Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital — IMD

◁ Projeto de Programação • IMD0030 ▷

Simulador do Jogo da Vida de Conway

14 de abril de 2015

Objetivo

O objetivo deste trabalho é implementar um sistema que realize a simulação do jogo da vida de Conway (`http://en.wikipedia.org/wiki/Conway's_Game_of_Life`). Para isso, devem ser utilizadas pelo menos uma estrutura de dados simples (matriz dinâmica) na forma de uma classe em C++. Espera-se que, com este trabalho, possibilitar a aplicação prática dos conhecimentos adquiridos sobre classes, organização de projeto em vários arquivos, alocação dinâmica, gerenciamento de compilação por meio de `Makefile`, documentação adequada de código com o Doxygen, versionamento de software através de Git e implementação de um sistema de simulação simples.

A equipe precisa planejar adequadamente o sistema, para que ele seja eficiente e eficaz. Portanto, é necessário refletir sobre possíveis soluções, para descobrir qual a que resolve o problema da melhor maneira. Além disso, deve existir a preocupação em desenvolver um software de qualidade.

Sumário

1 Introdução

O jogo da vida é, na verdade, uma simulação, não um jogo com jogadores. A simulação ocorre sobre uma **grade** retangular $n \times m$. Cada posição da grade é denominada de **célula**, a qual pode ou não estar ocupada por um organismo. Células ocupadas são chamadas de *vivas* e células desocupadas são chamadas de *mortas*. A cada **geração**, algumas células morrerão, outras se tornarão vivas, e outras irão manter seu estado atual, dependendo de sua vizinhança. Os **vizinhos** de uma determinada célula são as oito células que a tocam verticalmente, horizontalmente ou diagonalmente.

O estado de uma célula na próxima geração é determinado pelas seguintes regras:-

Regra 1: Se uma célula está viva, *mas o número de vizinhos vivos é menor ou igual a um*, na próxima geração a célula *morrerá de solidão*.

Regra 2: Se uma célula está viva e *tem quatro ou mais vizinhos vivos*, na próxima geração a célula *morrerá sufocada* devido a superpopulação.

Regra 3: Uma célula viva com *dois ou três vizinhos vivos*, na próxima geração *permanecerá viva*.

Regra 4: Se uma célula está morta, então na próxima geração ela *se tornará viva* se possuir *exatamente três vizinhos vivos*. Se possuir uma quantidade de vizinhos vivos diferente de três, a célula permanecerá morta.

Regra 5: Todos os nascimentos e mortes acontecem *exatamente ao mesmo tempo*, portanto células que estão morrendo podem ajudar outras a nascer, mas não podem prevenir a morte de outros pela redução da superpopulação; da mesma forma, células que estão nascendo não irão preservar ou matar células vivas na geração anterior.

Um determinado arranjo de células vivas e mortas em uma grade é chamado de **configuração**. As regras acima determinam como uma configuração muda para outra a cada geração.

A simulação, portanto, consiste em aplicar as regras acima sucessivamente, criando novas gerações a cada iteração. A simulação deve ser finalizada em três situações diferentes: *(i)* o usuário pede para finalizar o programa; *(ii)* a configuração da geração atual é considerada *estável*; e *(iii)* a configuração da geração atual é *extinta*. A seguir são apresentadas alguns exemplos de configuração e definições para os conceitos de configuração *estável* e *extinta*.

Exemplos

Nas próximas figuras, para cada célula, um ponto significa que ela está viva e um número indica quantos vizinhos da célula estão vivos. Considere a configuração apresentada na Figura 1. Pela Regra 1, as células vivas morrerão (de solidão) na próxima geração e a Regra 4

mostra que nenhuma célula se tornará viva. Desta forma, todas as células estarão mortas e nenhuma mudança de configuração ocorrerá nas próximas gerações. Neste caso afirmamos que a configuração tornou-se **extinta**. A última configuração antes da extinção propriamente dita é chamada de configuração **moribunda**.

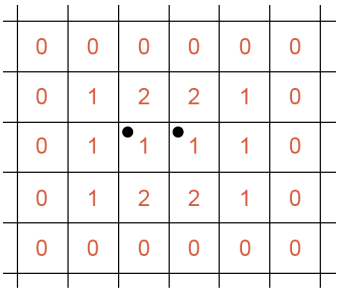


Figura 1: Exemplo de uma configuração moribunda.

Considere agora a configuração da Figura 2. Nela, cada uma das células vivas tem três vizinhos vivos, e, portanto, permanecerá viva, de acordo com a Regra 3. As células mortas têm dois vizinhos vivos ou menos, e, portanto, nenhuma delas se tornará viva (Regra 4). Trata-se de uma configuração **estável**, ou seja, não ocorrerão mudanças de estado mesmo existindo células vivas na configuração.

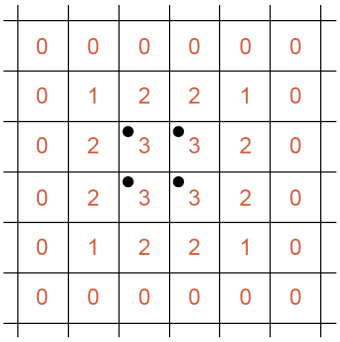
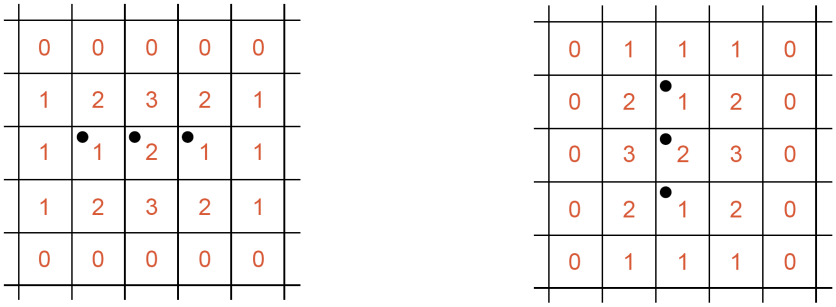


Figura 2: Exemplo de uma configuração estável.

Por fim, observemos as configurações da Figura 3. As duas configurações permanecem se alternando geração após geração, como pode ser percebido pela quantidade de vizinhos vivos.

2 Algoritmo

A Listagem 1 apresenta um possível algoritmo principal em alto nível para realizar a simulação do jogo da vida. A parte mais importante deste algoritmo é a de atualizar a configuração, determinando qual será o estado das células na próxima geração. A estrutura de dado do sistema deve manter a informação sobre as células atualmente vivas na configuração.



(a) Vizinhos horizontais. **(b)** Vizinhos verticais.

Figura 3: Configurações alternáveis.

Listagem 1 Possível algoritmo em alto nível para realizar a simulação do jogo da vida.

```
Carregar uma configuração inicial de um arquivo.
Exibir a configuração.
Enquanto o usuário quiser ver as próximas gerações:
    Atualizar a configuração, aplicando as regras do jogo da vida.
    Exibir a configuração atual.
    Se a configuração atual for estável, parar.
    Se a configuração atual for extinta, parar.
```

Mas como determinar as células que devem viver e as que devem morrer? A dica está em perceber o seguinte fato: só mudarão de estado as células que estiverem vivas ou que forem vizinhas de células vivas. As células vivas são fáceis de serem obtidas, já que elas deverão estar armazenadas de alguma forma. Quando acessamos uma célula sabemos qual a sua coordenada. Para consultar seus vizinhos incrementamos/decrementamos sua coordenada. Por exemplo, se uma célula ocupa a posição $(\mathbf{l}, \mathbf{c})$, seus oito vizinhos são:

$(l-1, c-1)$	$(l-1, c)$	$(l-1, c+1)$
$(l, c-1)$	$(\mathbf{l}, \mathbf{c})$	$(l, c+1)$
$(l+1, c-1)$	$(l+1, c)$	$(l+1, c+1)$

A Figura 4 exibe um exemplo, onde as células com pontos estão vivas e as células preenchidas representam seus vizinhos.

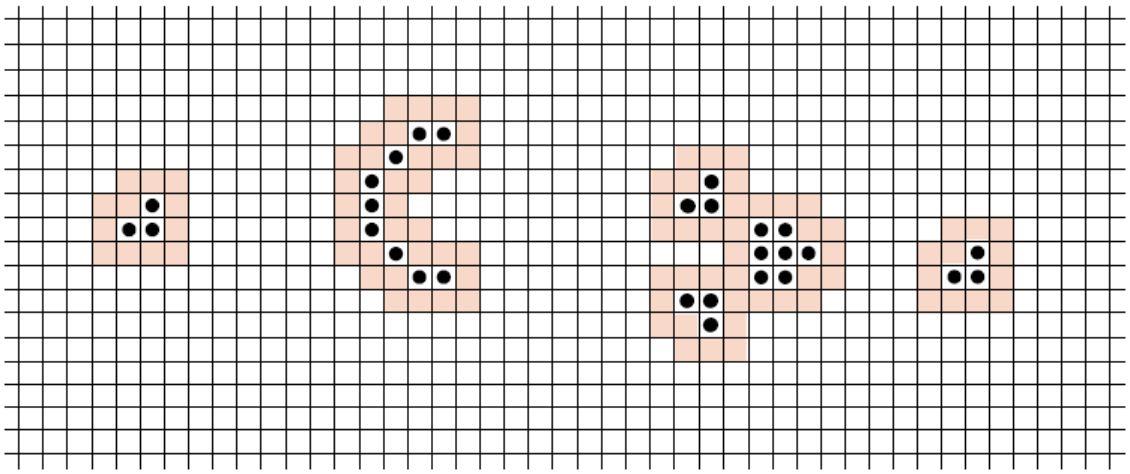


Figura 4: Identificando os vizinhos das células vivas.

Um cuidado especial deve ser tomado com a regra número 5, que diz que todos os nascimentos e mortes devem acontecer ao mesmo tempo. Ou seja, as células que já foram analisadas não devem influenciar na análise das células posteriores. É responsabilidade da equipe planejar uma implementação que respeite esta regra.

Outro cuidado especial deve ser tomado ao contabilizar os vizinhos de células vivas localizadas em uma das quatro *bordas* de uma configuração, visto que tais células não possuem todos os oito vizinhos.

3 Modelagem do problema

Uma sugestão para o gerenciamento dos dados é criar uma classe chamada `Life` que armazena uma matriz cujos elementos correspondem a células que podem ou não estar vivas. Esta classe representaria uma *configuração* no jogo da vida.

Alguns métodos para esta classe discutidos em sala de aula são:-

- ⋆ `Life(nLin, nCol)`: construtor que criar um grade `nLin` × `nCol`.
- ⋆ `setAlive(...)`: método que informa quais células de uma configuração estão vivas. A forma de passar esta informação para a classe fica a cargo de cada equipe.
- ⋆ `update()`: aplica as regras da simulação e atualizada a configuração.
- ⋆ `print()` ou `operator<<()`: imprime o status da configuração na saída padrão.
- ⋆ `stable()`: indica se uma configuração encontra-se estável.
- ⋆ `operator=()` e `operator==()`: podem ser usadas em conjunto para permitir que o cliente verifique se uma configuração encontra-se estável ao manter dois objetos do tipo `Life`, um para a geração atual e outro para a geração anterior.

Note que a implementação do `stable()` implica que a classe `Life` deve manter duas configurações internamente, para saber se a anterior é igual a atual. Alternativamente, a

equipe pode optar por implementar o par `operator=()` e `operator==()` para que o *cliente* seja responsável por manter duas configurações, uma atual e outra anterior, para avaliar se a configuração atual encontra-se estabilizada.

Cada equipe é livre para alterar ou remover os métodos ora sugeridos ou criar novos métodos, se julgar necessário.

4 Entrada de dados

A entrada de dados será feita por meio da leitura de um arquivo texto. O arquivo segue o seguinte formato:

```
<nLin> <nCol>
<caractere_vivo>
<linha_de_dado_1_de_comprimento_nCol>
<linha_de_dado_2_de_comprimento_nCol>
<linha_de_dado_3_de_comprimento_nCol>
...
<linha_de_dado_nLin_de_comprimento_nCol>
```

A primeira linha contém dois inteiros indicando a dimensão da configuração, por meio do número de linhas e colunas da grade. A segunda linha contém um caractere que vai representar células vivas no mapeamento de configuração que se segue. As próximas `nLin` linhas contém informações sobre o mapeamento das células vivas. Cada célula viva é representada pelo caractere informado na linha 2, enquanto que cada célula morta é representada por qualquer caractere *diferente* do caractere de linha 2.

Por exemplo, um arquivo de entrada, `cfg1.dat` válido seria:

```
8 8
*
........
........
..*.*...
..***...
..*.*...
........
........
........
```

O arquivo de entrada deve ser informado ao programa `life_game.cpp` por meio de *argumentos de linha de comando* com a seguinte sintaxe:

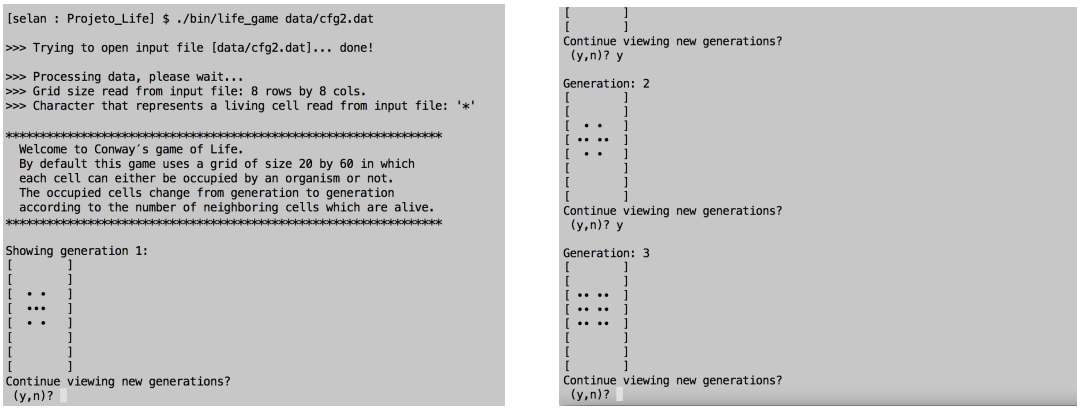
```
$ ./life_game

    Wrong syntaxe!
    Use: life <input_cfg_file> [<output_cfg_evolution_file>]
```

Opcionalmente pode-se gravar em um arquivo texto de saída as diversas gerações criadas durante a simulação, servido como uma espécie de *"histórico"* da evolução da configuração inicial.

5 Execução

A Figura 5 apresenta algumas capturas de tela do programa em funcionamento.



(a) Tela de abertura. **(b)** Após 2 gerações.

Figura 5: Captura da telas de uma possível interface com o usuário.

6 Avaliação

Seu programa deve ser escrito em um *bom estilo de programação*. Isto inclui (mas não fica limitado a) o uso de nomes significativos para identificadores e funções, um cabeçalho de comentário no início de cada arquivo, cabeçalho no formato Doxygen para cada função/método criado, uso apropriado de linhas em branco e identação para ajudar na visualização do código, comentários significativos nas linhas de código, etc.

Para a implementação deste projeto **é obrigatório** a utilização de pelo menos uma classe. O programa completo deverá ser entregue sem erros de compilação, testado e totalmente documentado. O projeto é composto de 100 pontos e será avaliado sob os seguintes critérios:-

- ⋆ Trata corretamente os argumentos de linha de comando (5 pts);
- ⋆ Lê uma configuração a partir de um arquivo ascii e inicializa um objeto `Life` (ou equivalente) de maneira correspondente aos dados (10 pts);
- ⋆ Exibe corretamente uma configuração na tela (10 pts);
- ⋆ Aplica corretamente as regras de evolução descritas na Seção 1 (20 pts);
- ⋆ Executa a evolução da configuração em um laço, cuja iteração é controlada pelo usuário e pelo estado atual da configuração, isto é, em evolução, estável ou extinta (10 pts);
- ⋆ Detecta corretamente quando uma configuração encontra-se estável (10 pts);
- ⋆ Detecta corretamente quanto uma configuração é extinta (10 pts);
- ⋆ Grava arquivo de saída com histórico da evolução da configuração inicial (10 pts);
- ⋆ Programa apresenta código com bom estilo de programação (10 pts);
- ⋆ Programa apresenta pelo menos uma classe (5 pts).

A pontuação acima não é definitiva e imutável. Ela serve apenas como um guia de como o trabalho será avaliado em linhas gerais. É possível a realização de ajustes nas pontuações indicadas visando adequar a pontuação ao nível de dificuldade dos itens solicitados.

Os itens abaixo correspondem à descontos, ou seja, pontos que podem ser retirados da pontuação total obtida com os itens anteriores:-

- ◦ Presença de erros de compilação e/ou execução (até -20%)
- ◦ Falta de documentação do programa com Doxygen (até -10%)
- ◦ Vazamento de memória (até -10%)
- ◦ Falta de um arquivo texto `README` contendo, entre outras coisas, identificação da dupla de desenvolvedores; instruções de como compilar e executar o programa; lista dos erros que o programa trata; e limitações e/ou problemas que o programa possui/apresenta, se for o caso (até -20%).

Boas práticas de programação

Recomenda-se fortemente o uso das seguintes ferramentas:-

- ⋆ Doxygen: para a documentação de código e das classes;
- ⋆ Git: para o controle de versões e desenvolvimento colaborativo;
- ⋆ Valgrind: para verificação de vazamento de memória;
- ⋆ gdb: para depuração do código; e
- ⋆ Makefile: para gerenciar o processo de compilação do projeto.

Recomenda-se também que sejam realizados testes unitários nas suas classes de maneira a garantir que elas foram implementadas corretamente. Procure organizar seu código em

várias pastas, conforme vários exemplos apresentados em sala de aula, com pastas como `src` (arquivos `.cpp`), `include` (arquivos `.h`), `bin` (arquivos `.o` e executável) e `data` (arquivos de entrada e saída de dados).

7 Autoria e Política de Colaboração

O trabalho pode ser realizado **individualmente** ou em **duplas**, sendo que no último caso é importante, dentro do possível, dividir as tarefas igualmente entre os componentes.

Qualquer equipe pode ser convocada para uma entrevista. O objetivo da entrevista é duplo: confirmar a autoria do trabalho e determinar a contribuição real de cada componente em relação ao trabalho. Durante a entrevista os membros da equipe devem ser capazes de explicar, com desenvoltura, qualquer trecho do trabalho, mesmo que o código tenha sido desenvolvido pelo outro membro da equipe. Portanto, é possível que, após a entrevista, ocorra redução da nota geral do trabalho ou ajustes nas notas individuais, de maneira a refletir a verdadeira contribuição de cada membro, conforme determinado na entrevista.

O trabalho em cooperação entre alunos da turma é estimulado. É aceitável a discussão de ideias e estratégias. Note, contudo, que esta interação **não** deve ser entendida como permissão para utilização de código ou parte de código de outras equipes, o que pode caracterizar a situação de plágio. Em resumo, tenha o cuidado de escrever seus próprios programas.

Trabalhos plagiados receberão nota **zero** automaticamente, independente de quem seja o verdadeiro autor dos trabalhos infratores. Fazer uso de qualquer assistência sem reconhecer os créditos apropriados é considerado **plagiarismo**. Quando submeter seu trabalho, forneça a citação e reconhecimentos necessários. Isso pode ser feito pontualmente nos comentários no início do código, ou, de maneira mais abrangente, no arquivo texto `README`. Além disso, no caso de receber assistência, certifique-se de que ela lhe é dada de maneira genérica, ou seja, de forma que não envolva alguém tendo que escrever código por você.

8 Entrega

Você deve submeter um único arquivo com a compactação da pasta do seu projeto. Se for o caso, forneça também o link Git para o seu projeto. O arquivo compactado deve ser enviado **apenas** através da opção Tarefas da turma Virtual do Sigaa, em data divulgada no sistema.

◄ FIM ►