Graduação em Ciência da Computação

Trabalho Multidisciplinar (2^o **TP)**

Disciplinas:
Prog-2, AED-1

Professores:
Mário Oliveira, Patrícia Proença, Walace Rodrigues

Instruções Gerais Para o Trabalho:

1. Esta atividade poderá ser resolvida em grupo com no máximo 2 integrantes.
2. Caso você ache que falta algum detalhe nas especificações, você deverá fazer as suposições que julgar necessárias e escrevê-las no seu relatório. Pode acontecer também que a descrição dessa atividade contenha dados e/ou especificações supérfluas para sua solução. Utilize sua capacidade de julgamento para separar o supérfluo do necessário.
3. Como produtos da atividade serão gerados dois artefatos: códigos fontes da implementação e documentação da atividade.
4. Cada arquivo-fonte deve ter um cabeçalho constando as seguintes informações: nome(s) do(s) aluno(s), matrícula(s) e data.
5. O arquivo contendo a documentação da atividade (relatório) deve ser devidamente identificado com o(s) nome(s) e matrícula do(s) autor(es) do trabalho. O arquivo contendo o relatório deve, obrigatoriamente, estar no formado PDF.
6. Devem ser entregues os arquivos contendo os códigos-fontes e o arquivo contendo a documentação da atividade (relatório). Compacte todos os artefatos gerados num único arquivo no formato ZIP.
7. Entregue apenas uma resolução por grupo em dois ambientes diferentes: (1) via portal acadêmico (`https://meu.ifmg.edu.br/`) e (2) via portal run.codes (`https://run.codes/`).
8. O prazo final para entrega desta atividade é até 23:59:00 do dia 31/07/2016.
9. O envio é de total responsabilidade do aluno. Não serão aceitos trabalhos enviados fora do prazo estabelecido.
10. Trabalhos plagiados serão desconsiderados, sendo atribuída nota 0 (zero) a todos os envolvidos.
11. O valor desta atividade é 20 pontos para AED-1 e 30 pontos para Prog-2.
12. As apresentações dos trabalhos acontecerão nos dias 1 e 2 de agosto, conforme cronograma a ser divulgado na véspera.

---

1 Introdução

Este trabalho será dividido em três partes:

1. Manutenção na TAD para grafos que foi desenvolvida no 1^o TP;
2. Utilização da 2^a versão da TAD grafos (desenvolvida na 1^a parte deste trabalho);
3. Desenvolvimento de um estudo comparativo sobre métodos de ordenação, segundo especificação apresentada a seguir.

A terceira parte do trabalho é endereçada apenas para alunos da disciplina de AED1.

2 Primeira Parte

Esta tarefa consiste em realizar uma manutenção no TAD grafo, implementado no primeiro trabalho prático, segundo a especificação que segue:

Modificar a estrutura de dados: A nova versão do TAD deve seguir a estrutura apresentada na figura 1:

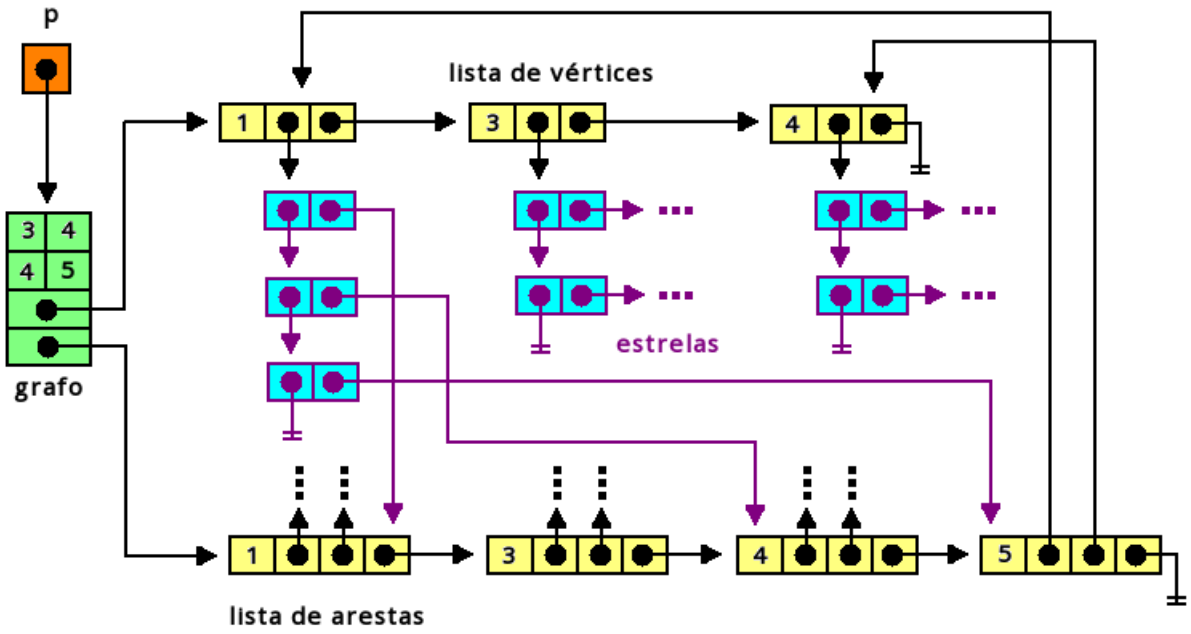

Figura 1: Estrutura de dados a ser utilizada no TAD Grafo.

A figura 1 ilustra a representação parcial do dígrafo **V = {1, 3, 4}** e **A = {(1,1,3), (3,4,3), (4,1,1), (5,1,4)}** sendo cada tripla em **A** definida como **(id-da-aresta, id-do-vértice-alfa, id-do-vértice-omega)**. Na ilustração da figura 1 vemos o vértice 1 cujas arestas incidentes nele são as arestas 1, 4 e 5. Vemos também que a aresta 5 é definida pelos vértices 1 e 4.

O código em C para o tipo Grafo é:

```
struct grafo {
   int numVertices, numArestas;
   int sementeVertices, sementeArestas;
   ListaVertices pv;
   listaArestas pa;
   };
typedef struct grafo *Grafo;
```

Examinando a figura 1, percebemos que:

1. Todas as listas implementadas utilizam encadeamento simples. Todavia, a forma de implementação das listas dinâmicas é livre, segundo critério do grupo. A utilização de listas que utilizam encadeamento duplo é sugerida, mas não é obrigatória.
2. Existem 3 tipos distintos de listas: lista de vértices, lista de arestas, lista de arestas incidentes (as estrelas dos vértices).
3. Cada nodo da lista de vértices armazena: o identificador do vértice, um apontador para a lista das arestas incidentes (a estrela do vértice), e um apontador para o próximo nodo da lista.
4. Cada nodo da lista das arestas incidentes (a estrela do vértice) armazena: um apontador para uma aresta incidente no vértice e um apontador para o próximo nodo da lista.
5. Cada nodo da lista de arestas armazena: o identificador da aresta, um apontador para o vértice alfa da aresta, um apontador para o vértice omega da aresta, e um apontador para o próximo nodo da lista.

Acrescentar ou modificar operações: Para a implementação das rotinas que executam as operações são adotadas as mesmas convenções utilizadas no primeiro trabalho prático. As operações que devem ser acrescentadas ou modificadas seguem listadas abaixo. Todas as outras operações que constavam no primeiro trabalho prático, mas não são apresentadas aqui, devem continuar funcionando exatamente como foram especificadas para o primeiro trabalho prático.

Operação :: GGcriaGrafo	
descrição :: cria um grafo	
entradas :: – não tem	**saídas ::** – *p* : ponteiro para o grafo criado G(V,A)
prerequisitos :: – não tem	**posrequisitos ::** – |V| = 0, |A| = 0 – G tem memória alocada para v vértices e a arestas

Operação :: GVcriaVertice	
descrição :: cria um vértice no grafo	
entradas :: – *p* : ponteiro para um grafo G(V,A)	**saídas ::** – *v* : identificador do vértice criado
prerequisitos :: – |V| < número máximo para um inteiro	**posrequisitos ::** – |V'| = |V| + 1 – v = menor inteiro nunca utilizado como id de vértice em G

Operação :: GAcriaAresta	
descrição :: cria uma aresta no grafo	
entradas :: – *p* : ponteiro para um grafo G(V,A) – *v1* : identificador do vértice de partida – *v2* : identificador do vértice de chegada	**saídas ::** – *a* : identificador da aresta (v1,v2), ou {v1,v2}, criada
prerequisitos :: – |A| < número máximo para um inteiro	**posrequisitos ::** – |A'| = |A| + 1 – a = menor inteiro nunca utilizado como id de aresta em G

Operação :: GVdestroiVertice	
descrição :: destroi um vértice no grafo	
entradas :: – *p* : ponteiro para um grafo G(V,A) – *v* : identificador do vértice	**saídas ::** – *v'* : identificador nulo, zero
prerequisitos :: – não tem	**posrequisitos ::** – se o vértice existir, elimina ele do grafo – A consistência de G é mantida

Operação :: GAdestroiAresta	
descrição :: destroi uma aresta no grafo	
entradas :: – *p* : ponteiro para um grafo G(V,A) – *a* : identificador da aresta	**saídas ::** – *a'* : identificador nulo, zero
prerequisitos :: – não tem	**posrequisitos ::** – se a aresta existir, elimina ela do grafo – A consistência de G é mantida

Modifique as funções **GGsalvaGrafo()** e **GGcarregaGrafo()** para utilizarem o seguinte formato de arquivo texto: a primeira linha do arquivo traz o número atual de vértices e o número atual de arestas no grafo; a segunda linha do arquivo traz o valor da semente dos vértices e o valor da semente das arestas no grafo; nas próximas seguem os identificadores dos vértices, um por linha; as linhas seguintes trazem uma aresta por linha, indicando o identificador da aresta, vértice de partida e o vértice de chegada. Por exemplo, o grafo representado na figura 1 seria armazenado num arquivo texto com o seguinte conteúdo:

```
3 4
4 5
1
3
4
1 1 3
3 4 3
4 1 1
5 1 4
```

3 Segunda Parte

Esta tarefa consiste em implementar algoritmos para realizar buscas em grafos e colorir grafos, utilizando o TAD desenvolvido na primeira parte deste trabalho. O termo "busca" é utilizado neste contexto como conceito, porque embora nenhum valor específico esteja sendo efetivamente procurado no grafo, a ordem em que os elementos do grafo (vértices e arestas) são examinados estabelece um padrão de varredura, e uma vez estabelecido esse padrão será fácil realizar quaisquer buscas no grafo.

As instruções para implementação dos algoritmos são apresentadas a seguir:

- **Busca em Largura** — a escolha dos vértices em cada iteração é controlada por uma fila.

  Considere a existência de 3 classes de vértices: brancos, cinzas, pretos. Os vértices brancos são aqueles ainda não visitados. Os vértices cinzas são aqueles que já foram visitados mas ainda não foram esgotados. Os vértices pretos são aqueles que já foram esgotados. Um vértice é considerado esgotado quanto todas as arestas na sua estrela já foram visitadas. Considere também a existência de 2 classes de arestas: visitadas e não visitadas. Segue o algoritmo em alto nível:

```
void buscaLargura(Grafo g, int partida) {
    fila = criaFila();
    "torne todos os vértices brancos";
    "torne todas as arestas não visitadas";
    v = partida;
    "torne v cinza";
    insere(fila, v);

    while ( "existir vértice cinza" ) {
       v = retira(fila);
       "ative v";
       while ( "existe aresta não visitada em estrela(v)" ) {
          a = "escolha aresta em estrela(v)";
          "torne a visitada";
          w = vizinho(g,a,v);
          if ( "w é branco" ) {
             "torne w cinza";
             insere(fila, w);
          }
       }
       "torne v preto"
       "desative v";
    }
} // fim da busca em largura
```

  Seu algoritmo deve imprimir quando um vértice muda de cor e quando é ativado. O algoritmo também deve imprimir quando uma aresta é visitada.

- **Busca em Profundidade** — a escolha dos vértices em cada iteração é controlada por uma pilha. É praticamente o mesmo algoritmo anterior, apenas substitua a fila por uma pilha.

- **Coloração com k cores** — o objetivo é colorir cada vértice no grafo de tal modo que (i) sejam utilizadas no máximo k cores distintas e (ii) nenhum par de vértices vizinhos sejam coloridos com a mesma cor. Encontrar a solução ótima (menor k possível) é um problema difícil, mas encontrar uma solução razoável é simples utilizando uma estratégia gulosa.

  Considere a existência de 3 classes de vértices: os já coloridos e os ainda não coloridos. Cada cor será representada por um número inteiro $n \leq k$. Segue o algoritmo em alto nível:

```
void coloracaoGulosa(Grafo g, int k) {
    pilha = criaPilha();
    "torne todos os vértices não coloridos";

    // fase 1 : define ordem de coloração
    while ( "existe vértice em g" ) {
       w = "escolhe o vértice com menor grau em g";
       destroiVertice(g, w);
       insere(pilha, w);
    }

    // fase 2 : colore os vértices
    while ( !vazia(pilha) ) {
       v = retira(pilha);
       if ( grau(v) >= k ) {
          "não tem solução"
          return;
       } else {
          "colore v com a menor cor possível";
       }
    }

    // imprime solução
    "imprime os pares associados (vértice, cor)";
} // fim da coloração
```

  A estratégia desse algoritmo é, repetidamente, remover cada vértice do grafo e inseri-lo numa pilha. Usa-se dois mecanismos distintos para selecionar o vértice a ser removido: (i) selecionar vértice com grau menor que k, esse vértice não é problemático; (ii) a remoção de um vértice vai diminuir o grau dos seus vizinhos, e isso pode torná-los não problemáticos. Se ao final todos os vértices se tornarem não problemáticos, então o grafo pode ser colorido com k cores e problema tem solução.

4 Terceira Parte

Esta tarefa consiste em implementar e avaliar o desempenho de alguns métodos de ordenação. Siga os requisitos:

(a) Implemente uma rotina de ordenação utilizando o método da bolha tradicional;

(b) Implemente uma rotina de ordenação utilizando o método da inserção;

(c) Implemente uma rotina de ordenação utilizando o método quicksort com seleção de pivô por mediana de 3 sorteios;

(d) Implemente uma rotina de ordenação utilizando o método "quicksort turbinado" que consiste em utilizar o método de inserção quando o tamanho da partição é menor que 40;

(e) Implemente uma rotina para gerar vetores de tamanho n contendo valores gerados aleatoriamente;

(f) Na implementação de todas as rotinas de ordenação, utilize funções para (i) trocar valores dentro do vetor, (ii) comparar valores dentro do vetor. A razão para tal é que a quantidade de comparações e de trocas efetuadas durante a ordenação deverão ser contabilizadas, porque serão utilizadas como métricas na comparação dos métodos de ordenação. Serão utilizadas as métricas: número de comparações, números de trocas, tempo de execução.

(g) Para comparar os métodos, gere vetores de tamanhos 500, 1000, 10000, 100 mil e 300 mil. Execute cada método de ordenação 3 vezes para cada tamanho do vetor, calcule a média e o desvio padrão para cada métrica. Apresente os resultados em forma de tabela.

5 Critérios de Correção

Serão adotados os seguintes critérios de correção para o trabalho:

1. **correção:** somente serão corrigidos códigos portáveis e sem de erros de compilação;
2. **precisão:** execução correta numa bateria de testes práticos; (70%)
3. **modularização:** uso adequado da TAD e das estruturas de dados; (30%)
4. **qualidade do código fonte:** legibilidade, indentação, uso adequado de comentários; (requisito)
5. **documentação:** relatório, conforme instruções apresentadas a seguir. (requisito)

O critério "correção" é obrigatório e condiciona a avaliação toda, o seu código deve compilar senão seu trabalho não será aceito e vai receber nota zero. Alguns critérios não receberão pontuação específica, mas funcionarão como "requisitos" que poderão afetar a pontuação se não forem atendidos adequadamente, portanto leve-os em conta durante o desenvolvimento do trabalho.

A primeira parte do trabalho será corrigida automaticamente pelo sistema `run.codes`, seguindo uma sequência pré-estabelecida (e não divulgada) de testes práticos. Para o portal run.codes, envie um arquivo no formato zip contendo o arquivo makefile e os demais necessários para a primeira parte do trabalho, com exceção do arquivo main.c que será disponibilizado pelo professor e já está armazenado no portal.

Para o portal acadêmico, envie todos os artefatos gerados para resolução do trabalho.

Haverá uma apresentação oral e individual do trabalho. Todos os integrantes do grupo devem participação dessa apresentação.

Na ausência de plágio, as notas dos trabalhos corretos serão computadas individualmente da seguinte forma: nota = nota_apresentacao $*$ nota_trabalho, ou seja, a nota final é ponderada pela nota da apresentação.

5.1 Documentação

Esta seção descreve o formato e o conteúdo do relatório que deve ser gerado como produto final do trabalho. Seja sucinto: em geral é possível escrever um bom relatório entre 2 e 4 páginas.

O relatório deve documentar apenas a terceita parte do trabalho e conter as seguintes informações:

1. **introdução:** Descrever o problema resolvido e apresentar uma visão geral da terceira parte do trabalho implementado;
2. **comparação:** Apresentar tabelas com os valores colhidos da execução de cada método de ordenação para as métricas exigidas: número de comparações, número de trocas e tempo de execução. Em cada caso, acrescer as respectivas médias e desvios padrão das amostragens.
3. **avaliação:** Dê a interpretação do grupo para os resultados obtidos (faça uso de tabelas ou gráficos para facilitar suas explicações). Compare os resultados obtidos experimentalmente com os resultados teóricos (complexidade). Os resultados obtidos experientalmente estão em conformidade com os resultados teóricos?
4. **conclusão:** Apresente as considerações do grupo indicando em quais situações cada um dos algoritmos comparados devem/podem ser utilizados. Avalie o trabalho considerando a experiência adquirida e a contribuição para o aprendizado da disciplina.

Referências

[1] ZIVIANI, N. *Projeto de algoritmos: com implementações em pascal e c*, 2a ed. rev. e ampl. São Paulo: Pioneira Thomson Learning, 2005. 552 p.