研究了一下 struc2vec 相关论文：

对于捕获网络中节点的结构身份的学习表示，一个有效的方法应具备：

节点的潜在表征（ latent representation）之间的距离应该与节点的结构相似性密切相关。具有相同局域网结构的两个节点应该具有相同的潜在表示而具有不同结构身份的节点应该相距很远（这里的距离是指节点 latent representation 之间的距离）潜在表示不应该依赖于任何节点或边缘属性，包括节点标签。因此，在结构上相似的节点应该有紧密的潜在表示，独立于节点和边缘的属性。

节点的结构身份必须独立于其在网络中的“位置”

struct2vec 框架的四个步骤：

1.对于不同的邻域大小，确定图中每个顶点对之间的结构相似性：

在度量节点之间的结构相似度时引入了层次，提供了更多的信息来评估层次的每一层的结构相似度。

2.构造一个加权多层图：

网络中的所有节点都存在于每一层，每一层对应于层次的一个层次，用于度量结构相似性每层内每个节点对之间的边权值与它们的结构相似度成反比

3. 使用多层图为每个节点生成上下文：

在多层图上使用有偏随机游走，用来生成节点序列结构上更相似的节点更大概率会在这些序列中

4. 应用一种技术，从节点序列给出的上下文学习潜在表示，例如 SkipGram：

和 deepwalk 一样，利用这些序列进行学习

struct2vec 是相当灵活的，因为它不要求任何特定的结构相似性测量或表征学习框架

总结：

struc2vec 模型完全是利用图的结构信息训练节点的向量表示，没有利用节点的任何标签信息。在实际运用中这可以是一个优化方案。

相对于 DeepWalk 和 Node2Vec 来说，Struc2Vec 是从另一个角度出发构造节点序列。更加侧重节点的同构信息。

附上源代码：

```
class Struc2Vec():
    def __init__(self, graph, walk_length=10, num_walks=100, workers=1, verbose=0, stay_prob=0.3, opt1_reduce_len=True, opt2_reduce_sim_calc=True, opt3_num_layers=None, temp_path='./temp_struc2vec/', reuse=False):
        self.graph = graph
        self.idx2node, self.node2idx = preprocess_nxgraph(graph)
        self.idx = list(range(len(self.idx2node)))

        self.opt1_reduce_len = opt1_reduce_len
        self.opt2_reduce_sim_calc = opt2_reduce_sim_calc
        self.opt3_num_layers = opt3_num_layers
```

```
        self.resue = reuse
        self.temp_path = temp_path

        if not os.path.exists(self.temp_path):
            os.mkdir(self.temp_path)
        if not reuse:
            shutil.rmtree(self.temp_path)
            os.mkdir(self.temp_path)

        self.create_context_graph(self.opt3_num_layers, workers, verbose)
        self.prepare_biased_walk()
        self.walker = BiasedWalker(self.idx2node, self.temp_path)
        self.sentences = self.walker.simulate_walks(
            num_walks, walk_length, stay_prob, workers, verbose)

        self._embeddings = {}

    def create_context_graph(self, max_num_layers, workers=1, verbose=0,):

        pair_distances = self._compute_structural_distance(
            max_num_layers, workers, verbose,)
        layers_adj, layers_distances = self._get_layer_rep(pair_distances)
        pd.to_pickle(layers_adj, self.temp_path + 'layers_adj.pkl')

        layers_accept, layers_alias = self._get_transition_probs(
            layers_adj, layers_distances)
        pd.to_pickle(layers_alias, self.temp_path + 'layers_alias.pkl')
        pd.to_pickle(layers_accept, self.temp_path + 'layers_accept.pkl')

    def prepare_biased_walk(self,):

        sum_weights = {}
        sum_edges = {}
        average_weight = {}
        gamma = {}
        layer = 0
        while (os.path.exists(self.temp_path+'norm_weights_distance-layer-' +
str(layer)+'.pkl')):
            probs = pd.read_pickle(
                self.temp_path+'norm_weights_distance-layer-' + str(layer)+'.pkl')
            for v, list_weights in probs.items():
                sum_weights.setdefault(layer, 0)
                sum_edges.setdefault(layer, 0)
```

```
                    sum_weights[layer] += sum(list_weights)
                    sum_edges[layer] += len(list_weights)

                average_weight[layer] = sum_weights[layer] / sum_edges[layer]

                gamma.setdefault(layer, {})

                for v, list_weights in probs.items():
                    num_neighbours = 0
                    for w in list_weights:
                        if (w > average_weight[layer]):
                            num_neighbours += 1
                    gamma[layer][v] = num_neighbours

                layer += 1

            pd.to_pickle(average_weight, self.temp_path + 'average_weight')
            pd.to_pickle(gamma, self.temp_path + 'gamma.pkl')

        def train(self, embed_size=128, window_size=5, workers=3, iter=5):

            # pd.read_pickle(self.temp_path+'walks.pkl')
            sentences = self.sentences

            print("Learning representation...")
            model = Word2Vec(sentences, size=embed_size, window=window_size, min_count=0,
hs=1, sg=1, workers=workers,
                             iter=iter)
            print("Learning representation done!")
            self.w2v_model = model

            return model

        def get_embeddings(self,):
            if self.w2v_model is None:
                print("model not train")
                return {}

            self._embeddings = {}
            for word in self.graph.nodes():
                self._embeddings[word] = self.w2v_model.wv[word]

            return self._embeddings
```

```
def _compute_ordered_degreelist(self, max_num_layers):

    degreeList = {}
    vertices = self.idx    # self.g.nodes()
    for v in vertices:
        degreeList[v] = self._get_order_degreelist_node(v, max_num_layers)
    return degreeList

def _get_order_degreelist_node(self, root, max_num_layers=None):
    if max_num_layers is None:
        max_num_layers = float('inf')

    ordered_degree_sequence_dict = {}
    visited = [False] * len(self.graph.nodes())
    queue = deque()
    level = 0
    queue.append(root)
    visited[root] = True

    while (len(queue) > 0 and level <= max_num_layers):

        count = len(queue)
        if self.opt1_reduce_len:
            degree_list = {}
        else:
            degree_list = []
        while (count > 0):

            top = queue.popleft()
            node = self.idx2node[top]
            degree = len(self.graph[node])

            if self.opt1_reduce_len:
                degree_list[degree] = degree_list.get(degree, 0) + 1
            else:
                degree_list.append(degree)

            for nei in self.graph[node]:
                nei_idx = self.node2idx[nei]
                if not visited[nei_idx]:
                    visited[nei_idx] = True
                    queue.append(nei_idx)
            count -= 1
        if self.opt1_reduce_len:
```

```
                orderd_degree_list = [(degree, freq)
                                      for degree, freq in degree_list.items()]
                orderd_degree_list.sort(key=lambda x: x[0])
            else:
                orderd_degree_list = sorted(degree_list)
            ordered_degree_sequence_dict[level] = orderd_degree_list
            level += 1

        return ordered_degree_sequence_dict

def _compute_structural_distance(self, max_num_layers, workers=1, verbose=0,):

        if os.path.exists(self.temp_path+'structural_dist.pkl'):
            structural_dist = pd.read_pickle(
                self.temp_path+'structural_dist.pkl')
        else:
            if self.opt1_reduce_len:
                dist_func = cost_max
            else:
                dist_func = cost

            if os.path.exists(self.temp_path + 'degreelist.pkl'):
                degreeList = pd.read_pickle(self.temp_path + 'degreelist.pkl')
            else:
                degreeList = self._compute_ordered_degreelist(max_num_layers)
                pd.to_pickle(degreeList, self.temp_path + 'degreelist.pkl')

            if self.opt2_reduce_sim_calc:
                degrees = self._create_vectors()
                degreeListsSelected = {}
                vertices = {}
                n_nodes = len(self.idx)
                for v in self.idx:    # c:list of vertex
                    nbs = get_vertices(
                        v, len(self.graph[self.idx2node[v]]), degrees, n_nodes)
                    vertices[v] = nbs    # store nbs
                    degreeListsSelected[v] = degreeList[v]    # store dist
                    for n in nbs:
                        # store dist of nbs
                        degreeListsSelected[n] = degreeList[n]
            else:
                vertices = {}
                for v in degreeList:
                    vertices[v] = [vd for vd in degreeList.keys() if vd > v]
```

```
                results = Parallel(n_jobs=workers, verbose=verbose,)(
                    delayed(compute_dtw_dist)(part_list, degreeList, dist_func) for part_list in
partition_dict(vertices, workers))
                dtw_dist = dict(ChainMap(*results))

                structural_dist = convert_dtw_struc_dist(dtw_dist)
                pd.to_pickle(structural_dist, self.temp_path +
                             'structural_dist.pkl')

            return structural_dist

        def _create_vectors(self):
            degrees = {}    # sotre v list of degree
            degrees_sorted = set()    # store degree
            G = self.graph
            for v in self.idx:
                degree = len(G[self.idx2node[v]])
                degrees_sorted.add(degree)
                if (degree not in degrees):
                    degrees[degree] = {}
                    degrees[degree]['vertices'] = []
                degrees[degree]['vertices'].append(v)
            degrees_sorted = np.array(list(degrees_sorted), dtype='int')
            degrees_sorted = np.sort(degrees_sorted)

            l = len(degrees_sorted)
            for index, degree in enumerate(degrees_sorted):
                if (index > 0):
                    degrees[degree]['before'] = degrees_sorted[index - 1]
                if (index < (l - 1)):
                    degrees[degree]['after'] = degrees_sorted[index + 1]

            return degrees

        def _get_layer_rep(self, pair_distances):
            layer_distances = {}
            layer_adj = {}
            for v_pair, layer_dist in pair_distances.items():
                for layer, distance in layer_dist.items():
                    vx = v_pair[0]
                    vy = v_pair[1]

                    layer_distances.setdefault(layer, {})
```

```
                    layer_distances[layer][vx, vy] = distance

                    layer_adj.setdefault(layer, {})
                    layer_adj[layer].setdefault(vx, [])
                    layer_adj[layer].setdefault(vy, [])
                    layer_adj[layer][vx].append(vy)
                    layer_adj[layer][vy].append(vx)

        return layer_adj, layer_distances

    def _get_transition_probs(self, layers_adj, layers_distances):
        layers_alias = {}
        layers_accept = {}

        for layer in layers_adj:

            neighbors = layers_adj[layer]
            layer_distances = layers_distances[layer]
            node_alias_dict = {}
            node_accept_dict = {}
            norm_weights = {}

            for v, neighbors in neighbors.items():
                e_list = []
                sum_w = 0.0

                for n in neighbors:
                    if (v, n) in layer_distances:
                        wd = layer_distances[v, n]
                    else:
                        wd = layer_distances[n, v]
                    w = np.exp(-float(wd))
                    e_list.append(w)
                    sum_w += w

                e_list = [x / sum_w for x in e_list]
                norm_weights[v] = e_list
                accept, alias = create_alias_table(e_list)
                node_alias_dict[v] = alias
                node_accept_dict[v] = accept

            pd.to_pickle(
                norm_weights, self.temp_path + 'norm_weights_distance-layer-' +
str(layer)+'.pkl')
```

```
                layers_alias[layer] = node_alias_dict
                layers_accept[layer] = node_accept_dict

        return layers_accept, layers_alias


def cost(a, b):
    ep = 0.5
    m = max(a, b) + ep
    mi = min(a, b) + ep
    return ((m / mi) - 1)


def cost_min(a, b):
    ep = 0.5
    m = max(a[0], b[0]) + ep
    mi = min(a[0], b[0]) + ep
    return ((m / mi) - 1) * min(a[1], b[1])


def cost_max(a, b):
    ep = 0.5
    m = max(a[0], b[0]) + ep
    mi = min(a[0], b[0]) + ep
    return ((m / mi) - 1) * max(a[1], b[1])


def convert_dtw_struc_dist(distances, startLayer=1):
    """

    :param distances: dict of dict
    :param startLayer:
    :return:
    """
    for vertices, layers in distances.items():
        keys_layers = sorted(layers.keys())
        startLayer = min(len(keys_layers), startLayer)
        for layer in range(0, startLayer):
            keys_layers.pop(0)

        for layer in keys_layers:
            layers[layer] += layers[layer - 1]
    return distances
```

```
def get_vertices(v, degree_v, degrees, n_nodes):
    a_vertices_selected = 2 * math.log(n_nodes, 2)
    vertices = []
    try:
        c_v = 0

        for v2 in degrees[degree_v]['vertices']:
            if (v != v2):
                vertices.append(v2)    # same degree
                c_v += 1
                if (c_v > a_vertices_selected):
                    raise StopIteration

        if ('before' not in degrees[degree_v]):
            degree_b = -1
        else:
            degree_b = degrees[degree_v]['before']
        if ('after' not in degrees[degree_v]):
            degree_a = -1
        else:
            degree_a = degrees[degree_v]['after']
        if (degree_b == -1 and degree_a == -1):
            raise StopIteration    # not anymore v
        degree_now = verifyDegrees(degrees, degree_v, degree_a, degree_b)
        # nearest valid degree
        while True:
            for v2 in degrees[degree_now]['vertices']:
                if (v != v2):
                    vertices.append(v2)
                    c_v += 1
                    if (c_v > a_vertices_selected):
                        raise StopIteration

            if (degree_now == degree_b):
                if ('before' not in degrees[degree_b]):
                    degree_b = -1
                else:
                    degree_b = degrees[degree_b]['before']
            else:
                if ('after' not in degrees[degree_a]):
                    degree_a = -1
                else:
```

```
                    degree_a = degrees[degree_a]['after']

            if (degree_b == -1 and degree_a == -1):
                raise StopIteration

            degree_now = verifyDegrees(degrees, degree_v, degree_a, degree_b)

    except StopIteration:
        return list(vertices)

    return list(vertices)


def verifyDegrees(degrees, degree_v_root, degree_a, degree_b):

    if(degree_b == -1):
        degree_now = degree_a
    elif(degree_a == -1):
        degree_now = degree_b
    elif(abs(degree_b - degree_v_root) < abs(degree_a - degree_v_root)):
        degree_now = degree_b
    else:
        degree_now = degree_a

    return degree_now


def compute_dtw_dist(part_list, degreeList, dist_func):
    dtw_dist = {}
    for v1, nbs in part_list:
        lists_v1 = degreeList[v1]    # lists_v1 :orderd degree list of v1
        for v2 in nbs:
            lists_v2 = degreeList[v2]    # lists_v1 :orderd degree list of v2
            max_layer = min(len(lists_v1), len(lists_v2))    # valid layer
            dtw_dist[v1, v2] = {}
            for layer in range(0, max_layer):
                dist, path = fastdtw(
                    lists_v1[layer], lists_v2[layer], radius=1, dist=dist_func)
                dtw_dist[v1, v2][layer] = dist
    return dtw_dist
```