Addendum to Textbook: Real World Networks

Michele Coscia (mcos@itu.dk)

September 20, 2018

Abstract

In this addendum we look at strategies to cope with noise and incomplete observation of networks data, specifically on projecting bipartite networks, extracting network backbones, and sampling networks.

1 The Hairball

Reality does not usually match expectations. Let's consider three examples. First, degree distribution. Many papers have been written on how power law degree distributions are ubiquitous. Chances are that any and all the networks you'll find on your way as a network analyst do not have even a hint of a power law degree distribution. In the best case scenario you are going to have shifted power laws, or exponential cutoffs – if you're lucky.

Second example: epidemics spread. In theory, SIS/SIR models tell us exactly when the next node is going to be activated. In practice, data about real activation times has (a) high levels of noise, (b) many exogenous factors that confound the network connections.

Third, and more famously, communities. When it comes to community discovery, the vast majority of papers will use as standard definition a very naive one: groups of nodes internally very dense and with few connections outside – with a claim that most networks have this kind of organization. 99% of networks will instead look like a blobbed mess, which we call hairballs, or spaghettigraph.

There are a few ways in which hairballs arise, which are the focus of this document. First, many networks are not observed directly: they are inferred. If the edge inference process you're applying does not fit your data, it will generate edges it shouldn't. Second, even if you observe the network directly, you might be catching a lot of noise, connections that are not real connections but due to some random fluctuations. Finally, you might have the opposite problem: you're looking at an incomplete sample, and thus missing crucial information.

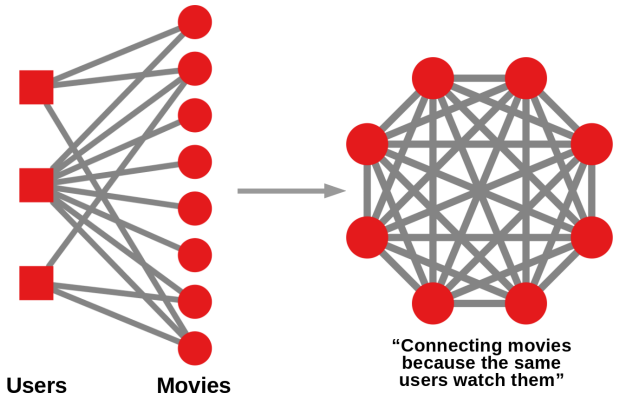


Figure 1: An example of naive bipartite projection, where we connect nodes of one type if they have a common neighbor.

2 Bipartite Projections

Let's consider the common case of projecting bipartite information. Bipartite projection means that you have a bipartite network with nodes of type A and B, and you want to create a unipartite network with only nodes of type A (or B). If you're Netflix, all you observe is people watching movies. This is a bipartite network: nodes of type A are people, nodes of type B are movies, and edges go from a person to a movie if the person watched the movie. However, the end objective is to know which movies are similar, to make recommendations to users.

Naively, you might think that you can connect movies because the same people watched them – as in Figure 1. The problem is that – as we saw – degree distributions are broad. This means that there are going to be some users in your bipartite user-movie network with a very high degree. These are power users, people who watched everything. They are a problem: under the rule we just gave to project the bipartite networks, you'll end up with all movies connected to each other. A hairball. The key lies in recognizing that not all users contribute equally to the similarity of two movies. The users who watched fewer movies should count more.

There are a few ways to tackle this problem which all boil down to the same strategy: we use a different criterion to give the projected edges a weight, we establish a threshold, and drop the edges below a this minimum acceptable weight. The criteria we cover here are:

- Simple Weight;
- Hyperbolic Weight;
- Resource Allocation;
- Random Walks.

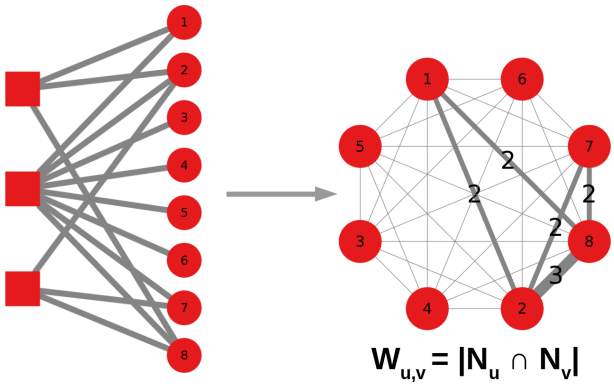


Figure 2: An example of Simple Weight bipartite projection, where we connect nodes of one type with the number of their common neighbors.

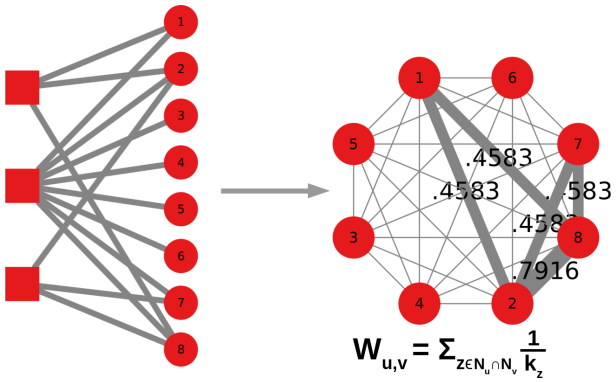


Figure 3: An example of Hyperbolic Weight bipartite projection, where each common neighbor z contributes k_z^{-1} to the sum of the edge weight.

The easiest way to do it is **simple weighting**. For each pair of nodes you identify the number of common neighbors they have, and that's the weight of the edge – see Figure 2. In practice, you don't simply require that movies are connected if there is at least one person who has watched both of them. You connect movies with a weighted link, and the weight is the number of people who watched them both: $w_{u,v} = |N_u \cap N_v|$. This weighting scheme is similar to Common Neighbor in link prediction, and of course you can do a Jaccard correction by normalizing it with the size of the union of the neighbor sets: $w_{u,v} = |N_u \cap N_v|/|N_u \cup N_v|$.

In **hyperbolic weight** we recognize that hubs contribute less to the connection weight than non-hubs. If you're in a CERN paper, you coauthor with hundreds of people, but you don't really know all of them. The weight scheme is similar to the Adamic-Adar link prediction strategy: each common neighbor z contributes k_z^{-1} rather than 1 to the weight of the edge connecting the two

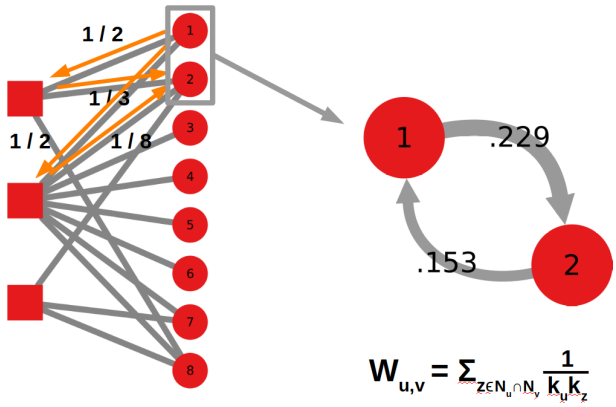


Figure 4: An example of Resource Allocation bipartite projection, where each common neighbor z contributes $(k_u k_z)^{-1}$ to the sum of the edge weight. When connecting node 1 to node 2, from node 1's perspective the edge weight is $(1/2 * 1/3) + (1/2 * 1/8)$, because the two common neighbors have degree of 3 and 8, respectively, and node 1 has degree of two. However, from node 2's perspective, the edge weight is $(1/3 * 1/3) + (1/3 * 1/8)$, because node 2 has three neighbors.

nodes: $w_{u,v} = \sum_{z \in N_u \cap N_v} \frac{1}{k_z}$. The final result in this example is similar to simple weight – see Figure 3 –, but it exaggerates the differences, so that thresholding becomes easier.

In **resource allocation** we do the same thing as hyperbolic weight, but considering two steps instead of one. Rather than only looking at the degree of the common neighbor, we also look at the degree of the originating node. In the paper-writing example, not only it is unlikely to be strongly associated with a co-author in a paper with hundreds of authors, it is also difficult to give attention to a particular co-author if you have many papers with many other people. So each common neighbor z that node u has with node v contributes not k_z^{-1} as in hyperbolic weights, but $(k_u k_z)^{-1}$ – see Figure 4. The weight is then $w_{u,v} = \sum_{z \in N_u \cap N_v} \frac{1}{k_u k_z}$. Note that in this case the projection is not symmetric any more: in the scenario with a single common neighbor z, u's score for v would be $(k_u k_z)^{-1}$, while v's score would be $(k_v k_z)^{-1}$. If $k_u \neq k_v$, then the scores are different. You can solve this issue by considering either the minimum or maximum of the two, or have a directed network as result.

In **Random Walks**, we take the resource allocation to the extreme. Rather than looking at 2-step walks, we look at infinite length random walks. Which means that the strength between u and v is the probability of visiting v starting from u. If we have infinite random walks, this means that we can use the stationary distribution to estimate the edge weight: $w_{u,v} = \pi A_{u,v}$, where A is a transition probability matrix (recording the probability of the path $u \to z \to$

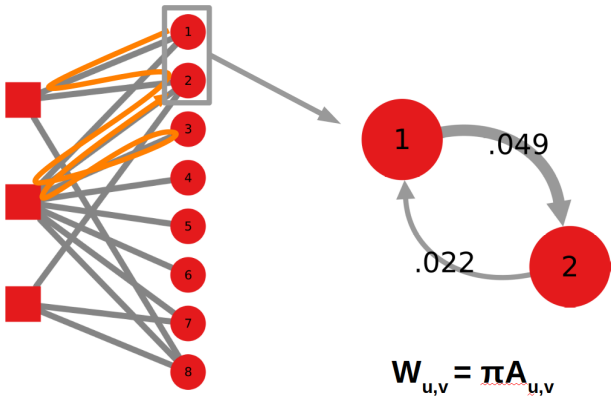


Figure 5: An example of Random Walks bipartite projection, where the connection strength between u and v is dependent on the stationary distribution π, telling us the probability of ending in v after a random walk.

v, for any z). As in the resource allocation case, this means the measure is not symmetric, and the differences between nodes now are more extreme than before: the $1 \to 2$ edge weight is now more than twice as $2 \to 1$, while in resource allocation it was just about 50% higher. See Figure 5 for an example.

If we want to avoid hairballs and related problems, these techniques – while necessary – are not usually sufficient. In fact, the process we implied so far has two steps: first one performs the bipartite projection, and then she applies a threshold to throw away low-weighted edges. The next section expands on how to perform this second step properly.

3 Network Backboning

Network backboning is the problem of taking a network that is too dense and removing the connections that are likely to be not significant – or "strong enough". If you ever found yourself in a situation thinking "there are too many edges in this network, I'm going to filter some out", then you performed network backboning. Even if it is rarely explicitly labeled like that, network backboning is one of the most common tasks performed in network analysis.

The reason not many network researchers mention it is because they usually apply a limited set of naive strategies and do not recognize it as a problem in itself. In fact, there is an easy naive solution that most researchers apply without a second thought. If we have a weighted network and we want to keep the "strongest connections", we sort them in decreasing order of intensity. We decide a threshold, a minimum strength we accept in the network. Everything not meeting the threshold is discarded.

There are two problems with the naive strategy. First is that, in real world networks, much like the degree also edge weights distribute broadly, in a fat-

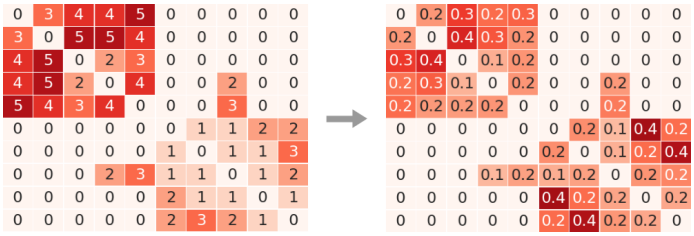


Figure 6: An example of Doubly Stochastic network backboning. The adjacency matrix on the left has two areas of the network with different edge weight scales. Once the matrix is transformed into its doubly stochastic counterpart, such correlations are eliminated.

tail highly skewed fashion. If 60% of your edges have weight of 1, the smallest possible hard threshold would remove 60% of your network, without allowing for any nuance. Moreover, if we have a power-law edge weight distribution, it is hard to motivate the choice of a threshold. A power-law distribution lacks of a well-defined average value and has undefined variance. You cannot motivate your threshold choice by saying that it is "x standard deviations from the average" or anything resembling this formulation.

The second problem is that edge weights are usually correlated. Nodes that connect strongly tend to connect strongly with everybody. So the weight of an edge is correlated with the weights of the edges of the nodes it connects. This means that there are areas of the network with high edge weights and areas with low weights. If we impose the same threshold everywhere, some nodes will retain all their connections and others will lose all of theirs, without making the structure any clearer.

There are several network backboning methods which aim to fix these problems. Here we consider four:

- Doubly Stochastic;
- High Salience Skeleton;
- Disparity Filter;
- Noise Corrected.

The first approach we look at is the **doubly stochastic** strategy. Remember what a stochastic matrix is: it is the adjacency matrix normalized such that the sum of the rows is 1. A *doubly* stochastic matrix is a matrix in which the sums of both rows *and* columns are equal to one. You can transform an adjacency matrix into its corresponding doubly stochastic by alternatively normalizing rows and columns until they both sum to 1. After you perform such normalization, the scale of all edges is the same, and you break local correlations – as we show in Figure 6. You can now threshold the edges without fearing for the issues we mentioned before. The downside of this approach is that not all matrices can

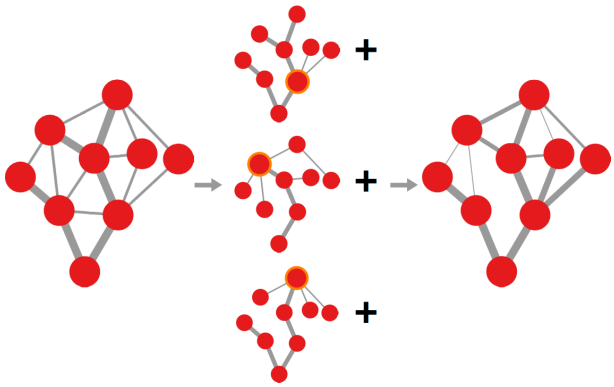


Figure 7: An example of High Salience Skeleton network backboning. The original graph is used to create a shortest path tree for each node in the network. The trees are then summed, and the result is new edge weights for the original graph that can be thresholded.

be transformed into a doubly stochastic form so long as they are sparse, and most real world networks are. So this solution cannot be always applied.

To build the **high salience skeleton** (HSS) we loop over the nodes and we build their shortest path tree: a tree originating from a node, touching all other nodes in the minimum number of hops possible and maximum amount of edge weight possible. In practice we start exploring the graph with a BFS and note down the total edge weight of each path. When we reach a node that we already visited we consider the edge weights of the two paths and the one with the highest one wins. We perform this operation for all nodes in the network and we obtain a set of shortest path trees. We sum them so that each edge now has a new weight: the number of shortest path trees in which it appears. The network can now be thresholded with these new weights.

The HSS makes a lot of sense for networks in which paths are meaningful, like infrastructure networks. However, it requires a lot of shortest path calculations – which makes it computationally expensive. Moreover, the edges are either part of (almost) all trees or of (almost) none of them. This can be nice, because it means HSS can be almost parameter free: the thresholding operation does not have many degrees of freedom. On the other hand, when there are few edges with weights close to one your skeleton might end up being too sparse and it is difficult to add more edges without lowering the threshold close to zero.

One implicit assumption of all these methods is that edge weights have to be made on the same scale across the entire network. The **disparity filter** (DF) takes a different approach. Rather that modifying the edge weights so that you can have a global threshold, it takes a node-centric approach. Each node has a different threshold to accept or reject its own edges. This is done by modeling an expected typical "node strength", for instance the average of its edge weights. Then we keep only those edges which are higher than the expected edge weight

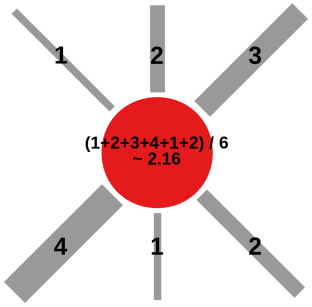


Figure 8: A schematic simplification of Disparity Filter network backboning. The node determines its customized threshold by building an expectation of its average connection strength. Every edge weight higher than this expectation in a statistically significant way is kept.

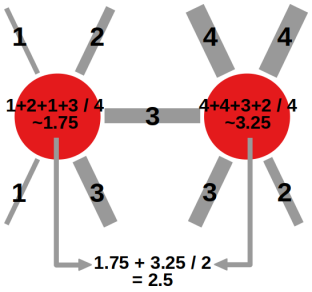


Figure 9: A schematic simplification of Noise Corrected network backboning. The edge determines its customized threshold by building an expectation of the average connection strength of its two nodes. If its weight is higher than the expectation in a statistically significant way then the edge is kept.

for this node, making sure that this difference is statistically significant. Figure 8 depicts a simplification of the method.

However, the disparity filter fails to take into account that some nodes have inherently stronger connections. For instance, consider a mobility network, tracking commuters between cities in the United States. New York has a lot of people and thus will have strong mobility links with any place in the US. In the disparity filter, edges are checked twice from both nodes' perspectives: few of New York's links are stronger than its average, but almost all of them are the strongest in the perspective of the smaller towns to which New York connects. Since you need one success to keep the edge, you end up with strong hubs connected to the entire network, and few peripheral connections (hub-spoke structure, or core-periphery, with no communities).

The **noise-corrected** (NC) approach attempts to fix this issue. In spirit, it is very similar to the disparity filter. However, the focus is shifted towards an edge-centric approach: each edge has a different threshold it has to clear if

it wants to be included in the network. The assumption is that an edge is a collaboration between the nodes. It has to surpass the weight we expect given both nodes' typical connection strength. Again, we have to make sure that this difference is statistically significant. Figure 9 depicts a simplification of the method.

As you might expect, these methods exist because they give very different results. It is up to you to decide which of their assumptions best fit the network you are analyzing and the type of things you want to say about the network. A naive threshold fixes the same obstacle for all nodes no matter how strong, favoring the connections of the hub; HSS can include weaker links if they're the only path to a node; DF is similar to naive, but can recognize important weak edges; and NC overweights peripheries and communities: it is the most punishing method for the central hubs.

4 Network Sampling

Sometimes, having a good edge induction or network backboning technique still doesn't help you. Sometimes you're observing a network directly and it's just a hairball. In these cases, it's useful to make a step back and consider that the act of observation in itself is not neutral. We decide what to focus on, whether we do it because of our interests, or simply because of data availability. If we could zoom out, we would see the structure. When you're unable or unwilling to look at the entire network you have to perform network sampling. A proper network sampling will ensure that the tiny sliver you observe is carrying the properties of the whole structure you're interested in.

To put in perspective how bad the problem is, consider Twitter. As of writing this paragraph, Twitter has more than 300 million active users. According to its API, it takes a bit more than a minute on average to fully know the connections of a user. This means that it takes more than 20 billion seconds to crawl the entirety of Twitter, or just a bit less than 700 years. If you were to crawl constantly for one year, you'd get a bit more than 0.1% of Twitter. You can understand that what ends up in your 0.1% has to be the best possible representation of the whole, and thus it has to be chosen carefully.

We already saw some ways to explore a graph: BFS and DFS. They are reasonable ways to explore a graph, but their underlying assumption is that, eventually, they will cover the entire network. Here we focus on a slightly different perspective. We don't want the entire network: we want to prevent biases to creep into our sample. The three network sampling classes we focus on are:

- Random Walks;
- Snowball;
- Forest Fire.

In **Random Walk** (RW) sampling, we take an individual and we ask them to name one of their friends at random. Then we do the same with her and so on. Basically, we're doing a random walk through the network. This is an easy approach which can be very effective, but it has problems. First, you might end up trapped in an area of the network where you already explored all nodes, thus unable to find new ones. More importantly, RW sampling has a degree bias. Remember the stationary distribution: the probability of ending in a node with a random walk is known and constant no matter where we started. And the stationary distribution has a 1-to-1 correspondence to the degree. This means that high degree nodes are very likely to be sampled, while low degree nodes not so much. Thus, with RW, your sample is not representative – at least when it comes to representing nodes with all degrees fairly.

So we make a correction to the random walk. In the Metropolis-Hastings Random Walk (MHRW), when we get a new neighbor we do not accept it with probability 1. Instead, we look at its degree. If its degree is higher than the one of the node we are visiting, we have a chance of rejecting this neighbor and trying a different one. This probability is the old node's degree over the new node's degree. The exact formula for this decision is $p = k_v/k_u$, assuming that we visited v and we're considering u as a potential next step. Thus, if the current node v has degree of 3, and its u neighbor has degree of 100, the probability of transitioning to u is only 3%. A random walk with this rule will generate a uniform stationary distribution.

In Re-Weighted Random Walk (RWRW), rather than modifying the way the random walk works, we perform a normal one. Once we're done exploring the network, we correct the result for the property of interest. Say we are interested in the degree. We want to know the probability of a node to have degree equal to i. We correct the observation with the following formula:

$$p_i = \frac{\sum_{v \in V_i} i^{-1}}{\sum_{v' \in V} x_{v'}^{-1}}.$$

The formula tells us the probability of a node to have degree equal to i (p_i). This is the sum of i^{-1} for all nodes in the sample with degree i (V_i), over 1 / degree ($x_{v'}^{-1}$) of all nodes in the sample (V). This is also known as Respondent-Driven Survey, because it is used in sociology to correct for biases in the sample when the properties of interest are rare and non-randomly distributed throughout the population.

In **Snowball** sampling we start by taking an individual and asking her to reveal k of her connections. She might have more than k friends, but we only take k. Then we use these new individuals and we ask them the same question: to name k friends. We do so with a BFS strategy. In practice, Snowball is BFS, but imposing a maximum degree: k. Snowball has some advantages. It is cheap to perform in the real world, where the cost of identifying nodes is high, because the nodes identify themselves as a part of the survey process. This is less relevant for social media, where node discovery is relatively easy. Snowball has a smaller

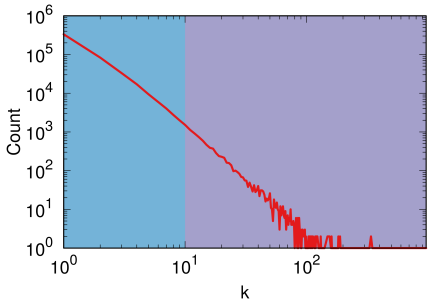


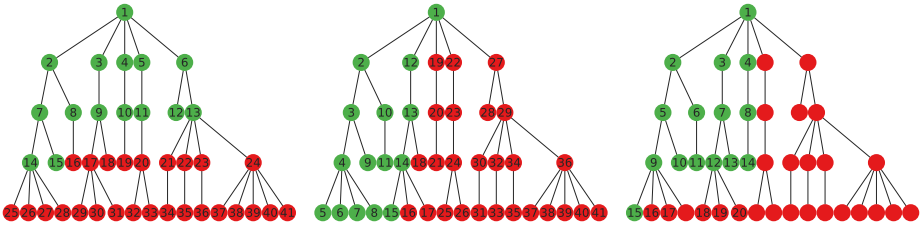
Figure 10: A power law degree distribution, showing the count of nodes (y-axis) with a given degree (x-axis). The colors in the plot represent in which cases the first API policy described in the text is faster than the second (purple) and when the second is faster than the first (blue).

degree bias, because you never have hubs in the network, since the maximum degree is k. However, this also generates weird degree distributions. Finally, it works well with pagination: in social media, when you ask the connections of a node, you rarely get all of them. Social media paginate results, so you only get the first k connections. With Snowball you can easily decide the maximum number of pages you want.

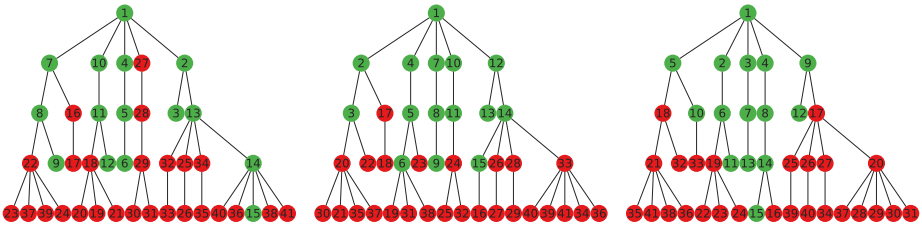
Since we're talking about pagination, bear in mind that it can be tricky to know how it will affect crawl time. Imagine two different API policies. The first returns big pages – say 100 edges per page –, but requires a large waiting time between queries – say two seconds. The second policy returns small pages – ten edges per page –, but more often – you only need to wait one second between queries. We can calculate the edge throughput of these two policies. In this case, the one with big pages returns more edges per unit of time, on average: 50 edges per seconds versus 10 edges per second. However, how do these policies behave on a real world network?

As we saw, real world networks have broad degree distributions, like the one we show in Figure 10. For some of these nodes, the second policy is better: if they have 10 or less edges, we can fully explore them with a single query, thus we're going faster because we have lower waiting times between nodes. In Figure 10, we color in blue the part of the degree distribution for which this holds true. If the node has a degree higher than 10, then the second policy is better, because it requires to perform fewer queries, even if they are spaced out more in time. In Figure 10, we color in purple the part of the degree distribution for which this holds true. In a broad degree distribution, we have way more nodes with low degree. In Figure 10, out of 500k nodes, 492k have degree of 10 or less. The second policy is in theory 5 times slower than the first (10 edges/sec versus 50 edges/sec, on paper), however it will allow you to crawl this network in half of the time.

In **Forest Fire**, like in Snowball, the base exploration is a BFS. However,



(a) BFS (b) DFS (c) Snowball



(d) Random Walk (e) MHRW (f) Forest Fire

Figure 11: Examples of how different network sampling strategies explore a given network. Each node is labeled with the order in which it is explored. The node color shows whether the node was sampled (green) or not (red), assuming a budget of 15 units and a constant cost of 1 unit per node. Snowball assumes $n = 3$ (unlabeled nodes are not explored due to this parametric restriction), while Forest Fire has a burn probability of .5.

once we get all neighbors of a node, we do not explore them all. Instead, for each of them, we flip a coin and we explore the node only with probability p. The advantage of Forest Fire is usually linked with a proper estimation of the clustering coefficient of the network, since with a BFS we would overestimate it – because we fully explore the neighborhood of nodes.

Figure 11 shows an example of how some of these different strategies would explore a simple tree.