Universidade de São Paulo
Instituto de Matemática e Estatística
MAC 5789 - Laboratório de Inteligência Artificial

Exercício Programa 2: Orquestrador de elevadores

Autor:

Walter Perez Urcia

São Paulo

Maio 2015

Resumo

Neste trabalho o objetivo foi implementar um orquestrador para despachar elevadores utilizando heurísticas e tendo como critérios a redução de consumo de energia, o aumento do confort e satisfação dos usuários e a combinação de ambos. A primeira parte do trabalho consiste em explicar a data de entrada e o gerador de dados. A seguinte parte consiste em explicar as funções utilizadas para satisfazer cada um dos critérios e a implementação do algoritmo. Por último, tem experimentos feitos com o orquestrador. A continuação presentamos os experimentos, resultados e conclusões.

Sumário

Lista de Figuras

1 Problema

1.1 Critérios

O problema que se quer resolver é o seguinte:
Implementar um orquestrador que para um número N de elevadores e M andares, tem que cumplir com os seguintes critérios:

- Redução de consumo de energia: medido pelo percurso total de cada elevador
- Aumento do conforto e satisfação dos usuários: medido pelo tempo de espera e pelo tempo dentro do elevador
- Combinação dos dois critérios anteriores

1.2 Definições previas

Esta implementação deve usar heurísticas para cumplir com os critérios anteriores. Então para entender

Heurística Método com o objetivo de encontrar soluções para um problema, ainda suas respostas não sempre são ótimas.

Função objetivo Função utilizada por uma heurística como critério de comparação.

Espaço de búsqueda Conjunto de estados possíveis desde onde a heurística está em certo momento.

2 Dados de entrada

O orquestrador vai receber os dados de entrada dum archivo de dados gerado aleatoriamente que conterá as chamadas dos elevadores e os tempos de estas. A continuação se explicará os aspectos relevantes dos dados.

2.1 Restrições

O archivo de entrada vai ter a seguinte forma:

1 T
2 NC_1
3 $in1_1 out1_1$
4 $in1_2 out1_2$
5 ...
6 $in1_{NC_1} out1_{NC_1}$
7 .
8 .
9 .
10 NC_T

11 inT_1outT_1
12 inT_2outT_2
13 ...
14 $inT_{NC_T}outT_{NC_T}$

A primeira linha tem T que será o tempo para o orquestrador, ou seja que vai funcionar no intervalo $[0, T]$. Logo se tem T grupos, onde para o grupo t, NC_t é o número de chamadas dos elevadores no tempo t, seguido de NC_t linhas descrevendo cada uma das chamadas como *inout* onde *in* é o andar onde a pessoa está e *out* o andar onde a pessoa quer ir. Em quanto as restrições numéricas temos:

- N será o número de elevadores e M o número de andares
- *in* e *out* sempre estão no intervalo $[1, M]$ e não podem ser iguaís
- O número máximo de chamadas para um determinado $t \in [0, T]$ é max_{NC}
- Todos os elevadores tem a mesma capacidade C
- Todos os elevadores servem todos os andares

2.2 Gerador aleatório de dados

O gerador de dados está escrito na linguagem de programação Java e recebe 4 parâmetros: T , max_{NC} , M e o nome para o archivo gerado. A função principal que gera uma soa chamada é a seguinte:

```
1  public ElevatorCall generateSingleElevatorCall( Integer currentTime ){
2    Integer in = Utils.randomBetween( 1 , numFloors ) ;
3    Integer out = Utils.randomBetween( 1 , numFloors ) ;
4    while( out == in ) out = Utils.randomBetween( 1 , numFloors ) ;
5    return new ElevatorCall( in , out , currentTime ) ;
6  }
```

Esta é usada por a função:

```
1  public List<ElevatorCall> generateElevatorCalls( Integer currentTime ){
2    Integer numCalls = Utils.randomBetween( 1 , maxNumCalls ) ;
3    List<ElevatorCall> lstCalls = new ArrayList<ElevatorCall>() ;
4    for( Integer i = 0 ; i < numCalls ; i++)
5      lstCalls.add( generateSingleElevatorCall( currentTime ) ) ;
6    return lstCalls ;
7  }
```

Esta última função é usada T vezes para gerar todas as chamadas e guardá-las no archivo especificado como parámetro.
O gerador de dados está no paquete Utils, no archivo CallGenerator.java do projeto.

3 Resolução do problema

Nesta parte se explicaram as funções objetivo para cumplir com cada critério sempre considerando N elevadores, M andares e o tempo de entrada e saída dos elevadores como 0.

3.1 Algoritmo

O algoritmo implementado para a resolução do problema é um algoritmo construtivo basado no algoritmo GRASP (Greedy Randomized Adaptative Search Procedure) que utiliza uma função objetivo para encontrar uma solução aproximada ao problema. O pseudocódigo do algoritmo para este trabalho é o seguinte:

Listing 1: Pseudocódigo do algoritmo

```
1  Para cada chamada nova faça
2    solução = vazio
3    Para um número de iterações
4      Para cada elevador i , faça
5        opção = Calcular solução com elevador i
6        Adicionar opção a lista de candidatos possíveis
7      Fim Para
8      Ordenar a lista de candidatos possíveis comparando seus valores para a função objetivo
9      Reducir lista de candidatos possíveis com parámetro alpha ∈ [0, 1]
10     Escolher um candidato aleatoriamente
11     Se a nova solução é melhor que solução até então conhecida
12       grave(solução)
13     Fim Se
14   Fim Para
15 Fim Para
```

Neste caso o espaço de búsqueda será cada elevador, a função objetivo será definida para cada critério e o parámetro alpha vai reducir a lista de candidatos da seguinte forma:

Listing 2: Redução de lista de candidatos

```
1   a = min(lista)
2   b = max(lista)
3   novaLista = {c ∈ lista | FO(c) ≤ a + alpha * (b − a)}
```

A continuação tem a implementação:

Listing 3: Função heurística

```
1  private Building heuristicFunction( Building initialState , List<
       ElevatorCall> lstCalls ){
2    Building currentState = new Building( initialState ) ;
3    for( Integer i = 0 ; i < lstCalls.size() ; i++){
4      Building bestSol = null ;
5      ElevatorCall call = lstCalls.get( i ) ;
6      for( Integer j = 0 ; j < numIterations ; j++){
7        List<Building> options = new ArrayList<Building>() ;
8        for( Integer k = 0 ; k < currentState.getNumElevators() ; k++){
9          Elevator elevator = currentState.getElevators().get( k ) ;
10         if( elevator.getCurrentCapacity() == 0 ) continue ;
11         Building currentSol = new Building( currentState ) ;
12         currentSol.takeNewCall( k ,  call ) ;
13         options.add( currentSol ) ;
14       }
15       if( options.isEmpty() ) continue ;
16       Collections.sort( options ) ;
17       options = filterList( options ) ;
18       Integer selectedIndex = randomBetween( 0 ,  options.size() ) ;
```

```
19          Building selection = new Building( options.get( selectedIndex ) ) ;
20          if( bestSol == null  selection.isBetterThan( bestSol ) )
21            bestSol = new Building( selection ) ;
22        }
23        currentState = new Building( bestSol ) ;
24      }
25      return currentState ;
26    }
```

Além, para o cálculo do função objetivo para uma solução será executado o seguinte método:

Listing 4: Calculo de custo para uma solução

```
1  private Integer getElevatorsCost(){
2    Integer cost = 0 ;
3    for( Elevator e : this.elevators )
4      cost += ( e.getCost() == Integer.MAX_VALUE ? 0 : e.getCost() ) ;
5    return cost ;
6  }
```

A função *getCost* terá a seguinte forma:

Listing 5: Custo para um elevador

```
1  public Integer getCost(){
2    if( COST_BY_DISTANCE && COST_BY_WAITING_TIME ){
3      // Calc cost
4      return ;
5    }
6    if( Simulation.COST_BY_DISTANCE ){
7      // Calc cost
8    }else if( Simulation.COST_BY_WAITING_TIME ){
9      // Calc cost
10   }
11 }
```

As linhas 7, 9 e 3 seram explicadas em 3.2, 3.3 e 3.4 respectivamente.

3.2 Critério 1: Distancia

3.2.1 Descrição

Redução do consumo de energia, que em este caso será medido pelo percurso total de cada elevador.

3.2.2 Função

Como o que se quer é reducir a distancia total recorrida por os elevadores, temos que ver os movimentos que cada elevador faz.

$$\sum_{i=1}^{N}(D_i = Distancia\ total\ recorrida\ por\ elevador\ i)$$

Para calcular D_i temos que ver os movimentos para cada t, mas para cada unidade de tempo, a máxima distancia recorrida vai ser 1, portanto não só temos que ver os movimentos actuais, se não também os seguintes. Então se pode definir D_i da seguinte forma:

$$D_i = m_{stops_i} + \sum_{t=0}^{T} dist_{ti}$$

Onde m_{stops} é a suma dos movimentos que vai ter que fazer para terminar de deixar todas as pessoas em seus respetivos andares logo do tempo T. Então a função objetivo para cumplir o critério será a seguinte:

$$F.O = Min(\sum_{i=1}^{N}(m_{stops_i} + \sum_{t=0}^{T} dist_{ti}))$$

3.2.3 Implementação

Para o calculo da função objetivo não se tem um método definido no código, se não que este valor é atualizado cada vez que se executa a linha 12 de código **??** para algum objeto Building porque este é o método que adiciona a nova chamada na lista de paradas que tem que fazer um elevador e portanto modifica o percurso do elevador. As instruções que vão na linha 7 do código 5 são:

Listing 6: Calculo de custo por percurso do elevador

```
1    Integer previous = this.currentFloor ;
2    this.cost = this.distanceMoved ;
3    for( ElevatorCall call : this.stops ){
4      this.cost += Math.abs( call.getIncomingFloor() - previous ) ;
5      previous = call.getIncomingFloor() ;
6    }
```

Onde *currentFloor* é o actual andar onde o elevador está e *distanceMoved* é a distancia recorrida até agora pelo elevador. Para cada chamada não atendida até esse momento se calcula os próximos movimentos do elevador e adiciona a seu custo.

3.3 Critério 2: Tempo espera

3.3.1 Descrição

Aumento do conforto e satisfação dos usuários que neste caso é medido pelo tempo de espera e pelo tempo dentro do elevador.

3.3.2 Função

Nesta vez se quer reducir o tempo de espera e o tempo no elevador, então podemos fazer o seguinte:

$$\sum_{i=1}^{N}(I_i + O_i)$$

Onde I_i = Tempo de espera total elevador i e O_i = Tempo total no elevador i. A forma implementada para diferenciar os tempos de espera dos tempos no elevador é ter dois tipos de chamadas, uma para as chamadas que não tem sido atendidas e outra para as chamadas que não tem sido entregadas a seu destino. Então podemos definir novamente:

$$I_i = Suma(\{x \mid x \in C_i \wedge x.in \neq x.out\})$$

$$O_i = Suma(\{x \mid x \in C_i \wedge x.in = x.out\})$$

Com C_i o conjunto de chamadas do elevador i.
As chamadas com andar de ingresso (in) e andar destino (out) diferentes são as que suma I_i, e as que tem ingresso e destino igual são as que suma O_i. Por tanto a função objetivo será:

$$F.O. = Min(\sum_{i=1}^{N}(I_i + O_i))$$

3.3.3 Implementação

Da mesma forma que em 3.2 temos que as instruções que vão na linha 9 do código 5 são:

Listing 7: Calculo de custo por tempo de espera do elevador

```
1    this.cost = this.waitingTime ;
2    for( ElevatorCall call : this.stops ){
3      this.cost += call.getWaitingTime() ;
4    }
```

Onde *waitingTime* é o tempo de espera e tempo no elevador total de todas as chamadas que atendeu esse elevador. Além, se adiciona a seu custo o tempo de espera para todas as chamadas não atendidas até esse momento. Este valor é incrementado durante cada instante de tempo que pasa até o tempo T.

3.4 Critério 3: Ambos critérios

3.4.1 Descrição

Combinação dos dois critérios anteriores, ou seja, percurso total e tempo de espera.

3.4.2 Função

Neste caso só vamos ter que sumar ambas funções objetivo descritas em 3.2 e 3.3. Por tanto a função neste caso será:

$$F.O = Min(\sum_{i=1}^{N}(m_{stops_i} + \sum_{t=0}^{T} dist_{ti}) + \sum_{i=1}^{N}(I_i + O_i))$$

3.4.3 Implementação

Para o calculo da função objetivo esta vez só é necessário sumar ambos valores vistos em 3.2 e 3.3. Então as instruções que vão na linha 3 do código 5 são:

Listing 8: Calculo de custo por percurso e tempo de espera do elevador

```
1    Integer previous = this.currentFloor ;
2    this.cost = this.distanceMoved + this.waitingTime ;
3    for( ElevatorCall call : this.stops ){
4      this.cost += Math.abs( call.getIncomingFloor() - previous ) ;
5      this.cost += call.getWaitingTime() ;
6      previous = call.getIncomingFloor() ;
7    }
```

Onde as variáveis são como em seções anteriores.

4 Experimentos e resultados

O algoritmo implementado é dependente principalmente de dois parámetros: alpha e número de iterações, por o que foram feitos alguns experimentos para saber quais são os melhores valores para cada um. Para fazer os experimentos foi gerado um archivo de entrada com os seguintes parámetros:

- $N = 50$
- $M = 50$
- $T = 50$
- $max_{NC} = 20$

Os experimentos foram feitos para os tres critérios mencionados em 3 e para todos os casos os elevadores tem capacidade 20. Os tempos de execução e os custos são comparados em cada um. Finalmente, a quantidade de execuções por experimento é 20 para todos os experimentos.

4.1 Experimentos com Alpha

Os experimentos nesta secção variaram o valor de alpha de 0.05 a 0.95 e tinham o valor de número de iterações igual a 100 para cada um dos tipos de custo. A continuação os resultados.

4.1.1 Para distancia

A figura 1 mostra que o melhor valor tendo como custo a distancia (ou percurso total dos elevadores) é $\alpha = 0.45$. Além, tem um dos melhores tempos de execução como se mostra na figura 2

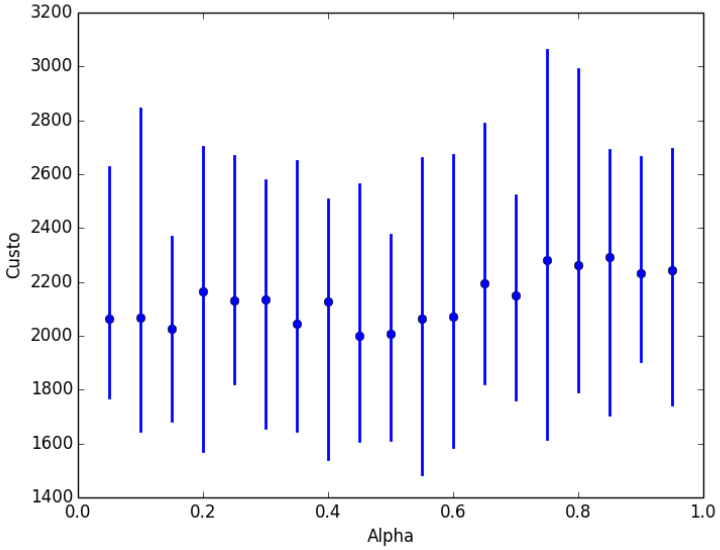

Figura 1: Custos com distancia

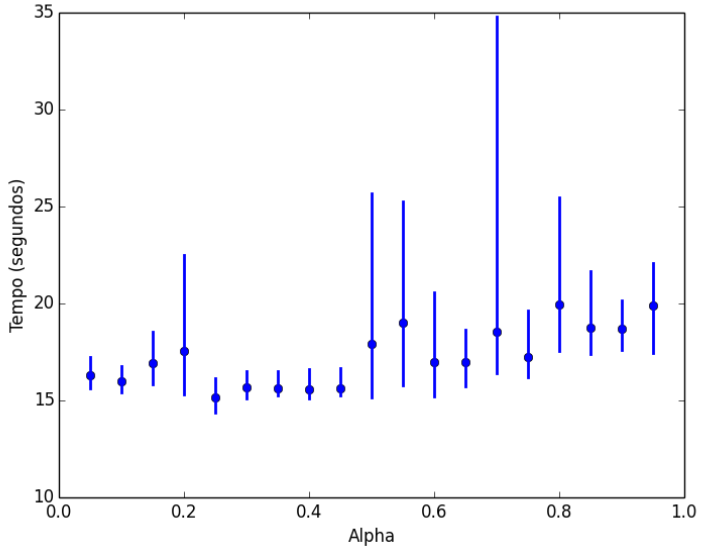

Figura 2: Tempo com distancia

4.1.2 Para tempo de espera e tempo no elevador

Neste experimento o melhor valor para alpha é $\alpha = 0.25$ como mostra a figurañeffig:costwaitingalpha e tem um bom tempo de execução comparado com os demais.

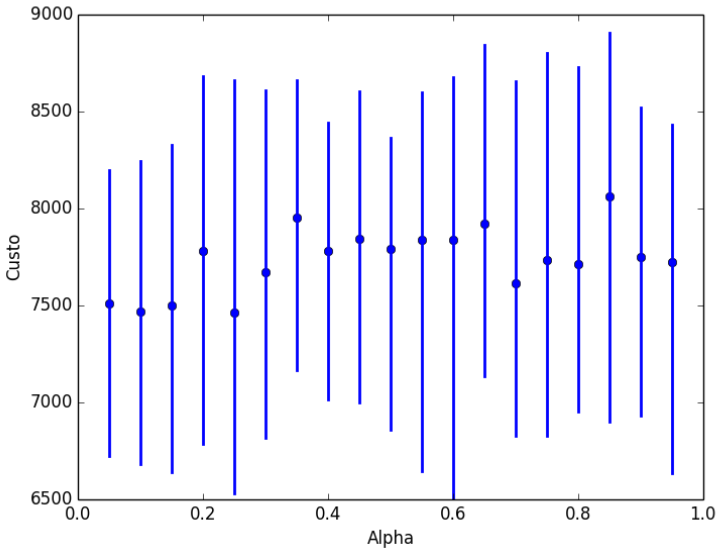

Figura 3: Custos com tempo de espera e tempo no elevador

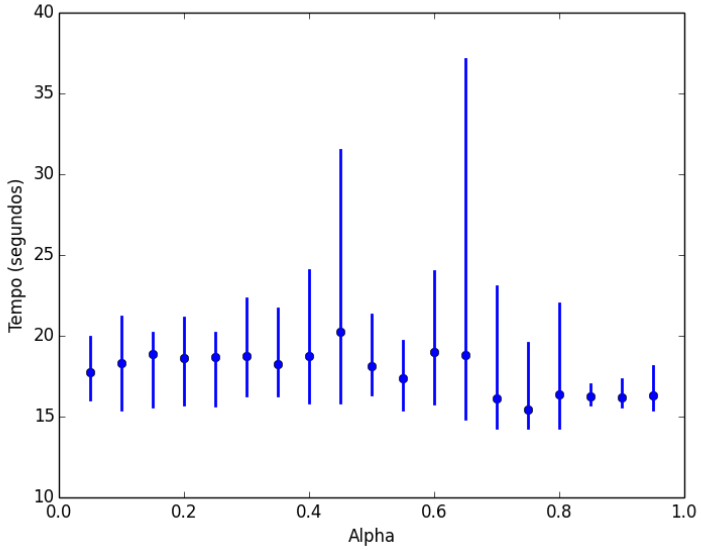

Figura 4: Tempo com tempo de espera e no elevador

4.1.3 Para ambos critérios

Por último, a figura 5 mostra que o melhor valor para usar com ambos critérios para calculo de custo é $\alpha = 0.4$ porque tem o menor valor para custo e também a melhor media, mas tem um dos maiores tempos de execução em promedio (ver figura 6).

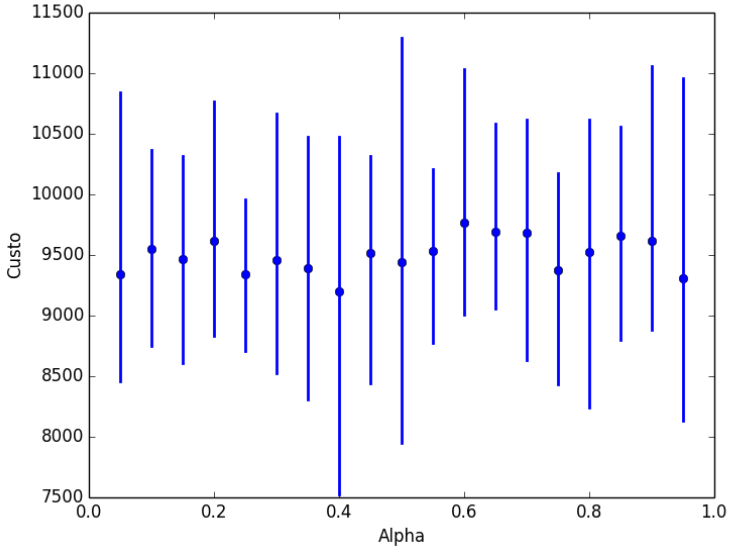

Figura 5: Custos com ambos critérios

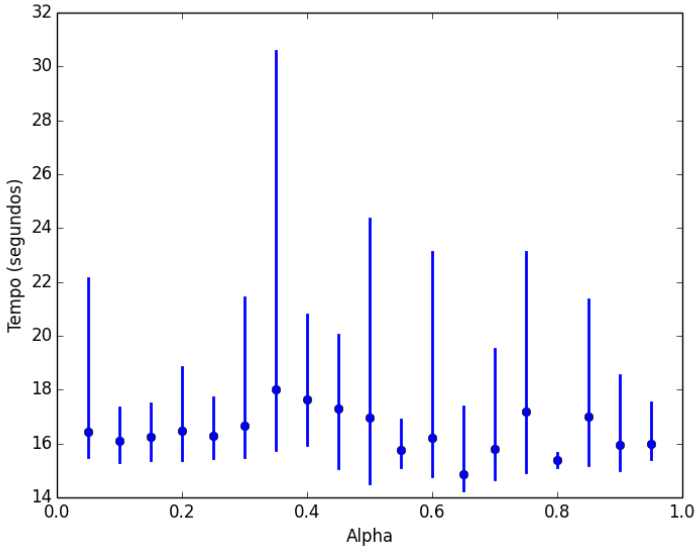

Figura 6: Tempo com ambos critérios

4.2 Experimentos com número de iterações

Da mesma forma que em 4.1 temos experimentos com o número de iterações, esta vez os valores de número de iterações variaram de 100 a 1000. Além, o valor de alpha foi o melhor valor resultado dos experimentos 4.1. A continuação os resultados.

4.2.1 Para ambos critérios

Por último, a figura 7 mostra que o melhor valor para usar com ambos critérios para calculo de custo é número de iterações igual a 500, mas os valores para todos os casos não são muito diferentes. Além, o tempo de execução é consideravelmente maior cada vez que o número de iterações é maior.

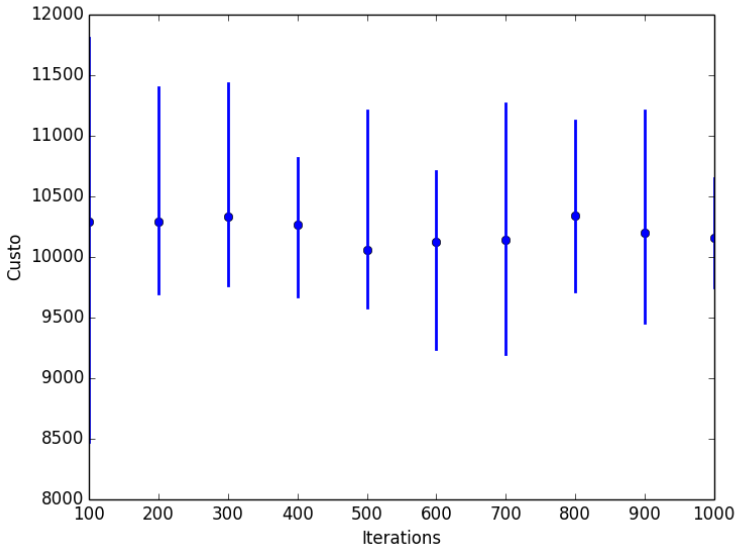

Figura 7: Custos com ambos critérios

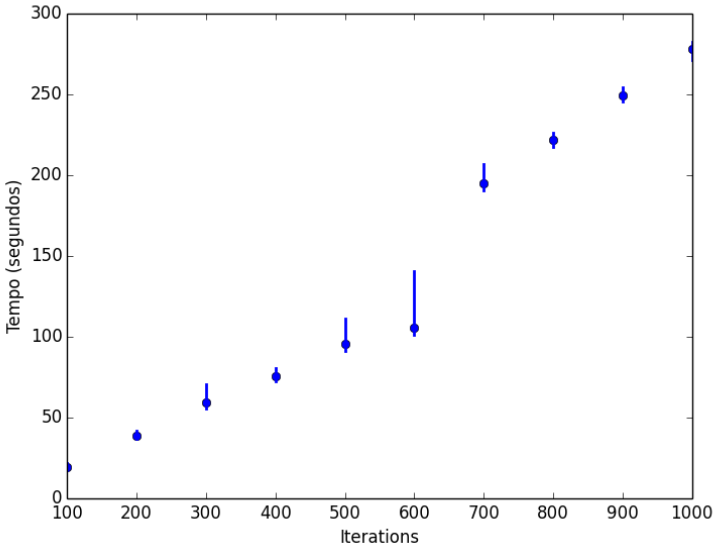

Figura 8: Tempo com ambos critérios

5 Conclusões

Pode-se concluir que:

- O valor de alpha não é irrelevante para o tempo de execução, mas o número de iterações se é muito importante
- Com valores maiores para o número de iterações os resultados não variam muito, só adiciona tempo de execução
- Ao ter partes aleatorias o algoritmo não sempre dá uma solução muito ótima
- Os valores ótimos para el calculo de custo com percurso total dos elevadores são $\alpha = 0.45$
- Os valores ótimos para el calculo de custo com tempo de espera e tempo no elevador são $\alpha = 0.25$
- Os valores ótimos para el calculo de custo com ambos critérios anteriores são $\alpha = 0.4$ e número de iterações igual a 500