1

Community Detection

Liu Hongzun

Department of Computer Science, Tsinghua University

2020.02.10

Social, technological and information systems can often be described in terms of complex networks that have a topology of interconnected nodes combining organization and randomness.The typical size of large networks such as social network services, mobile phone networks or the web now counts in millions when not billions of nodes and these scales demand new methods to retrieve comprehensive information from their structure.A promising approach consists in decomposing the networks into sub-units or communities, which are sets of highly inter-connected nodes. The identification of these communities is of crucial importance as they may help to uncover a-priori unknown functional modules such as topics in information networks or cyber-communities in social networks. Moreover, the resulting meta-network, whose nodes are the communities, may then be used to visualize the original network structure. The problem of community detection requires the partition of a network into communities of densely connected nodes, with the nodes belonging to different communities being only sparsely connected. Precise formulations of this optimization problem are known to be computationally intractable. Several algorithms have therefore been proposed to find reasonably good partitions in a reasonably fast way. This search for fast algorithms has attracted much interest in recent years due to the increasing availability of large network data sets and the impact of networks on every day life. One can distinguish several types of community detection algorithms: divisive algorithms detect inter-community links and remove them from the network, agglomerative algorithms merge similar nodes/communities recursively and optimization methods are based on the maximisation of an objective function.

定义图 G 的模块度 (modularity):

$$
\begin{aligned}
Q &= \frac{1}{2m} \sum_{i,j \in 1,2,\ldots n} \left(A_{ij} - \frac{d_i d_j}{2m} \right) \delta(C_i, C_j) \\
&= \sum_i \left(\frac{d_{in}(C_i)}{2m} - \left(\frac{d_{tot}(C_i)}{2m} \right)^2 \right)
\end{aligned}
\tag{1}
$$

* liu-hz18@mails.tsinghua.edu.cn

将节点 i 从社群 C_k 移出，移入社群 C_j, 对模块度 Q 的贡献为:

$$\begin{aligned}
\triangle Q(i, C_j) &= \frac{1}{2m}\left[\left(d_{in}(C_j) + d(k, C_j) - \frac{(d_{tot}(C_j)+d_k)^2}{2m}\right) + \left(d_{in}(C_i) + d(k, C_i) - \frac{(d_{tot}(C_i)-d_k)^2}{2m}\right)\right] \\
&\quad - \frac{1}{2m}\left[\left(d_{in}(C_j) - \frac{d_{tot}(C_j)^2}{2m}\right) + \left(d_{in}(C_i) - \frac{d_{tot}(C_i)^2}{2m}\right)\right] \\
&= \frac{1}{2m}\left[d(C_j, k) - d(C_i, k) - \frac{2}{2m}\left(d_k \cdot (d_{tot}(C_j) - d_{tot}(C_i) + d_k)\right)\right]
\end{aligned} \tag{2}$$

下面我们给出图的社群发现算法之一: Fast Unfolding

Algorithm 1: Fast Unfolding Method

Input: $G = (V, E)$;
Output: clustering of G;
$k = 0, G^0 = G$;
while *community partition (modularity) can still be changed (improved)* **do**
  make a simple clustering C^k of G^k such that $C_i^k = \{i\}$;
  for *node* $i \in G^k$ **do**
    remove the node i from its community C_i^k;
    C_N = set of neighbour communities of node i;
    $C_j^k = \arg\max_{C_{j'}^k \in C_N} \triangle Q(i, C_{j'}^k)$;
    if $\triangle Q(i, C_{j'}^k) > 0$ **then**
      add node i to the community C_j^k;
    else
      leave node i in the community C_i^k;
  build a new graph G^{k+1} whose nodes are the communities of C^k;
  $k = k + 1$;
make a clustering C_{final} of G;
return C_{final}

This approach reduces the computation time, especially on large networks, and still provides a high coefficient of modularity.

helloWorld.java

```java
public class Main {
    public static void main(String[] args){
        System.out.println("Hello World");
    }
}
```

bubbleSort.c

```c
#include <iostream>
#define LENGTH 8
using namespace std;
//测试用的代码，bubbleSort函数
int main() {
```

```cpp
    int temp,number[LENGTH]={95, 45, 15, 78, 84, 51, 24, 12};
    for(int i = 0; i < LENGTH; i++){
        for(int j = 0; j < LENGTH - 1 - i; j++){
            if(number[j] > number[j+1]) {
                temp = number[j];
                number[j] = number[j+1];
                number[j+1] = temp;
            } //if end
        }
    }
    for(int i = 0; i < LENGTH; i++)
        cout << number[i] << " ";
    cout << endl;
    /* the following code computes $\sum_{i=0}^{n} i$ */
    for (i = 1; i <= limit; i++) {
        sum += i;
    }

    return 0;
}
```

test.py

```python
import random
n = 100
s = [0 for i in range(n)]

for j in range(1000):
count = [0 for m in range(n)]
i = 0
while i < n:
count[i] += 1
if random.randint(0, 1) == 0:
i += 1
for k in range(n):
s[k] += count[k]

print(s)
```

Algorithm 2: How to write algorithms

Input: this text

Output: how to write algorithm with LaTeX2e

initialization;

while *not at end of this document* **do**

    read current;

    if *understand* **then**

        go to next section; current section becomes this one;

    else

        go back to the beginning of current section;

Algorithm 3: identify Row Context

Input: r_i, $Backgrd(T_i)$=$T_1, T_2, \ldots, T_n$ and similarity threshold θ_r

Output: $con(r_i)$

$con(r_i) = \Phi$

for $j = 1; j \leq n; j \neq i$ **do**

    float $maxSim = 0$

    $r^{maxSim} = null$

    while *not end of* T_j **do**

        compute Jaro$(r_i, r_m)(r_m \in T_j)$

        if $(Jaro(r_i, r_m) \geq \theta_r) \wedge (Jaro(r_i, r_m) \geq r^{maxSim})$ **then**

            replace r^{maxSim} with r_m

        end

    end

    $con(r_i) = con(r_i) \cup r^{maxSim}$

end

return $con(r_i)$

Algorithm 4: algorithm caption

```
Input: input parameters A, B, C
Output: output result
some description;
for condition do
    only if
    if condition then
        1
while not at end of this document do
    if and else
    if condition then
        1
    else
        2
foreach condition do
    if condition then
        1
```