PCC173/BCC463 - Otimização em Redes

Marco Antonio M. Carvalho

Departamento de Computação
Instituto de Ciências Exatas e Biológicas
Universidade Federal de Ouro Preto

Conteúdo

Aviso

Fonte

Este material é baseado no material

- Red Blob Games. *Introduction to the A^* Algorithm*. Disponível em `https://www.redblobgames.com/pathfinding/a-star/introduction.html`.
- Ami Patel. *Amit's A^* Pages*. Disponível em `http://theory.stanford.edu/~amitp/GameProgramming/`.

Licença

Algoritmo A^*

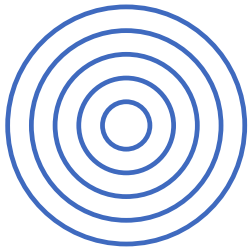
Histórico

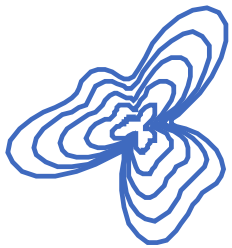
O algoritmo A^* foi proposto por Peter Hart, Nils Nilsson e Bertram Raphael do *Stanford Research Institute* em 1968, para determinar o caminho a ser navegado pelo robô Shakey, em uma sala com obstáculos.

O mesmo algoritmo pode ser utilizado para determinar o caminho mais curto entre um par específico de vértices.

Oriundo da inteligência artificial, este algoritmo é considerado uma extensão do algoritmo de Dijkstra e também é relacionado ao *Best-First Search*.

Este algoritmo de assemelha ao algoritmo de Dijkstra na medida em que favorece vértices mais próximos ao vértice inicial e também se assemelha ao *Best-First Search* na medida em que usa informações sobre os vértices mais próximos do destino.





Algoritmos

- BFS: Explora igualmente todas as direções.
- Dijkstra: Prioriza caminhos de menor custo durante a exploração.
- A^*: Prioriza caminhos que parecem levar mais rapidamente ao destino.

O Robô Shakey

Shakey em ação em outubro de 1969.

O Robô Shakey

O robô Shakey.

Algoritmo de Dijkstra

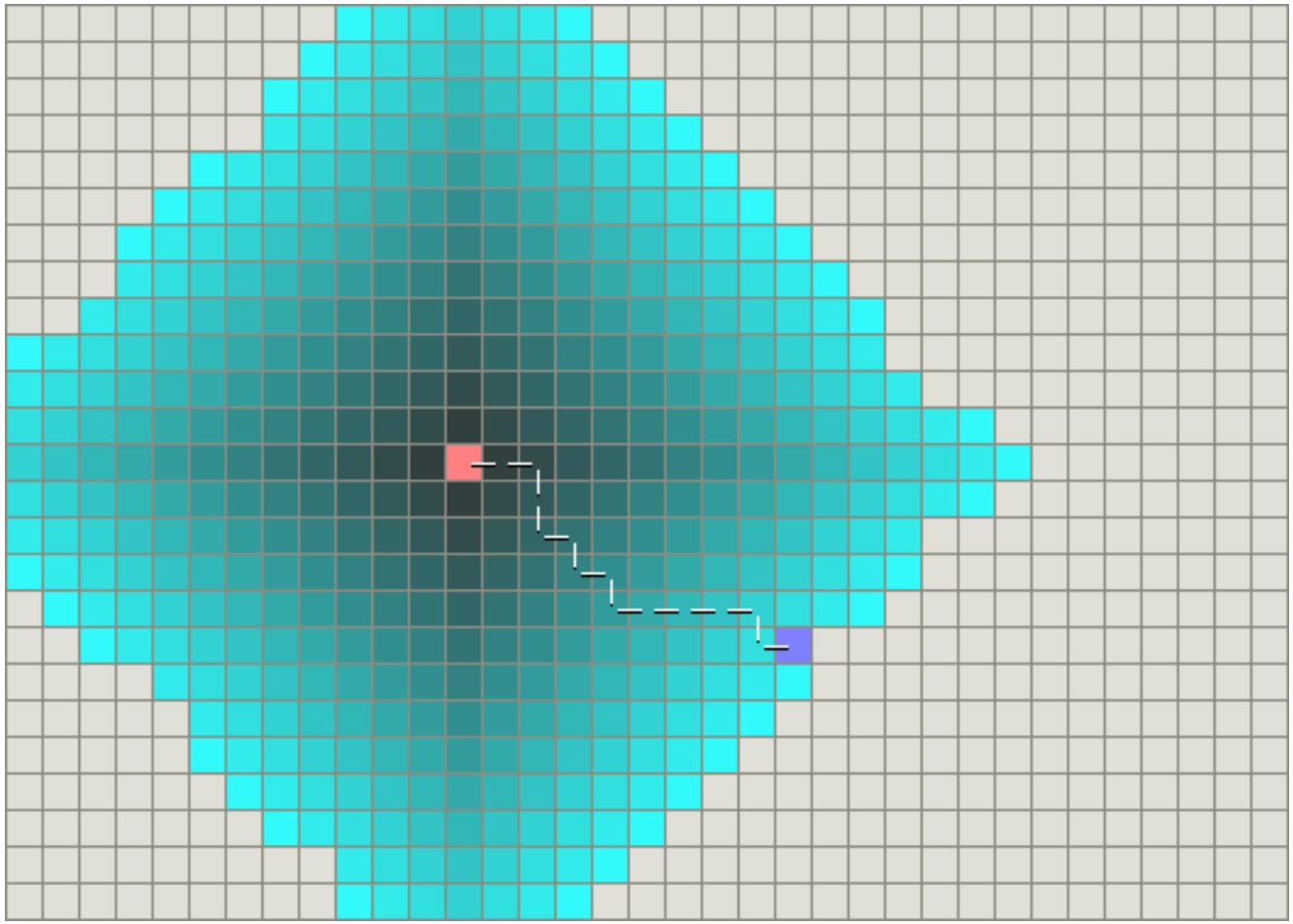
Princípio

Utiliza uma função $g(n)$ que representa o custo exato do caminho entre o vértice de origem e qualquer vértice n.

O algoritmo explora o grafo considerando sempre o vértice de menor $g(n)$.

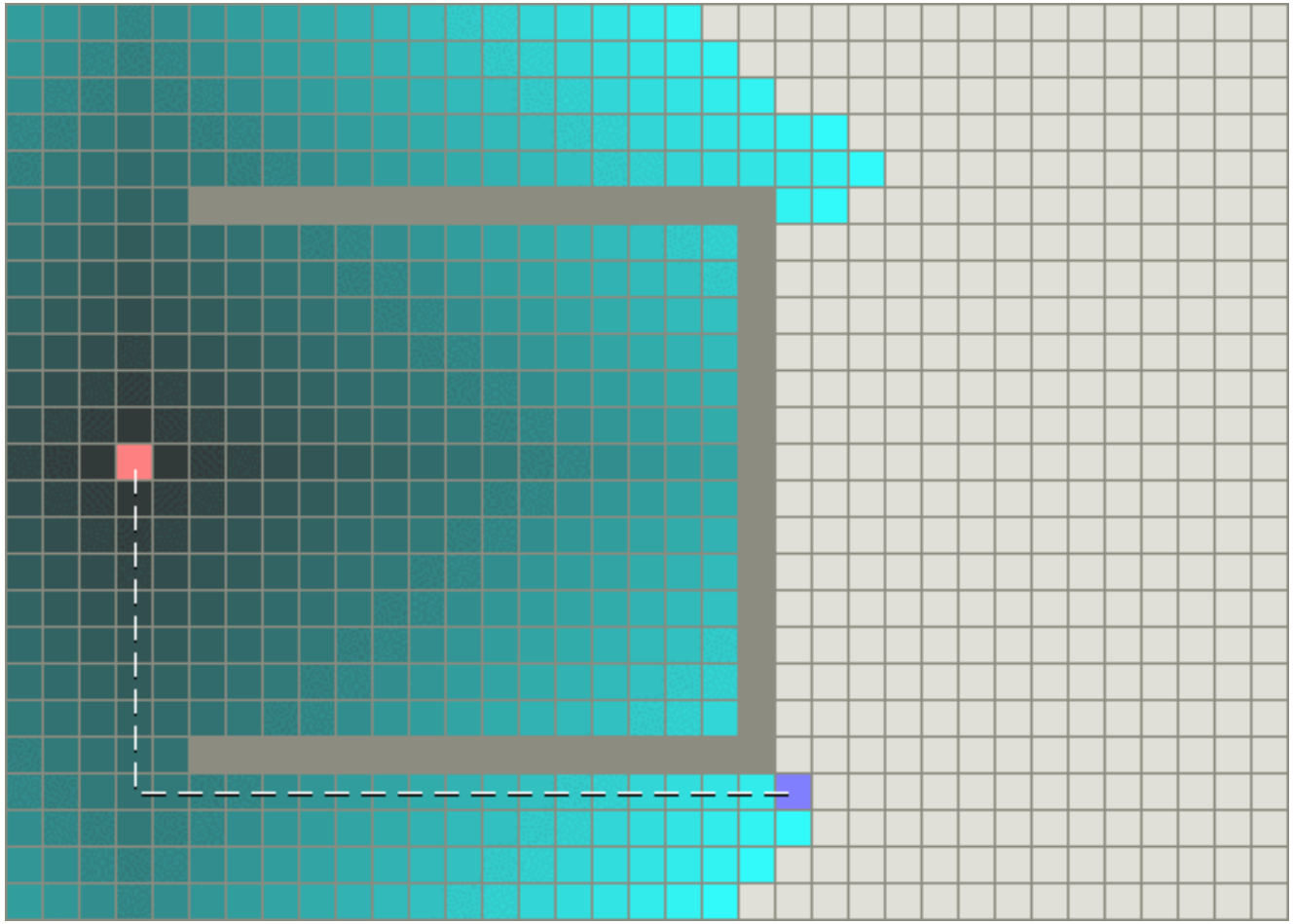
A partir do vértice inicial, expande os caminhos até encontrar o destino desejado e sempre encontra o menor caminho entre os vértices de origem e destino.

Algoritmo de Dijkstra



O vértice rosa é a origem e o vértice roxo o destino.
A área azul mostra caminhos explorados pelo algoritmo, quanto mais claros os vértices, mais longe do vértice de origem.

Algoritmo de Dijkstra

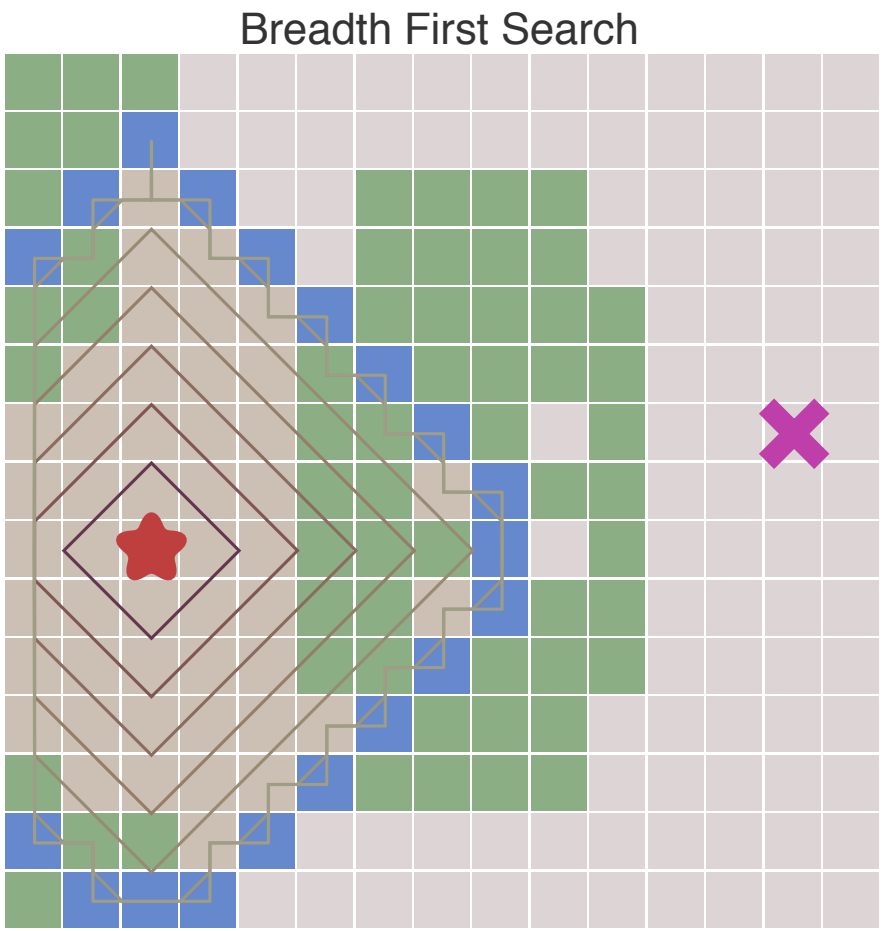


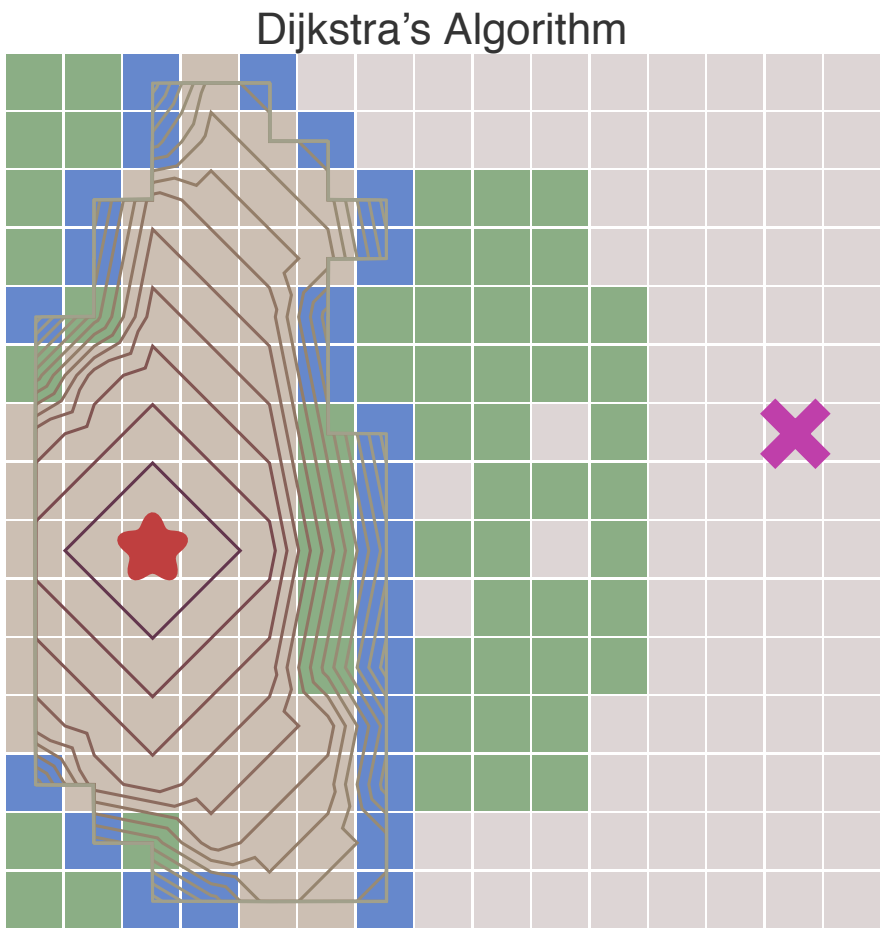
O vértice rosa é a origem e o vértice roxo o destino.
No caso com obstáculos, o algoritmo de Dijkstra é lento, mas encontra o caminho mais curto.

Algoritmo de Dijkstra

Animação

Ver animação *Frontier Expansion*.





Best-First Search

Princípio

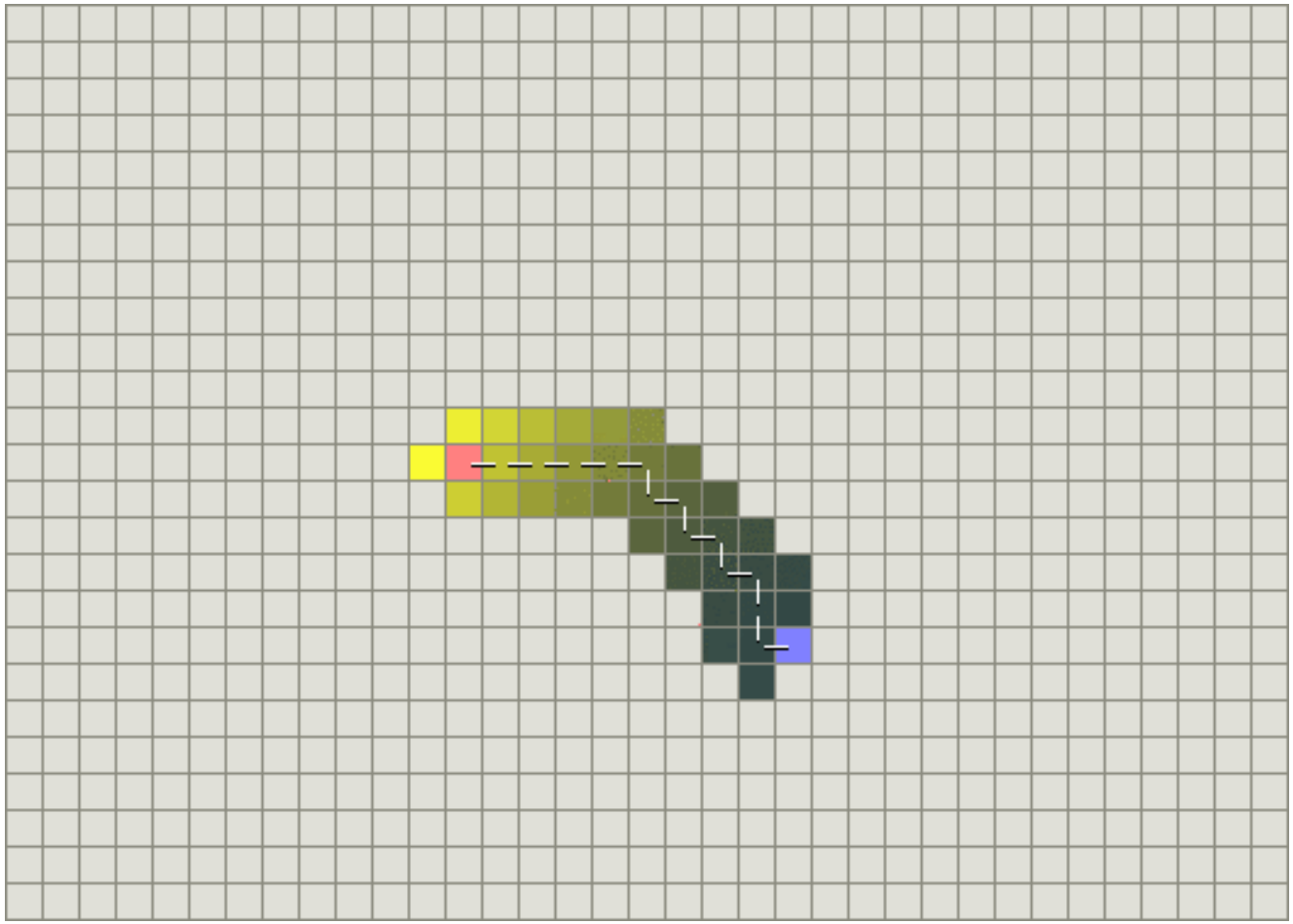
Utiliza uma função $h(n)$ que representa uma estimativa do custo do caminho entre o vértice de destino e qualquer vértice n.

O algoritmo explora o grafo considerando sempre o vértice de menor $h(n)$.

A partir do vértice inicial, expande um único caminho até encontrar o destino desejado e não necessariamente encontra o menor caminho entre os vértices de origem e destino.

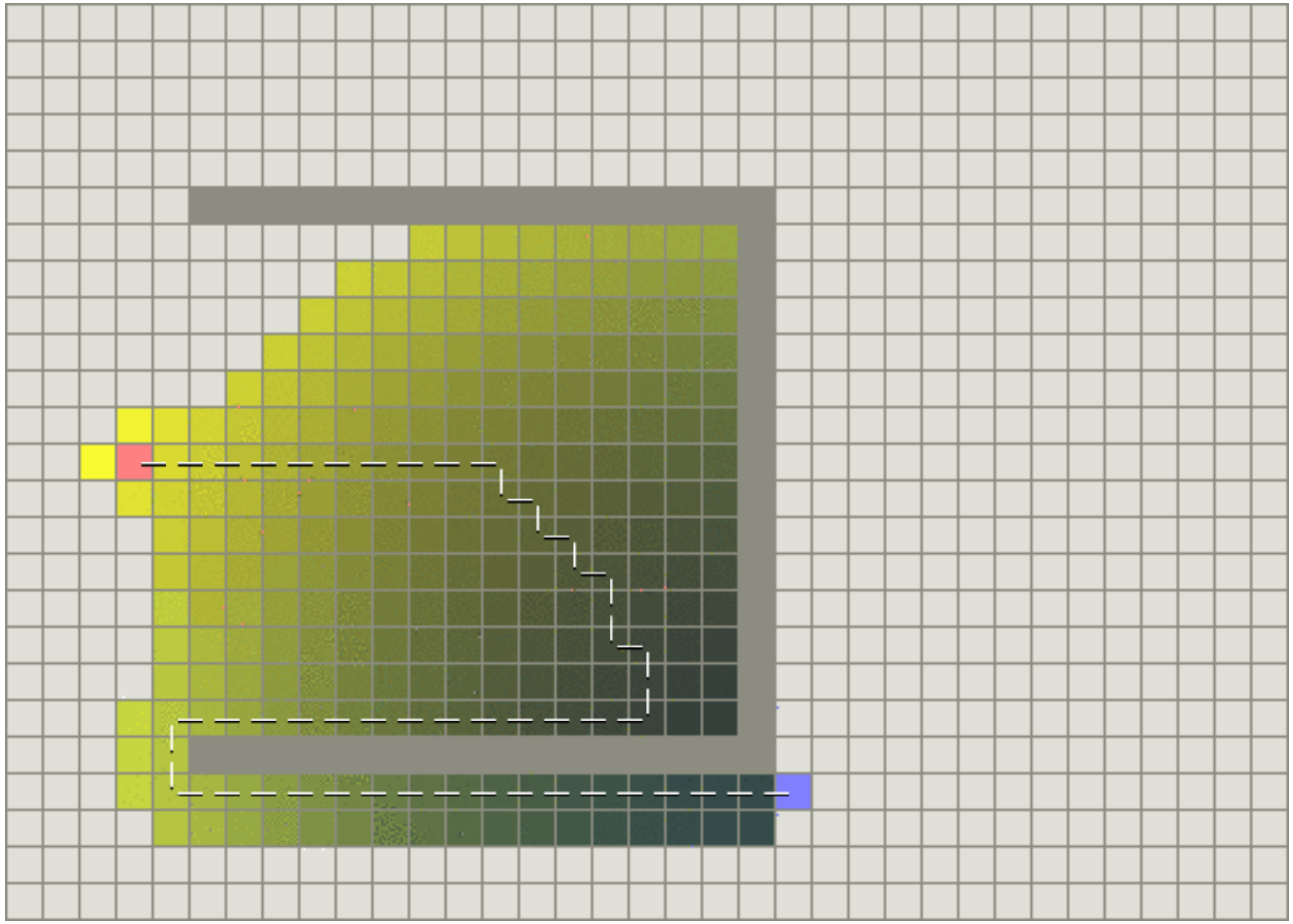
Embora seja muito mais rápido do que o algoritmo de Dijkstra, o *Best-First Search* ignora o comprimento do caminho enquanto o expande e, portanto, pode expandir um caminho mais longo.

Best-First Search



O vértice rosa é a origem e o vértice roxo o destino.
Vértices amarelos possuem alto valor de estimativa e vértices cinza possuem baixo valor de estimativa de custo para chegar ao destino.

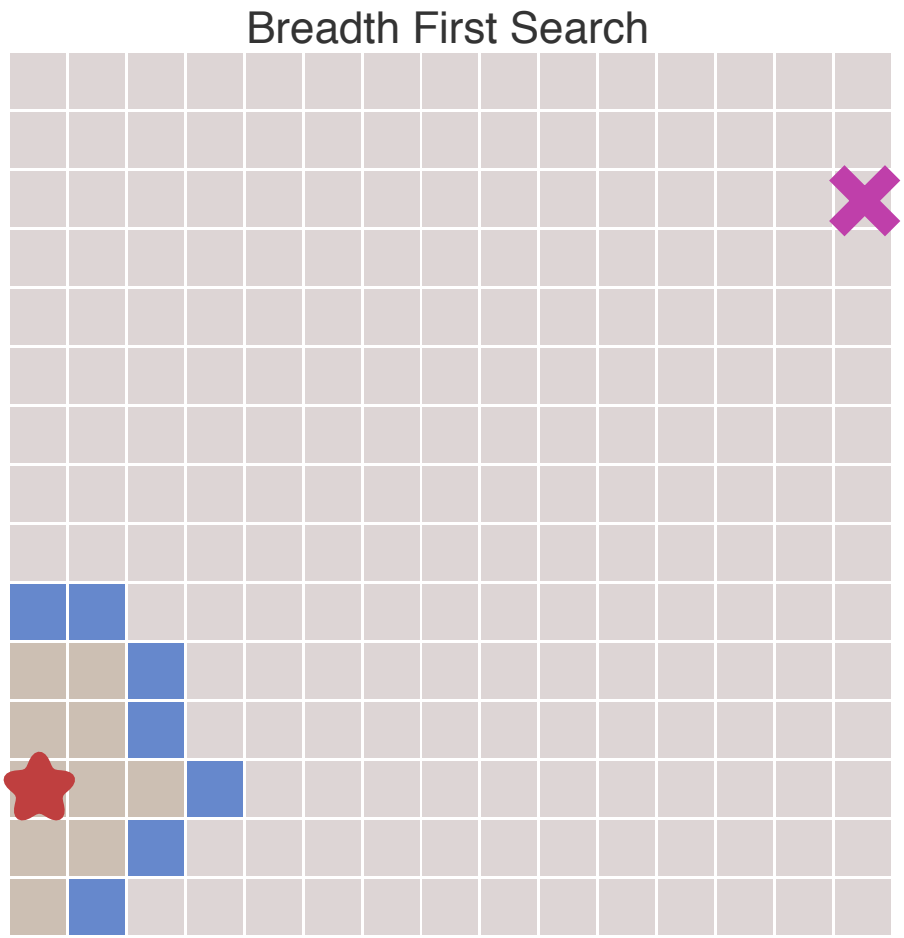
Best-First Search

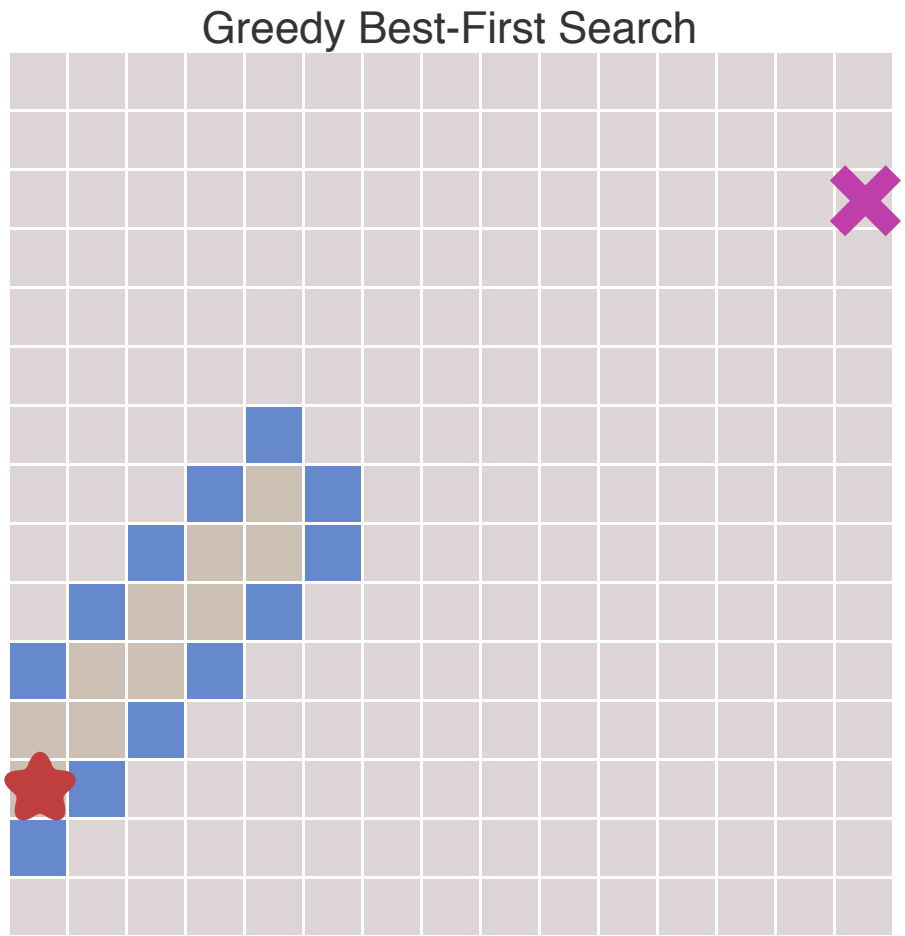


O vértice rosa é a origem e o vértice roxo o destino.
No caso com obstáculo, o *Best-First Search* é mais rápido, mas não possui corretude.

Animação

Ver animações *Greedy Best-First Search*.





Algoritmo A^*

Princípio

O algoritmo A^* utiliza uma função **conhecimento-mais-heurística** $f(n)$ para estimar o custo do caminho mais curto entre origem e destino que passa pelo vértice n.

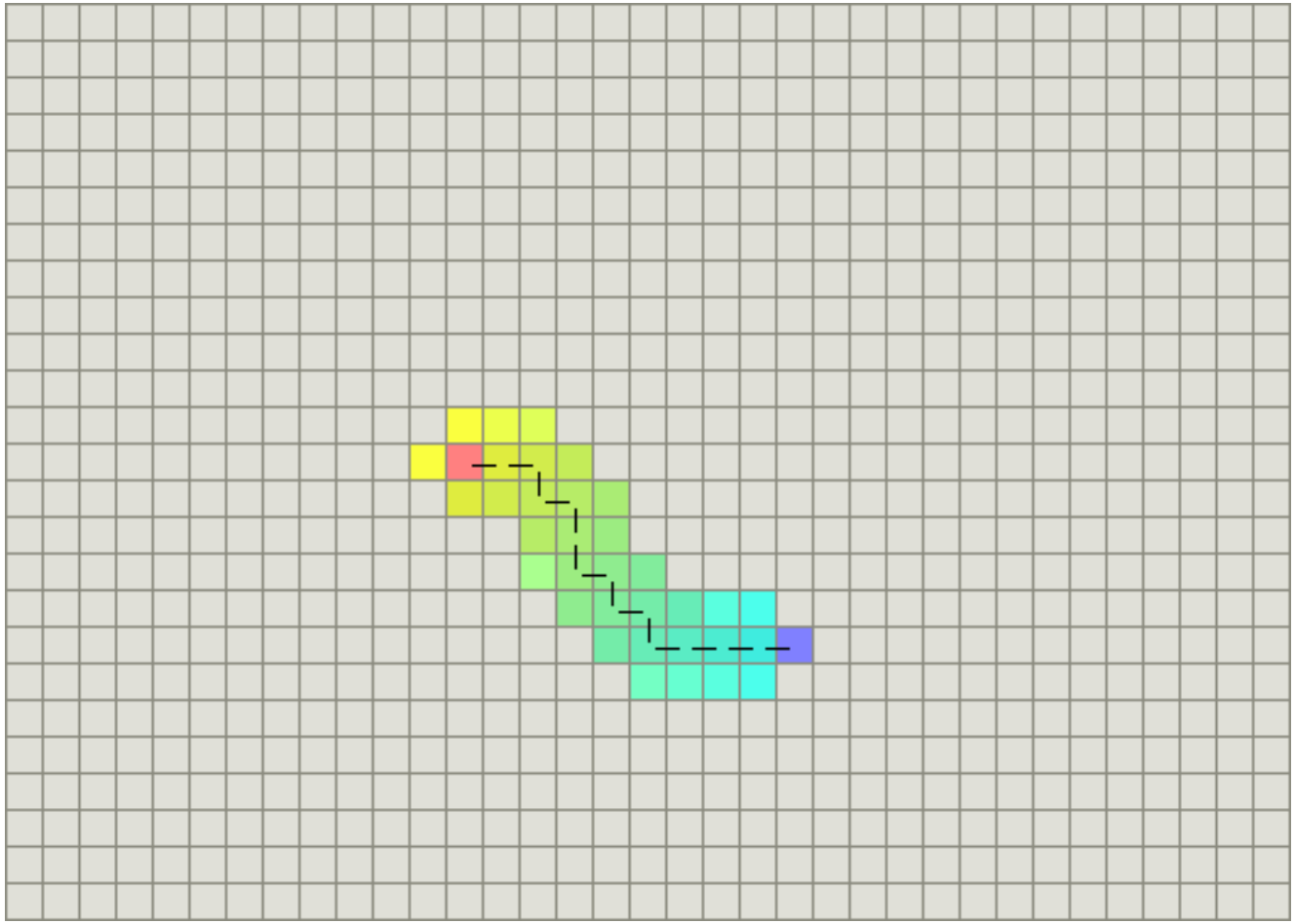
A função possui dois componentes, sendo definida da seguinte maneira:

$$f(n) = g(n) + h(n)$$

Em que $g(n)$ representa o custo exato do caminho entre o vértice inicial e qualquer vértice n e $h(n)$ representa uma estimativa heurística do custo do caminho entre o vértice n e o vértice de destino.

O algoritmo explora o grafo considerando sempre o vértice de menor $f(n)$.

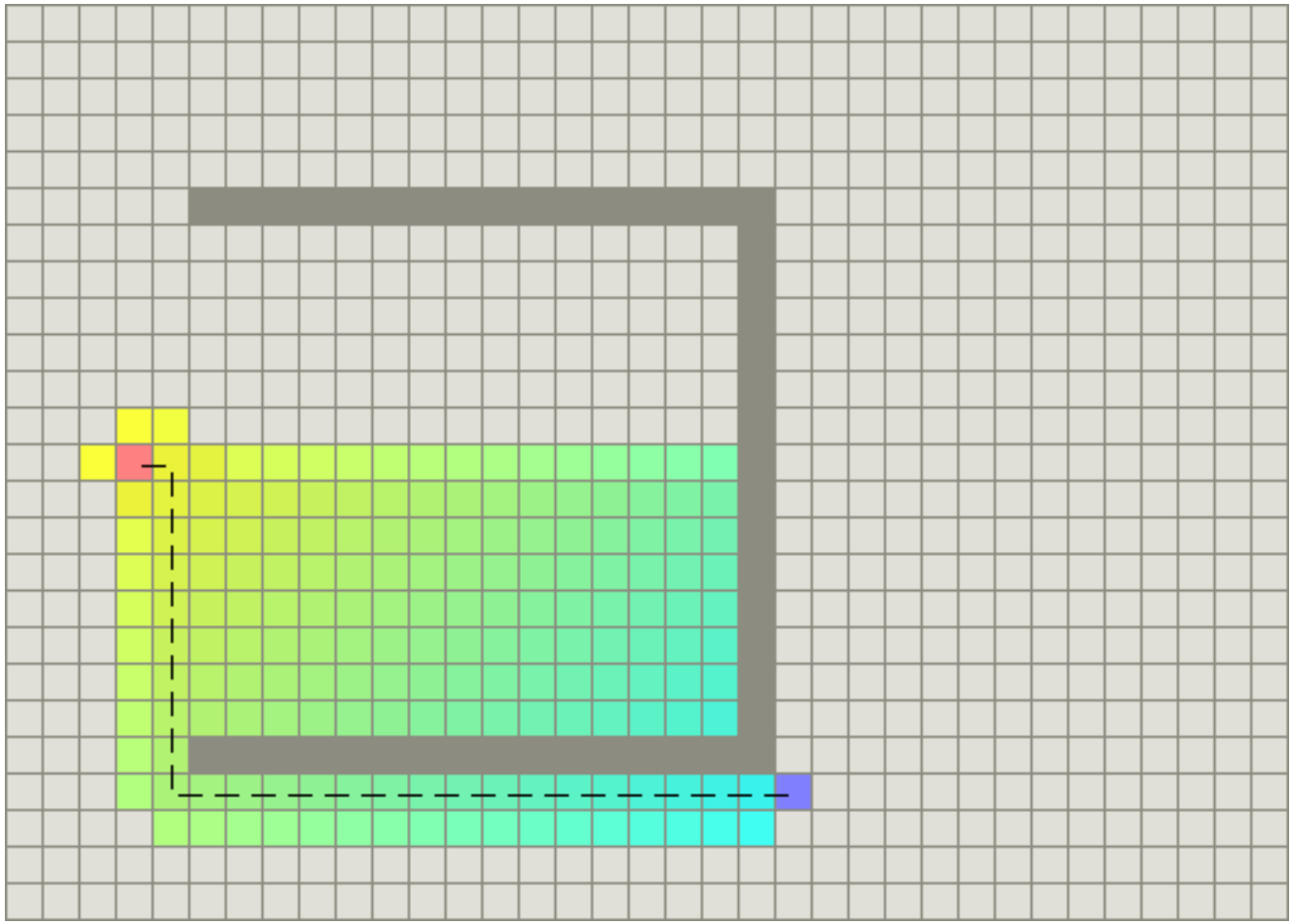
Algoritmo A^*



O vértice rosa é a origem e o vértice roxo o destino.

No caso simples, o A^* é tão rápido quanto o *Best-First Search*.

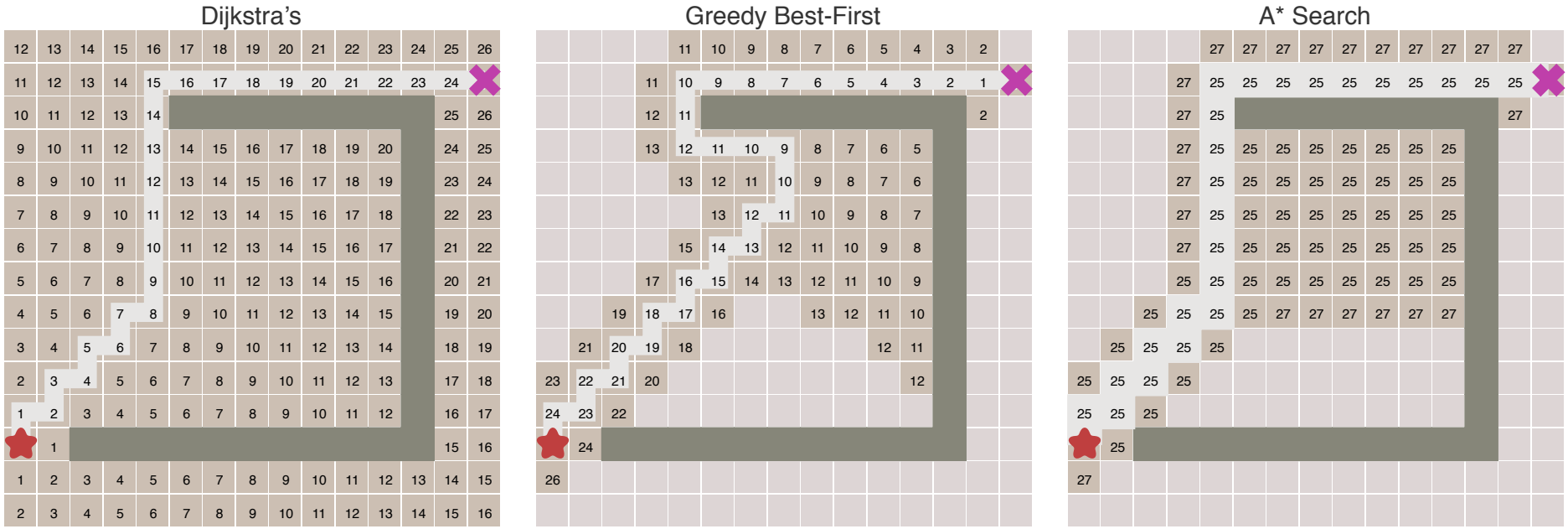
Algoritmo A^*



O vértice rosa é a origem e o vértice roxo o destino.
No caso com obstáculos, o A^* determina um caminho tão bom quanto o algoritmo de Dijkstra.

Animação

Ver animações A^* *Search*.



Considerações sobre a Estimativa Heurística

Algumas considerações sobre $h(n)$:

- Em um caso extremo, se $h(n) = 0$, então apenas $g(n)$ guiará o algoritmo, tornando-o equivalente ao algoritmo de Dijkstra;
- Se $h(n)$ é sempre menor ou igual ao custo do caminho mais curto entre n e o destino, então o algoritmo A^* garantidamente encontrará o caminho mais curto – denominamos $h(n)$ **admissível**;
- Quanto menor o valor de $h(n)$ (para todo n), mais vértices serão expandidos pelo A^*, tornando-o mais lento.

Considerações sobre a Estimativa Heurística (continuado)

- Se $h(n)$ é exatamente igual ao custo do caminho mais curto entre n e o destino, então o A^* expandirá somente o caminho mais curto, se tornando muito rápido;
- Se $h(n)$ em algum momento for maior que o custo do caminho mais curto entre n e o destino, então não há garantia de que o caminho mais curto será encontrado. Porém, será rápido;
- Em outro caso extremo, se $h(n)$ é muito maior do que $g(n)$, então apenas $h(n)$ guiará o algoritmo, tornando-o equivalente ao *Best-First Search*.

Algoritmo A^*

Terminologia

- F: conjunto dos vértices fechados (já examinados);
- A: conjunto dos vértices abertos (ainda não examinados);
- $f(v)$: função de custo do vértice v;
- $g(v)$: distância da origem até o vértice v;
- $g'(v)$: distância da origem até o vértice v, usando um vértice intermediário;
- $h(v)$: estimativa heurística do custo do caminho do vértice v até o vértice de destino;
- $N(v)$: conjunto de vértices adjacentes ao vértice v;
- d_{ij}: distância entre os vértices i e j, de acordo com a matriz D;
- rot: vetor utilizado para reconstrução do caminho determinado pelo algoritmo.

Algoritmo A^*

Entrada: Grafo $G = (V, E)$, vértices de origem e destino o, d, matriz de distâncias D

```
1  F ← ∅;
2  A ← A ∪ {o};
3  Crie um vetor vazio rot;
4  g(o) ← 0;
5  f(o) ← g(o) + h(o);
6  enquanto A ≠ ∅ faça
7  |   atual ← i|f(i) < f(j) ∀j ∈ A;
8  |   se atual = d então  retorna rot;
9  |   A ← A \ {atual};
10 |   F ← F ∪ {atual};
11 |   para cada vértice v ∈ N(atual) faça
12 |   |   se v ∈ F então continue;
13 |   |   g'(v) ← g(atual) + d_{atual,v};
14 |   |   se v ∉ A ou g'(v) < g(v) então
15 |   |   |   rot[v] ← atual;
16 |   |   |   g(v) ← g'(v);
17 |   |   |   f(v) ← g(v) + h(v);
18 |   |   |   se v ∉ A então A ← A ∪ {v};
19 |   |   fim
20 |   fim
21 fim
22 retorna ERRO;
```

Algoritmo A^*

Complexidade

A complexidade de tempo do A^* depende diretamente da função $h(v)$ usada.

No pior caso, a quantidade de vértices explorados é exponencial no tamanho do menor caminho.

Porém, o algoritmo possui complexidade de tempo polinomial se $|h(v) - h^*(v)| \leq O(log\ h^*(v))$, em que $h^*(v)$ é a heurística perfeita.

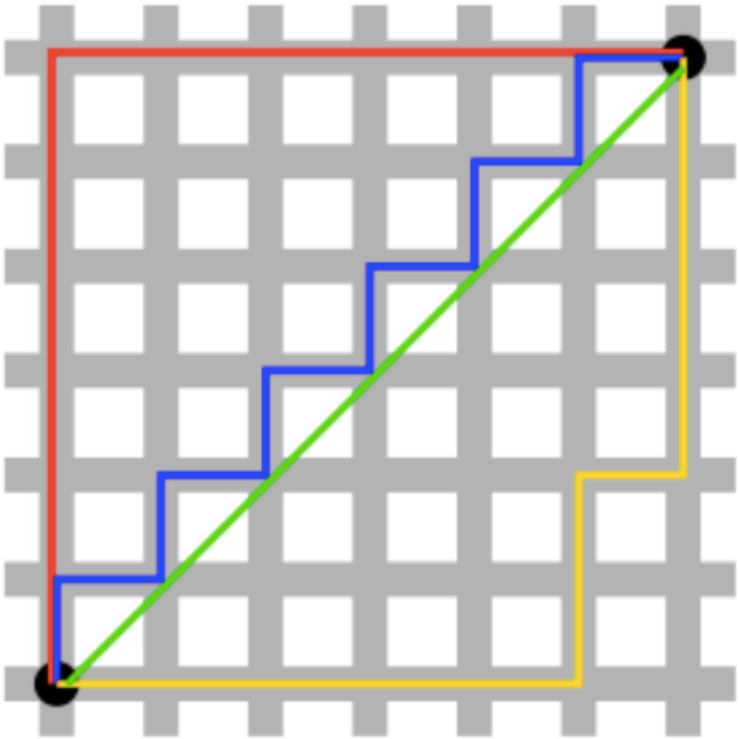
Em outras palavras, o algoritmo possui complexidade de tempo polinomial se o erro de $h(v)$ não crescer mais rapidamente do que o logaritmo da heurística perfeita.

Mapas em Grade

Para mapas em grade, existem estimativas heurísticas adotadas amplamente:

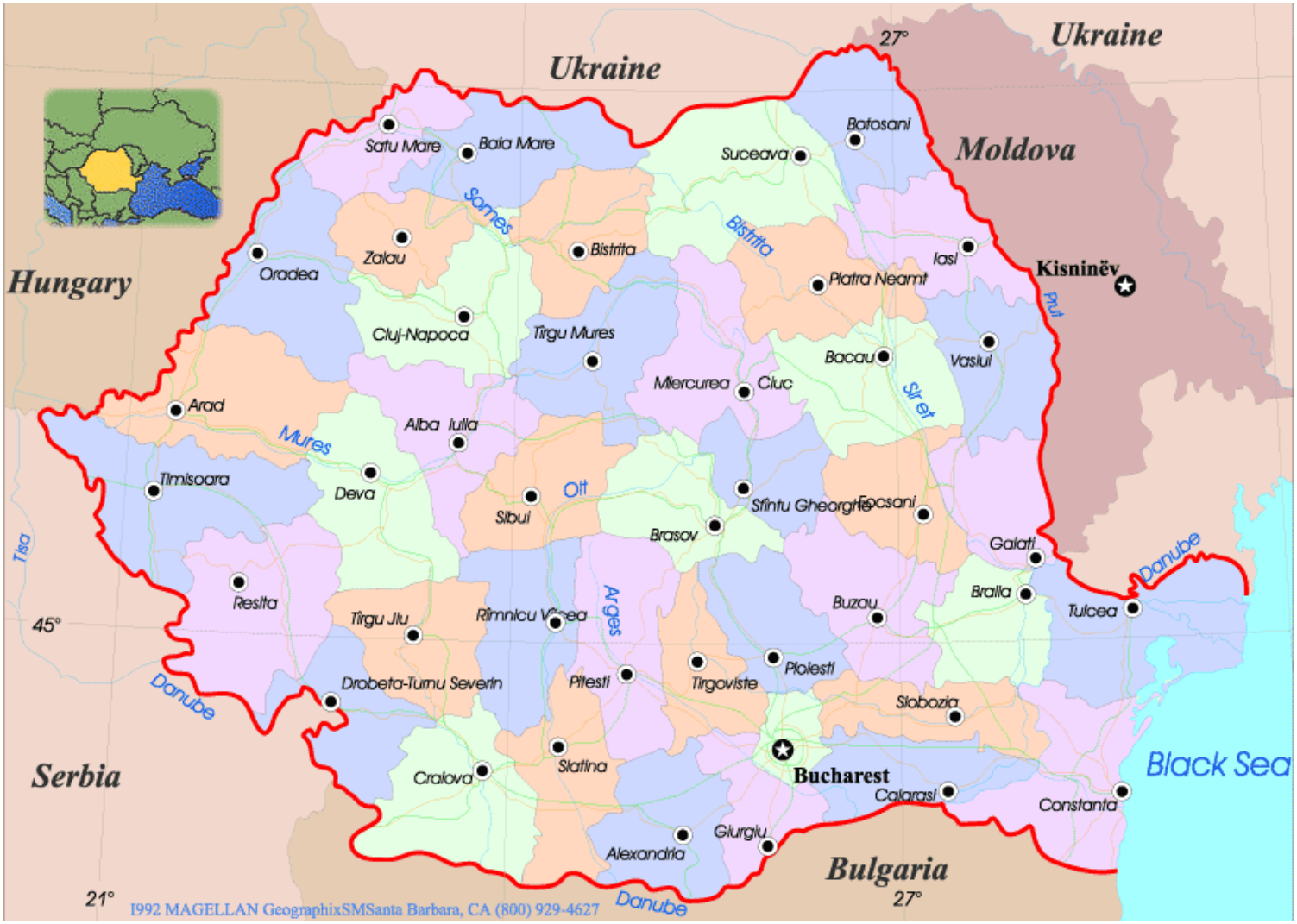
- Em grades quadradas, que permitem movimentos em 4 direções, usamos a distância Manhattan;
- Em grades quadradas, que permitem movimentos em 8 direções, usamos a distância *Chebyshev* (ou distância Diagonal);
- Em grades quadradas, que permitem movimentos em todas as direções, usamos a distância Euclidiana (que permitem movimentos fora da grade);
- Em grades hexagonais, que permitem movimentos em 6 direções, usamos a distância Manhattan adaptada para grades hexagonais.

Manhattan vs. Euclidiana



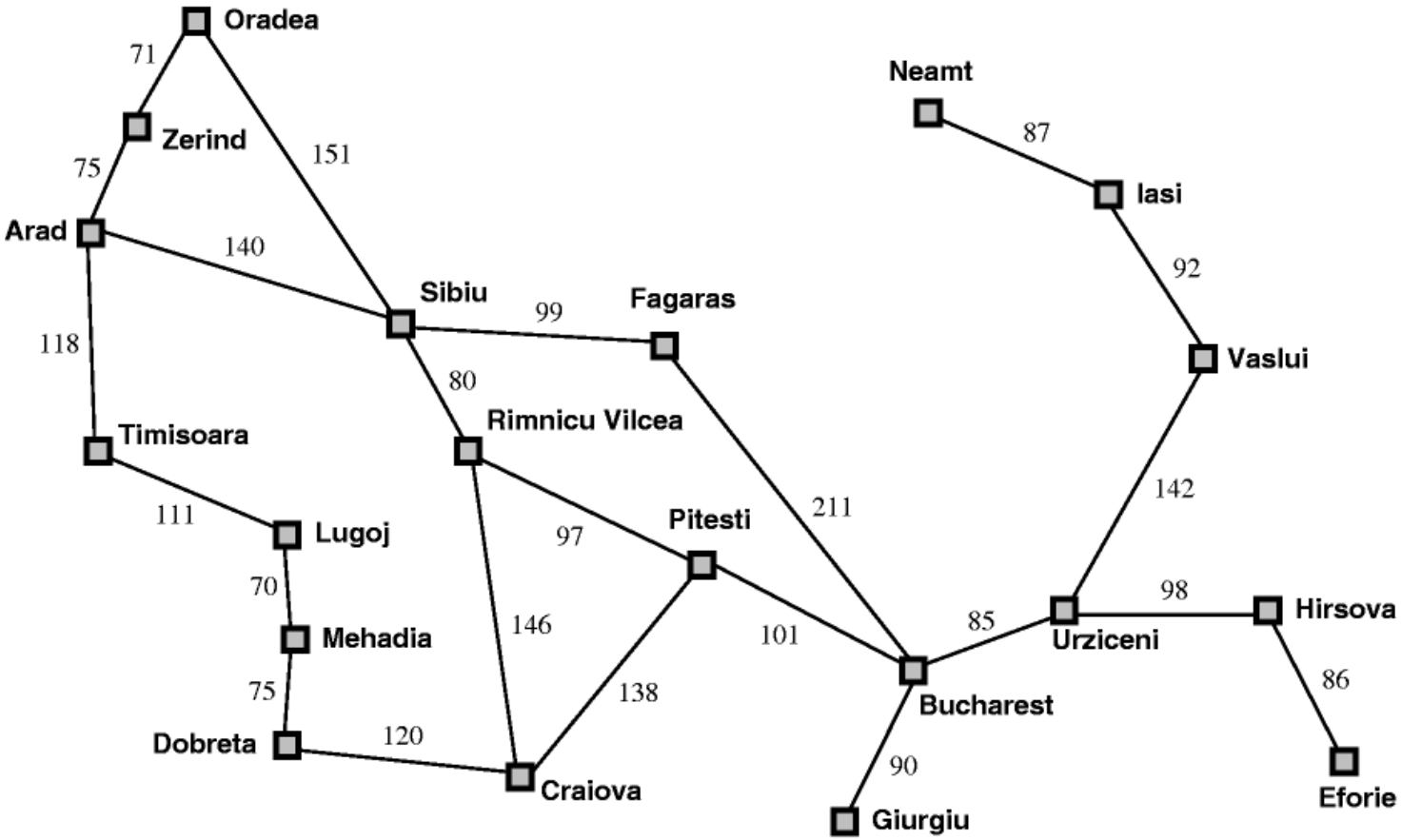
Distância Manhattan (vermelho, azul e amarelo) e distância Euclidiana (verde).

Exemplo



Mapa das cidades da Romênia, um exemplo clássico para demonstração do algoritmo A^*.

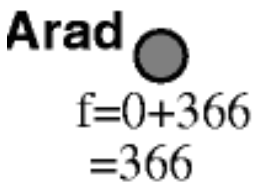
Exemplo



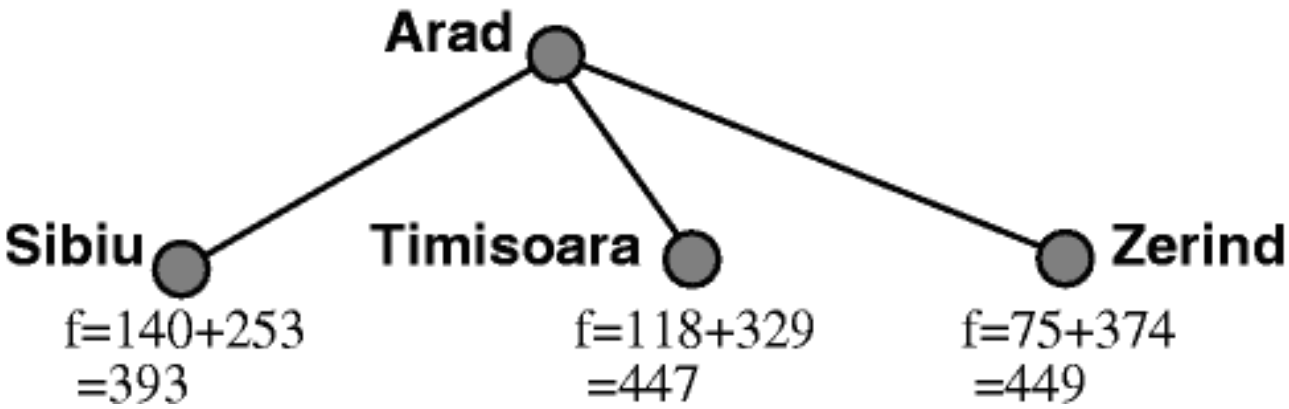
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Vamos executar o algoritmo A^* para determinar o caminho mais curto entre as cidades de Arad e Bucharest, utilizando como heurística a distância em linha reta, que nunca superestima a distância real.

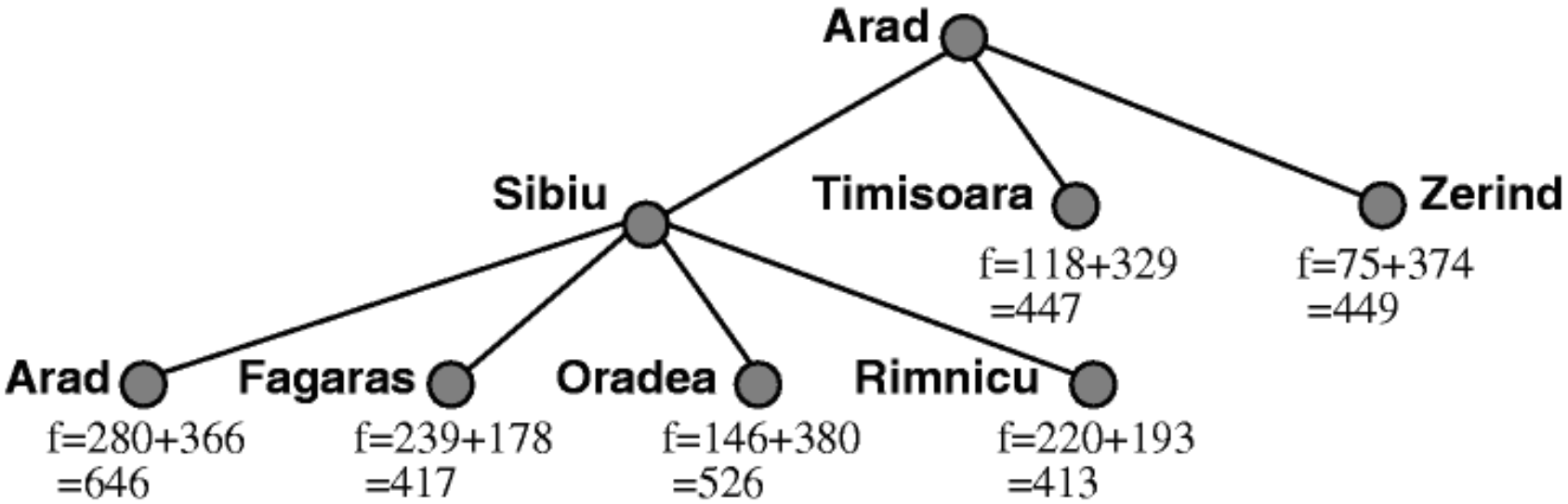
Exemplo



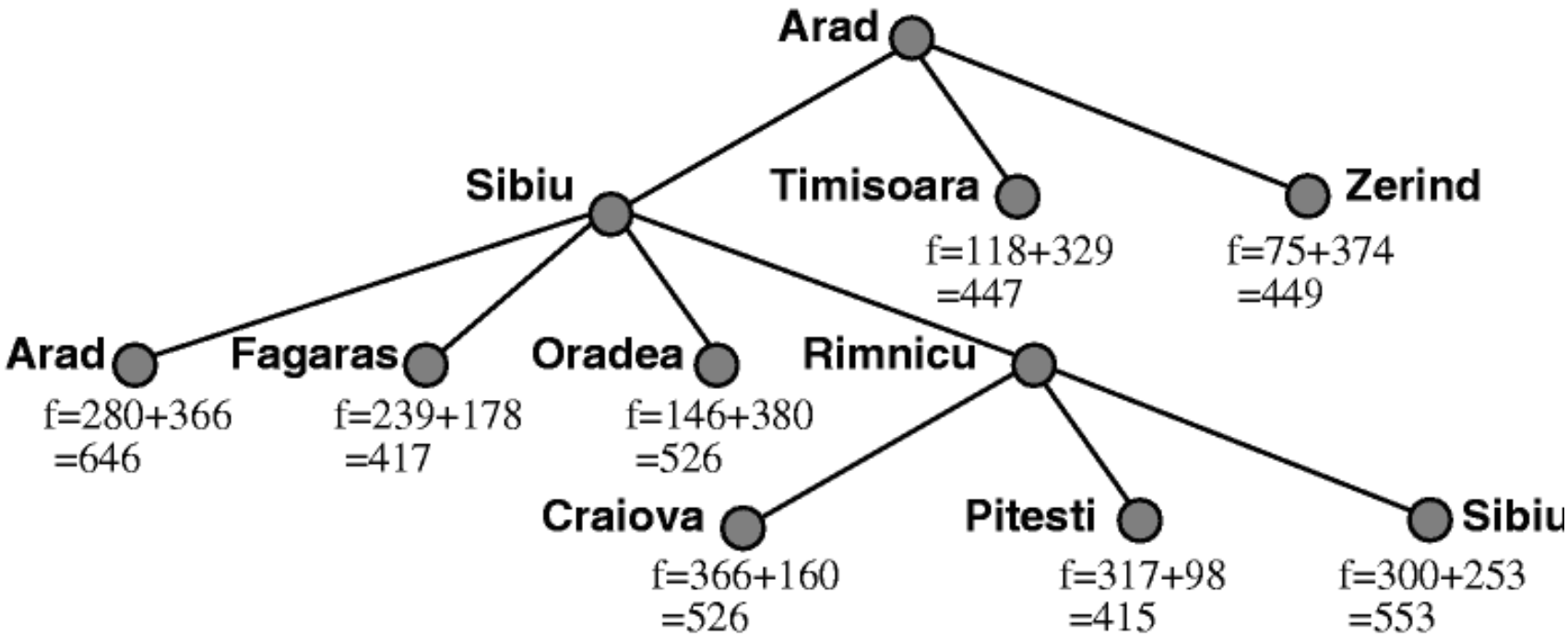
Exemplo


Arad
Sibiu
Timisoara
Zerind
f=140+253
=393
f=118+329
=447
f=75+374
=449

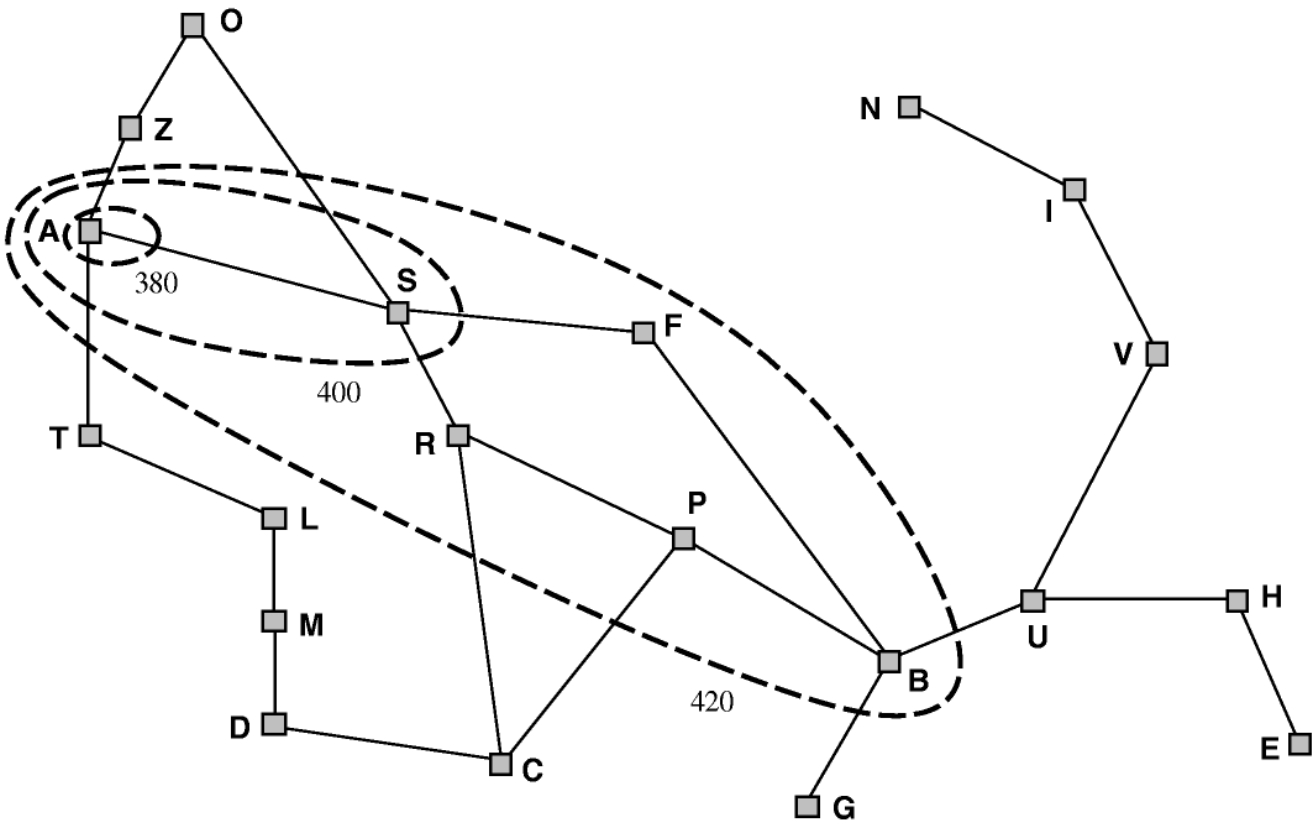
Exemplo


Arad
Sibiu
Timisoara
Zerind
f=118+329
=447
f=75+374
=449
Arad
Fagaras
Oradea
Rimnicu
f=280+366
=646
f=239+178
=417
f=146+380
=526
f=220+193
=413

Exemplo


Arad
Sibiu
Timisoara
Zerind
f=118+329
=447
f=75+374
=449
Arad
Fagaras
Oradea
Rimnicu
f=280+366
=646
f=239+178
=417
f=146+380
=526
Craiova
Pitesti
Sibiu
f=366+160
=526
f=317+98
=415
f=300+253
=553

Exemplo



A função $f(v)$ deve ser monotonicamente crescente ao longo do caminho. Vértices internos à área contornada possuem $f(v)$ menor ou igual aos vértices externos. Nos contornos, $f = 380$, $f = 400$ e $f = 420$.

Dúvidas?