Estruturas de Dados e Algoritmos II

Licenciatura em Engenharia Informática

Trabalho Prático 2

Palm Island Neighbours

Grupo: g206

Miguel Pombeiro, 57829 | Miguel Rocha, 58501

Departamento de Informática
Universidade de Évora
Abril 2025

Índice

1 Introdução

O problema proposto em "Palm Island Neighbours" descreve uma situação em que se pretende analisar a topologia de ilhas artificiais no Dubai. Estas ilhas têm a forma de palmeiras, o que faz com que dois dos seus habitantes, mesmo quando geograficamente perto, possam ter de percorrer grandes distancias para visitarem a casa um do outro. De forma a fazer esta análise, é pedido que seja calculada a distancia relativa, em número de habitantes que é preciso passar, do maior caminho mais curto entre quaisquer dois habitantes da ilha.

Para melhor analisar a topologia da ilha, é possível modelar os vários habitantes e ligações como um grafo conexo, não orientado e acíclico - uma árvore. Neste grafo, as casas dos habitantes são representadas por vértices, enquanto que as ligações entre as casas são as arestas. Assim, o objetivo final é encontrar a distância máxima, em número de conexões, entre quaisquer dois vértices, ou seja, o diâmetro da árvore.

2 Algoritmo

2.1 *Input*

Os *inputs* disponíveis para cada teste são:

- O número de habitantes da ilha (**n**).
- Um conjunto de $n-1$ ligações entre dois habitantes.

2.2 Grafo Construído

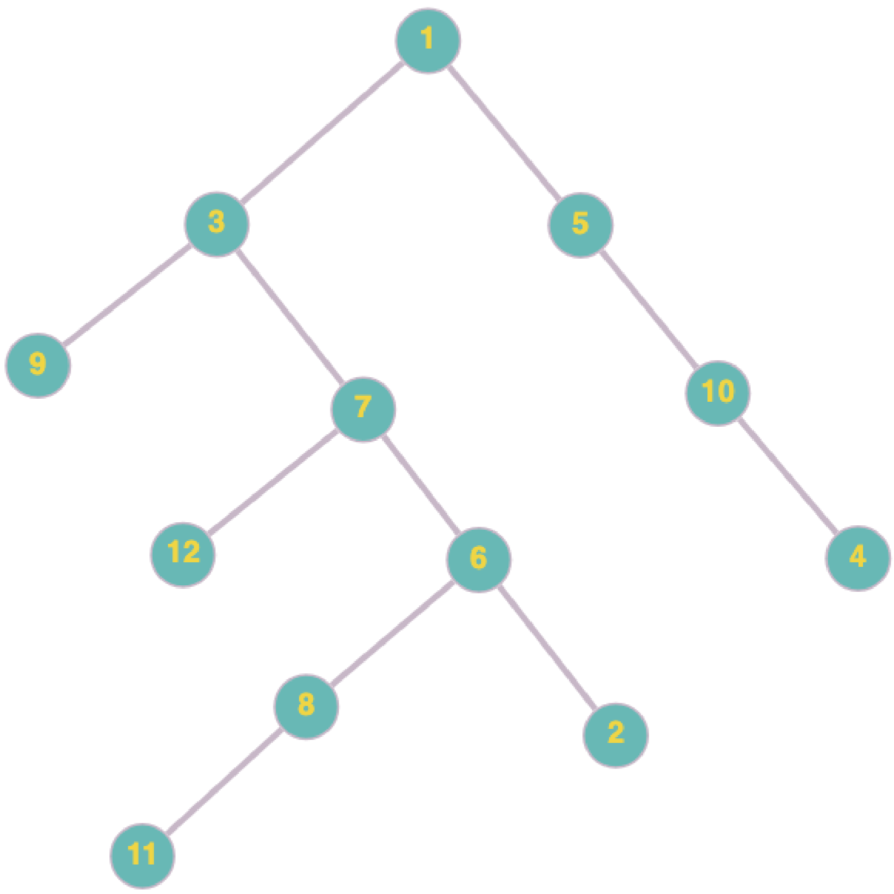


Figura 1: Grafo construído a partir do "Sample Input 1" do enunciado.

2.3 Descrição do Algoritmo

O algoritmo implementado para estudar a topologia das ilhas recorre a grafos para representar os habitantes e as suas vizinhanças. Desta forma, e com base no *input* dado, é necessário construir uma representação deste grafo. Uma vez que este se trata de um grafo não orientado e acíclico, basta utilizar uma representação em lista de adjacências, uma vez que são esperadas, no máximo, $V - 1$ arestas.

O algoritmo de pesquisa em largura, BFS (*Breath-First Search*), é uma técnica de travessia de um grafo que permite explorar todos os seus vértices nível a nível, ou seja, explora todos os vértices que se encontram a uma mesma distancia do vértice inicial, antes de explorar os vértices que se encontram a distancias superiores. Ao fazer a travessia desta forma, o BFS permite, sempre, encontrar o menor caminho entre dois nós, o que o torna apropriado para resolver este problema. Para utilizar este algoritmo são necessárias várias estruturas de dados:

- **Vetor de booleanos**, onde são registados os vértices que já foram visitados.
- **Fila de vértices**, que guarda os vértices a serem explorados, bem como a sua ordem.
- **Vetor de inteiros**, onde serão armazenadas as distancias do nó inicial a qualquer um dos outros nós. Utilizado para procurar a maior distância.

Desta forma, o BFS começa por explorar um dado vértice inicial, que é marcado como visitado. De seguida, são visitados todos os seus vértices vizinhos, que ainda não estão marcados como visitados. Ao serem visitados, estes vizinhos, são, também eles, marcados como visitados e colocados dentro da fila para serem, posteriormente, explorados. Este procedimento é repetido para todos os vértices que estão na fila, até que esta esteja vazia, ou seja, até que todos os vértices tenham sido explorados. A distância de cada vértice, que está a ser visitado, relativa ao vértice inicial é calculada com base na distância do vértice que está a ser explorado.

Numa primeira abordagem para resolver este problema, foi utilizada uma técnica *brute force*, que iria aplicar o algoritmo de BFS a cada um dos vértices do grafo. Assim, seria possível encontrar a distância máxima de cada um dos vértices a todos os restantes, sendo que a maior das distâncias encontrada representaria o *largest shortest path* do grafo. Contudo, devido às restrições temporais impostas para a resolução do problema, verificou-se que esta não era a solução ótima.

De forma a reduzir a complexidade dos algoritmos a implementar e, assim, respeitar os requisitos temporais impostos, foi explorada uma solução alternativa, também ela baseada no algoritmo de BFS. Esta solução baseia-se no facto de todos os grafos representativos deste problema serem árvores, o que permite concluir que ambos os nós que estão nos extremos do maior caminho da árvores, são folhas (apenas têm uma ligação).

Desta forma, aplicando uma BFS a partir de qualquer nó da árvore vai sempre resultar que o último nó visitado nesta pesquisa (maior distância ao nó inicial) será um dos extremos do seu diâmetro, uma vez que os nós são percorridos nível a nível. A partir do exemplo do enunciado, ao aplicar uma BFS a partir de um qualquer nó, o nó mais distante do inicial será sempre um dos nós que estão nos extremos do maior caminho da árvore (4 ou 11). Na Figura 2 foi aplicado o BFS a partir do nó 1, para encontrar o vértice que está mais distante deste.

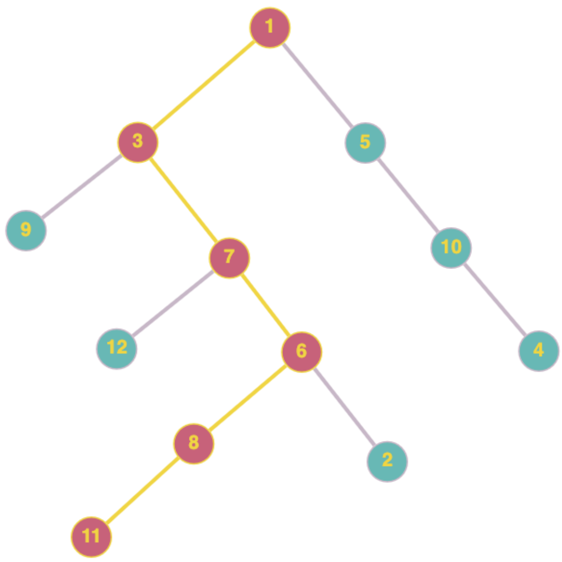
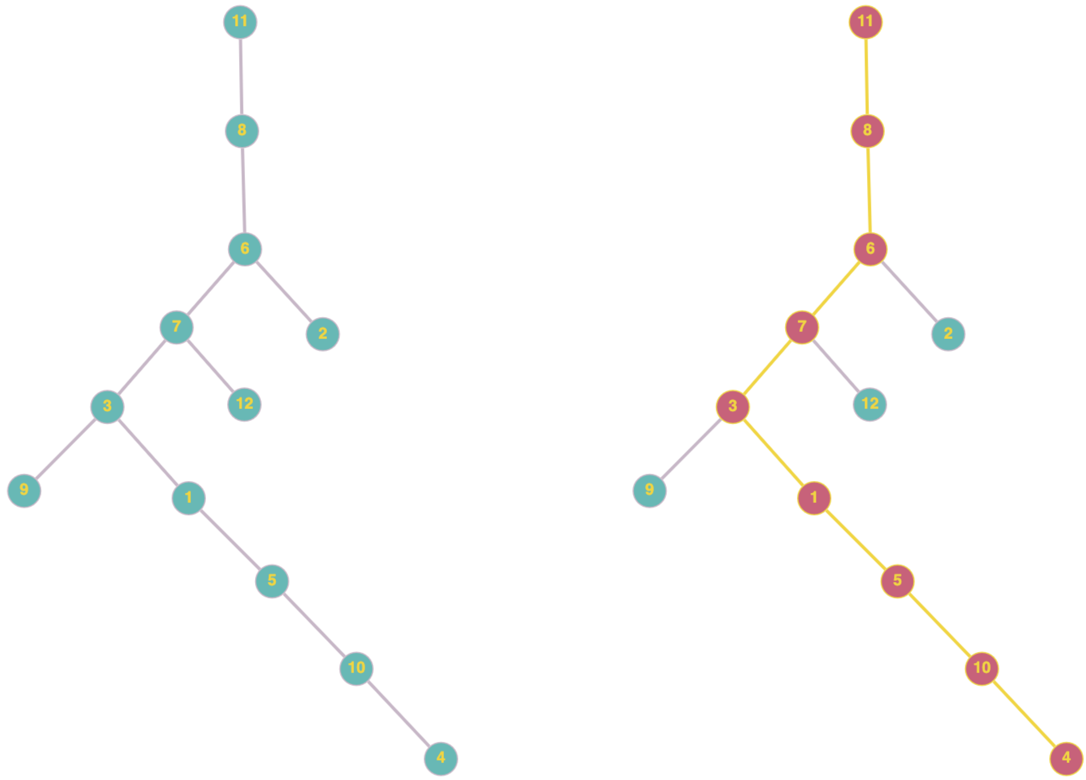


Figura 2: Encontrar o vértice mais distante do vértice 1 do grafo construído a partir do "Sample Input 1" do enunciado, através de BFS.

Depois de encontrado um dos nós dos extremos do diâmetro do grafo, é possível encontrar tanto o seu comprimento, como o outro extremo. Para isso, basta voltar a aplicar uma pesquisa em largura no grafo, a partir do extremo encontrado anteriormente. Novamente, tal é possível porque o último nó visitado pelo BFS será sempre aquele que está mais distante do nó inicial. Assim, voltando ao exemplo do enunciado, é possível tomar o nó 11 (extremo encontrado ao aplicar BFS a partir do nó 1) como raiz, Figura 3a, e aplicar o BFS a partir deste. No final, é possível concluir que o outro extremo do maior caminho da árvore será o nó 4 e que este terá comprimento oito, Figura 3b.



(a) (b)

Figura 3: Grafo (3a) e o seu diâmetro (3b) para o "Sample Input 1", com o vértice 11 como raiz.

2.4 Pseudocódigo

Para a resolução do problema proposto, foi necessário implementar dois algoritmos, sendo estes:

- O algoritmo de pesquisa em largura (BFS), cujo objetivo é achar as distâncias de todos os outros vértices relativamente a um vértice inicial.
- O algoritmo de procura do *largest smallest path* entre quaisquer dois habitantes. Para resolver este problema recorrendo a uma representação em árvore, é necessário fazer apenas duas procuras *BFS*. Contudo, se o grafo tivesse ciclos, seria necessário utilizar outra abordagem que fizesse uma procura *BFS* por cada vértice.

Algorithm 1 Algoritmo de procura do diâmetro de um grafo para *Palm Island Neighbours*

Require: G, um grafo com a sua lista de adjacências; *nNodes*, número de vértices de G
Ensure: G é um grafo conexo e acíclico (árvore)

```
startNode ← 1
maxDistance ← −∞
let distance[0...nNodes + 1] be a new array

distance ← BFS(G, startNode)
for i ← 1 to nNodes do
    if distance[i] > distance[startNode] then
        startNode ← i
    end if
end for

distance ← BFS(G, startNode)
for j ← 1 to nNodes do
    if distance[j] > maxDistance then
        maxDistance ← distance[j]
    end if
end for
return maxDistance
```

Algorithm 2 BFS(G, start) - Percurso em largura

Require: G, um grafo com a sua lista de adjacências; *nNodes*, número de vértices de G; *start*, vértice onde se inicia o percurso

```
let visited[0...nNodes + 1] be a new array
let distance[0...nNodes + 1] be a new array

for each vertex u in G.V - start do
    visited[u] ← false
end for

distance[start] ← 0
visited[start] ← true
let Q be a new EMPTY queue

ENQUEUE(Q, start)
while Q ! = EMPTY do
    u ← DEQUEUE(Q)
    for each vertex v in G.adj[u] do
        if !visited[v] then
            visited[v] ← true
            distance[v] ← distance[u] + 1
            ENQUEUE(Q, v)
        end if
    end for
end while
return distance
```

3 Complexidades

3.1 Temporal

Complexidade do BFS (Algorithm 2): Considera-se que as operações de criação de uma *queue* vazia, ENQUEUE e DEQUEUE, têm custo $\Theta(1)$. Todas as outras para além das mencionadas terão custo constante.

- O ciclo das linhas **4 - 6** tem custo $\Theta(V)$, sendo V o número de vértices do grafo G (variável *nNodes*).
- O ciclo das linhas **13 - 22** tem custo $\mathcal{O}(V + 2E)$, no seu pior caso. Onde E, é o número de arestas do grafo G. Como G, neste caso, é uma árvore, $E = V - 1$.
  - $\mathcal{O}(V)$ na linha **14**
  - O ciclo das linhas **15 - 21** é $\mathcal{O}(2E)$

Assim a complexidade deste algoritmo é $\mathcal{O}(V + 2E) = \mathcal{O}(V + E)$.

Complexidade do Algorithm 1, para encontrar o comprimento do *longest shortest path*:

Considerando **n** o número de habitantes. No algoritmo apresentado, apenas as operações seguintes fazem variar a complexidade temporal:

- Linha **5** tem custo O(n + e), sendo **e**, o número de arestas do grafo. Esta é a complexidade do algoritmo BFS descrita acima.
- Ciclo **6 - 10** é executado n vezes (variável i).
  - Todas as operações dentro deste ciclo tem custo constante.
- Linha **12** tem custo O(n + e), sendo **e**, o número de arestas do grafo. Esta é a complexidade do algoritmo BFS descrita acima.
- Ciclo **13 - 17** é executado n vezes (variável j).
  - Todas as operações dentro deste ciclo tem custo constante.

Para o ciclo das linhas **6 - 10**:

$$\sum_{i=1}^{n} 1 = n$$
$$= \mathcal{O}(n).$$

Para o ciclo das linhas **13 - 17**:

$$\sum_{j=1}^{n} 1 = n$$
$$= \mathcal{O}(n).$$

Para além das operações já descritas, todas as restantes têm custo constante, $\mathcal{O}(1)$.

Assim, o cálculo final de complexidade temporal é o seguinte:

$$\mathcal{O}((n+e)+(n)+(n+e)+(n)) = \mathcal{O}(n+e)$$

3.2 Espacial

Para fazer a análise da complexidade espacial, é necessário ter em conta todo o espaço de memória, que varia com o *input*, ao longo de todo o programa.

- Quando é criado um grafo, é necessário a criação de um *array* de listas, com *n+1* posições, para a lista de adjacências. Aqui estão representadas as arestas bidirecionais, **e**, do grafo, ou seja cada aresta é representada duas vezes, uma em cada vértice. $\Theta(n+2e) = \Theta(n+e)$.
- Quando é realizado o Algorithm 2 (*BFS*) são criados dois *arrays* com n+1 posições e uma *queue* (*LinkedList* na implementação em *Java*) que terá, no máximo, *n-1* elementos. $\mathcal{O}((n-1)+(n+1)+(n+1)) = \mathcal{O}(n)$
- No Algorithm 1 (Algoritmo principal) é usado um *array* de tamanho *n+1* para guardar os valores de distâncias, retornado pelo *BFS*. $\Theta(n+1)$.

As restantes variáveis utilizadas são números inteiros e não variam com *input*, tendo complexidade espacial constante. Assim sendo, a complexidade espacial deste programa é $\Theta(n+e)$.

4 Comentários e fontes consultadas

Este programa cumpriu os requisitos propostos, de modo a calcular o maior caminho mais curto entre quaisquer dois habitantes da ilha.

Foram consultados alguns *sites* para solidificar o algoritmo utilizado para resolver o problema, nomeadamente [1] e [2].

Referências

[1] Tamer Badr. Tree diameter — why does two bfs solution work? https://medium.com/@tbadr/tree-diameter-why-does-two-bfs-solution-work-b17ed71d2881, 2020. Acedido: 18 Abril 2025.

[2] GeeksForGeeks. Diameter of n-ary tree using bfs. https://www.geeksforgeeks.org/diameter-n-ary-tree-using-bfs/, 2020. Acedido: 18 Abril 2025.