人工智能实验

变量消除算法

16 级计科教务 2 班

16337327

郑映雪

实验题目

变量消除算法

实验内容

算法原理

在贝叶斯网络中，我们需要计算某个变量的概率，此时利用各个情况相加，可以计算：

$$\mathrm{P(a = a_1)} = \sum_{x1}\sum_{x2}\dots\dots\sum_{xn} p(a = a_1, x1, x2, \dots xn)$$

由上面的式子我们可以知道，我们要是想求得一个变量的概率，则需要将其它变量边缘化。这种“边缘化”是由求各种情况的和实现的。但是，随着网络中节点规模的增大，使用全概率公式计算需要耗费大量时间，这就出现了由贝叶斯网络的性质得到的变量消除算法。

我们可以定义一个 factor（因子），视作一个多维表格，这个表格表示了概率分布。在贝叶斯网络中，这些因子对应的是当前结点及其父节点的独立（没有父节点）的概率或条件概率（当前结点和父节点）分布。而变量消除算法则是重复地执行两个因子的乘法和边缘化的运算。下面就两个因子定义变量消除算法中需要的运算：

1、乘积运算

对于两个因子 f(X,Y)和 g(Y,Z),二者的乘积 h（X,Y,Z）=f（X,Y）*g（Y,Z）。注意此处的乘积需要共同的变量取值相同。乘积运算得到一个新的因子，即得到一个新的概率分布。

2、求和运算

假设在因子 f（X,Y）中，我们需要消除 X，求得 Y，则采取求和公式h(Y) = $\sum_x f(x,Y)$。由因子的定义，我们知道此时 f 是条件概率，所以该式符合全概率公式。

3、投影运算

投影运算是指在一个因子 f（X,Y）中，其中一个变量取特殊值得到的新的因子。比如在 f（X,Y）中，取 X=a，则得到新的因子h = $f_{x=a} = f(a,Y)$

有了以上三种运算之后，我们可以描述变量消除算法：

对于给定消除顺序中的变量 Zj，我们可以按下面的步骤消除：

①初始化包含每个变量的因子 f1，f2,…fk；

②计算新的因子$g_j = \sum_{z_j} f1 * f2 * \dots fk$**；**

③将已经计算过的因子删除，对于变量 j 添加新的因子 gj。

最终得到希望计算的因子的分布，再进行归一化处理，即得到最终的分布。

上面的步骤我们可以知道，变量消除算法是将一些乘法步骤推进了加法步骤里，这样可以做到减少计算的复杂度。

但是，即使使用了变量消除算法，变量消除的顺序也会对算法的复杂程度有影响。我们需要根据网络图的结构决定消除变量的顺序。消除变量的顺序可以由以下几个方面决定：

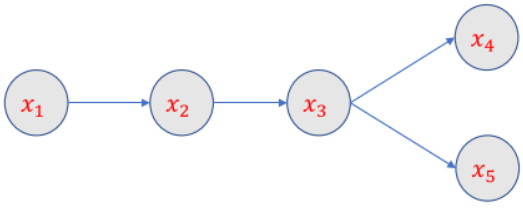
①最少邻结点：某个点相连结点最少则优先消除该结点；

②最小权重：一条边的权重即为该条边表示的因子值，消除因子值之和最小的结点；

③最少补充：在消除该结点后，补充的边最少即为优先消除的结点；

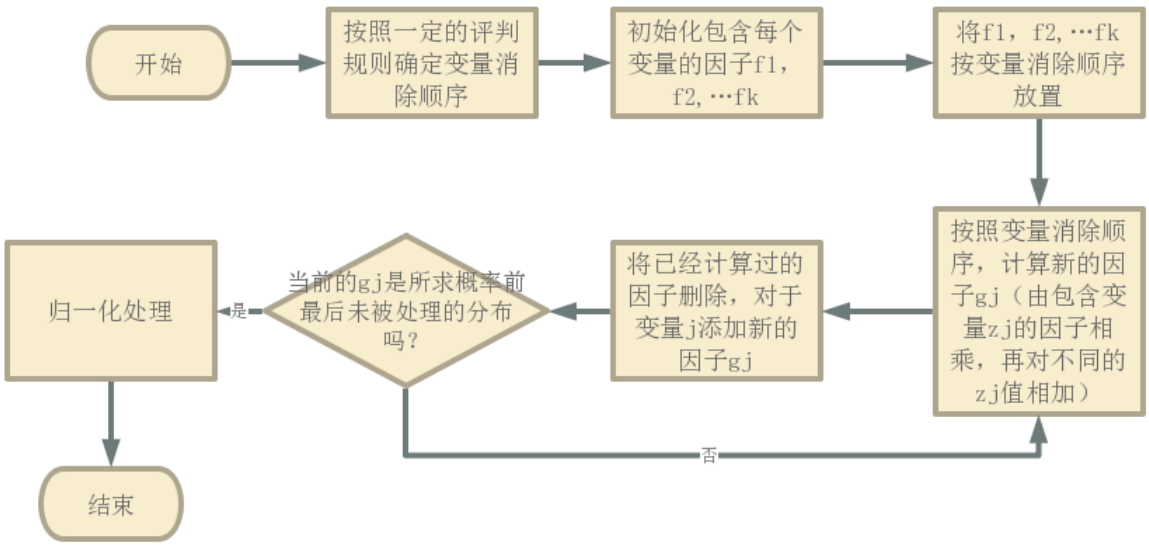
④最小补充边权重：在消除该结点后，补充边的权重（可算作两节点权重的积）最小，即为优先消除的结点。

如下图所示：



在上图中，先消除 x1 是优于先消除 x3 的。以此类推，我们可以在子图中逐步确认消除顺序，得出 x1、x2、x3、x4、x5 的顺序是较优的。

流程图



关键代码

1、乘积运算

乘积运算对列表和分布字典的操作是：将另一个因子的列表并入当前要计算乘积的因子列表当中。同时，判断两个因子相同的元素，对相同元素的位置进行判断，如果字典键名的位置相同，则对这些键值进行相乘。具体代码如下：

```
def multiply(self, factor):
    '''''function that multiplies with another factor'''
    # Your code here
    new_list = []
    for i in range(len(self.var_list)):
        if self.var_list[i] in factor.var_list:
            theindex = factor.var_list.index(self.var_list[i])
```

```
            new_list = self.var_list + factor.var_list[:theindex] + factor.var_list[theindex + 1:]#对新的变量进行结合两个因子的命名
            break
    new_cpt = {}
    for key1, value1 in self.cpt.items():
        for key2, value2 in factor.cpt.items():
            if key1[i] == key2[theindex]:#如果当前变量在两个因子里值相等
                new_cpt[key1 + key2[:theindex] + key2[theindex + 1:]] = value1 * value2#则计算乘积
    new_node = Node('f' + str(new_list), new_list)
    new_node.set_cpt(new_cpt)
    return new_node
```

2、求和运算

求和运算对因子列表和分布字典的操作是：找到因子中需要进行求和的元素位置，在元素列表里删除。并根据之前找到的位置，在字典键名中的相应位置进行判断，该位置的字符相等的则进行相加，并去掉该字符，作为新的字典键值对。具体代码如下：

```
def sum_out(self, variable):
    '''''function that sums out a variable given a factor'''
    # Your code here
    new_var_list=[]
    new_cpt={}
    for vars in self.var_list:
        new_var_list.append(vars)
    theindex=self.var_list.index(variable)
    new_var_list.remove(variable)#删除需要求和的变量
    for key,value in self.cpt.items():
        if key[:theindex]+key[theindex+1:] in new_cpt.keys():#如果求和变量已经在列表里面了
            new_cpt[key[:theindex]+key[theindex+1:]]+=value#则相加
        else:
            new_cpt[key[:theindex]+key[theindex+1:]]=value#否则创建新的键值
    new_node = Node('f' + str(new_var_list), new_var_list)
    new_node.set_cpt(new_cpt)
    return new_node
```

3、投影运算

投影运算对于因子列表和分布字典的操作是：判断需要投影的变量在列表中的位置，将此变量从列表中删除。在分布字典里，对于字典键搜索该位置，如果该位置的值是需要投影的值，则新字典添加该键减去变量位置得到的新键，并将原来的值赋给该字典，得到新的键值对。代码如下：

```
1. def restrict(self, variable, value):
2.     '''''function that restricts a variable to some value
3.     in a given factor'''
4.     # Your code here
5.     new_var_list=[]
6.     new_cpt = {}
7.     index = self.var_list.index(variable)
8.     for var in self.var_list:
9.         new_var_list.append(var)
10.    new_var_list.remove(variable)#删除需要投影的变量名
11.    for key, values in self.cpt.items():
12.        if key[index] == value:
13.            new_cpt[key[:index] + key[index + 1:]] = values#如果当前变量值是
   投影值，则赋值
14.    new_node = Node('f' + str(new_var_list), new_var_list)
15.    new_node.set_cpt(new_cpt)
16.    return new_node
```

实验结果及分析

实验结果展示

此处展示源代码中已有的结果：

1、P（A）

```
P(A) **********
RESULT:
Name = f['A']
 vars ['A']
   key: 1 val : 0.0025164420000000002
   key: 0 val : 0.997483558
```

2、P (B|J&~M)

```
P(B | J, ~M) **********
RESULT:
Name = f['B']
 vars ['B']
   key: 0 val : 0.9948701418665987
   key: 1 val : 0.0051298581334013015
```