Universidade Federal do Rio Grande do Norte
Depto. de Informática e Matemática Aplicada — DIMAp

◁ Projeto de Programação #1 • DIM0120 ▷

Simulador do Jogo da Vida de Conway

2 de novembro de 2022

O objetivo deste trabalho é implementar um sistema que realize a simulação do jogo da vida de Conway, que é um exemplo bem conhecido de celullar automaton (CA). Para isso, devem ser utilizadas pelo menos uma estrutura de dados simples (matriz dinâmica) na forma de uma classe em C++. Espera-se, 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 `CMake` ou `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

Apesar do nome, o jogo da vida é, na verdade, uma simulação e 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:-

Regras Jogo da Vida

- ⋆ **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.

Outra forma mais compacta de representar as regras do jogo da vida e outros tipos de CA parecidos com o jogo da vida é utilizar o código *"B3/S23"*, que significa: uma célula passa a viver ("**B***orn*") se tem exatamente 3 vizinhos vivos, sobrevive ("**S***urvives*") se tem 2 ou 3 vizinhos vivos, e morre, caso contrário.

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 número máximo de gerações foi alcançado (se esse valor for especificado pelo usuário); *(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*.

1.1 Configuração Extinta

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**.

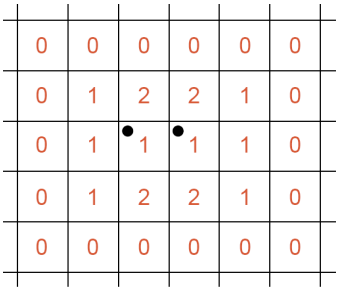


Figura 1: Exemplo de uma configuração moribunda. O ponto indica uma célula vida e a numeração indica quantos vizinhos da célula estão vivos.

1.2 Configuração Estável

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. Esta é um exemplo de configuração estável de *frequência* $n = 0$, ou seja, na próxima iteração da simulação a configuração será a mesma.

De maneira geral, a estabilidade pode ocorrer a uma frequência n (menor), ou seja, a cada $n > 0$ iterações a mesma sequência de configurações se repete. Por exemplo, observemos as configurações da Figura 3. As duas configurações permanecem se alternando geração após geração, como pode ser percebido pela quantidade de vizinhos vivos. Esta é uma configuração estável de *frequência* $n = 1$, ou seja, após uma interação (diferente), a configuração se repete.

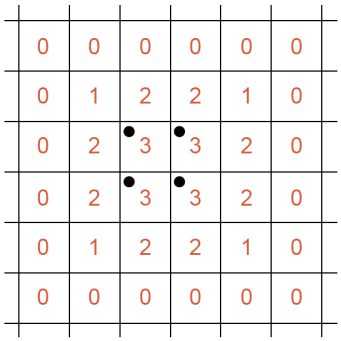
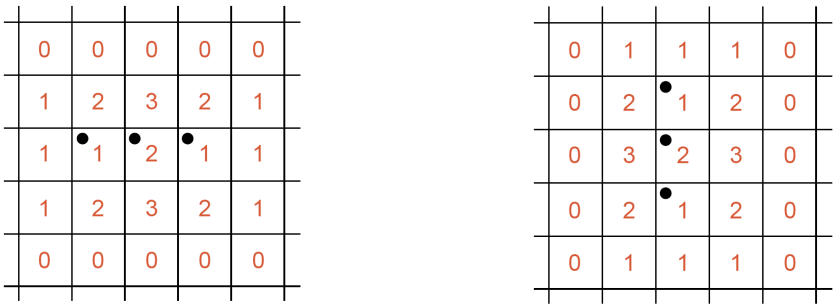


Figura 2: Exemplo de uma configuração estável com *frequência* $n = 0$, ou seja, a configuração não será modificada nas próximas rodadas.



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

Figura 3: Configurações alternáveis, o que implica em um configuração estável com *frequência* $n = 1$.

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. As estruturas de dados do sistema devem manter a informação sobre as células atualmente vivas na configuração.

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 não alcançarmos o número máximo de 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 demais permanecem no mesmo estado (mortas). 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.

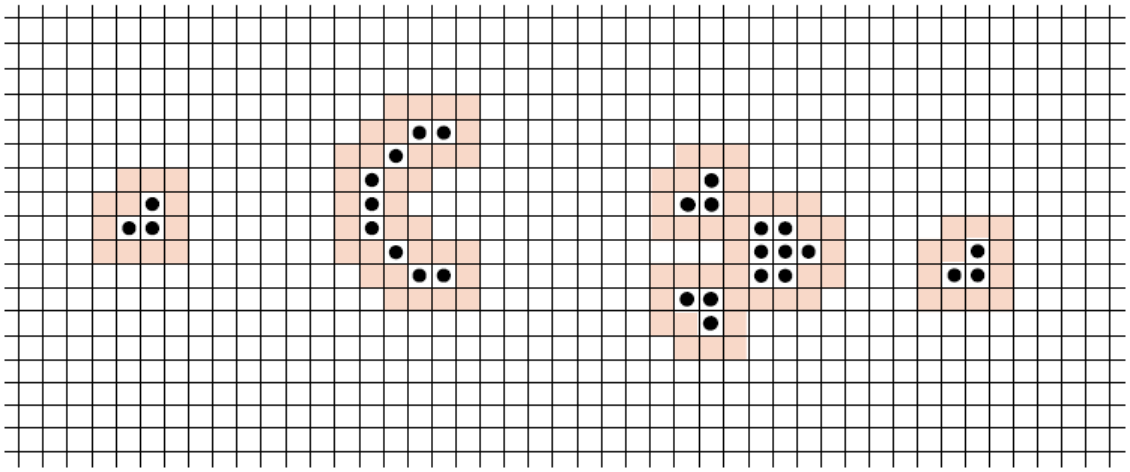


Figura 4: Identificando os vizinhos das células vivas.

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. Alternativamente, é possível fazer uma implementação em que as bordas são "conectadas" (*wrapped*), ou seja, a borda da direita se conecta com a da esquerda, a de cima se conecta com a de baixo, e os cantos se conectam em diagonais—desta forma qualquer célula sempre terá um vizinho.

3 Modelagem do problema

Uma sugestão para o gerenciamento dos dados é criar uma classe chamada `Life` que armazena uma matriz alocada dinamicamente, de acordo com a entrada, e 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 poderiam ser:-

- ⋆ `Life(nLin, nCol)`: construtor que criar um grade `nLin` $\times$ `nCol`.
- ⋆ `set_alive(...)`: 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 ou arquivo texto.
- ⋆ `stable()`: indica se uma configuração encontra-se estável.
- ⋆ `extinct()`: indica se uma configuração encontra-se extinta.
- ⋆ `operator=()`: copia uma configuração para outra.
- ⋆ `operator==()`: compara se duas gerações são distintas ou iguais, com o objetivo de determinar estabilidade.

A implementação do `stable()` requer uma tomada de decisão da equipe de desenvolvimento no sentido de determinar uma estratégia (estrutura de dados?) que permita a identificação de estabilidade ao londo das gerações, ou seja, com frequência $n > 1$.

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 **deve** seguir 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:

```
7 9
*
.........
.........
..*.*
..***
..*.*
......
.............
..............
..
```

A posição de uma célula viva em uma linha deve corresponder a posição que ela vai ocupar na configuração inicial. Uma linha não precisa, necessariamente, estar completamente preenchida, como é o caso das linhas 5, 6, 7 e 8 da configuração. O que importa é que cada linha **da configuração** deve ter o tamanho especificado no cabeçalho, mesmo que as linhas do arquivo de entrada de dados não estejam completas ou ultrapassem o comprimento indicado, como é o caso das linhas 8, 9, 10 e 11 no exemplo acima. O caracteres ‘`.`‘ foi usado apenas para facilitar a visualização, ou seja, poderíamos ter usado qualquer outro caracteres, como por exemplo espaços em branco. Qualquer linha extra deve ser ignorada.

O arquivo de entrada deve ser informado ao programa `glife` por meio de *argumentos de linha de comando*, de acordo com a sintaxe apresentada na Seção 6

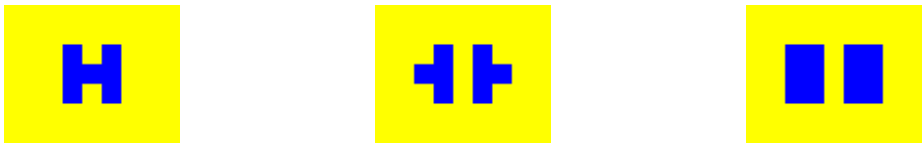
5 Saída de dados

A saída do programa deve ser a exibição de cada geração criada durante a simulação. O programa deve suportar 2 tipos de saída de dados, mutualmente exclusivos: apresentação das gerações no modo texto ou por meio de um conjunto de imagens. Por sua vez, a saída no modo texto pode ser enviada para um arquivo texto informado pelo usuário ou simplesmente enviada para a saída padrão `std::cout`.

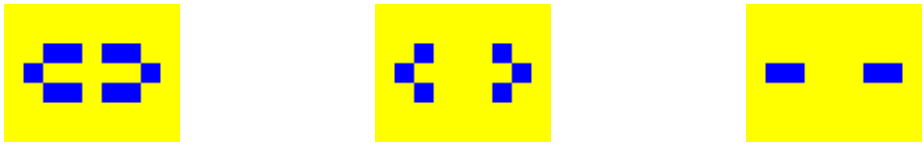
Veja abaixo um exemplo de saída (parcial) da simulação para 50 gerações no modo texto enviado para `std::cout`.

```
$ ./glife  --maxgen 50 ../source/data/cfg2.dat
>>> Trying to open input file [../source/data/cfg2.dat]... done!
>>> Running simulation up to 50 generations, or until extinction/stability is reached, whichever comes first.
>>> Processing data, please wait...
>>> Grid size read from input file: 7 rows by 9 cols.
>>> Character that represents a living cell read from input file: '*'
>>> Finished reading input data file.

****************************************************************
  Welcome to Conway's game of Life.
  Running a simulation on a grid of size 8 by 9 in which
  each cell can either be occupied by an organism or not.
  The occupied cells change from generation to generation
  according to the number of neighboring cells which are alive.
****************************************************************

Generation 1:
[         ]
[         ]
[   * *   ]
[   ***   ]
[   * *   ]
[         ]
[         ]
Generation 2:
[         ]
[         ]
[   * *   ]
[  ** **  ]
[   * *   ]
[         ]
[         ]
```

A Figura 5 apresenta a saída na forma de imagens para o mesmo arquivo de entrada apresentado anteriormente. Foram gerados apenas 6 gerações, visto que a $7^{\underline{a}}$ geração é extinta.



(a) Geração #1. **(b)** Geração #2. **(c)** Geração #3.



(d) Geração #4. **(e)** Geração #5. **(f)** Geração #6.

Figura 5: Sequência de 6 imagens correspondendo a simulação para o arquivo de entrda apresentado anteriormente.

6 Interface

A equipe pode implementar a interface com usuário de duas formas diferentes. A primeira é através do uso de *argumentos de linha de comando* e a segunda e por meio de um *arquivo de*

configurações. Descreveremos cada uma delas a seguir.

6.1 Argumentos de Linha de Comando

No caso da dupla optar pelo uso de argumentos de linha de comando, o programa deve adotar a seguinte interface:

```
$./glife -h
Usage: glife [options] input_cfg_file
  Running options:
    --help               Print this help text.
    --maxgen <num>       Maximum number of generations to simulate. No default.
    --fps <num>          # of generations presented p/ second. Default = 2 fps.
    --imgdir <path>      Images output directory.
    --blocksize <num>    Pixel size of a square cell. Default = 5.
    --bkgcolor <color>   Color name for the background. Default = GREEN.
    --alivecolor <color> Color name for the alive cells. Default = RED.

  Available colors are:
    BLACK BLUE CRIMSON DARK_GREEN DEEP_SKY_BLUE DODGER_BLUE GREEN LIGHT_BLUE
    LIGHT_GREY LIGHT_YELLOW RED STEEL_BLUE WHITE YELLOW
```

O único argumento que é obrigatório é o nome do arquivo de entrada: `inpu_cfg_file` Portanto se o cliente executar o programa sem argumentos, você de indicar que falta informar o nome do arquivo com a configuração inicial e imprimir a descrição da interface exibida via opção *help*.

Independentemente do método de interface adotado, seja via argumentos de linha de comando ou por meio de um arquivo de configuração, o programa sempre exibe as gerações sequencialmente numeradas na saída padrão `stdout`. Se o cliente desejar salva a sequência de gerações em arquivo ascii, basta redirecionar a execução com o símbolo '>' na linha de comando, como em

```
$./glife  --maxgen 50 data/cfg2.dat > saida_simulacao.txt
```

Parâmetros

⋆ Opção `-maxgen`: indica quantas gerações a simulação deve gerar no máximo. A simulação deve ser interrompida se uma dessas condições acontecerem: (1) alcançar a quantidade máxima de gerações solicitadas, ou (2) atingir estabilidade, ou (3) entrar em

extinção. Se não for especificado um limite para número de geração o programa deve simular gerações de maneira indefinida até alcançar estabilidade ou uma geração extinta.

- ⋆ Opção `-fps` (frames por segundo, em Inglês): indica a velocidade de exibição das gerações no modo texto na saída padrão `std::cout`. Por exemplo, se o usuário informar `-fps 2` devemos imprimir na saída padrão 2 gerações a cada segundo.
- ⋆ Opção `-imgdir`: indica a pasta onde as imagens correspondentes a cada geração simulada deverá ser gravada. Se essa opção for indicada, o programa deve gerar uma imagem para cada geração, todas com as mesmas dimensões, cujo nome deve ter o mesmo prefixo, seguido de uma numeração sequencial no formato: `gen_00001.png`, `gen_00002.png`, ..., etc. A adoção desse formato facilitará a posterior renderização dessas imagens em um vídeo. Por outro lado, se essa opção não for especificada pelo usuário isso significa que **não é preciso gerar as imagens de saída**.
- ⋆ Opção `-blocksize`: indica o tamanho de uma célula na simulação. Todas as células são quadradas. O valor *default* é 5 pixels (isto é, um célula de 5×5 pixels)
- ⋆ Opção `-bkgcolor`: indica a cor das células da grade que não estão vivas. Na prática essa cor representa a cor de fundo da imagem gerada.
- ⋆ Opção `-alivecolor`: indica a cor das células vivas da grade.

Notem que as opções `-blocksize`, `-bkgcolor` e `-alivecolor` estão diretamente ligadas a geração e gravação de imagens correspondentes às gerações simuladas. Portanto, se a opção `-imgdir` não for especificada pelo usuário, essas 3 opções perdem o sentido de existir e devem, desta forma, serem ignoradas, caso tenham sido especificadas.

Os argumentos podem ser especificados em qualquer ordem. Veja abaixo alguns exemplos de linhas de comando:

Exemplo 01

```
$./glife  --maxgen 50 data/cfg2.dat -imgdir imgs --bkgcolor Yellow --alivecolor BLUE -
blocksize 10
```

O comando acima pede que os dados de entrada seja lido do arquivo `data/cfg2.dat`, que as imagens sejam geradas na pasta `imgs`, sendo cada imagem gerada com fundo amarelo, células vivas pintadas de azul, com cada bloco representando uma célula ocupando 10×10 pixels e um máximo de 50 gerações (ou menos, se estabilidade ou extinção forem detectados).

Note que como foi solicitado um bloco de pixel com tamanho 10 e a configuração possui 7 linhas por 9 colunas, cada imagem terá o tamanho de 70×90 pixels no total.

Exemplo 02

```
$./glife  virus.dat
```

Na execução acima os dados de entrada serão lidos do arquivo `virus.dat` e a simulação rodará de forma indefinida até extinção ou estabilidade, sendo que as gerações deverão ser enviadas para a saída padrão apenas.

6.2 Arquivo de Configuração

Uma interface alternativa para configurar a execução da simulação é por meio da criação de um arquivo de configuração do programa. Se você adotar essa opção, é preciso ler um arquivo especificado no formato `INI`.

Normalmente arquivos de configurações de programas no Linux ficam escondidos na pasta do projeto, colocando-se um ponto na frente do nome do arquivo. Alternativamente, tal arquivo pode ser armazenado dentro de uma pasta escondida, usualmente chamada de `.config`.

Abaixo segue uma sugestão de arquivo de configuração que contempla todos os parâmetros de execução indicados e descritos na Seção 6.1.

```
; -------------------------------------
; Game of Life Configuration File
; -------------------------------------

; Arquivo de configuração
input_cfg = "data/cfg1.dat"   ; Path para o arquivo com configuração.

; Indica se devem ser geradas imagens ou não.
generate_image = true         ; Se usar 'false' apenas texto é gerado.

; Número máximo de gerações.
; Use zero ou omita, para não limitar a quantidade máxima de gerações.
max_gen = 30

;  Available colors are:
;   BLACK BLUE CRIMSON DARK_GREEN DEEP_SKY_BLUE DODGER_BLUE GREEN LIGHT_BLUE
;   LIGHT_GREY LIGHT_YELLOW RED STEEL_BLUE WHITE YELLOW

; Seção de controle da geração de imagem
[Image]
alive = BLUE      ; Cor da célula viva
```

```
bkg = YELLOW      ; Cor do tabuleiro (célula morta)
block_size = 10   ; Tamanho do pixel virtual
path = "/home/selan/glife/imgs" ; Onde as imagens serão gravadas

; Seção de controle da exibição textual
[Text]
fps = 2           ; Velocidade de exibição da saída padrão.
```

Note que nessa tipo de interface o usuário precisa apenas informar, via linha de comando, o nome do arquivo de configuração, como em:

```
$./glife glife.ini
```

Assim todas as opções de simulação e o arquivo com uma configuração válida serão lidos a partir de tal arquivo de configuração.

Claro, se o usuário esquecer de informar um arquivo de configuração via linha de comando, seu programa deve enviar uma mensagem de erro indicando o problema e instruindo o cliente como proceder.

```
$./glife
Missing path to configuration file!
  Usage: glife setting.ini
```

Para fazer a leitura, validação e processamento de arquivos `.INI`, recomendo usar a biblioteca `TIP`, disponível em (https://github.com/selan-ufrn/project_TIP).

7 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 OU coleta corretamente os parâmetro via arquivo de configuração INI (10 pts);
- ⋆ Lê uma configuração a partir de um arquivo ascii e inicializa um objeto `Life` (ou equivalente) de maneira correspondente aos dados (5 pts);
- ⋆ Exibe corretamente uma configuração na saída padrão (10 pts);
- ⋆ Exibe corretamente uma configuração na imagem gravada (10 pts);
- ⋆ Aplica corretamente as regras de evolução descritas na Seção 1 (10 pts);
- ⋆ Executa corretamente a evolução da configuração, obedecendo o limite de gerações (se informado), ou parando quando a geração alcançar estabilidade ou ser extinta (10 pts);
- ⋆ Detecta corretamente estabilidade da simulação, com a indicação da frequência (i.e. o número da geração onde o ciclo se inicia) (15 pts);
- ⋆ Detecta corretamente extinção da simulação (5 pts);
- ⋆ Gera corretamente os arquivos de imagens, uma imagem para cada geração do histórico de evolução da configuração inicial (10 pts);
- ⋆ Programa apresenta pelo menos uma classe (5 pts).
- ⋆ Trata corretamente erros de entrada do usuário (10 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 `author.md` 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%).

Bônus

Os programas que **estiverem funcionando corretamente e completamente** e conseguirem receber do usuário uma indicação de regra de evolução, como por exemplo $B2/S14$ ou algo nesse formato, poderão receber até 1pt de ponto extra.

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
- ⋆ CMake/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` e `.h`), `lib` (arquivos de bibliotecas externas ao projeto), `build` (arquivos `.o` e executável) e `data` (arquivos de entrada e saída de dados).

8 Autoria e Política de Colaboração

O trabalho pode ser realizado **individualmente** ou em **dupla**, sendo que no último caso é importante, dentro do possível, dividir as tarefas igualmente entre os componentes. A divisão de tarefas deve ficar evidente através do histórico de *commit* do git.

Após a entrega, 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.

9 Entrega

Você pode submeter seu trabalho de duas formas possíveis: via GitHub Classroom (GHC), ou via tarefa do SIGAA. Se decidir enviar via GHC é importante enviar também um arquivo texto via SIGAA indicando a URL do seu projeto no GHC. Se decidir enviar apenas pelo SIGAA,

monte um arquivo ZIP com todo o código fonte do seu projeto e tudo o mais que for necessário para compilar e executar o projeto.

Em qualquer uma das duas opções descritas, lembre-se de remover todos os arquivo executáveis do seu projeto (a pasta `build`) antes de submeter seu trabalho.

◀ FIM ▶