Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Disciplina Algoritmos e Estruturas de Dados III - 2018 / 2

Trabalho Prático 1: Resgate das memórias do Auto-Cone

Aluna: Ana Luiza de Avelar Cabral
Matrícula: 2013007080

O Trabalho

Este trabalho aborda o tema de gerar a ordem de classificação de pilotos em eventos de corrida de automobilismo, as corridas do tipo Auto-Cone. As corridas foram realizadas porém não há registros históricos sobre elas, e um historiador gostaria de resgatar e documentar os eventos. Sendo assim, consultou os pilotos que participaram das corridas sobre a classificação final das corridas que eles participaram.

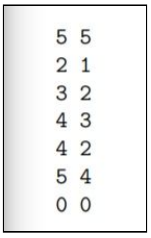
Tem-se todos os pilotos que participaram de cada corrida certo, sem dúvida. O que está em dúvida é a ordem de classificação deles ao final de cada corrida. Sendo assim, o historiador entrevista cada um dos pilotos a respeito do que lembram da ordem de classificação, por exemplo de quem ficaram na frente e quem ficou atrás deles em cada corrida.

Para cada corrida foi gerado um arquivo contendo as informações / memórias de todos os pilotos participantes.

O programa a ser construído por esse trabalho deve construir a ordem de classificação final da corrida com base nas informações das memórias, ou informar que não é possível gerar uma classificação porque as informações se contradizem. Quando as informações não se contradisserem porém for possível mais de uma ordem de classificação, foi instruído para que a ordem crescente do *id* do piloto seja preservada.

Entrada dos Dados

Em cada arquivo, haverá na primeira linha o número de pilotos participantes da corrida seguido pelo número de memórias resgatadas. Então, nas linhas seguintes, estão cada uma dessas memórias no formato M N, que significa uma memória de que o piloto de *id* M estava atrás do piloto de *id* N. Um exemplo de arquivo de entrada está na Figura 1 abaixo.



Arquivo de Memórias da Corrida:

Resultado / Saída dos Dados

O programa deve gerar uma saída no formato

X A,B,C,D,E

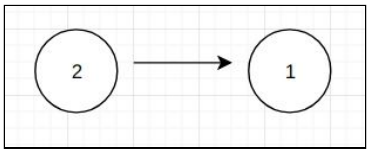
Onde A,B,C,D,E é a ordem de classificação dos pilotos na corrida, sendo A o primeiro classificado e E o último a chegar. No lugar das letras estarão os números *id* de cada piloto.

X pode ser 0, 1 ou 2 no resultado. Quando X for zero significa que não foi possível gerar uma ordem de classificação a partir das memórias dos pilotos. Quando X for 1 significa que foi possível gerar um ordem de classificação e essa ordem é única. Quando X for 2 significa que mais de uma ordem de classificação pode ser gerada com as memórias (então a ordem que segue é apenas uma entre as possíveis, a que foi gerada seguindo os critérios de desempate informados na especificação.

Solução do Problema

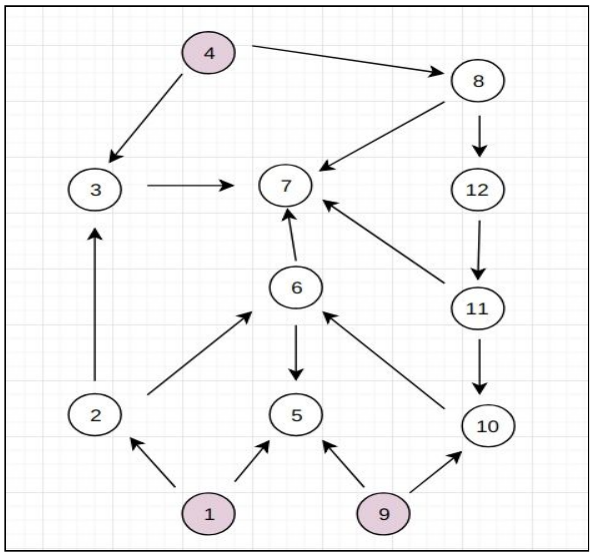
Solução

Para resolver o problema foi decidido modelar o problema como um grafo, onde os pilotos são os vértices do grafo e as memórias são arestas direcionadas: arestas com o elemento que ficou na frente recebendo a aresta e o com elemento anterior à ele de onde sai a aresta.

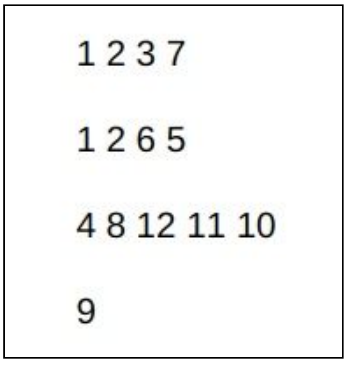


Piloto 2 ficou atrás do Piloto 1 na classificação
Memória: 2 1

Com o grafo pronto, é feita uma Busca em Profundidade (Algoritmo DFS) nele, começando sempre pelos vértices nos quais não entra nenhuma aresta (últimos classificados na corrida). É seguida a ordem das arestas do grafo direcionado ao fazer a busca em profundidade. Assim, cada ramo de uma árvore gerado no DFS é uma ordem de classificação possível. Todos os ramos possíveis são gerados.



Caso de teste do moodle toy 6



4 Ramos gerados pelo DFS sobre ele

Então é feito o desempate por *id* do piloto entre todos os ramos obtidos.
No exemplo acima:

- Entre todos os pilotos, 9 é o de maior id.
  Insere 9 na classificação e o retira do seu ramo ---------- **9**
- Agora, entre todos os ramos, entre 4, 1 e 1, o número 4 é o maior id.
  Insere 4 na classificação e o retira do seu ramo ---------- **9 4**
- Agora, entre todos os ramos, entre 8, 1 e 1, o número 8 é o maior id.
  Insere 8 na classificação e o retira do seu ramo ---------- **9 4 8**
- Agora, entre todos os ramos, entre 12, 1 e 1, o número 12 é o maior id.
  Insere 12 na classificação e o retira do seu ramo ---------- **9 4 8 12**
- Agora, entre todos os ramos, entre 11, 1 e 1, o número 11 é o maior id.
  Insere 11 na classificação e o retira do seu ramo ---------- **9 4 8 12 11**
- Agora, entre todos os ramos, entre 10, 1 e 1, o número 10 é o maior id.
  Insere 10 na classificação e o retira do seu ramo ---------- **9 4 8 12 11 10**
- Agora, entre todos os ramos, entre 1 e 1, o número 1 é o maior id.
  Insere 1 na classificação e o retira do seu ramo ---------- **9 4 8 12 11 10 1**
  Observação: Ao retirar 1 do ramo, retira 1 do ramo em questão e de todos os ramos em caso de “empate” (dois 1s). Dessa forma, retira 1 dos dois ramos em que foi comparado.
- Agora, entre todos os ramos, entre 2 e 2, o número 2 é o maior id.
  Insere 2 na classificação e o retira do seu ramo ---------- **9 4 8 12 11 10 1 2**
  Observação: Como foi empate (2 comparado com 2), retira o 2 dos dois.
- Agora, entre todos os ramos, entre 6 e 3, o número 6 é o maior id.
  Insere 6 na classificação e o retira do seu ramo ---------- **9 4 8 12 11 10 1 2 6**
- Agora, entre todos os ramos, entre 5 e 3, o número 5 é o maior id.
  Insere 5 na classificação e o retira do seu ramo ---------- **9 4 8 12 11 10 1 2 6 5**
- Agora, no ramo restante, 3 é o id próximo.
  Insere 3 na classificação e o retira do seu ramo ---------- **9 4 8 12 11 10 1 2 6 5 3**
- Agora, no ramo restante, 7 é o id próximo.
  Insere 7 na classificação e o retira do seu ramo ---------- **9 4 8 12 11 10 1 2 6 5 3 7**

  Assim, foi gerada a classificação: 7,3,5,6,2,1,10,11,12,8,4,9.

  <u>Agora para saber X:</u>

- X = 0

  Quando houver ciclos no grafo. Isto porque para que a busca represente uma classificação entre os nós eles devem estar em hierarquia na frente - atrás. Se um piloto (vértice) está na frente e atrás de outro piloto ao mesmo tempo (um ciclo), isto não é possível.

  A identificação de ciclos no grafo é feita com o mesmo DFS. Sempre que for encontrada uma aresta de retorno na execução do DFS há ciclo no grafo, logo ele é impossível. Aresta de retorno é quando, durante o DFS, ao buscar na lista de adjacências de um vértice seus vizinhos encontramos um vizinho cinza (Isto é: um vizinho ainda não fechado e também não é novo, um vizinho ancestral desse).

- X = 1

  X será um quando primeiro não houver ciclos no grafo (isto é, há ordem de classificação possível) e quando houver um caminho no grafo pelas arestas que passe por todos os vértices.

  Se naturalmente existe um caminho que passa por todos os vértices, há ordem única e as outras arestas fora desse caminho complementam essa ordem. A não ser que as

arestas contradigam essa ordem: isso vai acontecer quando houver aresta que faça o caminho inverso. Mas quando houver aresta que faça o caminho inverso haverá ciclos, logo será o caso de X=0.

- X = 2

  X será 2 sempre que houver caminhos possíveis (X não for zero) e sempre que não houver um caminho único (Caso X = 1).

Implementação

Modularização

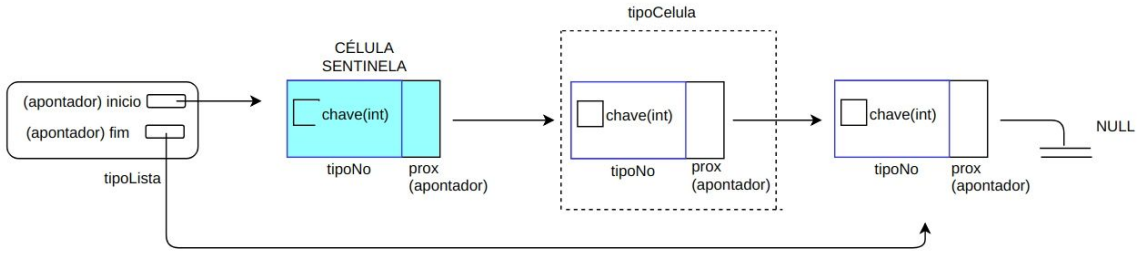
O trabalho está no arquivo *tp1.c* e usa as duas bibliotecas criadas com funções úteis para o trabalho: bibliotecas *grafos.h* e *listaEncadeada.h*. Para a execução do programa é fornecido um *Makefile*, que permite apenas digitar o comando *make* na pasta e a compilação do trabalho ser feita gerando o executável **tp1**. Para a execução então é feito

./**tp1** input.txt output.txt

onde *input.txt* tem os dados sobre os pilotos e as corridas, e *output.txt* será criado para armazenar o resultado gerado pela execução do programa.

Biblioteca listaEncadeada.h

Biblioteca que cria uma estrutura de dados de lista encadeada simples, que é alocada dinamicamente quando criada:



As estruturas de dados criadas na biblioteca estão exibidas acima:

- tipoLista
- tipoCelula (contém um nó do tipoNo e prox do tipo apontador)
- tipoNo (Item da lista, tem uma chave (int) que é o que armazena)
- apontador (endereço de uma tipoCelula)

As listas encadeadas sempre tem uma célula sentinela para melhorar o custo das operações.

As operações da biblioteca são:

- **ImprimeNo** (Para testes - imprime a chave no nó)
- **InicializaNoSentinela** (atribui valor inválido para a chave do nó sentinela)
- **CriaListaVazia** (Dada uma variável do tipoLista, inicializa a lista criando uma sentinela e aguardando a próxima inserção)
- **EstaVazia** (Checa se a lista passada como argumento está vazia ou tem elementos.)
- **Insere** (Insere um item na lista. **A inserção é feita sempre ordenada pela chave**, chave menor na primeira posição, maior na última, e intermediárias no lugar delas.)
- **ImprimeLista** (Para testes - Imprime os valores na lista)

Biblioteca grafos.h

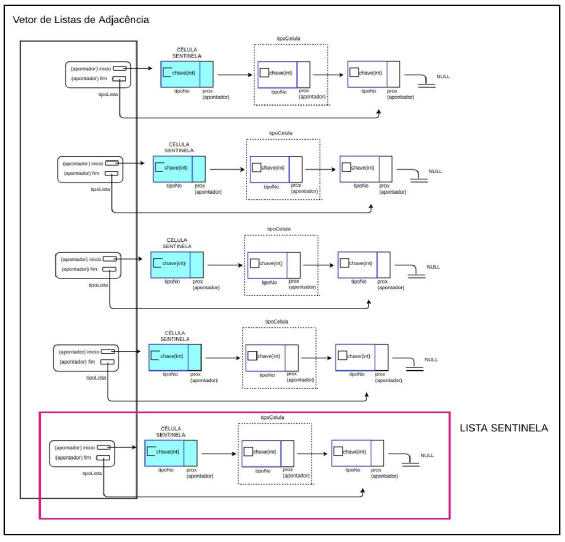
Biblioteca que para lidar com grafos e operações sobre ele.
Ela usa a biblioteca *listaEncadeada.h* para implementar suas operações.

Estruturas de dados:

O grafo, estrutura de dados dessa biblioteca, é implementado como listas de adjacências. Isto é: o **tipoGrafo** é um vetor de listas (tipoLista). Cada lista representa um vértice do Grafo. Em cada lista de um dado vértice estão os vértices vizinhos à ele segundo as arestas orientadas, isto é, os vértices para os quais ele aponta.

Para facilitar o trabalho, em que não existe vértice zero entre os pilotos, **foi colocado o vértice zero como sentinela: o elemento zero do vetor de listas de adjacência é uma lista sentinela, não representa nenhum vértice.** Essa adaptação fez diminuir o custo das operações e torná-las mais intuitivas.

As **constantes** BRANCO, PRETO e CINZA são usadas na busca em profundidade. A constante INICIALIZACAO é usada para inicializar variáveis no programa. A constante DIFERENTE é usada para detecção de ciclo no DFS: o DFS retorna esse valor ao encontrar uma aresta de retorno e detectar um ciclo, o que causa a saída do programa X = 0.



Operações:

- **InicializaGrafoSemArestas** (Aloca o numero de vertices para aquele Grafo e faz listas vazias neles).
- **InsereAresta** (Insere uma aresta no grafo, logo a insere na lista de adjacência do vértice um vértice adjacente à ele para onde ele aponta (grafo direcionado). A inserção é ordenada: as arestas estão inseridas em ordem crescente - para o DFS reduzir o custo na hora de buscar).
- **VisitaDFS** (Função que executa o DFS: visita da Busca em profundidade em um nó, adaptada ao trabalho para atender o que precisa ser gerado e detectado no trabalho).

Execução e pseudo-código

No arquivo *tp1.c* está o main e a execução do programa.

Execução

- Declara variáveis que serão usadas no main e as inicializa.
- Abre os arquivos passados na linha de comando (./tp1 input.txt ouput.txt) para leitura e escrita.

- Lê o número de vértices e arestas do grafo e com base nisso inicializa estruturas do programa.
- Lê as arestas do grafo no arquivo de entrada e a cada aresta lida:
    - Insere ela no grafo (lista de adjacências)
    - Atualiza o número de setas entrando nesse vértice (*vetor setasEntrando*)
- Passa em todos os vértices buscando quem deles não tem arestas apontando para ele e serve para ser raiz do DFS adaptado. *(Usa o vetor setasEntrando)*.
    - Ao encontrar vertices assim (Exemplo: vertices 1, 4 e 9 da Figura 1 aqui acima), os insere na lista de verticesRaiz.
- Inicializa estruturas para fazer a busca DFS no grafo.
- Checa se a lista de verticesRaiz esta vazia
    - Caso esteja, retorna X = 0 porque não é possível criar classificação e encerra.
      (Grafo tem ciclos porque <u>todo</u> vértice entra uma aresta)
    - Caso tenha vertices raiz, pega o primeiro vértice raiz a ser visitado
- Percorre a lista de vertices Raiz toda para executar o DFS adaptado[1] a partir destes. Em cada vértice raiz:
    - Executa VisitaDFS() no vertice.
    - Checa o valor retornado por VisitaDFS
        - Se for a constante DIFERENTE é porque achou ciclo no grafo e já imprimiu X=0. Então encerra o programa porque já tem o resultado.
        - Caso não seja a constante DIFERENTE, pega na lista de vertices raiz o próximo vértice a ser visitado.
- Checa se foi encontrado pelo menos um caminho que passa por todos os vértices do grafo. Caso sim, atribui o valor 1 a X porque existe ordem de classificação única no Grafo *(ver aqui em cima seção como calcular X)*. *[Usa a estrutura verticesAPartirDaqui]*.
- Inicializa vetor com o resultado do trabalho.
  (Obs.: Inicializa aqui para melhorar o tempo de execução, para casos de interrupção ele não precisar ser gerado.)
- Gera a classificação da corrida: Enquanto não tiver executado numVertices vezes, faz:
  (Isto é: enquanto todos os vértices não tiverem sido incluídos no resultado)
    - Visita a primeira posição em cada ramo gerado pelo DFS para determinar o maior valor entre estes vértices primeira posição de cada ramo.
    - Insere esse vértice na classificação, na última posição da classificação. *(vetorResultado e variável verticesInseridos)*.
    - Retira esse vértice do espaço de busca.
    - Procura entre as primeiras posições de cada vértice se ele tem vértice igual à ele. Caso tenha, retira essa duplicata do ramo que a tiver e do espaço de soluções.
    - Reinicia o loop para achar o próximo vértice classificado.
- Imprime os resultados calculados pelo programa: X já calculado seguido da ordem de classificação. *(Armazenada no vetor resultado)*.

---

[1] Uma busca DFS normal escolhe como raízes para começar a busca os vértices brancos não visitados <u>em ordem crescente de id</u>. O DFS adaptado para o programa aqui escolhe como raízes para a busca os vértices que não entra nenhuma aresta neles (vértices-piloto última posição da classificação da corrida).

Detalhamento da Execução do VisitaDFS

Ao visitar um dado vértice com a função VisitaDFS:

- Cria e Inicializa estruturas daquela visita
- Insere o vértice no ramo de busca do DFS *(vetor ramoBusca e tamanhoRamoBusca)*
- Tarefas de DFS normal:
    - Torna o vértice cinza
    - Soma 1 ao tempo percorrido até agora (entrou em um vértice, andou 1 no tempo)
    - Atribui tempo de descoberta atual ao tempo de descoberta do vértice
- Confere se o vértice tem vizinhos. Caso sim:
    - Escolhe como primeiro adjacente o vértice da lista de adjacentes dele com menor id.
      (Obs.: Já inseriu ordenado, está ordenado.)
    - Enquanto não tiver percorrido toda a lista de vértices adjacentes à ele, faz no adjacente / vizinho escolhido:
        - Checa se esse vizinho é branco.

          Se sim, é branco
            - Coloca como antecessor do vizinho o vértice atual.
            - VisitaDFS este adjacente.
            - Checa o valor retornado pela Visita:
                - Se for a constante DIFERENTE, é porque encontrou aresta de retorno em alguma busca abaixo dele, o que indica ciclo. Então já imprimiu o resultado X = 0 e está retornando a função: retornar.
                - Se não for a constante DIFERENTE, o valor retornado por VisitaDFS é o número máximo de vértices alcançáveis em um caminho a partir deste vértice. Esse valor é usado para checar se X = 1 ao final, classificação única possível.

                  Assim, checa se (esse valor + 1) é maior do que o atual número de vértices alcançáveis pelo vértice em questão. Caso seja, então atribui ao vértice em questão esse valor pois é possível alcançar mais vértices.

          Se não é branco:
            - Checa se o vizinho é cinza.

              Se o vizinho é cinza:
                - imprime X = 0 como resultado (achou aresta de retorno logo ciclo no grafo, o que o torna impossível de ser resolvido).
                - encerra a visita ao vértice, retornando a constante DIFERENTE

              Se o vizinho não é cinza:
                - Checa se o vizinho é preto. Se sim:
                    - Checa se (vértices alcançáveis a partir dele + 1) é maior do que o atual número de vértices alcançáveis a partir do vértice atual. Se for maior, atribui esse valor maior ao número de vértices alcançáveis a partir do vértice atual.
        - Seleciona o próximo vértice adjacente da lista de vértices adjacentes e repete o loop.

- Mais tarefas de DFS normal:
    - Torna o vértice preto (indica que já foi visitado e encerrado)
    - Soma 1 ao tempo (Executou verificação de vizinhos no mínimo)
    - Coloca tempo de saída do vértice em sua estrutura para armazenar este valor
- Checa se esse vértice é o último vértice de um ramo de busca. (Isto é: se ele é uma folha ou se ele é um vértice que só possui vizinhos pretos ou cinzas). Se sim:
    - Insere o ramo encontrado no vetor de ramos
    - Caso esse nó seja um nó folha, atribui que o número de vértices alcançáveis a partir dele é 1.
- Remove o vértice do ramo de busca (já foi visitado e colocado preto, agora volta recursiva)
- Retorna como resultado da visita o inteiro que diz quantos vértices no máximo é possível serem alcançados em algum caminho a partir deste vértice, contando com ele.
  (Exemplo: nó folha retorna 1. Um nó que só tem uma folha depois: retorna 2).

Estruturas de dados acima, detalhadas:

As estruturas citadas no pseudo-código acima estão detalhadas aqui.

Elas são necessárias para a execução no programa principal e como parâmetros para executar as funções das bibliotecas.

Assim, temos:

Para as funções do trabalho:

- **vetor setasEntrando[vertice i]**:
    - Armazena, para cada vértice, quantas setas arestas apontam para ele.
    - Vetor de inteiros: cada inteiro do vetor representa um vértice do grafo. (Logo índice zero é sentinela).
- **Lista verticesRaiz**:
    - Lista que armazena os vértices que serão raizes do DFS adaptado: vértices de onde começará a busca DFS no grafo.
    - Lista encadeada do tipoLista, lista encadeada que tem sentinela na primeira posição. (Todas as operações das bibliotecas levam a sentinela em consideração nas operações).

Para o cálculo de X especificamente:

- **verticesAPartirDaqui[vertice i]**
    - Armazena, para cada vértice, quantos vértices são possíveis de se chegar a partir dele em um caminho no grafo no máximo, e contando com ele mesmo.
      Exemplo: Nó folha: 1 (só ele). Nó antes da folha: 2. (ele + folha).
    - Vetor de inteiros: cada inteiro do vetor representa um vértice do grafo. (Logo índice zero é sentinela).

Para a classificação final da corrida:

- **vetorResultado**: vetor de inteiros que armazena os vértices na ordem de classificação na corrida, resultado do programa.
  A primeira posição tem o último colocado na corrida e assim sucessivamente, até que no fim do vetor está o piloto-vértice que ganhou a corrida.
- **verticesInseridos**: auxiliar para preenchimento da próxima posição do vértice, armazena quantos vértices já foram inseridos para saber em que posição inserir no vetor o próximo resultado encontrado.
- **verticeMax**: auxiliar que armazena o maior vértice em uma posição inicial entre todos os ramos de busca gerados.

- **ramoVerticeMax**: armazena em qual dos ramos gerado ramo foi encontrado o verticeMax acima.
- **vetorRamos**: vetor de vetores de inteiro. Cada vetor de inteiro nele é um ramo encontrado no VisitaDFS. Ou seja: cada item [i] dele é um dos vetores encontrados na busca DFS pelo grafo todo.
- **numRamos**: armazena quantos ramos há no vetor **vetorRamos**.
- **tamanhoDesseRamo[ramo]**: armazena o tamanho de um ramo de busca encontrado no DFS. É um vetor de inteiros onde cada posição armazena o tamanho do respectivo [ramo de busca encontrado].
- **inicioRamo**: armazena a partir de onde do ramo começar a busca pelo próximo vértice resultado. É a forma de retirar o vértice do ramo com menor custo computacional. É um vetor de inteiros.

Para executar o DFS:
- **tempo**: inteiro, variável para armazenar instantes de entrada e saída no DFS nos vértices
- Os vetores **com o ínidice 0 sentinela** e os outros índices associados à cada vértice do grafo:
    - **cor [vertice i]**: cor do vértice no DFS (Branco não visitado, Cinza visitado e não finalizado e Preto os vértices visitados e finalizados).
    - **antecessor[vertice i]**: antecessor
    - **d[vertice i]**: tempo que o vertice foi descoberto na busca em profundidade
    - **f[vertice i]**: tempo que o vertice ficou preto na busca
- **ramoBusca**: vetor que armazena cada ramo gerado pelo DFS até chegar à uma folha no Grafo ou até chegar à um vértice que não tenha vizinhos possíveis de serem visitados.
  Vetor de inteiros: cada inteiro nele é um vértice que já foi percorrido na buscaDFS.
- **tamanhoRamoBusca**: variável que armazena o tamanho do ramo de busca até agora (isto é: quantos vértices já estão nele).

Auxiliares para as funções:
- **int vOrigem, int vDestino**: para inserir arestas
- **noAux**: auxiliar para uso da função Insere ao inserir na lista de verticesRaiz
- **aux**: apontador usado para percorrer a lista de verticesRaiz.
- **indicaCiclo**: inteiro que armazena o valor retornado por VisitaDFS.
  Se esse valor é igual à constante DIFERENTE então há ciclo e encerra o programa.

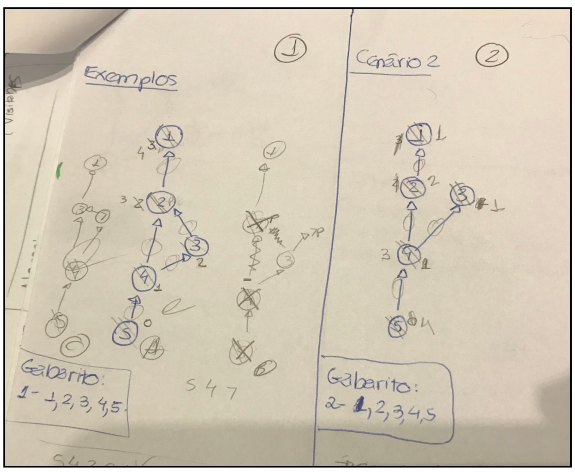
Análise de Complexidade da Solução

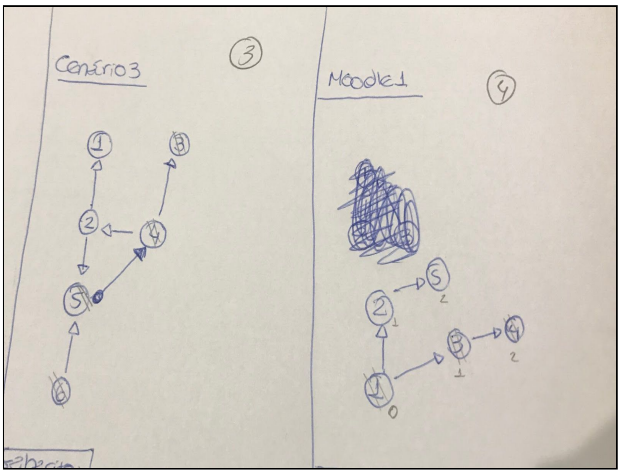
Seguindo o pseudocódigo da seção acima, podemos concluir que o algoritmo tem custo $O(n^2)$ e custo de espaço $O(n^5)$, onde n é o número de vértices / pilotos.

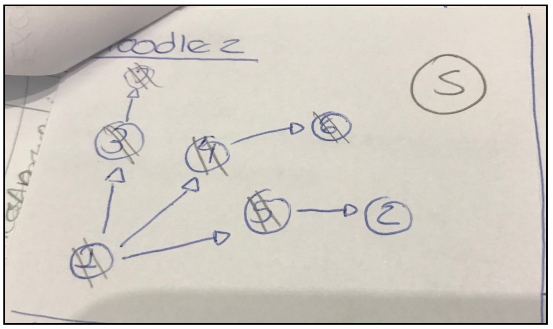
Análise Experimental

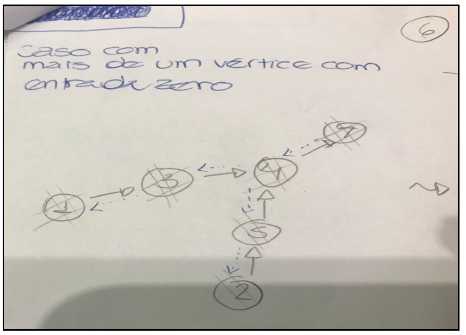
A máquina em que os testes foram realizados foi um notebook com Sistema Operacional Ubuntu 16.04 64 bits instalado, com processador Intel Core i5 2.3 GHz, 8GB de memória RAM e 70 GB de disco.

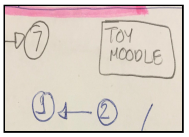
Os onze casos de teste abaixo, que incluem os testes disponibilizados no moodle mais casos meus específicos para cenários específicos de travamento, foram usados para validar cada nova etapa do algoritmo. A cada nova funcionalidade ela foi testada nos 11 cenários.

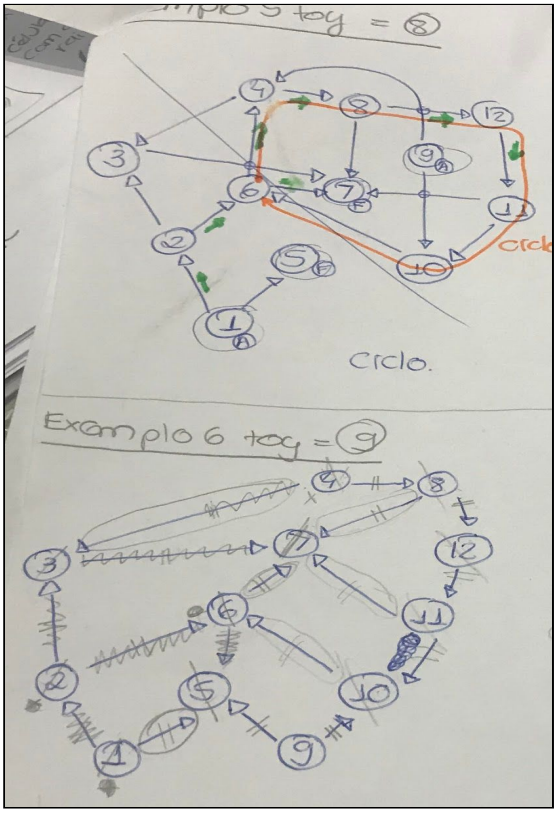


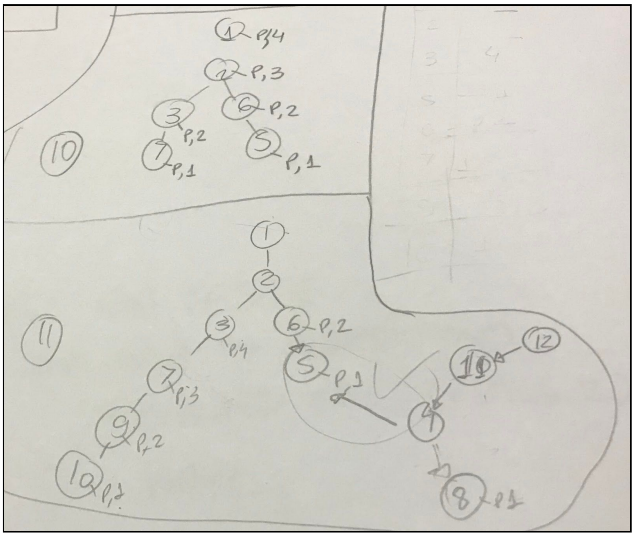












Conclusão

O trabalho foi interessante de implementar. Foi bom para fixar os conceitos de Grafos e de Busca em profundidade. O maior desafio deste trabalho foi o algoritmo e a compreensão do que deveria ser implementado, trouxe bastante experiência.