BCC202 – Estruturas de Dados I

Departamento de Computação - Universidade Federal de Ouro Preto - MG
Professor: **Pedro Silva** (`www.decom.ufop.br/`)

Trabalho Prático III (TP III) - 10 pontos, peso 1.

- Data de entrega: disponível na plataforma de entraga. O que vale é o horário do Moodle, e não do *seu*, ou do *meu* relógio!!!

- Clareza, identação e comentários no código também vão valer pontos. Por isso, escolha cuidadosamente o nome das variáveis e torne o código o mais legível possível.

- O padrão de entrada e saída deve ser respeitado exatamente como determinado no enunciado. Parte da correção é automática, não respeitar as instruções enunciadas pode acarretar em perda de pontos.

- Durante a correção, os programas serão submetidos a vários casos de testes, com características variadas.

- A avaliação considerará o tempo de execução e o percentual de respostas corretas.

- Eventualmente serão realizadas entrevistas sobre o trabalho para complementar a avaliação;

- O trabalho é em grupo de até 2 (duas) pessoas.

- Será aceito trabalhos após a data de entrega, todavia com um decréscimo de 0,05 a cada 10min.

- Os códigos fonte serão submetidos a uma ferramenta de detecção de plágios em software.

- Códigos cuja autoria não seja do aluno, com alto nível de similaridade em relação a outros trabalhos, ou que não puder ser explicado, acarretará na perda da nota.

- Códigos ou funções prontas específicas de algoritmos para solução dos problemas elencados não são aceitos

- Não serão considerados algoritmos parcialmente implementados.

- Procedimento para a entrega:.

1. Submissão: via **Moodle**.
2. Os nomes dos arquivos e das funções devem ser especificados considerando boas práticas de programação.
3. Funções auxiliares, complementares aquelas definidas, podem ser especificadas e implementadas, se necessário.
4. A solução deve ser devidamente modularizada e separar a especificação da implementação em arquivos *.h* e *.c* sempre que cabível.
5. Os arquivos a serem entregues, incluindo aquele que contém *main()*, devem ser compactados (*.zip*), sendo o arquivo resultante submetido via **Moodle**.
6. Você deve submeter os arquivos *.h*, *.c* e o *.pdf* (relatório) na raiz do arquivo *.zip*. Use os nomes dos arquivos *.h* e *.c* exatamente como pedido.
7. Caracteres como acento, cedilha e afins não devem ser utilizados para especificar nomes de arquivos ou comentários no código.

- Bom trabalho!

Autômato Celular

Introduzidos pelo matemático John Von Neumann na década de 1950 como modelos simples de auto reprodução biológica, os autômatos celulares [1] podem representar diversos sistemas evolutivos, sendo aplicado na modelagem de sistemas físicos, biológicos e sociológicos, simulações do comportamento de gases e fluidos, processamento de imagem e outros. Suas principais características são a computação descentralizada, onde cada célula é capaz de computar sua evolução apenas baseando-se nos estados anteriores do sistema de células, e a simplicidade de suas regras que dificilmente podem ter suas consequências globais previstas.

Um autômato celular é definido por seu espaço celular e sua regra de transição. O espaço celular é composto por um reticulado de N células idênticas dispostas em um arranjo d-dimensional, tendo um padrão idêntico de conexões locais entre as células e condições de contorno bem definidas. A regra de transição fornece o estado da célula no próximo passo de tempo baseado na configuração da vizinhança atual, de modo que todas as células são atualizadas de acordo com essa regra.

Um dos autômatos mais conhecidos foi proposto pelo matemático John Horton Conway em 1970 [2]. Esse autômato ficou conhecido como o jogo da vida (the game of life), com regras fixas e simples que podem gerar comportamentos bastante complexos. O reticulado bidimensional possui células em dois estados: viva ou morta. A cada transição (ou iteração), o estado de uma célula pode ou não ser alterado de acordo com as células vizinhas. O conjunto de regras incluem:

- Uma célula viva se mantém viva se tiver 2 (duas) ou 3 (três) células vizinhas vivas;
- Uma célula viva torna-se morta se houver mais de 3 (três) células vizinhas vivas por superpopulação (sufocamento);
- Uma célula viva torna-se morta se houver menos que 2 (duas) células vizinhas vivas por subpopulação (solidão);
- Uma célula morta torna-se viva se houver exatamente 3 (três) células vizinhas vivas, por reprodução (renascimento).

A Figura 1 mostra alguns exemplos do jogo da vida. O reticulado bidimensional 5x5 apresenta células vivas na cor preta e células mortas na cor branca. As regras especificadas anteriormente são aplicadas a cada transição, alterando o estado das células. Os reticulados à direita são as gerações posteriores. Note que a vizinhança para definir o estado da célula na próxima geração corresponde às oito células que perfazem um quadrado em torno dela. Os exemplos 1 e 2 são de populações que se extinguem. O exemplo 3 retrata uma configuração inicial de células que se estabiliza, tendendo a não sofrer mais alterações ao longo das transições. O exemplo 4 representa uma formação oscilatória, que pode voltar a sua configuração original após algumas gerações.

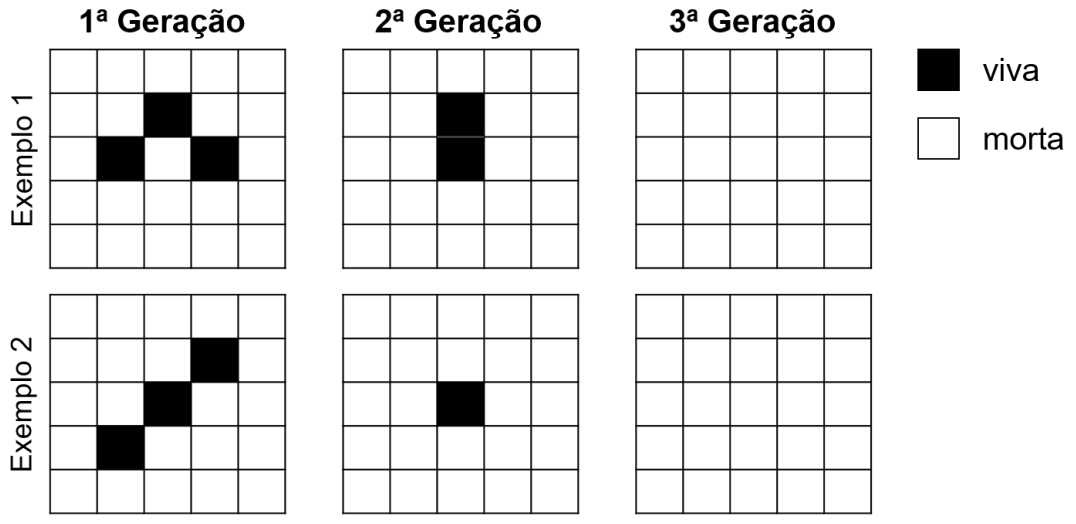

Figura 1: Evolução de reticulados no autômato celular “jogo da vida”. Estados: célula viva (preto) e célula morta (branco).

Pode-se utilizar uma simples matriz para armazenar o reticulado, todavia, para este trabalho, propõe-se a substituição da matriz por uma Tabela Hash.

O seu objetivo é implementar um programa em linguagem C que permita ler reticulados do jogo da vida de modo a retornar a malha de alguma geração seguinte utilizando Tabelas Hash com endereçamento aberto e utilizando a técnica de *hashing* duplo. A atualização das células deve seguir as regras especificas anteriormente. Importante: utilize suas habilidades de programação para implementar um tipo abstrato de dados (TAD) para o problema proposto.

Imposições e comentários gerais

Neste trabalho, as seguintes regras devem ser seguidas:

- Seu programa não pode ter *memory leaks*, ou seja, toda memória alocada pelo seu código deve ser corretamente liberada antes do final da execução. (Dica: utilize a ferramenta *valgrind* para se certificar de que seu código libera toda a memória alocada)
- Um grande número de *Warnings* ocasionará a redução na nota final.

O que deve ser entregue

- Código fonte do programa em C (**bem identado e comentado**).
- Documentação do trabalho (relatório[1]). A documentação deve conter:
  1. **Implementação:** descrição sobre a implementação do programa. Não faça "*print screens*" de telas. Ao contrário, procure resumir ao máximo a documentação, fazendo referência ao que julgar mais relevante. É importante, no entanto, que seja descrito o funcionamento das principais funções e procedimentos utilizados, bem como decisões tomadas relativas aos casos e detalhes de especificação que porventura estejam omissos no enunciado. Muito importante: os códigos utilizados na implementação devem ser inseridos na documentação.
  2. **Impressões gerais**: descreva o seu processo de implementação deste trabalho. Aponte coisas que gostou bem como aquelas que o desagradou. Avalie o que o motivou, conhecimentos que adquiriu, entre outros.
  3. **Análise**: deve ser feita uma análise dos resultados obtidos com este trabalho.
  4. **Conclusão**: comentários gerais sobre o trabalho e as principais dificuldades encontradas em sua implementação.
  5. **Formato:** PDF ou HTML.

Como deve ser feita a entrega

Verifique se seu programa compila e executa na linha de comando antes de efetuar a entrega. Quando o resultado for correto, entregue via *Moodle* até a data disponível na plataforma de entraga um arquivo **.ZIP** com o nome e sobrenome do aluno. Esse arquivo deve conter: (i) os arquivos *.c* e *.h* utilizados na implementação, (ii) instruções de como compilar e executar via terminal, e (iii) o relatório em **PDF**.

Detalhes da implementação

Para atingir o seu objetivo, você deverá construir um Tipo Abstrato de Dados `AutomatoCelular` como representação do reticulado que você quer analisar. O TAD deverá implementar, pelo menos, as seguintes operações:

1. `alocarReticulado`: aloca um (ou mais) TAD `AutomatoCelular`.
2. `desalocarReticulado`: desaloca um TAD `AutomatoCelular`.

[1]Exemplo de relatório: `https://www.overleaf.com/latex/templates/modelo-relatorio/vprmcsdgmcgd`.

3. `LeituraReticulado`: inicializa o TAD `AutomatoCelular` a partir de dados do terminal.

4. `evoluirReticulado`: função que retorna o reticulado presente no TAD `AutomatoCelular` após o número de gerações. **Esta função pode ou não ser recursiva!**

5. `imprimeReticulado`: imprime o TAD `AutomatoCelular`.

Alocação de um ou mais TADs `AutomatoCelular` fica a critério do aluno. Contudo, o autômato celular deve ser representado como uma **Tabela Hash de Endereçamento Linear com endereçamento duplo**. Você deve usar um vetor de pesos para a primeira função *hash* com valores $[1, 2]$ e um vetor de pesos para a segunda função *hash* com valores $[3, 4]$

O TAD deve ser implementado utilizando a separação interface no *.h* e implementação *.c* discutida em sala, bem como as convenções de tradução. Caso a operação possa dar errado, devem ser definidos retornos com erro, tratados no corpo principal. A alocação da TAD necessariamente deve ser feita de **forma dinâmica**.

É obrigatório o uso de alocação dinâmica de memória para implementar as listas de adjacência que representam as matrizes. A análise de complexidade de uma geração deve ser feita em função de m (dimensão da matriz).

O código-fonte deve ser modularizado corretamente em cinco arquivos: *tp.c*, *automato.h*, *automato.c*, *double_hash.h* e *doubla_hash.c*. O arquivo *tp.c* deve apenas invocar e tratar as respostas das funções e procedimentos definidos no arquivo *automato.h*. A separação das operações em funções e procedimentos está a cargo do aluno, porém, **não deve haver acúmulo** de operações dentro de uma mesma função/procedimento.

O limite de tempo para solução de cada caso de teste é de apenas **um segundo**. Além disso, o seu programa não pode ter *memory leaks*, ou seja, toda memória alocada pelo seu código deve ser corretamente liberada antes do final da execução. (Dica: utilize a ferramenta *valgrind* para se certificar de que seu código libera toda a memória alocada). *Warnings* ocasionará a redução na nota final. Assim sendo, utilize suas habilidades de programação e de análise de algoritmos para desenvolver um algoritmo correto e rápido!

Entrada

A entrada é dada por meio do terminal. Para facilitar, a entrada será fornecida por meio de arquivos. [2] A primeira linha especifica as dimensões do reticulado D que tem o mesmo número de linhas e colunas, logo será informado um único valor que corresponde a ordem do reticulado (ex. $D = 10$, reticulado 10 x 10), e também é fornecido o número de gerações a serem processadas. Em seguida, é apresentada uma matriz de valores binários que reproduz o reticulado do jogo da vida a ser analisado. A matriz informa o estado inicial das células, sendo (0) se a célula estiver morta e (1) se a célula estiver viva. A Figura 2 ilustra a equivalência entre o reticulado e a matriz de valores. Considere a borda do reticulado inicial como sempre tendo células mortas.

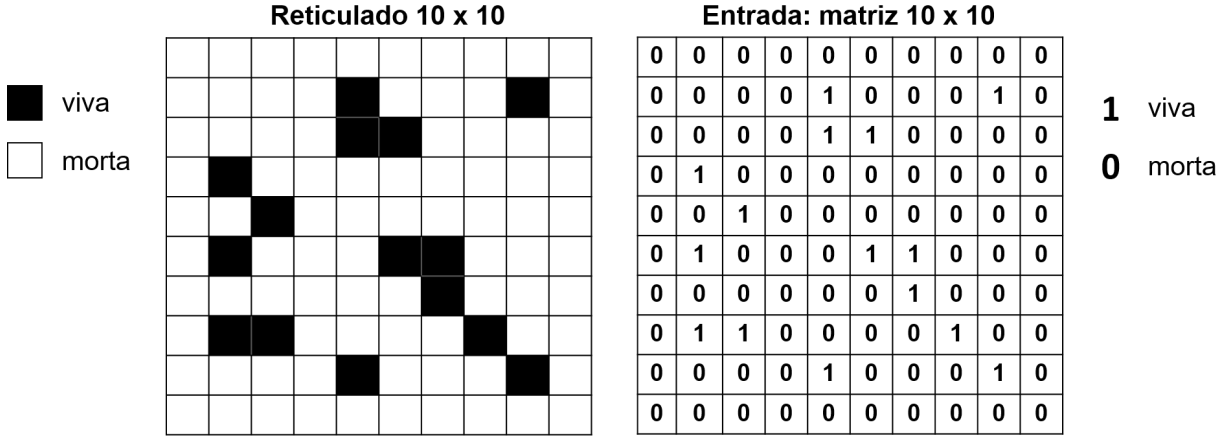

Figura 2: Representação da entrada.

[2] Para usar o arquivo como entrada no terminal, utilize `./executavel < nome_do_arquivo_de_teste`.

Saída

A saída deve ser uma matriz com a mesma dimensão da entrada contendo o estado das células na próxima geração (**Aqui devem ser impressos os zeros**). A definição do estado das células deve ser com base no conjunto de regras do jogo da vida. A Figura 3 mostra como seria a saída para o reticulado da Figura 2. Note a matriz com o estado das células na próxima geração e o reticulado que ela representa.

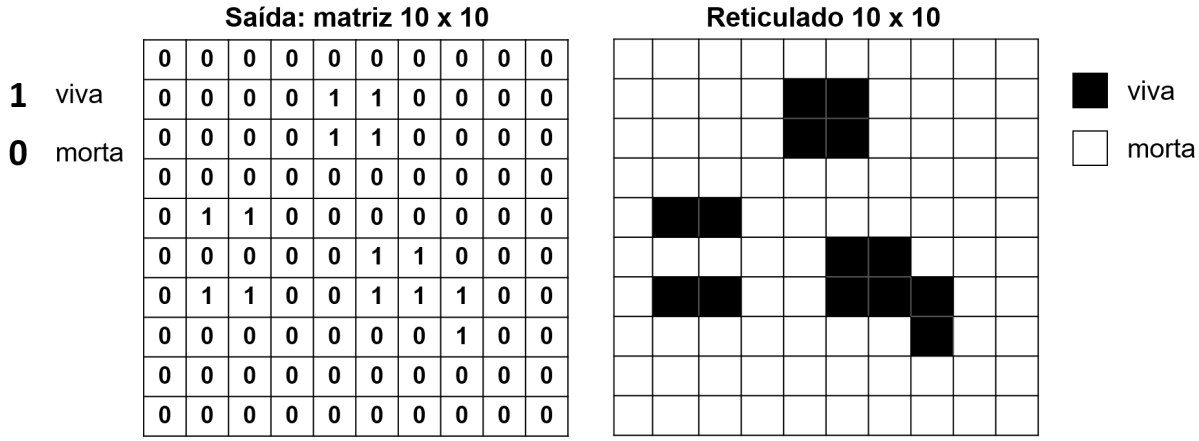

Figura 3: Representação da saída.

Exemplo de um caso de teste

Exemplo da saída esperada dada uma entrada:

Entrada	Saída
10 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 0 0 0 0 0 0 1 1 1 0 0 1 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 1 0 1 1 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 1 1 0 0 0

Entrada	Saída
5 5 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 0 0 0 0	0 0 1 0 0 0 1 0 1 0 1 0 0 0 1 0 1 0 1 0 0 0 1 0 0

A SAÍDA DA SUA IMPLEMENTAÇÃO DEVE SEGUIR EXATAMENTE A SAÍDA PROPOSTA.

Diretivas de Compilação

As seguintes diretivas de compilação devem ser usadas (essas são as mesmas usadas no run.codes).

```
$ gcc -c double_hash.c -Wall
$ gcc -c automato.c -Wall
$ gcc -c tp.c -Wall
$ gcc double_hash.o automato.o tp.o -o exe
```

Avaliação de *leaks* de memória

Uma forma de avaliar se não há *leaks* de memória é usando a ferramenta *valgrind*. O *valgrind* é um *framework* de instrumentação para análise dinâmica de um código e é muito útil para resolver dois problemas em seus programas: **vazamento de memória e acesso a posições inválidas de memória** (o que pode levar a *segmentation fault*). Um exemplo de uso é:

```
1 gcc -g -o exe *.c -Wall
2 valgrind --leak-check=full -s ./exe < casoteste.in
```

Espera-se uma saída com o fim semelhante a:

```
1 ==xxxxx== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Para instalar no Linux, basta usar: `sudo apt install valgrind`.

Referências

[1] Andrew Ilachinski. *Cellular automata: a discrete universe*. World Scientific, 2001.

[2] Martin Gardner. *Mathematical games: The fantastic combinations of john conway's new solitaire game "life"*. Scientific American, 223(4):120–123, 1970.

PONTO EXTRA

Será concedido 0,1 extra para quem implementar a função de *hashing* quadrática, além de comparar o número de colisões obtidos com a versão proposta neste trabalho e a versão do ponto extra.