基于 GPU 的大尺度网络布局显示①

李甜甜, 卢 罡, 许南山, 郭俊霞

(北京化工大学 信息科学与技术学院, 北京 100029)

摘 要: 复杂网络的可视化是复杂网络研究中的重要手段. 随着 Web2.0 时代和大数据时代的来临, 作为研究对象的复杂网络的规模越来越大, 这对复杂网络可视化布局算法的布局效果和运算速度提出了新的挑战. 本文针对复杂网络布局的力导引算法, 从布局效果和算法效率两方面对该算法进行了改进和实现. 布局效果方面, 利用复杂网络中的关节点, 对网络数据进行抽象合并, 从而实现分层次的网络布局显示. 算法效率方面, 针对压缩后的网络采用具有强大浮点运算能力的 GPU 进行计算, 对力导引算法需要斥力计算、引力计算和坐标更新三个部分均实现了基于 GPU 的并行计算, 大大提高了计算效率.

关键词: 复杂网络; 可视化; 布局算法; 力导引; GPU; 关节点

Compressed Layout for Large Scale Networks Based on GPU

LI Tian-Tian, LU Gang, XU Nan-Shan, GUO Jun-Xia

(Infomation Science and Technology, Beijing University of Chemical Technology, Beijing 100029, China)

Abstract: One of the ways to study complex networks is making them visualized. With the advent of Web2.0 era and the big data era, the scale of the complex networks is becoming larger. That brings the new challenge for the visualization of complex networks on the layout effect and speed. This paper aims at improving the FDA(Force-Directed Algorithm) from this two aspects. On the layout effect, the articulation points of the complex networks are used to compact the networks, which achieves showing the layout result hierarchically. On the speed of the algorithm, repulsive force and attraction are computed on GPU, and the operation of updating the coordinates of vertex is also implemented on GPU. That improves the efficiency of the algorithm greatly.

Key words: complex network; visualization; layout algorithm; force-directed algorithm; articulation points

将网络图进行可视化最早出现在二十世纪三十年代, Moreno 在纸上画出了一副体现男女小朋友社交关系选择图. 此后, Freeman[1]在文章中详细描述了社交网络可视化的发展过程. 社交网络可视化作为复杂网络可视化的一个重要部分, 因而促进了复杂网络可视化的发展. 随着研究网络的规模越来越大, 传统的数据表达方式已难以满足需求, 复杂网络可视化随之得到了广泛的应用和研究. 网络数据可视化便于人们对数据关系排布、节点密度、社区结构等属性进行分析[2], 同时探索网络数据内部关系, 挖掘数据信息, 进行链路预测等. 这些在社会化关系分析, 生物网络探索等方面都起到了重要作用. 因此, 越来越多的学者们致力于复杂网络有效的可视化的研究[3]. 复杂网络可视化包括复杂网络理论, 信息统计及数据挖掘等诸多理论内容, 其中, 网络布局算法是复杂网络可视化研究的核心问题.

网络布局算法就是将网络的非结构化数据用点边关系均匀地分布在二维平面或三维空间中[4], 从而显示网络结构, 反应网络信息. 网络布局算法在整体上分为节点-链接法、相邻矩阵布局法和混合布局法等. 其中属于节点-链接法中的力导引布局算法由于其布局的美观性、易理解等优点成为了应用最广泛的算法. 力导引布局最初由 P.Eades 在 1984 年提出[5]. 该算法引入了物理弹簧模型, 将网络中的点和边看做是用弹簧

① 基金项目:北京高等学校青年英才计划(YETP0506)
收稿时间:2015-01-26;收到修改稿时间:2015-03-19

相连的钢球, 两点间的距离会受到弹簧弹力的控制, 当整个网络中的点达到受力平衡状态时, 点的位置被最终确定. 此后, Kamada T 和 Kawai S 在弹簧模型的基础上提出了能量求解公式[6], 将求解节点受力平衡状态转换为求解系统能量最小化的问题. Fruchterman 和 Reingold 在前人研究的基础上提出了 FR 算法[7], 引入了粒子物理理论. 他们将节点模拟成为原子, 彼此之间具有斥力和引力, 节点在力的作用下不断移动, 直至稳定状态. LinLog[8,9]算法结合了 BH 算法优化能量函数, 该算法能够较好的显示网络结构, 同时在一定程度上降低了时间复杂度. 在以上几种经典力导引算法的基础上, 诸多专家和学者进一步提出了改进算法以适应不同的数据需求. 曲建华等人提出的 PSOGD 算法[10], 将粒子群算法和 FR 算法相结合, 使布局结果得到了全局最优. Gilbert[11]等人提出了可视化压缩算法, 根据节点和边的重要程度缩减网络数据规模. 但是针对大规模网络在计算方面存在一定的性能问题. MA 教授提出的 OntoVis[12]系统将数据可视化与语义相结合, 压缩重复出现的路径, 取得了较好的可视化效果. 但是该算法对节点和边的重要性的定义不够明确, 并且需要预先对数据进行语义分析. 吴鹏提出的 SAL 算法[13], 将力导引算法结合社会网络分析, 将网络中的节点赋予角色和属性, 从而将网络划分成为不同子群, 突出了网络结构特征, 但是该方法对于大规模网络计算的复杂度较高. 此外, Gansner 等人将多维尺度更新的方法应用到 KK 算法中[14], 将 KK 算法的坐标位置更新方式变为“成批”更新, 提高了算法效率. 王勇献等人在此基础上进一步对坐标更新公式调整[15], 实现了将“成批”更新坐标位置的 KK 算法移植到 GPU 进行实现. Jezowicz 等人将 FR 算法中的斥力计算部分进行了 GPU 加速计算[16], 在一定程度上提高大规模网络的计算速度. 这些算法未关注具有大量节点和边的大尺度网络在有限的显示面积中的布局显示问题.

本文将处理大尺度网络显示和提高算法计算速度两个方面相结合, 对大尺度复杂网络的快速可视化问题进行了研究. 处理大尺度网络显示方面, 本文基于网络中的关节点, 实现对网络节点和边的压缩处理, 从而在提高大规模网络布局算法运算速度的同时, 能够分层次显示出网络结构. 在此基础上, 将 FR 算法计算的引力计算、斥力计算和坐标更新三个部分均基于 GPU 实现并行计算, 以加快计算速度.

1 FR算法原理

对于网络 $G=(V, E)$, V 是其顶点集合, E 是其边集合. FR 算法将网络中点与点之间的关系类比成原子之间的斥力和引力, 任意两个节点间均存在斥力, 有边相连的两个节点间存在引力. 斥力和引力的具体定义公式分别如下:

$$F_r = \frac{k^2}{d} \tag{1}$$

$$F_a = \frac{d^2}{k} \tag{2}$$

其中 F_r 为斥力, F_a 为引力, d 表示两个顶点之间的实际距离, k 表示网络中每个点所占据的理想空间的大小. k 的计算公式如下:

$$k = c\sqrt{\frac{area}{|V|}} \tag{3}$$

其中 c 是由实验确定的常数, $area$ 表示显示区域的面积, $|V|$表示网络中节点数量. 在 FR 算法中还引入了“温度”来控制迭代次数. 温度以逆序的方式下降, 当达到设定值时结束迭代, 获得节点最终坐标值, 是模拟退火算法的一个特例. 在 FR 算法的每一轮迭代中都需要计算斥力位移变化量、引力位移变化量, 并根据二者的位移变化量更新节点坐标值. 首先利用公式(4)计算任意两个节点 v_i 和 v_j 之间的欧式距离 Δ, 在此基础上, 根据公式(5)计算节点 v_i 和其他所有节点间斥力引起的位移变化量 $Disp(i)$, 以及根据公式(6)计算分别节点 v_i 和节点 v_j 在引力作用下的位移变化量 $Disp(i)$ 和 $Disp(j)$. 最终在更新坐标值部分, 将位移变化量与当前温度值作比较, 并限定坐标值在屏幕显示范围内, 完成坐标值的更新.

$$\Delta = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \tag{4}$$

$$Disp(i) = Disp(i) + \frac{\Delta}{|\Delta|} F_r(|\Delta|) \tag{5}$$

$$\begin{cases} Disp(i) = Disp(i) + \frac{\Delta}{|\Delta|} F_a(|\Delta|) \\ Disp(j) = Disp(j) - \frac{\Delta}{|\Delta|} F_a(|\Delta|) \end{cases} \tag{6}$$

2 算法改进及实现

2.1 网络数据存储

对于有向网络 $G=(V, E)$, 采用如图 1 所示的存储结构. 其中, 数组 S 存储网络数据中的源节点编号, S 中每个节点的邻居节点按顺序存储在数组 T 中. 数组 $Index$ 的索引编号与数组 S 的索引编号对应, 记录 S 中

相应节点的邻居节点在 T 中的起始索引值, 即节点 $S[i]$ 的邻居节点为 $T[Index[i]]$至 $T[Index[i+1]-1]$. 对于无向网络, 节点顺序排列, 按照同样的结构存储, 不考虑边的方向性即可. 这实际上是邻接表的数组实现.

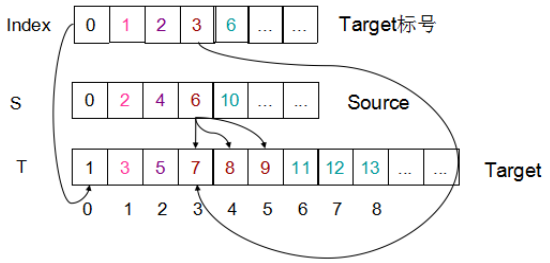


图 1 网络数据存储结构

2.2 基于关节点的大尺度网络层次化布局

网络可视化的目的是帮助人们理解网络的整体模式, 而不是将网络中的每个点每条边都显现出来. 尤其是对于大尺度复杂网络, 由于显示区域尺寸有限, 往往无法有效地显示大尺度网络布局, 从而失去网络可视化的意义. 因此, 采用一定的策略对网络中的节点和边进行聚合抽象, 从而反映网络整体结构概要是非常必要的. 本文引入关节点的概念, 实现网络数据层次化布局显示. 在本文算法中主要分为两个层次, 在上层结构中, 对网络中节点集合进行分类, 针对不同类别的网络节点和边进行聚合抽象, 从而减小网络规模提高屏幕空间利用率, 在宏观上显示网络整体结构. 在下层结构中, 根据需求恢复节点原始结构, 用于显示网络具体内部细节信息.

定义 1. 关节点[17]. 如果在去掉节点 v 以及和 v 相关联的各边之后, 将网络的一个连通分量分割成两个或两个以上的连通分量, 则节点 v 是该网络的一个关节点.

从关节点定义可以得知, 如果将网络数据集中关节点去掉, 原始的网络数据集将被分解成两个或多个连通分量. 如图 2 所示, 图 2(a)表示网络集合的结构图, 图中较大节点表示网络中的关节点. 图 2(b)中虚线表示与关节点相连边. 图 2(c)显示了去掉图关节点及连边后得到的结构图. 从图 2(c)中可以看出, 在得到的连通分量中, 有些是包含若干个节点和边的连通集合, 而有些连通分量仅包含一个孤立节点. 针对以上不同性质的连通分量, 本文算法首先对网络节点集合进行分类, 再对各个不同的分类集合采用不同的方式进行处理.

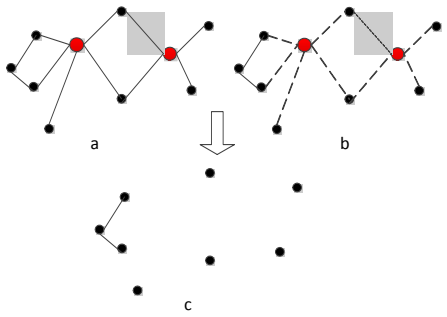


图 2 节点类别示意图

利用深度优先搜索算法遍历网络 $G=(V, E)$, 得到关节点集合 V_A. 为了保证图结构的整体连通性, 不对关节点集合做处理. 针对非关节点 v_i, 进一步计算其节点度值 $Degree(v_i)$, 统计该节点的邻居关节点个数 $Neigharticnum(v_i)$, 然后根据 $Degree(v_i)$ 和 $Neigharticnum(v_i)$的关系将其进一步分类. 具体如下:

(1)V_A 表示关节点集合, 如图 2(a)中较大节点;

(2)$V_B=\{v|Degree(v)=Neigharticnum(v),\ v\in V-V_A\}$ 表示的非关节点集合, 如图 2(c)中孤立节点;

(3)$V_C=V-V_A-V_B$ 表示其他非关节点集合, 如图 2(c)中连通在一起的节点.

同时, 三个集合满足条件如下:

$$\begin{cases} V_A \cup V_B \cup V_C = V; \\ V_I \cap V_J = \phi; (I, J = A, B, C); \end{cases} \tag{7}$$

将 $V_A \cup V_B$ 中的节点及相关连边构成的子图记为 $G_1=(V_1,E_1)$, 则有 $V_1=V_A \cup V_B$. 对于节点 $v_i,v_j\in V_B$, 用 $Neighartic(v_i)$表示 v_i 的邻居关节点集合. 当 v_i,v_j 满足公式(8)时, 就将它们合并为一个节点. 最终, V_B 中的结点将被划分在不同的聚集内, 每个聚集用 V_{Bi} 表示.

$$|Neighartic(v_i) \cap Neighartic(v_j)| \geq 1 \tag{8}$$

具体算法如下. 其中 $Artiflag(v_{bi})$表示 v_{bi} 所在的聚集编号; $Replace(V_{Bi})$表示代替每个聚集的非关节点; $Ifhavereplace(V_{Bi})$表示每个聚集是否已有代替节点; $Ifreplace(v_{bi})$表示 v_{bi} 是否已经做过替代节点. 其 $v_b\in V_B$.

算法 1 将 $V_B=\{v_{b1},v_{b2}...v_{bm}\}$划分成不同的聚集

输入: $G_1=(V_1, E_1)$, $newartiflag$=1;

```
For v_bi in V_B
    Artiflag(v_bi)=initialartiflag;//初始化 Artiflag(v_bi)
End For
For v_bi in V_B
    For v_bj in V_B
```

```
6        //满足公式(8)
         If(|Neighartic(v_bi)∩ Neighartic(v_bj)|>=1)
7          //两个节点均未被合并到聚集内
           If(Artiflag(v_bi)=initialartiflag&&
             Artiflag(v_bi)=initialartiflag)
8            //将两个节点放入同一个新聚集内
             Artiflag(v_bi)= Artiflag(v_bi)=newartiflag;
9            //更新聚集编号
             newartiflag= newartiflag+1;
10         End if
11         Else //已有一个节点合并到聚集内
12           //将新节点合并到聚集内
             Artiflag(v_bi)=max(Artiflag(v_bi),Artiflag(v_bj))
       ;
13           Artiflag(v_bi)=max(Artiflag(v_bi), Artiflag(v_bj));
14         End else
15       End if
16     End for
17   End for
18   For e_ab in E_1 do
19     If(!Ifhavereplace(V_Bi) && !Ifreplace(v_bi))
20       Replace(V_Bi)= v_bi; //得到 v_bi 的替换节点
21     End If
22   End For
输出: Replace(V_Bi), Artiflag(v_bi)
```

算法 1. 首先获取每个节点的邻居关节点, 再对每个节点的邻居关节点进行比较. 对于满足条件的节点判断其所在聚集情况. 如果两个节点都没有加入聚集内的则分配到新的聚集内. 对于只有一个节点被分配的情况则将另一个节点加入到聚集. 在对所有节点分配完聚集后, 再为每个节点获取替代节点. 对 V_B 中的每个节点进行替换, 得到关于 V_A 和 V_B 新的网络集合 $G'_1=(V'_1, E'_1)$.

在完成对 $G_1=(V_1,E_1)$的压缩处理后, 将 V_C 及相关的边 E_C 重新存储成为新的集合 $G_2=(V_C, E_C)$. 遍历边集 E_C, 将 V_C 分类聚集到不同的连通集合内. 定义 *Connectflag*(v_i)表示 v_i 节点所在的连通集合的编号, *Ifconnect*(v_i)标记 v_i 节点是否已被分类到连通集合内, 用 *newconnectflag* 表示新连通集合的编号, 用 Replace(*Connectflag*(v_i))表示 v_i 节点的代替点. 具体算法如下:

算法 2 将 $V_C=\{v_{c1},v_{c2}...v_{cn}\}$分成不同的连通集合

```
输入: G_1=(V_2,E_2), newconnectflag=0, Ifconnect(v_i)
1    For e_ij in E_1 do
2      //两个节点均未被分类到连通集合
       If(!Ifconnect(v_i) && !Ifconnect(v_j)) do
3        //将两个新节点放入新的连通集合
4        Connectflag(v_i)= newconnectflag;
         Connectflag(v_j)=newconnectflag;
5        //更新新的连通集合编号
         newconnectflag=newconnectflag+1;
6      End If
7      //v_i 为新节点, v_j 已被分类到连通集合
       If(!Ifconnect(v_i) && Ifconnect(v_j)) do
8        //将新节点放入相连节点所在的连通集合
         Connectflag(v_i)=Connectflag(v_j);
9      End If
10     //两个节点均已被分类到连通集合
       If(Ifconnect(v_i) && Ifconnect(v_j)) do
11       //两个节点在不同的连通集合
         If(Connectflag(v_i)!=Connectflag(v_j))
12         //将两个节点所在的连通集合合并
           For v_n in V_1 do
13           Connectflag(v_n)=min(Connectflag(v_i),
                                 Connectflag(v_j));
14         End If
15       End For
16   End For
输出: Connectflag(v_i)
```

在算法 2 中通过访问边集的方式来判断节点被分配到各个连通集合的情况. 如果一条边的两个节点均未被分配, 则将这两个节点放入到新的连通集合中. 如果仅有一个节点被分配, 则将另一个节点直接加入该连通集合. 对于两个节点都已经被的情况, 则把所在的两个连通集合合并. 最终将 V_C 节点集合分类到不同的连通集合内, 并在每个连通集合内部选取第一个节点 v_{first} 作为该连通集合内节点的代替点, 即 Replace(*Connectflag*(v_i))=v_{first}, 得到新的 V_C 节点网络集合 $G'_2=(V'_2,E'_2)$. 最后, 将 G'_1 和 G'_2 合并, 得到压缩后的网络集合 $G'=(V',E')$.

在下层结构中, 可以根据用户需求, 将节点内部的结构依据 FR 布局算法再次显示出来, 进而显示网

络详细信息. 由于大尺度网络数据在压缩后也具有较大的数据量, 对于包含节点数目较多的节点, 本文算法可以对该节点内部结构进行再次压缩, 更加清晰的显示网络细节信息.

2.3 基于 GPU 的加速 FR 算法

传统的 FR 算法需要进行多迭代后产生最终布局结果, 针对规模较大的网络, 更需要迭代几百甚至上千次才能够得到令人满意的结果. 每一次迭代过程中都需要计算斥力、计算引力、更新坐标三个过程才能保证最终结果的有效性. 对于压缩后的大尺度复杂网络而言, 要想得到满意的布局效果依然需要较大的时间开销, 难以在可接受时间内得到可视化结果, 这些都给复杂网络可视化带来了诸多不便. 因而, 本文针对这一问题对 FR 布局算法进行了改进. 针对压缩后的网络数据 $G'=(V', E')$, 将算法中每次迭代的三个过程均移植到 GPU 上进行, 大大的缩短了运算时间, 使算法效率得到显著提高.

在斥力计算部分, 需要对任意两点之间的斥力进行计算, 其时间复杂度为 $O(|V|^2)$, 是整个 FR 算法中时间复杂度最高的部分. 针对大规模网络数据, 即使在压缩后也需要巨大的时间开销. 因此, 首先将斥力部分的计算移植到 GPU 上, 将 GPU 并行线程数量设置为与节点个数相同, 为每个需要计算的节点 v_i 分配一个 GPU 线程, 负责计算节点 v_i 与$\{v_{i+1},v_{i+2}...v_n\}$之间的斥力及位移变化量. 由于斥力计算部分与连边关系无关, 因而仅需要将节点集 $V'=\{v_1,v_2...v_n\}$拷贝到 GPU 显存, 此外, 还需将初始化后用来存储坐标值的 X 数组和 Y 数组, 表示位移变化量的 *Disp_x* 数组和 *Disp_y* 数组依次拷贝到 GPU 显存, 具体计算过程如下:

算法 3 计算斥力作用下的位移变化量

输入: $V'=\{v_1,v_2...v_n\}$, *Disp_x,,Disp_y*, X,Y

```
1       //为每一个节点分配一个 GPU 线程
        Parallel For i=0 to nodescount-1 do
2         For j = i+1 to nodescount-1 do
3           根据公式(5)计算两点间距离 Δij;
4           根据公式(2)计算引力 Frij;
5           根据公式(6)计算 vi 节点位移变化量 Disp(i);
6           根据公式(6)计算 vj 节点位移变化量 Disp(j);
7         End For
8       End Parallel For;
```

输出: *Disp_x,,Disp_y*

在算法 3 中, *nodecount* 值用于表示参与计算的节点个数, 每个线程分别根据第 1 节中的公式(5)和公式(2)计算两点间距离、两点间斥力, 进而根据公式(6)求得节点的位移变化量, 最终更新存储位移变化量的 *Disp_x* 数组和 *Disp_y* 数组.

在完成 FR 算法的斥力计算后, 本文也将 FR 算法引力计算部分移植到 GPU 进行, 进一步提高算法效率. FR 算法的引力计算部分需要计算所有相邻节点之间的引力及位移变化量, 无连边关系的节点间在计算过程中互不影响. 为了保证算法的准确性, 必须保证节点 v_i 与其邻居节点之间的引力计算的串行执行, 因而本文采用如图 1 所示的存储结构, 将压缩后的 $G'=(V',E')$中的 $E'=\{e_1',e_2'...e_m'\}$关系重新存储为 *S',T',Index'*三个数组, 并为 *S'*数组中的每个节点分配一个 GPU 线程, 用于计算所有与 *S'*[*i*]节点相邻的节点之间的位移变化量. 具体算法如下:

算法 4 计算引力作用下的位移变化量:

输入: *S',T',Index'*

```
1       //为 S'数组每个节点分配一个 GPU 线程
        Parallel For i=0 to S'.size do
2         For j=Index'[i] to Index'[i+1] do
3           根据公式(5)计算两点间距离 Δij;
4           根据公式(3)计算引力 Faij;
5           根据公式(7)分别计算 vi 节点和 vj 节点位移变化量 Disp(i)和 Disp(j);
6         End For;
7       End Parallel For;
```

输出: *Disp_x,,Disp_y*

引力计算部分采用的 GPU 线程分配方式与斥力计算部分不同, 但是在计算部分相类似. 根据第 1 节中的公式(5)和公式(3), 分别计算出节点间距离和节点间引力. 利用公式(7)计算位移变化量, 并对位移变化量数组更新, 最终得到该迭代次数内的总位移变化量.

在完成算法 3 和算法 4 的计算后, 需要根据斥力计算和引力计算部分所得出的位移变化量对坐标值进行更新. 由于每一个节点都需要进行坐标值更新, 因而将其移植到 GPU, 使各个节点坐标的更新计算可以并行, 提高计算效率. 本文算法为每个节点分配一个 GPU 线程, 用于判断节点 v_i 的坐标变化量 *Disp_x*[v_i]、

$Disp_y[v_i]$与“温度“t之间的关系, 完成节点v_i的坐标值更新, 提高算法的并行性. 具体如下:

算法 5 更新节点坐标值

输入: t

```
//为每个节点分配一个 GPU 线程
Parallel For i=0 to nodescount do
    计算 Δij;
    If Δij≤t then do
        Δij=t/Δij;
    End If;
    更新 Xi;
    更新 Yi;
End Parallel For;
iteration=iteration-1;
End for;
```

输出: 节点坐标位置数组 $X[i]$和 $Y[i]$

在上述算法 5 中, 主要是应用算法 4 得到的坐标变化量与温控系数 t 进行比较, 从而完成整个坐标值的更新. 上述三个部分的算法均针对压缩后的网络数据, 实现了将 FR 算法中的三个部分移植到 GPU 上完成并行计算, 相比于传统的 FR 算法, 大大缩短了运算时间. 尤其是针对网络数据规模较大, 迭代次数较多的情况下, 算法效率提高效果更为显著.

3 实验及分析

3.1 实验环境

本文算法选取斯坦福大学网站上提供的复杂网络数据进行测试, 数据节点集大小从约 200 到约 70000 个顶点. 本文采用随机布局作为初始布局进行测试. 硬件环境为一台具有 Intel® Core™ i7-4770 的主频为 3.40GHz 的 CPU、32G 内存、NVIDIA GeForce GTX480 的 GPU 的工作站, 装有 64 位的 Windows Server 2008 R2 Enterprise 操作系统. 算法采用 C++ AMP 实现, 布局结果显示采用 OpenGL 编程接口.

3.2 实验结果及分析

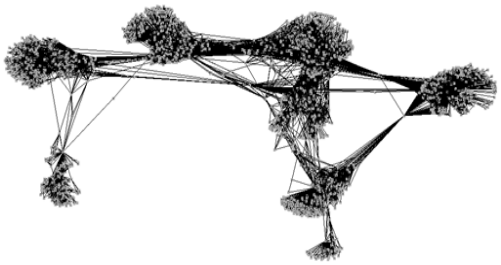
表 1 显示了在本文算法下对网络数据进行压缩的比较结果, 其中 G 为原始网络, G’为压缩后的网络, 可以看到实现了较大的压缩比, 得到了较好的数据压缩效果. 表 2 显示了压缩前后网络图的关节点数量和连通分量数量的情况. 从表中可以看出, 依据网络中关节点的相关概念, 对网络节点和边进行聚合抽象, 保证了网络在压缩前后关节点数目和连通分量数目都保持不变, 从宏观角度保证了网络整体结构不变. 图 3(a)和图 2(b)显示了 G1 被压缩合并前后的布局结果. 从图中可以看出, 压缩效果较为显著. 算法将该网络压缩到 37 个节点, 104 条边, 大大缩减了数据量. 图 3(b)中较大节点表示网络集中的关节点, 较小节点表示被压缩合并过的节点. 整个网络由关节点贯穿, 保证了其连通性, 较小节点分布密集的区域表示与该关节点相关联的节点较多. 图 3(c)和图 3(d)中显示了 G3 被压缩前后的结果. 该网络为非连通的, 但是在图 3(c)中很难反映出来, 在压缩后的网络中可以看出离散部分的节点信息, 较大节点分布密集的区域表示关节点集中的区域. 通过基于关节点对大尺度网络的压缩, 可以比较清晰的显示网络数据的内部结构和整体布局, 减少了节点密集重叠的现象.

表 1　网络数据压缩比较表

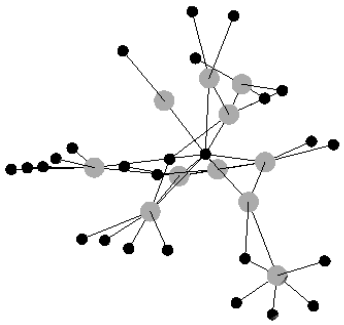
G=(V,E)	G’=(V’,E’)	节点集压缩比(%)	边集压缩比(%)
G1=(4039,88234)	G1’=(37,71)	9.1	0.8
G2=(13498,101565)	G2’=(1052,34793)	7.8	34.2
G3=(8298,103689)	G3’=(2080,24590)	25.0	23.7
G4=(76757,499802)	G4’=(3278,207064)	4.2	41.4
G5=(22687,54705)	G5’=(8613,25886)	37.9	47.3
G6=(75888,508837)	G6’=(32922,323178)	43.4	63.5
G7=(77360,905468)	G7’=(52185,505770)	67.5	55.9
G8=(82168,948464)	G8’=(51771,469716)	63	49.5

表 2　网络关节点数和连通分量数比较表

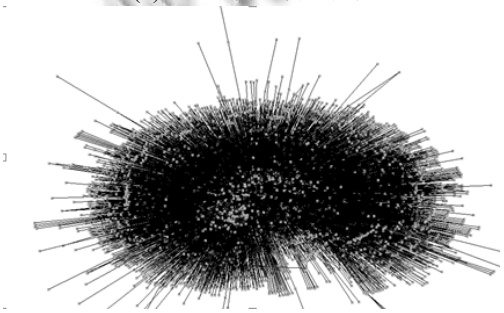
G	关节点数	连通分量数	G’	关节点数	连通分量数
G1	11	16	G1’	11	16
G2	525	18	G2’	525	18
G3	1033	21	G3’	1033	21
G4	1635	8	G4’	1635	8
G5	4273	76	G5’	4273	76
G6	1584	863	G6’	1584	863
G7	2096	1395	G7’	2096	1395
G8	4273	760	G8’	4273	760



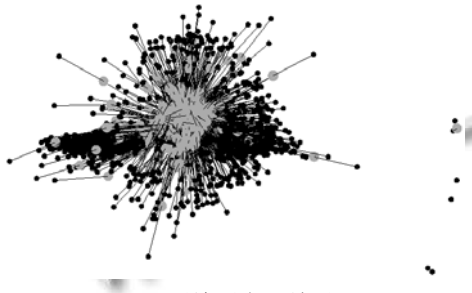
(a) G1=(4039,88234)布局结果



(b) G1 压缩后布局结果



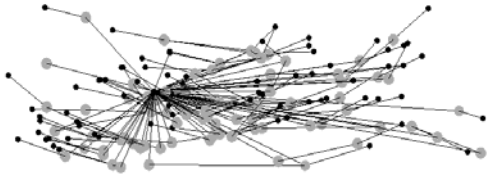
(c) G3=(8298,103689)布局结果



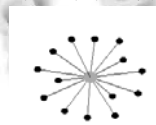
(b) G3 压缩后布局结果

图 3 布局结果

在下层结构中, 由于大尺度网络数据在压缩后也具有较大的数据量, 在可视化界面中提供节点编号会影响可视化效果. 因而本文算法提供了关于被压缩数据的具体信息, 可以根据需要, 进一步显示相应节点的细节信息. 图 4(a)显示了图 3(b)中 1 号节点的详细结构, 1 号节点中包含了 3688 个节点, 采用了再次压缩的方式较为清晰的显示了 1 号节点的内部结构. 图 4(b)显示了图 3(b)中 11 号节点的内部结构, 11 号节点中包含了 14 个节点, 因而采用原始 FR 布局算法可视化其真实结构信息.



(a) G1’中 1 号节点内部结构



(b) G1’中 11 号节点内部结构

图 4 G1 下层结构信息显示

对于不同规模的网络数据, 分别在 CPU 和 GPU 上运行 FR 算法, 均采用迭代 500 次得出布局结果. 图 5 显示了不同规模的原始网络 G 在 CPU 上运行和移植到 GPU 上运行时间的比较情况, 图 6 显示了 G 在 CPU 和 GPU 运行时间的加速比. 从实验结果中可以看出, FR 算法在 GPU 上运行的时间比在 CPU 上运行大大缩短, 算法在 GPU 上并行计算的效果较为显著. 图 7 显示了压缩后的网络 G’分别在 CPU 上运行和移植到 GPU 上运行的实验结果, 图 8 对应显示了 G’在 CPU 和 GPU 上运行时间的加速比. 从实验数据中可以看出, 针对较大规模的网络数据, 即使在压缩后采用 FR 布局算法进行布局依然需要较大的时间开销. 而算法在 GPU 上运行速度比在 CPU 上运行提高数倍, 得到了较大的加速比. 以上实验结果均反映出算法在 GPU 上进行并行计算的效果显著性.

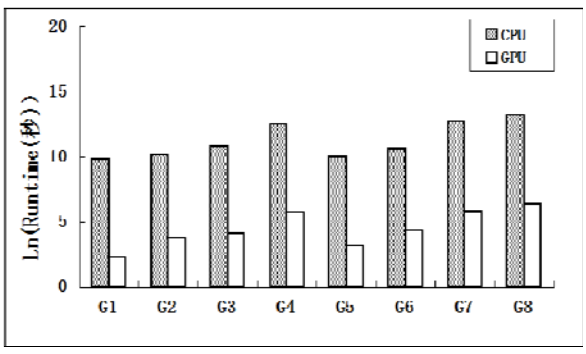


图 5 原始网络 G 的 CPU 和 GPU 运行时间比较

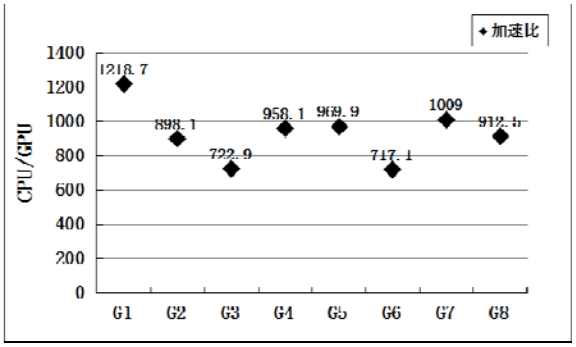


图 6　原始网络 G 的 CPU 和 GPU 运行时间加速比

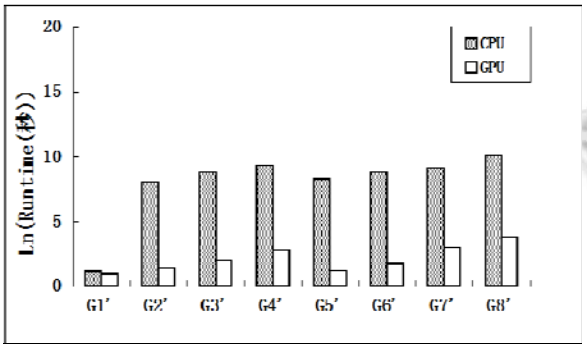


图 7　压缩网络 G'的 CPU 和 GPU 运行时间比较

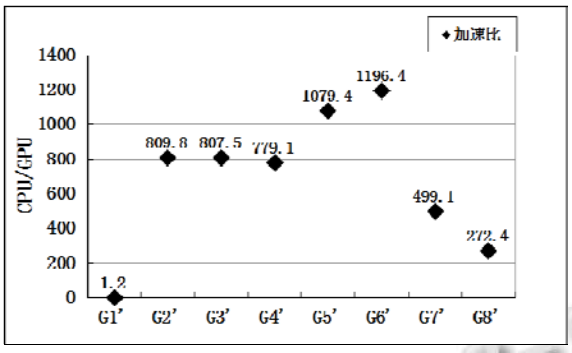


图 8　压缩网络 G'的 CPU 和 GPU 运行时间加速比

4　结语

复杂网络规模的不断扩大对网络可视化的要求在不断提升，一方面要求能够将布局结果在有限的显示范围内比较清晰显示出来，另一方面需要较高的数据处理速度，能够在用户可接受的时间内得到布局结果. 本文结合了关节点的概念，遍历网络数据得到关节点. 根据顶点与关节点的关系将节点集合分成三类，采用不同的规则对节点集合进行抽象合并，在保证原网络连通性，关节点数量和连通分量不变的前提下，对数据量进行压缩，从而实现分层次显示网络结构，充分利用显示区域的有效空间. 同时，实验表明，采用文中所述的算法可以较大程度的减小数据规模，在显示数据整体结构方面得到较好的效果，同时支持网络细节信息显示，从不同层次观察网络结构. 另一方面，在压缩数据规模的基础上，对网络数据进行重新存储，针对应用最广泛的 FR 布局算法进行改进，将 FR 算法中的计算斥力、计算引力和更新坐标均移植到 GPU 上进行并行运算. 实验表明使用 GPU 对 FR 算法进行加速可以大大提高算法的运行效率，缩短运行时间.

未来的工作主要可以在以下两方面开展：1)考虑到 GPU 的显存大小毕竟有限，对于任意大尺度的复杂网络，相关布局算法策略的设计与实现；2)进一步优化布局效果.

致谢　在此我向对本文的工作给予支持和建议的同行，以及北京化工大学网络数据库实验室老师和同学表示感谢.

参考文献

1 Freeman LC. Visualizing social networks. Journal of social structure. 2000,1(1):4.

2 Akhtar N. Social Network Analysis Tools. Communication Systems and Network Technologies, Bhopal: IEEE, 2014. 388–392.

3 Dunne C. Measuring and improving the readability of network visualizations[Ph.D. Thesis]. State of Maryland. University of Maryland, College Park. 2013.

4 Liu K, Ma C, Muelder W. Large-Scale Graph Visulization and Analytics. IEEE Computer Society, 2013, 46(7): 39–46.

5 Eades P. A heuristic for graph drawing. Congressus Nutnerantiunt, 1984, 42: 149–160.

6 Kamada T, Kawai S. An algorithm for drawing general undirected graphs. Infomation Processing Letters, 1989, 31(1): 7–15.

7 Fruchterman T, Reingold E. Graph drawing by force-directed placement. Software: Practice and Experience, 1991, 21(11): 1129–1164.

8 Noack A. An energy model for visual graph clustering. Graph Drawing, 2004: 425–436.

9 Noack A. Energy Models for Graph Clustering. Graph Algorithms and Applications, 2007(2):453–480.

10 Qu JH, Song Y, Bressan S. PSOGD: A new method for graph

drawing. Data Engineering Workshops. Brisbane QLD: IEEE, 2013:229–235.

11 Gibert AC, Levchenko K. Compressing network graphs. LINK KDD 2004.

12 Shen ZQ, Ma KL, Eliassi-Red. Visual Analysis of Large Hereogeneous Social Networks by Semantic and Structural Abstraction. IEEE Trans. on Visualization and Computer Graphs, 2006,12(6):1427–1439.

13 吴鹏,李思昆.适于社会网络结构分析与可视化的布局算法.软件学报,2011,22(10),2467–2475.

14 Jezowicz T, Kudelka M. Visualization of Large Graphs Using GPU Computing. Intelligent Networking and Collaborative Systems. Xi'an: IEEE , 2013: 662–667.

15 王勇献,曹维.基于 GPU 的力导引图绘制算法加速.北京,中国科技论文在线,2011:1–8.

16 Gansner E, Koren Y, North S. Graph drawing by stress majorization. Lecture Notes in Computer Science, 2004: 239–250.

17 Kuang J, Young EFY. An efficient layout decomposition approach for triple patterning lithography. Design Automation Conference. Austin TX: IEEE, 2013:1–6.