DeepWalk & node2vec

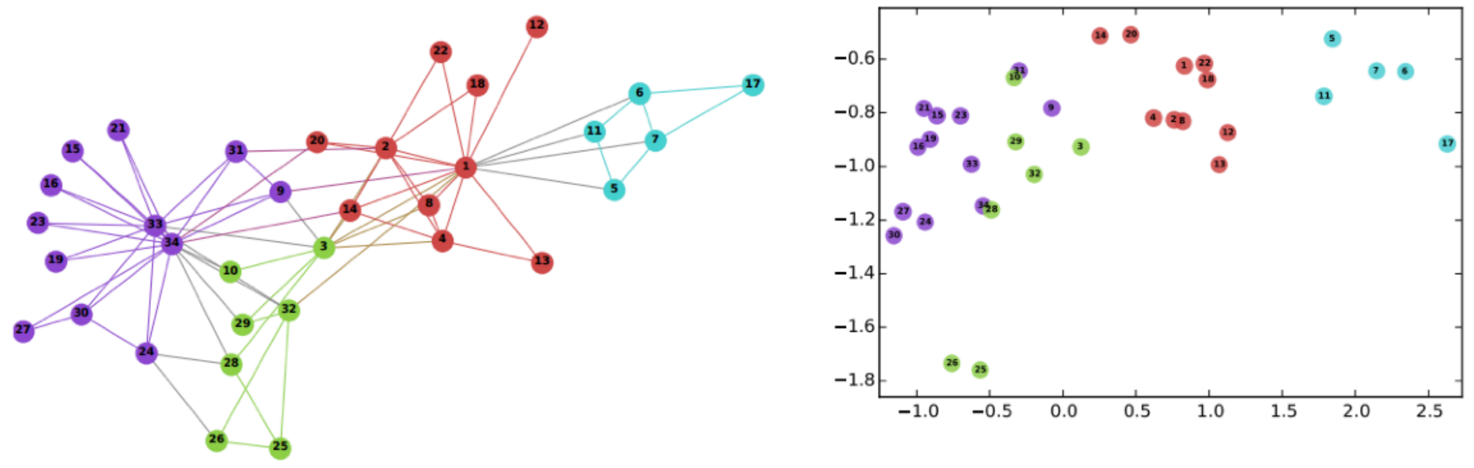
Deepdog 08.25.21

Abstract

- Learning latent representations of vertices in a network.
- These latent representations encode social relations in a continuous vector space
- DeepWalk uses local information obtained from truncated random walks to learn latent representations by treating walks as the equivalent of sentences.

Introduction

- NLP에서 검증된 deep learning 방법을 graph structure에 적용하려고 한다.
- DeepWalk learns social representations of a graph's vertices.
- Social representations are latent features of the vertices that capture neighborhood similarity and community membership



(a) Input: Karate Graph (b) Output: Representation

Figure 1: Our proposed method *learns* a latent space representation of social interactions in $\mathbb{R}^d$. The learned representation encodes community structure so it can be easily exploited by standard classification methods. Here, our method is used on Zachary's Karate network [44] to generate a latent representation in $\mathbb{R}^2$. Note the correspondence between community structure in the input graph and the embedding. Vertex colors represent a modularity-based clustering of the input graph.

Problem Definition

- Classifying members of a social network into one or more categories.
  This is known as the relational classification(collective classification)

- Unsupervised method which learns features that capture the graph structure independent of the labels' distribution

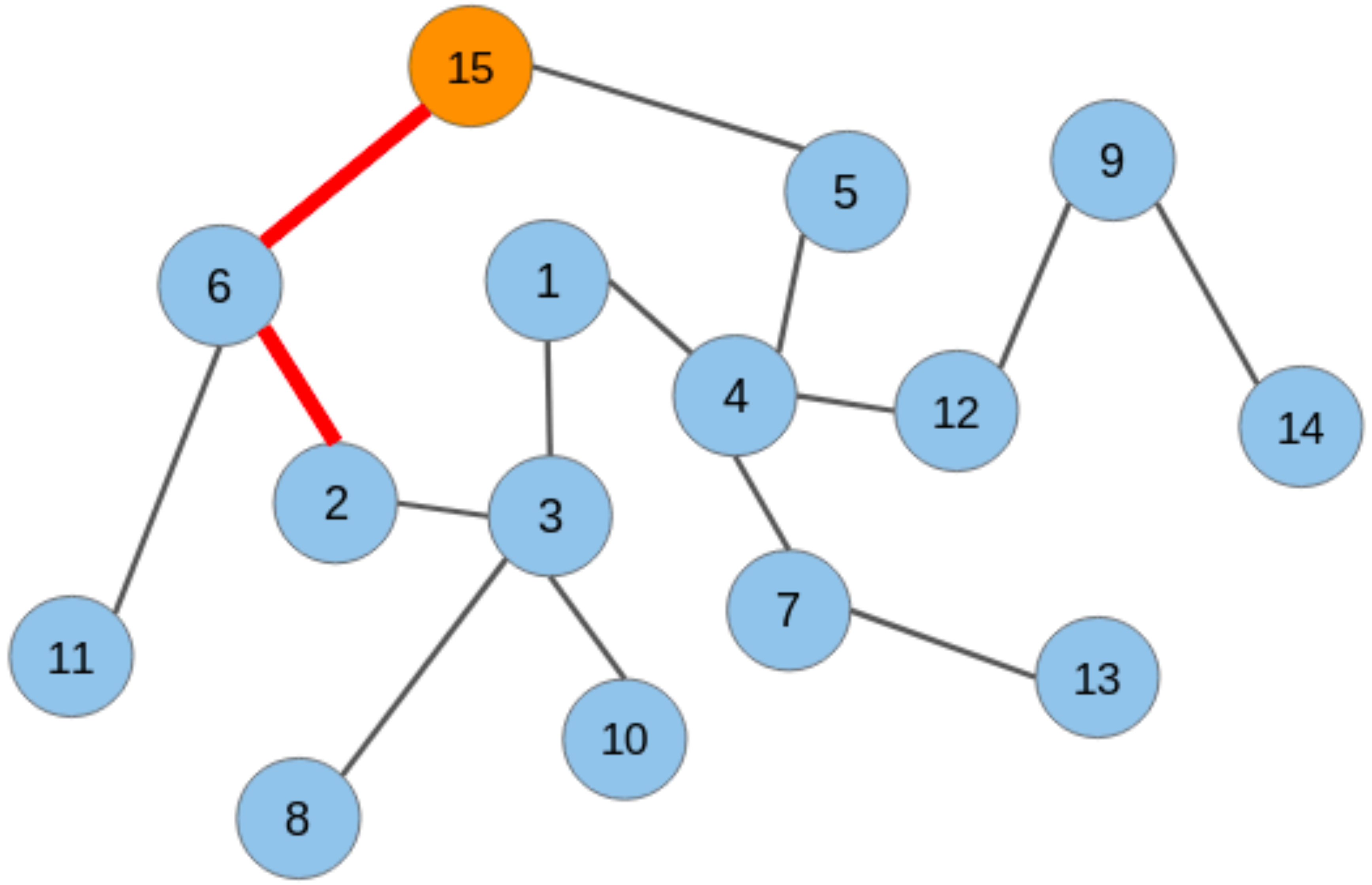
- The goal is to learn $X_E \in \mathbb{R}^{|V| \times d}$, where d is small number of latent dim.

$$G = (V, E)$$
$$G_L = (V, E, X, Y)$$
$$X \in \mathbb{R}^{|V| \times S}$$
$$Y \in \mathbb{R}^{|V| \times |y|}$$

Learning Social Representations

- Adaptability - new social relations should not require repeating the learning
- Community aware - The distance between latent dimensions should represent a metric for evaluating social similarity.
- Low dimensional - when labeled data is scarce, low-dim. models generalize better, and speed up convergence
- Continuous - We require latent representations to model partial community membership in continuous space

Learning Social Representations
- Random Walks

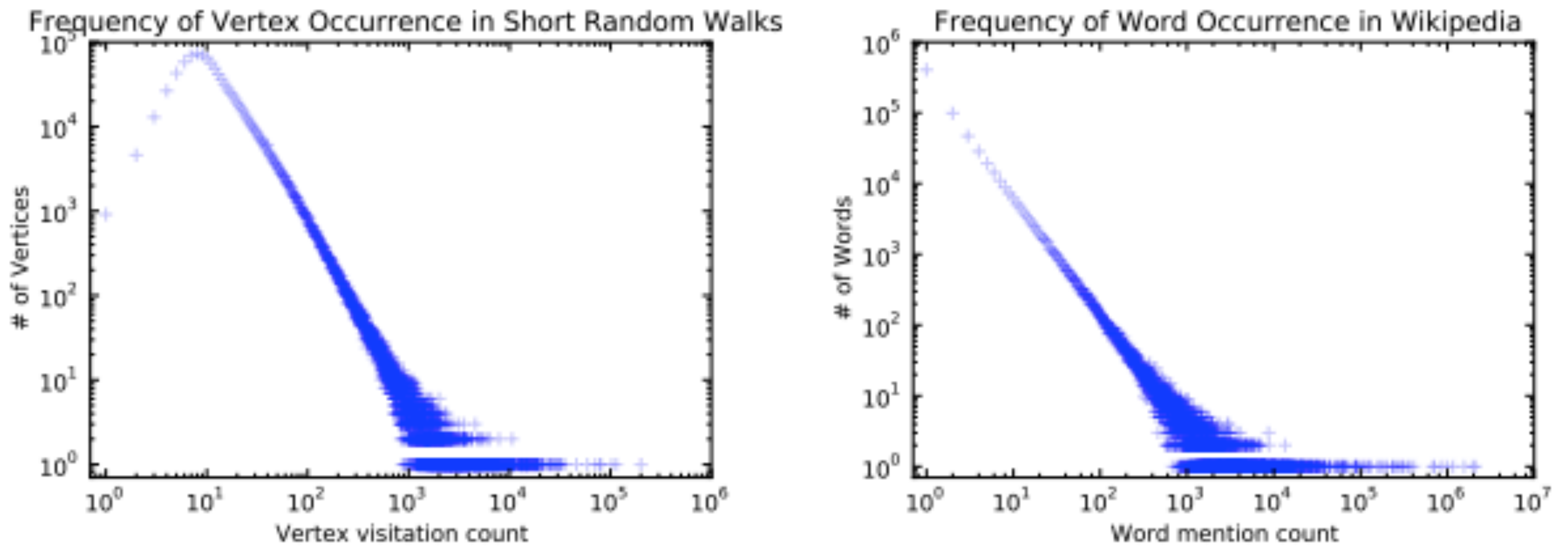


- Use a stream of short random walks as our basic tool for extracting information from a network.
- Local exploration is easy to parallelize
- It possible to accommodate small changes in the graph structure without the need for global re-computation

Learning Social Representations
- Connection : Power laws

- If the degree distribution of a connected graph follows a power law, we observe that the frequency which vertices appear in the short random walks will also follow a power-law distribution
- Word frequency in nature language follows a similar distribution.

A core contribution of our work is the idea that techniques which have been used to model natural language (where the symbol frequency follows a power law distribution (or *Zipf's law*)) can be re-purposed to model community structure in networks.



(a) YouTube Social Graph (b) Wikipedia Article Text

Figure 2: The power-law distribution of vertices appearing in short random walks (2a) follows a power-law, much like the distribution of words in natural language (2b).

Learning Social Representations
- Language Modeling

- The goal of language modeling is estimate the likelihood of a specific sequence of words appearing in a corpus.

$$W_1^n = (w_0, w_1, \dots, w_n)$$

$$w_i \in V (\text{V is the vocabulary})$$

$$\Pr(w_n | w_0, w_1, \dots, w_{n-1})$$

Generalization

$$\Pr(v_i | (v_1, v_2, \dots, v_{i-1}))$$

Learning Social Representations
- Language Modeling

- Our goal is to learn a latent representation, not only a probability distribution of node co-occurrences.

$$\Pr(v_i | (v_1, v_2, \ldots, v_{i-1}))$$

$$\Big\downarrow \text{Mapping function}$$

$$\Phi : v \in V \mapsto \mathbb{R}^{|V| \times d}$$

$$\Pr(v_i | (\Phi(v_1), \Phi(v_2), \ldots, \Phi(v_{i-1})))$$

Learning Social Representations
- Language Modeling

- Optimization problem:

$$\underset{\Phi}{\text{minimize}} \;\; -\,log\Pr(\{v_{i-w},\ldots,v_{i-1},v_{i+1},\ldots,v_{i+w}\}|\Phi(v_i))$$

DeepWalk

Algorithm 1 DeepWalk(G, w, d, γ, t)

Input: graph $G(V, E)$
window size w
embedding size d
walks per vertex γ
walk length t
Output: matrix of vertex representations $\Phi \in \mathbb{R}^{|V| \times d}$

Initialization: Sample Φ from $\mathcal{U}^{|V| \times d}$
Build a binary Tree T from V
for $i = 0$ to γ **do**
  $\mathcal{O} = \text{Shuffle}(V)$
  for each $v_i \in \mathcal{O}$ **do**
    $\mathcal{W}_{v_i} = RandomWalk(G, v_i, \text{t})$
    SkipGram(Φ, $\mathcal{W}_{v_i}$, w)
  end for
end for

Algorithm 2 SkipGram(Φ, $\mathcal{W}_{v_i}$, w)

for each $v_j \in \mathcal{W}_{v_i}$ **do**
  for each $u_k \in \mathcal{W}_{v_i}[j - w : j + w]$ **do**
    $J(\Phi) = -\log \Pr(u_k \mid \Phi(v_j))$
    $\Phi = \Phi - \alpha * \frac{\partial J}{\partial \Phi}$
  end for
end for

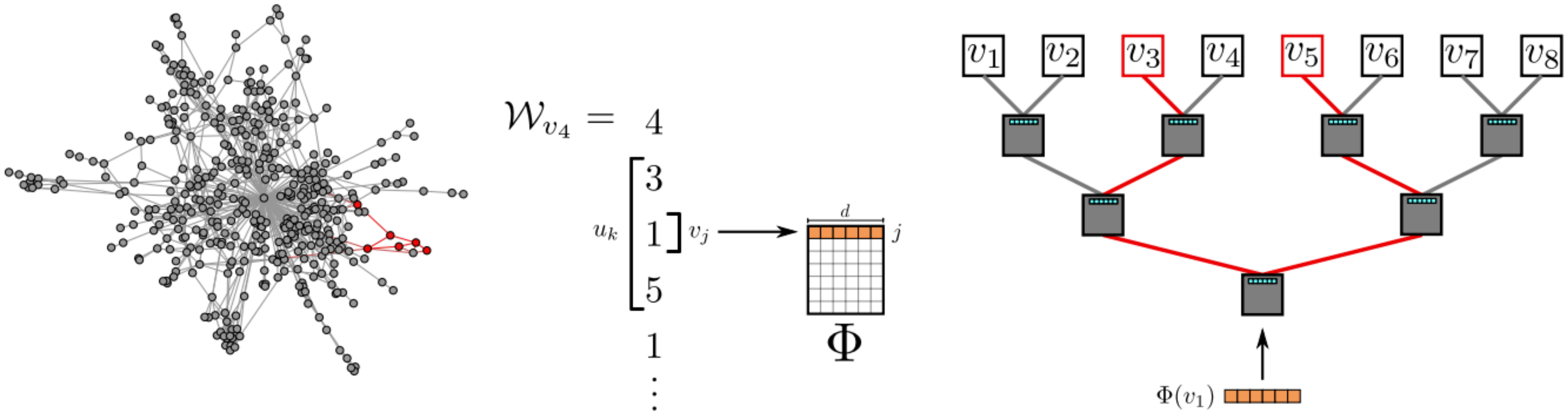
DeepWalk

$$\Pr_{\Phi}(u|\Phi(v_i)) := \frac{exp(\Phi(u)^T \Phi(v_i))}{\sum_{w \in V} exp(\Phi(w)^T \Phi(v_i))}$$

$$\Pr(W_{v_i}|\Phi(v_i)) := \prod_{u \in W_{v_i}} \Pr_{\Phi}(u|\Phi(v_i))$$

엄밀하게는 window 범위 안에 있는 nodes

DeepWalk



(a) Random walk generation. (b) Representation mapping. (c) Hierarchical Softmax.

Figure 3: Overview of DeepWalk. We slide a window of length $2w + 1$ over the random walk $\mathcal{W}_{v_4}$, mapping the central vertex v_1 to its representation $\Phi(v_1)$. Hierarchical Softmax factors out $\Pr(v_3 \mid \Phi(v_1))$ and $\Pr(v_5 \mid \Phi(v_1))$ over sequences of probability distributions corresponding to the paths starting at the root and ending at v_3 and v_5. The representation Φ is updated to maximize the probability of v_1 co-occurring with its context $\{v_3, v_5\}$.

Node2Vec

Deepdog 08.25.21

Related work

- Skip-gram model의 방법을 graph structure에 적용하려 함.
  ordered sequence of words => ordered sequence of nodes
- There are many possible sampling strategies for nodes
  => there is no clear winning sampling strategy that works across all networks and all prediction tasks.
- node2vec overcomes this limitation by designing a flexible objective that is not tied to a particular sampling strategy and provides parameters to tune the explored search space

Feature Learning Framework

- $G = (V, E)$

- $f : V -> \mathbb{R}^d$: mapping function from nodes to feature representations
  f is a matrix of size $|V| \times d$ parameters.

- $N_s(u) \subset V$: network neighborhood of node u
  S is a neighborhood sampling strategy

Feature Learning Framework
- Object function

$$\max_{f} \sum_{u \in V} \log Pr(N_s(u) \,|\, f(u)) \,.$$

In order to make the optimization problem tractable, we make two standard assumptions:

- Conditional independence

$$Pr(N_S(u) \,|\, f(u)) = \prod_{n_i \in N_S(u)} Pr(n_i \,|\, f(u)) \,.$$

- Symmetry in feature space

  A source node and neighborhood node have a symmetric effect over each other in feature space.

$$Pr(n_i \,|\, f(u)) = \frac{exp(f(n_i) \cdot f(n))}{\sum_{(u \in V)} exp(f(v) \cdot f(u))}$$

Feature Learning Framework
- Object function

$$\max_f \sum_{u \in V} log Pr(N_s(u) | f(u)) .$$

$$\downarrow$$

$$\max_f \sum_{u \in V} [-log Z_u + \sum_{n_i \in N_S(u)} f(n_i) \cdot f(u)] .$$

$$Z_u = \sum_{u \in V} exp(f(u) \cdot (fv))$$

$$Pr(N_S(u) | f(u)) = \prod_{n_i \in N_S(u)} Pr(n_i | f(u)) .$$

$$Pr(n_i | f(u)) = \frac{exp(f(n_i) \cdot f(n))}{\sum_{(u \in V)} exp(f(v) \cdot f(u))}$$

Feature Learning Framework
- Search strategies

- Text는 linear해서 sliding window를 사용할 수 있다.
- Network는 linear하지 않기 때문에 이웃 노드들에 대한 풍부한 notion이 필요하다.
- Source node u에 대해서 랜덤하게 다양한 neighborhoods를 sampling할 수 있는 방법을 소개한다.
- $N_S(u)$ 는 sampling strategy S에 따라 다른 구조를 가지게 된다.

Feature Learning Framework
- Classic search strategies

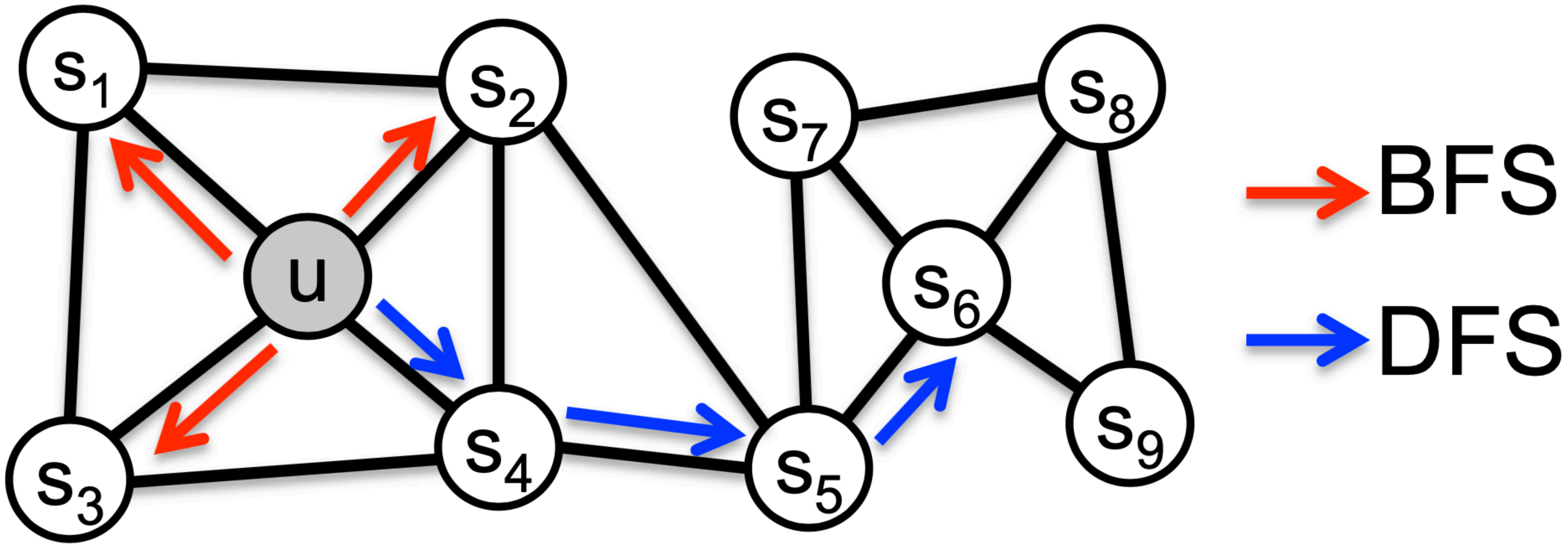


Figure 1: BFS and DFS search strategies from node u ($k = 3$).

Feature Learning Framework
- Classic search strategies

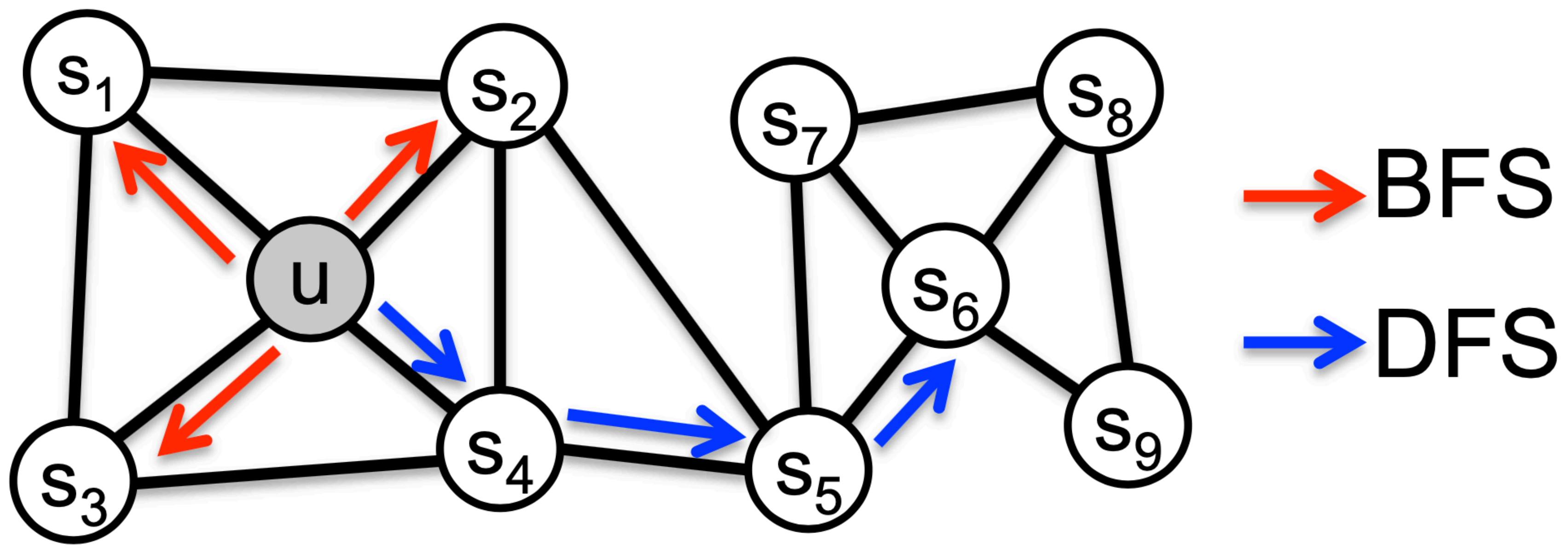


Figure 1: BFS and DFS search strategies from node u ($k = 3$).

- Breadth-first Sampling : 근접한 neighbors로 제한된다.
- Depth-first Sampling : sequentially sampled 된다.

- Homophile hypothesis
  nodes that are highly interconnected and belong to similar network clusters or communities should be embedded closely together(e.g., nodes s1 and u)
- Structural equivalence hypothesis
  nodes that have similar structural roles in networks should be embedded closely together (e.g., nodes u and s6)
- Importantly, unlike homophile, structural equivalence does not emphasize connectivity; nodes could be far apart in the network and still have the same structural role.

Feature Learning Framework
- node2vec

$$P(c_i = x \mid c_{i-1} = v) = \begin{cases} \frac{\pi_{vx}}{Z} & \text{if (v,x)} \in E \\ 0 & \text{otherwize} \end{cases}$$

- π_{vx} is the unnormalized transition probability between nodes v and x
- Z is the normalizing constant
- $\pi_{vx} = w_{vx}$ and Z : Whole the weight of the edges

Feature Learning Framework
- Search bias α

- 2^{nd} order random walk with two parameters p and q
- t에서 시작해서 egde(t, v) 를 지나 v에 도착한 상황을 가정.
- v에서 next step을 위한 edge를 선택해야함.
- The unnormalized transition probability to

$\pi_{vx} = \alpha_{pq}(t, x) \cdot w_{vx}$ where

$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0 \quad (t,t) \\ 1 & \text{if } d_{tx} = 1 \quad (t, x_1) \\ \frac{1}{q} & \text{if } d_{tx} = 2 \quad (t, x_2), (t, x_3) \end{cases}$$

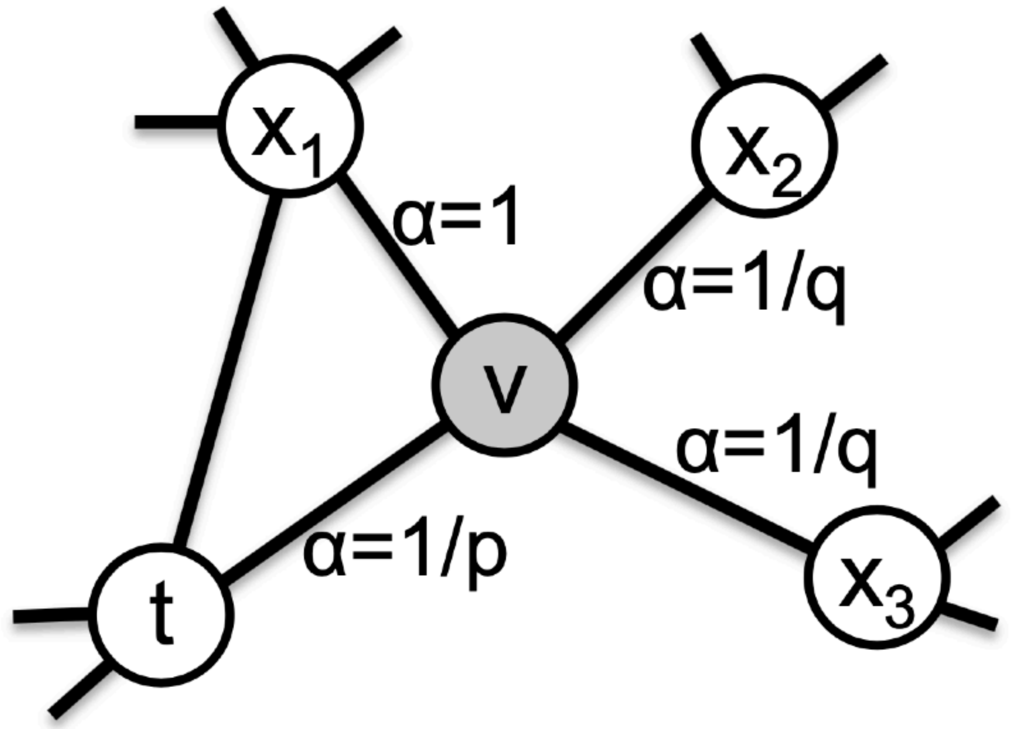
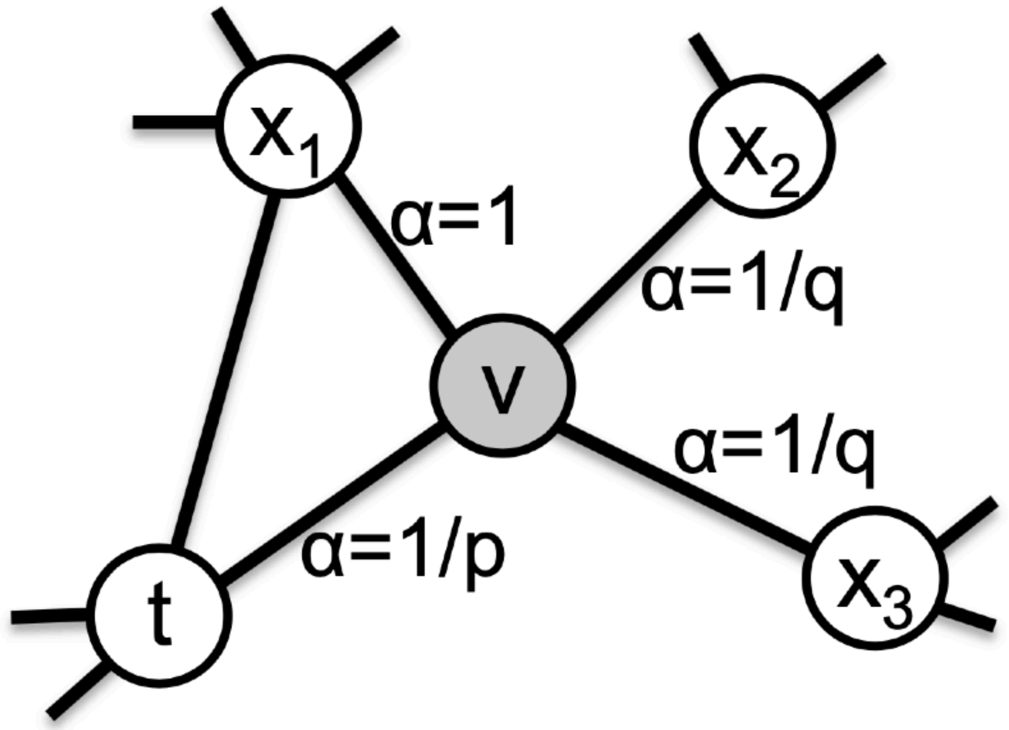


Figure 2: Illustration of the random walk procedure in *node2vec*. The walk just transitioned from t to v and is now evaluating its next step out of node v. Edge labels indicate search biases α.

d_{tx} denotes the shortest path distance between nodes t and x.

$$P(c_i = x \mid c_{i-1} = v) = \begin{cases} \frac{\pi_{vx}}{Z} & \text{if } (v,x) \in E \\ 0 & \text{otherwize} \end{cases}$$

Feature Learning Framework
- Search bias α



- Return parameter, p
  p controls the likelihood of immediately revisiting a node in the walk.
  (> max(q,1)) : 2-hop redundancy in sampling을 피하고 이미 방문한 노드는 가능한 다시 방문 하지 않으려 함.
  (< min(q,1)) : 다시 돌아가게 하려는 경향이 커지고 이는 계속해서 starting node u의 'local' 주변을 탐색하게 함

- In-out parameter, q
  if q > 1, the random walk is biased towards nodes close to node t.

Feature Learning Framework
- The node2vec algorithm

Algorithm 1 The *node2vec* algorithm.

LearnFeatures (Graph $G = (V, E, W)$, Dimensions d, Walks per node r, Walk length l, Context size k, Return p, In-out q)
    π = PreprocessModifiedWeights(G, p, q)
    $G' = (V, E, \pi)$
    Initialize $walks$ to Empty
    for $iter = 1$ **to** r **do**
        for all nodes $u \in V$ **do**
            $walk$ = node2vecWalk(G', u, l)
            Append $walk$ to $walks$
    f = StochasticGradientDescent(k, d, $walks$)
    return f

Start node u의 선택에 있어서 bias가 존재함
=> 모든 node를 다 representations 해야하기 때문에,
모든 nodes에 대하여 r 번 길이 l의 random walk를 수행.

node2vecWalk (Graph $G' = (V, E, \pi)$, Start node u, Length l)
    Inititalize $walk$ to $[u]$
    for $walk\_iter = 1$ **to** l **do**
        $curr = walk[-1]$
        V_{curr} = GetNeighbors($curr$, G')
        s = AliasSample(V_{curr}, π)
        Append s to $walk$
    return $walk$

```
model = node2vec(edge_index=data.edge_index,
                 embedding_dim=128,
                 walk_length=20, # l
                 context_size=10, # k
                 walks_per_node=20, # r
                 num_negative_samples=1,
                 p=200, q=1,
                 sparse=True).to(device)
```

```
loader = model.loader(batch_size=128, shuffle=True, num_workers=2)
```

```python
def loader(self, **kwargs):
    return DataLoader(range(self.adj.sparse_size(0)),
                      collate_fn=self.sample, **kwargs)
```

```python
def sample(self, batch):
    if not isinstance(batch, torch.Tensor):
        batch = torch.tensor(batch)
    return self.pos_sample(batch), self.neg_sample(batch)
```

```python
def pos_sample(self, batch):
    batch = batch.repeat(self.walks_per_node)
    rowptr, col, _ = self.adj.csr()
    rw = random_walk(rowptr, col, batch, self.walk_length, self.p, self.q)
    if not isinstance(rw, torch.Tensor):
        rw = rw[0]

    walks = []
    num_walks_per_rw = 1 + self.walk_length + 1 - self.context_size
    for j in range(num_walks_per_rw):
        walks.append(rw[:, j:j + self.context_size])
    return torch.cat(walks, dim=0)
```

```python
def neg_sample(self, batch):
    batch = batch.repeat(self.walks_per_node * self.num_negative_samples)

    rw = torch.randint(self.adj.sparse_size(0),
                       (batch.size(0), self.walk_length))
    rw = torch.cat([batch.view(-1, 1), rw], dim=-1)

    walks = []
    num_walks_per_rw = 1 + self.walk_length + 1 - self.context_size
    for j in range(num_walks_per_rw):
        walks.append(rw[:, j:j + self.context_size])
    return torch.cat(walks, dim=0)
```