Faculdade de Engenharia da
Universidade do Porto

Mestrado Integrado em Engenharia
Informática e Computação

Inteligência Artificial

Otimização de uma Conferência

Autores
João Conde up201503256
José Borges up201503603
Miguel Mano up201503538

Universidade do Porto
Faculdade de Engenharia
FEUP

Contents

1 Objetivo

O trabalho prático realiza-se no âmbito da unidade curricular de Inteligência Artificial e é proposto que se otimize a programação de uma conferência através de, por exemplo, algoritmos genéticos.

No parecer do grupo, o programa de uma conferência científica é constituída pela apresentação de um conjunto de *papers* (artigos científicos). Um *paper* tem um tema.

Cada apresentação tem um orador que deve ser um dos autores do *paper*. É possível que um autor de um *paper* seja coautor noutros *papers*. Existem dois tipos de *papers*, *full-papers* e *short-papers*, sendo que a única diferença entre os mesmos é a duração da apresentação de cada um. Para um *full-paper* são gastos 30 minutos e para um *short-paper* apenas 20 minutos. As apresentações são então agrupadas em sessões temáticas.

Cada sessão tem uma duração máxima de 2 horas e possui um tema com o qual os *papers* apresentados se devem relacionar. Cada sessão decorre numa das salas disponíveis, sendo que podem ocorrer tantas sessões em paralelo quanto salas.

A conferência decorre ao longo de 3 dias sendo que em cada um destes há no máximo 4 sessões. Estas podem ser intervaladas por breaks (para almoço ou *coffee-breaks*).

Assim, o objetivo é montar o programa da conferência ao longo de 3 dias, sendo que em cada um destes existem sessões e, em cada uma destas, que podem decorrer em paralelo, existem apresentações de diferentes *papers*.

Procura-se garantir sessões equilibradas. Para tal, cada sessão deve ter pelo menos 2 apresentações de *full-papers*. Não só mas também, para que um programa seja válido, é necessário garantir que não há sessões paralelas com apresentadores em comum.

2 Especificação do trabalho

2.1 Abordagem

Devido às diversas especificações do enunciado, o grupo teve algumas dúvidas quanto a como codificar a informação necessária.

Assim, foi necessário definir algumas características como constantes e concentrarmo-nos nas restantes como variáveis. De facto, o utilizador deve, *à priori*, definir uma série de constantes através de macros do programa. Assim, vários aspetos do nosso programa são parametrizados e facilmente mudados, apresentando soluções corretas independentemente das restrições desejadas. Entre tais características, o utilizador pode definir:

1. o limite de gerações
2. o número de indivíduos da população
3. a probabilidade de mutação
4. o número de salas disponíveis
5. o *fitness* objetivo (de 0 a 100%)
6. ativar ou desativar certas componentes do cálculo do *fitness* ou atribuir-lhes diferentes pesos
7. o horário dos *coffee-breaks* e do almoço
8. o horáro das sessões, sendo que considerámos haver uma matinal e uma da tarde

Por fim, deve especificar um ficheiro de input com os *papers* a apresentar.

O desenvolvimento deste projeto foi incremental, ou seja, primeiramente assegurar que o fluxo do programa é funcional, antes de implementar funções de *fitness* ou cruzamento.

Após o grupo ficar satisfeito com a arquitetura criada, rapidamente adicionou um sistema de cruzamento e mutação simples e incorporou uma

heurística de avaliação básica, para inferir sobre a potencialidade de desempenho do algoritmo.

Foi visível, numa fase inicial, um dos principais condicionamentos da tarefa: o facto do cruzamento de dois horários de conferência com *fitness* máxima, ou seja, perfeitamente adaptados às restrições impostas, poder gerar horários catastroficamente incompatíveis. Iremos explicitar, mais à frente neste relatório, como contornámos esta dificuldade.

Deslindando o problema, focou-se em elaborar parâmetros de avaliação de *fitness*, concentrando-se em minimizar colisões de apresentações e, numa fase posterior, garantir o cumprimento das regras das sessões.

A fase final cingiu-se a balancear as probabilidades de cruzamento, mutação e melhorar a função heurística, com o objetivo de convergir para uma solução perfeita o mais rápido possível.

2.2 Representação de um indivíduo

A elevada quantidade de informação a representar num único cromossoma captou, nesta fase, a maioria da atenção do grupo, dadas as vastas possibilidades expostas e a sua direta influência nas fases conseguintes.

Numa etapa preliminar, haver-se-á considerado uma **representação estritamente orientada a objetos**, em que cada cromossoma seria um objeto de uma classe que encapsula toda a informação do indivíduo. Rapidamente foi percetível, na nossa ótica, a **complexidade desta metodologia**, principalmente em operações de cruzamento e mutação.

Numa fase posterior, ficou claro que a quantidade de informação significativa a fornecer ao algoritmo genético é mais reduzida do que o anteriormente expectável. Há, efetivamente, **informação não mutável inerente a cada paper**. Exemplificando, não fará sentido requisitar um orador de um *paper* que não o tem para maximizar o valor de fitness de um dado horário. É, assim, necessário assegurar um conjunto de estratégias que **não modifiquem informação definitiva do *paper***.

Até à etapa da entrega e apresentação intercalares, o grupo haveria recorrido à tradução da população através de **binário**. Esta abordagem foi bem sucedida quando testada numa fase preliminar, porém as constantes traduções de genótipo para fenótipo (e vice-versa) revelaram-se redundantes. De facto, se o problema exige uma representação ordenada e sensível e, para tal, a aplicação de metodologias de cruzamento como *Linear Order Crossover* não são aplicáveis, não se justificava o trabalho extra.

O grupo acabou por relembrar a complexidade observada relativamente à organização por objetos, reconsiderou as alternativas, optando por representar um indivíduo como uma lista de dicionários de Python, cada elemento correspondendo à configuração de certa conferência.

Explicitemos as chaves...

- **Paper** aponta para o objeto Paper associado à conferência, contendo ele mesmo um nome, autor, tema e duração;
- **Day** refere-se ao dia em que a apresentação se insere;

- **Room** refere-se à sala em que a apresentação decorrerá;
- **Time** diz respeito ao bloco de 10 minutos em que se dará o início da talk.

Através desta simples representação foi possível assegurar a validade da informação nas operações de mutação e cruzamento.

2.3 Função de Cruzamento

Para uma adequada seleção da função de cruzamento, é necessário compreender o problema em questão. Após averiguação, concluiu-se que o problema proposto é do tipo *job shop scheduling*, em que tarefas são atribuídas a recursos em períodos particulares. No nosso caso, apresentações são atribuídas a oradores e salas em blocos e dias.

O passo seguinte seria adaptar uma metodologia de cruzamento otimizada para problemas *job shop*. De acordo com *Evolutionary Scheduling*, publicado por *Keshav Dahal, Kay Chen Tan e Peter I.*, **linear order crossover** é eficiente o suficiente para impactar positivamente problemas deste tipo.

Infelizmente, a aliança entre estes dois conceitos, anexada à representação proposta, não se revelou viável. Seria impossível garantir informação válida após cada cruzamento. Exemplificando, LOX poderia tentar cruzar, em indivíduos distintos, um dia por uma sala, resultando em descendentes com um par de dias e sem salas.

A metodologia implementada cinge-se a, por cada par de apresentações entre horários de conferências, ocorrer uma troca de apresentações, mediante uma percentagem estabelecida de 50%, garantindo a unicidade de cada *paper*.

2.3.1 Seleção para Cruzamento

Uma realidade desagradável descoberta aquando do desenvolvimento deste projeto é que o cruzamento de dois indivíduos perfeitos, contra-intuitivamente, poderá resultar em indivíduos sub-par! Para visualizar este problema basta imaginar a troca de horários na FEUP, em que trocar cadeiras de dois horários ótimos poderá gerar conflitos por sobreposição.

Assim, assegurar a imortalidade do melhor horário tornou-se obrigatório, receando gerações de pura aleatoriedade, em que se poderia assistir a evoluções, seguidas de regressões drásticas. O melhor indivíduo de cada geração passa não só por elitismo, imutado, para a geração seguinte como ainda possui a possibilidade de se reproduzir, propiciando uma constante evolução entre gerações, e a retenção da melhor solução possível até ao momento.

2.4 Função de Mutação

A função de mutação encarrega-se de voltar a gerar parâmetros em apresentações de conferência. Houve um grande cuidado em balancear a chance de mutação para permitir tanto o fluxo natural de progressão evolutiva dos indivíduos e resgatar gerações de vales evolutivos.

- Decidiu-se mutar cada indivíduo de uma população consoante a constante das definições de genética MUTATION.
- De seguida, é apenas dada ao cromossoma a oportunidade de mutar um dos seus parâmetros, escolhido aleatoriamente.
- Finalmente, a mutação é aplicada a apenas um dos seus *papers*, selecionado aleatoriamente.

2.5 Função de Avaliação

A função de avaliação procura favorecer as programações da conferência por nós consideradas mais favoráveis e penalizar de forma mais ou menos acentuada as desfavoráveis. Um dos principais desafios foi distinguir a informação a representar no cromossoma e a informação a encargo da função de *fitness*.

Assim, procurar-se-á **penalizar indivíduos que não apresentem características desejáveis**. Para tal, a penalização será tanto maior quanto:

1. Maior o número de conflitos entre apresentações de papers;
2. Maior o número de apresentações que colidem com os tempos de pausa estipulados;
3. Maior o número de oradores em situações de conflito, ou seja, com duas ou mais apresentações atribuídas simultaneamente;

4. Maior o número de sessões que não possuam um tema em comum;

5. Maior o número de sessões com menos de duas apresentações de *full-papers*

Foi-nos vital **inflacionar o peso de restrições específicas**, no cálculo da fitness final. Ilustrando, o cromossoma possuir limites de tempo excedidos poderá não resultar num horário de conferência otimizado, mas válido. Em contraste, o número de conflitos entre apresentações deverá pesar abundantemente, visto a existência de uma única colisão invalidar o cromossoma inteiro. Foi através da implementação destes multiplicadores que nos foi possível convergir para soluções perfeitas.

Optou-se por uma **escala de avaliação percentual**, em que uma pontuação de 0 pontos representa o pior horário hipoteticamente gerável e de 100 um horário que cumpre todas as caraterísticas impostas anteriormente.

2.6 Critérios de paragem

Neste problema de otimização de alocação de horários de uma conferência, poder-se-á considerar um pouco dúbio o critério de paragem.

Contudo, identificaram-se duas principais formas de ordenar a paragem de um algoritmo genético: através de um número limite de gerações ou atingindo um valor de *fitness* estabelecido *à priori*.

O algoritmo conta com duas macros para o efeito, declaradas no ficheiro `macros.py`. GENERATIONS limita o número de gerações, enquanto que DESIRED_FITNESS impõe o limite mínimo ao qual o algoritmo termina e guarda o melhor indivíduo, ou seja, a melhor programação da conferência para uma folha de *excel*.

É também de destacar que o envio do sinal `SIGINT`, comummente associado ao comando `CTRL+C`, terminará o programa e guardará o indivíduo mais adaptado até ao momento.

2.7 Algoritmos de otimização a aplicar

Para a otimização da alocação das diferentes apresentações da conferência utilizou-se um algoritmo genético.

Como referido previamente, a utilização de uma política elitista que não permite a morte do melhor cromossoma em cada geração foi vital para assistir a uma progressão genética.

3 Trabalho efetuado

3.1 Ambiente de Desenvolvimento

O projeto foi desenvolvido nos sistemas operativos de Windows 10 e Manjaro 4.9, mas na realidade o projeto pode ser corrido em qualquer sistema operativo, desde que tenha instalado Python versão 3.6. A linguagem de programação utilizada foi Python.

3.2 Estrutura e módulos de aplicação

3.2.1 paper.py

Ficheiro onde é definida a class Paper. Contém apenas a função `__init__(self, id, title, speaker, themes, duration)`.

3.2.2 crossover.py

Ficheiro responsável por todas as funções relacionadas com o crossover.

`generate_roulette(population, scores)`
Gera uma roleta baseada na pontuação de cada indivíduo. Esta abordagem é baseada no operador genético Fitness Proportionate Selection. Retorna uma lista de objetos individuais e pares de fatias da roleta.

`spin_roulette(roulette, population)`
Roda a roleta previamente gerada x vezes, em que x é o número de cromossomas da população multiplicado por 2. Retorna uma lista de pares de pais que irão participar no crossover.

`xover_parents(mother, father)`
Função feita para realizar o crossover entre dois cromossomas. Retorna um par de filhos resultantes.

`xover_population(couples)`
Realiza o crossover da população inteira de uma geração. Retorna uma lista de filhos resultantes de um crossover.

`immortality_policy(children, scores, fittest)`

A política de imortalidade que preserva o elemento com maior fitness intacto para a próxima geração. Ainda assim, permite crossover com o elemento com maior fitness.

3.2.3 fitness.py

Ficheiro responsável por todas as funções relacionadas com o fitness do projeto.

`calculate_pop_fitness(population)`
Computa a pontuação de fitness da população da geração recebida. Retorna uma lista da pontuação de indivíduos.

`calculate_fitness(individual)`
Aplica um número de testes a um indivíduo providenciado. Retorna a média de pontuações calculadas.

`construct_interval(talk)`
Recebe a entrada de dicionário de uma talk e retorna o intervalo de tempo alocado a esse recurso.

`score_collisions(individual)`
Pontua um indivíduo, baseado no número de talks em conflito.

`count_paper_collisions(intervals)`
Conta colisões entre os intervalos dos papers.

`score_break_collisions(individual)`
Avalia as colisões das talks com os intervalos do horário da conferência.

`score_sessions_theme(individual)`
Penaliza sessões com uma grande variedade de temas diferentes.

`score_speaker_occupation(individual)`
Avalia apresentadores tendo em conta se existe o mesmo apresentador requerido em 2 ou mais salas ao mesmo tempo, por dia.

`score_collisions_speaker(day_talks)`

Avalia apresentadores tendo em conta se existe o mesmo apresentador requerido em 2 ou mais salas ao mesmo tempo.

`score_sessions_balance(individual)`
Avalia se cada sessão que possua apresentações tem pelo menos 2 de *full-papers*

3.2.4 genetic.py

Ficheiro principal do projeto.

`main()`
Ponto de entrada do projeto.

`manage_generation(population)`
Processa cada geração, aplicando fitness, crossover baseado nisso e mutação. Retorna a nova população.

`parse_paper_file(file_path)`
Analisa cada linha de um ficheiro criado pelo utilizador contendo papers. Retorna uma lista de objetos do tipo Paper.

`init_population(papers)`
Gera uma população semi aleatória composta por um número NUMBER_OF_CROMOSSOMES de indivíduos. A população é uma lista de horários de uma conferência, o qual é uma lista de horários de talks.

`init_conference(papers)`
Gera um indivíduo semi aleatório. O genótipo é composto por um dicionário contendo um apontador para um objeto de paper, a sala da talk, o dia e começo da talk em blocos de 10 minutos.

3.2.5 mutation.py

`mutate_room(conference)`
Muta uma sala baseado numa talk. Atribui uma nova baseado nas salas disponíveis.

`mutate_day(conference)`
Muta o dia de uma talk. Atribui um novo baseado nos dias disponíveis.

`mutate_time(conference)`
Muta o tempo de início de uma talk. Atribui um novo baseado nos blocos de tempo disponíveis.

`mutate_conference(conference)`
Muta a conferência. Muta uma talk aleatória na conferência em dia, sala ou tempo de início.

`mutate_population(conference)`
Muta a população. A probabilidade de mutação para cada indivíduo é fixa.

3.2.6 macros.py

Ficheiro com todas as macros utilizadas ao longo do projeto. Permite ao utilizador facilmente aplicar novas restrições ao problema

3.2.7 utilities.py

Ficheiro que contém a função `export_to_spreadsheet`, responsável por exportar a conferência para uma folha de cálculo.

4 Experiências

4.1 Objetivos

Dada a natureza dos algoritmos genéticos, foi-nos fundamental comprovar que a evolução constatada não seria apenas fruto de natureza probabilística.

Adicionalmente, conduziram-se testes para convergir numa regulação mais precisa dos valores de mutação e número de cromossomas.

Foi permitido, a cada experiência, um máximo de 1000 gerações até a execução do programa ser interrompida. Caso um indivíduo perfeito seja encontrado, a execução é quebrada e o número da geração é registada. A experiência é repetida 10 vezes e a média ponderada é o valor de cada célula.

4.2 Resultados

Foi observável, aquando da remoção total de operações de mutação, uma estagnação das gerações num valor de fitness potencialmente elevado, mas não ótimo. Isto é expectável, a falta de diversidade genética entre gerações conduz a um vale evolutivo. Em média, a partir da geração 25, a pontuação do indivíduo mais fit manteve-se constante até ao fim da experiência.

Foi importante balançar estes valores, correndo o risco de, caso a mutação seja demasiado elevada, o algoritmo se torne numa pesquisa primitiva aleatória.

Valores `DNF` significam que o algoritmo excedeu o limite de 1000 gerações sem convergir numa solução ótima.

	10 Cromossomas	20 Cromossomas
0% Mutação	DNF	DNF
20% Mutação	364.1	273.3
50% Mutação	295.5	211.9
100% Mutação	520.6	494.3

Através das experiências conduzidas, infere-se que um valor demasiado baixo de mutações levará a um estagnamento das gerações, porém numa situação de pesquisa primitiva aleatória (com 100% de mutação), além da variância dos resultados ser vastamente superior, o número médio de gerações até alcançar um indivíduo perfeito aumenta.

Relembre-se que, mesmo que estes valores de mutação pareçam altos, a mutação apenas ocorre em apenas um paper de uma conferência, num dos seus parâmetros. O impacto revela-se baixíssimo.

O aumento de cromossomas origina valores proporcionais e aparenta indicar que o sobrelotar a população favorece o algoritmo. Porém, é discutível que o contrário sucede. O tempo de computação aumenta consideravelmente, atingindo, em média, o dobro e o número de cromossomas gerados até à conquista de uma solução ótima é superior na ordem dos milhares.

Conclui-se que é fundamental um balanço entre a ativação da função de mutação e o número de cromossomas. A redução destes e o nivelamento de mutações aparenta estar na base da melhor solução. De modo algum o algoritmo beneficia de aleatoriedade, dada a forte variância observada.

5 Conclusões

Com o fim deste trabalho o grupo pode afirmar que o seu conhecimento de algoritmos genéticos aumentou substancialmente, graças às várias horas dedicadas ao planeamento e execução deste projeto. Foi também uma boa maneira de interiorizar o porquê da importância da utilização destes algoritmos em certas ocasiões da vida real.

Pode-se afirmar que o grupo se encontra orgulhoso do resultado final obtido.

5.1 Software

Segue a lista de software utilizado para desenvolvimento:

1. Sistema operativo: Windows 10/ Manjaro 4.9
2. Linguagem: Python 3.6

Adicionalmente, para o desenvolvimento do programa foram usados dois módulos de python, um para trabalhar com intervalos e outro para exportar a informação para uma folha de cálculo. Ambos podem ser instalados correndo o *script install.sh* como super-utilizador.

5.2 Contribuição de cada elemento do grupo

Considera-se que este trabalho foi esforço conjunto dos três elementos que compunham a equipa.

Assim, foi possível comprovar que cada membro, João Dias Conde Azevedo, José Pedro da Silva e Sousa Borges e Miguel Mano Fernandes, contribuiu em 1/3 para o resultado final.

References

[1] Mitsuo Gen and Runwei Cheng.
Genetic Algorithms and Engineering Optimization, 2000.

[2] Keshav Dahal, Kay Chen Tan and Peter I.
Evolutionary Scheduling, 2007.

[3] Introduction to Genetic Algorithms,
`https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3`

[4] Using a Genetic Algorithm to Optimize Developer Conference Schedules,
`https://medium.com/@filiph/using-a-genetic-algorithm-to-optimize-developer-conference-schedules-27f13d97fa9a`

[5] Time Table Scheduling,
`https://medium.com/@vijinimallawaarachchi/time-table-scheduling-2207ca593b4d`

[6] Learning: Genetic Algorithms,
`https://www.youtube.com/watch?v=kHyNqSnzP8Y`

6 Apêndice

6.1 Manual do utilizador

Para começar, deve ser garantida a presença dos dois módulos requeridos pelo programa. Para isso, criámos um *shell script* para rapidamente instalar as dependências do programa. Garanta que possui o instalador *pip* instalado e atualizado. Corra o *script* como super-utilizador. Caso já possua o necessário instalado, nada será feito.

Em Windows, procure instalar os módulos **intervals** e **openpyxl** através do *pip* manualmente.

Para correr o programa, execute com python3.6 ou uma versão superior o *script "genetic.py"*

O programa será inicializado. Ser-lhe-á requerido a localização do ficheiro de *papers* e com que nome e onde deseja que a folha de cálculo de output seja guardada.

Para efeitos de teste, usando a estrutura atual do repositório,

1. instale as dependências
2. corra o script *genetic.py*
3. especifique um ficheiro da pasta *files* com o caminho relativo à pasta do código-fonte "../files/" + um dos ficheiro exemplo
4. especifique o nome e o local do ficheiro output

Enquanto o programa corre, pode optar por deixá-lo atingir o valor heurístico que definiu ou terminá-lo e o seu melhor resultado será guardado numa folha de cálculo em que as pausas (*coffee-breaks* e almoço) estão coloridas a amarelo e as sessões da manhã e da tarde a verde.

Para além disso, um ficheiro nomeado *logs.txt* é criado na pasta *files* contendo o número da geração e o valor heurístico da melhor solução até ao momento.