

**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](#_Toc514592539)

[2. Especificação 3](#_Toc514592540)

[2.1. Abordagem 3](#_Toc514592541)

[2.1.1. Representação do problema 3](#_Toc514592542)

[2.1.2. Algoritmo Genético 6](#_Toc514592543)

[2.1.3. Simulated Annealing 7](#_Toc514592544)

[3. Desenvolvimento 8](#_Toc514592545)

[3.1. Ferramentas/APIs Utilizadas 8](#_Toc514592546)

[3.2. Estrutura da Aplicação 8](#_Toc514592547)

[4. Experiências 8](#_Toc514592548)

[5. Conclusões 10](#_Toc514592549)

[6. Melhoramentos 11](#_Toc514592550)

[7. Recursos 11](#_Toc514592551)

[8. Apêndice 12](#_Toc514592552)

# 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

## 2.1.1. 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*.

Por fim, tentamos procurar a solução ótima através dos 2 algoritmos Algoritmo Genético e Simulated Annealing, sendo que em ambos a população será constituída por diferentes versões de um jantar, ou seja, diferentes combinações de pessoas nas mesas.

## 2.1.2. Algoritmo Genético

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)

Em genetic.py:

def mate(selectedPopulation):

cruzProb = 0.5

l = len(selectedPopulation)

selectedForMating = list()

for x in range(0, l):

r = random()

if (r < cruzProb):

selectedForMating.append(selectedPopulation[x])

l2 = len(selectedForMating)

for y in range(0, l2,2):

if(y == l2-1):

selectedPopulation[y].mate(selectedPopulation[y])

else:

selectedPopulation[y].mate(selectedPopulation[y + 1])

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.

## 2.1.3. Simulated Annealing

Cenas cenas cenas.

# Desenvolvimento

## 3.1. Ferramentas/APIs Utilizadas

Desenvolvemos o projeto no Sistema Operativo Ubuntu 18.04, no IDE PyCharm e a linguagem de programação foi Python.

## 3.2. Estrutura da Aplicação

Des

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

Detalhes relevantes da implementação

# Experiências

No quadro seguinte, estão apresentados os diferentes valores de afinidade total do jantar obtidos para cada uma de 6 experiências repetidas 5 vezes segundo os algoritmos Algoritmo Genético e Simulated Annealing.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algoritmo Genético | | | | | |
| Experiência 1 | 12 pessoas inscritas em 6 grupos, 3 mesas de 5 lugares cada | | | | |
| Afinidade Total: | 5 | 3.5 | 4.5 | 9 | 6 |
| Experiência 2 | 15 pessoas inscritas em 6 grupos, 3 mesas de 5 lugares cada | | | | |
| Afinidade Total: | 9 | 8 | 8.5 | 5.5 | 9 |
| Experiência 3 | 12 pessoas inscritas em 5 grupos, 3 mesas de 5 lugares cada | | | | |
| Afinidade Total: | 5.5 | 7 | 5 | 8 | 6.5 |
| Experiência 4 | 15 pessoas inscritas em 5 grupos, 3 mesas de 5 lugares cada | | | | |
| Afinidade Total: | 10.5 | 5.5 | 11.5 | 5 | 7.5 |
| Experiência 5 | 45 pessoas inscritas em 9 grupos, 9 mesas de 5 lugares cada | | | | |
| Afinidade Total: | 30 | 28.5 | 41 | 28 | 33.5 |
| Experiência 6 | 45 pessoas inscritas em 6 grupos, 9 mesas de 5 lugares cada | | | | |
| Afinidade Total: | 26.5 | 29.5 | 28.5 | 28 | 33.5 |
|  | | | | | |
| Simulated Annealing | | | | | |
| Experiência 1 | 12 pessoas inscritas em 6 grupos, 3 mesas de 5 lugares cada | | | | |
| Afinidade Total: | 8.5 | 9.5 | 10.5 | 11 | 9.5 |
| Experiência 2 | 15 pessoas inscritas em 6 grupos, 3 mesas de 5 lugares cada | | | | |
| Afinidade Total: | 15.5 | 14 | 17 | 13.5 | 18 |
| Experiência 3 | 12 pessoas inscritas em 5 grupos, 3 mesas de 5 lugares cada | | | | |
| Afinidade Total: | 11 | 9.5 | 11.5 | 9.5 | 9 |
| Experiência 4 | 15 pessoas inscritas em 5 grupos, 3 mesas de 5 lugares cada | | | | |
| Afinidade Total: | 14 | 13 | 14.5 | 15.5 | 13.5 |
| Experiência 5 | 45 pessoas inscritas em 9 grupos, 9 mesas de 5 lugares cada | | | | |
| Afinidade Total: | 36,5 | 38,5 | 40,5 | 43 | 42,5 |
| Experiência 6 | 45 pessoas inscritas em 6 grupos, 9 mesas de 5 lugares cada | | | | |
| Afinidade Total: | 36 | 41 | 35.5 | 37 | 41 |

As experiências 1 e 2 mantêm exatamente as mesmas condições à exceção do número de pessoas inscritas, de forma a mostrar a relação da afinidade conforme o número de pessoas.

As experiências 1 e 3, 2 e 4 ou 5 e 6 mantêm exatamente as mesmas condições à exceção do número de grupos em que as pessoas se inscreveram, de forma a mostrar a relação da afinidade conforme o número de grupos.

# Conclusões

Através das experiências 1 e 2, em que todas as condições são mantidas exatamente à exceção do número de pessoas, conseguimos observar que conforme o número de pessoas inscritas aumenta, também a afinidade total do jantar aumenta.

Através dos pares de experiências 1 e 3, 2 e 4 ou 5 e 6, em que todas as condições são mantidas exatamente à exceção do número de grupos, procurávamos identificar o efeito do número de grupos em que as pessoas se inscreveram na solução fina. No entanto, os resultados são bastante dispersos, sendo que em algumas situações o aumento do número de grupos não afeta a solução, noutros aumenta a afinidade e noutros até a diminui. Isto verificou-se ao longo de mais testes dos que os apenas representados para efeito deste relatório.

Ainda, concluímos que através do algoritmo Simulated Annealing conseguimos chegar a soluções muito mais otimizadas do que com o Algoritmo Genético, sendo que a diferença das médias da mesma experiência é sempre bastante elevada entre ambos, à exceção da experiência 5 em que a diferença foi de apenas 0,7.

# Melhoramentos

Futuramente, poderíamos desenvolver uma interface para que se tornasse mais intuitivo e orgânico para um utilizador correr o programa, mostrar a disposição do jantar final em formato de diagrama e permitir correr ambos os algoritmos devolvendo algumas estatísticas de comparação entre as soluções obtidas e tempo demorado.

Em termos de algoritmos, ainda que estejamos contentes por termos tido a oportunidade de explorar o Algoritmo Genético e o Simulated Annealing,pensamos que seria também uma mais valia implementar alguns algoritmos adicionais que não tivessem sido tão abordados nas aulas para podermos expandir o nosso conhecimento.

# 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)