Internship at Snap

Satyaki Sikdar

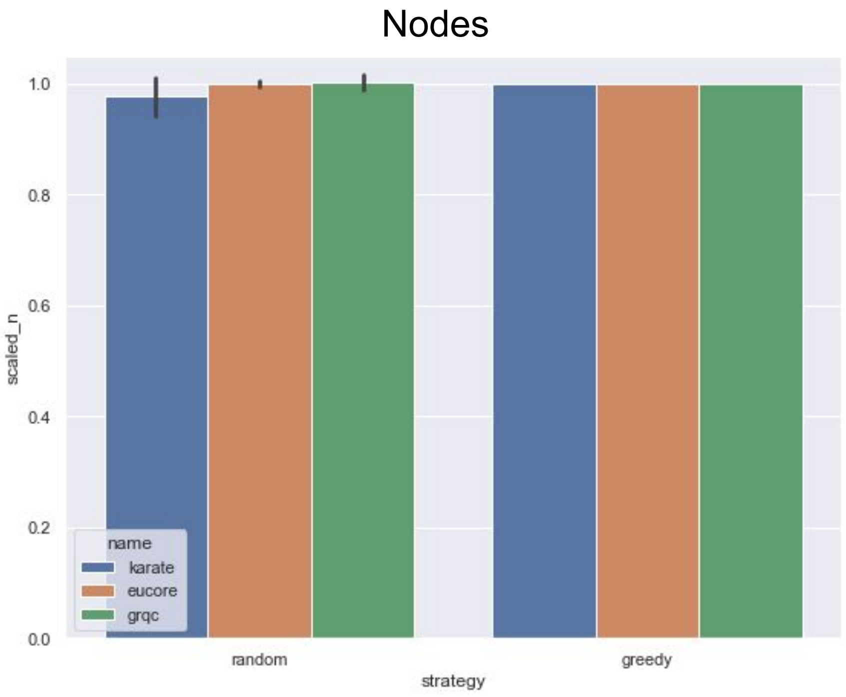
Main Ideas and Downstream Tasks

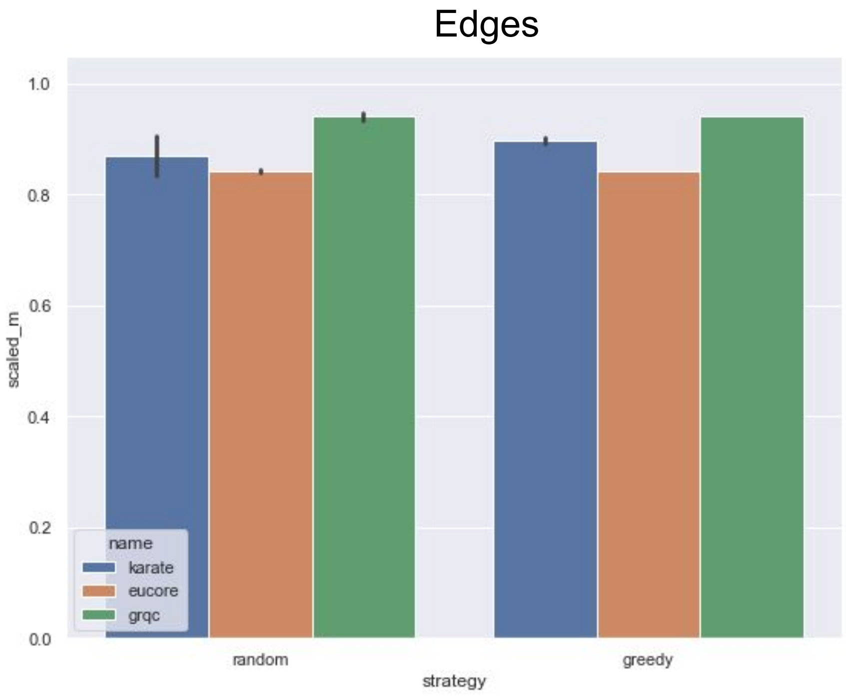
- **Faithful graph generation**
- Graph summarization combining SUBDUE and KGist
- Privacy preservation
- Anomaly detection - KGist does that to some extent - looks at the regions with high error

Updates (05/28)

- Waiting on data access
- Naive random generator and greedy generator works
- Truly isomorphic implementation in progress
- Some results on graph generation quality

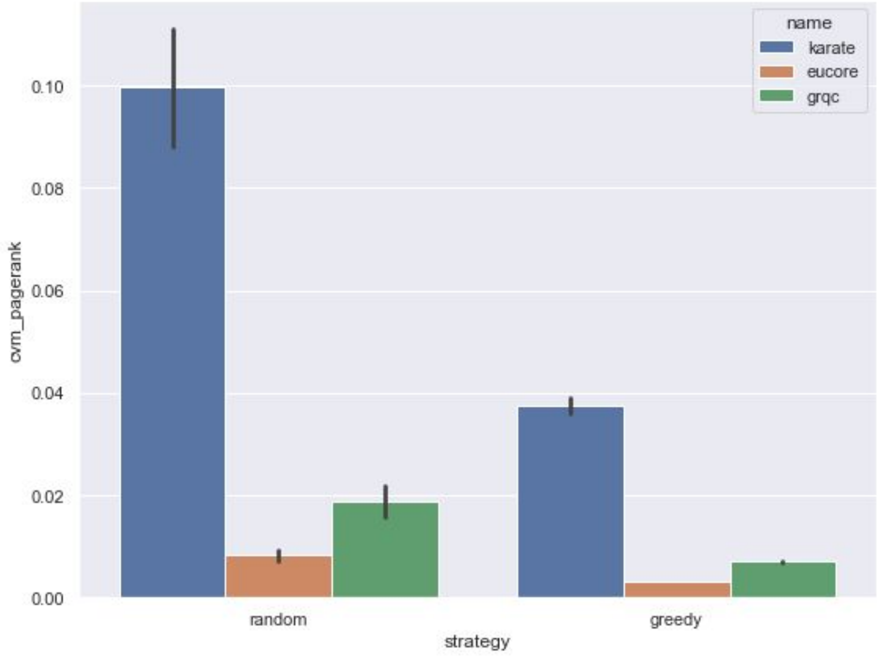
Number of nodes and edges for different strategies



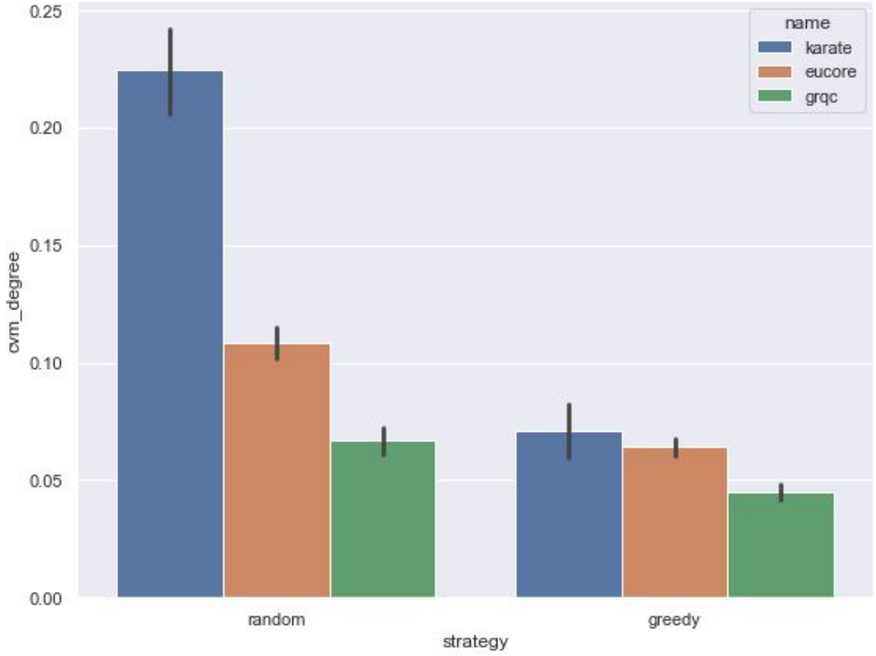


CVM pagerank and degree for different strategies

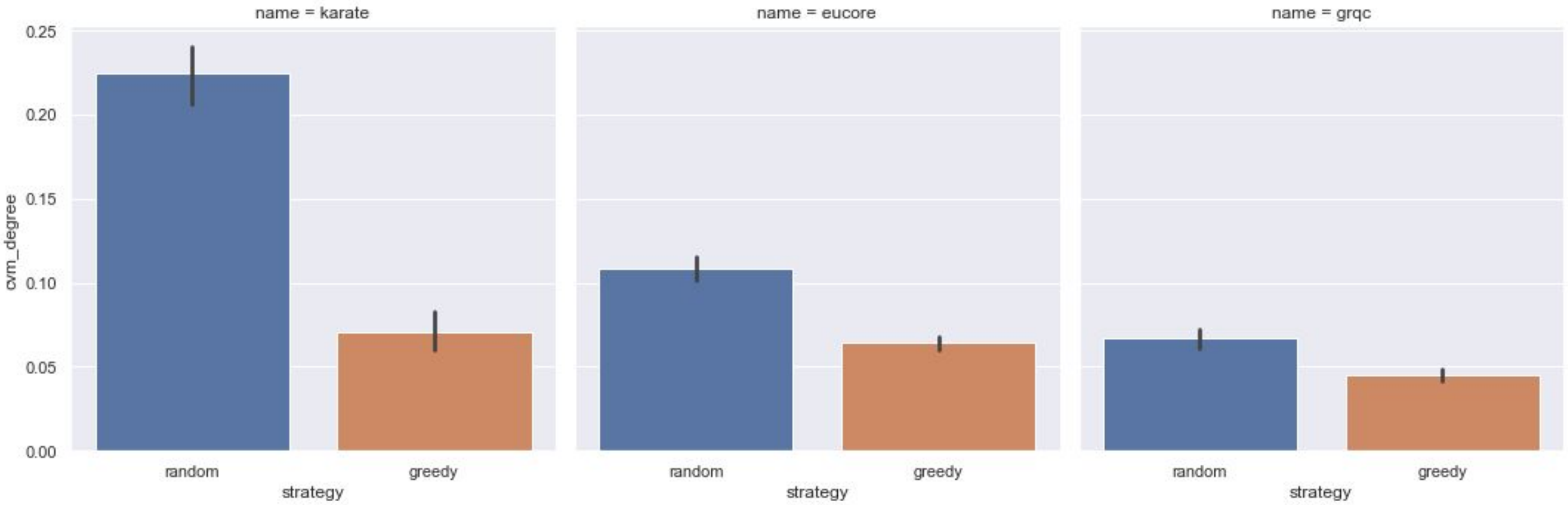
Pagerank



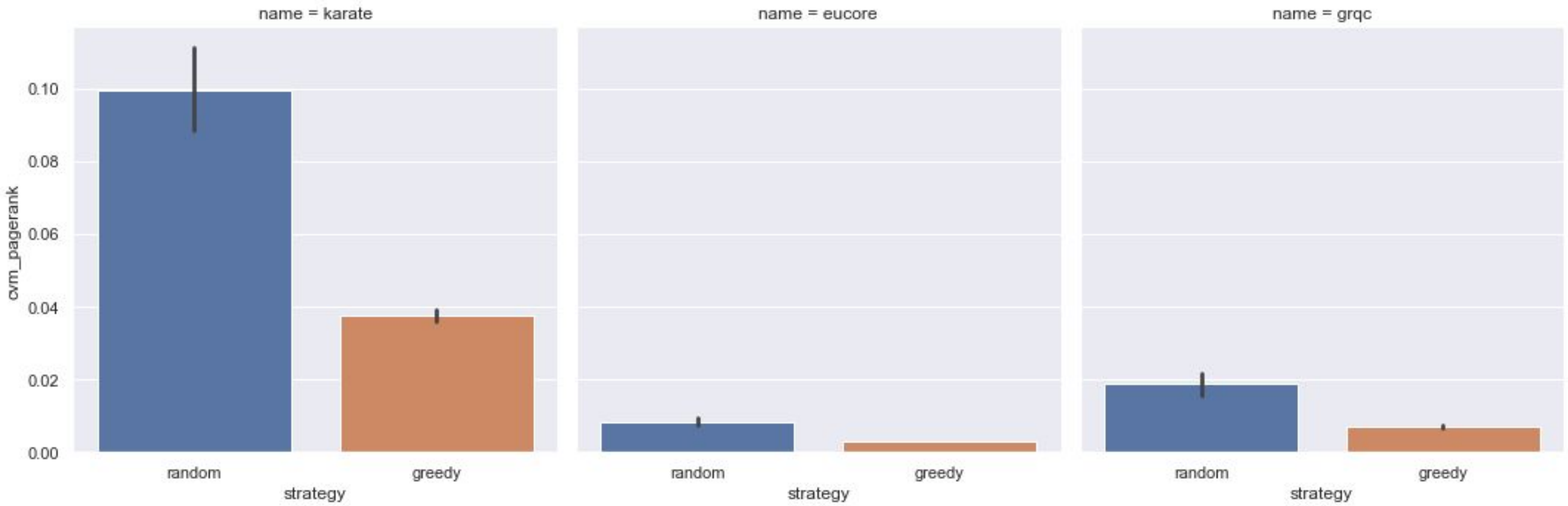
Degree



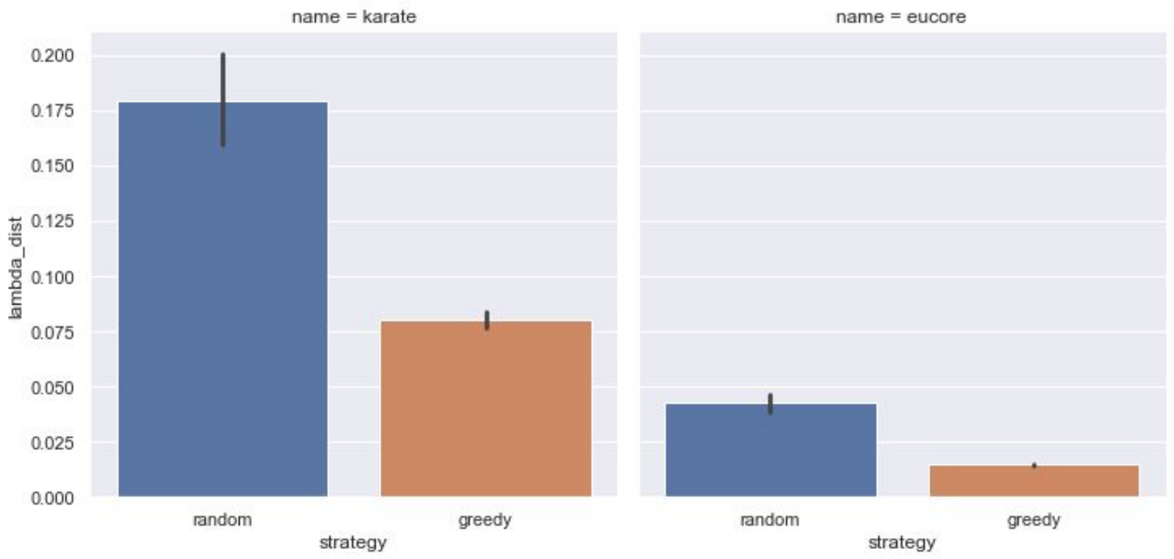
CVM degree for different graphs



CVM pagerank for different graphs



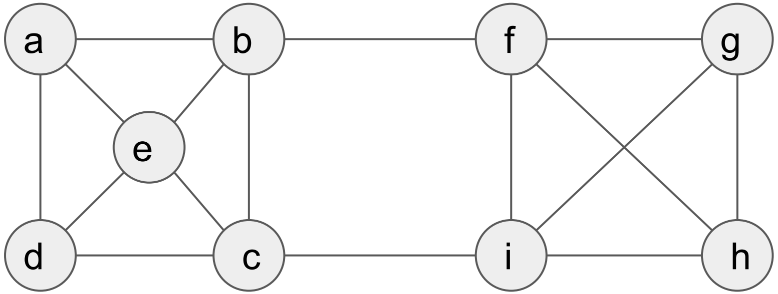
Lambda Dist for different graphs



Updates 06/04

- Re-wrote large sections of code to make it more accessible and easy to use
- Still working on fractional correspondence - full and zero works
- GraphGen - generating labeled graphs with LSTMs - DFS codes - similar to gSpan
- VRGs should be able to do that too.
- If we have a series of graphs, we can look to find grammar rules covering similar sets of nodes => mining structural patterns

Example VRG

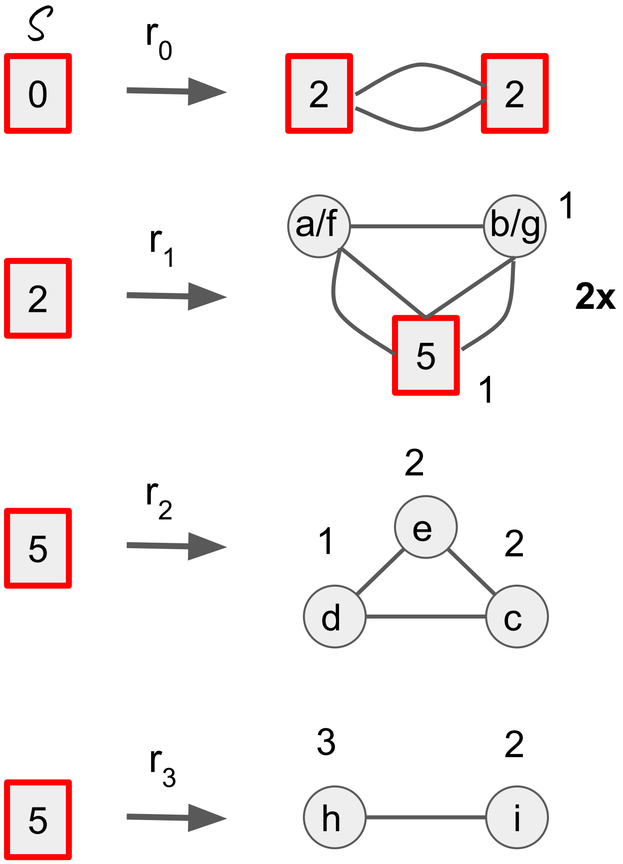


Input Graph

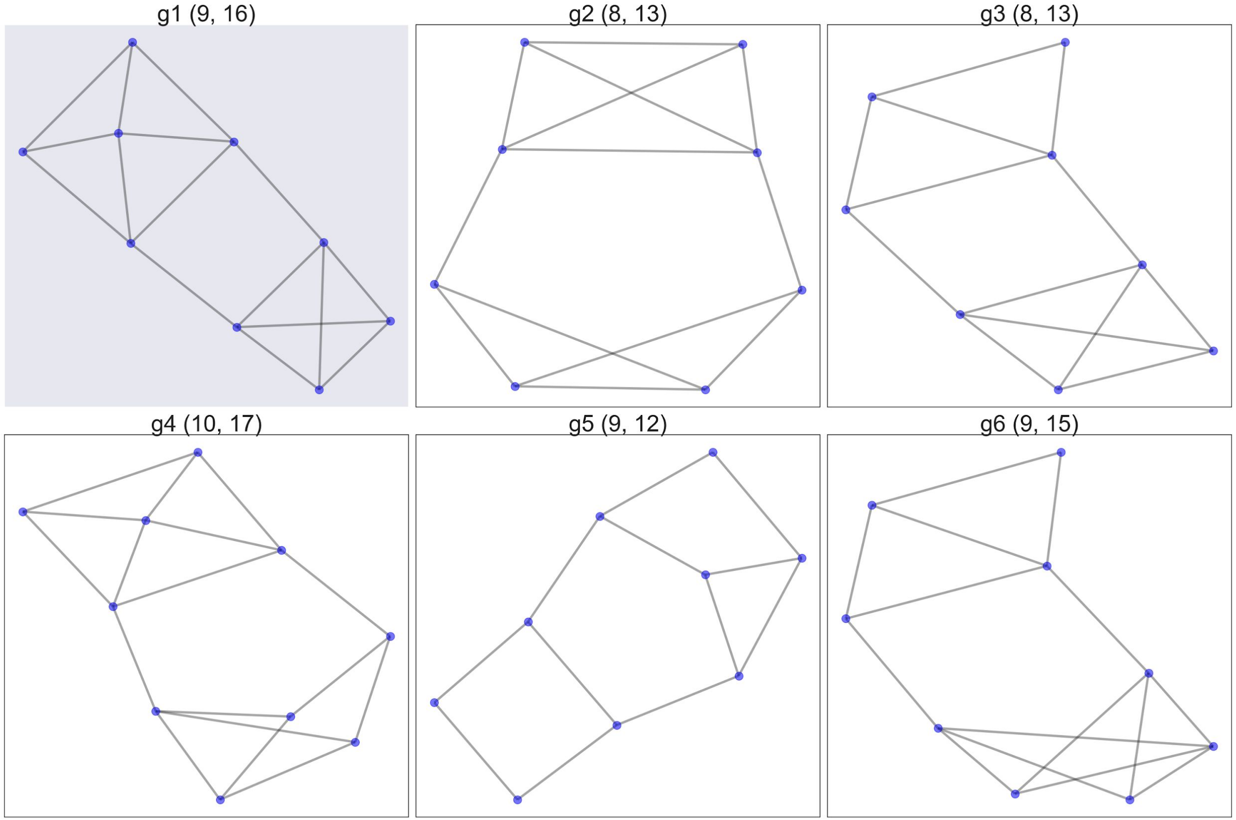
Order of discovery:

r_2 , r_1 , r_3 , r_1 , r_0

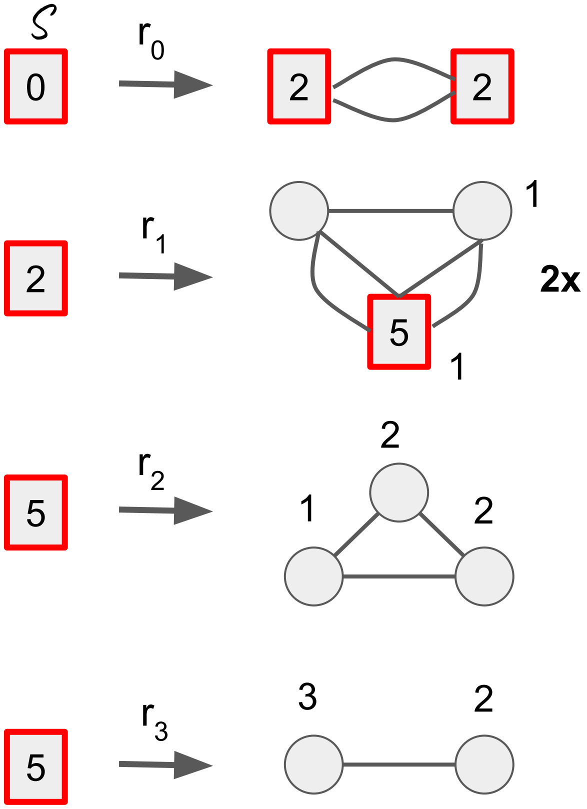
Grammar Rules



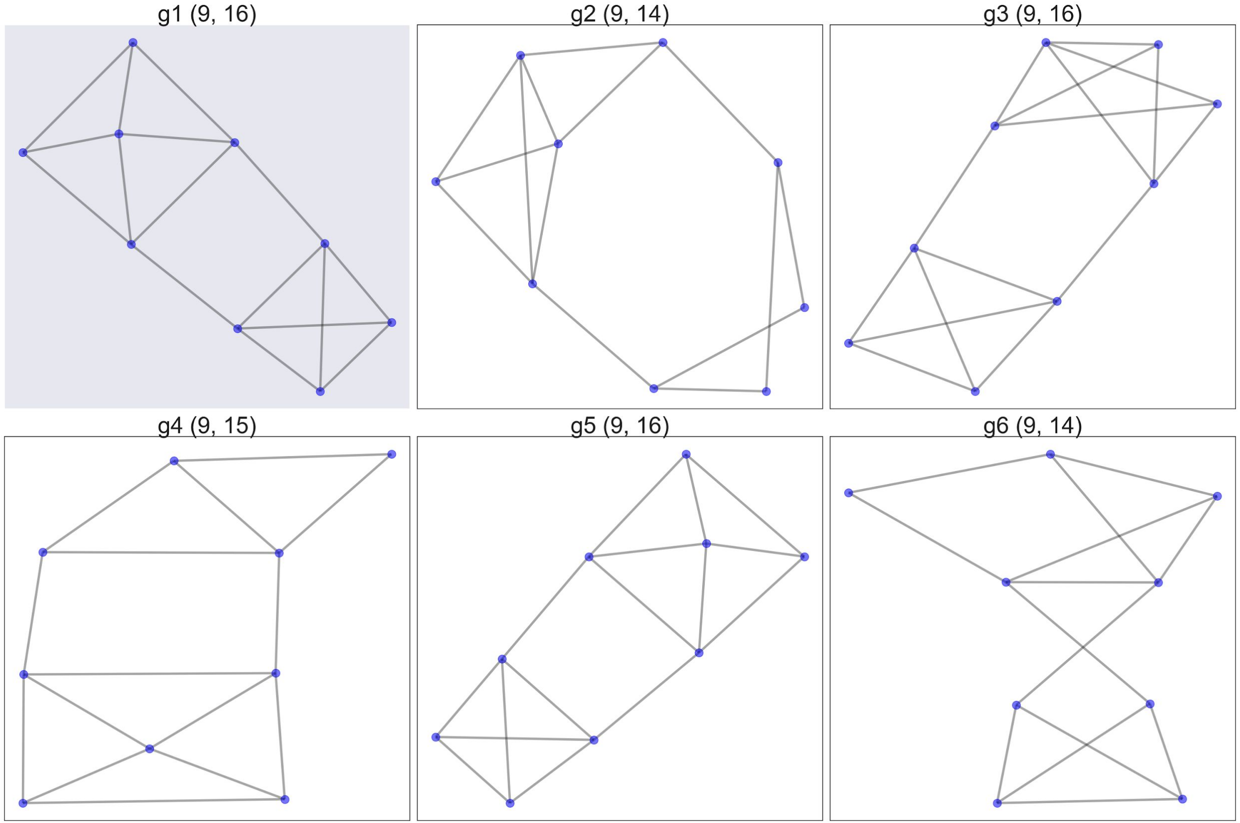
Generated Graphs - Random



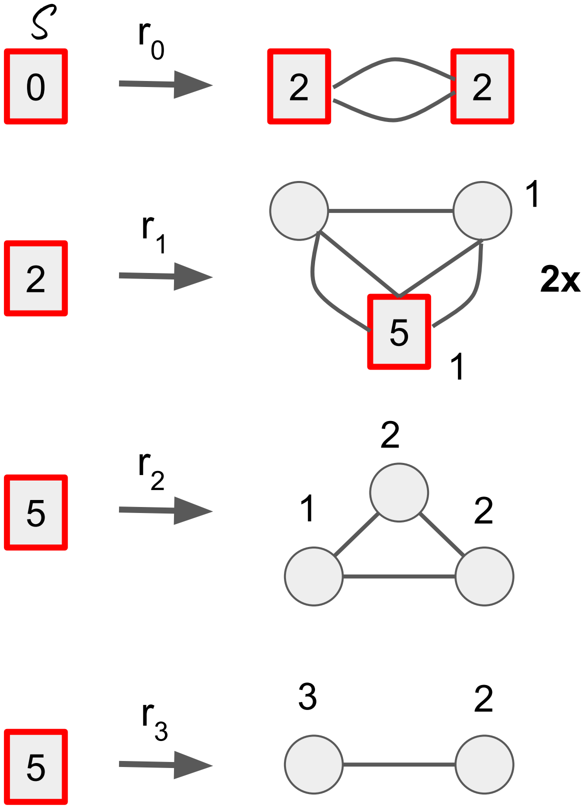
Grammar Rules



Generated Graphs - Greedy



Grammar Rules



Some observations - Extraction

- Each rule encodes a region of the graph
- Each node appears in exactly one rule
- We have the order of discovery of rules
- Apply them in reverse order, and you guarantee node coverage
- For isomorphism, we need to store local boundary information for each rule

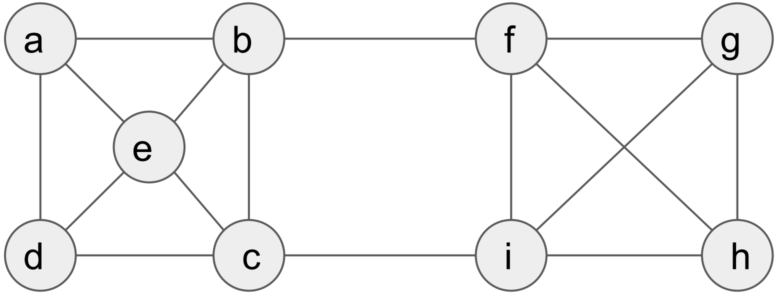
Some observations - Generation

- The order of rule firings determine the final graph
- Not every rule can be fired at a given time
  - Corresponding non-terminal should exist for a rule to fire
- Terminal nodes, once introduced, remains in the graph
- Edges connected to non-terminals do not exist in the final graph
- In the final graph,
  - Probability of a given node - directly tied to the probability of the rule firing
  - Probability of a given edge - slightly more complicated
    - If it exists in a rule, then same as node
    - If it isn’t then it depends on how edges are re-wired

TODOs

- Work on partial correspondence
- Formalize the probability of node / existence
- Intern presentation?

Example VRG

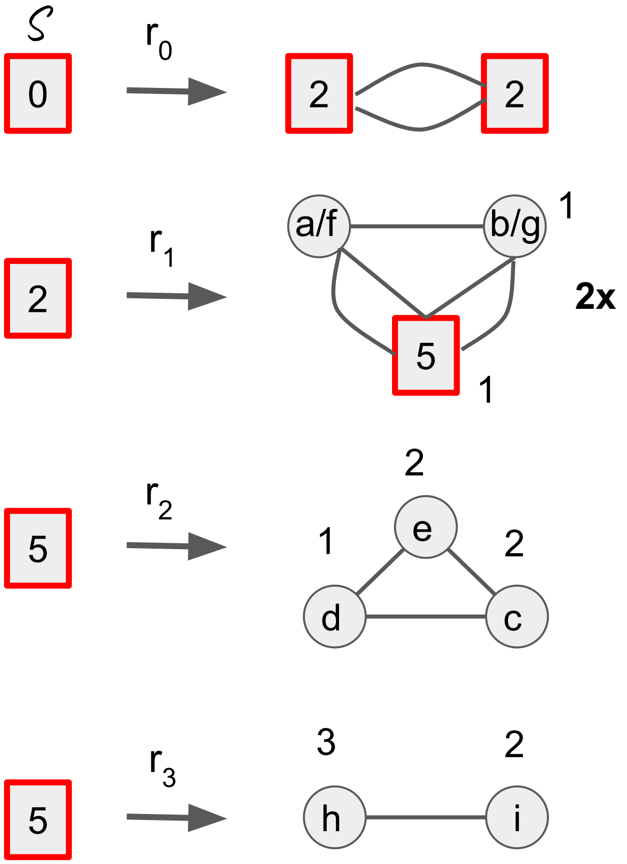


Input Graph

Order of discovery:

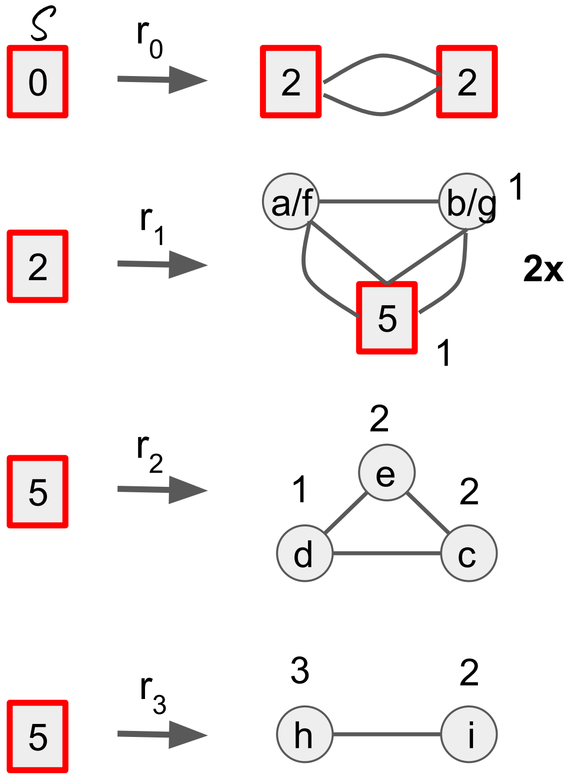
r_2 , r_1 , r_3 , r_1 , r_0

Grammar Rules



Node distribution in Rules

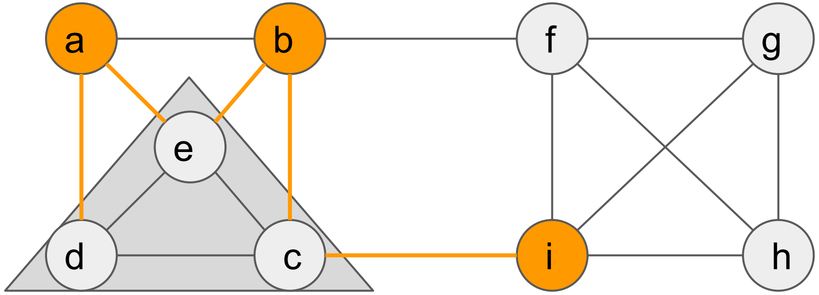
Grammar Rules



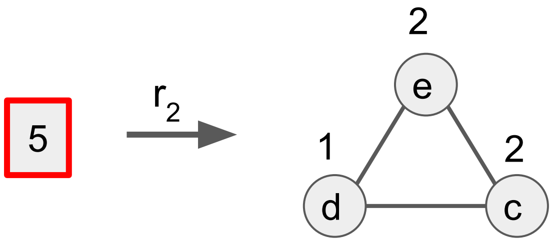
Node	Rule ID
a	r_{11}
b	r_{11}
c	r_2
d	r_2
e	r_2
f	r_{12}
g	r_{12}
h	r_3
i	r_3

Rule /node	r1	r2	r3
a	1		
b	1		
c		1	
d		1	
e		1	
f	1		
g	1		
h			1
i			1

Example VRG Rule Extraction



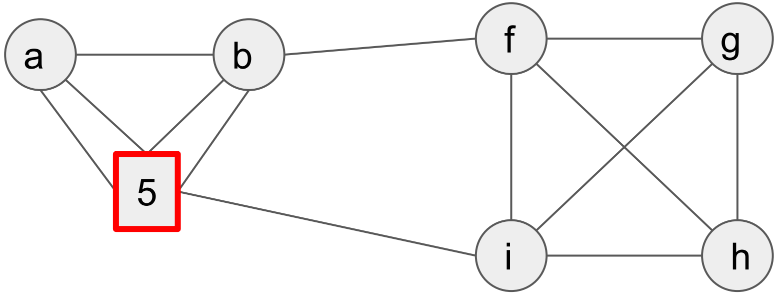
Current Graph



Boundary nodes: {a, b, i}

Boundary edges: {(e, a), (e, b), (d, a), (c, b), (c, i)}

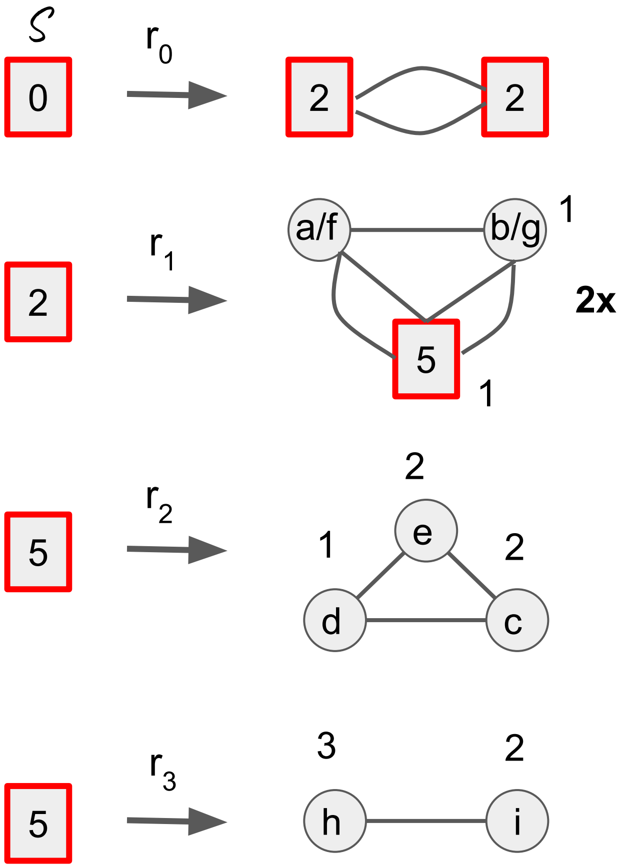
Extracted Rule



Updated Graph

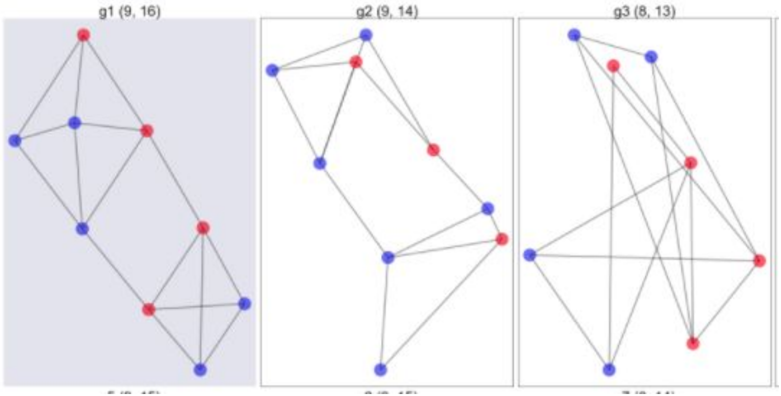
Notes regarding node and edge preservation

- Greedy generation wrt node preservation and correspondence
  - 0%, 50%, …., 100%
  - Or a specific set of nodes
- For guaranteeing a specific set of nodes
  - Each node belongs in exactly one rule
  - Firing that rule guarantees the presence of that node in the generated graph
  - If a rule *covers* a (sub)set of the nodes, applying that rule *at the right time* guarantees node preservation - eg rule r_2
- Isomorphism can be achieved
  - Store boundary info for each rule
  - Apply rules in reverse order - guarantees that boundary nodes exist when a rule is applied

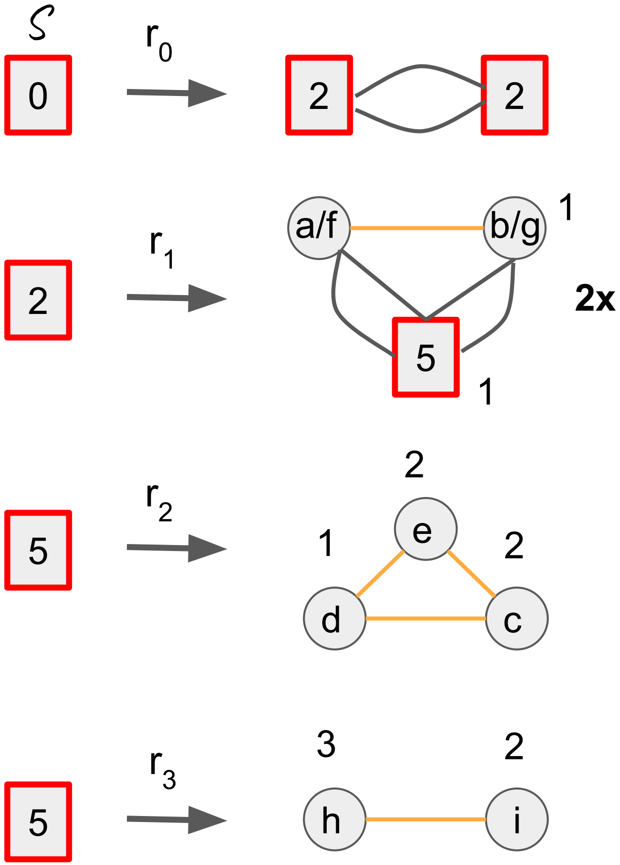


Notes regarding node and edge preservation

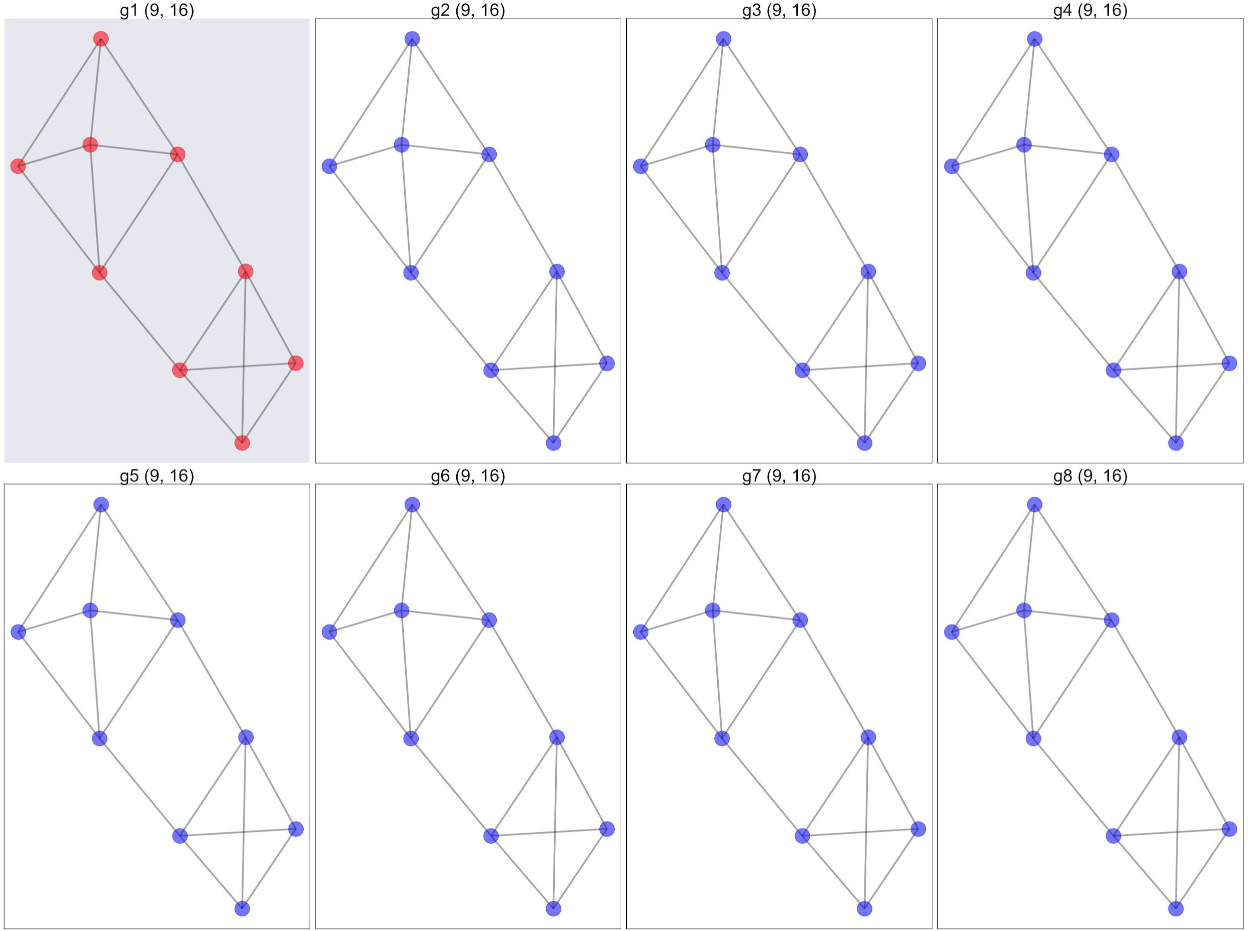
- What if nodes are not contained in the rules?
  - {a, b, f, i}? Sometimes you get lucky, sometimes not so much



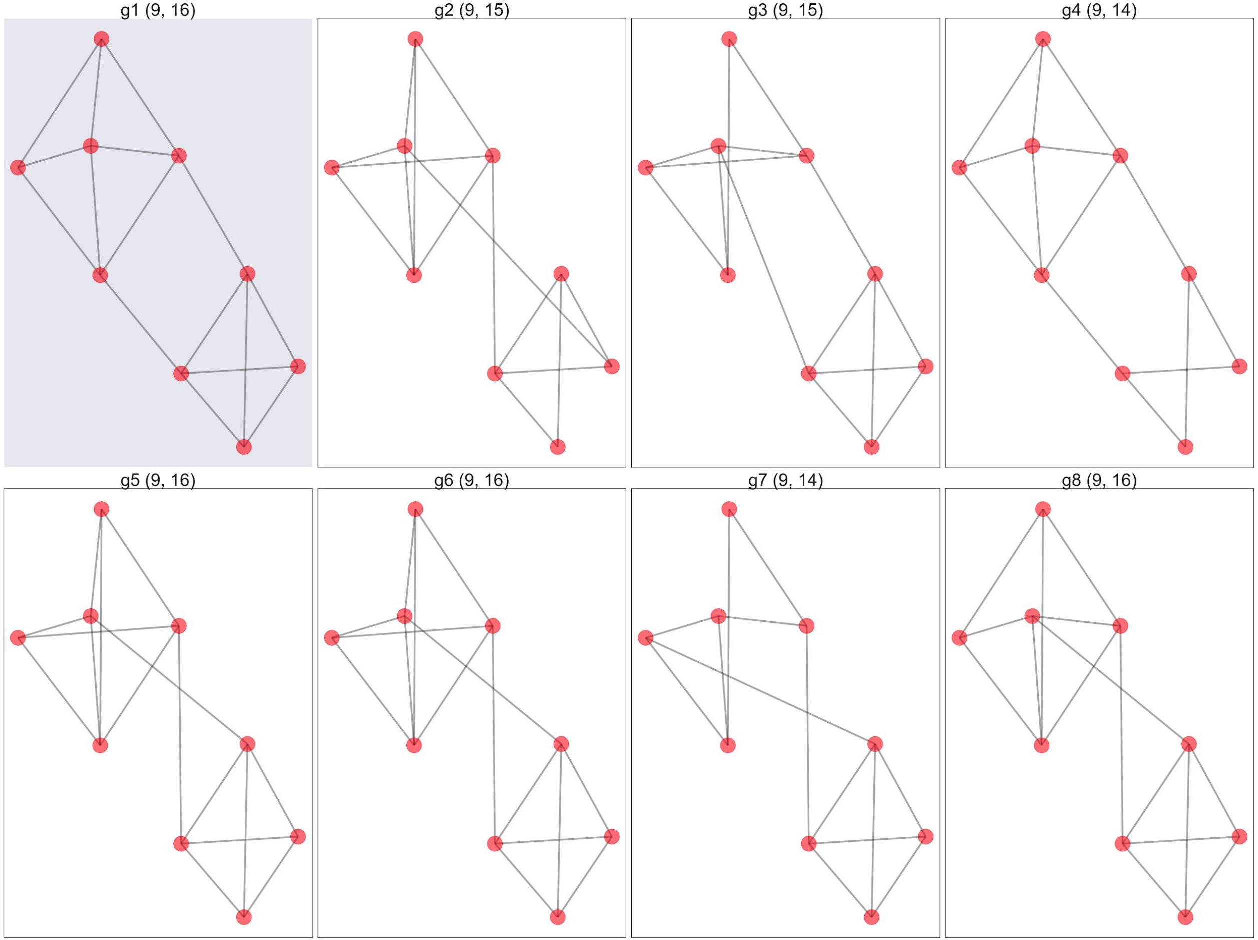
- Extending that idea to edges
- Subgraphs consisting of only *terminal* nodes in a rule are preserved once fired -- rule r_2 and r_3



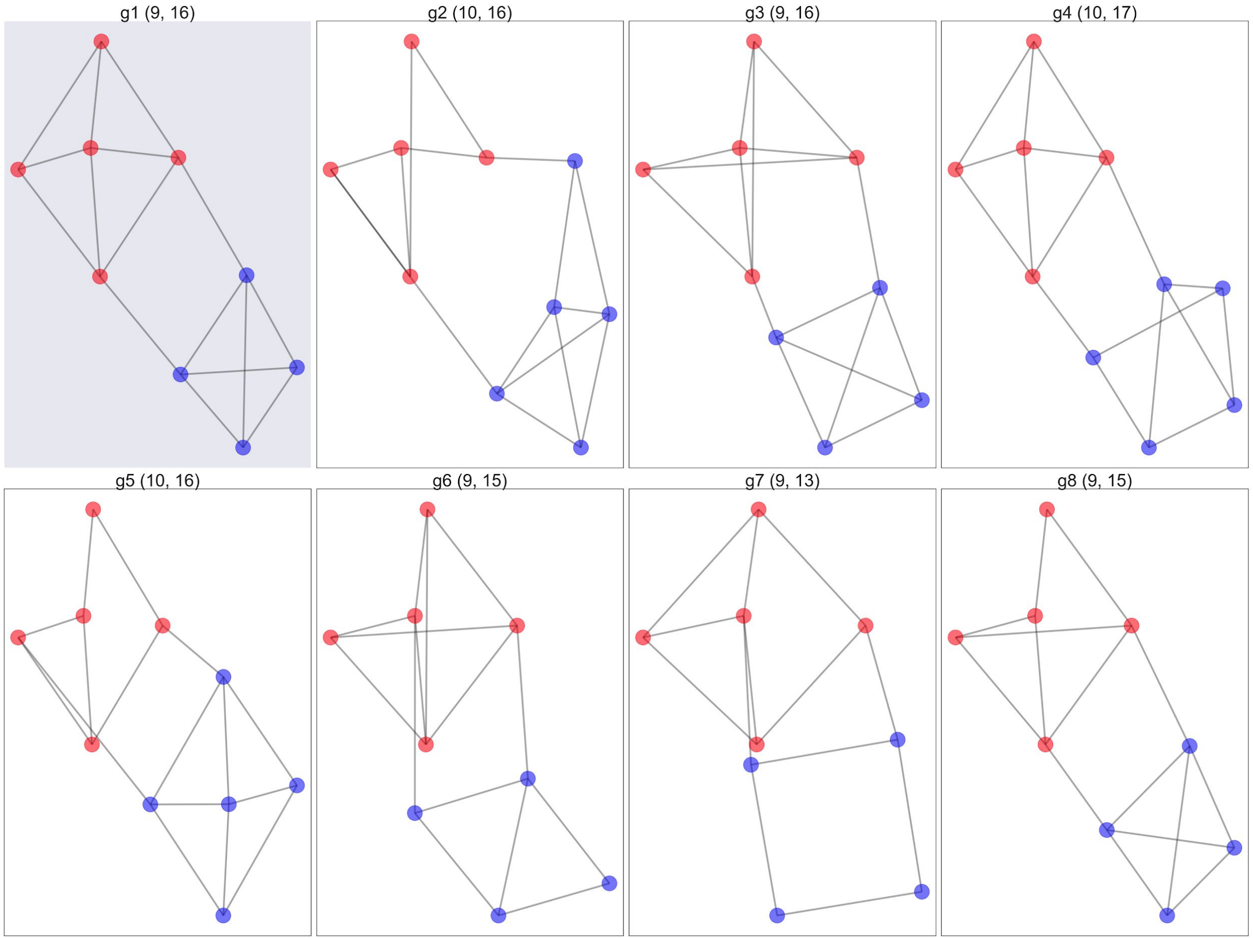
Guaranteeing Isomorphism



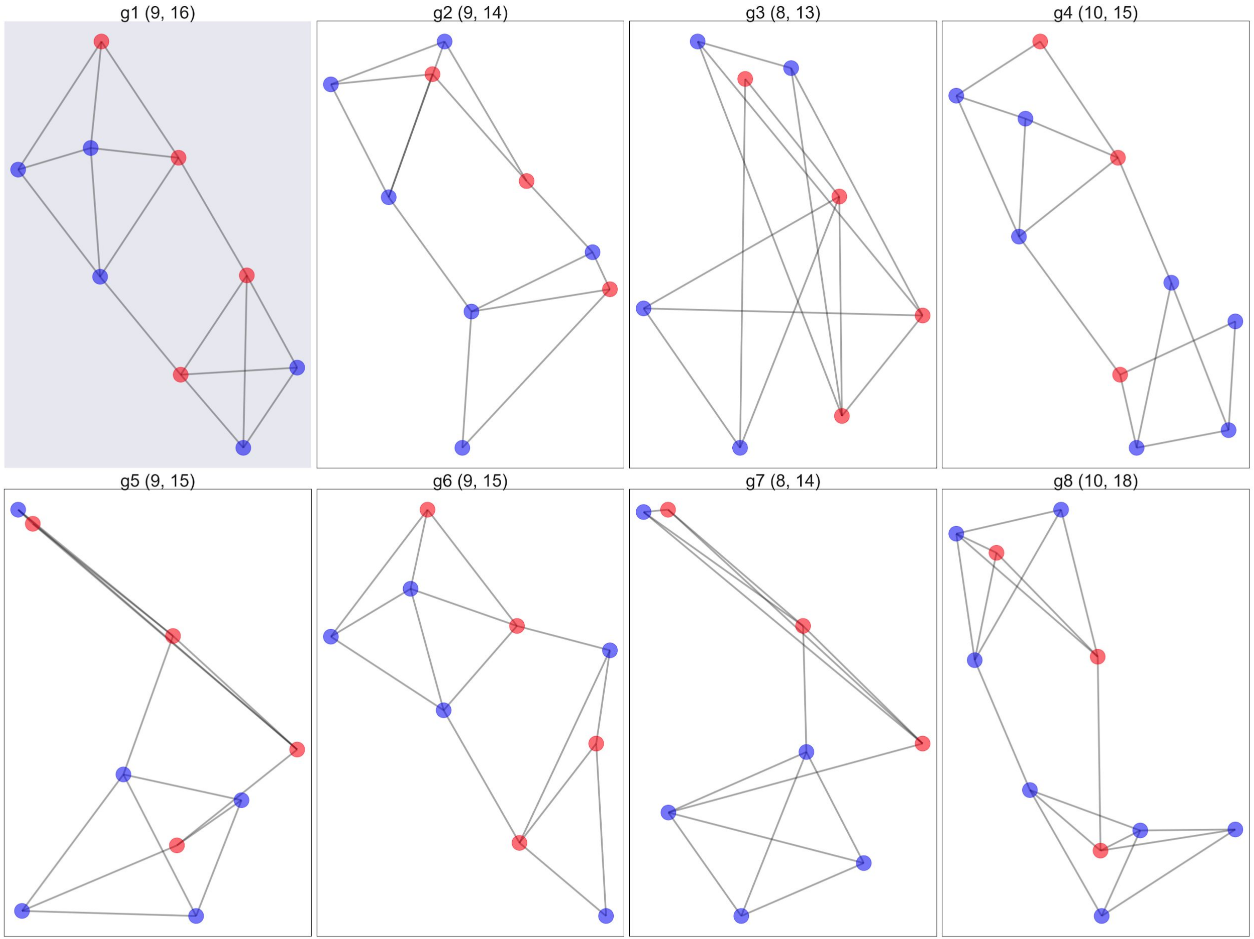
100% node correspondence - retain all nodes



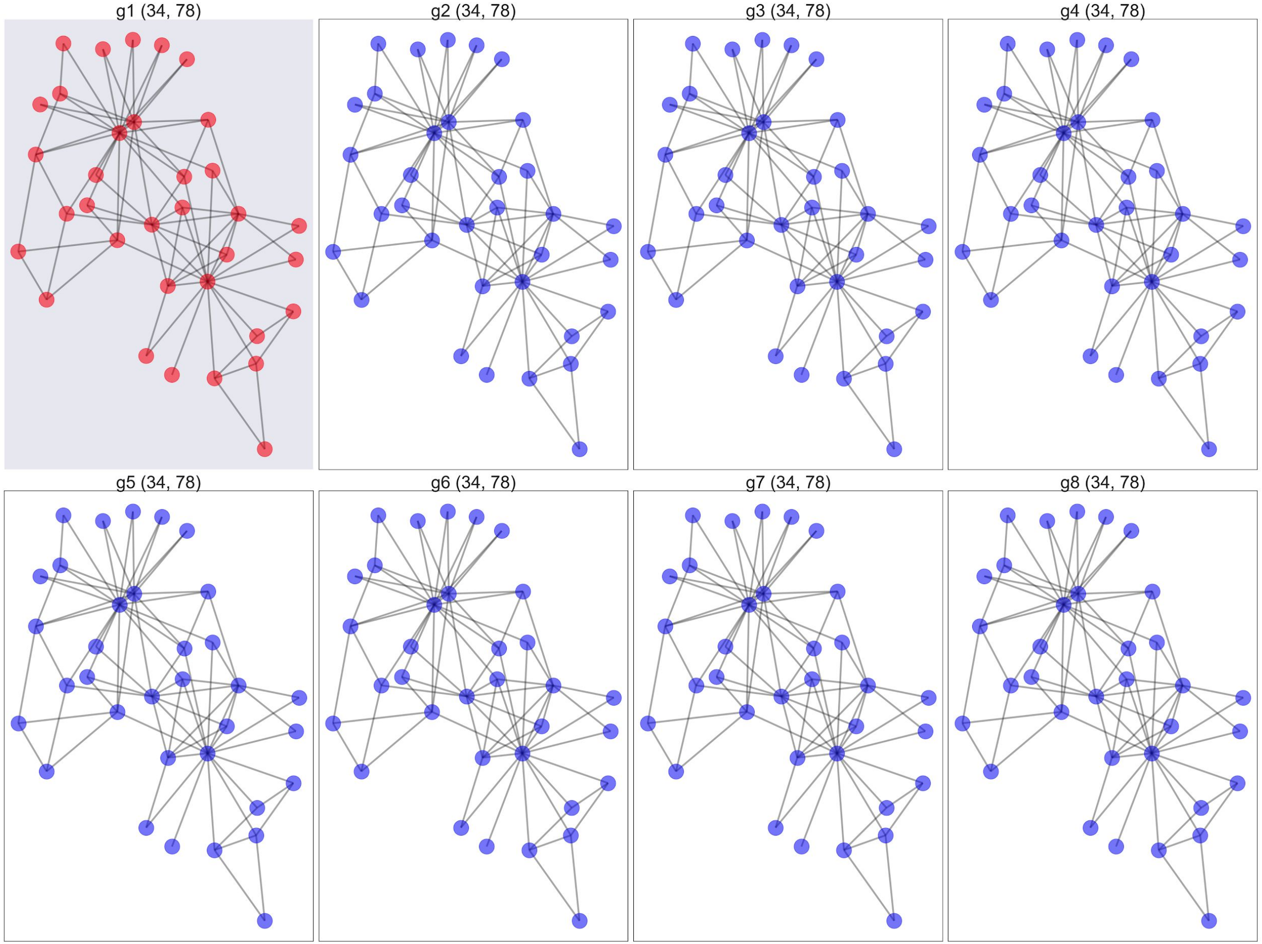
50% node correspondence - retain only red nodes



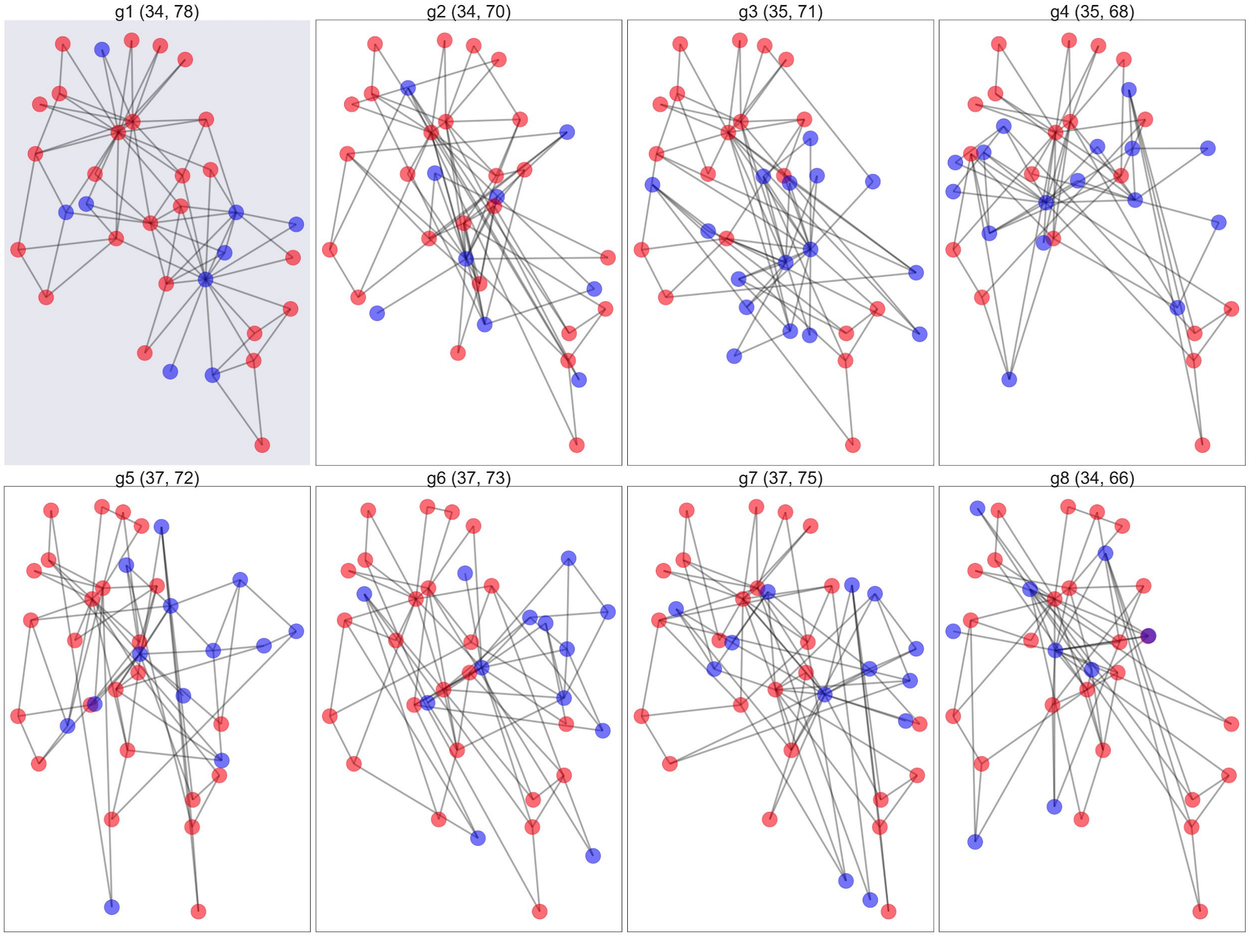
50% node correspondence - retain only red nodes



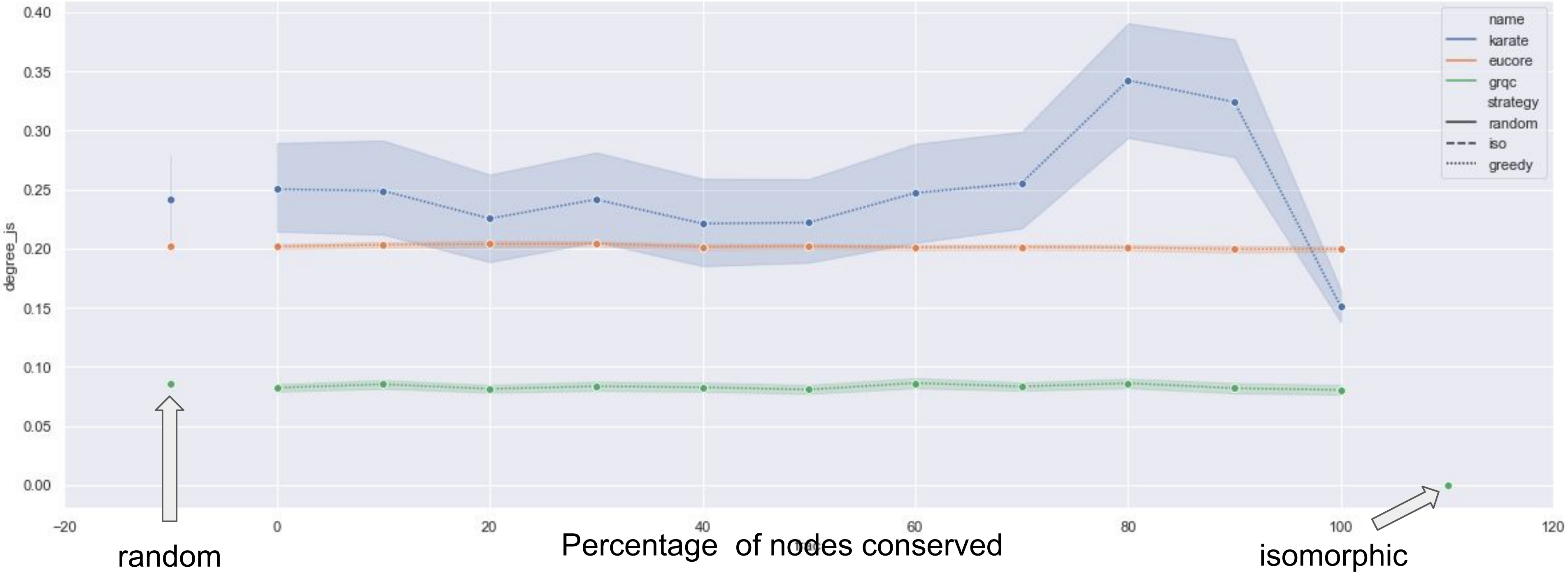
Guaranteeing Isomorphism - Karate Club



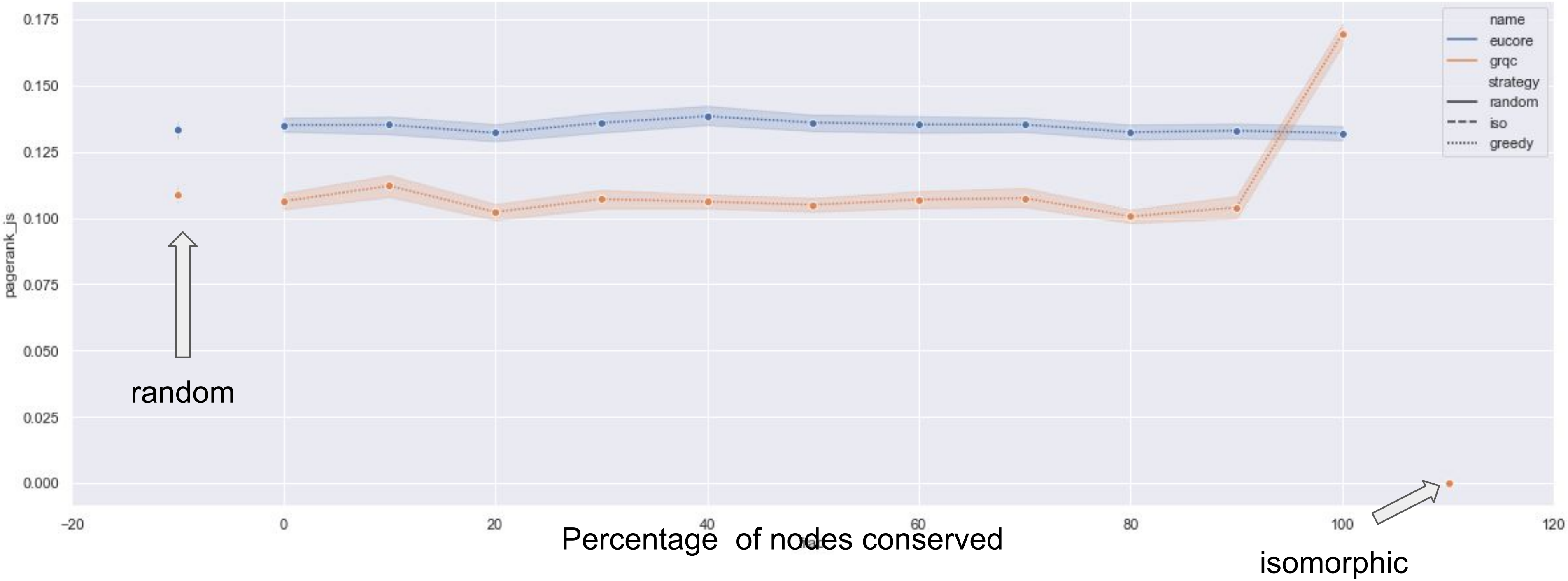
Preserve 75% of the nodes - Karate Club



JS Divergence Degree for 3 graphs (lower is better)



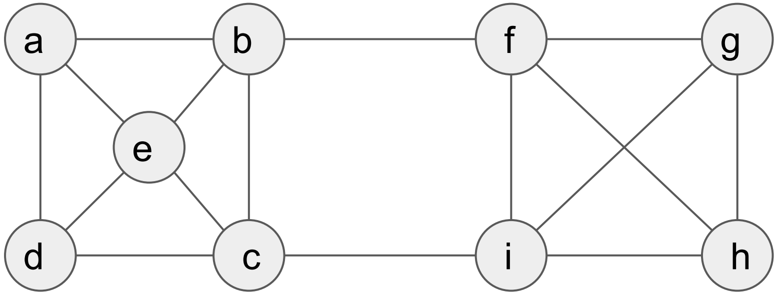
JS Divergence PageRank for 2 graphs (lower is better)



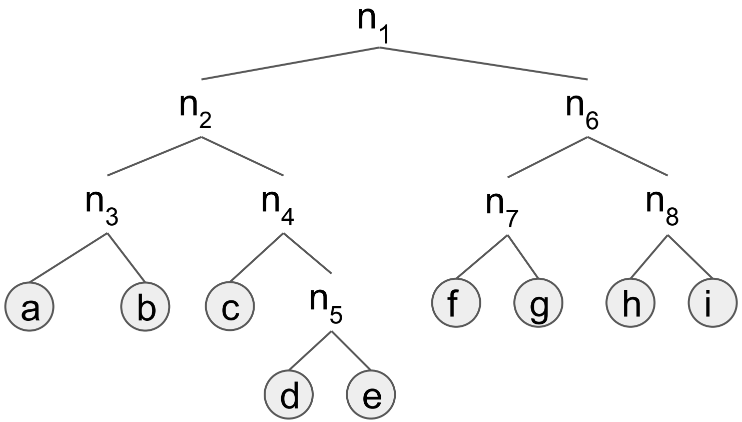
TODOs

- Stress-test the framework - offload the work to a server on Google Cloud
- Analyze the generated graphs more thoroughly - NetLSD / graphlet counts
- Start analyzing the rules and compare with KGist rules
- Tradeoffs - why would we want to do this?
- Downstream applications - classification tasks
- Provide invariants and guarantees about certain nodes and edges and conditional guarantees on edges
- Intern presentation? July 9

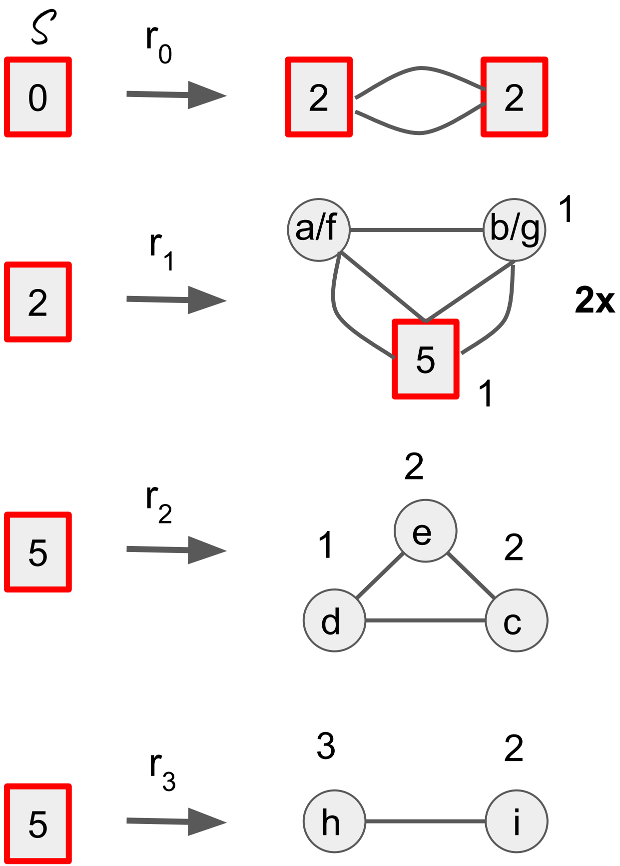
Where did the rules come from?



Input Graph

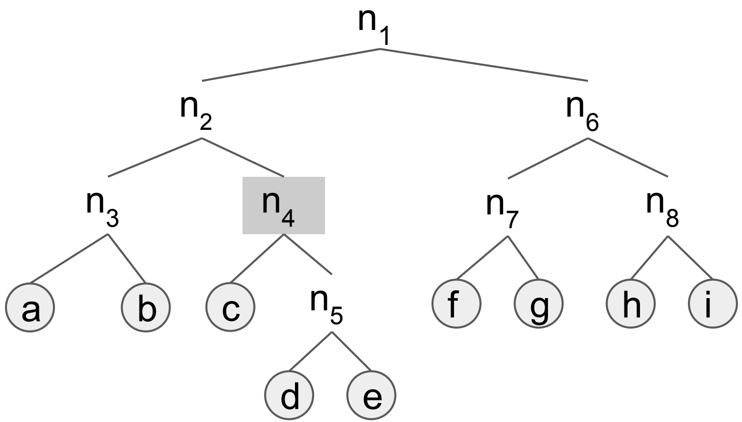


A dendrogram

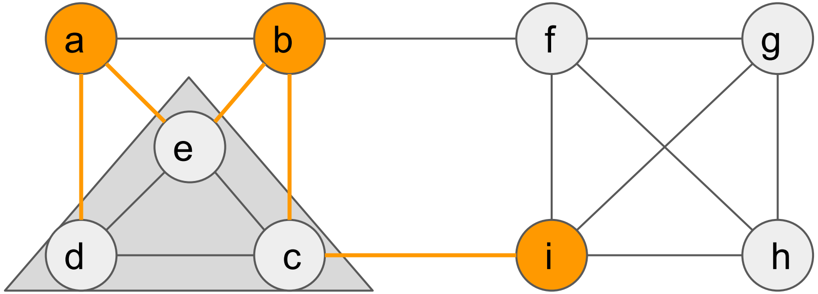


Grammar Rules

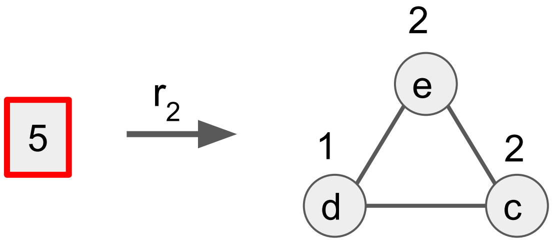
Example VRG Rule Extraction



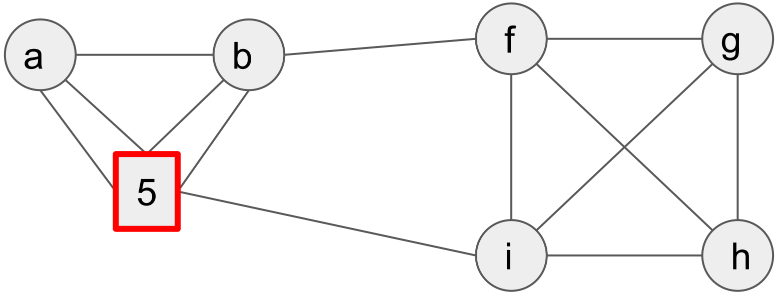
Current Dendrogram



Current Graph



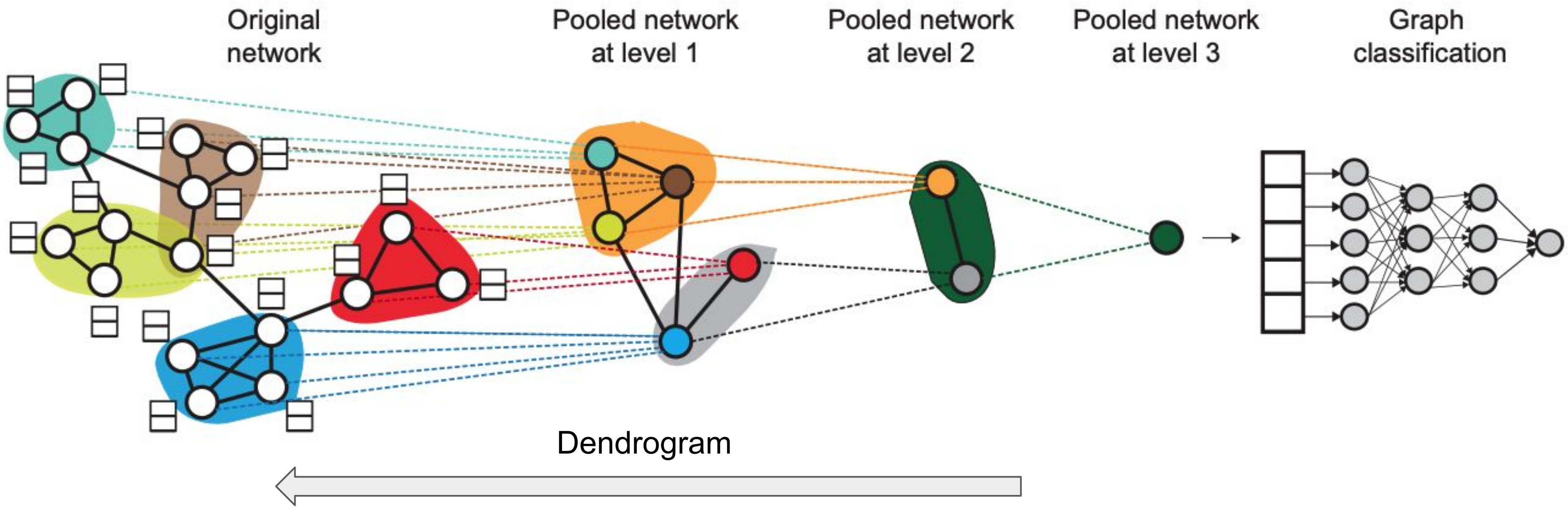
Extracted Rule



Updated Graph

Graph Pooling

[hierarchical]() and [self-attention]() and [differential pool]()



unlike these previous approaches, we seek to *learn* the hierarchical structure in an end-to-end fashion, rather than relying on a deterministic graph clustering subroutine.

Attributed VRGs with DiffPool?

- Incorporate attribute similarity into the dendrogram
- Use both topological and attribute similarity like in AA-cluster ASONAM 17
- Use DiffPool to *learn* the dendrogram
    - Handle node (and possibly) edge attributes
    - Train by minimizing topological / attribute similarity loss or generation quality instead of classification performance
- Use the dendrogram to generate VRG rules like before

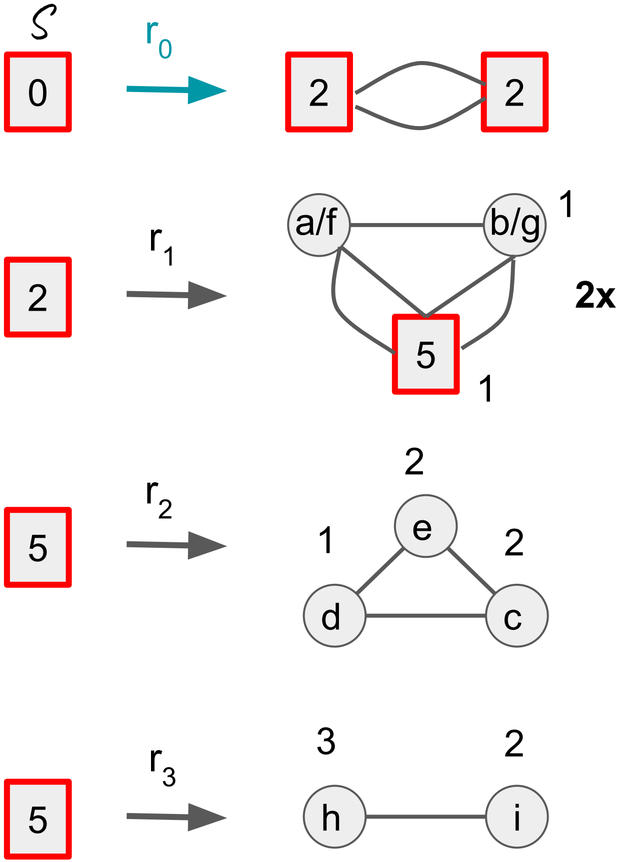
Graph Classification using VRGs

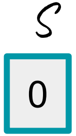
- Use a hierarchical pooling strategy to learn a dendrogram from labeled attributed graphs
- Use dendrogram to create a labeled VRG
  - So for Enzymes dataset, we have ~600 graphs with 6 classes with smallish graphs (n=100, m=150)
  - Learn separate VRGs for each class and generate new graphs - data augmentation
    - VRGs with varying levels of information - inspect the rules - compare with KGist?
  - Augment imbalanced classes?

Possible ideas

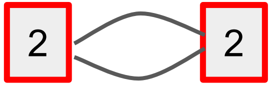
- Compare the quality of generated graphs obtained by using
  - Classical techniques - Louvain, spectral clustering, …
  - Neural network based techniques - obtained by performing spatial hierarchical clustering on the node embeddings
    - Preliminary results on Node2Vec show that hierarchical clustering on embeddings don’t correspond to a topologically faithful dendrogram - leading to disconnected rules
- Using node embeddings lets us use node / edge attributes more easily
- Challenge: we want to ensure that the spatial dendrogram is faithful to the graph topology - DiffPool should help with that since it aggregates connected nodes

Graph Generation using VRGs



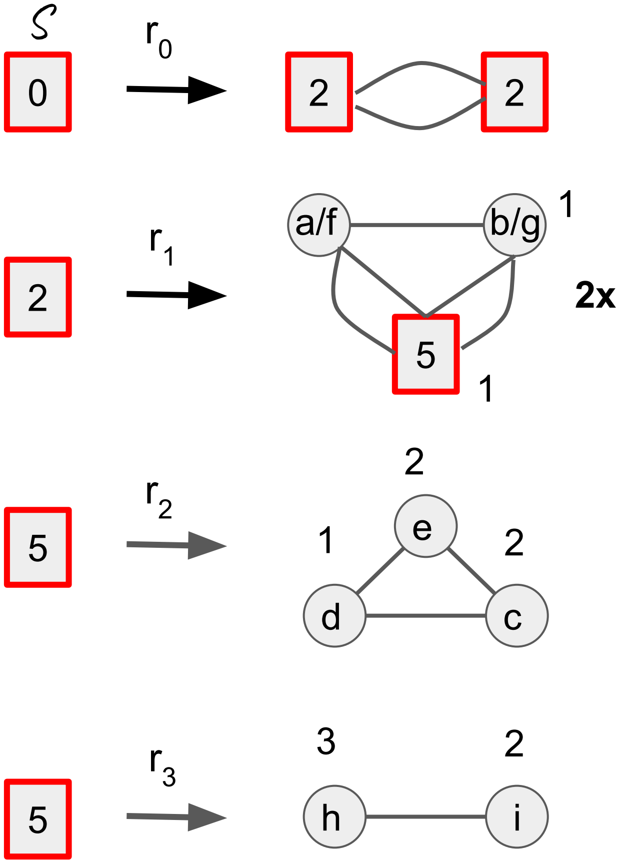


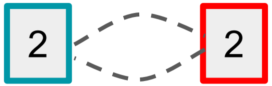
Current Graph



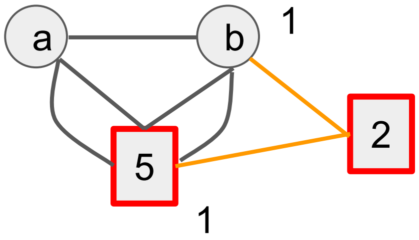
Next Graph
(applying r_0)

Graph Generation using VRGs



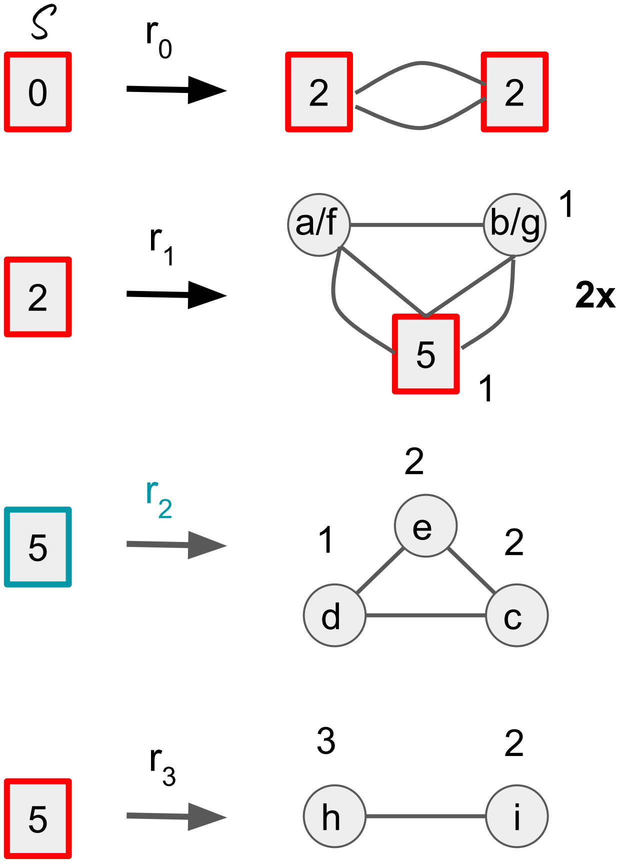


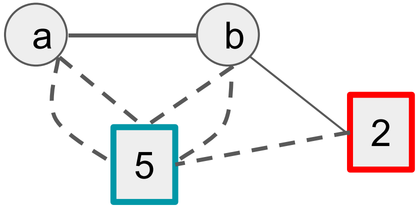
Current Graph



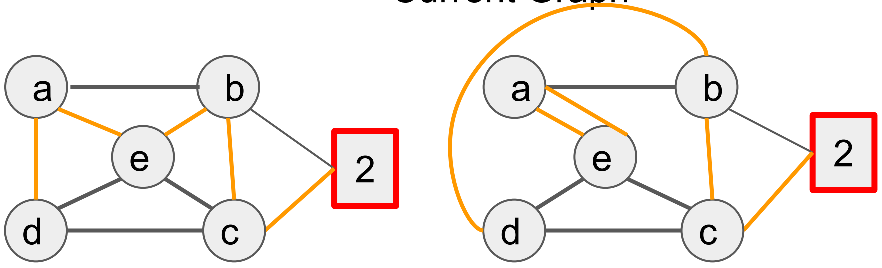
Next Graph
(applying r_1)

Graph Generation using VRGs



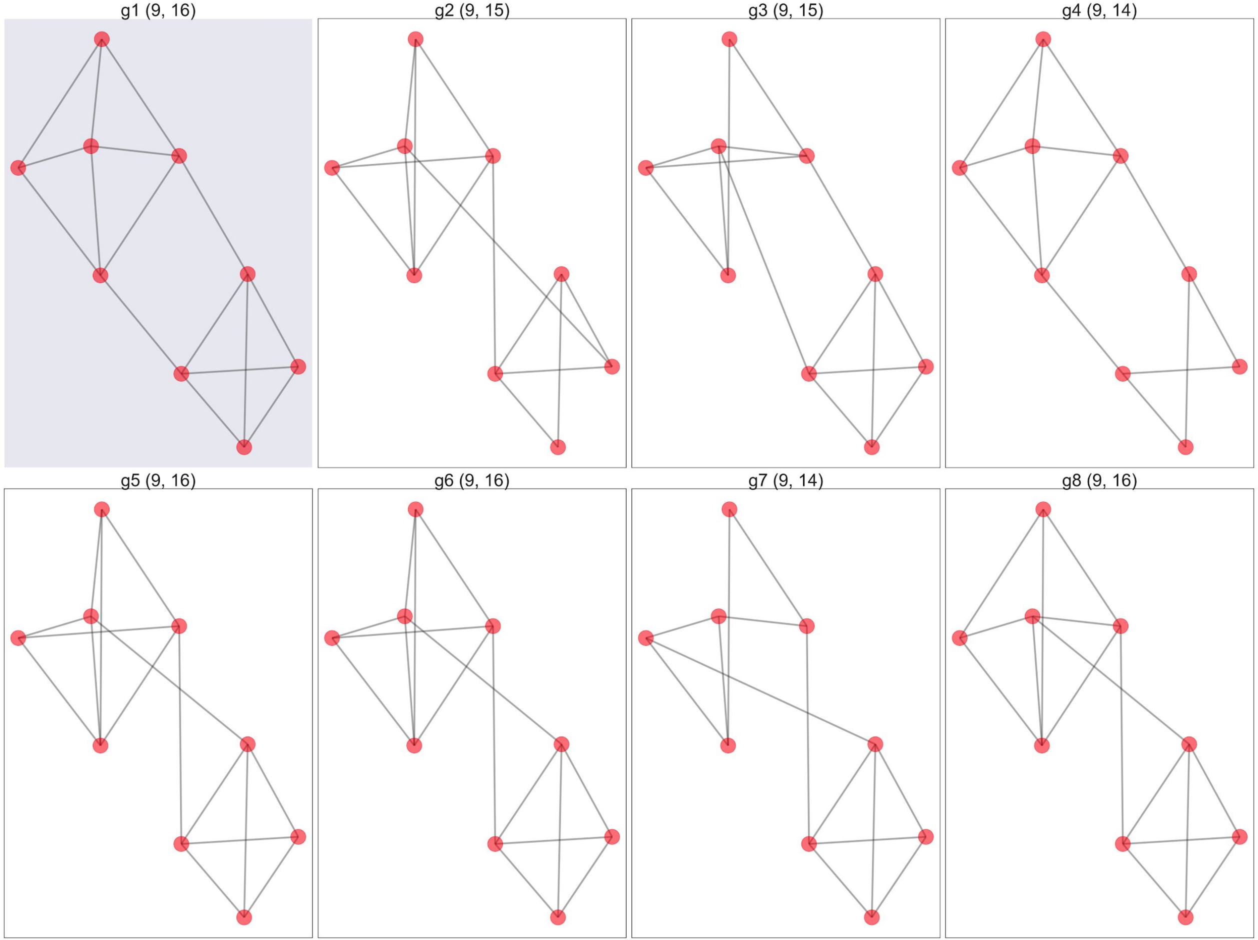


Current Graph



Next Graphs (equally likely)
(multiple choices)

100% node correspondence - retain all nodes



New Ideas for VRG w/ node correspondence

- Use reinforcement learning to generate graphs - pick rules based on partial similarity with the final graph
  - Subgraphs induced by terminal nodes stay constant during generation -- use this and maybe other indicators to assign partial rewards
- Markov assumption doesn’t really hold - current state of the graph has sufficient information for the next step, but past rewiring of edges influence future graphs
  - Rule sequences don’t uniquely determine a graph
- Hard to define the model and the states in advance - depends on rule selection history and current state of the graph
  - Need Model free RL algorithms

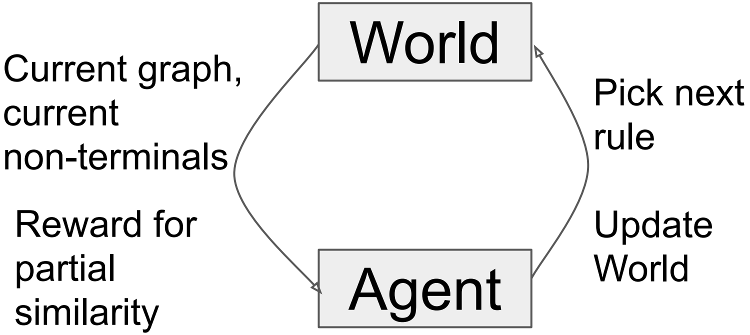
VRG Graph Generation as a RL problem

- Optimization: could be topological / functional similarity to input graphs
- **Delayed consequences**: choosing a rule now, has consequences in the future -- esp the rules extracted from the top of the dendrogram
- Exploration: Trying out different grammar rules with same matching LHS within or across different grammars
- Generalization: you want the rule firings / VRG generation to generalize and not overfit

The World and the Agent

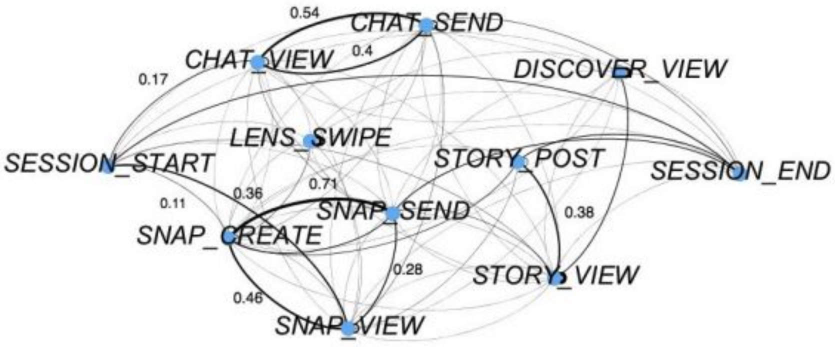
Some observations

- Each input graph has its own grammar
- Each grammar has rules with a LHS and RHS
- Explore: pick a rule from a different grammar (with the same LHS)
- Exploit: keep picking rules from the same grammar



User-level Action Graphs

- 12 nodes -- different actions in the app
- Transition probabilities between the states
- Have state machines for each user
- Individual VRG is not super suitable since it's only 12 nodes - also edge weights are unsuitable



TODOs

- Read more on model free RL
- Make slides for Research Meeting next week
- Get DiffPool to give back dendrograms as an output
- Think of graph level classification tasks for Snap - start with predicting properties of user habits from ego nets?

Model-free RL

- We don’t know the state space in advance
- Known action space - grammar rules - rewiring is still random
- Both action and state spaces are finite and enumerable
- Graph generation is an episode with a fixed beginning and end
  - Start with non-terminal of size 0 and end when no more non-terminals remain
- Markovian dynamics - probability of reaching the next step depends only on the current configuration (kinda)
- Reward dynamics - unknown
- Policy dynamics - unknown

Similarities and differences with Graph Convolutional Policy Networks

- Similarities
  - RL framework for finding optimal policy
  - Step by step graph construction with a defined beginning and end - episodic
  - Constraints during generation - valency and boundary degrees
  - Similar construction of state spaces
- Differences
  - Builds graphs one node / edge at a time
  - Domain specific - optimization policies based on chemical properties
  - Message passing
  - Adding one edge at a time - more granular than VRGs - adding a subgraph at a time
  - Message passing to determine policy - policy gradient training
  - Uses a GAN - adversarial training - distance measure is adversarially trained discriminator
  - Learns from example graphs

Model-free RL: MDP <S, A, P, R, γ>

- States S - different graph configurations
  - First state / initial state - one node with nonterminal of size 0
  - Final state(s) / terminating state(s) - states with no nonterminals
  - Finite number of states
- Action space A
  - Selecting a rule from grammar to replace one of the nonterminals in current graph
  - Rewiring the new subgraph in the existing graph while respecting boundary degrees
  - Finite number of actions since #rules is finite
- Transition dynamics P
  - Infeasible actions are ignored - picking invalid rules for example
  - Computing the probability of the next graph (state) given the previous state
- Reward dynamics R
  - Intermediate rewards - comparing regions of the graph that would remain the same (subgraphs induced by terminal nodes for example)
  - Final rewards - domain specific and adversarial?

VRGs & link prediction on a given set of nodes

- An achievable goal if we do 100% node coverage
  - Train a VRG on 90% edges and 100% nodes - while keeping the graph connected
  - Test on 10% edges and non-edges held out
- Hard to predict performance - depends on rules and the relative position of the held out edges
- For a batch of graphs - we can look into mixing VRG rules from multiple graphs (having the same set of nodes) - to predict the nature of future graphs - train the weights of different rules that would optimize link prediction in future graphs

Updates 07/09

- Making slides for Research Intro and for SIAM Network Science
- Setting up the RL environment
  - Getting familiar with OpenAI’s gym and spinningup packages
- Trying to get RL based graph generation code to work
- Reading up on policy gradient methods with function approximation
- Start prototyping basic RL - initially with no intermediate rewards
- On and off policy learning - graph generation is fast
- Value function formulation - degree_js, pagerank_js,
  - But diversity is important

Updates 07/16

- Struggling with different RL environments and packages :(
  - Chatting with Nils regarding best practices -
- Sticking with OpenAI’s gym environment for now
  - Defining state spaces is kinda tricky - taking inspiration from the RL paper
- Gym is mature - a lot of papers uses Gym and stable-baselines - has a lot of models are already implemented
  - A lot of codebases are overly specific
- Internship duration update - Aug 14 end date

VRGs & link prediction on a given set of nodes

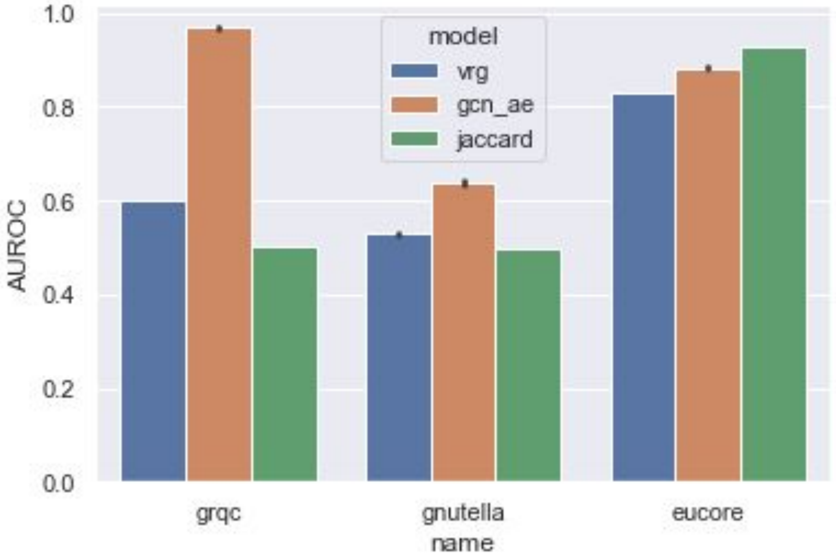
- An achievable goal if we do 100% node coverage
  - Train a VRG on 90% edges and 100% nodes - while keeping the graph connected
  - Test on 10% edges and non-edges held out
- Hard to predict performance - depends on rules and the relative position of the held out edges
- For a batch of graphs - we can look into mixing VRG rules from multiple graphs (having the same set of nodes) - to predict the nature of future graphs - train the weights of different rules that would optimize link prediction in future graphs
- Performance With and without diffpool - effect of attributes on link prediction performance

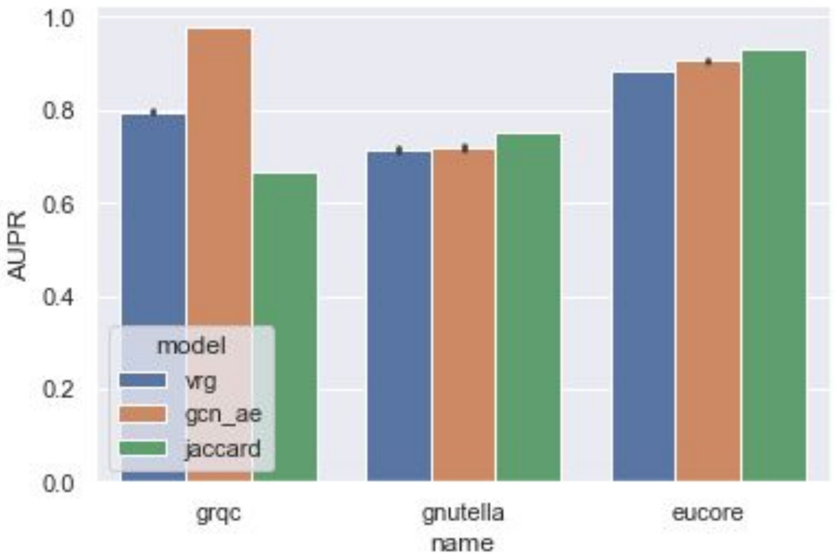
- Low attributed networks - small cardinality
- User demographic info - also link info
- Like a autoencoder

VRGs and link prediction (07/23)

- Basic framework works!!
- Autoencoders also work
- (Hyper?)-parameters
  - Max size of rules
  - Choice of clustering algorithm
- No need for validation edges for VRGs -- no way to integrate feedback

Link prediction performance





grqc: (4k, 13k)   gnutella: (6k, 21k)   eucore: (1k, 16k)

Next steps

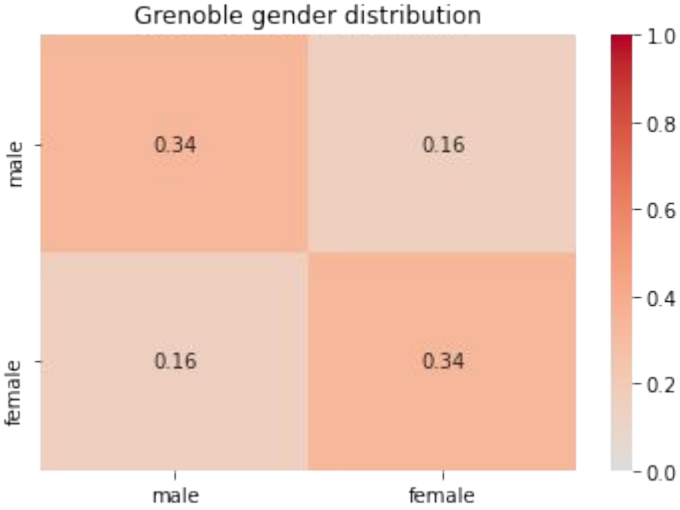
- Integrate attributes in the framework
- Observe the impact of different train/test edges on performance
  - If the clustering of the original graph vs. training graph is very different, the grammar will not match well
  - Use NMI to find similarity of different clusterings
- Get small graphs from Snap data
- Make sure VRG’s extraction process use attribute information
  - Hierarchically cluster attributed graphs
  - Dependent on attribute space -- some algs work only with discrete attributes
  - Diffpool or otherwise
- Link prediction + topological faithfulness

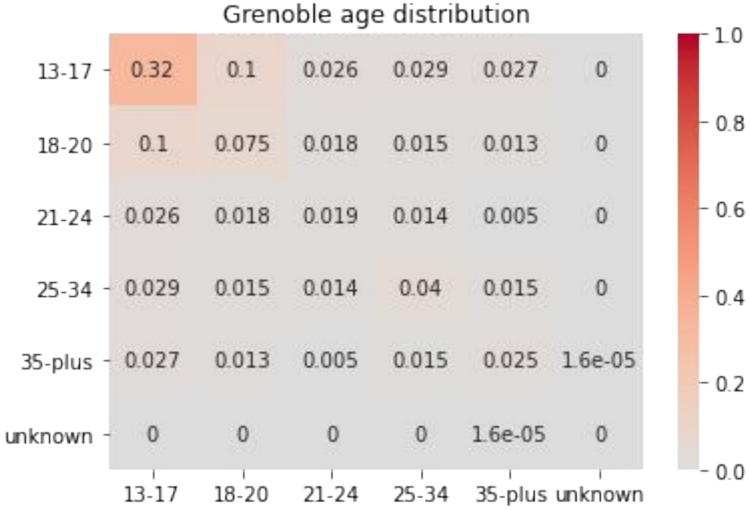
Progress 07/30

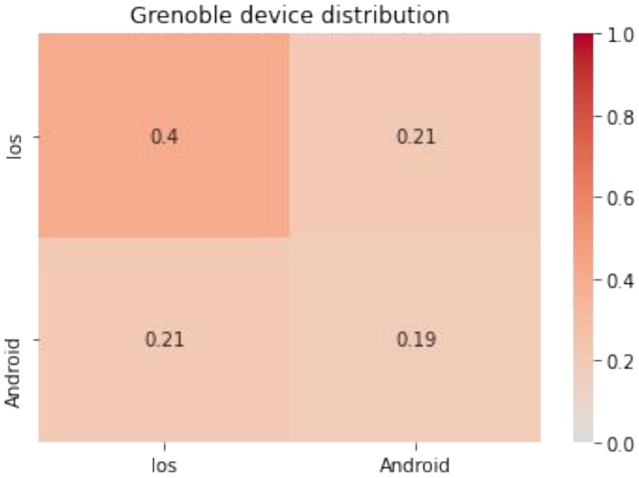
- Attributed clustering
  - AMEN - Brian Perozzi - really nice idea - MATLAB code :(
  - PAICAN - out of TUM - only binary attributes - needs #clusters as input
  - Diffpool requires a batch of graphs -
- Cleaned up Grenoble and Waterloo
  - Treating the graph as undirected w/o self-loops
  - Only users with verified phones and defined genders
  - Taking the largest weakly connected component
- Grenoble extraction takes about 2-3 mins - generation takes 20 secs

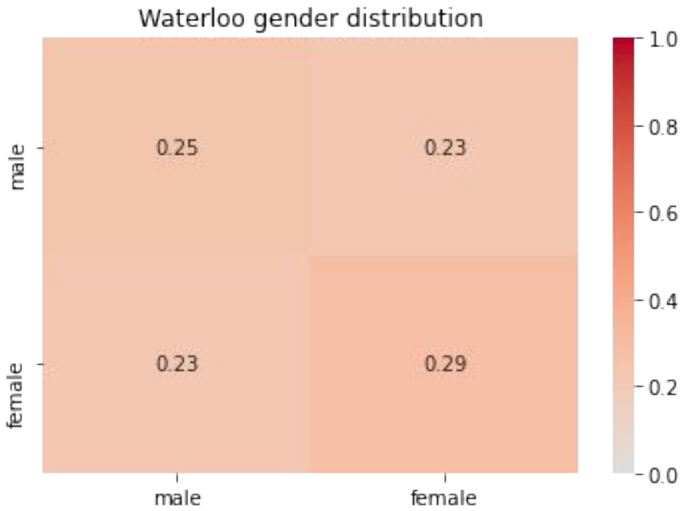
Dataset stats

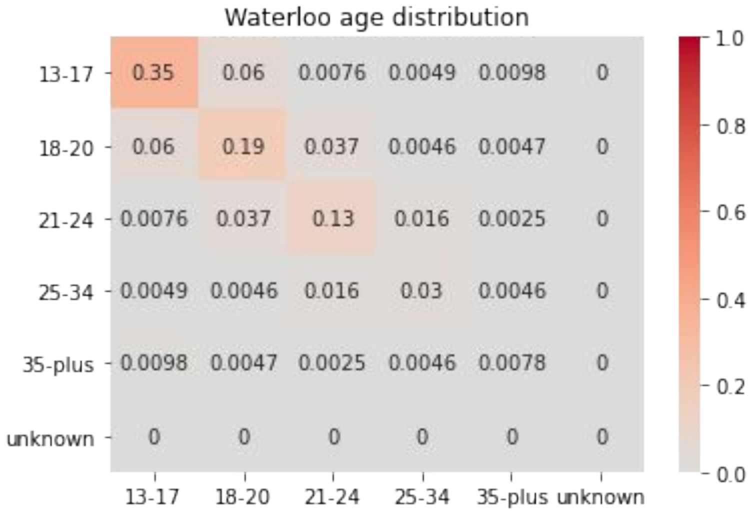
Name	#nodes	LCC #nodes	#edges	Lcc #edges
Grenoble, FR	33.8k	22.8k	304k	151k
Waterloo, CA	47k	23k	277k	188k

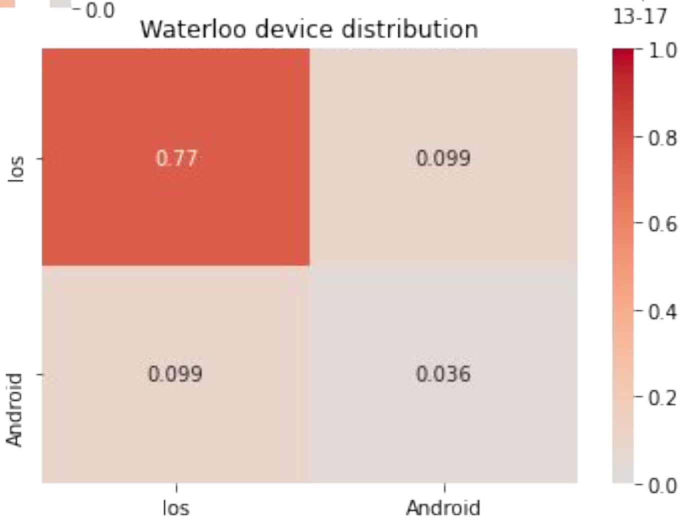

Grenoble gender distribution
male
female
0.34
0.16
0.16
0.34
male
female
1.0
0.8
0.6
0.4
0.2
0.0


Grenoble age distribution
13-17
18-20
21-24
25-34
35-plus
unknown
0.32
0.1
0.026
0.029
0.027
0
0.1
0.075
0.018
0.015
0.013
0
0.026
0.018
0.019
0.014
0.005
0
0.029
0.015
0.014
0.04
0.015
0
0.027
0.013
0.005
0.015
0.025
1.6e-05
0
0
0
0
1.6e-05
0
13-17
18-20
21-24
25-34
35-plus
unknown
1.0
0.8
0.6
0.4
0.2
0.0

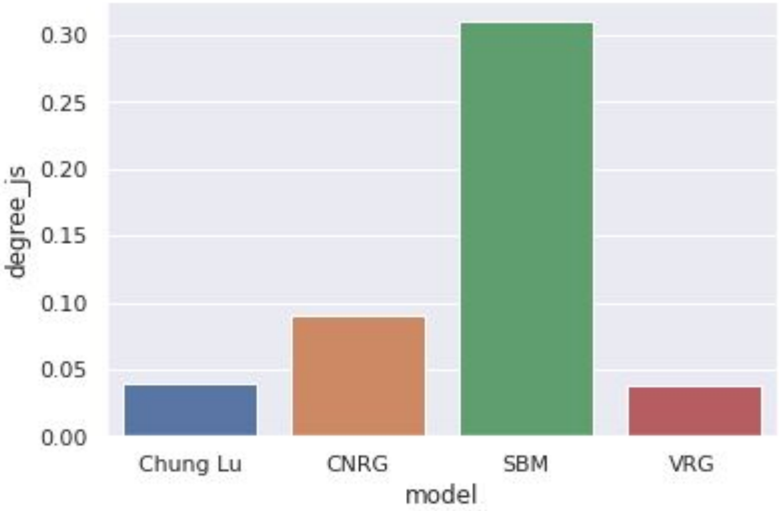

Grenoble device distribution
Ios
Android
0.4
0.21
0.21
0.19
Ios
Android
1.0
0.8
0.6
0.4
0.2
0.0

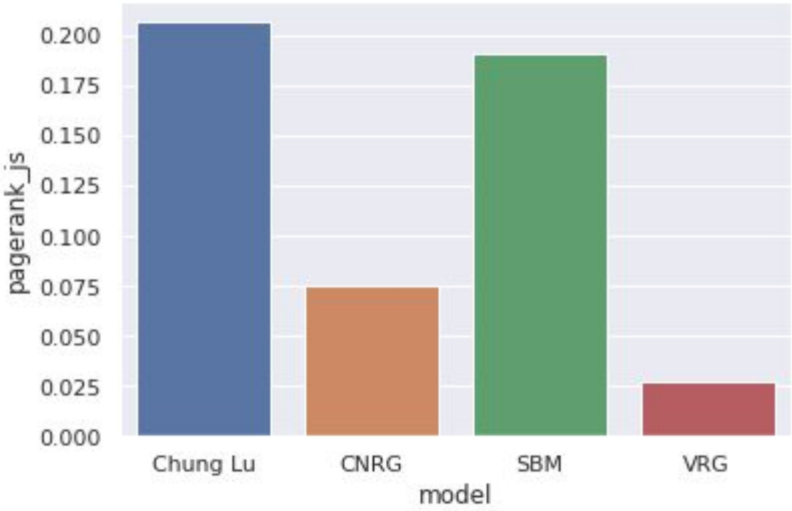

Waterloo gender distribution
male
female
0.25
0.23
0.23
0.29
1.0
0.8
0.6
0.4
0.2
0.0


Waterloo age distribution
13-17
18-20
21-24
25-34
35-plus
unknown
0.35 0.06 0.0076 0.0049 0.0098 0
0.06 0.19 0.037 0.0046 0.0047 0
0.0076 0.037 0.13 0.016 0.0025 0
0.0049 0.0046 0.016 0.03 0.0046 0
0.0098 0.0047 0.0025 0.0046 0.0078 0
0 0 0 0 0 0
1.0
0.8
0.6
0.4
0.2
0.0


Waterloo device distribution
Ios
Android
0.77
0.099
0.099
0.036
1.0
0.8
0.6
0.4
0.2
0.0

Grenoble Graph generation quality (w/o attributes)





Next Steps

- Incorporate edge attributes - as edge weights / artificial edges
- GCNs running out of memory - bump up RAM
- Promising results thus far
- VRG is scalable - clustering takes a little bit of time - Leiden is the fastest
- Inspect the individual rules
- Start an overleaf for WWW (Oct 12) or AAAI (Sep 1) or SDM (Sep 21)….?

Progress (08/06)

- Started to organize all the code into a single repo with runner scripts
- Optimized the weighted clustering algorithms
- Incorporate edge attributes - as edge weights / artificial edges
    - More work than I thought :(
    - Using Jaccard / Adamic-Adar as topological weighting schemes
    - Using Gender + deviceType + age_bucket as attribute weighting schemes
- Enforcing connectivity during extraction greatly decreases #rules
    - 500 - 1,000 of rules get reduced to < 100, with about 10-15 unique RHS subgraphs
    - Generation is unaffected
- There is 1 giant initial rule with LHS 0
    - The dendrograms are bushy and not tall - lots of medium sized clusters & not a lot of big clusters with a lot of subclusters

Grammar Extraction Pipeline

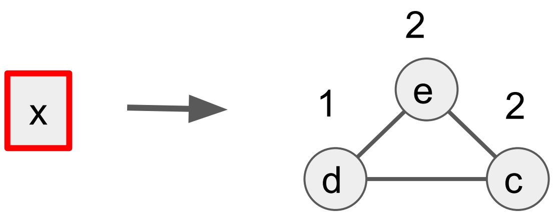
1. Read (attributed) graph
2. Preprocess graph
    a. Adding weights to existing edges
    b. Introducing artificial edges
3. Run hierarchical clustering to obtain dendrogram
4. Iteratively compress subtrees of the dendrogram to generate rules

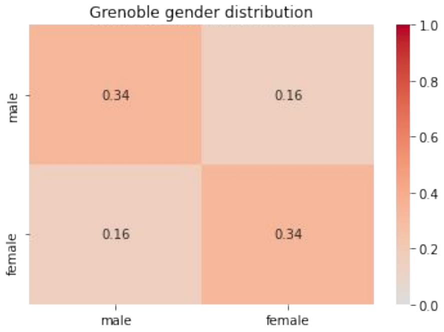
Introducing Artificial edges (in progress)

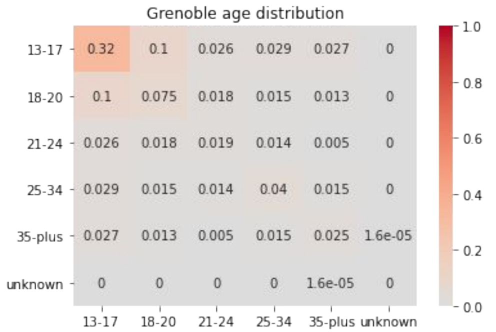
- Computing pairwise similarity for ALL users in the LCC
  - Attribute similarity using Hamming Distance S_1
  - Topological similarity using Jaccard / Adamic-Adar S_2
- Combine the two similarities $S = f(S_1, S_2)$ in any manner $\lambda S_1 + (1 - \lambda) S_2$
- Set a threshold τ to establish artificial edges in the input graph
  - Add weights to all existing edges from S
  - Add artificial edges from S whose weights exceed τ
- Considerations
  - Weighted edges influence the clustering
  - Artificial edges make the graph dense and change the original topology
  - Throw away the edges **before** rule extraction? May result in disconnected rules

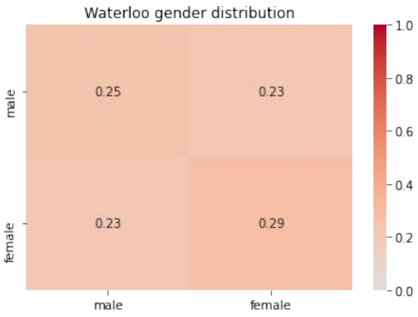
Most Frequent Rule Graphs

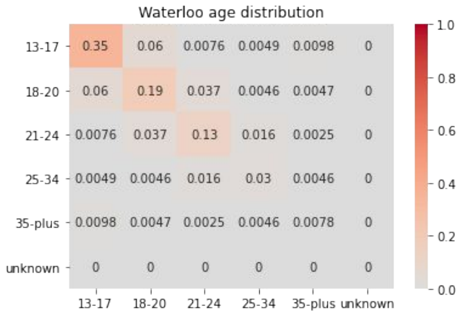
1. Simple Edges between same gendered users / same age groups
2. Lots of triangles











Overall structure of the project

- Scalable and **interpretable** modeling of attributed graphs
  - Extension of CNRG's framework
  - Write / plan / execute the interpretable section before the internship is over
- Augmenting attribute similarity into the topology of the input graph
- Learning a grammar with (hopefully) interpretable rules
- Generating new graphs with topological faithfulness
  - Partial or full node correspondence
  - Isomorphism
  - Competitive link prediction performance

Next Steps

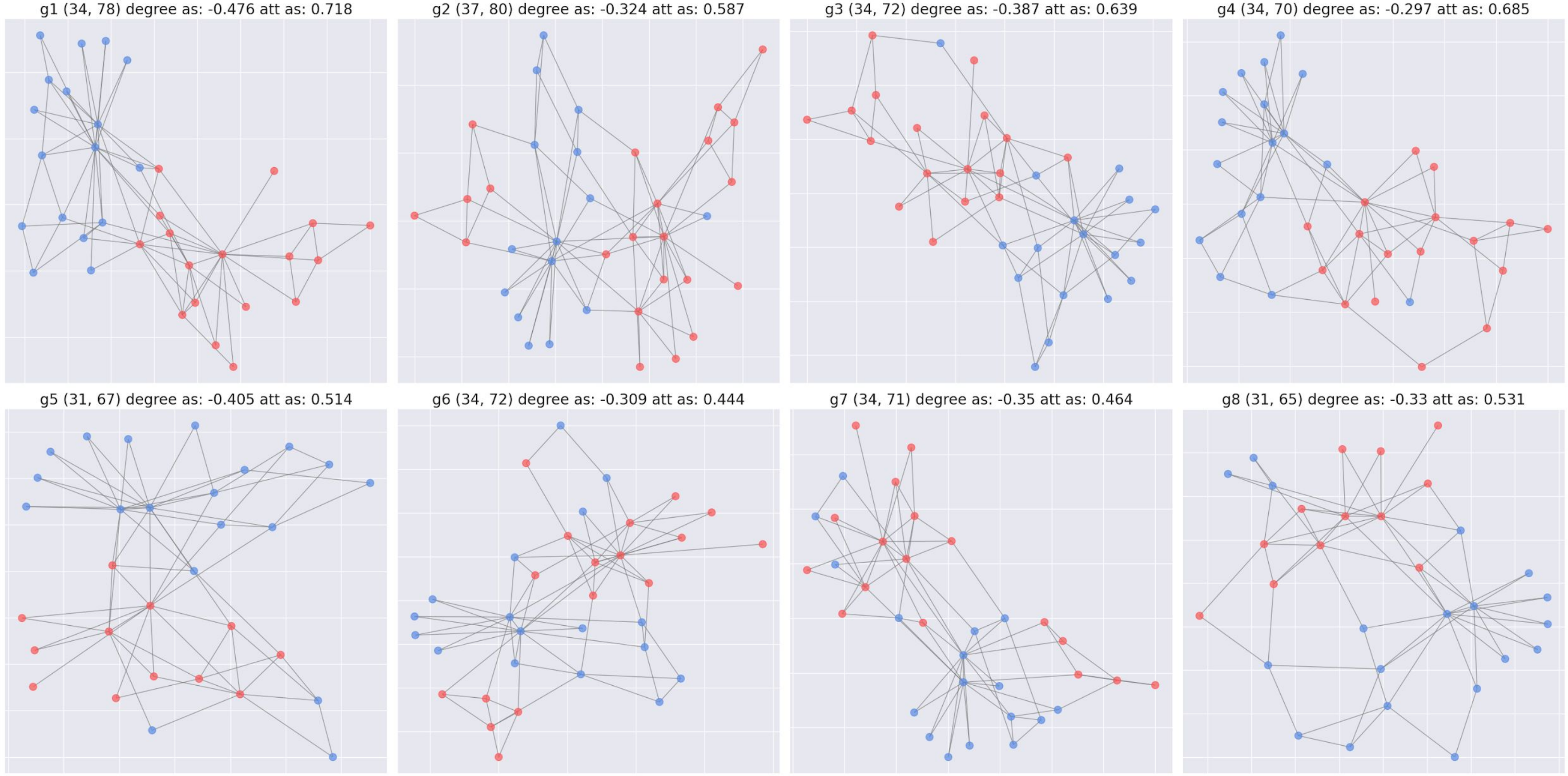
- Incorporate artificial edges
- Inspect the individual rules and try to come up with a narrative for the interpretability section
- Make the dendrograms long and skinny?
- Formalize a plan for offboarding -- esp the codebases
- An extra meeting in the middle of next week - maybe Tuesday?
  - Go through the final set of experiments and the paper narrative

Progress (08/07)

- Incorporating node attributes is not difficult :)
- We can give up node correspondence for flexibility
- **Rule isomorphism** becomes critical
  - We can use ALL / NO / subset of attributes to distinguish the rules
- Rule selection during generation depends on the frequency
  - More popular rules are more likely to get picked during generation
- Compare the generated graphs
  - Topological similarity
  - Distribution of attribute similarity
    - MMD of distributions
    - Assortativity

Karate Club

Original



Assortativity Stats

Name	Attribute	Degree		Attribute	
		Original	Generated	Original	Generated
Karate	Club	-0.476	-0.365 ± 0.047	0.718	0.563 ± 0.084
Grenoble	Gender	0.295	0.14 ± 0.007	0.355	0.004 ± 0.003
	Age Bucket	0.295	0.14 ± 0.007	0.131	0.002 ± 0.003
	Device Type	0.295	0.14 ± 0.007	0.228	0.082 ± 0.003

Attribute assortativity is low - doesn't match

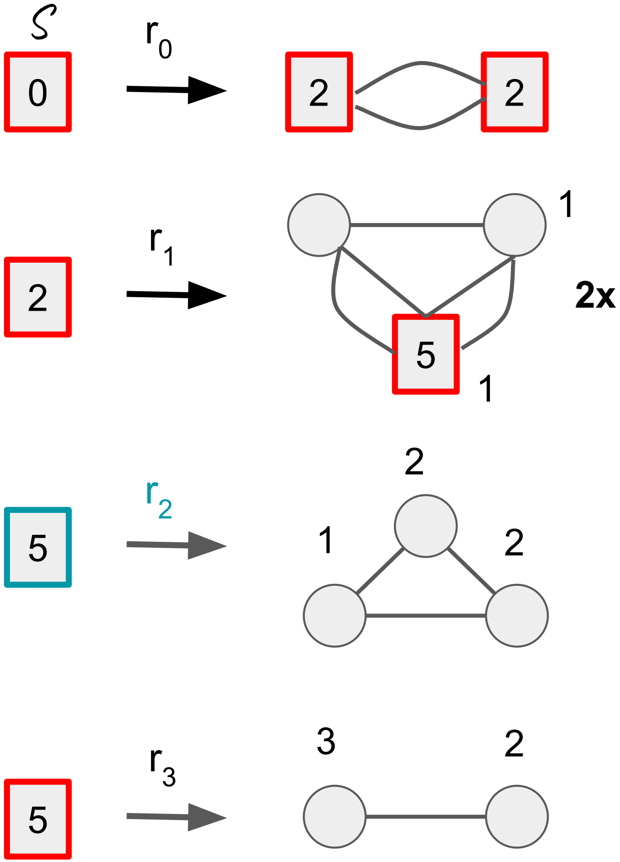
Plans

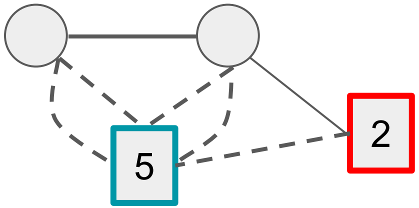
- Take a small graph
    - Shuffle attributes randomly and see the impact on rules - and the results
        - Ideally input and output assortativity / MMD should be strongly correlated
- Run it on Grenoble, Waterloo, …
- Make the graph generation process smarter to incorporate distribution of attributes while picking rules
- Partial node correspondence makes sense in way -- we can preserve connectivity and attribute distributions better?

Progress (08/11)

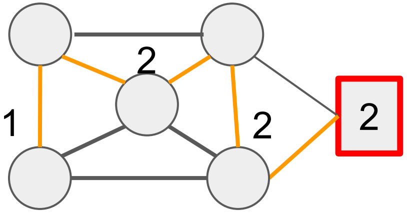
- A LOT of random rewiring happens during generation
  - Messing up assortativity
- Not all edges during generation are created equal
  - Probability of edges between (existing terminal node, new terminal node from rule) should incorporate node assortativity while respecting boundary degrees
- We know what the reference distribution is - from the input graph
- We can nudge the distribution in the right direction

Graph Generation using VRGs



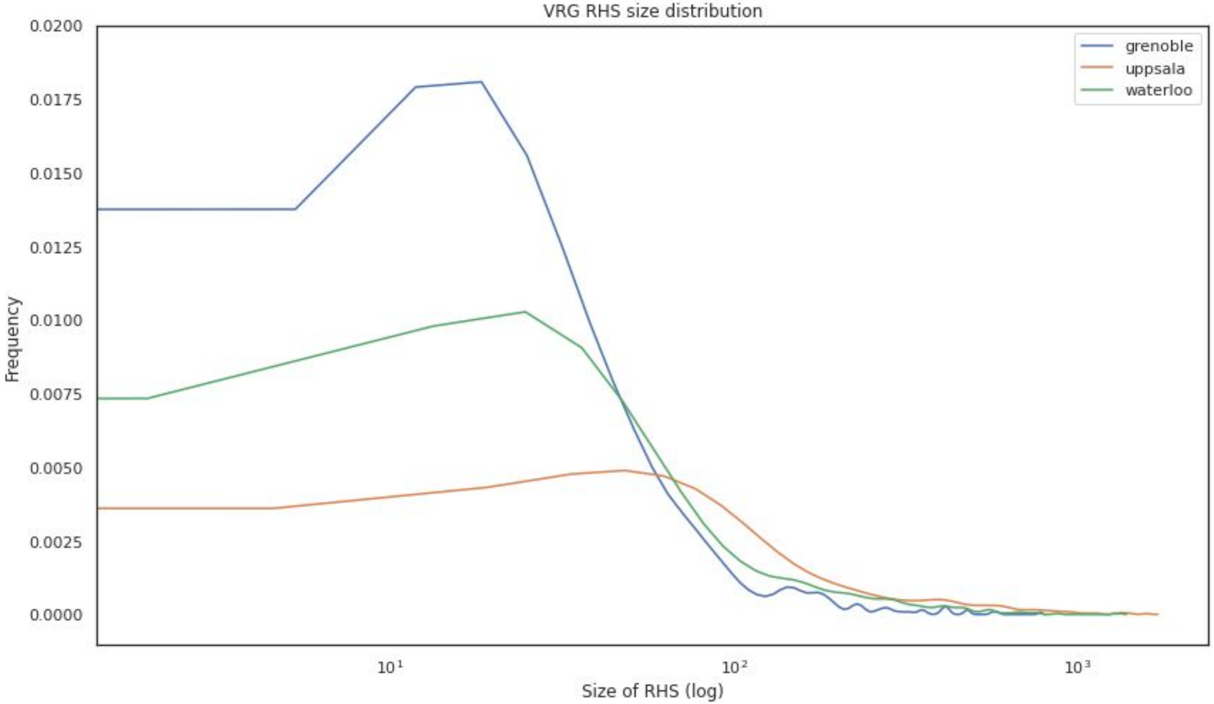


Current Graph



Next Graph

Distribution of #boundary edges



Assortativity Stats with random rewiring

Name	Attribute	Degree		Attribute	
		Original	Generated	Original	Generated
Karate	Club	-0.476	-0.365 ± 0.047	0.718	0.563 ± 0.084
Grenoble	Gender	0.295	0.14 ± 0.007	0.355	0.004 ± 0.003
	Age Bucket	0.295	0.14 ± 0.007	0.131	0.002 ± 0.003
	Device Type	0.295	0.14 ± 0.007	0.228	0.082 ± 0.003

Attribute assortativity is low - doesn’t match

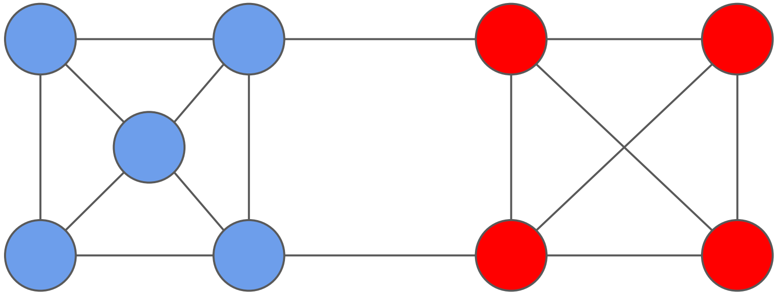
Plans

- Incorporate assortativity as a factor while rewiring boundary edges
- Came across this interesting paper on ranges on assortativity
- Make slides for intern showcase thingy
- Chart out a narrative in the paper
- Talk about

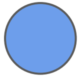
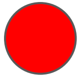
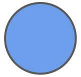
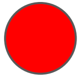
Progress (08/13)

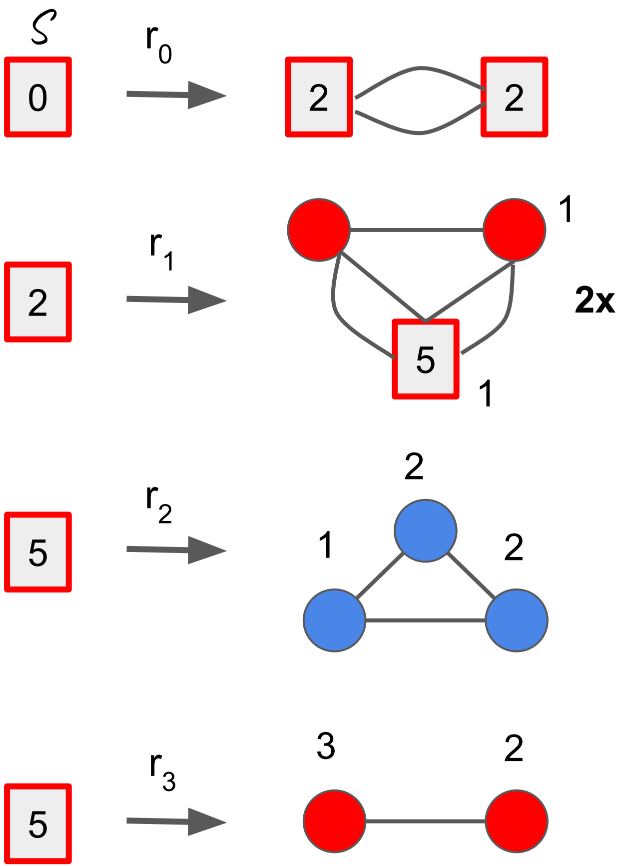
- Incorporate assortativity during rewiring
  - modify CL on a bipartite graph
  - Edges involving nonterminals get picked uniformly
- Put some words in the doc -- very basic sketch of the narrative

Where did the rules come from?



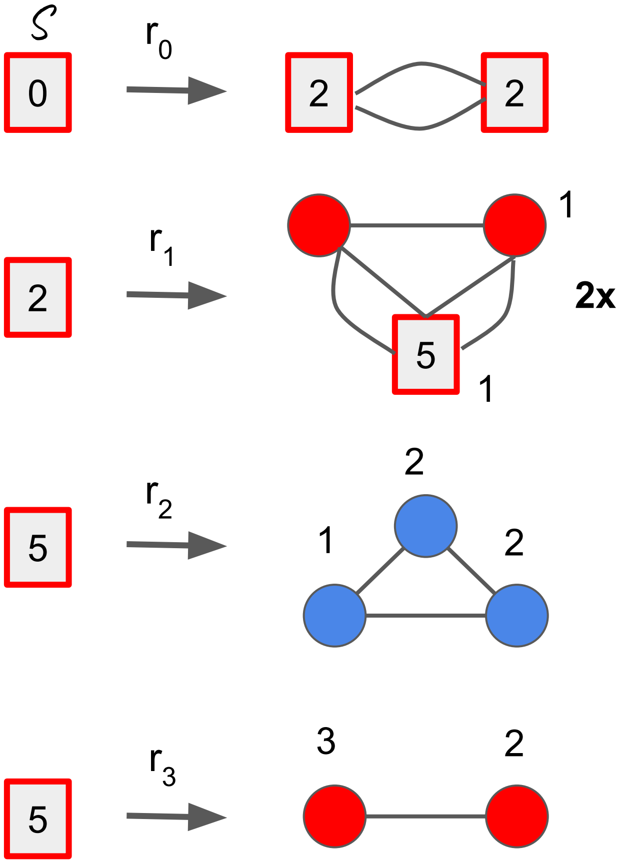
Input Graph

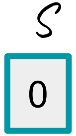
%edges		
	8/16 = 50%	2/16 = 12.5%
	2/16 = 12.5%	6/16 = 37.5%



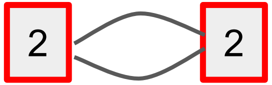
Grammar Rules

Graph Generation using VRGs



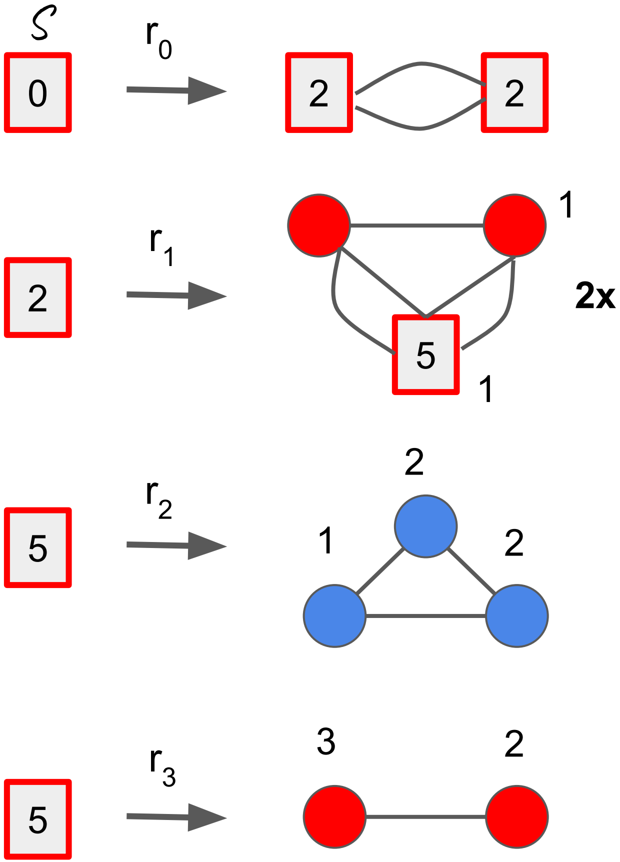


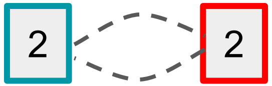
Current Graph



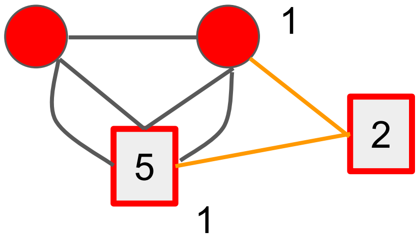
Next Graph
(applying r_0)

Graph Generation using VRGs





Current Graph

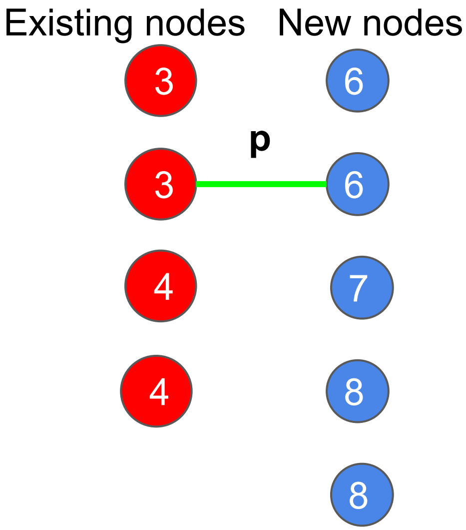


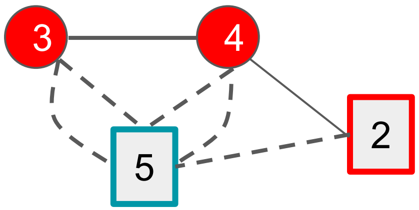
Next Graph
(applying r_1)

Boundary edges are randomly rewired

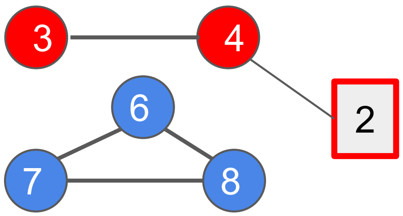
Graph Generation using VRGs

	50%	12.5%
	12.5%	37.5%





Current Graph



Next Graph

Use a CL style rewiring process to assign boundary edges where probability **p** depends on the mixing matrix -- bipartite graph

Graph Generation using VRGs

	50%	12.5%
	12.5%	37.5%

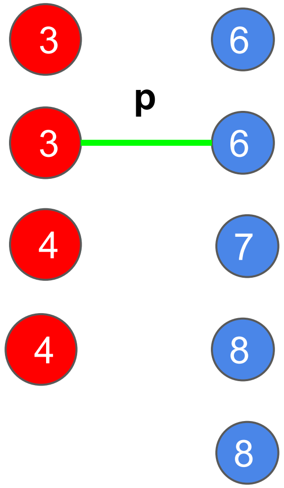
Use a CL style rewiring process to assign boundary edges where probability **p** depends on the mixing matrix -- bipartite graph

Sketch of an algorithm for only nonterminal nodes (both sets)

```
edge_added = True
While edge_added:
    edge_added = False
    Pick an existing node `v_e` uniformly at random
        Pick a new node `v_n` with prob `p(a_e,
    a_n)`
        Add edge (v_e, v_n) to the graph
        Remove nodes v_e and v_n
        edge_added = True

Rewire remaining broken_edges randomly (like before)
```

Existing nt nodes New nt nodes



Plans

- Moved all the code from local machine to Github
- What about the datasets? Currently only in server ssikdar-attr-graphs
- Put some words in the overleaf doc
- Record this about Snap graphs
    - Approx #nodes, #edges
    - Avg degree
    - Density
    - Power-law exponent
    - Distribution of attributes
        - Individual + Joint