代码：[github]

第一题：

简答计算部分：

我们收到了一封信邮件。在未统计分析之前， 我们假定它是垃圾邮件的概率为 50%。 我们用 S 表示垃圾邮件（spam），H 表示正常邮件（healthy）。因此，P(S)和 P(H)的先验概率，均为 50%——P(S) = P(H) = 50%

对这封新邮件的内容进行解析，发现其中含有“中奖”这个词，那么这封邮件属于垃圾邮件的概率提高到：

我们用 W 表示“中奖”这个词，那么问题变成了如何计算 P(S|W)的值，即在某个次于（W）已经存在的情况下，垃圾邮件（S）的概率有多大。

根据条件概率公式我们可以得到：

P(垃圾|中奖)=P(S|W) =

P(W|S)P(S)/(P(W|S)P(S)+P(W|H)P(H))=

0.05*0.5/(0.05*0.5+0.001*0.5)=5/5.1≈0.98039

因此这封邮件是垃圾邮件的概率约等于 98%

误判问题：

仅仅从一个关键词“中奖”来判断是否是垃圾邮件太绝对了，因为一封邮件包含很多词语，一些词语（比如“中奖”）是这是垃圾邮

件，另一些说不是，毕竟我们无法确定以哪个词为准。对于误判问题，解决的思路是“多特征判断”，就像我们分辨猫和老虎，单看颜色和花纹很难区分这两种动物，毕竟老虎是猫科动物，如何判断呢？那就从颜色大小花纹体重等特征一起来判断。

因此，提出用“贝叶斯推断”过滤垃圾邮件的 Paul GraHam 的做法是：选出这封邮件中 P(S|W)最高的 15 个词作为特征，计算它们的联合概率。(如果有的词是第一次出现,无法计算 P(S|W),Paul Graham 就假定这个值等于 0.4。因为垃圾邮件用的往往都是某些固定的词语，所以如果从来没有见过某个词，它多半是一个正常的词。）

联合概率，就是指在多个事件发生的情况下，另一个事件发生的概率有多大。比如，已知 W1 和 W2 是两个不同的词语，它们都出现在某封电子邮件之中,那么这封邮件是垃圾邮件的概率,就是联合概率。

在已知 W1 和 W2 的情况下，无非就是两种结果：垃圾邮件（事件 E1）或正常邮件（事件 E2）。

事件	W1	W2	垃圾邮件
E1	出现	出现	是
E2	出现	出现	否

其中，W1,W2 和垃圾邮件的概率：

事件	W1	W2	垃圾邮件
E1	P(S|W1)	P(S|W2)	P(S)
E2	1-(S|W1)	1-P(S|W2)	1-P(S)

如果假定所有事件都是独立事件（严格地说，这个假定不成立，

但是这里可以忽略），那么就可以计算P(E1)和P(E2)：

P(E1) = P(S|W1)P(S|W2)P(S)

又由于E1是在W1和W2同时出现的情况下是垃圾邮件的事件，E2是W1和W2同时出现的情况下是正常邮件的事件，这里的前提都是“在W1和W2同时出现的情况下”。那么，P = P(E1)/(P(E1)+P(E2))，其意思是W1和W2共同出现的情况下是垃圾邮件的概率，而P(W1,W2)实际上就是P(E1)+P(E2)，所以P = P(E1)/(P(E1)+P(E2))。

所以，垃圾邮件的概率等于下面的式子：

P = P(E1)/(P(E1)+P(E2))

也就是：

$$P=\frac{P(S|W1)P(S|W2)P(S)}{P(S|W1)P(S|W2)P(S)+(1-P(S|W1))(1-P(S|W2))(1-P(S))}$$

将P(S) = 0.5代入其中，化简为

$$P=\frac{P(S|W1)P(S|W2)}{P(S|W1)P(S|W2)+(1-P(S|W1))(1-P(S|W2))}$$

将上面的公式扩展，就得到了最终的概率计算公式：

$$P=\frac{P1P2\cdots Pn}{P1P2\cdots Pn+(1-P1)(1-P2)\cdots(1-Pn)}$$

我们将公式扩展到n个词，通过计算就得到了最终的是垃圾邮件的概率。当然要把一封邮件是不是归到垃圾邮件，我们还需要一个用于比较的门槛值，Paul 的门槛值是0.9，当计算得出的概率大于0.9，表示这n个词语联合认定，这封邮件有90%以上的可能属于垃圾邮件，概率小于0.9，就表示是正常邮件，有了上面这个公式，一封正常的邮件即使出现了“中奖”这个词，也不会被误判为垃圾邮件了。

算法设计：

首先，在评测算法中，我们需要用到；

准确率（Accuracy）

描述的是所有样本中识别正确的比例

精确率（Precision）

描述在预测为垃圾邮件的样本中，有多少是预测正确的

召回率（Recall）

描述在所有的垃圾邮件中，有多少被预测识别到了

具体算法思路：

1.输入所有邮件，

然后得到邮件中每个单词

出现在垃圾邮件中的次数概率，

出现在正常邮件中的次数概率，

模型就训练出来了（训练集数量已知各为 350）。

2.然后输入一封在测试集随机找一个待测试邮件测试，

找到里面所有出现的关键词。

求出 P(A|T1,…Tn)P(A|T1,…Tn)，

A 为一封邮件是垃圾邮件的事件，

T 为关键词出现在一封邮件中的事件。

T1,…TnT1,…Tn 是多个关键词。

A 和 T 是关联的事件。

T1,…TnT1,…Tn 每个关键词根据朴素贝叶斯的假设，

是相互独立的。

P(A|T1,…Tn)P(A|T1,…Tn)

为 T1,…TnT1,…Tn

这些关键词同时出现的情况下

A 是垃圾邮件的概率。

实验中我设置 n 为 EN 为 15，

P(A|T1,…Tn)P(A|T1,…Tn)求出之后，

就得到一个概率，

我们可以自己设置一个阈值，

比如说概率大于 0.9 时，

认为此邮件为垃圾邮件。

3.一封邮件可以确定之后，

我们可以用同样的方法来测试待测试数据集，

由于还要标记每个邮件很麻烦，

TA 给出的所有数据文件名都可以辨别是否是垃圾邮件，

所以具体测试时，

把所有的测试邮件按序（先垃圾后正常）输入，

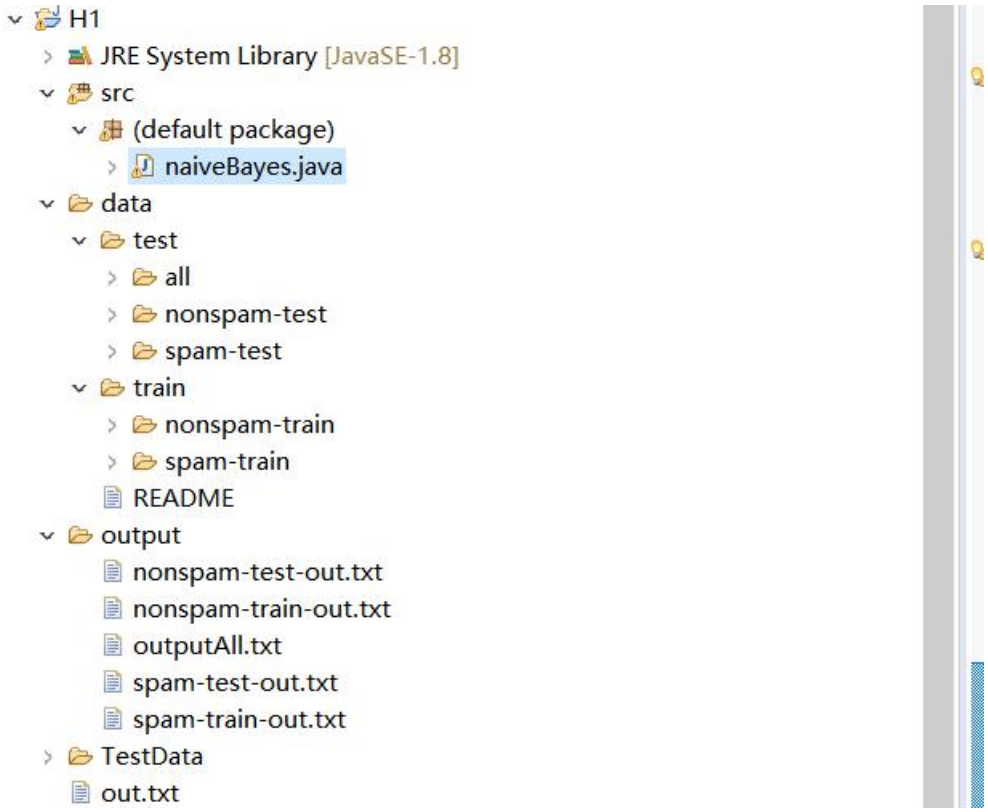
看垃圾邮件多少被标记错为正常，

正常标记为垃圾邮件就可以测试这些邮件的效果了，

得到测试的召回率和准确率，

然后评价算法的效果。

具体文件说明：



以上文件，主程序在 naiveBayes. java 中，使用朴素贝叶斯算法完成实验，代码中都加有详细注释说明，删除一些注释符号可以看到具体的一些计算过程和细节，比如每个单词在某个类别下的概率等具体参见代码。

测试数据和训练数据都放在了 data 下面，test 中新增的 all 是 nonspam-test 和 spam-test 混合文件，用作随机测试方便。

output/outputAll 是所有测试结果，“是”代表判断为垃圾邮

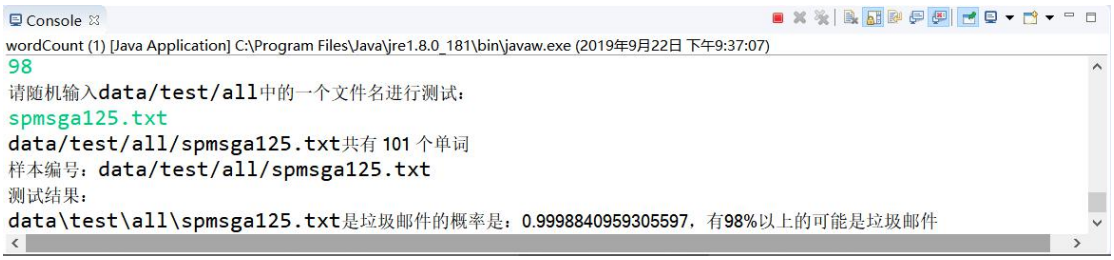
件,其中 5~134 行是输入的垃圾邮件测试集，在设置阈值为 0.9 时，其中被误判的有 2 个，后面的是输入的正常邮件，被误判的有 7 个。

```
265 不是
266 正常邮件被错判为垃圾邮件的7个
267 误判率为3%
268 准确率为96%
269 精确率为94%
270 召回率为1%
```

output 下其余的是在训练的过程中记录的四个文件夹下面的文件集的单词词频和概率和单词总数统计结果。（两个 test 文件夹在训练的时候顺便一起统计了，但是没有用它来训练算法，留下测试时用了）

```
1 统计结果:
2 共计单词:30314个,不重复单词：7393个
3
4 单词          出现次数          概率
5 language      395         1.303%
6 university    362         1.194%
7 linguistic    183         0.603%
8 english   165         0.544%
9 workshop      119         0.392%
.0 one   118         0.389%
.1 conference    117         0.385%
.2 address   98          0.323%
```

Out.txt 是随机测试文件的结果，这里我将所有的测试文件混合放在了 test/all/里面随机测试时方便测试，同时不用跨目录方便一些。



TestData 是一开始用来测试的小数据集

第二题：

A*算法原理

从图的特定起始节点开始，A*旨在找到从起始节点到目标节点见具有最小代价的路径（最少行驶距离、最短时间等）。A*算法维护源自起始节点的路径树，并且一次一个地延伸这些路径直到满足其终止标准。

在 A*算法主循环的每次迭代中，需要确定对哪条路径进行扩展，A*算法根据路径的成本和评估点到目标点的的成本的估计来选择，具体来说，A*算法选择待评估集合中能使下式最小的点作为扩展路径的点：

其中是路径上的下一个点，是从起始点到节点的路径代价，是一个启发式函数，它是节点到目标点的估算代价。

A*的典型实现使用优先级队列来重复选择的最小成本（即 f(n)）节点以进行扩展。此优先级队列称为 open set 或 fringe。在算法的每个步骤中，从队列中移除具有最低 f(n)值的节点，相应地更新其邻居的 f 和 g 值，并且将这些邻居添加到队列中。循环执行以上步骤，

直到目标点被扩展，或者队列为空。目标点的 f 值是即为最短路径的成本，因为目标处的 h 值为零。

为了找到最短路径的节点序列，可以使路径上的每个节点指向其前趋。运行此算法后，结束节点将指向其前趋，依此类推，直到某个节点的前趋为起始节点。

关于 h 值

下面介绍在平面栅格地图中 h 值的三种计算方法：

曼哈顿距离当智能体只能在 4 个方向（无对角线）上移动时，可以使用曼哈顿距离作为 h 值。计算方法如下：

h = abs(current.x - goal.x) + abs(current.y-goal.y)

对角线距离当智能体能在 8 个方向上移动时，可以使用对角线距离作为 h 值。计算方法如下：

h = max{abs(current.x - goal.x), abs(current.y - goal.y) }

欧式距离 当智能体能在任意方向上移动时，可以使用欧式距离作为 h 值。计算方法如下：

h = sqrt(abs(current.x - goal.x)^2 + abs(current.y - goal.y)^2)

h 值大小的会影响 A*算法的效果：

h(n) := 0，则只有 g(n)起作用，此时 A*演变成 Dijkstra 算法，这保证能找到最短路径。

如果 h(n)比从 n 移动到目标的实际代价小（或者相等），则 A*确定能找到一条最短路径。但是 h(n)越小，A*扩展的结点越多，运行就得越慢。

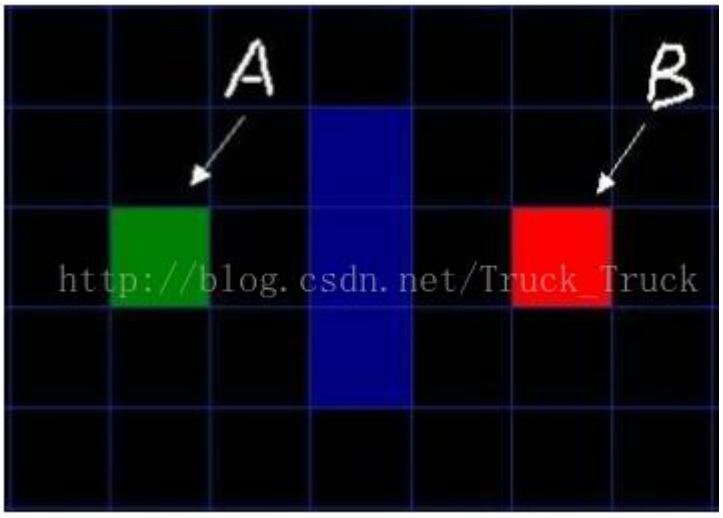
如果 h(n)精确地等于从 n 移动到目标的代价，则 A*将会仅仅寻找最佳路径上的节点而不扩展别的任何结点，这会运行得非常快。尽管这不可能在所有情况下发生，但仍可以在一些特殊情况下让它们精确地相等。只要提供完美的信息，A*算法会运行得很完美。

如果 h(n)比从 n 移动到目标的实际代价高，则 A*不能保证找到一条最短路径，但它运行得更快。

如果 h(n)比 g(n)大很多，则只有 h(n)起作用，A*演变成 BFS 算法。

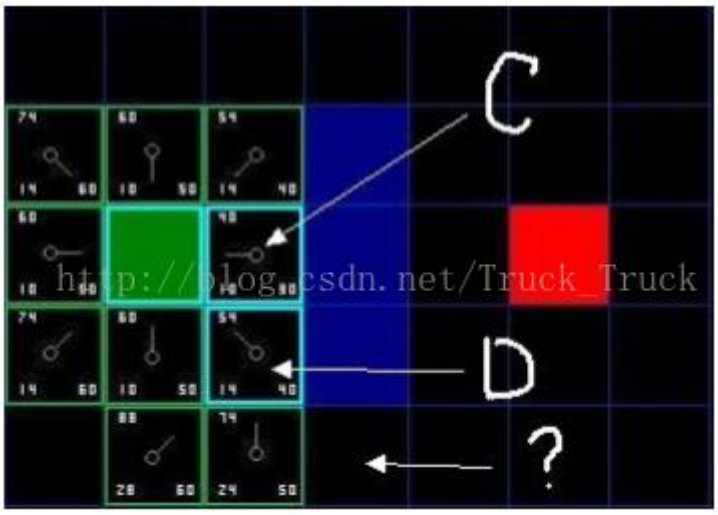
A*算法理解：

1.简易地图



如图所示简易地图，其中绿色方块的是起点，中间蓝色的障碍物，红色方块表示目的地，我们用一个二位数组来表示地图。

2.寻路步骤



1. 从起点 A 开始，把它作为待处理的方格存入一个"开启列表"，开启列表就是一个等待检查方格 的列表.

2. 寻找起点 A 周围可以到达的方格，将它们放入"开启列表"，并设置它们的"父方格"为 A.

3. 从"开启列表"中删除起点 A，并将起点 A 加入"关闭列表"，"关闭列表"中存放的都是不需要再次检查的方格

从 "开启列表" 中找出相对最靠谱的方块，什么是最靠谱? 它们通过公式 F=G+H 来计算.

F = G + H

G 表示从起点 A 移动到网格上指定方格的移动耗费 (可沿斜方向移动).

H 表示从指定的方格移动到终点 B 的预计耗费 (H 有很多计算方法，这里我们设定只可以 上下左右移动).

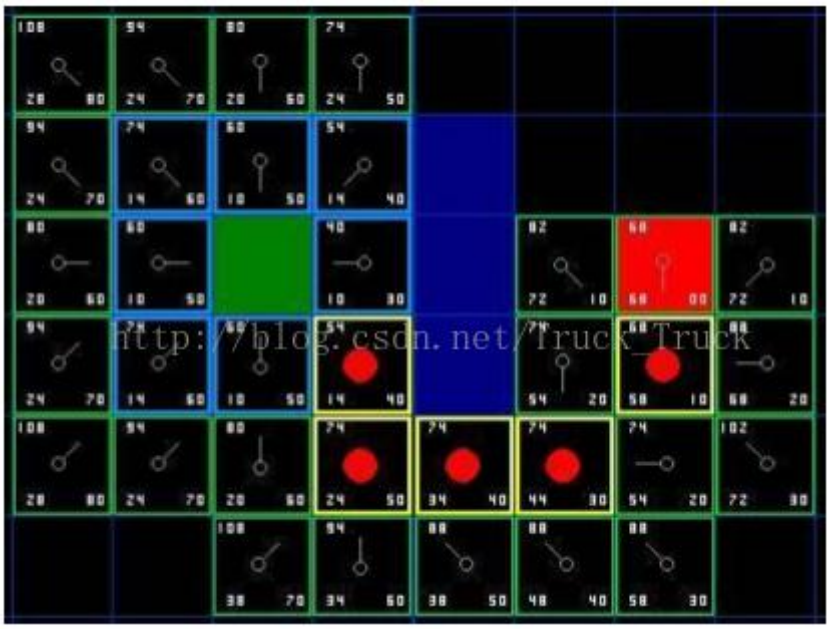
从 "开启列表" 中选择 F 值最低的方格 C (绿色起始方块 A 右边的方块)，然后对它进行如下处 理:

4. 把它从 "开启列表" 中删除，并放到 "关闭列表" 中.

5. 检查它所有相邻并且可以到达（障碍物和 "关闭列表" 的方格都不考虑）的方格. 如果这些方格 还不在 "开启列表" 里的话，将它们加入 "开启列表"，计算这些方格的 G，H 和 F 值各是多少，并设置 它们的 "父方格" 为 C.

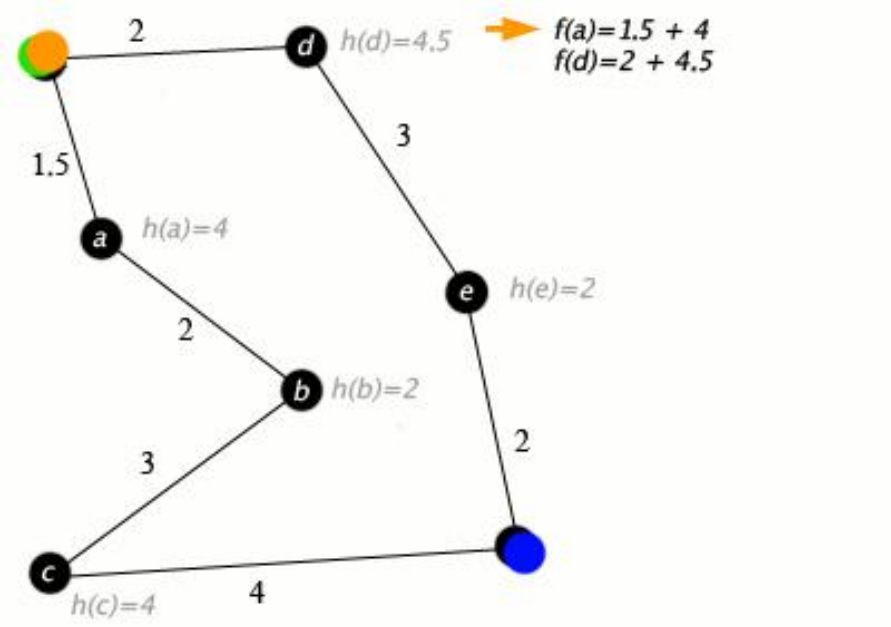
6. 如果某个相邻方格 D 已经在 "开启列表" 里了，检查如果用新的路径（就是经过 C 的路径）到 达它的话，G 值是否会更低一些，如果新的 G 值更低，那就把它的 "父方格" 改为目前选中的方格 C，然 后重新计算它的 F 值和 G 值（H 值不需要重新计算，因为对于每个方块，H 值是不变的). 如果新的 G 值比较高，就说明经过 C 再到达 D 不是一个明智的选择，因为它需要更远的路，这时我们什么也不做.

3.如何找回路径



除了起始方块，每一个曾经或者现在还在 "开启列表" 里的方块，它都有一个 "父方 块"，通过 "父方块" 可以索引到最初的 "起始方块"，这就是路径.

动态演示图（打开网页动态图）：



算法设计：

伪代码：（来自维基百科）

```
function reconstruct_path(cameFrom, current)
    total_path := {current}
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.prepend(current)
    return total_path
// A* finds a path from start to goal.// h is the heuristic function. h(n) estimates the cost to reach goal from node n.function A_Star(start, goal, h)
    // The set of discovered nodes that need to be (re-)expanded.
    // Initially, only the start node is known.
    openSet := {start}
```

```
    // For node n, cameFrom[n] is the node immediately preceding it on the cheapest path
from start to n currently known.
    cameFrom := an empty map

    // For node n, gScore[n] is the cost of the cheapest path from start to n currently known.
    gScore := map with default value of Infinity
    gScore[start] := 0

    // For node n, fScore[n] := gScore[n] + h(n).
    fScore := map with default value of Infinity
    fScore[start] := h(start)

    while openSet is not empty
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        closedSet.Add(current)
        for each neighbor of current
            if neighbor in closedSet
                continue
            // d(current,neighbor) is the weight of the edge from current to neighbor
            // tentative_gScore is the distance from start to the neighbor through current
            tentative_gScore := gScore[current] + d(current, neighbor)
            if tentative_gScore < gScore[neighbor]
                // This path to neighbor is better than any previous one. Record it!
                cameFrom[neighbor] := current
                gScore[neighbor] := tentative_gScore
                fScore[neighbor] := gScore[neighbor] + h(neighbor)
                if neighbor not in openSet
                    openSet.add(neighbor)

    // Open set is empty but goal was never reached
    return failure
```

算法主要以实现以下过程为主：

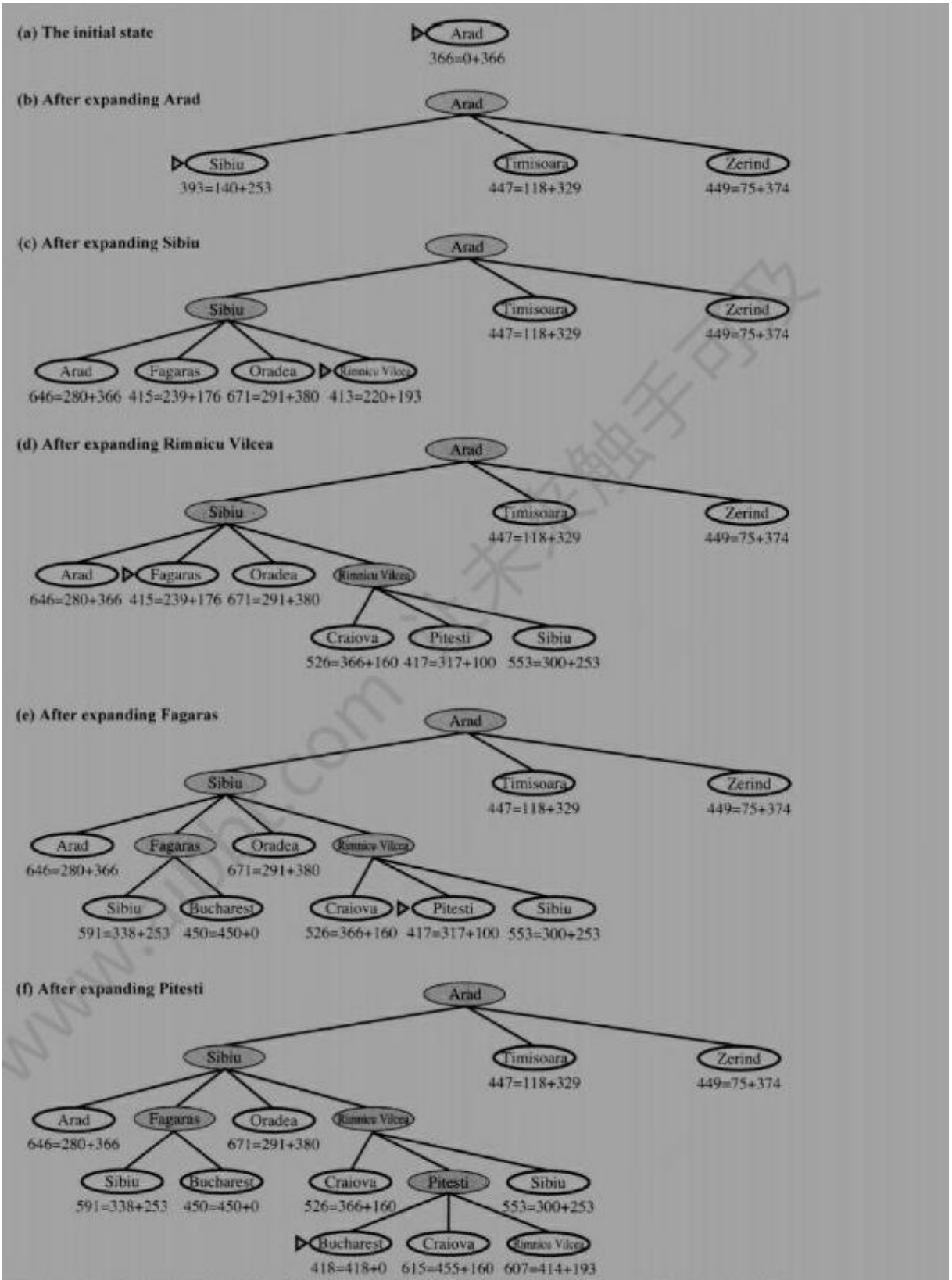


图 3.24　使用 A*搜索求解罗马尼亚问题。结点都用 $f = g + h$ 标明。h 值是图 3.22 给出的到 Bucharest 的直线距离

具体文件说明：

项目中有两个源文件：Astar.java Count.java（程序入口）

执行程序输出如下：

默认将全部城市作为起点输出结果：

第一行是城市名称

第二行到探索下标之间是 起点下标 g(n) h(n) f(n)

探索下标是对应得路径的下标顺序

访问过程就是访问路径

总长度为 A*算法计算出来的最短路径长度

```
终点城市：Bucharest

请输入一个数字代表您的起点城市：

0-Arad
1-Zerind
2-Oradea
3-Timisoara
4-Sibiu
5-Lugoj
6-Rimnicu Vilcea
7-Fagaras
8-Mehadia
9-Drobeta
10-Craiova
11-Pitesti
12-Bucharest
13-Giurgiu
14-Urziceni
15-Hirsova
16-Eforie
17-Vaslui
18-Iasi
19-Neamt

Arad
0 0 366 366
4 140 253 393
6 220 193 413
11 317 100 417
12 418 0 418
探索下标：0-->4-->6-->11-->12
访问过程：Arad-->Sibiu-->Rimnicu Vilcea-->Pitesti-->Bucharest
总长度为：418
```

```
总长度为：418

Zerind
1 0 374 374
0 75 366 441
4 215 253 468
6 295 193 488
11 392 100 492
12 493 0 493
探索下标：1-->0-->4-->6-->11-->12
访问过程：Zerind-->Arad-->Sibiu-->Rimnicu Vilcea-->Pitesti-->Bucharest
总长度为：493

Oradea
2 0 380 380
4 151 253 404
6 231 193 424
11 328 100 428
12 429 0 429
探索下标：2-->4-->6-->11-->12
访问过程：Oradea-->Sibiu-->Rimnicu Vilcea-->Pitesti-->Bucharest
总长度为：429

Timisoara
3 0 329 329
5 111 244 355
8 181 241 422
9 256 242 498
10 376 160 536
11 514 100 614
12 615 0 615
探索下标：3-->5-->8-->9-->10-->11-->12
访问过程：Timisoara-->Lugoj-->Mehadia-->Drobeta-->Craiova-->Pitesti-->Bucharest
总长度为：615
```

这里将全部结果都已输出，

如果需要随机调试，将main程序中注释段的注释符号删除即可按照提示输入测试