Trabalho 2 - Algoritmo e Estrutura de Dados II

Leticia Brasil Flores*
Escola Politécnica — PUCRS

16 de junho de 2022

Resumo

Este relatório descreve a implementação da solução para o segundo trabalho proposto na disciplina de Algoritmo e Estrutura de Dados II, que consiste em desenvolver um programa que realiza a contagem da quantidade de casas que cada player pode explorar em um cenário específico. Para o seu desenvolvimento, foi atilizada a linguagem Python, e para a representação da estrutura do labirinto foi utilizado Grafo não dirigido, lista de adjacências e caminhamento em profundidade.

Introdução

O objetivo do presente trabalho é desenvolver um programa que irá realizar a contagem da quantidade de casas que cada player pode explorar saindo do seu ponto de início em determinado cenário. Existem algumas regras para movimentação que devem ser seguidas, como por exemplo, os players (1-9) não se movem em diagonal, e cada cenário possui portas (A-Z) que só podem ser ultrapassadas pelo player se a respectiva chave tiver sido coletada (a-z).

Após compreensão da atividade proposta, foram elencados quatro desafios principais para serem abordados:

(1) leitura do arquivo de teste e armazenamento em uma estrutura de matriz de caracteres a fim de extrair uma lista com as arestas;

(2) implementar uma lista de adjacências, considerando grafo não dirigido;

(3) realizar o caminhamento pelo cenário coletando chaves e informações de portas disponíveis guardando em uma nova lista; e

(4) implementar a lógica que fará a contagem de casas exploradas.

Para solucionar os desafios, foi utilizada a linguagem Python. A implementação do programa está contida no arquivo t2.py em anexo.

A partir do desenvolvimento do programa, os resultados obtidos com a execução de cada caso de teste serão analisados.

Primeira solução

O desenvolvimento do programa foi iniciado com a leitura do arquivo e armazenamento em uma matriz de caracteres, considerando cada item da matriz uma posição no labirinto, nomeada Nodo, que possui os seguintes parâmetros: tipo; se foi visitado; se está disponível; e uma lista de nodos vizinhos, ou seja, as arestas.

Foi utilizada uma lista para armazenar os nodos e cada uma de suas arestas. Foi realizada a iteração em cada elemento da matriz, verificando o nodo atual e os nodos adjacentes. Se o tipo de

*leticia.flores@edu.pucrs.br

caractere do nodo verificado for diferente de “#” (parede do labirinto), deve-se adicionar as arestas do nodo atual e do próximo à lista de adjacências, considerando que se trata de um grafo não direcionado e arestas não valoradas. Ao final desse processo, foi obtido como resultado uma lista com as seguintes características:

Nodo . Visitado: False Disponível: True Total de arestas: 4 Arestas: [., ., ., .]
Nodo . Visitado: False Disponível: True Total de arestas: 4 Arestas: [., A, ., .]
Nodo A Visitado: False Disponível: False Total de arestas: 2 Arestas: [., .]
Nodo . Visitado: False Disponível: True Total de arestas: 4 Arestas: [A, ., ., .]

A partir da lista de adjacências, foi necessário implementar a função de caminhamento, que deve iniciar a partir de uma posição específica, neste caso, a posição de cada jogador. Para isso, foi utilizado Busca em profundidade (BFS), que irá explorar todo o grafo, marcando as posições já visitadas e realizando chamadas recursivas, visitando todos os vértices não marcados e seus adjacentes. Assim, é possível encontrar todos os Nodos conectados a um nodo de origem. Adicionalmente, na função de caminhamento é necessário incluir a verificação se o nodo visitado é uma chave, se sim, precisamos adicionar a lista de chaves do jogador, ou se é uma porta, é necessário verificar se existe a respectiva chave na lista do jogador, a fim de alterar a posição da porta como disponível.

A função para caminhamento está descrita a seguir:

```
1  procedimento Caminhamento(posição)
2    posição visitado = TRUE
3
4      se posição estiver disponível então
5          adiciona posição na lista Visitados
6
7          se tipo da posição for igual a letra minúscula então
8              adiciona posição na lista chavesJogador
9          fim
10
11         para i em arestas da posição faça
12             se i não for visitado então
13             caminhamento(i)
14         fim
15     fim
16     se posição não estiver disponível então
17         se posição é letra maiúscula então
18             se posição minúscula está na lista chavesJogador então
19                 posição disponível = TRUE
20                 caminhamento(posição)
21             fim
22         fim
23     fim
24 fim
```

Com a execução de um primeiro teste, utilizando o caso de teste do enunciado do trabalho e passando as posições de cada jogador para a função caminhamento(), obteve-se o resultado aproximado. Sendo que a contagem de posições que cada jogador pode explorar, referente ao item 4 de atividades elencadas anteriormente, pode ser facilmente obtida a partir do tamanho da lista "visitados".

Segunda solução

Visando corrigir o resultado obtido na primeira implementação, foi identificado que alguns parâmetros não estavam sendo zerados ao chamar a função para um novo jogador. Logo, foi criada a função reinicia, que será responsável por zerar todos os parâmetros a cada novo jogador. A função pode ser descrita da seguinte forma:

```
1  procedimento reinicia()
2      para posição em lista de adjacentes faça
3          posição visitado = FALSE
4          para p em lista de arestas da posição faça
5              p visitado = FALSE
6          fim
7          se posição é letra maiúscula então
8              posição disponível = FALSE
9          fim
10     fim
11     limpa lista de visitados
12     limpa lista de chaves do jogador
13 fim
```

Após a implementação correta da limpeza dos parâmetros, obtve-se o resultado esperado para o referido caso de teste.

Player: 1 pode explorar 96 casinhas
Player: 2 pode explorar 541 casinhas
Player: 3 pode explorar 72 casinhas

A partir da obtenção desse resultado, foi necessário ajustar outro ponto: a chamada de cada jogador. Para tanto, foi implementada uma lógica que busca a posição de cada jogador e realiza as chamadas de funções para possibilitar a execução dos demais casos de testes. Nesse sentido, foi implementado o seguinte código:

```
1  para cada linha da matriz faça
2      para cada elemento da linha da matriz faça
3              se elemento for igual a número
4                  chama a função reinicia ()
5                  chama a função caminhamento(elemento)
6                  imprimi no terminal quantas casas cada player pode explorar
7              fim
8      fim
9  fim
```

Após isso, foi possível executar o programa para os demais casos de testes, porém, a partir do caso de teste caso08.txt foi percebido um erro de profundidade máxima de recursão excedida: *"RecursionError: maximum recursion depth exceeded while calling a Python object"*. Após pesquisa, foi constatado que essa exceção é levantada quando o interpretador detecta que a profundidade máxima de recursão foi excedida, e no caso, o limite padrão é 1000. Como alternativa para contornar essa exceção, foi ampliado o limite de recursão, utilizando a método *setrecursionlimit()* do módulo *sys*, todavia, como será apresentado na seção de resultados, não foi possível executar todos os casos de teste.

Resultados

A seguir são apresentados os resultados obtidos com a execução dos casos de testes. Para cada caso, o resultado é exibido no terminal informando a quantidade de casas que podem ser exploradas por cada jogador. Para os casos de teste 6, 7 e 8, foram obtidos os seguintes resultados:

Caso06
Player: 4 pode explorar 669 casinhas
Player: 1 pode explorar 57 casinhas
Player: 2 pode explorar 954 casinhas
Player: 3 pode explorar 954 casinhas
Player: 5 pode explorar 33 casinhas
Player: 6 pode explorar 527 casinhas
Player: 7 pode explorar 129 casinhas
Player: 9 pode explorar 145 casinhas
Player: 8 pode explorar 105 casinhas

Caso07
Player: 2 pode explorar 707 casinhas
Player: 3 pode explorar 285 casinhas
Player: 1 pode explorar 81 casinhas
Player: 4 pode explorar 213 casinhas
Player: 6 pode explorar 2453 casinhas
Player: 5 pode explorar 2954 casinhas
Player: 7 pode explorar 2954 casinhas
Player: 8 pode explorar 21 casinhas
Player: 9 pode explorar 1433 casinhas

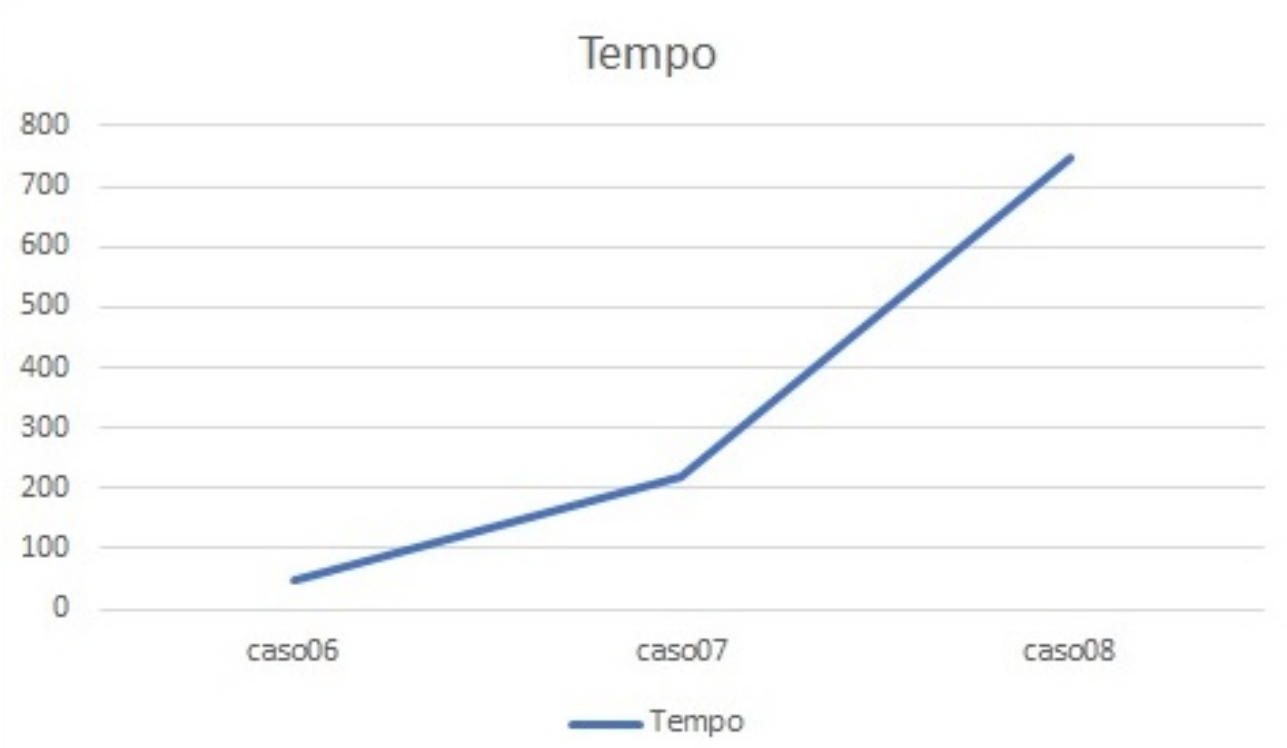
Caso08
Player: 2 pode explorar 105 casinhas
Player: 1 pode explorar 601 casinhas
Player: 6 pode explorar 225 casinhas
Player: 5 pode explorar 129 casinhas
Player: 3 pode explorar 1077 casinhas
Player: 7 pode explorar 849 casinhas
Player: 4 pode explorar 1305 casinhas
Player: 9 pode explorar 1209 casinhas
Player: 8 pode explorar 1161 casinhas

A execução dos casos de teste 9 e 10 não retornaram o resultado esperado, e também não retornaram uma mensagem de erro na execução. Até o envio deste trabalho não foi possível identificar a falha para correção.

Conclusões

Mesmo faltando a execução dos dois últimos casos de teste, foi possível compreender a aplicação e o funcionamento da estrutura de grafos para o problema apresentado. Em relação a complexidade da Busca em profundidade, conforme visto, é linear, sendo O(vertices + arestas). A fim de verificar a complexidade no programa, com base nos resultados da implementação, foi realizada a execução contando o tempo para cada teste, e elaborado o gráfico que consta na Figura 1 com as informações da evolução de cada caso de teste, representando a quantidade de caracteres, e o tempo de execução.

Figura 1: Tempo de execução



Referências

[1] A Biblioteca Padrão do Python: "**Exceções embutidas**". Disponível em: <https://docs.python.org/pt-br/3/library/exceptions.htmlRecursionError> Acesso em 14 Jun. 2022.

[2] A Biblioteca Padrão do Python: "**Parâmetros e funções específicas do sistema**". Disponível em: <https://docs.python.org/pt-br/3/library/sys.htmlsys.getrecursionlimit> Acesso em 14 Jun. 2022.