Simulação da Propagação do Vírus

Programação

Rafael Tavares Ribeiro – 2019131989

Relatório no âmbito da Licenciatura em Engenharia Informática para o projeto desenvolvido para a unidade curricular de Programação

Índice

Introdução

O presente projeto tem como principal objetivo a implementação de um programa através da linguagem de programação em C (por intermédio dos conhecimentos adquiridos ao longo do primeiro ano de licenciatura) que simule a propagação de um vírus entre uma população durante vários dias. A população, fornecida em ficheiros de texto, deverá encontrar-se ligada a locais que constituem um espaço e que por sua vez estão interligados entre si, locais esses também fornecidos por ficheiros binários.

No desenvolvimento do programa, o mesmo foi dividido na fase de preparação e na de simulação. A primeira, faz a leitura da população e locais (que o utilizador escolhe no início do programa), a partir dos ficheiros para listas ligadas e vetor dinâmico respetivamente. De seguida, é possível avançar e/ou recuar nas iterações (dias) que pretendidas e o programa calcula a taxa de disseminação dos doentes para com os saudáveis, a probabilidade de recuperação e duração máxima de infeção dos doentes e a taxa de imunidade de um saudável, depois de doente.

Para estas e outras ações foi construído um menu que apresenta as várias hipóteses ao utilizador. Quando este entender que a simulação deve acabar, será gerado um relatório completo da simulação e um ficheiro de texto que apresenta as características da população mais recente.

Sempre que o utilizador decida recuar até 3 iterações no programa, o mesmo guarda a população e espaço do respetivo dia, tornando-o como o dia atual, descartando os outros.

Estruturas de dados

- **Estrutura do tipo sala, fornecida no enunciado**
  - Caracteriza cada local e irá constituir o array dinâmico.

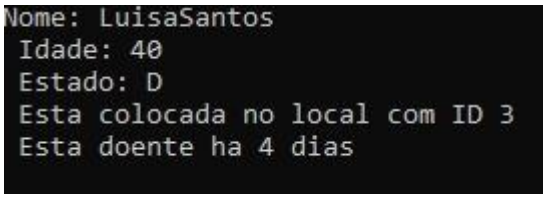
```
typedef struct sala local, *plocal;
struct sala{
    int id;
    int capacidade;
    int liga[3];
};
```

- **Estrutura do tipo pessoa**
  - Caracteriza cada pessoa que constitui uma população e será usada para construir listas ligadas.
  - Sabendo que cada linha do ficheiro de texto representava uma pessoa a partir do seu nome, idade, estado e por fim caso seja um doente, o número de dias que está doente, foi criada esta estrutura que apresenta um campo para cada elemento referido anteriormente.
  - Apresenta ainda um campo que chama a estrutura "local" para ser assim possível atribuir um local a cada pessoa, como pedido no enunciado.
  - Por fim, como serão estas as estruturas que farão parte das listas ligadas, existe um campo (pPessoa prox) do tipo ponteiro da própria estrutura "pessoa" para ser possível apontar para o próximo elemento nas listas ligadas.

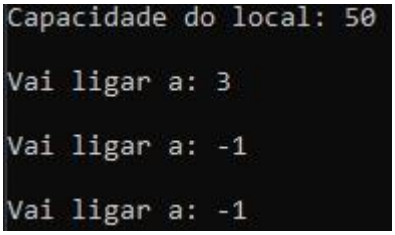
```
typedef struct Pessoas pessoa, *pPessoa;
struct Pessoas
{
    char identificador[30];
    int idade;
    char estado;
    int diasDoente;
    plocal   localRandom;
    pPessoa prox;
};
```

- Para uma melhor perceção de como é construída uma pessoa e local a partir destas estruturas, temos os seguintes outputs.

    - Exemplo de uma pessoa já com uma sala atribuída.



    - Exemplo de um local, com o seu ID e as respetivas ligações (se ligação for -1 significa que liga a nenhum local a partir dessa ligação).



Explicação de funções

- **Transferir N pessoas de um local para outro**
  - Optei por dividir a função em 3 fases:

    1. Validar os Ids indicados;

    2. Verificar capacidades e pessoas totais;

    3. Por fim, se tudo estiver correto, mover as pessoas.

1. Validar os ids

- Obter primeiro o id do local de origem, percorrer o array dinâmico e comprovar que existe. Se não existir, o programa volta a pedir um novo e não avança. De seguida, obtém o id do local de destino e faz a mesma confirmação que a origem.

- Se estiver tudo certo, percorre o array de ligações do local destino e verifica se tem uma ligação à origem, se não tiver, a função termina e apresenta uma mensagem de erro.

- Por fim, confirma se a sala origem tem de facto pessoas para serem transferidas, percorrendo então toda a lista ligada e sempre que existir uma pessoa que tenha no seu campo da estrutura "localRandom" a sala origem, contamos mais uma pessoa no contador. Se esse contador estiver a zero quando saímos da lista, é apresentada mensagem de erro.

- Todas estas ações encontram-se num do – while para garantir que não avançamos para troca de pessoas se algum destes aspetos estiver incorreto.

2. Verificar capacidades

- Começar por contar pessoas que existem no local da mesma forma que fizemos para a origem.

- Obter o número de pessoas a transferir.

- Se o número de pessoas pretendido para transferir for menor que as pessoas que existem na sala, não é possível proceder à transferência e é apresentada essa mensagem de erro.

- Se o passo anterior estiver correto, verificar se a sala origem pode receber todas as pessoas indicadas. Com um ciclo for anteriormente feito que encontrava a sala destino e guardava numa variável auxiliar a capacidade dessa sala, conseguimos saber se há vagas.

  - Ciclo for para guardar capacidade da sala

```
for (x = 0; x<total; x++)
  {
     if (tab[x].id == idDestino)
     {
        capaMaxDest = tab[x].capacidade;
        break;
     }
  }
```

  - Se retirarmos à capacidade total da sala, o número de pessoas, sabemos quantas vagas ainda temos. Se esse número de vagas forem pelo menos igual ao número de pessoas a transferir, podemos avançar.

```
vagas = capaMaxDest - contaPessoasDest;

if (vagas >= numTransf)
{
  flagMover =1;
}

else
{
   printf ("\nErro, nao ha vagas suficientes neste local\n");
}
```

3. Mover as pessoas

- Começar com ciclo for para encontrar o índice da sala destino, para depois sabermos onde fica no array.

- Passar o número de pessoas a mover para uma variável auxiliar, assim não destruímos a original e podemos mexer nesta nova.

```
while (contaPessMover !=0)
{
   pessoaRandom = intUniformRnd(1, contaPessoasOrig);
   aux3 = lista;

        while (aux3 != NULL)
        {
           if (aux3->localRandom->id == idOrigem)
           {
              pessoaRandom-- ;
           }
           if (pessoaRandom == 0)
           {
              aux3->localRandom = &tab[indice];
              contaPessMover--;
              flagSucesso = 1;
              break;
           }
           aux3 = aux3->prox ;
        }


}
```

Enquanto não pedirmos um número aleatório para todas as pessoas a transferir, decrementamos o auxiliar referido anteriormente e só paramos em 0

Pedir número aleatório, esse número será a posição na lista da pessoa que queremos transferir. Ex: número dado = 3, então andamos 3 pessoas que estejam no local da origem na lista, decrementando sempre que saltarmos uma pessoa e paramos quando chegarmos a 0. Ou seja, encontramos a 1ª pessoa que está na origem, saltamos para a 2ª, como ainda não é a 3ª voltamos a saltar.

Atribuir o novo local à pessoa, dando-lhe o novo endereço com o índice encontrado no for anterior.

➢ Atribuir salas a cada pessoa

1. Criar array dinâmico de capacidades
2. Confirmar que existe uma sala que tenha vagas
3. Gerar número aleatório que corresponderá a uma sala
4. Atribuir sala à pessoa

1. Array de capacidades

- Para garantir que a capacidade de cada sala não era ultrapassada ao colocar as pessoas, construi um array dinâmico onde apenas ia colocar cada capacidade de cada local. Assim, temos acesso a esses valores e podia trabalhar com eles sem alterar diretamente na estrutura da sala.

2. Confirmar vagas

- Dentro de um ciclo que percorre toda a lista de pessoas, começar por verificar se existe no array de capacidades, alguma que seja diferente de 0. Se existir, quer dizer que há pelo menos uma sala com vaga.

```
while (lista != NULL)      //percorrer a lista de pessoas
{
   for (j = 0; j < total; j++)
   {
      if (totalCapacidades[j] != 0)          //poderá não haver qaundo voltarmos ao incio deste while
      {
         flag = 1;      //confirmamos que há esaço para colocar mais uma nova pessoa
         break;          //encontramos uma vaga, podemos parar de procurar
      }
   }

   if (!flag)     // se não houver espaço paramos logo
   {
      printf("Nao ha espaço para colocar mais pessoas neste espaco\n");
      break;
   }
```

3. Gerar número aleatório

- Ao guardar um número aleatório que esteja no intervalo [0, total de salas - 1], estamos a obter um índice de uma sala de entre todas as que existem.

- Porém, temos de confirmar se ela essa sala obtida tem vagas no array de capacidades criado anteriormente, se não está igual a zero.

- Sendo assim, com um ciclo do – while, só saímos do mesmo quando dele obtermos um índice que não esteja a zero no vetor de capacidades.

- Fazer isto para cada pessoa pois estamos a percorrer a lista de pessoas com um while (lista!=NULL).

```
do
{
   posiRandom = intUniformRnd(0, total - 1); /

} while (totalCapacidades[posiRandom] == 0);
```

Guardar número aleatório gerada em "posiRandom". Esse número será o índice de uma sala no array de salas. Confirmar com esse mesmo índice se no array de capacidades há vaga.

Se não houver vaga, pedir novamente um índice.

4. **Atribuir sala**

   - Depois de obtido o número aleatório (o índice da sala), apenas é necessário atribuir o endereço da mesma ao campo "localRandom" que é a estrutura do tipo local que está em cada estrutura do tipo pessoa.

   - Por fim, retirar uma vaga da sala no array de capacidades para garantir que para as pessoas seguintes, esse array tem o número de vagas certas.

   - Avançar nos nós da lista para voltarmos a fazer o mesmo processo para todas as pessoas.

```
lista->localRandom = &tab[posiRandom];
totalCapacidades[posiRandom] -= 1;
lista = lista->prox;
```

➢ Funcionalidade de recuar iterações – 3 funções

○ Discrição geral

- Iremos criar um array com 3 ponteiros que apontam exatamente para as 3 listas mais recentes, ou seja, para as 3 últimas populações dos 3 ultimos dias. Sempre que é avançada uma iteração, colocamos essa nova lista no índice 2 do array de ponteiros (índice 2 será sempre a população mais recente). Quando esse array já estiver cheio, sairá a população do índice 0.

- Assim, sempre que recuamos até 3 dias na simulação, podemos obter a população desejada. Se recuarmos apenas 1, será a população do índice 2 a ficar como a atual e o mesmo para os outros dias.

➢ Funções

- Criar array de ponteiros para listas
- Adicionar novas populações ao array
- Recuar iterações

○ Criar array de ponteiros para listas

- Quando o utilizador indica que ficheiro de texto irá ser usado, para além de ser necessário criar lista ligada com esse conjunto de pessoas para o programa geral, também é necessário criar 3 listas ligadas do tipo pessoa, uma para cada um dos 3 ponteiros do array de ponteiros. Criamos assim o array com ponteiros para listas.

```
espaco = cria_vetor_dadosLocais(espaco, &total);
valida_espaco(espaco, total);
lista = cria_listas_dadosPessoas (lista, tabListas);
prepara_arrayListas (lista, tabListas);
initRandom();
selecionaLocal_paraPessoa  (lista, espaco, total);
```

Chamar em simultâneo com as funções que leem os ficheiros logo no início do programa

```
//criar array de ponteiros para listas do tipo pessoa
//onde iremos colocar cada população de cada 3 dias
void prepara_arrayListas (pPessoa lista, pPessoa tabListas[])
{
   pPessoa novo = NULL, aux;
   int i;

   aux = lista;
   while (aux != NULL)
   {
      for (i=0; i<3; i++)     //alocar até 3 listas
      {
         novo = malloc (sizeof (pessoa));

         if (novo == NULL)
         {
            printf ("\nErro a alocar memória\n");
            return;
         }

         novo->prox = tabListas[i];          //alocar no fim
         tabListas[i] = novo;
         novo = NULL;
      }

      aux = aux->prox;
   }
}
```

Para cada um dos 3 ponteiros do array de ponteiro, associar-lhe uma lista ligada do tipo pessoa.

Garantir que fica a NULL para depois ser possível colocar cada nó da lista que queremos copiar

- **Adicionar novas populações ao array**
  - Antes de avançarmos um dia na simulação (chamar todas as funções que podem alterar as características), guardamos no array de ponteiros essa mesma população.
  - Será guardado no índice 2. Se necessário a mais antiga sai para haver espaço para a nova.

```
case '1':
   array_listasNovas (lista, tabListas);
   avancar_iter (espaco, lista, total);
   contaIter++;
```

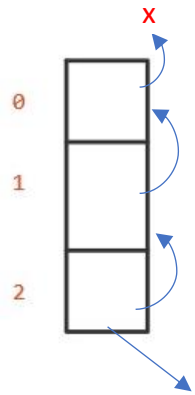
Opção de avançar um dia

Guardar a lista daquele dia no array de listas e só depois chamar as outras funções

```c
//em cada iteração, meter a nova lista desse dia no array de listas.
void array_listasNovas (pPessoa lista, pPessoa tabListas[])
{
    pPessoa novo = NULL, auxA, auxB;

    novo = tabListas[0];
    auxA = lista;
    auxB = novo;

    while (auxA != NULL)
    {
        strcpy(auxB->identificador, auxA->identificador);
        auxB->estado = auxA->estado;
        if (auxB->estado == 'D')
        {
            auxB->diasDoente= auxA->diasDoente;
        }
        auxB->idade = auxA->idade;
        auxB->localRandom = auxA->localRandom;

        auxA = auxA->prox;
        auxB = auxB->prox;
    }

    tabListas[1] = tabListas[2];        //andar com todos para cima
    tabListas[0] = tabListas[1];
    tabListas[2] = novo;
}
```

Como iremos tirar o ponteiro de índice 0 do array, não precisando do conteúdo dele, atribuímos a lista que lhe pertence ao novo, para depois esse novo poder ter uma lista para ser lá colocada nova informação

Copiar todas as características da população atual para a lista que vai entrar no entrar no array de ponteiros, o "novo"



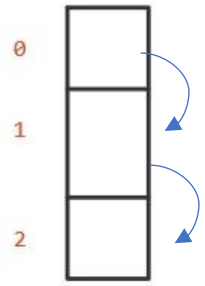
Este ponteiro receberá a lista nova já preenchida

- **Recuar dias**
  - Começar por obter o número de dias que o utilizador deseja recuar e garantir que são no máximo 3
  - Se o utilizador escolhe recuar 1 dia, necessitamos de encontrar a lista que se encontra no array de ponteiros com o índice 2 e colocá-la como a atual.
  - Por fim, é necessário construir de novo o array de ponteiros, visto que o índice 2 será o array atual, não faz sentido ainda se encontrar no array de ponteiros, então teremos de o tirar e colocar o que estava no índice 1 como o do dia exatamente anterior.
  - O processo é parecido para os restantes números de dias a recuar.

```
if (dias == 1)
{

      //fazer com o tab[2] seja o nosso atual agora

   auxA = tabListas[2];
   auxB = lista;

   while (auxA != NULL)
   {

      strcpy(auxB->identificador, auxA->identificador);
      auxB->estado = auxA->estado;
      if (auxB->estado == 'D')
      {
          auxB->diasDoente= auxA->diasDoente;
      }
      auxB->idade = auxA->idade;
      auxB->localRandom = auxA->localRandom;


      auxA = auxA->prox;
      auxB = auxB->prox;

   }

   //retirar tab[2] do array de ponteiros e colocar o 1 no local do 2

   aux = tabListas[2];
   tabListas[1] = tabListas[0];
   tabListas[2] = tabListas[1];
   tabListas[0] = aux;
}
```

A lista que está no índice 2 deve ser a atual, a lista principal. Assim, copiar a lista 2 para a lista do programa.

Iremos descer no array para tornar o que estava no índice 1 como o dia anterior mais recente, andando para baixo. Para o ponteiro do índice 0 não se perder, colocamos lá a lista que estava no 2



Explicação geral

Este projeto está concebido de maneira a serem constituídos grupos de pessoas e um conjunto de salas onde essas pessoas ficarão alojadas para ser possível o cálculo da propagação do vírus que certas pessoas inicialmente o contêm.

É possível calcular a taxa de disseminação (quando uma pessoa fica doente, o programa assume que está há 0 dias), a probabilidade de recuperação dos doentes, a duração máxima em dias que um dia fica infetado, sendo que depois da passagem desses mesmos dias, o doente ficará recuperado. Por fim ainda é possível calcular a taxa de imunidade de uma pessoa recuperada.

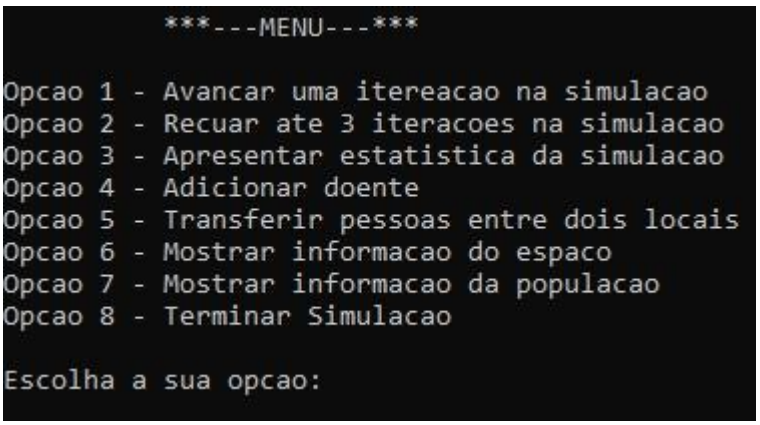
É possível apresentar uma lista que representa a estatística a qualquer momento da simulação, com os seguintes conteúdos: taxas das pessoas imunes, doentes, saudáveis e recuperados e a contagem dos mesmos. Estas informações apenas aparecem se de facto existirem, não aparecendo informação lixo. Na mesma lista, ainda é possível obter a idade da pessoa mais velha e nova de cada estado, isto se facto existirem. A média de idades dos estados também é apresentada bem como a distribuição das salas, sendo possível ver quantas pessoas cada sala tem.

Ações como a adição de doentes ou troca de pessoas entre salas também são possíveis de realizar e a qualquer momento o utilizador pode ver as características de toda a população e locais. Tudo sempre atualizado.

Para uma melhor interação entre o utilizador e programa, foi construído um menu que facilita os pedidos do utilizador para com estas ações que será explicado a seguir.

Manual de utilização

Pela seguinte imagem, podemos perceber melhor como é apresentado o menu no programa.



Explicação de cada opção:

1 . Avançar uma iteração

- Esta opção serve para avançar um dia na simulação, efetuando todas as taxas e cálculos descritos em cima. Sempre que escolhida, iremos alterar as características que tínhamos na população. Irá também ser acrescentada a nova lista ao array de ponteiros para que ao recuar, todas as listas estejam atulizadas.

- Sempre que escolhida esta opção é apresentada uma mensagem em cima do menu apresentando o dia atual da simulação.

2 . Recuar até 3 iterações

- É possível recuar até 3 iterações na simulação. Ao recuarmos, tornamos esse dia como o atual e toda a população volta às características do mesmo dia.

- Apenas é possível recuar até 3 dias, sendo que se for escolhido um número superior é apresentada uma mensagem de erro e o programa volta a questionar.

- Sempre que escolhida esta opção é apresentada uma mensagem em cima do menu apresentando o dia atual da simulação.

3 . Apresentar estatística da simulação

- Quando o utilizador escolhe esta opção, toda a estatística descrita em cima (explicação geral) é apresentada. Uma estatística completa e atualizada do dia em que a simulação se encontra.

4 . Adicionar doente

- É possível adicionar um doente à simulação nesta opção. Ao utilizador, será questionado qual a sala que pretende colocar a nova pessoa e o programa confirma se o ID da sala é positivo, se de facto existe e se tem pelo menos uma vaga. Caso não se confirmem estas situações, será apresentada uma mensagem de erro, o programa questiona novamente o utilizador e o doente não será colocado na simulação.

- Por outro lado, se for possível colocar o doente, o programa irá recolher as características do mesmo.

5 . Transferir pessoas de dois locais

- Nesta opção, o utilizador está possibilitado de trocar pessoas entre duas salas desde que: o ID do local de origem e destino sejam ambos positivos e façam parte do conjunto de salas, existam o mesmo número de pessoas que o utilizador deseja deslocar na sala de origem e por fim, se ao serem colocadas as pessoas na sala de destino, a sua capacidade máxima não será ultrapassada.

- A transferência apenas é realizada se estes aspetos forem confirmados, caso contrário aparecem mensagens de cada erro.

- O utilizador apenas indica quantas pessoas deseja mover e de que sala. As mesmas são escolhidas pelo programa aleatoriamente e depois transferidas.

6 . Mostrar espaço

- Escolher esta opção para a qualquer momento serem apresentados todos os locais e as características de cada um: ID, capacidade total e a que outros locais tem ligação.

7 . Mostrar população

- Escolher esta opção para a qualquer momento serem apresentadas todas as pessoas que fazem parte da simulação, com as suas características: nome, idade, estado (D - doente, S - saudável, I - Imune, R - recuperado), ID sala onde está colocada e por fim, se for doente, há quantos dias está.

8 . Terminar

- Esta opção gera um relatório completo da simulação: estatísticas, último espaço e população, idades, distribuição das salas. Este relatório ficará com o nome “report”.

- Ainda é gerada um ficheiro txt com um nome escolhido pelo utilizador que apresenta as características da última população.
- Por fim encerra o programa.

Conclusão

Com a realização deste projeto foi possível uma forte consolidação das matérias abordadas durante a aprendizagem da linguagem em C e vejo a realização do mesmo como uma boa conclusão deste capítulo de aprendizagem.