Programação Evolutiva aplicada ao problema de Alocação de Turmas em Salas de Aula

DAVI KOOJI UEZONO *

*Engenharia da Computação - Graduação
E-mail: davi.uezono@students.ic.unicamp.br

***Resumo* – O problema de encontrar uma alocação eficiente de turmas em salas de aula é um problema bastante complexo. Imagine um lugar como a Unicamp, onde a relação de disciplinas de graduação oferecidas em toda a universidade nos remete a aproximadamente oito mil disciplinas, cada uma contém um número de créditos diferente, algumas turmas tem mais ou menos alunos, outras exigem salas especiais com bancadas maiores, computadores, mesas de laboratório ou um vão livre, os professores têm preferências por aulas no período diurno ou no noturno por causa de outros compromissos pessoais. Há inúmeras variáveis que podem ser consideradas, e para efeito de simplificação, nem todas foram consideradas no momento deste trabalho. A solução foi implementada em Python e Shell Script com Programação Evolutiva utilizando mutações entre cada geração para chegar em uma que seja factível, a partir de uma base de dados que contém a preferência de dias e horários dos professores que lecionam a matéria. O projeto ainda permite parametrizar alguns valores, como o número de salas disponíveis, de turmas a serem alocadas, e o número mínimo de gerações para o critério de parada. Os resultados são obtidos em menos de um minuto para uma entrada contendo cinco mil turmas, mínimo de vinte e cinco gerações e trinta salas de aula disponíveis, onde sobram cerca de trinta turmas sem alocação em aproximadamente trinta gerações.**

***Palavras-chave* – programação evolutiva, alocação de salas, otimização**

I. INTRODUÇÃO

Durante um trabalho voluntário com foco educacional desenvolvido na faculdade, uma das minhas funções era realizar a alocação de professores nas nossas salas de aula do projeto. Na época, ainda no início da graduação, eu sabia que deveria ter como resolver este problema computacionalmente, mas não sabia exatamente como isso poderia ser feito. A motivação para resolvê-lo ficou inerte durante algum tempo, mas após ter estudado mais sobre o problema nas disciplinas de Teoria de Computação e notar que a modelagem do problema não era tão complicada como parecia ser, surgiu a oportunidade de resolvê-lo neste trabalho.

O problema de alocação de salas de aula é um dos problemas comumente estudados em disciplinas de Teoria de Computação, e é um dos problemas *NP-completo* na versão de se determinar a solução exata. O problema consiste em partir de um conjunto de dados que representa a disponibilidade/preferência de professores para lecionar suas disciplinas ao longo dos dias da semana e horários e evoluir uma entrada que satisfaz alguns casos, até que se chegue em algo factível, maximizando o casamento com a disponibilidade/preferência recebida. Esta evolução é realizada aplicando-se mutações em alguns indivíduos da população que não são satisfeitos pela atribuição atual.

Ao longo deste trabalho serão apresentados com maiores detalhes a implementação dos algoritmos que geram os dados de entrada (seção 2) e do algoritmo que evolui a solução (seção 3). Alguns experimentos foram realizados na tentativa de validar o algoritmo, e os resultados são apresentados na seção 4. A seção 5 abordará algumas sugestões de como melhorar a solução, incluindo alguns outros parâmetros que foram deixados de lado nesta versão inicial. Conclusões serão abordadas na seção 6.

II. ALGORITMOS PARA DADOS DE ENTRADA

Tem-se dois algoritmos de entrada, um que lista as disponibilidades/preferências dos professores, e outro que cria uma versão inicial do problema. Ambos os arquivos tem a mesma estrutura, contendo M linhas e onze colunas, onde M é o número de salas de aula, com cinco dias da semana e seis horários de aula por dia (totalizando as onze colunas).

O arquivo de disponibilidade contém valores 0, 1 ou 2 que indicam *indisponibilidade*, *disponibilidade* e *preferência*, respectivamente, e foi gerado totalmente ao acaso. Já o arquivo de entrada contém apenas os valores 0 e 1, indicando uma *não atribuição* e uma *atribuição*, respectivamente, e contém apenas um valor atribuição nas colunas de dia da semana e apenas um valor atribuição nas colunas de horário de aula. Dada as suas restrições, este também foi gerado aleatoriamente. Optou-se pela implementação de ambos em Python devido a facilidade da linguagem na manipulação de dados armazenados em arquivos ou strings em memória.

A. *subseção*

Se precisar, você pode usar listas, tais como

- Item 1
- Item 2

ou

1) Item 1
2) Item 2

III. ALGORITMO DE PROGRAMAÇÃO EVOLUTIVA

O algoritmo responsável por determinar uma solução aproximada é pequeno e consideravelmente simples, porém o tratamento dos dados provenientes dos arquivos de entrada deve ser feito cuidadosamente. O volume de dados é grande

e pode não ser fácil distinguir a região do arquivo em que se encontra pois os dados são aleatórios e em baixa variabilidade.

Inicia-se o algoritmo com um *loop* principal, onde cada iteração deste é o tratamento dos dados de uma geração. Vamos chamá-la de geração *corrente*. No início do problema, o arquivo de entrada é tratado como geração zero. A cada iteração, examina-se os dados da geração corrente com os arquivos de disponibilidade/preferência, produzindo o arquivo da próxima geração. Durante a análise, cada linha de um arquivo representa um indivíduo na população, que tem uma nota atribuída conforme a satisfação.

Caso não haja satisfação, a nota é zero e o indivíduo é um candidato a sofrer mutação para a próxima geração. Caso ocorra a satisfação, verifica-se como este ocorreu, isto é, se o dia da semana era um de *disponibilidade* ou de *preferência* e se o horário era um de *disponibilidade* ou de *preferência*; se ambos os critérios forem de disponibilidade apenas, a nota é um; se apenas um dos critérios for de disponibilidade, mas o outro for de preferência, a nota é dois; se, porém, ambos os critérios forem de preferência, então o indivíduo recebe a nota máxima, que é três.

Depois de calcular a nota para cada indivíduo, caso a nota seja um valor positivo (isto é, caso tenha ocorrido uma satisfação), verifica-se se ainda existe salas de aula disponíveis para o horário. Se ainda existir salas disponíveis para o horário, a turma fica definitivamente alocada; caso contrário, não. A função de *fitness* é calculada como a somatória das notas de todas as turmas (indivíduos) que estão devidamente alocados na solução (geração) corrente. Isto significa que mesmo que haja satisfação de disponibilidade/preferência dos professores, caso não haja salas disponíveis, então a nota desta atribuição não é contabilizada no *fitness* da geração.

O critério de mutação é uma reordenação aleatória da condição não satisfeita, tanto dentre os dias da semana quanto dentre os horários. A taxa de mutação controla o percentual de indivíduos que sofrem mutação entre uma geração e outra. Idealmente, ela deve ser próxima de 15%.

Existem dois critérios de parada implementado no algoritmo. Um deles estabele um número mínimo de gerações para ocorrer. Neste caso, as mutações devem continuar ocorrendo ainda que não haja ganhos na função de fitness. O outro critério diz que se o fitness da geração corrente for exatamente igual ao das duas últimas gerações, então o algoritmo não precisa continuar varrendo as próximas gerações, porque já se pode observar uma tendência de estagnação deste valor.

A. *Tabelas*

Uma tabela pode ser posicionada em qualquer lugar no texto, como no exemplo seguinte.

Para citar esta tabela, em qualquer ponto no texto, como Tabela I.

IV. Experimentos

Utiliza-se dois conjuntos de testes, com três testes em cada. No primeiro conjunto de testes, do primeiro para o segundo teste varia-se (dobra) a quantidade de salas disponível e compara-se o resultado. Depois varia-se (dobra) a taxa de mutação e compara-se o segundo com o terceiro resultado. Já no segundo conjunto de testes, com uma população maior, mantém-se fixo todos os parâmetros e varia-se apenas a quantidade de salas, dobrando a cada quantidade de salas disponíveis a cada teste. Adicionalmente, pode-se comparar o segundo teste do primeiro conjunto com o primeiro teste do segundo conjunto, onde a variação se dá apenas pela ordem de grandeza do tamanho da população.

Tabela I
Exemplo de texto de uma tabela.

	Texto		Sem #21	
X	Y	z	$\mathcal{A}$	valor-z
1	0,491	3,66	0,367	2,46
2	0,732	4,21	0,354	1,50
3	0,000	-	0,000	-
4	0,000	-	0,000	-
5	0,421	1,94	0,668	2,79
6	0,421	1,94	0,668	2,79
7	0,938	3,92	1,295	4,67
8	0,000	-	0,000	-
9	0,356	1,40	0,491	1,87

Uma figura pode ser posicionada em qualquer lugar no texto, como no exemplo seguinte da Figura 1.

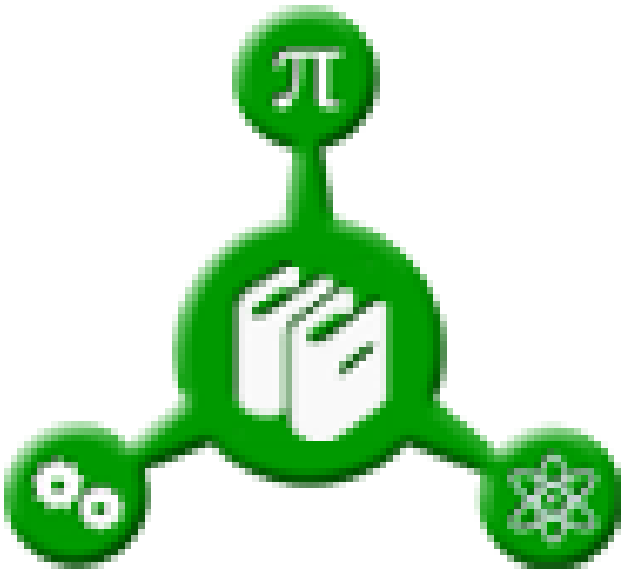

Figura 1. Um exemplo de figura.

Use o comando “cite” para citar itens na sua lista de referências através dos seus rótulos. Exemplo: [1][2][3].

V. Resultados e Discussão

Nesta seção você deve apresentar claramente os resultados obtidos para os testes efetuados. Procure organizar os dados

utilizando uma linguagem científica. Algumas opções são o uso de tabelas e gráficos, para que a compreensão seja fácil e rápida.

VI. Melhorias

VII. Conclusões

Nesta seção, faça uma análise geral de seu trabalho, levando em conta todo o processo de desenvolvimento e os resultados. Quais os seus pontos fortes? Quais os seus pontos fracos? Quais aspectos de sua metodologia de trabalho foram positivas? Quais foram negativas? O que você recomendaria (ou não recomendaria) a outras pessoas que estejam realizando trabalhos similares aos seus?

+————————-+

Referências

[1] J. K. Rowling, *Harry Potter and the Philosophers Stone*, 1st ed. London: Bloomsbury Publishing, 1997. 2

[2] J. H. Reynolds and D. J. Heeger, "The Normalization Model of Attention," *Neuron Review*, vol. 61, no. 2, pp. 168–185, 2009. 2

[3] M. P. Michalowski and R. Simmons, "Multimodal person tracking and attention classification," in *Proceedings of the 1st ACM SIGCHI/SIGART Conference on Human-robot Interaction*, ser. HRI 06. New York, NY, USA: ACM, 2006, pp. 347–358. 2

Submissão

Seu trabalho deve ser submetido via moodle em conjunto com o código fonte.

Prazo: 15/05/20126