
1: Common mistakes in math proof strategies

Adapted from <https://matheducators.stackexchange.com/questions/17900/taxonomy-of-bad-proofs>

1. **Assuming the conclusion:** Proofs that (explicitly or implicitly) assume the result you are trying to prove in the proof.

Typically, these won't outright say that they claim the conclusion, but it will be implicitly part of a statement. For example, if I am trying to prove that a graph is connected, and then I say, "take the path from s to t " without proving such a path exists, that implicitly assumes the graph is connected.

2. **Backwards reasoning:** Proofs that start from the desired conclusion and work backwards.

Although it can be helpful when you're figuring out the solution to work both backwards and forwards to make ends meet, you should make sure when writing the proof to write it forwards. It's easy to introduce errors working backwards when some steps aren't reversible.

3. **Proof by Example / Missing Case / "Obvious" worst case:** Proofs that don't fully analyze all cases of a problem.

Of course, you can't prove a theorem by just giving a single example. But, it is *sometimes* true that you can prove a theorem (or often, an asymptotic analysis) if you can prove that there is some worst-case and just analyze that. However, in those cases, you must *prove* that the worst-case really is the worst-case, and not just assume it. A similar problem appears if you try to divide a problem into multiple cases and then prove it for each of them, but you might make a mistake because you missed a case.

4. **The tangle:** Proofs that just keep going and going until somehow mysteriously reaching the right conclusion.

Strictly speaking, these proofs can sometimes be technically correct, but very often, what happens is that somewhere along the way, a mistake is made in the middle but it's hard to find since it's buried in a pile of more-or-less correct but irrelevant or unnecessary details—this is why I have in my syllabus that proofs should be succinct.

5. **Inappropriate types:** Proofs that substitute an object for a similar looking one without realizing.

Maybe you remember some fact about trees, but think that it applies to graphs, and then use that in your proof.

6. **The incantation:** Proofs that try to force a theorem or algorithm to work in situations it's not designed for.

Often happens in class when you've just learned a theorem/algorithm, so you might guess that you need to use that theorem on the homework or test. But, remember that to use a theorem, you need to check the hypotheses for applicability. For example, using Dijkstra with negative edges fails, because you didn't check the requirements of Dijkstra.

7. **Implication confusion:** Proofs that use incorrect logical implication.

These are just common classic logical mistakes. For example, you are trying to prove that X implies Y , but you instead prove Y implies X , or not X implies not Y .

2: A few common signs of errors in algorithms solutions

1. **Too much power:** the logic in the solution or the algorithm would apply just as well for proving/solving an impossible problem, so even if you can't spot where the logic went wrong, you know it must've gone wrong somewhere.

Example: if I see an algorithm for $n \times n$ matrix multiplication in $O(n \log n)$ time, I don't need to check the proof at all to know that this is wrong somewhere, because that would imply I can read the matrix in less than n^2 time, which is impossible.

2. **Hidden loops/runtime:** in your runtime analysis, you might forget about a step that causes the algorithm to be much slower than you think.

Example: this will often happen when you implicitly assume something about the way your data is structured (e.g. that your data is sorted). For example, if I am trying to find the runtime of Kruskal, and I say to pick the lowest cost edge, this is a free step if my edge weights are sorted, but is expensive if I don't do that.

3. **Unproven greedy choice:** Local optimality doesn't imply global optimality. That has to be proven. Note that so far in class, we've only really dealt with greedy algorithms, where local optimality works, but we're moving beyond that now.
4. **Not using the input:** An algorithm that barely uses the input and outputs an answer is unlikely to be correct. For example, if you have a counting problem, but the answer is always 0 or 1, something probably went wrong.
5. **Not using the hypothesis:** An algorithm or proof that doesn't use the conditions and properties given is either doing too little work or solving a different problem.

We haven't really gotten to reductions yet, but including here since it's super common later.

6. **Improper reductions:** Often, we'll convert a problem we don't know how to solve into one we do know how to solve, but you need to make sure the conversion works, is not too expensive, and is in the right direction. We won't talk about this one much yet, but this will be by far the biggest error when we get to NP-hardness reductions.

3: Exercises

For each of the following incorrect attempts, find and classify the errors. Note that many of the error types overlap, and the same attempt may contain multiple errors.

Problem 1. In tree $T = (V, E)$, if we remove edge $e = (u, v) \in E$ from T , then the tree is no longer connected. In other words, in $T \setminus \{e\}$, many node pairs would become disconnected (i.e., there would be no path between some node pairs). We therefore define the importance of edge $e = (u, v) \in E$ as the number of node pairs that will become disconnected if we remove e from T . Design an $O(V)$ algorithm to find the most important edge in any given tree.

Attempt (a). To find the edge that disconnects the most pairs, we just need to analyze the worst-case shape for a tree, which is a “line” or “path” graph (where every node has degree 2, except the ends). In a line graph of V nodes, the edge that splits the tree into the most balanced components ($V/2$ and $V/2$) is the one exactly in the physical center of the line. Therefore, for any general tree, we just find the longest path (the diameter), and return the middle edge of the diameter.

Solution: *Obvious worst case / insufficient cases. The proof above only proves things for one case, which is asserted to be the worst case.*

Attempt (b). We need to know how many node pairs are disconnected by removing an edge. We can just explicitly find all paths in the tree. We initialize a 2D matrix of size $V \times V$. For every node u , we run a DFS to find the simple path to every other node v , and we increment a counter for every edge that appears on that path. After checking all pairs, we just scan our edge counters and return the edge with the highest number. Since DFS is $O(V)$, the algorithm is $O(V)$.

Solution: *Hidden runtime. If you have an algorithm that explicitly goes over all V^2 pairs of nodes, it's not possible for it to be $O(V)$ no matter how fast the inner loop is.*

Problem 2. Let G be a connected graph where every edge has a different weight (as we assumed in class). Show that there is only one, unique minimum spanning tree in this case.

Attempt (a). We can model the graph using the Laplacian Matrix, $L = D - A$, where D is the degree matrix and A is the adjacency matrix. By Kirchhoff's Matrix Tree Theorem, the total number of spanning trees in G is equal to any cofactor of the Laplacian. Since all edge weights in G are distinct numbers, the eigenvalues of the Laplacian will also be strictly distinct. Because the characteristic polynomial has no repeating roots, the minimum eigenvalue corresponds to a unique eigenvector. Thus, projecting the spanning trees onto this basis yields only 1 minimum spanning tree.

Solution: *The Tangle. Kirchhoff's theorem, the Laplacian, and eigenvalues are all real things, but they have nothing to do with minimum spanning trees. This is just a mess of fancy-sounding words from linear algebra but not a proof of any kind.*

Attempt (b). Suppose there are two edges e_1 and e_2 with the exact same weight. If e_1 and e_2 both cross the same cut, and are the lightest edges in that cut, we could choose either one to include in our spanning tree. This would result in two different valid minimum spanning trees, meaning that we don't have a unique minimum spanning tree. Therefore, the claim above is false.

Solution: *Implication Confusion. The biggest problem here is that the proof structure tries to prove $P \implies Q$, where P is "all edge weights are distinct" and Q is "the MST is unique", but actually only proves that $\neg P \implies \neg Q$. A second mistake is claiming that if there are two equal weight lightest edges in the same cut, that forces non-uniqueness. Consider the triangle with weights 1, 1, 2.*

Attempt (c). Let G be a graph with distinct edge weights. We want to prove the MST is unique. Suppose G has a unique MST, let's call it T . Because T is unique, Kruskal's algorithm must never encounter a situation where it has to arbitrarily choose between two edges of the exact same weight that both connect different components. If it did, it could branch into two different trees. Since it doesn't, all the edges evaluated must have distinct weights. Therefore, if the edge weights are distinct, the MST is unique.

Solution: *Assuming the conclusion. This proof starts from the conclusion (that you have a unique MST). It hides it a bit behind sounding like a proof by contradiction, but it never reaches a contradiction.*

Attempt (d). We want to show the MST is unique. The MST is unique if Kruskal's algorithm can only ever produce one tree. Kruskal's can only produce one tree if it never has to break a tie, since sorting the edges is the only point at which the algorithm makes a choice. And it never has to break a tie if no two edges have equal weight, which is exactly our hypothesis. Reading the chain from the bottom up, the MST is unique.

Solution: *Backwards reasoning. The chain is assembled from the goal downward, and the author never travels any link in the direction the argument needs. The rest of the chain is fine: distinct weights means no ties, no ties means the sorted order is forced, so Kruskal produces exactly one tree. The bad link is the first one, "the MST is unique if Kruskal can only ever produce one tree." The easy direction is the other one. Getting the direction actually needed requires knowing that every MST arises from some run of Kruskal, which is never stated here and is no easier than the claim being proved. Working backwards is what let the author cross that link without noticing there was anything to prove.*

Problem 3. Find the MST of a connected graph in $O(E \log V)$ time.

Attempt (a). We can find the MST using the Cut Property, which states that the lightest edge crossing any cut must be in the MST. Because we are looking for exactly $V - 1$ edges to form our spanning tree, and the Cut Property guarantees that lightest edges belong to the MST, we can just sort all the edges in the graph globally from lightest to heaviest. The globally lightest edge is the lightest edge of some cut. The second lightest edge is also the lightest edge of some cut, and so on. Therefore, we can just take the $V - 1$ absolutely lightest edges in the entire graph. Because they are the lightest edges, the Cut Property guarantees they are in the MST. Sorting takes $O(E \log E) = O(E \log V)$ time, and picking the first $V - 1$ edges takes $O(V)$ time.

Solution: *The Incantation. The Cut Property only says that the lightest edge crossing a cut belongs to the MST. The attempt needs a further claim, that each of the $V - 1$ lightest edges is lightest across some cut, which is never proved but just handwaved with “and so on”. That claim is false, and the attempt does not rule out the possibility of the lightest $V - 1$ edges containing a cycle.*

Problem 4. You are given an undirected graph $G = (V, E)$ and two nodes $u, v \in V$. All edges have weight 1. Design an $O(V + E)$ algorithm to find the number of shortest paths between u, v . If u and v are not connected, the number should be zero.

Attempt (a). We are looking for shortest paths between u and v in an undirected graph where all weights are 1. Since all weights are equal, we can just run Kruskal's or Prim's algorithm to find the Minimum Spanning Tree (MST) of G . In a tree, there is always exactly one unique path between any two nodes. Therefore, if v is in the MST, the number of shortest paths is 1. If it's not, the number is 0.

Solution: *Inappropriate types. A path in an MST is not the same as a shortest path in an undirected graph. Also an example of barely using the input.*

Problem 5. Let $G = (V, E)$ be a directed acyclic graph (DAG) where every edge has a positive integer weight. Design an $O(V + E)$ algorithm to find the longest path from a given start node s to a target node t .

Attempt (a). To find the longest path, we can just use a modified greedy search. We start at node s . At each step, we look at all outgoing edges from our current node. We select the edge with the maximum weight and traverse it. We repeat this process until we reach t . Because we are always picking the heaviest available edge at every single step, we will accumulate the maximum possible weight by the time we reach t .

Solution: *Unproven greedy choice / The Incantation. The proof assumes that a sequence of locally optimal choices (picking the heaviest adjacent edge) will yield a globally optimal solution. This is not inherently true for longest paths, as a heavy edge might lead to a dead end or a series of very light edges.*

Also, Too Much Power / Not using hypothesis. Notice that the argument never uses the fact that it's a DAG, but would apply to a general graph. It turns out the problem for a general graph of finding the longest non-repeating path is NP-hard, so we definitely can't solve it with a greedy algorithm.

Also, Assuming the conclusion. We have the line "we repeat this process until we reach t ", which assumes that this process reaches t .