Trabalho prático 2 – Infiltração em Rede Social

1) Informação geral

O objetivo do trabalho prático 2 é avaliar a capacidade do estudante para analisar um problema algorítmico, utilizando estruturas derivadas das apresentadas na unidade curricular, e implementar em C uma solução correta e eficiente.

Este trabalho deverá ser feito de forma autónoma por cada grupo na aula prática e completado fora das aulas até à data limite estabelecida. A consulta de informação nas diversas fontes disponíveis é aceitável. No entanto, o código submetido deverá ser apenas da autoria dos elementos do grupo e quaisquer cópias detetadas serão devidamente penalizadas. A incapacidade de explicar o código submetido por parte de algum elemento do grupo implicará também numa penalização.

O prazo de submissão no Moodle de Programação 2 é 5 de junho às 23:59.

2) Descrição

Com as tuas proezas de programação conseguiste-te infiltrar nos servidores de uma rede social de renome, a Redibookchan. Obtiveste uma cópia de uma porção da base de dados de mensagens, que está disponível para fazeres o que te apetecer com ela. A tua mente já fervilha de ideias de como a usar.

A estrutura de dados que guarda as informações dos utilizadores está já implementada, e tem por base numa tabela de dispersão, que se denomina `tabela_dispersao` (ver "tabdispersao.h" para mais detalhes).

O registo `mensagem`, que é referenciado num apontador pertencente ao registo `elemento` da tabela de dispersão de encadeamento fechado, contém informação relativa a uma mensagem de chat da plataforma. O registo `mensagem`, apresenta os seguintes campos:

- *remetente*: nome de utilizador que envia a mensagem (caracteres alfanuméricos, único por utilizador)
- *destinatario*: nome de utilizador que recebe a mensagem (caracteres alfanuméricos, único por utilizador)
- *texto*: informação transmitida (caracteres alfanuméricos)

Por sua vez, cada `elemento` terá como conteúdo:

- *msg*: apontador para um registo `mensagem`,
- *proximo*: apontador para elemento seguinte na mesma "linha" da tabela, em caso de colisão.

Deves implementar:

1. Sabes que consegues descodificar a quem pertencem as mensagens graças a uma função de dispersão que te aponta para a posição certa dentro da estrutura a que tens acesso. Usa o nome do remetente como chave e a função de dispersão presente no registo '`tabela_dispersao`', para aceder à linha certa. Implementa as funções:

```
int tabela_existe(tabela_dispersao *td, const char
    *remetente)
```

Retorna o número de mensagens enviadas por um dado 'remetente' ou -1 em caso de erro. Um pedido de remetente inexistente deverá retornar 0.

```
mensagem **tabela_listagem(tabela_dispersao *td, const
    char *remetente)
```

Retorna um vetor de apontadores para mensagens de um 'remetente' ou NULL em caso de erro. Estes apontadores deverão referir-se às mensagens presentes na tabela e não potenciais cópias. A última posição do vetor deverá ser NULL, de forma a indicar o final do vetor. Assim, se um dado remetente não existir ou não tiver mensagens, será retornado um vetor com apenas um apontador, com valor NULL.

2. Vais precisar de perceber as relações entre utilizadores. Cria uma função para contar número de interações entre 2 utilizadores pré-selecionados:

```
void ligacao2(tabela_dispersao *td, char *nomeU1, char
  *nomeU2, int totMsg[2])
```

Descobre o número total de mensagens enviadas do utilizador 'nomeU1' para 'nomeU2' e retorna-o, na posição 0 de 'totMsgs' e as enviadas de 'nomeU2' para 'nomeU1', na posição 1 de 'totMsgs'. Caso algum dos utilizadores não exista como remetente, deve retornar o valor -1 na respetiva posição de 'totMsg', i.e, se *'nomeU1'* não existe então totMsg[0] = -1.

3. Com base no número de mensagens trocadas consegues perceber quão bem quaisquer dois utilizadores se conhecem. Recorrendo a um vetor de adjacências, cria um grafo pesado e direcionado, onde cada nó corresponde a um utilizador. O peso da ligação de um nó 'X' para um nó 'Y' deve ser igual ao número de mensagens que 'X' enviou para 'Y', não devendo existir ligação caso não haja mensagens enviadas de 'X' para 'Y'. O grafo deverá incluir todos os utilizadores registados na tabela.

```
grafo * criaGrafo(tabela_dispersao *td)
```

Cria e retorna um grafo direcionado, implementado por um vetor de adjacências, com arestas pesados pelo número de mensagens enviadas para os respetivos nós.

O grafo deverá ter os seguintes componentes e campos:

```
Estrutura grafo:

  int tamanho          /* número de posições válidas de 'nos' */
  no_grafo **nos       /* vetor de apontadores para 'no_grafo' */

Estrutura no_grafo:

  char *nome_user      /* String com nome do utilizador */
  int tamanho          /* número de posições válidas de 'ligacoes' */
  ligacao **ligacoes /* vetor de apontadores para 'ligacao' */

Estrutura ligacao:

  int peso_ligacao   /* número de mensagens enviadas do 'nome_user'
                        respetivo, para este 'destino' */
  no_grafo *destino  /* apontador para o no receptor de mensagens */
```

Para acompanhar a estrutura do grafo, deverás também implementar a biblioteca que a manipula:

`grafo *grafo_novo()`

Cria uma instância nova da estrutura 'grafo', retornando um apontador para o mesmo. Retorna NULL em caso de erro.

`void grafo_apaga(grafo *g)`

Apaga o grafo 'g', eliminando-o e libertando toda a memória associada.

`no_grafo *no_insere(grafo *g, char *nomeU)`

Cria uma instância de 'no_grafo' com a informação da String 'nomeU' e insere-a no final do vetor de apontadores para nós. Retorna apontador para o nó criado ou NULL em caso de insucesso ou tentativa de introdução de nó já existente.

`int cria_ligacao(no_grafo *origem, no_grafo *destino, int peso)`

Cria uma ligação entre os nós 'origem' e 'destino', com ponderação 'peso', e insere-a no final do vetor. Retorna 0 se bem-sucedido e -1 em caso de erro ou ligação já existente (imagina que a tabela nunca será actualizada depois de completamente criada).

4. Tens um alvo em mente. Pesquisa o grafo para encontrares esse utilizador. Cria outra função e lista os seus contatos principais para poderes analisar as suas interações:

`no_grafo * encontra_no(grafo *g, char *nomeU)`

Retorna apontador para o nó do grafo 'g' com o nome nomeU. *Retorna NULL se não for encontrado ou em caso de erro.*

`no_grafo **lista_amigos(grafo *g, char *nomeU, int *n)`

Retorna um vetor de apontadores para todos os nós do grafo 'g' que são "amigos" do nó com o nome 'nomeU'. Considera-se que dois nós são amigos se tiverem trocado 4 (quatro) ou mais mensagens em cada direção. Retorna NULL se não forem encontrados

amigos ou em caso de erro. O tamanho do vetor deve ser retornado por referência através do argumento 'n'.

5. Queres tentar seguir o fluxo de informação entre contactos e perceber se há ligações cíclicas, independentemente de quaisquer contactos directos entre os utilizadores. Implementa uma função que liste contactos que criem pelo menos um ciclo:

```
no_grafo ** identifica_ciclo(grafo *g, char *nomeU, int
    M, int *n)
```

*Retorna vetor de apontadores para nó do grafo '*g*'. Os nós retornados deverão constituir o primeiro ciclo encontrado que inclua o utilizador '*nomeU*' e, no mínimo, 2 outros utilizadores. De forma a limitar a pesquisa, a função deve retornar um ciclo com o máximo de 'M' utilizadores, incluindo 'nomeU'. Retorna NULL se não for encontrado nenhum ciclo ou em caso de erro. Retorna o tamanho do vetor retornado via o argumento 'n', que deverá contabilizar a inclusão do nó de partida na primeira posição do vetor. Exemplo: Retorno de [U->V->X->Z], com 'n' retornando 4.*

6. As pessoas têm demasiados contactos bidirecionais e queres-te organizar da melhor forma para os investigar individualmente, de forma rápida. Interessa analisares primeiro as ligações mais importantes de cada utilizador, i.e., os com maior interação. Escolhe a estrutura (ou combinação de estruturas), que te pareça mais apropriada para o fazeres. Não te importa que a elaboração da estrutura seja lenta, mas, quando for para analisar os dados, queres aceder rapidamente aos que interessam (avaliação temporal da extração).

   Reutiliza as estruturas '*elemento*' e '*mensagem*' como blocos básicos a guardar, aproveitando a informação da tabela. Garante que o campo '*proximo*' de cada *elemento* só deverá apontar para outra instância do mesmo remetente e destinatário ou ser NULL. Desenvolve a biblioteca para o que precisas, podendo criar as funções e estruturas que achares adequadas, mas usando o seguinte conjunto de interfaces para interagir com o código de teste:

```
estrutura * st_nova()
```

Cria e inicializa a estrutura que possa alojar as instâncias pretendidas. Retorna o apontador para a estrutura se bem-sucedida ou NULL em caso contrário.

```
int st_insere(estrutura *st , elemento *elem)
```

Tenta inserir o elemento elem *na estrutura '*st*', garantindo acesso mais rápido aos elementos com maior número total de mensagens trocadas. Retorna 0 se bem-sucedida ou -1 em contrário.*

```
int st_importa_tabela(estrutura *st, tabela_dispersao
  *td)
```

*Importa todo o conteúdo da tabela '*td*' para o novo formato de acesso em 'st'. A estrutura 'st' deverá ser preenchida com elementos novos, deixando o conteúdo da tabela de dispersão inalterado. Retorna 0 se bem-sucedido ou -1 em contrário.*

`elemento *st_remove(estrutura *st, char *remetente)`

Extrai a instância do par remetente-destinatário com maior soma de mensagens enviadas e recebidas entre eles (definição de prioridade), levando juntamente só as mensagens enviadas (dados extraídos). A instância retornada deverá ser consumida, i.e., não deverá ficar uma cópia dela em 'st'. Retorna apontador para o primeiro elemento de uma lista ligada de 'elemento', terminada com o campo 'proximo' do último elemento apontando para NULL. Retorna NULL se não for encontrado ou em caso de erro. Esta função será avaliada pelo tempo de execução.

`int st_apaga(estrutura *st)`

Retorna 0 se bem-sucedido e -1 se ocorrer algum erro. Elimina todas as instâncias presentes na estrutura st e desaloca toda a memória da estrutura.

Para todas as funções a implementar, uma descrição da própria, dos seus argumentos e retornos poderá ser consultada nos próprio *header files* "tabdispersao.h", "grafo.h" e "stnova.h".

Os dados relativos ao servidor de mensagens estão disponíveis no ficheiro **"dados.txt"** e estão no formato: **remetente | destinatário| mensagem**.

3) Avaliação

A classificação do trabalho é dada pela avaliação feita à implementação submetida pelos estudantes. A classificação final do trabalho (T2) é dada por:

$$T2 = 0.60\ Implementação + 0.25\ Eficiência + 0.15\ Memória$$

A classificação da implementação e da performance serão essencialmente determinadas por testes automáticos adicionais. Serão avaliados os outputs produzidos pelo código dos estudantes e o tempo de execução face a uma implementação de referência. No caso da implementação submetida não compilar, esta componente será de 0%. Haverá uma diferenciação da classificação, de acordo com a eficiência, como explícito na fórmula, essencialmente focada no exercício 6.

A gestão de memória também será avaliada, tendo a respetiva cotação parcial a contemplar 3 patamares: 100% nenhum *memory leak*, 50% alguns, mas pouco significativos, 0% muitos *memory leaks.*

4) Submissão da resolução

A submissão é apenas possível através do Moodle e até à data indicada no início do documento. Deverá ser submetido um ficheiro *zip* contendo:

- ficheiros **tabdispersao.c, grafo.c, stnova.c e stnova.h**
- **funções adicionais** que desejem usar devem ser implementadas nos ficheiros **stnova.c/.h**
- ficheiro **autores.txt**, indicando o nome e número dos elementos do grupo

Nota importante: apenas as submissões com o seguinte nome serão aceites: `T2_G<numero_do_grupo>.zip`. Por exemplo, `T2_G999.zip`

5) Exemplo de resultados esperados

IINICIO DOS TESTES: Boa sorte

Teste com poucos utilizadores

...verifica_tabela_existe(Remetente inexistente): tabela_existe deu 0 (OK)
...verifica_tabela_existe(Remetente que existe): tabela_existe deu 7 (OK)
OK: verifica_tabela_existe passou

...verifica_tabela_listagem(Remetente inexistente): tabela_listagem a primeira posição deu NULL (OK)
...verifica_tabela_listagem(Remetente que existe): tabela_listagem deu 7 mensagens(OK)
OK: verifica_tabela_listagem passou

...verifica_ligacao2(Os remetentes sao inexistente): ligacao2 deu -1 -1 (OK)
...verifica_ligacao2(Remetentes existentes): ligacao2 deu 0 1 (OK)
OK: verifica_ligacao2 passou

...verifica_criaGrafo(numero de nos): criaGrafo deu 7 (OK)
OK: verifica_criaGrafo passou

...verifica_lista_amigos(sem amigos): lista_amigos deu 0 (OK)
...verifica_lista_amigos(com amigos): lista_amigos deu a amiga MAFALDA (OK)
OK: verifica_lista_amigos passou

...verifica_identifica_ciclo(com ciclo): identifica_ciclo deu um numero de nos igual a 4 (OK)
Os no´s foram: [1 MARIA] : [2 PEDRO] : [3 TERESA] : [4 ANA]
OK: verifica_identifica_ciclo passou

...verifica_st_remove(): st_remove removeu a amiga MAFALDA com 4 mensagens (OK)
OK: verifica_st_remove passou

Teste com muitos utilizadores

...verifica_tabela_existe(Remetente que existe): tabela_existe deu 778 (OK)
OK: verifica_st_remove passou

...verifica_tabela_listagem(Remetente que existe): tabela_listagem deu 521 mensagens(OK)
OK: verifica_tabela_listagem passou

...verifica_ligacao2(Remetentes existentes): ligacao2 deu 135 109 (OK)
OK: verifica_ligacao2 passou

...verifica_criaGrafo(numero de nos): criaGrafo deu 1545 nós (OK)
OK: verifica_criaGrafo passou

...verifica_lista_amigos(com amigos): lista_amigos deu CHESS PLAYER,JACK,JACOB,JOE,MILES,ROB (OK)
OK: verifica_lista_amigos passou

...verifica_identifica_ciclo(com ciclo): identifica_ciclo deu um numero de nos está entre 3 e 8 (OK)
Os no´s foram: [1 ERNEST] : [2 LILLIAN] : [3 FRANCES] : [4 SYMINGTON] : [5 FRANCES] : [6 SYMINGTON] : [7 FRANCES] : [8 LILLIAN]
OK: verifica_identifica_ciclo passou

...verifica_st_remove(): st_remove removeu a amiga CALVIN com 67 mensagens (OK)
Tempo a remover: 0.00000100
OK: verifica_st_remove passou

FIM DOS TESTES: Todos os testes passaram