

**3ºano – MIEIC – 19 de Maio 2018**

**Otimização na Organização de um Jantar**

Relatório

Inteligência Artificial

Grupo B2\_3

**Bruno Alexandre Oliveira Dias, up201504859@fe.up.pt**

**Fernando André Bezerra Moura Fernandes, up201505821@fe.up.pt**

**Maria Eduarda Santos Cunha, up201506524@fe.up.pt**

# Índice

[1. Objetivo 3](#_Toc514457867)

[2. Especificação 3](#_Toc514457868)

[2.1. Abordagem 3](#_Toc514457869)

[3. Desenvolvimento 7](#_Toc514457872)

[4. Experiências 7](#_Toc514457873)

[5. Conclusões 7](#_Toc514457874)

[6. Melhoramentos 7](#_Toc514457875)

[7. Recursos 8](#_Toc514457876)

[8. Apêndice 8](#_Toc514457877)

# Objetivo

Este trabalho tem como objetivo resolver o problema de otimização da distribuição de pessoas por mesas num jantar solidário, onde centenas de pessoas se registaram. Idealmente, os grupos de pessoas que se inscreveram juntas não deverão ser separados e é importante a afinidade entre pessoas de grupos diferentes quando juntas na mesma mesa. Essa afinidade é determinada segundo interesses ou características em comum: idade, trabalho, hobby, interesse e família.

Pretende-se estabelecer quantas mesas de cada tamanho devem ser utilizadas e a respetiva distribuição das pessoas pelos lugares.

# Especificação

Análise detalhada do tema, ilustração de cenários, explicação datasets...

Abordagem: técnicas, algoritmos e sua breve explicação, esquemas de representação do conhecimento, métricas, heurísticas, gramáticas, arquiteturas...

O problema em questão inclui-se nos problemas de otimização dado que se pretende maximizar a afinidade existente entre pessoas no conjunto de todas as mesas.

## 2.1. Abordagem

## Representação do problema:

De forma a representar o problema proposto, procedemos à criação de 3 classes *Person*, *Table* e *Dinner*, representativas de cada indivíduo, mesa e cenário de jantar possível respetivamente.

Atribuímos às classes os seguintes atributos:

* *Person*: *id*, *name*, *age*, *group* (id do grupo de pessoas com que se inscreveu), *hobie*, *family*, *job* e *interest*;
* *Table*: *id*, *size* e *people* (lista de pessoas a sentar nessa mesa);
* *Dinner*: *id*, *tables*, *fitness*, *afinity*, *probabilityMin*, *probabilityMax* e *probability*.

Uma vez que a disposição das pessoas pelas mesas tem em conta a afinidade entre elas, procedemos à criação da função *getAfinity(self, person)* na classe *Person* responsável por devolver essa mesma afinidade. Calcula-a fazendo várias comparações entre 2 pessoas e incrementado o atributo *afinity* segundo o seguinte sistema de prioridades definido por nós:

Se as 2 pessoas a ser comparadas...

* Estiverem no mesmo grupo, incrementa 1 unidade;
* Tiverem o mesmo interesse, incrementa 0.5 unidades;
* Tiverem a mesma família, incrementa 0.5 unidades;
* Tiverem a mesma idade, incrementa 0.5 unidades;
* Tiverem o mesmo emprego, incrementa 0.5 unidades;
* Tiverem o mesmo hobby, incrementa 0.5 unidades;

Isto, pois o mais importante é manter um grupo de pessoas inscritas juntas, já que partiu da sua iniciativa própria juntarem-se e, consequentemente, deverão querer manter-se na mesma mesa, em detrimento da companhia de pessoas com quem possam ter outras relações de proximidade. As restantes características têm todas a mesma importância, dado que todas constituem um elemento comum e passível de ser usado como tema de conversa.

Em Person.py:

def getAfinity(self, person):

afinity = 0

if(self.id != -1 and person.id != -1):

if (person.group == self.group):

afinity += 1.0

if (person.interest == self.interest):

afinity += 0.5

if (person.family == self.family):

afinity += 0.5

if (person.age == self.age):

afinity += 0.5

if (person.job == self.job):

afinity += 0.5

if (person.hobie == self.hobie):

afinity += 0.5

return afinity

Como a afinidade total do jantar corresponde à soma da afinidade em cada mesa, *getAfinity(self)* na classe *Table* trata de retornar a afinidade total de uma mesa ao somar as afinidades entre todas pessoas sentadas na mesma e *calcAfinity(self)* na classe *Dinner* soma as afinidades das mesas todas, gerando assim a afinidade total do jantar.

Em Table.py:

def getAfinity(self):

afinity = 0

if (self.size > len(self.people)):

for x in range(0, len(self.people)):

if (x < len(self.people) - 1):

afinity += self.people[x].getAfinity(self.people[x + 1])

else:

for x in range(0, self.size):

if (x < self.size - 1):

afinity += self.people[x].getAfinity(self.people[x + 1])

return afinity

Em Dinner.py:

def calcAfinity(self):

self.afinity = 0

for x in range(0, len(self.tables)):

self.afinity += self.tables[x].getAfinity()

Ainda, é tratado que não se podem sentar mais pessoas numa mesa do que o número de lugares existentes nela, representado por *size*.

## Algoritmo Genético:

A população será constituída por diferentes versões de um jantar, ou seja, diferentes combinações de pessoas nas mesas.

O **cruzamento** entre dois “jantares” é feito trocando entre dois jantares uma pessoa pela mesma no outro jantar, ficando essa pessoa numa posição diferente e, consequentemente, gerando uma nova afinidade.

Em Dinner.py:

def mate(self, other):

firstTableChoiceIndex = choice(range(0,len(self.tables)))

secondTableChoiceIndex = choice(range(0,len(other.tables)))

firstPersonChoiceIndex = choice(range(0,len(self.tables[firstTableChoiceIndex].people)))

secondPersonChoiceIndex = choice(range(0,len(other.tables[secondTableChoiceIndex].people)))

ownSubs = self.tables[firstTableChoiceIndex].people[firstPersonChoiceIndex]

otherSubs = other.tables[secondTableChoiceIndex].people[secondPersonChoiceIndex]

self.subs(firstTableChoiceIndex,firstPersonChoiceIndex,otherSubs)

other.subs(secondTableChoiceIndex,secondPersonChoiceIndex,ownSubs)

A **mutação** consiste em trocar duas pessoas de sítio dentro de cada versão do jantar.

Em Dinner.py:

def mutate(self):

firstTableChoiceIndex = choice(range(0, len(self.tables)))

secondTableChoiceIndex = choice(range(0, len(self.tables)))

firstPersonChoiceIndex = choice(range(0, len(self.tables[firstTableChoiceIndex].people)))

secondPersonChoiceIndex = choice(range(0, len(self.tables[secondTableChoiceIndex].people)))

ownSubs = self.tables[firstTableChoiceIndex].people[firstPersonChoiceIndex]

self.subs(secondTableChoiceIndex,secondPersonChoiceIndex, ownSubs)

Em genetic.py:

def mutate(population):

probMut = 0.2

for x in range(0, len(population)):

r = random()

if(r < probMut):

population[x].mutate()

O processo descrito é repetido ao longo de uma série de iterações, cujo critério de paragem é quando a diferença entre a afinidade máxima anterior e a atual for mínima ou nula.

# Desenvolvimento

Ferramentas/APIs utilizadas, linguagens de programação, ambiente de desenvolvimento (SO, IDE...)

Estrutura da aplicação, módulos, diagrama de classes...

Detalhes relevantes da implementação

# Experiências

Objetivo de cada experiência

Resultados

# Conclusões

Análise dos resultados das experiências levadas a cabo

# Melhoramentos

Sugestões para trabalho futuro

# Recursos

Biblioteca

Software

**Contributo dos elementos do grupo para o projeto final:**

|  |  |
| --- | --- |
| Bruno Dias | X% |
| Fernando Fernandes | X% |
| Maria Eduarda Cunha | X% |

# Apêndice

Manual do utilizador (sucinto)