Mapeamento de Rede utilizando Algoritmo Genético: Implementação em Versões Sequencial, Paralela e Distribuída

Eurico D. N. Júnior[1], **Lucas S. Lopes**[1]

[1]Departamento de Sistemas de Informação
Universidade Federal do Piaui - Picos, Brasil

euricodelmondes@ufpi.edu.br, lucaslopes092009@gmail.com

1. Introdução

A alocação de recursos é um tema presente em diversas áreas da computação, e está diretamente relacionada com o problema de mapeamento. Este é um problema combinacional que envolve escolhas, e é abordado em disciplinas como sistemas operacionais, redes de computadores e engenharia de software. No contexto do sistema 5G, o mapeamento é realizado no processo de criação de redes virtuais sob demanda, conhecido como network slicing as a service (NSaaS). Nesse contexto, as fatias de redes são um exemplo do problema de virtual network embedding (VNE), que tem sido investigado em matemática por um longo período de tempo.

Assim como em algoritmos genéticos, onde é preciso encontrar uma solução ótima a partir de um conjunto de possíveis soluções, a alocação de recursos e o problema de mapeamento também requerem a busca pela melhor solução dentre diversas opções. Em ambos os casos, é necessário avaliar o desempenho das soluções candidatas em relação a um conjunto de critérios e aplicar operadores de seleção, recombinação e mutação para gerar novas soluções.

Este trabalho aborda o desafio de associar redes virtuais em redes reais no contexto do sistema 5G, considerando a alocação de recursos e o problema de VNE. Na alocação de recursos e no problema de mapeamento, a busca pela melhor solução é afetada por diversas restrições e limitações, tais como a disponibilidade de recursos, a capacidade de processamento, a largura de banda e outros fatores. Dito isso, utilizar algoritmos genéticos e técnicas de paralelismo e distribuição pode ser uma solução promissora para otimizar o mapeamento de redes virtuais e reduzir o tempo de processamento.

2. Implementação e Testes

Para solucionar o problema, o algoritmo genético busca encontrar o melhor mapeamento para uma determinada requisição em uma rede física. Esse mapeamento é considerado o melhor quando consegue utilizar de forma eficiente os recursos disponíveis em cada link e nó da rede, de modo que sobrem o máximo de recursos possíveis.

De inicio temos a nossa rede fisíca(substrato), a qual sera representada por meio do Grafo 1 o qual contém 112 nós interligados, logo em seguida carregamos 10 requisições, a qual o conteúdo de cada uma corresponde a seguinte formatação de exemplo:

'cpu': 89, **'bandwidth'**: 5621, **'qtd-nos'**: 3, **'topologia'**: [[0, 2], [1, 2]]

onde a **cpu** corresponde ao valor minimo de processamento solicitado, a **bandwith**(largura de banda) ao tamanho da quantidade de banda minima entre os nós,

a **qtd-nos** que se refere a quantidade de nós que devem ser mapeados e por fim a **topologia** a qual define a topologia em que esses nós estarão totalmente ou parcialmente interligados entre si. Porém a rede fisíca ainda não possuia os valores de CPU e largura de banda. Dito isso, foram gerados com valores aleatório, de 10 a 100 para cpu e de 100 a 10000 para largura de banda. Ademais, a população inicial foi gerada de modo a ser fixa, para evitar que a aleatoriedade influencie de maneira errônea na comparação dos testes, onde a mesma possui 20 individuos com 3 nós cada. Em relação ao tratamento das requisições, inicialmente, foram removidas as que solicitavam mais do que a rede fisica dispõe, em sequencia foram ordenadas de modo a ser atendidas primeiramente as que solicitavam menos processamento e largura de banda, ou seja, de maneira crescente.

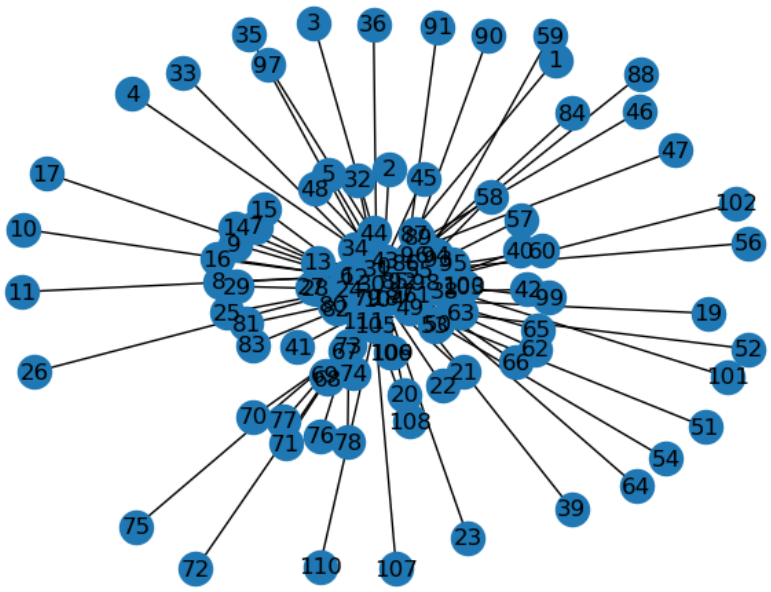

Figura 1. Grafo do substrato.

De modo a espelhar a explicação do código com a ideia do algoritmo genético, em seguida será explicado tal relação tomando como base a Figura 2 que ilustra bem a sequencia de passos seguidos para a exemplificação do algoritmo.

O ciclo de um algoritmo genético geralmente é composto pelos seguintes passos:

1. **População Inicial**: Inicia-se o algoritmo com a criação de uma população inicial de indivíduos. Cada indivíduo é representado por um cromossomo, que é uma cadeia de bits ou valores numéricos que codificam uma solução para o problema em questão. A população inicial nesse caso, como ja mencionada anteriormente, possui 3 nós em cada individuo.
2. **Avaliação**: Cada indivíduo da população é avaliado de acordo com sua aptidão, que é uma medida da qualidade da solução representada pelo cromossomo. A aptidão pode ser definida de várias formas, dependendo do problema em questão. No nosso caso se o mapeamento não atingir a quantidade minima o resultado da função de avaliação(fitness) sera negativo, caso mapeie na mesma quantidade o resultado será 0, e caso ultrapasse a quantidade minima o resultado será positivo.

Figura 2. Metodologia do algoritmo genético.

Ademais, nos 2 casos de resultados da avaliação, tanto o negativo como o positivo quanto mais extremos ou distantes do ponto 0, maior será a descriminação da sua avaliação.

3. **Seleção**: Os indivíduos mais aptos são selecionados para a próxima geração. Existem vários métodos de seleção, sendo os mais comuns a seleção por torneio, a seleção por roleta e a seleção por classificação. No nosso caso, foi utilizado o metodo seletivo da roleta, o método funciona selecionando indivíduos para reprodução com base em uma probabilidade proporcional à sua aptidão (função fitness) em relação à população total. O processo começa calculando a aptidão de cada indivíduo na população. Em seguida, é calculada a soma total de todas as aptidões. Em seguida, cada indivíduo é atribuído a uma fatia da roleta, proporcional à sua aptidão relativa. Isso significa que indivíduos com maior aptidão terão uma fatia maior da roleta, aumentando suas chances de serem selecionados.
4. **Cruzamento**: Os indivíduos selecionados são cruzados para gerar novos indivíduos. O cruzamento é realizado trocando partes dos cromossomos entre dois indivíduos. O objetivo é criar novas soluções a partir das soluções já existentes na população. No nosso caso, o código implementa um crossover de ponto único(função crossover), em que o ponto de corte é definido pela metade do número de nós do indivíduo. Ele seleciona aleatoriamente qual dos pais fornecerá os nós para cada posição no novo indivíduo. Se um nó já estiver presente no novo indivíduo, a função escolhe aleatoriamente um nó que ainda não foi adicionado. O resultado é um novo indivíduo que é uma combinação dos pais selecionados.
5. **Mutação**: Em alguns casos, é introduzido um pequeno grau de aleatoriedade no processo de cruzamento, para evitar a convergência prematura para uma solução sub-ótima. A mutação consiste em alterar aleatoriamente alguns dos bits ou valores numéricos dos cromossomos. Em nosso código a mutação é aplicada em

cada filho gerado pelo operador de crossover, com uma probabilidade definida pela variavel 'prob-mutacao' que é parâmetro na função **fit**. Como o operador de crossover é executado em todos os indivíduos da população, cada indivíduo tem uma chance de ser mutado a cada geração. Isso ajuda a aumentar a diversidade da população e a evitar que a busca fique presa em um mínimo local.No entanto, como a probabilidade de mutação colocada foi 0.1 sendo relativamente baixa, espera-se que a maioria dos indivíduos na população permaneça relativamente semelhante ao seu estado original.

6. **Atualização**: A nova população é criada a partir dos indivíduos resultantes do cruzamento e da mutação. A população anterior é descartada.
7. **Finalização**: O algoritmo é executado por um número fixo de gerações ou até que uma solução satisfatória seja encontrada. O resultado final é a melhor solução encontrada durante a execução do algoritmo. No nosso caso, nos só usamos o número de época.
8. **Solução final**: A solução final é obtida selecionando o individuo mais apto da ultima geração.

Dito isso, neste trabalho foram implementadas a versão Sequencial, Paralelo e Distribuido. Na versão paralela foi utilizada **concurrent.futures.ThreadPoolExecutor** onde é uma classe da biblioteca padrão do Python que permite a execução concorrente de tarefas em threads. Um executor é basicamente um objeto que gerencia um pool de threads para executar tarefas. As tarefas são colocadas em uma fila (chamada de "fila de tarefas"), e quando um thread estiver livre, ele pega a próxima tarefa da fila e a executa. Já na versão Distribuida, foi utilizada **Docker**. Com o Docker, foi possível criar imagens de contêineres que continham todo o ambiente necessário para executar o código e a criação diversos container utilizados para processamento, ajudando a processar as funções **crossover**, **mutação** e **calculo da pontuação**.

	Épocas		
Versão	30	750	2000
Sequencial	2,61 segundos	38,08 segundos	238,09 segundos
Paralelo	1,61 segundos	58,97 segundos	176 segundos
Distribuído	36,88 segundos	830,67 segundos	X

Tabela 1. Tempo de execução em segundos de cada versão em diferentes épocas.

Na Tabela 1 estão os resultados dos testes com ambas versões e diferentes épocas com seu tempo de processamento gasto respectivamente.

3. Análise e Resultados da Implementação

A partir da análise da Tabela 1, é possível observar que o algoritmo genético paralelo obteve resultados superiores aos demais algoritmos em duas situações específicas: com 30 e 2000 épocas. Por outro lado, o algoritmo sequencial se destacou como o melhor em uma terceira situação, com 750 épocas. Já o algoritmo distribuído apresentou os piores resultados em todas as situações avaliadas.

O algoritmo genético paralelo pode ter se destacado com 2000 épocas devido à sua capacidade de calcular múltiplas soluções simultaneamente. Já em situações com

menos épocas (750), o algoritmo sequencial pode ter sido mais eficiente por conseguir encontrar uma solução ótima mais rapidamente, já que possui menos operações do que um algoritmo paralelo.

Por outro lado, o algoritmo distribuído pode ter apresentado desempenho inferior devido a comunicação em rede, o que pode ter aumentado o tempo de processamento.

4. Conclusão

O propósito desse esperimento foi realizar o mapeamento com os três métodos, de forma squencial, paralela e distribuida, de forma que mapear uma rede física utilizando os métodos sequencial, paralelo e distribuído Os resultados indicaram que o algoritmo distribuído foi o mais rápido em termos de tempo de execução, seguido pelo algoritmo paralelo e, por último, pelo algoritmo sequencial. Além disso, foi observado que a utilização de hardware especializado pode ser benéfica para aumentar ainda mais a eficiência do processo. Em suma, esses resultados podem ser úteis para profissionais de redes de computadores que buscam otimizar o tempo de mapeamento de suas redes físicas.

Qual o propósito do experimento? Tal propósito foi alcançado? Quais foram os erros e dificuldades? Quais foram superados e como? Quais foram os resultados do experimento? Que informação foi descoberta com a execução dos programas? O que foi aprendido com a realização dos experimentos? Com os experimentos foram adquiridos conhecimentos sobre algoritmo genético, slicing de rede, paralelização em python, comunicação em rede em python e conteiner em docker. Além disso, também

Algumas melhorias podem ser exploradas, como: mapear outra forma de distribuição do algoritmo genético e otimizar o código em python de todas as versões

5. Apêndice

Fragmento dos codigos:

```python
def atualiza_individuo(self, individuo):
    individuo['cpu'] = min([self.substrato.nodes[no]['cpu'] for no in individuo['nos']])

    try:
        bandwidths = []

        for no1, no2 in self._VNR['topologia']:
            caminhos = nx.dijkstra_path(self.substrato, individuo['nos'][no1], individuo['nos'][no2])
            bandwidths += [self.substrato.edges[caminhos[i], no]['bandwidth'] for i, no in enumerate(caminhos[1:])]

        individuo['bandwidth'] = min(bandwidths)
        individuo['qtd_saltos'] = len(bandwidths)
    except:
        individuo['bandwidth'] = 0
        individuo['qtd_saltos'] = 0

    return individuo
```

Figura 3. Função para atualizar individuo.

Referências

```python
def fitness(self, VNR):
    cpu = (VNR['cpu'] - self._VNR['cpu']) / self._max_cpu
    bandwidth = (VNR['bandwidth'] - self._VNR['bandwidth']) / self._max_bandwidth
    saltos = (VNR['qtd_saltos'] - len(self._VNR['topologia'])) / self.substrato.number_of_edges()

    if cpu >= 0 and bandwidth < 0:
        return bandwidth
    elif cpu < 0 and bandwidth >= 0:
        return cpu

    return cpu + bandwidth - saltos
```

Figura 4. Função fitness sequencial.

```python
def calcular_pontuacao(self, populacao):
    return [self.fitness(x) for x in populacao]

def calculo_roleta(self, pontuacoes):
    menor_pontuacao = min(pontuacoes)

    pontuacoes_positivadas = list(map(lambda pontuacao: pontuacao-menor_pontuacao, pontuacoes))

    maior_pontuacao = max(pontuacoes_positivadas)

    pontuacoes_positivadas_reverso = list(map(lambda pontuacao: (maior_pontuacao-pontuacao) * 5, pontuacoes_positivadas))

    roleta = []
    total = sum(pontuacoes_positivadas_reverso)
    for i, valor in enumerate(pontuacoes_positivadas_reverso):
        repeticao = round(abs((total+1)/(valor + 1)))

        roleta.extend([i]*repeticao)

    return roleta
```


Figura 5. Funções de calculo da pontuação e da roleta.

```python
def crossover(self, populacao, g1, g2):
    # {'nos': [109, 105, 66], 'cpu': 0, 'bandwidth': 100, 'qtd_saltos': 5}
    qtd_nos = len(populacao[g1]['nos'])

    new_individuo = {'nos': [-1 for _ in range(qtd_nos)]}

    metade = int(np.ceil(qtd_nos/2))

    indices_dos_nos = list(range(qtd_nos))
    random.shuffle(indices_dos_nos)

    if random.random() < 0.5:
        g1, g2 = g2, g1

    g = g1
    for i, indice in enumerate(indices_dos_nos):
        if i == metade:
            g = g2

        if populacao[g]['nos'][indice] not in new_individuo['nos']:
            new_individuo['nos'][indice] = populacao[g]['nos'][indice]
        else:
            nos_possiveis = list(set(range(self.substrato.number_of_nodes())).difference(set(new_individuo['nos'])))
            new_individuo['nos'][indice] = random.choice(nos_possiveis)

    return self.atualiza_individuo(new_individuo)
```

Figura 6. Funções de crossover sequencial.

Figura 7. Funções de mutação paralela.

```
servers = []
for i in range(2, 2+5):
    servers.append(Pyro4.core.Proxy(f"PYRO:Server@192.167.1.{i}:9090"))
```

Figura 8. Conectando containers utilizando Pyro.

```
  server5:
    image: server
    ports:
      - "9094:9090"
    networks:
      dist-net:
        ipv4_address: 192.167.1.6

  cliente:
    image: alg_genet
    networks:
      - dist-net
    depends_on:
      - server1
      - server2
      - server3
      - server4
      - server5

networks:
  dist-net:
    driver: bridge
    ipam:
      config:
        - subnet: 192.167.1.0/12
          gateway: 192.167.1.1
```

Figura 9. Esboco do docker$_{compose}$.

```
FROM python:3

RUN pip install Pyro4 && pip install futures && pip install networkx

WORKDIR /app/

COPY ./alg_genet.py .
COPY ./infra.gml .
COPY ./populacao_inical.json .
COPY ./vnr.json .

CMD ["python", "/app/alg_genet.py"]
```

Figura 10. Dockerfile.