Relabeling nodes according to the structure of the graph

Ronan Hamon, Céline Robardet, Pierre Borgnat, Patrick Flandrin

Relabeling nodes according to the structure of the graph.

Ronan Hamon, Céline Robardet, Pierre Borgnat, Patrick Flandrin

1 Proposed method

1.1 Algorithm

We propose to solve this problem thanks to a two-step algorithm. The first step is based on the depth-first search algorithm and enables us to obtain a collection of independent paths. From a starting node with minimal value of closeness centrality, the algorithm jumps from another node according to the neighborhood of the considered node. The neighborhood is computed such that a node already taken into account in a path is not included. If one or more of his neighbors have a degree equal to 1, that means the neighbor node is only linked to the considered node, the node is added to the path and another neighbor is considered. If all neighbors have a degree greater than 1, the next node is chosen taking the highest value of a criterion based on the Jaccard index between neighborhood of the considered node and each of its neighbors. This criterion determines which neighbors is the most similar to the current node in order to stay in the same part of the graph. The other neighbors are stacked in a pile and the algorithm repeats the same procedure from the chosen node. When no neighbors are available, the procedure stops and the path is closed. A new path is opened and starts from the last node put in the pile and so on. At the end of step 1, there is a collection of paths which are independent i.e. no vertex is in two different paths.

The second step aims to aggregate these paths in order to minimize the cyclic bandwidth sum. The paths are considered following their decreasing lengths. The longest path is first considered and inserted into a empty list called labeling. The second longest path is then considered and inserted at all available indices in the labeling : for each insertion, a criterion based on the cyclic bandwidth sum is computed. The path is inserted definitively at the index which minimized this criterion. The algorithm goes on until the collection of paths is empty.

Algorithm 1 `Minimization_Cyclic_Bandwidth_Sum`

Require: $G = (V, E)$
Ensure: π a one-to-one and onto mapping of V to $\{0 \dots n-1\}$. L, labeling two piles. Paths a heap.

for all $u \in V$ **do**
    color[u] $\leftarrow$ white
    $\pi[u] \leftarrow$ `nil`
end for
centrality $\leftarrow$ `Closeness_Centrality`(G)
for all Connected components C of G **do**
    $V_C \leftarrow$ Vertices of C
    for all $u \in V_C$ **do**
        `Heap_Push`(S, (centrality[u], u))
    end for
    while S is not empty **do**
        $u_0 \leftarrow$ `Heap_Pop`(S)
        if color[u_0] = white **then**
            $P \leftarrow$ `Find_best_path`(u_0, C, color, centrality)
            `Heap_Insert`(Paths, (`length`(P), P))
        end if
    end while
    while Paths is not empty **do**
        path $\leftarrow$ `Max_Heap_Extract`(Paths)
        `Insert_path`(labeling, path, C, color)
        for all $u \in$ path **do**
            color[u] $\leftarrow$ black
        end for
    end while
end for
for $i \in [0, \dots, n-1]$ **do**
    $\pi[i] \leftarrow$`Index`(labeling, i)
end for

Algorithm 2 `Find_best_path`(u_0, C, color, centrality)

Ensure: P a pile. H a heap.

1: u $\leftarrow u_0$
2: **while** u $\neq$ -1 **do**
3:   `Push`(P, u)
4:   **for all** $v \in$ adj$[u]$ **do**
5:     **if** color$[v]$ = white **then**
6:       **if** degree$(v) = 1$ **then**
7:         `Push`(P, v)
8:         color$[v] \leftarrow$ gray
9:       **else**
10:         $j \leftarrow$ `Modified_Index_Jaccard`(u, v)
11:         $c \leftarrow$ centrality$[v]$
12:         `Heap_Insert`$(H, (v, c, j))$
13:       **end if**
14:     **end if**
15:   **end for**
16:   color$[u] \leftarrow$ gray
17:   **if** H not empty **then**
18:     $u \leftarrow$ `Min_Heap_Extract`(H)
19:   **else**
20:     $u \leftarrow -1$
21:   **end if**
22: **end while**
23: **return** P

Lines 8-17 concerns the step 1 of the algorithm whereas lines 18-25 concerns the step 2.

Algorithm 3 `Modified_Index_Jaccard`(u, v)

Ensure: nb_u, nb_v two piles.

1: **for all** $w \in$ adj$[u]$ **do**
2:   **if** color$[w]$ = white **then**
3:     `nb_u, w`
4:   **end if**
5: **end for**
6: **for all** $w \in$ adj$[v]$ **do**
7:   **if** color$[w]$ = white **then**
8:     `nb_v, w`
9:   **end if**
10: **end for**
11: **return** $\frac{\#(\text{nb\_u} \cup \text{nb\_w})}{\#(\text{nb\_u} \cap \text{nb\_w})}$

Algorithm 4 `Insert__path`(*labeling*, *path*, C, *color*)

```
best_index ← 0
best_cbs ← Criterion(labeling, path, C, color)
for all i ∈]0, ..., length(labeling)] do
    cbs ← Criterion(Insert(labeling, path, i), path, color)
    if cbs < best_cbs then
        best_index ← i
        best_cbs ← cbs
    end if
end for
return labeling ← INSERT(labeling, path, best_index)
```

Algorithm 5 `Criterion`(labeling, path, C, color)

```
CBS ← 0
n ← #V
for all u ∈path do
    for all v ∈ adj[u] do
        if color[v] = black then
            label_u ← Index(labeling, u)
            label_v ← Index(labeling, v)
            CBS ← CBS + min (|label_u − label_v|, n − |label_u − label_v|)
        end if
    end for
end for
return cbs
```