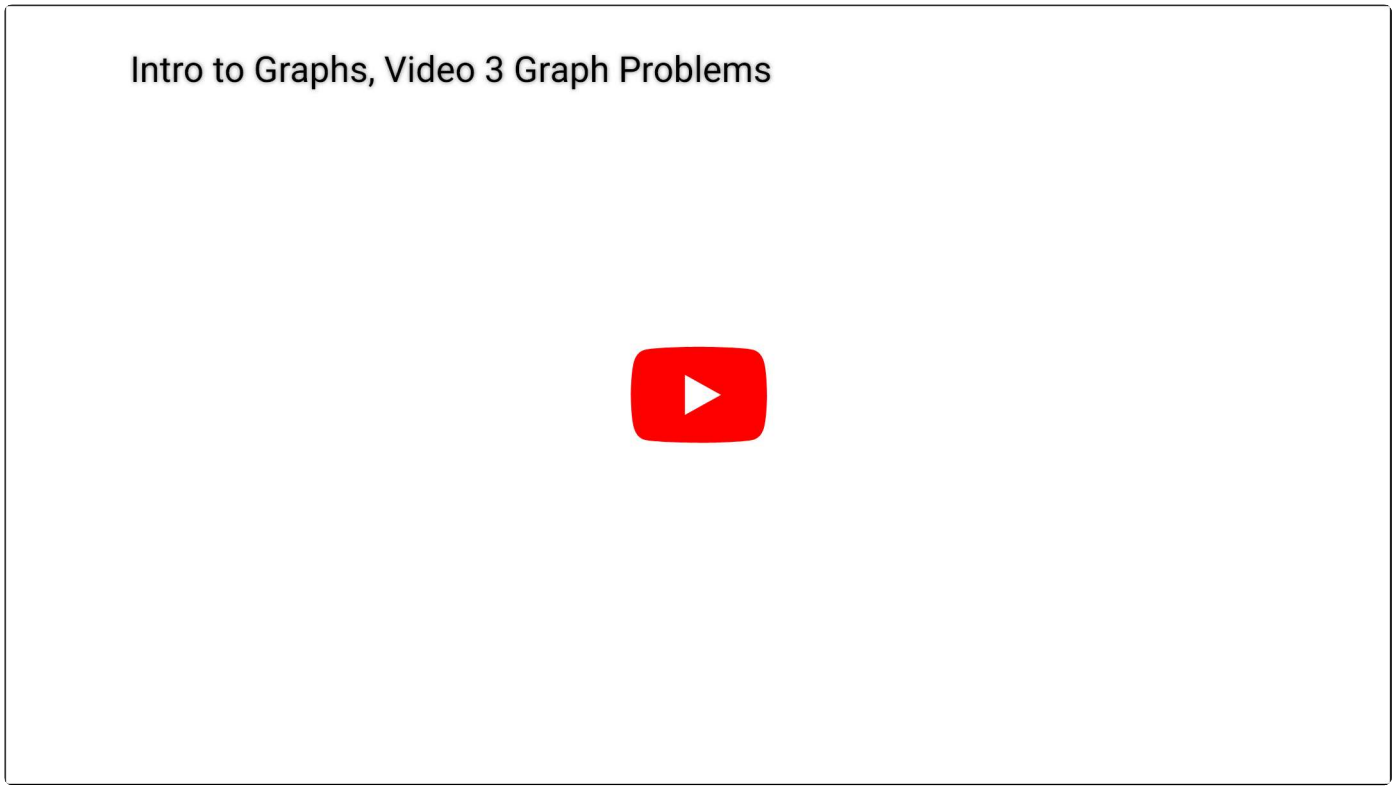
22.4 Graph Problems



Graph Problems

There are many questions we can ask about a graph.

For example,

- **s-t Path**: Is there a path between vertices s and t?
- **Connectivity**: Is the graph connected, i.e. is there a path between all vertices?
- **Biconnectivity**: Is there a vertex whose removal disconnects the graph?
- **Shortest s-t Path**: What is the shortest path between vertices s and t?
- **Cycle Detection**: Does the graph contain any cycles?
- **Euler Tour**: Is there a cycle that uses every edge exactly once?
- **Hamilton Tour**: Is there a cycle that uses every vertex exactly once?
- **Planarity**: Can you draw the graph on paper with no crossing edges?
- **Isomorphism**: Are two graphs isomorphic (the same graph in disguise)?

What's cool and also weird about graph problems is that it's very hard to *tell* which problems are very hard, and which ones aren't all that hard.

For example, consider the Euler Tour and the Hamilton Tour problems. The former... is a solved problem. It was solved as early as 1873. The solution runs in O(E) where E is the number of edges in the graph.

The latter? If you were to solve it efficiently today, you would win every Math award there was, become one of the most famous computer scientists, win a million dollars, etc. No one has been able to solve this **efficiently**. The best known algorithms run in exponential times. People have been working on it for many decades!

One step at a time!

Alright, well, before we solve the million dollar problem, let's solve the first one on the list. Given a source vertex s and a target vertex t, is there a path between s and t?

Graph Connectivity

In other words, write a function `connected(s, t)` that takes in two vertices and returns whether there exists a path between the two.

To begin, let's guess that we have the following code:

```
if (s == t):
    return true;

for child in neighbors(s):
    if isconnected(child, t):
        return true;

return false;
```

Exercise 17.4.1. Before you move on, please read the code above, and spend time thinking about it. Does it work? Is it efficient? Run through a couple scenarios.

Alright, so, let's try it out.

We start with `connected(0, 7)`? That recursively calls `connected(1, 7)`, which then recursively calls `connected(0, 7)`. Uh-oh. Infinite looping has occurred.

Exercise 17.4.2. How could we fix this? Once again, thinking about this. What was the problem? We visited `s` again... but did we need to?

Alright, let's try a "remember what we visited" approach.

```
mark s  // i.e., remember that you visited s already
if (s == t):
    return true;

for child in unmarked_neighbors(s): // if a neighbor is marked, ignore!
    if isconnected(child, t):
        return true;

return false;
```

As it turns out, this does work! Follow the example in these slides to see how.

Woah, what did we just develop?

Depth First Paths

You may not have realized it, but we just developed a **depth-first traversal** (like pre-order, post-order, in-order) but for graphs. What did we do? Well, we marked ourself. Then we visited our first child. Then our first child marked itself, and visited its children. Then our first child's first child marked itself, and visited its children.

Intuitively, we're going deep (i.e., down our family tree to our first child, our first child's first child aka our first grandchild, our first grandchild's first child, and so on... visiting this entire lineage), before we even touch our second child.

Up next, we'll see the opposite notion, where first we visit all our children, then our grandchildren, and so on.

Previous
22.3 Graphs

Next
23. Graph Traversals and Implementations

Last updated 1 year ago