Detecting Communities in Dynamic Networks

Sanjukta Bhowmick
Shweta Bansal
Kelly Fermoyle
Padma Raghavan

The Pennsylvania State University

(06/09/2009)

Introduction

- Networks represent patterns of interaction
  - Social Interactions, Citations, Internet, etc.
- Community detection and analysis provides important information about the network
- Communities represented by highly connected vertices

Community Detection

Hierarchical Clustering

Divisive

Top-down

Deleting edges from network

Girvan Newman (2002)

Based on edge betweenness

O(|E|^2*|V|)

Agglomerative

Bottom-up

Adding edges to network

Clauset, Newman, Moore(2004)

Based on modularity

O((|E|+|V|)*|V|)

Modularity-I

- Modularity: Improvement on random connectivity
- High modularity is good---better connectivity than random
- Goal: To form communities that maximize modularity

Modularity-II

- Fraction of edges connecting community i to community j : ***C(i,j)***

- Fraction of edges attached to community i : ***a(i)=***$\sum_j$ ***C(i,j)***

- Probability of an edge existing between community i and community j : ***a(i)*a(j)***

- Modularity: ***Q =*** $\sum_i$ ***(C(i,i)-a(i)^2)***
  - Fraction of within community edges – expected edges for random connections

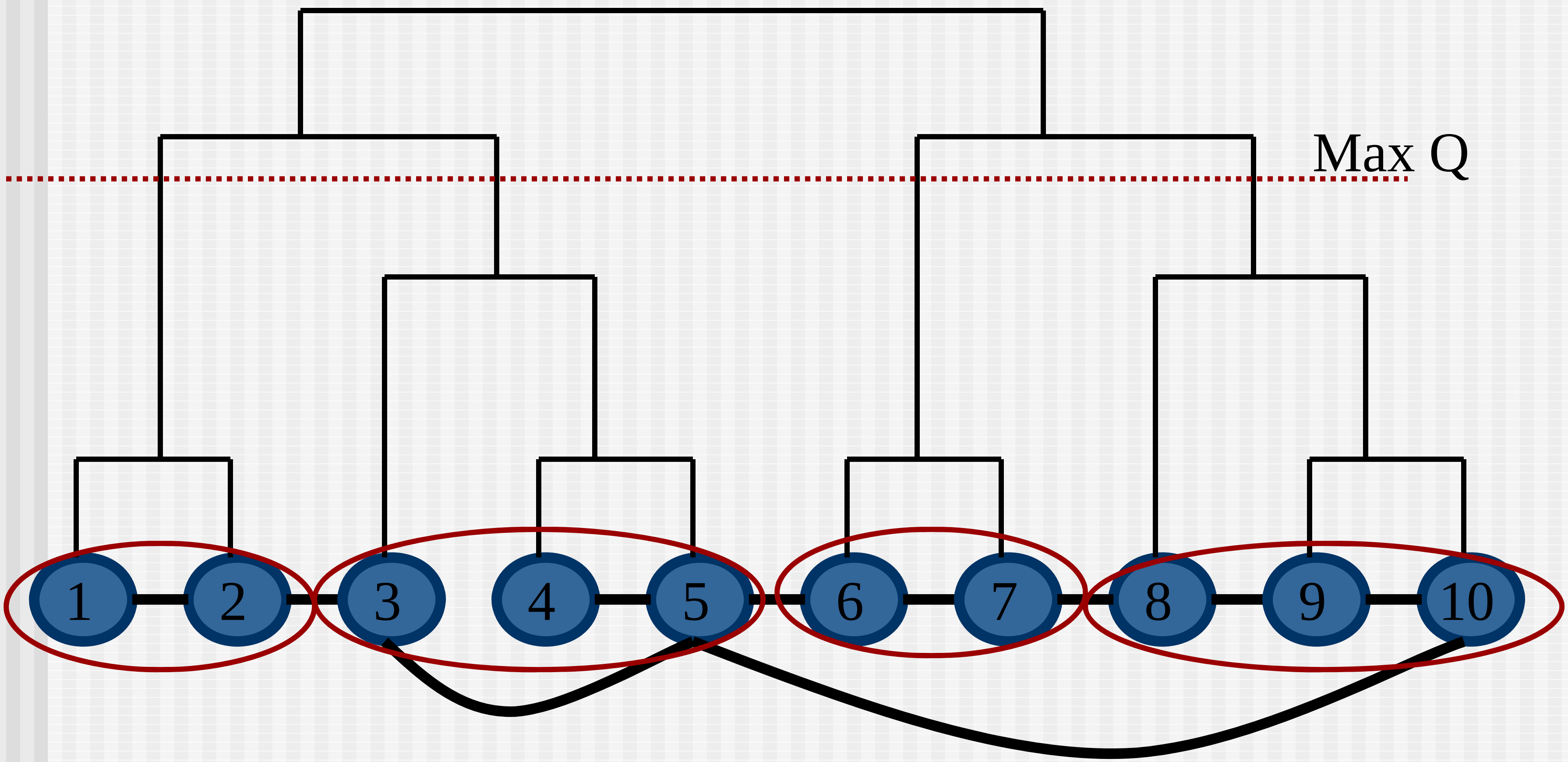
Agglomerative Clustering

- Optimization of Q is expensive
  - Exhaustive search of all possible communities is exponential to number of vertices
- Agglomerative Clustering (greedy)
  - Change in Q on joining community i and community j
    - $\Delta Q(i,j) = 2*(C(i,j)-a(i)*a(j))$
  - Join communities to maximize ΔQ

Agglomerative Clustering

- Algorithm-I
  - Initialize each vertex as a community
  - While (community > 1)
    - Find ΔQmax $O(|E|)$
    - Combine corresponding communities $O(|V|)$
      - C(i,:)=C(i,:)+C(j,:)
  - End
- Optimal community given by maximum Q
- Maximum Iterations $O(|V|)$
- Complexity: $O((|E|+|V|)*|V|)$

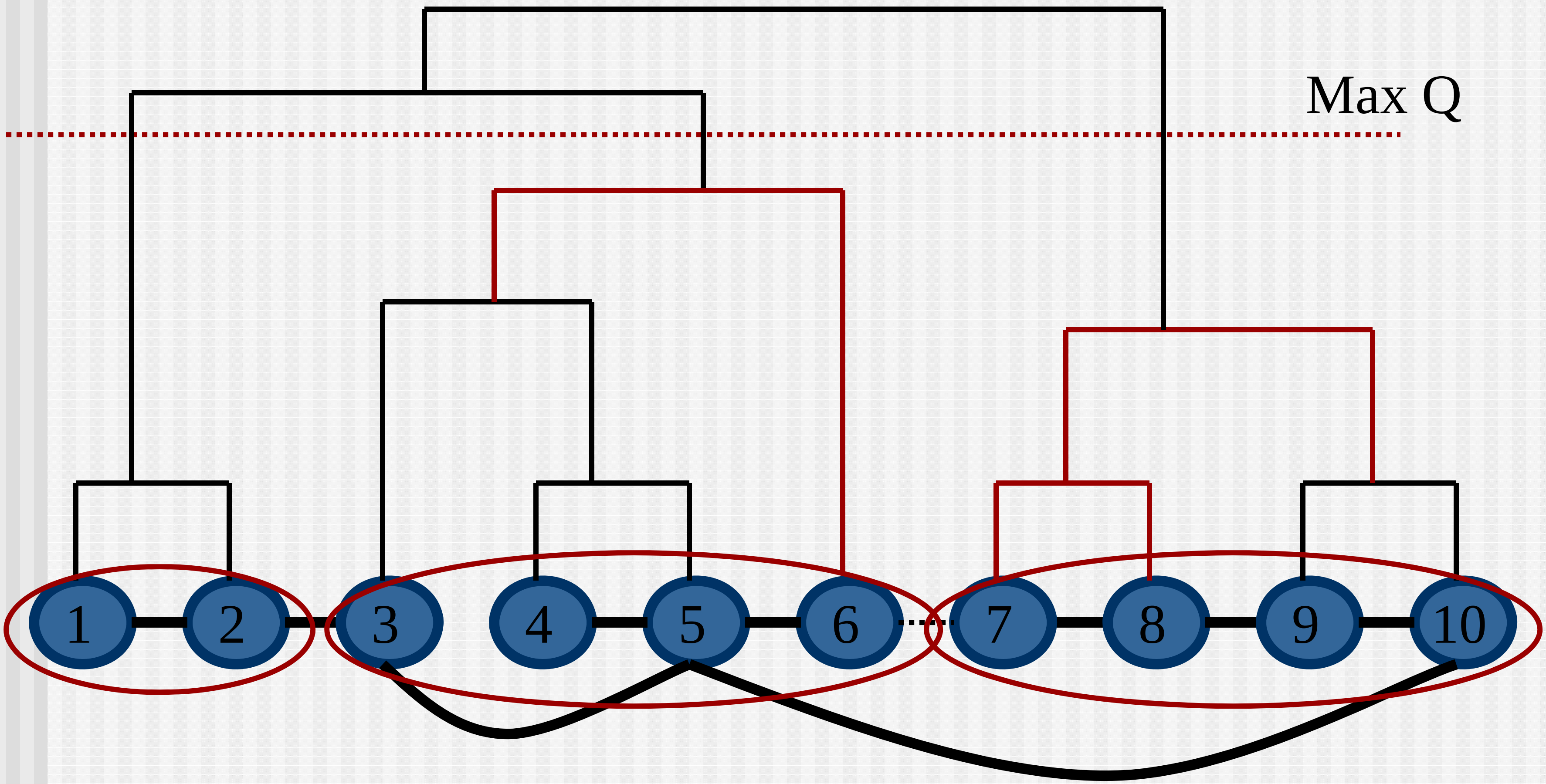
Agglomerative Clustering



Dynamic Networks

- Most networks are not static
  - Social interactions change, web pages added, etc.
- Most community detection algorithms recalculate the entire network for each change
- Goal: Incremental update of communities based on network perturbations
- Memory/Time Trade-Off:
  - Some information from the previous network

Communities in Perturbed Networks



Things to Observe

- Perturbation does not affect all communities
  - Possible to duplicate many combination operations
- Perturbation does not affect only communities that contained perturbed vertices
  - Must consider new combinations for communities beyond the perturbed vertices
- Instead of the final community structure consider the combination operations
  - Both require O(|V|) memory space

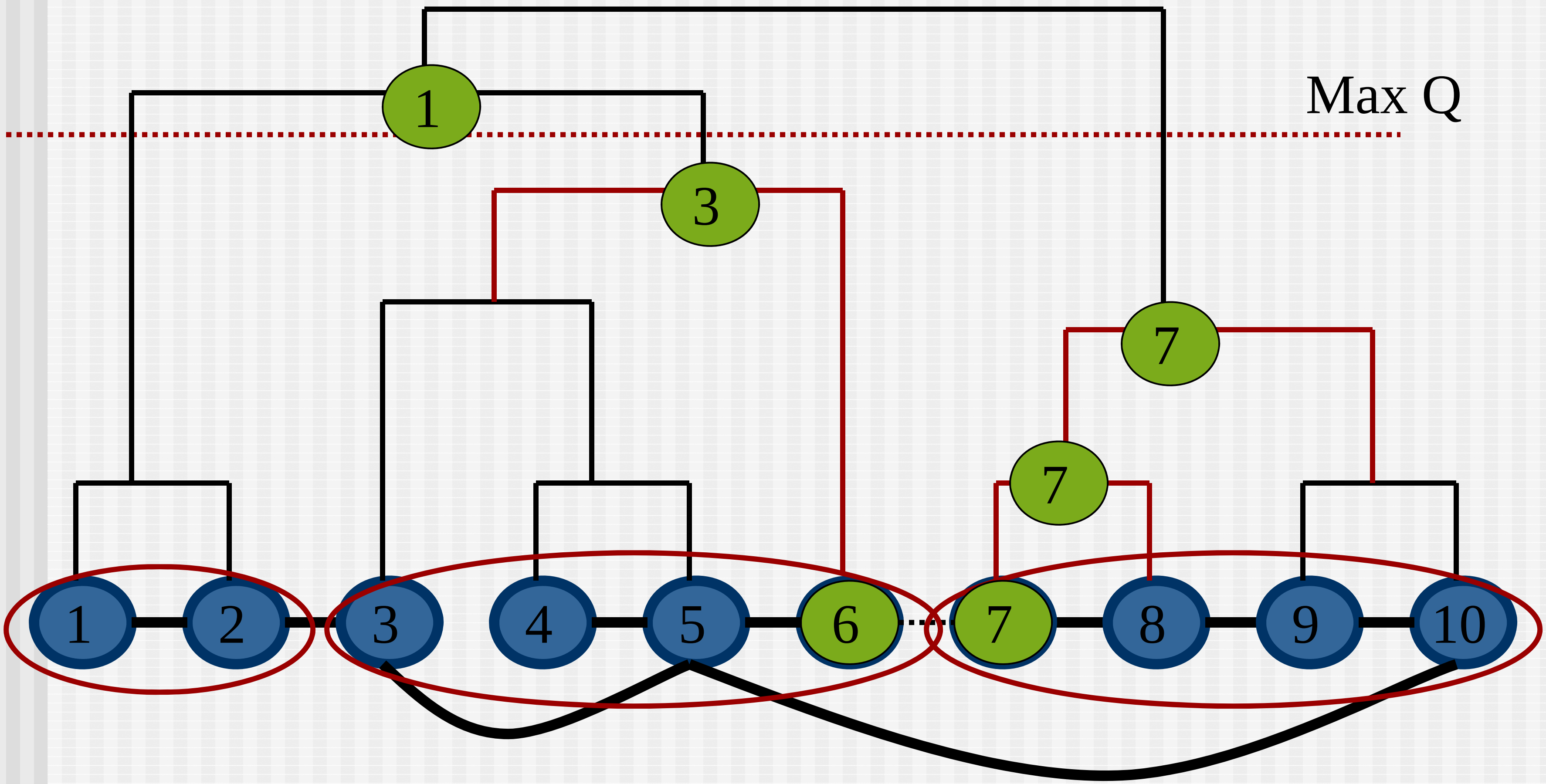
Perturbed Communities

- Perturbed Communities: composition is affected by network perturbations
- Set of perturbed communities evolve through the agglomeration process
- Need to find highest ΔQ only for perturbed communities
  - All other communities maintain almost the same ΔQ

Identifying Perturbed Communities

- Initial perturbed communities
  - Perturbed vertices
  - All vertices whose ΔQ values are changed
- Recall, $\Delta Q(i,j) = 2*(C(i,j)-a(i)*a(j))$
  - All neighbors of perturbed vertices are potential perturbed communities
  - In practice; only perturbed vertices in initial list works
- In subsequent agglomeration steps
  - Any communities that combined with perturbed communities are perturbed

Communities in Perturbed Networks



Agglomerative Clustering on Perturbed Networks

- Algorithm -II
  - Initialize each vertex as a community
  - Create list of perturbed communities
  - While (community > 1)
    - If previous step combines perturbed communities $O(|P|)$
      - Find ΔQmax $O(|E|)$
      - Add new community to perturbed list $O(1)$
    - End
    - Combine communities $O(|V|)$
  - End
  - Maximum Iterations O(|V|)
  - Complexity O(|V|*(|P|+|V|)) +O(|E|*|P|)
    - P=perturbed communities

Improving Algorithm-II

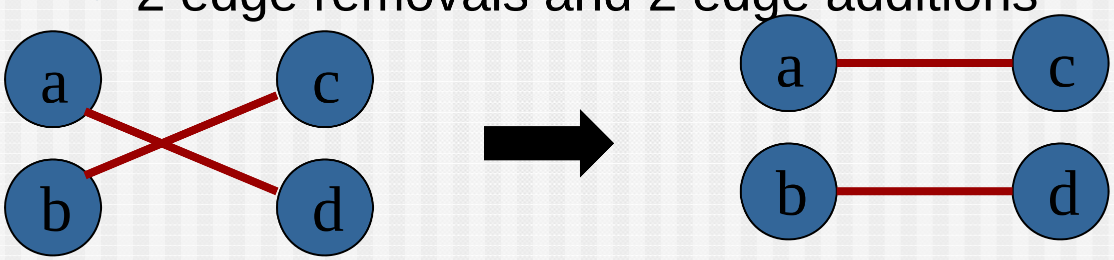
- Smaller |P| leads to faster update
  - Focus on limiting number of perturbed communities
  - Revert to Algorithm-I when |P| is large

- Need not recompute all combination steps
  - Store some updated C values for some combinations (Memory/Time Trade-Off)

Experimental Setup

- Synthetically generated graph
  - 100 vertices, 506 edges
  - Vertex degrees 1-11 (mean 5, mode 3)

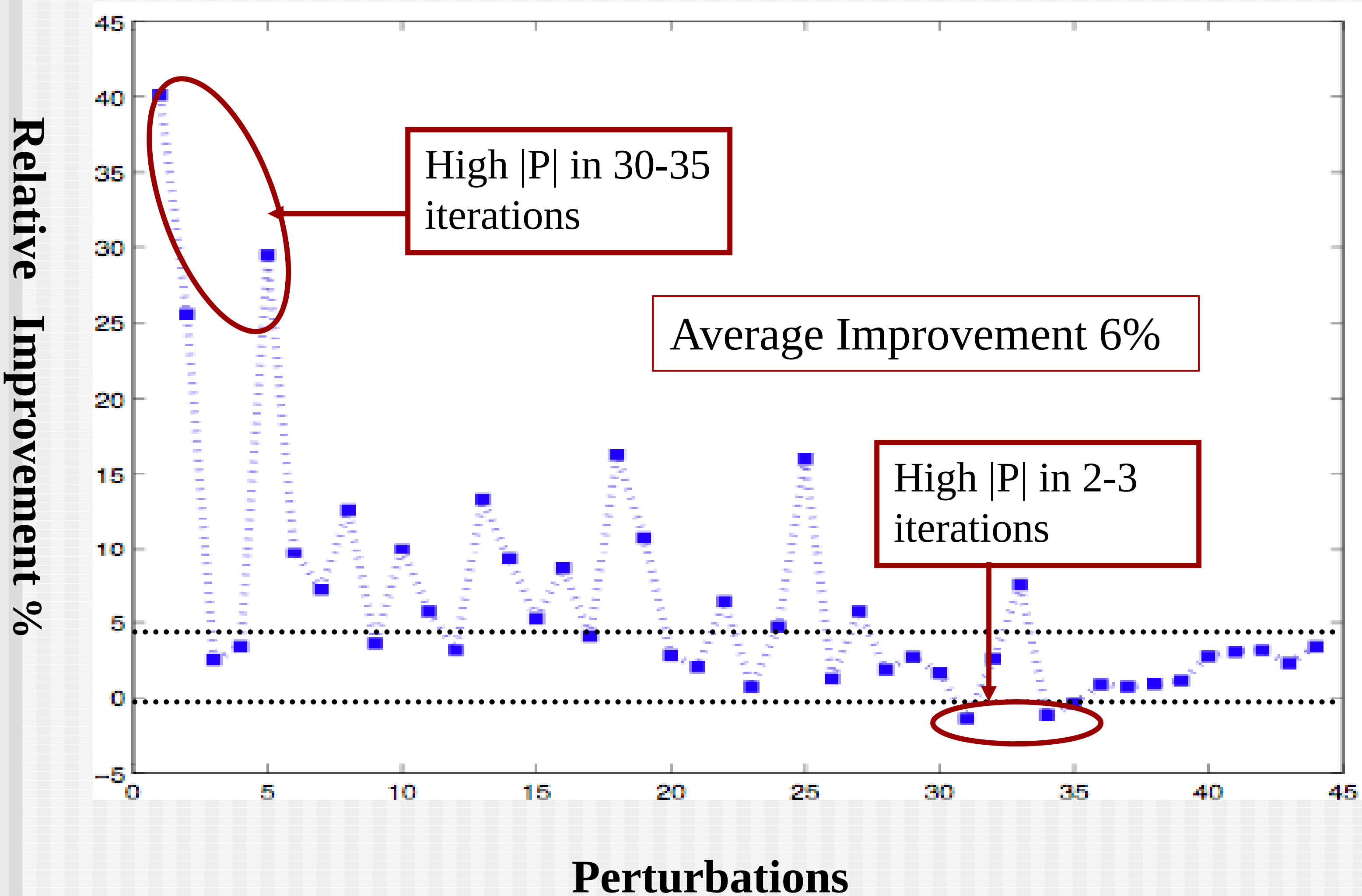
- Network Perturbation
  - Shuffling (preserves degree distribution)
  - 11 shuffles
    - 2 edge removals and 2 edge additions



Implementation Issues

- Selecting ΔQ_{max} is dependent on the floating point precision
- In the initial stages there can be many contenders for ΔQ_{max}----this affects the community structure
- Almost correct data---> Almost correct answer

Dynamic Update



Summary

- Key to efficient dynamic update is to identify vertices that are affected
- Hierarchical(local) algorithms help to isolate perturbed regions
- Memory/Time trade-off
  - previous step information vs recomputing
  - size of the perturbed subgraph