Sugestões para resolução de alguns problemas

1. Problema: **Construção de Mapa**

- Construir o grafo por análise dos percursos indicados. Usar um *array* de contadores para guardar informação sobre o número de adjacentes de cada nó.
- Notar que cada arco (x, y) só pode ser inserido uma vez. A função `insert_new_arc` não testa se o arco já existe, mas há uma outra função `find_arc` que permite verificar se existe.
- Quando o arco (x, y) é inserido, incrementar o número de adjacentes do nó x.
- **Estrutura do programa principal:**

```
ler(nverts); ler(nperc);
criar array de contadores nadjs e inicializá-lo a zero;
criar grafo g com nverts (sem ramos);        //usar a biblioteca grafos0.h
Enquanto (nperc > 0) fazer
    analisa_percurso(g,nadjs);        // insere os ramos novos no grafo e atualiza ndajs
    nperc ← nperc − 1;
Para v ← 1 até nverts fazer
    escrever(nadjs[v]);
```

2. Problema: **Reservas**

- Construir o grafo (biblioteca `grafos2.h`) a partir da informação sobre os seus ramos.
- 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 constante, aceder de novo a cada um desses arcos).
- **Estrutura do programa principal:**

```
g ← ler_construir_grafo();
ler(t);
Enquanto (t > 0) fazer
    processar_reserva(g);  t ← t − 1;
```

- Esqueleto da função de processamento da reserva para um grupo:

```
void processar_reserva(GRAFO *g) {
   int lugares, nnos, x, y, custoporpessoa, i, j;
   ARCO *arco;

   custoporpessoa = 0;
   scanf("%d%d%d",&lugares,&nnos,&x);

   // reserva vetor para arcos a repor
   ARCO **repor = (ARCO **)malloc(sizeof(ARCO *) *(nnos-1));
   j=0;

   for (i=2; i<= nnos; i++) {
      scanf("%d",&y);
      arco = find_arc(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
      }

      repor[j++] = arco;     // equivale a:  repor[j] = arco;  j++;

      // reduzir o número de lugares na ligação apontada por arco
      VALOR2_ARCO(arco) = VALOR2_ARCO(arco) - lugares;

      acrescentar o preco do bilhete da ligação ao custoporpessoa

      y = x;
   }

   if (i <= nnos) {       // o ciclo foi interrompido

      ler os restantes nos do percurso (se houver)

      somar lugares a cada arco guardado no array repor

   }  else   printf("Total a pagar: %d\n",custoporpessoa*lugares);

   // libertar o espaço reservado para o array repor
   free(repor);

}
```

3. Problema: **Ilhas**

- Construir o grafo (biblioteca `grafos0.h`). Notar que as ligações são **bidirecionais**, pelo que se terá de inserir (x, y) e (y, x) para cada ramo dado. Não é necessário verificar se o ramo já existe, porque o *input* não terá ligações repetidas.

- O enunciado diz que "queremos processar a informação para responder **imediatamente** a questões do tipo "Qual é o conjunto a que o nó v pertence?", para vários nós v não necessariamente distintos".

  Isto sugere que a resposta a cada questão deve ser dada em **tempo** $O(1)$, o que não é possível se, de cada vez que se tiver de dar a resposta se determinar a *ilha* (i.e., componente conexa) e o seu representante.

- Portanto, há que *pré-processar* para pré-calcular as respostas. Ou seja, **aplicar BFS**(G) **apenas uma vez** para determinar todas as componentes conexas do grafo G. Nessa pesquisa, para cada vértice v, guardamos $repCom[v]$, o **representante** da componente conexa em que está.

- **Propriedade muito interessante:** na execução de BFS_VISIT a partir de s visita todos os vértices da componente conexa de s e apenas esses. Assim, se em BFS(G) se procurar componentes por **por ordem decrescente de representante (nó raíz)**, todos os nós visitados em BFS_VISIT a partir de s terão s como representante.

  BFS_ADAPTADO(G)
  Para cada $v \in G.V$ fazer
  $visitado[v] \leftarrow$ `false`;
  $repComp[v] \leftarrow v$; // até prova em contrário
  $Q \leftarrow$ MKEMPTYQUEUE();
  Para $s \leftarrow |G.V|$ até 1 com decremento de 1 fazer
  Se $visitado[s] =$ `false` então
  BFS_VISIT$(s, G, Q, repComp, visitado)$;

  Na função BFS_VISIT$(s, G, Q, repComp, visitado)$, não inicializa $visitado$ e, para cada novo nó w encontrado, memoriza que $repComp[w]$ é s.

4. Problemas: **Bons e maus caminhos**, **Quantas depois** e **Rare Order**.

- Seja Σ um alfabeto e seja $<$ uma **relação de ordem total em** Σ. Essa relação induz uma relação de ordem total sobre as palavras de alfabeto Σ, que se designa por **ordem lexicográfica**, e é assim definida:

  > a palavra $a_1 a_2 \ldots a_k$ é **lexicograficamente menor** do que a palavra $b_1 b_2 \ldots b_m$ se e só se para o menor $i \leq \min(k, m)$ tal que $a_i \neq b_i$ se tiver $a_i < b_i$, ou se $a_1 a_2 \ldots a_k$ é prefixo de $b_1 b_2 \ldots b_m$ e $k < m$.

  Exemplos (ver também Sopa de Letras)

  - Se $\Sigma = \{1, 2, 3\}$ e se assumir que $1 < 2 < 3$ então 1112123113 é lexicograficamente menor do que 11121311.
  - Se $\Sigma = \{a, b, c, d\}$ e se assumir que $d < a < c < b$ então a palavra $abbabb$ não é lexicograficamente menor do que a palavra d.

- Nos problemas **Rare Order**, **Sopa de Letras**, **Bons e maus caminhos** e **Quantas depois** são dadas sequências de palavras sobre um alfabeto Σ (que é o conjunto das letras maiúsculas) e assume-se que essa sequência está ordenada lexicograficamente mas não se diz o que é a relação $<$.

  Em **Rare Order**, o objetivo é descobrir a relação $<$ que se definiu em Σ.

- Os problemas **Bons e maus caminhos** e **Quantas depois**, inspirados em **Rare Order**, envolvem também a análise do **grafo de precedências para** $<$, o qual se deduz da análise da sequência de palavras dadas como *input*.

  - **Bons e maus caminhos**: construir o grafo de precedências e analisar percursos nesse grafo.
  - **Quantas depois**: construir o grafo de precedências e analisar quantos letras teriam de ser maiores do que uma letra dada. Notar que essas letras são as acessíveis no grafo de precedências a partir da letra dada.

- As **bibliotecas disponibilizadas para grafos** prevêem que os nós sejam numerados (por números consecutivos a partir de 1).

  Se 'A' corresponder a 1, 'B' a 2,..., 'Z' a 26, o grafo pode ser suportado pela biblioteca.

  Recordar que os códigos das letras maiúsculas são inteiros consecutivos. Pelo que, o nó correspondente a uma letra maiuscula x é o identificado por $x - \texttt{'A'} + 1$.

- Como só se terá no máximo 26 nós, é possível usar uma representação do grafo baseada numa **matriz de incidências** em vez de listas de adjacências.

5. Spreading News

- Aplicação de propriedades de BFS para cálculo de distâncias mínimas a partir de um nó s, sendo a distância dada pelo número de ramos do caminho.

6. Seleção de rota

- Construir o grafo (biblioteca `grafos.h`). As ligações são **unidirecionais**. O valor em cada ramo (1 ou 0) indica se a ligação correspondente tem problemas ou não.

- O grupo pretende escolher uma rota que satisfaz as três condições seguintes:

  - passa na **origem do grupo** e depois **no destino do grupo**;
  - entre a **origem do grupo** e o **destino do grupo** tem lugares suficientes para o grupo;
  - tem o *menor número de problemas* desde a **origem da rota** até ao **destino do grupo** (se houver várias possíveis, escolhe a primeira).

- **Nenhuma rota passa duas vezes no mesmo local.** Por isso, sendo n o número de nós da rede, a inicialização de minProbs com $n + 1$ é um valor correto (e que pode simplificar o programa).

- Escrever uma função para analisar uma rota. Retornará o número de problemas que afetam o grupo ou $n + 1$ se não puder ser usada pelo grupo. Como em **Reservas**, é necessário consumir a rota até ao fim, mesmo que se conclua que não será adequada para o grupo.

- **Estrutura do programa principal:**

```
ler(nelem); ler(origem); ler(destino);
g ← ler_construir_grafo();
minProbs ← NUM_VERTICES(g) + 1;
rotaOtima ← −1;      // não conhecida ainda
ler(p);
Para i ← 0 até p − 1 fazer
    probsRota ← processar_rota(g, nelem, origem, destino, minProbs);
    Se (probsRota < minProbs) então
        minProbs ← probsRota;
        rotaOtima ← i;
escrever_resposta(rotaOtima);
```

A função processar_rota$(g, nelem, origem, destino, minProbs)$ pode usar o valor de $minProbs$ para evitar analisar até ao fim uma rota que já tenha mais problemas do que a $rotaOtima$.

Na implementação desta função:

- **processar a rota** até encontrar a origem **ou** o destino do grupo **ou** o número de problemas já exceder $minProbs$ **ou** a rota acabar;
  - se encontrou a origem, **prosseguir** até encontrar o destino ou a rota acabar ou o número de problemas exceder $minProbs$, tendo em conta a necessidade de lugares para o grupo e os problemas das ligações até ao destino do grupo;
  - se em vez da origem, encontrou o destino do grupo, a rota não será adequada;
  - analogamente, se o número de problemas exceder $minProbs$ antes de chegar à origem, a rota não será adequada;
- **quando se interrompe a análise** por se concluir que a rota não é adequada ou por se chegar ao destino do grupo, é necessário consumir o resto da rota.