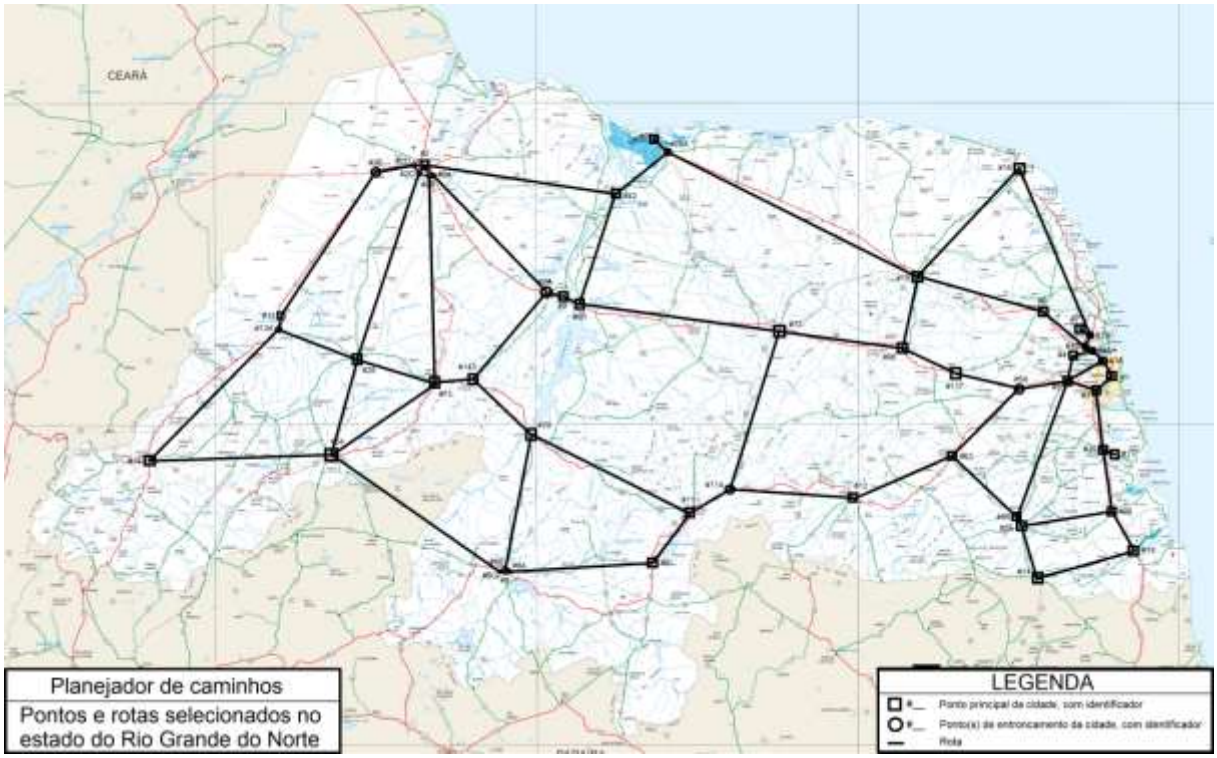
PLANEJADOR DE CAMINHOS
PROFESSOR: ADELARDO ADELINO DANTAS DE MEDEIROS



O objetivo é utilizar a biblioteca STL de C++ para desenvolver um programa, baseado no algoritmo A*, para determinar o caminho mais curto entre dois pontos (origem e destino) em um mapa. O mapa contém uma série de pontos, com conexões entre alguns deles (rotas). Cada rota tem um comprimento associado.

CAMINHO DE MENOR CUSTO EM GRAFO

O algoritmo A* encontra o caminho de menor custo em um grafo no qual a transição entre nós conectados tem um custo associado. O A* mantém um conjunto dos nós já visitados (***Fechado***) e um conjunto dos nós ainda não analisados (***Aberto***). No início, ***Fechado*** está vazio e ***Aberto*** contém apenas o nó de origem.

A cada passo, o A* retira um nó de ***Aberto*** para ser analisado. O algoritmo coloca esse nó em ***Fechado***, verifica se ele é o destino e, se não for, gera os sucessores, que correspondem aos nós que estão conectados ao nó que está sendo analisado. Os sucessores válidos são colocados em ***Aberto***. O algoritmo prossegue até que o destino seja alcançado.

PLANEJADOR DE CAMINHOS

Neste planejador de caminhos, cada ponto está conectado a um número variável de outros pontos, que são aqueles para os quais existe uma rota entre eles. Os pontos conectados são os únicos diretamente atingíveis a partir do ponto anterior.

Cada nó do grafo está associado a um ponto do mapa e ao caminho que levou da origem até ele, contendo as seguintes informações:

- A identificação do ponto associado ao nó.
- A identificação da rota que trouxe do nó anterior até ele.
- O custo (comprimento) do caminho da origem até o nó.

Cada nó n_k tem um custo associado. Esse custo é denominado de custo passado (***g***). Ele é igual ao custo passado do seu nó antecessor n_{k-1} adicionado ao custo da movimentação do antecessor até ele, que é igual ao comprimento da rota que os interliga:

$$g(n_k) = g(n_{k-1}) + \text{comprimento\_rota}(n_{k-1}, n_k)$$

O nó inicial (a origem do caminho) tem custo passado nulo (***g*** = 0). O comprimento total do caminho será o custo do seu último nó, ou seja, o custo passado (***g***) do nó destino.

INSERÇÃO DE NOVOS NÓS

Para garantir que o caminho mais curto seja encontrado, o nó retirado de ***Aberto*** deve ser sempre o de menor custo. Por isso, os nós em ***Aberto*** são ordenados em ordem crescente de custo e retira-se sempre o primeiro deles.

Como o conjunto estava ordenado no passo anterior, a cada inserção procura-se o local do novo nó no conjunto, que é:

- **antes** do primeiro nó de custo **maior** que ele, caso exista; ou
- ao final do contêiner, caso o novo nó tenha custo maior ou igual que todos os atuais.

Não se deve reordenar o conjunto inteiro a cada vez que um novo nó é inserido.

Cada ponto do mapa só pode ser representado por um único nó em ***Aberto*** ou em ***Fechado***. Quando um sucessor é gerado, verifica-se se um nó que representa o mesmo ponto (mesma id) já não existe em um dos conjuntos:

- Caso o sucessor seja igual (mesma id) que um nó em ***Fechado***, ignora-se o sucessor.
- Caso o sucessor seja igual (mesma id) que um nó em ***Aberto***:
  - Caso o sucessor tenha custo igual ou maior que o nó existente, ignora-se o sucessor.
  - Caso o sucessor tenha custo menor que o nó existente, exclui-se o nó existente de ***Aberto*** e coloca-se o sucessor na posição apropriada em ***Aberto***.

ESTIMATIVA DO CUSTO FUTURO

Ao ordenar os nós apenas pelo custo passado, está sendo utilizado o algoritmo de Dijkstra. Isso garante que o caminho mais curto será encontrado primeiro, mas o algoritmo pode ser lento para encontrar o caminho ótimo.

Para acelerar a busca, o algoritmo A* ordena os nós pelo custo total ***(f)***, que é a soma do custo passado ***(g)*** com o custo futuro ***(h)***:

$$f(n_k) = g(n_k) + h(n_k, \text{destino})$$

O custo futuro ***(h)*** é baseado em uma estimativa aproximada (heurística) do custo adicional para ir diretamente do nó que está sendo analisado até o nó destino. Garante-se que o caminho mais curto será encontrado se for utilizada uma *heurística otimista*, com valor sempre menor ou igual do que o custo real até o destino. Caso não se use heurística (***h*** = 0), o A* recai no algoritmo de Dijkstra.

Neste projeto, será utilizada como heurística a distância euclidiana entre os dois pontos, pois o comprimento da reta entre dois pontos será sempre menor do que o comprimento do caminho efetivo que precisará ser percorrido usando as rotas existentes. O cálculo da distância euclidiana entre os pontos pode ser feito a partir da latitude e da longitude dos pontos, que são conhecidas.

A distância *d* entre dois pontos, dadas suas latitudes (φ_1, φ_2) e longitudes (λ_1, λ_2), pode ser calculada de forma aproximada, considerando-se que a Terra uma esfera, pela fórmula de haversine:

$$d = R \cos^{-1}[\sin\varphi_1 \sin\varphi_2 + \cos\varphi_1 \cos\varphi_2 \cos(\lambda_1 - \lambda_2)]$$

onde R é o raio da Terra (6371 km) e os ângulos de latitude e longitude devem ser expressos em radianos para que o cálculo computacional seja correto.

IMPLEMENTAÇÃO

Para auxiliar na implementação do planejador de caminhos, algumas classes e funções, parcialmente ou totalmente declaradas e implementadas, já estão disponibilizadas no SIGAA:

- `IDPonto`: identificação única (id) de um ponto no mapa.
  - `t`: cadeia de caracteres, iniciada por `#`.

  Já está declarada e implementada.
- `IDRota`: identificação única (id) de uma rota no mapa.
  - `t`: cadeia de caracteres, iniciada por `&`.

  Já está declarada e implementada.
- `Ponto`: representa um ponto no mapa.
  - `id`: identificador (`IDPonto`).
  - `nome`: denominação usual (`string`)
  - `latitude`, `longitude`: coordenadas geográficas (em graus).

  Parcialmente declarada e implementada.
- `Rota`: representa uma rota existente entre 2 pontos.
  - `id`: identificador (`IDRota`).
  - `nome`: denominação usual (`string`).
  - `extremidade`: as 2 `id`'s dos pontos extremos da rota (2 x `IDPonto`).
  - `comprimento`: comprimento (em km).

  Parcialmente declarada e implementada.
- `Noh`: necessária para implementação dos contêineres ***Aberto*** e ***Fechado*** do A*. Não está declarada nem implementada.
- `Planejador`: planejador de caminhos
  - `pontos`: conjunto de pontos do mapa.
  - `rotas`: conjunto de rotas do mapa.

  Todas as funções membro da classe `Planejador` já estão declaradas e quase todas implementadas, com exceção de:
  - `getPonto`: incompleta.
  - `getRota`: incompleta.
  - `ler`: incompleta.
  - `calculaCaminho`: função que deve implementar o A*. Deve ser implementada quase que inteiramente, seguindo o algoritmo que se encontra a seguir.
- Função `main`, uma interface de acesso às funcionalidades da classe `Planejador`.

Tendo em vista que um dos objetivos principais do projeto é praticar a utilização das estruturas de dados básicas e dos algoritmos fundamentais da biblioteca STL, algumas regras devem ser **obrigatoriamente** seguidas:

- As classes (`IDPonto`, `IDRota`) e funções (`main`) que já estão completamente declaradas e implementadas **NÃO** devem ser modificadas de nenhuma forma (por alteração, supressão ou acréscimo).
- As classes (`Ponto`, `Rota`, `Planejador`) que estão parcialmente declaradas e/ou implementadas devem receber acréscimos, mas as partes já disponibilizadas **NÃO** devem ser modificadas ou suprimidas.
- Deve(m) ser utilizado(s) o(s) contêiner(es) mais adequado(s) dentre os contêineres sequenciais (`vector`, `deque` ou `list`) e/ou as adaptações simples desses contêineres (`stack` ou `queue`); **NÃO** devem ser utilizados os contêineres baseadas em *heaps* (`priority_queue`) ou em árvores binárias de busca (`set`, `multiset`, `unordered_set`, `map`, `multimap`, `unordered_map`, etc.), pois são estruturas de dados não estudadas com profundidade nessa disciplina introdutória.
- Sempre que possível, os algoritmos genéricos da STL devem ser utilizados e **NÃO** substituídos por um trecho de código similar, implementado pelo próprio programador, utilizando laços de controle. Por exemplo, **NÃO** faça:

```
for (...) {
  if (...) {
    ... // O que precisa fazer
  }
}
```

  quando poderia fazer:

```
itr = algoritmo_STL( ... );
if (itr != ___.end()) {
  ... // O que precisa fazer
}
```

- Caso possa ser empregado mais de um algoritmo STL, deve ser utilizado o que implementa mais diretamente a função desejada e/ou requer menos programação adicional. Por exemplo, **NÃO** utilize `find_if` com uma função de teste quando um `find` simples poderia resolver.

ALGORITMO A*

```
// TIPOS DE DADOS

// Identificador de um ponto
IDPonto:
  string    t: #__
// Identificador de uma rota
IDRota:
  string    t: &__
// Características de um ponto
Ponto:
  IDPonto   id: identificador
  string    nome: denominação
  double    latitude: em graus
  double    longitude: em graus
// Características de uma rota
Rota:
  IDRota    id: identificador
  string    nome: denominação
  IDPonto   extremidade[2]: id das
            extremidades da rota
  double    comprimento: em km
// Noh: elementos dos contêineres
// Aberto e Fechado
Noh:
  IDPonto   id_pt: id do ponto
  IDRota    id_rt: id da rota do
            antecessor até o ponto
  double    g: custo passado
  double    h: custo futuro
  double    função f() = g+h :
            custo total

// DADOS DE ENTRADA

// Pontos do mapa
contêiner<Ponto>  pontos
// Rotas do mapa
contêiner<Rota>   rotas
// ID dos pontos de origem e
// destino do caminho
IDPonto     id_orig,
            id_dest

// DADOS DE SAÍDA

// N° final de nós nos contêineres
// Aberto e Fechado
int         NumAberto,
            NumFechado
// Sequência de rotas e pontos do
// caminho encontrado
contêiner< par<IDRota,IDPonto> >
            caminho
// Comprimento do caminho
// encontrado
double      compr
```

```
// ALGORITMO A*

// Obtém pontos de origem e destino
pt_orig ← Ponto cuja id=id_orig
pt_dest ← Ponto cuja id=id_dest

// atual ← Noh inicial
atual.id_pt ← id_orig
atual.id_rt ← ∅  // Rota()
atual.g ← 0.0
atual.h ← haversine(pt_orig,
                    pt_dest)

// Inicializa os conjuntos de Noh's
Aberto ← ∅
Fechado ← ∅
incluir(atual,Aberto)

// Laço principal do algoritmo
REPITA
|
| // Lê e exclui o 1° Noh (o de
| // menor custo) de Aberto
| atual ← primeiro(Aberto)
| excluir_primeiro(Aberto)
|
| // Inclui "atual" em Fechado
| // (no início ou outra posição)
| incluir(atual,Fechado)
|
| // Expande se não é solução
| SE (atual.id_pt ≠ id_dest)
| |
| | // Gera sucessores de "atual"
| | REPITA
| | |
| | | // Busca "rota_suc", próxima
| | | // Rota conectada a "atual"
| | | rota_suc ← próxima Rota com
| | |  uma das extremidades = atual
| | |
| | | // Achou uma Rota?
| | | SE ( EXISTE(rota_suc) )
| | | |
| | | | // Gera Noh sucessor "suc"
| | | | suc.id_pt ← extremidade de
| | | |            rota_suc ≠ atual
| | | | // Ponto do Noh "suc"
| | | | pt_suc ← Ponto cuja
| | | |          id = suc.id_pt
| | | | suc.id_rt ← rota_suc.id
| | | | suc.g ← atual.g +
| | | |        rota_suc.comprimento
| | | | suc.h ← haversine(pt_suc,
| | | |                   pt_dest)
| | | |
| | | | // Inicialmente, assume que
| | | | // não existe Noh igual a
| | | | // "suc" nos contêineres
| | | | eh_inedito ← TRUE
```

```
| | | |
| | | | // Procura Noh igual a
| | | | // "suc" em Fechado
| | | | old ← buscar(suc.id_pt,
| | | |              Fechado)
| | | |
| | | | // Achou algum Noh?
| | | | SE ( EXISTE(old) )
| | | | |
| | | | | // Noh já existe
| | | | | eh_inedito ← FALSE
| | | | |
| | | | CASO CONTRÁRIO
| | | | |
| | | | | // Procura Noh igual a
| | | | | // "suc" em Aberto
| | | | | old ← buscar(suc.id_pt,
| | | | |              Aberto)
| | | | |
| | | | | // Achou algum Noh?
| | | | | SE ( EXISTE(old) )
| | | | | |
| | | | | | // Menor custo total?
| | | | | | SE (suc.f() < old.f())
| | | | | | |
| | | | | | | // Exclui anterior
| | | | | | | excluir(old, Aberto)
| | | | | | |
| | | | | | CASO CONTRÁRIO
| | | | | | |
| | | | | | | // Noh já existe
| | | | | | | eh_inedito ← FALSE
| | | | | | |
| | | | | | FIM SE (suc.f() < ...
| | | | | |
| | | | | FIM SE ( EXISTE(old) )
| | | | |
| | | | FIM SE ( EXISTE(old) )
| | | |
| | | | // Já existe?
| | | | SE ( eh_inedito )
| | | | |
| | | | | // Acha "big", 1° Noh de
| | | | | // Aberto com custo total
| | | | | // f() maior que o custo
| | | | | // total f() de "suc"
| | | | | big ← maior_que(suc.f(),
| | | | |                 Aberto)
| | | | |
| | | | | // Insere "suc" em Aberto
| | | | | // antes de "big"
| | | | | inserir(suc,big,Aberto)
| | | | |
| | | | FIM SE ( eh_inedito )
| | | |
| | | FIM SE ( EXISTE(rota_suc) )
| | |
| | // Repita enquanto há rotas com
| | // "atual" em uma extremidade
| | ENQUANTO ( EXISTE(rota_suc) )
| |
| FIM SE (atual.id_pt ≠ ...
|
| // Repita enquanto não encontrou
| // solução e há nós em Aberto
ENQUANTO ( NÃO(VAZIO(Aberto)) E
    (atual.id_pt ≠ id_dest) )

// Calcula n° de nós da busca
NumAberto ← tamanho(Aberto)
NumFechado ← tamanho(Fechado)

// Inicialmente, caminho vazio
caminho ← ∅

// Encontrou solução ou não?
SE (atual.id_pt ≠ id_dest)
|
| // Não existe solução
| compr ← -1.0
|
CASO CONTRÁRIO
|
| // Calcula comprimento do caminho
| compr ← atual.g
|
| // Refaz o caminho, procurando
| // Nohs antecessores em Fechado.
| ENQUANTO (atual.id_rt ≠ ∅)
| |
| | // Acrescenta par atual no topo
| | // (início) de "caminho"
| | incluir_topo( par(atual.id_rt,
| |                   atual.id_pt),
| |               caminho )
| |
| | // Recupera o antecessor.
| | //
| | // Obtém "rota_ant", Rota que
| | // levou até "atual".
| | rota_ant ← Rota cuja
| |            id = atual.id_rt
| | // Calcula id do antecessor
| | id_pt_ant ← extremidade de
| |             rota_ant ≠ atual
| |
| | // Procura Noh igual a
| | // "id_pt_ant" em Fechado
| | atual ← buscar(id_pt_ant,
| |                Fechado)
| |
| FIM ENQUANTO (atual.id_rt ≠ ∅)
|
| // Acrescenta origem no topo
| // (início) de "caminho"
| incluir_topo( par(atual.id_rt,
|                   atual.id_pt),
|               caminho )
|
FIM SE (atual.id_pt ≠ id_dest)
```