**Descrição das Soluções dos Requisitos**

Numa primeira tentativa de resolução do problema começámos por conceptualizar uma abordagem "top-down" que partisse de uma pesquisa dfs modificada a começar por cada vértice órfão, e que fosse acumulando para cada vértice o caminho até lá percorrido, de forma recursiva. Contudo, rapidamente nos apercebemos que tal solução iria implicar uma complexidade temporal e espacial longe de desejável. Sendo assim, decidimos inverter o problema e desenhar uma solução "bottom-up", que começasse pelos vértices v1 e v2, explorando subsequentemente os seus ancestrais.

Para o requisito (1), em que é preciso determinar se a árvore é válida, começamos por ler a partir do input as relações de parentesco. Guardamos por cada vértice uma referência para cada um dos progenitores, numa estrutura de dados “vertex”. Ao tentar atribuir a um vértice um terceiro progenitor, é acusada a invalidez da árvore. Dentro de “vertex” também é guardada um booleano "clear", indicador da existência ou não de um ciclo atingível a partir do vértice (note-se que esta propriedade é hereditária). Deste modo, ao realizar uma pesquisa dfs recursiva a começar por cada vértice, com um unordered\_set que vai mantendo registo do caminho percorrido, atribuindo "clear = true" para cada vértice através do qual a dfs chega somente a antepassados órfãos, e retornando de imediato "false" assim que um caminho vá ter a um vértice já descoberto, conseguimos eficazmente detetar a existência de ciclos. Equivalentemente a uma DFS típica, estamos na presença de um limite assimptótico de grandeza O(V+E). Contudo, se a DFS tiver início num vértice pertencente a um loop, o número de iterações é constante (Ω(1)).

Por último, no que concerne o requisito 2, guardamos também para cada vértice uma variável "color", que pode ser “branco”, “azul”, “vermelho”, “roxo”, ou “preto”. Para v1 realizamos uma dfs em que marcamos todos os seus antepassados, originalmente a branco, com a cor azul. De seguida, uma outra DFS com início em v2 marca todos os antepassados brancos de v2 a vermelho. Ao encontrar um antepassado azul, marca-o a roxo, colorindo seguidamente todos os seus antepassados de preto, independentemente da sua cor. No final, sabemos que os vértices roxos vr constituem a solução do problema, pois constam na interseção dos antepassados de v1 e v2 e é impossível haver um vértice vr’ roxo descendente deste, porque se esse fosse o caso, vr  teria sido marcado a preto. Como se trata de uma DFS, a complexidade típica pertence a O(V+E), conquanto na maioria dos casos apenas uma fração do grafo é explorada, ocorrendo ainda um número constante de iterações se o ancestral comum mais próximo for órfão e progenitor de v1 e v2.

**Pseudocódigo de partes das lógicas por detrás das soluções apresentadas:**

Verificação de loops: Determinação do LCA:

**for** i=1 to #vertices **do DFS**(int v, color c, vertex vertices[])

**let** path be a new unordered\_set **if** vertices[v].color && c == blue **then**

**if** (has\_cycle(i, path, vertices)) **then**  c = purple

print(“0”) **end if**

**end if** vertices[v].color = c

**end for** **for** p in v.parents do

**if** p.color == white || p.color == red &&

c == blue **do**

**has\_cycle(**int v, unordered\_set path, vertex vertices**[])**  DFS(p, c, vertices)

**if** vertices[v].clear == true **then**  **end if**

**return** false **else if** (c == purple || black) **do**

**end if** DFS(p, black, vertices)

**if** path.contains(v) **then** **end if**

**return** true; **end for**

**end if** **end**

**else do**

path.insert(v)

**end else**

bool res = res

**for** p in v.parents **do**

res = res || has\_cycle(p, path, vertices)

**end for**

vertices[v].clear = !res

**return** res

**end**

**Análise Teórica**

Como numa árvore válida existem no máximo duas arestas por vértice, temos que no pior caso E = 2V.

* Leitura de arestas para array de vértices: O(E) = O(2V)
* Algoritmo que verifica se há loops: O(V+E) = O(3V) = O(V), Ω(1)

Este algoritmo percorre em média cada vértice e aresta uma única vez. Se encontrar de imediato o loop, corre apenas três iterações.

* DFS para determinar ancestral comum mais próximo: O(V+E) = O(V), Ω(1)

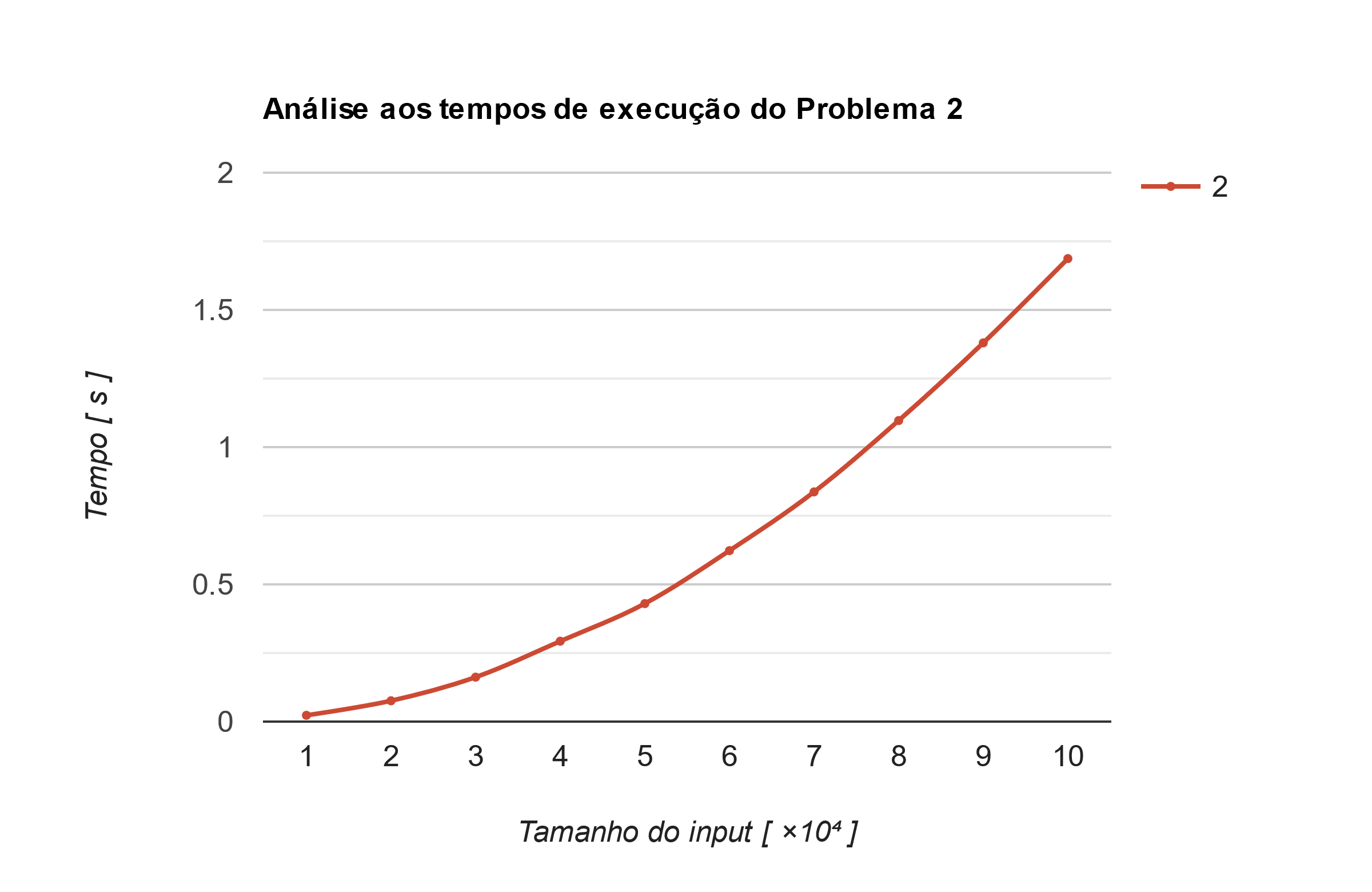
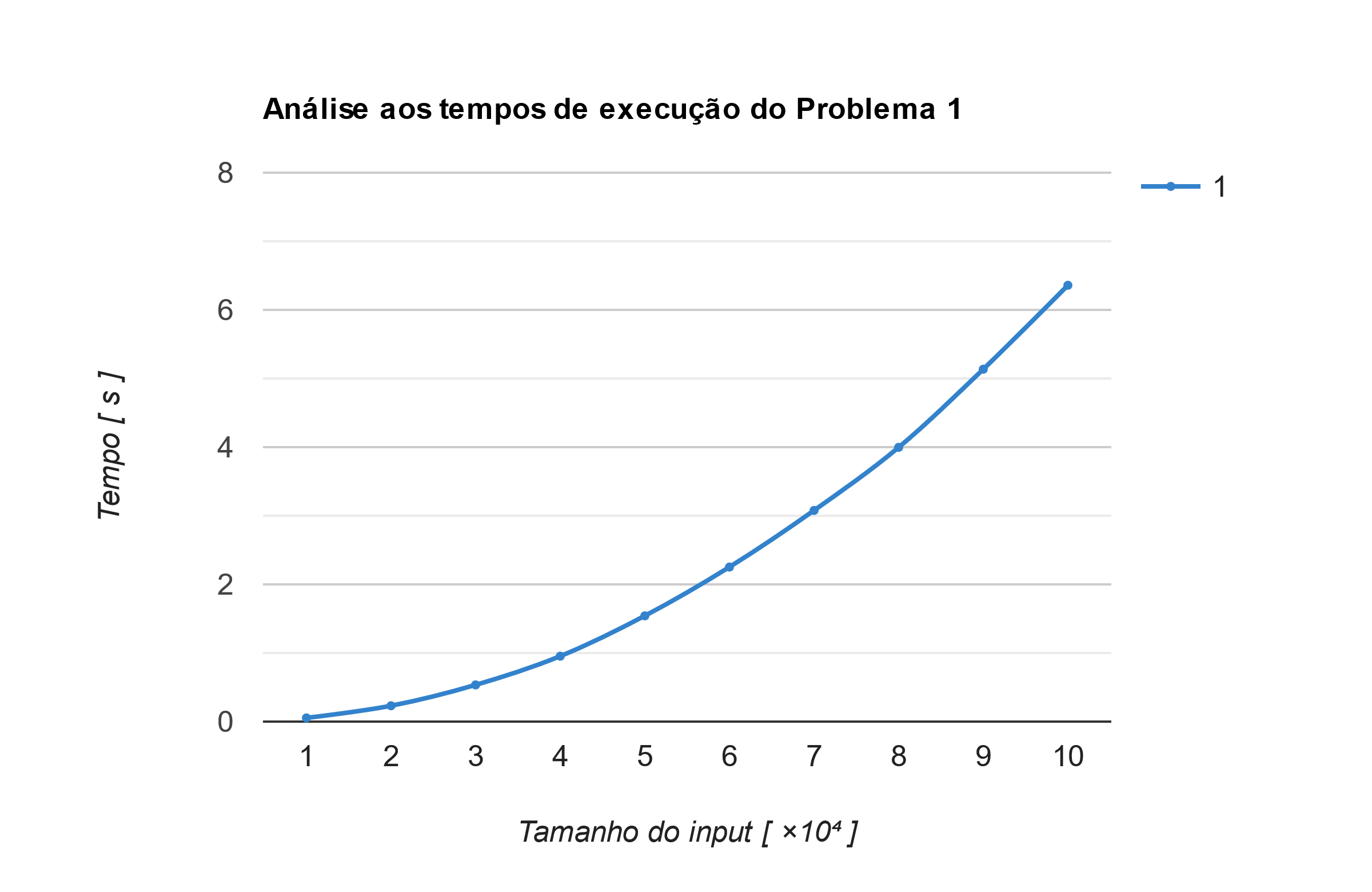
Em média, este algoritmo percorre uma fração do número de arestas e vértices do grafo. No melhor caso, encontra de imediato um antepassado órfão, no pior terá que percorrer todos os vértices e arestas, tal como uma DFS normal.

Complexidade de tempo: O(V + E) = O(V), complexidade de espaço: O(2V) = O(V).

**Avaliação Experimental dos Resultados**

A análise aos tempos de execução foi realizada com base em gráficos com tempos associados. Para cada problema, foram registados 10 resultados, um para cada tamanho da sequência de *input* e de forma incremental, desde 1×104 até 1×105, de 10000 em 10000, e calculados os tempos para cada instância. No 2º problema, ambas as sequências dadas têm o mesmo tamanho.

Para cada tamanho, foram realizados 50 testes com inputs sempre diferentes com o **random\_k** e calculada a média, com recurso à ferramenta **hyperfine**. O tempo encontra-se em segundos (eixo dos YYs) e o tamanho na potência 104 (eixo dos XXs).



Recordando a complexidade das soluções dos 2 problemas, as curvas dos gráficos gerados estão, ambas, de acordo com aquilo que seria de esperar de um algoritmo com limite assimptótico apertado de grandeza n2.