RBE 510 – Multi-Robot Systems

Lecture 3: Modeling Robots and Distributed Algorithms

Kevin Leahy

August 29, 2025

Admin

- HW0 Due today

- HW1 is out
  – Take a look soon
  – There is a programming portion
  – Confirm that the example runs!

- Find partner/group of 3 for next class 9/2
  – Send me group info via email
  – Send me email if no group

Updates

- Office hours:
  - Wednesdays: 3 PM – 3:45 PM UH 250 D
  - As before, also by appointment

- Annotated lecture 2 from last year is posted

Recap

- Wrapped up consensus
  - Directed graphs
  - Time-varying topologies

- Considered formations
  - $\dot{x}_i = \sum_{j \in \mathcal{N}_i} a_{ij}(x_j - x_i - d_{ij})$
  - Conditions on d_{ij} to ensure equilibrium

Today

- Modeling multi-robot systems
- Networks and communication
- Lecture draws heavily from *Distributed Control of Robotic Networks* by Bullo, Cortés, and Martínez
- Taking off our controls hat (mostly) and putting on our computer science hat

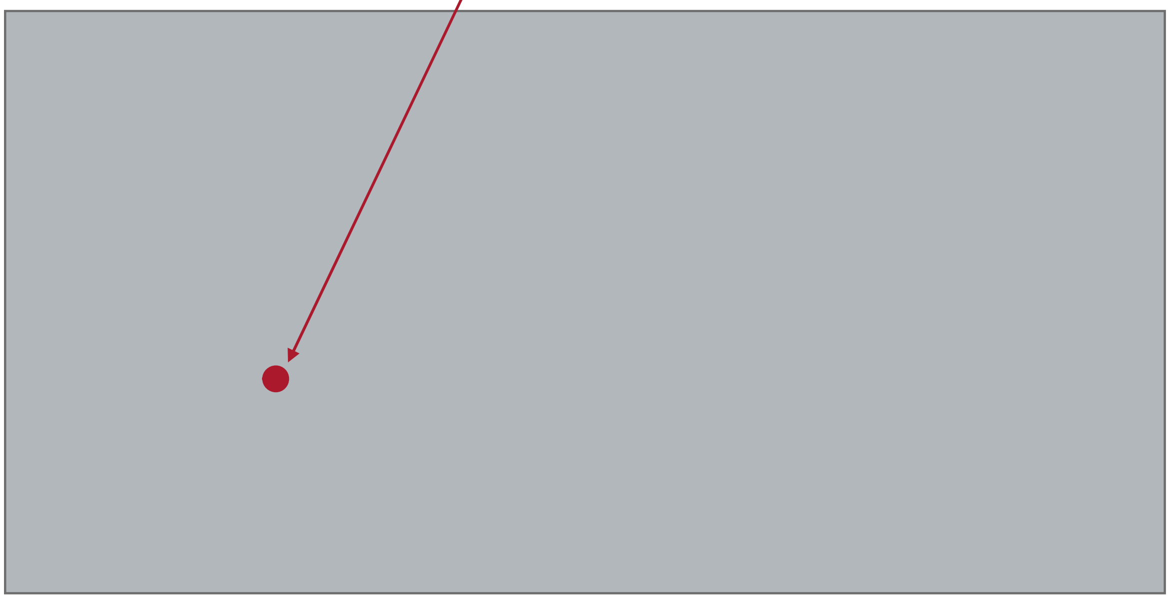
Modeling Robots

Modeling Robots

The World ($\mathbb{R}^2$ or a subset thereof–$D \subseteq \mathbb{R}^2$)

Modeling Robots

A single robot (sometimes "agent")



The World ($\mathbb{R}^2$ or a subset thereof–$D \subseteq \mathbb{R}^2$)

State $x \in \mathbb{R}^2$

Robot's position in the plane, plus associated state information (velocity, heading, acceleration, etc.)

Dynamics $\dot{x} = f(x, u)$

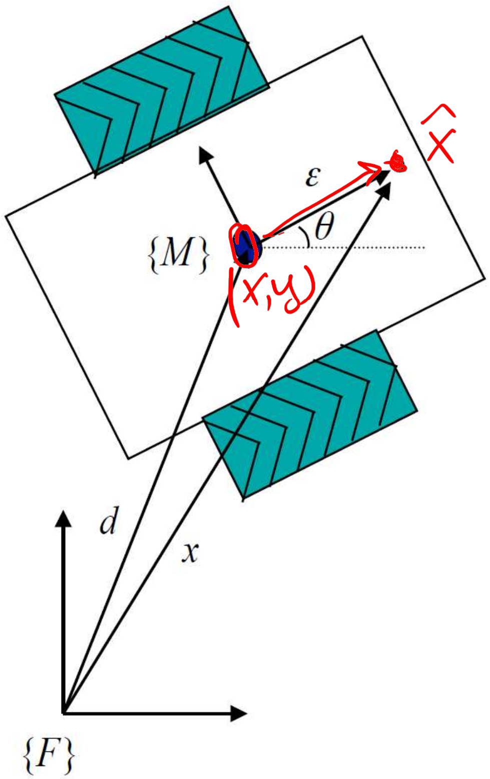
For now, $\dot{x} = u$
(single integrator)

Aside About Single Integrators

- Is the single integrator a good "general" model?
  - Easy to analyze
  - Doesn't represent difficult control regimes

- For some systems, we can construct a **reduced-order model**
  - Construct a controller that can track a simpler model
  - Then design a protocol that works for the simpler model

Reduced-Order Models for Ground Robots



- Underactuated system and nonholonomic!

$$\left\{\begin{aligned} \cancel{\dot{x}}\ \dot{d}_x &= u_s cos\theta \\ \cancel{\dot{y}}\ \dot{d}_y &= u_s sin\theta \\ \dot{\theta} &= u_\omega \end{aligned}\right.$$

$$\hat{x} = \begin{bmatrix} x+\varepsilon\cos\theta \\ y+\varepsilon\sin\theta \end{bmatrix}$$

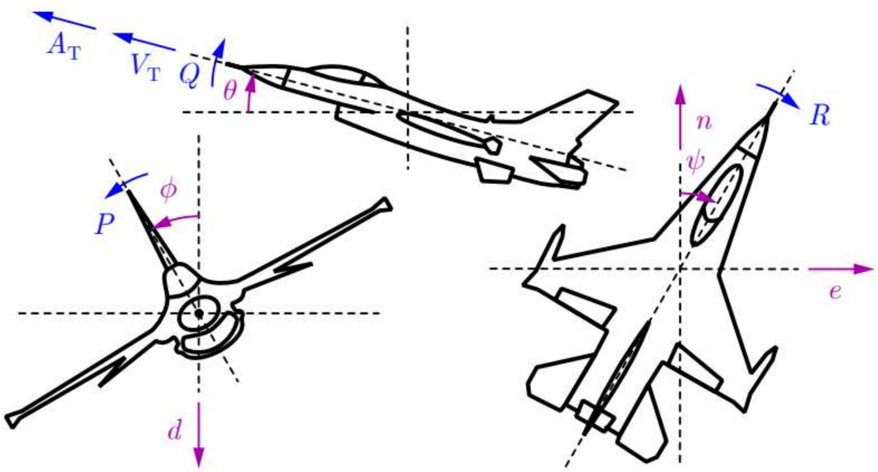
$$\dot{\hat{x}} = \begin{bmatrix} \dot{x} - \varepsilon\sin\theta\,\dot{\theta} \\ \dot{y} + \dot{\theta}\varepsilon\cos\theta \end{bmatrix}$$

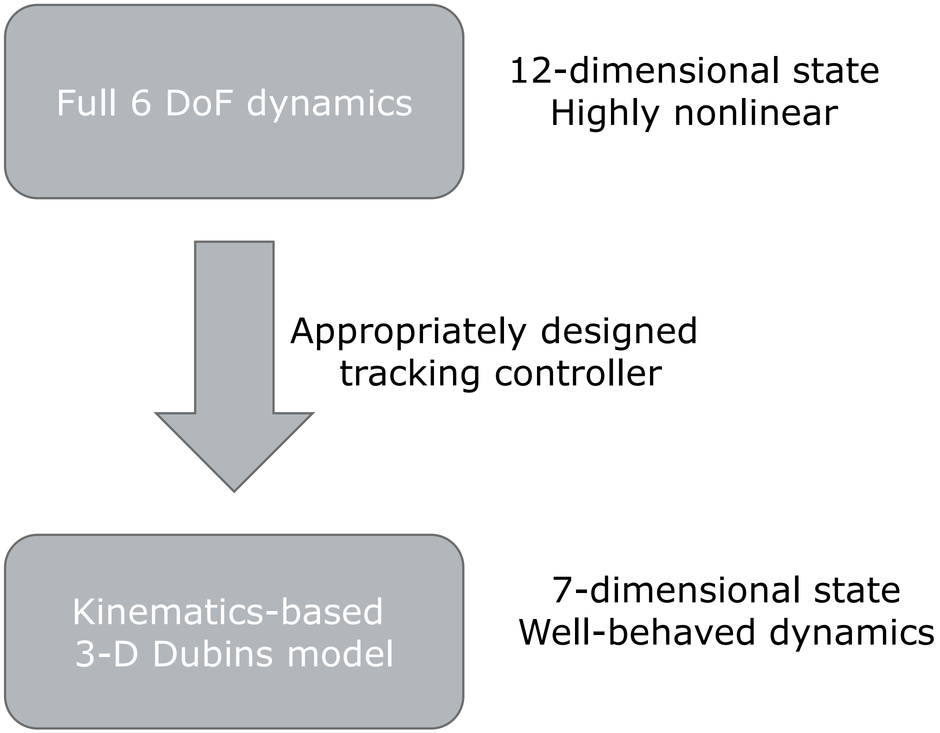
$$\dot{\hat{x}} = \begin{bmatrix} u_s\cos\theta - \varepsilon\sin\theta\, u_\omega \\ u_s\sin\theta + \varepsilon\cos\theta\, u_\omega \end{bmatrix}$$

$$= \underbrace{\begin{bmatrix} \cos\theta & -\varepsilon\sin\theta \\ \sin\theta & \varepsilon\cos\theta \end{bmatrix}}_{R_\theta} \begin{bmatrix} u_s \\ u_\omega \end{bmatrix}$$

J. Desai, J.P. Ostrowski, and V. Kumar. ICRA, 1998.

Reduced-Order Models for Complex Robots



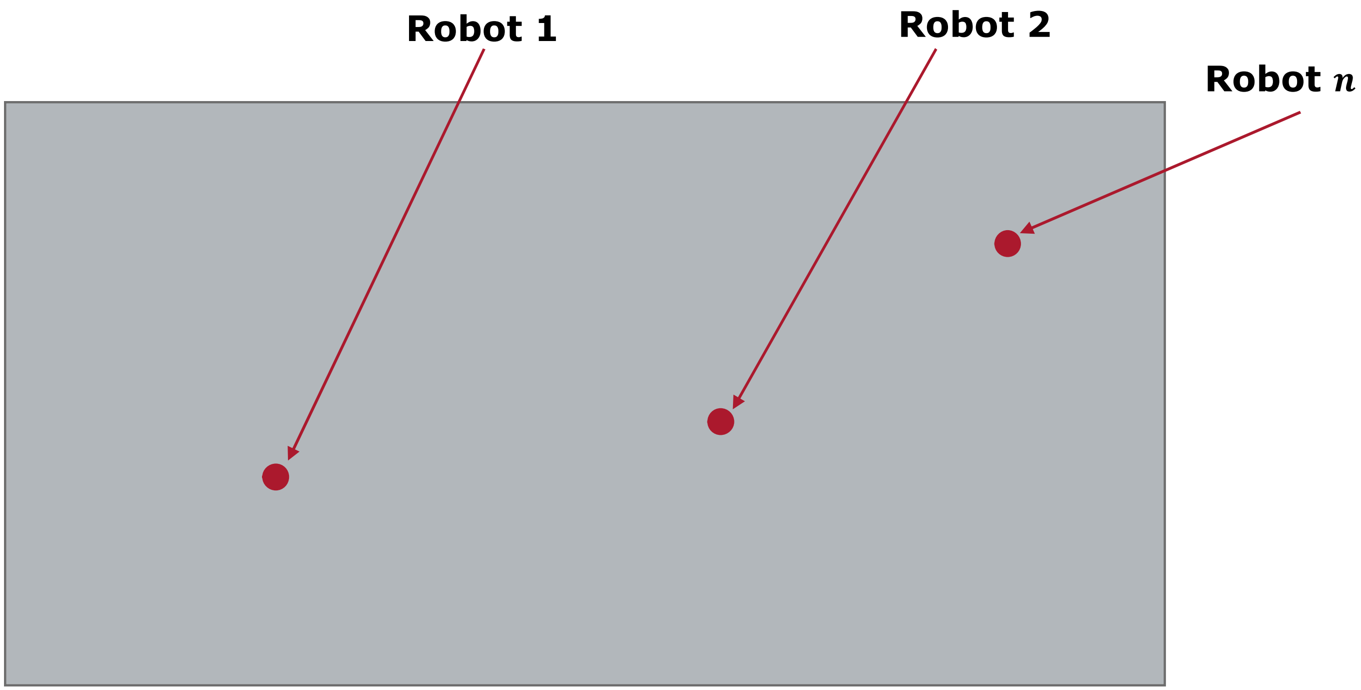


Molnar et al. “Collision Avoidance and Geofencing for Fixed-wing Aircraft with Control Barrier Functions”

Single integrators can do this...

...but not this

Modeling Multiple Robots



The World ($\mathbb{R}^2$ or a subset thereof–$D \subseteq \mathbb{R}^2$)

State $\boldsymbol{x} \in \mathbb{R}^{2n}$

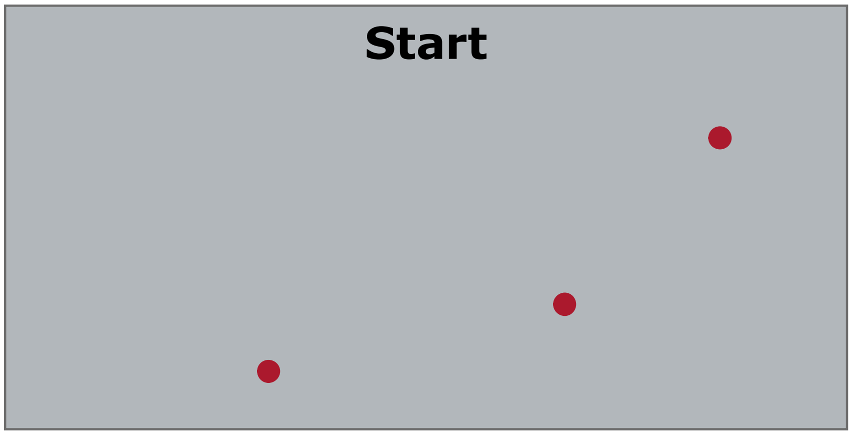
$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

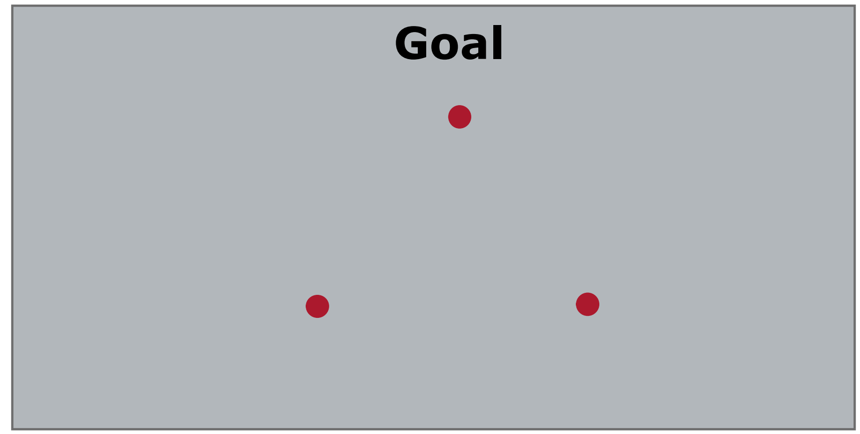
Dynamics $\dot{\boldsymbol{x}} = \boldsymbol{u}$

$$\dot{\boldsymbol{x}} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \vdots \\ \dot{x}_n \end{bmatrix}$$

$$\boldsymbol{u} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix}$$

Modeling Multiple Robots





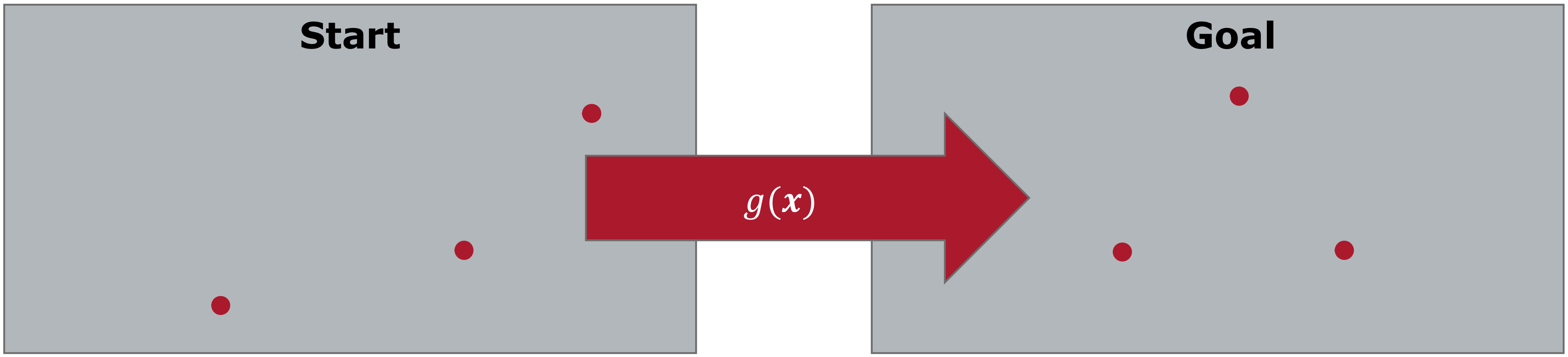
Example: From start configuration reach goal configuration

How: Design a feedback policy $\dot{\boldsymbol{x}} = \boldsymbol{g}(\boldsymbol{x})$

Where $\boldsymbol{g}(\boldsymbol{x}) = \begin{bmatrix} g_1(\boldsymbol{x}) \\ g_2(\boldsymbol{x}) \\ \vdots \\ g_3(\boldsymbol{x}) \end{bmatrix}$

Note, in general:
1. Agents may have individual feedback policies
2. Policies depend on multi-robot state (i.e., the *full* state)

Modeling Multiple Robots



How to design $g(\boldsymbol{x})$?
What are its properties for stability?
Convergence?
To what?
Under what conditions?

$g_3(\boldsymbol{x})$

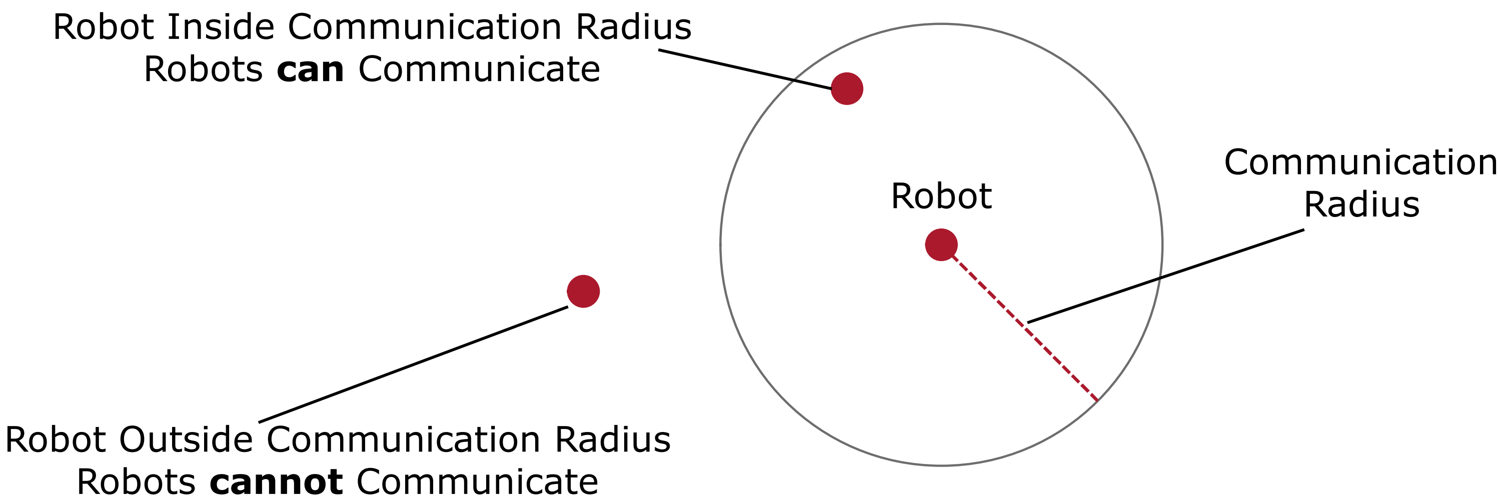
Modeling Robots

- How will we define a robot?
  - How it moves
  - How it senses
  - How it communicates
  - (Eventually) how it decides

Modeling Robots

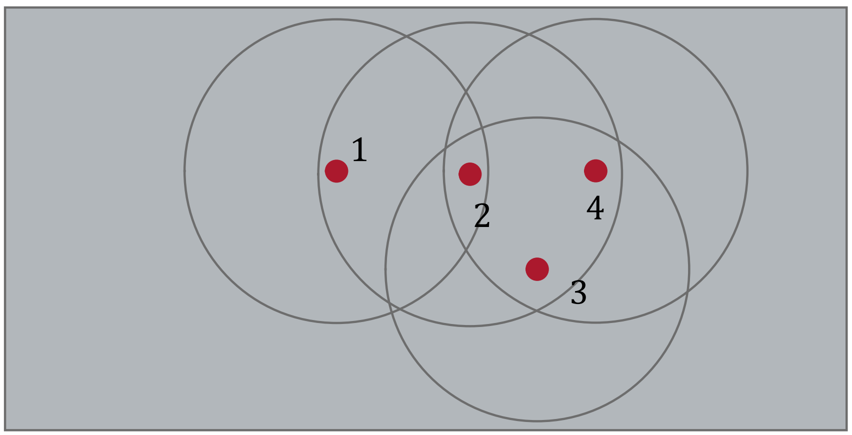
- How will we define a robot?
  - How it moves
  - How it senses
  - **How it communicates**
  - (Eventually) how it decides

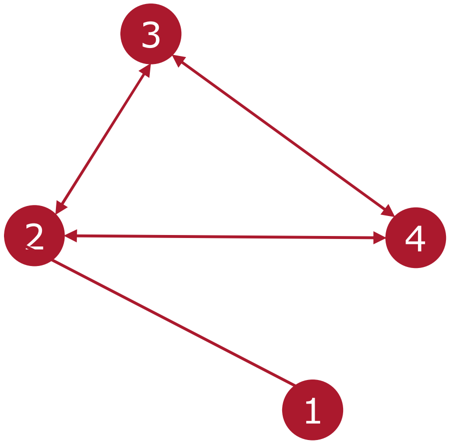
Modeling Communication



- Simple, radius-based method for modeling comms (for now)
- Robots in the radius of agent i are the **neighbors** of agent i, denoted $\mathcal{N}_i$

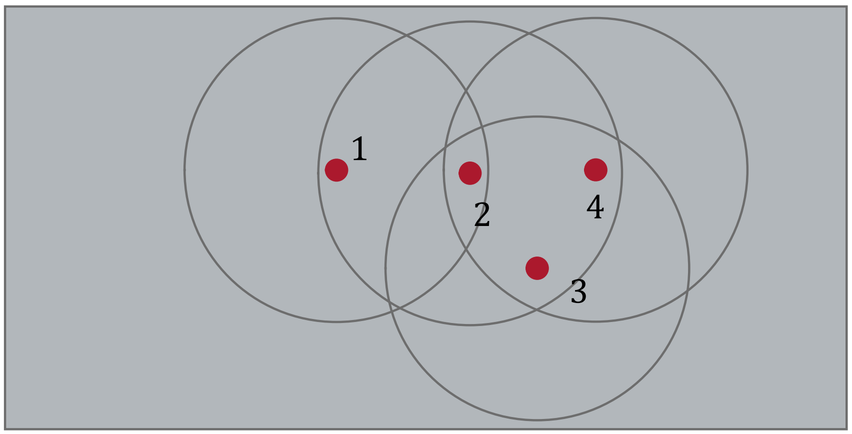
Modeling Communication



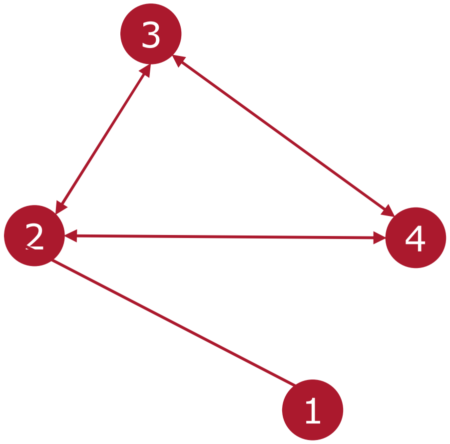


- If $i \in \mathcal{N}_j$ then $j \in \mathcal{N}_i$
- This induces an undirected graph $G = (V, E)$
- Now, $g_i(\boldsymbol{x}) = g_i(x_i, x_j \mid j \in \mathcal{N}_i)$
- *This is how decentralization happens*

Modeling Communication

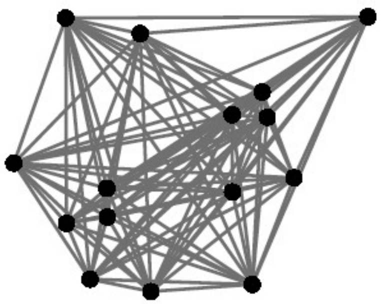


- Spatially embedded in $\mathbb{R}^2$
- Has dynamics

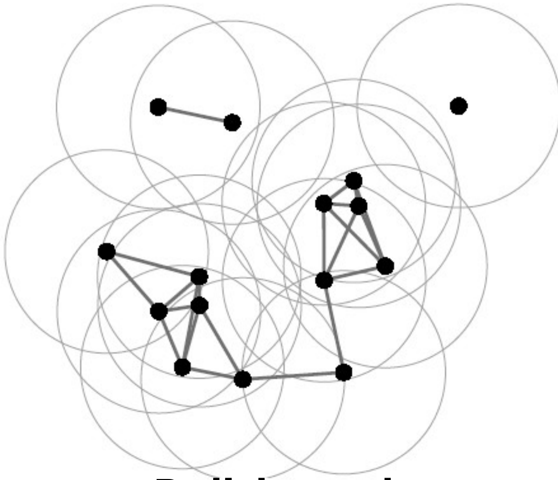


- Purely a combinatorial object (no geometry or dynamics)
- Information flow modeled via edges

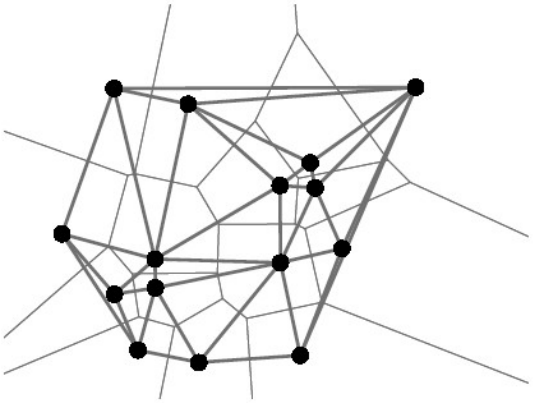
Other Networks



Complete Graph
Definitely connected
May suffer from congestion

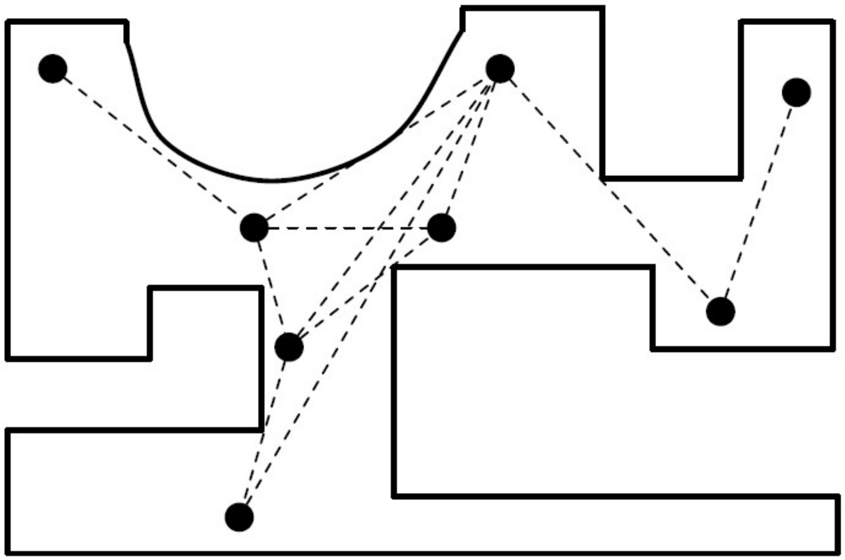


R-disk graph
Might be disconnected
Sparser

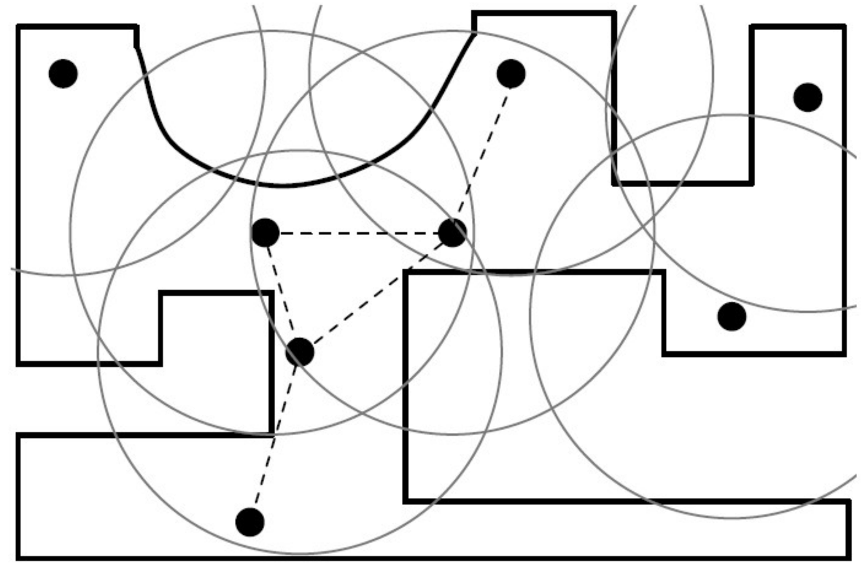


Delaunay Graph
Connected
Sparse

Other Networks



Visibility Graph



Range-Limited Visibility Graph

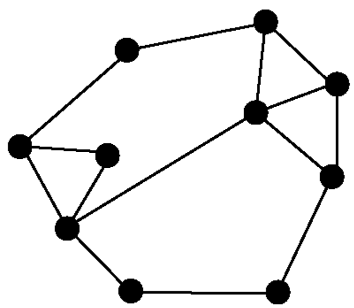
Models so far

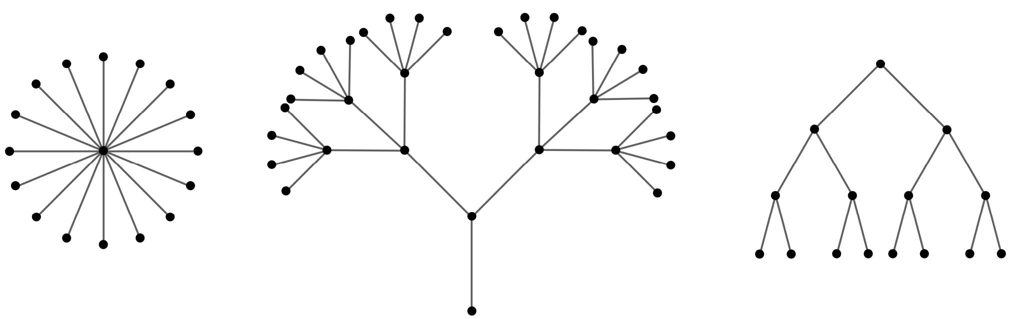
- So, there are two things we care about so far
  - Motion dynamics
  - Information processing via comms
- We often (but not always) treat them as the same thing
  - Agents communicate freely and update controls and so on and so forth
  - There are some subtle differences between their interactions, so let's model comms more precisely
- What can we compute over such a network?
  - Let's model as a "processor" state

Graph Algorithms

Trees

Graph



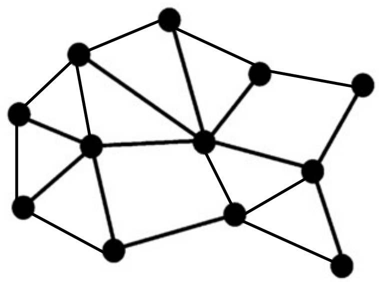


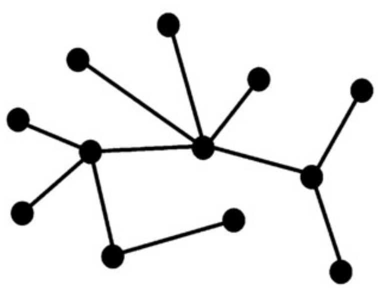
Trees

- A **tree** is an undirected graph in which any two vertices are connected by exactly one path
- Properties
  - Connected
  - Acyclic
  - For v vertices, there are $v-1$ edges

Spanning Tree

Graph



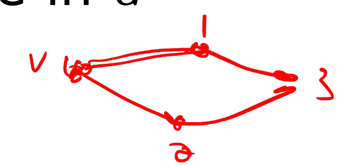


Corresponding Spanning Tree

- For an undirected graph G a **spanning tree** is a subgraph that
  - Is a tree
  - Includes all the vertices of G
- Useful for things like shortest-path computation
  - Used internally for Dijkstra, A^* algorithms
  - Also for things like telecommunication protocols, etc.

Example – BFS tree (Centralized)

- A **breadth-first spanning (BFS) tree** for a digraph G with respect to a node v, written T_{BFS} is a spanning directed tree rooted at v that contains the shortest path from v to every other node in G



1. Initialize subgraph as $(\{v\}, \emptyset)$
2. Attach all out-neighbors of the subgraph as well as a single edge to connect each out neighbor
3. Repeat step 2 until there are no out-neighbors to add

$(\{v\}, \emptyset)$

$(\{v, 1, 2\}, \{(v,1), (v,2)\})$

BFS Tree Formalized

```
function BFS(G, v)
  (V_1, E_1) := ({v}, ∅)
  for k = 2 to radius(v, G) do
      find all vertices w_1, ..., w_m not in V_{k-1} that are out-neighbors of
      some vertex in V_{k-1} and, for j ∈ {1, ..., m}, let e_j be an edge
      connecting a vertex in V_{k-1} to w_j
      V_k := V_{k-1} ∪ {w_1, ..., w_m}
      E_k := E_{k-1} ∪ {e_1, ..., e_m}
  return (V_n, E_n)
```

BFS Tree Formalized

`function` BFS(G, v)

$(V_1, E_1) := (\{v\}, \emptyset)$

for $k = 2$ to radius(v, G) **do**

  find all vertices $w_1, \ldots, w_m$ not in V_{k-1} that are out-neighbors of some vertex in V_{k-1} and, for $j \in \{1, \ldots, m\}$, let e_j be an edge connecting a vertex in V_{k-1} to w_j

  $V_k := V_{k-1} \cup \{w_1, \ldots, w_m\}$

  $E_k := E_{k-1} \cup \{e_1, \ldots, e_m\}$

return (V_n, E_n)

BFS Tree Formalized

This is a *global* property of the graph

$\texttt{function}\ \mathrm{BFS}(G, v)$

$(V_1, E_1) := (\{v\}, \emptyset)$

for $k = 2$ to radius(v, G) **do**

    find all vertices $w_1, \ldots, w_m$ not in V_{k-1} that are out-neighbors of some vertex in V_{k-1} and, for $j \in \{1, \ldots, m\}$, let e_j be an edge connecting a vertex in V_{k-1} to w_j

    $V_k := V_{k-1} \cup \{w_1, \ldots, w_m\}$

    $E_k := E_{k-1} \cup \{e_1, \ldots, e_m\}$

return (V_n, E_n)

BFS Tree Formalized

Find all adjacent nodes that haven't been added yet

`function` $\mathrm{BFS}(G, v)$

$(V_1, E_1) := (\{v\}, \emptyset)$
for $k = 2$ to radius(v, G) **do**
    find all vertices $w_1, \ldots, w_m$ not in V_{k-1} that are out-neighbors of some vertex in V_{k-1} and, for $j \in \{1, \ldots, m\}$, let e_j be an edge connecting a vertex in V_{k-1} to w_j
    $V_k := V_{k-1} \cup \{w_1, \ldots, w_m\}$
    $E_k := E_{k-1} \cup \{e_1, \ldots, e_m\}$
return (V_n, E_n)

BFS Tree Formalized

function $\mathrm{BFS}(G, v)$

$(V_1, E_1) := (\{v\}, \emptyset)$

for $k = 2$ to radius(v, G) **do**

  find all vertices $w_1, \ldots, w_m$ not in V_{k-1} that are out-neighbors of some vertex in V_{k-1} and, for $j \in \{1, \ldots, m\}$, let e_j be an edge connecting a vertex in V_{k-1} to w_j

  $V_k := V_{k-1} \cup \{w_1, \ldots, w_m\}$

  $E_k := E_{k-1} \cup \{e_1, \ldots, e_m\}$

return (V_n, E_n)

BFS Tree Formalized

```
function BFS(G, v)
    (V_1, E_1) := ({v}, ∅)
    for k = 2 to radius(v, G) do
        find all vertices w_1, ..., w_m not in V_{k-1} that are out-neighbors of
        some vertex in V_{k-1} and, for j ∈ {1, ..., m}, let e_j be an edge
        connecting a vertex in V_{k-1} to w_j
        V_k := V_{k-1} ∪ {w_1, ..., w_m}
        E_k := E_{k-1} ∪ {e_1, ..., e_m}
    return (V_n, E_n)
```

BFS Tree Formalized

`function` $\mathrm{BFS}(G, v)$

$(V_1, E_1) := (\{v\}, \emptyset)$

for $k = 2$ to radius(v, G) **do**

  find all vertices $w_1, \ldots, w_m$ not in V_{k-1} that are out-neighbors of some vertex in V_{k-1} and, for $j \in \{1, \ldots, m\}$, let e_j be an edge connecting a vertex in V_{k-1} to w_j

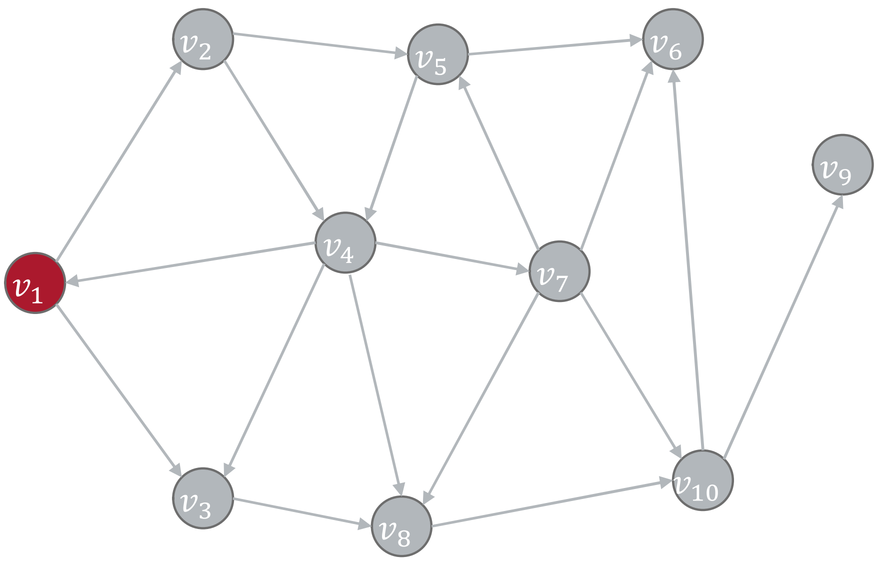
  $V_k := V_{k-1} \cup \{w_1, \ldots, w_m\}$

  $E_k := E_{k-1} \cup \{e_1, \ldots, e_m\}$

return (V_n, E_n)

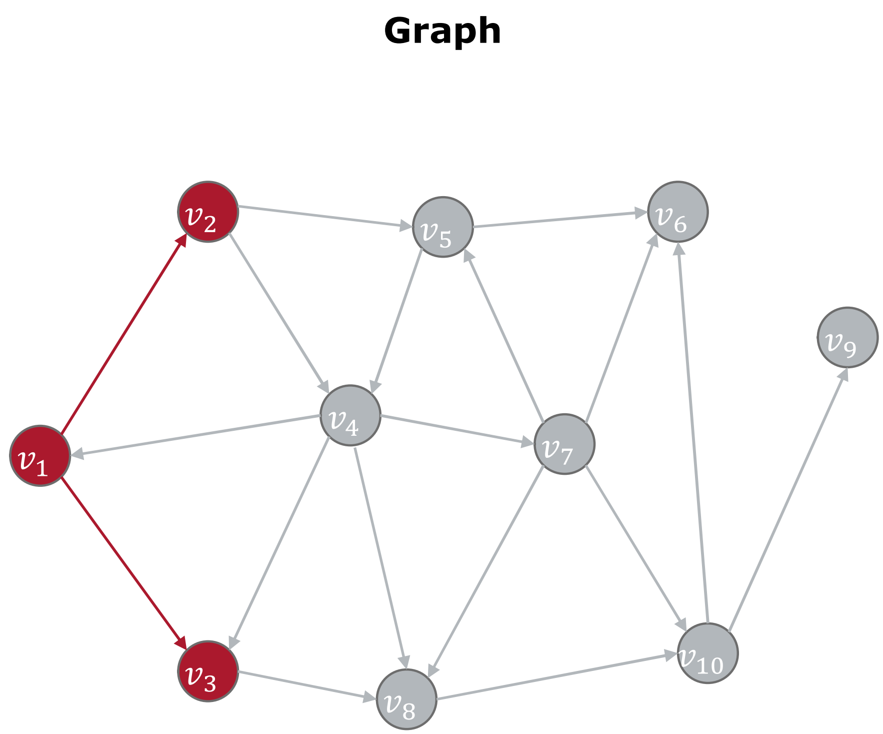
BFS Tree Visualized

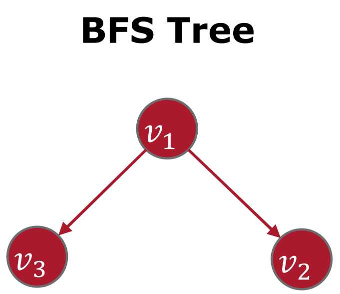
Graph



BFS Tree

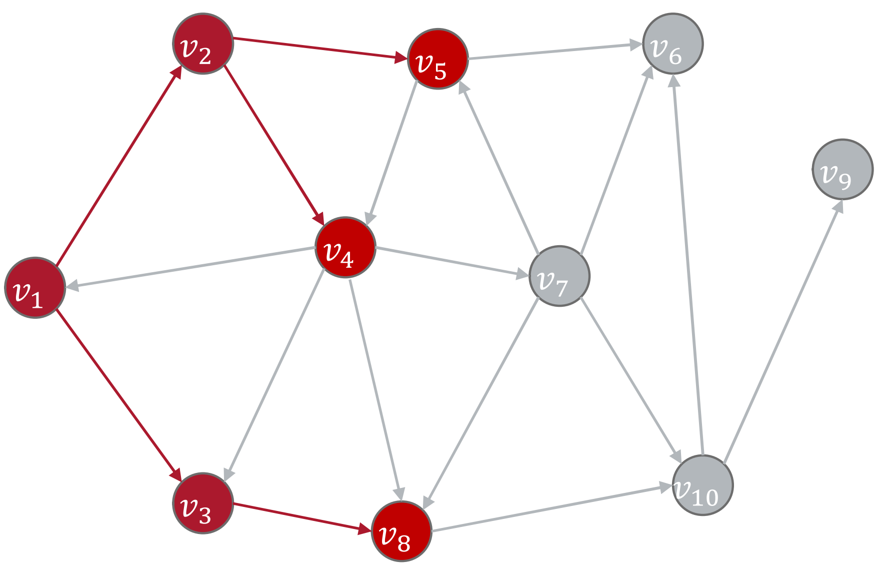
BFS Tree Visualized



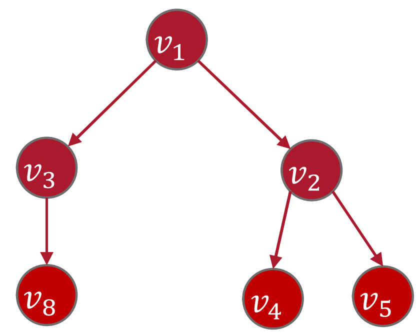


BFS Tree Visualized

Graph

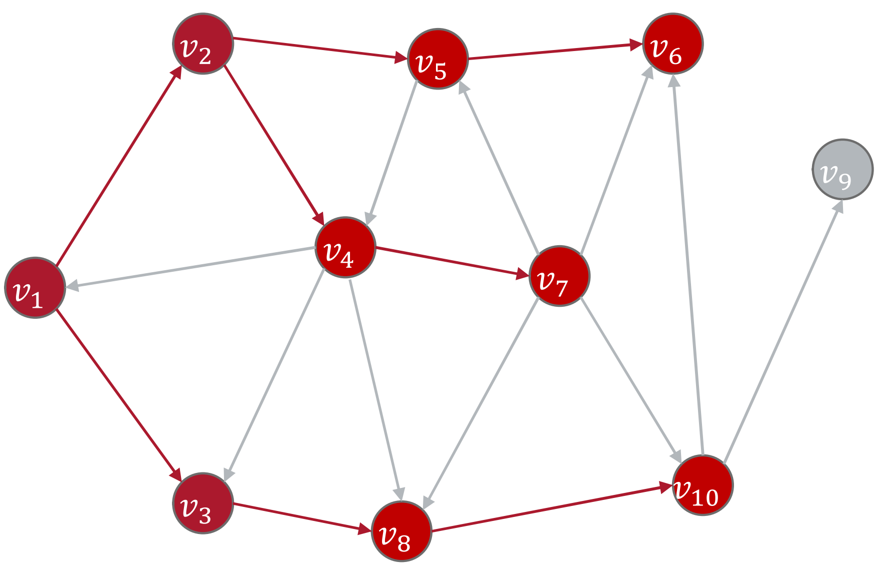


BFS Tree

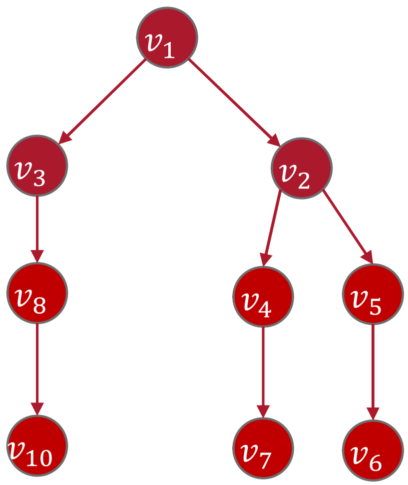


BFS Tree Visualized

Graph

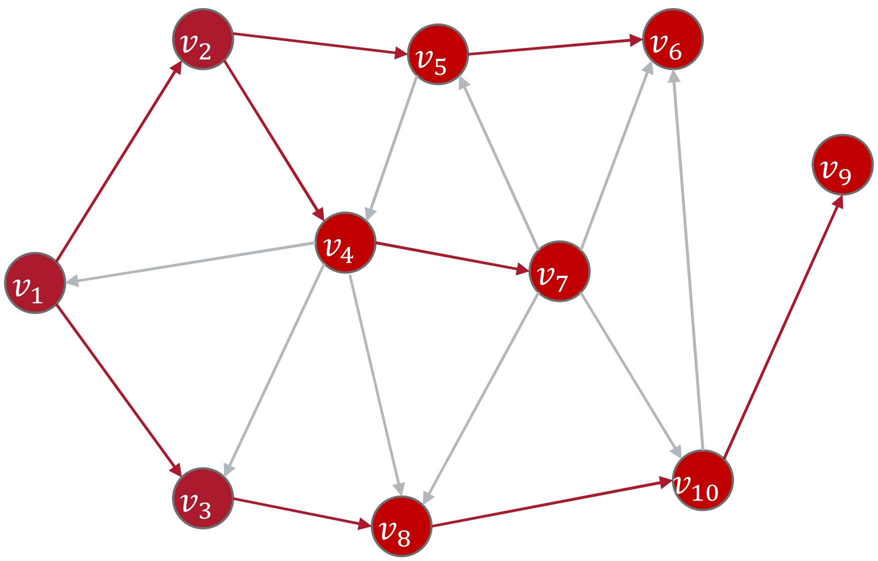


BFS Tree

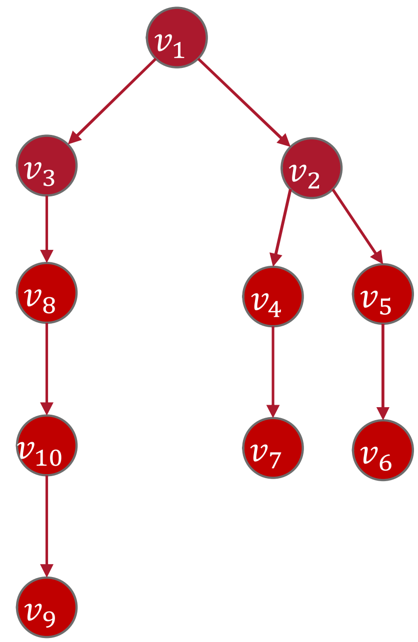


BFS Tree Visualized

Graph

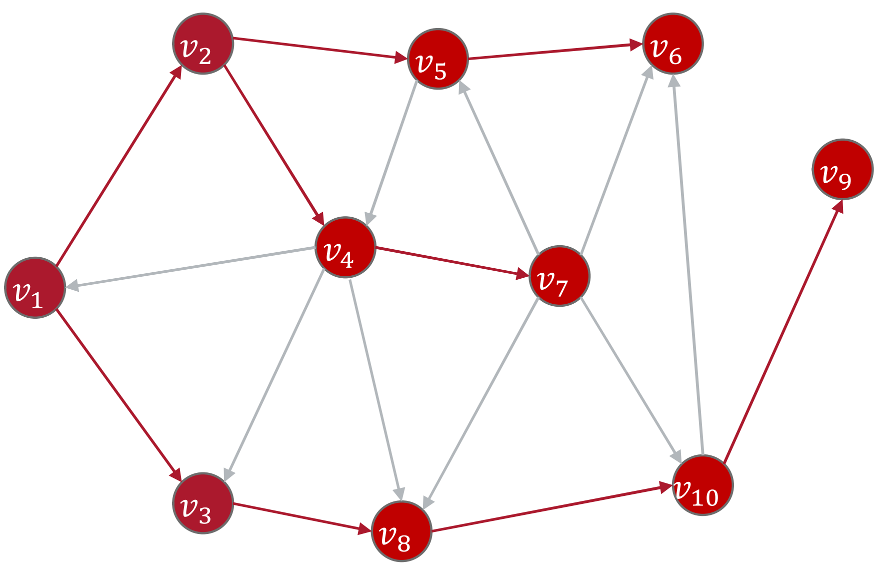


BFS Tree

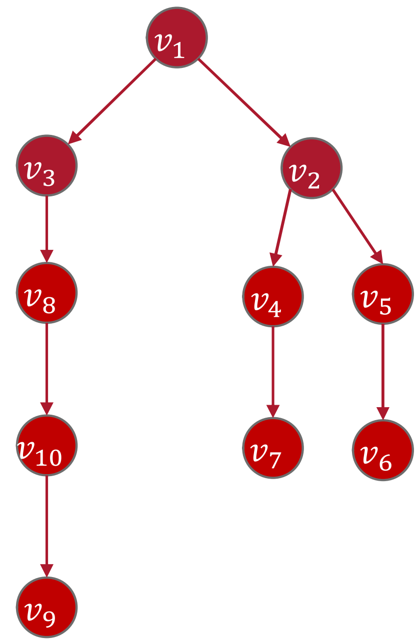


BFS Tree Visualized

Graph



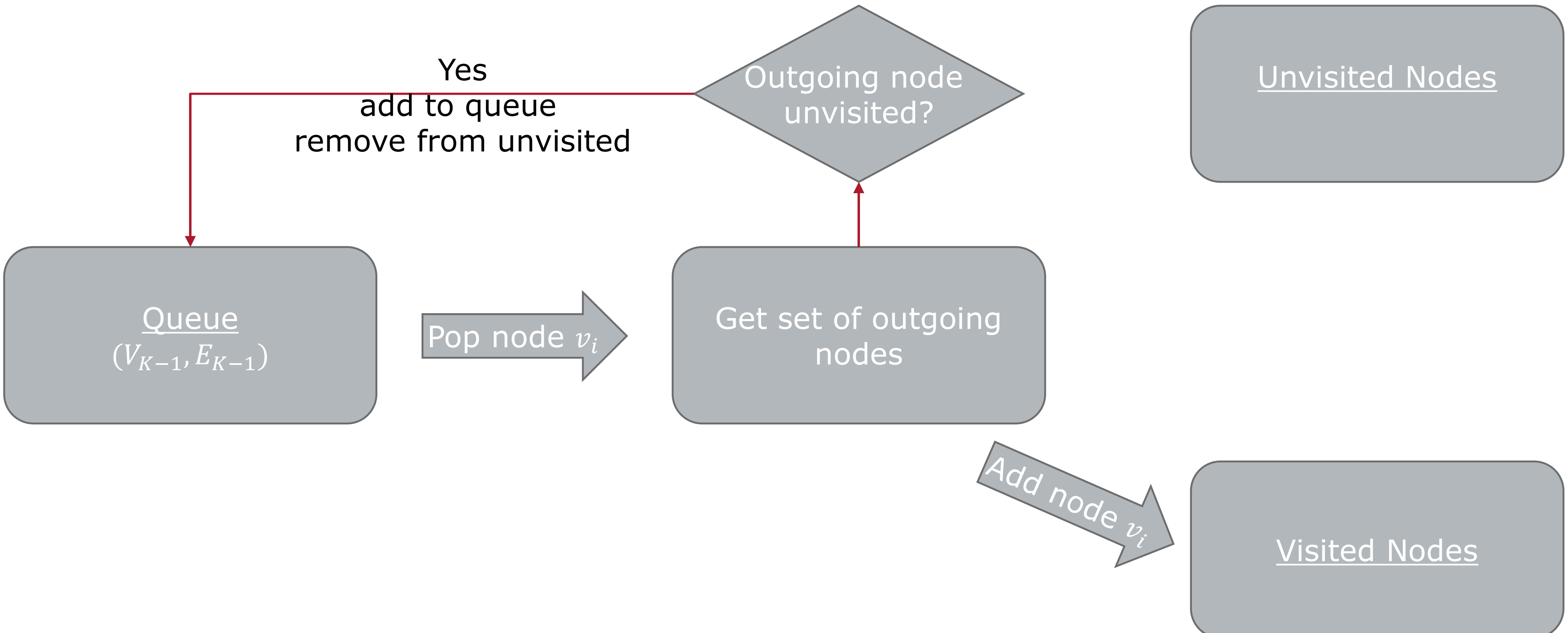
BFS Tree



BFS Tree

- This approach is centralized
- How is it computed? What is different about the pseudocode vs how you would implement it?
- What information is required?

Computationally



What makes this difficult for a team of multiple robots?

Distributed Algorithms

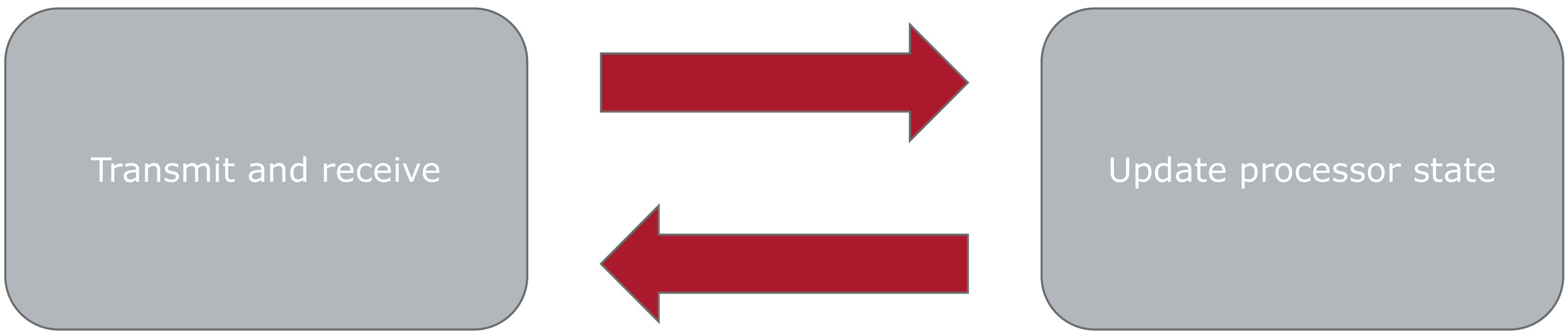
Distributing an algorithm

- We don’t have centralized processor, queue, etc.
- How can we distribute an algorithm?
- What are the ingredients?

Modeling Communication – Networks

- We will model a **synchronous network** S as a digraph (I, E_{cmm}), where:
  - $I = \{1, \dots, N\}$ is the set of **unique identifiers** (UIDs)
  - E_{cmm} is a set of directed edges over the vertices $\{1, \dots, N\}$, known as communication links
- Each $i \in I$ represents a **processor** (i.e., robot *brain*) and E_{cmm} represents the communication topology among the processors
- Processor i can send a message to processor j if $(i, j) \in E_{cmm}$

Communication on a Network



Distributed Algorithm

- A **distributed algorithm** DA for a network S consists of the sets
  - $\mathbb{A}$: a set containing the **alphabet**, including the `null` symbol
  - $W^{[i]}, i \in I$: the **processor state sets**
  - $W_0^{[i]} \subseteq W^{[i]}, i \in I$: the **allowable initial values**

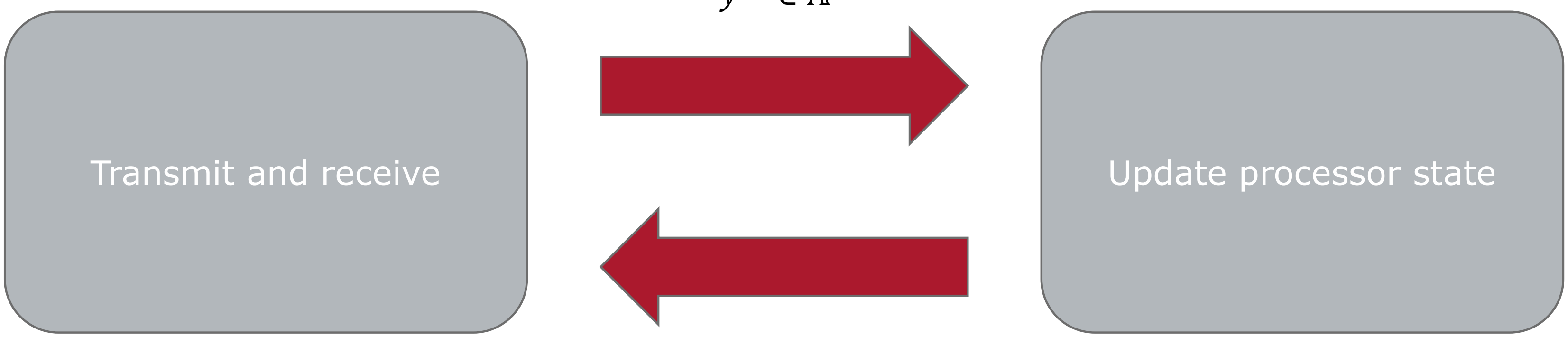
- It also has the maps
  - $msg^{[i]}: W^{[i]} \times I \rightarrow \mathbb{A}, i \in I$: the **message-generation functions**
  - $stf^{[i]}: W^{[i]} \times \mathbb{A}^n \rightarrow W^{[i]}, i \in I$: the **state-transition functions**

Communication on a Network

State: $w^{[i]} \in W^{[i]}$

Update state:
$w^{[i]} \leftarrow stf^{[i]}(w^{[i]}, y^{[i]})$

Receive messages:
$y^{[i]} \in \mathbb{A}^n$



Generate messages:
$y_i^{[j]} \leftarrow msg^{[i]}(w^{[i]}, j)$ if $(i,j) \in E_{cmm}$

Network Evolution

- For a distributed algorithm DA on a network S, the **evolution** of (S, DA) from initial conditions $w_0^{[i]} \in W_0^{[i]}, i \in I$, is a collection of trajectories $w^{[i]}: \mathbb{Z}_{\geq 0} \to W^{[i]}, i \in I$, with

$$w^{[i]}(l) = stf^{[i]}(w^{[i]}(l-1), y^{[i]}(l))$$

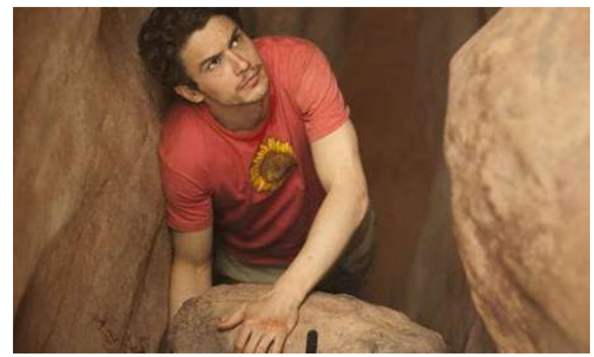
$$y_j^{[i]}(l) = \begin{cases} msg^{[j]}(w^{[j]}(l-1), i) & if\ (j,i) \in E_{cmm} \\ null & otherwise \end{cases}$$

Assumptions of this model

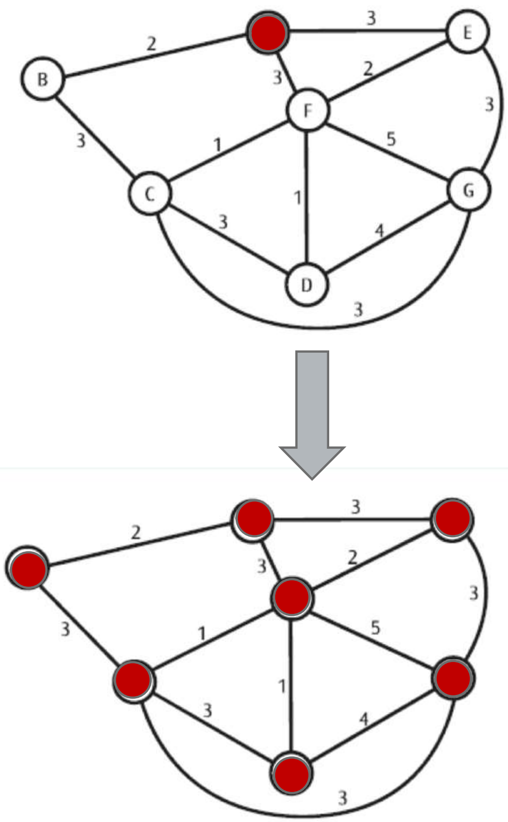
1. S and DA are **synchronous** because communication takes place at the same time for all processors
2. Communication is **point-to-point**: processor i can send different messages to different neighbors and identify the origin of messages it receives
3. Information is transmitted as **messages** from an alphabet $\mathbb{A}$. This includes null, logical, integers, reals; we do not consider how to effectively transmit this information
4. In many instances $msg_{std}(w, j) = w$—agents transmit their states. We will call this the **standard message-generation function**

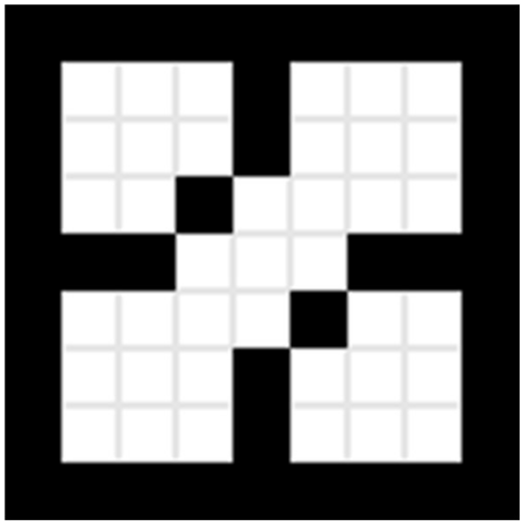
Example – Search

I found him,
his location is
38°24'17.0"N
110°14'46.7"W



Flooding





Flooding

Synchronous Network: $\mathcal{S} = (\{1, \ldots, n\}, E_{\text{cmm}})$

Distributed Algorithm: FLOODING

Alphabet: $\mathbb{A} = \{\alpha, \ldots, \omega\} \cup \texttt{null}$

Processor State: $w = (\texttt{parent}, \texttt{data}, \texttt{snd-flag})$, where

$$
\begin{array}{llll}
\texttt{parent} & \in \{0, \ldots, n\}, & \text{initially:} & \texttt{parent}^{[1]} = 1, \\
 & & & \texttt{parent}^{[j]} = 0 \text{ for all } j \neq 1 \\
\texttt{data} & \in \mathbb{A}, & \text{initially:} & \texttt{data}^{[1]} = \mu, \\
 & & & \texttt{data}^{[j]} = \texttt{null} \text{ for all } j \neq 1 \\
\texttt{snd-flag} & \in \{\texttt{false}, \texttt{true}\}, & \text{initially:} & \texttt{snd-flag}^{[1]} = \texttt{true}, \\
 & & & \texttt{snd-flag}^{[j]} = \texttt{false} \text{ for } j \neq 1
\end{array}
$$

Flooding

Synchronous Network: $\mathcal{S} = (\{1, \ldots, n\}, E_{\text{cmm}})$

Distributed Algorithm: FLOODING

Alphabet: $\mathbb{A} = \{\alpha, \ldots, \omega\} \cup \texttt{null}$

Processor State: $w = (\texttt{parent}, \texttt{data}, \texttt{snd-flag})$, where

$$
\begin{array}{llll}
\texttt{parent} & \in \{0, \ldots, n\}, & \text{initially:} & \texttt{parent}^{[1]} = 1, \\
 & & & \texttt{parent}^{[j]} = 0 \text{ for all } j \neq 1 \\
\texttt{data} & \in \mathbb{A}, & \text{initially:} & \texttt{data}^{[1]} = \mu, \\
 & & & \texttt{data}^{[j]} = \texttt{null} \text{ for all } j \neq 1 \\
\texttt{snd-flag} & \in \{\texttt{false}, \texttt{true}\}, & \text{initially:} & \texttt{snd-flag}^{[1]} = \texttt{true}, \\
 & & & \texttt{snd-flag}^{[j]} = \texttt{false} \text{ for } j \neq 1
\end{array}
$$

Flooding

Synchronous Network: $\mathcal{S} = (\{1, \ldots, n\}, E_{\mathrm{cmm}})$

Distributed Algorithm: FLOODING

Alphabet: $\mathbb{A} = \{\alpha, \ldots, \omega\} \cup \texttt{null}$

Processor State: $w = (\texttt{parent}, \texttt{data}, \texttt{snd-flag})$, where

$$
\begin{array}{llll}
\texttt{parent} & \in \{0, \ldots, n\}, & \text{initially:} & \texttt{parent}^{[1]} = 1, \\
 & & & \texttt{parent}^{[j]} = 0 \text{ for all } j \neq 1 \\
\texttt{data} & \in \mathbb{A}, & \text{initially:} & \texttt{data}^{[1]} = \mu, \\
 & & & \texttt{data}^{[j]} = \texttt{null} \text{ for all } j \neq 1 \\
\texttt{snd-flag} & \in \{\texttt{false}, \texttt{true}\}, & \text{initially:} & \texttt{snd-flag}^{[1]} = \texttt{true}, \\
 & & & \texttt{snd-flag}^{[j]} = \texttt{false} \text{ for } j \neq 1
\end{array}
$$

Flooding

Synchronous Network: $\mathcal{S} = (\{1, \ldots, n\}, E_{\text{cmm}})$

Distributed Algorithm: FLOODING

Alphabet: $\mathbb{A} = \{\alpha, \ldots, \omega\} \cup \texttt{null}$

Processor State: $w = (\texttt{parent}, \texttt{data}, \texttt{snd-flag})$, where

parent	$\in \{0, \ldots, n\}$,	initially:	$\texttt{parent}^{[1]} = 1$, $\texttt{parent}^{[j]} = 0$ for all $j \neq 1$
data	$\in \mathbb{A}$,	initially:	$\texttt{data}^{[1]} = \mu$, $\texttt{data}^{[j]} = \texttt{null}$ for all $j \neq 1$
snd-flag	$\in \{\texttt{false}, \texttt{true}\}$,	initially:	$\texttt{snd-flag}^{[1]} = \texttt{true}$, $\texttt{snd-flag}^{[j]} = \texttt{false}$ for $j \neq 1$

Flooding

Synchronous Network: $\mathcal{S} = (\{1, \ldots, n\}, E_{\mathrm{cmm}})$

Distributed Algorithm: FLOODING

Alphabet: $\mathbb{A} = \{\alpha, \ldots, \omega\} \cup \texttt{null}$

Processor State: $w = (\texttt{parent}, \texttt{data}, \texttt{snd-flag})$, where

parent	$\in \{0, \ldots, n\}$,	initially:	$\texttt{parent}^{[1]} = 1$,
			$\texttt{parent}^{[j]} = 0$ for all $j \neq 1$
data	$\in \mathbb{A}$,	initially:	$\texttt{data}^{[1]} = \mu$,
			$\texttt{data}^{[j]} = \texttt{null}$ for all $j \neq 1$
snd-flag	$\in \{\texttt{false}, \texttt{true}\}$,	initially:	$\texttt{snd-flag}^{[1]} = \texttt{true}$,
			$\texttt{snd-flag}^{[j]} = \texttt{false}$ for $j \neq 1$

Flooding

```
function msg(w, i)
  if (parent ≠ i) AND (snd-flag = true) then
      return data
  else
      return null
```

Flooding

```
function stf(w, y)
case
   (data = null) AND (y contains only null messages):
   % The node has not yet received the token
      new-parent := null
      new-data := null
      new-snd-flag := false
   (data = null) AND (y contains a non-null message):
   % The node has just received the token
      new-parent := smallest UID among transmitting in-neighbors
      new-data := a non-null message
      new-snd-flag := true
   (data ≠ null):
   % If the node already has the token, then do not re-broadcast it
      new-parent := parent
      new-data := data
      new-snd-flag := false
return (new-parent, new-data, new-snd-flag)
```

Flooding

```
function stf(w, y)
case
  (data = null) AND (y contains only null messages):
  % The node has not yet received the token
    new-parent := null
    new-data := null
    new-snd-flag := false
  (data = null) AND (y contains a non-null message):
  % The node has just received the token
    new-parent := smallest UID among transmitting in-neighbors
    new-data := a non-null message
    new-snd-flag := true
  (data ≠ null):
  % If the node already has the token, then do not re-broadcast it
    new-parent := parent
    new-data := data
    new-snd-flag := false
return (new-parent, new-data, new-snd-flag)
```

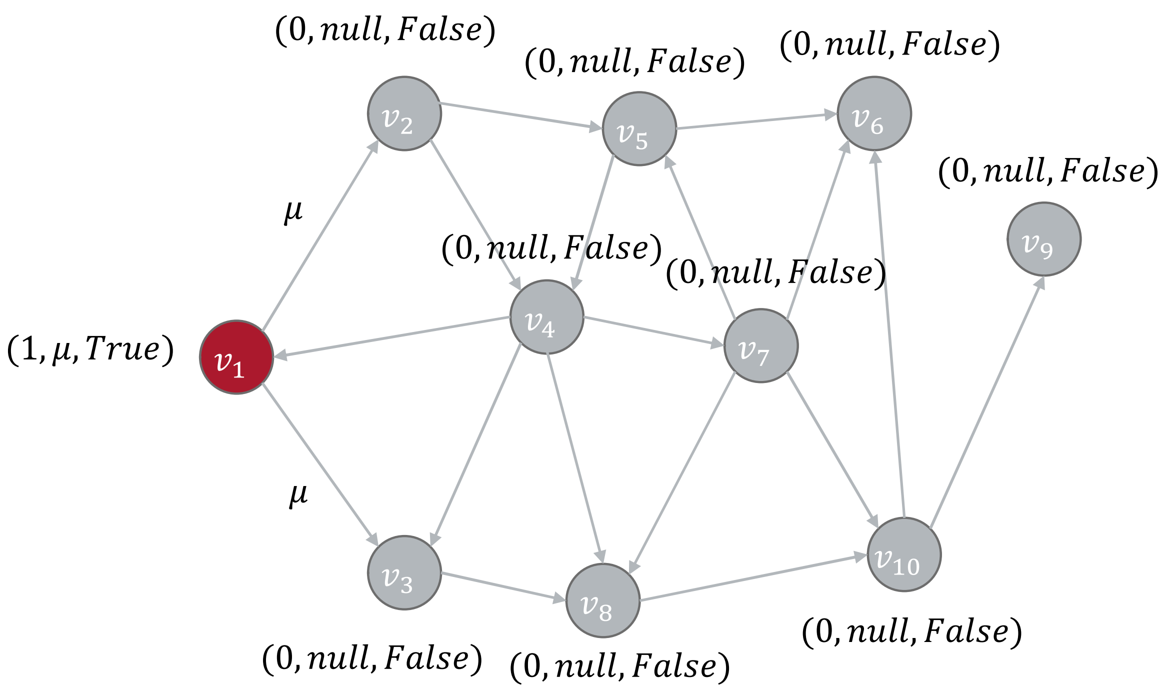
Flooding

```
function stf(w, y)
case
  (data = null) AND (y contains only null messages):
   % The node has not yet received the token
    new-parent := null
    new-data := null
    new-snd-flag := false
  (data = null) AND (y contains a non-null message):
   % The node has just received the token
    new-parent := smallest UID among transmitting in-neighbors
    new-data := a non-null message
    new-snd-flag := true
  (data ≠ null):
   % If the node already has the token, then do not re-broadcast it
    new-parent := parent
    new-data := data
    new-snd-flag := false
return (new-parent, new-data, new-snd-flag)
```

Flooding

```
function stf(w, y)
case
  (data = null) AND (y contains only null messages):
  % The node has not yet received the token
    new-parent := null
    new-data := null
    new-snd-flag := false
  (data = null) AND (y contains a non-null message):
  % The node has just received the token
    new-parent := smallest UID among transmitting in-neighbors
    new-data := a non-null message
    new-snd-flag := true
  (data ≠ null):
  % If the node already has the token, then do not re-broadcast it
    new-parent := parent
    new-data := data
    new-snd-flag := false
return (new-parent, new-data, new-snd-flag)
```

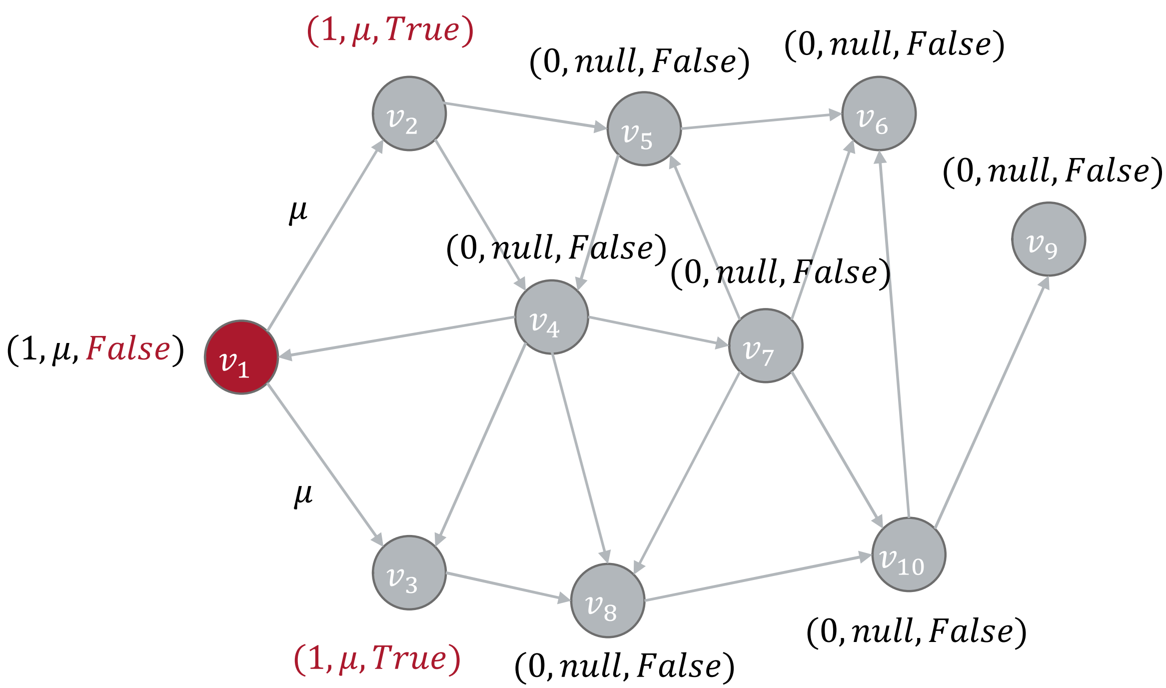
Flooding Msg Round 1



Processor State:
(parent, data, snd-flag)

Assume unlabeled edges are null messages

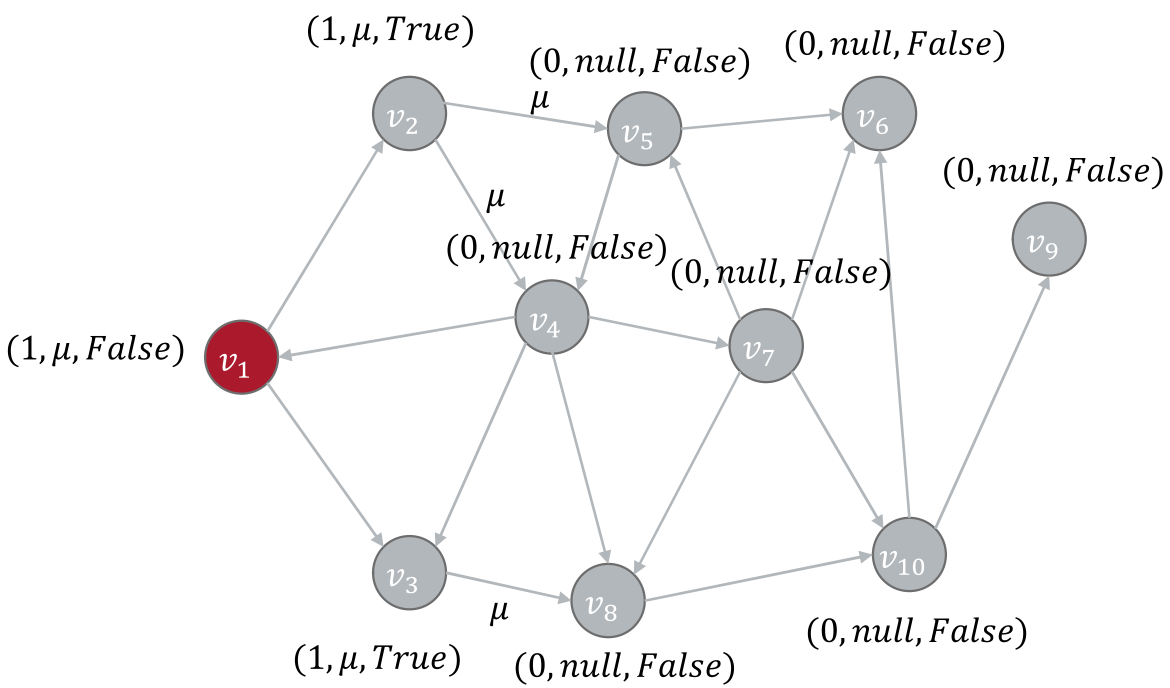
Flooding Stf Round 1



Processor State:
(parent, data, snd-flag)

Assume unlabeled edges are null messages

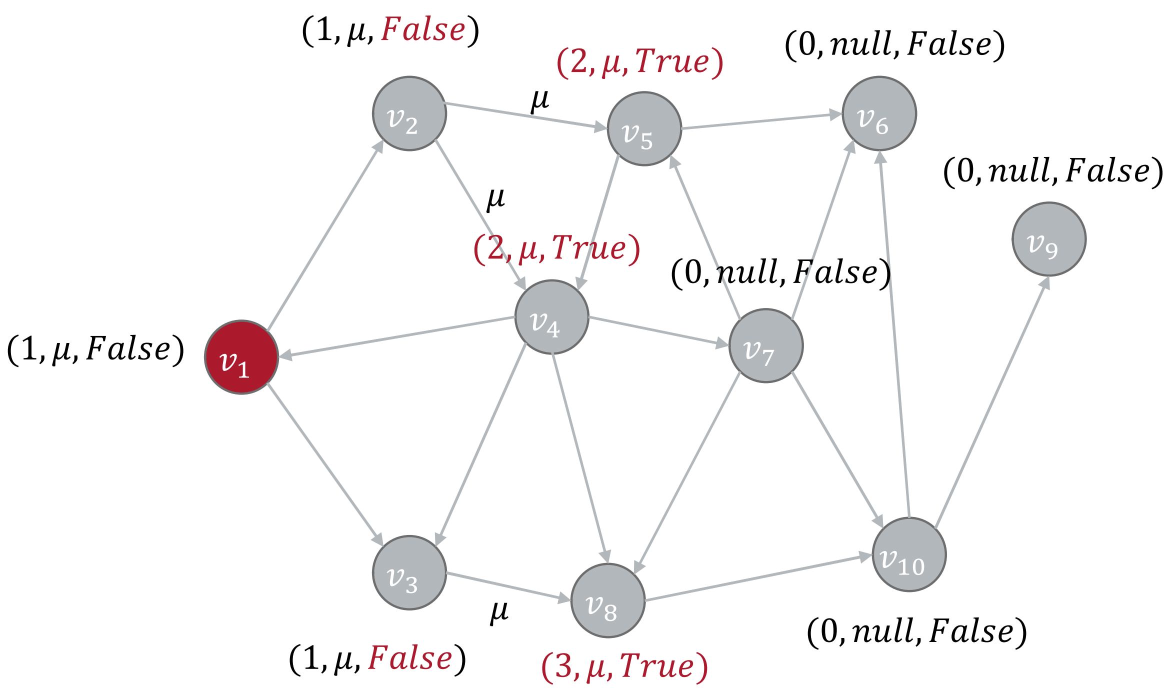
Flooding Msg Round 2



Processor State:
(parent, data, snd-flag)

Assume unlabeled edges are null messages

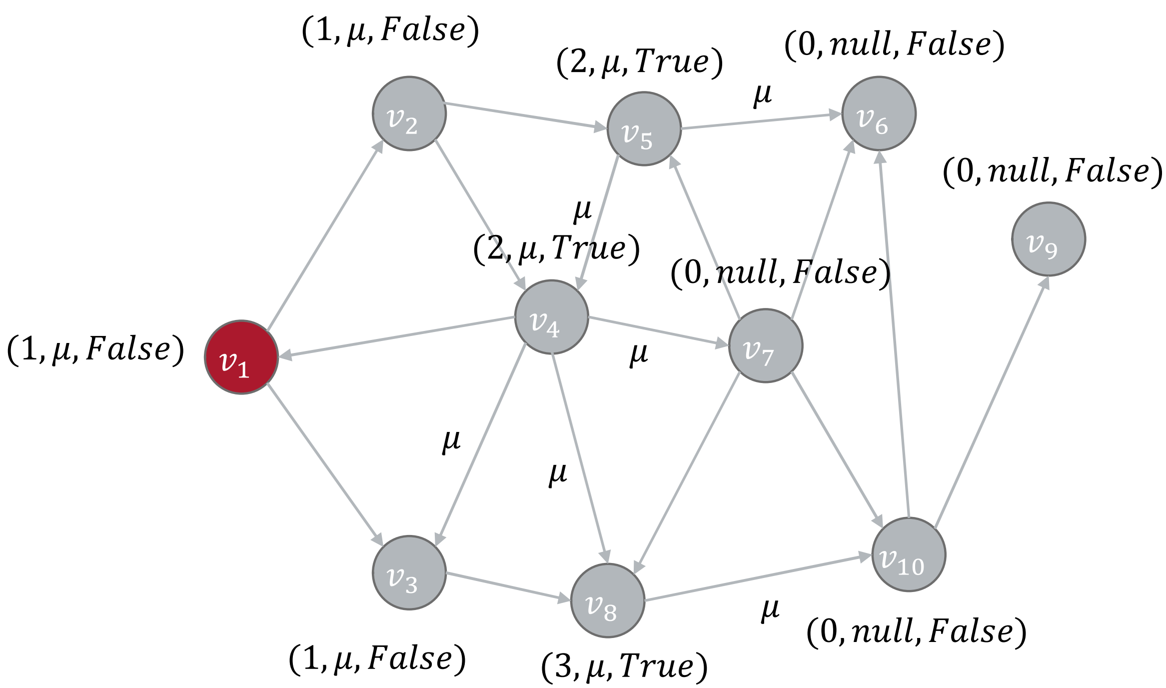
Flooding Stf Round 2



Processor State:
(parent, data, snd-flag)

Assume unlabeled edges are null messages

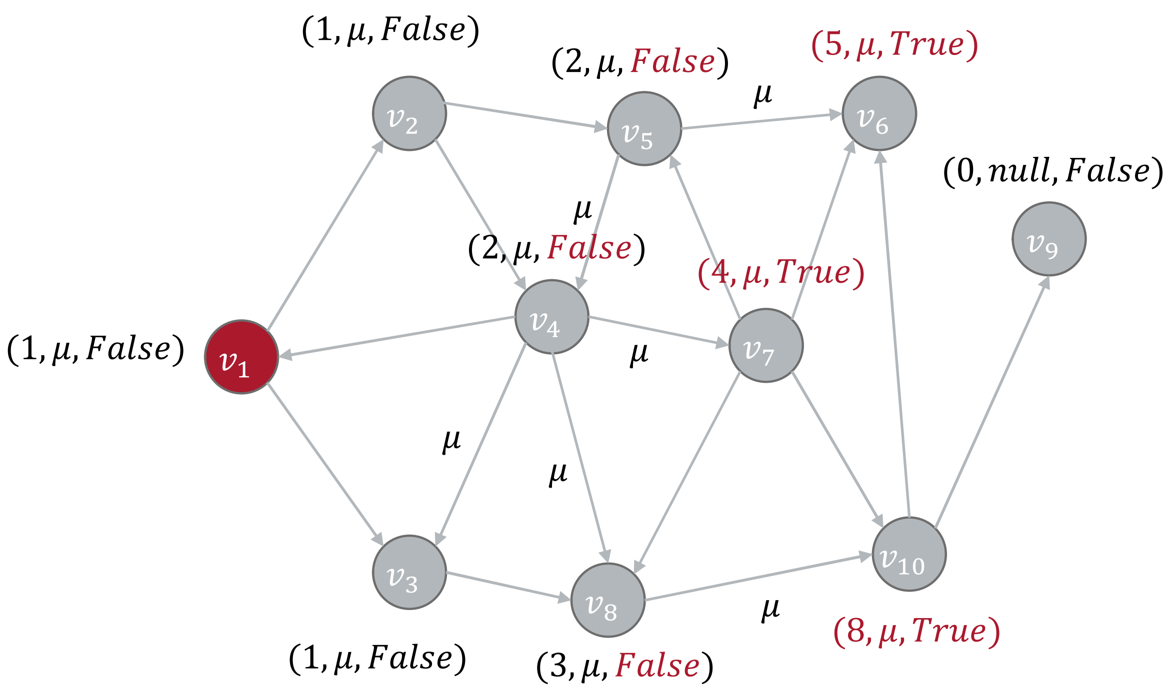
Flooding Msg Round 3



Processor State:
(parent, data, snd-flag)

Assume unlabeled edges are null messages

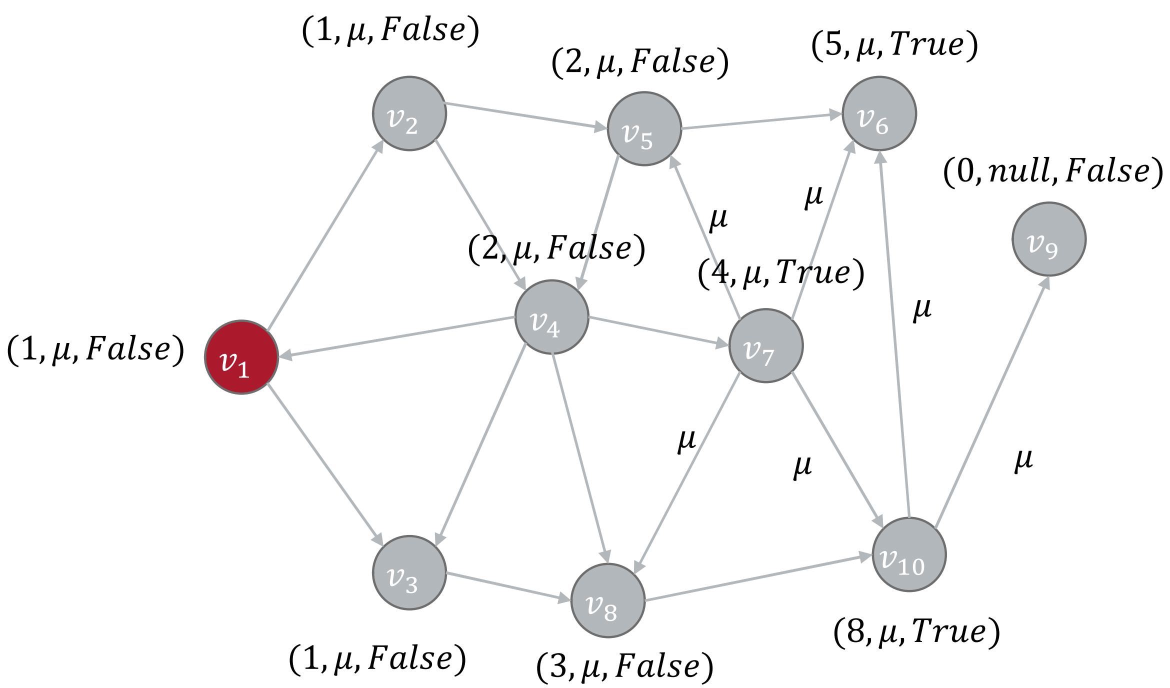
Flooding Stf Round 3



Processor State:
(parent, data, snd-flag)

Assume unlabeled edges are null messages

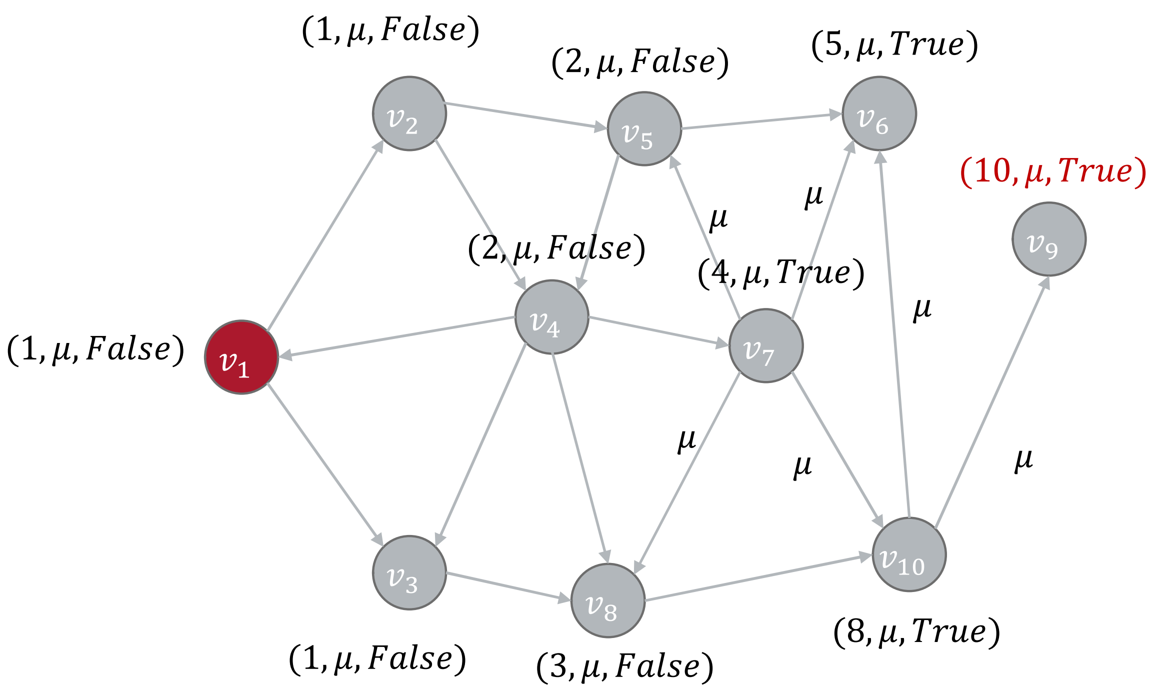
Flooding Msg Round 4



Processor State:
(parent, data, snd-flag)

Assume unlabeled edges are null messages

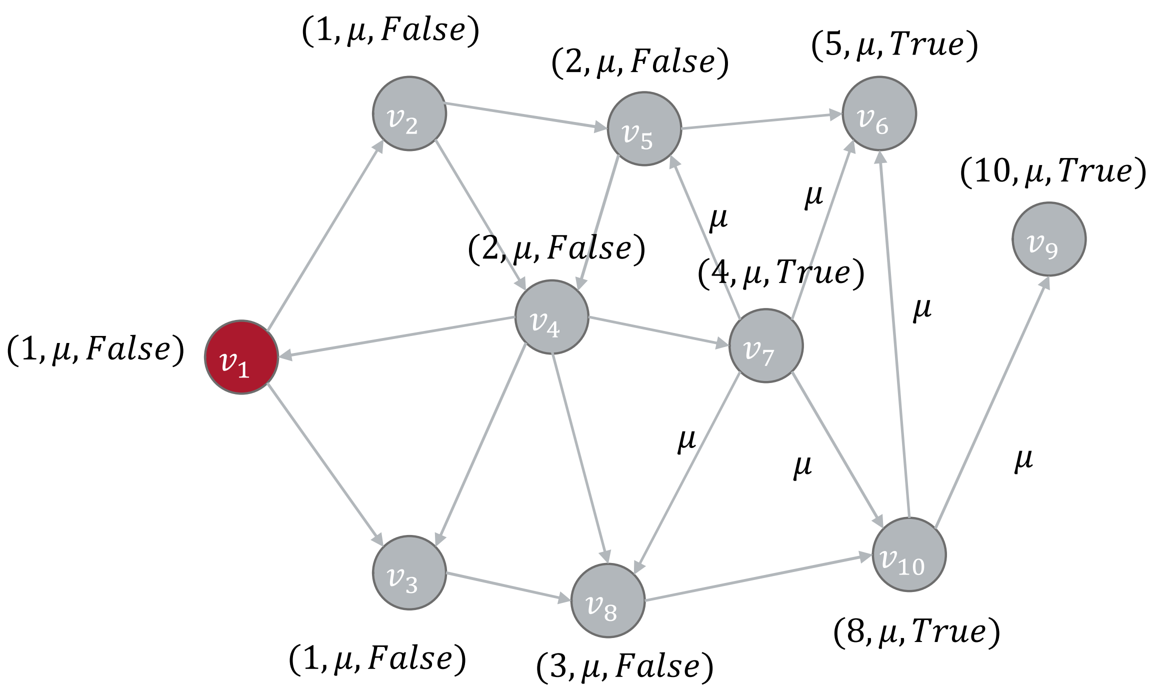
Flooding Stf Round 4



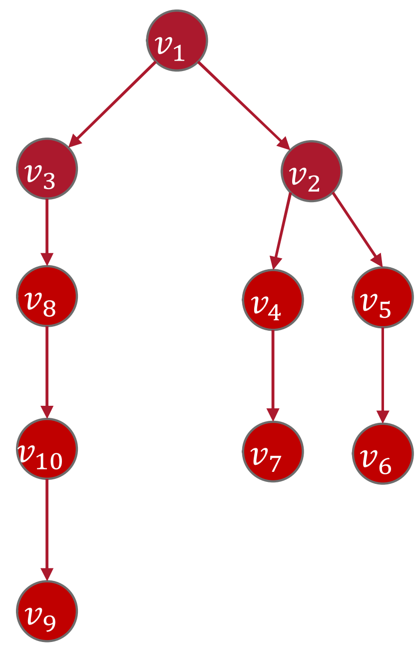
Processor State:
(parent, data, snd-flag)

Assume unlabeled edges are null messages

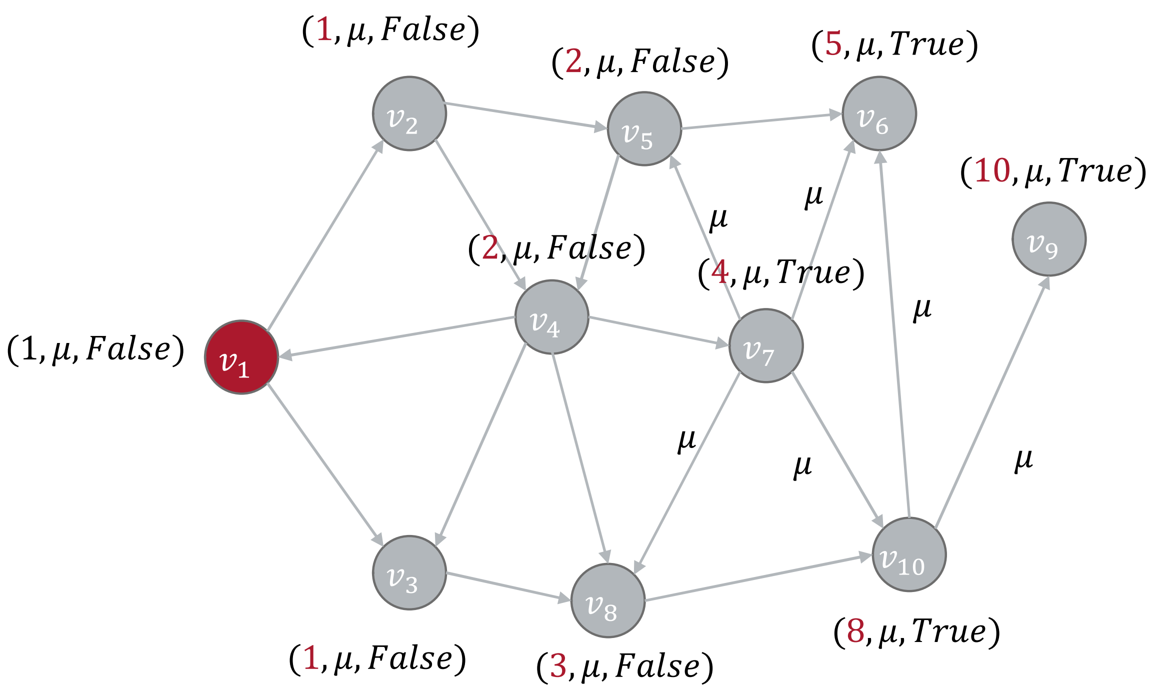
Comparison to Centralized Algorithm



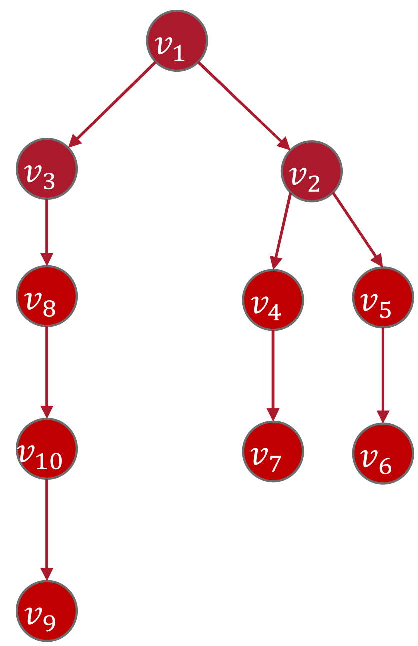
BFS Tree



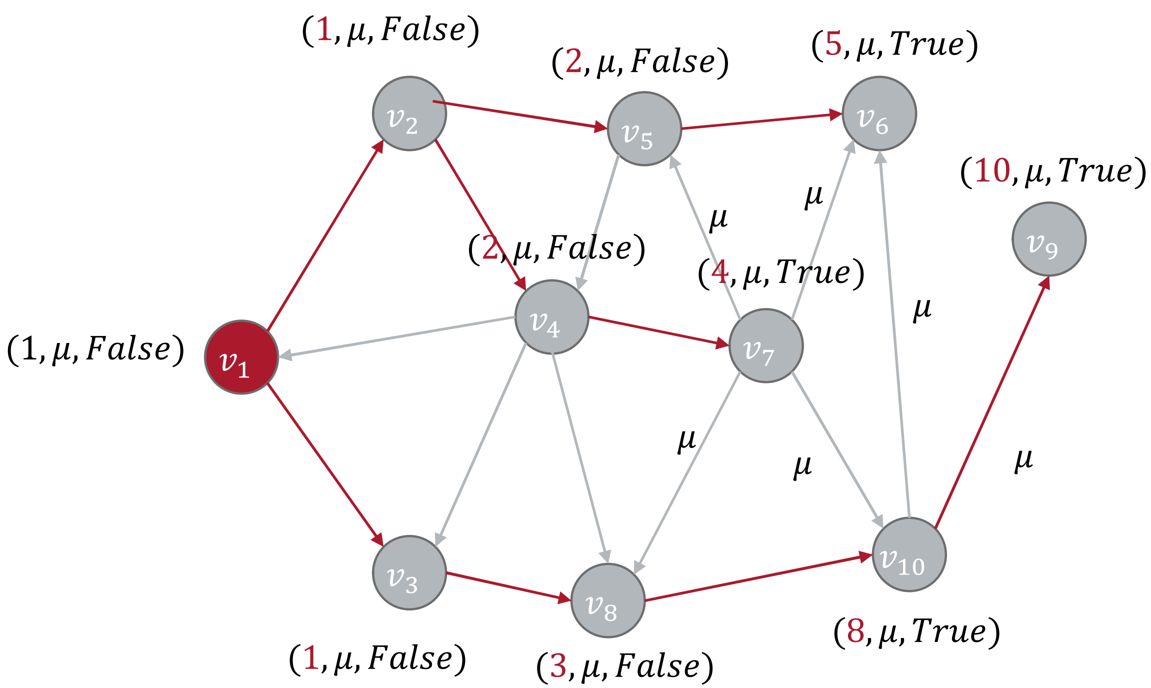
Comparison to Centralized Algorithm



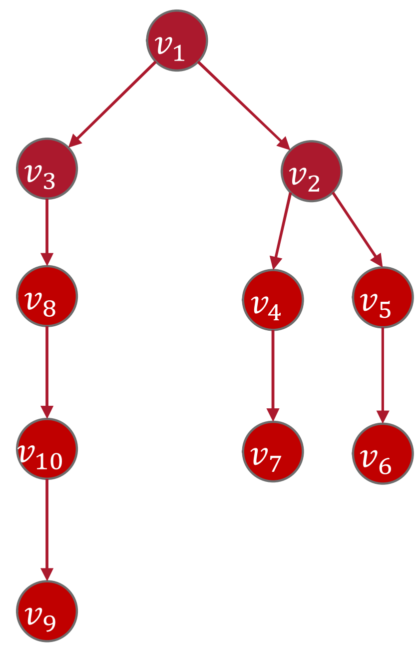
BFS Tree



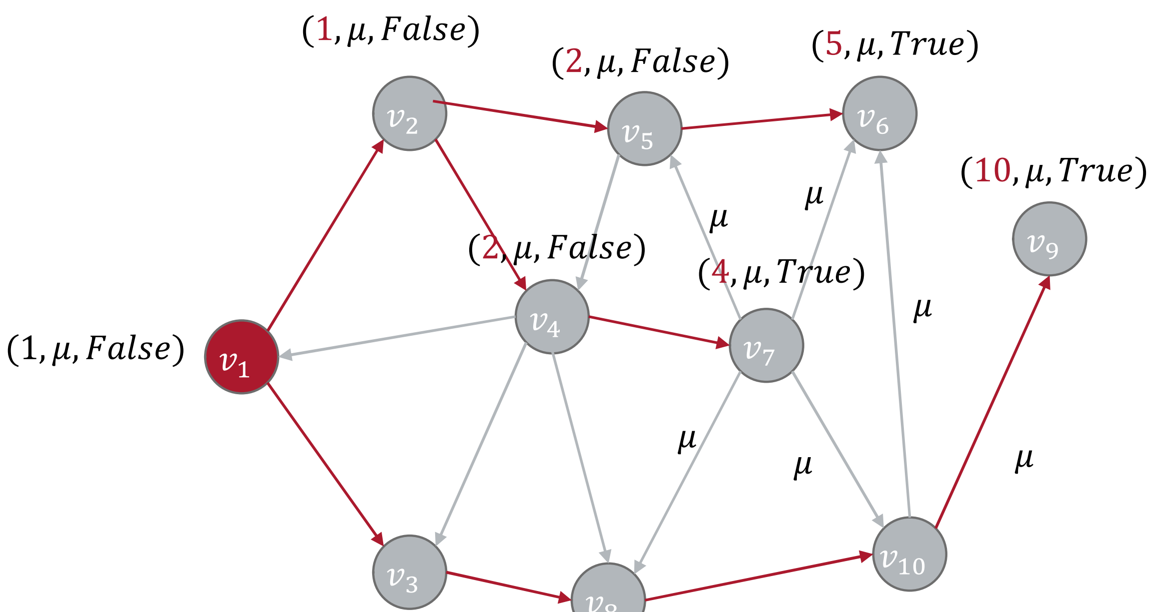
Comparison to Centralized Algorithm



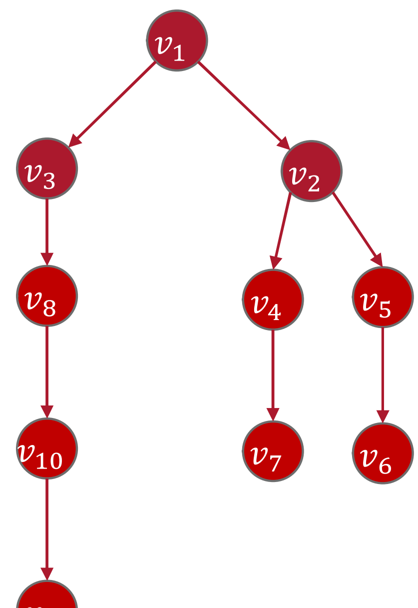
BFS Tree



Comparison to Centralized Algorithm

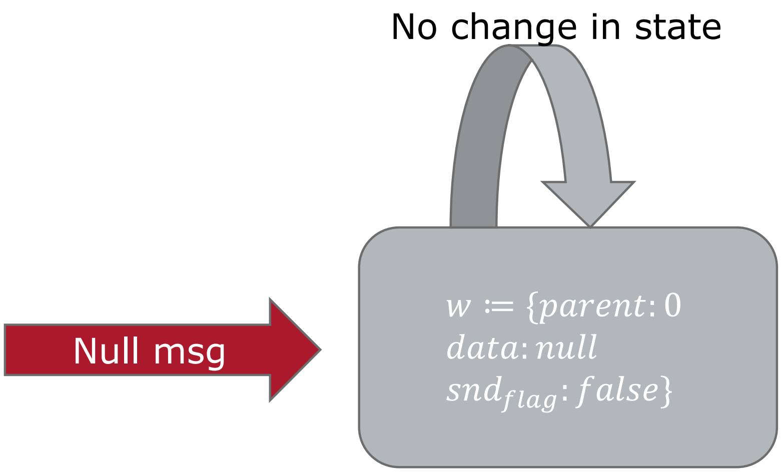


BFS Tree

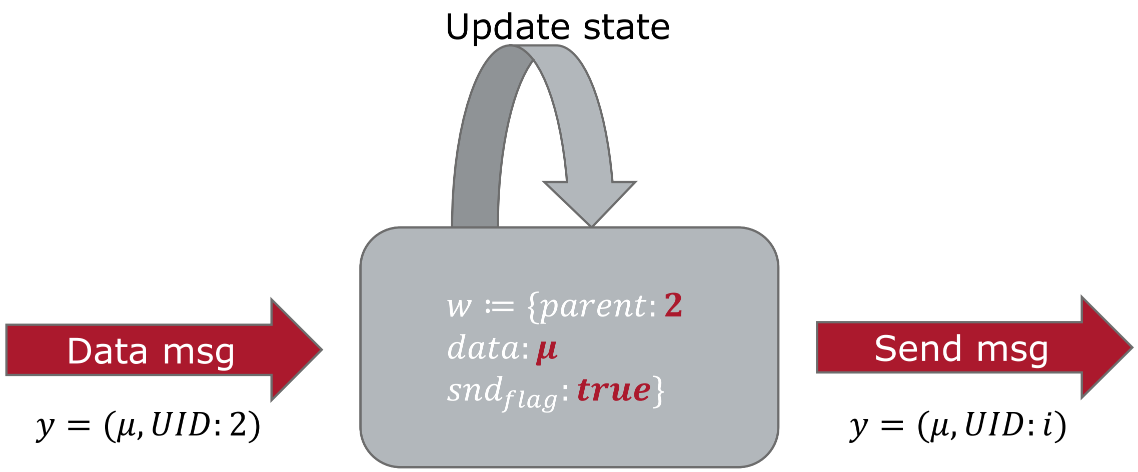


The result is the same, but how can we quantify the difference in the process?

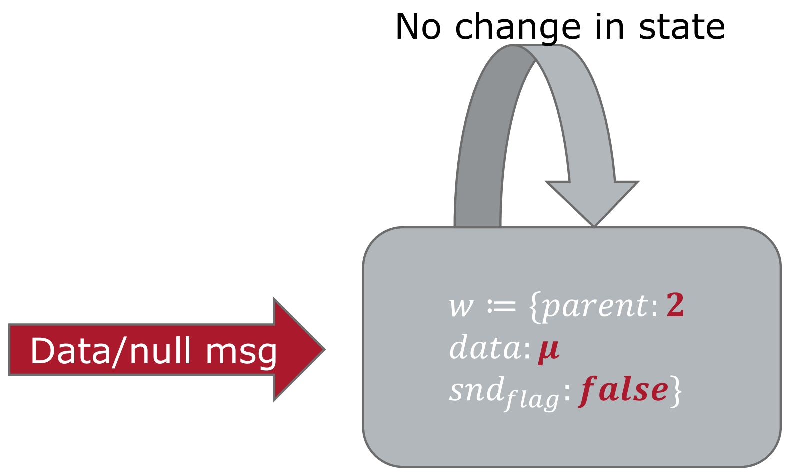
Individual Agent i Perspective



Individual Agent i Perspective

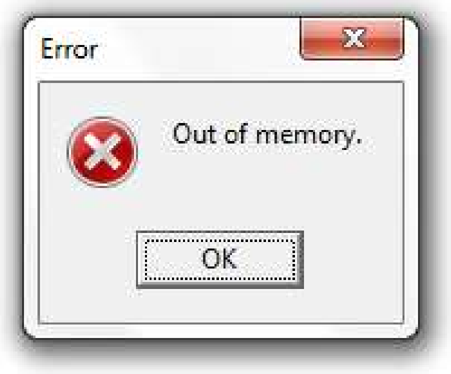


Individual Agent i Perspective



Types of Complexity

Time



Space

Communication

Types of Complexity

- **Time complexity** – the maximum number of rounds required by execution of the algorithm for an arbitrary initial state before the algorithm terminates

- **Space complexity** – the maximum number of basic memory units required by a processor executing a distributed algorithm among all processors and initial states before the algorithm terminates

- **Communication complexity** – the maximum number of basic messages transmitted over the entire network during execution of the algorithm among all initial states before the algorithm terminates

Complexity – some useful notation

- For $f, g: \mathbb{N} \to \mathbb{R}_{\geq 0}$, we say that

  Upper-bounding

  - $f \in O(g)$ if there exist $n_0 \in \mathbb{N}$ and $K \in \mathbb{R}_{>0}$ such that $f(n) \leq Kg(n)$ for all $n \geq n_0$

  Lower-bounding

  - $f \in \Omega(g)$ if there exist $n_0 \in \mathbb{N}$ and $k \in \mathbb{R}_{>0}$ such that $f(n) \geq kg(n)$ for all $n \geq n_0$

  - If $f \in O(g)$ and $f \in \Omega(g)$, we write $f \in \Theta(g)$ Upper- and lower-bounding

- O, Ω, and Θ are called **Bachmann-Landau symbols**

General Complexity Notions on Graphs

- $\mathrm{diam}(S) \in \Theta(n)$ – you can determine the diameter of a graph in linear time compared to the number of nodes
  - *Diameter is the greatest distance between any two nodes*

- $|E_{cmm}(S)| \in \Theta(n^2)$ – you can determine the size of the edge set in squared (polynomial) time compared to the number of nodes

- $\mathrm{radius}(v, S) \in \Theta(\mathrm{diam}(S))$ – you can determine the radius of the graph with the same complexity as computing the diameter
  - *Radius of a graph is the minimum over all vertices of the maximum distance to any other vertex*

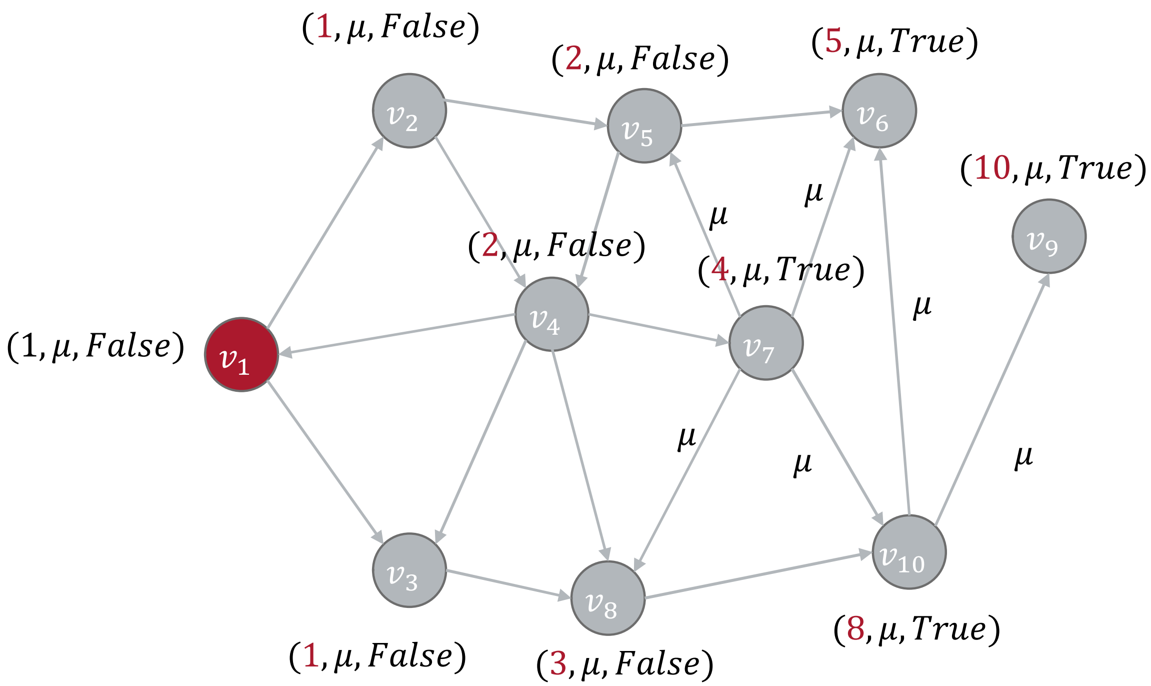
Complexity of Flooding Algorithm

- Flooding algorithm has communication complexity in $\Theta(|E_{cmm}|)$
  - Why?
  - Each edge is traversed at most one time

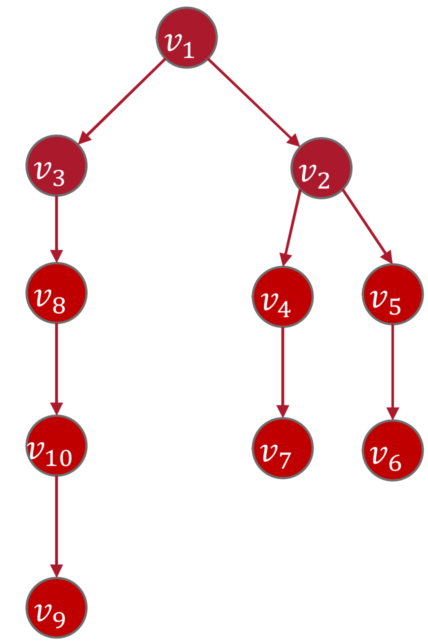
- Time complexity in $\Theta(\mathrm{radius}(v, S))$
  - Why?
  - If you get unlucky, you might have to perform $\mathrm{radius}(v, S)$ rounds

- Space complexity in $\Theta(1)$
  - Why?
  - The size of the processor state for each agent is fixed
  - What types of algorithms might cause this to change?

Comparison to Centralized Algorithm



BFS Tree



How is the complexity different? How is this related to the end result?

Our model now

- We have a network model that supports distributed algorithms
  - Physical state
  - Processor state
- In upcoming homework, we will tie all three together
  - Network
  - Physical (control) system
  - Processing system

Programming Assignment Overview

Simulating a distributed system

```
for t in time steps:
        for agent in agents:
                update state for each agent
        for agent in agents:
                send messages for each agent
        for agent in agents:
                receive messages for each agent
```

- Could spawn individual processes/threads for each agent
- Pass messages in between
- Create a bunch of ROS nodes
- Especially useful for asynchronous systems

- Run as single process
- Simple way to implement synchronous process
- Computationally inefficient
- Run time is longer than actual distributed system

FloodMAX algorithm

Synchronous Network: $\mathcal{S} = (\{1, \dots, n\}, E_{\text{cmm}})$

Distributed Algorithm: FLOODMAX

Alphabet: $\mathbb{A} = \{1, \dots, n\} \cup \{\texttt{null}\}$

Processor State: $w = (\texttt{my-id}, \texttt{max-id}, \texttt{leader}, \texttt{round})$, where

my-id	$\in \{1, \dots, n\}$,	initially: $\texttt{my-id}^{[i]} = i$ for all i
max-id	$\in \{1, \dots, n\}$,	initially: $\texttt{max-id}^{[i]} = i$ for all i
leader	$\in \{\texttt{false}, \texttt{true}, \texttt{unknwn}\}$,	initially: $\texttt{leader}^{[i]} = \texttt{unknwn}$ for all i
round	$\in \{0, 1, \dots, \text{diam}(\mathcal{S})\}$,	initially: $\texttt{round}^{[i]} = 0$ for all i

function $\text{msg}(w, i)$

1: **if** round $< \text{diam}(\mathcal{S})$ **then**
2:     **return** max-id
3: **else**
4:     **return** null

function $\text{stf}(w, y)$

1: new-id$:= \max\{\texttt{max-id}, \text{largest identifier in } y\}$
2: **case**
3:     round $< \text{diam}(\mathcal{S})$:   new-lead := unknwn
4:     round $= \text{diam}(\mathcal{S})$ AND max-id $=$ my-id: new-lead := true
5:     round $= \text{diam}(\mathcal{S})$ AND max-id $>$ my-id: new-lead := false
6: **return** (my-id, new-id, new-lead, round $+1$)

- This version requires some global information, namely the diameter of the graph

Programming Assignment Info

- Homework starter code
  - RaiseExceptionError
  - Example Code
- Packages
  - Networkx https://networkx.org/documentation/stable/tutorial.html
  - Itertools
- Python conventions and help
  - Python Like You Mean It: https://www.pythonlikeyoumeanit.com/
- VS Code and Anaconda

Wrap Up

Recap

- Formalized our model to distinguish between
  - Physical dynamics
  - Communication network
  - Processor dynamics
- Modeled distributed algorithms and complexity thereof
- Demonstrated code for programming assignment

Next Time

- Controlling groups of robots
- Moving to reference-frame invariant control