Trabalho de implementação

Computação Concorrente (MAB-117) - 2021/2

Problema do caminho mínimo - Algoritmo de Floyd-Warshall

Ellen Almeida de Souza 120041556
Kevin Sena de Andrade 120085762

1. Descrição do problema

O *problema do caminho mínimo ou mais curto* consiste em encontrar a menor distância entre dois pontos. Nesse problema geralmente temos um grafo dirigido ponderado $G = (V, E)$ com função peso $f: E \rightarrow R$ que mapeia arestas para pesos de valores reais. O peso do caminho $p = \left(v_1, v_2, ..., v_k\right)$ é dado pela soma do valor de cada aresta nesse caminho. Dessa forma, o problema do caminho mínimo pode ser representado matematicamente por:

$$f(p) = \sum_{i=1}^{k} f\left(v_{i-1}, v_i\right).$$

A partir dessa definição geral para o problema, é possível desenvolver algumas vertentes, como por exemplo, ***problema do caminho mínimo para um par*** que busca o menor caminho entre dois nós dados; ***problema dos caminhos mínimos com destino único***, o qual tem por objetivo determinar o menor caminho entre cada um dos nós do grafo e um nó de destino final; ***problema de única origem*** que busca o menor caminho entre um nó dado e todos os demais nós do grafo e ***problema de caminhos mínimos para todos os pares de vértices*** que procura o menor caminho para cada par de nós contidos no grafo.

Há diversos algoritmos para solucionar estes problemas mencionados acima, mas daremos enfoque apenas para o último problema: menor caminho para todos os pares de nós de um grafo. Usaremos o algoritmo de Floyd-Warshall, cuja complexidade é $O(n^3)$, pois ele possui 3 laços aninhados para chegar ao resultado final. Além disso, o algoritmo de Floyd-Warshall aceita pesos negativos, mas não ciclos negativos.

O algoritmo de Floyd, também conhecido como algoritmo de WFI, apesar de não especificar quais vértices (ou nós) configuram os caminhos, é bem eficiente ao retornar o valor da distância mais curta, permitindo

encontrar o caminho propriamente dito com uso de outros métodos caso queira.

Esse algoritmo parte da matriz de distância e faz uma quantidade $k = |V|$ de passos (que é quantidade de vértices do grafo). A cada passo o objetivo é determinar uma matriz M^k, cuja k-ésima coluna, k-ésima linha e diagonal principal (que sempre será preenchida por 0, pois a distância de um nó para ele mesmo é 0) serão fixas. Para determinar tal matriz devemos seguir a seguinte condição:

$$d_{ij} = \begin{cases} d_{ik} + d_{kj} & \text{se } d_{ij} > d_{ik} + d_{kj} \\ d_{ij} & \text{se } d_{ij} \leqslant d_{ik} + d_{kj} \end{cases}$$

Ou simplesmente,

$$d_{ij} = min(d_{ij}, d_{ik} + d_{kj})$$

Após essa visão inicial do problema, temos como dado de entrada um arquivo que contém duas informações: primeiro o tamanho da matriz e, segundo, a própria matriz distância. O algoritmo deve retornar também uma matriz, a qual relaciona cada par de vértice a seu caminho de menor custo. A saída do algoritmo não será a saída padrão do terminal, mas sim em um arquivo separado.

Para exemplificar melhor todo esse cenário, tomemos a matriz distância abaixo.

$k = 0 = A, M^k = M^0 = M^A$:

	A	**B**	**C**	**D**
A	**0**	**3**	∞	**7**
B	**8**	**0**	**2**	∞
C	**5**	∞	**0**	**1**
D	**2**	∞	∞	**0**

Como $k = 0 = A$, então a primeira linha e a primeira coluna ficarão fixadas, e por padrão, a diagonal principal também. Agora temos

que percorrer a matriz e verificar cada célula. Por exemplo, para calcular a posição d_{12}, ou seja d_{BC}, temos que calcular:

$$d_{ij} = min(d_{ij}, d_{ik} + d_{kj})$$

$$d_{12} = min(d_{12}, d_{10} + d_{02})$$

$$d_{BC} = min(d_{BC}, d_{BA} + d_{AC})$$

$$d_{BC} = min(2, 8 + \infty)$$

Como $2 < 8 + \infty$, então a posição d_{BC} se mantém. Entretanto, ao calcularmos a posição d_{13}, ou seja d_{BD}, teremos:

$$d_{ij} = min(d_{ij}, d_{ik} + d_{kj})$$

$$d_{13} = min(d_{13}, d_{10} + d_{03})$$

$$d_{BD} = min(d_{BD}, d_{BA} + d_{AD})$$

$$d_{BD} = min(\infty, 8 + 7)$$

Como $8 + 7 < \infty$, então a posição $d_{BD} = 8 + 7 = 15$. Portanto, a matriz agora será:

	A	**B**	**C**	**D**
A	**0**	**3**	∞	**7**
B	**8**	**0**	**2**	**15**
C	**5**	∞	**0**	**1**
D	**2**	∞	∞	**0**

E após o algoritmo percorrer todas as linhas para $k = 0 = A$, teremos:

	A	B	C	D
A	**0**	**3**	∞	**7**
B	**8**	**0**	**2**	**15**
C	**5**	**8**	**0**	**1**
D	**2**	**5**	∞	**0**

Agora para $k = 1 = B$ devemos repetir novamente todo o percurso para cada linha da matriz, mas com a linha e coluna B fixadas e a diagonal principal. Faremos todas essas repetições até que finalmente executemos os passos para $k = 3 = D$. No fim do algoritmo o resultado será:

	A	B	C	D
A	**0**	**3**	**5**	**6**
B	**5**	**0**	**2**	**3**
C	**3**	**6**	**0**	**1**
D	**2**	**5**	**7**	**0**

Abaixo o algoritmo de Floyd-Warshall em pseudocódigo que recebe como entrada uma matriz distância e retorna a própria matriz com os caminhos mais curtos para cada par de vértice:

FUNC FLOYD-WARSHALL($dist$)
    PARA k DE 1 A $|V|$
        PARA i DE 1 A $|V|$
            PARA j DE 1 A $|V|$
                $dist_{ij} = min(dist_{ij}, dist_{ik} + dist_{kj})$
    RETORNE $dist$

Analisando com cautela o funcionamento do algoritmo, podemos perceber que como a k-ésima linha, k-ésima coluna e a diagonal principal

não mudam, então podemos pulá-las quando $i = k$, $j = k$ e $i = j$. Além disso, para cada iteração de k, cada passo de i é executado de forma independente. Isso porque o cálculo de cada célula é baseado em duas células da k-ésima linha e coluna, que são fixas para cada k. Logo, podemos concluir que o algoritmo de Floyd-Warshall pode ser implementado em uma versão concorrente.

2. Projeto e implementação da solução concorrente

Tendo em vista que há uma possível solução concorrente para solucionar o problema do caminho mínimo, podemos analisar como implementar essa solução.

Uma primeira solução para resolver o problema usando concorrência é: para cada iteração de k, criamos uma thread e atribuímos a ela a tarefa de calcular a M^k. Assim, teríamos um balanceamento de tarefas igualitário entre as threads. Porém, como cada passo depende do passo anterior, não haveria a possibilidade de paralelização entre os fluxos de execução, já que cada thread teria que esperar a thread anterior terminar para fazer seu processamento. Logo, a execução desse algoritmo ficaria completamente sequencial, mas com um processamento maior com a criação das threads e com a sincronização entre elas.

Pensando em outra proposta de solução temos que, como dito anteriormente, a cada passo de k, as iterações de i são realizadas de forma independente, logo, a solução concorrente é: para cada i, criamos uma thread e atribuímos a ela a tarefa de calcular uma linha da matriz. Logo, pensando em balanceamento de processamento entre as threads, cada thread executará a mesma quantidade de linhas (nesse caso $|V|$ linhas, pois $0 < k < |V|$) ao final do processo. Isso porque, por mais que possamos ignorar a linha fixada, sabemos que o k avança a cada iteração, assim, cada thread se beneficiará do "privilégio" de pular uma linha, garantindo que todas trabalhem igualmente.

Entretanto, pensando em possíveis entradas grandes, ficaria muito custoso criar uma thread para cada i. Por exemplo, numa matriz de distância 50x50, criar 50 threads não trará bons ganhos para a aplicação, visto que não será possível paralelizar os 50 fluxos de execução.

Nesse sentido, uma outra solução concorrente para esse problema é: ao invés de criar uma thread para cada linha, criamos um número menor de threads (respeitando os limites da máquina), permitindo uma

paralelização de fluxos de execução. Dessa forma, separamos as tarefas entre as threads dividindo o número de linhas (ou colunas) da matriz pelo número de threads, assim, cada thread ficaria encarregada de executar as $\frac{|V|}{N° \, threads}$ linhas consecutivas da matriz.

Contudo, pensando novamente em balanceamento de execuções entre as threads, teremos casos em que uma thread trabalhará menos que as outras, pois nem sempre essa divisão será exata, logo, pelo menos uma thread receberá menos linhas para computar. Além disso, cada thread precisa saber por onde começar, então é necessário tratar essa divisão e atribuir corretamente para cada thread a linha inicial e final da sua tarefa.

Por fim, uma última proposta de solução é: novamente criamos um número menor de threads, permitindo uma paralelização de fluxos de execução, porém, diferentemente da proposta anterior, não dividimos o número de linhas da matriz pelo número de threads, mas sim atribuímos às threads linhas alternadas da matriz. Por exemplo, em uma matriz de distância 4x4 e utilizando apenas 2 threads, podemos distribuir as tarefas da seguinte forma: a thread 1 executa as linhas 0 e 2 e a thread 2 executa as linhas 1 e 3.

Novamente, em relação ao balanceamento de tarefas entre as threads, o mesmo ocorre na solução anterior, uma thread pode receber menos linhas para calcular em caso de a quantidade de threads for par e o tamanho da matriz ser ímpar, ou vice-versa. Entretanto, também em ambos os cenários, não será uma diferença tão significativa.

Desse modo, por conta de uma pequena facilidade no momento da implementação e um melhor desempenho, em relação às outras propostas, decidimos implementar essa última solução em nosso trabalho.

Considerando a proposta de resolução concorrente apresentada, podemos analisar mais especificamente o problema a nível de implementação. Já sabemos que a cada k iteração, as threads executarão suas respectivas i linhas. Porém, não pode ocorrer de, por exemplo, enquanto um thread executa suas linhas no passo $k = 1$, outra estar executando no passo $k = 2$, pois assim, teríamos linhas e colunas diferentes fixadas, podendo ocasionar em sobrescrita de resultado. Desse modo, é necessário que as threads percorram sempre em fase (mesma iteração de k) durante a execução e um jeito de garantir essa situação é utilizando **sincronização coletiva com barreiras**, em que todas as threads só executarão a nova bateria quando todas estiverem terminado o

passo atual. Em outras palavras, após terminar suas linhas em um certo passo, cada thread deve esperar pela(s) outra(s) para avançar para o próximo passo, correspondendo exatamente ao arranjo de uma barreira em códigos concorrentes.

3. Casos de teste

Pensando em facilitar a atividade de execução dos programas sequencial e concorrente, desenvolvemos um código que retorna uma matriz distância, recebendo como entrada dois dados: primeiro, o nome de um arquivo com extensão que permita escrita e, segundo, um número inteiro representando a dimensão da matriz distância. Dessa forma, automatizamos a criação de entradas para o nosso programa que calcula as distâncias mínimas.

O algoritmo de geração de entradas explicado acima, gera uma matriz com dados aleatórios (número ou infinito), respeitando a regra de que a diagonal principal é composta apenas de zeros, pois consideramos que a distância de um nó para ele mesmo é 0. Por consequência, nenhum outro campo que não esteja na diagonal principal pode constar 0.

Esse código foi feito aplicando, em cada posição [x][y] da matriz, uma probabilidade de 50% para um número (entre 1 e 9) e 50% para infinito, sendo o infinito a representação de que não há caminho direto do vértice x para o vértice y.

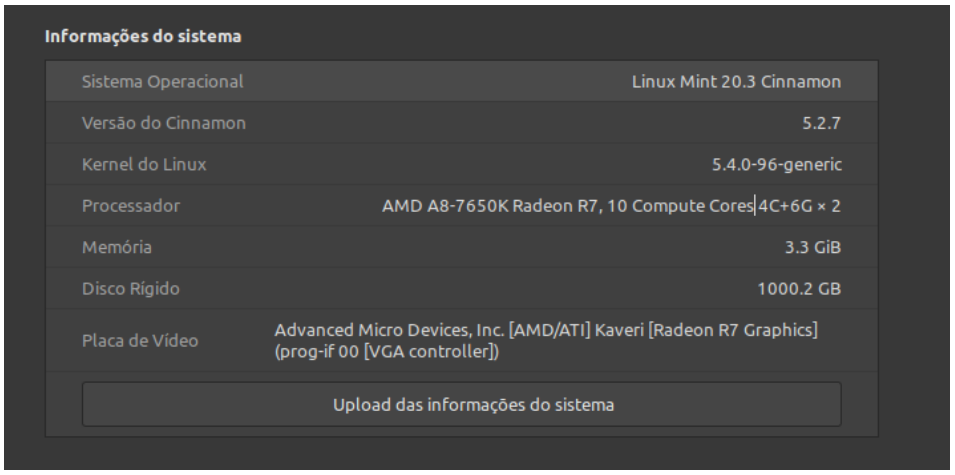
A partir dessa aplicação, foram geradas 12 matrizes de distância para 3 casos de testes com dimensões 250x250, 500x500 e 1000x1000. É importante ressaltar que o código principal, o qual calcula as distâncias mínimas, apenas funciona em matrizes quadradas. Então, caso não se use o algoritmo de geração de entradas, é necessário passar como entrada um arquivo que contenha na primeira linha o tamanho da matriz e em seguida uma matriz quadrada que respeite também as regras características de uma matriz de distância. A aplicação principal não verifica estes casos, pois entende-se que o requisito para o cálculo é uma matriz de distância quadrada, além disso, o custo operacional para essa checagem seria alto em termos de tempo de execução.

Para análise de corretude, criamos um programa em python que irá executar o código que gera as matrizes e os algoritmos de Floyd-Warshall (sequencial e concorrente). Para isso, definimos um array contendo as dimensões citadas acima e, para cada dimensão, o algoritmo em python

executará o gerador de matrizes e rodará ambos programas de Floyd-Warshall e comparando, ao fim, a saída deles (dois arquivos com extensão .txt) tirando a diferença com o comando *diff*. Se o comando não retornar nada, significa que os arquivos são iguais, caso contrário, o programa em python será interrompido e a mensagem de erro sairá no terminal.

4. Avaliação de desempenho

Todos os resultados experimentais foram executados em um computador doméstico com o sistema operacional Linux Mint, cujas configurações encontram-se na figura abaixo.



Para analisar o desempenho de um forma mais fácil e garantida, também foi desenvolvido um algoritmo em python que roda os programas sequencial e concorrente com os tamanhos de matriz referente aos casos de teste e com diferentes configurações de concorrência: 1, 2, 3 e 4 thread(s). Os tempos de execução foram obtidos em segundos tomando-se o menor valor decorrente das 5 execuções para cada tamanho de matriz. A aceleração foi calculada por meio da Lei de Amdahl, em que dividimos o tempo da execução sequencial pelo tempo da execução concorrente. Por exemplo, tomando uma execução sequencial para a dimensão 500x500 e uma execução concorrente utilizando 2 threads para a mesma dimensão, temos: $T_{seq} = 0.769480$ e $T_{conc} = 0.414383$, logo a aceleração será:

$$A = \frac{T_{seq}}{T_{conc}} = \frac{0.769480}{0.414383} = 1.856929$$

Portanto, a solução concorrente nesse caso foi 1.856929 vezes mais rápida que a solução sequencial. Abaixo temos uma tabela com os resultados obtidos nos testes realizados para 1, 2, 3 e 4 threads.

N° threads	Dimensão	Tempo sequencial	Tempo concorrente	Aceleração
1	250x250	0.099669	0.101017	0.986656
	500x500	0.769480	0.763217	1.008206
	1000x1000	6.243462	6.179035	1.010427
2	250x250	0.099669	0.056061	1.777867
	500x500	0.769480	0.414383	1.856929
	1000x1000	6.243462	3.215943	1.941409
3	250x250	0.099669	0.041589	2.396523
	500x500	0.769480	0.276588	2.782044
	1000x1000	6.243462	2.178887	2.865436
4	250x250	0.099669	0.036076	2.762751
	500x500	0.769480	0.227274	3.385693
	1000x1000	6.243462	1.934262	3.227826

5. Discussão

Analisando-se os resultados obtidos no teste de desempenho, conclui-se que a aceleração é maior quando temos 4 threads, isso acontece porque na máquina usada para a testagem temos 4 processadores, assim, com essa configuração, usamos o máximo dos recursos aplicando a concorrência para o cálculo das linhas a cada passo. Portanto, a aplicação concorrente consegue dividir a tarefa e distribuir entre as threads, possibilitando a paralelização da execução das subtarefas, as quais as threads ficam responsáveis e, consequentemente, fazendo com que o tempo fique menor em relação ao código sequencial.

Apenas para confirmar que atingimos o melhor desempenho com 4 fluxos de execução, rodamos o código de desempenho em python com 5 threads e obtivemos os seguintes resultados:

N° threads	Dimensão	Tempo sequencial	Tempo concorrente	Aceleração
5	250x250	0.099094	0.055032	1.800661
	500x500	0.768124	0.372103	2.064278
	1000x1000	6.223599	2.572633	2.419155

Comparando com a aceleração atingida com 4 threads, constata-se que 5 fluxos de execução não provoca melhoria no desempenho, pelo contrário, o resultado é pior. Logo, é possível afirmar que o máximo de otimização de tempo é alcançado com uso de uma quantidade de threads relacionada ao número de processadores da máquina utilizada.

O código concorrente atende às expectativas de otimização de tempo para se alcançar o resultado desejado (a matriz com as menores distâncias) por conta da paralelização, ou seja, há vários fluxos de execução trabalhando ao mesmo tempo em cima da matriz, respeitando a barreira implementada. Isso faz com que, apesar da necessidade da sincronização coletiva entre as threads, a concorrência possibilita ganho de desempenho por conta de várias linhas da matriz serem executadas de forma simultânea.

Devido ao fato de ser necessário que todas as threads trabalhem usando como base as mesmas informações fixadas a cada passo, pois caso contrário, obteremos resultados inconsistentes, não é possível remover o mecanismo de sincronização com barreiras. Sendo assim, a aplicação concorrente apresentada alcança o máximo de ganho de desempenho possível para resolver o problema do caminho mínimo.

Um outro ponto importante a ser mencionado é referente à dificuldade inicial enfrentada para fazer a tomada de tempo de forma adequada. Inicialmente tentamos usar a função *clock()* da biblioteca *time.h*, mas os resultados gerados não estavam atendendo às expectativas, pois em alguns casos o programa concorrente tinha desempenho pior que o sequencial, mas após algumas pesquisas, constatamos que esse recurso

não funciona bem com códigos que fazem uso de fluxos de execução e não é adequado para testes sucessivos.

Por fim, conseguimos implementar o código disponibilizado pela docente Silvana Rossetto, responsável pela disciplina Computação Concorrente, para fazer a análise do tempo. Então, finalmente, foi possível observar corretamente a atuação dos códigos concorrente e sequencial.

Levando-se em consideração o exposto acima, conclui-se que a programação concorrente foi bastante útil para ganho de desempenho na aplicação que resolve o problema dos caminhos mínimos para cada par de vértice de um grafo dirigido e ponderado, utilizando o algoritmo de Floyd-Warshall.

6. Referências bibliográficas

[1] CORMEN, Thomas H. et al. Algoritmos: teoria e prática. 3ª edição. Rio de janeiro: Elsevier, 2012.

[2] Wikipédia: a enciclopédia livre. Problema do caminho mínimo. Disponível em <https://pt.wikipedia.org/wiki/Problema_do_caminho_m%C3%ADnimo> . Acesso em 21 de janeiro de 2022.

[3] Wikipédia: a enciclopédia livre. Algoritmo Floyd – Warshall - Floyd–Warshall algorithm. Disponível em <Algoritmo Floyd – Warshall - Floyd–Warshall algorithm - abcdef.wiki> . Acesso em 22 de janeiro de 2022.

[4] STUMM, Valdir. Argumentos da linha de Comando. Python Help, 2011. Disponível em <https://pythonhelp.wordpress.com/2011/11/15/argumentos-da-linha-de-comando/ >. Acesso em 28 de janeiro de 2022.

[5] MARINHO, Thiago. Como fazer um bom README. Blog da Rocketseat, 2020. Disponível em <https://blog.rocketseat.com.br/como-fazer-um-bom-readme/>. Acesso em 1 de fevereiro de 2022.