Storage capacity of labeled graphs

Dana Angluin[1] James Aspnes[1] Rida A. Bazzi[2]
Jiang Chen[3] David Eisenstat[4] Goran Konjevod[2]

[1]Department of Computer Science, Yale University

[2]Department of Computer Science and Engineering, Arizona State University

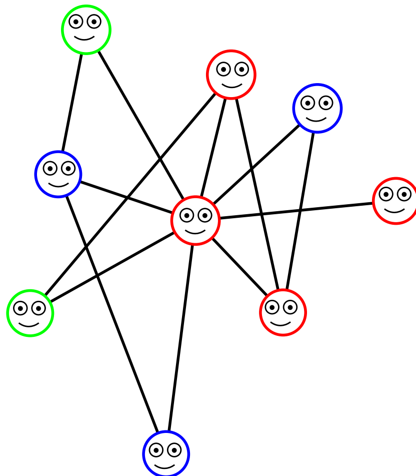
[3]Google

[4]Department of Computer Science, Brown University

September 22, 2010

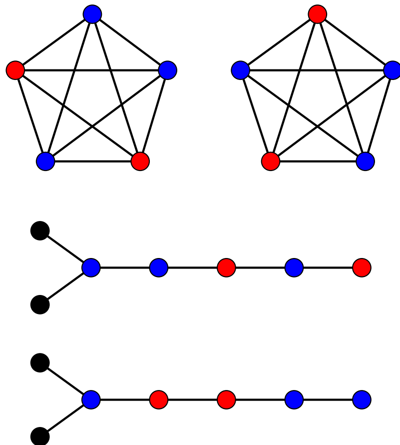
The basic picture

- Network of n anonymous finite-state machines.
- Graph structure is fixed.
- State of each machine $\Rightarrow$ one label per node.
- How much information can we store?



Structure of the graph matters!

- Lots of symmetries:
  - $\Rightarrow$ can't tell nodes with the same label apart
  - $\Rightarrow O(\log n)$ bits are enough to describe the state.
- Few symmetries:
  - $\Rightarrow$ most nodes are distinguishable
  - $\Rightarrow \Theta(n)$ bits are needed to describe the state.



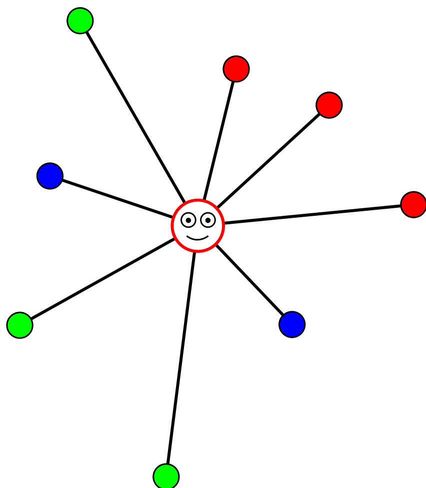
Information-theoretic vs effective capacity

- **Information-theoretic capacity** = log (number of distinguishable labelings) = how much space a program running *outside* the network can get.
- **Effective capacity** = how much space a program running *inside* the network can get.
- Information-theoretic capacity $\geq$ effective capacity.
- When are they equal?

Graph Turing machines

Graph Turing machine is like a regular Turing machine with the tape replaced by a graph.

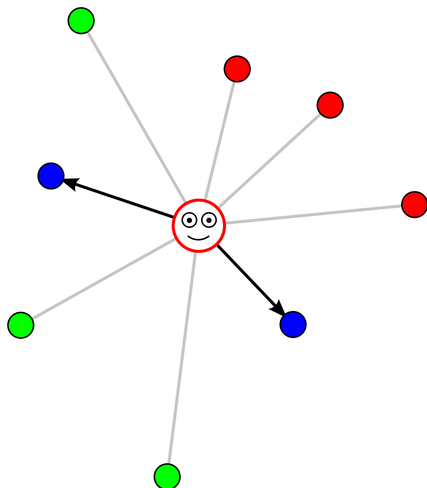
- Finite-state controller sees label on current node and *set* of labels on adjacent nodes (without multiplicities!)
- No sense of direction: controller moves by choosing a label.
- If more than one neighbor has that label, adversary chooses which to move to.



Graph Turing machines

Graph Turing machine is like a regular Turing machine with the tape replaced by a graph.

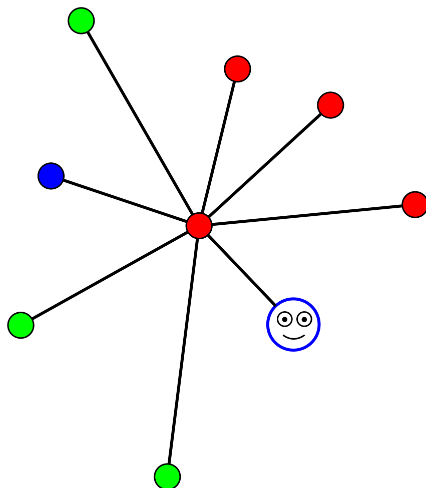
- Finite-state controller sees label on current node and *set* of labels on adjacent nodes (without multiplicities!)
- No sense of direction: controller moves by choosing a label.
- If more than one neighbor has that label, adversary chooses which to move to.



Graph Turing machines

Graph Turing machine is like a regular Turing machine with the tape replaced by a graph.

- Finite-state controller sees label on current node and *set* of labels on adjacent nodes (without multiplicities!)
- No sense of direction: controller moves by choosing a label.
- If more than one neighbor has that label, adversary chooses which to move to.



Graph Turing machines: formal version

- Input: Graph G and initial vertex v_0.
- Specification: Tuple (Σ, Q, q_0, δ) where
  - Σ is the label set,
  - Q is the state space for the controller,
  - q_0 is the initial state, and
  - $\delta : Q \times \Sigma \times \mathcal{P}(\Sigma) \to (Q \cup \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Sigma \times \Sigma$ is the transition function.
- Transitions:
  - Let $\delta(q, \ell(v), \{\ell(v') | (v, v') \in E\}) = (q', s, t)$.
  - $q \leftarrow q'$.
  - $\ell(v) \leftarrow s$.
  - Move controller to some v' where $\ell(v') = t$.

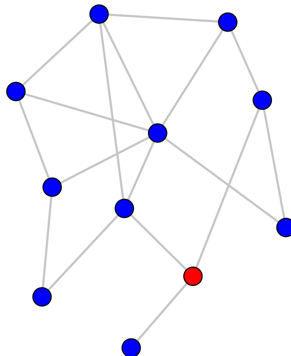
Effective capacity

- Idea: Measure capacity by size of the largest ordinary Turing machine we can simulate.
- Problem: Not well-defined for a single graph (can put as much storage as we like in the controller).
- **Effective capacity** is defined for *classes* of graphs $\mathcal{G}$.
- A class $\mathcal{G}$ has effective capacity $f(G)$ if:
  - For any standard Turing machine M,
  - There is a graph Turing machine M', such that
  - For any graph G in $\mathcal{G}$ and vertex v_0 of G,
  - $M'(G, v_0)$ simulates M running on a blank tape with $f(G)$ cells.

Tool: graph traversal

Subroutine allows controller to visit every node in the graph.

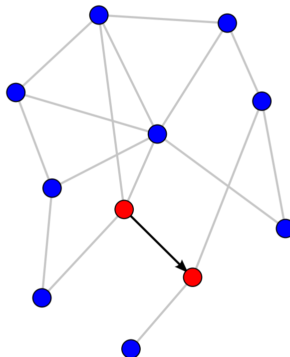
- Basic idea: use depth-first search, storing stack in graph itself.
- But stack may get tangled up.
- Solution: Assign a layer number mod 3 to each node using breadth-first search. (Itkis and Levin, 1994).
- DFS now only considers children in next layer.
- Total time is $O(n)$.



Tool: graph traversal

Subroutine allows controller to visit every node in the graph.

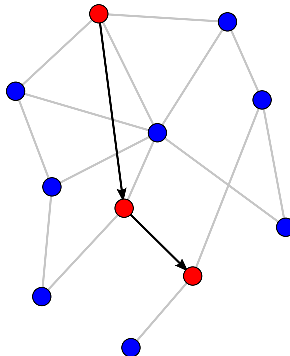
- Basic idea: use depth-first search, storing stack in graph itself.
- But stack may get tangled up.
- Solution: Assign a layer number mod 3 to each node using breadth-first search. (Itkis and Levin, 1994).
- DFS now only considers children in next layer.
- Total time is $O(n)$.



Tool: graph traversal

Subroutine allows controller to visit every node in the graph.

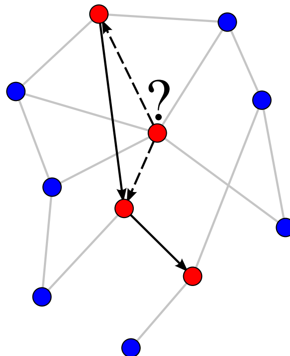
- Basic idea: use depth-first search, storing stack in graph itself.
- But stack may get tangled up.
- Solution: Assign a layer number mod 3 to each node using breadth-first search. (Itkis and Levin, 1994).
- DFS now only considers children in next layer.
- Total time is $O(n)$.



Tool: graph traversal

Subroutine allows controller to visit every node in the graph.

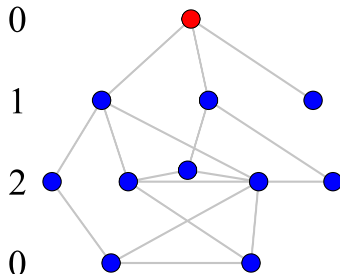
- Basic idea: use depth-first search, storing stack in graph itself.
- But stack may get tangled up.
- Solution: Assign a layer number mod 3 to each node using breadth-first search. (Itkis and Levin, 1994).
- DFS now only considers children in next layer.
- Total time is $O(n)$.



Tool: graph traversal

Subroutine allows controller to visit every node in the graph.

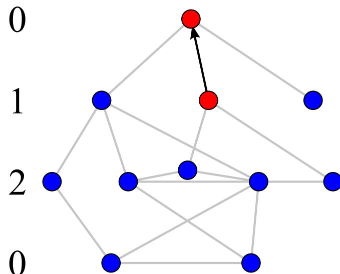
- Basic idea: use depth-first search, storing stack in graph itself.
- But stack may get tangled up.
- Solution: Assign a layer number mod 3 to each node using breadth-first search. (Itkis and Levin, 1994).
- DFS now only considers children in next layer.
- Total time is $O(n)$.



Tool: graph traversal

Subroutine allows controller to visit every node in the graph.

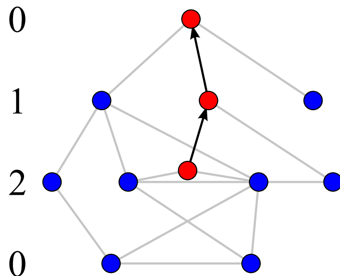
- Basic idea: use depth-first search, storing stack in graph itself.
- But stack may get tangled up.
- Solution: Assign a layer number mod 3 to each node using breadth-first search. (Itkis and Levin, 1994).
- DFS now only considers children in next layer.
- Total time is $O(n)$.



Tool: graph traversal

Subroutine allows controller to visit every node in the graph.

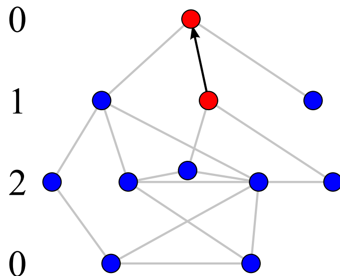
- Basic idea: use depth-first search, storing stack in graph itself.
- But stack may get tangled up.
- Solution: Assign a layer number mod 3 to each node using breadth-first search. (Itkis and Levin, 1994).
- DFS now only considers children in next layer.
- Total time is $O(n)$.



Tool: graph traversal

Subroutine allows controller to visit every node in the graph.

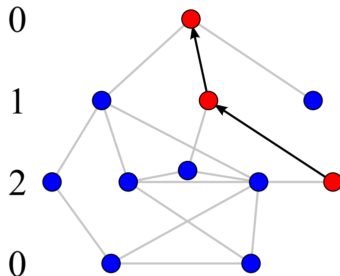
- Basic idea: use depth-first search, storing stack in graph itself.
- But stack may get tangled up.
- Solution: Assign a layer number mod 3 to each node using breadth-first search. (Itkis and Levin, 1994).
- DFS now only considers children in next layer.
- Total time is $O(n)$.



Tool: graph traversal

Subroutine allows controller to visit every node in the graph.

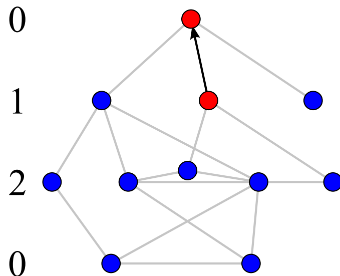
- Basic idea: use depth-first search, storing stack in graph itself.
- But stack may get tangled up.
- Solution: Assign a layer number mod 3 to each node using breadth-first search. (Itkis and Levin, 1994).
- DFS now only considers children in next layer.
- Total time is $O(n)$.



Tool: graph traversal

Subroutine allows controller to visit every node in the graph.

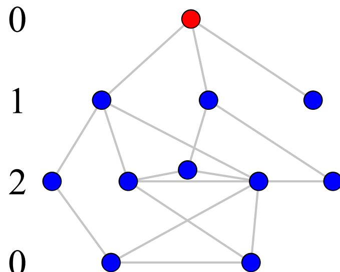
- Basic idea: use depth-first search, storing stack in graph itself.
- But stack may get tangled up.
- Solution: Assign a layer number mod 3 to each node using breadth-first search. (Itkis and Levin, 1994).
- DFS now only considers children in next layer.
- Total time is $O(n)$.



Tool: graph traversal

Subroutine allows controller to visit every node in the graph.

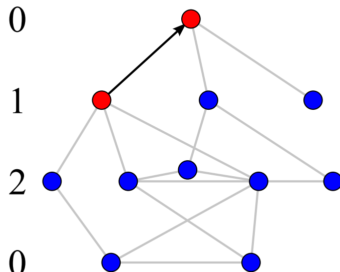
- Basic idea: use depth-first search, storing stack in graph itself.
- But stack may get tangled up.
- Solution: Assign a layer number mod 3 to each node using breadth-first search. (Itkis and Levin, 1994).
- DFS now only considers children in next layer.
- Total time is $O(n)$.



Tool: graph traversal

Subroutine allows controller to visit every node in the graph.

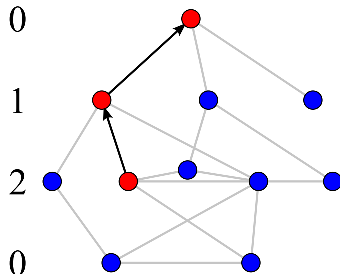
- Basic idea: use depth-first search, storing stack in graph itself.
- But stack may get tangled up.
- Solution: Assign a layer number mod 3 to each node using breadth-first search. (Itkis and Levin, 1994).
- DFS now only considers children in next layer.
- Total time is $O(n)$.



Tool: graph traversal

Subroutine allows controller to visit every node in the graph.

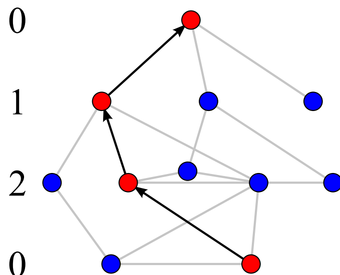
- Basic idea: use depth-first search, storing stack in graph itself.
- But stack may get tangled up.
- Solution: Assign a layer number mod 3 to each node using breadth-first search. (Itkis and Levin, 1994).
- DFS now only considers children in next layer.
- Total time is $O(n)$.



Tool: graph traversal

Subroutine allows controller to visit every node in the graph.

- Basic idea: use depth-first search, storing stack in graph itself.
- But stack may get tangled up.
- Solution: Assign a layer number mod 3 to each node using breadth-first search. (Itkis and Levin, 1994).
- DFS now only considers children in next layer.
- Total time is $O(n)$.



Every graph has effective capacity $\Omega(\log n)$

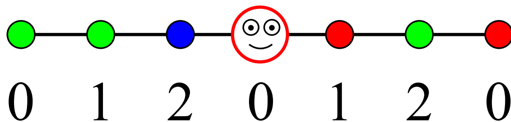
- Count from 0 to n by storing values in unary, with one bit per node.
- Each of these operations takes one graph traversal:

$c \leftarrow c/2$	Change every other 1 to a 0.
$c \leftarrow c-1$	Find a 1 and change it to 0.
$c \leftarrow c+1$	Find a 0 and change it to 1.
$c \stackrel{?}{=} 0$	Traverse the graph looking for a 1.
$c \bmod 2 \stackrel{?}{=} 0$	Sum up all bits mod 2.

- Doubling can be accomplished in quadratic time by repeatedly adding 2 to a second counter.
- This gives an $O(\log n)$-bit counter in any graph.
- $\Rightarrow$ (Minsky, 1967) Any graph can simulate an $O(\log n)$-space Turing machine.

Capacity of lines is $\Theta(n)$

- Use a mod-3 slope to orient the line.
- Now we have an ordinary Turing machine.

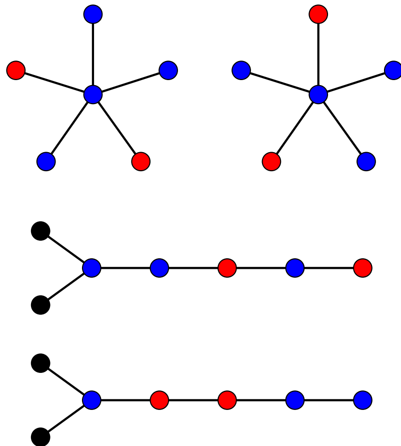


Capacity of trees

Information-theoretic capacity varies from

- $\Theta(\log n)$ for stars, to
- $\Theta(n)$ for lines.

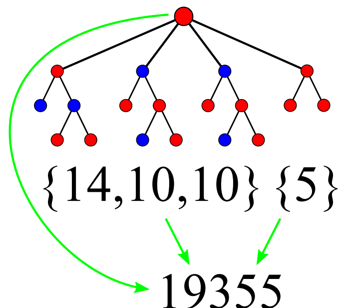
We'd like to get the same values for effective capacity.



Information-theoretic capacity

Recursively ranking a rooted tree:

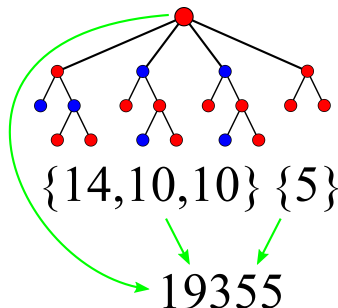
- Group subtrees into isomorphism classes.
- Convert each subtree labeling into a number: this gives a multiset for each equivalence class.
- Multisets + root → rank for whole tree.



Information-theoretic capacity = log(number of possible ranks) = $\log_2(31200) \approx 14.93$.

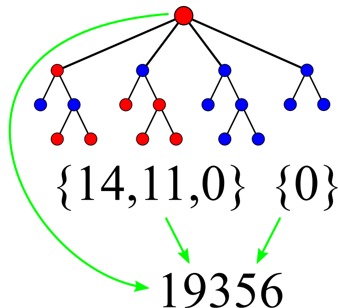
Extracting the information-theoretic capacity

- Increment operation:
  - Find lowest-valued subtree in rightmost isomorphism class that is not maxed out.
  - Increment it.
  - Reset less-significant subtrees to zero.
- All of this can be done using finite-state controller plus previous tricks for storing stacks in tree etc.
- Other counter operations are similar.



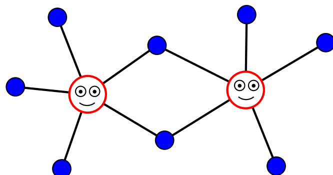
Extracting the information-theoretic capacity

- Increment operation:
  - Find lowest-valued subtree in rightmost isomorphism class that is not maxed out.
  - Increment it.
  - Reset less-significant subtrees to zero.
- All of this can be done using finite-state controller plus previous tricks for storing stacks in tree etc.
- Other counter operations are similar.



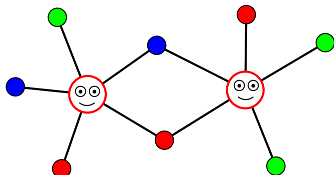
Capacity of random graphs is $\Theta(n)$

- Problem: Can't tell nodes apart.
- Solution: Assign random labels.
- Sort nodes by degree and neighborhood labeling and use them as TM tape.
- Works for $G_{n,p}$ model where edge probability p is polynomial in n.



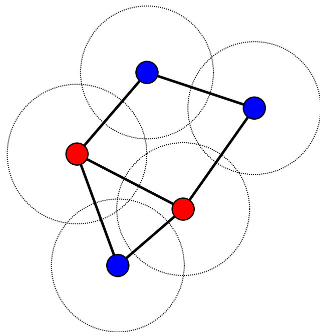
Capacity of random graphs is $\Theta(n)$

- Problem: Can't tell nodes apart.
- Solution: Assign random labels.
- Sort nodes by degree and neighborhood labeling and use them as TM tape.
- Works for $G_{n,p}$ model where edge probability p is polynomial in n.



Conclusions and open problems

- We extract full information-theoretic capacity from:
  - Highly-symmetric graphs.
  - Trees.
  - Random graphs with polynomial edge probabilities.
- Observation: GRAPH ISOMORPHISM is not hard on these graphs.
- What about:
  - Random geometric graphs?
  - Planar graphs?



Warning!

Proceedings version omits many details.
Full paper (and these slides) available at:
`www.cs.yale.edu/homes/aspnes/graph-capacity.html`