In fact, this equation is equivalent to Jacobi's theorem. To see this, let us first rewrite the right-hand side. The standard formula for the sum of a geometric series implies:

$$\sum_{n=1}^{\infty}\left(\frac{x^{4n-3}}{1-x^{4n-3}}-\frac{x^{4n-1}}{1-x^{4n-1}}\right)$$
$$=\sum_{n=1}^{\infty}\Big(\sum_{k=1}^{\infty}x^{(4n-3)k}-\sum_{k=1}^{\infty}x^{(4n-1)k}\Big)$$
$$=\sum_{n\in\mathbb{N}\setminus\{0\},k\in\mathbb{N}}(x^{(4n-3)k}-x^{(4n-1)k})=\sum_{k=1}^{\infty}(d_1(k)-d_3(k))x^k.$$

Now let us rewrite the left-hand side:

$$\Big(\sum_{n=-\infty}^{\infty}x^{n^2}\Big)^2=\Big(\ldots+x^{(-1)^2}+x^{0^2}+x^{1^2}+\cdots\Big)^2=1+\sum_{k=1}^{\infty}r_2(k)x^k.$$

Thus, we have proved that

$$1+\sum_{k=1}^{\infty}r_2(k)x^k=1+\sum_{k=1}^{\infty}(4d_1(k)-4d_3(k))x^k,$$

which completes the proof of Jacobi's theorem. □

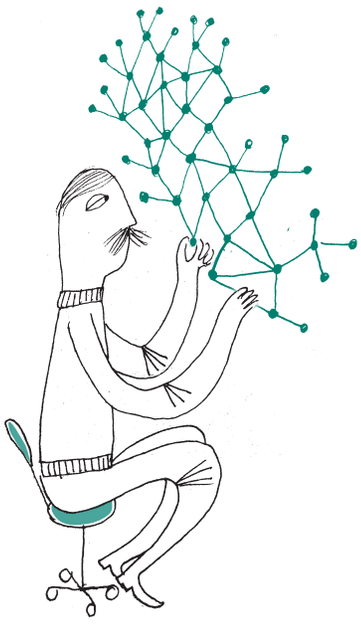
Paul Erdős, one of the greatest mathematicians of the 20th century, often referred to The Book, where God keeps the perfect proofs for mathematical theorems. He famously said: "You don't have to believe in God, but, as a mathematician, you should believe in The Book". If such a book truly exists, I believe that these proofs, which are an excellent example of the deep connections between different areas of mathematics, would certainly deserve a special place in it.

Can you see a graph here?

*Sylwia SAPKOWSKA**

* Student, Faculty of Mathematics, Informatics, and Mechanics, University of Warsaw

Graph search algorithms are everywhere. Without them, your favorite web search engine wouldn't work, and you probably optimize your own "graph paths" subconsciously – for example, when planning your day. The connections between you, your friends, and your family also form a graph, which we feverishly search through during social gatherings.



Since graph search algorithms have many applications in everyday life, it's no surprise that numerous math olympiad problems can also be solved using this idea.

Depth First Search

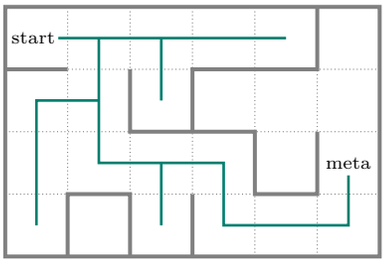
Before moving forward, let's discuss the simplest graph search algorithm – **Depth First Search**, or DFS for short.

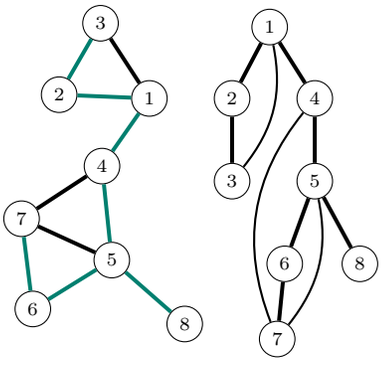
We can think of it as a "walk" through a graph. While performing the algorithm, we keep track of whether a given vertex has already been visited. We start by selecting an initial vertex. When we reach some vertex v during the execution of the algorithm, we follow these steps:

1. Mark vertex v as visited.
2. For each adjacent vertex u of the current vertex v, if u has **not** been visited yet – recursively perform this procedure for u.
3. After exploring all adjacent vertices of v, backtrack to the previous vertex – the one from which we arrived at v.

DFS can be thought of as an analogy to sightseeing. Imagine you're a tourist in a bustling city, holding a map and eager to explore every hidden gem the city

The first version of depth-first search was used in the 19th century by the French mathematician Charles Pierre Trémaux as a strategy for solving mazes, like the one below. The cells of the grid can be interpreted as graph vertices. The edges in the graph represent possible movements, so we connect vertices if and only if the cells they represent are adjacent by their sides.





On the left side of the figure, we have a graph with 8 vertices. Suppose we start our DFS at vertex 1. One possible traversal order could be:
$1 \to 2 \to 3 \to 2 \to 1 \to 4 \to 5 \to 6 \to 7 \to 6 \to 5 \to 8 \to 5 \to 4 \to 1$. Notice that if there were an edge between 3 and 4, this traversal would have gone from 3 to 4 instead of returning to 2. The edges used to reach unvisited vertices form a spanning tree (these edges are marked in red on the left diagram). The diagram on the right shows the same graph with the rooted spanning tree structure. All additional edges – those not used during DFS traversal – are back edges.

has to offer. You start at a famous museum, soaking in the art and history, then look around for another place you haven't visited yet. If you discover a new café, a charming alleyway, or a historical landmark, you head there next, enjoying the sights and adding them to your "visited" list. When you reach a point where everything nearby has been explored, it's time to backtrack – to return to the last place that had unexplored paths. From there, you continue, uncovering new attractions and moving forward until you've fully explored every possible location. This method of sightseeing ensures that you don't miss anything. By the end of your journey, you will have covered the entire city, leaving no landmark undiscovered and no corner unexplored! The same thing happens in a connected graph – after performing DFS, the entire graph becomes "fully visited."

It's easy to see that during the algorithm's execution, each vertex has three possible states – it is either unvisited (not yet reached by DFS), visited (reached but still being explored), or fully explored (all of its adjacent vertices have been examined).

Back to the Problems

Using the DFS algorithm, we can divide the edges of a graph into **tree edges**, also called spanning edges (those traversed during the walk, forming a rooted spanning tree), and **back edges** (the remaining edges, which complete cycles). It is helpful to think of tree edges as directed downwards and back edges as directed upwards.

Why is this way of analyzing a graph useful? To see this, let's prove the most important property concerning DFS traversal. Suppose we have a connected graph G and perform DFS on it, starting from an arbitrary vertex r.

Our goal is to show that for any edge (u, v), the vertex v is either an ancestor or a descendant of u in the DFS tree – meaning that v either lies on the path from u to the root r or is in the subtree of u.

Why does this property hold? Suppose there is an edge (u, v), and without loss of generality, DFS reaches u while v is still unvisited. Then:

- If DFS moves from u to v using (u, v), then (u, v) is a tree edge.
- If DFS does not proceed to v from u using (u, v), then v must have already been visited when this edge was considered. This could have happened only if v was reached and explored earlier, during the traversal of some other branch originating from u.

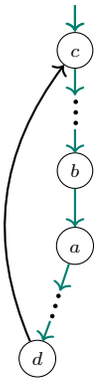
In both cases, v is a descendant of u in the DFS tree.

The key advantage of the DFS tree is that it simplifies graph analysis. Instead of handling various types of edges, we can focus on a tree structure with a few additional edges connecting ancestors to descendants. This makes the graph much easier to analyze.

Problem: Let G be a connected, undirected graph. Prove that the edges of G can be oriented in such a way that the resulting directed graph is strongly connected if and only if G has no bridges, i.e., edges whose removal disconnects the graph.

Solution: If G has a bridge (u, v), directing this edge from u to v (without loss of generality) means there is no path from v to u. If such a path existed, (u, v) would not be a bridge. Now, let's focus on the other implication.

Assume that G has no bridges. Perform DFS on the graph, starting from an arbitrary vertex r. This process forms a rooted spanning tree using tree edges, along with additional back edges. Let's orient tree edges downward (from the root) and back edges upward (towards the root). To prove that such a graph is strongly connected, we need to show that there exists a directed path from r to



every vertex and, symmetrically, a path from every vertex to r. The first part is easy – since DFS visits every vertex, there exists a path using only tree edges from r to each vertex.

Define the depth of a vertex v as the number of tree edges from the root r to v. Consider a vertex a distinct from the root. In the DFS tree, there is a tree edge (b, a) connecting a to its parent b. Since the graph has no bridges, for the tree edge (b, a), there must exist some back edge (d, c) connecting some vertex d from the subtree of a (possibly $d = a$) to some ancestor c of b (possibly $c = b$). Otherwise, (b, a) would be a bridge.

To find a path from a to the root, start by traveling from a to d using tree edges. Then, follow the back edge from d to c. Since the depth of c is smaller than that of a, we repeat this process until reaching the root.

Problem (LXVII Polish Mathematical Olympiad, second stage): Given a connected, undirected graph with an even number of edges, prove that its edges can be paired in such a way that each pair shares exactly one common vertex (in other words, each pair forms a path of length two).

Solution: Perform the DFS algorithm on the graph starting from an arbitrary vertex r and define the depth of a vertex v as the number of tree edges from the root r to v. We will now describe an algorithm for processing the vertices. After pairing two edges, we can remove them. Initially, all vertices are marked as unprocessed:

- Select an unprocessed vertex v with the greatest depth that is not the root.
- Consider the edges from v that have not yet been removed. There are three types of such edges:
  – back edges connecting v with its ancestors,
  – tree edges connecting v with its children,
  – a single edge connecting v with its parent via a tree edge.
  If the number of remaining edges is even, we pair them arbitrarily and remove all of them. Otherwise, we pair them in such a way that only the edge from v to its parent remains.
- Mark v as processed.

It remains to pair up the edges outgoing from r. There is only one type of such edges – those connecting r to its children. Initially, the graph had an even number of edges, and each removed pair consisted of exactly two edges. Thus, r has an even number of children, and we can pair them arbitrarily.

Now, we should discuss why this construction is correct. When considering a vertex v, the vertices in its subtree are already processed. Since in one step of the algorithm we remove all children of a vertex, the only remaining edges in the subtree of v before pairing must be precisely its children, and they will certainly be paired when processing v.

On a Special Vertex in Trees

Everyone knows what trees are. They grow in parks (somehow upside-down – why is the root at the bottom?). Usually, such trees grow symmetrically in every direction – rarely does one branch have noticeably more leaves than others. Is this also the case with graph trees? It turns out that the answer is yes. More formally, we will show that in every tree with n vertices, there exists a vertex v such that if we root our tree at v, then each of its subtrees has size at most $\lfloor \frac{n}{2} \rfloor$. Any such vertex is called a **centroid** or a **tree center**.

We will attempt to find such a vertex recursively. Suppose we root our tree at some vertex r. If all subtrees of r have sizes at most $\lfloor \frac{n}{2} \rfloor$, then we are done, as we have found a vertex with the desired property. Otherwise, there must be a vertex w that is a child of r whose subtree has size greater than $\lfloor \frac{n}{2} \rfloor$. Moreover, such a vertex must be unique – if there were at least two subtrees with sizes

$s_1, s_2 > \lfloor \frac{n}{2} \rfloor$, then we would have:

$$n \geqslant s_1 + s_2 + 1 \geqslant 2\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) + 1 > n,$$

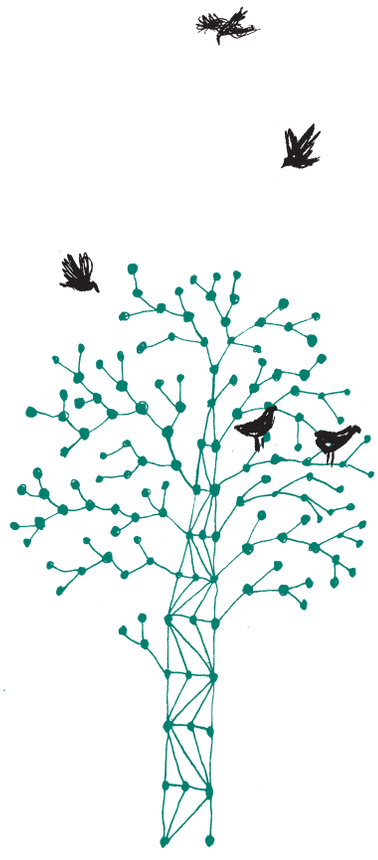
which contradicts the total size of the tree. Similarly, we can show that if the subtree of w has size greater than $\lfloor \frac{n}{2} \rfloor$, then after rooting the tree at w, the subtree containing r will have size at most $\lfloor \frac{n}{2} \rfloor$.

Thus, we can traverse through all neighbors of r, and since w is unique, we recursively check whether w is a suitable candidate for a centroid. This algorithm must terminate because, in each step, the size of the largest subtree decreases.

Observe that a centroid is not necessarily unique. For example, in a graph consisting of two vertices connected by an edge, both vertices are centroids – each of them has a subtree of size $1 = \frac{2}{2}$. From this example, we can conclude that a tree has exactly two centroids if and only if there exists an edge whose removal splits the tree into two smaller trees, each with exactly $\frac{n}{2}$ vertices (the proof is left as an easy exercise).

Problem: Assume that in a certain country where the MBL camp will be held, there are n cities connected by $n-1$ roads forming a tree. As the organizer of the camp, your task is to efficiently accommodate the participants. Exactly $2k$ people from different cities $v_1, \ldots, v_{2k}$ have been admitted. During the camp, participants will form pairs (teams of two friends). The pairing has not yet been determined. Each pair will require accommodation. There is exactly one hotel in each city. Participants from the same pair should stay in the same hotel (and there may be other pairs in that hotel as well). The condition that must be met is as follows: if people from cities u and w form a pair, their hotel must be located in a city on the shortest path connecting u and w (possibly in u or w). Since renting many hotels in different cities would be an organizational challenge, pair up the participants in a way that minimizes the number of different hotels used while satisfying the above condition.

Solution sketch: In this problem, we are given a tree with $2k$ special vertices $W = \{v_1, \ldots, v_{2k}\}$. Let's assign a weight to each vertex: $c_v = 1$ if $v \in W$ and 0 otherwise. If we find a vertex v such that in every subtree formed by removing v, there are at most k special vertices, then the problem is solved. Why? Because we can then greedily pair up vertices from different subtrees with the highest number of special points, and such paths will always pass through v. The question remains: how do we find such a vertex v? The definition of v resembles the definition of a centroid. Instead of calculating subtree size, we will compute the sum of weights in the subtree. Then, the centroid-finding algorithm will also find our desired vertex v. Therefore, we conclude that it is always sufficient to rent just one hotel.



Here is another problem, this time left as an exercise for the reader:

Problem (based on a task from the second stage of the XXX Polish Olympiad in Informatics): Antek and Marysia are playing a game on a tree. Initially, all vertices are white, except for one vertex x, which is colored red (belonging to Antek), and another vertex y, which is colored blue (belonging to Marysia). Each player takes turns selecting any vertex u of their color and then coloring an adjacent white vertex v with their color. The player who cannot make a move loses the game. Determine who has a winning strategy depending on the initial positions x and y of Antek and Marysia.

Summary

As we have seen, there are many applications of DFS and centroids in mathematical olympiad problems. However, DFS is much more than that – we use graph search algorithms almost constantly without even realizing it! So next time you are solving a sudoku puzzle or rushing to school or work, remember that you are actually recursively following some path in a graph, hoping to find the right way out.