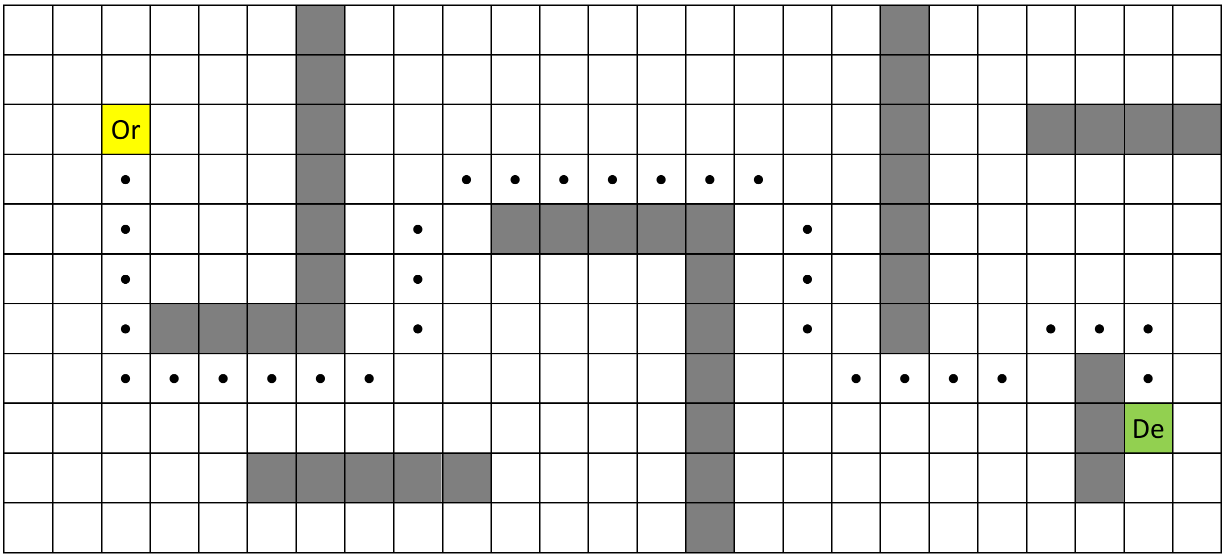
UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO

PLANEJADOR DE CAMINHOS EM LABIRINTOS

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 de menor custo entre células de origem e destino, dentro de um ambiente com obstáculos.

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:

- 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`, `map`, etc.), pois são estruturas de dados não estudadas nessa disciplina introdutória.
- **NÃO** devem ser utilizados recursos avançados de C++ não abordados na disciplina, tais como expressões lambda[1].
- Sempre que possível, os algoritmos genéricos da STL (`for_each`, `find`, `sort`, etc.) devem ser utilizados e **NÃO** substituídos por um trecho de código similar implementado pelo próprio programador utilizando laços de controle (`for`, `while`, etc.).

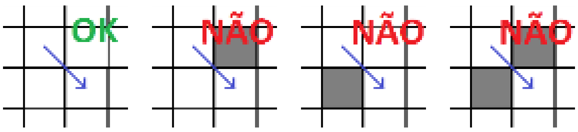
CAMINHO DE MENOR CUSTO EM GRAFO

O algoritmo A* encontra o caminho de menor custo em um grafo no qual a transição entre nós conectados tem um custo associado. No exemplo, cada célula do mapa é um nó do grafo. As células estão conectadas às 8 células vizinhas e o custo de ir de uma célula para outra é a distância entre os centros das células:

- 1, para movimento horizontal ou vertical; e
- $\sqrt{2}$, se o movimento for diagonal.

Só é possível o movimento de uma célula para uma célula vizinha se:

1) A célula vizinha estiver livre; e
2) Caso o movimento seja em diagonal, nenhuma das "quinas" seja um obstáculo.



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***, coloca em ***Fechado***, verifica se ele é o destino e, se não for, gera até 8 sucessores, que correspondem às possíveis direções de movimen-

[1] Não se preocupe se não souber do que se trata.

tação. Os sucessores válidos são colocados em ***Aberto***. O algoritmo prossegue até que o destino seja alcançado.

Cada nó tem um custo associado, que é o tamanho do caminho percorrido da origem até ele. Esse custo é denominado de custo passado ***(g)***. Ele é igual ao custo passado do seu antecessor mais o custo da movimentação do antecessor até ele:

$$g(n_k) = g(n_{k-1}) + \text{custo}(n_{k-1}, n_k)$$
$$\text{custo}(n_{k-1}, n_k) = \begin{cases} \sqrt{2}, \text{se diagonal} \\ 1, \text{caso contrário} \end{cases}$$

O comprimento (custo) do caminho é o custo do último nó do caminho, ou seja, o custo do nó destino. A profundidade do caminho é a quantidade de nós percorridos para chegar da origem (profundidade 0) até o destino (profundidade do caminho). Sempre vale a relação:

Profundidade ≤ Comprimento

No exemplo da figura:

- Comprimento (custo) do caminho: 27x1 + 5x$\sqrt{2}$ = 34,07
- Profundidade do caminho: 32

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

Como o conjunto estava ordenado no passo anterior, o que é feito para manter a ordem é, a cada inserção, procurar (com `find_if` ou outro algoritmo STL adequado) o local do novo nó no conjunto (antes do primeiro nó de custo maior que ele ou no fim do container) e inseri-lo (com `insert`) nessa posição. **Não é necessário, e constitui uma má técnica de programação, reordenar o conjunto inteiro (`sort`) a cada vez que um novo nó é inserido**.

Cada célula do mapa só pode ser representada por um único nó em ***Aberto*** ou em ***Fechado***: ao ser gerado um novo nó que represente uma mesma célula (mesmas coordenadas, ou seja, mesmas linha e coluna) já presente em um desses conjuntos, deve-se decidir qual dos nós será mantido: o anteriormente existente ou o novo. Da mesma forma, nenhuma célula pode estar simultaneamente representada em ambos os conjuntos ***Aberto*** e ***Fechado***.

Quando um sucessor é gerado, verifica-se se um nó que representa a mesma célula já não existe em ***Aberto*** ou em ***Fechado***. Caso exista, significa que foi encontrado outro caminho para chegar ao mesmo nó. Nesse caso, deve ser mantido o caminho de menor custo:

- Caso o sucessor tenha custo maior que o nó existente, ignora-se o novo sucessor.
- Caso o sucessor tenha custo menor que um nó em ***Fechado***, exclui-se o nó de ***Fechado*** e coloca-se o sucessor em ***Aberto***.
- Caso o sucessor tenha custo menor que um nó em ***Aberto***, exclui-se o nó de ***Aberto*** e coloca-se o sucessor em ***Aberto***.

Ordenando os nós apenas pelo custo passado, o algoritmo A* se torna equivalente ao algoritmo de Dijkstra, que se assemelha a uma busca em largura: são analisados primeiro todos os vizinhos da origem, depois todos os vizinhos dos vizinhos e assim sucessivamente. Isso garante que o caminho mais curto será encontrado primeiro, mas pode ser lento.

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)***. O custo futuro é baseado em uma estimativa (heurística). O caminho mais curto será encontrado se o valor da heurística ***h*** for sempre menor ou igual do que o custo real para mover até o destino. Caso não se use heurística (***h*** = 0) o A* recai no algoritmo de Dijkstra.

POSSÍVEIS HEURÍSTICAS

- Distância Manhattan: usada quando só se move nas 4 direções principais:
$$h = |\Delta x| + |\Delta y|$$
- Distância diagonal: usada quando se move nas 8 direções vizinhas (é nosso caso):
$$h = \sqrt{2} \cdot \min(|\Delta x|, |\Delta y|) \\ + \text{abs}(|\Delta x| - |\Delta y|)$$
- Distância Euclidiana: usada quando os movimentos em todas as direções são possíveis, não só para o centro das células:
$$h = \sqrt{\Delta x^2 + \Delta y^2}$$

ALGORITMO A*

```
// Dados de entrada:
origem: célula inicial
destino: célula final
mapa: células livres e obstáculos

// Dados de saída:
compr: comprimento do caminho
prof: profundidade do caminho
NA: n° final de nós em Aberto
NF: n° final de nós em Fechado

// Tipo de dado Coord
// Coordenadas de uma célula
Coord:
  int lin: linha
  int col: coluna

// Tipo de dado Noh
Noh:
  Coord pos: posição atual
  Coord ant: antecessor
  double g: custo passado
  double h: custo futuro
  double f(): custo total (=g+h)

// Cria os conjuntos de Noh
// inicialmente vazios
Container<Noh> Aberto
Container<Noh> Fechado

// Cria o Noh inicial
Noh atual;
atual.pos ← origem
atual.ant ← posição_inválida
atual.g ← 0.0
atual.h ← heurística(destino)

// Inicializa o conjunto Aberto
inserir(atual,Aberto)

// Posições (var. auxiliares)
Coord dir, prox;

// Iteração: repita enquanto houver
// Noh's em Aberto e ainda não
// houver encontrado a solução
REPITA
|
| // Atualiza atual com o Noh de
| // menor custo (o 1°) de Aberto
| atual ← primeiro(Aberto)
|
| // Remove o 1° Noh de Aberto
| remove_primeiro(Aberto)
|
| // Insere o Noh em Fechado
| inserir(atual,Fechado)
|
| // Testa se é solução
| SE (atual.pos ≠ destino)
| |
| | // Gera sucessores de atual
| | PARA dir.lin de -1 a 1
| | PARA dir.col de -1 a 1
| | SE dir ≠ (0,0)
| | |
| | | prox ← atual.pos + dir
| | |
| | | // Testa se pode mover de
| | | // atual para prox
| | | SE ( movVálido(atual.pos,
| | | |                 prox) )
| | | |
| | | | // Gera novo sucessor:
| | | | suc.pos ← prox
| | | | suc.ant ← atual.pos
| | | | // A norma de dir dá o
| | | | // custo do movimento:
| | | | // 1.0 ou sqrt(2.0)
| | | | suc.g ← atual.g+norma(dir)
| | | | suc.h ← heurística(destino)
| | | |
| | | | // Existe em Fechado um nó
| | | | // igual a suc (mesma pos)?
| | | | oldF ← procura(suc.pos,
| | | |                Fechado)
| | | | SE (oldF ≠ não_existe)
| | | | |
| | | | | // Testa qual tem menor
| | | | | // custo total f=g+h
| | | | | SE (suc < oldF)
| | | | | |
| | | | | | remove(oldF,Fechado)
| | | | | | oldF ← não_existe
| | | | | |
| | | | | FIM SE
| | | | |
| | | | FIM SE
| | | |
| | | | // Se ñ existe em Fechado,
| | | | // existe em Aberto um nó
| | | | // igual a suc (mesma pos)?
| | | | SE (oldF = não_existe)
| | | | |
| | | | | oldA ← procura(suc.pos,
| | | | |                Aberto)
| | | | | SE (oldA ≠ não_existe)
| | | | | |
| | | | | | // Testa qual tem menor
| | | | | | // custo total f=g+h
| | | | | | SE (suc < oldA)
| | | | | | |
| | | | | | | remove(oldA,Aberto)
| | | | | | | oldA ← não_existe
| | | | | | |
| | | | | | FIM SE
| | | | | |
| | | | | FIM SE
| | | | | |
```

```
| | | | FIM SE
| | | |
| | | | // Insere suc em Aberto se
| | | | // não existe nem em Aberto
| | | | // nem em Fechado, seja pq
| | | | // não existia antes, seja
| | | | // pq foi removido
| | | | SE (oldF = não_existe E
| | | | |   oldA = não_existe)
| | | | |
| | | | | // Determina onde inserir
| | | | | // suc, que deve ficar na
| | | | | // frente de big, o 1° nó
| | | | | // de Aberto com custo
| | | | | // total f maior que o
| | | | | // custo total f de suc
| | | | | big ← acha_maior(suc.f(),
| | | | |                  Aberto)
| | | | |
| | | | | // Insere suc em Aberto
| | | | | // na frente de big
| | | | | inserir(suc,big,Aberto)
| | | | |
| | | | FIM SE
| | | |
| | | FIM SE
| | |
| | FIM SE,PARA,PARA
| |
| FIM SE
|
ENQUANTO ( atual.pos ≠ destino E
           NÃO(vazio(Aberto)) )

// Disponibiliza estado final da
// busca, quer encontre ou não o
// caminho. Se não existe caminho,
// tamanho(Aberto) deve ser 0
NF ← tamanho(Fechado)
NA ← tamanho(Aberto)

// Pode ter terminado porque
// encontrou a solução ou porque
// não há mais nohs a testar
SE (atual.pos ≠ destino)
|
| // Disponibiliza informação de
| // que não existe caminho
| // (comprimento e profundidade
| // inválidos)
| compr ← -1.0
| prof ← -1
|
CASO CONTRÁRIO
|
| // Calcula e depois disponibiliza
| // comprimento e profundidade
| compr ← atual.g
| prof ← 1
| ENQUANTO (atual.ant ≠ origem)
| |
| | // Assinala a célula como
| | // pertencente ao caminho
| | marca_caminho(atual.ant)
| | // Procura a célula anterior
| | // à atual em Fechado
| | atual ← procura(atual.ant,
| |                 Fechado)
| | prof++
| |
| FIM ENQUANTO
|
FIM SE
```