As misteriosas chaves do reino perdido

Mariela Pontes Cordeiro[*], Lucas Dellatorre de Freitas[†]
Faculdade de Informática — PUCRS

6 de junho de 2022

Resumo

Este relatório descreve nossa solução para o problema proposto no segundo trabalho da disciplina de Algoritimos e Estruturas de Dados II, no qual deve-se desenvolver um algoritmo que percorre os espaços livres de um mapa coletando chaves e abrindo portas.

Palavras-chaves: Algoritmos. Estrutura de Dados. Jogos.

Introdução

```
##################################
#................#...............#
#................#...............#
#................B...............#
#.....1..........#...............#
#..........c....#...............#
#................#...............#
##################.............#
#..............................#
#..............................#
#..............................#
#..............................#
###########b################C#####
#a................#............#
#.................#............#
#...............2..#............#
#.................A............#
#.................#............#
####################............#
#.................#............#
#.....3...........####A##########
#.................#............#
#.................#............#
##################################
```

A fim solucionar o problema, foi solicitado o desenvolvimento de um jogo que funciona através do percurso de diferentes jogadores em um mapa, onde há espaços livres, rochedos por onde não se pode passar, chaves que podem ser coletadas para abrirem suas respectivas portas, e que deve terminar quando todas as posições alcançáveis por cada jogador forem visitadas.

Foram disponibilizados cinco mapas, os quais podem ser exemplificados pela figura ao lado, com os jogadores representados por números de um até nove em suas posições iniciais. Em cada mapa, os pontos são os espaços por onde se pode passar, as cerquilhas são os rochedos e as chaves e as portas têm a mesma letra como indentificador, mas as portas estão em maiúsculo.

As regras do jogo são simples: não há como movimentar-se na diagonal, podem haver chaves e portas repetidas, uma chave pode ser usada várias vezes e uma porta a fechada não conta como posição explorada, mas uma aberta sim. Quando objetivo final é alcançado, deve-se retornar quantos pontos no cenário foram visitados por cada jogador.

[*]mariela.c@edu.pucrs.br
[†]lucas.dellatorre@edu.pucrs.br

Estruturando o mapa

Para iniciar o desenvolvimento, deve-se em primeiro lugar escolher a linguagem a ser utilizada e a estrutura de dados que irá conter o mapa ao longo do percurso; decidimos que iríamos usar a linguagem Python e, em um primeiro momento, começamos a estudar como grafos são manipulados através dela. Ao longo da pesquisa, concordamos que uma busca de profundidade seria adequada para manipular cada jogador, mas que talvez não fosse necessário utilizar um grafo e, então, encontramos o conceito de algoritmos de inundação, o que nos fez acreditar que poderíamos implementar algo parecido, criando um código que percorre uma matriz - o mapa, nesse caso - e realiza uma busca de profundidade mantendo o histórico nas quatro posições vizinhas, considerando que foi especificado não ser permitido o movimento diagonal no jogo, embora esse pudesse ser implementado com a nossa solução.

Basicamente, há dois vetores 2D, um que contém o mapa em si e outro que armazena apenas zeros e uns, sendo um o inteiro que indica que uma posição já foi explorada; essas duas estruturas principais permitem que tenhamos as coordenadas (x, y) específicas de cada caractere, logo seus vizinhos sempre serão (x + 1, y), (x - 1, y), (x, y + 1), (x, y - 1). Ainda, especificamos por parâmetro que os pontos são os caracteres que podem ser visitados, e os que não podem ser visitados são tratados diferentemente de acordo com o seu conteúdo. Assim, iniciamos com uma função que insere um caso de teste em uma matriz e com outra função que a atravessa mostrando no terminal seu conteúdo para fins de teste.

Encontrando os jogadores em suas posições iniciais

Cada jogador deve jogar separadamente, então desenvolvemos uma função que antes de iniciar o jogo, percorre o mapa buscando os números contidos nele e guardando-os em um dicionário como chave e com o valor sendo sua coordenada, uma tupla. Assim, quando o iterador for iniciar o jogo naquele cenário específico, já saberá em qual posição cada jogador deve iniciar a sua trajetória. A função em pseudocódigo encontra-se a seguir:

```
1  procedimento encontraJogadores(List lista)
2
3    jogadores ← Conjunto
4
5    para x em (LENGTH(lista) −1)
6      para y em LENGTH(lista[0])
7        se lista[x][y] éum número
8          numero ← lista[x][y]: int
9          jogadores[numero] = x , y
10       se y = (LENGTH(lista[0]) −1)
11         sair
12
13   retorna jogadores
```

Lidando com as chaves e portas

Nossa solução utiliza um conjunto (chavesDasPortas) para guardar as chaves que o usuário for encontrando no caminho, pois nesse tipo de coleção a ordem das letras não importa, não há letras repetidas, o que nesse caso é adequado pois uma chave pode abrir qualquer porta correspondente, não sendo necessário armazenar repetidas, e a maior vantagem é que o método

para conferir se uma certa chave está contida nele é muito otimizado; para guardar as portas, inicialmente havíamos utilizado um dicionário parecido com o dos jogadores, uma chave que contém uma coordenada como valor, pois o jogador pode acabar encontrando uma porta antes da chave que pode abrí-la, portanto, caso isso ocorresse, realizaríamos uma chamada recursiva da função passando a tupla (x, y) de onde a porta se encontra para acessar o conteúdo dentro dela, mas testando os casos percebemos que ele pode encontrar mais de uma porta com o mesmo nome antes da sua respectiva chave, então mudamos os valores do dicionário para uma lista que pode conter inúmeras tuplas como valor.

Dessa maneira, ao passo que o jogador for se movimentando, ele pode voltar para as portas que podem ser abertas com a chave correspondente encontrada através de um *loop* na função que chama a si mesma para cada coordenada do mapa que contém a porta específica. Essas duas coleções, além da matriz que marca as posições visitadas, são reiniciadas para cada jogador; o dicionário de jogadores é reiniciado e recebe novos valores após a leitura e armazenamento de um novo mapa.

Função principal

Abaixo iremos mostrar a função principal e após explicaremos mais detalhadamente como ela funciona.

```
1  procedimento inundacao(int x, int y, String posLivre, Conjunto chavesDasPortas, Dic portas)
2
3    se a coordenada não for válida ou não estiver no mapa do jogo
4      retorna
5
6    se matrizVisitados[x][y] = 1
7      retorna
8
9    global posVisitadas
10
11   posAtual ← mapaDoJogo[x][y]
12
13   se posAtual ≠ posLivre
14       se posAtual for uma letra
15         se posAtual for minúscula
16           chaveDasPortas.add(posAtual)
17           se posAtual em maiúsculo estiver em portas
18             matrizVisitados[x][y] ← 1
19             para valores de portas.get(posAtual em maiúsculo):
20                 inundacao(valor[0], valor[1], posLivre, chaveDasPortas, portas)
21         se posAtual for maiúscula e ainda não tiver uma chave correspondente
22             se ((x,y)) não já estiver em portas[posAtual]
23                 portas[posAtual].append((x,y))
24             retorna
25         se posAtual for maiúscula e tiver uma chave correspondente
26             continua
27       se posAtual for um número
28         continua
29       se for uma cerquilha
30         retorna
```

```
31
32    matrizVisitados[x][y] ← 1
33    posVisitadas ← posVisitadas + 1
34
35    inundacao(x+1 , y , posLivre , chaveDasPortas , portas)
36    inundacao(x−1 , y , posLivre , chaveDasPortas , portas)
37    inundacao(x , y+1 , posLivre , chaveDasPortas , portas)
38    inundacao(x , y−1 , posLivre , chaveDasPortas , portas)
39
40    retorna posVisitadas
```

O método inicia recebendo o x e o y da posição a ser verificada, o ponto, que irá representar as posições livres do mapa, e as duas coleções com chaves e portas. No início, instanciamos a variável global que será o nosso contador de posições visitadas por cada jogador (posVisitadas), verificamos se a coordenada é válida para aquele mapa e se ela já foi visitada, caso algum desses dois casos for verdade, ela não continua.

Se a posição for um espaço livre, ela é marcada como visitada, o contador acresce uma unidade e seus vizinhos são chamados. Caso contrário, há três casos principais diferentes:

- Se for uma cerquilha, a função retorna para onde estava antes dessa posição e tenta visitar os outros vizinhos existentes, sem marcar como visitado porque no jogo os rochedos não podem ser marcados como visitados;
- Se for um número, ele apenas continua e marca como visitado; achamos importantes mencionar que com essa verificação, a posição inicial do jogador, um número, é marcada como visitada e conta no valor final retornado.
- Caso seja uma letra, há três opções:
    1. Se for uma chave, ela é adicionada no conjunto de chaves e após verifica-se se já foi encontrada uma porta que a mesma pode abrir, se a resposta for verdade, duas atitudes são tomadas: a posição da chave deve ser marcada como visitada, pelo motivo de que um mapa pode conter, por exemplo, a porta já visitada em uma posição vizinha da chave, o que resultaria em um *loop* infinito e o código quebraria, e um *for* é feito sobre o dicionário de portas, voltando para abrir uma ou mais posições onde essas portas se encontram, como comentado na seção acima;
    2. Se for uma porta que o jogador tem chave para abrir, ela é aberta, marcada como visitada e o jogador continua sua exploração;
    3. Se for uma porta nunca visitada e que não pode ser aberta, guardamos suas coodernadas e seu nome no dicionário de portas e retornamos, sem marcar como visitada, para que no caso anterior possamos voltar nela sem que a condição de se já foi visitada impeça o processo.

Ao final, após o jogador ter visitado todas as posições que pode, a função retorna o que foi pedido no problema, o número de casas visitadas.

Resultados

O enunciado do problema nos informa os resultados pro caso de teste de exemplo, o mesmo que mostramos no início deste relatório, são eles:

- Jogador 1: pode visitar 96 casas;

- Jogador 2: pode visitar 541 casas;
- Jogador 3: pode visitar 72 casas;

Nosso algoritmo encontra os mesmos resultados para o caso de teste em pauta; já para os outros mapas disponibilizados, foi necessário aumentar o limite de recursão para 15000, pois a linguagem Python, por padrão, permite uma profundidade máxima de 1000 na recursividade e nossas saídas encontram-se a seguir:

	Caso 5	Caso 6	Caso 7	Caso 8	Caso 9	Caso 10
P1	57	57	81	601	453	2733
P2	91	954	701	105	1713	81
P3	33	954	285	1077	93	2481
P4	1454	669	213	1305	4700	2301
P5	1454	33	2954	129	561	3393
P6	113	527	2453	225	117	21
P7	349	129	2954	849	153	5817
P8	45	105	21	1305	513	6379
P9	1454	145	2373	1209	33	28130

Conclusões

Em um quesito de complexidade de tempo do algoritmo, se analisarmos de uma formal geral, visitamos cada posição no máximo duas vezes, e como há N x M posições, o trabalho é feito em 2*N*M. Esse resultado pode ser escrito em notação assintótica como O(N*M).

Enfim, considerando as diferentes soluções possíveis de serem encontradas, acreditamos que nosso código recursivo que utiliza busca em profundidade e vetores multidimensionais atende o problema proposto de uma maneira efetiva.