BFS Concorrente

Gabriel Martins de Freire - 123449127
Matheus Avila Abreu de Lima - 123255293
Paulo Roberto Ferreira de Godoy Moreira - 123451653

Relatório Parcial

Programação Concorrente (ICP-361) – 2025/2

1. Descrição do problema geral

Um grafo é uma estrutura matemática usada para representar relações entre elementos. Ele é formado por um conjunto de **vértices** (também chamados de *nós*) e um conjunto de **arestas**, que representam as conexões entre pares de vértices.

Formalmente, podemos definir um grafo como:

$$G = (V, E)$$

onde:

- V é o conjunto de vértices;
- $E \subseteq V \times V$ é o conjunto de arestas.

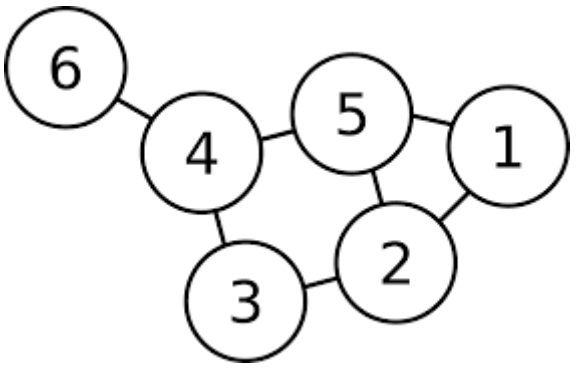

Figura 1: Esquema de um grafo, em que cada vértice é ilustrado por um círculo numerado e cada aresta por uma linha conectando pares de vértices.

Pela possibilidade de implementações de grafos em diversas problemáticas do mundo real, é do interesse de cientistas da computação analisar e resolver problemas baseados em grafos. Um desses problemas é o de percorrer todos os

nós de um dado grafo, obtendo as distâncias mínimas – em quantidade de nós visitados – de um nó em relação aos demais.

Existem diversos algoritmos que percorrem essas estruturas matemáticas a fim de extrair essa informação, sendo um dos mais famosos a Busca em Largura (mais conhecido como Breadth First Search).

Durante a execução deste programa, é calculada a relação de distância entre o vértice inicial e todos os demais vértices alcançáveis no grafo. Essas distâncias correspondem ao número **mínimo** de arestas que precisam ser percorridas para ir do vértice de origem até cada outro vértice, conforme determinado pelo algoritmo.

O algoritmo, então, percorre o grafo **nível a nível.** Primeiro, ele visita todos os vértices a uma distância 1 do vértice inicial, depois os vértices a uma distância 2, e assim sucessivamente. Dessa forma, ao final da execução, é possível associar a cada vértice um **nível** (ou distância mínima) em relação ao vértice de origem.

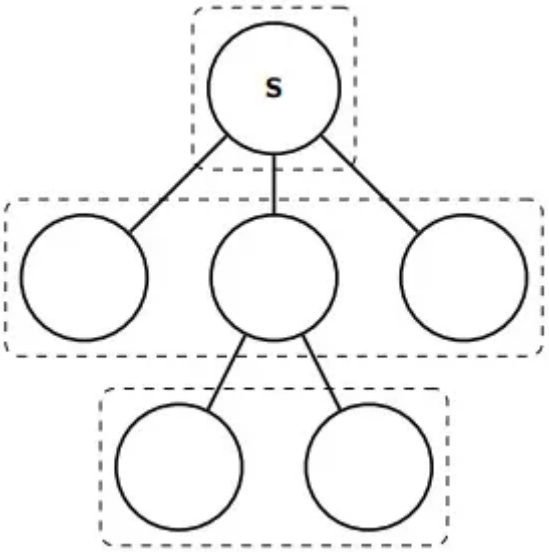

Figura 2: Representação visual da exploração em camadas realizada pelo algoritmo, mostrando a divisão do grafo nível a nível.

A entrada para o programa inclui o **número de vértices do grafo** e a **quantidade total de arestas**, seguidos de tuplas de dois números que representam **cada aresta**, e, na última linha, um valor que representa o **vértice inicial** para o BFS. Por exemplo, se o usuário fornecer como entrada:

– Input –
4 5
1 2
3 4
3 2

2 4
1 3
1

Teremos um grafo com 4 vértices, e 5 arestas, e o vértice de partida para o BFS será o nó de número 1.

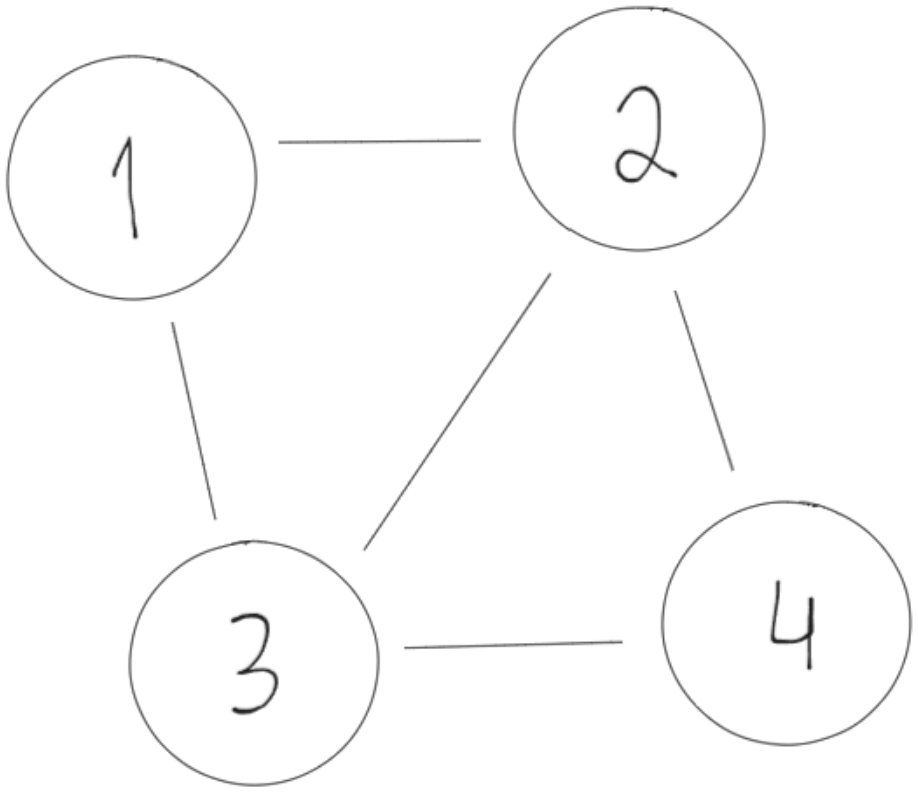

Figura 3: Ilustração do grafo exemplo de entrada do programa

Portanto, a saída do programa será uma série de linhas, com cada linha representando uma das **camadas calculadas no BFS**, com os vértices pertencentes aquela camada. No exemplo anterior, teríamos como saída:

– Output –
Nível 0: 1
Nível 1: 2, 3
Nível 2: 4

Como é possível notar, o algoritmo de Busca em Largura, em sua forma sequencial, apresenta um fluxo bastante linear: a próxima camada apenas será calculada quando a camada atual for completamente finalizada. Dessa forma, o

programa pode ser computacionalmente custoso em grafos densos e com muitos vértices.

Perceba que vértices pertencentes à mesma camada são independentes entre si (suas arestas apontam para camadas já percorridas ou para a próxima camada, com sua execução não afetando as demais do mesmo nível). Aproveitando-se deste fato, podemos utilizar a programação multi-thread. Assim, dividimos o trabalho de exploração entre várias unidades de execução, e diminuímos o tempo total necessário para percorrer o grafo.

Para casos de grafos com poucos vértices e arestas, o uso de mecanismos de sincronização pode impactar negativamente no uso do paralelismo, tornando a solução sequencial mais eficiente. Contudo, espera-se que, em grafos densos, o algoritmo paralelo obtenha um desempenho significativamente melhor, por se aproveitar do paralelismo de dados, por reduzir o tempo total médio de execução, e por utilizar melhor os recursos computacionais disponíveis.

2. Projeto da solução concorrente

Enquanto discutia sobre o problema, o grupo pensou em duas principais estratégias para projetar a solução concorrente:

- **Estratégia 1**: Fila compartilhada globalmente pelas threads

  Nessa estratégia, uma fila global é compartilhada por todas as threads. Essa estrutura de dados armazena os vértices que ainda não foram selecionados para explorar no algoritmo do BFS, e as threads retiram os valores dos vértices, quando disponíveis, processa seus vizinhos, e os adiciona para a fila. Na verdade, esta implementação segue um padrão na forma de produtor/consumidor, sendo cada thread consumidora e produtora simultaneamente. Ademais, o acesso a fila precisaria estar protegido por um lock.

  Os benefícios dessa abordagem incluem a distribuição dinâmica do trabalho (threads que “terminam mais cedo” são capazes de consumir mais vértices), e a simplicidade da implementação. Contudo, o constante lock e unlock da fila global poderia diminuir a eficiência do programa concorrente frente ao sequencial.

- **Estratégia 2**: Divisão da exploração por nível

  Nessa estratégia, aproveita-se o funcionamento de “camadas” do BFS, fazendo todos os vértices de um determinado nível serem explorados em paralelo, mas apenas um nível por vez (para garantir a corretude do programa). Por meio de uma **barreira** (como visto em sala), podemos garantir a sincronização entre as threads. Ademais, os vértices que deverão

ser explorados na próxima camada são incorporados a uma fila local para cada thread, e depois incorporadas (junto de todas as outras threads) a uma fila global, reduzindo a quantidade de locks e unlocks necessários (em comparação com a primeira estratégia).

Como benefícios desta estratégia, pode-se citar que evitamos problemas de concorrência e sincronização na fila global (ocorrem menos locks e unlocks necessários). Assim, podemos diminuir o overhead necessário para a comunicação entre as threads, também reduzindo, consequentemente, o tempo total de execução. Mais do que isso, a própria "natureza" do problema já é em camadas, tornando a implementação de uma barreira uma abordagem mais "natural".

Portanto, o grupo decidiu projetar a solução com base na **Estratégia 2**, considerando que seus benefícios superam os oferecidos pela Estratégia 1.

3. Casos de teste de corretude e desempenho

Para os casos de teste de corretude e desempenho, dividiremos os grafos de teste com base nos seguintes critérios:

- Tamanho: os grafos podem ser "pequenos" (menos do que 1000 vértices), "médios" (entre 1000 e 100000 vértices), e "grandes" (mais de 100000 vértices).

- Densidade: os grafos podem ser "densos" ($|E| \cong |V|^2$) ou "esparsos" ($|E| \ll |V|^2$).

O computador que será utilizado para a realização dos casos de teste possui um processador **i5-12500H** da Intel, com **12 núcleos físicos**, e **16 processadores lógicos**, utilizando o sistema operacional **Windows 11**.

Para a realização dos casos de teste de corretude, usaremos uma série de grafos já pré-determinados (cujos valores da distância mínima do BFS já serão conhecidos, tendo sido realizados pelo algoritmo sequencial do BFS). Para cada um destes casos, utilizaremos a solução concorrente do problema 5 vezes (seguindo o padrão estabelecido nos laboratórios desenvolvidos ao longo da disciplina), e checaremos se as distâncias obtidas pelo algoritmo eram, de fato, as esperadas em todos os casos.

Com os testes variados, e se o programa apresentar a saída correta em todos os casos, consideraremos que o algoritmo tem sua corretude garantida.

Caso	Vértices	Arestas	Threads
1	10	15	1, 2, 4
2	100	250	1, 2, 4, 8
3	250	300	1, 2, 4, 8
4	500	500	1, 2, 4, 8
5	1000	2500	1, 2, 4, 8

Para avaliar o desempenho do programa, iremos variar tanto o tamanho do grafo (que implica em uma variação do número de vértices e arestas) quanto o número de threads. Além disso, seguiremos o padrão de determinar o tempo médio de execução com base na média de 5 execuções de cada caso de teste. Inicialmente, imaginamos os seguintes cenários para teste:

Caso	Vértices	Arestas	Threads	Classificação
1	100	250	1, 2, 4	Pequeno Esparso
2	100	4000	1, 2, 4	Pequeno Denso
3	1000	5000	1, 2, 4, 8	Médio Esparso
4	1000	400000	1, 2, 4, 8, 12	Médio Denso
5	100000	500000	1, 2, 4, 8, 12, 16	Médio Esparso
6	100000	5000000	1, 2, 4, 8, 12, 16	Médio Denso
7	1000000	5000000	1, 2, 4, 8, 12, 16	Grande Esparso
8	1000000	10000000	1, 2, 4, 8, 12, 16	Grande Denso

Tabela 2: Casos de teste que serão realizados para medir o desempenho do programa.

Observação: Não estamos usando o número de arestas que teoricamente definem um grafo como denso, pois acreditamos não ter recursos computacionais suficientes para rodar o programa (sequencial ou concorrente) neste caso.

Ademais, o grupo espera que o algoritmo concorrente apresente desempenho variável conforme o tamanho e a densidade do grafo. Em grafos pequenos, o algoritmo sequencial tende a ser mais eficiente, pois o custo de paralelismo (como criação de threads e sincronização) pode superar os ganhos. Já em grafos médios e grandes, especialmente os mais densos, o paralelismo deve reduzir significativamente o tempo de execução, pois conseguimos processar cada camada (com milhares de nós) em uma velocidade muito maior que o sequencial pela divisão da tarefa entre as diferentes threads.

No entanto, é bom salientar que também sabemos que o aumento excessivo de threads pode levar à saturação, onde o desempenho deixa de melhorar ou até piora devido ao overhead de gerenciamento. E, assim, aumentando progressivamente o número de threads, poderemos ter uma evidência empírica dos melhores cenários para uso concorrente do algoritmo na versão que segue a estratégia de sincronização por barreira.

4. Referências bibliográficas

1 - *LSPD Mackenzie – Marathon.* Disponível em: http://lspd.mackenzie.br/marathon/20/problemset.pdf. Acesso em: 10 out. 2025.

2 - LOUREIRO, Antonio Alfredo Ferreira. *Grafos.* Disponível em: https://homepages.dcc.ufmg.br/~loureiro/alg/071/paa_Grafos_1pp.pdf. Acesso em: 10 out. 2025.

3 - FIGUEIREDO, Daniel Ratton. *Grafos – Aula 2*. Disponível em: https://www.cos.ufrj.br/~daniel/grafos-2024/slides/aula_2.pdf. Acesso em: 10 out. 2025. PESC

4 - STANFORD. *Graphs, DFS and BFS*. Disponível em: https://paulomann.github.io/files/lecture_notes/stanford/graphs_dfs_bfs.pdf. Acesso em: 10 out. 2025.

5 - A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson and U. Catalyurek, "A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L," SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, Seattle, WA, USA, 2005, pp. 25-25, doi: 10.1109/SC.2005.4.