UNIVERSIDADE ESTADUAL DE MARINGÁ

CENTRO DE TECNOLOGIA

DEPARTAMENTO DE INFORMÁTICA

CIÊNCIA DA COMPUTAÇÃO

MODELAGEM E OTIMIZAÇÃO DE ALGORITMOS

(6903/01)

Algoritmo Genético

Classroom Assignment Problem & Job Scheduling Problem

ALUNO	RA
Andrei Roberto da Costa	107975
Joao Gilberto	112684
Felipe Piassa Antonucci Esperanca	112647

SUMÁRIO

1. Introdução

O mundo atual está em constante evolução e as demandas por soluções otimizadas para os mais diversos problemas têm aumentado significativamente. Dentre esses problemas, podemos citar o Job Scheduling Problem e o Classroom Assignment Problem, ambos com alta complexidade computacional e grande importância prática.

O Job Scheduling Problem é um problema clássico de otimização, em que se busca alocar recursos limitados para uma série de tarefas que devem ser executadas em um determinado período de tempo. Já o Classroom Assignment Problem envolve a alocação de salas de aula para turmas de estudantes, levando em consideração diversos fatores como tamanho da turma, disponibilidade de equipamentos, localização e horários disponíveis.

Para solucionar esses problemas de forma eficiente, pode-se utilizar algoritmos genéticos, que são uma técnica de busca baseada em princípios evolutivos. Algoritmos genéticos são capazes de gerar soluções próximas ao ótimo global, mesmo em problemas de alta complexidade computacional, como é o caso do Job Scheduling Problem e do Classroom Assignment Problem.

Assim, O propósito deste trabalho consiste em apresentar uma abordagem baseada em algoritmos genéticos para a resolução do problema de agendamento de tarefas (Job Scheduling Problem) e do problema de atribuição de salas de aula (Classroom Assignment Problem), visando a obtenção das soluções ótimas para duas configurações distintas, seguida da análise dessas soluções.

2. Metodologia

2.1 Classroom Assignment Problem

O código é uma implementação de um Algoritmo Genético (AG) que resolve um problema de Alocação de Salas (Classroom Assignment Problem - CAP). O objetivo do problema é atribuir um horário e sala para cada aula, de forma que não ocorram conflitos de horário ou sala para aulas diferentes.

O algoritmo é definido pela classe CAP, que contém os parâmetros e funções necessárias para a execução do AG. A seguir é explicado o funcionamento das principais funções :

public CAP(...) É o construtor da classe que define os parâmetros do algoritmo e recebe como entrada o tamanho da população (populationSize), o número de gerações (numGenerations), as taxas de mutação (mutationRate) e de crossover (crossoverRate), o número de aulas (numClasses), o número de salas (numRooms), o número de horários (numTimeslots) e uma matriz com as informações das aulas (classes).

public void initializePopulation() é a função que inicializa a população aleatoriamente. Ela cria uma matriz population com populationSize indivíduos, cada um com numClasses genes. Cada gene representa o índice de uma sala e horário atribuídos a uma aula. Os valores dos genes são gerados aleatoriamente com Random.nextInt().

private int evaluateFitness(int[] individual) é a função que avalia a aptidão de um indivíduo. Ela recebe como entrada um vetor individual com numClasses genes, e retorna um valor inteiro que representa a qualidade do indivíduo. A função avalia se ocorrem conflitos de horário ou sala para as aulas do indivíduo. Se houver conflito, a aptidão é decrementada em 1. Caso contrário, a aptidão é mantida.

private int[] evaluateFitness() é a função que avalia a aptidão de todos os indivíduos da população. Ela percorre a matriz population e calcula a aptidão de cada indivíduo usando a função evaluateFitness(). O resultado é armazenado em um vetor fitnessValues com tamanho populationSize.

private int[][] selection(int[] fitnessValues) é a função que realiza a seleção de indivíduos para o crossover. Ela recebe como entrada o vetor fitnessValues com as aptidões de todos os indivíduos da população, e retorna uma matriz parents com dois indivíduos selecionados aleatoriamente. A seleção é feita por torneio simples, ou seja, dois indivíduos são escolhidos aleatoriamente e o mais apto é selecionado para o crossover.

private int[][] crossover(int[] parent1, int[] parent2) é a função que realiza o crossover entre dois indivíduos. Ela recebe como entrada os vetores parent1 e parent2 com numClasses genes cada, e retorna uma matriz offspring com dois filhos gerados pelo crossover. O crossover é feito com uma taxa de crossover definida pelo parâmetro crossoverRate. Se a taxa for menor que o valor gerado aleatoriamente, o crossover não é realizado e os filhos são iguais aos pais. Caso contrário, um ponto de crossover é escolhido aleatoriamente e os primeiros genes até o ponto são copiados de um pai e os restantes do outro pai para cada filho.

private void mutation(int[] individual) é a função que realiza a mutação em um indivíduo. Ela recebe como entrada um vetor individual com numClasses genes, e realiza a mutação com uma taxa definida pelo parâmetro mutationRate. Para cada gene, se a taxa for maior que o valor gerado aleatoriamente, o gene é alterado para um valor aleatório.

public void evolve() é a função que executa a evolução da população. Ela chama as funções initializePopulation(), evaluateFitness(), selection(), crossover() e mutation() em um loop de numGenerations vezes, e armazena o melhor indivíduo encontrado na variável bestIndividual.

public void printSchedule(int[] individual) é a função que imprime o horário e sala atribuídos a cada aula para um indivíduo. Ela recebe como entrada um vetor individual com numClasses genes, e imprime as informações formatadas em uma tabela.

public void printBestSchedule() é a função que imprime o melhor horário e sala atribuídos a cada aula encontrado durante a execução do algoritmo. Ela chama a função printSchedule() com o melhor indivíduo armazenado na variável bestIndividual.

Em resumo, o algoritmo genético para o problema de Alocação de Salas funciona da seguinte forma: é criada uma população de indivíduos aleatórios, que representam possíveis soluções para o problema. Cada indivíduo é avaliado em termos de qualidade, levando em consideração se ocorrem conflitos de horário ou sala para as aulas. Os indivíduos mais aptos são selecionados para o crossover, onde ocorre a troca de informações entre dois indivíduos para gerar filhos que podem ter características de ambos. Por fim, é realizada a mutação em cada indivíduo para adicionar novas características à população. Esse processo é repetido por um número de gerações definido pelo usuário, e o melhor indivíduo encontrado é retornado como solução para o problema.

2.2 Job Scheduling Problem

O código é uma implementação do algoritmo genético que resolve o problema de JSP (Job Scheduling Problem) para resolver o problema de escalonamento de tarefas em máquinas. O algoritmo começa criando uma população inicial de soluções aleatórias, cada uma contendo uma lista de tarefas a serem realizadas em máquinas específicas. Em seguida, as soluções são avaliadas e selecionadas para gerar novas soluções através dos operadores genéticos de crossover e mutação. O processo é repetido até que uma solução satisfatória seja encontrada ou até que um número máximo de iterações seja atingido.

A classe JSA (Job Scheduling Algorithm) possui alguns atributos privados, incluindo:

populationSize: o tamanho da população de soluções.
numMachines: o número de máquinas disponíveis.
numTasks: o número de tarefas a serem realizadas.
taskTimes: uma lista de inteiros que representam o tempo de processamento de cada tarefa.
population: uma lista de listas de inteiros que representam as soluções da população.
random: um objeto da classe Random utilizado para gerar valores aleatórios.

JSA() recebe como parâmetros o tamanho da população, o número de máquinas e a lista de tempos de processamento das tarefas. Ele inicializa os atributos correspondentes e chama o método initializePopulation() para criar e inicializar a população de soluções.
initializePopulation() é responsável por criar uma nova solução para cada indivíduo da população. Ele itera o número de vezes especificado em populationSize, criando uma nova lista de inteiros para cada solução e preenchendo-a com valores aleatórios gerados pelo objeto random.
fitness() calcula a aptidão de uma solução passada como parâmetro. Ele cria um array machineTimes com o número de elementos igual ao número de máquinas e itera sobre as tarefas da solução, adicionando o tempo de processamento de cada tarefa ao tempo total de processamento da máquina correspondente. A função retorna o tempo de processamento da máquina com o maior tempo de processamento.
selection() seleciona aleatoriamente dois indivíduos da população e retorna aquele com menor valor de fitness.
crossover() realiza o operador genético de crossover em dois pais passados como parâmetros, gerando um novo filho. Ele seleciona aleatoriamente um ponto de crossover, adiciona as primeiras tarefas do pai 1 até o ponto de crossover na lista do filho, e adiciona as tarefas restantes do pai 2 na lista do filho a partir do ponto de crossover.
mutation() realiza o operador genético de mutação em uma solução passada como parâmetro, alterando aleatoriamente um dos genes da solução. Ele seleciona aleatoriamente uma posição do gene a ser mutado e altera seu valor para um valor aleatório entre 0 e o número total de máquinas disponíveis.
solve() é o principal método da classe e resolve o problema de escalonamento de tarefas em máquinas utilizando o algoritmo genético JSA. Ele recebe como parâmetro o número máximo de iterações que o algoritmo deve executar. Ele itera sobre as iterações, selecionando e combinando as soluções da população atual para gerar novas soluções, e em seguida aplicando o operador de mutação em uma porcentagem dos filhos gerados. A cada iteração, o método mantém a melhor solução encontrada até o momento e a retorna como resultado final.

Em resumo, a classe JSA implementa um algoritmo genético para resolver o problema de escalonamento de tarefas em máquinas. O algoritmo começa criando uma população inicial de soluções aleatórias, e em seguida aplica operadores genéticos para gerar novas soluções a partir da população atual. O

processo é repetido até que uma solução satisfatória seja encontrada ou até que um número máximo de iterações seja atingido. O resultado final é a melhor solução encontrada pelo algoritmo.

3. Resultados

3.1 Job Schedulling Problem

Para a *configuração 1*, do Job Scheduling Problem (JSP), foi utilizado os seguintes valores de entrada :

- População Inicial → **500**
- Máquinas Disponíveis → **10**
- Lista de Tarefas → **(8, 10, 5, 7, 9, 4, 6, 3, 5, 7, 6, 9, 8, 10, 7, 6, 8, 5, 3, 4)**

Após executar a *Configuração 1*, por 3 vezes, obtivemos os seguintes resultados de soluções ótimas :

[9, 6, 0, 4, 0, 3, 8, 9, 8, 5, 4, 2, 5, 7, 6, 3, 1, 8, 4, 2]
[9, 4, 6, 3, 2, 9, 6, 6, 7, 0, 7, 0, 3, 8, 5, 1, 2, 1, 5, 5]
[1, 3, 6, 8, 0, 2, 9, 1, 9, 3, 5, 7, 6, 4, 2, 9, 8, 5, 2, 6]

Para a *configuração 2*, do Job Scheduling Algorithm (JSP), foi utilizado os seguintes valores de entrada :

- População Inicial → **1000**
- Máquinas Disponíveis → **4**
- Lista de Tarefas → **(7, 4, 9, 6, 8, 5, 10, 3, 7, 6, 4, 9, 8, 5, 6, 7, 4, 10, 7, 9)**

Após executar a *Configuração 2*, por 3 vezes, obtivemos os seguintes resultados de soluções ótimas :

[3, 2, 3, 2, 0, 1, 3, 3, 1, 2, 3, 0, 2, 2, 1, 1, 2, 1, 0, 0]
[1, 2, 3, 3, 1, 0, 2, 0, 0, 3, 3, 1, 3, 0, 2, 0, 2, 1, 0, 2]
[0, 0, 2, 3, 0, 1, 1, 3, 3, 2, 2, 3, 0, 1, 2, 0, 1, 1, 2, 3]

3.1.1 Análise

No Job Scheduling Problem, o objetivo é determinar a ordem em que as tarefas devem ser executadas em cada uma das máquinas disponíveis, de modo a minimizar o tempo total de execução. Para isso, são utilizados algoritmos genéticos, que partem de uma população inicial de soluções aleatórias e evoluem ao longo de diversas gerações, combinando as soluções mais promissoras.

Observando as soluções ótimas encontradas para cada configuração, podemos ver que as ordens das tarefas variam entre as diferentes execuções. Isso indica que o algoritmo genético está explorando diferentes soluções possíveis e que não está preso em mínimos locais.

Além disso, é possível notar que, na Configuração 1, as tarefas 9 e 8 aparecem no início e no final da sequência, respectivamente, em todas as soluções ótimas. Isso pode indicar que essas tarefas são críticas para a minimização do tempo total de execução e que o algoritmo está dando prioridade a elas. Já na Configuração 2, não há uma tarefa que se destaque em todas as soluções ótimas encontradas.

Em resumo, os resultados obtidos mostram que o algoritmo genético está explorando diferentes soluções possíveis e que é capaz de encontrar soluções ótimas distintas em cada execução.

3.2 Classroom Assignment Problem

Para a *configuração 1*, do Classroom Assignment Problem (CAP), foi utilizado os seguintes valores de entrada :

Valores de entrada do Algoritmo Genético :

- Quantidade de indivíduos (soluções) que serão gerados e avaliados em cada geração do algoritmo → **100**
- Número de gerações que o algoritmo genético irá percorrer antes de terminar a busca por uma solução ótima → **500**
- Probabilidade de um indivíduo sofrer uma mutação em cada geração do algoritmo genético → **0.01**
- Probabilidade de dois indivíduos cruzarem seus genes para gerar um novo indivíduo em cada geração do algoritmo genético → **0.8**

Valores de entrada do Problema em questão :

- Número total de classes que precisam ser programadas em um horário → **10**
- Número total de salas disponíveis para programar as aulas → **3**
- Número total de intervalos de tempo disponíveis para programar as aulas → **5**
- Matriz que contém informações sobre as classes que precisam ser programadas em um horário → **{ {1, 2, 3, 4}, {2, 3, 4, 1}, {3, 4, 1, 2}, {4, 1, 2, 3}, {1, 2, 3, 4}, {2, 3, 4,1}, {3, 4, 1, 2}, {4, 1, 2, 3}, {1, 2, 3, 4}, {2, 3, 4, 1} }**

Após executar a Configuração 1, por 3 vezes, obtivemos os seguintes resultados de soluções ótimas :

[12, 13, 6, 4, 0, 2, 3, 5, 14, 8]
[5, 6, 11, 2, 9, 3, 14, 7, 1, 10]
[14, 9, 2, 6, 5, 3, 11, 13, 1, 0]

Para a *configuração 2*, do Classroom Assignment Problem (CAP), foi utilizado os seguintes valores de entrada :

Valores de entrada do Algoritmo Genético :

- Quantidade de indivíduos (soluções) que serão gerados e avaliados em cada geração do algoritmo → **150**
- Número de gerações que o algoritmo genético irá percorrer antes de terminar a busca por uma solução ótima → **700**
- Probabilidade de um indivíduo sofrer uma mutação em cada geração do algoritmo genético → **0.01**
- Probabilidade de dois indivíduos cruzarem seus genes para gerar um novo indivíduo em cada geração do algoritmo genético → **0.8**

Valores de entrada do Problema em questão :

- Número total de classes que precisam ser programadas em um horário → **8**
- Número total de salas disponíveis para programar as aulas → **4**
- Número total de intervalos de tempo disponíveis para programar as aulas → **5**
- Matriz que contém informações sobre as classes que precisam ser programadas em um horário → **{ {1, 2, 3, 4}, {2, 3, 4, 1}, {3, 4, 1, 2}, {4, 1, 2, 3}, {1, 2, 3, 4}, {2, 3, 4,1}, {3, 4, 1, 2}, {4, 1, 2, 3}, {1, 2, 3, 4}, {2, 3, 4, 1} }**

Após executar a *Configuração 2*, por 3 vezes, obtivemos os seguintes resultados de soluções ótimas :

[18, 3, 16, 6, 8, 13, 4, 9]
[12, 4, 17, 3, 14, 10, 6, 15]
[2, 6, 10, 18, 12, 1, 8, 19]

3.2.1 Análise

Os resultados apresentados indicam as soluções ótimas encontradas pelo algoritmo genético para resolver o problema de alocação de salas do Classroom Assignment Problem (CAP) para duas configurações diferentes.

Na primeira configuração, foram utilizados 100 indivíduos por geração, durante 500 gerações. A probabilidade de mutação foi de 0,01 e a probabilidade de cruzamento foi de 0,8. Para esse problema, havia 10 classes para serem programadas em 3 salas, em um total de 5 intervalos de tempo disponíveis. A matriz que contém informações sobre as classes que precisam ser programadas em um horário foi dada como entrada.

Os resultados apresentados para a *configuração 1* mostram as soluções ótimas encontradas em três execuções independentes do algoritmo genético, cada uma indicando a ordem em que as 10 classes devem ser programadas. Por exemplo, na primeira execução, a ordem correta seria alocar a classe 1 na primeira sala no primeiro intervalo de tempo, alocar a classe 2 na segunda sala no segundo intervalo de tempo, e assim por diante, até alocar a classe 8 na terceira sala no quinto intervalo de tempo.

Na segunda configuração, foram utilizados 150 indivíduos por geração, durante 700 gerações. A probabilidade de mutação foi de 0,01 e a probabilidade de cruzamento foi de 0,8. Para esse problema, havia 8 classes para serem programadas em 4 salas, em um total de 5 intervalos de tempo disponíveis. A matriz que contém informações sobre as classes que precisam ser programadas em um horário foi dada como entrada.

Os resultados apresentados para a *configuração 2* mostram as soluções ótimas encontradas em três execuções independentes do algoritmo genético, cada uma indicando a ordem em que as 8 classes devem ser programadas. Por exemplo, na primeira execução, a ordem correta seria alocar a classe 1 na primeira sala no primeiro intervalo de tempo, alocar a classe 2 na segunda sala no primeiro intervalo de tempo, e assim por diante, até alocar a classe 8 na quarta sala no quinto intervalo de tempo.

Os resultados sugerem que o algoritmo genético foi capaz de encontrar soluções ótimas para o problema de alocação de salas em ambas as configurações. No entanto, como o algoritmo genético depende de uma abordagem heurística, pode haver variações nos resultados encontrados em diferentes execuções do algoritmo. Por isso, é importante avaliar os resultados

em várias execuções e compará-los para garantir a robustez e a confiabilidade do algoritmo.

4. Conclusão

O trabalho apresenta uma implementação de um Algoritmo Genético (AG) para resolver o problema de Alocação de Salas (Classroom Assignment Problem - CAP), que consiste em atribuir um horário e sala para cada aula, de forma que não ocorram conflitos de horário ou sala para aulas diferentes. A implementação do AG é definida pela classe CAP, que contém os parâmetros e funções necessárias para a execução do AG. O processo consiste em criar uma população de indivíduos aleatórios, avaliar a aptidão de cada indivíduo, selecionar os indivíduos mais aptos para o crossover, realizar a troca de informações entre dois indivíduos para gerar filhos e, por fim, realizar a mutação em cada indivíduo para adicionar novas características à população. A avaliação da qualidade de cada indivíduo leva em consideração a ocorrência de conflitos de horário ou sala para as aulas. O processo é repetido por um número de gerações definido. A implementação do AG é uma ferramenta eficaz para a solução do problema de Alocação de Salas.

Além disso, também foi implementado a classe JSA (Job Scheduling Problem) que apresenta uma solução eficiente para o problema de escalonamento de tarefas em máquinas, utilizando um algoritmo genético que começa criando uma população inicial de soluções aleatórias e, em seguida, aplicando operadores genéticos para gerar novas soluções. O método solve() itera sobre as iterações, selecionando e combinando as soluções da população atual para gerar novas soluções, e em seguida aplicando o operador de mutação em uma porcentagem dos filhos gerados. O algoritmo se repete até que uma solução satisfatória seja encontrada ou até que um número máximo de iterações seja atingido. A classe JSA é uma solução eficaz para resolver o problema de escalonamento de tarefas em máquinas.