Trabalho Prático 2: Andando na Física

Algoritmos e Estruturas de Dados III – 2016/2
Entrega: 01/11/2016

1 Introdução

O *ICEx*, prédio de ciéncias exatas da *UFMG* possui vários departamentos, *(Matemática, Física, Computação e Estatística)*. Dentre esses, o mais fácil de se perder é o departamento da Física. Vários mitos correm pelos alunos mais antigos de pessoas que se perderam nos corredores deste e nunca mais foram encontradas. Vinícius (Vinizinho para os intimos) nunca acreditou nesses mitos e foi tirar a prova, e o óbvio ocorreu: *ele se perdeu*. O objetivo deste trabalho é salvá-lo e trazê-lo de volta ao *DCC*. Você tem em mãos um mapa com a localização do Vinícius e do *DCC* e, como é aluno de Computação, quer trazê-lo de volta pelo caminho mais curto.[1]

Para dificultar o problema, o mapa da física é bem problemático: existem diversos obstáculos e portas trancadas pelo departamento (é possível passar por dentro de sala dos professores para cortar caminho e eles não ligam pra isso). Para destrancar as portas, é necessário ter a chave dessa porta, mas depois de destrancada não é mais necessária a chave para passar por essa porta. Para piorar, Vinícius consegue carregar apenas um número limitado de chaves de portas I, pois ele se confunde se andar com mais que I chaves. As chaves das portas estão espalhadas pela Física. Ao achar uma chave, é possível pegá-la ou não.

Além disso, encontramos na física **"buracos de minhoca"**. Os buracos de minhoca são pontos que vão de um ponto a outro do mapa que podem ou não ser adjacentes. Ao passar na posição que tenha um buraco de minhoca, a pessoa automaticamente passará pro outro ponto do mapa e o buraco de minhoca desaparece.

O mapa é um grid N x M. Dada a posição inicial de Vinícius (i, j), ele consegue se mover para as posições $(i+1, j), (i-1, j), (i, j+1), (i, j-1)$, desde que não tenha nenhum obstáculo nessas posições e elas estejam dentro do grid. Uma porta trancada funciona como um obstáculo caso o Vinicíus

[1] Trabalho baseado no problem F da Maratona Mineira de Programação de 2016

não possua a chave que destranque ela e funciona como uma posição livre caso possua a chave. Para pegar a chave, obviamente, é necessário ir até a posição da chave.

Mover para uma posição adjacente gasta exatamente uma unidade de tempo. Abrir portas ou pegar chaves e andar nos buracos de minhoca não gastam tempo (Vinicius é muito habilidoso!). Seu objetivo é informar com quantas unidades de tempo Vinicíus pode sair do departamento da Física ou, se não há como escapar, você terá que dar a má noticia dizendo que é *impossível* escapar.

Esse problema tem solução usando grafos, sendo uma delas por força bruta, porém não a solução mais inteligente. Alunos que implementarem uma solução mais adequada para o problema será recompensado com um acréscimo de 20% do valor do trabalho.

2 Entrada e saída

Entrada A entrada começa com uma linha contendo três valores N,M e T $0 < N, M < 9$, $0 < T < 4$ indicando respectivamente a dimensão do mapa da física (M *e* N) e a quantidade de chaves que Vinícius consegue carregar. Cada uma das N linhas seguintes contém M caracteres cada, a representação do mapa da física. No mapa, o caractere '.' indica posições livres. O caractere '#' indica posições com obstáculos. Os caracteres minúsculos 'c','d','h' e 's' indicam chaves. Há no máximo uma chave de cada tipo de porta por mapa. Os caracteres maíusculos 'C', 'D', 'H' e 'S' indicam portas trancadas. Uma porta de caractere 'C' só pode ser destrancada por uma chave 'c', uma porta 'D' por uma chave 'd' e assim por diante. O caractere 'V' indica a posição inicial do Vinícius. O caractere 'E' indica a saída do departamento de física. Os buracos de minhoca são representados por um valor X e Y $(0 < X, Y < 9)$, X representando a linha e Y a coluna sem espaço entre eles indicando o destino final do buraco de minhoca, sendo que no grid a posição $(0, 0)$ fica no canto inferior esquerdo. O caractere 'E' indica a saída da física.

Para cada entrada é garantido que $|V| = 1, |E| = 1, |XY| < 5, |C| + |D| + |H| + |S| < 9$, onde $|x|$ indica quantas vezes o caractere 'x' aparece na entrada.

Exemplos de entrada

Entrada 1

```
1 8 3
V s S S S S S E
```

Entrada 2

```
4  4  1
V  d  D  .
#  #  .  .
#  #  00  c
E  .  .  C
```

Uma melhor visualização do exemplo de Entrada 2 é observado na Figura 1.

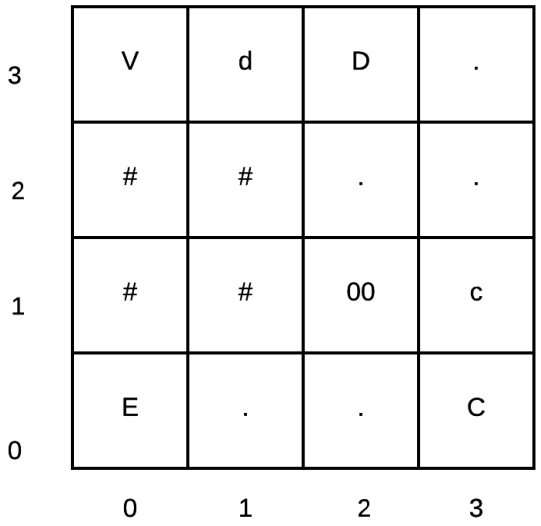

Figura 1: Representação da Entrada 2

Saída Para cada entrada imprima uma linha com o tempo mínimo para fugir da física. Caso seja impossível fugir da física imprima -1.

Exemplo de saída Saída 1

```
7
```

Saida 2

```
4
```

3 O que deve ser entregue

Deverá ser submetido um arquivo **.zip** contendo somente uma pasta chamada `tp2` e dentro desta deverá ter: (i) Documentação **em formato PDF** e (ii) Implementação.

Documentação Poderá ter no máximo 10 páginas e deverá seguir tanto os critérios de avaliação discutidos na Seção 4.1, bem como as diretrizes sobre a elaboração de documentações disponibilizadas no *moodle*. Além disso, a documentação deverá conter análise experimental validando as complexidades de tempo e espaço.

Implementação Código fonte do seu TP (*.c* e *.h*), com solução baseada em **grafos**.

Makefile Inclua um *makefile* na submissão que permita compilar o trabalho. É obrigatório o uso das *flags*: **-Wall -Wextra -Werror -std=c99 -pedantic** na compilação.

4 Avaliação

Eis uma lista **não exaustiva** dos critérios de avaliação que serão utilizados.

4.1 Documentação

Introdução Inclua uma breve explicação do problema que está sendo resolvido no seu trabalho e um resumo da sua solução.

Solução do Problema Você deve descrever a solução do problema de maneira clara e precisa, detalhando e justificando os algoritmos e estruturas de dados utilizados. Para tal, artifícios como pseudo-códigos, exemplos ou diagramas podem ser úteis. Note que documentar uma solução não é o mesmo que documentar seu código. **Não** é necessário incluir trechos de código em sua documentação nem mostrar detalhes de sua implementação, exceto quando estes influenciem o seu algoritmo principal, o que se torna interessante.

Análise de Complexidade Inclua uma análise de complexidade de tempo e espaço dos principais algoritmos e estrutura de dados utilizados. Cada complexidade apresentada deverá ser devidamente **justificada** para que seja aceita.

Avaliação Experimental Sua documentação deve incluir os resultados de experimentos que avaliem o tempo de execução de seu código em função de características da entrada. Cabe a você gerar entradas para esses experimentos. Por exemplo: se esse trabalho fosse sobre ordenação, seria interessante mostrar como o tempo de execução de cada algoritmo varia quando o número de items a serem ordenados aumenta. Para tal, um gráfico mostrando o tempo de execução em função do tamanho da entrada pode ser interessante. Você também deve interpretar os resultados obtidos. Comente sobre cada gráfico ou tabela que você apresentar mostrando o que é possível concluir a partir dele.

4.2 Implementação

Linguagem & Ambiente O seu programa deverá ser implementado na linguagem **C** e poderá fazer uso de funções da biblioteca padrão da linguagem. Trabalhos que utilizem qualquer outra linguagem de programação e/ou que façam uso de outras bibliotecas que não a padrão serão zerados. Além disso, certifique-se que seu código compile e funcione corretamente nas máquinas **Linux** dos laboratórios do DCC.

Casos de teste A sua implementação passará por um processo de correção automatizado, portanto, o formato da saída do seu programa deve ser idêntico aquele descrito nessa especificação. Saídas com qualquer divergência serão consideradas erradas, mesmo que as divergências sejam *whitespaces*. e.g. espaços, *tabs*, quebras de linha, etc. Para auxilia-lo na depuração do seu código, será fornecido um pequeno, **não-exaustivo**, conjunto de entradas e suas respectivas saídas. É seu dever certificar-se que seu código funciona corretamente para qualquer entrada válida.

Alocação Dinâmica Algoritmos e estruturas de dados deverão fazer uso de memória alocada dinamicamente (`malloc()` ou `calloc()`). Certifique-se que seu programa utiliza essas regiões de memória corretamente, pois os monitores penalizarão implementações que realizam *out-of-bounds access* e que tenham vazamento de memória (não desalocar memoria dinâmica).
A alocação dinâmica deverá fazer uso das funções malloc() ou calloc() da

biblioteca padrão C, bem como liberar tudo o que for alocado utilizando free(), para gerenciar o uso da memória. **DICA:** Utilize `valgrind` antes de submeter o seu TP.

Qualidade do código Seu código também será avaliado no quesito de legibilidade, dando atenção, porém não limitando-se, aos seguintes items: (i) **INDENTAÇÃO**; (ii) nomes de variável e função descritivos e claros; (iii) Modularização adequada; (iv) Comentários dentro de funções, explicando o que certos trechos mais complicados fazem; (v) Comentários fora de funções, explicando, em alto-nível, o que as funções mais importantes fazem; (vi) funções concisas que desempenham somente uma tarefa; (vii) **Proibido uso de variáveis globais**.

Atrasos Trabalhos poderão ser entregues após o prazo estabelecido, porém sujeitos a uma penalização regida pela seguinte fórmula:

$$\Delta_p = \frac{2^{d-1}}{0.32}\,\%$$

Por exemplo, se a nota dada pelo corretor for 70 e você entregou o TP com 4 dias corridos de atraso, sua penalização será de $\Delta_p = 25\%$ e, portanto, a sua nota final será: $N_f = 70 \cdot (1 - \Delta_p) = 52.2$. Note que a penalização é exponencial e 6 dias de atraso resultam em uma penalização de 100%.

5 Consideração Final

Assim como em todos os trabalhos dessa disciplina é estritamente proibida a copia parcial ou integra de códigos, seja da internet ou de colegas. Utilizaremos o algoritmo *MOSS* para detecção de plágio em trabalhos, seja honesto. Você não aprende nada copiando código de terceiros nem pedindo a outra pessoa que faça o trabalho por você. Se a cópia for detectada, sua nota será zerada e os professores serão informados para que as devidas providências sejam tomadas.

HAVE FUN!!!