Self Stabilizing Spanning Tree Algorithm Implementation

Tyler King
Kelsey Sandlin

Outline

- Introduction
- The Algorithm
- Correctness Proof
- Performance Proof
- Future Work
- Demo
- Questions

Introduction

- Spanning Trees and Connected Networks
  - G(V,E) into T(V,E')
- Self-Stabilization
  - Real world
  - Difficulties associated with it
  - Pros and Cons
- Related Work
- Benefits of our algorithm
  - Local Detection
  - Time
  - No prior knowledge

The Algorithm: Overview

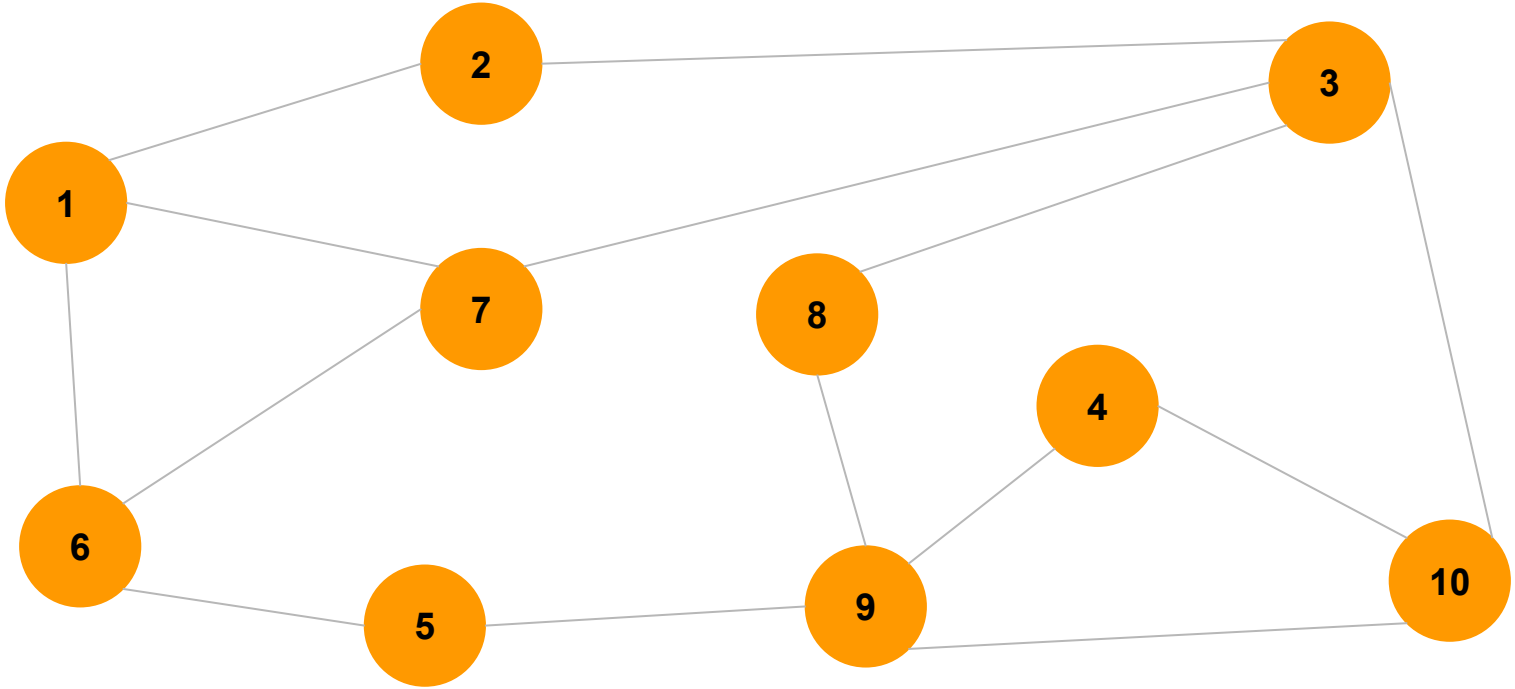
Each node maintains a priority, a parent pointer, distance from root and color.

Each node polls it’s neighbors and ranks them based on maximum priority and minimum distance from the root node.

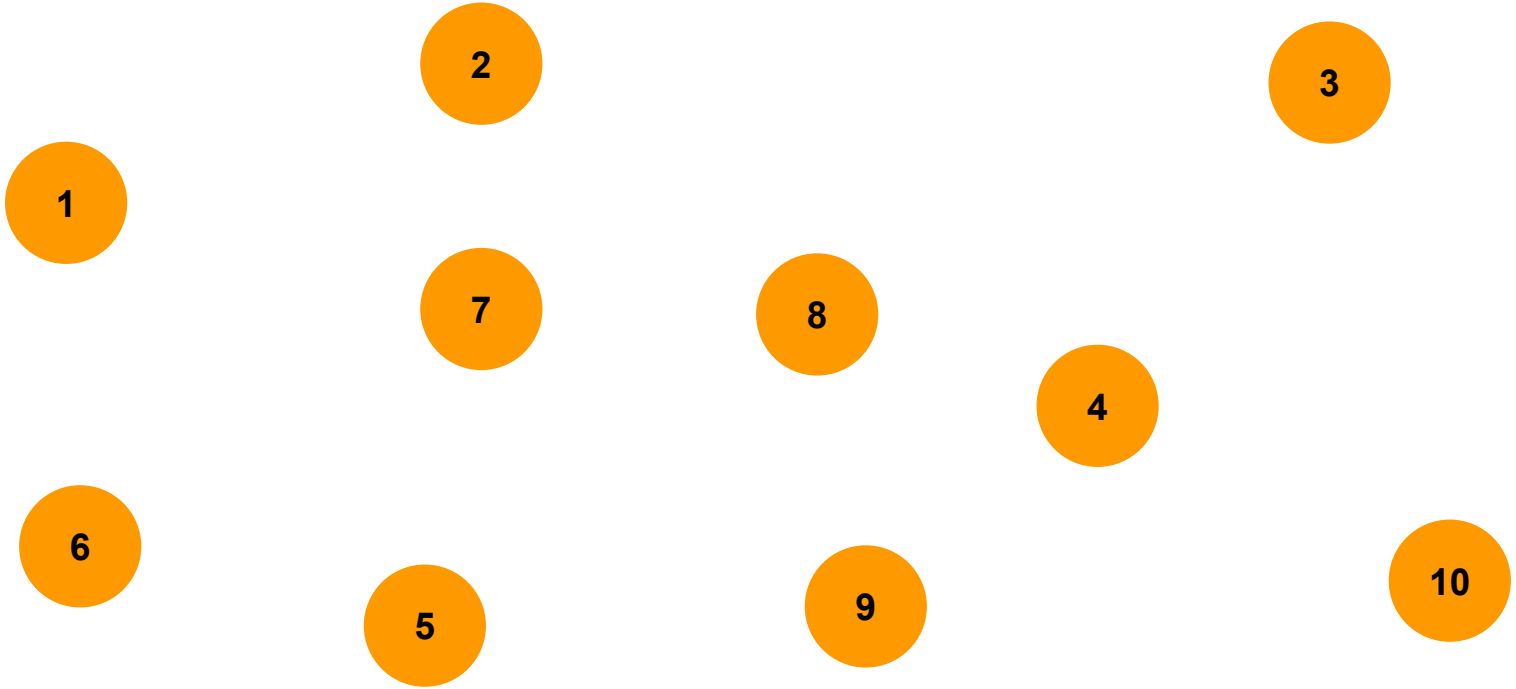
The highest ranked node becomes the current node’s parent, or if the current node posses the highest rank priority it becomes the root.

Many trees can exist in a system before stabilization; the trees gradually overrun each other until a single spanning tree exists.

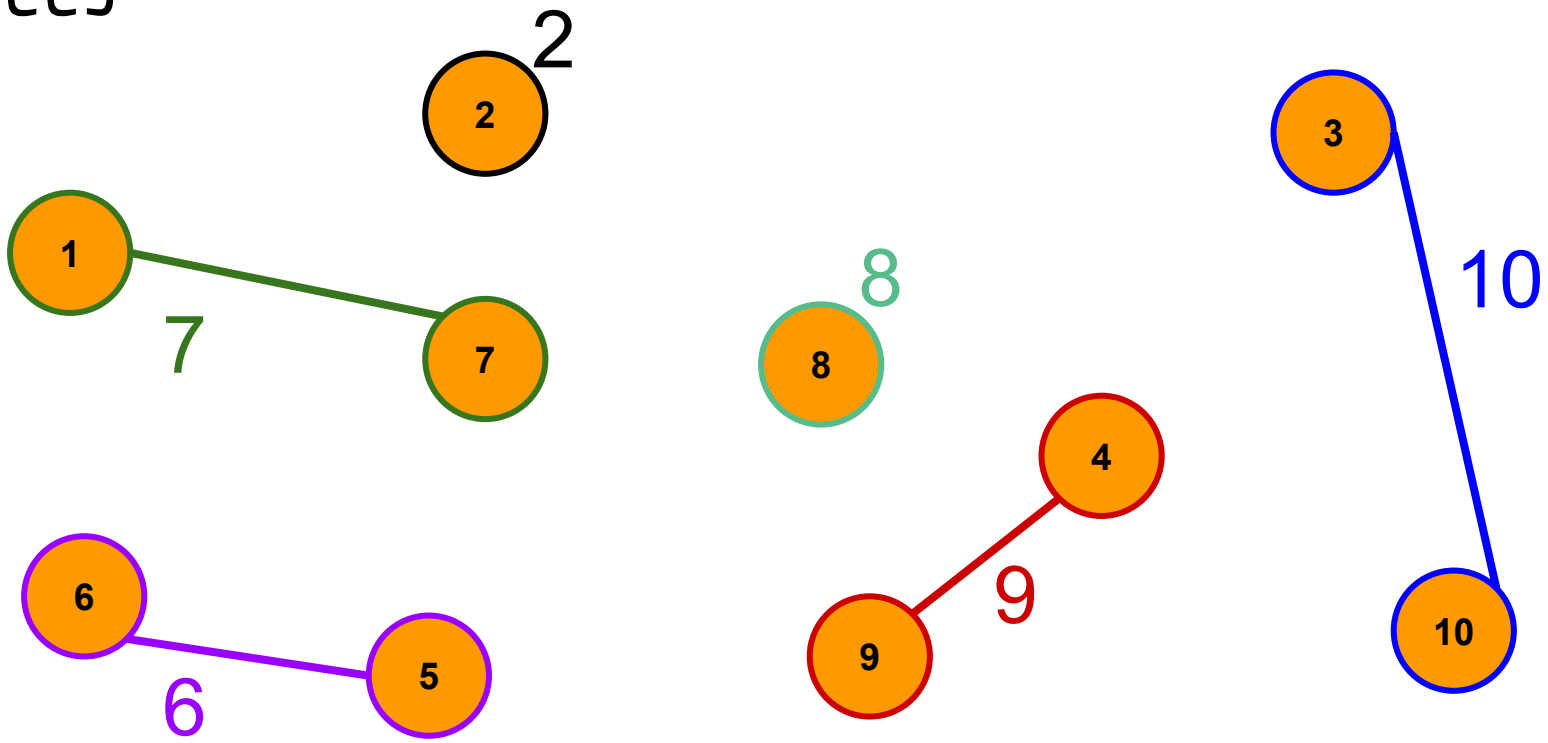
10 Node Network



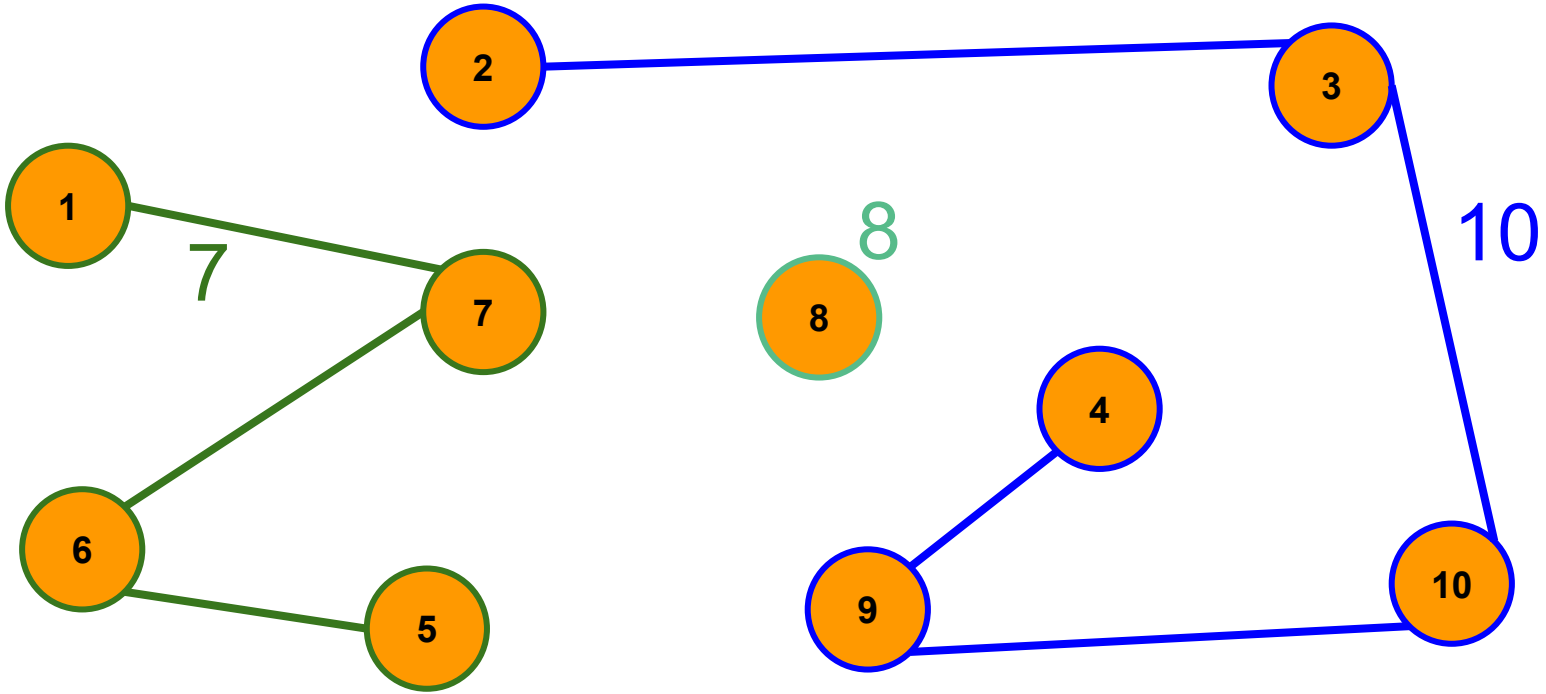
Initially: A Forest of 10 Trees



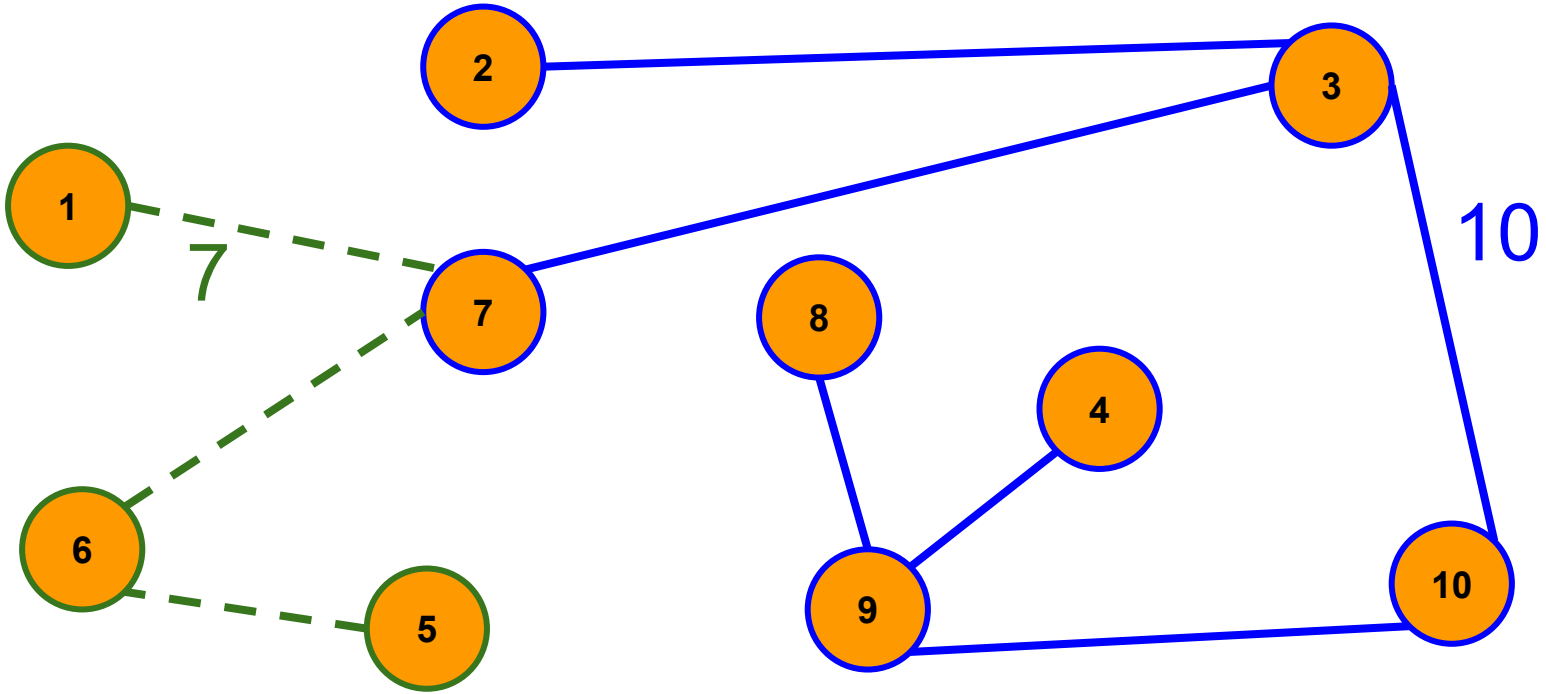
6 Trees



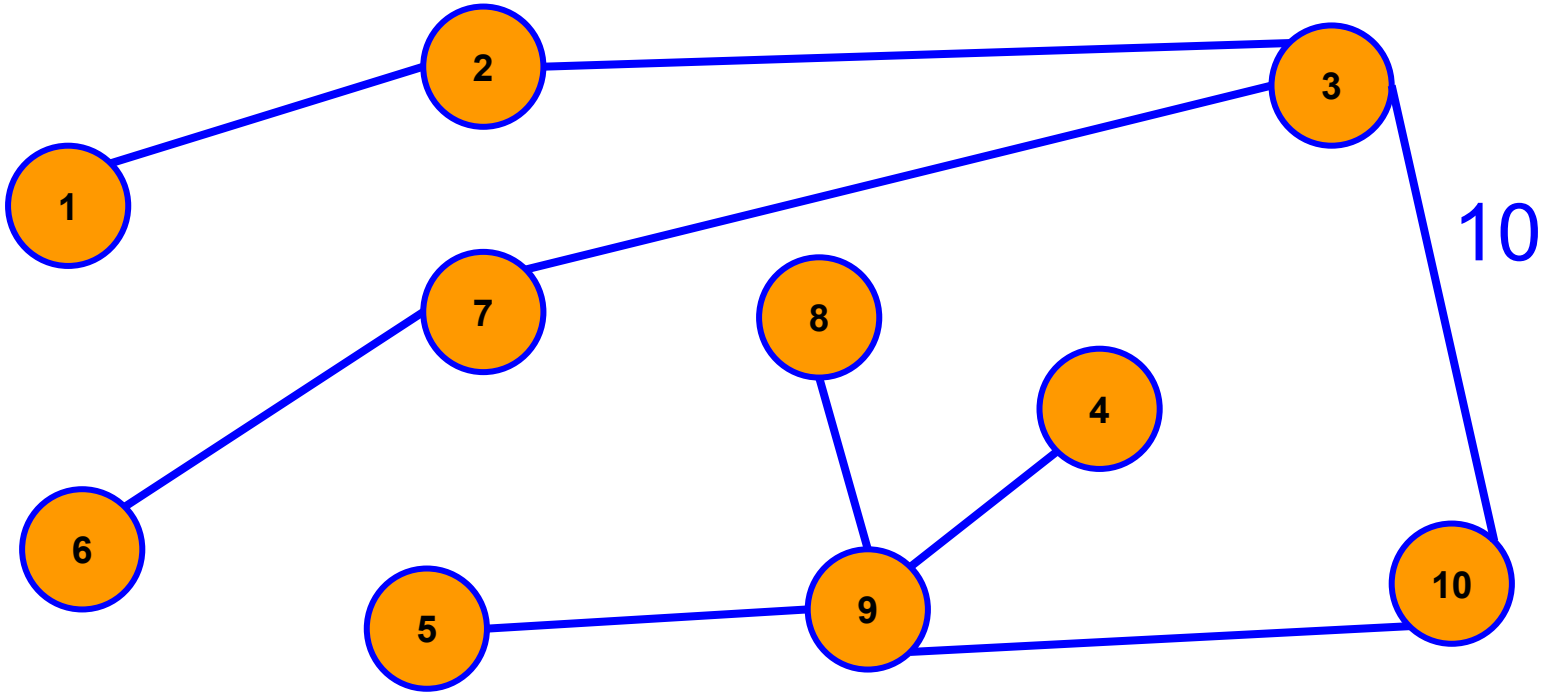
Trees overrunning -- highest priority (ID) wins



Trees overrunning



Stabilized



The Algorithm: Local Variables

```
// Needed for communication
String myHost;
int myPort;
ServerTable neighbors;

int ID;
TCPSocket tcpSocket;

// "shared" vars
priorityScheme priority;
int distance;
int parent;
int color;
boolean other_trees;
JSONObject neighbor_colors = new JSONObject();

// "local" vars
JSONObject neighbor_data;
```

The Algorithm: Execution

```
public void run() {
    while (true) {
        copy_neighbor_data();
        detect_trees();
        maximize_priority();
        next_color();
        extend_priority();
        try {
            Thread.sleep(2500);
        } catch (Exception e) {}
    }
}
```

There are five functions that run continuously to build or maintain the spanning tree

The Algorithm: Copy Neighbor Data

```
// copies neighbor data into local vars
//     & performs coloring tasks
void copy_neighbor_data() {
    for (int i=0; i<neighbors.size(); i++) {
        int ID_v = neighbors.get(i).ID;
        requestData(ID_v);
    }
}

// Sends request for data to v
void requestData(int ID_v) {
    String myData = this.packageSharedData(ID_v);
    this.tcpSocket._send(ID_v,"requestData " +
            this.ID + " " + myData);
}
```

Iterate through neighboring nodes, requesting their shared data.
Upon receiving data, determine if sending node is a part of current tree.

```
//receives data from v. copies to local vars. Coloring stuff
void receiveData(int ID_v, HashMap data) {
    this.neighbor_data.put(ID_v,data);
    priorityScheme priority_v = data.get("priority");
    int distance_v = data.get("distance");
    if ((this.priority.equals(priority_v.priority)) &&
        (Math.abs(distance - distance_v) <= 1)
    ) {
        //record color of neighbor if needed
        int color_v = data.get("color");
        if ((this.color != 0) && (color_v !=0)) {
            this.neighbor_colors.put(ID_v,color_v);
            if (this.color != color_v) {
                this.other_trees = true;
            }
        }
        // if parent, copy color
        if ((this.parent == ID_v) && (color != color_v)) {
            resetColor(color_v);
        }
    }
}
```

The Algorithm: Detect Trees

Examines each neighboring node and determines if it is in a different tree.

- Different priorities
- Difference in distance from root > 1
- Different color
- Child node with other_trees flag set

```
void detect_trees() {
    boolean same_tree = true;
    boolean color_same = true;
    boolean other_tree_detected = false;
    for (int i=0; i<neighbors.size(); i++) {
        int ID_v = neighbors.get(i).ID;
        HashMap data_v = (HashMap) this.neighbor_data.get(ID_v);
        priorityScheme priority_v = data_v.get("priority");
        int distance_v = data_v.get("distance");
        if ( (!(this.priority.equals(priority_v))) ||
            (Math.abs(this.distance-distance_v) > 1)
        ) {
            same_tree = false;
        }
        int color_v = data_v.get("color");
        if (color_v != this.color) {
            color_same = false;
        }
        int parent_v = data_v.get("parent");
        if (parent_v == this.ID) {
            if (data_v.get("other_trees") == true) {
                other_tree_detected = true;
            }
        }
    }
    if (same_tree) {
        if ((!(color_same) ||  other_tree_detected)) {
            this.other_trees = true;
        }
    }
}
```

The Algorithm: Maximize Priority

```
// become child of neighbor with max priority or become root
void maximize_priority() {
    int max_node = -1;
    priorityScheme max_priority = (-1);
    int max_distance = -1;
    int max_color = -1;
    // Get maximum priority and minimum distance
    for (int i=0; i<this.neighbors.size(); i++) {
        int ID_v = this.neighbors.get(i).ID;
        if (ID_v != this.ID) {
            HashMap data_v = (HashMap) this.neighbor_data.get(ID_v);
            priorityScheme priority_v = new priorityScheme(data_v.get("priority");
            int distance_v = data_v.get("distance");

            if ((priority_v.greaterThan(max_priority)) ||
                ((priority_v.equals(max_priority)) && (distance_v < max_distance))
            ) {
                max_priority = priority_v;
                max_distance = distance_v;
                max_node = ID_v;
                max_color = data_v.get("color");
            }
        }
    }
```

Rank neighboring nodes by maximum priority and minimum distance from root

Become child of highest ranked node become root

```
    // if u can improve its priority, by becoming child of another
    // neighbor, do so, otherwise become root
    if ((max_priority.greaterThan(this.priority)) ||
        ((max_priority.equals(this.priority)) && (max_distance < this.distance))
    ) {
        this.priority = max_priority;
        this.distance = max_distance + 1;
        if (this.parent != max_node) {
            System.out.println("=======> Node " + this.ID + " is child of " + max_node);
        }
        this.parent = max_node;

    } else {
        this.distance = 0;
        if (this.parent != -1) {
            System.out.println("=======> Node " + this.ID + " is root");
        }
        this.parent = -1;
    }
}
```

The Algorithm: Next Color

```
void next_color() {
    /* If root, choose new color if necessary */
    if ((this.parent == -1) &&
        (this.other_trees == false)
    ) {
        resetColor(this.ID);
    }
}

void resetColor(int newColor) {
    this.color = newColor;
    this.other_trees = false;
    for (int i=0; i<neighbors.size(); i++) {
        int ID_v = neghbors.get(i).ID;
        if (ID_v != this.ID) {
            this.neighbor_colors.put(ID_v,-1);
            HashMap data_v = (HashMap) this.neighbor_data.get(ID_v);
            data_v.put("self_color",-1);
            this.neighbor_data.put(ID_v, data_v);
            // if v is child, clear color
            if (data_v.get("parent") == ID) {
                data_v.put("color", -1);
                this.neighbor_data.put(ID_v, data_v);
            }
        }
    }
}
```

Root nodes use their ID to color trees.
Locally held child color data is invalidated proactively as children will be forced to either copy color or change parent nodes in the next round.

The Algorithm: Extend Priority

```
void extend_priority() {
    if ((this.parent == -1) &&
        (this.other_trees == true)
    ) {
        // always append UID
        this.priority.priority.add(this.ID);
        resetColor(this.ID);
    }

}
```

Use unique IDs to break any ties in priority between competing root nodes.

Correctness Proof

```
// become child of neighbor with max priority or become root
void maximize_priority() {
    int max_node = -1;
    priorityScheme max_priority = (-1);
    int max_distance = -1;
    int max_color = -1;
    // Get maximum priority and minimum distance
    for (int i=0; i<this.neighbors.size(); i++) {
        int ID_v = this.neighbors.get(i).ID;
        if (ID_v != this.ID) {
            HashMap data_v = (HashMap) this.neighbor_data.get(ID_v);
            priorityScheme priority_v = new priorityScheme(data_v.get("priority");
            int distance_v = data_v.get("distance");

            if ((priority_v.greaterThan(max_priority)) ||
                ((priority_v.equals(max_priority)) && (distance_v < max_distance))
            ) {
                max_priority = priority_v;
                max_distance = distance_v;
                max_node = ID_v;
                max_color = data_v.get("color");
            }
        }
    }
```

- Rank based on max priority and minimum distance
  - Priority is primary determinant and is extended by unique ID

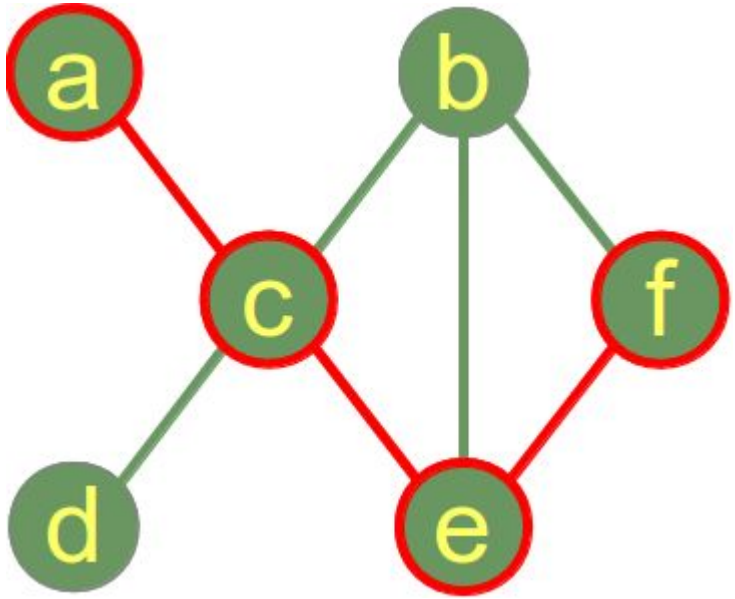
- Total order exists assuming unique IDs and continuous polling

```
    // if u can improve its priority, by becoming child of another
    // neighbor, do so, otherwise become root
    if ((max_priority.greaterThan(this.priority)) ||
        ((max_priority.equals(this.priority)) && (max_distance < this.distance))
    ) {
        this.priority = max_priority;
        this.distance = max_distance + 1;
        if (this.parent != max_node) {
            System.out.println("======> Node " + this.ID + " is child of " + max_node);
        }
        this.parent = max_node;

    } else {
        this.distance = 0;
        if (this.parent != -1) {
            System.out.println("======> Node " + this.ID + " is root");
        }
        this.parent = -1;
    }
}
```

Performance Proof

- A tree will emerge in O(D) rounds where D is the diameter of the graph
- At the end of the round, each node will have latest data
- The max rounds a node can remain oblivious to the existence of other trees is the distance from one end of the graph to the other as data propagates to each neighbor each round



Future Work?

- Verification of Stabilized tree
- Optimization of implementation
- Comparison of other Self-Stabilizing Spanning Tree Algorithms
- Analysis of longer-term highly volatile networks
- Prettier output

DEMO!

Questions?