Sugestões para resolução de alguns problemas

1. Problema: **A – Viagens na minha terra**

- Para representar o grafo, pode ser usada uma matriz (200×200), que não é muito grande. Se se optar pela representação por listas de adjacências, não se deve incluir os ramos que são indicados com valor 2 porque, de facto, não existem.
- O programa deve ser decomposto em funções.
  - `construirGrafo`: ler a informação sobre a rede e retornar a representação correspondente
  - `lerAnalisarPercurso`: ler e analisar um percurso, retornando quantos problemas tem (ou um valor que indique que seria inválido para o grupo). Deve ler os dados do percurso ramo a ramo e
    - determinar em que instante (em minutos) passaria na **origem do grupo** (se passar) e quantos problemas acumulou desde até a **origem da rota** até à **origem do grupo**;
    - pode-se (ou deve-se) interromper essa análise se esse instante for maior ou igual que o **final** para o grupo ou **o número de problemas for maior** do que o mínimo até ao momento ou **encontrar o destino** (antes da origem);
    - se não falhar e o instante em que passa na **origem do grupo** for *compatível* com **instante inicial** para o grupo, prossegue a análise da rota até ao **destino do grupo** (ou o fim dos dados da rota), enquanto for útil (i.e., enquanto não ultrapassar o fim do intervalo em que o grupo está disponível e o número de problemas não exceder o mínimo até ao momento).
    - se não tiver acabado de ler os dados da rota, terá de o fazer (para não baralhar os dados com os de outro percurso)
    - retorna o número de problemas (se a rota for válida para o grupo) ou um valor que indique que é inválida.
  - função `main` que:
    - lê os dados do grupo; os tempos devem ser convertidos a minutos;
    - chama a função que lê os dados da rede e a constrói;
    - processa todos os percursos, por chamada a `lerAnalisarPercurso` para tratar cada um; conta quantas alternativas o grupo teria com número mínimo de problemas;
    - imprime o resultado.

2. Problema: C – Mapa sem sentidos únicos

- Construir o grafo por análise dos percursos indicados. Se implementar em C, usar um *array* de contadores para guardar informação sobre o número de adjacentes de cada nó. Em Java, não é necessário, porque o método `size()` retorna o comprimento de uma `LinkedList` em $O(1)$.

- Notar que cada aresta (x, y) só pode ser inserida uma vez. A função `insert_new_edge` não testa se a aresta já existe, mas há uma outra função `find_edge` que permite verificar se existe. É dito qe as ligações são bidirecionais: se inserir (x, y), deve inserir (y, x), e incrementar o número de adjacentes dos nós x e y, se programar em C. Se se mantém o grafo simétrico, não é necessário verificar se (y, x) existe (quando (x, y) não existe).

- Para cada percurso, as arestas são inseridas à medida que o percurso é analisado. Sendo k o número de nós do percurso, a função de análise deve ter a seguinte estrutura:

  ler(x);
  $compr \leftarrow 0$;
  Para $i \leftarrow 1$ até $k - 1$ fazer
      ler(c); ler(y);
      $compr \leftarrow compr + c$;
      Se (x, y) não existir no grafo, colocar (x, y) e (y, x) no grafo;
      $x \leftarrow y$;
  retornar $compr$;

- **Estrutura do programa principal (para C):**

  ler($nverts$); ler($nperc$);
  criar array de contadores $nadjs$ e inicializá-lo a zero;
  criar grafo g com $nverts$ (sem ramos); // pode usar a biblioteca `graph.h` (ou `graph0.h`)
  Para $i \leftarrow 1$ até $nperc$ fazer
      $compr \leftarrow$ analisa_percurso(g,$nadjs$); // insere os ramos **novos** no grafo e atualiza $ndajs$
      escrever $compr$ no formato pedido;
  Para $v \leftarrow 1$ até $nverts$ fazer
      escrever $nadjs[v]$ no formato pedido;

- **Estrutura do programa principal (para Java):**

  ler($nverts$); ler($nperc$);
  criar grafo g com $nverts$ (sem ramos); //usar a biblioteca `Graph.java` (ou `Graph0.java`)
  Para $i \leftarrow 1$ até $nperc$ fazer
      $compr \leftarrow$ analisa_percurso(g,$nadjs$);
  // insere os ramos **novos** no grafo
      escrever $compr$ no formato pedido;
  Para $v \leftarrow 1$ até $nverts$ fazer
      escrever o comprimento da lista de adjacentes de v no formato pedido;

3. Problema: **D – Reservas**

- Construir o grafo a partir da informação sobre os seus ramos. **Adaptar** `graph.h` **(ou** `Graph.java`**) para poder ter dois valores em cada ramo.**
- Processar cada percurso:
  - Analisar cada ligação (x, y) num percurso:
    - Se (x, y) não existir ou não tiver lugares suficientes para o grupo, imprimir a mensagem correspondente.
    - Se existir e tiver lugares suficientes, reduzir o número de lugares disponíveis nessa ligação e acrescentar o custo do bilhete ao *montante total a pagar por cada bilhete*.
  - Se a reserva para o grupo falhar, é necessário:
    - **acabar de ler os dados do percurso** (se ficou a meio);
    - **repor a informação que foi alterada**. Para isto, será útil memorizar o prefixo do percurso já processado ou, preferencialmente (por tornar a reposição mais eficiente), guardar os identificadores (*apontadores*) dos arcos que foram alterados (para, em tempo $O(1)$, aceder de novo a cada um desses arcos).
- **Estrutura do programa principal:**

  $g \leftarrow$ ler_construir_grafo();
  ler(t);
  Enquanto ($t > 0$) fazer
      processar_reserva(g);
      $t \leftarrow t - 1$;

- Esqueleto da função de processamento da reserva para um grupo (em Java):

```
public static void processar_reserva(Graph g, Scanner stdin) {
   int custoporpessoa  = 0;
   int lugares = stdin.nextInt();
   int nnos = stdin.nextInt();
   LinkedList<Edge> paraRepor = new LinkedList<Edge>();  // arcos a repor
   int i, x =  stdin.nextInt();
   for (i=2; i<= nnos; i++) {
      int y = stdin.nextInt();
      Edge arco = g.find_edge(x,y);
      if (arco == null) {
          escrever a mensagem correspondente
          break;     // interrompe o ciclo
      }
      if (numero de lugares no arco é insuficiente){
          escrever a mensagem correspondente
          break;     // interrompe o ciclo
      }
      paraRepor.addFirst(arco);
      arco.newvalue2(arco.value2()-lugares);  // reduzir lugares no arco
      custoporpessoa  += ...   // acrescentar o preco do bilhete da ligação
      x = y;
   }
   if (i <= nnos) {       // o ciclo foi interrompido
      ler o resto do percurso (se não tiver terminado)
      somar lugares a cada arco guardado na lista paraRepor
   }  else System.out.println("Total a pagar: " + (custoporpessoa*lugares));
}
```

- Esqueleto da função de processamento da reserva para um grupo (em C):

```
void processar_reserva(GRAPH *g) {
   int lugares, nnos, x, y, custoporpessoa  = 0, i, j;
   EDGE *arco;

   scanf("%d%d%d",&lugares,&nnos,&x);
   EDGE **paraRepor = (EDGE **)malloc(sizeof(EDGE *)*(nnos-1)); // arcos a repor
   j=0;
   for (i=2; i<= nnos; i++) {
      scanf("%d",&y);
      arco = find_edge(x,y,g);
      if (arco == NULL) {
          escrever a mensagem correspondente
          break;     // interrompe o ciclo
      }
      if (numero de lugares no arco é insuficiente){
          escrever a mensagem correspondente
          break;     // interrompe o ciclo
      }
      paraRepor[j++] = arco;     // equivale a:  ParaRepor[j] = arco;  j++;
      // reduzir o número de lugares na ligação apontada por arco
      EDGE_VALUE2(arco)  = EDGE_VALUE2(arco) - lugares;   // segundo valor no arco
      acrescentar o preco do bilhete da ligação ao custoporpessoa
      x = y;
   }

   if (i <= nnos) {       // o ciclo foi interrompido
      ler o resto do percurso (se não tiver terminado)
      somar lugares a cada arco guardado no array paraRepor
   }  else printf("Total a pagar: %d\n",custoporpessoa*lugares);

   // libertar o espaço reservado para o array paraRepor
   free(paraRepor);
}
```

4. Problema: **E – Halloween**

- Ideia: para cada caso, se não houver abóboras no supermercado em que a família se encontra, deve localizar (por `BFS_Visit` ou `DFS_Visit`) o supermercado que tem maior número de abóboras entre os acessíveis desse.
- função `main`:
  - ler informação sobre o número de nós (i.e., supermercados) e número de abóboras existentes em cada nó;
  - ler informação sobre o grafo (inserir (x, y) e (y, x), para cada par referido)
  - ler o número de casos, k, a tratar e, para cada caso:
    - ler o nó s (supermercado em que está);
    - se não houver abóboras em s, aplicar `BFS_Visit` a partir de s para encontrar o nó que tem maior número de abóboras e é acessível de s (em caso de empate, escolhe o que tem menor identificador); essa função pode retornar 0 se não encontrar nenhum nó nessas condições (porque os nós da rede são numerados a partir de 1);
    - imprimir o resultado

5. Problema: **F – Onde está Wally?**

- Ideia: semelhante a **Halloween**.
- Na pesquisa em `BFS_Visit`, manter um *array* de **booleanos** que indica, para cada objeto, se foi ou não encontrado; além desse *array*, deve manter o número total dos objetos que encontrou (diferentes);
- Antes de sair, deve escrever a informação sobre a posição do Wally; e sobre os objetos encontrados (não é necessário ordenar; basta escrever os identificadores à medida que percorre esse array).

6. Problema: **G – Transporte Rápido**

- Ideia: Aplicar `BFS_Visit_Distancia`
- **O enunciado diz que se pode *optar por uma jaula maior ou mais pequena***, o que significa que a jaula pode ser tão pequena quanto o necessário para que possa passar na rede. Os máximos indicados para as dimensões da jaula são irrelevantes.
- **Não se deve incluir no grafo os ramos em que a jaula não poderia passar** por mais pequena que fosse. Ou seja, os ramos em que os máximos para a largura, comprimento e altura são inferiores aos mínimos correspondentes para a jaula não são úteis para a resposta. Devem ser descartados durante a leitura dos dados. Assim, o grafo não tem de ter valores nos ramos.
- Qualquer ligação é bidirecional. É necessário construir um grafo simétrico.
- O enunciado refere que **não terá mais do que 500 locais** e que são identificados por inteiros consecutivos, a partir de 1. Esta informação é útil se se usar as estruturas de dados para grafos que estão no arquivo `DAA2324_DataStructures.tgz` (no Sigarra). Para programas em C, note-se que há uma constante simbólica `MAXVERTS` que define a dimensão do *array* de vértices. Essa constante está definida como 1000. Se o número fosse maior, seria necessário mudar essa valor para `MAXVERTS`.

Exemplo de implementação de BFS_Visit (Java)

```
import java.util.LinkedList;
import java.util.Queue;

public static void bfs_visit(int s,Graph g, boolean [] visitados, int [] pai){
    // boolean [] visitados = new boolean[g.num_vertices() + 1];
    // int [] pai = new int[g.num_vertices() + 1];
    visitados[s] = true;
    Queue<Integer> q = new LinkedList<>();
    q.add(s);
    do {
        int v = q.poll();
        for (Edge e: g.adjs_no(v)) {
            int w = e.endnode();
            if (!visitados[w]) {
                q.add(w);
                visitados[w] = true;
                pai[w] = v;
            }
         }
     } while(!q.isEmpty());
}
```

Exemplo de implementação de BFS_Visit (C)

```
void bfs_visit(int s,GRAPH *g, int visitados[], int pai[]){
  int n = NUM_VERTICES(g);
  //  int visitados[n+1], pai[n+1];
  visitados[s] = 1;

  QUEUE *q = mk_empty_queue(n);
  enqueue(s,q);
  do {
    int v = dequeue(q);
    EDGE *e = ADJS_NO(v,g);
    while (VALID_ADJ(e)) {
      int w = EDGE_ENODE(e);
      if (!visitados[w]) {
        enqueue(w,q);
        visitados[w] = 1;
        pai[w] = v;
      }
      e = NXT_ADJ(e);
    }
  } while(!queue_is_empty(q));
}
```