Relatório – As Misteriosas Chaves do Reino Perdido

Giovane Bianchi Milani[1]

[1]Escola Politécnica - PUCRS

13 de junho de 2022

Resumo. Este artigo descreve uma alternativa para solucionar o problema proposto na disciplina de Algoritmos e Estrutura de Dados II, que consiste em descobrir quantas casas um Player pode explorar em um determinado cenário de um jogo. A modelagem do problema e o processo da solução é apresentado, juntamente com o pseudocódigo dos algoritmos mais importantes. Além disso, proponho um caminho para dar continuidade ao estudo do problema. Por fim, serão apresentados os resultados obtidos nos seis casos de teste disponibilizados pelo professor.

1. Introdução

No contexto de que fomos contratados, por uma grande empresa de games, para testar uma ideia de um novo jogo, chamado *"As Misteriosas Chaves do Reino Perdido"*, o problema que investigaremos pode ser descrito da seguinte forma: desenvolver a ideia básica do jogo, em que existe um cenário onde os *Players* andam livremente, realizando *Quests* e visitando quartos. Esses quartos podem conter objetos, tais como poções, pergaminhos, espelhos e palavras mágicas escritas nas paredes. O objetivo final, é descobrir quantas casas um *Player* pode percorrer dependendo de o lugar onde iniciar dentro do cenário.

O jogo tem algumas regras já definidas, para a movimentação do jogador e para a leitura do cenário.

1 - Os *Players* podem se mover pelo cenário, desde que não seja na diagonal;

2 - O caractere "#" representa um rochedo, por onde não é possível passar;

3 - O caractere "." representa um espaço livre;

4 - As letras minúsculas de “a” até “z”, são espaços livres que contém chaves e podem ser coletadas pelo *Player;*

5 - As letras maiúsculas de “A” até “Z”, são portas fechadas, que só podem ser abertas com a chave adequada;

6 - As chaves encontradas abrem qualquer porta de mesmo nome, por exemplo, a chave “a” abre a porta “A”, e podem ser usadas várias vezes;

7 - As portas fechadas não contam como casa explorada, mas as abertas sim;

8 - Pode haver chaves e portas repetidas pelo cenário;

9 - Os números de 1 até 9, são pontos de início para os *Players*.

A empresa ainda disponibiliza um exemplo de cenário, para fins de teste, veja abaixo:

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

Resultados para o cenário do exemplo:

-*Player* 1 pode explorar 96 casas;

-*Player* 2 pode explorar 541 casas;

-*Player* 3 pode explorar 72 casas.

Dadas as regras e o cenário do jogo, deve-se ler o cenário e para todos os pontos de início, e descobrir quantas casas o *Player* pode percorrer. Para isso, vamos analisar, a seguir, a característica de uma solução, apresentando as ideias e estruturas usadas, bem como uma análise do desempenho e os resultados obtidos após a implementação.

2. Primeira Solução

Após analisar o problema e o exemplo de cenário, decidimos seguir com uma ideia simples. Pensamos em organizar o cenário em uma estrutura de grafo não dirigido, em que os vértices representam os espaços livres e as arestas onde o *Player* pode se mover. Cada vértice armazenaria se possui uma chave, se é um ponto de início ou uma porta fechada, além de guardar a referência dos seus vizinhos, que são outros vértices que ele se conecta. Os rochedos não são armazenados nessa estrutura. Segue um exemplo:

Com a entrada:

```
############
#.....#....#
#.1...#..3.#
#######....#
#.....#....#
#.a...A....#
#...2.#....#
############
```

Obteríamos:

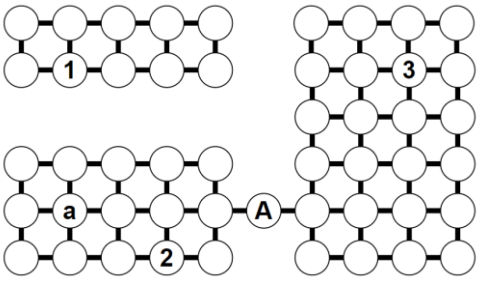


Figura 1. Exemplo da estrutura usada

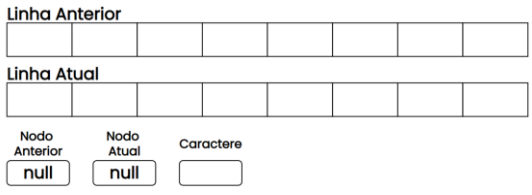
Com o objetivo de ler o arquivo de entrada e organizar na estrutura descrita, devemos gerar *vértices* para todos os espaços livres e conectá-los com os que estão ao lado dele. Para tal, poderíamos iterar sobre cada linha do arquivo e para cada espaço livre que encontrarmos, um novo *vértice* é adicionado. Quando isso acontece, precisamos verificar se esse *vértice* precisa se conectar com alguém, ou seja, olhamos se foi gerado um *vértice* anterior a ele, se sim, então podemos conectar os dois. A mesma coisa é feita para os nodos “acima” dele, olhamos para a linha anterior, e se um *vértice* foi gerado na mesma posição em relação a linha, então conectamos estes dois. Este algoritmo pode ser representado pelo pseudocódigo abaixo:

```
funcao criaGrafo(arquivo) {
    linhaAnterior = [];
    para cada (linha) do arquivo {
        nodoAnterior = null;
        linhaAtual = [];
        para cada (caractere) da linha {
            se caractere == '#' {
                nodoAnterior = null;
                linhaAtual.add(null);
            } senão {
                nodo = novo nodo com o caractere
                linhaAtual.add(nodo);
                se nodoAnterior != null {
                    conecta(nodo, nodoAnterior);
                }
                nodoAnterior = nodo;
            }
        }
        conectaNodosMesmoIndice(linhaAnterior, linhaAtual);
        linhaAnterior = linhaAtual
    }
}
```

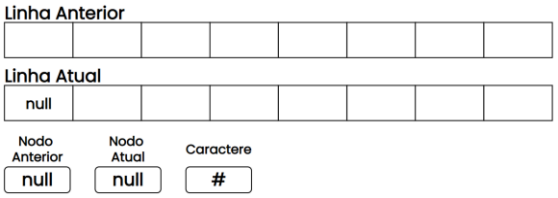
Por exemplo, se fossemos ler a entrada a seguir, o algoritmo teria os seguintes passos:

Ao ler:
```
#.1.#a.#
###A#..#
```

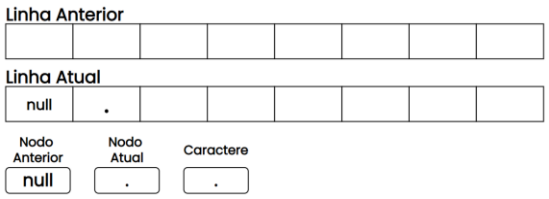
1º: começa lendo a primeira linha do arquivo:



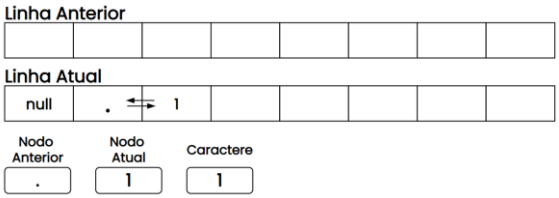
2º: lê o primeiro caractere da linha, como é um rochedo, nenhum *nodo* é adicionado, e a primeira posição da linha atual é preenchida com uma referência *nula:*



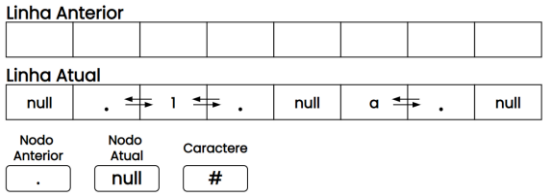
3º: quando lê o segundo caractere, que é um espaço livre, adiciona um *nodo* e tenta conectar com o anterior, como ele é uma referência *nula*, não terá conexão:



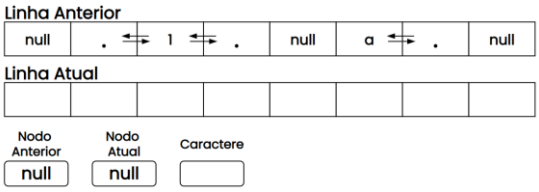
4º: ao ler o terceiro, que também é um espaço livre, adiciona mais um *nodo*, e tenta conectar com o anterior, dessa vez, o anterior é um outro *nodo*, então a conexão será feita:



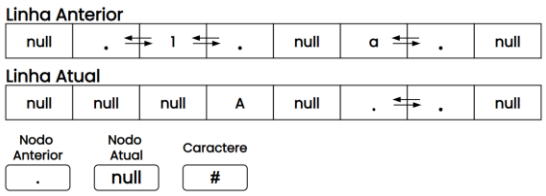
5º: continua lendo os caracteres da linha, ao chegar no final dela, o estado do algoritmo seria o seguinte:



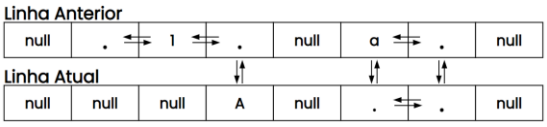
6º: antes de passar para a próxima linha, tenta conectar os *nodos* da linha atual com a anterior, como a anterior não tem nada, vai apenas copiar o estado da linha já iterada para a linha anterior:



7º: itera sobre a segunda linha, gerando os *nodos* quando precisa e conectando quando possível, ao chegar no final, o estado seria o seguinte:



8º: verifica se a linha anterior possui referências, como dessa vez tem, fará a conexão entre todos os *nodos* de mesmo índice das linhas anterior e atual:



Com o cenário já montado em forma de grafo, podemos prosseguir para o próximo passo do problema, que é descobrir quantas casas cada *Player* pode percorrer. Para isso, seguiremos com uma ideia baseada no caminhamento em largura, poderia ser sem problemas em profundidade ou qualquer outro tipo, escolhemos o em largura por achar de mais fácil compreensão. O caminhamento ainda tem uns detalhes a mais, pois queremos coletar chaves e abrir portas durante a execução.

A ideia é ter uma lista de *nodos que vou visitar*, e a partir de qualquer ponto do cenário, os seus vizinhos vão sendo adicionados sucessivamente nessa lista, até que todos os *nodos* tenham sido visitados e a lista estará vazia. O detalhe a mais, é que existem *nodos* que poderemos visitar somente se uma condição tenha sido satisfeita, que é o caso das *portas e chaves*, dado um *nodo* que é uma porta, poderemos acessá-lo e consequentemente acessar seus vizinhos, somente se tivermos a *chave* para ele.

Levando em conta que podemos achar uma *porta* antes da sua *chave* e que existem *portas* repetidas que usam a mesma *chave*. O algoritmo que pensamos, possui uma lista de *chaves* e uma de *portas*, que vão sendo preenchidas durante o caminhamento. A lista de *chaves*, representa as *chaves* que já achamos, e nunca vão ser tiradas dali. Já a lista de *portas*, representa as *portas* trancadas, que encontramos no caminho, e ficam lá até que elas possam ser abertas. No momento em que coletamos alguma *chave*, que abre alguma *porta* que está na nossa lista, essa *porta* é adicionada na lista de *nodos que vou visitar* e poderemos acessar seus vizinhos.

Também, é importante salientar duas coisas. A primeira, é que ao saber da existência de um *nodo*, ou seja, ao adicionar um *nodo* na lista de *nodos que vou visitar*, ele é marcado, pois não queremos visitar o mesmo *nodo* duas vezes. A segunda, é que contagem das casas é feita somente quando visitamos algum *nodo*. Dessa forma, não corremos o risco de encontrar uma *porta trancada* e contar como casa visitada.

O algoritmo descrito nos parágrafos anteriores, pode ser representado pelo pseudocódigo abaixo:

```
funcao caminha(pontoInicio) {
    casas = 0;
    chaves = [];
    portas = [];
    nodosParaCaminhar = [];
    pontoInicio.marca();
    nodosParaCaminhar.add(pontoInicio);

    enquanto nodosParaCaminhar não estiver vazia {
        nodo = nodosParaCaminhar.primeiro();
        se nodo é porta {
            se tem chave para porta {
                nodos = visita(nodo);
                casas++;
                nodosParaCaminhar.addTodos(nodos);
            } senão {
                portas.add(nodo)
            }
        } senão {
            se nodo é chave { chaves.add(nodo.caractere) }
            nodos = visita(nodo)
            casas++;
            nodosParaCaminhar.addTodos(nodos);
        }
        para cada (porta) em portas {
            se tem chave para porta {
                portas.remove(porta);
                nodosParaCaminhar.add(porta);
            }
        }
    }
    retorna casas;
}
```

Com isso, podemos caminhar a partir de cada ponto de início e descobrir quantas casas cada *Player* pode percorrer.

Após essa ideia ter sido implementada, apresentou resultados de eficiência satisfatórios. Apesar disso, percebemos que a forma como os cenários são construídos, apresentam um número grande de *vértices*, o que prejudica a eficiência do caminhamento, com um simples teste descobrimos que o caso número 10, tem 3111877 *vértices*, por exemplo.

3. Próximos passos

Nesta seção, discutiremos um problema que achamos merecer ser estudado, baseado no que aprendemos no desenvolvimento deste artigo, mas que não temos certeza de como resolvê-lo. Proporemos uma linha de pensamento, para uma ideia que ainda está bem abstrata, mas que tenta melhorar a eficiência do caminhamento no grafo.

O problema que vamos relatar, é sobre a estrutura como estamos representando o cenário do jogo, com muitos *vértices* que poderiam ser comprimidos. Olhe o exemplo abaixo para entender:

Este é um exemplo de cenário gerado pelo nosso algoritmo:

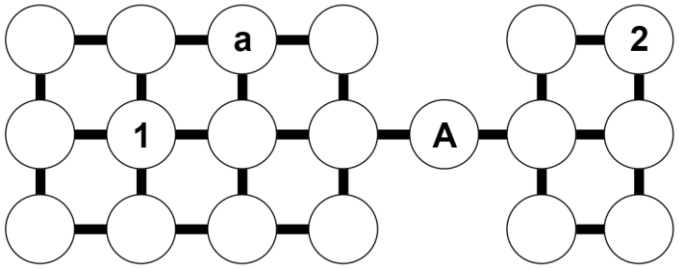


Figura 2. Estrutura gerada pelo algoritmo solução 1

Agora olhando para a representação abaixo, podemos perceber, que saindo de qualquer *vértice* é possível chegar em qualquer outro dentro do grupo em que o próprio está contido.

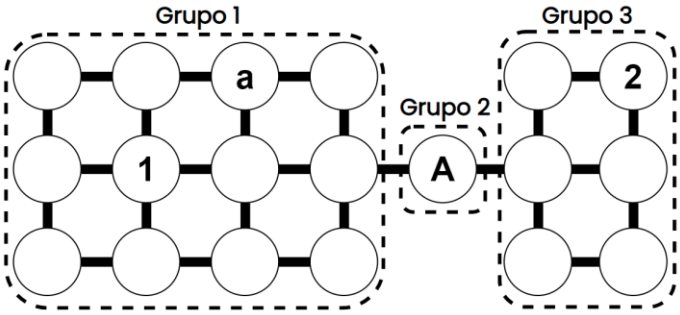


Figura 3. Representação do agrupamento de vértices

Primeiramente, vamos explicar o que são esses grupos e como sabemos formá-los. Um grupo é formado por todos os *vértices* que para serem visitados, não dependem de nenhuma condição ser satisfeita, ou seja, todos os *vértices* que são acessíveis sem a dependência de uma *chave*. Todos esses *vértices* poderiam ser convertidos em um só, que guarda os dados de todos eles (número de casas, chaves e *Players*).

Os *vértices* que são portas, não ficariam agrupados a nenhum outro, pois esses sim dependem de *chaves* para serem acessados, e por isso, não temos a certeza de que ele será visitado durante um caminhamento. No caso do exemplo mostrado anteriormente, não conseguimos ter a certeza de que os *Players* 1 e 2 vão conseguir coletar a *chave* para ter acesso a *porta A,* e por esse motivo, as *portas* nunca podem ser agrupadas.

A razão para fazer esse agrupamento de *vértices,* é para realizar menos visitas durante o caminhamento, olhe para a estrutura em que faríamos o caminhamento:

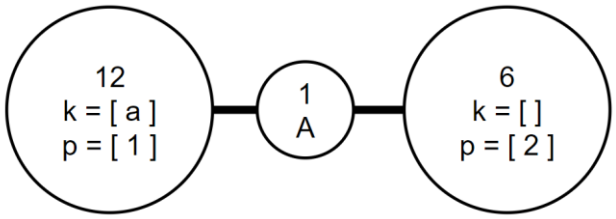


Figura 4. Exemplo da nova estrutura

Por exemplo, começando pelo *Player 1*, na primeira visita já conseguimos saber que ele caminha 12 *casas* e coleta a *chave "a",* com mais 2 visitas, podemos descobrir que ele abre a porta A e acessa os seus vizinhos, caminhando mais 7 *casas*. Portanto, com 3 visitas, obtivemos o resultado para o *Player 1*, comparando com o algoritmo da solução 1, precisaríamos de 19 visitas para chegar no resultado.

Dito isso, sabemos que essa ideia melhoraria a eficiência do caminhamento, mas não temos certeza de como funcionaria um algoritmo para construir essa estrutura.

4. Resultados

Resultados obtidos através da implementação da primeira solução:

	Caso 5	Caso 6	Caso 7	Caso 8	Caso 9	Caso 10
P1	113	608	167	2006	309	5817
P2	113	321	33	269	685	9523
P3	1065	189	2020	1603	4611	45
P4	1065	815	2483	873	2433	8611
P5	1065	21	571	105	4579	21
P6	21	45	2762	377	3653	2133
P7	77	9	115	1395	1437	847
P8	9	153	2762	913	825	1089
P9	57	33	525	1125	749	2505
Tempo Construção	0,01 s	0,02 s	0,12 s	0,50 s	1,52 s	4,3 s
Tempo Caminhamentos	0,01 s	0,01 s	0,05 s	0,03 s	0,12 s	0,49 s

5. Conclusão

A nossa abordagem inicial, apesar de simples, apresentou resultados eficientes para resolver o problema proposto. O desenvolvimento dessa alternativa, nos mostrou um possível caminho para dar continuidade ao estudo desse problema. Apresentamos isso como uma ideia bastante abstrata, mas foi até onde conseguimos chegar.

Concluímos dizendo que, a realização deste trabalho influenciou muito no nosso processo de entendimento dos *Grafos,* que é uma estrutura de dados que é muito importante e tem diversas aplicações no mundo real.