Universidade do Porto

Faculdade de Engenharia

FEUP

Otimização na Organização de um Jantar

Relatório final

Inteligência Artificial
3º ano do Mestrado Integrado em Engenharia Informática e Computação

Grupo B2_4:
Anabela Costa e Silva - up201506034 - up201506034@fe.up.pt
Beatriz Souto de Sá Baldaia - up201505633 - up201505633@fe.up.pt
João Francisco Veríssimo Dias Esteves - up201505145 - up201505145@fe.up.pt

20 de maio de 2018

Conteúdo

1 Objetivo

Este trabalho é desenvolvido no âmbito da unidade curricular de Inteligência Artificial e tem como principal objetivo o estudo e investigação de algoritmos de otimização de soluções.

No projeto em questão será abordada a otimização da gestão de pessoas, isto é, a distribuição de pessoas e grupos, sendo esta uma atribuição de pessoas pelas mesas de um jantar solidário. Sendo que tal processo ocupa um vasto campo de soluções, devido à dimensão da população alvo e diferentes fatores a tomar em consideração no momento de avaliação e decisão, recorremos a metodologias de otimização de soluções como algoritmos genéticos e arrefecimento simulado para alcançar o objetivo pretendido.

Considerando o grau de complexidade da organização de recursos orientados à gestão de pessoas, temos como objetivo obter um valor ótimo para o problema.

Teremos grupos compostos por pessoas com uma certa idade, área profissional, religião e hobbies. Os membros de um grupo têm de ficar na mesma mesa, existe um número máximo e mínimo de lugares em mesas e também nos são dados os números de mesas de n lugares. Para garantir o conforto dos convidados, os elementos de uma mesa devem ter uma certa afinidade, tomando-se em consideração, para tal efeito, as características de cada pessoa anteriormente referidas.

2 Especificação

2.1 Arquitetura

A arquitetura do nosso trabalho baseia-se nos seguintes pontos:

- Uma classe *Person* que representa uma pessoa e tem os seguintes atributos: o nome da pessoa (*string name*); o grupo a que pertence (*int group*); a sua idade (*int age*); o conjuto de *hobbies* (vector<Hobby> hobbies); a área de trabalho (JobArea job); e a sua religião (Religion religion).

- Uma classe *Group* que representa um grupo de pessoas (poderá haver grupos com apenas uma pessoa) e tem os seguintes atributos: o identificador do grupo (*int id*); o conjunto de membros do grupo (*vector<const Person *> members*); conjuntos com as distribuições/frequências de valores, dentro do grupo, para cada um dos fatores a analisar (*double ageDistribution[4], double jobDistribution[NUMBER_JOBS], double religionDistribution[NUMBER_RELIGIONS], double hobbiesDistribution[NUMBER_HOBBIES]*) sendo que, por exemplo, cada índice do array ageDistribution corresponde a uma faixa etária e o seu valor será a percentagem de pessoas nessa faixa etária dentro do grupo.

- Uma classe *Table* que representa uma mesa e tem como atributo o seu número de lugares (*numberOfSeats*).
- Uma classe *TablesManager* que representa o gestor de grupos pelas mesas, ou seja, tratará da parte lógica do programa, e tem os seguintes atributos: os conjuntos de todas as pessoas, grupos e mesas existentes (*vector<Person> people, vector<Group> groups e vector<Table> tables*, respetivamente); uma matriz com os valores de afinidade entre todos os grupos (*vector<vector<double> > groupsAffinity*) onde cada valor corresponde à afinidade entre dois grupos.
- Um *header file EnumCreation.h* com todos os enumeradores usados no programa e funções que fazem a conversão dos mesmo para *strings* e a correspondência destes com *strings*.

2.2 Ilustração do cenário

2.2.1 Input do programa

Para obtermos a informação útil para o programa, lemos dois ficheiros:

- **Ficheiro com informação sobre as pessoas:** Cada linha deste ficheiro contém a informação sobre uma pessoa seguindo a seguinte estrutura:

  <p_name>;<age>;<g_id>;<job_area>;<hobby1, hobby2, ..., hobbyN>

  Onde o significado de cada parâmetro é, repetivamente, o nome da pessoa, a idade, o identificador do grupo ao qual a pessoa pertence, a área de trabalho e sequência de interesses/ocupações. Um exemplo poderia ser:

  Ana;20;1;Science;Judaism;Fishing,Walking,TeamSports,RentingMovies

- **Ficheiro com informação sobre as mesas disponíveis:** Cada linha deste ficheiro contém a informação sobre um tipo de mesa seguindo a seguinte estrutura:

  <n_seats>;<n_tables>

  Onde o significado de cada parâmetro é, respetivamente, o número de lugares deste tipo de mesa e o número de mesas deste género (com este número de lugares). Um exemplo poderia ser:

  6;3

  ou seja, existem três mesas de seis lugares.

  Outras informações relevantes para o projeto e relativas aos algoritmos usados, como a probabilidade de cruzamento, o número máximo de iterações e o número de genes de cada geração, são fornecidos pelo o

utilizador aquando da iniciação do programa e serão descritos mais a frente.

2.2.2 Parse dos ficheiros

Para o parse do ficheiro relativo às pessoas, temos um número finitos de valores para as características área de trabalho, religião e *hobbies*:

- **Valores possíveis do enumerador *JobArea*:** *Science, Education e Politics;*
- **Valores possíveis do enumerador *Religion*:** *Christianity, Gnosticism, Islam, Judaism, Buddhism, Hinduism e Atheism;*
- **Valores possíveis do enumerador *Hobby*:** *Reading, WatchingTV, FamilyTime, GoingToMovies, Fishing, Computer, Gardening, RentingMovies, Walking, Exercise, ListeningToMusic, Entertaining, Hunting, TeamSports, Shopping e Traveling;*

2.3 Abordagem/resolução do problema

Como o nosso trabalho consiste na otimização da distribuição de pessoas por mesas, usaremos dois algoritmos: algoritmo genético e algoritmo de arrefecimento simulado.

Inicialmente geramos aleatoriamente uma população válida, isto é, cada indivíduo representa uma configuração possível de distribuição de pessoas por mesas, não ocorrendo casos em que o número de pessoas numa mesa exceda o número de lugares da mesma. Para cada indivíduo desta população aplicamos o segundo algoritmo referido e após ser atingida a condição de paragem deste obtemos uma população ótima, segundo o arrefecimento simulado, que será fornecida como população inicial para o algoritmo genético.

Segue-se nas próximas subsecções a explicação dos algoritmos e das estruturas, fases e funções dos mesmos.

2.3.1 Estados

Um estado será um vetor em que cada índice corresponde ao identificador de um grupo e o valor nesse vetor à mesa em que esse grupo se encontra.

Assim podemos assegurar que todo o grupo se encontra na mesma mesa, o que é um requisito.

2.3.2 Descrição dos algoritmos de otimização a aplicar

Usaremos dois algoritmos - genético e arrefecimento simulado - que serão analisados individualmente e em cooperação.

O **algoritmo genético** consiste em obter uma população (conjunto de soluções) melhor a partir de uma inicial, alterando-a progressivamente de geração em geração.
Considere-se N o tamanho da população. Primeiro, as K melhores soluções da geração atual são escolhidas para passar por Elitismo para a geração seguinte, caso se opte usar uma política de elitismo (secção 2.3.4). Note-se que, usando um algoritmo genético geracional, é necessário gerar *N-K* descendentes para a geração seguinte para que as populações tenham um tamanho constante.

Seguindo-se a fase do Emparelhamento (secção 2.3.5), esta consiste em escolher pares de indivíduos da geração atual para, na fase de Cruzamento, formarem um novo par de descendentes. Porém, nem todos os pares escolhidos cruzam: esta ocorrência dá-se aleatoriamente de acordo com uma probabilidade de cruzamento, sendo que os pais que não cruzam serão passados diretamente para a seguinte fase. Quanto aos cruzados, foi escolhido o método de dividir os pais em dois na mesma posição escolhida aleatoriamente e a troca destas partes entre pais resultam no par de filhos respetivo. Portanto, desta fase de Cruzamento passam os pais que não cruzaram e os filhos dos que cruzaram.

Segue-se agora a fase de Mutação (secção 2.3.6), importante para manter a diversidade de uma população face à junção de genes das populações em cada geração. No nosso projeto recorreremos a dois tipos de mutação distintos:

- Escolher aleatoriamente um bit do conjunto de cromossomas/indivíduos da população/geração e alterar o seu valor para outro também escolhido aleatoriamente. No contexto do problema, isto equivale a escolhermos aleatoriamente um grupo e mudá-lo para outra mesa também escolhida à sorte.

- Escolher aleatoriamente um cromossoma/indivíduo da população/geração e dois bits do mesmo e depois trocar os valores entre estes dois. No contexto do problema, isto equivale a escolhermos aleatoriamente uma possível solução obtida nesta geração e dela selecionar dois grupos, fazendo-os trocar de mesas entre si.

Todos os elementos da geração em análise são submetidos a este processo, contudo com uma baixa probabilidade de mutação, pelo que normalmente poucos deles a sofrem.

Tendo a população passado pela fase de Mutação, temos agora a geração seguinte e o processo repete-se com esta até que as condições de paragem descritas anteriormente se verifiquem e tenhamos finalmente a população final.

O **algoritmo de arrefecimento simulado** toma um único candidato a solução e altera-o progressivamente. Isto é, a vizinhança de cada estado, como será abordado na secção 2.3.7, é gerada e a cada elemento é dada uma probabilidade de ser o escolhido para a próxima iteração. O algoritmo tem

uma variável, a "temperatura", que começa num valor alto e vai diminuindo. A temperatura influencia a probabilidade de serem escolhidas soluções piores que a atual de forma a não prender a solução a ótimos locais, sendo que uma temperatura alta vai permitir mais facilmente a deslocação da solução para configurações piores que a atual.

A qualidade das soluções, tal como no algoritmo genético apresentado previamente, é calculada através da função de avaliação descrita na secção 2.3.3. Este algoritmo tem a vantagem de inicialmente, com uma alta temperatura, variar muito a solução no espaço de procura, podendo acabar num de vários ótimos locais ao invés de ficar preso em algum local que não seja o ótimo global.

Quanto à diminuição da temperatura a cada iteração, iremos testar múltiplas variações, nomeadamente logarítmica, geométrica e exponencial. Em termos de usar ambos os algoritmos no cálculo da solução, poderemos aplicar primeiro o arrefecimento simulado múltiplas vezes, a fim de obtermos os indivíduos que formarão a população inicial usada no algoritmo genético, que será invocado de seguida. Dada a natureza volátil do arrefecimento simulado nas soluções obtidas, devido à temperatura alta inicial, é de prever que para diferentes invocações deste algoritmo para o mesmo indivíduo inicial resultem soluções distintas.

2.3.3 Função de avaliação

A função de avaliação serve como ponte entre o problema em questão e o algoritmo evolutivo. Avalia os candidatos a solução e diz o quão bons são. Como referido anteriormente, convém os membros da mesma mesa terem uma certa afinidade e tal toma em consideração a idade, área profissional, *hobbies* e religião. Estes fatores terão diferentes pesos pois, de uma forma geral, há características que influenciam mais uma relação que outras. O nosso problema também restringe a distribuição de pessoas na medida que todos os membros de um grupo tem de, obrigatoriamente, ficar na mesma mesa. Desta forma, queremos calcular a afinidade entre grupos. Para tal, para cada um dos fatores já referidos, calculamos as frequências, isto é, cada fator tem um array onde cada índice é um valor possível para esse fator. Nesse índice estará a percentagem de pessoas do grupo que satisfaz esse valor. Esta função de avaliação é aplicada a todos os grupos existentes e no momento de distribuição destes pelas mesas verificamos a diferença dos valores obtidos para cada uma das características entre os grupos que partilharão a mesma mesa. Quanto menor for esta diferença, ou seja, quanto mais perto do zero nos encontrarmos, mais semelhantes são os grupos, portanto há uma maior afinidade entre eles. Assim, maior é o *fitness* pois na função de avaliação de uma característica entre dois grupos, o valor de retorno é 1 menos essa diferença calculada (a diferença só pode variar entre 0 e 1).

Muito frequentemente, uma população apresenta indivíduos inviáveis e

que por isso terão um impacto negativo. Assim, aplicamos uma penalização "mortal"(*"death penalty"heuristic*) onde eliminamos esses indivíduos da população. Ao fazê-lo simplificamos significativamente o algoritmo porque não há a necessidade de os avaliar, penalizar, reparar e compará-los com indivíduos viáveis. Esta estratégia pode resultar bem quando o espaço de pesquisa viável é convexo e contém uma parte razoável de todo o espaço de pesquisa. No nosso caso, um indivíduo inviável é uma distribuição de grupos por mesas que leva à existência de mesas com mais pessoas do que a sua capacidade o permite. Estas distribuições devem ser eliminadas pois nunca serão uma possível solução. Eliminamos em vez de corrigirmos pois esta segunda opção não seria boa, visto que há casos em que não podemos mover pessoas individualmente de uma mesa para outra (os membros de um grupo têm de estar todos na mesma mesa).

2.3.4 Política elitista

Uma variante do processo de construção de uma nova população é poder preservar, para a próxima geração, o(s) melhor(es) indivíduo(s) da geração atual sem lhe aplicar qualquer alteração. Esta estratégia é conhecida como seleção elitista e garante que a qualidade de solução obtida não diminuirá de uma geração para a seguinte.

No nosso projeto, damos a oportunidade ao utilizador de escolher o número da política elitista, sendo que se for zero significa que tal política não se aplica, se for um significa que será selecionado e preservado apenas um, o melhor, indivíduo, e assim consecutivamente.

2.3.5 Função de cruzamento

Para o cruzamento entre duas configurações será lançado um número aleatório entre 0 e N-1, sendo N o número de grupos. Esse número equivale à posição no array do estado, referido anteriormente, a partir da qual os valores serão trocados com os correspondentes do segundo candidato a emparelhamento. Ou seja, tendo os grupos ordenados, é escolhido um à sorte a partir do qual se altera a mesa que lhe seria atribuída com a mesa presente para o mesmo grupo na segunda configuração em questão.

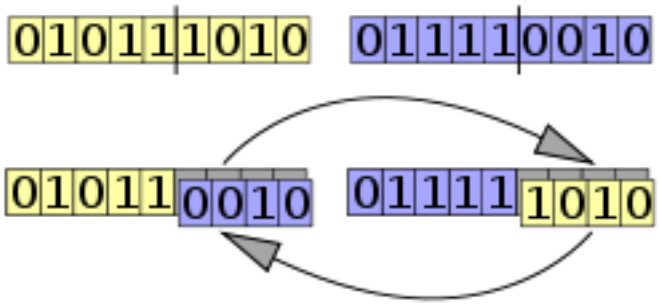


Figura 1: Cruzamento num ponto

2.3.6 Função de mutação

Mutação é uma operação usada para manter diversidade genética de uma geração para outra. É análogo à mutação biológica, alterando um ou mais genes de um cromossoma. A solução pode mudar por completo podendo tornar-se melhor que a anterior. A probabilidade de ocorrer mutação deverá ser baixa porque senão a procura tornar-se-ia meramente aleatória.

No nosso trabalho, a função de mutação altera a mesa em que um grupo se vai sentar. Temos dois tipos de mutação: ou, considerando todos os indivíduos da população, é escolhido aleatoriamente um grupo e alterada a mesa em que este se encontra para uma outra também escolhida à sorte (semelhante ao *bit mutation*), ou é escolhido aleatoriamente um indivíduo da população e dele são selecionados à sorte dois grupos que irão trocar de mesa entre si (semelhante ao *swap mutation*).

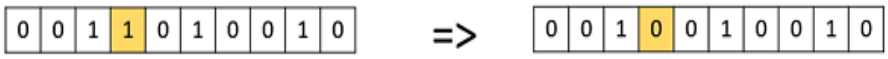


Figura 2: *Bit mutation*

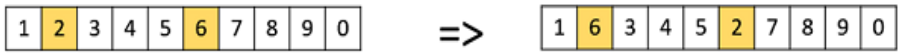


Figura 3: *Swap mutation*

2.3.7 Função de vizinhança

A solução ótima envolve a avaliação dos vizinhos de um estado/iteração. Os novos estados serão gerados a partir de pequenas alterações aplicadas ao elemento selecionado no estado atual a fim de progressivamente melhorar a solução final ao melhorar iterativamente as suas partes.

Assim, a nossa função de vizinhança será trocar a mesa de um grupo. Todos os estados que diferem na alocação de um grupo são então vizinhos.

2.3.8 Critério de paragem

Como será descrito na próxima secção, usaremos dois algoritmos de otimização diferentes: algoritmo genético e algoritmo de arrefecimento simulado (*simulated annealing*).

Para o algoritmo genético, a condição de paragem tomará em conta os seguintes pontos:

- Não se verificarem melhorias na população passadas X iterações consecutivas;
- Ser atingido um determinado número de gerações;
- A função de avaliação ter atingido um determinado valor predefinido como suficientemente ótimo.

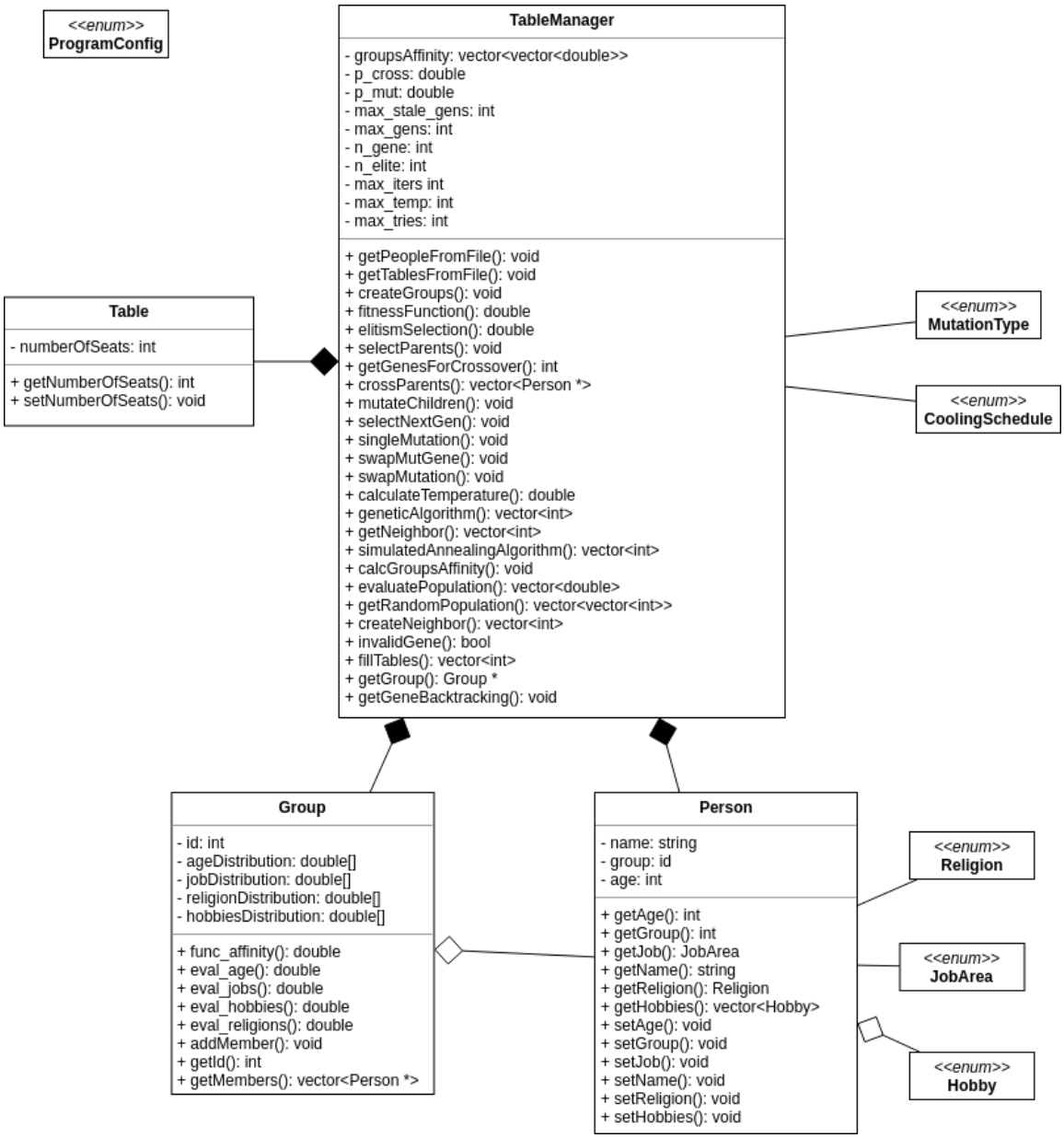
Para o algoritmo de arrefecimento simulado, os pontos a ter em consideração para a paragem será:

- Não se verificar uma mudança da solução passadas X iterações consecutivas;
- Ser atingido um determinado número de iterações;
- Temperatura chegar ao valor 0.

3 Desenvolvimento

O programa foi desenvolvido em C++, em Ubuntu com Visual Studio Code. No início recorremos a Windows, mas devido à falta de compatibilidade completa das threads de C++ no MinGW para Windows mudamos para Ubuntu.

3.1 Diagrama de classes



3.2 Fluxo do programa

3.2.1 Compilar e correr

Para compilar e correr o programa, na pasta /IART executar os seguintes comandos:

```
~/IART:$ g++ -Wall -pthread src/*.cpp -o distribution
~/IART:$ ./distribution <people_file> <tables_file>
```

```
<p_cross> <p_mut> <n_elite> <max_stale_gens>
<max_generations> <n_gene> <max_iters> <max_temp>
<schedule> <max_tries> <mut_type> <prog_config>
```

São pedidos muitos parâmetros devido a toda a informação necessária a passar para os dois algoritmos usados:

- people_file: Nome do ficheiro que contém a informação relativa às pessoas que serão sentadas.
- tables_file: Nome do ficheiro que contém a informação relativa às mesas disponíveis.
- p_cross: Probabilidade de cruzamento (para o algoritmo genético).
- p_mut: Probabilidade de mutação (para o algoritmo genético).
- n_elite: Número de indivíduos passados para a próxima geração por seleção elitista (para o algoritmo genético).
- max_stale_gens: Número máximo de sucessivas gerações sem qualquer melhoria (para o algoritmo genético).
- max_generations: Número máximo de gerações a gerar (para o algoritmo genético).
- n_gene: Número de genes de cada geração (para o algoritmo genético).
- max_iters: Número máximo de iterações (para o algoritmo de arrefecimento simulado).
- max_temp: Valor máximo/inicial da temperatura (para o algoritmo de arrefecimento simulado).
- schedule: Tipo de arrefecimento/diminuição do parâmetro temperatura (para o algoritmo de arrefecimento simulado). Pode ser *Logarithmic*, *Geometric* ou *Exponential*.
- max_tries: Número máximo de tentativas (para o algoritmo de arrefecimento simulado).
- mut_type: Tipo de mutação (para o algoritmo genético). Pode ser *Single* ou *Swap*.
- prog_config: Algoritmos a executar. Pode ser 'SimAnneal' para o *simulated annealing*, 'Genetic' para o algoritmo genético, ou 'All' para o primeiro seguido do segundo.

3.2.2 Criar o objeto TableManager

O programa começa por ler os ficheiros com a informação das pessoas e das mesas e cria o objeto TableManager que contém a lógica toda. No seu construtor ele cria e guarda todas os grupos (classe *Group*) e mesas (classe *Table*) e calcula todas as afinidades entre pares de grupos.

3.2.3 Gerar a população inicial

Geramos uma população inicial aleatória e válida, isto é, sem casos em que a capacidade das mesas é excedida. Como já referido anteriormente, cada indivíduo da população é um vetor em que um índice corresponde a um grupo, e o valor nessa posição à mesa em que ele fica.

3.2.4 Chegar a uma resposta com os algoritmos de otimização

Como o utilizador pode escolher que algoritmos usar, para esta etapa temos três cenários:

- **Usar o algoritmo de arrefecimento simulado e, seguidamente, o genético**

  Será aplicado a cada indivíduo da população inicial o primeiro algoritmo. Após se verificar a condição de paragem em todos os indivíduos, obtemos uma população melhor porque deriva do ótimo atingido no arrefecimento simulado. Esta nova população será a primeira geração do segundo algoritmo. Só após se verificar a condição de paragem do algoritmo genético e que podemos selecionar o melhor indivíduo da última geração, ou seja, o que tem maior valor de *fitness*, apresentando em que mesa cada grupo irá ficar.

- **Usar apenas o algoritmo de arrefecimento simulado**

  Aplicamos este algoritmo a todos os indivíduos da população inicial e, ao verificar-se a condição de paragem, selecionamos para solução final o indivíduo com maior valor de avaliação.

- **Usar apenas o algoritmo genético**

  Usamos a população inicial como primeira geração deste algoritmo e, ao verificar-se a condição de paragem, selecionamos para solução final o indivíduo da última geração com maior valor de avaliação.

3.3 Detalhes relevantes da implementação

- **Geração da população inicial**

  Para a população inicial é necessário que ela seja válida, não se excedendo a capacidade das mesas.

Para testarmos a prestação dos algoritmos é preciso ficheiro grandes (com muitas pessoas e mesas), portanto tivemos de criar um script que gera um ficheiro de texto para pessoas e outro para mesas, pois seria impraticável escrevê-los à mão. No entanto, podem ser gerados ficheiros em que as opções de distribuição de grupos sejam muito reduzidas. Caso tal acontecesse, se a população inicial tivesse indivíduos inviáveis, o mais provavel é que os seus vizinhos também o fossem, obtendo-se uma solução impossivel. Daí termos de fazer a verificação de validade.

Se fôssemos apenas atribuindo aleatoriamente mesas aos grupos existentes e verificando se todas estas distribuições eram possiveis (indivíduo válido) poderiamos ficar num ciclo muito longo, devido ao caso de o número de distribuições possíveis ser pequeno em comparação com o número total de configurações. Deste modo, optamos por recorrer a *backtracking* de forma a, de uma maneira rápida, extrairmos todos os indivíduos válidos, e se houverem mais do que os necessários para a população, eliminamos aleatoriamente indivíduos até satisfazermos o número pedido. Cada nível da árvore é um grupo e cada aresta uma mesa. De facto, esta estratégia é bastante rápida nos casos em que o número de opções viáveis não é exageradamente grande, o que, no entanto, pode ocorrer. Portanto, tivemos de atualizar o nosso plano aplicando um *timeout* à pesquisa em *backtracking*. Se este ocorrer, o *handler* do sinal ativará uma *flag* que quando a 1 abandona a pesquisa com retrocesso por geração aleatoria, como tinhamos inicialmente. Isto porquê? Porque se o tempo se esgotou, significa que há um número elevado de opções válidas o que nos permite facilmente gerar um indivíduo válido a partir da atribuição aleatória de mesas a grupos.

- **Calculo da afinidade entre dois grupos**

  A afinidade entre dois grupos é igual à soma dos valores obtidos na análise de cada uma das caracteristicas:

```
double Group::func_afinity(const Group &other) const {
        return this->eval_age(other)
                        + this->eval_jobs(other)
                        + this->eval_hobbies(other)
                        + this->eval_religions(other);
}
```

  O método de análise das caracteristicas é semelhante entre todas estas. Segue-se a função que analisa a semelhança entre os hoobies das pessoas de dois grupos distintos:

```
double Group::eval_hobbies(const Group &other) const {
        double res = 1;
        int notNullFields = 0;
```

```
        vector<double> diffs;
        bool common = false;

        for(int i = 0; i < NUMBER_HOBBIES; i++) {
                if(!(this->hobbiesDistribution[i] == 0 &&
        other.getHobbiesDistribution()[i] == 0)) {
                        notNullFields++;
                        diffs.push_back(
            abs(this->hobbiesDistribution[i] -
            other.getHobbiesDistribution()[i]));
                }
                if(!common && (this->hobbiesDistribution[i] != 0
        && other.getHobbiesDistribution()[i] != 0)) {
                        common = true;
                }
        }
        if(!common) {
                return 0;
        }

        for(unsigned int j = 0; j < diffs.size(); j++) {
                res -= diffs[j] / notNullFields;
        }
        return res;

}
```

Caso, para esta caracteristica, não haja nenhum valor em comum (*!common*) o valor será zero.

A variável *notNullFields* serve para o valor da diferença a subtrair a *res* não ser superior a 1.

- **Matriz de afinidades**

  No construtor do TableManager é criada uma matriz triangular onde ficam guardadas todas as afinidades entre pares de grupo. Assim, sempre que quisermos saber o valor de um indivíduo, só precisamos de consultar esta matriz.

- **Uso de threads**

  No caso do uso do algoritmo de arrefecimento simulado, iniciamos uma thread para cada indivíduo da população inicial para que este algoritmo seja aplicado a todos os genes ao mesmo tempo, atingindo-se uma solução mais rapidamente.

- **Diferentes formas de diminuir a temperatura no algoritmo de arrefecimento simulado**

  No algoritmo de arrefecimento simulado, o utilizador pode optar por três formas de arrefecimento, sendo i a iteração atual:

  - Logarítmica
    $T_i = \frac{\alpha * T_0}{log(1+i)}$, com $\alpha = 1.0$
  - Geométrica
    $T_i = \alpha^i * T_0$, com $\alpha = 0.85$
  - Exponencial
    $T_i = T_0 * e^{-\alpha * iteration}$, com $\alpha = 1.0$

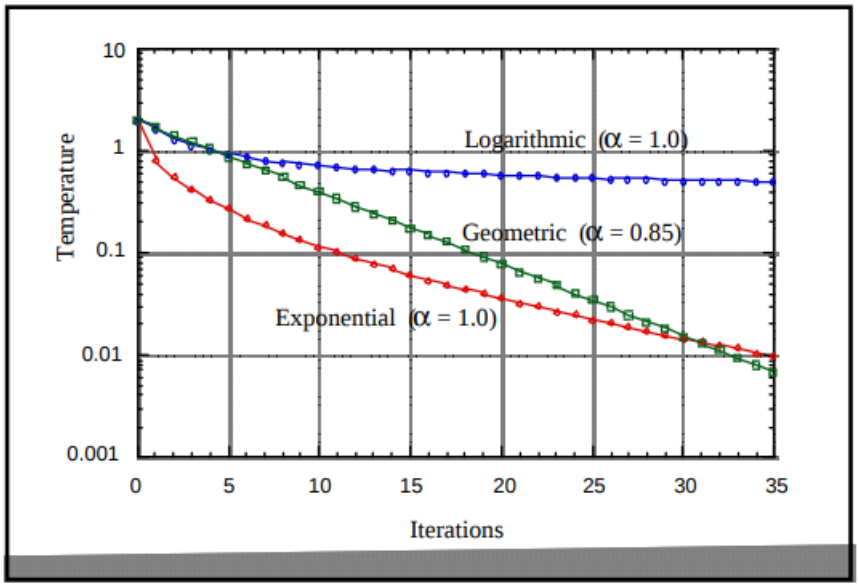


Figura 4: Redução da temperatura, retirado da referência [6]

4 Experiências

No decorrer da construção desta aplicação foram executadas várias experiências. Estas tiveram como objetivo analisar o comportamento dos algoritmos usados em várias circunstâncias.

Analisámos os algoritmos a correr sozinhos e usando resultados do algoritmo de arrefecimento simulado como forma de melhorarmos a população inicial usada pelo algoritmo genético.

Numa experiência alterámos a probabilidade de cruzamento:

	.1		.5		.9	
<p_cross>	Tempo(ms)	Resultado	Tempo	Resultado	Tempo	Resultado
Experiência 1	5608,9	3221.45	3827,2	3258.66	4760,8	3263.92
Experiência 2	3622,6	3256.15	3690,5	3271.77	4754,1	3247.2
Experiência 3	3503,9	3330.84	3587,2	3282.45	5609,2	3383.75
Experiência 4	3541,5	3258.36	3512,1	3299.9	6183,9	3346.82
Experiência 5	3520,6	3212.65	4984,6	3306.64	3544,7	3333.65

Os melhores resultados resultam de uma maior probabilidade de cruzamento, mas são também os mais custosos em tempo de execução. Foram os resultados que levaram maior número de iterações a estabilizar (ter 10 iterações sem melhoria).

Realizámos também experiências alterando a probabilidade de mutação:

	.01		.1		.5	
<p_mut>	Tempo(ms)	Resultado	Tempo	Resultado	Tempo	Resultado
Experiência 1	3554,6	3317.46	5301,4	3375.02	5897,0	3391.47
Experiência 2	6194,1	3222.27	4719,1	3326.15	4448,3	3291.66
Experiência 3	4116,1	3255.33	4478,3	3324.48	6858,3	3286.92
Experiência 4	5300,7	3380.69	5922,1	3359.16	4725,2	3354.99
Experiência 5	9454,9	3320.19	7986,9	3272.49	5019,5	3228.94

E outras alterando o número de elementos passados para a próxima gereção por seleção elitista:

	0		5		10	
<n_elite>	Tempo(ms)	Resultado	Tempo	Resultado	Tempo	Resultado
Experiência 1	4474,9	3280.06	5622,9	3436.22	7068,9	3503.58
Experiência 2	3842,1	3252.13	12989,8	3553.73	10966,3	3445.85
Experiência 3	6777,0	3317.82	10942,9	3573.2	20043,1	3527.33
Experiência 4	5905,3	3311.6	10303,0	3494.31	12403,3	3556.76
Experiência 5	4169,6	3356.13	7955,3	3500.93	8848,9	3544.74

Nota-se uma crescente melhoria dos resultados com número de elementos elitistas maiores.

Alterámos também a forma como realizamos a mutação dos genes:

	Single		Swap	
<mut_type>	Tempo(ms)	Resultado	Tempo	Resultado
Experiência 1	13156,7	3502.39	11123,8	3500.62
Experiência 2	12992,4	3587.18	6164,5	3453.46
Experiência 3	5804,9	3432.08	5605,6	3489.28
Experiência 4	12182,4	3499.78	16477,2	3526.09
Experiência 5	4430,9	3467.22	10590,7	3515.85

Para analisar o algoritmo de arrefecimento simulado realizamos também uma serie de esperiências alteramdo diversas propriedades: a temperatura,

número de tentativas, número de iterações, e a formula como diminuinos a temperatura.

	50		75		100	
max_temp	tempo	Resultado	tempo	Resultado	tempo	Resultado
Experiência 1	43632,5	3453.05	43263,5	3537.05	42994,0	3425.47
Experiência 2	43273,7	3401.55	43370,8	3367.48	43361,7	3502.24
Experiência 3	43663,7	3457.56	43493,8	3418.81	43273,3	3456.45
Experiência 4	43069,8	3460.05	43331,3	3443.71	43372,3	3375.88
Experiência 5	43235,9	3446.07	43119,7	3366.19	42849,7	3342.88

	100		500		1000	
max_tries	tempo	Resultado	tempo	Resultado	tempo	Resultado
Experiência 1	43288,2	3467.51	46790,5	3383.67	47581,4	3433.74
Experiência 2	44872,1	3400.58	46439,2	3416.4	43398,6	3299.04
Experiência 3	45674,8	3363.15	44303,3	3380.32	45598,5	3574.47
Experiência 4	44225,1	3393.2	43594,8	3358.89	45939,1	3372.55
Experiência 5	46478,8	3438.7	43294,2	3408.26	47311,8	3447.96

	Logarithmic		Geometric		Exponential	
schedule	tempo	Resultado	tempo	Resultado	tempo	Resultado
Experiência 1	42363,3	3447.54	43419,8	3400.58	43743,6	3417.91
Experiência 2	42668,8	3526.87	43214,7	3515.49	42917,7	3398.51
Experiência 3	42595,1	3437.18	44319,8	3380.53	42880,2	3409.1
Experiência 4	42208,5	3393.33	43037,4	3458.03	42746,5	3569.1
Experiência 5	42425,0	3381.56	43298,6	3436.11	43156,8	3453.26

	25		50		100	
max_iters	tempo	Resultado	tempo	Resultado	tempo	Resultado
Experiência 1	11195,2	3353.49	22047,2	3320.73	43624,4	3328.28
Experiência 2	11364,8	3386.79	22188,6	3393.8	43451,5	3496.6
Experiência 3	11208,7	3309.01	21797,6	3331.78	44208,3	3378.43
Experiência 4	11537,0	3345.45	21807,8	3354.11	45763,2	3394.88
Experiência 5	11551,0	3424.52	22387,7	3287.01	45092,3	3438.5

Realizámos uma comparação entre os algoritmos usados:

	Genético		Arrefecimento		Ambos			
	tempo	Resultado	tempo	Resultado	tempo			Resultado
					Arrefecimento	Genético	Total	
Experiência 1	20325,2	3635.57	43749,7	3462.9	43459,8	31315,9	74775,8	3669.78
Experiência 2	17299,8	3487.85	43489,9	3404.1	43785,8	5012,0	48797,8	3545.68
Experiência 3	7959,7	3652.07	43573,3	3340.4	43676,3	14481,1	58157,4	3587.55
Experiência 4	20930,6	3530.37	43570,3	3370.89	43543,8	9595,2	53139,1	3470.4
Experiência 5	17266,5	3535.74	43901,2	3384.19	43274,8	4145,5	47420,4	3513.43

O algoritmo genético foi o melhor, demorando menos tempo a chegar a resultados ótimos.

Mesmo a combinação de ambos os algoritmos não deu melhorias aos resultados, mas adicionou muitos custos pois para além da criação da população inicial, adicionou-se o tempo de execução do algoritmo de arrefecimento simulado.

Apesar de com a combinação dos dois algoritmos o tempo passado no algoritmo genético ter diminuido, o aumento de tempo que foi nessesário para a realização do arrefecimento simulado abateu quaisquers melhorias.

5 Conclusões

É interessante compreender detalhadamente e implementar os algoritmos tratados neste projeto face a um problema real, porém, consideramos incompreensível a disparidade de trabalho necessário entre os temas disponíveis devido ao possível uso de *frameworks*. No nosso caso decidimos implementar os algoritmos de raiz, julgando até que não hajam ou não sejam muito conhecidas implementações públicas para estes. Dedicámos um tempo bastante razoável à implementação, enquanto que, por exemplo, grupos com um tema de redes neuronais apenas necessitaram de recorrer ao *TensorFlow* concluindo o projeto em termos de implementação literalmente numa tarde.

Em relação ao nosso produto final, consideramos que foi concluído com sucesso, abordando individualmente e em conjunto os dois algoritmos de otimização estudados nas aulas: o algoritmo genético e o algoritmo de arrefecimento simulado.

6 Melhoramentos

Se tempo o permitisse, apostaríamos em melhorar o script que gera os ficheiros de teste para uma análise e exploração de dados mais precisa. Outro

fator a atacar, mas que, no entanto, também não está diretamente ligado com o tema proposto na unidade curricular, isto é, o problema de otimização, seria criar mais eficientemente uma população inicial aleatória e, ao mesmo tempo, válida, visto que de momento dependemos de *backtracking* e da interrupção causada pelo sinal *ALRM*.

De referir ainda, tipicamente, o que se revela ser difícil em problemas de otimização é a função de avaliação. Desta forma, num trabalho futuro focar-nos-íamos no melhoramento desta função para obtermos dados cada vez mais fidedignos.

Referências

[1] Özgür Yeniay. *PENALTY FUNCTION METHODS FOR CONSTRAINED OPTIMIZATION WITH GENETIC ALGORITHMS*, Hacettepe University, Faculty of Science, Department of Statistics, 06532, Beytepe, Ankara, 2005.
`http://www.cs.cinvestav.mx/~constraint/papers/yeniay05.pdf`

[2] Biruk Girma Tessema. *A SELF-ADAPTIVE GENETIC ALGORITHM FOR CONSTRAINED OPTIMIZATION*, Bahir Dar University, Ethiopia, 2004.
`http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.633.202&rep=rep1&type=pdf`

[3] Alice E. Smith and David W. Coit. *Penalty Functions*, Department of Industrial Engineering University of Pittsburgh, Pittsburgh, Pennsylvania 15261 USA, Revised January 1996.

[4] Vassilios Petridis and Anastasios Bakirtzis. *Varying Fitness Functions in Genetic Algorithm Constrained Optimization: The Cutting Stock and Unit Commitment Problems*, IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS—PART B: CYBERNETICS, VOL. 28, NO. 5, OCTOBER 1998.

[5] TutorialsPoint. *GENETIC ALGORITHMS TUTORIAL*.
`https://www.tutorialspoint.com/genetic_algorithms/index.htm`

[6] Rafael E. Banchs. *SIMULATED ANNEALING*.
`http://rbanchs.com/documents/THFEL_PR15.pdf`