PATH

- Given a directed graph G and two nodes s, t in G. Question:

  Is there a path from s to t?

- $PATH = \{(G, s, t): G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$

  Is $(G, s, t) \in PATH$?

- This is a stronger question than the first one.

It hides more questions: Is G a directed graph? Are s, t vertices in G?

Theorem: PATH ∈ P

- The language PATH is in the class P

- What is the language PATH exactly?

It is a collection of **binary strings** that represent triples (directed graph, vertex1, vertex2) where the vertex1 and vertex2 belong to the graph in the first component

- The first component, the graph itself, is a collection of vertices and edges.

This can be represented by an adjacency matrix (array)

- So, every triple can be represented as an array at the end, and we know that arrays get saved into bits (binary strings)

The input

- Given a binary string, can a computer decide if it corresponds to a triple (directed graph, vertex1, vertex2) ?

Yes

- Can then the computer decide if vertex1 and vertex2 are in the graph?

Yes

- After that, can the computer decide if there is a path from vertex1 to vertex2 in the graph? (Let me call this the surface question)

- Can all this be done in **polynomial time** in the size of the initially given binary string?

That's what the Theorem says

The input size

- A proof of the Theorem requires finding a **polynomial time** algorithm that **decides** PATH

- Moment of awareness before we start thinking of the algorithm:

How should we think of the size of an input here?

The input size: Theoretical vs Mechanical

- We feed an array representing (G, s, t) to our machine, which gets saved (coded) as some binary representation

- G is a set of vertices and edges, theoretically we write $G = (V, E)$

- Theoretically, it is practical to think of the size of a graph as the number of its vertices

- Mechanically, the size of a graph for a computer (TM) is the size of the graph's binary representation (including edges)

Theoretical is good enough

- When it comes to complexity analysis, it is safe to assume that the size of a graph is the number of its vertices

- Because: The size of the mechanical representation of a graph is polynomial in the number of vertices

- More precisely, there is a polynomial function $f(x)$ such that,

For an arbitrary directed graph G, if G has n vertices, then the size of the mechanical representation of G is $< f(n)$

Why safe?

- Suppose we have a graph G with n vertices
- For simplicity, assume for now it is loop-free, and not a multi-graph
- Worst case scenario for the number of edges is when every two vertices are connected (complete graph)
- Ignoring direction, that number is $\frac{n(n-1)}{2}$. Taking direction into account we have $n(n-1)$ edges
- Note that the number of edges follows a polynomial function of degree 2

- In case we have loops

Still poly of degree 2

- In case we have a multi-graph

Still poly of degree 2

- The information of vertices and edges can then be captured by arrays (adjacency matrix, say)

- Finally, switching all this to binary still results in a representation of size polynomial in n

- Note that final step is the same for natural numbers, symbols, or strings; all bits (which we always ignore)

Break

Now we are happy to simply think of the size of the graph as the number of its vertices

An Algorithm

- Recall, we want a polytime algorithm that decides PATH.

- Given a binary string:

1. Decide if it corresponds to a triple (directed graph, vertex1, vertex2)
2. Decide if vertex1 and vertex2 are in the graph?
3. Decide if there is a path from vertex1 to vertex2 in the graph?

- For ease, think of having a separate polytime algorithms for each of 1,2,3, and we run them after each other (if needed)

We focus only on 3

- Note that, in almost every decision problem, there are other hidden decision problems similar in nature to 1 or 2. Consider for example Sort we discussed last time. Or even something simpler, like addition.

- If you notice, those hidden problems concern how the data are coded into bits, and how the algorithm is designed to take in an input.

- Normally, if the input isn't valid (does not allow 3), a good program will quickly give an error within a short time (polynomial)

- This is why such hidden problems are not the main issue and do not change tractability

- In practice, deciding PATH means deciding if there is a path assuming that the given data correspond to a graph and two vertices in the graph.

So basically, like our very initial question.

At this point

- We are ready to consider time complexity based on the number of vertices instead of the size of the binary representation

- We are fine investigating 3 without worrying about 1,2

- Enough of the fuss!

Let’s start an algorithm for real

- First, let’s consider a brute-force algorithm

- Examine all potential sequences of vertices (edges)

- Check for each sequence if all the edges are valid direction-wise

- Check each sequence if it starts at *s* and ends at *t*

- Brute-force is clearly exponential

Let’s do better

1. Mark the vertex *s* (perhaps save it in a specific array called Marked)

2. Scan all the edges in the graph, and if any of them starts at a marked vertex, then mark its end vertex

3. If *t* gets marked, **accept** (there is a path). Otherwise, **reject** (no path).

Analysis

- Stage 1: Marking *s* takes a constant time (it is already given as an input).

This stage gets executed once and takes polynomial time (just creating Marked then writing *s* in Marked).

- Stage 2 is a loop work. This stage may get executed many times (How many?).

Once for each vertex in worst case.

- Stage 3: Executed once in polytime (is *t* in Marked?)

Stage 2

- Executed at most *n* times (the number of vertices).

This is because each time it marks at most one single extra vertex

- Involves scanning the input edges, checks if the start vertex is marked, marks the end vertex

The class NP

- $NTIME(f(n)) = \{L: L \text{ is a language decidable by an } O(f(n)) \text{ nondeterministic TM}\}$

- $NP = \bigcup_{k \in \mathbb{N}} NTIME(n^k)$