# Project Summary

This project is designed to analyze graph properties using propositional logic and SAT solvers. At its core, it defines a variety of graph-related propositions such as edges, adjacency, distances between nodes, and specific graph properties like being strongly connected or undirected. The logical constraints represent the structure and characteristics of a graph. For instance, constraints are used to establish the relationships between edges and nodes, ensure the symmetry of adjacency, and define the conditions for a graph to be considered strongly connected or undirected. The system also includes the functionality to handle custom graph configurations and to enforce specific conditions like graph disconnection. By compiling these constraints into a logical theory, the project can analyze and deduce various properties of graphs.

# Propositions

1. **edge\_x\_y**: Represents a directed edge from node x to node y. If an **Edge(x, y)** proposition is true, it indicates that there is a direct connection from node x to node y in the graph.
2. **adjacent\_x\_y**: Expresses that nodes x and y are adjacent to each other. The **Adjacent(x, y)** proposition being true means that there is a direct link between x and y, regardless of the direction.
3. **distance\_x\_y\_n**: Denotes there exists a path of length **n** between nodes x and y.
4. **reachable\_n**: Indicates whether a particular node n is reachable from a specified starting point. A true **Reachable(n)** proposition suggests that there exists a path within the graph that leads to node n from the starting node.
5. **property\_<name>**: Used to represent a general property or characteristic of the graph, where 'name' is a placeholder for the property's specific name. For instance, **Property("strongly\_connected")** would be used to denote whether the graph is strongly connected (every node is reachable from every other node), and **Property("undirected")** would represent whether the graph is undirected (each edge is bidirectional).

# Constraints

Just going to show the logic for some of these instead of all of them. (for a full project, you should list them all).

1. **No Self-Loops**: Constraints to prevent self-loops, ensuring that no node has an edge directed to itself.  
    - ¬Edge\_n\_n
2. **Edge Adjacency**: Constraints to define the relationship between edge existence and node adjacency.  
    - Edge\_x\_y → Adjacent\_x\_y
3. **Symmetry in Adjacency**: Constraints to ensure that adjacency is symmetric; if node X is adjacent to Y, then Y is also adjacent to X.
4. **Edge Direction for Undirected Graphs**: Constraints to ensure that if an edge exists in one direction, it exists in the opposite direction as well, applicable for undirected graphs.
5. **Distance Definition and Transitivity**: Constraints to define the distance between nodes and to ensure the transitivity of distance; if the distance from node X to Y is D1 and from Y to Z is D2, then the distance from X to Z is D1 + D2.  
    - Edge\_x\_y → Distance\_x\_y\_1  
    - (Distance\_n1\_n2\_d1 ∧ Distance\_n2\_n3\_d2) → Distance\_n1\_n3\_(d1+d2)
6. **Reachability**: Constraints to define which nodes are reachable from a given starting node.
7. **Graph Properties (Strongly Connected, Undirected)**: Constraints to determine whether the entire graph satisfies certain properties like being strongly connected or undirected.

# Model Exploration

## Distance -1 Bug

I encountered a bug where the following constraint led to every distance proposition being true:

A screen shot of a computer code

Description automatically generated

To diagnose things, I first added constraints saying every node reaches itself with distance 0, and nothing else can:

A computer code on a black background

Description automatically generated

This led to the theory being unsolvable. I tried printing the constraints being added, but it just looked like a mess. So, instead, I took a step back and tried to draw out the situation. The diagram ultimately led to my understanding of what was wrong, and looked like this:



My issue was with the for-loops and where I was using the n1, n2, n3 in the constraints. The revised and working code was this:

A computer screen with many colorful text

Description automatically generated with medium confidence

Interestingly, this caused the number of solutions with my first example graph to plummet from 80,190 to just 1. This makes intuitive sense, since there should be only a single solution when the graph is pre-determined.

## Strongly Connected

The idea with this property was to not use the Reachable proposition, but rather the distance ones. The reason for this is that the Reachable propositions may all be set true to satisfy those constraints – it helps to force a graph to be disconnected, but can’t be used to check it.

Consequently, this is how the strongly connected property was built:

A computer screen with text

Description automatically generated

Most interestingly for exploration of the model, the last two constraints tied together this “STRONGLY\_CONNECTED” proposition with the constraints that define the graph property. This notion led to the creation of a new type of propositions to explore things (and constrain it if I want):

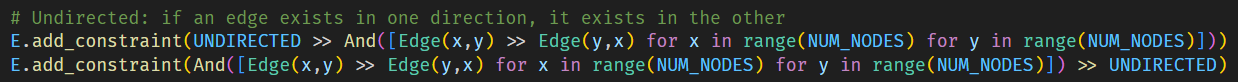
A computer screen shot of text

Description automatically generated

Also, as part of testing this property, I added a new graph that was a cycle on 4 nodes (which is strongly connected). This pattern will allow for other properties to be used as well.

## Undirected Bug

As another property, I tried to add a new proposition that indicated if a graph was undirected (every edge implies that the opposite edge exists). This was the first attempt:



…but it unfortunately didn’t fix the proposition. Every time it was run, it would randomly set to true or false, *even though it was running on a fixed graph of four nodes in a cycle.*

From this point, I looked at what the encoding object offered to print the constraints, and used “introspect”:

A black background with white text

Description automatically generated

Looked good, but still way too many constraints. Filtering just for “undirected” listed this:

A black background with many numbers

Description automatically generated with medium confidence

From this point, it was clear that the variables were named incorrectly, and I wasn’t using the right Edge objects. After fixing the edge creation (using parameters such as “f'n{x}',f'n{y}'”), it seemed to work and I tested this by re-running it several times and watching for the property. It rarely came up as a randomly generated undirected graph, and I manually confirmed this was correct from the adjacency lists.

## Exploring Likelihoods

With things in place, I was ready to start exploring the likelihood of a graph having certain properties:

A black screen with blue and orange text

Description automatically generated

A black background with white text

Description automatically generated

Interestingly, we can see that about half of the undirected graphs are completely disconnected – if they are connected, and undirected, then it should be strongly connected as well.

## { *Note From Prof* }

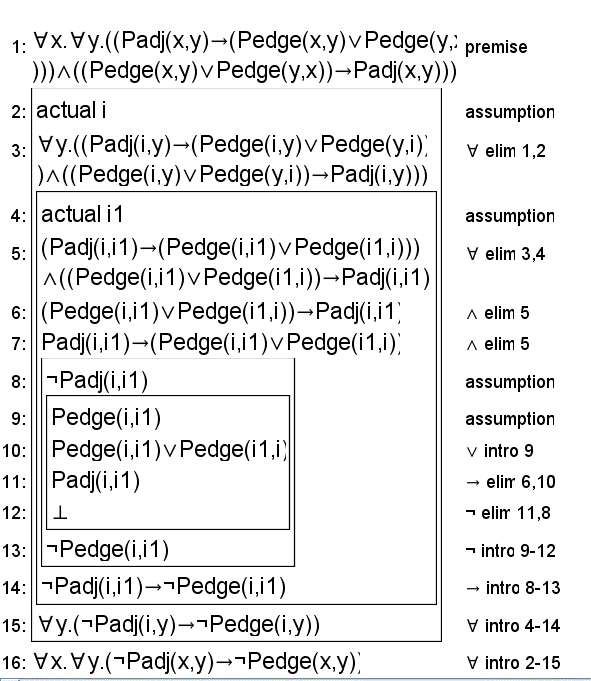
*The content in this section is precisely the type of things we’re looking for. It’s about 70-80% complete of what we’d expect from an A+ project, with more exploration of solutions and bugs being missing.*

# Jape Proof Ideas

## If there are a pair of nodes that aren’t adjacent, then there is no edge between them

* If an edge exists, then it is adjacent (optionally, adjacent means edge either way)

∀x.∀y.((Padj(x,y)→(Pedge(x,y) ∨Pedge(y,x))) ∧ ((Pedge(x,y) ∨ Pedge(y,x))→Padj(x,y))) ⊢ ∀x.∀y.(¬Padj(x,y) → ¬Pedge(x,y))



## If a node has a self-loop, then it can always reach itself with one higher distance

This one is more of an exercise in setting up the premises correctly, and then just removing the quantifiers in the appropriate way.

### Premises

* Distance update:  
  ∀xn1.∀xn2.∀xn3.∀xdi.∀xdj.∀xdk.((Pdist(xn1,xn2,xdi) ∧ Pdist(xn2,xn3,xdj) ∧ Psum(xdi,xdj,xdk))→Pdist(xn1,xn3,xdk))
* Constant predicates: Pdist(in, in, id1), Pdist(in0,in,idx), Psum(idx,id1,idy)
* Objects: actual in, actual id1, actual in0, actual idx, actual idy

### Conclusion

Pdist(in0,in,idy)

### Sequent

∀xn1.∀xn2.∀xn3.∀xdi.∀xdj.∀xdk.((Pdist(xn1,xn2,xdi) ∧ Pdist(xn2,xn3,xdj) ∧ Psum(xdi,xdj,xdk))→Pdist(xn1,xn3,xdk)), Pdist(in, in, id1), Pdist(in0,in,idx), Psum(idx,id1,idy), actual in, actual id1, actual in0, actual idx, actual idy ⊢ Pdist(in0,in,idy)

### Proof

A screenshot of a computer code

Description automatically generated

## Node with a self loop can reach itself in 3 steps

Similar to the previous, but requires smarter universal elimination to combine things appropriately.

∀xn1.∀xn2.∀xn3.∀xdi.∀xdj.∀xdk.((Pdist(xn1,xn2,xdi) ∧ Pdist(xn2,xn3,xdj) ∧ Psum(xdi,xdj,xdk))→Pdist(xn1,xn3,xdk)),  
Pdist(in, in, id1), Psum(id1,id1,id2), Psum(id2,id1,id3),  
actual in, actual id1, actual id2, actual id3  
 ⊢  
Pdist(in,in,id3)

A black text on a white background

Description automatically generated

# First-Order Extension

## Predicates

We will start with the propositions used for the base model, and extend to other predicates of interest.

* Edge(x,y): There is an edge from node x to node y
* Degree(x,n): Node x has degree n
* Connection(x,y,n): Node y is the nth connection starting at node x
* Connected: Graph is connected
* Distance(x,y,n): A path exists of length n from node x to y
* *Equality* : Implicitly lets us compare two objects for equality

To handle sorts, we would need the following predicates:

* Node(x): x is a node
* Number(n): n is a whole number

## Functions

* sum(n, n’): Function that returns an object that is the sum of n and n’
* sub(n, n’): Function that returns an object that is the subtraction of n and n’ (i.e., n – n’)

## Constraints

Some of the constraints we might have:

* If an edge exists between nodes, then their distance is 1
* If a node x can reach y with distance n, and y can reach z with distance n’, x can reach z with distance n+n’  
   )
* If distance\_x\_y\_n holds, then distance\_x\_z\_(n-1) must hold for some z connected to y
* If we can reach every node with some distance, then the graph is connected
* If a graph is connected, then between every pair of nodes there is a finite distance

# Conclusion

This report covers the type of elements we expect to see in a final report for a course project in CISC/CMPE 204. In particular, this project covered various properties one might explore on a graph, and how they interact with each other. While we might expect more sections to be expanded for an A+ project (e.g., the model exploration), this serves as a pretty solid example of a decent report.