Algoritmo A^*

O problema que um algoritmo A^* procura resolver é encontrar uma sequência de passos (= caminho) que leva de uma dado nó inicial `s` a um destino `t` em um grafo.

O grafo é tipicamente **muito grande** e muitas vezes não pode ser dado explicitamente. Ele é dado através de uma função ou iterador `viz()` que para cada nó fornece os nós que são seu vizinhos.

Frenquentemente os nós representam um conjunto de configurações possíveis.

O algoritmo é iterativa e no início de cada iteração temos uma partição dos seus nós em dois conjuntos:

- `vistos`: nós que já foram encontrados pelo algoritmos
- `não_vistos`: nós que o algoritmo não tem a menor ideia que existem.

O conjunto de nós `vistos` é ainda particionado em

- `vistos_não_examinados`: nós que foram descobertos, mas que ainda o algoritmo não terminou olhou com cuidado.
- `vistos_examinados`: nós que foram descoberto e o algoritmo não tem mais nada a fazer com eles.

O conjunto de nós `vistos_não_examinados` é o coração do algoritmo. Esse conjunto é mantido em uma fila priorizada que é o segredo para o desempenho do algoritmo.

Como tipicamente o número de nós é muito grande, infelizmente, não é possível manter o conjunto de nós examinados explicitamente. Desta forma, a divisão dos nós `vistos` em `vistos_não_examinados` e `vistos_examinados` pode não ser muito séria, pode não ser uma partição. Podemos ter nós `vistos` que estão em `vistos_não_examinados` e `vistos_examinados` o que atrapalha um pouco a metáfora e o consumo de tempo e espaço...

Descrição

Eis um descrição do algoritmo em um pseudo-código meia boca. Notem que os atores principais são, em ordem que aparecem no algoritmo:

- `s` : nó inicial
- `vistos`: conjunto de nós descobertos
- `vistos_não_examinados`: fila priorizada para manter o conjunto de nós vistos e que ainda precisam ser examinados
- `t`: nó destino
- `viz()`: iterador
- `vistos_examinados`: conjunto de nós vistos e examinados. Pode não ser muito sério e dependendo da aplicação não pode ser mantido.

Pré-condição: descrição supõe que é possível chegarmos do nó `s` ao nó `t`:

```
// inicialize a fila priorizada
vistos.insert(s)
vistos_não_examinados.insert(s)
while (true) {
    // 1. remova o primeiro elemento da fila
    // u é o nosso batedor
    u = vistos_não_examinados.delMin(); // dependendo da aplicação, pode ser delMax()

    // 2. examine u
    // 2.1 chegamos no nosso destino
    if (u == t): break;

    // 2.2 visite todos os vizinhos de u
    for (Node v: u.viz()) {
        if (!vistos.contains(v)) {
            vistos.insert(v);
            vistos_não_examinados.insert(v);
     }
    }

    // 2.3 acabamos de examinar u
    vistos_examinados.insert(u) // == vistos - vistos_não_examinados


// construa o caminho inverso de u até s
p = ...

// retorne esse caminho
return p
```