# Geração de Horários (Tema B1, Grupo 33)

Geração de Horários utilizando Algoritmos Genéticos em Linguagem Python (Tema B1, Grupo 33)

Nuno Lopes (201605337)
*Faculdade de Engenharia da Universidade do Porto*
Porto, Portugal
up201605337@fe.up.pt

Amadeu Pereira (201605646)
*Faculdade de Engenharia da Universidade do Porto*
Porto, Portugal
up201605646@fe.up.pt

João Alves (201605236)
*Faculdade de Engenharia da Universidade do Porto*
Porto, Portugal
up201605236@fe.up.pt

***Resumo*—Este artigo contém a descrição de um trabalho cujo objetivo é resolver um problema de otimização utilizando diferentes algoritmos de otimização/ metaheurísticas.**

**Os algoritmos utilizados são comparados entre si tendo em conta a qualidade média da solução obtida e o tempo médio despendido para obter as soluções.**

***Index Terms*—Inteligência Artificial, Algoritmos Genéticos, Otimização, Algorimos Hill-Climbing, Algoritmo de Arrefecimento Simulado,**

## I. Introdução

Os algoritmos genéticos são técnicas de procura utilizadas na ciência da computação para encontrar soluções aproximadas em problemas de otimização e procura. Neste trabalho os algortimos são utilizados para gerar horários tendo em conta um conjunto de restrições sendo que o objetivo é obter uma solução que não viole nenhuma restrição ou que as viole o menor número de vezes possível.

## II. Descrição do Problema

O problema da geração de horários foi proposto por Ben Paechter para a Metaheuristics Network. É uma redução de um problema típico de geração de horários dos cursos universitários. Trata-se de um conjunto de eventos a serem organizados em 45 períodos (5 dias, 9 horas cada), um conjunto de salas nas quais podem ocorrer eventos, um conjunto de alunos que participam nos eventos e um conjunto de recursos satisfeitos por salas e exigido por eventos. Cada aluno participa em vários eventos e cada sala tem um tamanho. Um horário viável é aquele em que todos os eventos são atribuídos a um período de atividade e a uma sala, de modo a que as seguintes restrições sejam cumpridas:

- Nenhum estudante pode participar em mais do que um evento ao mesmo tempo;
- A sala é grande o suficiente para todos os alunos e satisfaz todos os recursos exigidos pelo evento;
- Por cada período de tempo de cada sala apenas pode estar a ser realizado um evento.

Além disso, o horário de um estudante é igualmente penalizado por cada violação das seguintes restrições flexíveis:

- Um estudante tem uma aula no último período do dia;
- Um estudante tem mais do que duas aulas consecutivas;
- Um estudante tem apenas uma aula num dia.

## III. Formulação do Problema

Para a facilitar a resolução deste problema foi realizado um pré-processamento dos dados. Estes foram obtidos através de ficheiros fornecidos pela competição que continham a informação inicial dos horários a gerar.

Foram criadas as classes **Room**, **Student** e **Event** para o tratamento dos dados. A classe **Room** tem um **id** que representa o número da sala, uma variável **size** que representa o número máximo de estudantes que esta permite ter e uma lista com as **features** (recursos) que a sala dispõe. A classe **Student** contém um **id** que representa o número do estudante e uma lista **events** que contém todos os eventos em que o estudante em causa participa. Por fim a classe **Event** contém um **id** que representa o número do evento e uma lista **features** que contém os recursos requeridos pelo evento em questão. São então criadas 3 listas (**ROOMS**, **STUDENTS**, **EVENTS**) que contém todas as salas, estudantes e eventos, respetivamente.

Após a obtenção e armazenamento da informação proveniente dos ficheiros iniciais foram criadas três matrizes, sendo estas: **event_student**, é uma matriz com dimensão num_events*num_students em que a linha i indica quais os estudantes que participam no evento i; **event_rooms**, é uma matriz com dimensão num_events*num_rooms em que a linha i indica quais as salas que podem receber o evento i; **incidence**, é uma matriz booleana com dimensão num_events*num_events em que a entrada (i, j) indica se o evento i e j podem estar no mesmo período de tempo. Com estas matrizes existe a possibilidade de, mais facilmente, se obter uma solução que não viole nenhuma das restrições principais (exceto as restrições fléxiveis referidas na descrição do problema).

As soluções do problema são representadas pela classe **Allocation**. Esta classe contém uma lista com objetos da classe **Slot**, que por sua vez tem um **id** que representa o número do período de tempo e uma lista **distribution** que tem os eventos e as salas utilizadas por estes no período de tempo em questão. A classe **Allocation** também contém um atributo **alloc_value** que representa o valor de avaliação da solução em questão e uma lista **event_data** que facilita a obtenção do período

de tempo e da sala que foram atribuidos a um evento para a solução em questão, facilitando assim os algoritmos em termos temporais.

A função que obtém o valor de avaliação de uma solução (**value**) foi obtido através do ficheiro de testes fornecido pela competição. Esta função conta quantas vezes foram violadas as restrições flexiveis.

## IV. Trabalho Relacionado

De forma a percebermos melhor a melhor forma de conseguirmos implementar uma forma de resolver este problema, fomos à procura dos resultados da competição [here] e conseguimos encontrar o top 4 de melhores soluções. Em primeiro encontra-se Philipp Kostuch [1], segundo Brigitte Jaumard, Jean-François Cordeau and Rodrigo Morales [2], Yuri Bykov [3] em terceiro e Luca Di Gaspero and Andrea Schaerf [4] com o quarto lugar.

## V. Algoritmos

Neste trabalho foram implementados todos os algoritmos pedidos, sendo estes: Algoritmo Hill-Climbing, Algoritmo de Arrefecimento Simulado e Algoritmos Genéticos. A implementação destes algoritmos está no ficheiro *logic.py*.

Para serem utilizados estes algoritmos foram criadas algumas funções principais:

- **generateRandomAllocation** - gera uma solução fiável (não viola nenhuma das principais restrições) aleatória;
- **isFeasible** - verifica se uma solução é fiável;
- **value** - avalia a qualidade de uma solução, ou seja, é o número de vezes que as restrições flexíveis são violadas;
- **getBestNeighbour** - retorna o vizinho de uma solução que tem melhor avaliação através da função **value**;
- **getBetterNeighbour** - retorna o primeiro melhor vizinho que encontrar de uma solução;
- **getRandomNeighbour** - retorna um vizinho aleatório de uma solução.

O Algoritmo Hill-Climbing consiste em escolher um estado aleatoriamente do espaço de estados, gerar todos os vizinhos escolhendo o melhor e voltar a repetir até não haver vizinhos melhores. Quanto à sua implementação foram criados os dois tipos de Hill-Climbing: *Hill-Climbing básico*, que consiste em gerar um a um os sucessores do estado e selecionar o primeiro que tenha uma melhor avaliação, e *Steepest ascent*, que consiste em gerar todos os sucessores possíveis e escolher o que tem melhor avaliação. A função **hill_climbing_1** executa o tipo *Hill-Climbing básico*, enquanto que a função **hill_climbing_2** executa o tipo *Steepest ascent*.

No caso do Algoritmo de Arrefecimento Simulado a sua implementação pode ser encontrada na função **simulated_annealing**. Este algoritmo é parecido ao Hill-Climbing porém admite explorar vizinhos piores, consoante uma certa probabilidade. De forma a obter uma solução melhor no final do algoritmo decidimos utilizar um valor inicial de 30 para a temperatura e a cada iteração a temperatura é reduzida em $((1/math.log(1+i)) * 0.1)$.

O Algoritmo Genético encontra-se implementado na função **genetic_algorithm** e começa por criar uma população inicial, descendente de uma solução random (*seed*). Para facilitar a representação e interação com uma população foi criada a classe **Population**. Após a geração da população inicial é feita uma seleção de dois horários através de uma *Roulette Wheel Selection*, esta seleção é proporcional ao nível de fitness de cada elemento da população, sendo que os de fitness mais elevado têm mais probabilidade de serem escolhidos. Tendo dois individuos selecionados seguimos para a criação de um descendente que ocorre do *crossover* entre os seu pais. Para esta etapa percorremos todos os eventos existentes e alocamo-lo ou no slot em que estava alocado no pai ou no slot em que estava alocado da mãe. Durante esta parte também se processa a mutação do descendente uma vez que, com uma dada probabilidade, o evento em vez de ser alocado no slot de um dos seus pais é alocado num slot random possível. Também implementamos o conceito de elitismo, em que os N melhores escalonamentos são diretamente transferidos para a geração seguinte.

## VI. Experiências e Resultados

O desenvolvimento dos algoritmos para o trabalho foi sempre feito de forma incremental de forma a podermos testar vários aspetos computacionais. Para testar os algoritmos basta correr o ficheiro *schedule.py*, podendo escolher qual algoritmo executar através de um menu. Para escolher o ficheiro a testar, que se encontram dentro da pasta *competition*, é preciso alterar a variável **FILENAME** dentro do ficheiro *settings.py*.

No caso do Algoritmo Hill-Climbing não nos foi possível obter uma solução em tempo real para todas as instâncias provenientes da competição visto que o processo de calcular um vizinho melhor de um estado é bastante pesado temporalmente. Porém prevemos que uma solução deste algoritmo possuíria uma avaliação a rondar os 200. Em termos comparativos entre os dois tipos de algoritmo Hill-Climbing, o *Hill-Climbing básico* é mais rápido a calcular um vizinho para a próxima iteração porém utiliza mais iterações para chegar ao melhor vizinho possível enquanto que o tipo *Steepest ascent* demora muito mais tempo a calcular o vizinho mas demora menos iterações para chegar ao melhor vizinho possível.

No Algoritmo de Arrefecimento Simulado tivemos dificuldades em calcular qual seria a melhor temperatura inicial e qual seria o valor de decréscimo da temperatura em cada iteração. Porém escolhemos um valor que nos pareceu adequado utilizando tentativa e erro. Mesmo assim os resultados que obtivemos não foram de todo muito satisfatórios pois não melhorava assim tanto comparativamente com a solução inicial gerada. Este valor obtido da solução rondava entre os 500 e 600. O principal problema que encontramos na implementação/execução deste algoritmo foi o facto de ter uma probabilidade grande de aceitar maus vizinhos no inicio o que leva a subir muito o valor de avaliação da solução.

Nas experiências feitas com o algoritmo genético apercebemos-nos de que ao realizar um crossover entre dois individuos de uma população o resultado não era, na maior

parte das vezes, uma solução viável (o mesmo acontecia com a função de mutação). Isto é refletido depois de visualizados os valores correspondentes às soluções geradas, que aumentavam de forma exponencial após uma iteração do algoritmo. De notar que pode haver uma pequena chance de um resultado positivo (melhores valores) após a primeira passagem de geração, mas claro sempre condicionada com o fator sorte.

## VII. Conclusões e Perspetivas de Desenvolvimento

Depois de vários testes corridos o grupo chega à conclusão de que o trabalho realizado está dentro daquilo que era o objetivo principal. Tendo em conta que o problema faz parte de uma competição internacional e de que seria uma tarefa árdua, sentimos que o trabalho desenvolvido está num "bom caminho". Acreditamos que a solução desenvolvida para o problema tem potencial e foi bem desenvolvida apesar de estar um pouco longe do considerado óptimo, isto porque na nossa opinião o desenvolvimento do algoritmo genético para a resolução do problema não é uma forma viável devido ao crossover entre dois horários que acaba por ser condicionado por um fator de sorte. Algo que também se verifica nos outros algoritmos pois estão sempre dependentes da aleatoriedade o que condiciona bastante a solução final.

Dito isto, tendo em conta ambos os bons e maus aspetos do trabalho, todo o estudo envolvido para a realização do mesmo contribuiu para um aprofundamento dos conteúdos e conceitos relacionados com Machine Learning. Acima de tudo o grupo sentiu enorme interesse em aprender mais.

## Referências


[1] P A Kostuch, Timetabling Competition - SA-based Heuristic, January 2003

[2] Jean-François Cordeau, Brigitte Jaumard and Rodrigo Morales, Efficient Timetabling Solution with Tabu Search, March 2003

[3] Yuri Bykov, The Description of the Algorithm for International Timetabling Competition

[4] Luca Di Gaspero and Andrea Schaerf, Timetabling Competition TT-Comp 2002: Solver Description