始化棋盘格和加围墙
------------------
-2 -2 -2 -2 -2
-2 -1 -1 -1 -2
-2 -1 -1 -1 -2
-2 -1 -1 -1 -2
-2 -2 -2 -2 -2
------------------
请输入已经占据的位置(行坐标，列坐标)，代表此位置不能布线
例如输入 2 2(然后输入回车)，表示坐标(2,2)不能布线;当输入坐标为 0 0(然后输入回车)表示结束输入
2 1
2 3
3 3
0 0
布线前的棋盘格
------------------
-2 -2 -2 -2 -2
-2 -1 -1 -1 -2
-2 -3 -1 -3 -2
-2 -1 -1 -3 -2
-2 -2 -2 -2 -2
------------------
请输入起点位置坐标
1 1
请输入终点位置坐标
3 1

没有找到路线，第 1 次尝试

没有找到路线，第 2 次尝试
------------------
$ 代表围墙
# 代表已经占据的点
* 代表布线路线
= 代表还没有布线的点
------------------
$ $ $ $ $
$ * * = $
$ # * # $
$ * * # $
$ $ $ $ $
------------------
路径坐标和长度
(1,1) (1,2) (2,2) (3,2) (3,1)
路径长度: 5
布线完毕，共查找 3 次
算法运行时间为: 39ms
运行结束.
```

- 测试样例 2: 5×5 棋盘, 代码的交互输出过程如下:

```
---------分支限界算法之布线问题---------
在一个 m*n 的棋盘上, 请分别输入 m 和 n, 代表行数和列数, 然后输入回车
5 5
请输入已经占据的位置(行坐标, 列坐标), 代表此位置不能布线
例如输入 2 2(然后输入回车), 表示坐标(2,2)不能布线;当输入坐标为 0 0(然后输入回车)表示结束输
↪ 入
3 1
3 2
3 4
3 5
4 5
0 0
布线前的棋盘格
------------------
-2 -2 -2 -2 -2 -2 -2
-2 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -2
-2 -3 -3 -1 -3 -3 -2
-2 -1 -1 -1 -1 -3 -2
-2 -1 -1 -1 -1 -1 -2
-2 -2 -2 -2 -2 -2 -2
------------------
请输入起点位置坐标
1 1
请输入终点位置坐标
5 2
没有找到路线, 第 1 次尝试
没有找到路线, 第 2 次尝试
没有找到路线, 第 3 次尝试
------------------
$ 代表围墙
# 代表已经占据的点
* 代表布线路线
= 代表还没有布线的点
------------------
$ $ $ $ $ $ $
$ * = = = = $
$ * * * = = $
$ # # * # # $
$ = = * = # $
$ = * * = = $
$ $ $ $ $ $ $
------------------
路径坐标和长度
(1,1) (2,1) (2,2) (2,3) (3,3) (4,3) (5,3) (5,2)
路径长度: 8
布线完毕, 共查找 4 次
算法运行时间为: 61ms
```

- 测试样例 2: 10×10 棋盘, 代码的交互输出过程如下:

```
---------分支限界算法之布线问题---------
在一个 m*n 的棋盘上, 请分别输入 m 和 n, 代表行数和列数, 然后输入回车
10 10
布线前的棋盘格
------------------
-2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -3 -3 -3 -3 -1 -1 -3 -3 -3 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2
------------------
请输入起点位置坐标
1 1
请输入终点位置坐标
9 9

没有找到路线, 第 1 次尝试
没有找到路线, 第 2 次尝试
没有找到路线, 第 3 次尝试
没有找到路线, 第 4 次尝试
------------------
$ 代表围墙
# 代表已经占据的点
* 代表布线路线
= 代表还没有布线的点
------------------
$ $ $ $ $ $ $ $ $ $ $ $
$ * = = = = = = = = = $
$ * * * = = = = = = = $
$ = = * * = = = = = = $
$ = = = * * = = = = = $
$ # # # # * = # # # = $
$ = = = = * = = = = = $
$ = = = = * * * = = = $
$ = = = = = = * = = = $
$ = = = = = = * * * = $
$ = = = = = = = = = = $
$ $ $ $ $ $ $ $ $ $ $ $
------------------
路径坐标和长度
(1,1) (2,1) (2,2) (2,3) (3,3) (3,4) (4,4) (4,5) (5,5) (6,5) (7,5) (7,6) (7,7) (8,7)
↪ (9,7) (9,8) (9,9)
路径长度: 17
布线完毕, 共查找 5 次
算法运行时间为: 73ms
```

由此可见, 结合随机化和分支限界的 Las Vegas 算法的求解效率还是相当不错的.

Problem 4

判断正误:

- Las Vegas 算法不会得到不正确的解. ( )
- Monte Carlo 算法不会得到不正确的解. ( )
- Las Vegas 算法总能求得一个解. ( )
- Monte Carlo 算法总能求得一个解. ( )

Solution:

- 正确, 拉斯维加斯算法不会得到不正确的解. 一旦用拉斯维加斯算法找到一个解, 这个解就一定是正确解. 但有时用拉斯维加斯算法找不到解.
- 错误, Monte Carlo 算法每次都能得到问题的解, 但不保证所得解的正确性. 请注意, 可以在 Monte Carlo 算法给出的解上加一个验证算法, 如果正确就得到解, 如果错误就不能生成问题的解, 这样 Monte Carlo 算法便转化为了 Las Vegas 算法.
- 错误, Las Vegas 算法并不能保证每次都能得到一个解, 但是如果一旦某一次得到解, 那么就一定是正确的.
- 正确, Monte Carlo 算法每次运行都能给出一个解, 但正确性就不能保证了.

Problem 5

设 Las Vegas 算法获得解的概率为 $p(x) \geq \delta, 0 < \delta < 1$, 则调用 k 次算法后, 获得解的概率为:_____.
Solution: 不妨求一下调用 k 次算法后, 求解失败 (即 k 次调用都求解失败) 的概率:

$$P\left(\text{失败}\right) = (1 - p\left(x\right))^k \leq (1 - \delta)^k \Rightarrow P\left(\text{成功}\right) = 1 - P\left(\text{失败}\right) \geq 1 - (1 - \delta)^k$$

即获得解的概率至少为 $1 - (1 - \delta)^k \to 1(\text{当} k \to \infty)$.

Problem 6

对于判定问题 Π 的 Monte Carlo 算法, 当返回 false(true) 时解总是正确的, 但当返回 true(false) 时解可能有错误, 该算法是 ______.

(A) .偏真的*Monte Carlo*算法 (B) .偏假的*Monte Carlo*算法
(C) .一致的*Monte Carlo*算法 (D) .不一致的*Monte Carlo*算法

Solution: 答案选 B, 只要将偏真的 Monte Carlo 算法的定义中的 true/false 互换即可得到偏真的 Monte Carlo 算法的定义.

Problem 7

判断正误:

- 一般情况下, 无法有效判定 Las Vegas 算法所得解是否正确. ( )
- 一般情况下, 无法有效判定 Monte Carlo 算法所得解是否正确. ( )
- 虽然在某些步骤引入随机选择, 但 Sherwood 算法总能求得问题的一个解, 且所求得的解总是正确的. ( )
- 虽然在某些步骤引入随机选择, 但 Sherwood 算法总能求得问题的一个解, 但一般情况下, 无法有效判定所求得的解是否正确. ( )

Solution:

- 错误, Las Vegas 算法并不能保证每次都能得到解, 但是如果一旦某一次得到解, 那么就一定是正确的.
- 错误, 虽然 Monte Carlo 算法每次运行都能给出一个解, 可能是错的也可能是对的, 但是可以通过检验解来有效判定其正确性. 判定解的正确性跟算法本身没有多大关系, 只要代进去验证即可. 特殊点在于, 只要 Las Vegas 算法求得解了, 那么就一定是正确的, 就不用再浪费时间来判定了; 但是对于 Monte Carlo 算法的所得解, 必须要进行正确性检验.
- 正确, Sherwood 算法总能求得问题的一个解, 且所求得的解总是正确的.
- 错误.

Problem 8

装箱问题: 任给 n 件物品, 物品 j 的重量为 $w_j, 1 \le j \le n$, 限制每只箱子装入物品的总重量不超过 B, 其中 B 和 w_j 都是正整数, 且 $w_j \le B, 1 \le j \le n$. 要求用最少的箱子装入所有物品, 怎么装法?

考虑下述近似算法-**首次适合算法 (FF)**: 按照输入顺序装物品, 对每一件物品, 依次检查每一只箱子, 只要能装得下就把它装入, 只有在所有已经打开的箱子都装不下这件物品时, 才打开一个新箱子. 证明: **FF 算法**是 2-近似的, 即任给实例 I, 都有

$$\mathbf{FF}(I) < 2\mathrm{OPT}(I)$$

Solution: 当 $\mathbf{FF}(I) = 1$ 时, 显然 $\mathbf{FF}(I) = \mathrm{OPT}(I) = 1$. 如果 $\mathbf{FF}(I) > 1$, 记 $W = \sum_{i=1}^{n} w_i$. 因为任何两只箱子的重量之和大于 B.

- 因此当 $\mathbf{FF}(I)$ 为偶数时,
$$W > \frac{B}{2}\mathbf{FF}(I)$$
- 当 $\mathbf{FF}(I)$ 为奇数时, 设最重的箱子重量为 B_1, 则有
$$W > \frac{B}{2}\left(\mathbf{FF}(I) - 1\right) + B_1 > \frac{B}{2}\mathbf{FF}(I)$$

故总有 $W > \frac{B}{2}\mathbf{FF}(I)$, 即 $\mathbf{FF}(I) < \frac{2W}{B}$. 而显然 $\mathrm{OPT}(I) \ge \frac{W}{B}$, 因此证得 $\mathbf{FF}(I) < 2\mathrm{OPT}(I)$, $\square$.

Problem 9

设无向图 $G = \langle V, E\rangle, V_1 \cup V_2 = V, V_1 \cap V_2 = \emptyset$, 称

$$(V_1, V_2) = \left\{(u, v) \,\middle|\, (u, v) \in E, \text{且} u \in V_1, v \in V_2\right\}$$

为 G 的割集, (V_1, V_2) 中的边称为割边, 不在 (V_1, V_2) 中的边称作非割边.

最大割集问题: 任给无向图 $G = \langle V, E\rangle$, 求 G 的边数最多的割集. 考虑下述求最大割集问题的**局部改进算法 (MCUT)**: 令 $V_1 = V, V_2 = \emptyset$. 如果存在顶点 u, 在 u 关联的边中非割边多于割边, 如果 $u \in V_1$, 则把 u 移到 V_2 中; 如果 $u \in V_2$, 则把 u 移到 V_1 中, 直到不存在这样的顶点为止, 取此时得到的 (V_1, V_2) 作为解.

证明: MCUT 是 **2-近似算法**, 即对任一实例 I, 都有

$$\mathrm{OPT}(I) \le 2\mathbf{MCUT}(I)$$

Solution: 根据算法, 每一个顶点关联的割边数大于等于关联的非割边数. 对所有的顶点求和, 每条边出现 2 次, 故所有的割边数大于等于所有非割边数. 从而 $\mathbf{MCUT}(I) \ge \frac{|E|}{2}$. 又显然有 $\mathrm{OPT}(I) \le |E|$, 于是证得 $\mathrm{OPT}(I) \le 2\mathbf{MCUT}(I)$, $\square$.[1]

[1]最小割集问题是多项式时间可解的, 而最大割集问题是 $\mathcal{NPH}$ 的.

Problem 10

双机调度问题 (优化形式): 有 2 台相同的机器和 n 项作业 $J_1, J_2, \cdots, J_n$, 每一项作业可以在任一台机器上处理, 没有顺序的限制, 作业 J_i 的处理时间为正整数 $t_i, 1 \leq i \leq n$. 要求把 n 项作业分配给这 2 台机器使得完成时间最短, 即把 $\{1, 2, \cdots, n\}$ 划分为 I_1 和 I_2, 使得

$$\max\left\{\sum_{i \in I_1} t_i, \sum_{i \in I_2} t_i\right\}$$

最小.

令 $D = \left\lfloor \frac{1}{2}\sum_{i=1}^{n} t_i \right\rfloor, B(i) = \left\{ t \,\middle|\, t = \sum_{j \in S} t_j \leq D, S \subseteq \{1, 2, \cdots, i\} \right\}, 0 \leq i \leq n$. $B(i)$ 包括所有前 i 项作业中任意项 (可以是 0 项) 作业的处理时间之和, 只要这个和不超过所有作业处理时间之和的二分之一. 试给出关于 $B(i)$ 的递推公式, 并利用这个递推公式设计双机调度问题的伪多项式时间算法, 进而设计这个问题的完全多项式时间近似算法.

Solution: 递推公式如下所示:

$$B(0) = \{0\}, B(i) = B(i-1) \cup \left\{ t \,\middle|\, t - t_i \in B(i-1), t_i \leq t \leq D \right\}, i = 1, 2, \cdots, n$$

于是显然有

$$\mathrm{OPT}(I) = \sum_{i=1}^{n} t_i - \max B(n)$$

于是我们可以给出如下的**DP**算法:

Algorithm 1 **DP** 算法

Input: n 个作业的处理时间 $t_1, t_2, \cdots, t_n$
Output: 处理好的作业集 J

令 $D \leftarrow \left\lfloor \frac{1}{2}\sum_{i=1}^{n} t_i \right\rfloor, B(0) \leftarrow 0$;
for $i = 1; i \leq n; i{+}{+}$ **do**
    令 $B(i) \leftarrow B(i-1)$;
    for $t = t_i; t \leq D; t{+}{+}$ **do**
        if $t - t_i \in B(i-1)$ **then**
            $B(i) \leftarrow B(i) \cup \{t\}$;
        end if
    end for
end for
令 $t \leftarrow \max B(n), J \leftarrow \emptyset, i \leftarrow n$;
for $i = n; i \geq 1; i{-}{-}$ **do**
    if $t - t_i \in B(i-1)$ **then**
        $J \leftarrow J \cup \{i\}, t \leftarrow t - t_i$;
    end if
    if $t \leq 0$ **then**
        输出 J, **break**;
    end if
end for
end {DP};

DP 的时间复杂度为 $O(nD)=O\left(n^2 t_{\max}\right)$, 其中 $t_{\max}=\max\left\{t_i \middle| i=1,2,\cdots,n\right\}$. 这是伪多项式时间算法. 进而设计出下述的完全多项式时间近似算法-**FPTAS**:

Algorithm 2 FPTAS 算法

Input: n 个作业的处理时间 $t_1,t_2,\cdots,t_n$ 和 $\epsilon>0$
Output: 处理好的作业集 J

令 $t_{\max}\leftarrow\max\{t_i|i=1,2,\cdots,n\}$;
令 $b\leftarrow\max\left\{\left\lfloor t_{\max}\Big/\left(1+\frac{1}{\epsilon}\right)n\right\rfloor,1\right\}$;
令 $t_i'\leftarrow t_i/b, i=1,2,\cdots,n$;
以 $t_i'(i=1,2,\cdots,n)$ 为输入并运行 DP 算法;
输出处理好的作业集 J;
end {FPTAS};

FPTAS 的时间复杂度为 $O\left(n^2 t'_{\max}\right)=O\left(n^2 t_{\max}/b\right)=O\left(n^3\left(1+\frac{1}{\epsilon}\right)\right)$.

当 $b=1$, **FPTAS** 得到最优解. 不妨设 $b>1$, $b\left(t_i'-1\right)<t_i\le bt_i'$. 对任意的 $S\subseteq\{1,2,\cdots,n\}$, 有

$$b\sum_{i\in S}t_i'-b|S|<\sum_{i\in S}t_i\le b\sum_{i\in S}t_i',$$
$$0\le b\sum_{i\in S}t_i'-\sum_{i\in S}t_i<b|S|\le bn \tag{$*$}$$

记最优解为 J^*, **FPTAS** 得到的近似解为 J 和 $J'=\{1,2,\cdots,n\}\setminus J$, $\mathrm{OPT}(I)=\sum_{i\in J^*}t_i$, **FPTAS**$(I)=\sum_{i\in J'}t_i$, 于是有

$$\begin{aligned}\mathbf{FTPAS}(I)-\mathrm{OPT}(I)&=\sum_{i\in J'}t_i-\sum_{i\in J^*}t_i\\&=\left(\sum_{i\in J'}t_i-b\sum_{i\in J'}t_i'\right)+\left(b\sum_{i\in J'}t_i'-b\sum_{i\in J^*}t_i'\right)+\left(b\sum_{i\in J^*}t_i'-\sum_{i\in J^*}t_i\right)\end{aligned}$$

根据($*$)式可知上述第一项 ≤ 0. J' 是关于 $\left\{t_i'\right\}$ 的最优解, 第二项也 ≤ 0, 于是得到

$$\mathbf{FTPAS}(I)-\mathrm{OPT}(I)\le b\sum_{i\in J^*}t_i'-\sum_{i\in J^*}t_i<bn\le t_{\max}\Big/\left(1+\frac{1}{\epsilon}\right)\le\mathbf{FTPAS}(I)\Big/\left(1+\frac{1}{\epsilon}\right)$$

化简得到 **FTPAS**$(I)\le(1+\epsilon)\mathrm{OPT}(I)$, 因此 **FTPAS** 是双机调度问题的完全多项式时间近似算法,□.

Problem 11

顶点覆盖问题: 任给一个图 $G = \langle V, E\rangle$, 求 G 的顶点数最少的顶点覆盖. 复习顶点覆盖问题的近似算法及其证明.

Solution: MVC算法如下所示:

Algorithm 3 算法 **MVC**(G)

Input: 图 $G = \langle V, E\rangle$
Output: 最小顶点覆盖 V'

```
V' ← ∅, e_1 ← E;
while e_1 ≠ ∅ do
    从 e_1 中任选一条边 (u, v);
    V' ← V' ∪ {u, v};
    从 e_1 中删去与 u 和 v 相关联的所有边;
end while
return V';
end {MVC};
```

显然算法 MVC 的时间复杂度为 $O(m), m = |E|$. 记 $|V'| = 2k$, V' 中的顶点是 k 条边的端点, 这 k 条边互不关联. 为了覆盖这 k 条边则需要 k 个顶点, 从而 $\mathrm{OPT}(I) \geq k$. 于是有

$$\frac{\mathrm{MVC}\,(I)}{\mathrm{OPT}\,(I)} \leq \frac{2k}{k} = 2$$

故 MVC 是最小顶点覆盖问题的 2-近似算法,□.

Problem 12

判断正误:

- 旅行商问题存在多项式时间近似方案. (　)
- 0/1 背包问题存在多项式时间近似方案. (　)
- 0/1 背包问题的贪心算法 (单位价值高优先装入) 是绝对近似算法. (　)
- 多机调度问题的贪心近似算法 (按输入顺序将作业分配给当前最小负载机器) 是 ϵ-近似算法. (　)

Solution:

- 错误. 根据教材可知, 旅行商问题不存在多项式时间近似算法, 除非 $\mathcal{P} = \mathcal{NP}$. 如果存在的话, 那么就可以证得 $\mathcal{P} = \mathcal{NP}$, 即可以拿图灵奖了.
- 正确, **PTAS 算法**就是 0/1 背包问题的多项式时间近似方案.
- 错误, 0/1 背包问题的贪心算法**不是**绝对近似算法.
- 正确, 多机调度问题的贪心近似算法有 GMPS 和 DGMPS 分别是 2-近似和 3/2-近似算法.

Problem 13

设旅行商问题的解表示为 $D = F = \{S|S = (i_1, i_2, \cdots, i_n), i_1, i_2, \cdots, i_n$是$1, 2, \cdots, n$的一个排列$\}$, 邻域定义为 2-OPT(即 S 中的两个元素对换), 求 $S = (3, 1, 2, 4)$ 的邻域 $N(S)$.

Solution: 将 S 中的两个元素对换即可得到 $N(S)$:

$$N(S) = \{(1, 3, 2, 4), (2, 1, 3, 4), (4, 1, 2, 3), (3, 2, 1, 4), (3, 4, 2, 1), (3, 1, 4, 2)\}$$

Problem 14

0/1 背包问题的解记作 $X = (x_1, x_2, \cdots, x_n), x_i \in \{0, 1\}, i = 1, 2, \cdots, n$. 邻域定义为

$$N(X) = \left\{ Y \,\middle|\, \sum_{i=1}^{n} |y_i - x_i| \le 1 \right\}, X = (1, 1, 0, 0, 1)$$

求邻域 $N(X)$.

Solution: 每次只允许一个分量变化即可求出邻域 $N(X)$:

$$N(X) = \{(0, 1, 0, 0, 1), (1, 0, 0, 0, 1), (1, 1, 1, 0, 1), (1, 1, 0, 1, 1), (1, 1, 0, 0, 0)\}$$

Problem 15

写出禁忌搜索算法的主要步骤.

Solution: 禁忌搜索算法的主要步骤如下算法4中所示:

Algorithm 4 禁忌搜索算法步骤

1: 选定一个初始可行解 $\boldsymbol{x}^{\mathrm{cb}}$ 并初始化禁忌表 $H \leftarrow \{\}$;
2: **while** 不满足停止规则 **do**
3: 在 $\boldsymbol{x}^{\mathrm{cb}}$ 的邻域中选出满足禁忌要求的候选集 Can-$N\left(\boldsymbol{x}^{\mathrm{cb}}\right)$;
4: 从该候选集中选出一个评价值最佳的解 $\boldsymbol{x}^{\mathrm{lb}}$;
5: 令 $\boldsymbol{x}^{\mathrm{cb}} \leftarrow \boldsymbol{x}^{\mathrm{lb}}$ 并更新记录 H;
6: **end while**

Problem 16

禁忌对象特赦可以基于影响力规则: 即特赦影响力大的禁忌对象. 影响力大什么含义? 举例说明该规则的好处.

Solution: 影响力大意味着有些对象变化对目标值影响很大. 如 0/1 背包问题, 当包中无法装入新物品时, 特赦体积大的分量来避开局部最优解.

Problem 17

判断正误:

- 禁忌搜索中, 禁忌某些对象是为了避免领域中的不可行解. ( )
- 禁忌长度越大越好. ( )
- 禁忌长度越小越好. ( )

Solution:

- 错误, 选取禁忌对象是为了引起解的变化, 根本目的在于避开邻域内的局部最优解而不是不可行解.
- 错误, 禁忌长度短了则可能陷入局部最优解.
- 错误, 禁忌长度长了则导致计算时间长.

Problem 18

写出模拟退火算法的主要步骤.

Solution: 模拟退火算法的主要步骤如下算法5中所示:

Algorithm 5 模拟退火算法步骤

任选初始解 x_0 并初始化 $x_i \leftarrow x_0, k \leftarrow 0, t_0 \leftarrow t_{\max}$(初始温度);
while $k \leq k_{\max}$ && $t_k \geq T_f$ **do**
    从邻域 $N(x_i)$ 中随机选择 x_j, 即 $x_j \leftarrow_R N(x_i)$;
    计算 $\Delta f_{ij} = f(x_j) - f(x_i)$;
    if $\Delta f_{ij} \leq 0 \,\|\, \exp\left(-\Delta f_{ij}/t_k\right) > \mathrm{RANDOM}(0,1)$ **then**
        $x_i \leftarrow x_j$;
    end if
    $t_{k+1} \leftarrow d(t_k)$;
    $k \leftarrow k+1$;
end while

Problem 19

为避免陷入局部最优 (小), 模拟退火算法以概率 $\exp\left(-\Delta f_{ij}/t_k\right)$ 接受一个退步 (比当前最优解差) 的解, 以跳出局部最优. 试说明参数 $t_k, \Delta f_{ij}$ 对是否接受退步解的影响.

Solution: 很明显, 当 t_k 较大时, 接受退步解的概率越大; 当 Δf_{ij} 较大时, 接受退步解的概率越小.

Problem 20

下面属于模拟退火算法实现的关键技术问题的有 ______.

(A) .初始温度 (B) .温度下降控制 (C) .邻域定义 (D) .目标函数

Solution: 模拟退火算法实现的关键技术问题有**邻域的定义 (构造)、起始温度的选择、温度下降方法、每一温度的迭代长度以及算法终止规则**. 因此选择 (A), (B), (C).

Problem 21

用遗传算法解某些问题, fitnees $= f(x)$ 可能导致适应函数难以区分这些染色体. 请给出一种解决办法.

Solution: 采用非线性加速适应函数如下

$$\text{fitness}\,(x) = \begin{cases} \dfrac{1}{f_{\max} - f\,(x)}, & f\,(x) < f_{\max} \\ M > 0, & f\,(x) = f_{\max} \end{cases}$$

其中 M 是一个充分大的值, $f_{\max}$ 是当前最优值.

Problem 22

用非常规编码染色体实现的遗传算法, 如 TSP 问题使用 $1, 2, \cdots, n$ 的排列编码, 简单交配会产生什么问题? 如何解决?

Solution: 后代可能会出现非可行解, 因此需要通过罚值和交叉新规则来解决.

Problem 23

下面属于遗传算法实现的关键技术问题的有 ______.

(A) .解的编码　　(B) .初始种群的选择　　(C) .邻域定义　　(D) .适应函数

Solution: 遗传算法实现的关键技术问题有**解的编码、适应函数、初始种群的选取、交叉规则以及终止规则**. 因此选择 (A), (B), (D).

至此, 9-10-11 章的所有练习都已做完.