Algoritmo genético para cálculo da tabela de horários dos Componentes Curriculares do curso de Ciência da Computação da UFFS

Gabriel Batista Galli[1], **Matheus Antonio Venancio Dall'Rosa**[1], **Vladimir Belinski**[1]

[1]Ciência da Computação – Universidade Federal da Fronteira Sul (UFFS)
Caixa Postal 181 – 89.802-112 – Chapecó – SC – Brasil

{g7.galli96, matheusdallrosa94, vlbelinski}@gmail.com

Resumo. *O presente trabalho, apresentado ao curso de Ciência da Computação da Universidade Federal da Fronteira Sul - UFFS - Campus Chapecó - como requisito parcial para aprovação no Componente Curricular Inteligência Artificial, 2017.1, sob orientação do professor José Carlos Bins Filho, consiste em uma descrição detalhada da implementação de um algoritmo genético para calcular a tabela de horários dos Componentes Curriculares do curso de Ciência da Computação da UFFS.*

1. Descrição geral do algoritmo

A implementação do algoritmo genético encontra-se distribuída em dois arquivos: *main.cpp*, o arquivo principal, cujo conteúdo engloba as funções do programa; e *ia.h*, o arquivo de cabeçalho de *main.cpp*, onde se encontram declaradas as estruturas utilizadas e alguns protótipos de funções.

O fluxo de execução do algoritmo pode ser encontrado na função *main()* de *main.cpp*. Inicialmente, realiza-se uma chamada à função *read_professors()*, que realiza a leitura dos professores e horários que não desejam ministrar aulas, a partir do arquivo de entrada.

Em sequência é calculada a *fitness* máxima de um indivíduo, dada da seguinte maneira: inicia-se com uma *fitness* de 450 pontos, dos quais 30 consistem em pontos atribuídos ao professor pelo fato dele não estar dando aula nos horários que não deseja, 30 pelo fato de não estar dando aulas nos períodos matutino e noturno no mesmo dia, 30 por não estar dando aulas sequenciais e 360 por não estar dando aula no último horário de uma noite e primeiro do dia seguinte. Os três primeiros casos se relacionam às preferências estabelecidas na descrição do trabalho proposto e o último corresponde à última restrição elencada. Cabe destacar que, devido a maneira em que o algoritmo foi implementado, as demais restrições não são violadas.

Ao ser realizada uma violação da preferência que nos informa que um professor não deveria dar aula nos horários que não deseja, é realizado um desconto de 1 ponto por horário infringido (dentro de um limite de 30 possíveis horários de uma semana, motivo pelo qual a *fitness* do indivíduo inicia com 30 pontos atrelados a essa preferência). Por sua vez, ao ser violada a preferência que indica que um professor não deveria dar aula nos turnos matutino e noturno no mesmo dia, é realizado um desconto de 6 pontos na *fitness* do indivíduo por dia infringido (são 6 pontos, pois temos inicialmente 30 pontos para essa preferência e cinco dias na semana. Logo, $30/5 = 6$). Adicionalmente, ao ser realizada

uma violação da preferência que apresenta que um professor não deveria dar aulas de maneira sequencial (um turno inteiro), é realizado um desconto de 2 pontos por turno infringido (são 2 pontos, pois temos inicialmente 30 pontos para essa preferência e 15 possíveis casos de aulas sequenciais por semana. Logo, $30/15 = 2$). Às três preferências foram atribuídos 30 pontos iniciais para que, assim, elas possuíssem o mesmo peso no cálculo da *fitness*.

Por fim, a restrição que informa que um professor não pode dar aula no último horário de uma noite e no primeiro horário da manhã seguinte é levada em conta na função de *fitness* da seguinte forma: inicialmente são atribuídos 360 pontos na *fitness* de um indivíduo para a não violação dessa restrição. A cada infração dela 90 pontos são descontados da *fitness* do indivíduo (são 90 pontos, pois temos 360 pontos iniciais e 4 possibilidades de violação – $(21, 02)$, $(23, 04)$, $(25, 06)$, $(27, 08)$. Logo $360/4 = 90$). São 360 pontos para a restrição, pois, como uma violação de restrição não deve ocorrer, foi decidido que 80% da *fitness* seria atribuída a ela, de maneira que indivíduos que tivessem a restrição violada (caso com poucas ocorrências nos testes realizados) sofressem significativas penalidades e consequentemente tivessem sua probabilidade de reprodução amplamente diminuída.

Devido ao fato de não ter sido utilizado números de ponto flutuante para a representação da *fitness* e a fim de ser mantida a proporcionalidade de um valor entre 0 e 1, multiplicamos o valor inicial máximo da *fitness* explanado acima pelo número de professores dado na entrada.

Em seguida, é calculada a *fitness* máxima da população, dada pela multiplicação da *fitness* máxima de um indivíduo pelo tamanho da população.

Cabe enfatizar que as funções relacionadas ao cálculo da *fitness* de um indivíduo, de uma população e de suas penalidades são, respectivamente, *fitness()*, *set_population_fitness()*, *count_schedules_to_avoid_infringements()*, *count_morning_night_infringements()*, *count_consecutive_schedules_infringements()* e *count_restriction_infringements()*, todas presentes em *main.cpp*.

Sequencialmente, são lidos do arquivo de entrada os cursos/semestres, salas e horários disponíveis para a salas (função *read_room_schedules()*) e os CCRs, períodos, cursos/semestres e professores (função *read_subjects()*).

Após isso, é realizada a geração da população inicial, armazenada na estrutura *population_t* e montada através da chamada das funções *generate_people_permutation()* e *generate_population()*.

Na função *generate_people_permutation()* é realizada a construção de m permutações aleatórias, sem uso de *backtracking*. Seja $Professors = \{p_1, p_2, ..., p_m\}$ o conjunto de professores recebidos na entrada, e uma permutação P_i definida por $P_i = \{p_{i_1}, p_{i_2}, ..., p_{i_m}\}$, é feito $p_{i_1} = p1$ e sorteados os $m - 1$ professores restantes sem uso de *backtracking* para as posições $\{p_{i_2}, ..., p_{i_m}\}$ da permutação P_i.

A construção dos indivíduos (com tamanho igual ao parâmetro constante POPULATION_SIZE) acontece da seguinte forma: a população é dividida em m grupos de tamanho $g =$ POPULATION_SIZE$/m$. Um grupo m_i é construído a partir da permutação i, ou seja, a primeira permutação de m_i é $m_{i_1} = P_1$. Então, são geradas $m - 1$ permutações

lexicograficamente ordenadas a partir da permutação P_i, sendo utilizado para tal o algoritmo implementado na função *next_permutation()* da biblioteca *algorithm*. Cabe destacar que cada indivíduo é construído a partir de uma permutação.

Seja $schedule_i$ um par sala e horário, e $schedules$ um conjunto de pares de sala e horário, o conjunto $schedules$ é ordenado de tal forma que $schedule_i < schedule_j$ e $i < j$, caso a sala de $schedule_i$ possua menos horários disponíveis do que a sala de $schedules_j$. Dada a ordenação apresentada, à cada indivíduo é realizada a atribuição de seus horários, sendo esses tomados de maneira sequencial a partir de $schedule_1$.

Finalmente, é realizada uma chamada à função *evolve()*, que recebe como parâmetro a quantidade de gerações em que o algoritmo será executado e é responsável pela evolução da população. A cada geração uma nova população é inicializada, sendo os indivíduos dessa nova população resultantes do cruzamento de dois indivíduos da população atual, que, por sua vez, são sorteados de maneira com que os indivíduos que apresentem maior *fitness* tenham maior probabilidade de reprodução. São realizados cruzamentos até que a nova população atinja o tamanho da atual. Após cada cruzamento (detalhado na seção "Cruzamento dos indivíduos") é realizada a mutação do indivíduo resultante dentro de uma probabilidade especificada no código (a mutação será detalhada na seção "Mutação dos indivíduos").

Ainda a respeito da implementação do algoritmo, cabe destacar os critérios de parada estabelecidos para esse. O algoritmo é finalizado quando ocorre um dos seguintes casos:

1. Quando é gerado um indivíduo de *fitness* máxima, ou seja, uma resposta ideal/perfeita ao problema, na qual não haveria nenhuma violação de restrições e preferências;
2. Quando a *fitness* de uma população se mantém inalterada por um número estabelecido de gerações, definido no parâmetro constante MAX_UNMODIFIED_GENERATIONS;
3. Quando o algoritmo atinge um número máximo de gerações, definido no parâmetro constante MAX_GENERATIONS.

Cabe destacar que ao longo da execução do algoritmo são impressas no console a numeração das gerações, a *fitness* de sua população e a *fitness* máxima. Ao ser atingido um dos critérios de parada informados, também é destacado por qual motivo a execução foi finalizada e impressos no console os dados do indivíduo de maior *fitness* da última população gerada. Por fim, cabe destacar que para fins de debug pode ser descomentado o define de debug no arquivo *main.cpp*, o qual faz com que sejam impressas mais informações no console durante a execução do algoritmo.

2. Representação de um indivíduo

Um indivíduo é representado pela estrutura *person_t*, presente em *ia.h*, sendo definido por uma lista de genes e um valor de *fitness*.

Por sua vez, cada gene é representado pela estrutura *schedule_t*, presente em *ia.h*, que armazena as informações referentes à sala de aula (estrutura *room_t*, que contém o número da sala, o curso e os horários disponíveis da sala), ao componente curricular (estrutura *subject_t*, onde também pode ser encontrado, por exemplo, o professor associado ao componente curricular e o curso), ao horário da semana e o número do período.

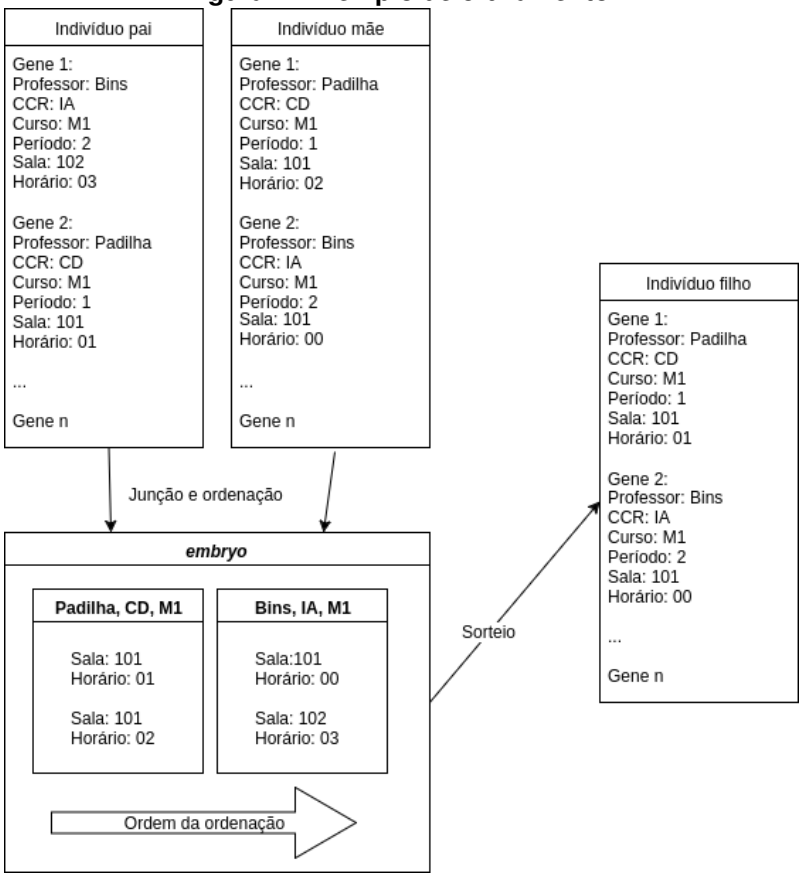
3. Cruzamento dos indivíduos

A implementação do cruzamento pode ser encontrada na função *cross()* de *main.cpp*. A chamada de *cross()* é realizada dentro de *evolve()*, de forma a gerar os indivíduos de uma nova população. *cross()* recebe como parâmetros um indivíduo pai e um indivíduo mãe, sorteados da forma apresentada na seção "Descrição geral do algoritmo".

Inicialmente, são agrupados na estrutura *embryo* os pares sala e horário do indivíduo pai e do indivíduo mãe que apresentam a mesma tripla professor, componente curricular e curso. Após isso, *embryo* é ordenado de forma não decrescente pela quantidade de períodos de um determinado componente curricular. O desempate da ordenação é realizado pela quantidade salas disponíveis, de forma com que as triplas com maior quantidade de salas são ocupadas primeiro.

Em seguida, para cada tripla professor, componente curricular e curso de *embryo*, são sorteados *n* pares de sala e horário, sendo *n* a quantidade de períodos necessários ao componente curricular da tripla. Após cada sorteio é realizada verificação de forma a garantir que um professor não ministre duas aulas no mesmo horário ou ocupe uma sala já reservada à outra tripla professor, CCR e curso. Assim, ao final do cruzamento é tido como resultado um indivíduo válido e que apresenta em seu cromossomo uma mistura de genes de seus indivíduos pai e mãe.

Figura 1. Exemplo de cruzamento.



Um exemplo simplório de cruzamento é apresentado na Figura 1 (cabe destacar que na imagem são apresentadas somente as informações necessárias para a explicação).

Nele, temos um indivíduo pai e um indivíduo mãe, cada um com uma sequência de genes. Conforme apresentado anteriormente, o passo inicial da função de cruzamento é realizar o agrupamento na estrutura *embryo* dos pares sala e horário do indivíduo pai e do indivíduo mãe que apresentam a mesma tripla professor, componente curricular e curso. No exemplo da Figura 1 percebe-se que para a tripla Padilha, CD, M1 têm-se os pares sala e horário $(101, 01)$ e $(101, 02)$ e para a tripla Bins, IA, M1 têm-se os pares $(101, 00)$ e $(102, 03)$. Para o exemplo, a ordenação apresenta o CCR CD antes do CCR IA, pois, conforme apresentado, CD apresentava um número menor de períodos. Após a montagem de *embryo* são realizados os sorteios para tomada dos genes que irão compor o cromossomo do indivíduo filho. No exemplo é apresentado no indivíduo filho o resultado de um possível sorteio, no qual temos que o indivíduo filho herdou genes do pai e da mãe.

4. Mutação dos indivíduos

A implementação da mutação pode ser encontrada em *mutate()* de *main.cpp*. Tal função é executada após o cruzamento de dois indivíduos e ocorre seguindo a probabilidade de mutação estabelecida no parâmetro constante MUTATION_RATE.

A função *mutate()* recebe um indivíduo candidato à sofrer uma mutação e constrói uma estrutura que armazena os horários atualmente livres dentre as salas de aula dadas na entrada do problema. Então, itera pelos horários alocados no indivíduo sendo analisado e, ao ser encontrada uma sala de aula com horário disponível e de mesmo curso (e.g. V1...) que o horário sendo considerado no momento, sorteia um horário dentre os disponíveis e realiza a troca de sala e horário no cromossomo do indivíduo.

5. Parâmetros utilizados na execução do algoritmo

No total, quatro parâmetros constantes (definido em *main.cpp*) são utilizados ao longo do algoritmo, sendo eles:

1. POPULATION_SIZE: representa o tamanho da população;
2. MUTATION_RATE: representa a taxa de mutação, a qual varia de 0 a 100;
3. MAX_GENERATIONS: representa o número máximo de gerações pelas quais o algoritmo será executado, caso os demais critérios de parada não sejam satisfeitos antes;
4. MAX_UNMODIFIED_GENERATIONS: representa o número máximo de gerações sem alteração de *fitness*, considerado como um dos critérios de parada do algoritmo.

6. Resultado final

Nos testes realizados observou-se que o algoritmo implementado se comporta conforme esperado e atendeu ao propósito ao qual foi desenvolvido, consistindo em um algoritmo genético para cálculo da tabela de horários dos Componentes Curriculares do curso de Ciência da Computação da UFFS. Ademais, foi observado que na maioria das execuções a *fitness* das populações geradas se mostrou não decrescente e convergiu rapidamente.