

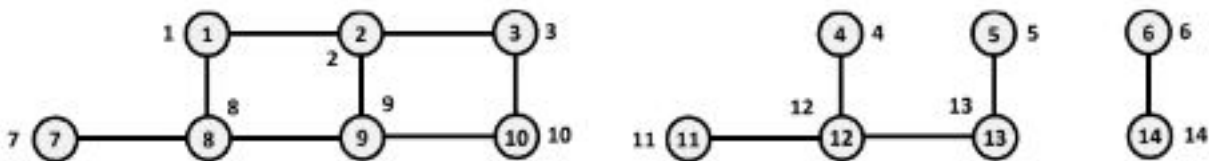
Componentes Conexas em Paralelo

Por João Curcio

Introdução

Seja $G = (V, E)$ um grafo representado por uma lista de adjacências, o objetivo deste relatório é descobrir em paralelo se G é conexo e se não for quantas e quais são suas componentes conexas.

Um grafo é conexo se para cada par (u, v) de seus vértices, existe um caminho com origem u e término v . Tome como exemplo o grafo G_1 a seguir, onde G_1 é numerado da esquerda pra direita, e de cima para baixo, com número contínuos de 1 até $|V|$ para todos os seus vértices.



Veja que a partir do vértice 1 não existe nenhum caminho para o vértice 4. Isso indica que G_1 não é conexo. Contudo, a partir do vértice 1 é possível alcançar os vértices 2, 3, 7, 8, 9, e 10, implicando que esses vértices formam um pedaço (ou componente) conexo do meu grafo G_1 . Visualmente podemos verificar que existem 3 componentes conexas no grafo G_1 , mas como determinar isso através de um algoritmo?

Abordagem Sequencial

Sequencialmente podemos resolver esse problema usando k buscas em profundidade, onde k é o número de componentes conexas do meu grafo. Dada uma raiz r , a busca em profundidade visita todos os vértices conectados a essa raiz (ou seja, acha uma componente conexa). Se após realizarmos uma busca em profundidade, procurarmos por vértices não visitados e repetirmos a busca caso algum exista, podemos encontrar todas as componentes conexas de um grafo. Isso gera o seguinte algoritmo:

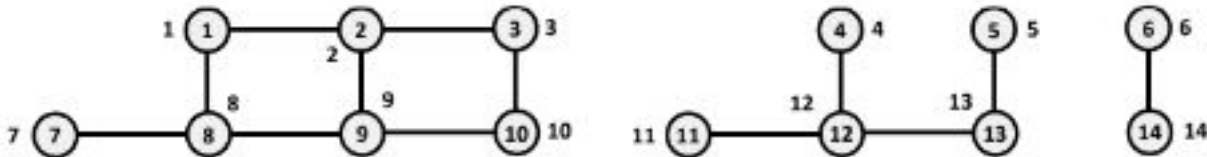
```
k ← 0
Para todo v em V, marque v como não visitado
Para todo v em V:
    Se v ainda não foi visitado
        k ← k + 1
        dfs(v)
```

O algoritmo acima possui uma complexidade de tempo de $O(m + n)$, onde m é o número de arestas e n é o número de vértices e será usado para comparar a eficiência do algoritmo paralelo implementado para este relatório.

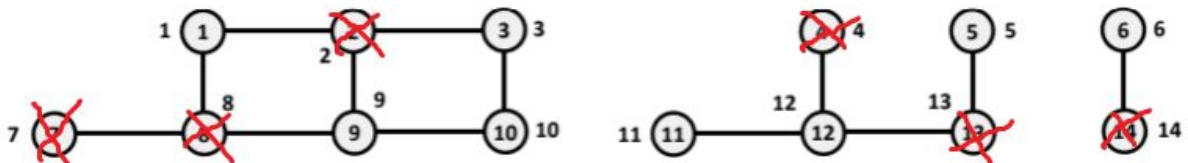
Abordagem Paralela

Existe um algoritmo chamado *Randomized Parallel Connected Components*^[1] que serve para achar as componentes conexas de um grafo. Esse algoritmo foi escolhido para ser implementado neste relatório.

Seja G_1 o grafo dado a seguir, podemos achar as componentes conexas de G_1 da seguinte forma:

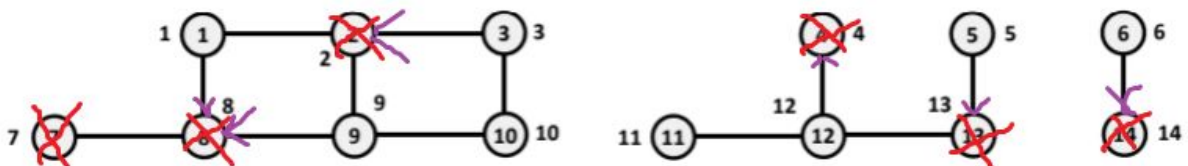


Passo 1: Para cada vértice v em V , jogue uma moeda aleatoriamente. Se der cara, marque esse vértice como um filho. Se der coroa, marque-o como pai..

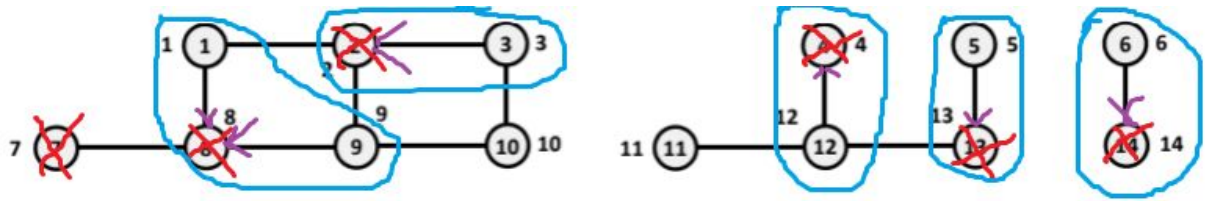


Os vértices marcados com um X vermelho são pais.

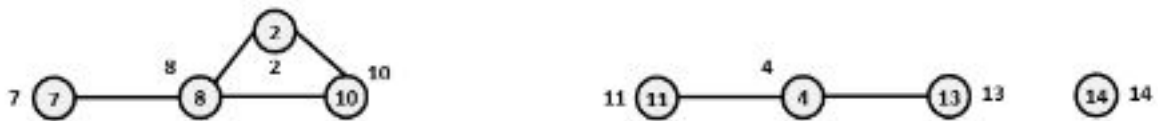
Passo 2: Para cada aresta (u, v) em E , se u é um pai e v é um filho, faça o filho apontar para o pai.



O conjunto formado por cada um dos pais e dos seus filhos formam um possível candidato a componente conexa.

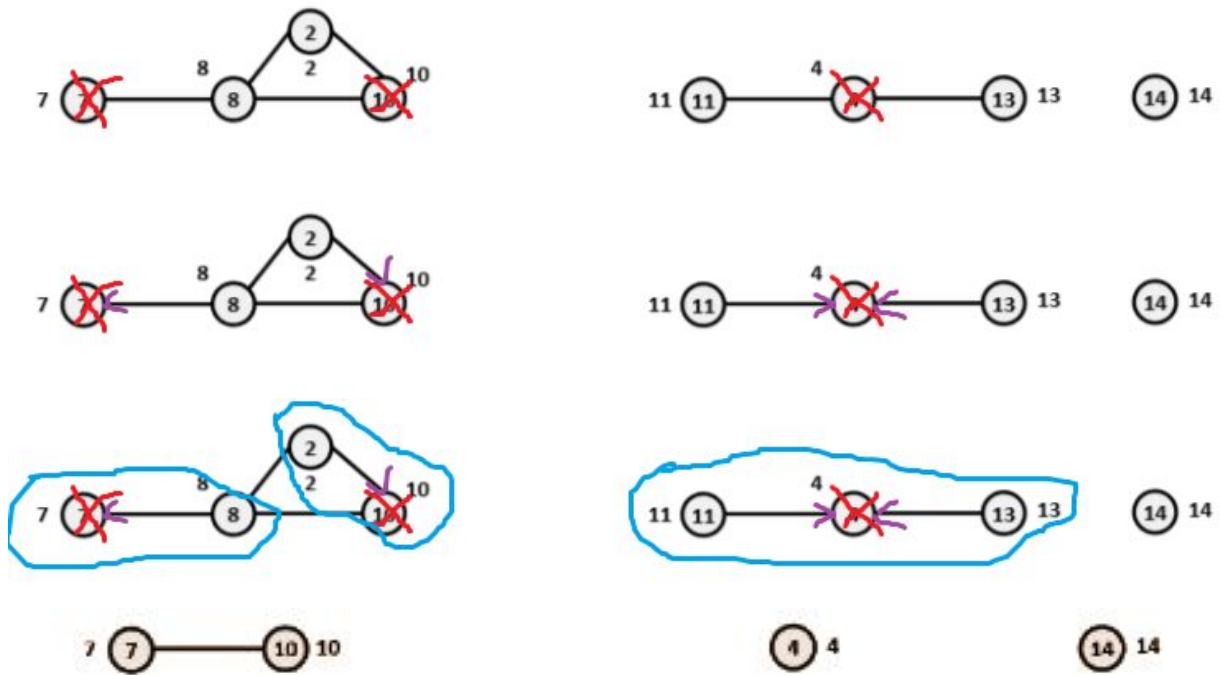


Passo 3: Junte cada pai com seus filhos em um único vértice. Isso é possível de ser feito com uma soma de prefixos.



Passo 4: Nesse novo grafo formado, se o número de arestas for 0 vá para o passo 5, senão repita o passo 1.

Iteração 2:



Iteração 3:





Passo 5: Quando o número de arestas for 0, a quantidade de componentes conexas foi encontrada e é possível mapear de volta para o grafo original.



Pseudo-código

Dado um grafo $G = (V, E)$, onde $|V| = n$, $|E| = m$. Seja L meu vetor resposta, C o vetor que guarda se um vértice é pai ou filho e S o vetor da soma de prefixos, o passo a passo explicado acima pode ser traduzido no seguinte pseudo-código:

randomizedCC(L, V, E)

1. **Inicialize** os arrays $C[1..n]$, $L[1..n] = V$, $S[1..m]$
2. **Se** $m = 0$, então retorne L
3. **Para** v de 1 até n em paralelo
 - a. $C[v] = \text{random}\{ \text{PAI}, \text{FILHO} \}$
4. **Para** (u, v) em E em paralelo
 - a. **Se** $C[u]$ é um pai e $C[v]$ é um filho
 - i. $L[u] = L[v]$
5. **Para** i de um até m em paralelo
 - a. **Se** $L[E[i].u] \neq L[E[i].v]$
 - i. $S[i] = 1$
 - b. **Senão**
 - i. $S[i] = 0$
6. $S = \text{soma_prefixos}(S, +)$
7. **Inicialize** $F[1..|S[n]|]$
8. **Para** i de um até m em paralelo
 - a. **Se** $L[E[i].u] \neq L[E[i].v]$
 - i. $F[S[i]] = (L[E[i].u], L[E[i].v])$
9. $L = \text{randomizedCC}(L, V, E)$
10. **Para** (u, v) em E em paralelo
 - a. **Se** $v = L[u]$

i. $L[u] = L[v]$
11. Retorne L

Implementação

Usando cilk++, o código acima foi implementado da seguinte forma:

```
void randomizedConectedComponents(vector<Vertex> &L, vector<Vertex> &V, vector<Edge> &E){
    if (E.size() == 0) return;
    int n = V.size(); int m = E.size();
    vector<Vertex> tmpS(m);

    Group *C = new Group[n+1];
    cilk_for(int i = 0; i < n; i++){
        C[V[i]] = ((double)rand()/RAND_MAX > 0.5) ? Group::PARENT : Group::CHILD;
    }

    cilk_for(int i = 0; i < m; i++){
        if (C[E[i].u] == Group::CHILD && C[E[i].v] == Group::PARENT){
            L[E[i].u] = L[E[i].v];
        }
    }

    cilk_for(int i = 0; i < m; i++){
        if ( L[E[i].u] != L[E[i].v] ) tmpS[i] = 1; else tmpS[i] = 0;
    }

    vector<Vertex> S(m);
    prefixSum(tmpS, S);
    vector<Edge> F(S[m-1]);

    cilk_for(int i = 0; i < m; i++){
        if(L[E[i].u] != L[E[i].v]){
            F[S[i]-1].u = L[E[i].u];
            F[S[i]-1].v = L[E[i].v];
        }
    }

    randomizedConectedComponents(L, V, F);

    cilk_for(int i = 0; i < m; i++){
        if(E[i].v == L[E[i].u]) L[E[i].u] = L[E[i].v];
    }
}
```

Onde a função `prefixSum` é implementada também em paralelo da seguinte forma:

```

void prefixSum(vector<int> &S, vector<int> &res){
    int n = S.size();
    vector<int> Sstar(n/2);
    vector<int> Sout(n/2);

    if(n == 1){
        res[0] = S[0];
        return;
    }

    cilk_for(int i = 0; i < n/2; i++){
        Sstar[i] = S[2*i] + S[2*i + 1];
    }

    prefixSum(Sstar, Sout);

    cilk_for(int i = 0; i < n; i++){
        if(i == 0){
            res[i] = S[i];
        }else if(i%2 != 0){
            res[i] = Sout[i/2];
        }else{
            res[i] = Sout[(i-1)/2] + S[i];
        }
    }
}

```

O programa implementado espera um arquivo de texto com o grafo $G = (V, E)$. A primeira linha do arquivo de texto possui a n e m separados por um espaço. Depois disso m linhas se seguem onde cada linha possui dois números indicando uma aresta no grafo.

O código completo pode ser encontrado em <https://github.com/johncurcio/AIPaCa>

Trabalho Total

Cada vez que o passo 3 do algoritmo é executado, o grafo é reduzido mais ou menos pela metade. Dessa forma, a cada iteração do algoritmo, estamos trabalhando com a metade da iteração anterior. Além disso, o grafo está representado por uma lista de adjacências, que tem complexidade $O(n + m)$ para ser percorrida. Portanto, o trabalho total desse algoritmo é de $O((n+m)*\log n)$.

Testes e Resultados

Para testar o algoritmo implementado foram usados os grafos encontrados em:

1. <https://snap.stanford.edu/data/wiki-Vote.html>
2. <https://snap.stanford.edu/data/email-Enron.html>
3. <https://snap.stanford.edu/data/roadNet-CA.html>

E o algoritmo para encontrar componentes conexas sequencialmente disponível em: <http://www.geeksforgeeks.org/connected-components-in-an-undirected-graph/>. Após algumas pequenas modificações no algoritmo do Geeks for Geeks para ficar no padrão do meu algoritmo, os seguintes resultados foram achados:

Arquivo	Meu algoritmo	Geeks for Geeks	Vertices x Arestas CC
wiki-Vote.txt	0.728755 s	0.00716278 s	7115 x 103689 903
email-Enron.txt	0.232372 s	0.0317371 s	36692 x 183831 1067
roadNet-CA.txt	16.5865 s	+ 10 min	1965206 x 2766607 8685

CC = número de componentes conexas

Os resultados mostram que conforme o tamanho do grafo vai crescendo, a solução sequencial se torna cada vez menos eficiente enquanto a paralela ainda é uma solução viável. Grafos monstruosos, como o do orkut (encontrado em <https://snap.stanford.edu/data/com-Orkut.html>) não terminaram de rodar nem sequencialmente nem paralelamente nos testes realizados na minha máquina de 8 cores.

Apesar de a solução que implementei ser eficiente, ela não chega a ser melhor que a sequencial em todos os casos. Veja que nos primeiros dois casos a versão paralela chega a ser cerca de 10 vezes pior que a sequencial. A versão paralela parece ficar mais eficiente quando o número de componentes conexas e o tamanho do grafo aumenta em comparação a versão sequencial.

Referências

- [1] <http://www3.cs.stonybrook.edu/~rezaul/Spring-2013/CSE638/CSE638-lectures-10-11.pdf>
- [2] Joseph Jajá, Introduction to Parallel Algorithms