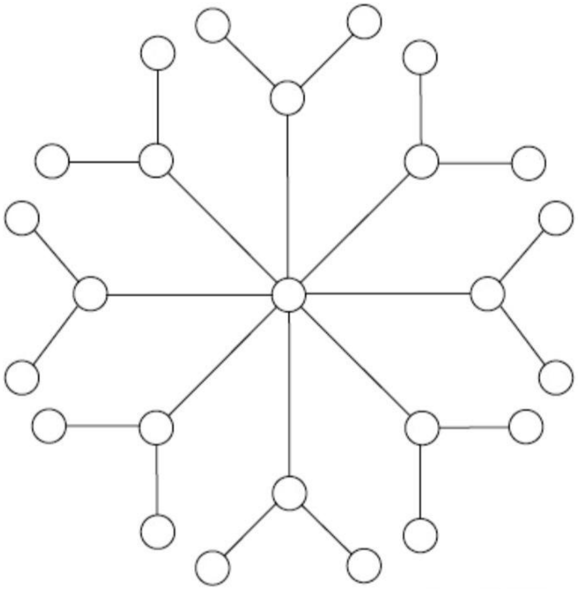
Vertex k-Labeling of Amalgamated Star Graphs Using Algorithmic Approach

- *Kylee Willis 16260630*
- *Suryansh Patel 16360312*
- *Blaine Steck 12324286*
- *Nathan Bailey 16179517*
- *Sowmya Reddy Adunuthula 16358422*
- *Akhilandaswari Tulasi Marisetti 16361218*
- *Yaswadeep Lanka 16365262*

Problem 1: Homogenous Amalgamated Star: $S_{n,3}$

The data structures for this graph are nested arrays

- The labels take the form of a list that begins with the centermost vertex and is followed by lists of each individual star/branch in the graph (e.g: if *a* is the centermost vertex, *b* is a centroid vertex, and *ba* and *bb* are pendant vertices)
- The array takes the following form: *[a, [b, ba, bb]]*
- The arrays containing weights have the same order as the vertex labels, with regard to their source vertex (e.g: if *a* is the centermost vertex, *b* is a centroid vertex, and *ba* and *bb* are pendant vertices)
- The array containing weights takes the following form: *[[a+b, b+ba, b+bb]]*



Algorithm 1 Design Strategy

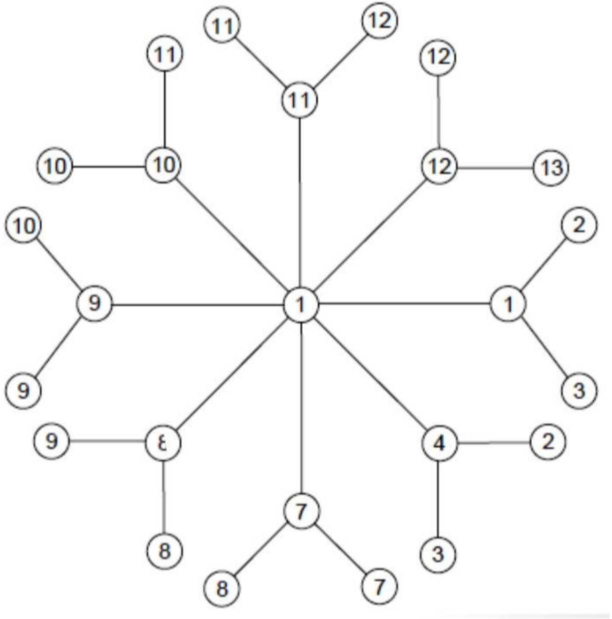
$$k = \left\lceil \frac{3^{*}n+1}{2} \right\rceil$$

- This algorithm is based on the *Edge irregular k-labeling of amalgamated stars* section from the provided research "Edge irregularity k-labeling for several classes of trees"
  - The center vertex is always labeled *1*
  - For centroid vertices where *1 ≤ i ≤ [ceil(n/4) + 1]*, vertex labels are calculated using the formula *(3i - 2)*
  - For centroid vertices where *[ceil(n/4) + 2]* ≤ i ≤ *n*, vertex labels are calculated using the formula *[2*ceil(n/4) + 1]*
  - For pendant vertices where *1 ≤ i ≤ [ceil(n/4)]* and *j = 1,2*, vertex labels are calculated using *(j + 1)*
  - For pendant vertices where *[ceil(n/4) + 1] ≤ i ≤ n* and *j = 1,2,* vertex labels are calculated using *[n + i + j - 1 - 2*ceil(n/4)]*

Graph Labeling/Traversal

- Traversal follows a greedy design strategy
  - Uses DFS (depth-first search); each individual star/branch is explored fully before moving to the next
- Labeling:

```
k value 16
lables [1, [1, 2, 3], [4, 2, 3], [7, 2, 3], [10, 8, 9], [11, 9, 10], [12, 10, 11], [13, 11, 12], [14, 12, 13], [15, 13, 14], [16, 14, 15]]
weights : [[2, 3, 4], [5, 6, 7], [8, 9, 10], [11, 18, 19], [12, 20, 21], [13, 22, 23], [14, 24, 25], [15, 26, 27], [16, 28, 29], [17, 30, 31]]
is valid lables: True
```



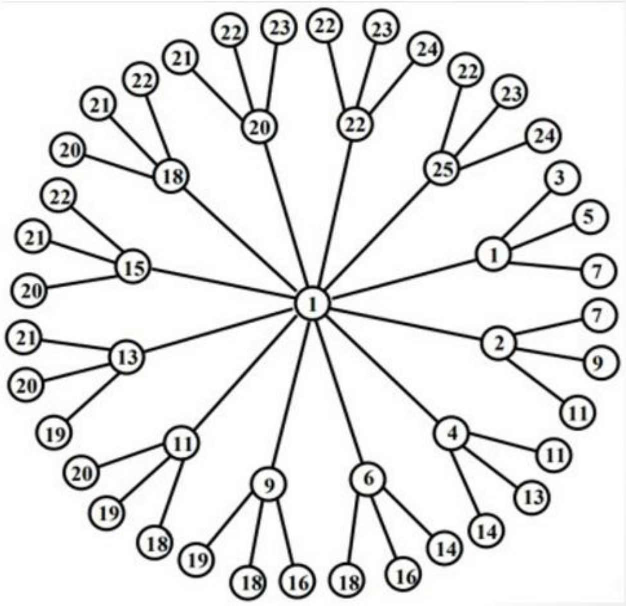
Algorithm 1 Pseudocode

```
Function edge_irregular_labeling(n):
  k = (3*n + 1) // 2                              O(1)
  labels = {}                                     O(1)

  if n % 4 == 0 or n % 4 == 2 or n % 4 == 3:      O(1)
    for i from 1 to n:                            O(n)
      if i <= floor(n/4) + 1:                     O(1)
        labels[i] = 3*i - 2
      else:
        labels[i] = 2*floor(n/4) + i

    for i from 1 to n:                            O(n*2)
      if i <= floor(n/4):
        for j from 1 to 2:                         O(n*m)
          labels[(i, j)] = j + 1
      else:
        for j from 1 to 2:
          labels[(i, j)] = n + i + j - 1 - 2*floor(n/4)

    return labels                                 O(1)
  else:
    return "Invalid n value for the given conditions"
```

Complexity: O(n)

Problem-2: Homogenous amalgamated Star : $S_{n,m}$

- Data Structure
  - The data structures for this graph are nested arrays.
  - The labels take the form of a list that begins with the centermost vertex and is followed by list of each individual star/branch in the graph (e.g: if *a* is the centermost vertex, *b* is a centroid vertex, and *ba* and *bb* are pendant vertices)
  - The array takes the following form: *[a, [aa, ba, bb]]*
  - The arrays containing the weights have the same order as the vertex labels, with regard to their source vertex (e.g: if *a* is the centermost vertex, *b* is the centroid vertex, and *ba* and *bb* are pendent vertices)
  - The array containing weights takes the following form: *[[a+b, b+ba, b+bb]]*



Algorithm 2 Design Strategy

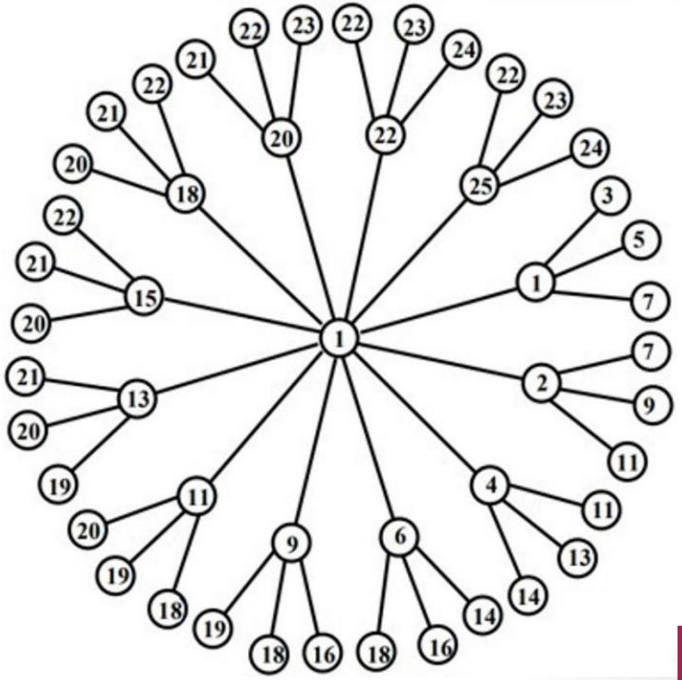
$$k = \lceil \frac{m^*n+1}{2} \rceil$$

- Based on Algorithm 4 from the provided research “Computing Edge Irregularity Strength of Complete M-ary Trees Using Algorithmic Approach”
  - A greedy approach is taken for each centroid vertex, where “d” is calculated using d = (ceil(V / 2) / (m - 1))
  - The center vertex, and first centroid are labeled with “1”
  - Each centroid vertex from i = 1 to i = n - 1 is labeled according to *floor(d * i)*
  - The final centroid vertex is labeled with “k”, where “k” is the maximum valid label for a vertex
    - Calculated with *k = ceil((m * n + 1) / 2)*
  - Pendant vertices are calculated by initially labeling the vertex with the minimum edge weight in the partially constructed graph, and iterating through potential labels until the resulting edge weight is valid

Graph Labeling/Traversal

- Traversal follows a greedy design strategy
  - Uses DFS (depth-first search); each individual star/branch is explored fully before moving to the next star/branch
- Labeling:

```
k value 25
labels : [1, [1, 3, 5, 7], [2, 7, 9, 11], [4, 11, 13, 14], [6, 14, 16, 18], [9, 16, 18, 19], [11, 18, 19, 20], [13, 19, 20, 21], [15, 20, 21, 22], [18, 20, 21, 2
], [20, 21, 22, 23], [22, 22, 23, 24], [25, 22, 23, 24]]
weights : [[2, 4, 6, 8], [3, 9, 11, 13], [5, 15, 17, 18], [7, 20, 22, 24], [10, 25, 27, 28], [12, 29, 30, 31], [14, 32, 33, 34], [16, 35, 36, 37], [19, 38, 39, 4
], [21, 41, 42, 43], [23, 44, 45, 46], [26, 47, 48, 49]]
is valid lables: True
```



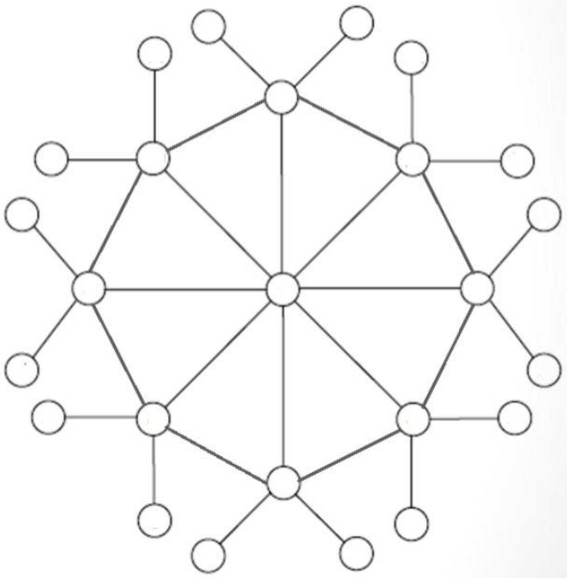
Algorithm 2 Pseudocode

```
Function generate_graph(n, m):
    labels = [1, [1]]                                   O(1)
    weights = [2]                                       O(1)
    v = m * n + 1                                       O(1)
    d = ceil(v / 2) / (n – 1)                           O(1)

    for i from 1 to n - 1:                              O(n)
        value = floor(d * i)                            O(1)
        labels.append([value])                          O(1)
        weights.append(value + 1)                       O(1)

    for i from 0 to n - 1:                              O(n)
        assumed_label = min(weights)                    O(n)
        for j from 0 to m - 2:                          O(m)
            while labels[i + 1][0] + assumed_label is in weights:      O(n*m)
                assumed_label += 1                      O(1)
            weights.append(labels[i + 1][0] + assumed_label)           O(1)
            labels[i + 1].append(assumed_label)         O(1)

    return labels, weights
```

Complexity: O(n*m)

Problem-3: For any number n as branches with centroid vertex

- Our chosen name for this data structure is "Homogeneous Trampoline"
- Order and size of graph
    - The order of the graph is determined with using $|V| = m * n + 1$
    - The size of the graph is determined using $|E| = (m + 1) * n$
        - The $(m + 1)$ term accounts for m branches, plus 1 to account for the additional edge linking each "branch" together



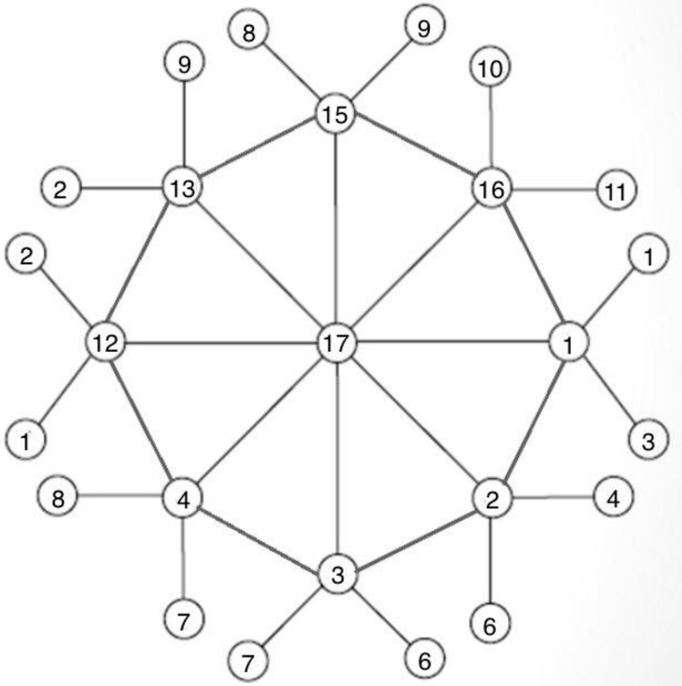
Data Structure

$$k = \left\lceil \frac{(m+1)^{*}n+1}{2} \right\rceil$$

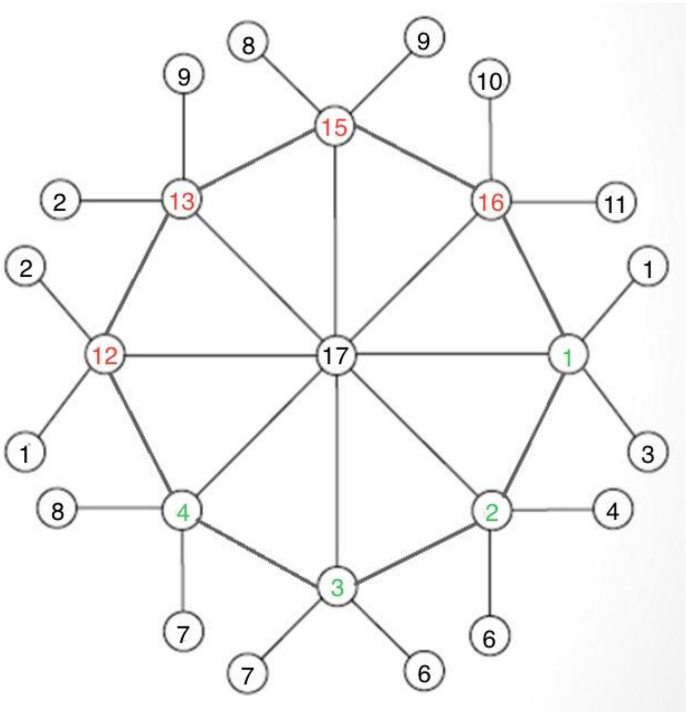
- Algorithm was adapted from Problem 2
- Vertex labels are stored in an array:
  - If *a* is the centermost vertex, *b* is a centroid vertex, and *ba* and *bb* are pendant vertices,
  - The array containing their labels would look like this: ***[a, [b, ba, bb]]***.
- Edge weights are stored in two separate arrays:
  - Non-centroid cycle weights work like the first two algorithms.
    - If *a* is the centermost vertex, *b* is a centroid vertex, and *ba* and *bb* are pendant vertices,
    - The array containing their weights would look like this: ***[[a+b, b+ba, b+bb]]***.
  - Cyclical weights are stored separately with a different structure.
    - If b and c are centroid vertices that are next to each other,
    - The array containing their cyclical weights would look like this: ***[a+b, (a, b)]***.

Assign the Labels

- Centermost vertex is labeled first
- Centroid vertices are labeled next
  - From i = 1 to n/2 (ceiling, inclusive) is found first
    - Start labels with 1, then work up
  - From i = n to n/2 (ceiling, exclusive) is then found
    - Start labels with k-1, then work down
  - Check edge weights with each label
    - If unused, it's fine
    - If used, increment/decrement label value and check again



Assign the Labels



- Pendant vertices get labeled, beginning with those connected to i=1
  - Retrieve lowest unused edge weight
  - Subtract centroid label from edge weight
  - Label pendant

- Output:

```
k value:         17
labels:          [17, [1, 1, 3], [2, 4, 6], [3, 6, 7], [4, 7, 8], [12, 1, 2], [13, 2, 9], [15, 8, 9], [16, 10, 11]]
weights:         [[18, 2, 4], [19, 6, 8], [20, 9, 10], [21, 11, 12], [29, 13, 14], [30, 15, 22], [32, 23, 24], [33, 26, 27]]
cycle weights:   [[3, (1, 2)], [5, (2, 3)], [7, (3, 4)], [16, (4, 12)], [25, (12, 13)], [28, (13, 15)], [31, (15, 16)], [17, (16, 1)]]
is valid labels: True
```

Algorithm 3 Pseudocode

```
generate_trampoline(n, m, k, e):
    Labels = [k]
    Weights = [e+1 values of False]
    Mid = ceiling(n/2)
    For i in range 1 to mid:
        For j in range i to k:
            If it's not used:
                Add j to labels
                Set weights[j+k] = True
                break
For i in range n to mid:
    For j in range k-1 to i-1:
        If it's not used:
            Add j to labels
            Set weights[j+k] = True
            break
```

Algorithm 3 Pseudocode

```
Prev = 2
For i in range 0 to n:
        For j in range 0 to m-1:
                Avail = -1
                For z in range prev to e:
                        If z is unused:
                                Avail = z
                                Prev = avail +1
                Calc = avail - centroid label
                Weights[avail] = True
                Append calc to labels[i+1]
```

Time Complexity

- Loop 1: O(mid * k)
- Loop 2: O((n - mid) * (k - i))
- Loop 3: O(n * m * (e - prev))
- So, the overall time complexity is approximately O(V + E), where V represents the number of vertices and E represents the number of edges in the graph.

```
prev = 2
for i from 0 to n - 1:                              O(n)
    for j from 0 to m - 2:                          O(m)
        low_avail = -1
        for z from prev to e:                       O(e-prev)
            if not used_weights[z]:
                low_avail = z
                prev = low_avail + 1
```

Thank you