CPA: Lista de Exercícios Obrigatórios 1a

A resolução completa das questões desta lista implica na entrega dos códigos-fonte nas linguagens C ou C++ conforme as regras definidas no *Google Classroom* e nos próprios enunciados dos problemas.

Para resolver as Questões 1 a 3 desta lista, você deverá escrever os algoritmos solicitados usando obrigatoriamente as linguagens C ou C++. Nesses programas, o diagrama de um autômato determinístico de estados finitos (ADF) poderá ser armazenado na forma de uma matriz de adjacências. Tal matriz deverá ser um *array* bidimensional do tipo *char*, em que cada elemento é um caractere ASCII representando os eventos que ligam os estados; se não houver transição entre dois deles, o valor da posição correspondente da matriz será NULL (0x00). Os estados não são armazenados explicitamente, eles simplesmente são associados às posições das linhas e colunas da matriz. Assim, supondo que a mesma seja designada por m, o caractere contido em $m[i][j]$ corresponderia ao evento que rotula a transição que sai do estado i para o estado j. Pode-se também adotar a convenção de que o estado inicial é atrelado ao índice 0. Ainda, sem perda de generalidade, assuma que cada transição está associada a um único evento[1].

Nessa convenção, se o número de estados não for muito grande, o conjunto daqueles marcados poderia ser armazenado como uma máscara de *bit*. Por exemplo, se os estados 0, 3 e 4 são marcados, tal informação poderia ser armazenada como o número binário $Xm = 11001 = 0\text{x}19$, ou seja, se o i-ésimo estado é marcado, o i-ésimo *bit* menos significativo da máscara é 1. Esse método é útil, pois permite verificar facilmente se certo estado j é marcado, bastando para isso fazer a operação de deslocamento à direita (*right shift*) da máscara por j *bits*, seguida da operação "E" *bit* a *bit* (*bitwise "AND"*) entre a máscara deslocada e 1 (0x01). Na notação de C/C++, se $\big((Xm \gg j)\ \&\ 0\text{x}01\big)$ for diferente de zero, então o estado representado pelo índice j é marcado. A informação de quais estados são marcados também pode ser armazenada num *array* unidimensional. Por exemplo, o *array xm* do tipo *int* poderia indicar que o estado j é marcado armazenando o valor 1 na posição j, ou seja, $xm[j] = 1$. No caso de C++, poder-se-ia utilizar alternativamente um *array* do tipo *bool* e os valores *true* ou *false*.

Para facilitar o uso da matriz de adjacências e, portanto, a análise do ADF correspondente, cada programa deverá armazenar palavras, isto é, sequências de eventos, na forma de *strings* (*array* de caracteres ASCII). Em C++, a classe padrão *string* pode ser adotada, possuindo a sobrecarga dos operadores "+" e "+=", de modo que para adicionar um caractere ou palavra (operação de concatenação) a uma *string*, basta "somá-

[1] Dado um ADF G com transições rotuladas por mais de um evento, sempre é possível obter um ADF G' equivalente por linguagem a G cujas transições são rotuladas por um único evento apenas. Tomando uma transição associada a múltiplos eventos, basta separar o estado para onde ela se dirige em vários outros, um para cada evento dessa transição, sendo que todos devem possuir comportamento futuro idêntico ao estado original que foi dividido.

la" com os mesmos. Por exemplo, se $s = ab$, a operação $s \mathrel{+}= c$, que equivale a $s = s + c$, resulta em $s = abc$. Na linguagem C, a biblioteca *string.h*, com funções como *strcat*(), pode ser adotada.

Questão 1 - Tomando um ADF acessível G na forma de matriz de adjacências, de acordo com o que foi convencionado anteriormente, construa um algoritmo na linguagem C ou C++ capaz de obter todas as palavras de $\mathcal{L}(G)$ de comprimento até N, ou seja, ele deve encontrar a linguagem $L_N = \{s \in \mathcal{L}(G): |s| \leq N\}$.

O algoritmo deve receber N como argumento de entrada fornecido pelo usuário para a função *main*(). A matriz de adjacência de G pode ser uma variável de escopo global e as palavras de L_N deverão ser armazenadas num *array* de objetos *string* ou num *array* de ponteiros *char*,* o qual, para simplificar, também pode ser declarado como variável global. As palavras de L_N deverão ser impressas de forma organizada na tela no fim da execução.

Seu algoritmo também deve armazenar em outro *array* – também global – as palavras de $\mathcal{L}_m(G)$ que forem obtidas ao longo do processo de obtenção de L_N, ou seja, o conjunto de palavras $L_{N_m} = L_N \cap \mathcal{L}_m(G)$. Você pode usar uma macro, função, método, etc., além da estratégia de máscara de *bits* apresentada anteriormente, ou ambos, para verificar se certo estado é marcado. As palavras de L_{N_m} também deverão ser impressas na tela organizadamente.

Por fim, simule a execução do seu algoritmo usando um ADF G de exemplo e um valor $N \geq 5$. A matriz desse autômato G deverá estar presente no código-fonte entregue. Comente seu programa para que ele seja compreensível.

Atenção! Seu código será testado usando o compilador GNU GCC no Linux. Certifique-se do seu correto funcionamento e inclua na entrega do código-fonte em .c ou .cpp um arquivo README.txt com instruções de compilação e execução. Nomeie o código associando-o à questão da lista. Por exemplo: q1_lista1a.c.

Dica: lembre-se dos algoritmos recursivos usados para percorrer as listas, árvores, etc., aprendidos na disciplina Estrutura de Dados.

Questão 2 - A estratégia de criar um *firmware* na forma de máquina de estados finita (MEF) é bastante vantajosa. Nessa estratégia, o comportamento do programa é modelado pelos estados da MEF, os quais são 'executados' por determinadas funções do código. Assim, em cada ciclo do laço principal, executa-se uma função/estado e a partir de determinada ocorrência (interna ao código ou referente a algo externo, como uma interação com o usuário, interrupções assíncronas, etc.), muda-se a função/estado atual para outra, a qual será executada no laço seguinte. Portanto, a partir da ocorrência de um evento, o autômato associado à MEF do firmware transita de um estado para outro.

Nessa estratégia, convém armazenar a MEF explicitamente em alguma estrutura de dados ou *array* de modo a facilitar a mudança de estado, tornando, ainda, o código mais organizado e legível. Contudo, o armazenamento da MEF como matriz de adjacências, embora poupe espaço de memória, pode tornar a transição menos ágil, pois, sabendo-se o estado atual, o código deve percorrer a linha correspondente da matriz até encontrar o evento ocorrido, determinando, por conseguinte, o estado futuro.

Sacrificando-se eventualmente espaço de memória, pode-se evitar a varredura da linha da matriz armazenando a MEF de uma maneira tal que a partir do estado atual e do evento ocorrido a estrutura de dados retorne imediatamente o estado futuro. Por exemplo, se f é um array bidimensional, então $f[estado\_atual][evento]$ contém o índice associado ao estado futuro. Esse *array* corresponderia à função de transição f vista na teoria de autômatos.

Nessa questão, você deve implementar um código em C ou C++ que gere o "array de transição" f descrito no parágrafo anterior a partir de uma matriz de adjacências definida de acordo com a convenção estabelecida anteriormente. O *array* f deverá conter elementos do tipo *int* e ser alocado dinamicamente de acordo com a matriz de adjacências, a qual pode ser uma variável global. Ao término do programa, o conteúdo de f deverá ser impresso na tela de forma organizada, mostrando o resultado da operação.

Simule a execução do seu algoritmo usando um ADF G de exemplo cuja matriz de adjacências deverá estar presente no código-fonte entregue. Comente seu programa para que ele seja compreensível.

Atenção! Seu código será testado usando o compilador GNU GCC no Linux. Certifique-se do seu correto funcionamento e inclua na entrega do código-fonte em .c ou .cpp um arquivo README.txt com instruções de compilação e execução. Nomeie o código associando-o à questão da lista. Por exemplo: q2_lista1a.c.

Questão 3 - Tomando um ADF acessível G armazenado como uma matriz de adjacências, de acordo com o que foi convencionado anteriormente, ou como uma função de transição f, da forma descrita na Questão 2, crie um programa em C ou C++ capaz de eliminar os estados em que ocorre bloqueio (*livelock* e *deadlock*) e as transições que a eles levam. A saída do seu algoritmo deve ser uma nova matriz de adjacências ou função de transição f correspondente ao autômato $CoAc(G)$.

Os *arrays* que representam G e $CoAc(G)$ podem ser variáveis globais. Você pode usar uma macro, função, método, etc., além da estratégia de máscara de *bits* apresentada anteriormente, ou ambos, para verificar se certo estado é marcado.

Simule a execução do seu algoritmo usando um ADF de exemplo, o qual deverá estar presente no código-fonte enviado. Comente seu programa para facilitar sua compreensão.

Atenção! Seu código será testado usando o compilador GNU GCC no Linux. Certifique-se do seu correto funcionamento e inclua na entrega do código-fonte em .c ou .cpp um arquivo README.txt com instruções de compilação e execução. Nomeie o código associando-o à questão da lista. Por exemplo: q3_lista1a.c.

Questão 4 - A Universidade de Brasília contratou a empresa onde você trabalha para desenvolver um aplicativo para uso dos alunos e que permita o acesso a diversas informações relevantes como: disciplinas matriculadas, histórico, livros emprestados da biblioteca, créditos no restaurante universitário, mapas dos *campi*, além da própria identidade estudantil, entre outras coisas. Ademais, as seguintes regras devem ser atendidas:

- A primeira tela do aplicativo deve solicitar o *login* (CPF) e uma senha de acesso. Deve haver a opção de as informações do usuário ficarem salvas, de modo que ao acessar o aplicativo, a tela de *login* é pulada, sendo então apresentada a tela principal. Nessa última, e apenas nela, deve ser dada ao usuário a opção "Sair", o que levaria novamente à tela de *login*.
- A tela inicial após o *login*, ou a primeira, caso os dados do usuário já estejam salvos, é a já mencionada tela principal. Nela deve haver um menu com botões correspondentes a todas as opções disponíveis ao usuário. Esses botões direcionam o aplicativo para as telas secundárias associadas a essas opções.
- Em cada tela secundária, além dos botões próprios que possam ser necessários para o fornecimento das informações desejadas, deve haver um botão "Início", que leva o aplicativo de volta para a tela principal.
- Para cada tela para a qual o aplicativo for direcionado a partir das secundárias, ou outras posteriores, deve haver o botão "Voltar", que volta uma tela, e o botão "Início", que leva o aplicativo para a tela principal.

Você ficou responsável por implementar a camada de apresentação[2] do aplicativo, incluindo o *layout* das telas. Sendo assim, faça o que se pede nos itens abaixo.

a) Crie um autômato que modele o comportamento do aplicativo do ponto de vista das telas exibidas, tomando como eventos as entradas do usuário: informações de *login* e senha (que podem ser corretas ou incorretas), um botão pressionado, etc. Esboce o *layout* de cada tela fazendo as associações com os estados do autômato, usando ainda, uma nomenclatura compacta para facilitar a desenho do seu diagrama. Seu autômato deve contemplar no mínimo as opções descritas no enunciado bem como as quatro regras exigidas. Nas telas secundárias, dê as opções que achar relevantes. Explicite os conjuntos X e E, bem como o significado de cada estado e evento. Escolha o conjunto X_m como quiser.

a) A implementação da camada de apresentação não se resume à definição dos *layouts* das telas e da interconexão entre elas. Para que você possa usar o autômato criado no item "a" como base para o desenvolvimento do código, modifique-o de modo a incluir a ocorrência de erros que podem surgir nas transa-

[2] Um padrão comumente adotado para o desenvolvimento de um *software* consiste em dividi-lo em três camadas principais: camada de apresentação (interface gráfica do usuário), camada de negócios (lógica do programa que visa cumprir seu propósito) e camada de persistência (acesso a bancos de dados). Um padrão bastante popular e que de certo modo emprega essa estratégia é o MVC, *Model-View-Controller*.

ções com a camada de negócios, e entre essa e a camada de persistência. Esses erros poderiam incluir o lançamento de exceções, falhas de acesso ao banco de dados da UnB, ausência de conexão com a internet, etc. Do ponto de vista da interface gráfica, considere que quando da ocorrência de um problema devem ser exibidos no ecrã uma mensagem de erro e dois botões, um com a opção "Voltar", que leva o aplicativo para a tela a partir da qual o erro ocorreu, e outro com a opção "Início", que faz com que a aplicação volte para a tela principal. Além de desenhar o novo diagrama, lembre-se de explicitar os conjuntos X e E do autômato modificado, descrevendo o significado dos novos estados e eventos adicionados. Escolha o conjunto X_m como quiser.

Atenção! Nas respostas dos itens *a* e *b* desta questão, você também deve incluir a representação gráfica dos autômatos.