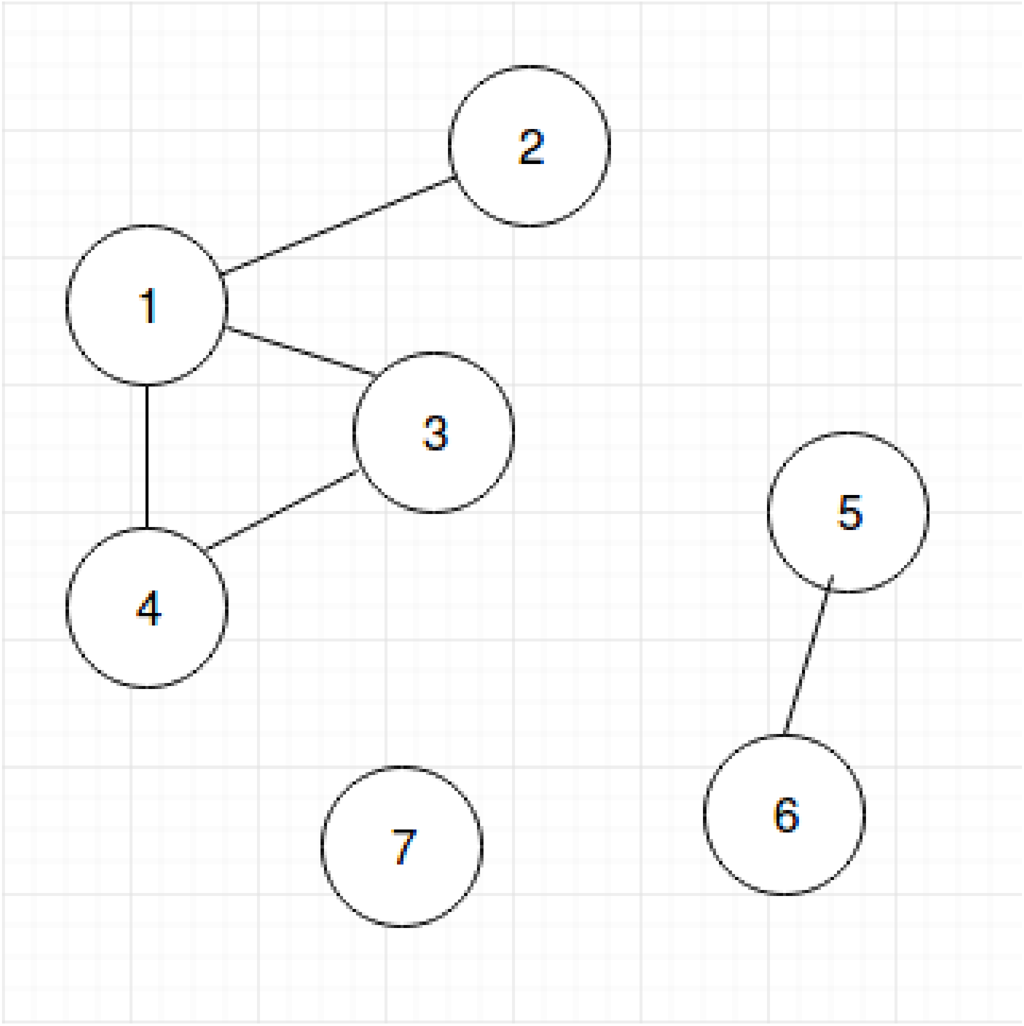
Connected Components

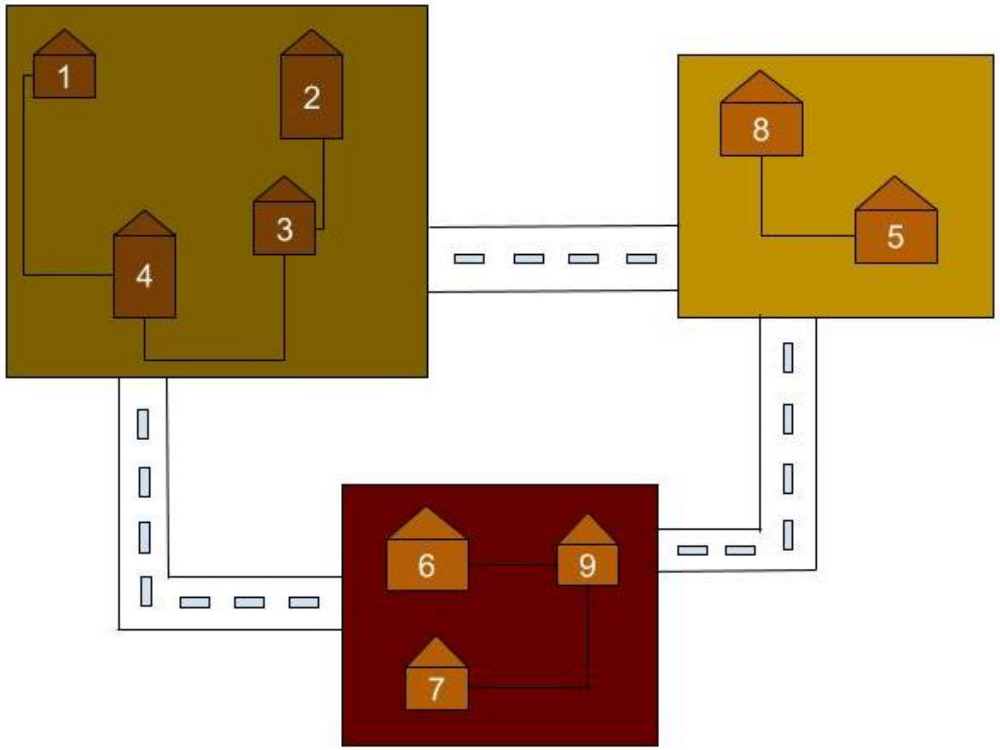
By: Ethan Church, Daniel Liao, Sia Puri



Sector Shuffle

- Billy is tasked with inspecting different buildings every day on Tesla's manufacturing plant
- The plant has an interesting layout.
  - Buildings are connected by two-way walkable paths
  - Cluster of connected buildings = One sector
  - All sectors are connected to others by road

GOAL: Given a list of buildings he must inspect, what is the minimum number of times he must drive between sectors.



About this problem

- **Inspiration:** The problem presents a world aspect and is realistic.
- **Illustration vs. Figure:** We chose a **figure** because layout of the plant may be hard to understand at first and the figure is essential in clarifying the problem statement.
- **Problem title:** "Sector Shuffle" explains the actions and goals of the problem.
- **Input constraints:** We wanted a realistic question while also creating some difficulty. So, we decided on 0 < buildings < 1000.
- **Input/output example:** The specific example shown is short and easy to understand. Ideal for a competition setting.

2.9 DIFFICULTY
Medium

Sample Input 1

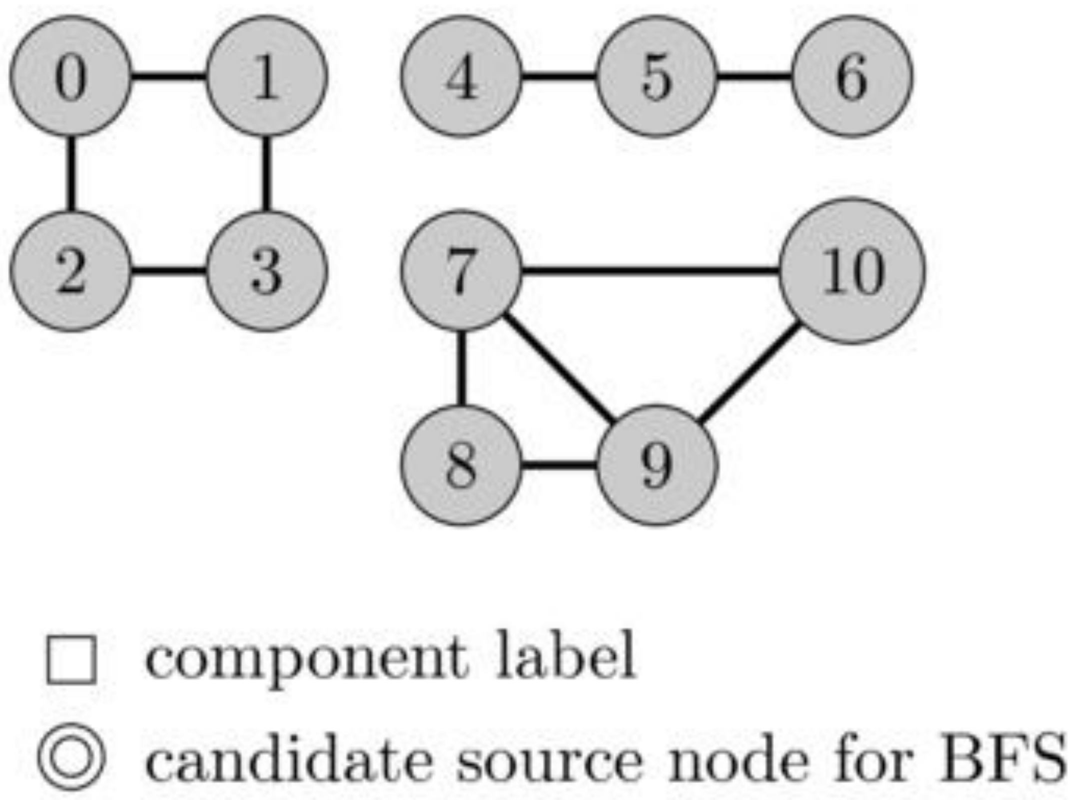
```
5
1 3 5
1 1 4
2 1 5
3 1 4
4 2 1 3
5 1 2
```

Sample Output 1

```
1
```

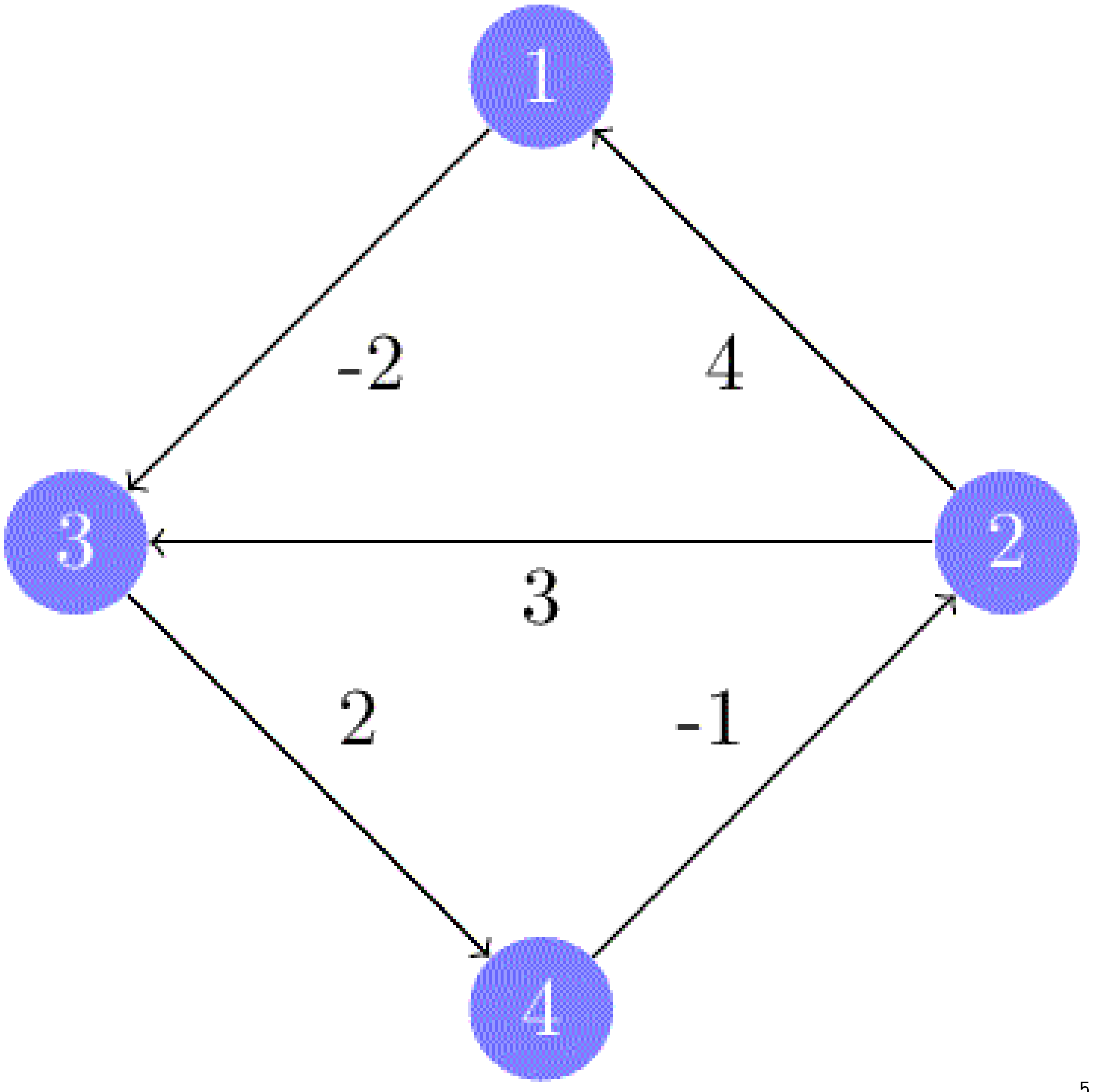
Correct Solution

- Implements a BFS algorithm with adjacency lists for O(V + E) complexity in Time and Space.
- C++ performed best among the three programming languages (then Python, Java)
- Best case scenario for a BFS is
  - All inspected buildings are isolated nodes with degree = 0: O(V + E + k)
  - All inspected buildings are in the same connected component: O(V + E)
  - Only 1 building to inspect: O(V + E)



Inefficient Solution

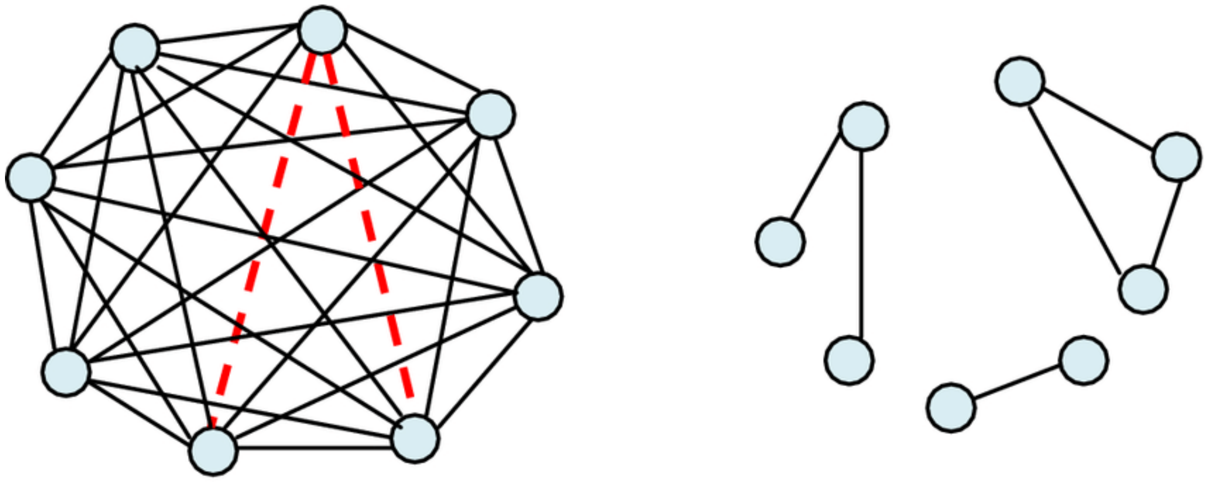
- Implements an Algorithm that checks the connectivity of all Building pairs
  - Runs in O(V^3) Time and O (V^2) Space complexity
  - Inspired by Floyd-Warshall style algorithms
- Solution yields a TLE



```python
def generate_test_case(filename, location):
    output_dir = os.path.join("data", location)
    os.makedirs(output_dir, exist_ok=True)

    n = random.randint(3, max_buildings)
    buildings = list(range(1, n+1))

    inspect_count = random.randint(1, n)
    to_inspect = random.sample(buildings, k=inspect_count)

    adjacency = {b: set() for b in buildings}
    for i in range(n):
        for j in range(i+1, n):
            if random.random() < 0.006:  # Creating a low percentage change of being connected so that buildings aren't always in the same sector.
                adjacency[buildings[i]].add(buildings[j])
                adjacency[buildings[j]].add(buildings[i])

```

Input and Output Files

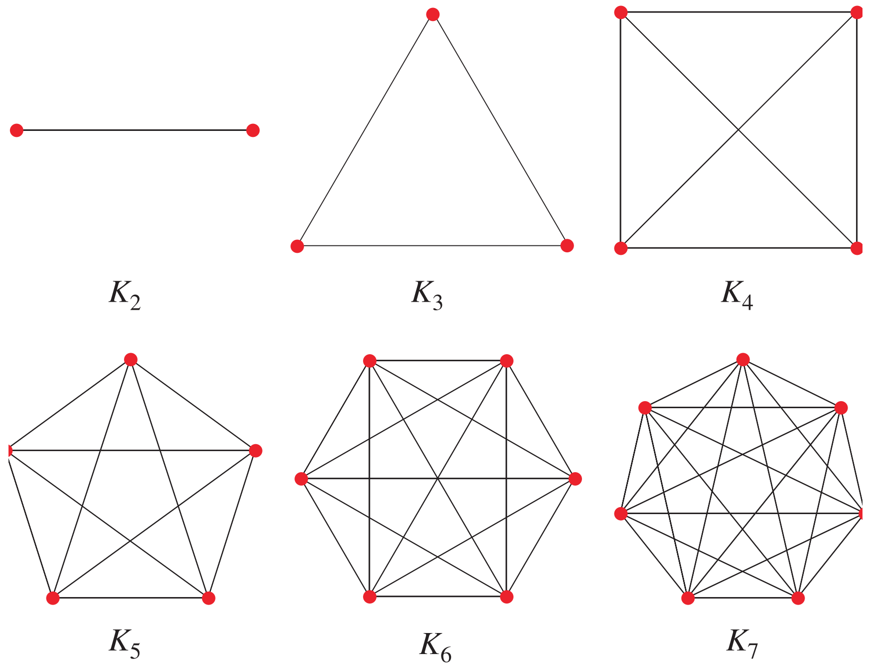
- **General Cases:**
  - Random number of buildings and random edges between buildings
  - Large and small number of buildings
  - Sparse and Dense graphs

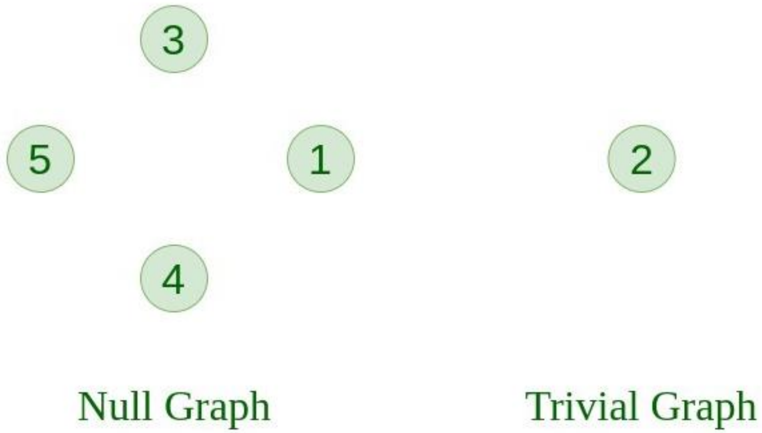


Dense Vs Sparse

Input and Output Files

- Edge Cases:
    - Every building connects with every other building (Complete Graph). Does the solution correctly identify one large component?
    - Every building does not connect to any other (Null Graph). Does the solution correctly handle zero edges?
    - Trivial Graphs with one or no buildings (To be implemented)





Test Case Generator

- The biggest challenge was understanding why each test case consistently produced an output of 0 and realizing that the issue was caused by the probability of buildings connecting being set too high
- Generators have different functions to generate Edge and General cases
- Corresponding output files were created by running our solution code on the input files generated by the Generator

```python
def generate_test_case(filename, location):
```

```python
def edge_case_only_one_sector(filename, location):
```

```python
def edge_case_all_sectors(filename, location):
```

```python
for i in range(1, 4):
    generate_test_case(f"test{i}", "sample")

for i in range(4, 25):
    generate_test_case(f"test{i}", "secret")

edge_case_only_one_sector("test25", "secret")
edge_case_all_sectors("test26", "secret")
```

Input Validator

- Challenging to create cogent and correct code in checktestdata format
- Validators handle regular and edge cases well, while rejecting malformed inputs:
  - Enforce integer size limits, uniqueness, etc.

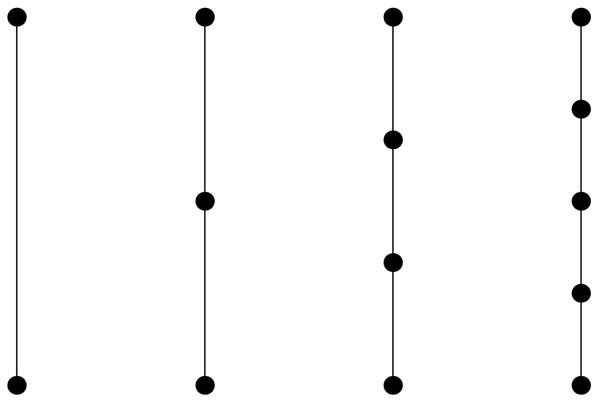
```
integer  := 0|-?[1-9][0-9]*
float    := -?[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)?
string   := ".*"
varname  := [a-z][a-z0-9]*
variable := <varname> | <varname> '[' <expr> [',' <expr> ...] ']'
value    := <integer> | <float> | <string> | <variable> | <function>
compare  := '<' | '>' | '<=' | '>=' | '==' | '!='
logical  := '&&' | '||'
expr     := <term> | <expr> [+-] <term>
term     := <factor> | <term> [*%/] <factor>
factor   := <value> | '-' <factor> | '(' <expr> ')' | <factor> '^' <factor>
test     := '!' <test> | <test> <logical> <test> | '(' <test> ')' |
            <expr> <compare> <expr> | <testcommand>
```

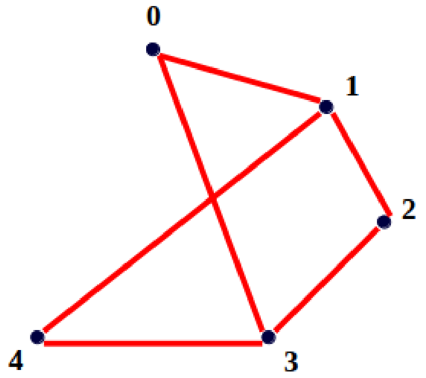
```
# Verify uniqueness of building IDs to inspect
assert len(inspect_ids) == len(set(inspect_ids)), "Building IDs to inspect must be distinct"
```

Presentation Design & Delivery

- The main strategy we used to effectively deliver our presentation was using **visual aids** to demonstrate the algorithms used in our correct and inefficient solutions
- **Graph problems** in particular benefit from visualization
- We also limited the amount of code in our presentation to keep the slides interesting

Takeaways





- This project challenged us creatively, forcing us to create unique test cases and incorrect solutions.
- We were able to branch out into different programming languages
  - Observed differences in efficiency using the same algorithmic paradigm in Python, C++, and Java
- With more time, we could create more comprehensive test cases including:
  - Path graphs
  - Cycles
  - Very Large Sparse Graphs

Thank You!