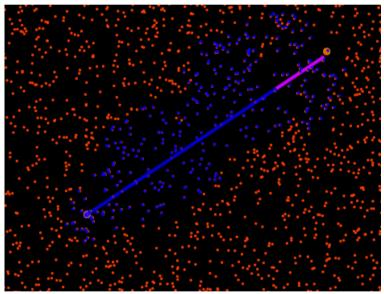
Discrete Approximation & Self-Healing

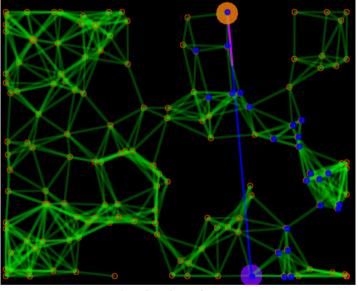
Jacob Beal Lecture 3 of 5 on Spatial Computing ISC-PIF Summer School, 2009



or: scalability and robustness cheap!



2000 devices

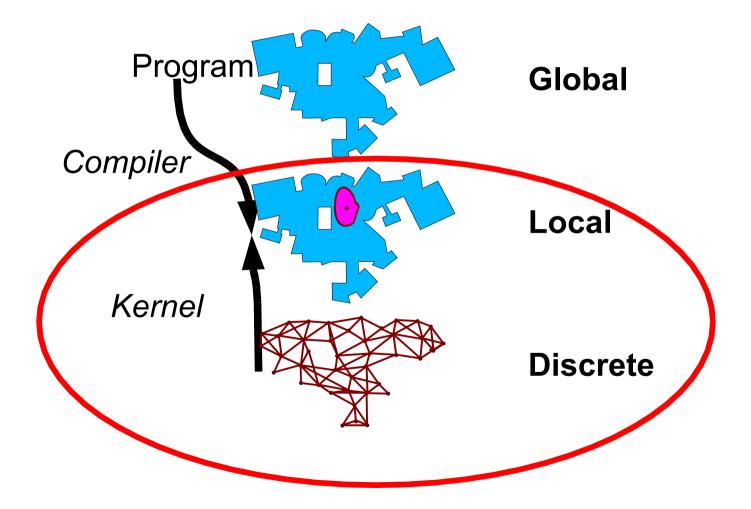


150 devices

Agenda

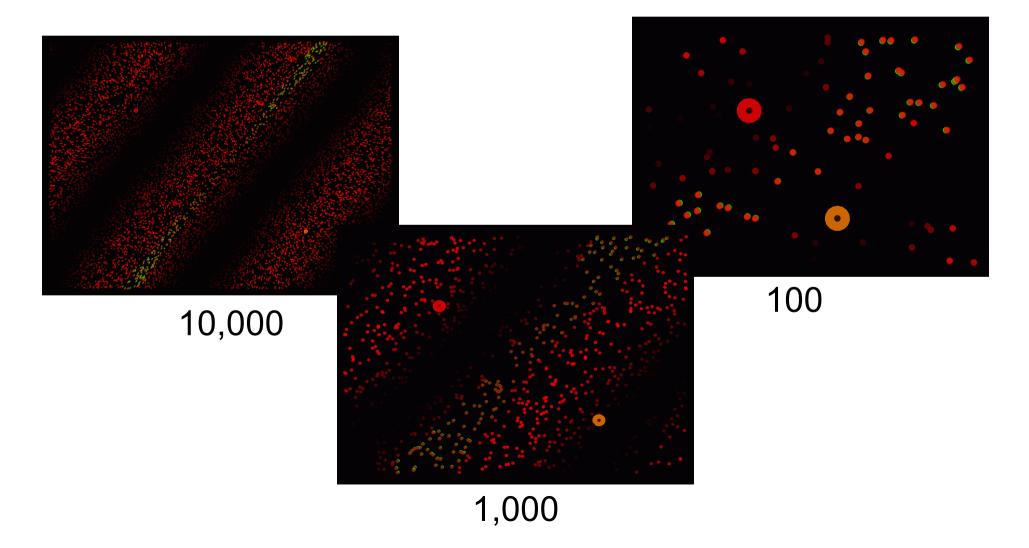
- Discrete Approximation
- Self-Healing Distance-To
- Proving Self-Stablization
- Correction Rate vs. Consistency

Global v. Local v. Discrete



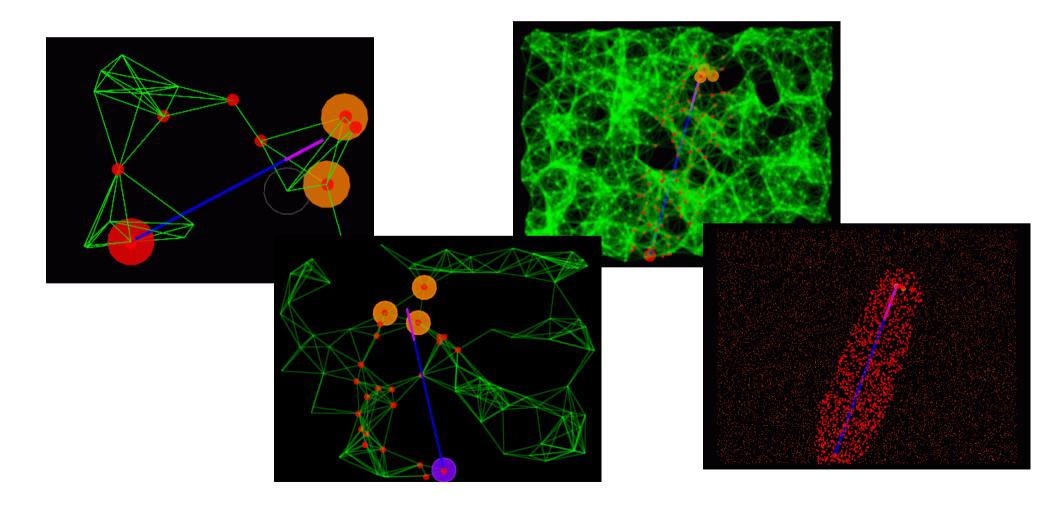
Gradual Degradation

• Plane wave at different resolutions:



Automatic Scaling

• Target tracking on 20 to 10,000 nodes:



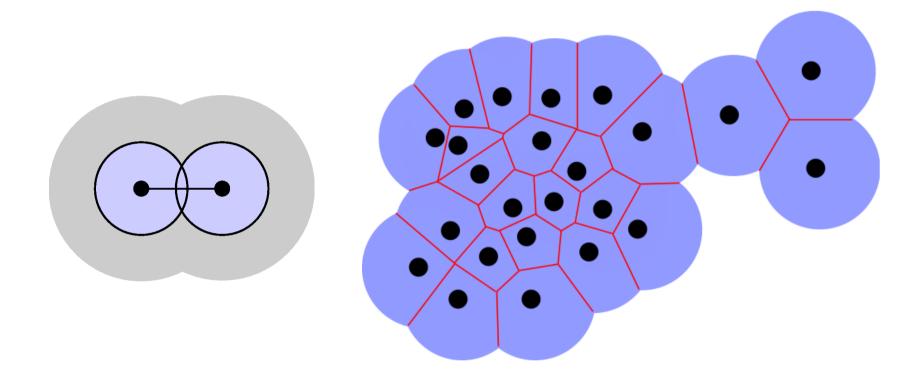
Discrete Model

- Dozens to billions of simple, unreliable agents
- Distributed through space, communicating by local broadcast
- Agents may be added or removed
- No guaranteed global services (e.g. time, naming, routing, coordinates)
- Relatively cheap power, memory, processing
- Partial synchrony

Kernel

- Responsibilities:
 - Emulate amorphous medium
 - Time evolution
 - Interface with sensors, actuators
 - Viral reprogramming
- Current platforms: simulator, Mica2 Mote, McLurkin's SwarmBots, Topobo, iRobot Create + Meraki

Discrete Space

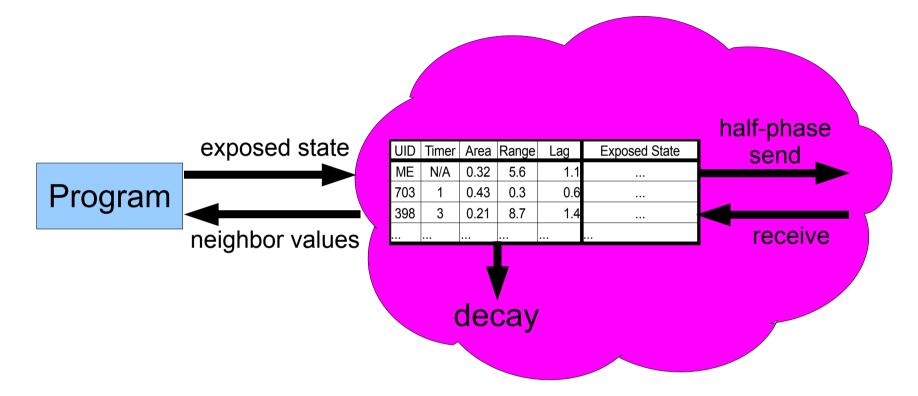


- Each devices represent nearby space
 - Best-effort space-time metrics from sensors
 - Each summary has a discrete equivalent

Discretizable Neighborhood Ops

- Space-Time Metrics:
 - nbr-range, nbr-angle, nbr-vec, is-self
 - nbr-lag, nbr-delay
 - density, infinitesimal, curvature
 - nbr
- Summary functions:
 - min-hood, max-hood, any-hood, all-hood
 - Same four "+" hole at self: e.g. min-hood+
 - int-hood
 - Abstraction breakers: sum-hood, fold-hood

Neighborhood Abstraction



- Aggregate access to best-effort estimate of neighbor state, space-time properties
- Neighbors decay without updates

Sensors & Actuators

- Indirect access via space-time operators
- Direct access by extending kernel
- Virtual sensors/actuators can connect to other programs running in parallel
 - I/O must be interpretable as CT stream of values
 - e.g. interface with a high-level planner with sensors "plan-ready" and "best-plan-to-take"

Agenda

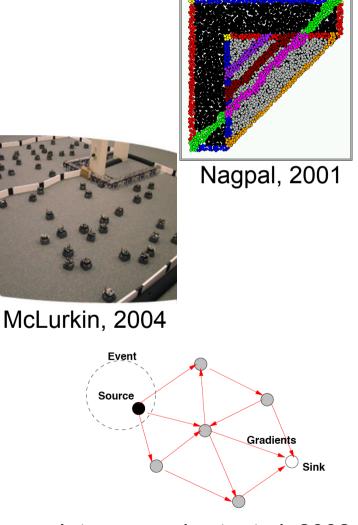
- Discrete Approximation
- Self-Healing Distance-To
- Proving Self-Stablization
- Correction Rate vs. Consistency

Distance-To (A.K.A. "gradient")

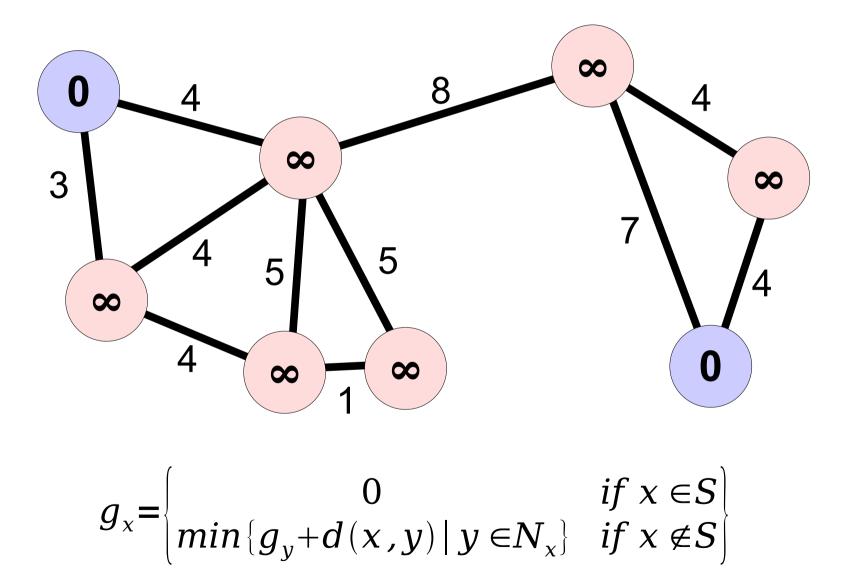
Common SA/SO building block

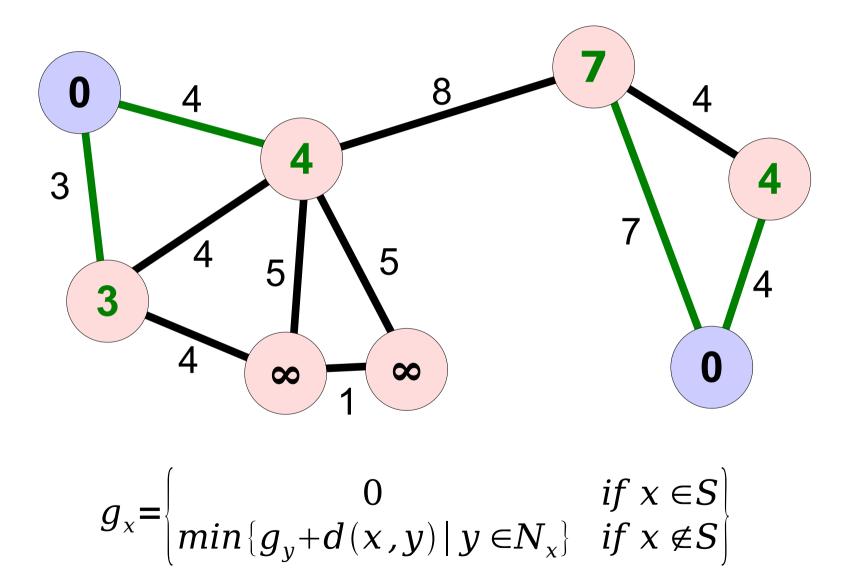
- Pattern Formation
 - Nagpal, Coore, Butera
- Distributed Robotics
 - Stoy, Werfel, McLurkin
- Networking
 - DV routing, Directed Diffusion

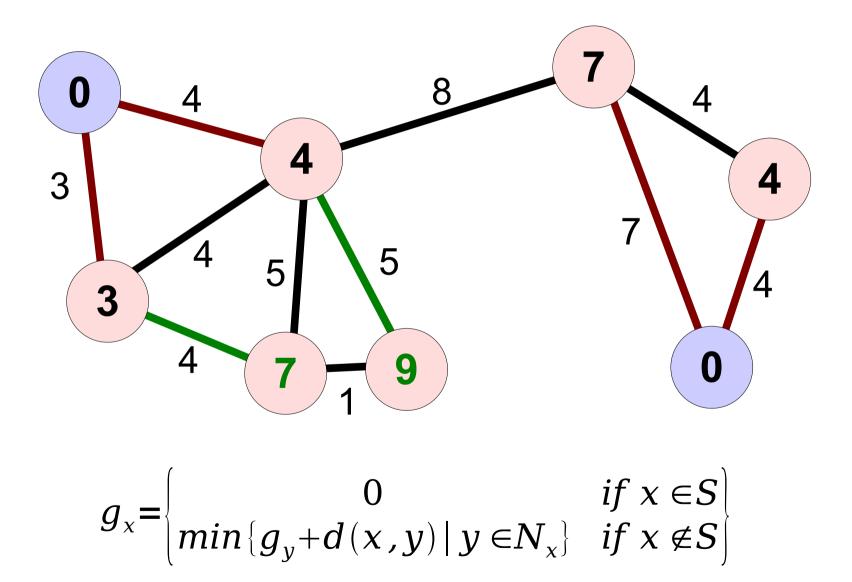
Need to adapt to changes

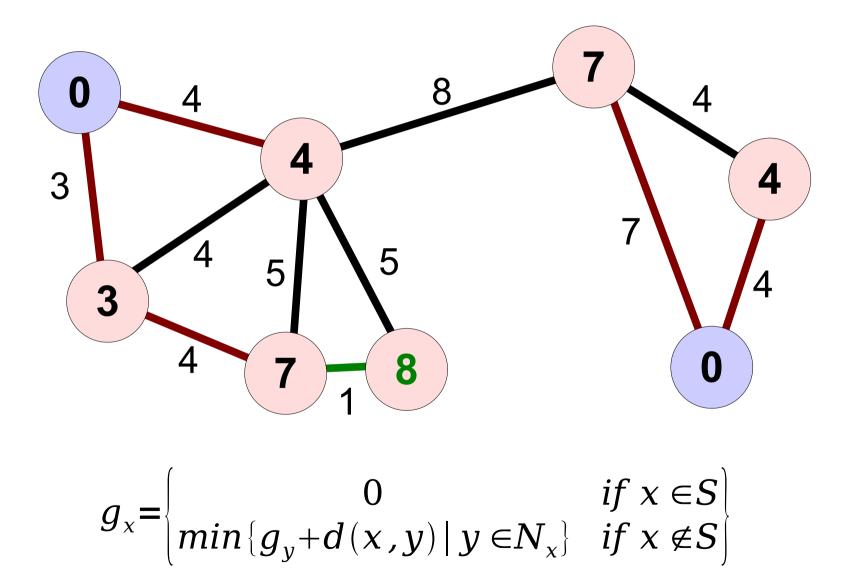


Intanagonwiwat, et al. 2002

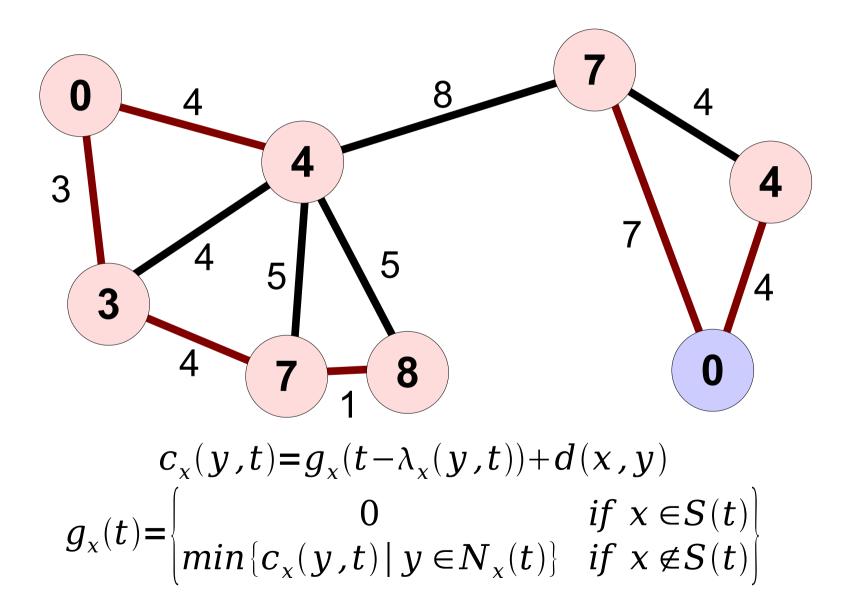


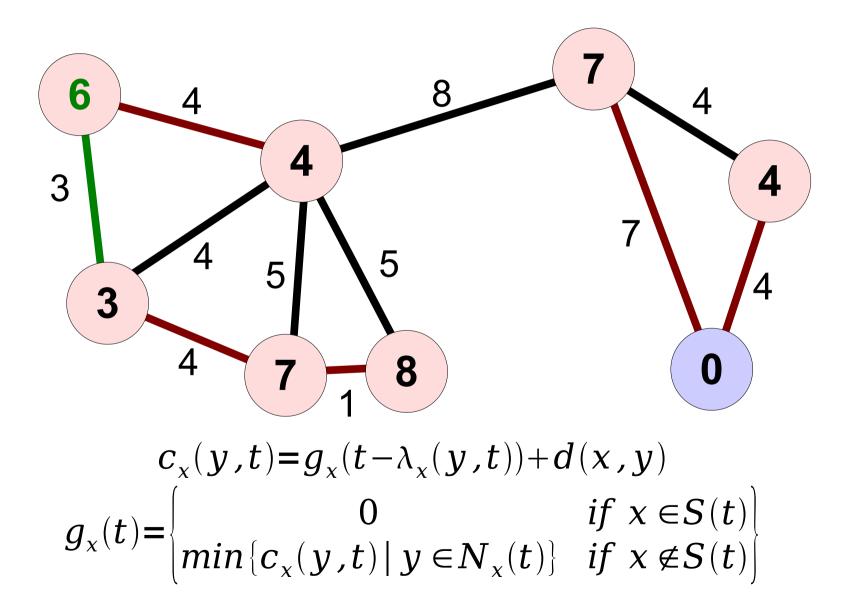


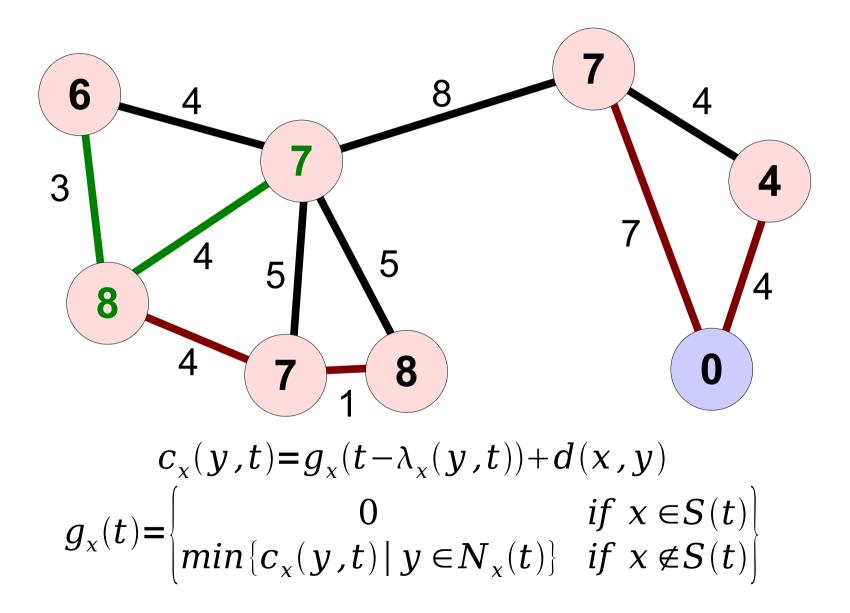


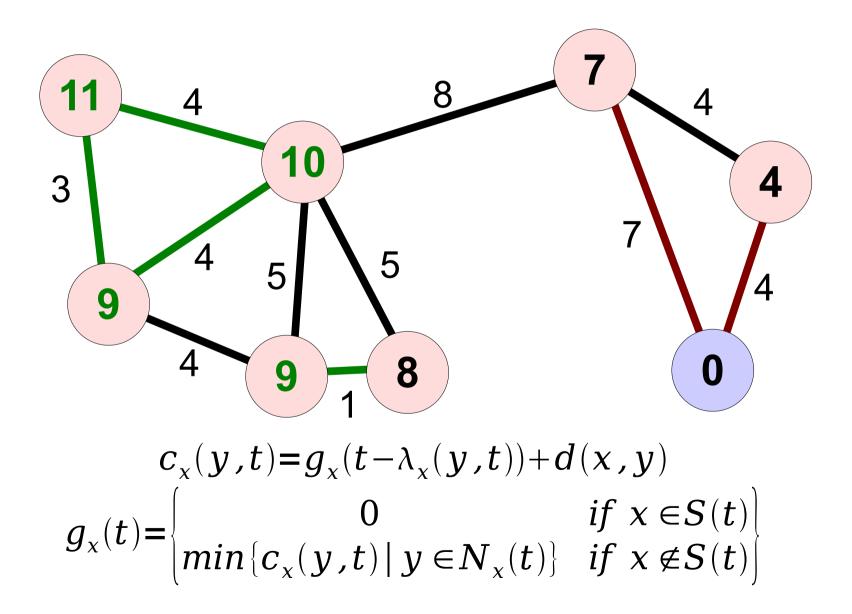


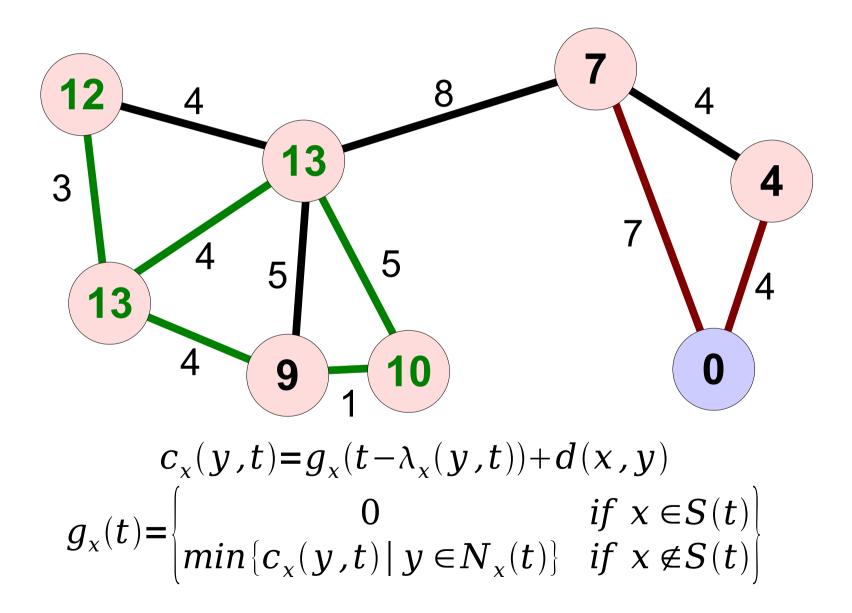
Distance-To + Communication Lag Falling Rising $\boldsymbol{\infty}$ $c_x(y,t)=g_x(t-\lambda_x(y,t))+d(x,y)$ $g_{x}(t) = \begin{cases} 0 & \text{if } x \in S(t) \\ \min\{c_{x}(y,t) \mid y \in N_{x}(t)\} & \text{if } x \notin S(t) \end{cases}$

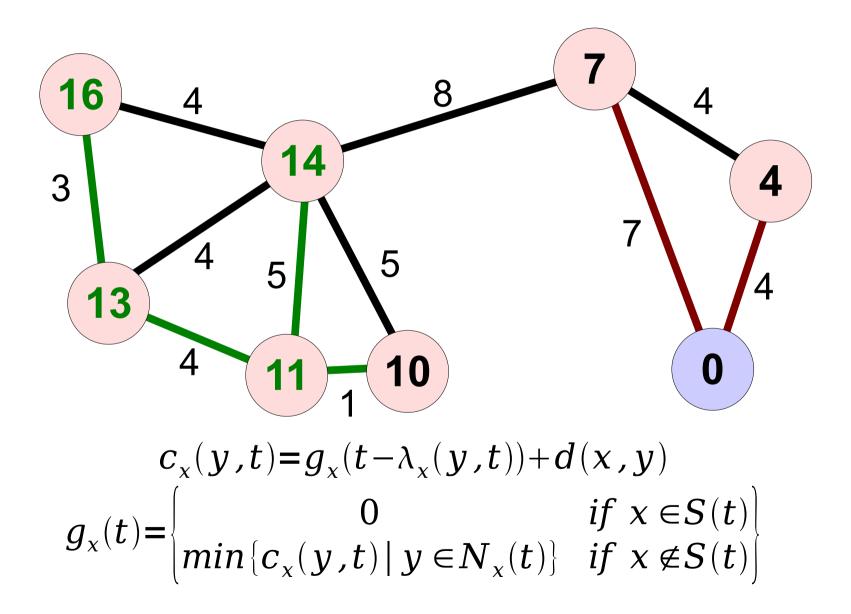


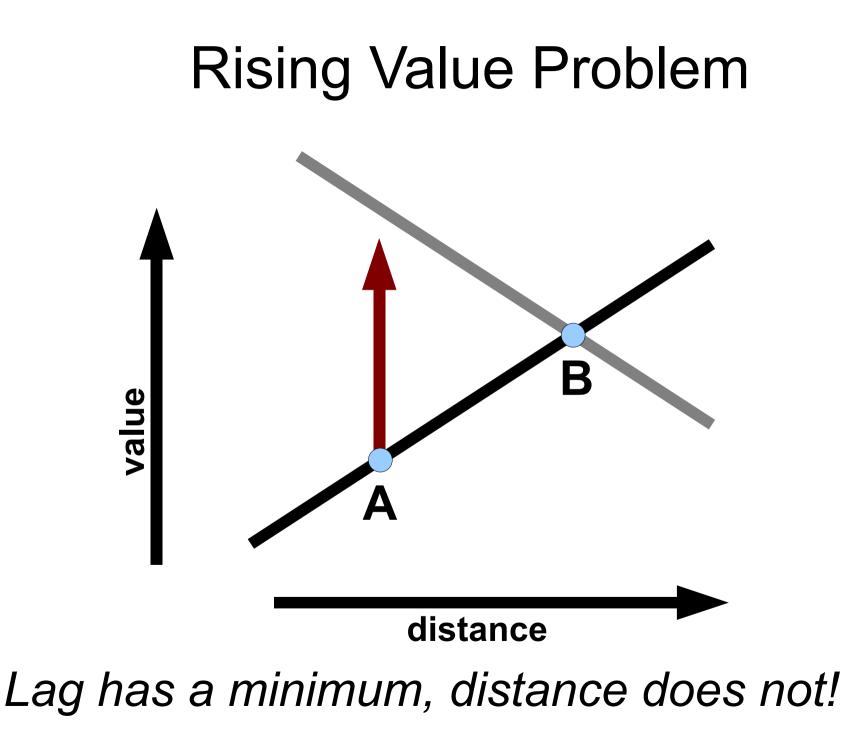










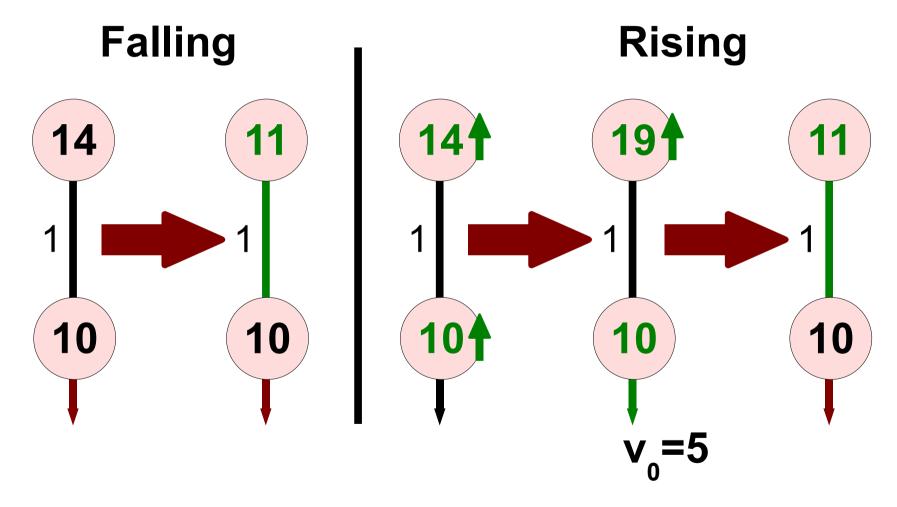


Previous Algorithms

- "Invalidate and Rebuild"
 - GRAB: single source, rebuild on high error
 - TTDD: static subgraph, rebuild on lost msg.
- "Incremental Repair"
 - Hopcount: Clement & Nagpal, Butera
 - Distorted Measure: Beal & Bachrach (naïve generalization of hopcount to continuous)

Can't exploit distance info in large nets

CRF-Distance-To: Local Deconstraint

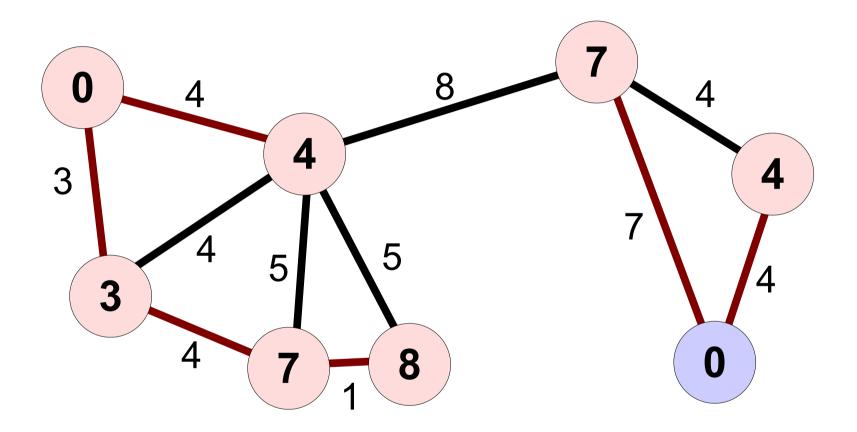


Self-stabilization in O(diameter)

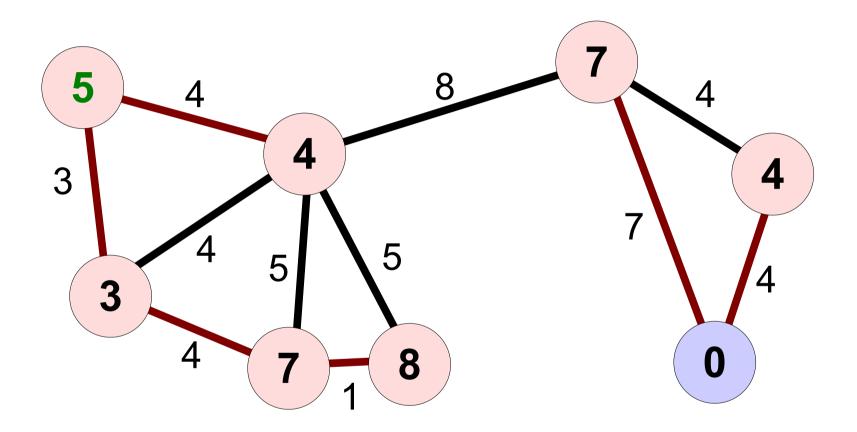
CRF-Distance-To: Local Deconstraint

$$\begin{split} c_{x}(y,t) &= g_{x}(t - \lambda_{x}(y,t)) + d(x,y) \\ c'_{x}(y,t) &= c_{x}(y,t) + (\lambda_{x}(y,t) + \Delta_{t}) \cdot v_{x}(t) \\ N'_{x}(t) &= \{ y \in N_{x}(t) \mid c'_{x}(y,t) \leq g_{x}(t - \Delta_{t}) \} \\ g_{x}(t) &= \begin{cases} 0 & if \ x \in S(t) \\ min\{c_{x}(y,t) \mid y \in N'_{x}(t)\} & if \ x \notin S(t), N'_{x}(t) \neq \emptyset \\ g_{x}(t) + v_{0} \cdot \Delta_{t} & if \ x \notin S(t), N'_{x}(t) = \emptyset \end{cases} \\ v_{x}(t) &= \begin{cases} 0 & if \ x \in S(t) \\ 0 & if \ x \notin S(t), N'_{x}(t) \neq \emptyset \\ v_{0} & if \ x \notin S(t), N'_{x}(t) = \emptyset \end{cases} \end{split}$$

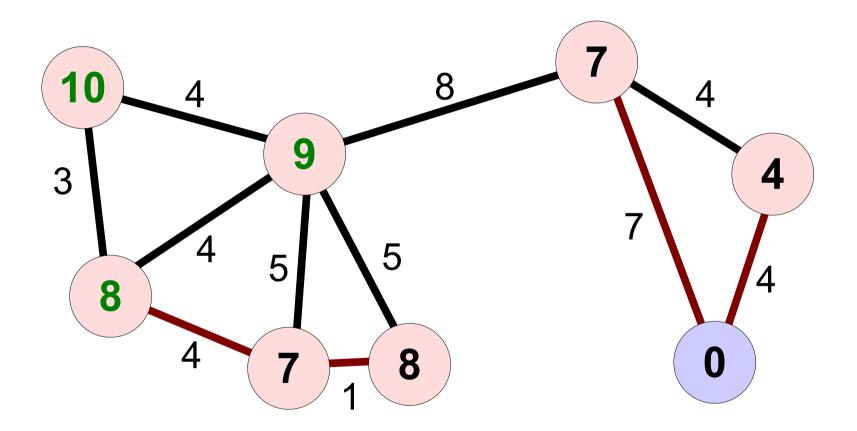
• Self-stabilization in O(diameter)



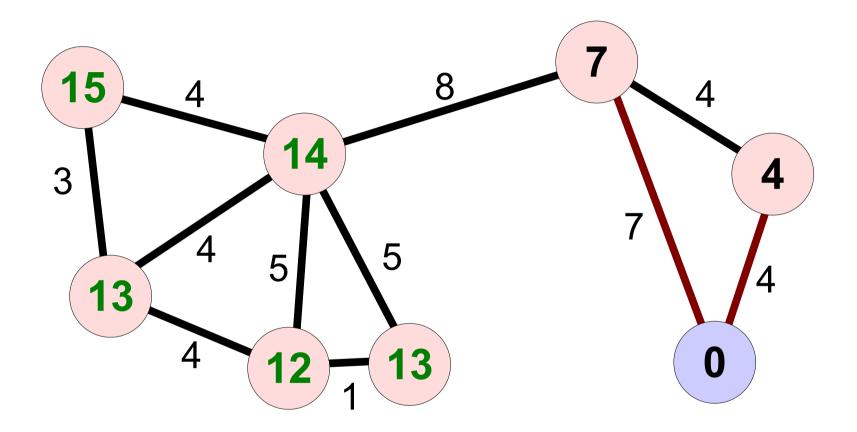
- zero at source
- rise at v_0 with relaxed constraint
- otherwise snap to constraint



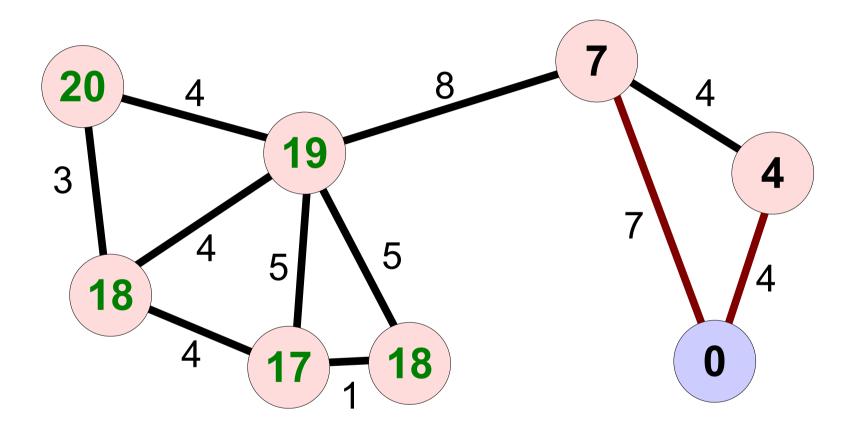
- zero at source
- rise at v_0 with relaxed constraint
- otherwise snap to constraint



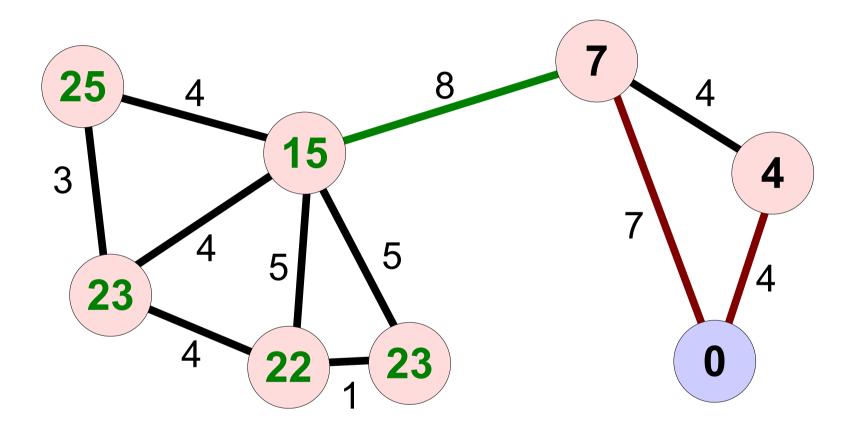
- zero at source
- rise at v_0 with relaxed constraint
- otherwise snap to constraint



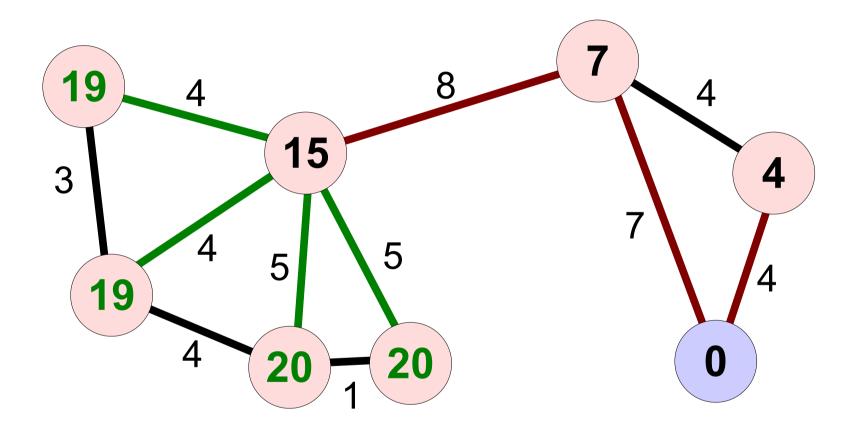
- zero at source
- rise at v_0 with relaxed constraint
- otherwise snap to constraint



- zero at source
- rise at v_0 with relaxed constraint
- otherwise snap to constraint

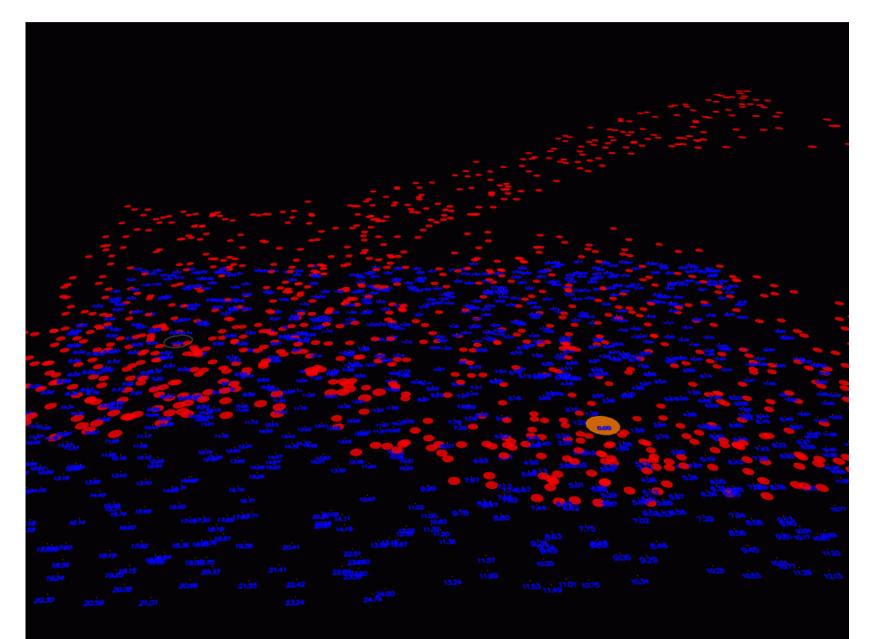


- zero at source
- rise at v_0 with relaxed constraint
- otherwise snap to constraint

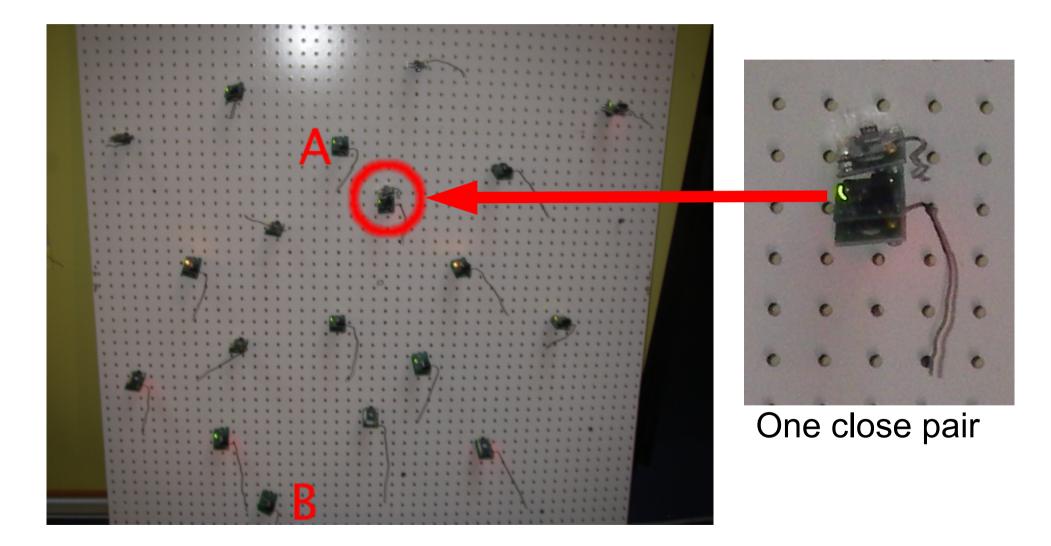


- zero at source
- rise at v_{o} with relaxed constraint
- otherwise snap to constraint

Simulated CRF-Distance-To

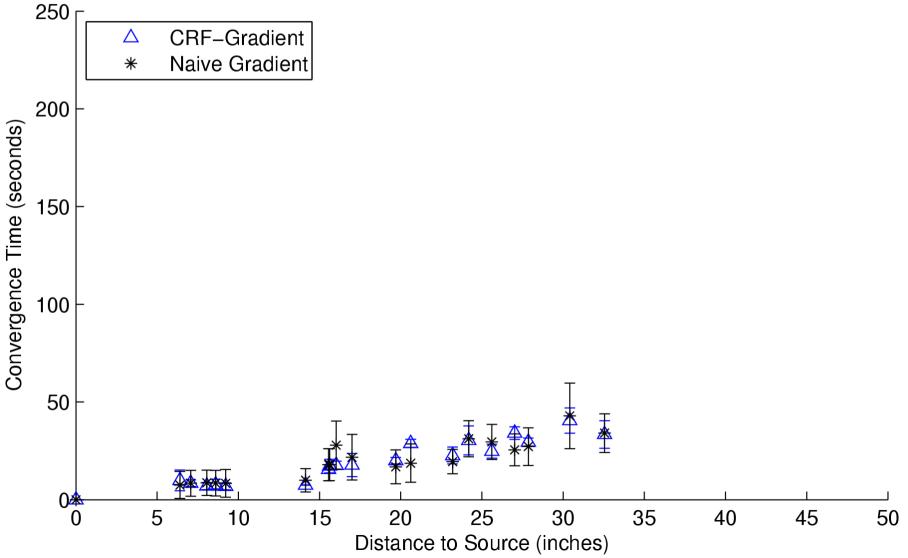


Experimental Setup



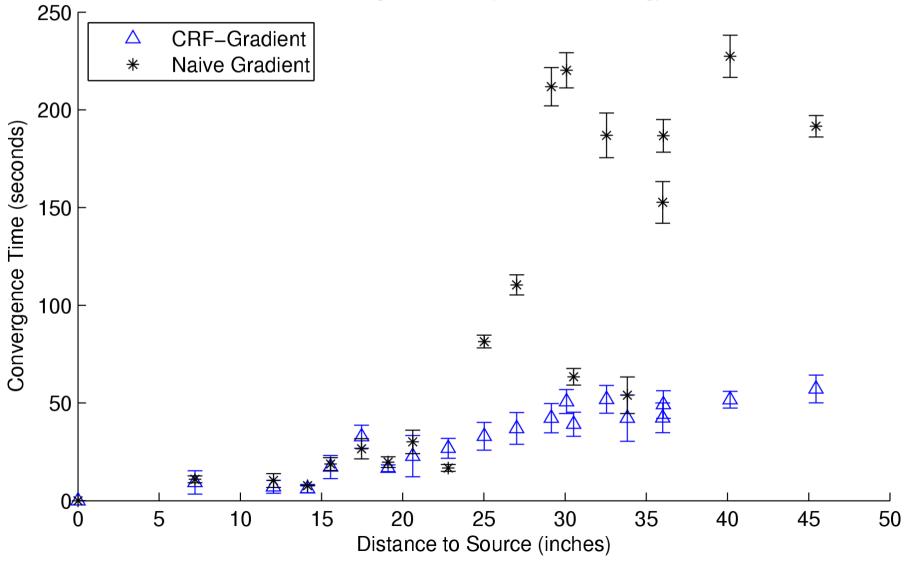
Experimental Results: Falling

Convergence Time (Close Pair Falling)

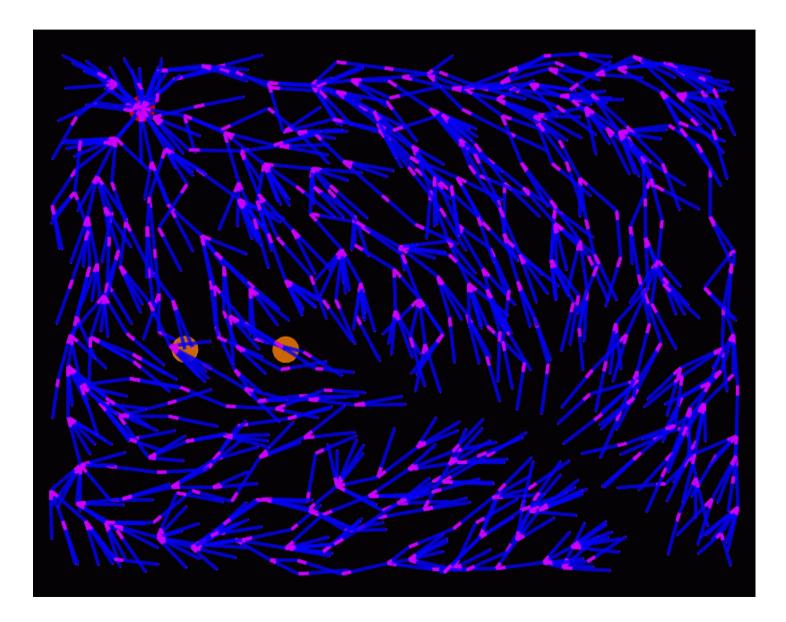


Experimental Results: Rising

Convergence Time (Close Pair Rising)



Generalized CRF



Feed-Forward Self-Stabilization



A composition of self-stabilizing components with no feedback is itself self-stabilizing in the sum of the times along the longest path

Agenda

- Discrete Approximation
- Self-Healing Distance-To
- Proving Self-Stablization
- Correction Rate vs. Consistency

Self-Stabilization vs. Self-Healing

An algorithm is *self-stabilizing* iff, given an arbitrary starting state, it converges to a correct state in finite time.

An algorithm is *self-healing* if it always incrementally adjusts its state towards a more correct state.

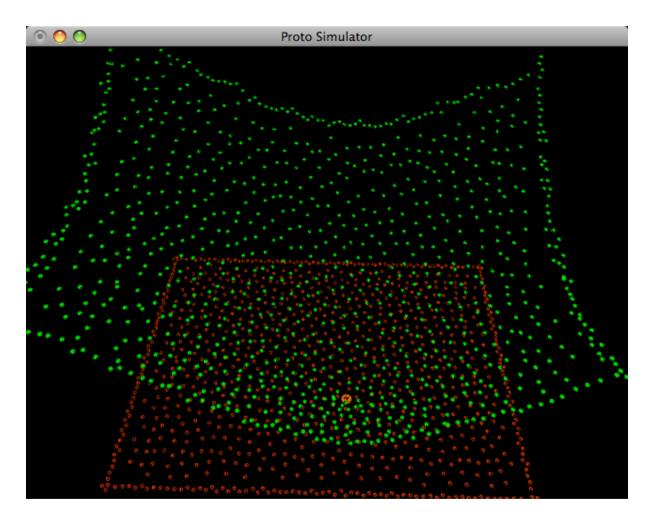
Proofing Self-Stablization for CRF

Let's work this proof out together...

Agenda

- Discrete Approximation
- Self-Healing Distance-To
- Proving Self-Stablization
- Correction Rate vs. Consistency

Perfection is expensive and "twitchy"



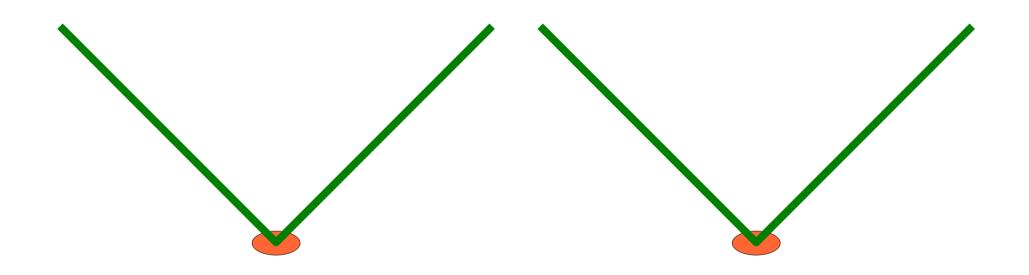
But most applications don't need perfection...

proto -n 1000 -r 10 "(all (mov (* 0.1 (disperse))) (green (distance-to (sense 1))))" -l -s 1 -m -w

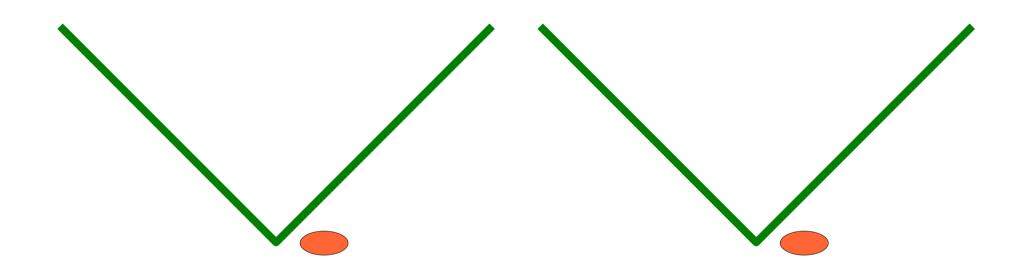
Making distance-to tolerate error

- Hysteresis?
 - Past a threshold, unbounded communication
- Low-pass filtering?
 - Worse! Value change != msg cost

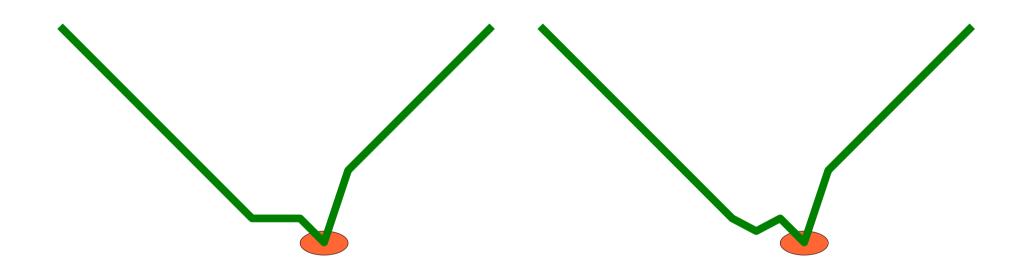
- "Elastic" connections!
 - Absorb error incrementally



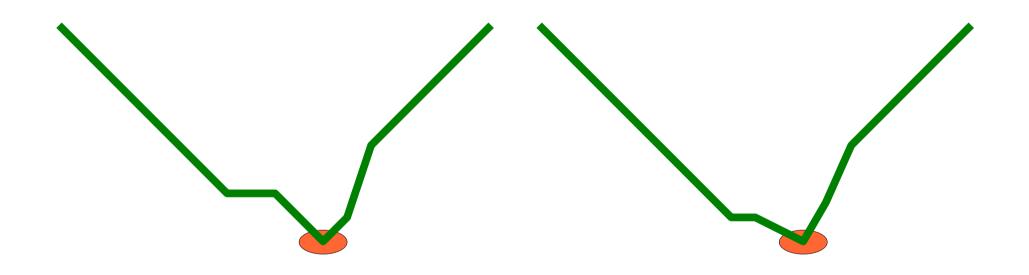
Attemped Perfection



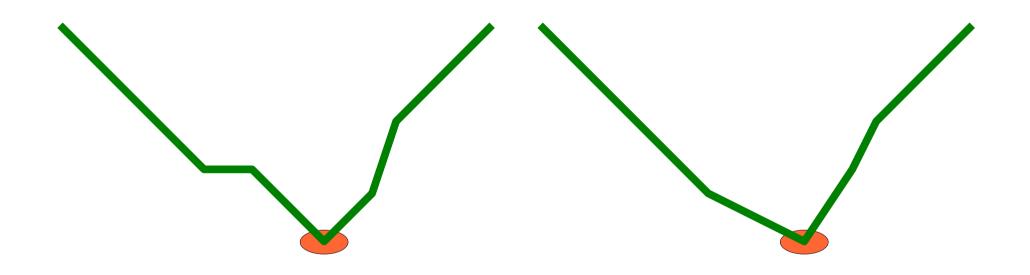
Attemped Perfection



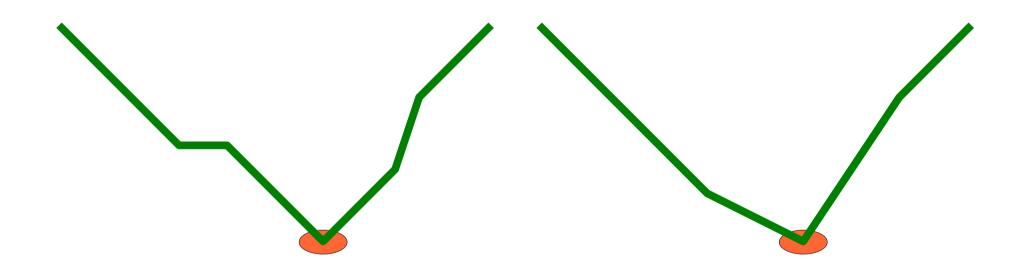
Attemped Perfection



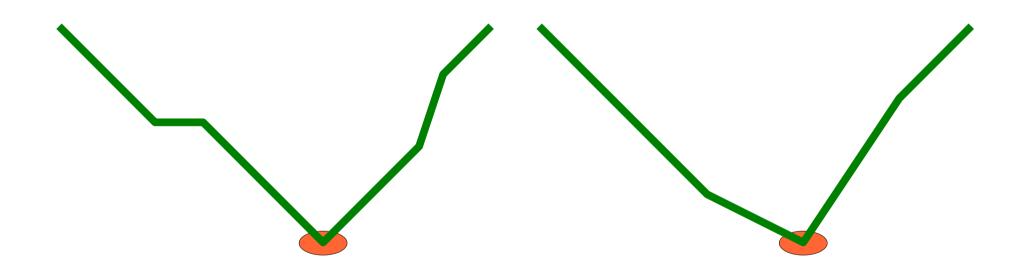
Attemped Perfection



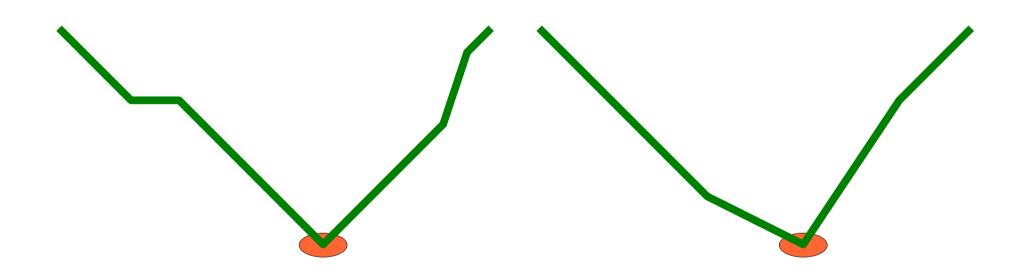
Attemped Perfection



Attemped Perfection



Attemped Perfection



Attemped Perfection

Managing error through slope

Goal: ε-acceptable values

 $\bar{g_x}(t) \cdot (1 - \epsilon) \leq g_x(t) \leq \bar{g_x}(t) \cdot (1 + \epsilon)$

• Add local constraint via slope:

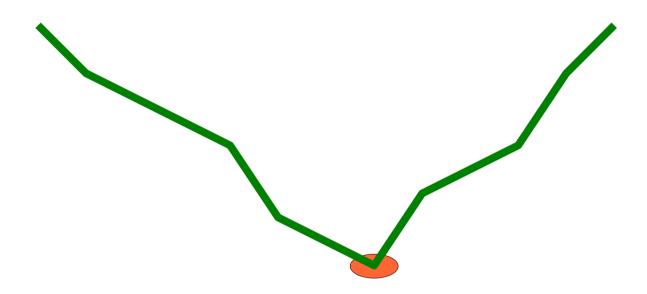
$$s_{x}(t) = max \left\{ \frac{g_{x}(t - \Delta_{t}) - g_{y}(t_{x,y})}{d(x, y, t_{x,y})} | y \in N_{x}(t) \right\}$$

 \rightarrow "flexible" distance-to

(allow small distortion for rising value problem)

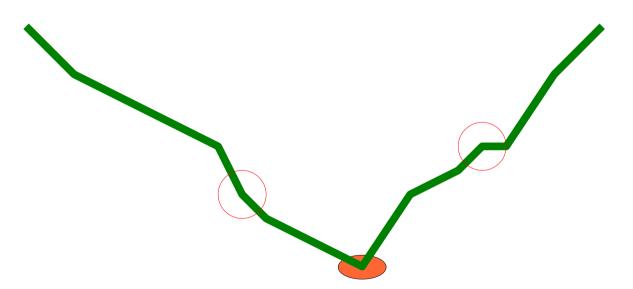
Getting the kinks out

- Flexed regions cannot absorb error
- Want eventual correctness



Getting the kinks out

- Flexed regions cannot absorb error
- Want eventual correctness



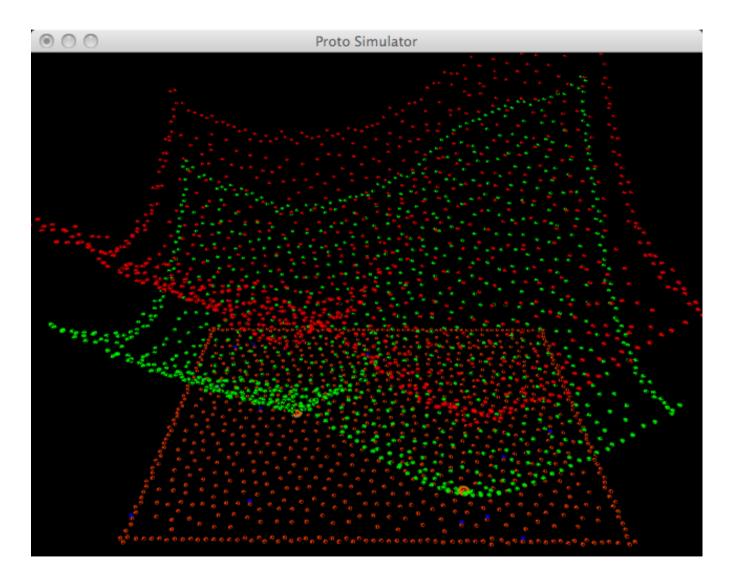
Solution: occasional ε =0 steps

Flex-Distance-To Algorithm (simplified)

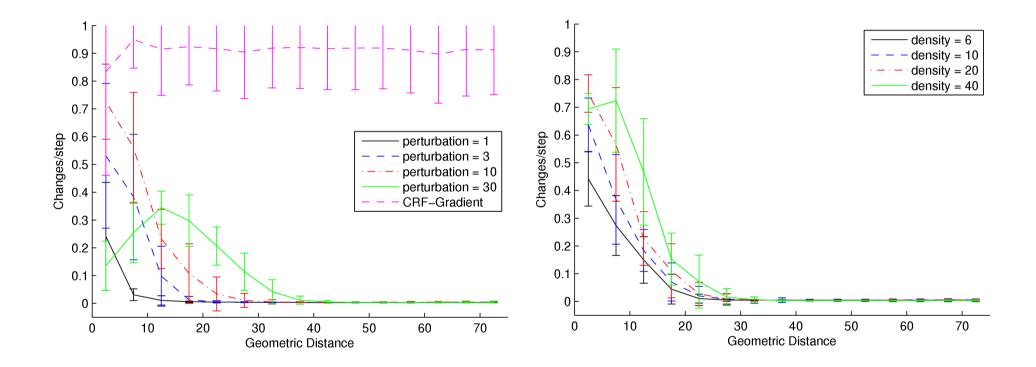
- Sources take $g_x(t)=0$
- Else measure maximum slope and minimum distance through neighbors (w. r/δ distortion):
 - If value is more than 2x lowest value through neighbor, snap to slope=1
 - Else if slope is not ε-acceptable, make ε-acceptable

– Once every $g_{x}(t)$ updates, use $\epsilon=0$

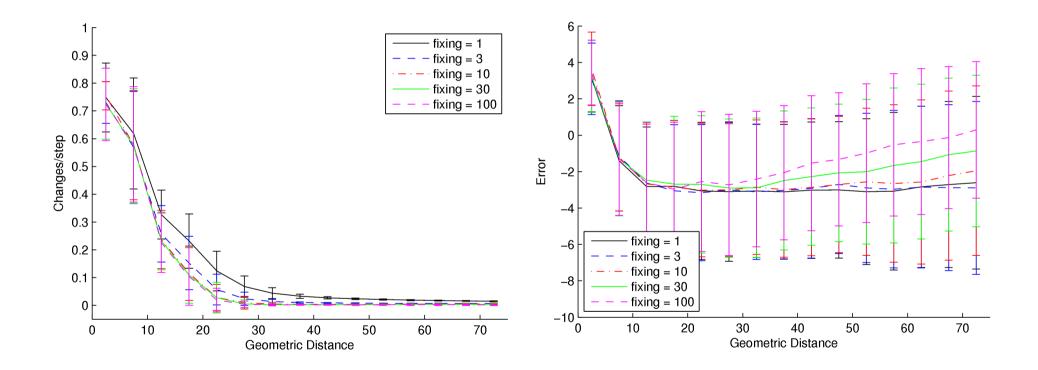
Flex-Distance-To vs. CRF-Distance-To



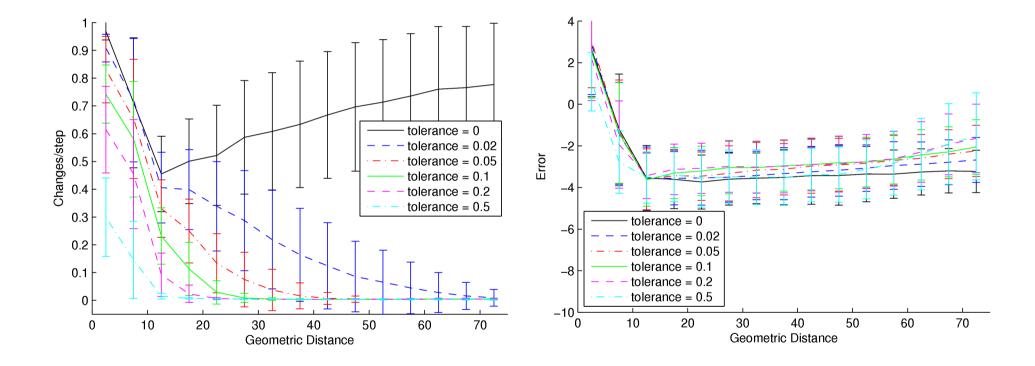
Perturbations affect limited range



Even infrequent repair helps



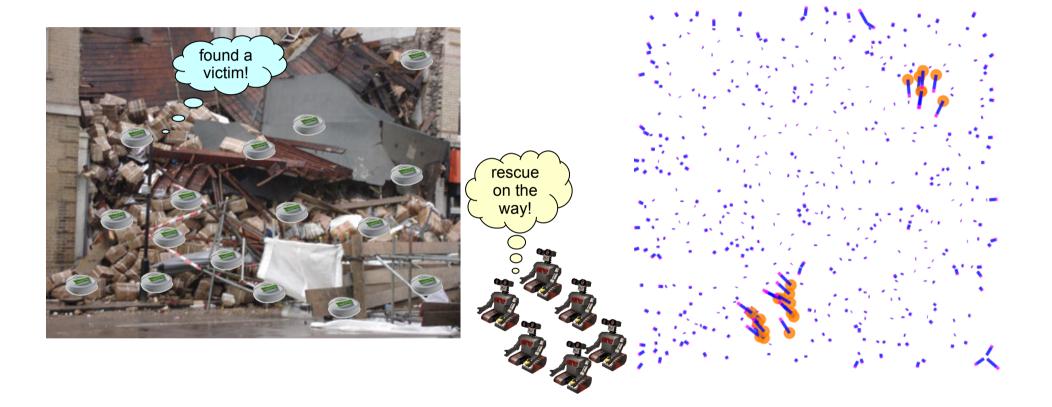
A little tolerance goes a long way



Summary

- Appropriately choosen amorphous medium operations discretize naturally.
- Self-healing algorithms adapt gracefully to changes in environment or program state.
- Feed-forward compositions of self-stabilizing algotihms are self-stabilizing.
- Healing rate and consistency can be traded off.

Tomorrow: Moving Devices



Robot motion = vector fields

Further Questions

- What is the optimal replacement policy when there are more neighbors than table memory?
- What is the optimal decay rate?
- How much energy can be saved by throttling update and decay rates?
- What are good ways to expose the cost/responsiveness tradeoff to the programmer?

Further Questions

- Are the neighborhood summary functions a cover of all useful approximable functions?
- Are there other basic space-time metrics needed for neighborhood computations?
- What is the best way to represent random number generation in continuous space-time?