INE5413 - Atividade 2

Gabriel da Silva Cardoso e Hans Buss Heidemann

Maio 2022

1 Código-fonte

O código utilizado esta disponível no Github em https://github.com/Cardoz-0/Graph-Theory-Study

2 Representação

Para esta entrega a biblioteca foi reescrita em Rust, mudando o paradigma utilizado que era programação orientada a objeto e passa a ser programação estruturada.

Uma das principais features de Rust é a segurança de memória onde código com acesso inválido a memória não será compilado, além disso é uma linguagem sem garbage collection, onde variáveis serão invalidadas assim que saírem do escopo. Isto cria várias dificuldades para a implementação de um grafo, que é em essência uma *(*linked list) e por isso é insegura. Para evitar que as variáveis saiam de escopo enquanto ainda queremos acessa-las utilizamos *(*Reference Counters), que só permitirão que uma variável saia de escopo se sua contagem de referencias zerar, toda vez que utilizamos Rc::clone(&x) estamos somando 1 para a contagem da referencia de x.

Como o uso de ponteiros é desincentivado na linguagem e o uso de referências (que são ponteiros inteligentes) causaria bastante complicação, optamos por separar os vertices e edges como listas separadas que armazenam o index de cada variavel no vetor. Essa não é a forma linguística de lidar com o problema, mas agiliza a implementação e diminui a complexidade.

Nossa representação de grafo ficou da seguinte maneira:

```
struct Vertex {
    id: usize,
    name: String,
}

struct Edge {
    w: f32,
    v: Rc<Vertex>,
```

```
    u: Rc<Vertex>,
}

struct Graph {
    verts: Vec<Rc<Vertex>>,
    edges: Vec<Rc<Edge>>,
}
```

3 Componentes Fortemente Conexas

Para detectar SCCs o algoritmo de Tarjan que prevê o uso de uma stack com os nodos que precisam ser processados. Este método geraria um nivel maior de complexidade porque as posições dos nodos nessa stack não precisariam corresponder com a posição dos vertices no vetor do grafo. Para isso, foi criado um struct que mantém informações sobre cada nodo e se ele deveria estar na stack e esta estrutura é posta numa lista na mesma ordem que a lista de nodos do grafo.

Como também precisamos saber se visitamos um nodo adicionamos ao struct mais uma estrutura, que é determina se o nodo já foi visitado. Ela é separada porque é utilizada para estruturas de outros algoritmos, podendo assim, reaproveitar código.

```
struct AuxNodeSCC {
    node: VisitableNode,
    on_stack: bool,
    lowest: usize,
}


struct VisitableNode {
    v: Rc<Vertex>,
    visited: bool,
}
```

4 Ordenação Topológica

A saída de uma ordenação topologíca pode ser um pouco complicada de fazer em Rust, visto que inicializar um array de tamanho N seria perigoso. Para evitar esse problema atrelamos uma posição para cada nodo em um struct e editaremos ela, sem nunca deixar memória corrompida exposta.

```
struct OrderedNode {
    visitable: VisitableNode,
    pos: usize,
}
```

Note que ele também utiliza um VisitableNode, porque também precisa manter informações se um nodo já foi visitado.

5 Prim

O algoritmo de Prim precisa de uma priority queue, para isso, usaremos uma fila ao invés de um simples vetor. A std do Rust tem uma implementação de fila, utilizaremos ela: VecDeque. Apesar da necessitar de uma PQ, ele precisa de uma estrutura muito mais simples que só precisa manter informações se um nodo já foi visitado. Para isso utilizamos mais uma vez o struct de VisitableNode.