算法设计与分析-作业 1

--- Random-walk Domination In Large Graphs

学号： **1130310128**

姓名： 杨尚斌

哈工大计算机科学与技术学院

一、关注的问题

大图两种随机游走支配集问题的生成

1.在 L 随机游走中从其他节点开始向目标节点 k 的命中中总命中时间是最小的

2.在 L 的随机游走中找到 k 节点最大限度的打击任何一个目标节点的预期数量

二、解决思想

动态规划（DP）+ 贪心（Greedy）+随机取样物化技术 -> 近似贪婪算法

一般来说，基于 DP 的贪心由于昂贵的边缘收益导致不是非常有效。在这篇文章中，提出了一个基于随机取样技术上的，精心设计的一个近似贪心算法，拥有线性的时间复杂度，并且在图的空间大小上也保证了最佳性。

三、论文概览介绍

1. 算法应用场地

1. 在社交网络中项目安放位置问题（Item-placement Problem in online social networks）

在人们的日常社交网络活动中，人们通过浏览网页获得信息。而作为一个用户，去发现其他的信息是需要靠自己的社会联系来完成的。比如在一个照片分享的网站上，用户可以在他的主页上看见他的朋友的照片，一旦用户抵达他的朋友的主页上，他就能够在他的

朋友的主页上看到他的朋友的照片。换句话说，下一次要访问的主页完全取决了现在所访问的主页。因此，用户的访问网站过程可以认为是一个在社交网络上的随机游走问题，在这个问题的背后，又有一个比较隐含的限制就是时间上的限制，即用户不可能有无限的时间去浏览朋友的主页或者说是朋友的朋友的网页。

于是，我们就可以通过每个用户在社交网络中可以访问 L 多的主页地址建立一个 L 随机游走的模型。基于用户浏览网页的行为出现了两个比较有趣的问题：1.如何去放置这些东西才能让其他的用户比较董怡的通过浏览网页发现他们 2. 如何去放置这些东西才能更多的用户可以找到他们。

举个更加具体的例子，在 facebook 中，有个 APP 的开发者想通过 facebook 让更多的人使用上他的应用，于是，他需要选择 k 个人去安装他的应用，如过有人安装了他的应用，那么这个人在 facebook 上的朋友就可以子啊 facebook 上看他安装这个应用的消息。因此，最开始摆在面前的问题就是如何去选择这 k 个人的问题，即如何去选择这 k 个人才能让更多的人去看到这个应用。我们也可以就这个问题建立一个 L 随机游走的模型。

2. 网络中优化广告位置问题（Optimizing Ads-placement in advertisement networks）

和前面的例子类似，如何把广告投递给普通人（会支付一些报酬）能让更多的人看到这个广告。我们可以就这个“用户-信息-发现”这一个广告传播过程建立一个 L 随机游走问题模型

3. 在 P2P 网络中资源的加速搜寻问题(Accelerating resources search in P2P networks)

在 p2p 网络中，如何去安置资源在一小部分人上才能让其他的人也快速的找到资源通过这种预先指定的策略。在 P2P 网络中，普通用户搜索资源的这种策略是基于随机游走的。于此同时，更为重要的是，对于每个用户所下载的资源，都应该有一个时间使用期，即在这个时间之内是要向别人去分享资源的，超出这个时间之后就不再继续提供资源。因此，我们也就可以以这个 p2p 资源搜索问题建立一个 L 随机游走的模型。同时，运用这篇文章中所提到的方法能更加容易的优化这种策略。

2.本文提到算法的主要贡献

这个算法构造出两种分别基于两种离散最优化子结构的随机游走支配集问题。证明了这是基数约制模型集合函数下最大问题的实例。一般来说，这是一个 NP 难的问题。在文章中，采取了发展中的近似算法，比较有效的解决了这个问题。

在数据上，我们所知道的基于 DP 的贪心达到了 0.63 的近似估算，是时间复杂度上是以 O（n^3）的形式，这样的话，这种算法只能用于比较小的图形中。

而对于文中的算法，是基于精心设计的随机取样物化技术的一种近似算法，在时间和空间上都是线性的，最后可以达到 0.63-x 的近似估算（x 是一个很小的常熟）。

最后经过验证，发现这种近似贪婪算法线性扩展于图的大小。

四、基本算法的描述

前提：G（V,E）,即对于图 G,有 V 个顶点 E 条边，属于无向无权图（后续算法也非常容易向有向有权的图进行扩展，为讨论简单起见，取最简单的无向无权图进行算法的相关说明）

1.几个定理

给定一个无向无权图 G 和一个起始节点 u,随机游走挑选一个邻居节点并且移动到这个邻居节点，然后就按照这种方式进行递归。因此，在这种工作中，我们可以把精力集中在一个称作是 L 随机游走的通用游走模型，在这项工作中，随机游走路径的长度是由一个非负整数所控制的。

传统的随机游走是通过设置参数 L 来限制无限随机游走的特殊情况。

在文章中，定义了一个重要的概念**击中时间**（hitting time）：在源和目标节点中击中时间就是从源节点到目标节点的 L 随机游走中第一个跳的预期数量；记 z(t, u)就是 u 节点随机游走在离散时间 t 时候的位置；让 T（L, u, v）这个随机变量定义为 T（L, u, v） = min{min{t: Z(t, u) = v, t >= 0}, L}。

这样的话在 u 和 v 之间的击中时间定义为 h_{uv}^{L}，他也就是 T（L, u, v）的期望值.因此就有下面的几个定理：

1.对任意的节点 u 和 v，击中时间 h(L, u, v)都是由 L 限制的，即 h_{uv}^{L} = E[T(L, u, v)] <= L

2.定义 d（u）为节点 u 的度，N(u)是 u 的邻居节点的集合，p(u, w) = 1 / （ d(u) ）为 u 到 N（u）中 w 节点的可能性，如果 N 中不存在 w，就定义 p(u, w) = 0。即 h_{uv}^{L} = {0, u = v; 1+}，即：

$$h_{uv}^{L} = \begin{cases} 0, u=v \\ 1+\sum_{w \in V} p_{vw} h_{wv}^{L-1}, u \neq v \end{cases}$$

h_{uv}^{L} 表示 w 到 v 基于 L-1 随机游走的击中时间。

2.几个问题

随机游走问题：

基于 L 随机游走的模型，引入两种类型的随机游走问题。

1） 定义$S \subseteq V$，从$u \in V$有一个 L 随机游走，在这个随机游走中，如果在任何离散的时间[0, L]中抵达 S 中的任意一个点，就称之为 u 击中了 S。举个例子：假如 S = {v5, v6}, L= 4, 有一个从 v1 开始的 L 随机游走（v1, v2, v3, v2, v6），这个随机游走最后抵达了 v6 这个节点，我们就称之为 v1 击中了 S. 当然，如果 u 是 S 中的一个元素，我们就认为 u 击中了 S.

2） 定义一种新的概念叫做**广义上的击中**, 即从单一的一个源节点到目标集合 S，即：

$$T_{\mathrm{uS}}^{L} \triangleq \min\{\min\{t : Z_u^t \in S, t >= 0\}, L\}$$

T_{uS}^{L}是在源和目标节点中击中时间就是从源节点到任意 S 目标节点中的一个节点的 L 随机游走中第一个跳的预期数量。

显然，如果 S 是空集，则T_{uS}^{L} = L(如果 S 是空集，u 就不可能击中 S，则按照 $T_{\mathrm{uS}}^{L} \triangleq \min\{\min\{t : Z_u^t \in S, t >= 0\}, L\}$ 来说他的值是无限)，除此之外，如果 L = 0，$T_{\mathrm{uS}}^{L} \triangleq \min\{\min\{t : Z_u^t \in S, t >= 0\}, L\}$ = 0。

基于 $T_{\mathrm{uS}}^{L} \triangleq \min\{\min\{t : Z_u^t \in S, t >= 0\}, L\}$，广义上从 u 到 S 的击中通过 h_{uS}^{L} 定义成了期望的 T_{uS}^{L}。我们通过这种定义就能比较清楚的知道当 h_{uS}^{L} 比较小的时候，u 是更容易击中 S 通过 L 随机游走，通过递归，我们的目的就可以达到。

第一种随机游走分配集问题就是去找到最小的广义打击时间从 v-s 集合里面向 S 集合，|S|<=k,于是，问题就变成了下面：

$$\begin{aligned} &\min \sum_{u \in V \setminus S} h_{uS}^{L} \\ &s.t. \quad |S| \leq k. \end{aligned}$$

紧接着的就是第二种随机游走分配集的问题，我们让 X_{uS}^{L} = 1 如果击中了 u，否则就让他等于 0，第二种问题是求解最大的期望值有 S 所击中的，同样这，我们让 |S|<=k，则问题就变成了：

$$\begin{aligned} &\max \mathbb{E}[\sum_{u \in V} X_{uS}^{L}] \\ &s.t. \quad |S| \leq k. \end{aligned}$$

我们让 p_{uS}^{L} 为在 L 随机游走中从 u 节点开始成功击中 S 中一个节点的可能性，即我们会比较容易的得到：

$$E(X_{uS}^{L}) = P_{uS}^{L}$$

通过定义，我们定义 L>0，则有：

$$p_{uS}^{L} = \begin{cases} 1, u \in S \\ \sum_{w \in V} p_{uw} p_{wS}^{L-1}, u \notin S. \end{cases}$$

3. 一些关键性的代码

1、贪心算法

Input: A graph $G = (V, E)$, and a parameter k
Output: A set of nodes S

$S \leftarrow \emptyset$;
for $i = 1$ to k **do**
  $v \leftarrow \arg\max_{u \in V \setminus S} \{F(S \cup \{u\}) - F(S)\}$;
  $S \leftarrow S \cup \{v\}$;
return S;

2. 倒排索引算法

Input: A graph $G = (V, E)$, two parameters L and R
Output: An inverted index $I[1:R][1:n]$

Initialize an inverted list $I[1:R][1:n] \leftarrow$ null;
for each node $w \in V$ **do**
  for $i = 1:R$ **do**
    Initialize $visited[1:n] \leftarrow 0$;
    $u \leftarrow w$;
    $visited[u] \leftarrow 1$;
    for $j = 1:L$ **do**
      Randomly select a neighbor of u, denoted by v;
      if $visited[v] = 0$ **then**
        $visited[v] \leftarrow 1$;
        $Object.id \leftarrow w$;
        $Object.weight \leftarrow j$; /*$w$ hits v at j-th step*/
        /*$Object.weight \leftarrow 1$; for Problem (2)*/;
        $I[i][v].push\_back(Object)$;
      $u \leftarrow v$;
return $I[1:R][1:n]$;

3. 约束边缘增益算法

Input: The inverted index $I[1:R][1:n]$, the array $D[1:R][1:n]$, a node u and parameter R
Output: Approximate marginal gain σ_u

Initialize $\sigma_u \leftarrow 0$;
for $i = 1 : R$ **do**
  $\sigma_u \leftarrow \sigma_u + D[i][u]$;
  /*$\sigma_u \leftarrow \sigma_u + 1 - D[i][u]$; for Problem (2)*/
  while $Object \leftarrow I[i][u].pop()$ **do**
    $v \leftarrow Object.id$;
    if $Object.weight < D[i][v]$ **then**
      $\sigma_u \leftarrow \sigma_u + D[i][v] - Object.weight$;
    /*for Problem (2), use line 8-9 to replace line 6-7*/
    if $Object.weight > D[i][v]$ **then**
      $\sigma_u \leftarrow \sigma_u + Object.weight - D[i][v]$;
$\sigma_u \leftarrow \sigma_u / R$;
return σ_u;

4. 更新图算法

Input: The inverted index $I[1:R][1:n]$, the array $D[1:R][1:n]$, a node u and parameter R
Output: The updated array $D[1:R][1:n]$

for $i = 1 : R$ **do**
  $D[i][u] \leftarrow 0$; /*$D[i][u] \leftarrow 1$; for Problem (2)*/
  while $Object \leftarrow I[i][u].pop()$ **do**
    $v \leftarrow Object.id$;
    if $Object.weight < D[i][v]$ **then**
      $D[i][v] \leftarrow Object.weight$;
    /*for Problem (2), use line 7-8 to replace line 5-6)*/
    if $Object.weight > D[i][v]$ **then**
      $D[i][v] \leftarrow Object.weight$;

5. 近似贪婪算法

Input: A graph $G = (V, E)$, and parameters k, R
Output: A set of nodes S

1: $I[1:R][1:n] \leftarrow$Invert_Index(G, L, R);
2: $S \leftarrow \emptyset$;
3: Initialize $D[1:R][1:n] \leftarrow L$;
/*$D[1:R][1:n] \leftarrow 0$; for Problem (2)*/
4: **for** $i = 1$ to k **do**
5: $v \leftarrow \arg\max_{u \in V \setminus S}$ Approx_Gain($I[1:R][1:n], D[1:R][1:n], u, R$);
6: $S \leftarrow S \cup \{v\}$;
7: Update($I[1:R][1:n]$, $D[1:R][1:n]$, v, R);
8: **return** S;

四、算法分析结论

这个模块我们进行时间和空间复杂度的分析：

倒排索引算法：

时间：O（RLn）

约束边缘增益算法：

时间：O(n)

图的更新算法：

时间：最大是 O（Rn）

近似贪婪算法：

时间：O（KRLn）

空间：O（nRL+m）

最后的近似贪婪算法可以达到：$1-\frac{1}{e}-\zeta$的近似度，当 R 比较小的时候这个算法的效果以及消耗是很好的

五、一个实际例子

简单起见，我们设 R=1，L=2， k=2。即假定这是一个 2-随机游走，对于每种游走的方式，我们会有（v1, v2, v3）, (v2, v3, v5), (v3, v2, v5), (v4, v7, v5), (v5, v2, v6), (v6, v7, v5), (v7, v5, v7)和（v7, v8, v4）。我们把他进行倒排索引，则我们很容易的会得到一张表：

v_1:	
v_2:	$< v_1, 1 >, < v_3, 1 >, < v_5, 1 >$
v_3:	$< v_1, 2 >, < v_2, 1 >$
v_4:	$< v_8, 2 >$
v_5:	$< v_2, 2 >, < v_3, 2 >, < v_4, 2 >, < v_6, 2 >, < v_7, 1 >$
v_6:	$< v_5, 2 >$
v_7:	$< v_4, 1 >, < v_6, 1 >, < v_8, 1 >$
v_8:	

在（v7， v5， v7）中，v7 是一个重复的节点，于是第二个 v7 将不会被插入到倒排索引中去。于是，所谓的倒排索引就像上面的表一样被我们所构建好。

然后我们通过一个近似贪婪算法将 S 这个集合进行初始化，给与他 D[1][1:8]中的数据。然后他就调用约束边源增大的算法。经过这不之后，我们就能估计出每个节点的边缘增益，在这个例子中，我们可以得到：

$\sigma_{v1}(\varnothing)=2$， $\sigma_{v2}(\varnothing)=5$，$\sigma_{v3}(\varnothing)=3$， $\sigma_{v4}(\varnothing)=2$， $\sigma_{v5}(\varnothing)=3$，$\sigma_{v6}(\varnothing)=2$， $\sigma_{v7}(\varnothing)=5$， $\sigma_{v8}(\varnothing)=2$。

举个例子，对于 v2 这个节点，在 I[1][2]中有三个元素。V1, v3, v5 这三个节点都能够击中 v2，则 D[1][1], D[1][3], D[1][5]=2，于是我们就可以得到 D[1][2]+3=5.

和上述一样，我们去分析其他的节点。

最后我们得到，v2 和 v7 得到了最大的边际收益，图的首次随机游走也就停止。

递归着，我们选择 v2 这个节点，让他加入到 S 中去，然后我们就去更新图（用上面呢更新图的那个算法）,则 D[1][2], D[1][1], D[1][3]和 D[1][5]都将被更新,他们在经过更新之后将变为 0, 1, 1, 1. 然后重复之前的过程，进行图的第二次随机游走。、

选择 v7 加入 S 中……

我们就可以输出 v2, v7……