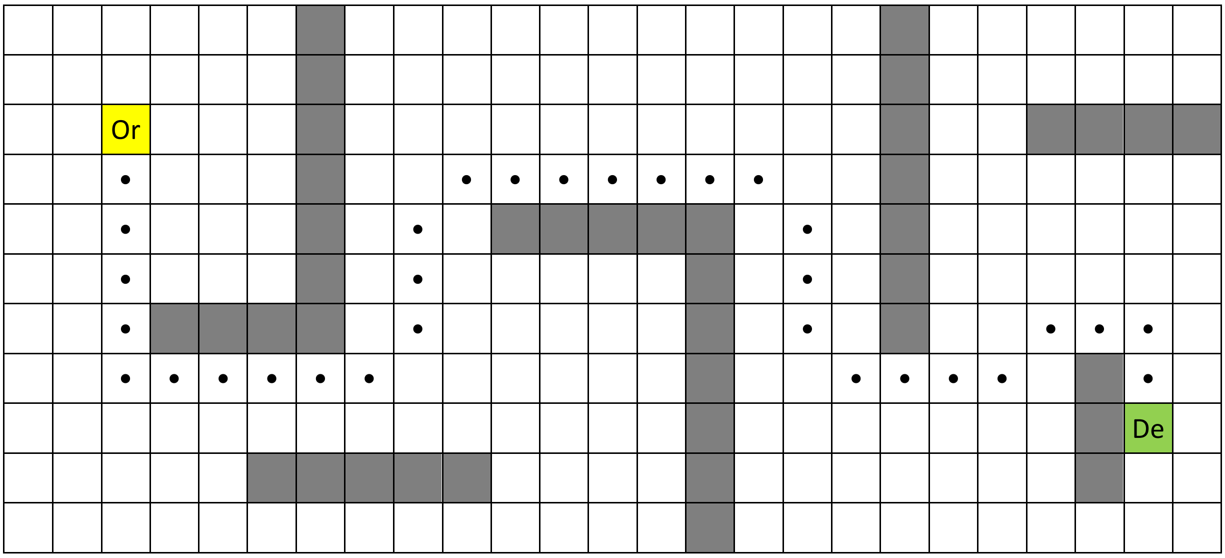
UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLGIA
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO

PLANEJADOR DE CAMINHOS EM LABIRINTOS

PROFESSOR: ADELARDO ADELINO DANTAS DE MEDEIROS



O objetivo é desenvolver em C++ um programa para determinar o caminho de menor custo (mais curto) entre células de origem e destino, dentro de um ambiente descrito por um mapa com obstáculos, utilizando o algoritmo A* e as estruturas de dados da biblioteca STL de C++.

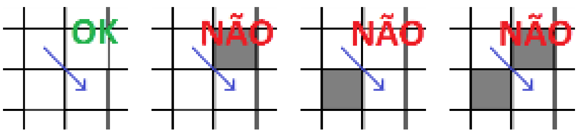
CAMINHO DE MENOR CUSTO EM GRAFO

O algoritmo A* encontra o caminho de menor custo em um grafo no qual a transição de cada nó do grafo para outro nó ao qual ele esteja conectado tem um custo associado. No exemplo do labirinto, 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, ou seja, ambas as células vizinhas simultaneamente da origem e destino estejam livres.



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 movimentação. 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 (no caso do labirinto, 1 ou $\sqrt{2}$).

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

O comprimento (custo total) 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:
  $27 \times 1 + 5 \times \sqrt{2} = 34{,}07$
- 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, normalmente os nós em ***Aberto*** são mantidos ordenados em ordem crescente de custo e retira-se sempre o primeiro.

Para cada célula do mapa, só pode haver um nó armazenado em ***Aberto*** ou ***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 apenas 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 a heurística ***h*** for admissível, isto é, se o seu valor for sempre menor ou igual do que o custo real para mover do nó até o destino. Caso não se use heurística, ou seja, ***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):
$$h = \sqrt{2} \cdot \min(|\Delta x|, |\Delta y|) + \mathrm{abs}(|\Delta x| - |\Delta y|)$$
- 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*

```
// Tipo de dado Noh
Noh:
  pos: célula atual (posição)
  ant: célula anterior (antecessor)
  g: custo passado
  h: custo futuro

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

// Cria o noh inicial
Noh atual;

atual.pos ← origem
atual.ant ← void()
atual.g ← 0.0
atual.h ← heurística()

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

// Iteração: repita enquanto houver
// nohs em Aberto e ainda não
// houver encontrado a solução
Repita
|
| // Remove o noh de menor custo
| // (o primeiro) de Aberto
| atual ← remove_primeiro(Aberto)
|
| // insere o noh em Fechado
| inserir(atual,Fechado)
|
| // Testa se é solução
| Se ( Não(é_destino(atual)) )
| |
| | // Gera sucessores de atual
| | Para dir.lin de -1 a 1
| | Para dir.col de -1 a 1
| | Se dir ≠ (0,0)
| | |
| | | dest ← atual.pos + dir
| | |
| | | // Testa se pode mover de
| | | // atual na direção dir
| | | Se ( movVálido(atual,dest) )
| | | |
| | | | // Gera novo sucessor:
```

```
| | | | suc.pos ← dest
| | | | suc.ant ← atual.pos
| | | | suc.g ← atual.g+custo(dir)
| | | | suc.h ← heurística()
| | | |
| | | | // Procura suc em Fechado
| | | | oldF ← procura(suc,Fechado)
| | | | Se (existe(oldF))
| | | | |
| | | | | // Testa qual o melhor
| | | | | Se (suc < oldF)
| | | | | | remove(oldF,Fechado)
| | | | | | oldF ← não_existe()
| | | | | Fim Se
| | | | |
| | | | Fim Se
| | | |
| | | | // Procura suc em Aberto
| | | | oldA ← procura(suc,Aberto)
| | | | Se (existe(oldA))
| | | | |
| | | | | // Testa qual o melhor
| | | | | Se (suc < oldA)
| | | | | |
| | | | | | remove(oldA,Aberto)
| | | | | | oldA ← não_existe()
| | | | | |
| | | | | Fim Se
| | | | |
| | | | Fim Se
| | | |
| | | | // Insere suc em Aberto se
| | | | // não existe nem em Aberto
| | | | // nem em Fechado, seja pq
| | | | // não existia mesmo, seja
| | | | // pq foi removido
| | | | Se ( Não(existe(oldF)) E
| | | | |    Não(existe(oldA)) )
| | | | |
| | | | | inserir_ordem(suc,Aberto)
| | | | |
| | | | Fim Se
| | | |
| | | Fim Se
| | |
| | Fim Se,Para,Para
| |
| Fim Se
|
Enquanto ( Não(é_destino(atual)) E
           Não(vazio(Aberto)) )

// Imprime estado final da busca
// quer encontre ou não o caminho
// Se não encontrou caminho,
// tamanho(Aberto) deve ser 0
exibir(tamanho(Fechado))
exibir(tamanho(Aberto))

// Pode ter terminado porque
// encontrou a solução ou porque
// não há mais nohs a testar
Se ( Não(é_destino(atual)) )
|
| exibir("Não existe caminho")
|
Caso contrário
|
| comprimento ← atual.g
| profundidade ← 1
| Enquanto (atual.ant != origem)
| |
| | incluir_caminho(atual.ant)
| | atual ← procura(atual.ant,
| |         Fechado)
| | profundidade++
| |
| Fim Enquanto
|
| // Dados do caminho encontrado
| exibir(comprimento)
| exibir(profundidade)
|
Fim Se
```