Trabalho Prático 2

Essa coloração é gulosa?

Arthur Araújo Rabelo
2022091552

1. Introdução

Neste trabalho, existem dois problemas: dizer se a coloração própria de um grafo é gulosa e ordenar os vértices com dois critérios: a sua cor e seu rótulo. Uma coloração é gulosa se cumpre a seguinte condição: cada vértice de cor c possui pelo menos um vizinho com cada uma das cores menores que c. A ordenação pode ser feita por sete métodos: Bolha, Seleção, Inserção, Quicksort, Mergesort, Heapsort e um algoritmo próprio, todos esses adaptados a estrutura de dados usada para representar o grafo. Também deve ser feita uma comparação experimental entre eles de modo a extrair informações importantes sobre cada um e sobre como afetam o algoritmo em geral. Por fim, será verificado qual o melhor método para resolver o problema da coloroção gulosa.

A solução a ser implementada para dizer se a coloração é gulosa será feita através de loops simples que percorrem os vértices do grafo e seus vizinhos e testam a condição determinante supracitada. A adaptação da ordenação a estrutura de dados foi feita por uma simples sobrecarga dos operadores ‘<’ e ’ >’.

2. Método

2.1 Tipos abstratos de dados e classes

Para este trabalho, foi implementado a classe *graph*, que é um grafo formado por vértices. O *struct* vértice foi então criado com o objetivo de definir o tipo a partir do qual o grafo seria construído. Cada vértice possui inteiros para o seu rótulo, sua cor e seu número de vizinhos. Além disso, pensando na implementação por lista de adjacência vista no curso, cada vértice possui um array dinâmico de inteiros que guarda os vizinhos. Além disso, foi feita uma sobrecarga para os operadores menor e maior para que o tipo do vértice pudesse ser comparado nas funções de ordenação.

A classe *graph* conta com as seguintes funções: o construtor, que inicializa os vértices de acordo com o tamanho n passado como parâmetro e insere os rótulos de 0 até n-1 em cada um deles; InicializaVizinhos, que aloca espaço para os vizinhos de determinado vértice passado como parâmetro; InsereVizinho, que insere os rótulos dos vizinhos; GetVizinhos retorna o array de inteiros dos vizinhos; GetNumVizinhos retorna o número de vizinhos de certo vértice; SetCor atribui determinada cor a um vértice; MenorCor retorna a cor mínima do grafo; GetCor retorna a cor de um determinado vértice; GetTamanho retorna a quantidade de vértices do grafo; por fim, o destrutor libera a memória alocada para os vizinhos e para os vértices.

Para o meu método de ordenação foi utilizada uma lista encadeada adaptada ao problema, de modo que cada nó (class Node) simula um vértice, possuindo inteiros de cor e rótulo e a mesma sobrecarga de operadores para o tipo Vertice. A lista encadeada possui apenas o construtor, o destrutor e a função Insert, a qual adiciona um nó antes do primeiro nó maior do que ele.

2.2 Funções

A principal função do código, além das funções relacionadas a classe graph, é a isGreedy, que recebe um parâmetro do tipo Graph. Ela retorna 1 se a coloração é gulosa e 0 caso contrário. Para realizar essa verificação, a função salva em um arranjo as cores menores que a cor do vértice atual (C) e percorre seus vizinhos, verificando se cada uma de suas diferentes cores menores que C estão no arranjo acima (essa operação é feita com o auxílio da função Find, que retorna 1 se um inteiro está contido em determinado arranjo e 0 caso contrário). Se isso é verdadeiro para todos os vértices, então quer dizer que a coloração é gulosa. Um pequeno elemento que diminui o custo do algoritmo é o teste se a menor cor do grafo é maior que 1. Se isso é verdadeiro, então a coloração certamente não será gulosa. Desse modo, não é necessário fazer todas as operações acima em vão. Ademais, se a cor do vértice atual for a cor mínima,

nenhuma verificação de gulosidade precisa ser feita para ele; esta também é uma melhoria que diminui o custo do algoritmo.

Além das funções já conhecidas de ordenação, o código conta com um algoritmo próprio para ordenar os vértices, o qual é feito baseado na estrutura de dados LinkedList (lista encadeada) adaptada a esse problema. A lógica do algoritmo é basicamente inserir os vértices na lista encadeada fazendo comparações com os elementos que já estão nela: um vértice sempre é inserido antes do primeiro elemento maior do que ele na lista. Dessa forma, o resultado final da lista serão os nós ordenados de acordo com a cor e o rótulo. Assim, esses valores são substituídos no vetor dos vértices, tornando-o ordenado.

3. Análise de Complexidade

3.1 Complexidade de Tempo

```
bool isGreedy(Graph* g){
    int min_color = g->MenorCor();                                          } O(n)
    int *v;
    int c;
    int a = 0;
    if(min_color > 1) {                                                     } O(1)
        return 0;
    }
    for(int i = 0; i < g->GetTamanho(); i++){                               } O(n)
        c = g->GetCor(i);                                                   } O(1)
        if(c == min_color){                                                 } O(1)
            a++;
            continue;
        }
        int colors[c-1];
        for(int k = 1; k < c; k++){                                         } O(c), tal que c <= n
            colors[k-1] = k;
        }
        int helper = 0;
        int *anteriores = (int *) calloc(g->GetNumVizinhos(i), sizeof(int));
        v = g->GetVizinhos(i);                                              } O(1)
        for(int j = 0; j < g->GetNumVizinhos(i); j++){                      } O(v), tal que v <= n-1
            int c_vizinho = g->GetCor(v[j]);                                } O(1)
            if((c_vizinho < c) && (!Find(c_vizinho, anteriores, g->GetNumVizinhos(i))) && Find(c_vizinho, colors, c-1)){ } O(n) + O(1) + O(n)
                anteriores[j] = c_vizinho;
                helper++;
            }
        }
        if(helper == c-1){                                                  } O(1)
            a++;
        }
        free(anteriores);
    }
    if(a == g->GetTamanho()){                                               } O(1)
        return 1;
    }
    return 0;
}
```

A imagem acima demonstra a complexidade de cada parte da função isGreedy. A parte fora do loop externo (função MenorCor) é O(n). Este loop percorre todos os vértices, ou seja, tem complexidade O(n). O resultado final será a soma da parte de fora do loop externo somada a complexidade de dentro do loop multiplicada pela complexidade do mesmo.

Dentro desse loop, temos o primeiro loop interno, que é O(n) no pior caso (c = n). O segundo loop interno seria O(n-1), pois o pior caso é v = n-1, multiplicado por O(n), que é a complexidade dos testes do if $\big(O(n) + O(n) + O(1) = O(n)\big)$. Este O(n) da condição vem da função da Find. Isso resulta em $O(n * (n-1)) = O(n^2)$. A soma da complexidade dos dois loops internos é $O(n^2)$.

Complexidade total da função isGreedy: $O(n) + (O(n) * O(n^2)) = \mathbf{O(n^3)}$.

```
void MySort(Vertice *v, int n){
    LinkedList list;

    for(int i = 0; i < n; i++){
        list.Insert(v[i].rotulo, v[i].cor);
    }

    Node* aux = list.head->next;
    int i = 0;
    while(aux != NULL){
        v[i].cor = aux->cor;
        v[i].rotulo = aux->rotulo;
        aux = aux->next;
        i++;
    }
}
```

} O(n)

} O(n)

} O(n)

A complexidade da função MySort é dada pela soma da complexidade do primeiro e do segundo loop. O primeiro insere os vértices na lista, fazendo as comparações necessárias até achar um vértice menor que o que está sendo inserido. O pior caso dessas comparações é quando o vetor está ordenado, ou seja, serão feitas n-1 comparações para inserir o último vértice. Considerando

este caso, o primeiro loop possui complexidade O(n) * O(n) = O(n²). O segundo faz 3 movimentações e percorre toda a lista, logo possui complexidade O(n). Sendo assim, a complexidade total é dada por: O(n²) + O(n) = **O(n²)**.

Como para este problema não existem elementos iguais (o rótulo é o critério de desempate), a ordenação dos vértices é sempre estável.

O método, entretanto, possui adaptabilidade. Se o vetor estiver inversamente ordenado, a inserção na lista é O(1), pois fará apenas uma comparação para achar um elemento maior do que o que será inserido. Neste caso, a complexidade do primeiro loop se torna O(n) e a complexidade final, portanto, se torna O(n) também.

3.2 Complexidade de Espaço

A alocação de memória extra no código é feita para a inicialização dos vértices do grafo e de seus vizinhos. Sendo ‘n’ o número de vértices e ‘e’ o número de arestas, temos que o espaço alocado é n + 2e, já que cada aresta liga dois vértices e, portanto, cria dois vizinhos. O pior caso é o grafo completo, que possui (n²-n)/2 arestas. Nesse caso, o espaço alocado seria n + n² - n, que é igual a n². Sendo assim, temos que a complexidade da classe graph é O(n²). A função isGreedy aloca um vetor de tamanho igual a quantidade de vizinhos de cada vértice do grafo, liberando-o a cada iteração do loop. Como o número de vizinhos máximo é n-1, o pior caso seria n² - n de espaço utilizado durante todo o algoritmo, o que é O(n²).

Caso haja a utilização do algoritmo próprio de ordenação, mais algum espaço será alocado. A lista encadeada irá alocar n elementos, ou seja, a complexidade será O(n). Comparada à complexidade das partes supracitadas, esse valor não é significativo: a complexidade final de espaço sempre será dominada por **O(n²)**.

4. Estratégias de Robustez

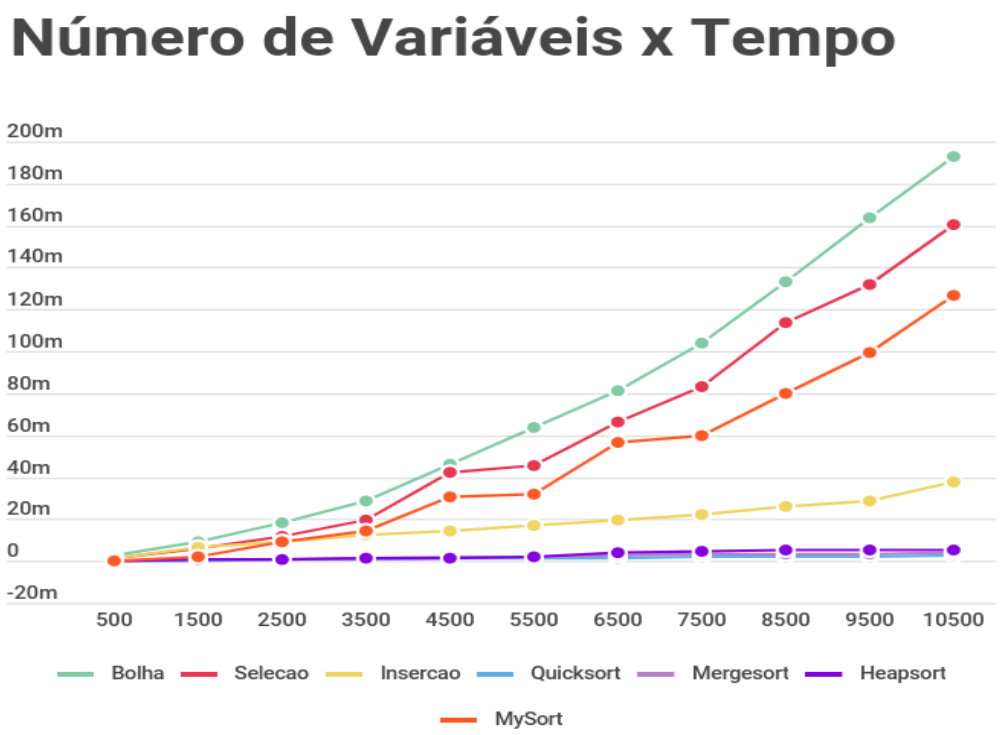
As medidas de programação defensiva foram implementadas apenas para tratar as exceções possíveis nos valores de entrada, ou seja, aquelas que estão sob a responsabilidade do usuário.

Considerando que cada vértice só pode ter n-1 vizinhos (não existe aresta laço neste problema), se o número de vizinhos ultrapassa esse valor ou é menor que 0, uma exceção do tipo NumVizinhosInvalido é lançada. Caso haja uma tentativa de inserir como vizinho um vértice fora do intervalo do número de vértices, a exceção VerticeInvalido é lançada. Por fim, se os vizinhos fornecidos ultrapassarem a quantidade inicialmente dada, a exceção do tipo NumVizinhosInvalido é lançada também.

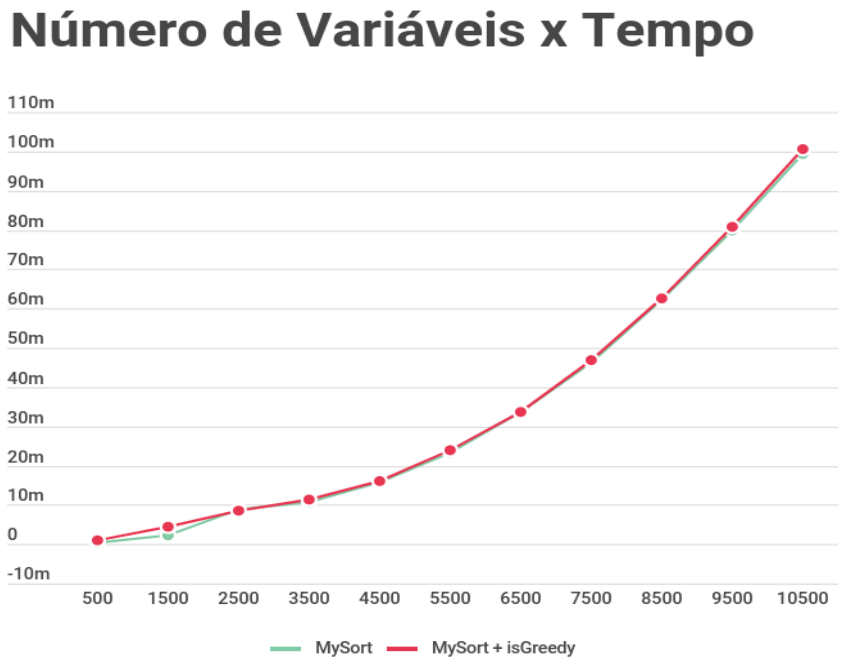
O programa também testa se o comando de ordenação está entre os possíveis (b, s, i, q, m, p, y). Caso contrário, um aviso de erro é lançado.

5. Análise Experimental

Para a análise experimental, primeiro foram medidos os valores de tempo para vetores de 500 até 10500 elementos em todos os métodos de ordenação. Com a imagem abaixo pode verificar que o método de ordenação próprio é de fato $O(n^2)$, visto que é dominado assintoticamente pelo método da bolha, o qual também é $O(n^2)$. Foi possível verificar também que os métodos mais eficazes (em termos de tempo) para o problema são o quicksort, o mergesort e o heapsort.



Também foi feita uma análise para verificar o quanto a função isGreedy impacta no tempo do algoritmo. Para tanto, foram comparados os tempos de relógio do método próprio de ordenação com e sem a função de gulosidade. Os testes feitos foram todos gerados pelo gerador disponibilizado.



Pode-se verificar pelo gráfico que o impacto é mínimo. Isso acontece porque o pior caso da função isGreedy (complexidade $O(n^3)$) raramente ocorre, uma vez que o número de vizinhos em todos os testes gerados dificilmente será um número tão grande quanto a quantidade de vértices. Sendo assim, o segundo loop interno não terá tanto impacto no algoritmo. Além disso, muitas cores são repetidas e, portanto, o teste de cor mínima ajuda bastante na diminuição do tempo gasto na função. Por fim, nos testes gerados também é raro que a quantidade de cores chegue perto do número de vértices, o que contribui para que o tempo no primeiro loop não caia no pior caso (ordem de n).

6. Conclusão

Neste trabalho foi feito um algoritmo que verifica se a coloração de um grafo é gulosa e foram implementados diversos métodos de ordenação adaptados a uma estrutura de dados, contando com um método de autoria própria.

A análise comparativa desses métodos de forma a verificar a melhor opção para solucionar o problema proposto foi útil para a testemunha prática do funcionamento de cada um em termos de complexidade de tempo. Além disso, a análise de tempo e espaço dos outros algoritmos e classes também possibilitou um maior aprofundamento em complexidade assintótica e uso de memória.

7. Bibliografia

Relational Operator Overloading. Disponível em: https://prepinsta.com/c-plus-plus/relational-operator-overloading/

Heap Sort - Data Structures and Algorithms Tutorials. Disponível em: https://www.geeksforgeeks.org/heap-sort/

Merge Sort - Data Structures and Algorithms Tutorials. Disponível em: https://www.geeksforgeeks.org/merge-sort/

STUDY WITH, Saumya. Merge Sort || GeeksforGeeks || Problem of the Day || SORTING. YouTube, 23/12/2021. Disponível em: https://www.youtube.com/watch?v=9Xaz01KOPuo

Program to implement Singly Linked List in C++ using class. Disponível em: https://www.geeksforgeeks.org/program-to-implement-singly-linked-list-in-c-using-class/

Destructor for a linked list. Disponível em: https://codereview.stackexchange.com/questions/20472/destructor-for-a-linked-list

CHAIMOWICZ, Luiz. PRATES, Raquel. Complexidade Assintótica. Apresentação do PowerPoint. Disponível em: https://virtual.ufmg.br/20232/pluginfile.php/283451/mod_resource/content/2/02%20-%20Complexidade%20Assint%C3%B3tica.pdf