Inteligência Artificial para Jogos

Aula 10 - Descoberta de caminho – Parte 04

Apresentação da aula

Olá! Você chegou a última e derradeira aula do curso! 👏 E para o final... hora de achar o menor caminho em um **grafo**!

Nas últimas aulas você viu o que era um **grafo** e como fazer um percurso entre os nós para ir de um ponto ao outro da estrutura. Mas nem sempre o caminho que o algoritmo encontrava era necessariamente o melhor! No exemplo que foi dado, do **grafo** do estado do RN, a melhor rota é a que gasta menos dinheiro, mesmo que ela não seja direta, não é verdade?

Nessa aula, você vai estudar dois algoritmos que podem ser usados quando se quer encontrar o menor caminho entre dois pontos de um **grafo**. Esses são algoritmos clássicos e bastante utilizados na área de jogos, mas também são amplamente utilizados nas demais áreas da computação!

Então amarre o tênis e se prepare para caminhar... pelo **grafo**. 😁 Simbora!

Objetivos

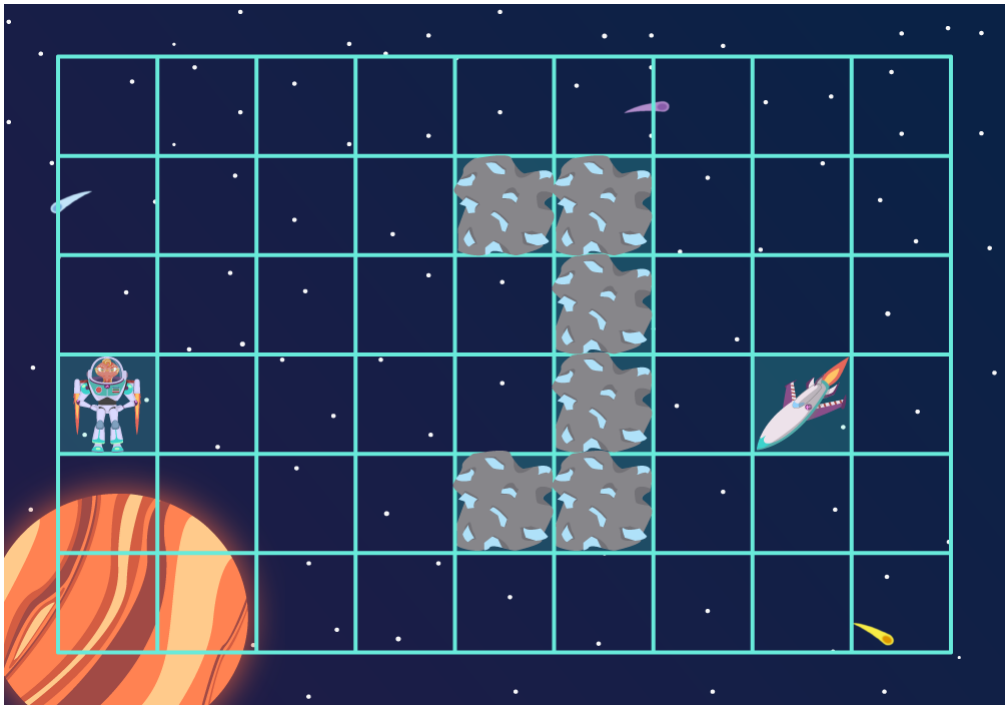
> Conhecer os conceitos do algoritmo de **Dijkstra** e do algoritmo **A***;
>
> Aprender o pseudocódigo que implementa esses algoritmos e entender as diferenças de execução entre eles;
>
> Distinguir a execução dos algoritmos **DFS/BFS**, **Dijkstra** e **A***.

1 - Caminhos pelo mapa

Bom, antes de mais nada, deixe-me preparar um cenário para você! Vou abandonar um pouco o **grafo** do RN (talvez ele seja citado em menção honrosa), e você vai trabalhar em um mapa que parece um pouco mais com o cenário de um jogo. Que rufem os tambores!

Eis que lhes apresento... a Figura 01!

Figura 01 - Mapa de exemplo da Aula...



Ehr... é que ficou tão bonito que eu não tive coragem de fazer outro (gulp!). 😎

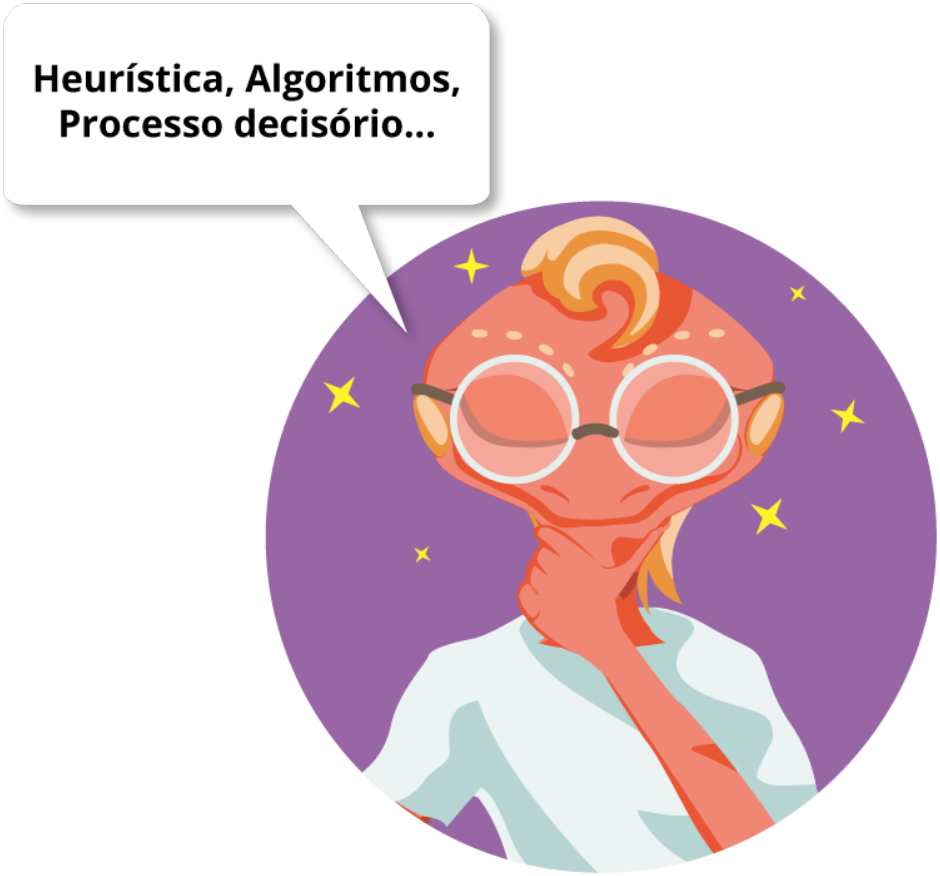
O que você tem aqui é um mapa em forma de malha - ou *grid*, que representará um cenário de um jogo. Os asteroides (esses quadrados com objetos em cinza) representam obstáculos que não permitem a passagem do jogador (o Patrulheiro Estelar), e a nave é o objetivo final que ele deseja alcançar. O personagem pode se mover nas oito direções, e o custo de se movimentar na vertical e horizontal é igual ao custo de mover nas diagonais (considere todos os custos unitários).

Você deve estar se perguntando: aqueles algoritmos da aula passada já não serviriam para encontrar um caminho nesse mapa?

Claro que sim! Pode não ser o melhor caminho possível, mas será um caminho, sem dúvidas. Que tal utilizar um deles como exemplo de como esse mapa seria representado por um **grafo**, deixando a base prontinha para o estudo dos próximos algoritmos? Parece uma boa ideia. 😉

Antes de começar, eu tenho uma confissão a fazer: eu deixei uma parte de fora da outra aula. Sabe como é, já tinha tanta coisa que fiquei com medo de embananar você. Lembra que na aula passada a escolha do próximo vértice foi meio aleatória? Não que não possa ser feito dessa forma, mas normalmente existe um critério para escolher quem será o próximo vizinho a ser explorado. Por exemplo, eu poderia ter estabelecido que o primeiro vértice a ser visitado fosse aquele cujo preço da passagem fosse mais barata. Não faria mal ao bolso, hein?

Essa estratégia de escolha do próximo vértice tem um nome lindo de doer: **heurística**. 😁 A **heurística** do algoritmo refere-se ao processo ou estratégia utilizada para tomar determinadas decisões em sua execução.



Então, que tal ilustrar no mapa acima como funcionaria uma **DFS** utilizando algumas **heurísticas**? Prepare o coração, que serão alguns passos de simulação!

Se rimou, é verdade. Considerando o mapa acima, uma primeira **heurística** que você poderia utilizar seria a **distância de Manhattan** (lembra da bichinha?). Ela seria uma boa **heurística** caso o movimento só pudesse ser realizado nas quatro direções. Na **distância de Manhattan**, o cálculo do valor da **heurística** seria a soma de quantos blocos faltam, na vertical e na horizontal, para chegar ao objetivo. Logo a distância de cada bloco para o objetivo seria dada por:

Distância = |posição_atual.x - destino.x| + |posição_atual.y - destino.y|

Mas eu gosto mesmo é de jogo com movimento nas oito! Quem não ama uma diagonal? Para esse tipo de cenário, existe uma **heurística** similar, a **distância diagonal**: será uma medição parecida com a **distância de Manhattan**, mas você terá de levar em conta os blocos na diagonal também, ok? Nesse caso, a conta fica um pouquinho mais simplificada:

Distância = MAIOR (|posição_atual.x - destino.x|, |posição_atual.y - destino.y|)

Onde MAIOR retorna o maior valor entre os dois existentes nos parênteses, no caso quantos quadrados faltam na horizontal e na vertical. Essa simplificação está levando em conta a possibilidade de o personagem dar o passo na diagonal, o que reduz muitas vezes o custo da distância comparado com a fórmula de **Manhattan**.

Uma outra **heurística** que é bastante utilizada é a **distância Euclidiana**. Nela você utiliza o **teorema de Pitágoras** para descobrir qual seria a distância em linha reta do ponto do mapa até o ponto de destino. Essa **heurística** é mais utilizada quando se tem um mapa contínuo, que não segue o padrão do *grid*, e onde os custos de movimentação podem ser mais variados:

Distância = RAIZ_QUADRADA (|posição_atual.x - destino.x|² + |posição atual.y - destino.y|²)

Que tal uma simulação de uma **DFS** usando a **heurística** de **distância diagonal**?

1.1 - Ponto Inicial

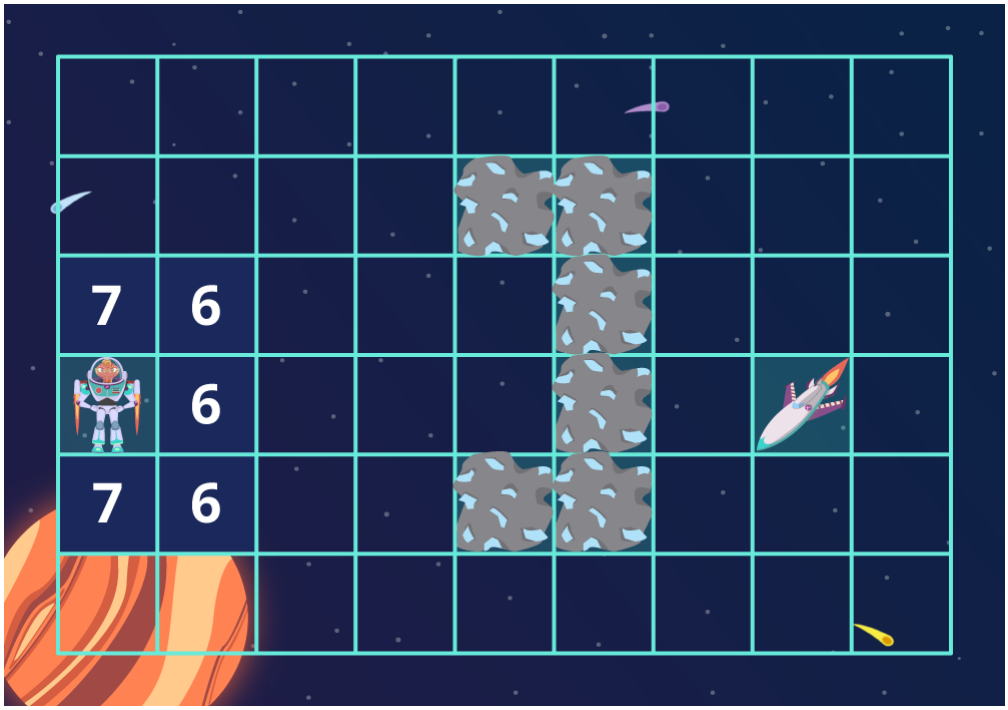
Calcule o custo de mover para cada um dos quadrados vizinhos usando a **heurística** da **distância diagonal**. Os referenciais são o patrulheiro estelar e a nave. Como é a primeira vez, vou explicitar as contas para você. Se você contar as linhas e as colunas, vai observar que:

O patrulheiro estelar está na posição (4,1)
A nave está na posição (4,8)

Os vizinhos do patrulheiro estelar são os blocos que representam as posições (3,1), (3,2), (4,2), (5,2) e (5,1). Aplicando a **heurística** você vai ter:

D(3,1) = MAIOR(|3-4|, |1-8|) = MAIOR(1, 7) = 7
D(3,2) = MAIOR(|3-4|, |2-8|) = MAIOR(1, 6) = 6
D(4,2) = MAIOR(|4-4|, |2-8|) = MAIOR(0, 6) = 6
D(5,1) = MAIOR(|5-4|, |1-8|) = MAIOR(1, 7) = 7
D(5,2) = MAIOR(|5-4|, |2-8|) = MAIOR(1, 6) = 6

Figura 02 - Ponto inicial do patrulheiro estela



Aqui você tem um empate entre várias casas, como vai ser o desempate? Você escolhe o critério. Você pode utilizar o seguinte: em caso de empate, avança primeiro para o vizinho que só muda em uma direção (vertical ou horizontal), para depois ir para o vizinho que altera as duas. É mais uma decisão que foi colocada no algoritmo, logo faz parte da **heurística**, mas não tem nenhum motivo especial para essa decisão. É só a ideia de tentar ir direto para o objetivo o mais rápido possível.

Seguindo a **heurística**, dê um passo para a frente (lembra do desempate? Vá reto para só atualizar o valor na horizontal). E, enquanto não chegar no destino, vá repetindo os passos. Lembre-se que na **DFS** se marcam os nós visitados para evitar que o personagem entre em um ciclo e fique igual a cachorro correndo atrás do próprio rabo! Calculando o valor para os novos vizinhos, você vai ter:

Figura 03 - Passo 1 do patrulheiro estelar

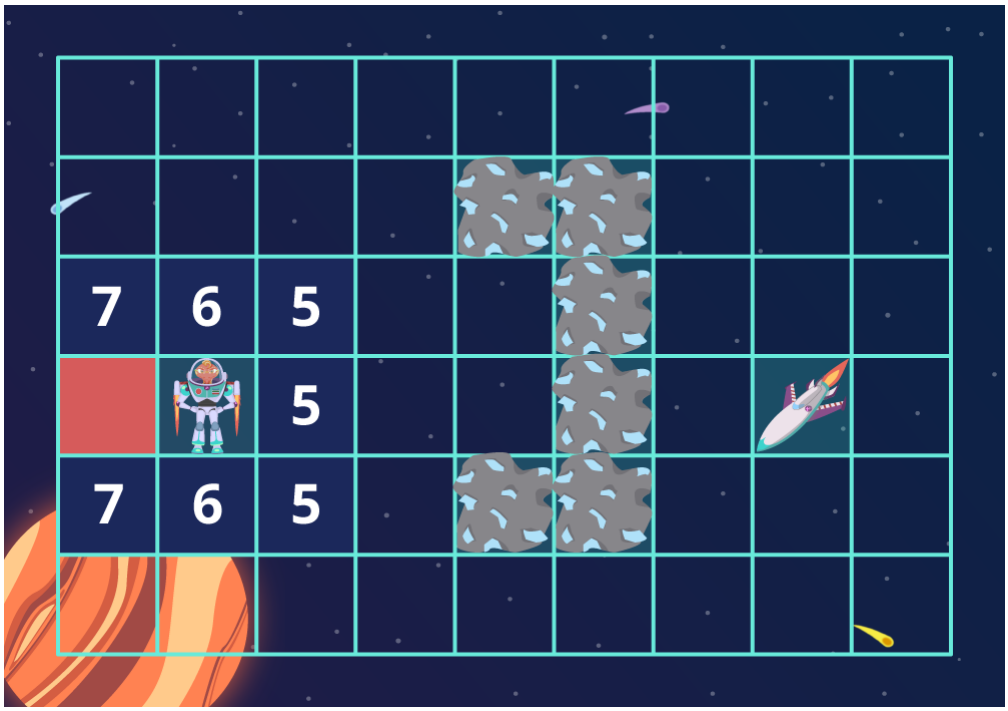


Figura 04 - Passo 2 do patrulheiro estelar

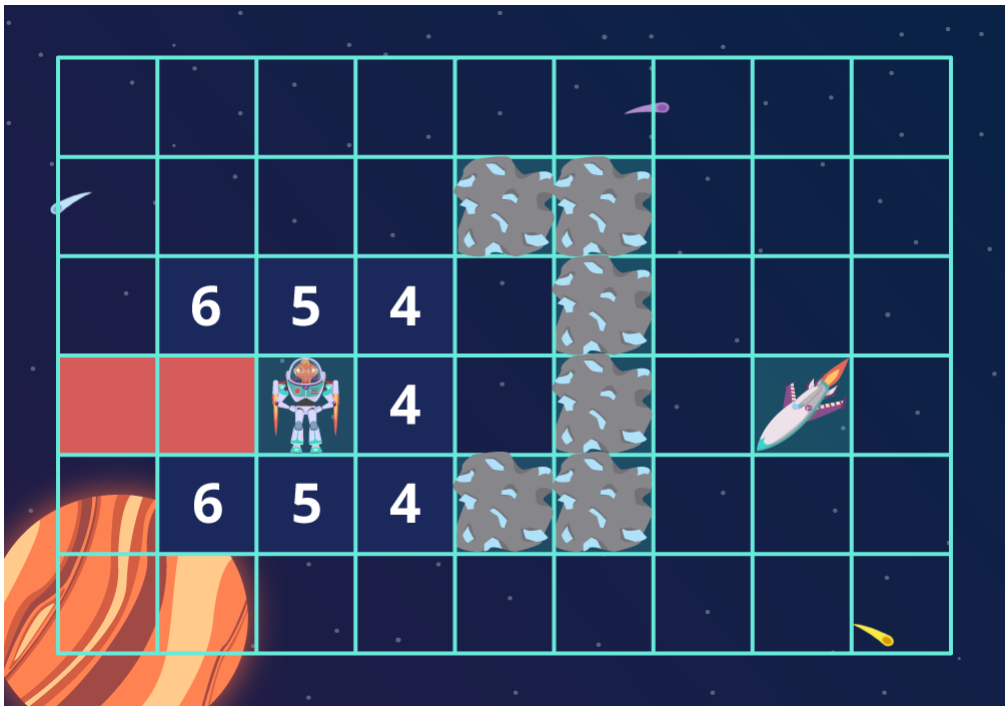


Figura 05 - Passo 3 do patrulheiro estelar

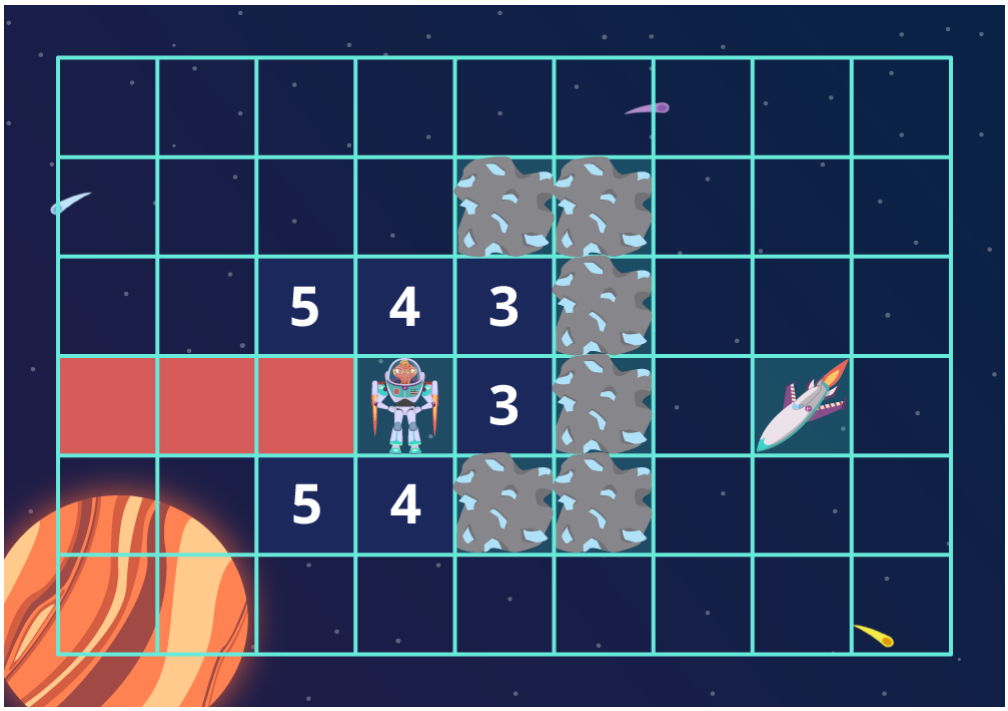
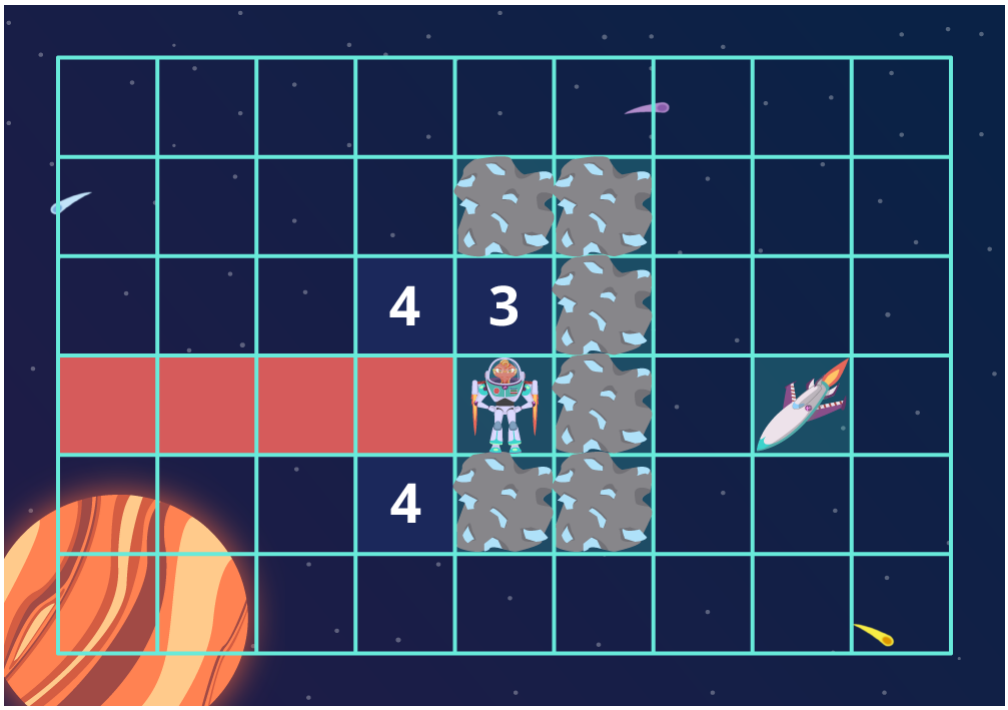


Figura 06 - Passo 4 do patrulheiro estelar



Opa! Bateu de cara com a parede! Continuando com o algoritmo, de acordo com as casas que ainda não foram visitadas.

Figura 07 - Passo 5 do patrulheiro estelar

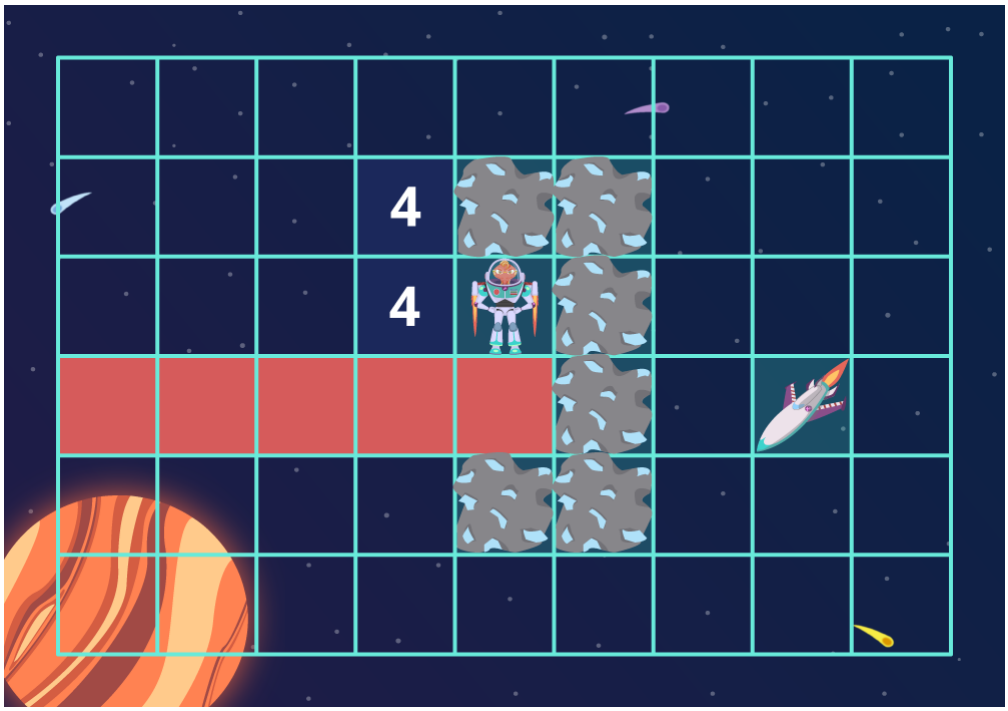


Figura 08 - Passo 6 do patrulheiro estelar

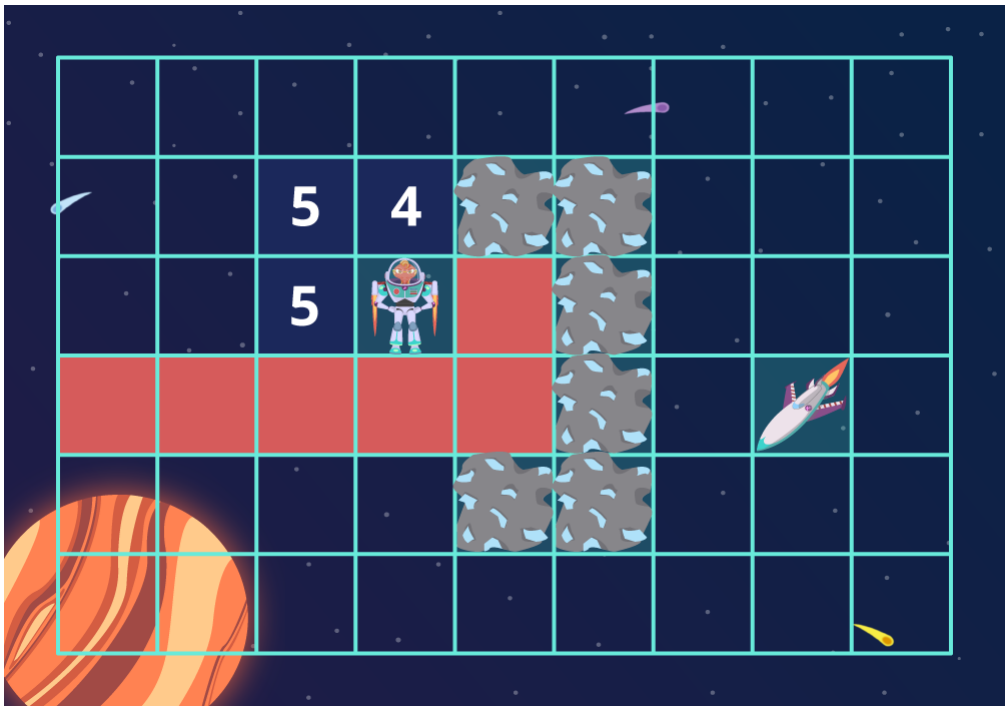


Figura 09 - Passo 7 do patrulheiro estelar

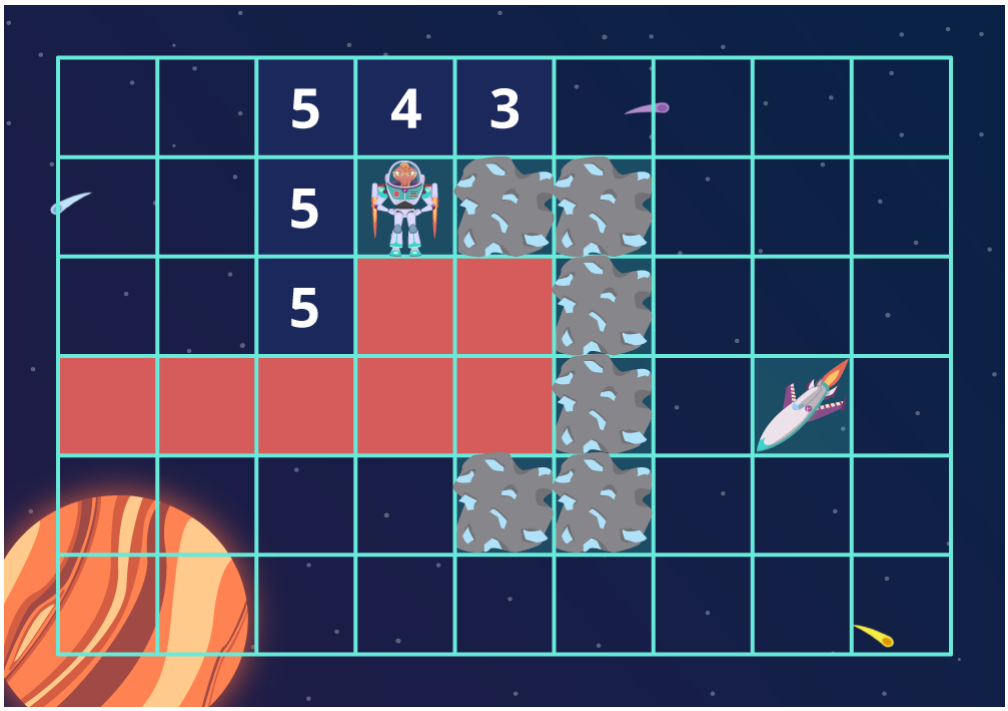


Figura 10 - Passo 8 do patrulheiro estelar

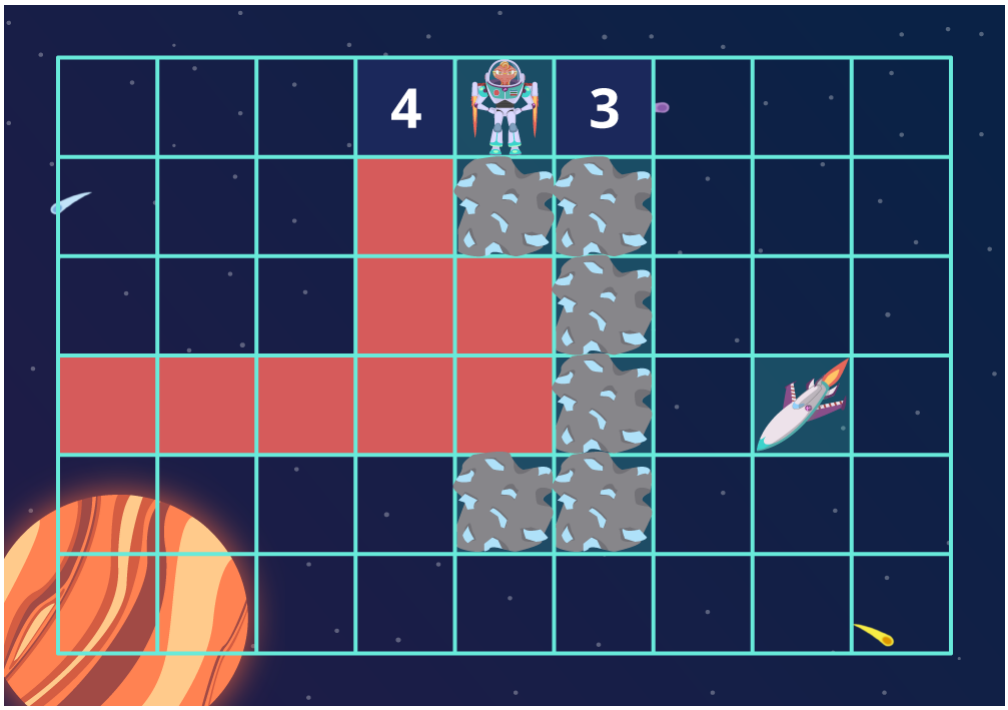
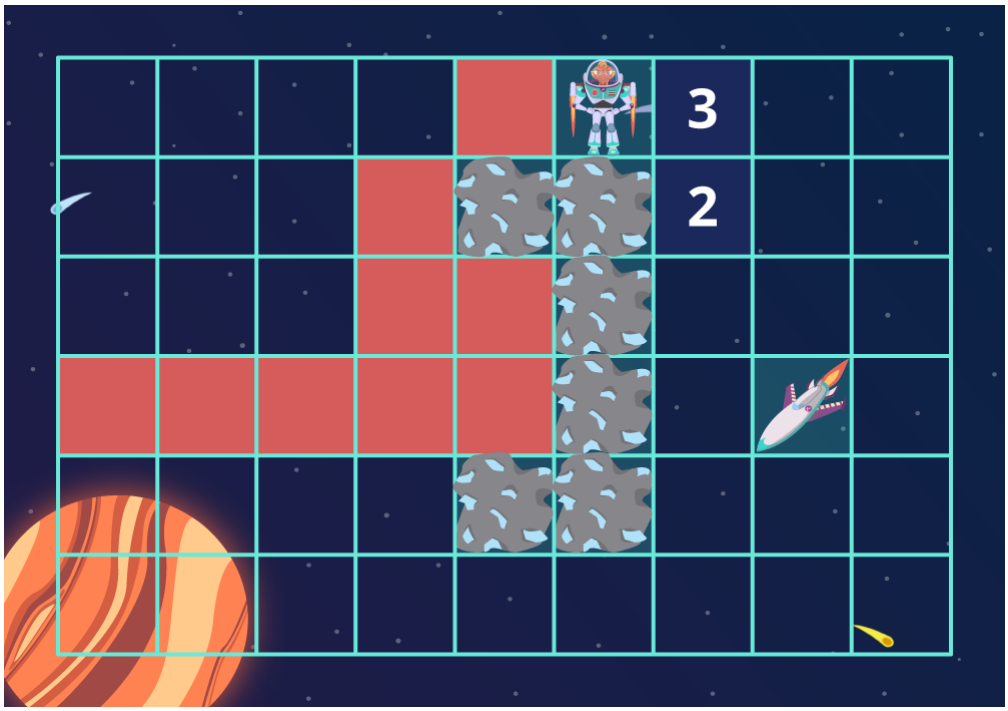


Figura 11 - Passo 9 do patrulheiro estelar



Ufa, que arrodeio! Mas parece que agora vai. 😁

Figura 12 - Passo 10 do patrulheiro estelar

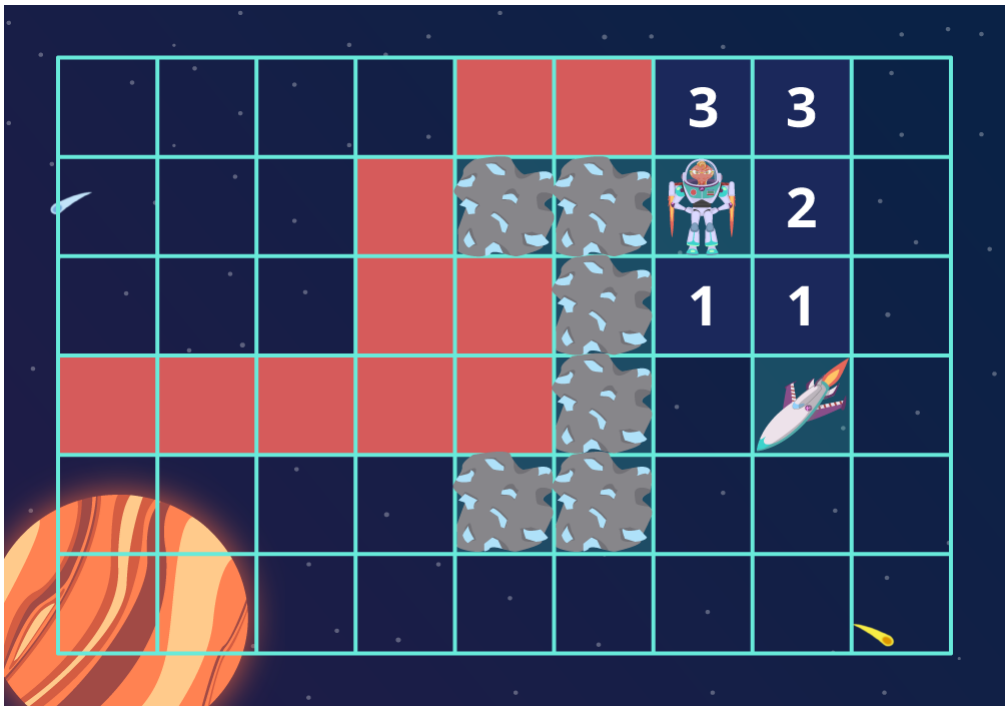


Figura 13 - Passo 11 do patrulheiro estela

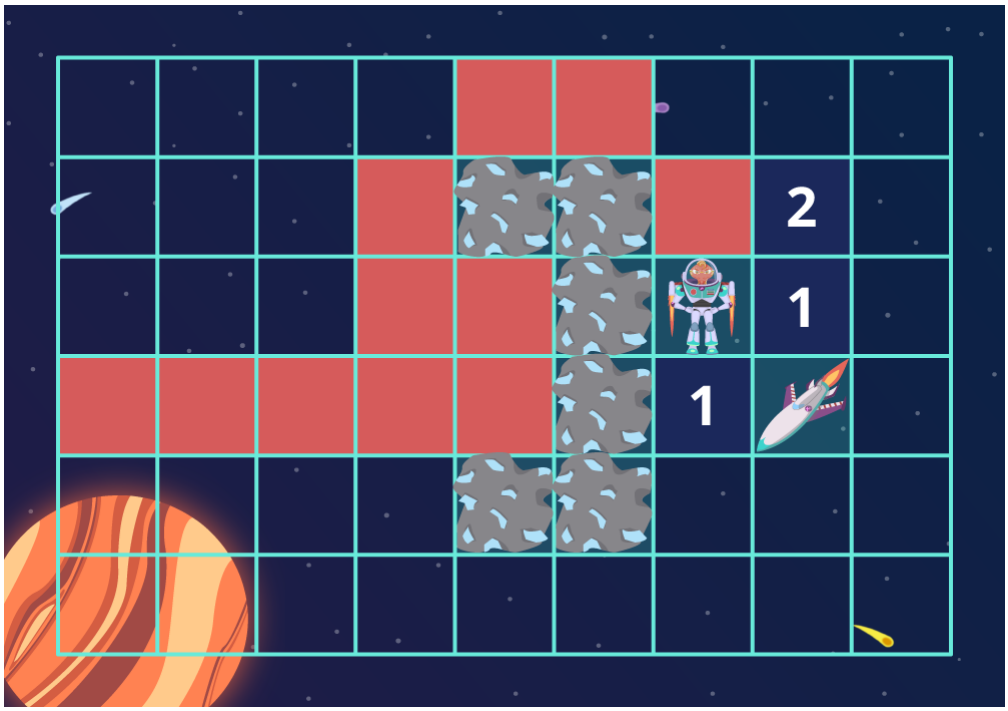
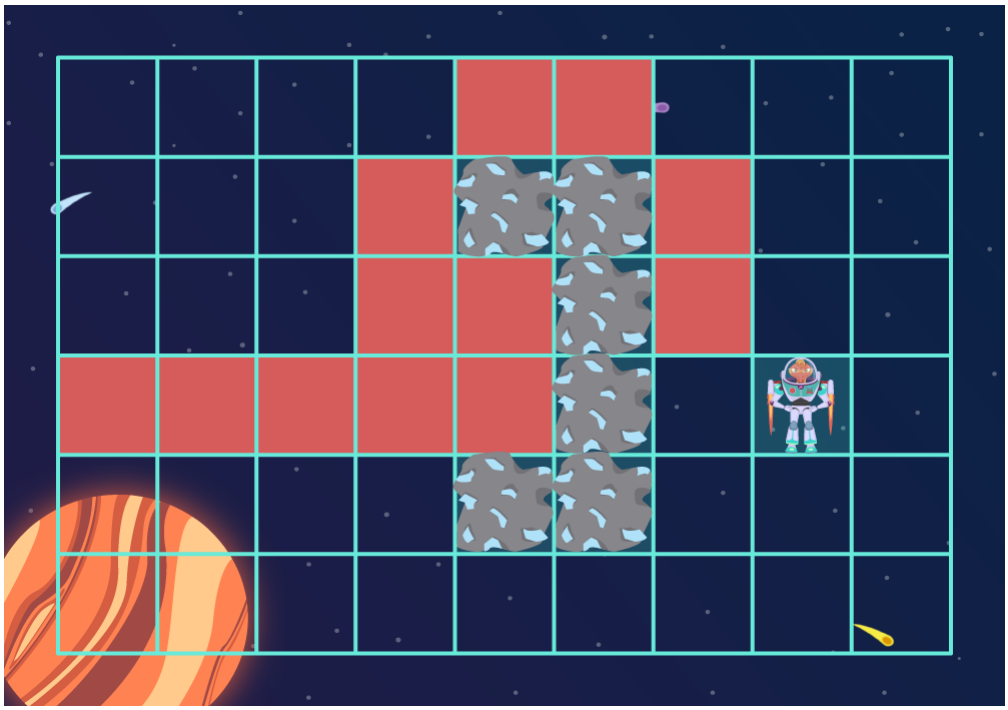


Figura 14 - Passo final!

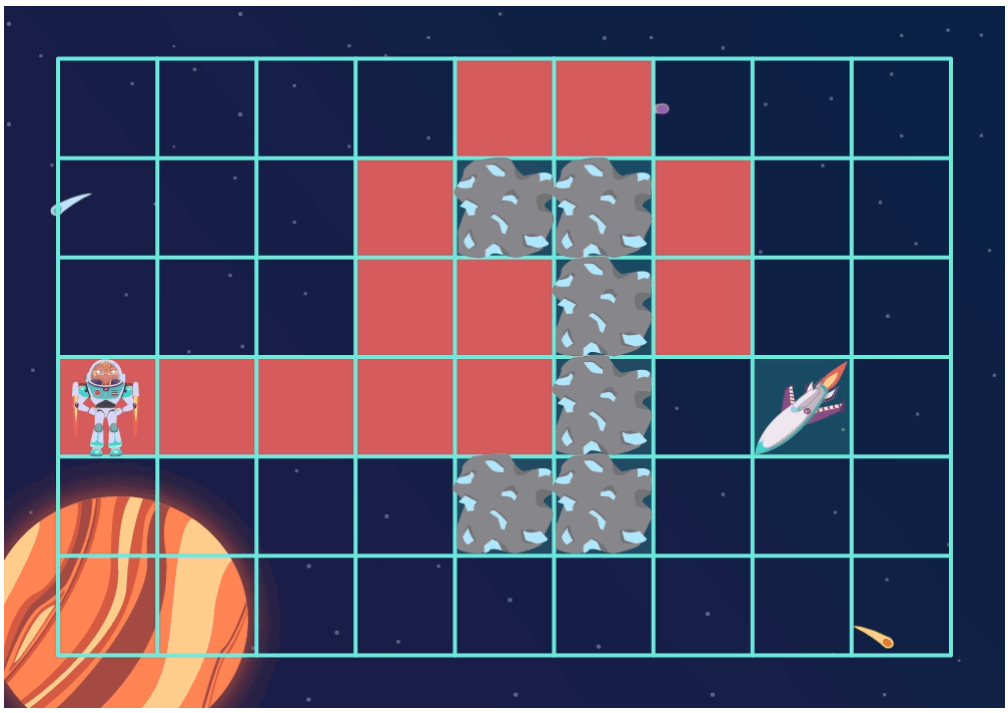


Chegou! 🎉

Então, a **DFS** consegue sim achar um caminho, mas com certeza dava para ser melhor! O problema é que o algoritmo usa uma **heurística** de decisão que opera em um nível local: ele só avalia a próxima casa! Como o obstáculo está lá na frente, ele

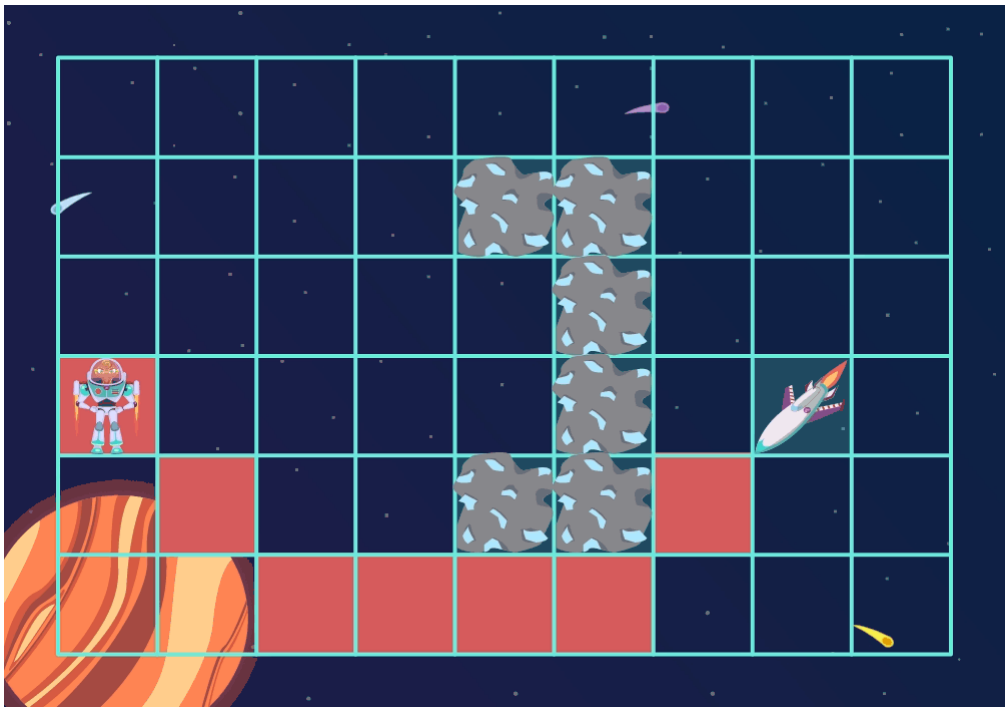
não consegue evitá-lo com antecedência, só quando já está de frente a ele.

Figura 15 - Opa!



Você deve estar se perguntando: mudar a **heurística** local afeta alguma coisa? Sim! Imagine, por exemplo que o critério de desempate fosse diferente: escolher sempre o vértice mais abaixo, forçando a procura inicial por um caminho na parte inferior do mapa. A sequência de execução levaria a um caminho bem diferente:

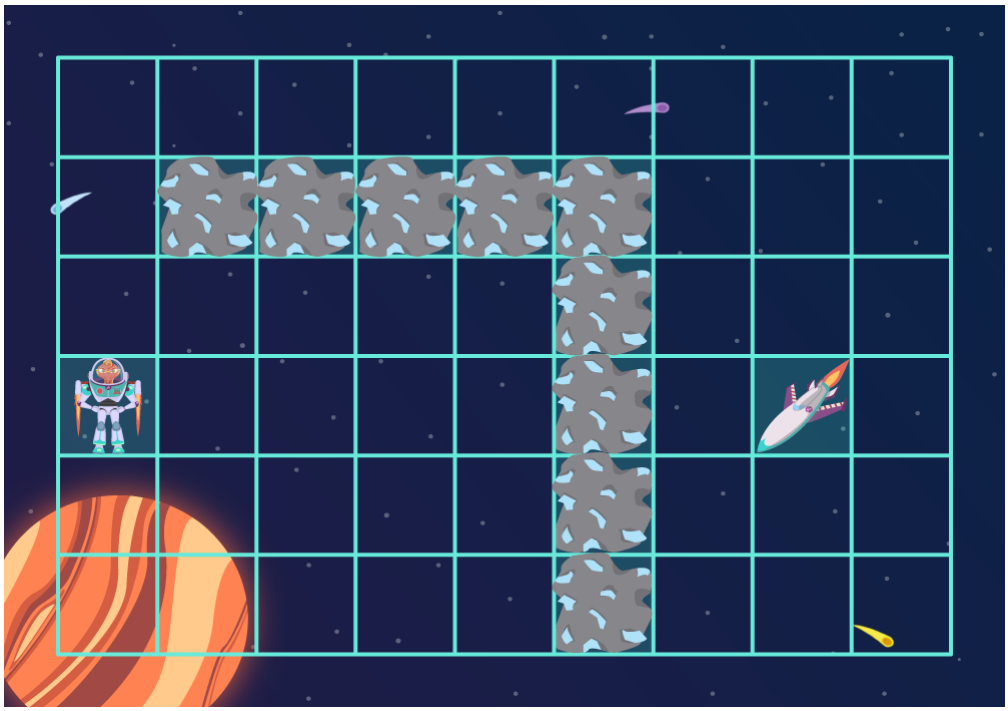
Figura 16 - Outra saída?



O problema é que não dá para ficar adivinhando qual a melhor **heurística** para cada mapa, principalmente quando os objetos e obstáculos podem mudar de posição em tempo real.

Mas, e se você calcular todos os caminhos possíveis e depois verificar qual é o melhor? Boa! Aí vai dar certo... só que com a **BFS/DFS**, você vai precisar executar o algoritmo passando por todos os vértices pelo menos uma vez para garantir que achou todos os diferentes caminhos. Isso significa que a operação vai ser MUITO custosa, e em jogos normalmente não existe esse luxo. Tem que rodar rápido, senão o jogador lhe xinga! Imagine se o cenário fosse um pouquinho diferente:

Figura 17 - Mudando o cenário!



É bem provável que na situação da figura acima, a busca vá primeiro percorrer todos os blocos mais internos dos obstáculos, e só no final consiga achar o caminho que vai por cima!

Por isso, é necessária uma classe de algoritmos que seja capaz de identificar o menor caminho dentro de um mapa (ou **grafo** 😁) em um tempo de execução que seja aceitável para o jogo. Esse algoritmo é o **A*** (pronuncia-se **A estrela**), mas antes de estudá-lo você precisa conhecer o algoritmo base sobre o qual ele é construído: o algoritmo de **Dijkstra**.

2 - Menor caminho: algoritmo de Dijkstra

O **Dijkstra** é um dos algoritmos clássicos da área de **grafos**. Ele é comumente utilizado em situações em que se deseja encontrar o menor caminho de um vértice origem para todos os outros vértices do **grafo**, mas também pode ser adaptado para encontrar o menor caminho entre dois vértices específicos. Como? É só parar ele no meio do caminho. 😁

O algoritmo é baseado nas técnicas de **programação dinâmica** para obter uma boa performance. Calma, eu não vou explicar em detalhes o que é **programação dinâmica**, isso em si daria uma disciplina de 60 horas do curso! A ideia bem simplificada (mas olhe, é BEM simplificada) é salvar e reutilizar resultados que você já encontrou uma vez na execução do algoritmo, para não ter de calcular de novo. O algoritmo vai "lembrando" dos resultados e, quando precisa de algo que ele já sabe, por exemplo, o custo para chegar em um vértice que ele já calculou não precisa refazer as contas, apenas usa o que já está armazenado. A ideia é simples, a implementação não. 😐

O algoritmo de **Dijkstra** segue o seguinte fluxo:

- Passo 1: Todos os vértices do **grafo** são considerados como não visitados;
- Passo 2: Uma distância é colocada para cada vértice. A origem do caminho tem distância 0 (você não gasta nada para chegar lá, é seu ponto de partida!), enquanto os outros vértices têm distância infinita (você ainda não chegou em nenhum deles). Além disso, para cada vértice vai ter uma informação anterior, para informar a partir de que outro ponto do **grafo** o caminho até ele foi construído;
- Passo 3: O vértice origem passa a ser seu vértice atual de análise, onde você vai realizar os seguintes passos:
    - Atribuir distância a todos os vizinhos não visitados. Essa distância é o custo de chegar até o vizinho (peso da aresta) mais o custo para chegar até o vértice atual. Caso já exista uma distância diferente de infinito, ficará a menor entre a que já existia e a que acabou de ser calculada;
    - Caso a distância atual seja a menor, alterar o vértice anterior do vizinho para o vértice atual.
- Passo 4: O vértice atual é marcado como visitado;

- Passo 5: Um novo vértice não visitado é escolhido para ser o vértice atual, e os procedimentos do passo 3 são repetidos até não existirem mais vértices não visitados.

Uma característica interessante desse algoritmo é que ele testa vários caminhos até cada vértice, mas salva apenas o menor que foi encontrado ao longo da execução. Por exemplo, para um vértice X, se ele salva o menor custo de chegar a X, então todos os caminhos que passam por X usarão esse custo. Ele não precisa ficar recalculando esse valor toda vez que encontrar um caminho que passa por X, ele apenas utiliza o que já foi calculado anteriormente. No Código 01 pode-se ver o pseudocódigo do algoritmo.

Código 01 – Pseudocódigo do algoritmo de **Dijkstra**

```
Djikstra(Grafo g, Vértice origem){
    distancia[origem] = 0

    Cria conjunto de vertices Não_Visitados

    Para cada vértice v do grafo g{
        Se v!= origem{
            distancia[v] = INFINITO
        }
        anterior[v] = INDEFINIDO
        Insere v em Não_Visitados
    }

    Enquanto Não_Visitados não for VAZIA{
        Procura vértice com menor distância, denominado u

        Remove u de Não_Visitados

        Para cada i vizinho de u{
            custo = distancia[u] + g[u, i]

            Se custo < distancia[i] {
                distancia[i] = custo
                anterior[i] = u
            }
        }
    }

    return distancia, anterior
}
```

É difícil visualizar como ele funciona, não é mesmo? Imagine como se fosse uma busca sistemática, só que ela vai salvando os valores do caminho encontrado, e não precisa retornar em nenhum momento. É quase como se o caminho fosse uma “onda” que vai varrendo os vértices do **grafo** e atualizando com os menores caminhos possíveis, até chegar ao ponto de origem. Veja um exemplo com o mapa da seção anterior em que você partirá do mesmo ponto.

Assuma que o custo de movimentação entre os blocos é 1, e não há diferença entre movimento vertical e horizontal (da mesma forma que foi feito no exemplo anterior):

Figura 18 - Voltando ao cenário!

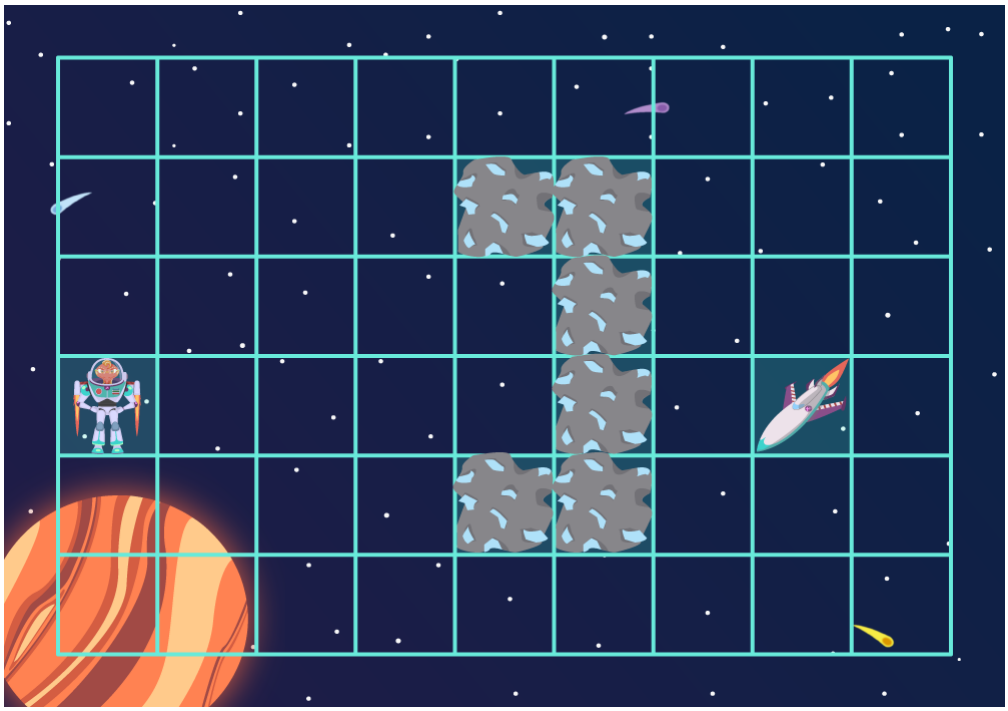
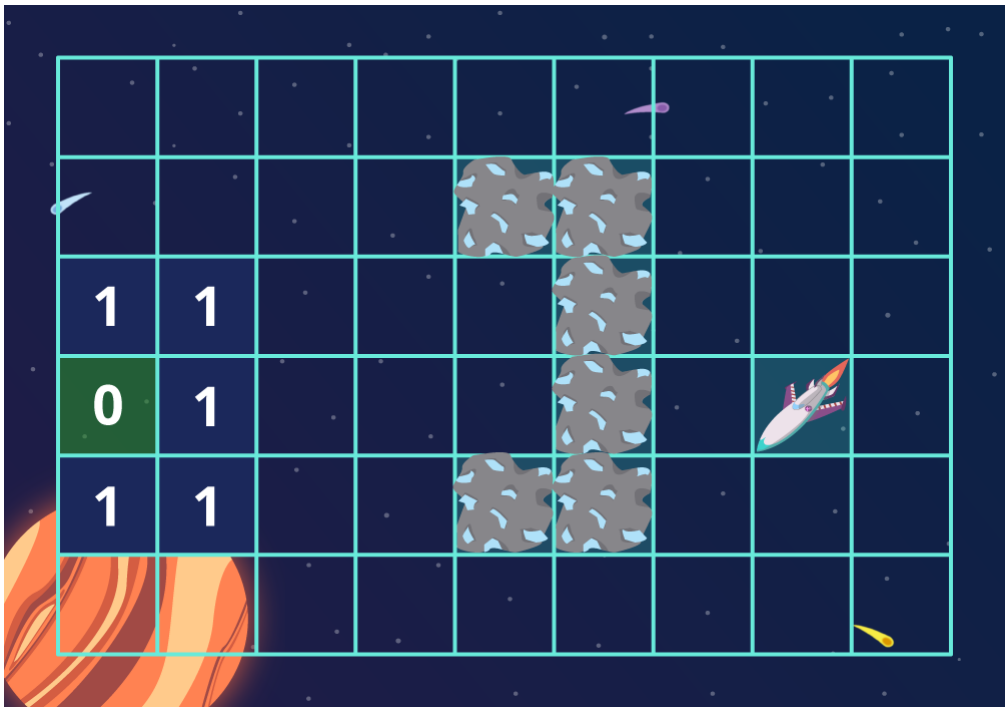
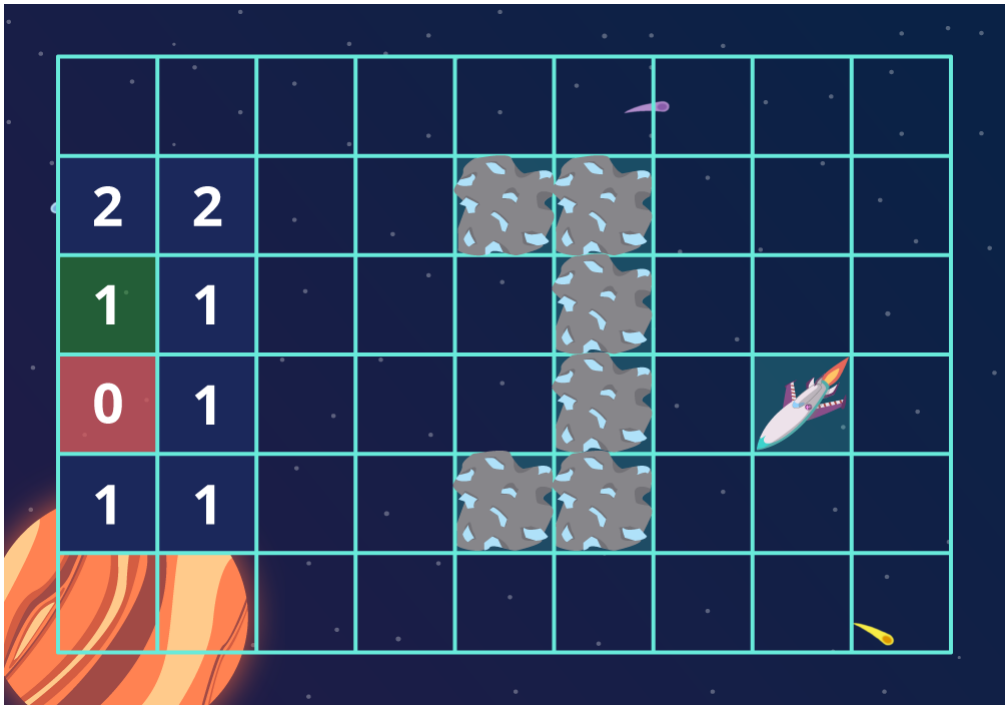


Figura 19 - Passo 01 do patrulheiro estelar



Foi processado o primeiro vértice (origem, marcado de verde) e atualizadas as distâncias dos vizinhos a partir dele. Esse processo será repetido até chegar no vértice destino.

Figura 20 - Passo 02 do patrulheiro estelar



Os vértices visitados passam a ser coloridos de vermelho e não serão mais explorados (o valor contido já é o menor caminho possível até ele). O vértice atual em exploração continua marcado como verde.

Figura 21 - Passo 03 do patrulheiro estelar

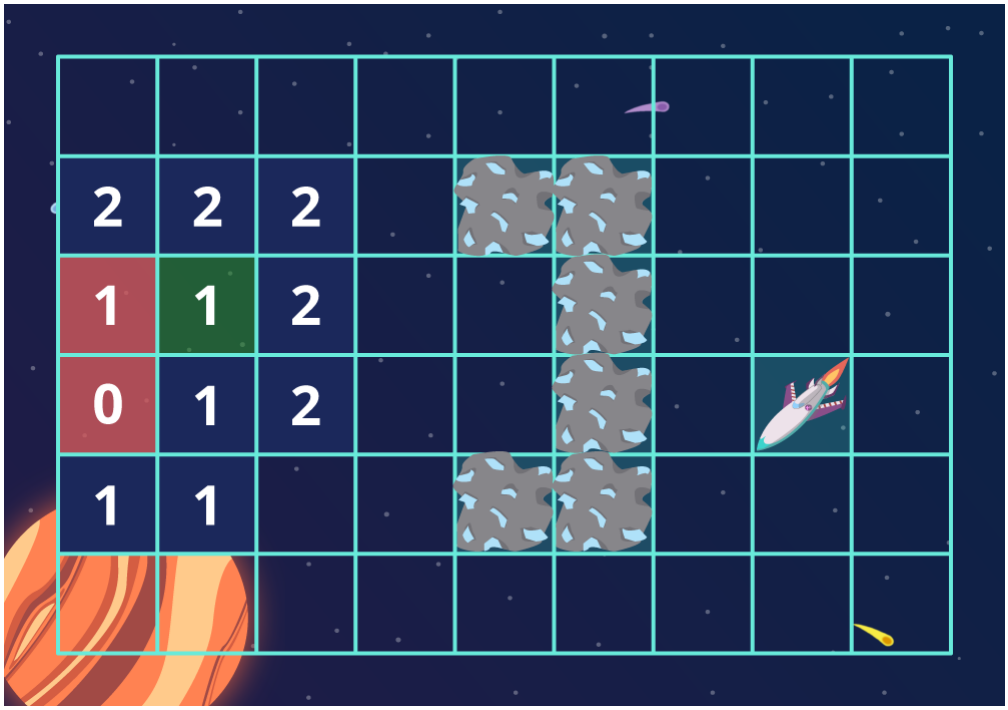


Figura 22 - Passo 04 do patrulheiro estelar

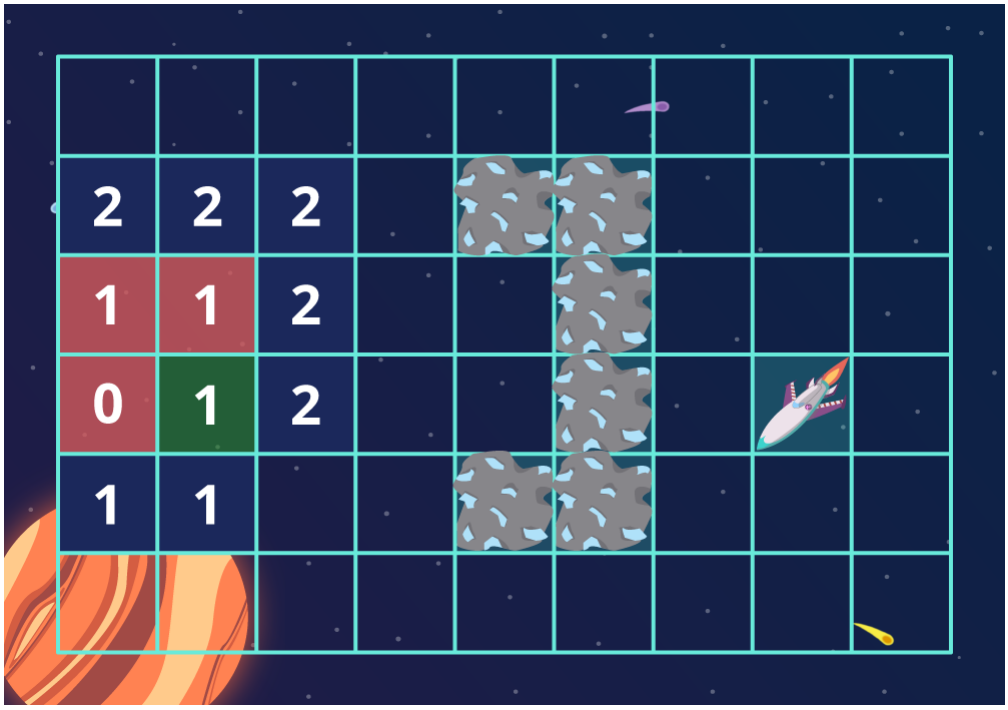


Figura 23 - Passo 05 do patrulheiro estelar

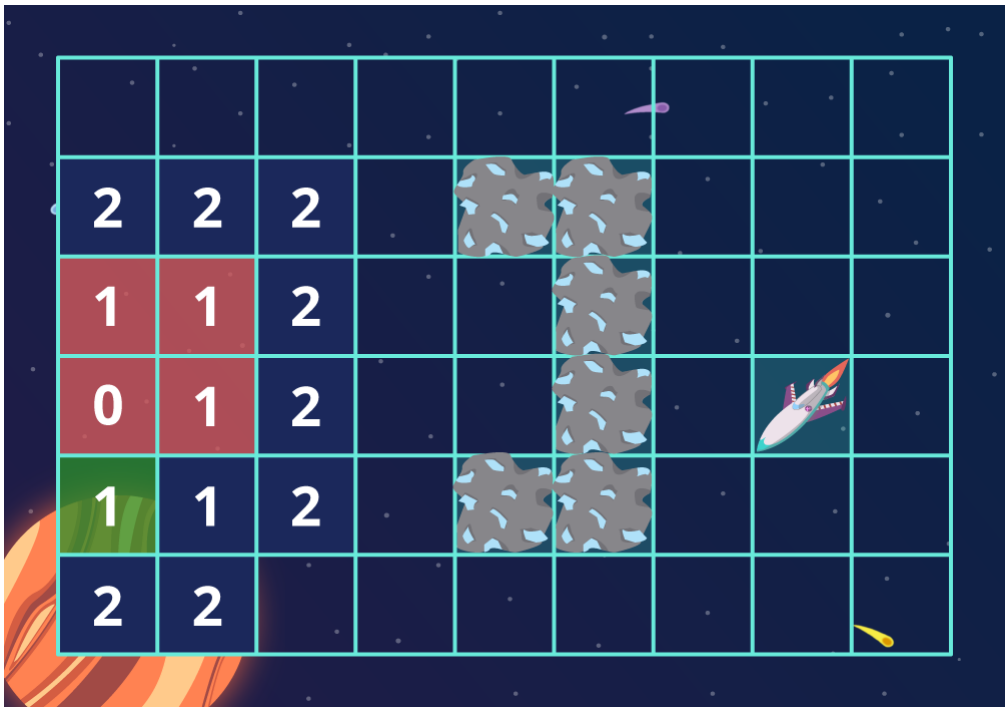


Figura 24 - Passo 06 do patrulheiro estelar

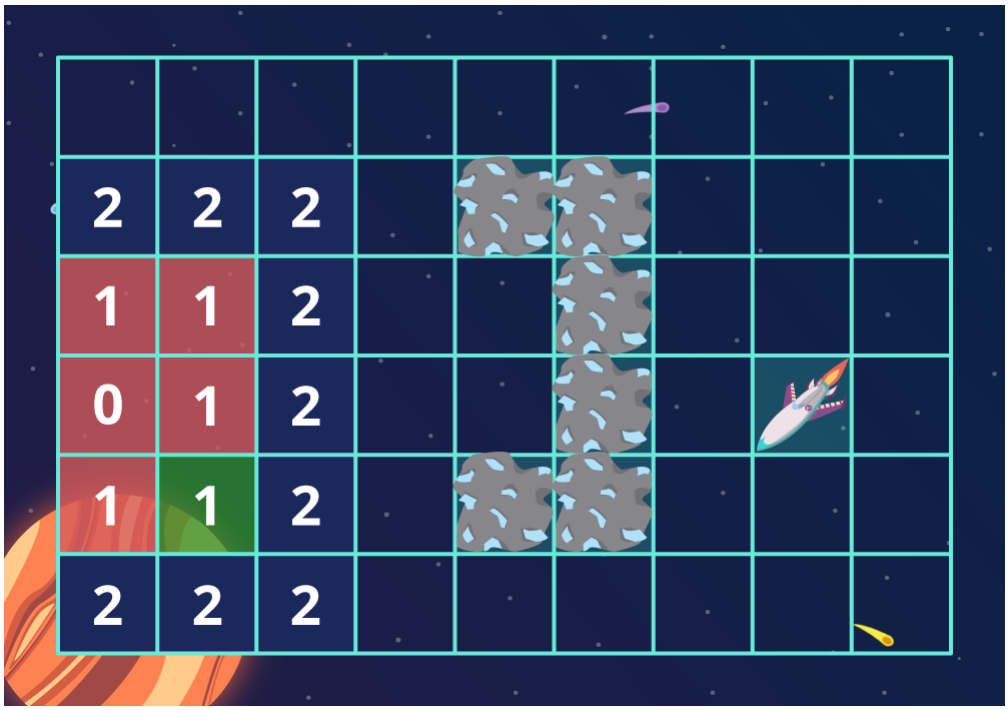


Figura 25 - Passo 07 do patrulheiro estelar

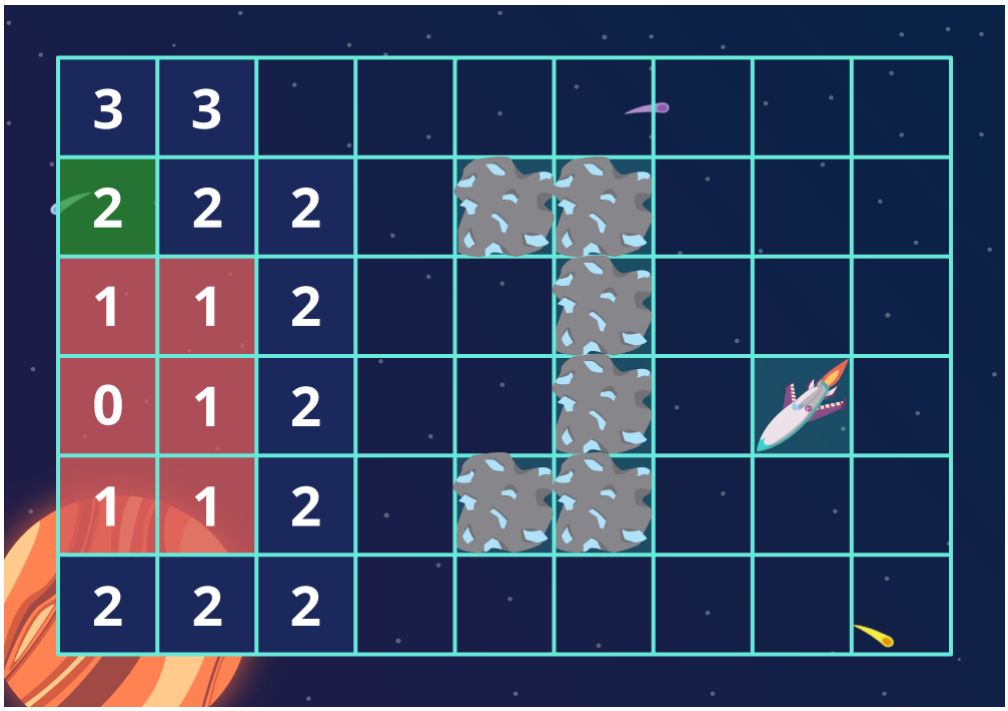


Figura 26 - Passo 08 do patrulheiro estelar

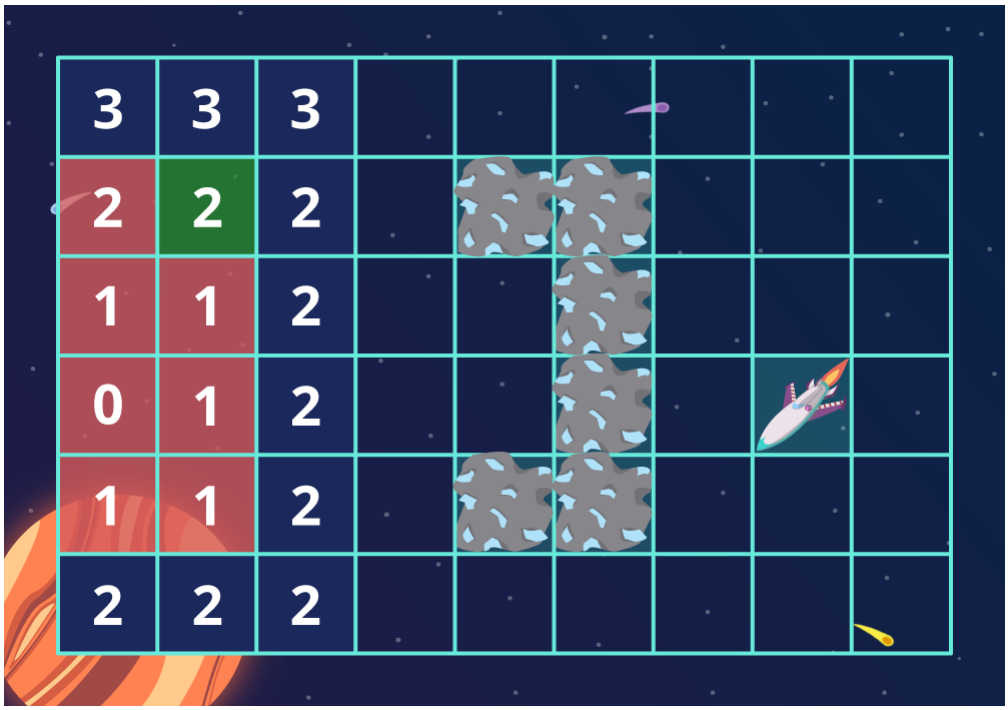


Figura 27 - Passo 09 do patrulheiro estelar

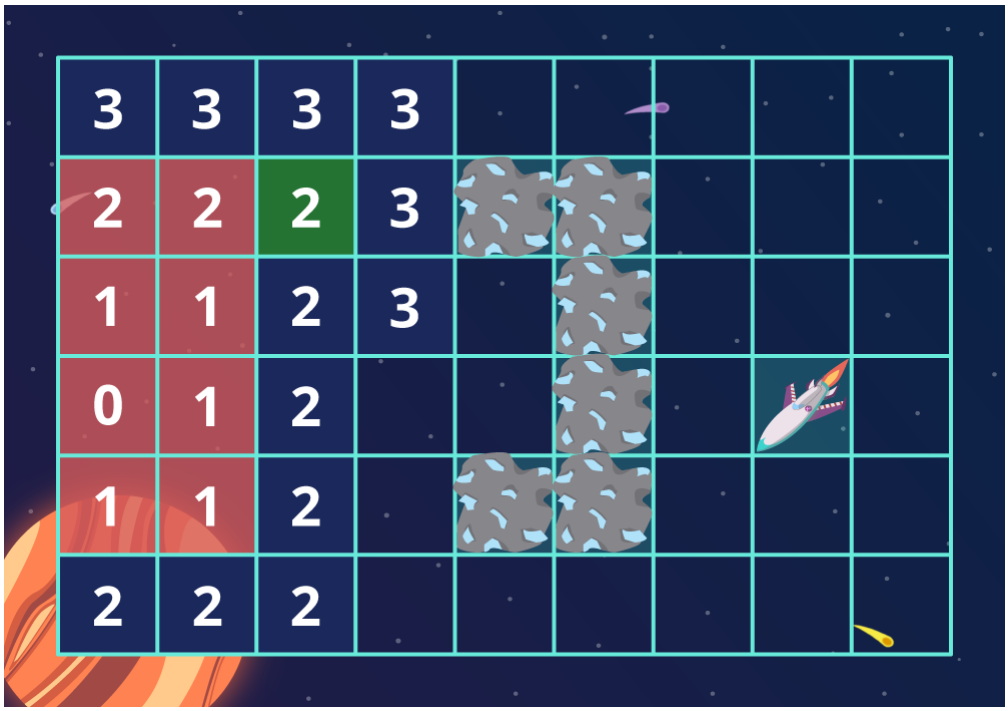


Figura 28 - Passo 10 do patrulheiro estelar

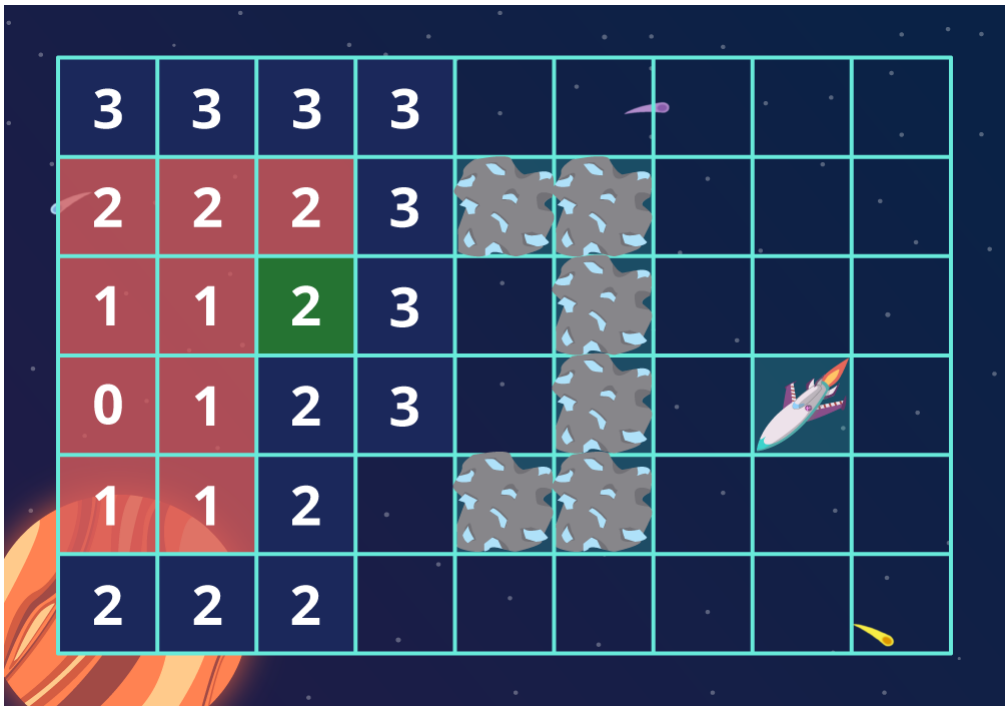


Figura 29 - Passo 11 do patrulheiro estelar

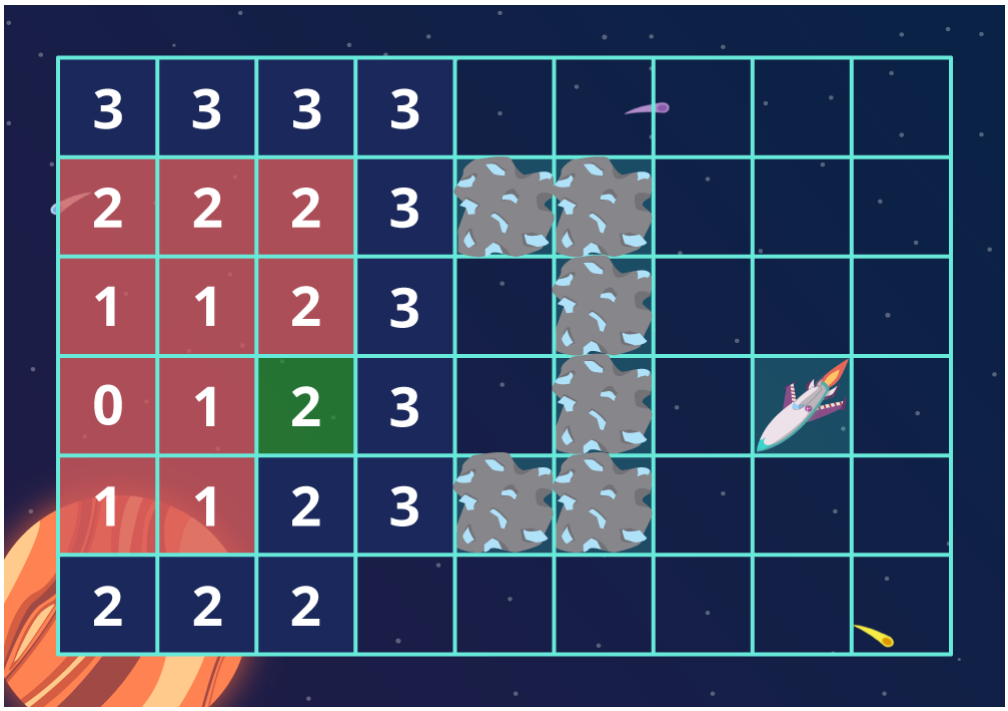
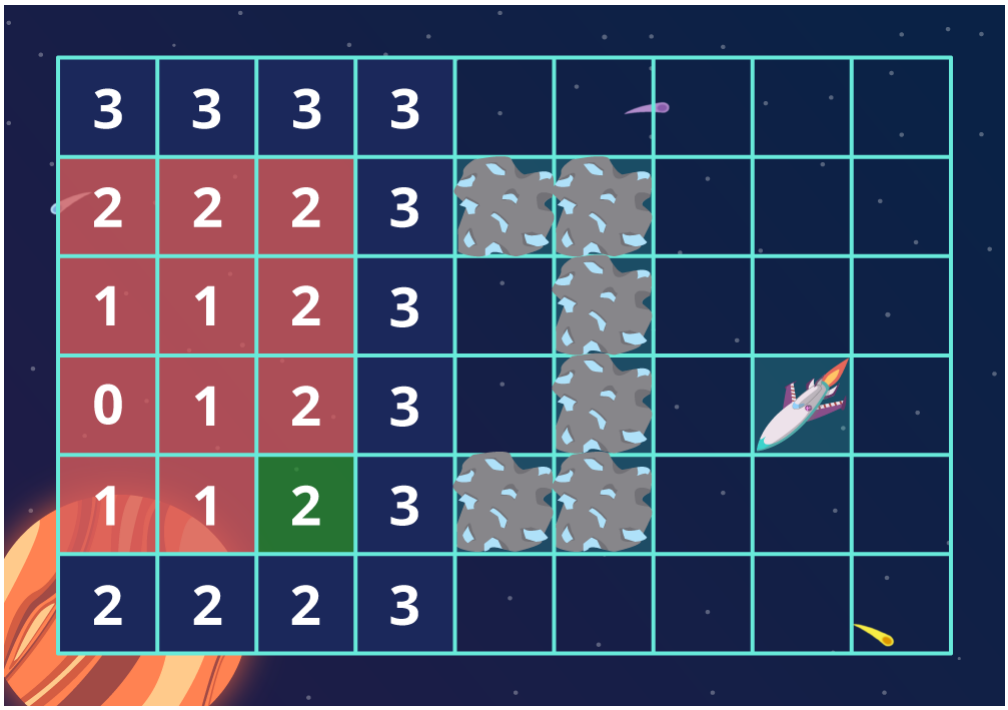
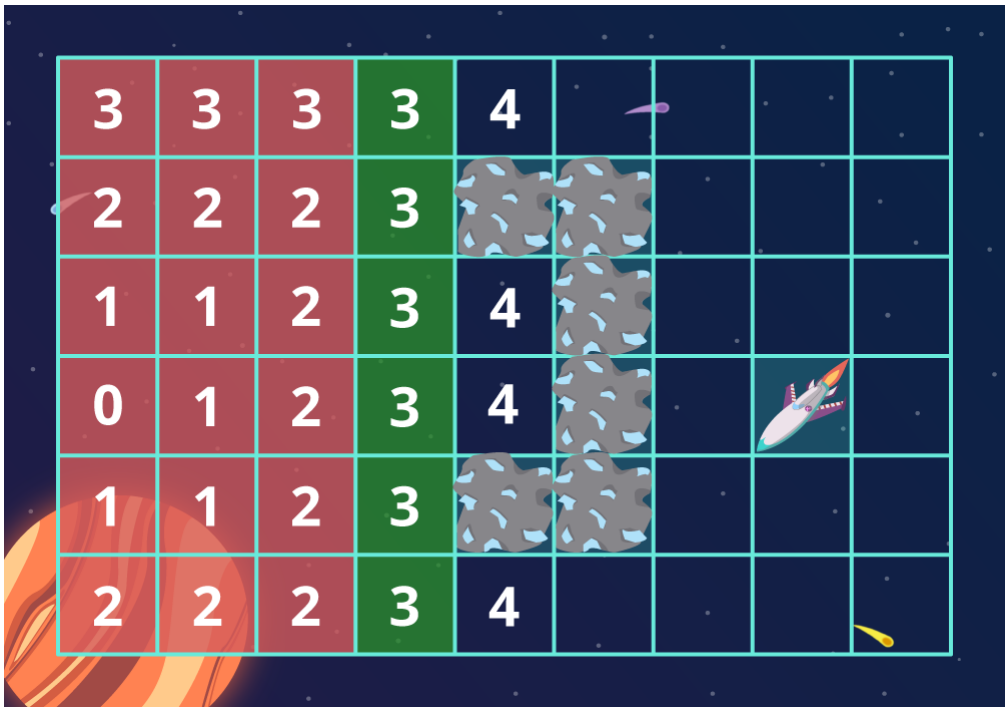


Figura 30 - Passo 12 do patrulheiro estelar



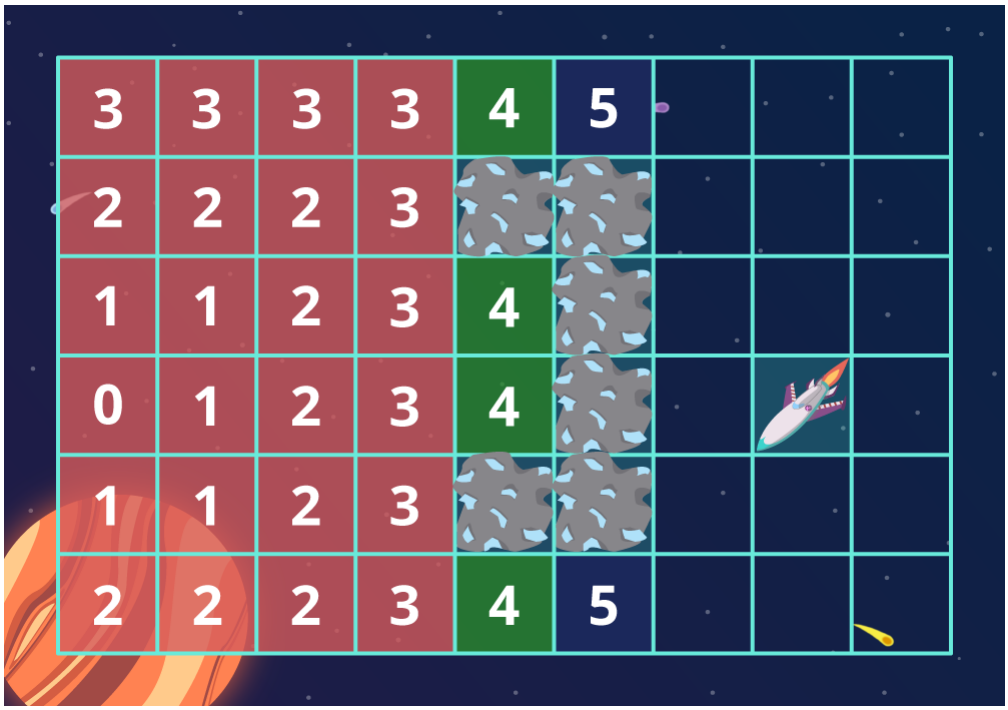
A partir de agora, eu vou “queimar” algumas etapas na sequência senão o pessoal que faz as imagens vai desmaiar de exaustão! 😉 Você verá a exploração de vários vértices simultaneamente, mas a ideia é repetir o processo de um por um. Dessa forma tem-se garantia de o caminho até o vértice será sempre o menor!

Figura 31 - Passo 13 do patrulheiro estelar



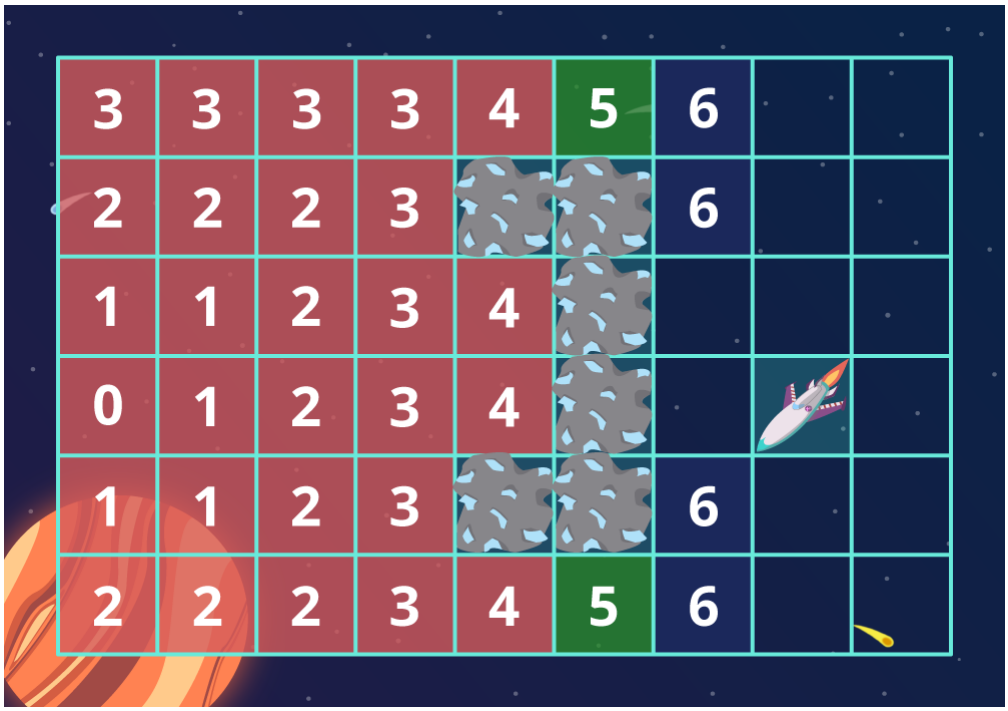
Todos os vértices em uma distância de 3 que tem vizinhos.

Figura 32 - Passo 14 do patrulheiro estelar



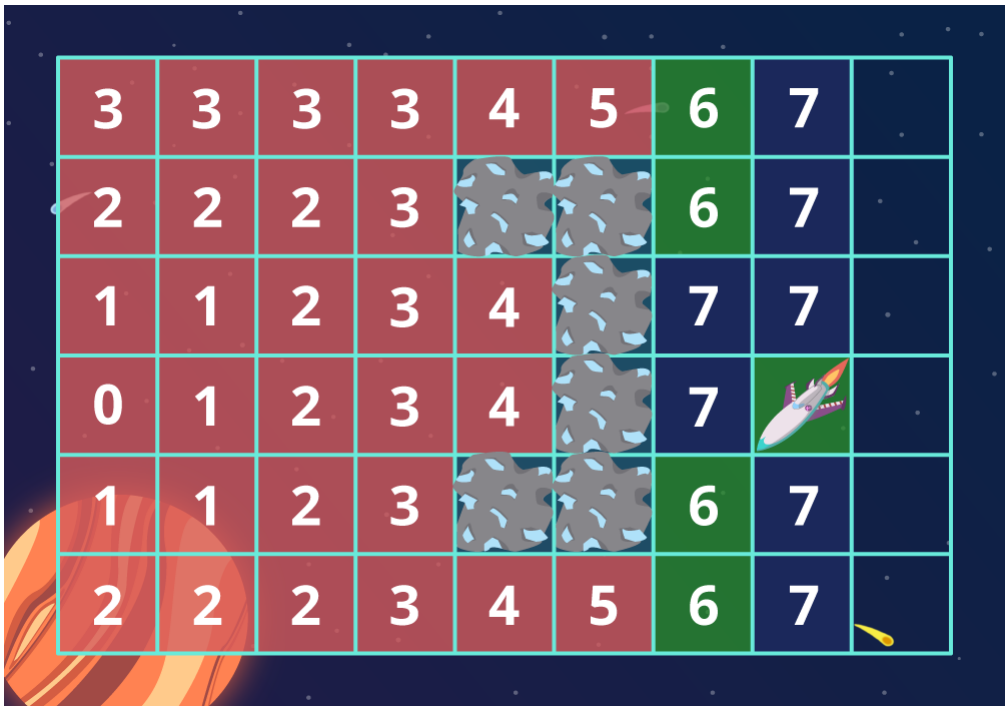
Todos os vértices que percorreram uma distância de 4 casas.

Figura 33 - Passo 15 do patrulheiro estelar



Todos os vértices que percorreram uma distância de 5.

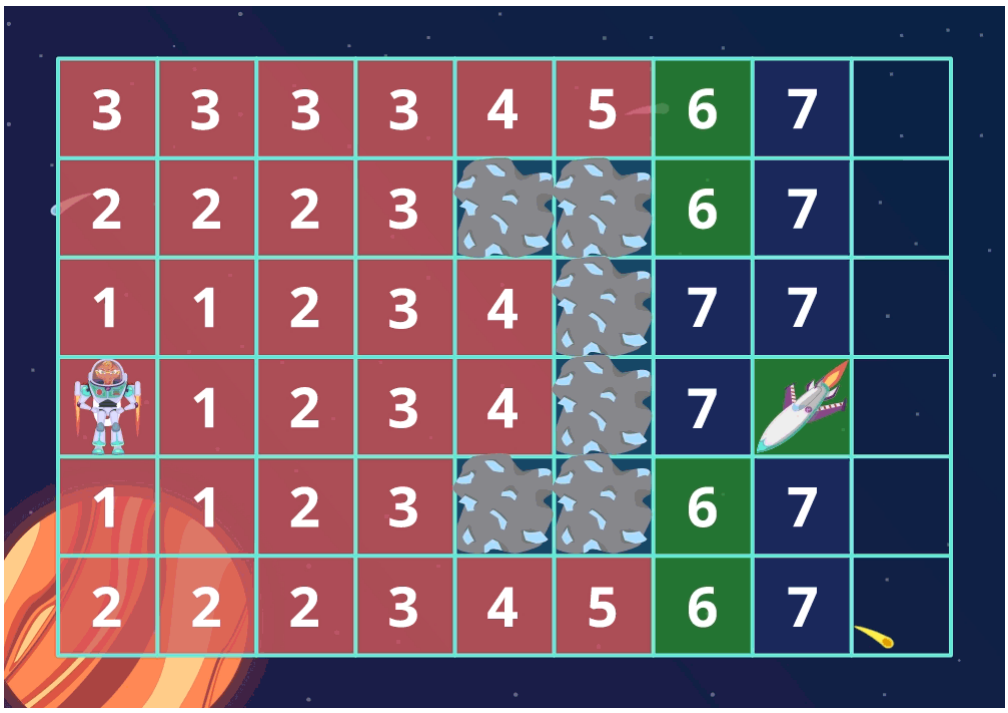
Figura 34 - Passo final!!



Ao explorar todos os vértices que percorreram uma distância de 6, você encontrará um que chega até o destino. Como todas as distâncias são unitárias, tem-se a garantia de que esse primeiro que chegou é o menor caminho, que

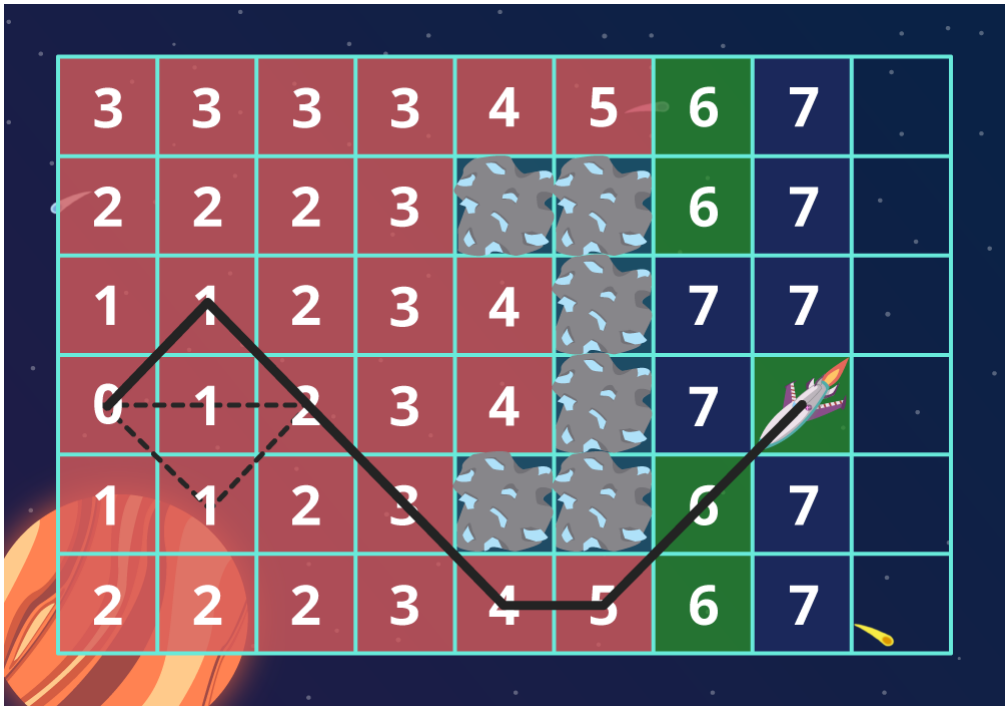
seguindo o algoritmo pode ser expresso pelo seguinte desenho:

Figura 35 - Chegada ao destino!



Perceba que existem mais dois caminhos que possuem o mesmo custo: o que passa por (4,2) e o que passa por (5,2). Qualquer um desses três caminhos satisfazem a condição de ser o menor caminho entre o bloco (4,1) e o bloco (4,8).

Figura 36 - Demarcação do menor caminho entre o bloco (4,1) e o bloco (4,8)

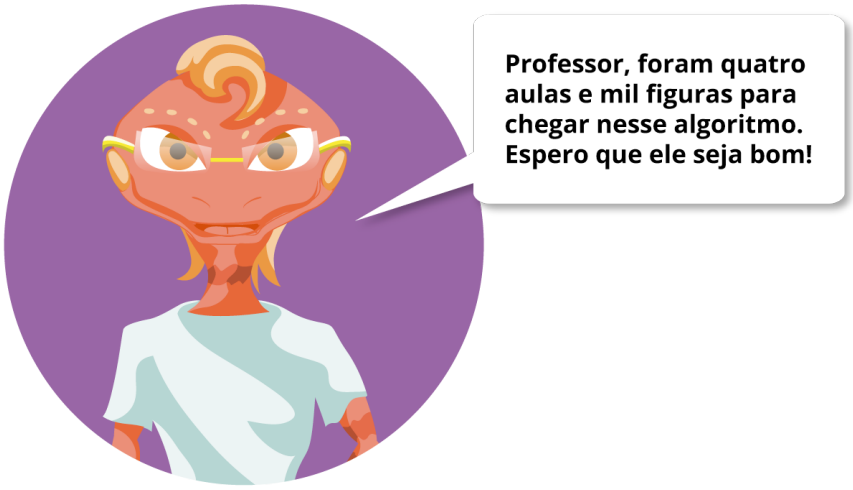


Para esse mapa do exemplo, não é possível ver tanta vantagem no algoritmo de **Dijkstra** frente às outras buscas. Porém, quando o cenário começa a ficar mais complexo, ele torna-se mais eficaz do que as buscas para a solução do caminho:

- Mapas com custos distintos de movimentação (por exemplo, terrenos diferentes);
- Mapas com *layout* mais intrincado, cheios de obstáculos.

Mesmo assim, esse algoritmo ainda precisa visitar um grande número de vértices para achar a resposta correta. Sem entrar em muitos detalhes de complexidade de algoritmos, suponha que, se o seu mapa tiver 5000 blocos, ele ainda quebra bem o galho. Mas em um mapa com 1.000.000 de blocos, aí os seus caminhos iam demorar muito tempo para serem calculados! E não é difícil pensar mapas grandes, dado o tamanho de alguns cenários em jogos atuais! Por esse motivo, o algoritmo de **Dijkstra** tradicional não é tão usado para achar o caminho entre dois pontos, mas sim uma otimização que foi criada a partir dele: o algoritmo **A***.

3 - Caminho eficaz: Algoritmo A*



Ehr... sem pressão. 😁

O algoritmo **A*** é um dos mais utilizados em situações de *pathfinding*, ou **busca de caminhos**. Ele otimiza o algoritmo de **Dijkstra** em dois aspectos para tornar o seu funcionamento mais eficiente:

- Ele utiliza uma estrutura de dados chamada **Fila** de prioridade para organizar os vértices que serão explorados;

- Além de salvar os caminhos já calculados (como **Dijkstra** faz), ele também utiliza **heurísticas** para estimar em cada ponto quanto ainda falta para o final, buscando direcionar a escolha do próximo vértice.

A ideia do algoritmo é tentar acertar o menor caminho o mais rápido possível! Por isso ele olha tanto para o passado (o quanto ele já percorreu) como para o futuro (o quanto ainda falta) na hora de estimar se um caminho é bom ou não. Calma, não estou dizendo que ele acerta de primeira, mas ele converge para o menor caminho muito mais rápida e precisamente do que **Dijkstra** ou as buscas por **DFS** e **BFS** conseguem fazer. De certa forma, esse algoritmo é uma combinação das duas abordagens que foram apresentadas nas seções anteriores dessa aula, utilizando o que há de melhor em cada um dos métodos.

Por convergir mais rápido, o algoritmo **A*** acaba explorando menos vértices para achar a solução, e consequentemente possui uma performance melhor (por isso o povo de jogos usa bastante ele!). Se você não usar nenhuma **heurística** para mensurar o caminho para a frente do vértice atual, estará olhando apenas o caminho já percorrido, e o algoritmo se iguala ao de **Dijkstra**.

Você deve estar se perguntando: o que é uma **Fila de prioridade**? É a mesma estrutura que eu já estudei?

Não! Essa seria uma estrutura de dados nova. Nessa estrutura, foi definido um critério de prioridade, e os elementos que mais atendem esse critério são os primeiros a serem acessados em uma consulta à estrutura. Por exemplo, se for definida uma **Fila de prioridade** para valores inteiros, e que o critério é o menor valor, então o menor valor existente entre os elementos da **Fila** será o primeiro a ser acessado em uma consulta. Independentemente da sequência em que você realiza a inserção dos valores, o menor valor sempre será o primeiro a sair da **Fila**. Ou seja, o menor critério sempre fura para o começo da fila, porque tem a prioridade, captaram?

A implementação desse tipo de estrutura exigiria que você tivesse estudado outras estruturas, como a *heap*... então essa parte eu vou abstrair. Tudo o que você precisa saber é que, numa **Fila de prioridade** eu sempre consigo pegar o menor elemento de acordo com o meu critério de forma imediata, ok?

Com esses elementos em mente, o pseudocódigo do **A*** pode ser descrito como no código abaixo.

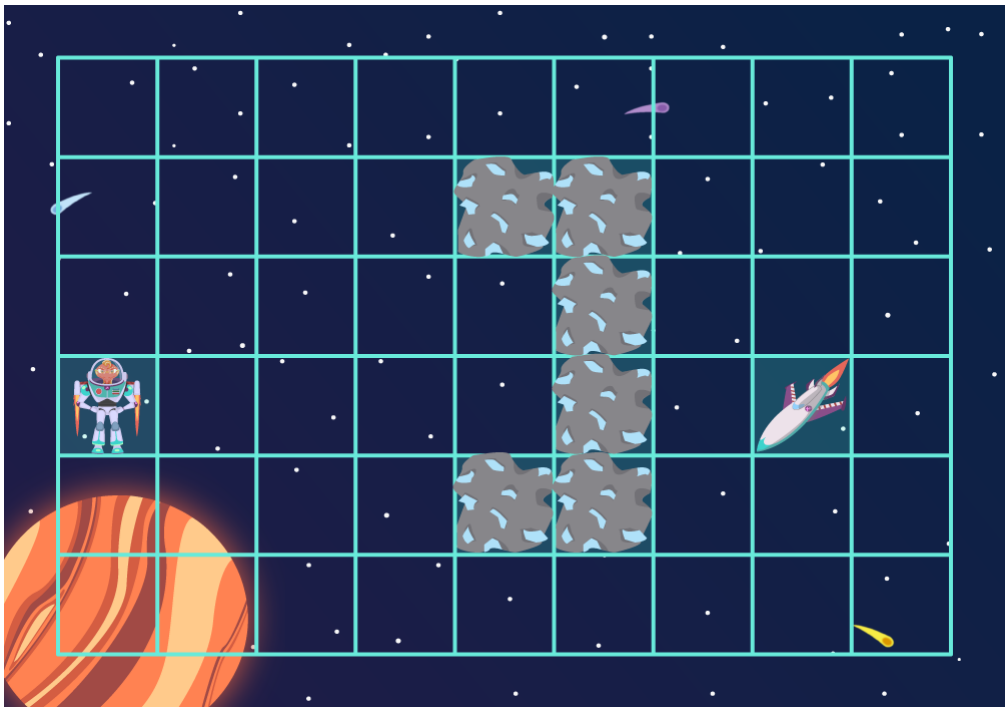
Código 02 – Pseudocódigo do algoritmo **A***

```
1  Busca_A*(Grafo g, Vértice inicio, Vértice fim){
2     Visitados = {} //Nenhum vértice foi visitado ainda
3     Abertos ={inicio} //Vértices na fila para exploração
4
5     custo[] // Vetor que representa os custos dos caminhos para cada
6
7     //vértice
8     anterior[] // Vetor que representa o vértice anterior no caminho
9     futuro[] //Vetor que representa o custo futuro do caminho
10
11    Enquanto Abertos não for vazio{
12       atual = vértice de Abertos que possui menor valor de caminho futuro
13
14       Se atual == fim
15          Retorne o caminho
16
17       Remove atual de Abertos
18       Insere atual em Visitados
19
20       Para cada vizinho de atual{
21          Se o vizinho não foi visitado{
22             Se ele não estiver em Abertos
23                Insere vizinho em Abertos
24
25             custo_vizinho = custo[atual] + g[atual, vizinho]
26
27             Se custo_vizinho < custo[vizinho]{
28                anterior[vizinho] = atual
29                custo[vizinho] = custo_vizinho
30                futuro[vizinho] = custo[vizinho]+ HEURÍSTICA
31             }
32          }
33       }
34    }
35    Retorna ERRO
36 }
```

Perceba que o código fica um pouco mais complexo na medida em que você adiciona as otimizações. Que tal uma simulação com o exemplo dos casos anteriores, para você poder visualizar melhor o funcionamento do algoritmo?

O bom e velho mapa!

Figura 37 - Passo inicial!



O primeiro passo é processar o vértice inicial e calcular os primeiros valores. Como **heurística**, resolvi fazer esse com a **distância Euclidiana**, que é uma **heurística** bastante utilizada em diversas aplicações da computação. A conta fica um pouquinho maior, mas como quem vai fazer é o computador, então tá tudo tranquilo. 😁

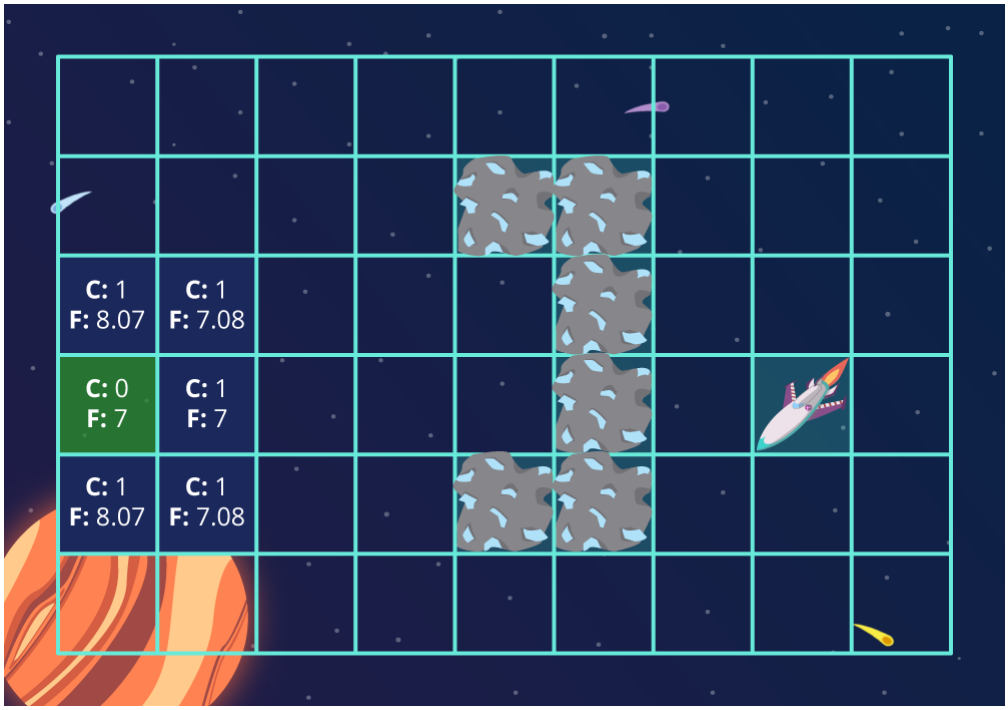
Lembrando: a posição do destino é (4,8) e a de partida do jogador é (4,1). Todas as contas de coordenadas são feitas da posição do vértice com a posição do destino, ok?

Aqui serão dois conjuntos de valores calculados para cada um dos vértices: Denomine de C o valor do caminho percorrido até o vértice, e F o valor estimado do restante do caminho a partir do vértice. O custo de movimentação C é unitário, então você precisa fazer a conta do valor da **heurística** para somar com esses custos:

F(4,1) = RAIZ_QUADRADA(0 + 49) + C(4,1) = 7 + 0 = 7
F(3,1) = RAIZ_QUADRADA(1 + 49) + C(3,1) = 7.07 + 1 = 8.07
F(3,2) = RAIZ_QUADRADA(1 + 36) + C(3,2) = 6.08 + 1 = 7.08
F(4,2) = RAIZ_QUADRADA(0 + 36) + C(4,2) = 6 + 1 = 7
F(5,1) = RAIZ_QUADRADA(1 + 49) + C(5,1) = 7.07 + 1 = 8.07
F(5,2) = RAIZ_QUADRADA(1 + 36) + C(5,2) = 6.08 + 1 = 7.08

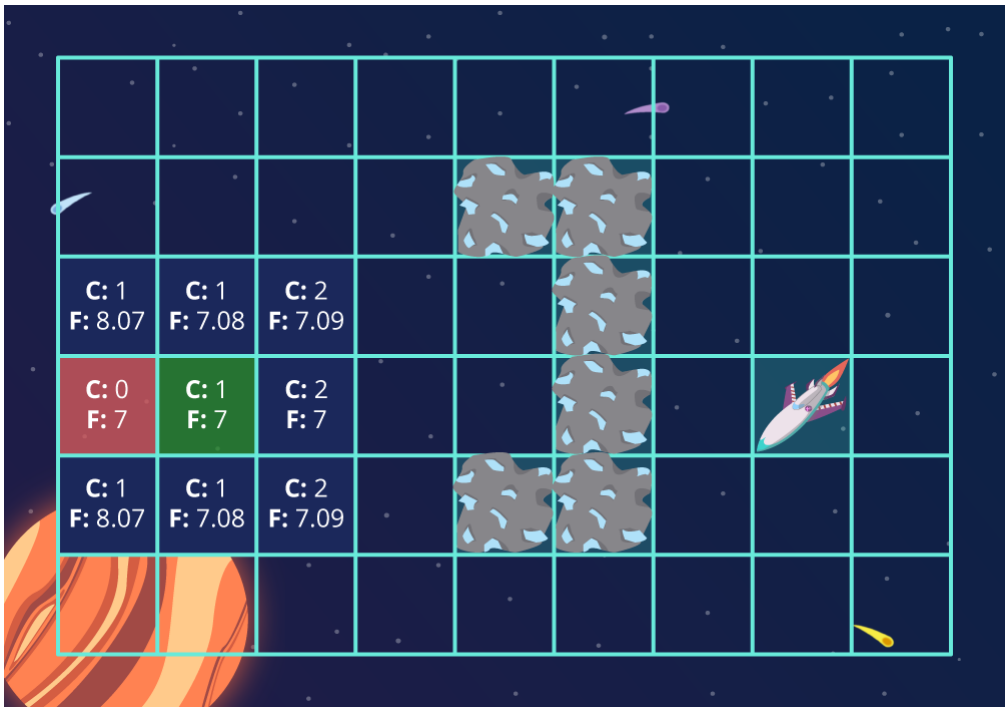
Entendeu a ideia? Cada vértice vai ter duas informações: C e F.

Figura 38 - Passo 01 do patrulheiro estelar



A partir desse ponto, escolhe-se sempre o vértice que tiver o menor F como o próximo a ser explorado. Em caso de empate, vai ser o primeiro que foi adicionado na fila.

Figura 39 - Passo 02 do patrulheiro estelar



Valide as contas do valor de F, para ver se você está seguro:

F(3,3) = RAIZ_QUADRADA(1 + 25) + C(3,3) = 5.09 + 2 = 7.09
F(4,3) = RAIZ_QUADRADA(0 + 25) + C(4,3) = 5 + 2 = 7
F(5,3) = RAIZ_QUADRADA(1 + 25) + C(5,3) = 5.09 + 2 = 7.09

A partir daqui vou colocar as contas dos passos diretamente, mas você pode fazer a conta para validar, certo?

Figura 40 - Passo 03 do patrulheiro estelar

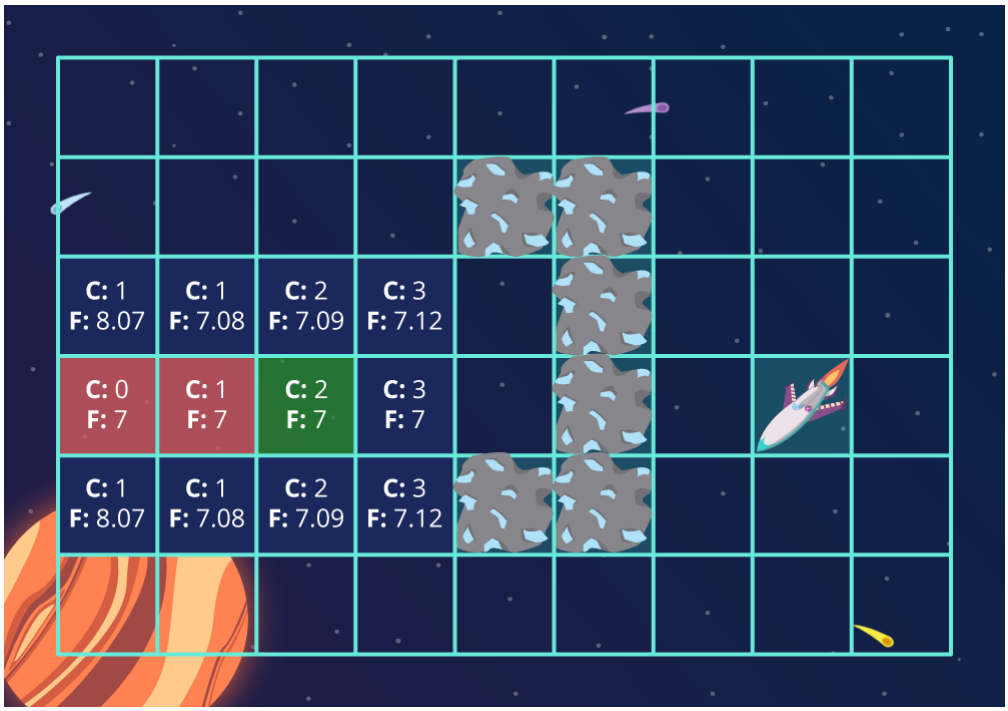


Figura 41 - Passo 04 do patrulheiro estelar

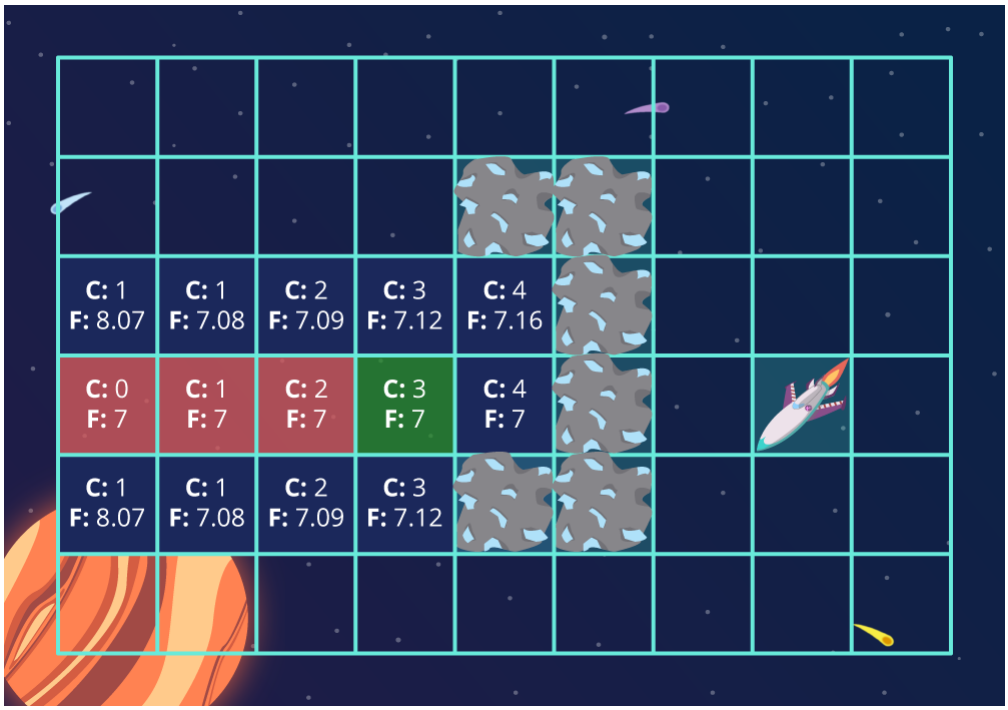
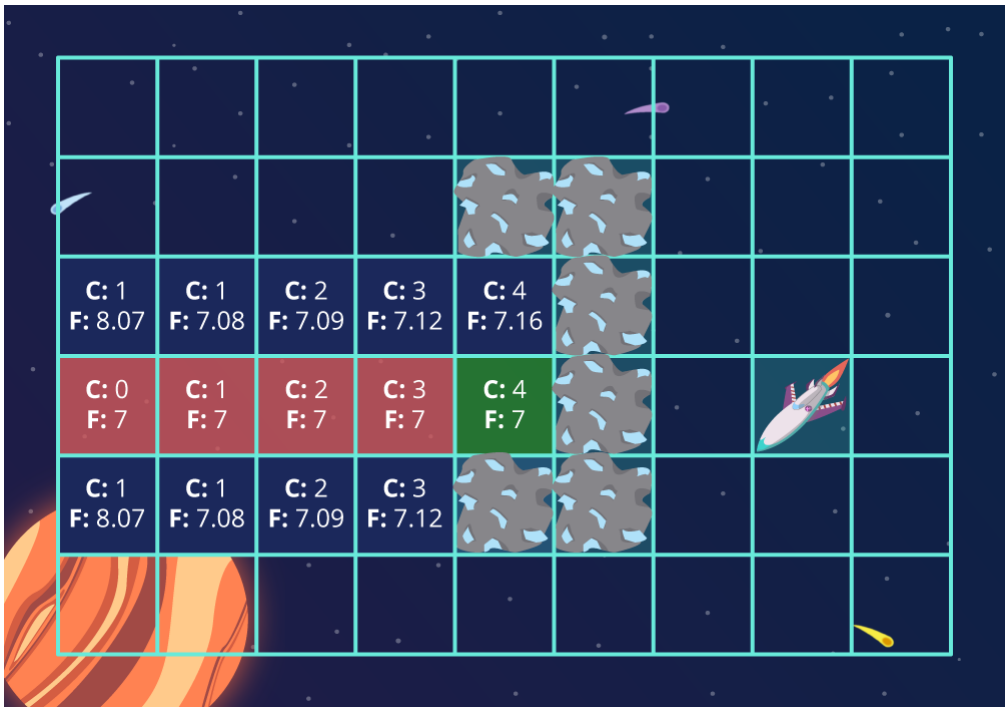


Figura 42 - Passo 05 do patrulheiro estelar



Esse vértice não tem vizinhos para atualizar: ou o caminho a partir dele fica pior, ou o vizinho já foi visitado. Você irá só processá-lo e seguir para o próximo menor F!

Figura 43 - Passo 06 do patrulheiro estelar

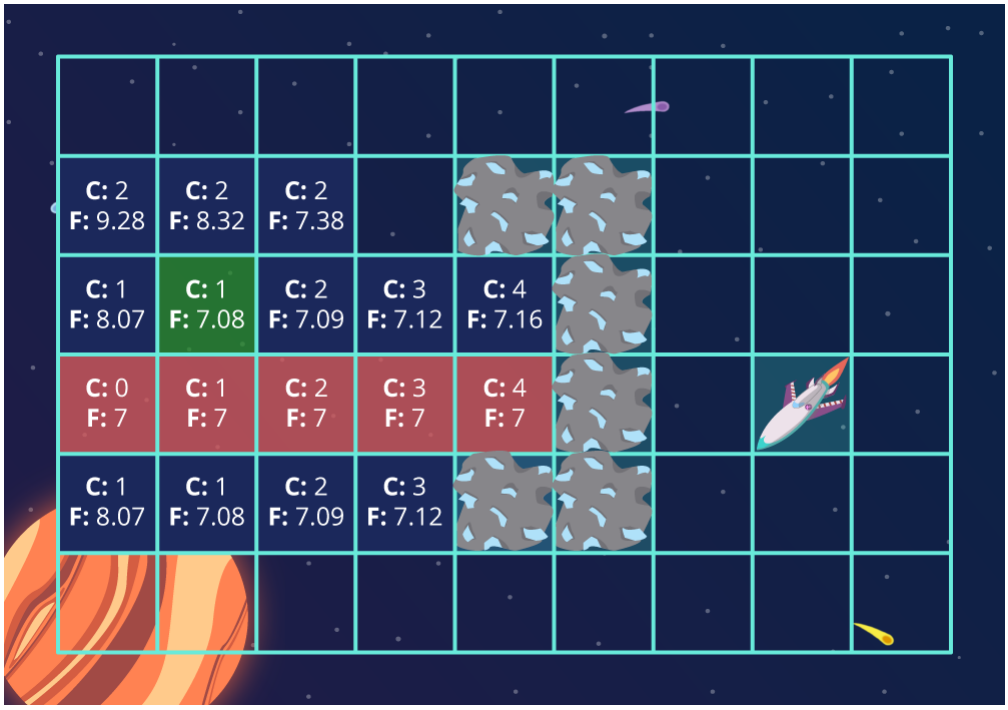


Figura 44 - Passo 07 do patrulheiro estelar

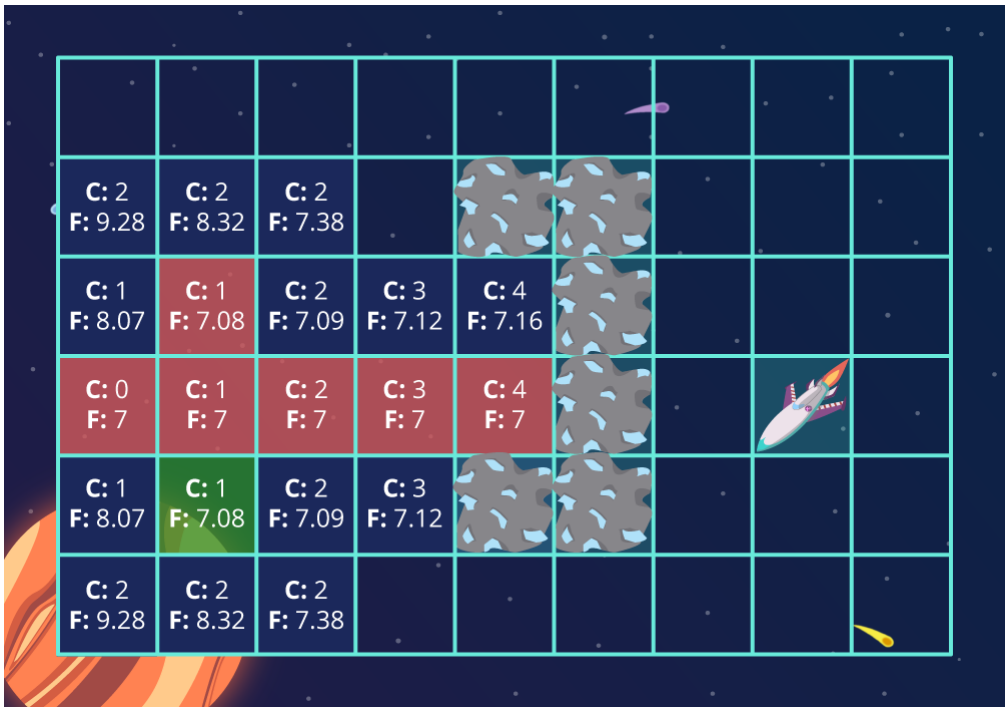


Figura 45 - Passo 08 do patrulheiro estelar

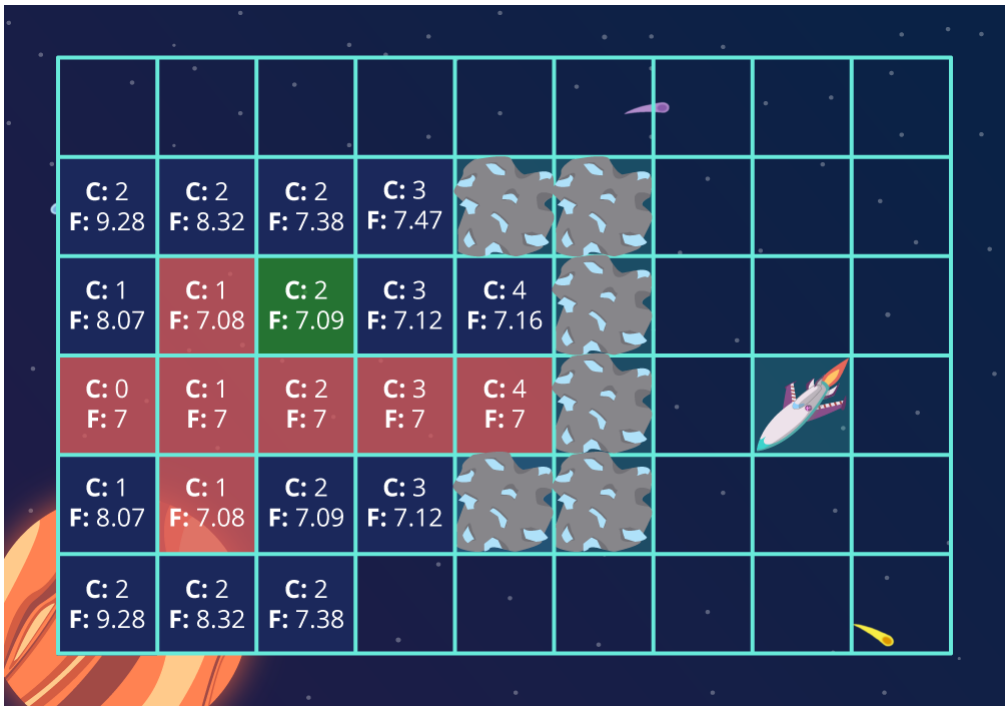


Figura 46 - Passo 09 do patrulheiro estelar

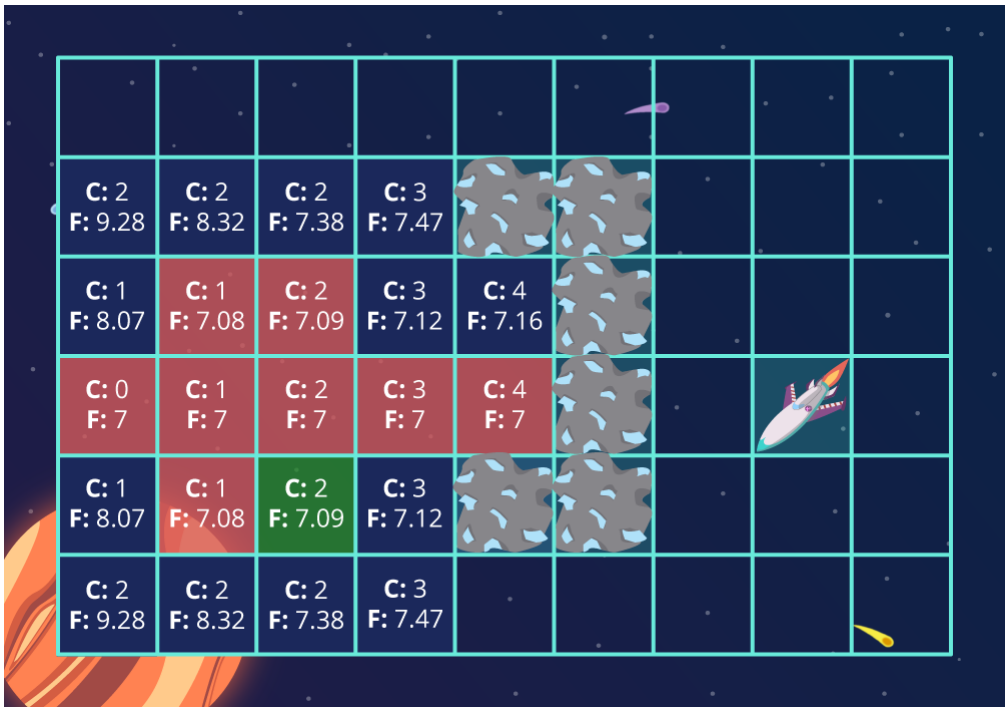


Figura 47 - Passo 10 do patrulheiro estelar

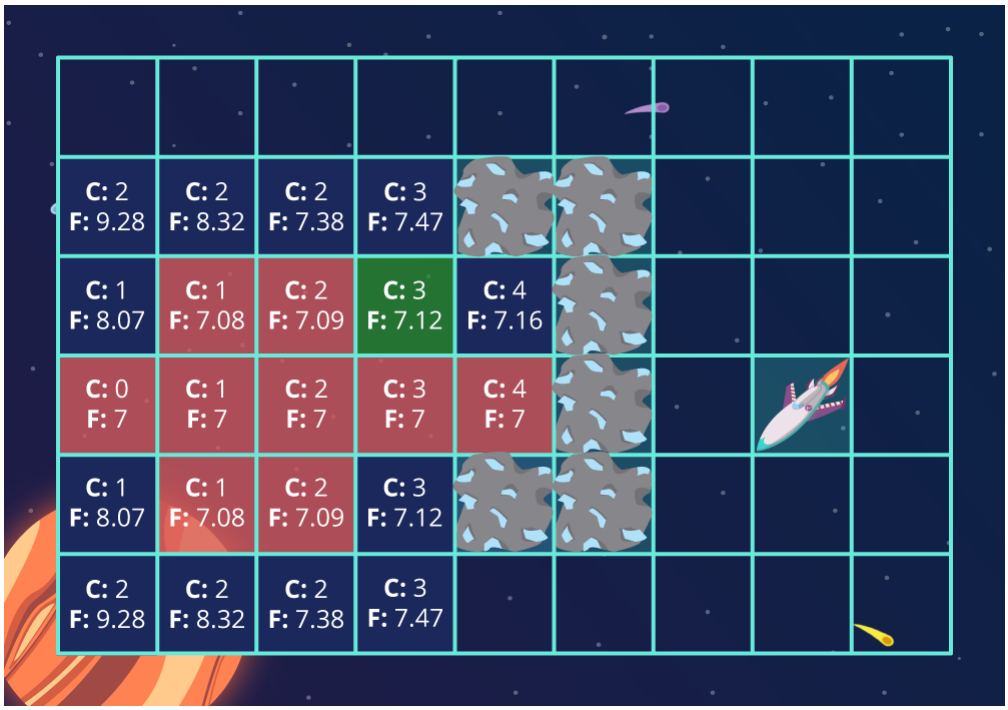


Figura 48 - Passo 11 do patrulheiro estelar

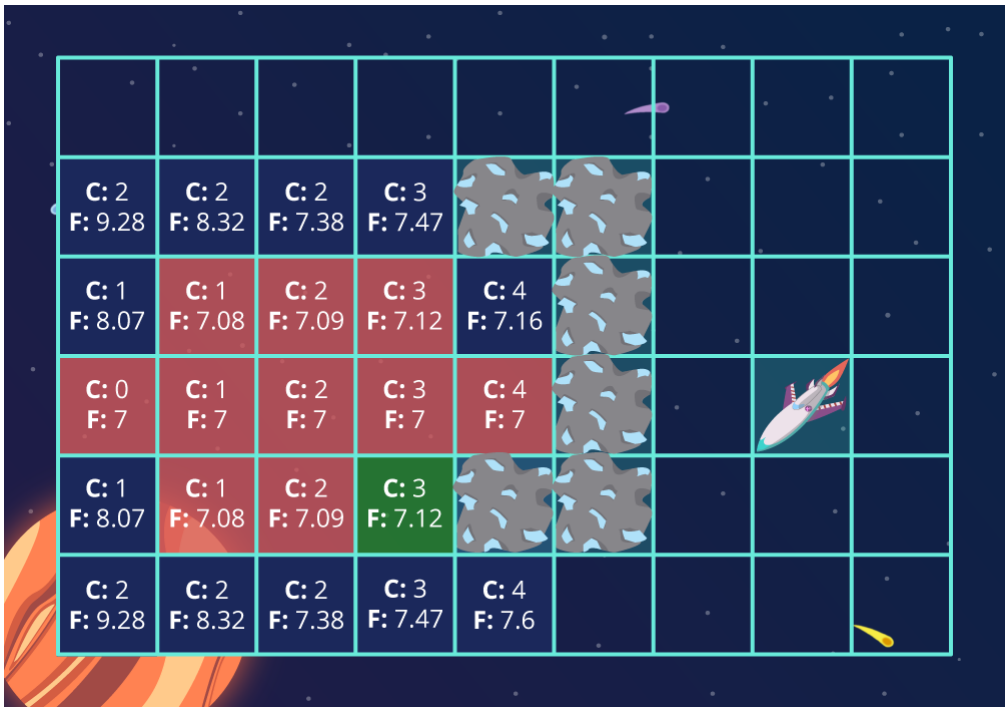


Figura 49 - Passo 12 do patrulheiro estelar

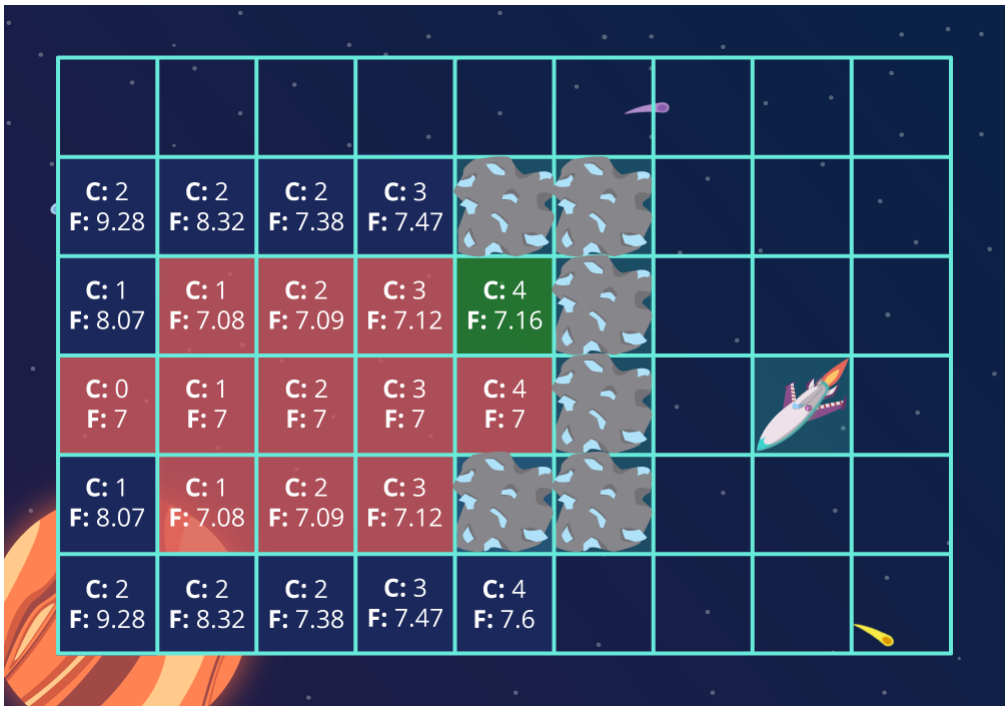


Figura 50 - Passo 13 do patrulheiro estelar

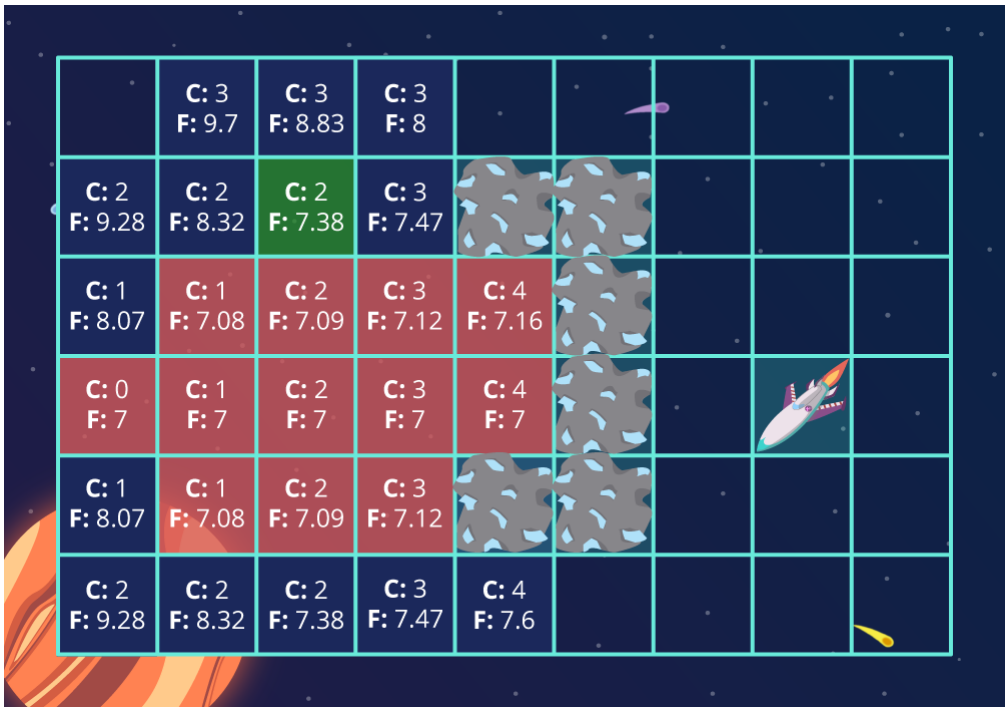


Figura 51 - Passo 14 do patrulheiro estelar

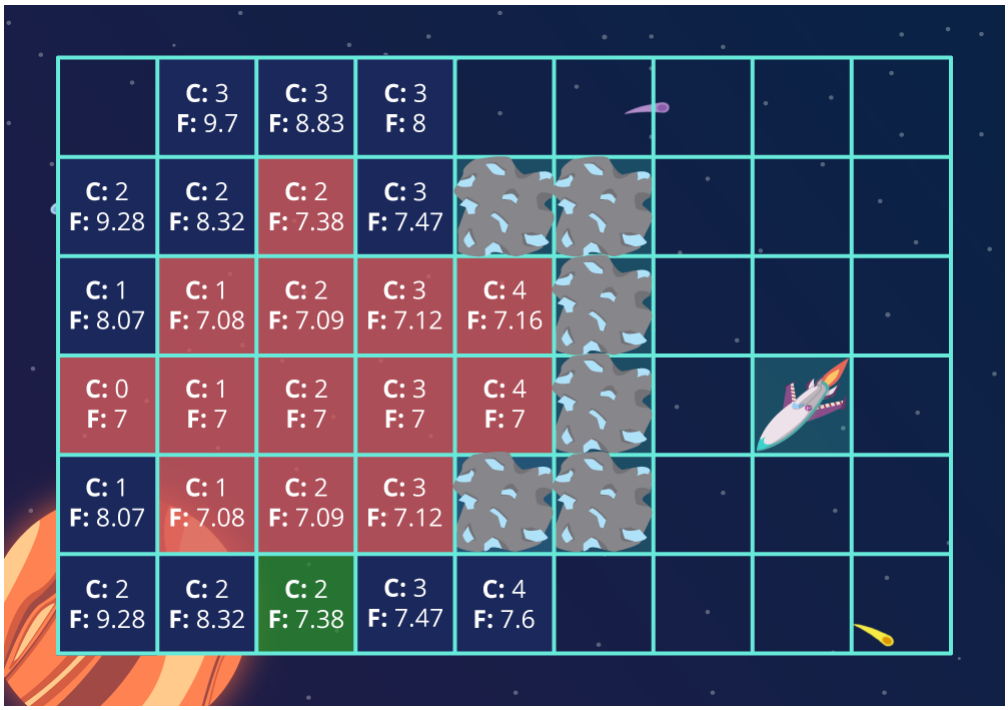


Figura 52 - Passo 15 do patrulheiro estelar

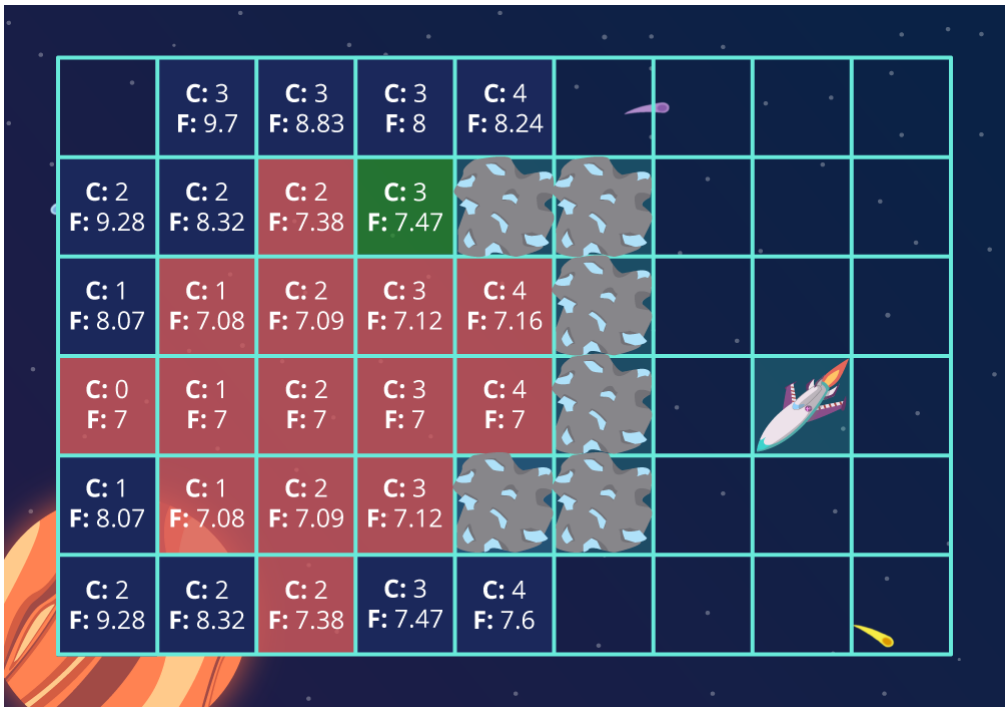


Figura 53 - Passo 16 do patrulheiro estelar

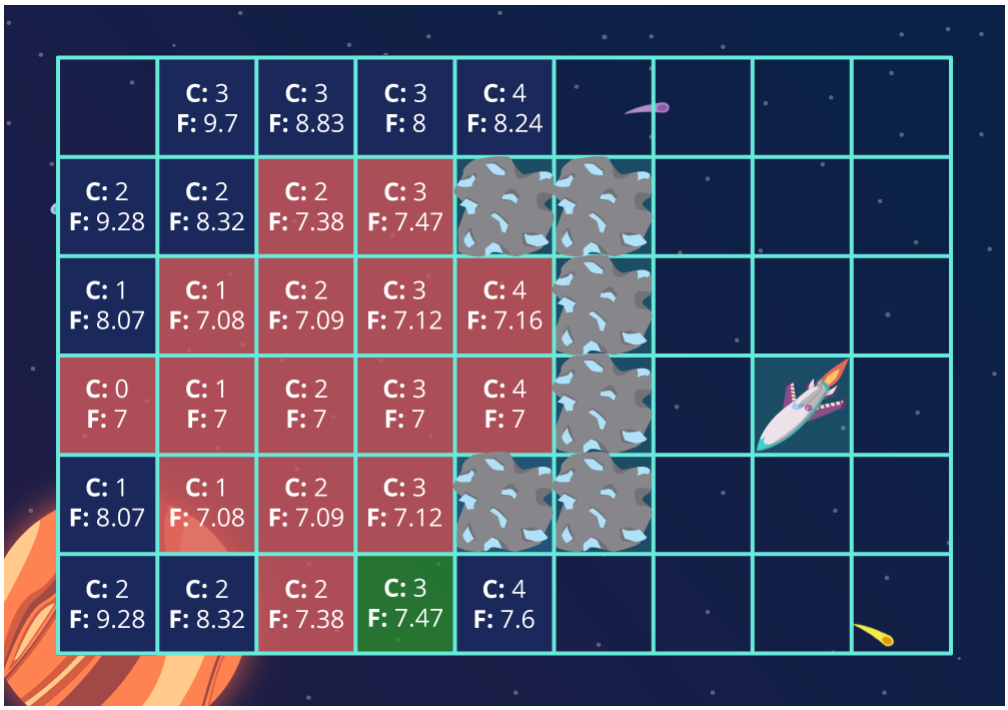


Figura 54 - Passo 17 do patrulheiro estelar

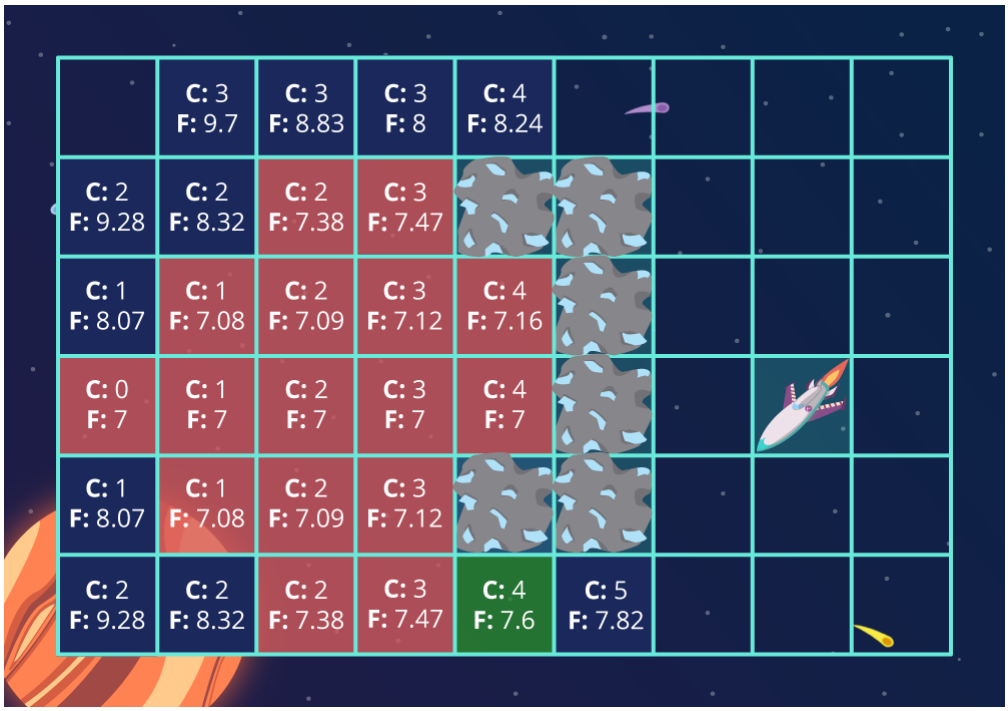


Figura 55 - Passo 18 do patrulheiro estelar

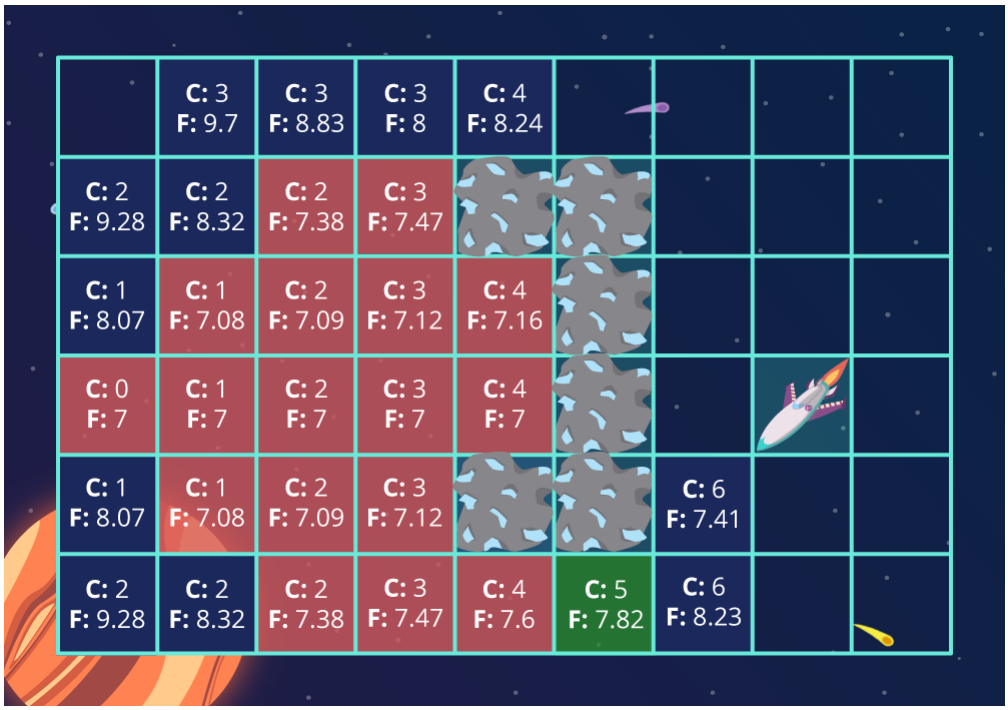
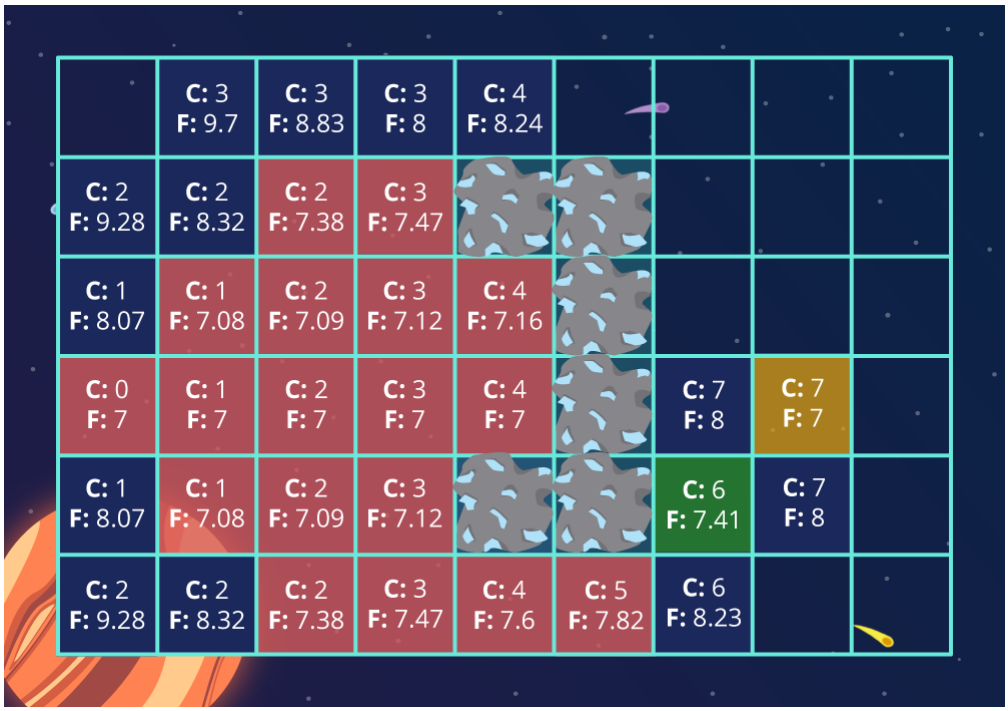
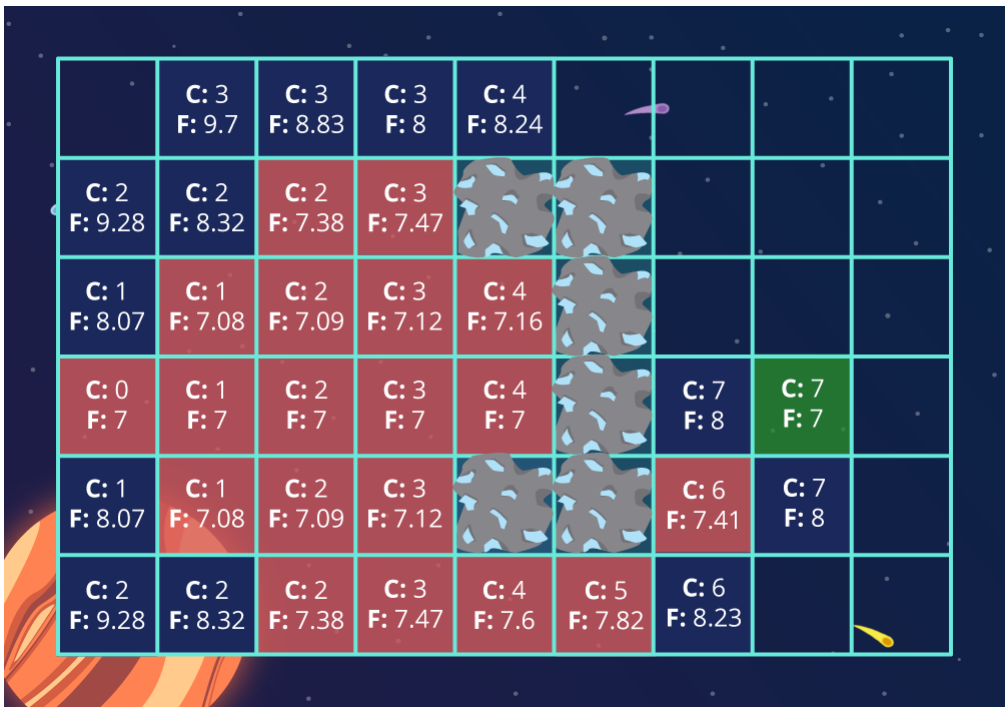


Figura 56 - Passo 19 do patrulheiro estelar



Quase lá! Substitui a figura da nave pelos valores que são calculados para o vértice destino, para mostrar que ele seria a escolha certa ao final do algoritmo.

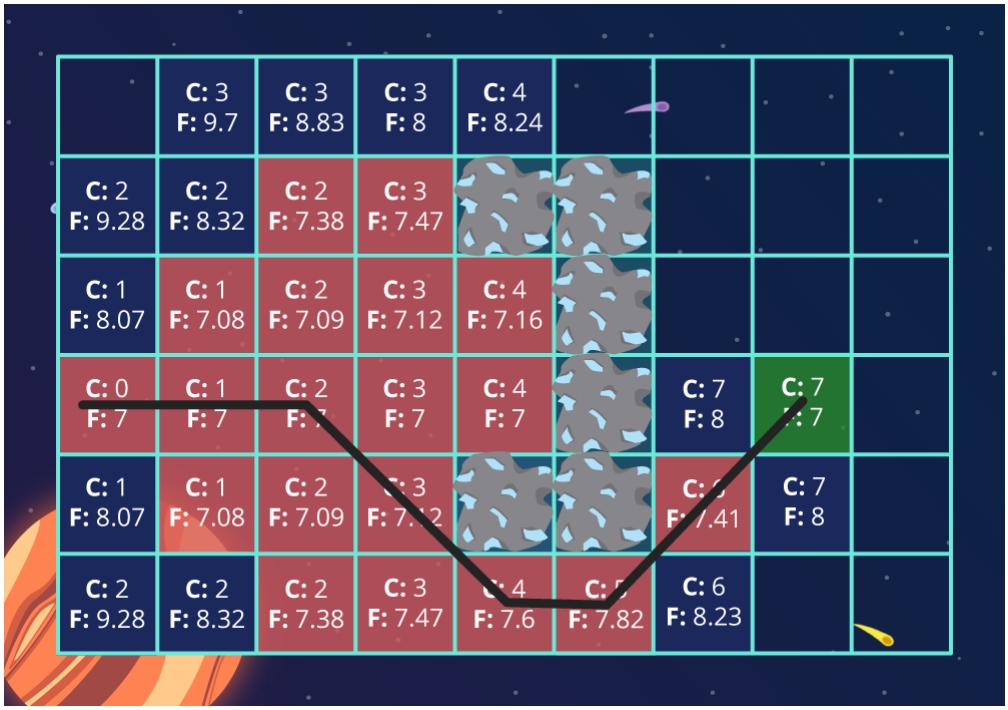
Figura 57 - Passo 20 do patrulheiro estela



Chegou! 👏 Ufa, 20 passos para convergir para o menor caminho. 😁

Eu não adicionei a informação do mapa anterior nessa imagem, mas o caminho traçado como menor foi o seguinte:

Figura 58 - Até que enfim!



Não deixe a quantidade de imagens lhe enganar: o algoritmo **A*** convergiu para o menor caminho após explorar 20 vértices. Eu não coloquei todos do **Dijkstra**, mas contei aqui: foi necessário explorar 35, quase o dobro para achar o caminho! Isso é uma pequena mostra de como esse algoritmo consegue aumentar a performance de execução mantendo a corretude do caminho encontrado. Mas é claro, tanto o meu nome como o dele começam com a mesma letra. 😁

E é com esse assunto que encerro o estudo do conteúdo de IA para jogos.

Fonte: TENOR. Disponível em: <https://tenor.com/search/desespero-gifs>. Acesso em: 07 jun. 2018.

É, eu também fico sentimental nessas horas...

Eu sei que alguns assuntos podem parecer muito complexos à primeira vista, mas lembre-se: para alguns dos tópicos da disciplina já existem ferramentas prontas, e você entender como elas funcionam, para quais situações são adequadas e a forma de utilizar já é metade do caminho andado!

E quem sabe alguns dos assuntos da disciplina tenham despertado seu interesse para outras áreas também. Que o conhecimento aqui sirva de plataforma para você progredir ainda mais como programador.

Até uma próxima aula! (Essa não é a última? 🤔)

Resumo

Nessa derradeira e última aula sobre descoberta de caminhos, você aprendeu um pouco sobre dois algoritmos amplamente utilizados na Ciência da Computação e na área de jogos: algoritmo de **Dijkstra** e **A***.

Inicialmente foi apresentado o conceito de **heurística** e exemplificadas três formas distintas de como o processo de decisão de escolha de caminho pode ser realizado. Um dos exemplos foi utilizado junto com uma **busca em profundidade** para ilustrar como seria a execução de um algoritmo que possui um processo decisório simples no percurso de um caminho em um **grafo**.

Em seguida, foi apresentado o algoritmo de **Dijkstra**, que permite o cálculo do menor caminho entre um vértice e todos os outros vértices do **grafo**, podendo ser adaptado para procurar o menor caminho entre dois pontos do **grafo**. Também foi observado que, apesar de garantir o menor caminho, esse algoritmo necessita explorar muitos vértices para gerar o resultado, o que limita o seu uso em cenários onde o mapa possui tamanho exagerado.

Uma otimização em cima desse algoritmo é a busca **A***, em que se une o algoritmo de **Dijkstra** com um passo de decisão **heurístico** para otimizar o processo de descoberta de caminhos. Com esse algoritmo, é possível traçar o melhor caminho entre dois pontos de um mapa com uma boa performance de execução, o que torna essa técnica uma das principais da indústria de jogos.

Referências

BELWARIER, Rachit. **A* search algorithm**. Disponível em: <https://www.geeksforgeeks.org/a-search-algorithm>. Acesso em: 04 jun. 2018.

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L. **Introduction to algorithms third edition**. 1. ed. Massachusetts: MIT Press, 2009.

FEOFILOFF, Paulo. **Algoritmo de Dijkstra**. Disponível em: <https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/dijkstra.html>. Acesso em: 28 maio 2018.

KYAW, Aung Sithu. **Unity 4. x game AI programming**. Birmingham: Packt Publishing Ltd, 2013.