Documentação do TP1 de Algoritmos 1

Lucas Resende Pellegrinelli Machado

Setembro 2019

1 Introdução

O trabalho proposto diz respeito a um grupo de alunos que, ao tentar ter mais chances de ganhar em um cassino, tenta se organizar de forma a coordenar ações dentro do cassino sem serem percebidos. Aos alunos são atribuidos "subordinados", que são outro alunos e dependendo do horário, essas relações hierárquicas podem se inverter caso a instrução "SWAP" seja dada. A todo momento, também pode ser necessário saber qual o membro mais novo que é hierarquicamente superior a algum membro (instrução COMMANDER) e também qual seria uma possível ordem de fala tal que pessoas que estão mais altos na hierarquia precisam falar primeiro que os mais baixos (instrução MEETING).

2 Modelagem

O problema pode ser modelado como um grafo direcionado acíclico, sendo cada vértice um membro da equipe e cada aresta uma relação entre dois membros, sendo direcionada na direção indo do membro mais alto na hierarquia para o mais baixo. Dessa forma para executar as ações necessárias para o problema será necessário apenas um DFS (Depth-First Search) para cada uma das instruções. Para o SWAP devemos apenas inverter a aresta e depois tentar encontrar um ciclo executanto um DFS para cada vértice (caso ele não tenha sido visitado ainda) e, caso ao percorrer o grafo com o DFS o algoritmo chegue em um vértice que está na pilha de execução do DFS, temos um ciclo, então a aresta não poderia ser invertida, logo a inversão será disfeita. Já no caso da instrução COMMANDER temos apenas que executar o DFS para o grafo invertido (onde cada vértice se liga aos vértices hierarquicamente superiores a ele ao invés do oposto) e guardar para cada vértice, o menor valor entre a idade dos vértices vizinhos e a chamada da recursão nesses vértices vizinhos, assim percorrendo todos os "comandantes" do membro a ser testado e encontrando o de menor idade. Por último, para o comando MEETING devemos encontrar uma das possíveis ordens topológicas do grafo executando o DFS para cada um dos membros (caso ele não tenha sido visitado ainda) e para cada um deles, o adicionando em um vetor que representa a ordem, resultado no final em um vetor com todos os membros em um ordem topológica.

3 Implementação

Para a implementação do problema, foi escolhido o uso de uma lista de adjacências utilizando um `std::map<int, std::vector<int>>` que mapeia o índice de cada membro com sua lista de membros adjacentes. Um detalhe importante na implementação utilizada foi que foram armazenadas também as arestas invertidas de cada um dos membros. Isso se da pois tanto a instrução COMMANDER se torna mais fácil de implementar caso voce use um grafo com arestas invertidas para ele, logo foi criado uma outra lista de adjacências apenas para guardar o grafo invertido. Para as instruções de SWAP e MEETING apenas o vetor original foram utilizados.

Todo o código foi encapsulado em uma classe chamada `Team` que representa o grupo de pesquisa, tendo armazenado as idades dos membros, a lista de adjacências e a lista de adjacências invertida.

Outro detalhe na implementação foi que a classe `Team` não imprime os resultados na tela, todos os resultados são retornados pelas suas respectivas funções (como no caso da função `meeting()` que retorna um `std::vector<int>` que representa a ordenação topológica).

4 Análise de Complexidade

A complexidade de espaço do problema é simples de ser calculada visto que os únicos espaços de memória utilizados no programa são destinados aos vetores de idade, lista de adjacência e lista de adjacência inversa. O vetor de idade como guarda a idade de cada um dos membros (que são modelados como vértices) tem complexidade de $O(V)$. Já as listas de adjacência também tem um tamanho igual ao número de membros, porém para cada uma de suas posições, temos um outro vetor, e esse representando cada uma das arestas conectadas a ele. Dessa forma é possível inferir que o número total de itens adicionados em cada uma dessas posições soma o número total de arestas, assim temos que a complexidade de espaço dessas listas são $O(V + A)$. Como a complexidade de espaço das listas domina assintóticamente a do vetor de idades, então a complexidade de espaço final do problema é $O(V + A)$.

Para a complexidade de tempo primeiro estudaremos qual a complexidade de um algoritmo de DFS visto que todo o problema é baseado nisso e ter essa informação guardada facilitará a compreensão da complexidade do resto do programa.

O DFS executa apenas cada uma vez para cada vértice, isso pode ser comprovado olhando para a implementação desse algoritmo e notar que sempre existe a cláusula

```
if(visited[v] == false){
    visited[v] = true;
    // Resto do algoritmo
}
```

Logo já sabemos que a complexidade desse algoritmo é algo parecido a $O(V)$. Porém em cada um dos vértices existe um `for` que passa por cada uma das arestas daquele vértice. Isso leva também a conclusão que cada uma das arestas também só é visitada uma vez visto que como é um grafo direcionado, apenas um único vértice tem uma dada aresta na sua lista de adjacências e visto que cada vértice só é visitado uma vez, cada aresta só é visitada uma vez também. Isso nos leva a complexidade final do DFS de $O(V + A)$.

Com isso, agora podemos calcular a complexidade de cada uma das instruções.

A instrução SWAP precisa primeiro remover e adicionar uma aresta no grafo para fazer a inversão dela. Isso é considerado $O(1)$. O outro passo para a instrução é detectar se existe um ciclo por meio do DFS. O DFS só precisa ser executado uma vez, ou seja, temos que a complexidade dessa parte é $O(V + A)$. Por fim, caso um ciclo seja encontrado, a inversão precisa ser desfeita e mais uma remoção de adição de arestas precisa ser feita, sendo isso $O(1)$. Como o DFS domina assintóticamente o processo, a complexidade dessa instrução é $O(V + A)$.

A instrução COMMANDER precisa apenas de um DFS que retorne a menor idade que encontrar no gráfico invertido, logo sua complexidade é $O(V+A)$. Um detalhe importante dessa instrução é que no DFS executado por ela, é possível que nem todos os vértices sejam visitados, mas como potencialmente todos são visitados, a complexidade continua a mesma.

Já a instrução de MEETING é também apenas um DFS que vai adicionando os vértices que encontra em um vetor, logo também é $O(V + A)$.

Logo, a complexidade final do programa, dependendo da quantidade I de instruções é dada por $(O(I(V + A))$ visto que para cada instrução temos $O(V + A)$.

5 Análise Experimental

Foram feitos 2 experimentos e para cada um deles foram obtidos duas informações.

Ambos experimentos se baseavam em executar o programa em diferentes casos de teste sendo que o número de vértices crescia linearmente mas o número de arestas geradas era igual ao número máximo de arestas possíveis para um dado número de vértices ($A = V(V-1)/2$), ou seja, crescia de forma quatrática.

5.1 Número de instruções constante

Para o primeiro teste, o número de vértices começava em 100 no primeiro caso de teste e ia até 1000 em incrementos de 100 em 100. O número de instruções foi mantido constante (3 swap, 3 commander e 1 meeting).

Ao realizar o teste, era esperado que o gráfico do tempo de execução pelo número de vértices fosse uma curva quadrática e o do tempo de execução pelo número de arestas fosse linear. Isso faz sentido visto que como o número de

arestas crescia de forma quadrática e o de vértices de forma linear, ela acaba dominando o número de vértices na complexidade do DFS, isso faz com o que a linearidade da complexidade do DFS seja em relação ao número de arestas e não mais ao número de vértices.

Esses foram os resultados encontrados:

Gráfico de Tempo de Execução x Número de Arestas

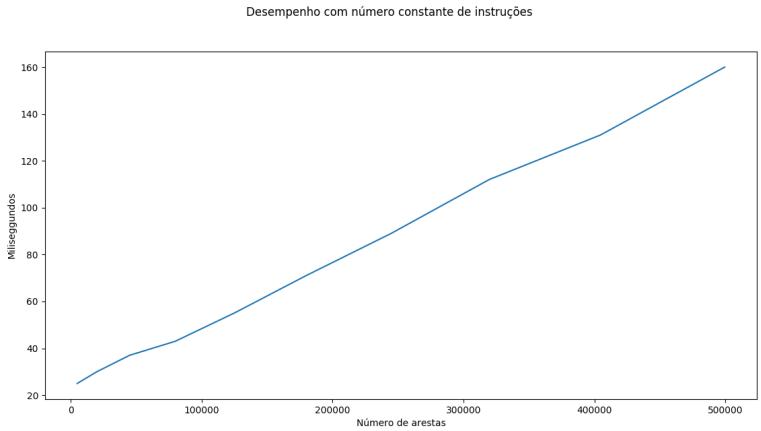
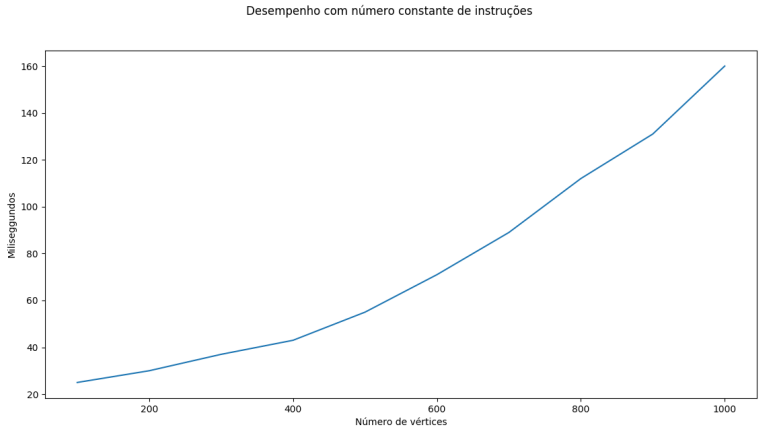


Gráfico de Tempo de Execução x Número de Vértices



Como podemos observar, os resultados foram como esperado.

5.2 Número de instruções variável

Foi também feito o teste com o número de instruções sendo igual a metade do número de vértices (que agora começam em 5 e incrementam de 5 em 5 até 50

para cada caso de teste). O resultado esperado era o mesmo do último teste visto que o número de instruções aumentando linearmente com os vértices não altera o fato do número de arestas crescer quadraticamente com o número de vértices e isso dominar a complexidade do DFS. O que mudou bastante foi o número de vértices que era possível executar em tempo hábil. Para comparação, com o teste de 50 vértices, o tempo gasto foi praticamente igual ao tempo gasto com o de 1000 vértices no teste anterior.

Esses foram os resultados encontrados:

Gráfico de Tempo de Execução x Número de Arestas

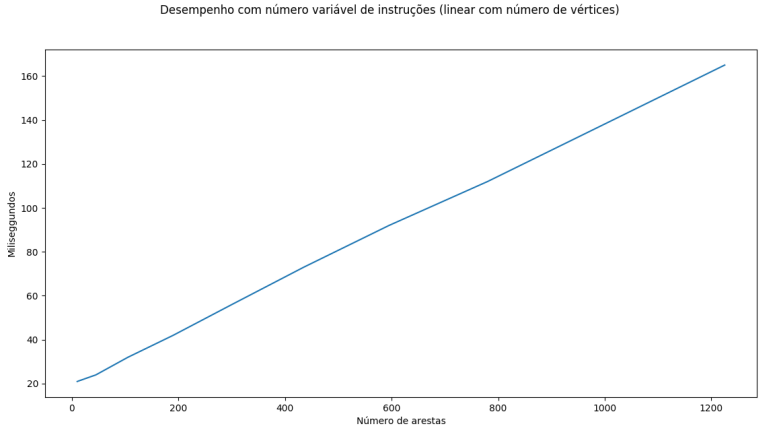
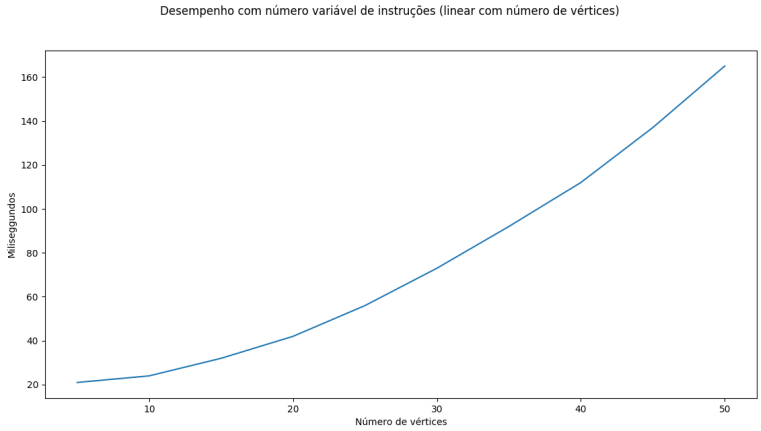


Gráfico de Tempo de Execução x Número de Vértices



Como podemos observar novamente, os resultados foram como esperado.

6 Conclusão

O problema foi modelado e implementado com sucesso e os testes representaram o que era esperado quando analisamos o algoritmo de forma teórica

O trabalho compriu o propósito de nos fazer praticar o uso de grafos para resolver um problema, tanto na parte de modelagem quanto na implementação, que é algo bem diferente do que estamos acostumados a lidar no curso até então no curso.

Acredito que o time de Blackjack da UFMG tem uma preocupação a menos para tentar burlar o sistema dos cassinos de Las Vegas. Quem sabe os programadores que tanto os ajudaram receberão uma parcela do dinheiro ganho por lá, só resta acreditar.

References

[1] Depth first search or dfs for a graph. https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/. Accessed: 2019-09-28.

[2] Detect cycle in a directed graph. https://www.geeksforgeeks.org/detect-cycle-in-a-graph/. Accessed: 2019-09-28.

[3] Graph representation. https://www.hackerearth.com/pt-br/practice/algorithms/graphs/graph-representation/tutorial/. Accessed: 2019-09-28.

[4] Measure execution time of a function in c++. https://www.geeksforgeeks.org/measure-execution-time-function-cpp/. Accessed: 2019-09-28.