Universidade Federal do Espírito Santo
Departamento de Informática

1ª Trabalho Prático - Etapa 1

Pac-Man revival

Programação II - 2023/2

05 de Outubro de 2023

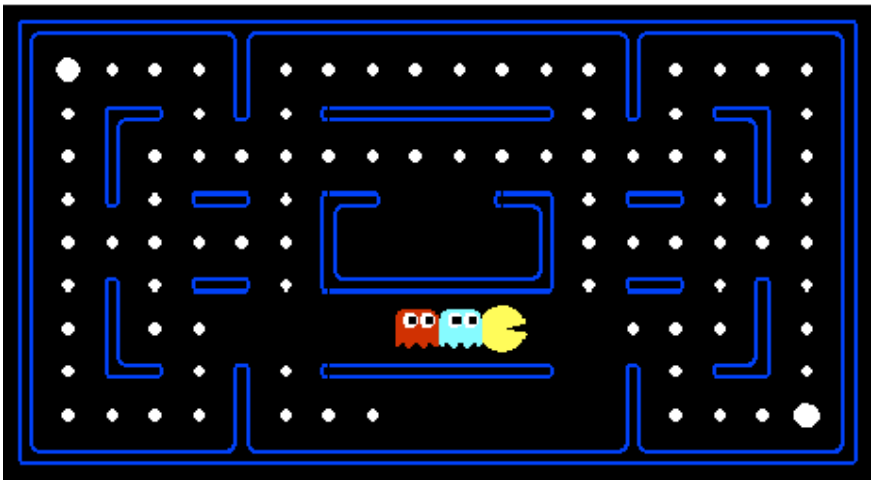
Errata 1 (19/10/2023) - Alterações em vermelho no texto

Introdução

O objetivo deste trabalho é colocar em prática as habilidades de programação na linguagem C adquiridas ao longo do curso. O trabalho foi elaborado para que você exercite diversos conceitos principalmente a construção de TADs e a alocação e manipulação de memória de maneira dinâmica. Como consta no plano de ensino da disciplina, o trabalho será realizado em duas etapas (a primeira realizada em casa e a segunda em sala de aula), sendo esta a primeira etapa. O somatório de ambas as etapas **equivale a 50% da nota final do curso**.

Nesta etapa a meta é implementar um jogo do gênero *Pac-Man*[1] seguindo as especificações descritas neste documento e no *template* de código disponibilizado. *Pac-Man* é um jogo single-player no qual o jogador controla um personagem que se move sobre um mapa bidimensional. O objetivo do jogo é obter a maior pontuação possível sem colidir com os fantasmas do jogo.

[1] https://en.wikipedia.org/wiki/Pac-Man



Descrição do trabalho

Neste trabalho, você deverá implementar um jogo do gênero *Pac-Man*. O programa deverá rodar no terminal, assim como todas as outras atividades realizadas em sala de aula. De maneira geral, o programa irá iniciar pela linha de comando (terminal) e irá ler o estado inicial do jogo, que consiste na definição do mapa, a posição inicial do personagem e dos fantasmas. Com o jogo inicializado, serão executados movimentos dados pelo usuário na entrada padrão. Durante a execução, o programa apresentará os estados parciais na saída padrão até o jogo terminar, hora em que ele apresentará o resultado final e salvará arquivos do jogo contendo outros resultados.

Funcionamento detalhado do programa

A execução do programa será feita através da linha de comando (i.e., pelo cmd, console, ou terminal) e permitirá a passagem de parâmetros. As funcionalidades a serem realizadas pelo programa são descritas a seguir: inicializar jogo, realizar jogo, gerar resumo de resultado, gerar estatísticas, gerar ranking, gerar trilha e tratamento de túneis. A descrição dos parâmetros de inicialização, da exibição do estado atual do programa, assim como, das operações a serem realizadas pelo programa, são apresentadas em detalhes a seguir.

Inicializar jogo

Para garantir o correto funcionamento do programa, o usuário deverá informar, ao executar o programa pela linha de comando, o caminho do diretório contendo os arquivos a serem processados (exemplo assumindo um programa compilado para o executável trab1, `./trab1 /maquinadoprofessor/diretoriodeteste`). Considere um caminho com tamanho máximo de `1000` caracteres. O programa deverá verificar a presença desse parâmetro informado na linha de comando. Caso o usuário tenha esquecido de informar o nome do diretório, o programa deverá exibir uma mensagem de erro (`ERRO: O diretório de`

`arquivos de configuração nao foi informado`) e finalizar sua execução. Dentro do diretório de teste, espera-se encontrar o arquivo de definição do mapa do jogo chamado `mapa.txt` (alguns exemplos serão fornecidos junto com esta especificação). O programa deverá ler o arquivo e preparar o ambiente de jogo. Caso o programa não consiga ler o arquivo (por exemplo, porque ele não existe), ele deverá ser finalizado e imprimir uma mensagem informando que não conseguiu ler o arquivo (nesta mensagem deve conter o caminho e nome do arquivo que ele tentou ler, por exemplo, `ERRO: nao foi possível ler o arquivo /minhapasta/arquivo.txt`). **Todas as saídas em forma de arquivo** do trabalho devem ser escritas em uma pasta com nome `saida` que, caso não exista, deve ser criada dentro do próprio diretório de teste fornecido.

O arquivo `mapa.txt` definirá todos os parâmetros do mapa: número máximo de movimentos e o conteúdo de cada célula, ou seja, o que deve estar em cada posição. O número máximo de movimentos define um limite de movimentos que o jogador pode executar sem que acarrete um *game over*. Um mapa é definido como um arranjo de `N` linhas contendo `M` colunas, semelhante a uma matriz, com tamanho variável, que deve ser lido dinamicamente. Cada posição dessa matriz (denominada célula) pode ser de diferentes tipos como descrito na tabela abaixo.

Tipo de célula	Representação	Em caso de colisão
Vazio	(espaço em branco)	Nada acontece
Parede	#	O personagem não se move
Comida	*	Ganha +1 ponto
Posição inicial do jogador	>	–
Fantasma	B, P, I, C	Fim de jogo
Túnel	@	Move para outro lado do túnel

Um exemplo desse arquivo de definição do mapa é ilustrado a seguir:

`mapa.txt`

```
20
#######
#*    *#
#   * B#
#   ##*#
#>     #
#######
```

O exemplo anterior define um mapa com 6 linhas e 7 colunas, limite de 20 movimentos, paredes (`#`) e quatro comidas (`*`) espalhadas pelo mapa. A posição inicial do personagem (`>`) está na quinta linha e segunda coluna. Um fantasma (`B`) está na linha 3 e coluna e 6. Repare que o arquivo de mapa não precisa ter todos os fantasmas. ~~Por fim, o caractere `F` representa o fina da matriz inicial do jogo~~. Um mapa sempre será delimitado por paredes (`#`).

Como resultado do passo de *inicializar* o jogo, o programa deve gerar o arquivo `inicializacao.txt`. Esse arquivo deve conter o mapa lido. Depois da representação do mapa, deve haver a linha de texto "`Pac-Man comecara o jogo na linha x e coluna y`", substituindo `x` pela linha e `y` pela coluna da posição de onde o personagem foi definida no arquivo `mapa.txt`. Um exemplo desse arquivo é mostrado a seguir.

`inicializacao.txt`

```
#######
#*    *#
#   * B#
#   ##*#
#>     #
#######
Pac-Man comecara o jogo na linha 5 e coluna 2
```

Realizar jogo

Uma vez preparado o jogo, as jogadas poderão começar a ser processadas. Existem três situações que fazem com que o jogo termine: (1) quando não houver mais comidas pelo mapa; (2) o Pac-Man morrer, ou seja, quando colidir com um fantasma; (3) o limite de movimentos é atingido. As jogadas deverão ser lidas da entrada padrão de dados, permitindo, dessa forma, um redirecionamento a partir de arquivo (em outras palavras, é possível usar o teclado diretamente ou redirecionamento via terminal utilizando, por exemplo, `./trab1 < comandos`).

O fluxo de jogada deve ser executada da seguinte forma:
(1) A jogada será fornecida pelo usuário através de um caractere que define o sentido do movimento, são eles: `a` para ir para esquerda, `w` para cima, `s` para ir para baixo e `d` para ir para a direita.

(2) Após ler a jogada do jogador, ela deve ser processada. O movimento que o personagem fará depende da posição em que ele estava e do movimento que o jogador definiu. Com isso os fantasmas também se movimentam, eles seguem uma política de movimento simples se movendo em apenas uma direção (vertical ou horizontal) e com um sentido inicial especificado:

Fantasma	Direção	Sentido inicial
B	Horizontal	Para esquerda
P	Vertical	Para cima
I	Vertical	Para baixo
C	Horizontal	Para direita

Um fantasma, ao estar de encontro como uma parede, deve ter seu sentido alterado para o oposto. Não haverá casos com fantasma preso entre duas paredes.

(3) Após um movimento, o programa deverá exibir na saída padrão (ou seja, na tela) o estado atual do jogo (isto é, o mapa com o Pac-Man) e a pontuação até o momento de acordo com o padrão:

```
Estado do jogo apos o movimento 'x'
<estado do mapa>
Pontuação: p
```

Substituindo `x` pelo movimento feito (`a`, `w`, `s` ou `d`), `<estado do mapa>` pelo estado do mapa e `p` pela pontuação no momento. Por exemplo, considerando a entrada de mapa anterior e que o jogador se movimentou para a direita (`d`) a saída esperada será:

```
Estado do jogo apos o movimento 'd'
#######
#*   *#
#  * B#
#  ##*#
# >   #
#######
Pontuação: 0
```

Observação 1: se um fantasma estiver na mesma posição de uma comida, o fantasma é impresso ao invés da comida. Além disso, não haverá casos de teste com a possibilidade de fantasmas se cruzarem no caminho.
Observação 2: A pontuação é determinada pela quantidade de comida que o Pac-Man comeu. Cada comida vale 1 ponto e para comê-la, basta mover o Pac-Man até a sua posição.

(4) Caso após o movimento executado não haja mais nenhuma comida no mapa, o jogo encerra com a seguinte mensagem:

```
Voce venceu!
Pontuacao final: p
```

Caso ainda exista comida no mapa, exibe-se a mensagem a seguir:

```
Game over!
```

```
Pontuacao final: p
```

Em ambos os casos, deve-se substituir `p` pela pontuação final obtida pelo jogador. As condições de *game over* são: (1) o personagem colidir com algum dos fantasmas (e neste caso o estado do mapa a ser impresso não exibirá o Pac-Man) ou o número de movimentos ultrapassar o limite estabelecido no arquivo `mapa.txt`.

Para facilitar a implementação, será fornecido também, com cada caso de teste, um arquivo de movimentos (denominado `jogadas.txt`), que poderá ser redirecionado pela entrada padrão para gerar uma saída esperada. Isso evitará ter que digitar todas as jogadas na mão. Veja a seguir um exemplo do conteúdo de um arquivo de movimentos. Considere que os casos dados sempre serão suficientes para finalizar o jogo, com uma vitória ou uma derrota. Cabe ao aluno testar outros casos.

`jogadas.txt`

```
w
w
w
d
d
d
d
d
s
s
w
a
a
```

Gerar resumo de resultado

O programa deverá também salvar na pasta de saída do caso de teste em questão, um arquivo, denominado `resumo.txt`, contendo o resumo do resultado do jogo. O arquivo de resumo deverá conter uma descrição do que houve em cada movimento onde houve alguma variação significativa: pegou comida, colidiu com a parede ou colidiu com um fantasma. Cada linha deve descrever a alteração causada por um movimento, sendo que cada alteração possui um padrão específico de saída, como mostrado a seguir.

No caso de um movimento que resulta na coleta de uma comida:

```
Movimento x (y) pegou comida
```

No caso de um movimento que resulta na colisão com a parede (em que o Pac-Man não muda de posição):

```
Movimento x (y) colidiu com a parede
```

No caso de um movimento que resulta no fim de um jogo por encostar em um fantasma:

```
Movimento x (y) fim de jogo por enconstar em um fantasma
```

Em todos os casos, os valores de `x` e `y` devem ser substituídos pelo número do movimento e pelo caractere que o representa, respectivamente. Um exemplo completo de um arquivo de resumo válido é ilustrado a seguir:

`resumo.txt`

```
Movimento 3 (w) pegou comida
Movimento 7 (d) pegou comida
Movimento 8 (d) colidiu na parede
Movimento 10 (s) pegou comida
Movimento 13 (a) pegou comida
```

Gerar ranking

O programa deverá também salvar na pasta de saída do caso de teste em questão, um arquivo, denominado `ranking.txt`, contendo um ranking de melhores movimentos junto de características desses movimentos ao fim do jogo. O melhor movimento (`a`, `w`, `s` ou `d`) é aquele que mais resultou em pegar comida. Em caso de empate usa-se o movimento que menos colidiu com a parede. Como segundo critério de desempate temos o movimento utilizado mais vezes. Se nesse último critério também houver empate, usa-se a ordem alfabética do caractere representando o movimento. O arquivo de ranking deverá apresentar cada linha, o movimento (`x`), o número de comidas pegas com esse movimento (`#1`), o número de colisões com parede (`#2`) e o número de movimentos do tipo realizado (`#3`), seguindo essa mesma ordem.

```
x1,#1,#2,#3
x2,#1,#2,#3
x3,#1,#2,#3
x4,#1,#2,#3
```

Um exemplo válido arquivo é exemplificado a seguir:

`ranking.txt`

```
w,1,0,4
a,1,0,2
s,1,0,2
d,1,1,5
```

Neste exemplo, o movimento `w` foi responsável pela coleta de uma comida, não colidiu com a parede e foi realizado 4. Observe que a ordenação respeita a sequência previamente descrita.

Gerar arquivo de estatísticas para análise

Ao final do jogo, o programa deverá também escrever, na pasta de saída do caso de teste em questão, um arquivo, denominado `estatisticas.txt`, contendo algumas estatísticas básicas sobre a partida. As informações que devem ser coletadas são: número de movimentos, número de movimentos sem pontuar, número de colisões com parede, número de movimentos para baixo, número de movimentos para cima, número de movimentos para a esquerda e número de movimentos para a direita. O padrão desse arquivo segue o formato:

```
Numero de movimentos: x1
Numero de movimentos sem pontuar: x2
Numero de colisoes com parede: x3
Numero de movimentos para baixo: x4
Numero de movimentos para cima: x5
Numero de movimentos para esquerda: x6
Numero de movimentos para direita: x7
```

Um exemplo válido arquivo é exemplificado a seguir:

estatisticas.txt

```
Numero de movimentos: 13
Numero de movimentos sem pontuar: 9
Numero de colisoes com parede: 1
Numero de movimentos para baixo: 2
Numero de movimentos para cima: 4
Numero de movimentos para esquerda: 2
Numero de movimentos para direita: 5
```

Gerar trilha

Ao final do jogo, o programa deverá também escrever, na pasta de saída do caso de teste em questão, um arquivo, denominado `trilha.txt`, contendo o mapa que descreve o número do movimento da última vez com que Pac-Man passou por cada célula. Isso será mostrado como uma matriz, onde o valor na linha `i` e coluna `j` denota o número do movimento em que o personagem passou pela linha `i` e coluna `j` do mapa pela última vez. A posição inicial é o movimento 0. Caso nunca tenha passado posição do mapa, exibi-se o carácter `#`. Um exemplo do arquivo de trilha é descrito a seguir:

trilha.txt

```
# # # # # # #
# 3 4 5 6 8 #
# 2 # 13 12 11 #
# 1 # # # 10 #
# 0 # # # # #
# # # # # # #
```

Tratamento túnel

Um mapa pode conter um túnel, representado por dois caracteres `@` posicionados em duas células diferentes no mapa. De forma simples, ao colidir com alguma dessa célula, o Pac-Man deve ser teleportado para a posição da outra célula. Por exemplo, considere o mapa a seguir que possui um túnel:

mapa.txt

```
20
#######
#* @ *#
#   * B#
#   ##*#
#> @   #
#######
```

Caso o Pac-Man colida em qualquer uma das pontas, ele deve ser teleportado para outra.

Observação: para o caso do arquivo de trilha, a última posição dele será escrita nas duas pontas do túnel.

Informações Gerais

Alguns arquivos de entrada e respectivos arquivos de saída serão fornecidos para o aluno. O aluno deverá utilizar tais arquivos para testes durante a implementação do trabalho. É de responsabilidade do aluno criar novos arquivos para testar outras possibilidades do programa e garantir seu correto funcionamento. O trabalho será corrigido usando, além dos arquivos dados, outros arquivos (específicos para a correção e não disponibilizados para os alunos) seguindo a formatação descrita neste documento. Em caso de dúvida, entre em contato com os professores e/ou monitores. O uso de arquivos com formatação diferente poderá acarretar em incompatibilidade durante a correção do trabalho e consequentemente na impossibilidade de correção do mesmo (sendo atribuído a nota zero). Portanto, siga estritamente o formato estabelecido.

Implementação e Verificação dos Resultados

A implementação deverá seguir o *template* de código disponibilizado no repositório do trabalho dentro da organização da disciplina no Github:

- Link para o repositório: **https://github.com/prog-II-ufes/template-TP-1-etapa-1**

A utilização do *template* segue os mesmos princípios dos exercícios disponibilizados ao longo da disciplina. Você deve utilizar os arquivos .h disponibilizados e, **obrigatoriamente**, implementar todas as funcionalidades requisitadas para construção destes TADs. Todavia, vocês são livres para implementar outros TADs e/ou bibliotecas que julgarem necessárias. Além disso, você já possui acesso ao *script* de correção automática de exercícios e trabalhos. Ele será utilizado para corrigir o seu trabalho. Desta forma, é extremamente recomendado que você avalie seu trabalho utilizando o *script* e os casos de teste disponibilizados.

Para fins de depuração, você também pode utilizar softwares de comparação de arquivos, como *diff* e o *Meld* (alternativa gráfica do *diff*). Diferenças na formatação poderão impossibilitar a comparação e consequentemente impossibilitar a correção do trabalho. O programa será considerado correto se gerar a saída esperada idêntica à fornecida com os casos de teste.

Pontuação e Processo de Correção

Pontuação: a primeira etapa do trabalho será pontuada de acordo com sua implementação e seguindo os pontos apresentados na tabela a seguir. A pontuação da tabela será utilizada para calcular a nota para cada caso de teste. A pontuação final será a média de pontos considerando todos os casos. Haverá o mesmo número de casos visíveis e ocultos (que serão divulgados durante a segunda etapa do trabalho em sala).

É importante observar que os pontos descritos na tabela não são independentes entre si, isto é, alguns itens são pré-requisitos para obtenção da pontuação dos outros. Por exemplo, gerar o arquivo de estatísticas depende de realizar o jogo corretamente. Observe também que modularização e gerenciamento de memória são pré-requisitos do trabalho. Se você não os cumprir, será descontado uma porcentagem da pontuação obtida (ver próxima sub-seção). Outros erros gerais de funcionamento ou lógica serão tratados de maneira individual.

Item	Quesitos	Ponto
Inicializar jogo	Ler o arquivo do mapa Gerar corretamente o arquivo de Inicialização (`inicializacao.txt`)	1
Realizar jogo	Receber corretamente as entradas do jogador Mostrar as informações do jogo como especificado Mostrar o resultado do jogo (A saída padrão deve ser idêntica a saída padrão esperada)	3
Gerar resumo do resultado	Gerar arquivo com o resumo dos resultados (resumo.txt)	1.5
Gerar ranking	Gerar arquivo com a ordenação solicitada (ranking.txt)	1.5
Gerar estatísticas	Gerar corretamente o arquivo com as estatísticas (estatisticas.txt)	1.5
Gerar trilha	Gerar corretamente o arquivo com a trilha (trilha.txt)	1.5

Descontos: a pontuação obtida de acordo com a tabela anterior será utilizada para calcular a nota final do seu trabalho. O gerenciamento de memória e modularização são itens obrigatórios. Sendo assim, não basta apenas acertar a saída, é necessário seguir os *templates* de código disponibilizados e gerenciar corretamente o uso da memória. Logo, considere que seu trabalho seja capaz de gerar todos os arquivos corretamente e obtenha pontuação igual a 10. A partir desta pontuação a sua nota final será calculada da seguinte maneira:

(1) primeiramente será verificado se você seguiu corretamente o *template* de código fornecido. Na tabela a seguir são descritos todos templates disponibilizados e o peso de cada um deles:

Item	Descrição	Peso
tPacMan.h	TAD relacionado ao PacMan	25%
tMapa.h	TAD relacionado ao mapa do jogo	25%
tPosicao.h	TAD relacionado a posição dentro do jogo	5%
tTunel.h	TAD relacionado a um túnel dentro do jogo	5%
tMovimento.h	TAD relacionado a movimentação dentro do jogo	5%

main.c + restante	Função principal do programa + restante do código necessário	35%

Se você implementou todos os *templates* corretamente não haverá nenhum desconto na sua pontuação nesta fase. Porém, se você deixar de implementar algum dos *templates*, sua pontuação será descontada de acordo com o peso do *template*. Por exemplo, considerando que sua pontuação foi 10, se você falhar na implementação do `tMapa.h`, sua nota será descontada em 30%. Portanto, a sua pontuação após esta fase será 7. Esse valor é transmitido para a próxima fase, que verifica o gerenciamento de memória.

(2) após a verificação dos *templates*, será verificado o gerenciamento de memória através do Valgrind. Caso o seu programa apresente vazamento de memória e/ou erro de memória será **descontado 30%** da pontuação que chegar até essa fase. Portanto, considerando o exemplo anterior que a pontuação foi 7, se o programa gerencia a memória corretamente, essa será a nota final. Porém, caso ocorra problema, será descontado mais 30%. Neste caso, a pontuação desta fase e, consequentemente, a nota final da etapa 1 para o seu trabalho será 4,9.

Sendo assim, **é de suma importância que seu trabalho siga a risca os *templates* e gerencie corretamente a memória** (lembre-se, esses são os principais objetivos desta disciplina). Todo esse processo de pontuação será realizado por meio do *script* de correção que vocês terão acesso. Observe que a compilação da pasta completo (dentro de resultados) no *script* não é utilizado para computar a pontuação. Porém, pode ser utilizado para depuração de erros no seu programa.

Processo de correção do trabalho: como consta no plano de ensino da disciplina, os trabalhos práticos são divididos em duas etapas. A primeira etapa é o desenvolvimento do projeto seguindo as regras e descrições apresentadas neste documento. A segunda etapa consiste na correção, atualização e/ou incremento de funcionalidades implementadas na etapa anterior. Esta última etapa é realizada em sala de aula de acordo com cronograma da disciplina. A nota final do trabalho é a média entre as duas etapas, ou seja, cada etapa vale 50% da nota.

É importante ressaltar que ao iniciar a etapa 2 vocês terão acesso à nota e aos casos de teste ocultos da etapa 1. Portanto, vocês poderão corrigir possíveis erros cometidos durante o desenvolvimento da etapa 1.

Regras Gerais

Para todas as etapas do trabalho, considere as seguintes regras:

- Trabalhos entregues após o prazo **não** serão corrigidos (recebendo a nota zero)
- O trabalho deverá ser feito **individualmente** e pelo próprio aluno, isto é, o aluno deverá necessariamente conhecer e dominar **todos** os trechos de código criados. Cada aluno deverá trabalhar independente dos outros, não sendo permitido a cópia ou compartilhamento de código.
    - Haverá verificação automatizada de plágio. Trabalhos identificados como iguais, em termos de programação (por exemplo, mudar nomes de variáveis e funções entre outros não faz dois trabalhos serem diferentes), **serão penalizados com a nota zero e poderão ser submetidos para penalidades**

adicionais em instâncias superiores. Isso também inclui a pessoa que forneceu o trabalho, sendo portanto, de sua obrigação a proteção de seu trabalho contra cópias ilícitas. Proteja seu trabalho e não esqueça cópias do seu código nas máquinas de uso comum.

- Não utilizar caracteres especiais (como acentos) em lugar nenhum do trabalho
- Para dados multidimensionais no qual o tamanho é desconhecido, é obrigatório o uso de alocação dinâmica de memória. Liberar a memória alocada é de sua responsabilidade.
    - O script de correção utiliza o Valgrind. Haverá descontos significativos da nota para erros e vazamentos de memória apontados pela ferramenta
- Modularização e organização são fundamentais na construção do código. Você é obrigado a implementar os TADs solicitados via o *template* dos `.h`. Todavia, isso não impede a criação de outros.
- Todos os trabalhos serão corrigidos utilizando o sistema operacional Linux. **Garanta que seu código funcione nos computadores do Labgrad**. Qualquer discrepância de resultado na correção por conta de Sistema Operacional, os computadores do Labgrad serão utilizados para solucionar o problema.

Prazo e submissão

Todas as informações referentes a prazo de entrega e forma de submissão estão disponíveis na atividade **Trabalho Prático 1 - Etapa 1** no *classroom* da disciplina.

Considerações finais

Esse documento descreve de maneira geral as regras de implementação do trabalho. É de responsabilidade do aluno garantir que o programa funcione de maneira correta e amigável com o usuário. Qualquer alteração nas regras do trabalho será comunicada em sala e no portal do aluno. É responsabilidade do aluno frequentar as aulas e se manter atualizado em relação às especificações do trabalho. Caso seja notada qualquer tipo de inconsistência nos arquivos de testes disponibilizados, comunique imediatamente ao professor para que ela seja corrigida e reenviada para os alunos.