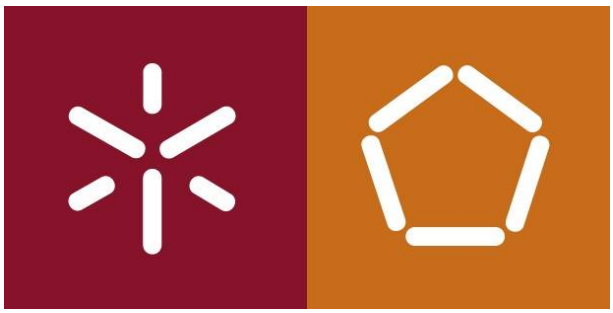

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Modelos Estocásticos de Investigação Operacional

Trabalho prático

Alexandre Costa (a78890)
Etienne Costa (a76089)
Pedro Costa (a85700)
Rui Azevedo (a80789)
11 de Maio de 2020

Conteúdo

Lista de Figuras

1 Introdução

O presente documento tem como objectivo descrever todo o processo envolvido na determinação de uma política ótima para a transferência diária de carros entre filiais do ramo automóvel. O problema foi resolvido usando programação dinâmica estocástica, uma vez que as variáveis do problema não são valores determinísticos, dependendo de acontecimentos incertos.

Numa primeira fase, irá ser apresentada a formulação do problema, definindo quais as suas variáveis e constantes. De seguida, irá ser descrito o programa desenvolvido para realizar os cálculos necessários para obter uma solução do problema. Por fim, irá ser feita uma análise detalhada sobre os resultados obtidos.

2 Formulação do problema

O problema pedido para resolver diz respeito a uma empresa do ramo automóvel, gerida por um determinado empresário, em que existem duas filiais em seu nome. Em cada filial existe um número determinado de carros em *stock*, onde, diariamente, chegam carros para serem alugados e clientes para alugar carros. Uma vez que as filiais têm um limite máximo de carros e que todos dias podem chegar novos carros para serem alugados, é possível haver transferência de carros entre as duas filiais, de maneira a que o empresário tenha menos perdas de venda. O objectivo é, então, definir a política óptima de transferência de carros entre as duas filiais.

Este problema vai ser formulado como um problema de programação dinâmica estocástica com alternativas e um número infinito de estágios, pois é um modelo que se adequa na perfeição para a resolução deste tipo de problemas, em que os acontecimentos de chegadas de clientes e carros para aluguer a cada uma das filiais, são incertos, dependendo de um conjunto de probabilidades. Este problema foi formulado como um problema de maximização do lucro que o empresário, que gere as filiais, poderá vir a obter. Para além do lucro óptimo, pretende-se também descobrir o número de transferências de carros entre as filiais, de modo a tornar este lucro, o maior possível.

Sendo um problema de programação dinâmica, antes de tudo, é necessário definir os seguintes conceitos:

- Estados : número de carros em *stock* de uma determinada filial.
- Estágio : fim de cada dia.
- Decisão : número de carros que devem ser transferidos entre filias.

Na secção seguinte será explicado, em detalhe, todo o processo de modulação do problema e, por fim, análise dos resultados obtidos.

3 Descrição do programa desenvolvido

Tendo esta combinação por base, começamos por criar as matrizes de transição para cada filial, considerando uma política em que nada se transfere. Simultaneamente, calculamos também as matrizes de custos.

Para este fim, desenvolvemos um algoritmo que percorre todos as transições possíveis e, inicialmente, deriva os mínimos e máximos de pedidos e entregas que podem ocorrer para aquele estado. De seguida, calcula a probabilidade com base em todas as combinações possíveis entre pedidos e entregas. Estas combinações divergem em 3 possibilidades:

- **Caso base**: Para uma dada transição que não se dirija ao estado 12, percorremos todas as entregas válidas (entre o valor mínimo e o máximo) e combinamos-las com o número correto de pedidos para satisfazer a transição em causa.
- **Alternativa ao caso base**: Numa transição que se dirija ao estado 12 é necessário ter em consideração casos adicionais. Nestas transições é possível que se ultrapasse o número máximo de carros que uma filial aguenta pelo que é necessário ter esses casos em consideração.
- **Pedidos não satisfeitos**: Para qualquer transição, é também necessário considerar que podem ocorrer pedidos que não se conseguem satisfazer. Para considerar esse caso, fixamos qualquer valor de entrega válida e consideramos a chance de ocorrer com um pedido inconcretizável.

Exemplo:

$$\begin{aligned} P(4,7) = {} & P(0) * E(3) + P(1) * E(4) + P(2) * E(5) + P(3) * E(6) + P(4) * E(7) \\ & + P(5) * E(7) + P(6) * E(7) + P(7) * E(7) + P(8) * E(7) + P(9) * E(7) \\ & + P(10) * E(7) + P(11) * E(7) + P(12) * E(7) \end{aligned}$$

A matriz de contribuições base foi calculada de maneira semelhante em que para cada possibilidade entre pedidos e entregas, multiplicamos a probabilidade com o lucro que se obterá.

A partir destes casos base, procedemos à utilização de uma estratégia de fazer *shifts* às matrizes anteriores para, de maneira muito mais simples, conseguir criar as das restantes decisões. Existem apenas dois cenários diferentes, o caso em que uma filial dá carros a outra e o caso em que uma filial recebe carros de outra.

- **Caso em que recebe**: No caso de uma filial receber N carros, o *shift* é feito para baixo. As primeiras N linhas da matriz passam a ser impossíveis uma vez que nunca se vai começar num estado menor que N. As restantes, como indicado, assumem o valor da Nésima linha anterior (ex. a linha 3 passa a ser igual à linha 3 - N).
- **Caso em que dá**: No caso de uma filial dar N carros, o *shift* é feito para cima. Torna-se impossível acabar num estado maior que o máximo menos o número de carros que se dá, ou seja, 12 - N. Da mesma maneira, as restantes linhas assumem o valor da Nésima linha que se segue (ex. a linha 0 passa a ser igual à linha 0 + N).

Nota: Note-se que as matrizes de contribuições, depois da fase dos shifts, sofreram ainda a consideração dos custos de transporte.

Como exemplo, apresentamos um esboço parcial da nossa rede de programação dinâmica para o caso em que uma qualquer filial recebe um carro. Neste cenário, torna-se impossível começar um estágio com 0 carros disponíveis.

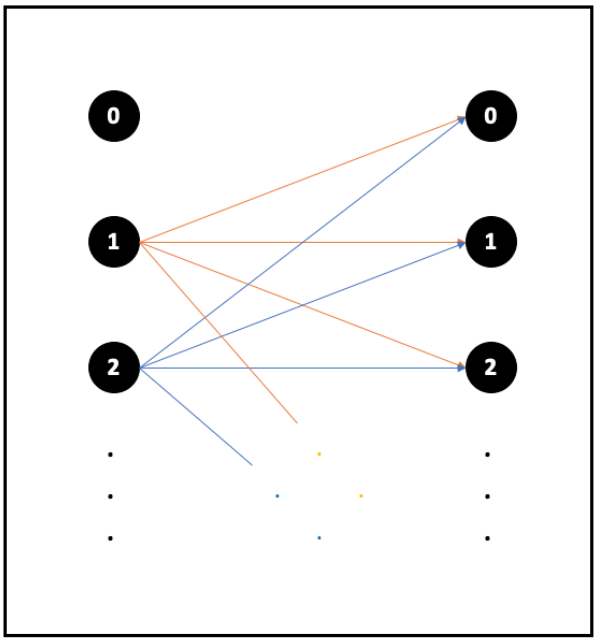

Figura 1: Esboço da rede de programação dinâmica

Com todas as matrizes, separadas por filial, o próximo passo lógico é fazer merge das mesmas numa só, combinando cada valor de uma com todos da outra. Assim, através de cada duas matrizes 13 por 13, criamos uma 169 por 169.

A partir daqui começa-se o algoritmo que nos leva ao resultado que procuramos. Este consiste em repetir o mesmo processo até a solução convergir, acabando aí o processo. Eis os passos que seguimos em cada iteração:

- **1**: Obtemos a esperança da contribuição de cada estágio, **Q**, através das matrizes de transições e das de contribuições obtemos a matriz Q.
- **2**: Obtemos o produto da esperança do total da contribuição quando o sistema parte de (n,i) e segue uma política ótima da última iteração com as transições, **Pn - F(n-1))**.
- **3**: Calculamos a esperança do total da contribuição, **V**, através do valor calculado em **2** e da esperança da contribuição de cada estágio, **Q**.
- **4**: Calculamos a esperança do total da contribuição quando o sistema parte de (n,i) e segue uma política ótima, **F**, através de **V**, obtendo o maior valor entre todas as decisões para cada transição.
- **5**: Subtraímos o **F** desta iteração pelo da anterior de modo a obtermos **D**.

- **6**: Caso **D** tenha convergido paramos o algoritmo. Caso não tenha convergido, avançamos para a próxima iteração.

4 Interpretação e análise crítica do resultado obtido

Uma vez feito todo o processo de modelação do problema, foi necessário recorrer a ferramentas para auxilio do cálculo da política a adoptar. Para isso, foi usada como ferramenta a linguagem de programação *Java*, onde foi muito fácil separar a parte do código que calcula probabilidades da parte do código que usa essas mesmas para iterar, usando o algoritmo estudado de programação dinâmica estocástica.

Abaixo apresenta-se o resultado devolvido pelo programa após onze iterações. É apresentado em forma de matriz para mais fácil visualização. O índice de cada linha representa o número de carros em *stock* da filial 1 e, os índices das colunas, o número de carros em *stock* da filial 2. Dado isto, o resultado deve ser interpretado, por exemplo, da seguinte maneira: $(7, 8) = 6$. Para este caso, quando a filial 1 tem 7 carros em *stock* e a filial 2 tem 8, a decisão a tomar é a 6, que tem um significado específico no contexto do problema como vai ser mostrado posteriormente.

```
Optimal policy = 37.01840690723759
line 0:   0  0  0  0  0  0  0  0  0  0  0  0  0
line 1:   0  0  0  0  0  0  0  0  0  0  0  0  0
line 2:   0  0  0  0  0  0  0  0  0  0  0  0  0
line 3:   0  0  0  0  0  0  0  0  0  0  0  0  0
line 4:   0  0  0  0  0  0  0  0  0  0  0  0  0
line 5:   0  0  0  0  0  0  0  0  0  0  0  0  0
line 6:   0  0  0  0  0  0  0  6  6  6  6  0  0
line 7:   0  0  0  0  0  6  6  6  6  6  6  6  0
line 8:   0  0  0  6  6  6  6  6  6  6  6  6  6
line 9:   0  0  4  6  6  6  6  6  6  6  6  6  6
line 10:  0  2  4  4  4  4  4  4  4  4  4  4  4
line 11:  0  2  2  2  2  2  2  2  2  2  2  2  2
line 12:  0  0  0  0  0  0  0  0  0  0  0  0  0
```

Figura 2: Resultados obtidos

Os valores em cada célula representam um tuplo, (x, y), correspondentes ao número de transições de carro entre filiais, onde o primeiro componente do tuplo indica quantos carros

foram transferidos para a filial 1, caso este valor seja positivo, ou o número de carros que a filial 1 transferiu, caso seja negativo. Analogamente, o segundo componente do par diz respeito à filial 2. Os valores têm o seguinte significado.

- 0 : (0,0)
- 1 : (+1,-1)
- 2 : (-1,+1)
- 3 : (+2,-2)
- 4 : (-2,+2)
- 5 : (+3,-3)
- 6 : (-3,+3)

O grupo achou que os resultados obtidos pelo programa não são muito legítimos. Pode-se observar que só começam a haver transferências quando a filial 1 tem 7 carros em *stock* e a filial 2 tem 5. Ao contrário do que aconteceu, o grupo estava à espera que os valores fossem mais equilibrados, isto é, para um número de carros em *stock* semelhante, por exemplo, $(7, 7)$, $(8, 7)$, o número transferidos de carros fosse 0. Por outro lado, para casos mais extremos, por exemplo, $(0, 12)$, $(2, 12)$, deveriam haver mais transições de carros por parte da filial que possui um número elevado dos mesmos, uma vez que pode acontecer perda de vendas. No entanto, as probabilidade de entregas e procuras para números mais elevados, é muito mais baixa do que para valores medianos, o que por sua vez, pode levar a não optar uma transição nestes casos. Contudo, mesmo assim, a distribuição das transferências não parece a mais correta.

De qualquer maneira, o lucro óptimo que o empresário tem com esta política de transferência é de 37.02 euros por dia, quer resulta no final do mês, em 1147.62 ao final do mês.

5 Conclusão

O trabalho desenvolvido foi muito elucidativo para perceber como pode ser formulado um problema do mundo real, e recorrente, através de programação dinâmica estocástica. Foi também interessante ver a diferença entre a programação dinâmica determinística, que foi abordada em cadeiras anteriores, e a estocástica.

Embora o grupo achar que os resultados obtidos não estão, na sua totalidade, correctos, o trabalho foi importante para saber modelar um sistema que depende de acontecimentos não determinísticos, estando dependentes de várias probabilidades para os seus acontecimentos. É importante ter estas noções porque, em casos reais, é muito provável desenvolver sistemas que dependam de probabilidades de acontecimentos pois, o mundo que nos rodeia acaba por ser mais estocástico do que determinístico.

6 Anexos

6.1 Ficheiro de input fornecido pelo docente

```
Grupo que inclui o Aluno com o Nº 85700
MEIO-TP1 - Tabelas de probabilidades de pedidos e entregas de automóveis

FILIAL 1
Número de clientes:       ;     0 ;     1 ;     2 ;     3 ;     4 ;     5 ;     6 ;     7 ;     8 ;     9 ;     10 ;     11 ;     12
Probabilidade (pedidos):  ; 0.0392 ; 0.0692 ; 0.1236 ; 0.1360 ; 0.1284 ; 0.1124 ; 0.1108 ; 0.0900 ; 0.0640 ; 0.0580 ; 0.0416 ; 0.0200 ; 0.0068
Probabilidade (entregas): ; 0.0352 ; 0.0700 ; 0.1152 ; 0.1412 ; 0.1452 ; 0.1032 ; 0.1024 ; 0.0892 ; 0.0740 ; 0.0568 ; 0.0376 ; 0.0228 ; 0.0072

FILIAL 2
Número de clientes:       ;     0 ;     1 ;     2 ;     3 ;     4 ;     5 ;     6 ;     7 ;     8 ;     9 ;     10 ;     11 ;     12
Probabilidade (pedidos):  ; 0.0460 ; 0.0816 ; 0.1244 ; 0.1492 ; 0.1136 ; 0.1108 ; 0.1020 ; 0.0876 ; 0.0664 ; 0.0508 ; 0.0384 ; 0.0212 ; 0.0080
Probabilidade (entregas): ; 0.0516 ; 0.1548 ; 0.2132 ; 0.2136 ; 0.1692 ; 0.1092 ; 0.0556 ; 0.0224 ; 0.0072 ; 0.0024 ; 0.0008 ; 0.0000 ; 0.0000
```

Figura 3: Ficheiro de input

6.2 Programa principal

```
1  public static void main(String[] args) throws FileNotFoundException {
2          /* calculate odds and costs */
3          Double[][] probB1 = Odds.deliveryRequestProbs(Odds.b1);
4          Double[][] probB2 = Odds.deliveryRequestProbs(Odds.b2);
5
6          Double[][] costB1 = Odds.costsProbs(Odds.b1);
7          Double[][] costB2 = Odds.costsProbs(Odds.b2);
8
9          /* calculate all the decisions */
10         Double[][][][] detachedDecisions = Odds.allDecisions(probB1,probB2,'
       P');
11         Double[][][] decisions = Odds.mergeDecisions(detachedDecisions, 'P')
       ;
12
13         /* calculate all the costs */
14         Double[][][][] detachedCosts = Odds.allDecisions(costB1,costB2,'C');
15         Odds.transferCost(detachedCosts);
16         Double[][][] costs = Odds.mergeDecisions(detachedCosts,'C');
17
18         /* calculates de best policy */
19         Policy policy = Calculate.policy(decisions,costs);
```

6.3 Algoritmo de programação dinâmica com decisões alternativas e com número infinito de casos

```
1    public static Policy policy(Double[][][] decisions, Double[][][] costs){
2        double decision;
3
4        Double[] Fn = new Double[169];
5        Double[] lastFn = new Double[169];
6        for(int i = 0; i < 169; lastFn[i++] = 0.0);
7        /* Vn for each decision */
8        Double[] Vn = new Double[1183];
9        /* Pn * Fn-1 for each decision */
10       Double[] PnLastFn = new Double[1183];
11       /* Qn for each decision */
12       Double[] Qn = new Double[1183];
13       /* Dn for each decision */
14       Double[] Dn = new Double[169];
15       /* Tracks the decision */
16       Integer[] D = new Integer[169];
17
18       while(true) {
19           /* Calculate Qn */
20           straightMul(decisions, costs, Qn);
21           /* Calculate Pn * Fn-1 */
22           mul(decisions, lastFn, PnLastFn);
23           /* Calculate Vn */
24           sum(Qn, PnLastFn, Vn);
25           /* Calculate Fn */
26           max(Vn, Fn, D);
27           /* Calculate Dn */
28           sub(Fn, lastFn, Dn);
29           /* Checks if it has converged */
30           if (converged(Dn, 0.01)){
31               decision = Dn[0];
32               break;
33           }
34           else System.arraycopy(Fn, 0, lastFn, 0, 169);
35       }
36
37       return new Policy(decision,D);
38   }
39
40   public static void straightMul(Double[][][] Pn, Double[][][] Rn, Double
     [] Qn){
41       double sum;
42       for(int d = 0; d < 7; d++) {
43           for (int i = 0; i < 169; i++) {
44               sum = 0.0;
```

```
                for (int j = 0; j < 169; j++)
                    sum += Pn[d][i][j] * Rn[d][i][j];
                Qn[169 * d + i] = sum;
            }
        }
    }

    public static void mul(Double[][][] Pn, Double[] lastFn, Double[]
    PnLastFn){
        double sum;
        int k;
        for(int d = 0; d < 7; d++) {
            for (int i = 0; i < 169; i++) {
                sum = 0.0;
                for (int j = 0; j < 169; j++)
                    sum += Pn[d][i][j] * lastFn[j];
                PnLastFn[169 * d + i] = sum;
            }
        }
    }

    public static void sum(Double[] Qn, Double[] PnLastFn, Double[] Vn){
        for(int i = 0; i < 1183; i++) {
            Vn[i] = Qn[i] + PnLastFn[i];
        }
    }

    public static void sub(Double[] Fn, Double[] lastFn, Double[] Dn){
        for(int i = 0; i < 169; i++)
            Dn[i] = Fn[i] - lastFn[i];
    }

    public static void max(Double[] Vn, Double[] Fn, Integer[] D){
        double max;
        int d = -1;

        for(int i = 0; i < 169; i++){
            max = Double.MIN_VALUE;
            for(int decision = 0; decision < 7; decision++)
                if( Vn[169 * decision + i] > max ) {
                    max = Vn[169 * decision + i];
                    d = decision;
                }
            Fn[i] = max;
            D[i] = d;
        }
    }

    public static boolean converged(Double[] Dn, double dx){
```

```
93          boolean conv = true;
94          double n = Dn[0];
95
96          for(int i = 1; i < 169; i++)
97              if (Math.abs(n - Dn[i]) > dx) {
98                  conv = false;
99                  break;
100             }
101         return conv;
102     }
```

6.4 Cálculo de probabilidades e de contribuições

```
public static Double[][] deliveryRequestProbs(Double[][] probs){
        Double[][] p = new Double[states][states];
        int i,j,k,d,r;
        int minRequest, maxDelivery, minDelivery;
        for(i = 0; i < states; i++){
            for(j = 0; j < states; j++){
                p[i][j] = 0.0;
                maxDelivery = j;
                minDelivery = (i > j) ? 0 : ( j - i );
                minRequest  = (j > i) ? 0 : ( i - j );
                System.out.print("P(" + i + "," + j + ") = ");
                /* x -> 12 */
                for(d = minDelivery, r = minRequest; d < maxDelivery; d++, r
    ++){
                    if( j == states - 1 )
                        for(k = d; k < states; k++) {
                            System.out.print("P(" + r + ")*E(" + k + ") + ")
    ;
                            p[i][j] += probs[REQUEST][r] * probs[DELIVERY][k
    ];
                        }
                        /* caso base */
                    else {
                        System.out.print("P(" + r + ")*E(" + d + ") + ");
                        p[i][j] += probs[REQUEST][r] * probs[DELIVERY][d];
                    }
                }
                    /* not done requests */
                for( ; r < states; r++) {
                    System.out.print("P(" + r + ")*E(" + d + ") + ");
                    p[i][j] += probs[REQUEST][r] * probs[DELIVERY][d];
                }
                System.out.println();
            }
        }
        return p;
    }

    public static Double[][] merge(Double[][] m, Double[][] t, char type){
        Double[][] p = new Double[totalStates][totalStates];
        int i = -1,j;
        int mi, mj, ti, tj;

        for(mi = 0; mi < states; mi++)
            for(mj = 0; mj < states; mj++){
                i++;
                for (ti = 0, j = 0; ti < states; ti++)
```

```
45                      for (tj = 0; tj < states; tj++, j++)
46                          if( type == 'P')
47                              p[i][j] = m[mi][ti] * t[mj][tj];
48                          else
49                              p[i][j] = m[mi][ti] + t[mj][tj];
50              }
51          return p;
52      }
53
54      public static Double[][] decisionMakerPositive(Double[][] probs, int
    decision, char type){
55          int i, dx = Math.abs(decision);
56          Double[][] p = new Double[states][states];
57
58          for(i = dx; i < states; i++)
59              System.arraycopy(probs[i-dx],0, p[i],0, states);
60
61          for(i = 0; i < dx; i++) {
62              Arrays.fill(p[i], 0.0);
63          }
64
65          return p;
66      }
67
68      public static Double[][] decisionMakerNegative(Double[][] probs, int
    decision, char type){
69          int i, dx = Math.abs(decision);
70          Double[][] p = new Double[states][states];
71
72          for(i = 0; i < states - dx; i++)
73              System.arraycopy(probs[i + dx],0, p[i],0, states);
74
75          for(i = states - dx; i < states; i++) {
76              Arrays.fill(p[i], 0.0);
77          }
78
79          return p;
80      }
81
82      public static Double[][][][] allDecisions(Double[][] probsB1, Double[][]
     probsB2, char type){
83          Double[][][][] decisions = new Double[DECISIONS][BRANCHES][states][
    states];
84
85          decisions[0][0] = Arrays.stream(probsB1).map(Double[]::clone).
    toArray(Double[][]::new);
86          decisions[0][1] = Arrays.stream(probsB2).map(Double[]::clone).
    toArray(Double[][]::new);
87
```

```
88          for(int d = 1, i = 1; i < DECISIONS; i+=2, d++){
89              /* (+d,-d) */
90              System.out.println("D" + i + ": (+" + d + ",-" + d + ")");
91              decisions[i][0] = decisionMakerPositive(probsB1,d,type);
92              decisions[i][1] = decisionMakerNegative(probsB2,d,type);
93              /* (-d,+d) */
94              System.out.println("D" + (i+1) + ": (-" + d + ",+" + d + ")");
95              decisions[i+1][0] = decisionMakerNegative(probsB1,d,type);
96              decisions[i+1][1] = decisionMakerPositive(probsB2,d,type);
97          }
98          return decisions;
99      }
100
101     public static Double[][][] mergeDecisions(Double[][][][] decisions, char
        type){
102         Double[][][] m = new Double[DECISIONS][totalStates][totalStates];
103
104         for(int d = 0; d < DECISIONS; d++)
105             m[d] = merge(decisions[d][0],decisions[d][1], type);
106
107         return m;
108     }
109
110     public static Double[][] costsProbs(Double[][] probs){
111         Double[][] c = new Double[states][states];
112         int i,j,k,d,r;
113         int minRequest, maxDelivery, minDelivery;
114         for(i = 0; i < states; i++){
115             for(j = 0; j < states; j++){
116                 c[i][j] = 0.0;
117                 maxDelivery = j;
118                 minDelivery = (i > j) ? 0 : ( j - i );
119                 minRequest  = (j > i) ? 0 : ( i - j );
120                 if( j > 8 )
121                     c[i][j] -= 10;
122                 for(d = minDelivery, r = minRequest; d < maxDelivery; d++, r
    ++)
123                     if( j == states - 1 )
124                         for(k = d; k < states; k++)
125                             c[i][j] += (probs[REQUEST][r] * probs[DELIVERY][
    k]) * (30 * r);
126                     else
127                         c[i][j] += (probs[REQUEST][r] * probs[DELIVERY][d])
    * (30 * r);
128                 for(; r < states; r++)
129                     c[i][j] += (probs[REQUEST][r] * probs[DELIVERY][d]) *
    (30 * r);
130             }
131         }
```

```
132         return c;
133     }
134 
135     public static void transferCost(Double[][][][] costs){
136         int c = 0;
137         for(int d = 1; d < DECISIONS; d += 2) {
138             c++;
139             for (int i = 0; i < states; i++)
140                 for (int j = 0; j < states; j++) {
141                     costs[d][1][i][j] -= TRANSFER * c;
142                     costs[d + 1][0][i][j] -= TRANSFER * c;
143                 }
144         }
145     }
```