■ Lecture 2-1

Search-based path finding

高飞

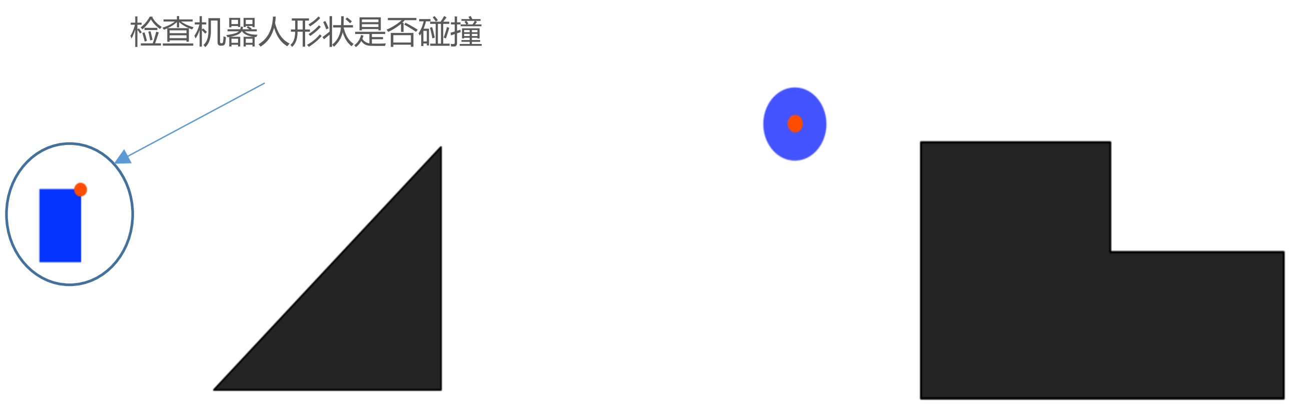
浙江大学 控制学院

构形空间

- 机器人构形: 机器人所有的位置点
- 机器人自由度(DOF): 用来表示机器人构形的最小实数坐标数
- 机器人构形空间: 包含所有可能的机器人构形的n维空间，记作C-space
- 每个机器人姿态都是C-Space中的一个点

构形空间障碍物

- 在工作空间中规划
  - 机器人具有不同的形状和大小
  - 碰撞检测需要知道机器人的几何信息——费时，难度大

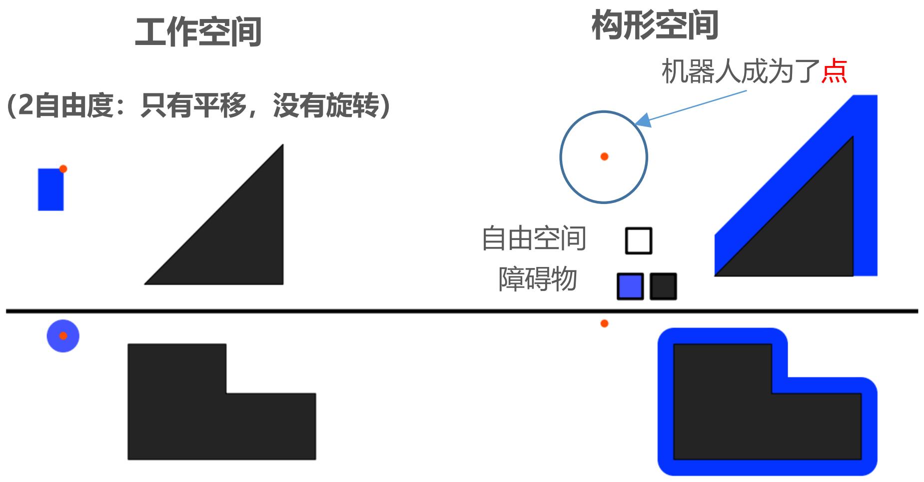


(1) 矩形移动机器人

(2) 圆形移动机器人

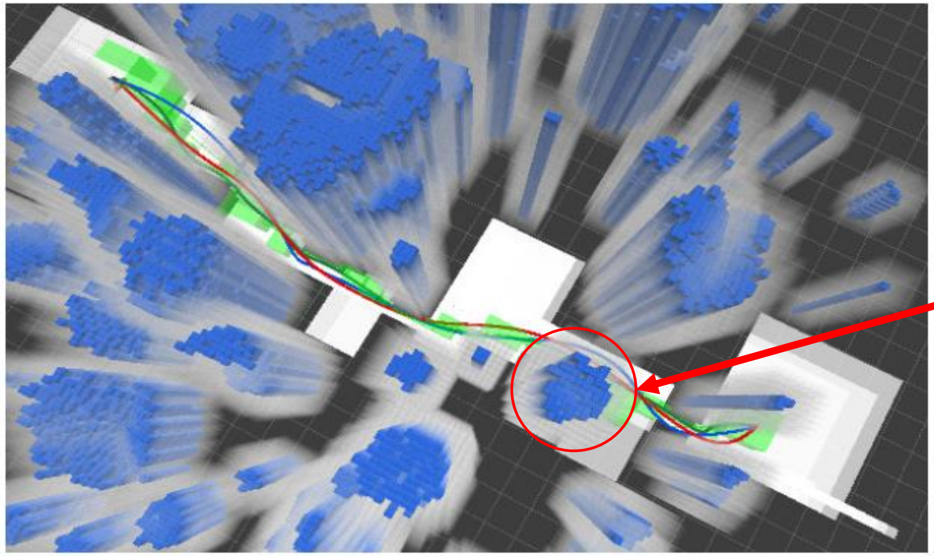
构形空间障碍物

- 在构形空间中规划
  - 机器人表示为C-space中的一个点，如位置（$\mathbb{R}^3$中的点），姿态（$SO(3)$中的点）
  - 障碍物需要在构形空间中表示（先于运动规划完成），称为构形空间障碍，或C-obstacle
  - C-space = （C-obstacle） ∪ （C-free）
  - 路径规划就是在C-free中寻找起点q_{start}至终点q_{goal}的路径



工作空间和构形空间障碍物

- 工作空间
  - 机器人有形状和大小（不利于运动规划）
- 构形空间： C-space
  - 机器人是一个点（便于运动规划）
  - 运动规划前，障碍物先在C-space中表示
- 在C-space中表示障碍物非常复杂。在实际中使用近似（但是更保守）的表示方法



如果我们保守地把机器人建模成一个半径为δ_r的球体，构建C-space可以把所有障碍物向各个方向膨胀δ_r。

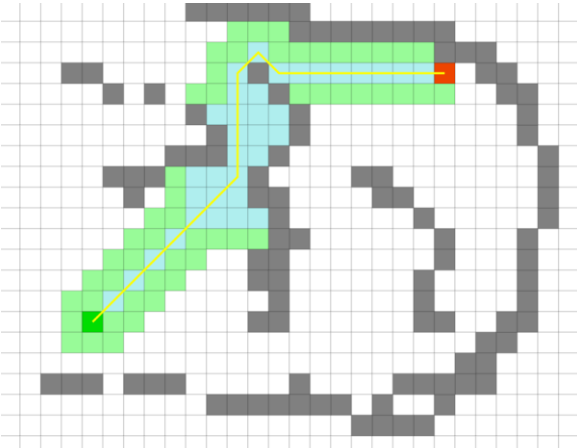
Path Finding

- **基于搜索的方法**
  - General graph search: DFS, BFS
  - Dijkstra and A* search
  - **Jump point search**
- **基于采样的方法**

  Probabilistic roadmap (PRM)

  Rapidly exploring random tree (RRT)

  RRT*, informed RRT*



A* search example

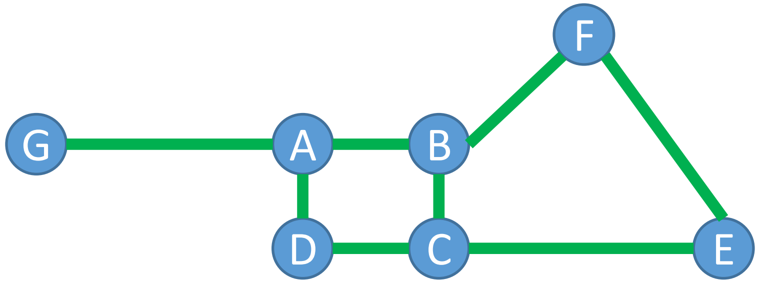
RRT example

图的定义

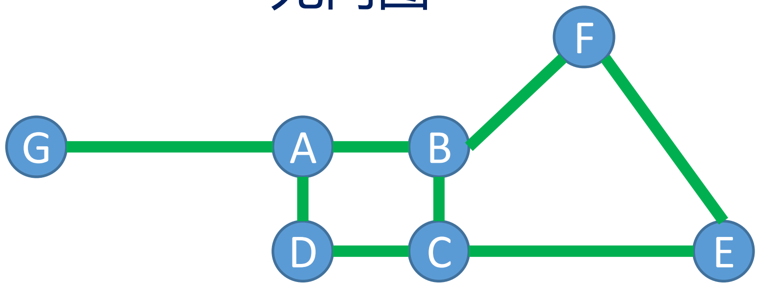
图（Graphs）

图由节点和边构成

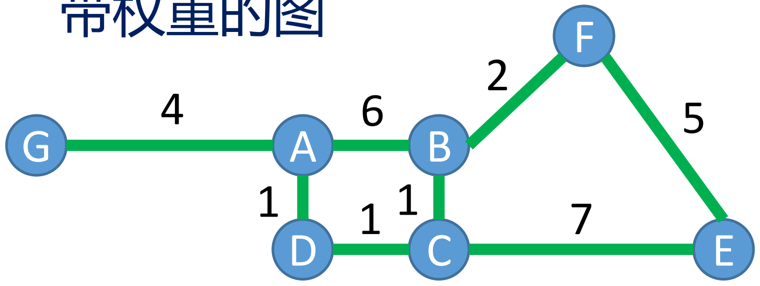
不带权重的图



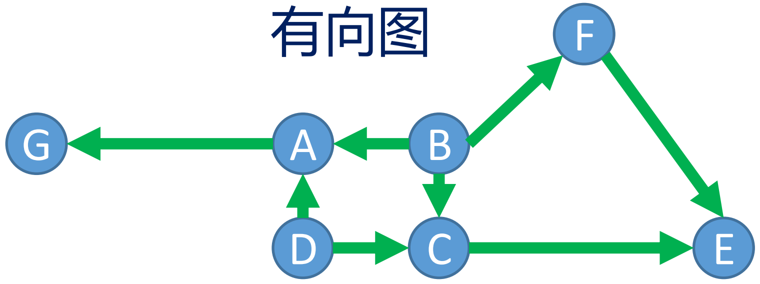
无向图



带权重的图

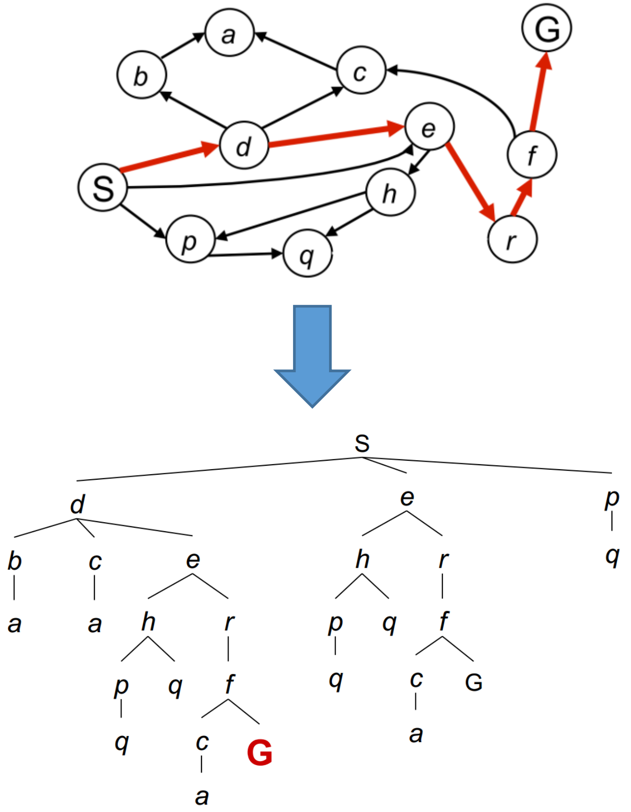


有向图



图搜索算法概述

- 搜索过程开始于初始点S
  - 搜索过程会生成一个搜索树
  - 对树中节点逆向搜索可以得到一条路径
  - 很多情况下，构建图的整棵搜索树是不明智的（工作量大且低效）——尽快到达目标点才是我们的目的



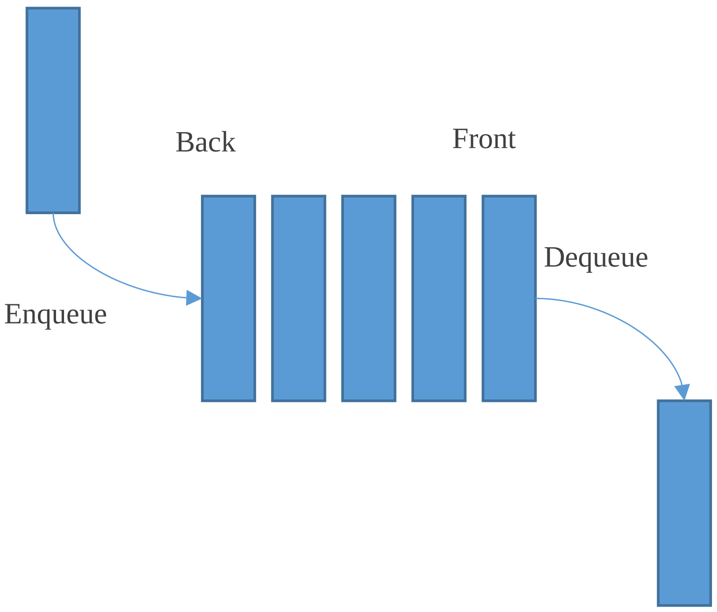
- 维护一个容器来存储所有待访问的节点
- 容器初始化时只存在初始状态Xs
- 开始循环
  - 删除Remove：根据指定的规则从容器中移出一个节点
  - 扩展Expansion：得到该节点的所有邻节点
  - 存入Push：将这些邻节点存入容器
- 结束循环

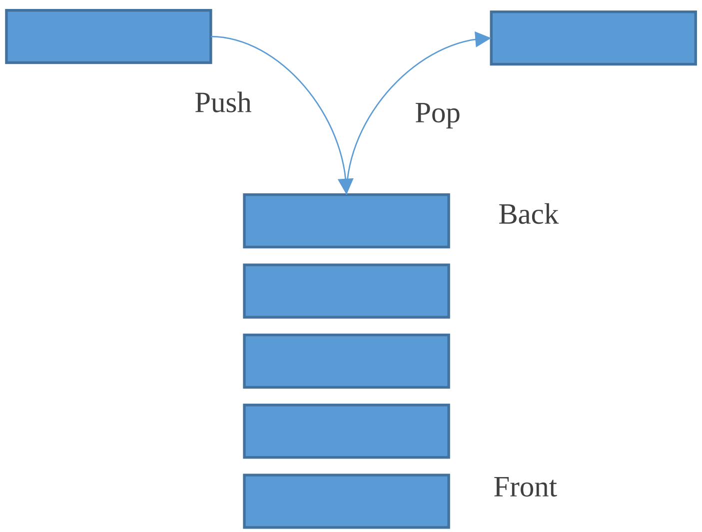
- Question 1:如何退出循环?
  - 例:容器空掉时退出循环
- Question 2:如果图中存在回环?
  - 禁止从容器中移出的节点再次进入容器
- Question 3:如何制定取出节点的规则使目标尽快被达到，且尽可能少的扩展结点?

图的遍历

广度优先搜索 vs. 深度优先搜索

- Breadth First Search (BFS) vs. Depth First Search (DFS)





This is a queue

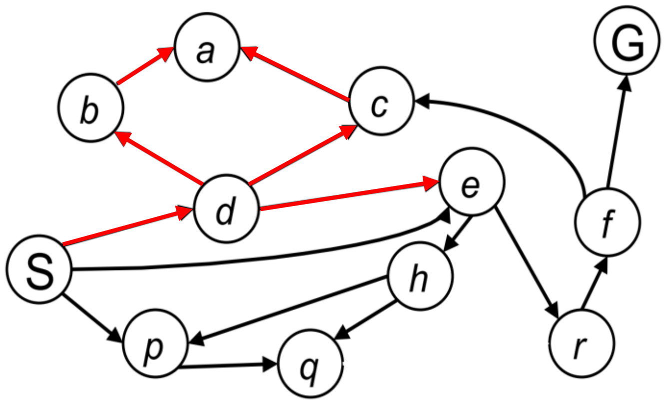
BFS： 队列→"先进先出"

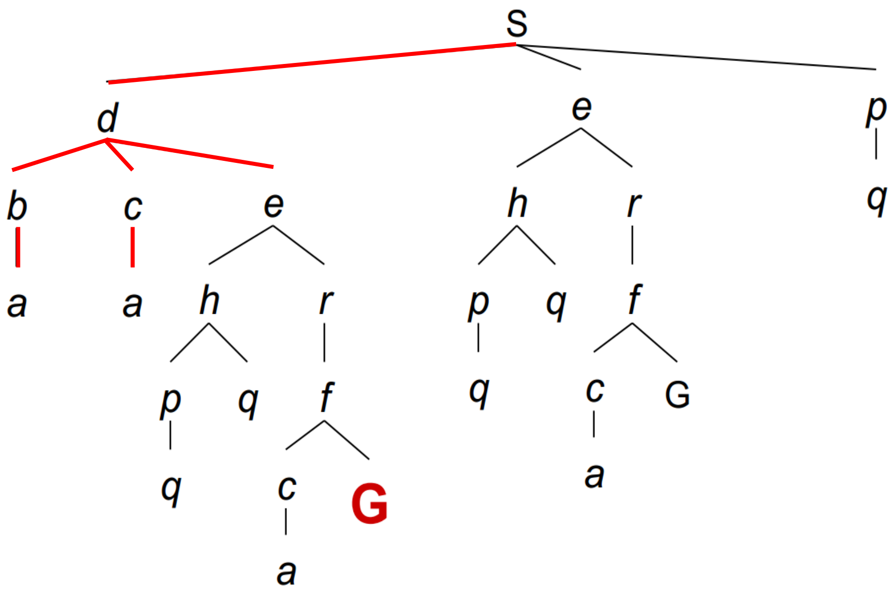
This is a stack

DFS： 堆栈→"后进先出"

深度优先搜索（DFS）

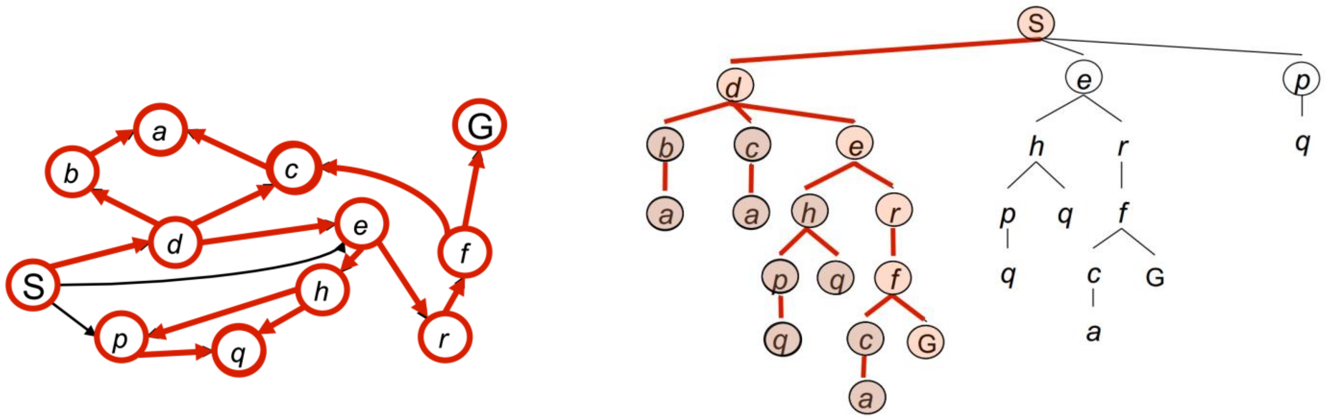
- 策略：移除/扩展搜索树中层级**最深的节点**

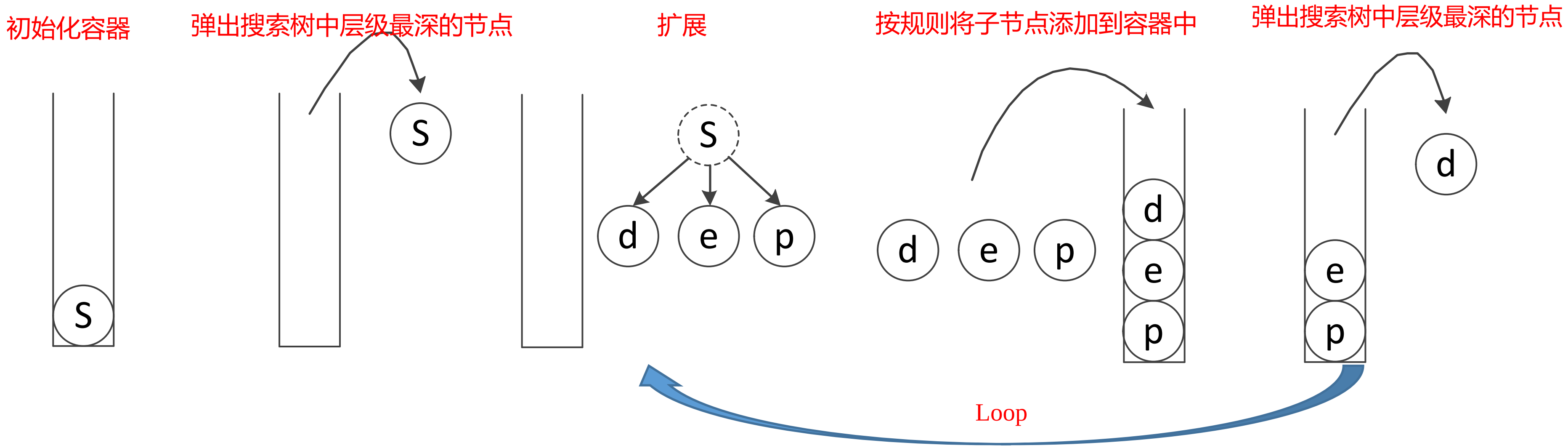




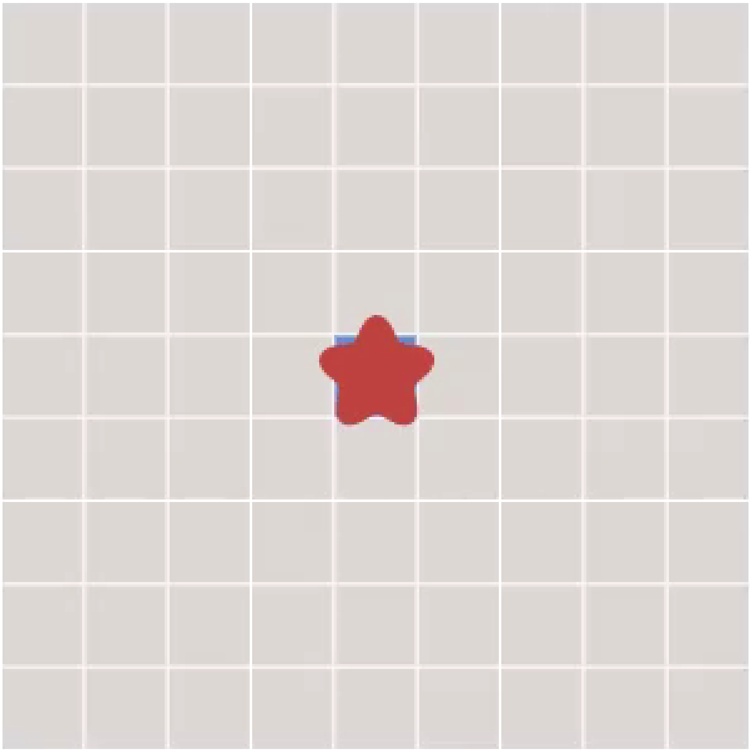
深度优先搜索（DFS）

- 实现：维护后进先出（LIFO）容器（即堆栈）





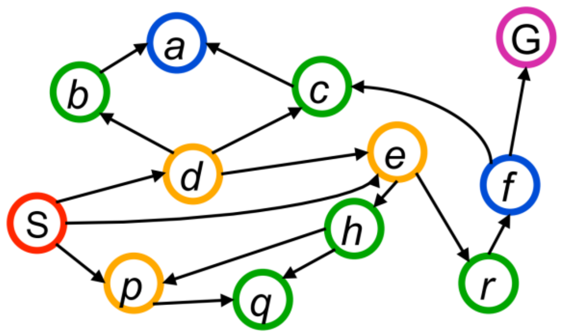
深度优先搜索（DFS）

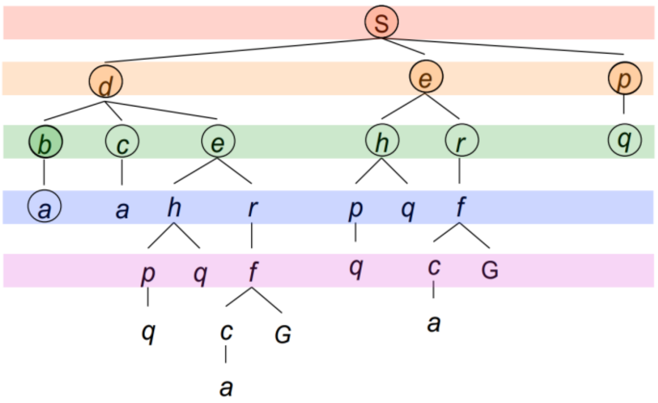


Courtesy: Amit Patel's Introduction to A*, Stanford

广度优先搜索（BFS）

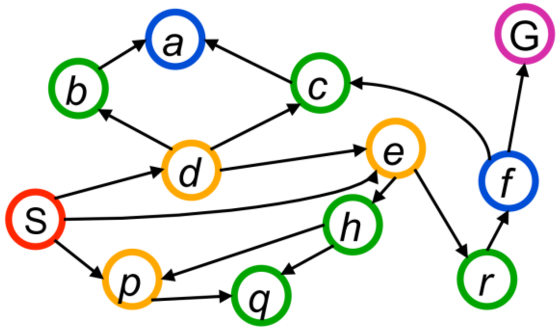
- 策略：移除/扩展搜索树中层级**最浅的节点**

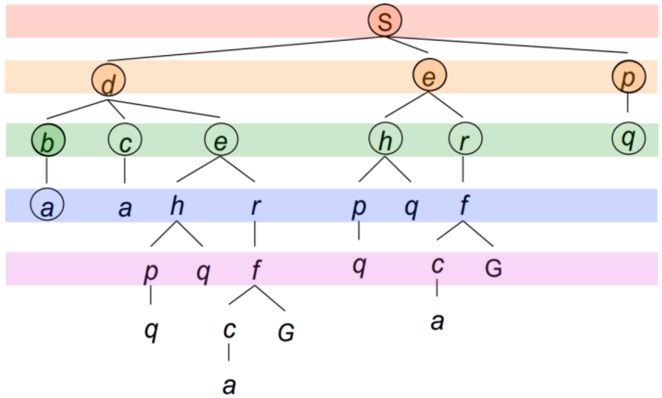




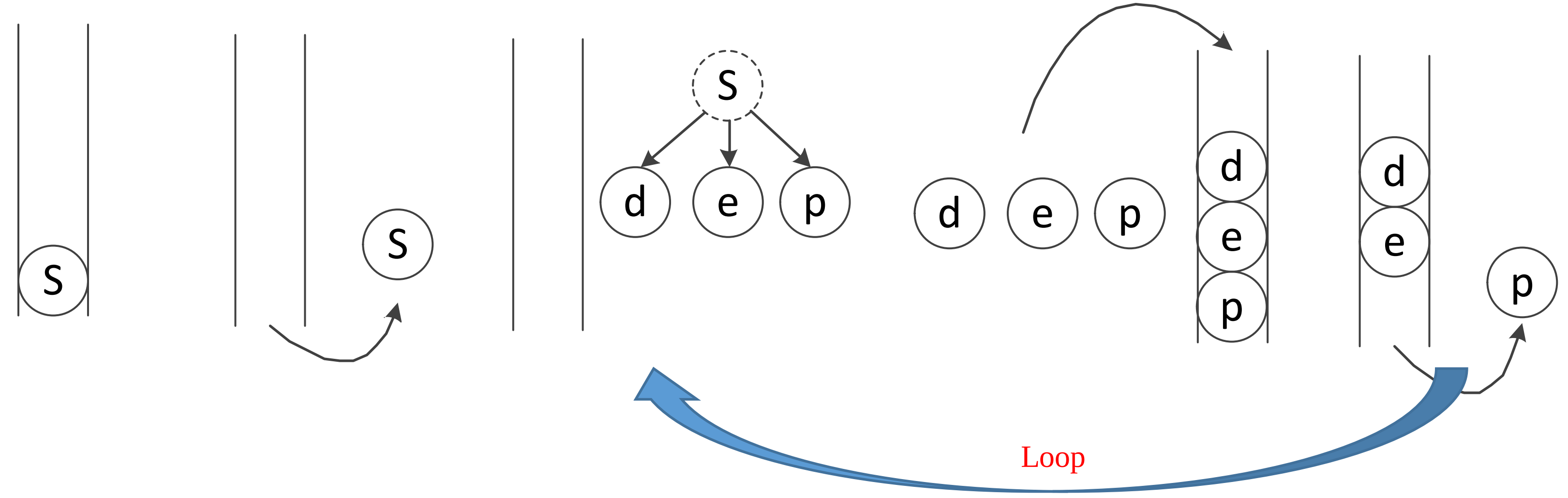
广度优先搜索（BFS）

- 实现：维护先进先出（FIFO）容器（即队列）

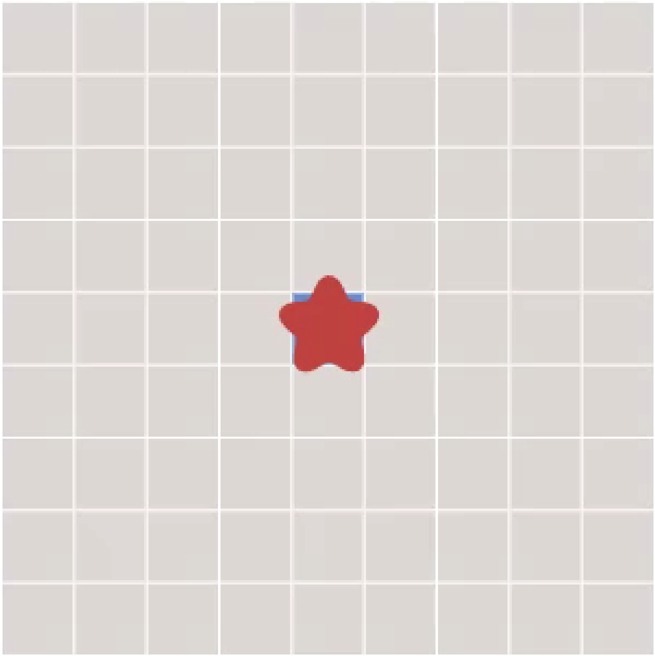




初始化容器　弹出搜索树中层级最浅的节点　扩展　按规则将子节点添加到容器中　弹出搜索树中层级最浅的节点

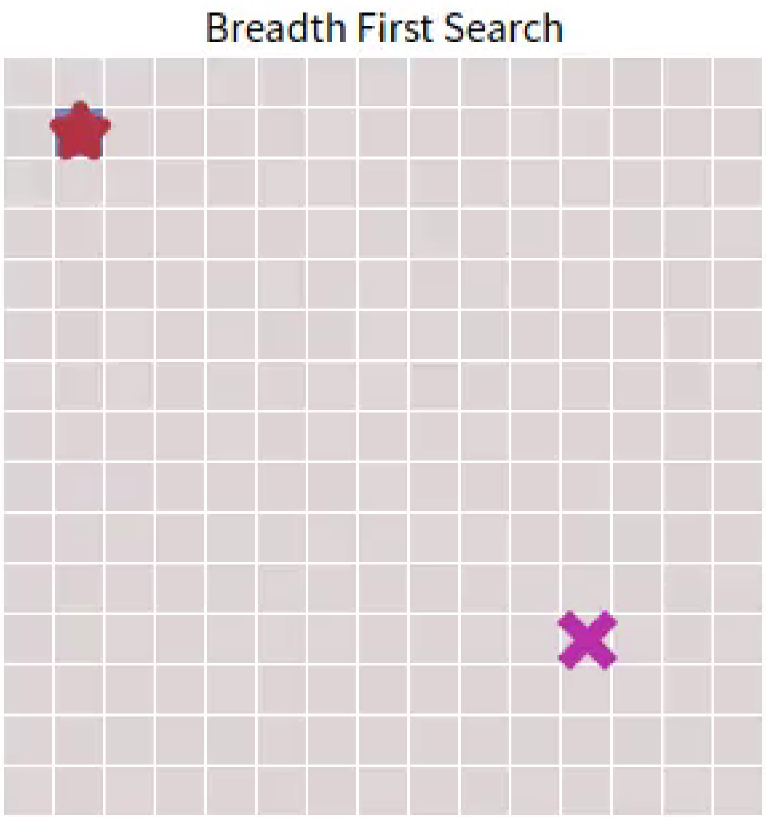


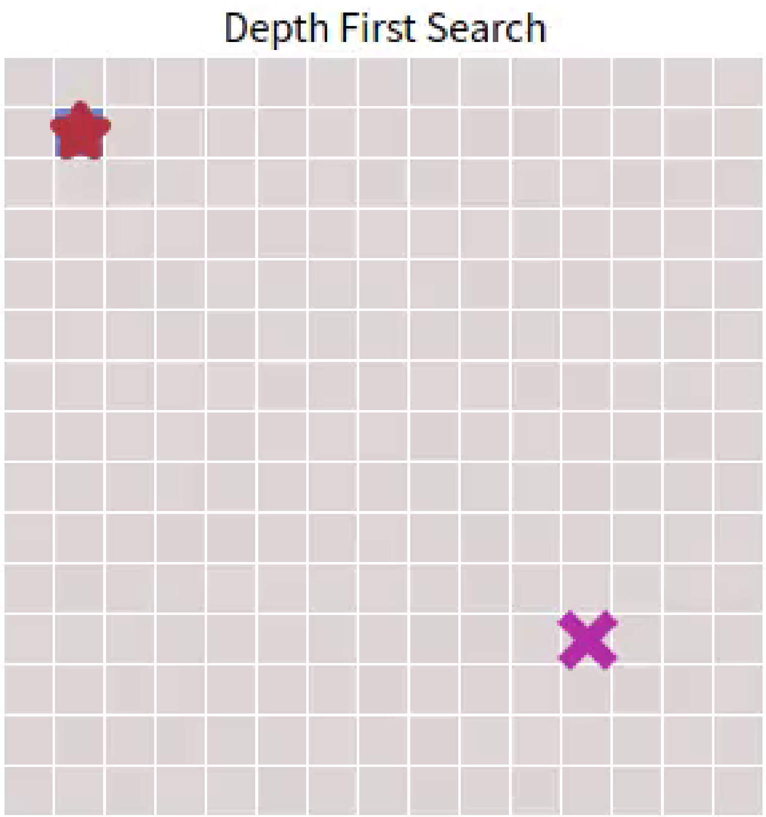
广度优先搜索（BFS）



Courtesy: Amit Patel's Introduction to A*, Stanford

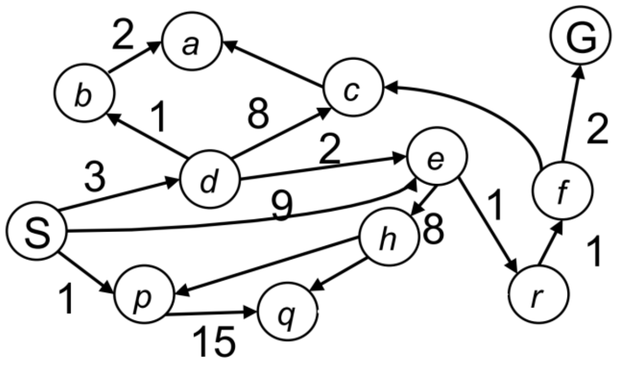
BFS vs. DFS

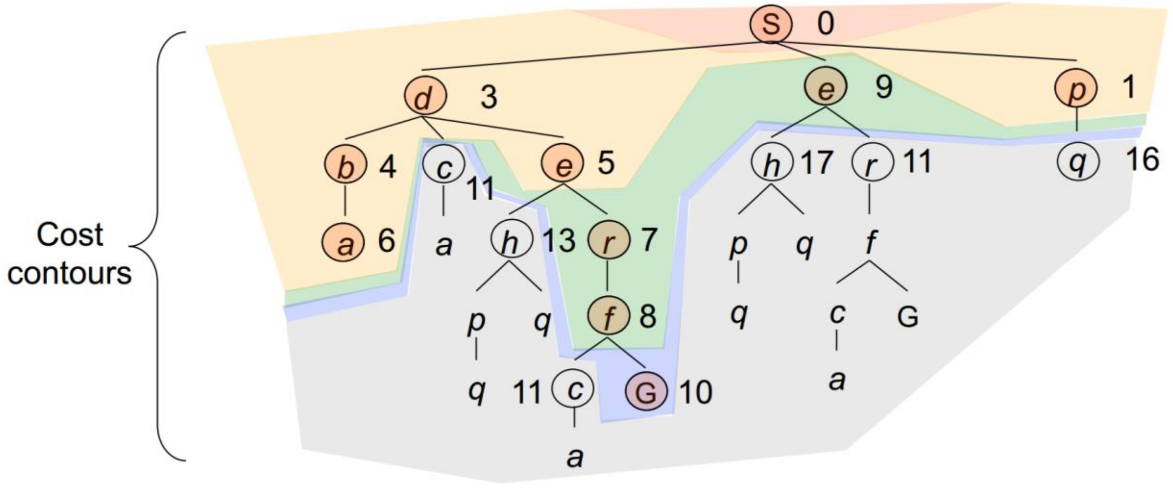




Remember BFS.

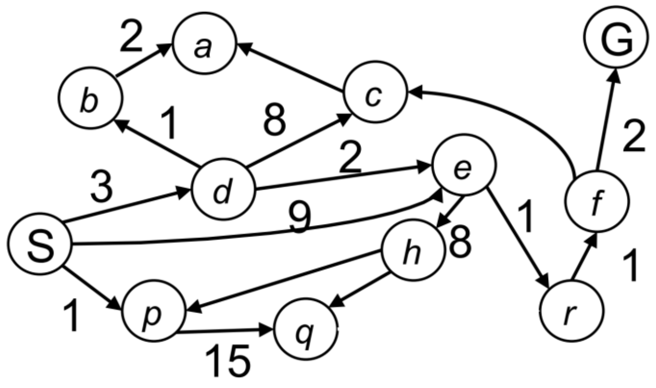
- 策略：每次取出容器中**累计代价g(n)最小**的节点
  - g(n):从初始点到点n的累计代价
  - 更新节点“n”的所有未拓展邻居“m”的累计成本g（m）
  - 已扩展节点的累计代价应为到起始点的最短路径代价

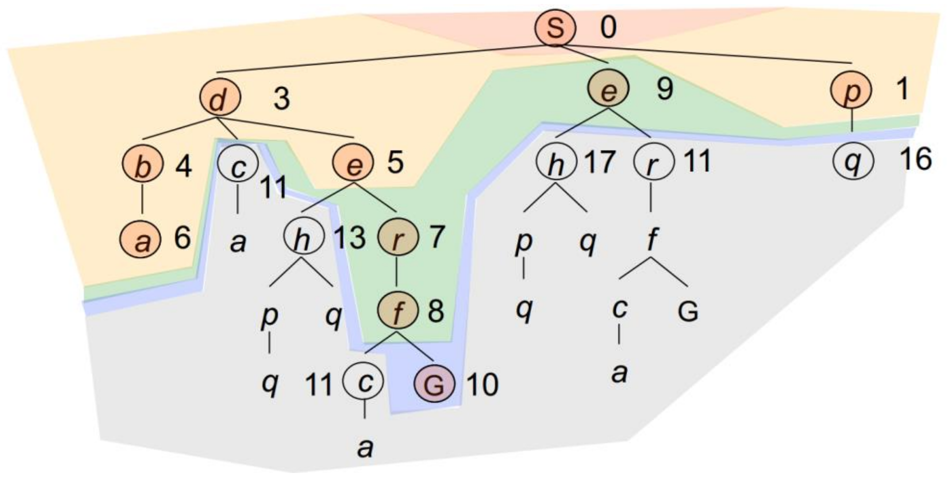


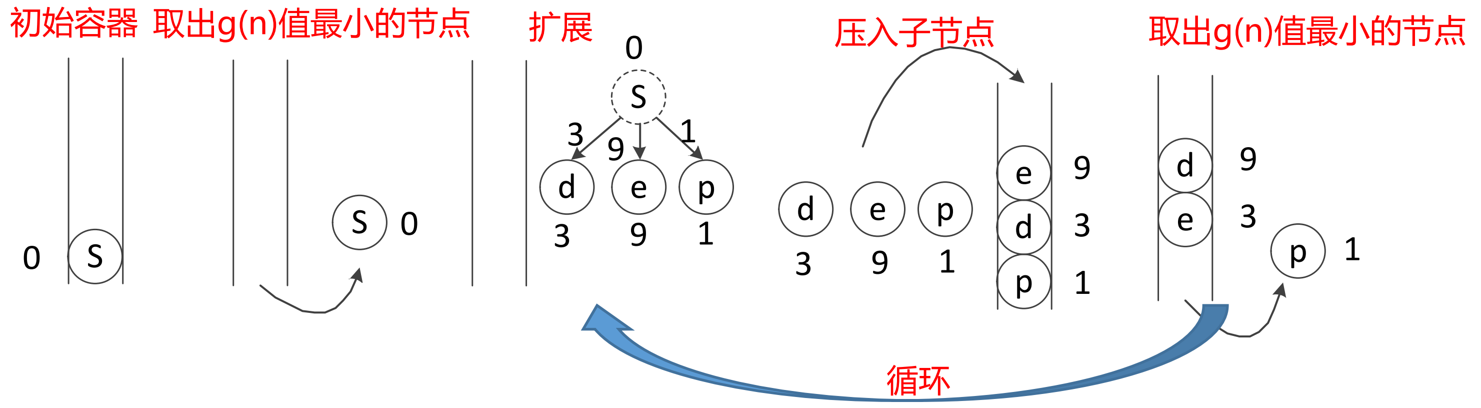


Dijkstra算法

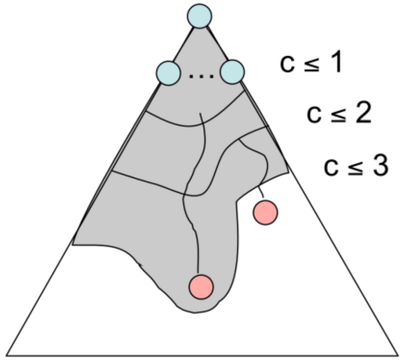
- 维护一个存储待扩展节点的优先队列
- 根据初始状态X_S初始队列
- 赋值$g(X_S)=0$, 且对于图中其他节点g(n)=无穷
- 循环
  - 如果队列为空，返回FALSE；退出循环
  - 从优先队列中移出最小g(n) 的节点"n"
  - 将节点"n"记作已扩展的节点
  - 如果节点"n"是终点，返回TRUE；退出循环
  - 对于所有未扩展的节点"n"的邻居节点"m"
    - 如果g(m) = 无穷
      - g(m)= g(n) + Cnm
      - 将节点"m"压入队列
    - 如果$g(m) > g(n) + C_{nm}$
      - $g(m)= g(n) + C_{nm}$
  - end
- 结束循环

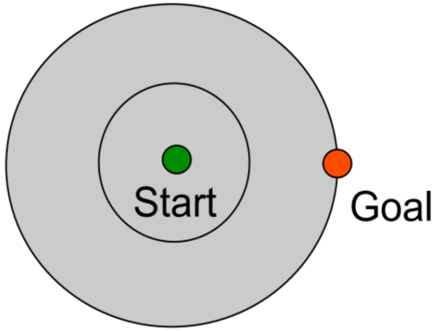






- 算法优缺点
  - 优点：
    - 具有完备性和最优性
  - 缺点：
    - 扩展不具有方向性，没有利用到目标点的信息





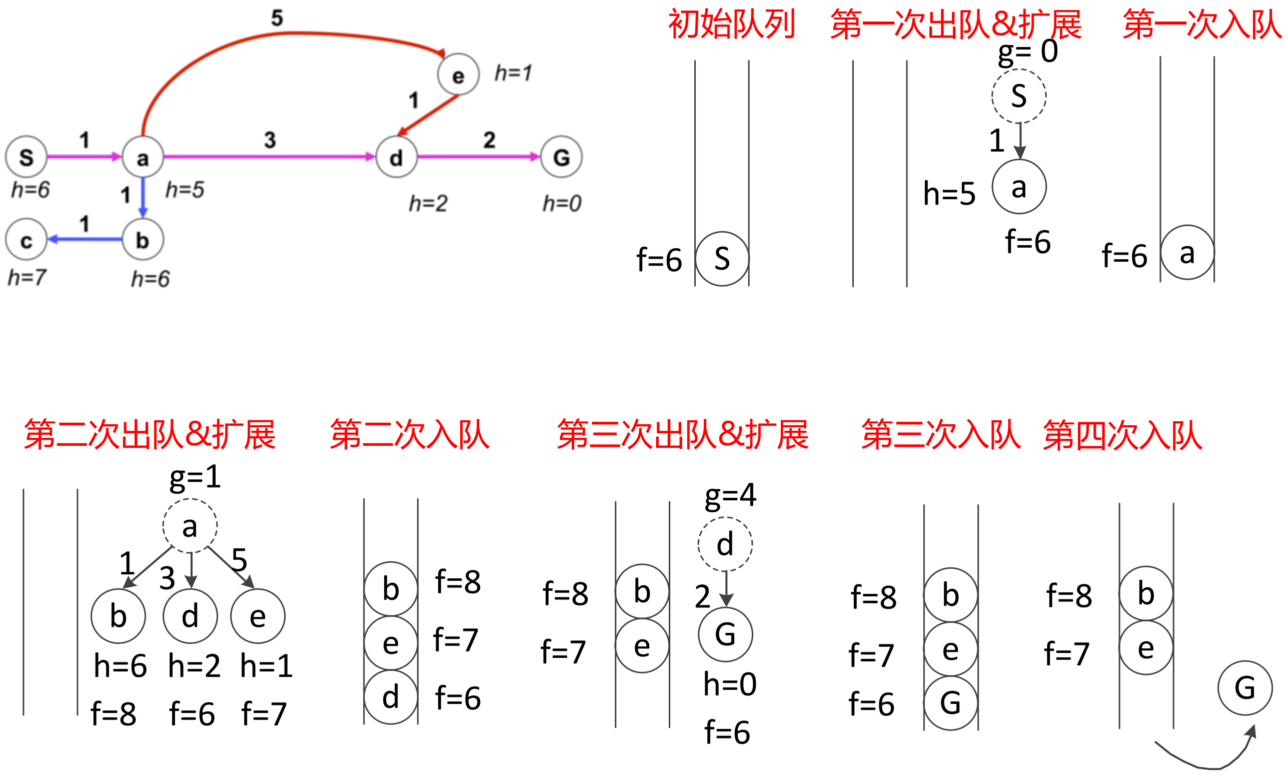
A*：Dijkstra算法+启发式函数

- 累计代价
  - g(n):从起始状态到节点“n”的最小估计代价
- 启发式函数
  - h(n)：从节点到目标点的最小估计代价
- 从起始状态到目标状态，经过节点“n”的最小估计代价为
- f（n）=g（n）+h（n）
- 策略：取出具有最小f(n)的节点

- 维护一个存储待扩展节点的优先队列
- 预先定义所有节点的启发式函数h(n)
- 根据初始状态X_S初始队列
- 赋值$g(X_S)=0$, 且对于图中其他节点g(n)=无穷
- 循环
  - 如果队列为空，返回FALSE；退出循环
  - 从优先队列中移出最小f(n)=g(n)+h(n) 的节点“n”
  - 将节点“n”记作已扩展的节点
  - 如果节点“n”是终点，返回TRUE；退出循环
  - 对于所有未扩展的节点“n”的邻居节点“m”
    - 如果g(m) = 无穷
      - g(m)= g(n) + Cnm
      - 将节点“m”压入队列
    - 如果$g(m) > g(n) + C_{nm}$
      - $g(m)= g(n) + C_{nm}$
  - end
- 结束循环

和Dijkstra算法的唯一区别

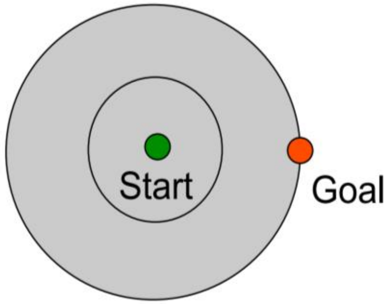
A*算法



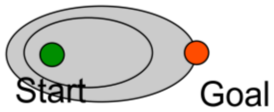
5
e h=1
1
S 1 a 3 d 2 G
h=6 h=5 h=2 h=0
1
c 1 b
h=7 h=6
初始队列
f=6 S
第一次出队&扩展
g= 0
S
1
h=5 a
f=6
第一次入队
f=6 a
第二次出队&扩展
g=1
a
1 3 5
b d e
h=6 h=2 h=1
f=8 f=6 f=7
第二次入队
b f=8
e f=7
d f=6
第三次出队&扩展
f=8 b
f=7 e
g=4
d
2
G
h=0
f=6
第三次入队
f=8 b
f=7 e
f=6 G
第四次入队
f=8 b
f=7 e
G

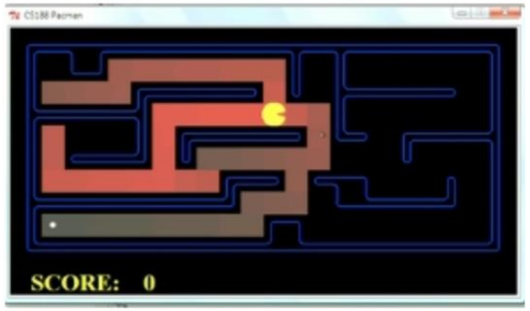
Dijkstra vs. A*

- Dijkstra算法朝各个方向探索



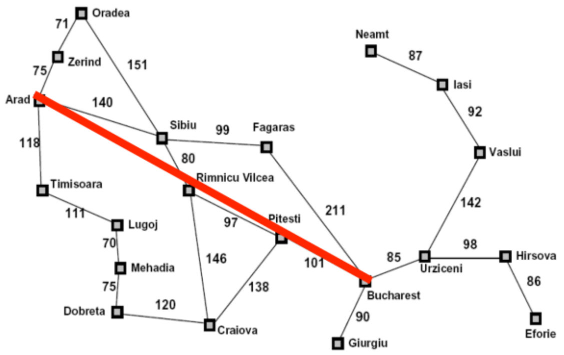
- A*算法主要朝着目标点方向探索

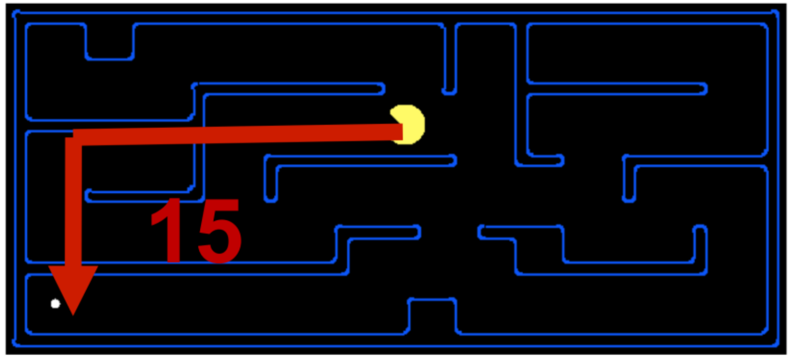




可采用的启发式函数

- 启发式函数h是可采用的（ admissible ）如果:
  - 所有的节点h(n) <= h*(n) ,h*(n) 是从节点n到终点的真实最小距离
- 如果启发式函数可采用，那么A*搜索是最优的
- 实际中想出可采用的启发式函数是使用A*中的主要环节
- 举例:





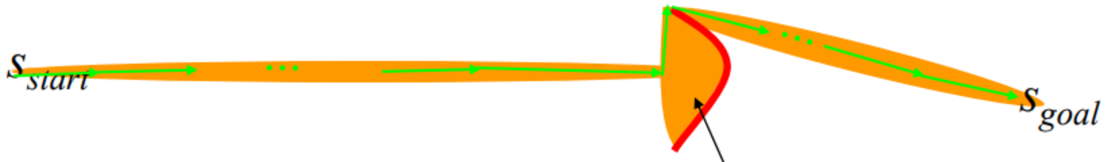
次优解

使用过分估计的启发式函数会怎样?

- 次优路径
- 更快

加权A*:（Weighted A*）

根据 $f = g + \varepsilon h, \varepsilon > 1$ 拓展搜索，更倾向于靠近终点的状态



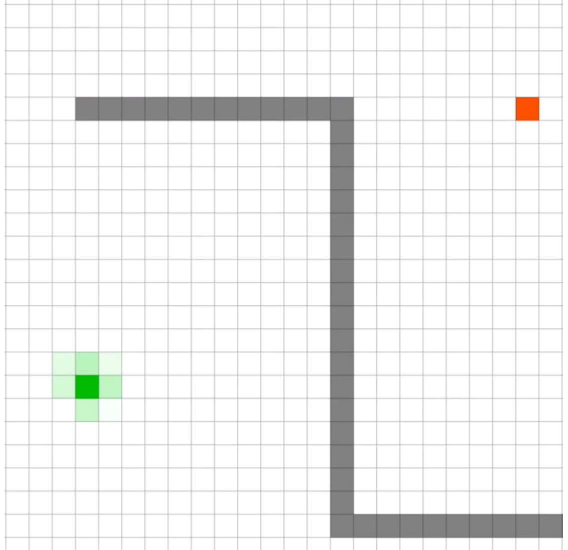
- 加权A* 搜索:

➢ 最优性 vs. 速度
➢ ε-suboptimal
➢ 比A*快几个数量级

$\underbrace{\text{加权A*-> Anytime A*-> ARA*->D*}}_{\text{超过本课程内容}}$

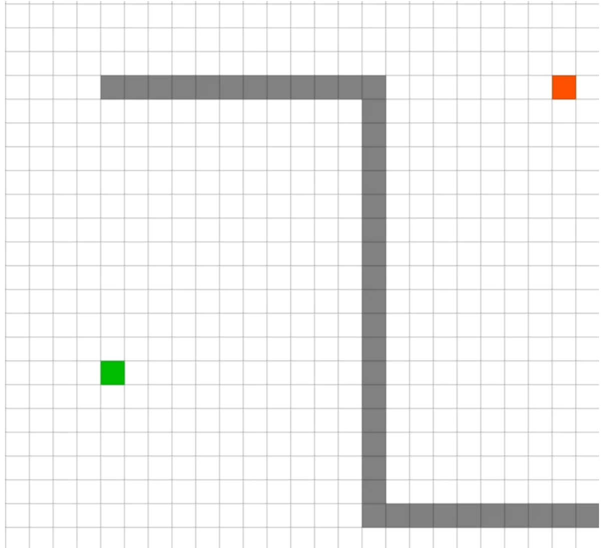
贪婪优先搜索 vs. 加权 A* vs. A*

$$f = a \cdot g + b \cdot h$$



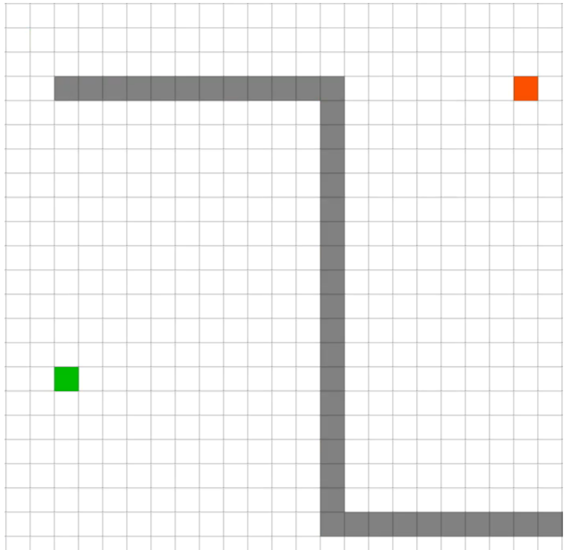
最贪婪的情况

$a = 0, b = 1$



可调整的贪婪

$a = 1, b = \varepsilon > 1$



最优

$a = 1, b = 1$

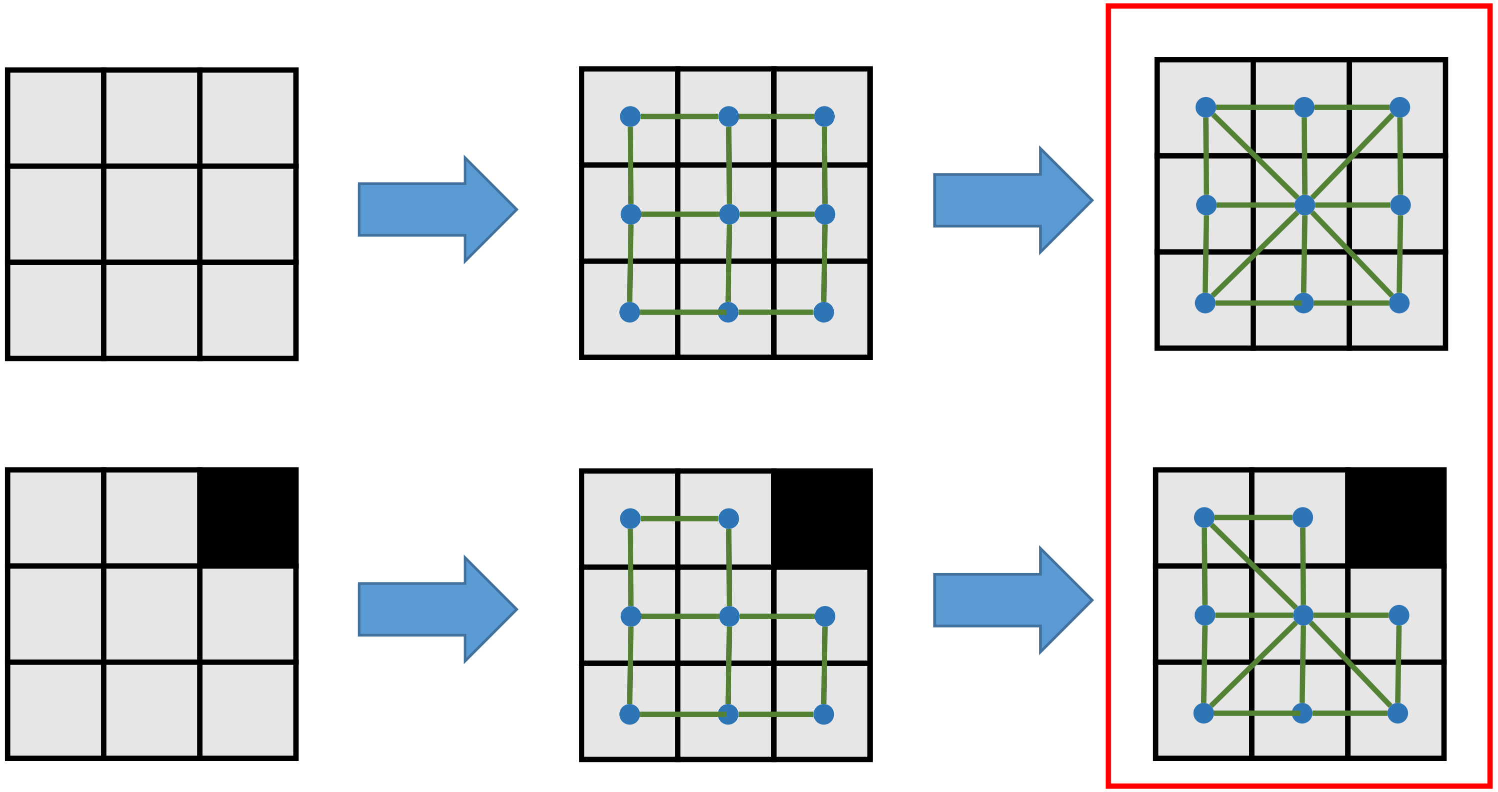
Dijkstra: $a = 1, b = 0$

Animation from: http://qiao.github.io/PathFinding.js/visual/

工程技巧

怎么把栅格表示为图？

每个单元是一个节点，边连接相邻的单元

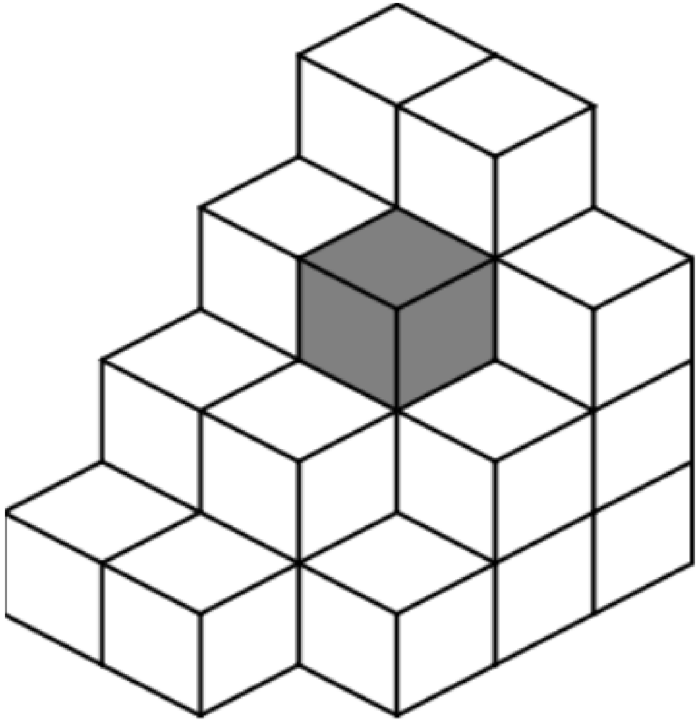


常用选择！

4 方向连接

8 方向连接

举例：基于栅格的路径搜索→机器人



- 创建一个稠密的栅格地图。
- 连接存储在栅格地图之间的状态。
- 通过栅格索引发现邻居
- 执行A*搜索。

- 优先级队列： C++
  - std::priority_queue
  - std::make_heap
  - std::multimap

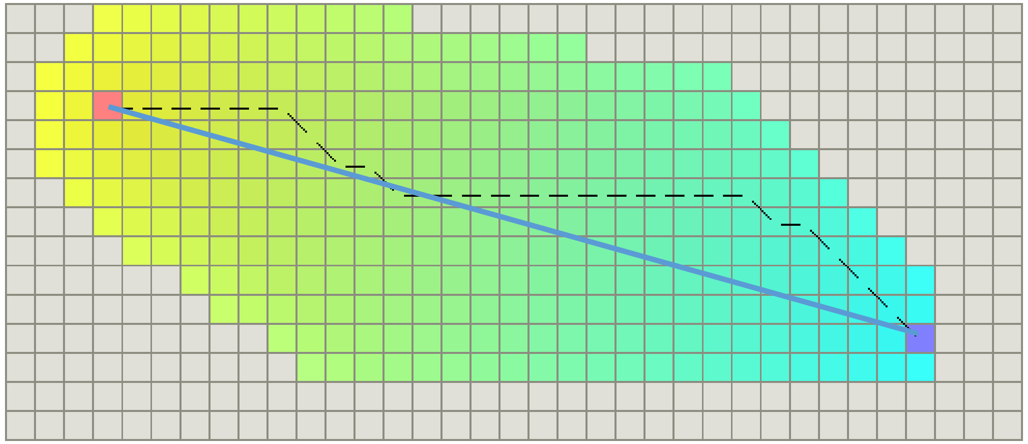
最优启发式函数

- 欧氏距离 ✓
- 曼哈顿距离 ✗
- L∞ 范数 ✓
- 0 ✓

➢ 它们是有用的，但没有一个是最好的选择，为什么？

- 因为没有一个是紧的。
- 紧是指测量的真正的最短距离。

欧氏距离



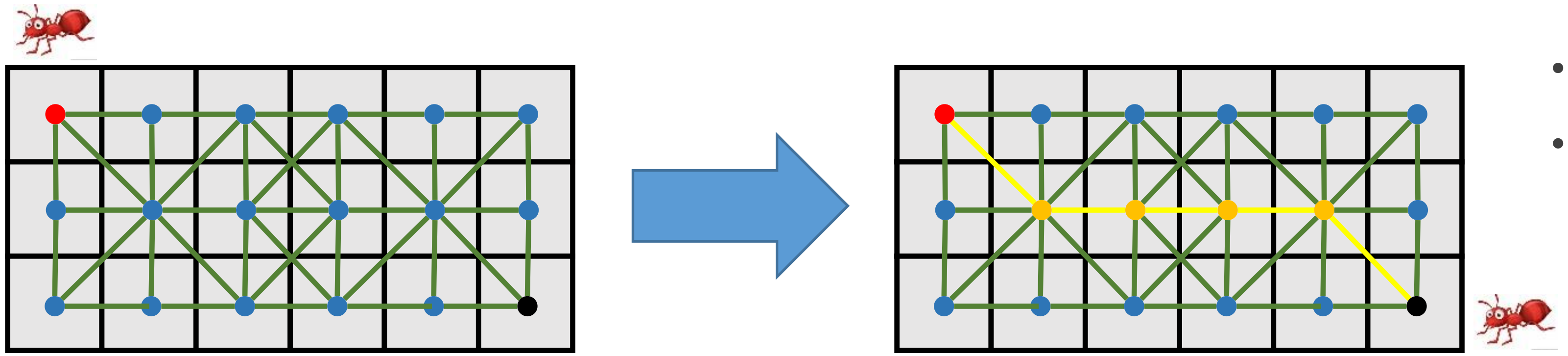
➢ 为什么扩展了这么多节点？

- 因为欧氏距离远不是真正的**理论最优解**。

最优启发式函数

如何获得真正的**理论最优解**？

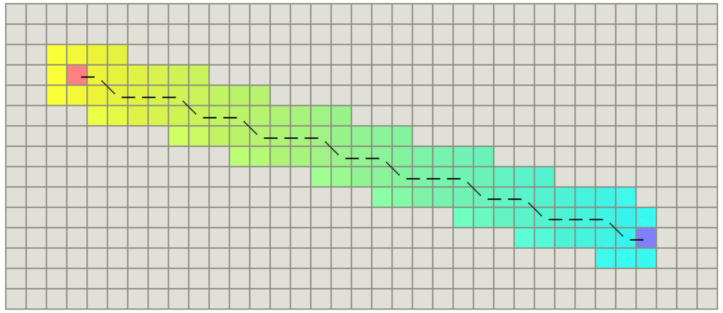
幸运的是，网格地图是高度结构化的。



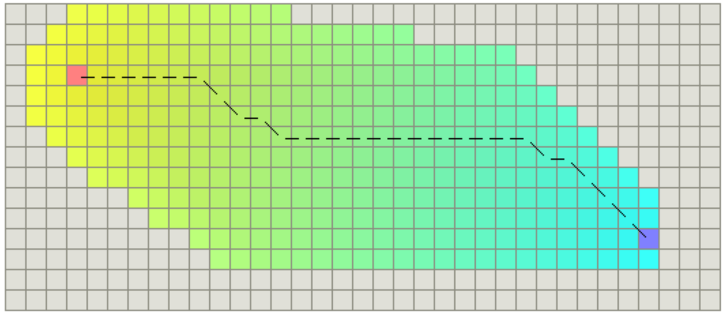
- 不需要搜索路径。
- 具备理论上的**闭式解！**

dx=abs(node.x −goal.x)
dy=abs(node.y −goal.y)
$h=(dx+dy)+(\sqrt{2}-2)*\min(dx,dy)$

对比



对角启发式函数：Diagonal Heuristic



欧氏距离

3D情况

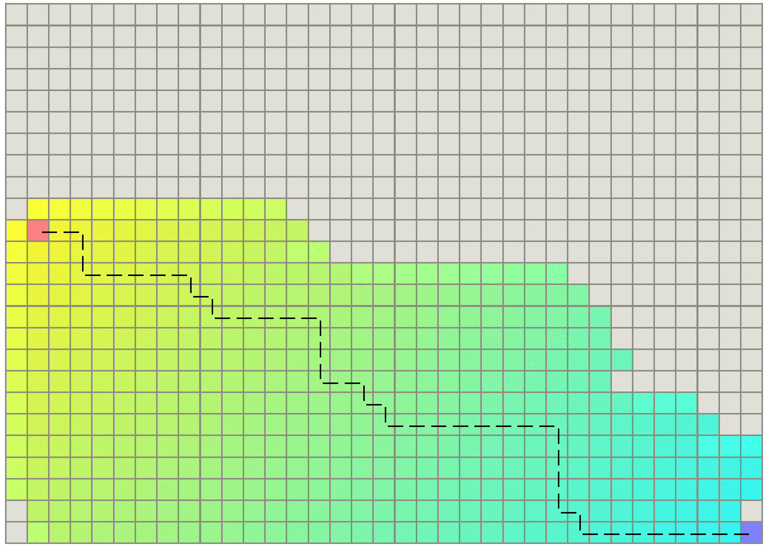
打破对称性：Tie Breaker

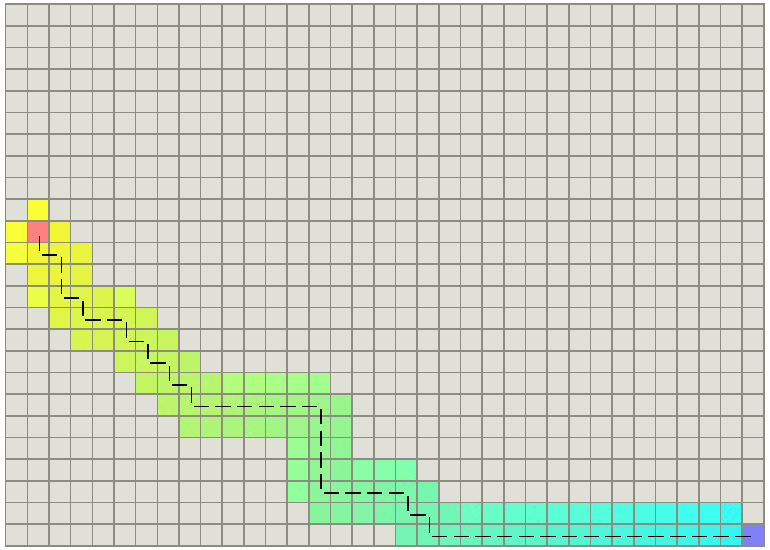
- 许多节点具有相同的 f 值。
- 它们之间没有差异，这使得A*平等地扩展它们。

- 修改 f 打破对称性。
- 使相同 f 值不同
- 轻微地放大 h

$$h = h \times (1.0 + p)$$

$$p < \frac{minimum\ cost\ of\ one\ step}{expected\ maximum\ path\ cost}$$





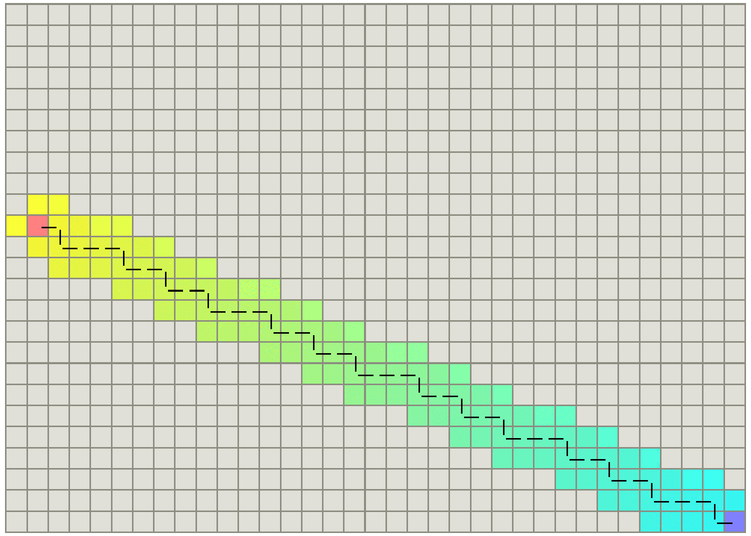
稍微打破了最优性，有关系吗？

打破对称性：Tie Breaker

核心思想：在相同 f 的节点中找到倾向性

- 当节点具有相同 f ,比较他们 h .
- 将确定性随机数添加到启发式中。
- 倾向从起点到目标的直线路径。

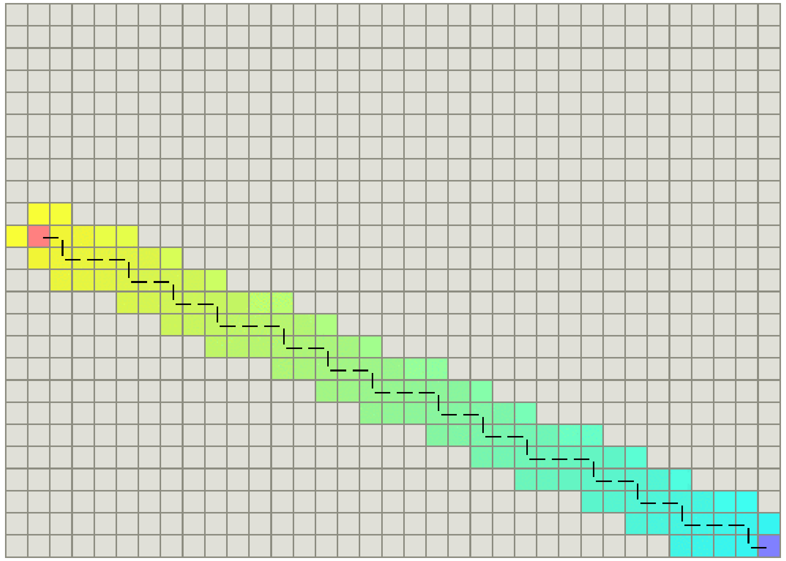
$$
\begin{gathered}
dx1 = abs(node.x - goal.x) \\
dy1 = abs(node.y - goal.y) \\
dx2 = abs(start.x - goal.x) \\
dy2 = abs(start.y - goal.y) \\
cross = abs(dx1 \times dy2 - dx2 \times dy1) \\
\mathrm{h} = h + cross \times 0.001
\end{gathered}
$$



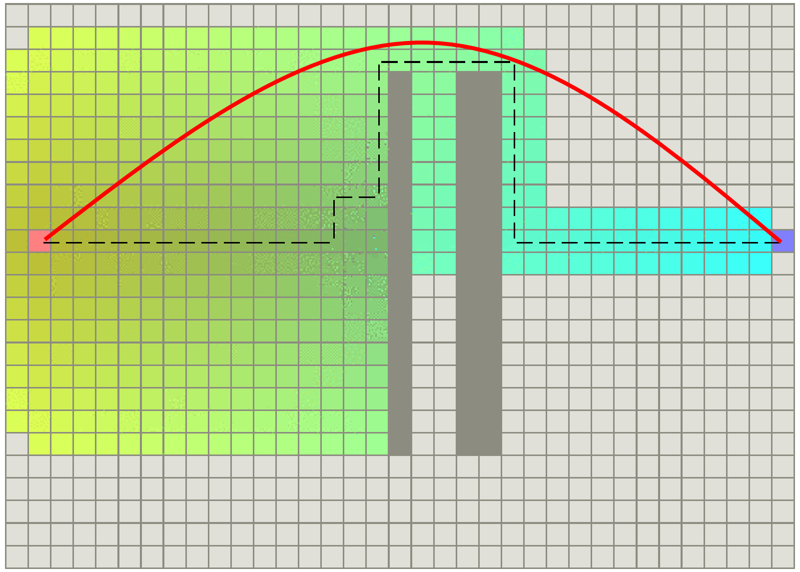
- ...许多自定义方式

回顾打破对称性：Tie Breaker

- 首选从起点到目标的直线路径。



两者都是**最优**的，
但是…



这是最短的路径，但不利于轨迹优化

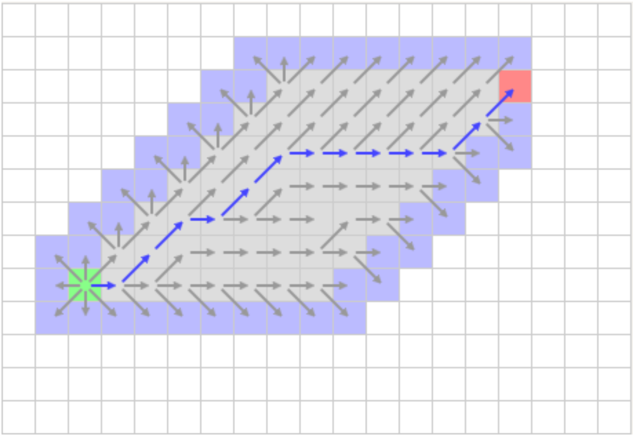
一种系统性实现打破对称性的方法：跳跃点搜索 Jump Point Search (JPS)

Jump Point Search

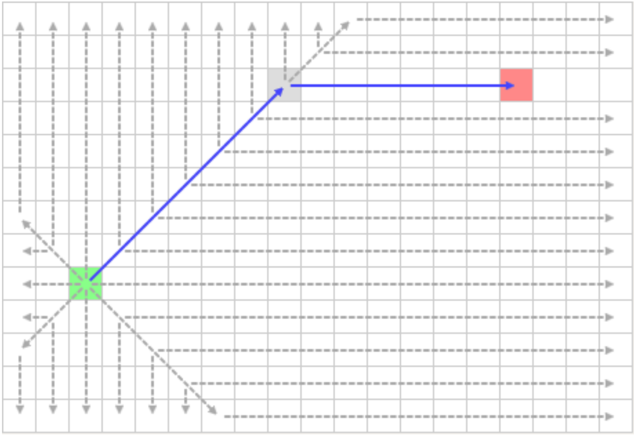
Algorithm Workflow

JPS 核心：找到对称性并打破它们。

A* 探索所有对称路径

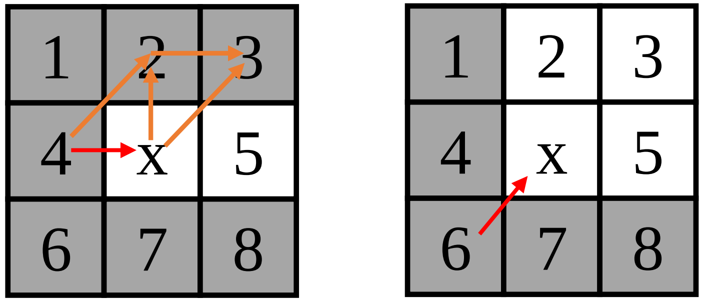


JPS 选择一条路径



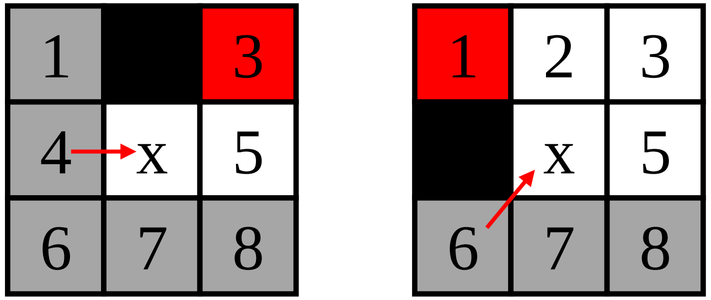
Look Ahead 规则

直线　　对角



考虑:

- 当前节点x
- x的扩展方向



直线修剪

- 灰色节点：较差的邻居，当去他们，路径没有x 更短。丢弃
- 白色节点： 自然节点。
- 在扩展搜索时，我们只需要考虑自然节点

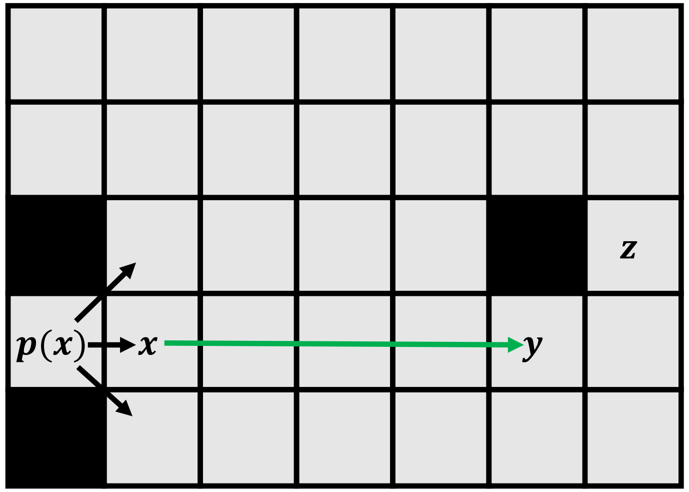
强制节点

- 附近有障碍物x
- 红色节点是强制节点。
- 从x’ 他们的父母被障碍物挡住了。

See: http://users.cecs.anu.edu.au/~dharabor/data/papers/harabor-grastien-aaai11.pdf Equation 1/2

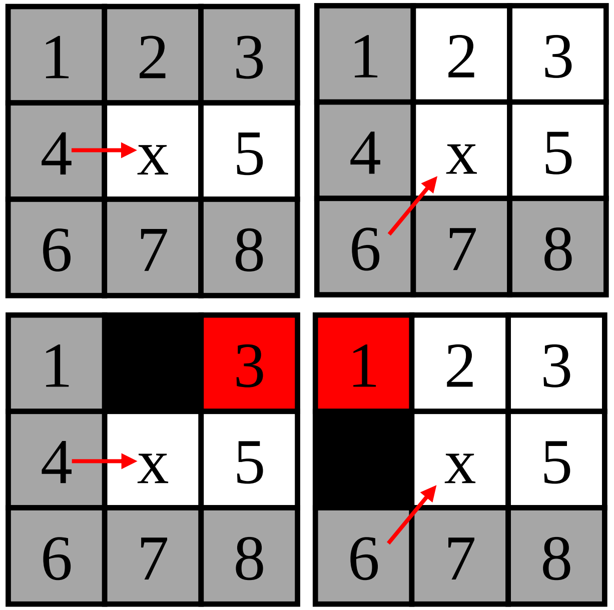
Jump Point Search

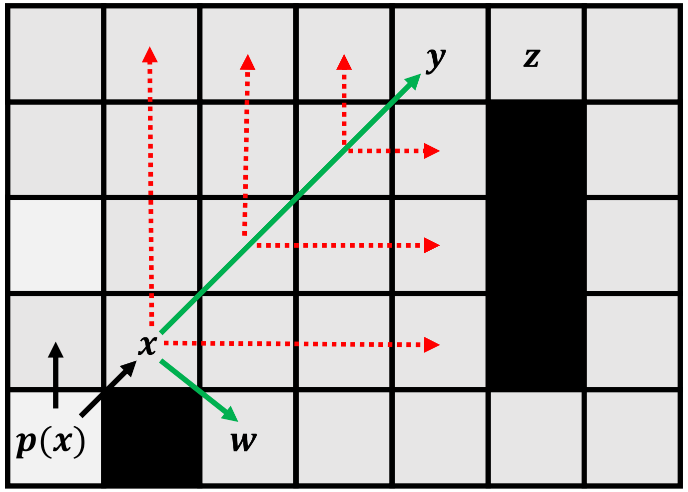
Jumping 规则



直线跳跃

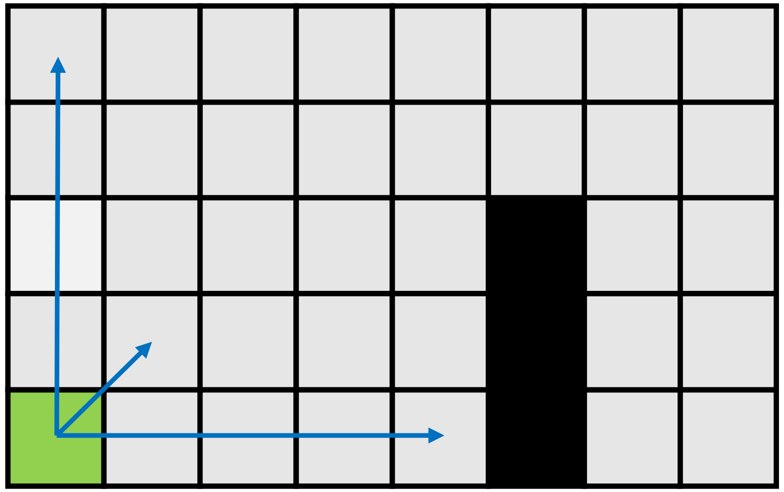
Look Ahead 规则



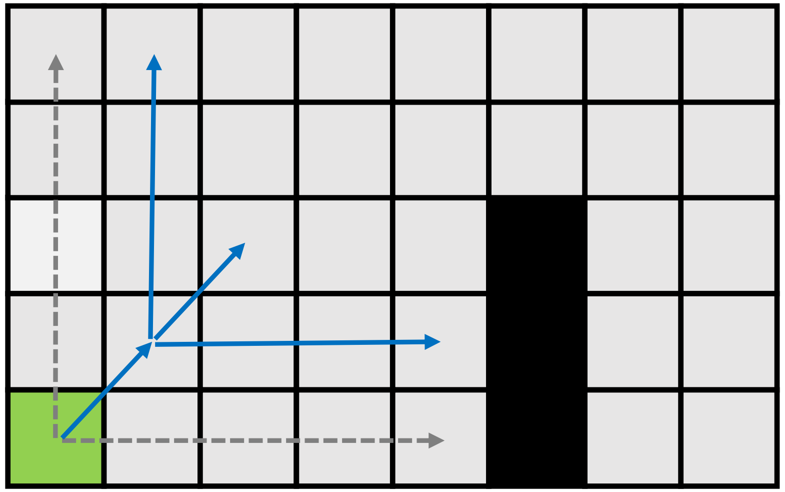


对角跳跃

- 递归地应用直线修剪规则，并将y识别为x的跳点后继。这个节点很有趣，因为它有一个邻居z，除了访问x然后访问y的路径之外，无法以最佳方式到达。
- 递归地应用对角修剪规则，并将y识别为x的跳点后继。
- 在每一个对角线步骤之前，我们首先直下弯。只有当两个直递归都不能识别跳跃点时，才能再次对角跨步。
- 节点w，x的强制邻居，被正常扩展。（也可推入打开的列表，即优先级队列）

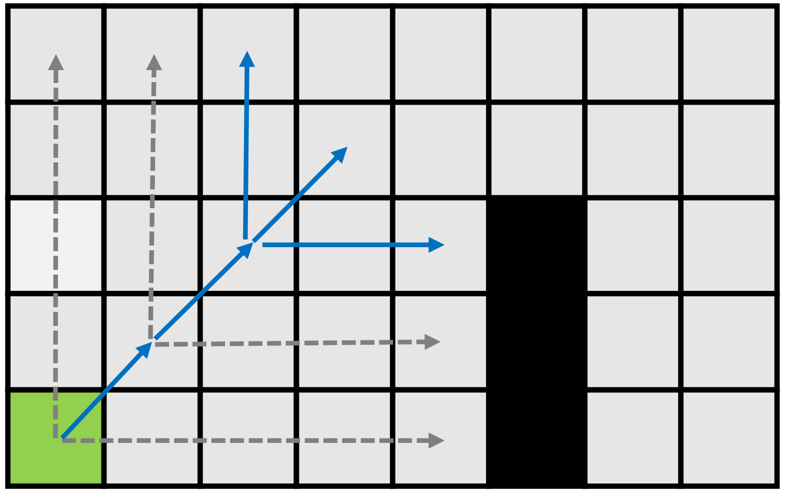


- 水平和垂直展开
- 两次跳跃都以障碍物结束
- 沿对角线移动

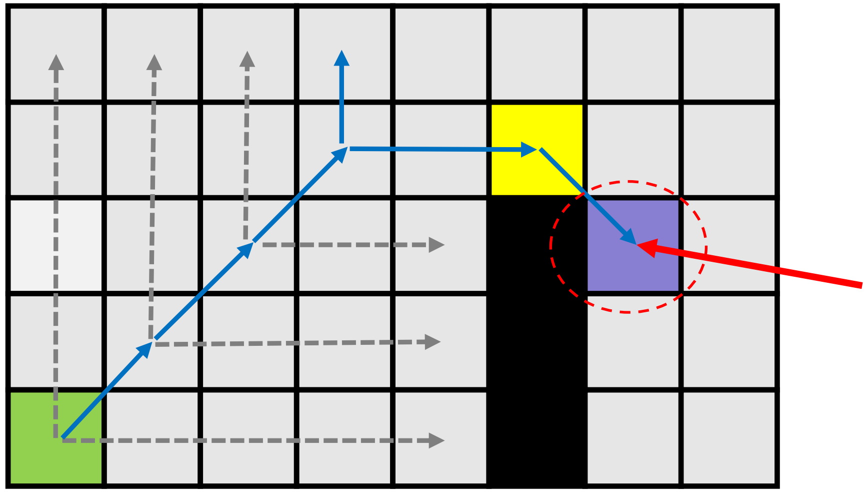


- 水平和垂直展开
- 两次跳跃都以障碍物结束
- 沿对角线移动

Jump Point Search

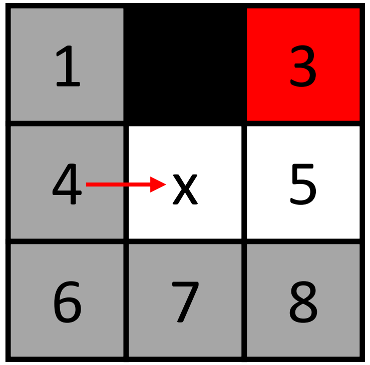


- 水平和垂直展开
- 两次跳跃都以障碍物结束
- 沿对角线移动

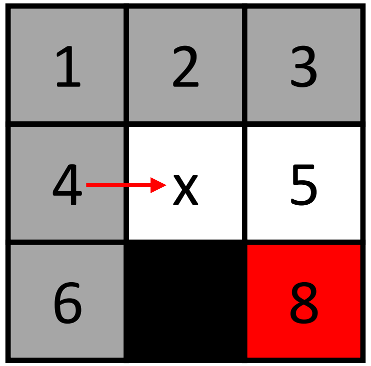


- 记住：你只能直跳或斜跳；从不分段跳跃
- 垂直扩展结束于障碍物中
- 向右展开查找具有
强迫邻居

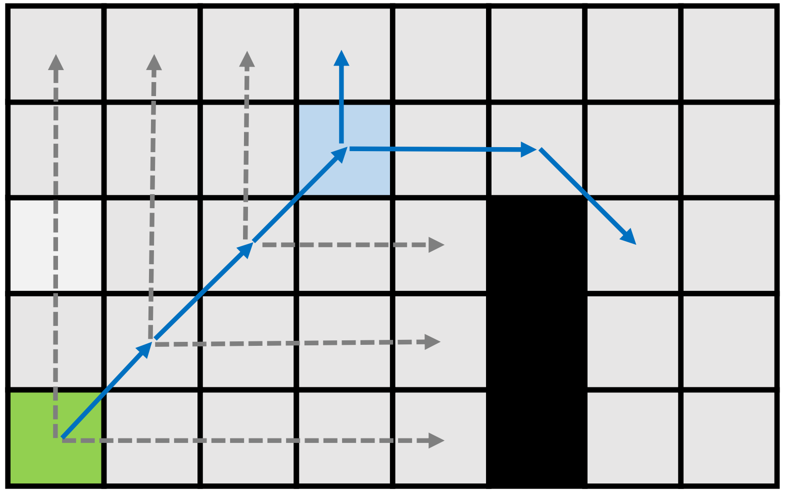
重新调用规则



得到



Jump Point Search

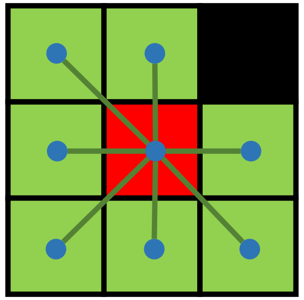


- 蓝色节点
- 将其放入 open list.

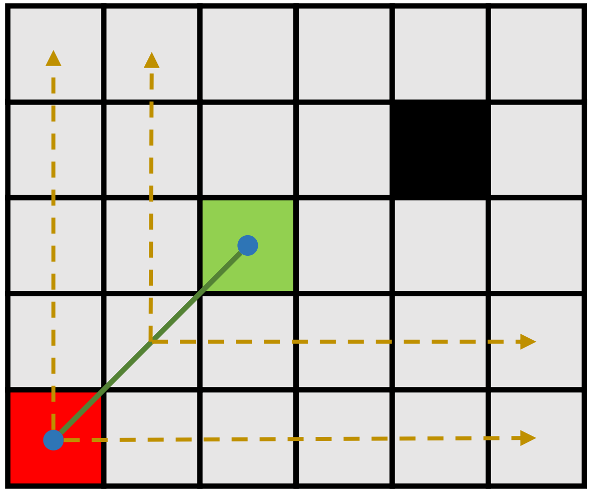
Recall A*’ s pseudo-code, JPS’ s is all the same!

- 维护一个存储待扩展节点的优先队列
- 预先定义所有节点的启发式函数h(n)
- 根据初始状态X_S初始队列
- 赋值$g(X_S)=0$, 且对于图中其他节点g(n)=无穷
- 循环
  - 如果队列为空，返回FALSE；退出循环
  - 从优先队列中移出最小f(n)=g(n)+h(n) 的节点“n”
  - 将节点“n”记作已扩展的节点
  - 如果节点“n”是终点，返回TRUE；退出循环
  - 对于所有未扩展的节点“n”的 邻居 节点“m”
    - 如果g(m) = 无穷
      - g(m)= g(n) + Cnm
      - 将节点“m”压入队列
    - 如果$g(m) > g(n) + C_{nm}$
      - $g(m)= g(n) + C_{nm}$
  - end
- 结束循环

A*: “Geometric” neighbors



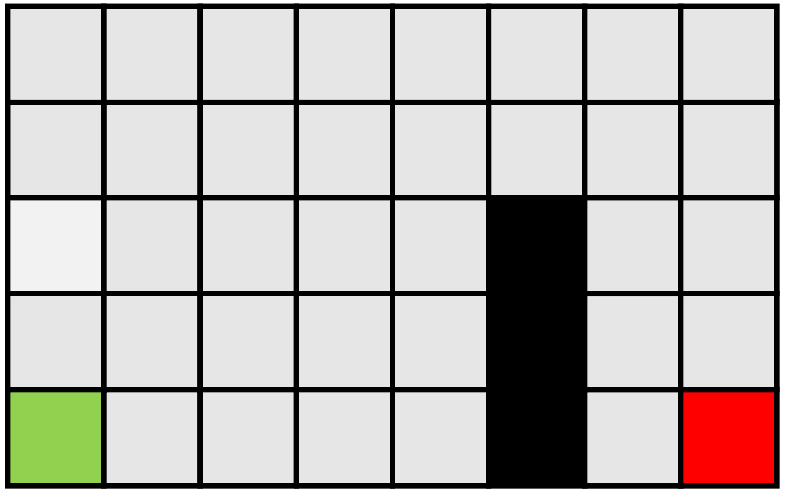
JPS: “Jumping” neighbors



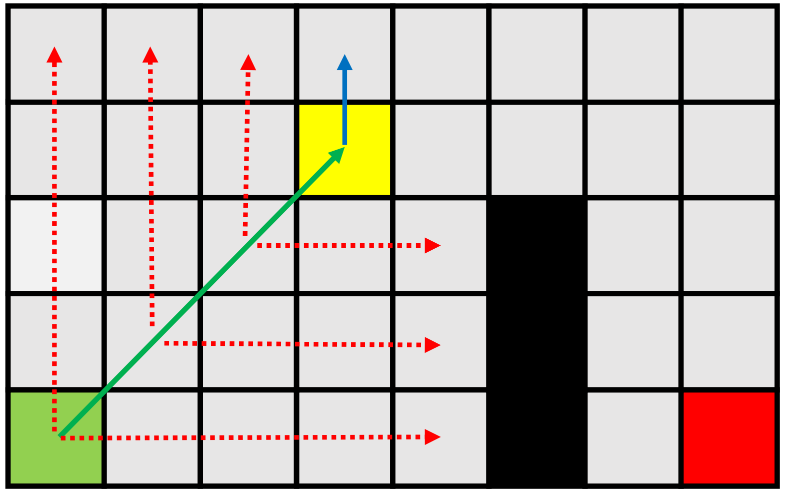
Example

Jumping Example

案例

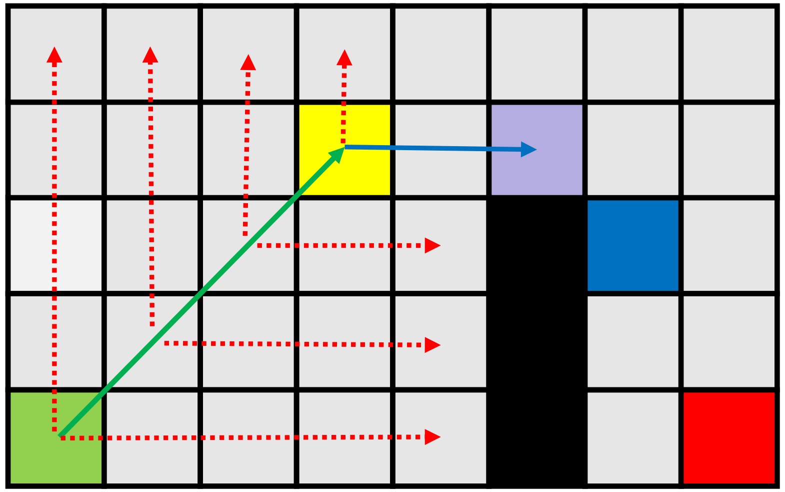


Jumping Example



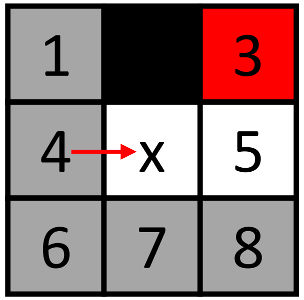
- 扩展-->对角移动
- 最后找到一个关键节点，将其添加到open list
- 从open list 中弹出它（唯一一个）。
- 垂直扩展，在障碍物处结束。

Jumping Example

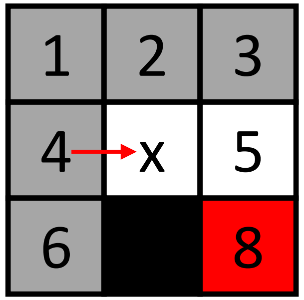


- 水平扩展，遇到具有强制邻居的节点。
- 将其添加到open list

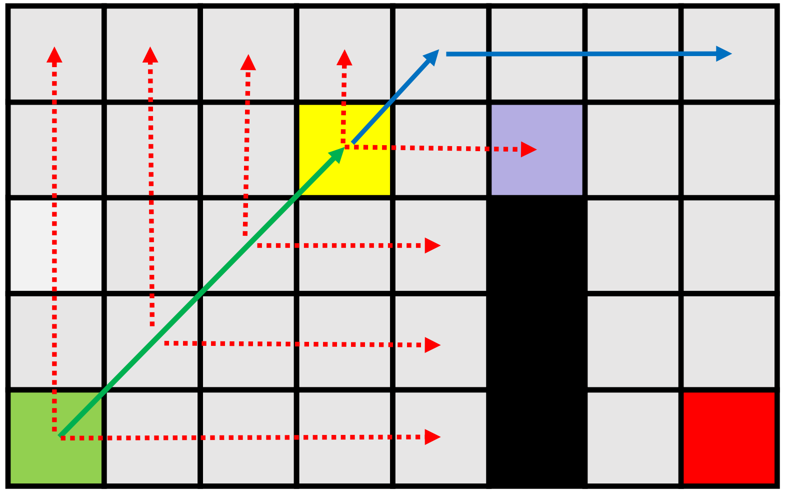
Recall the rule



So we have

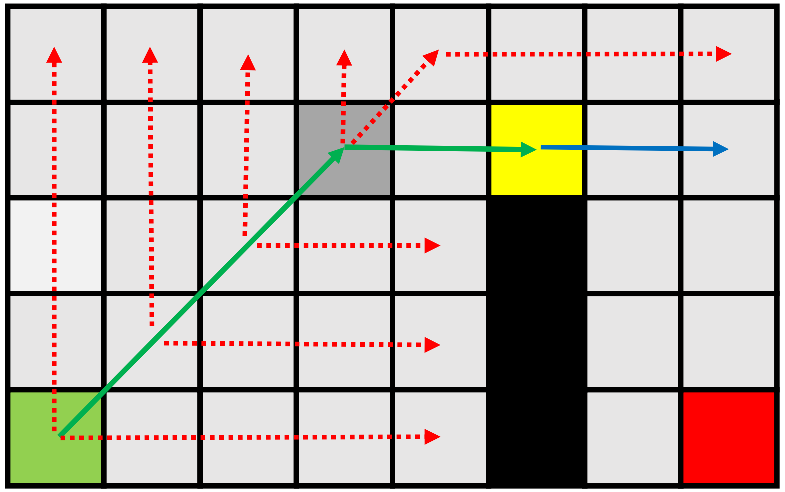


Jumping Example



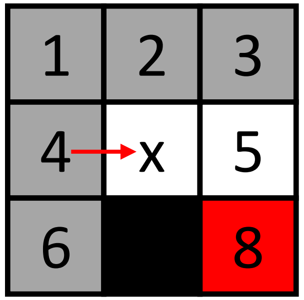
- 对角扩展，find nothing
- 完成当前节点的扩展。

Jumping Example

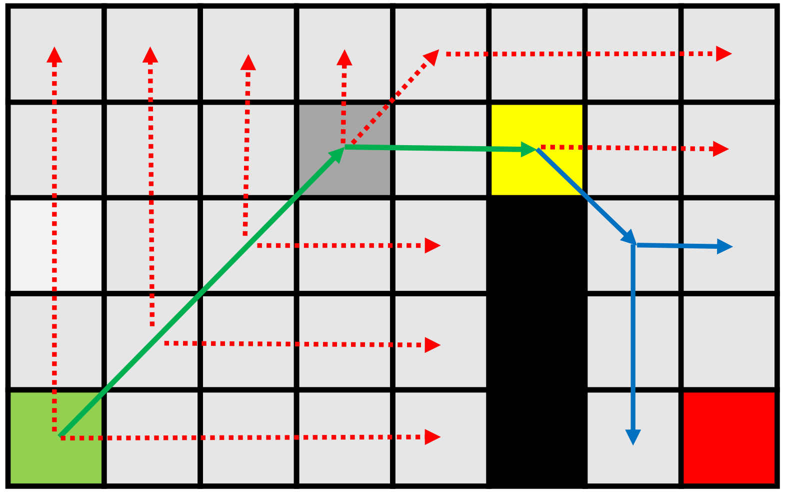


- 检查打开列表中的“新最佳”节点
- 水平扩展
- Finds nothing。

Remember the rule

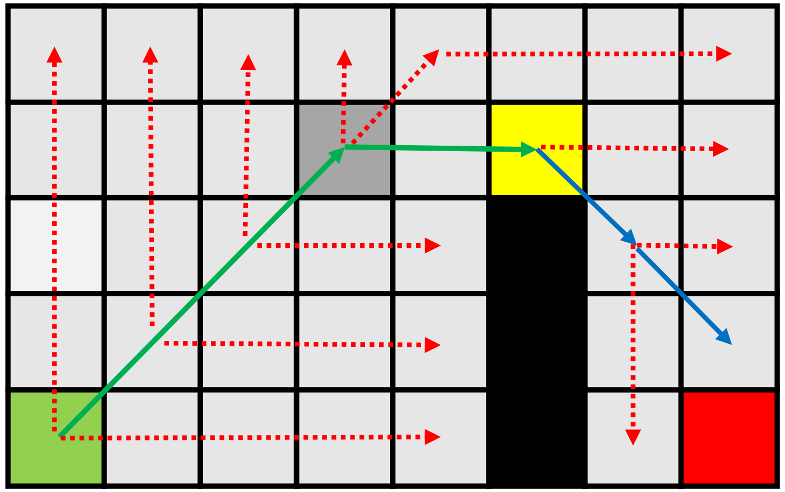


Jumping Example



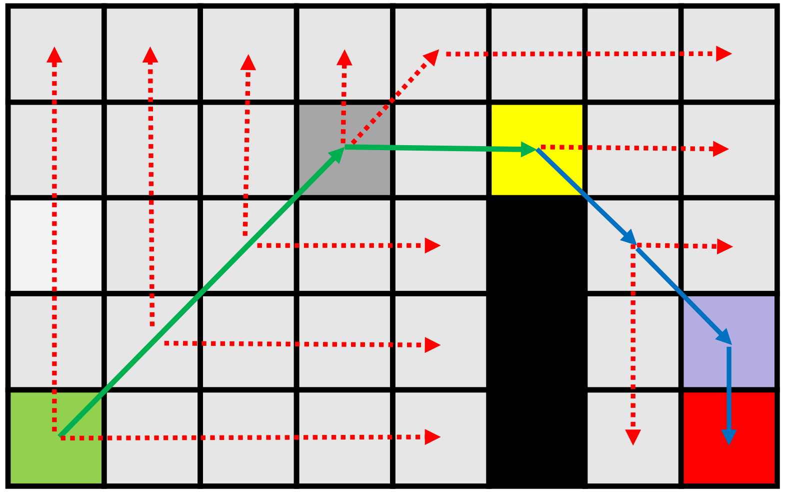
- 沿对角线移动
- 首先沿垂直和水平方向扩展

Jumping Example



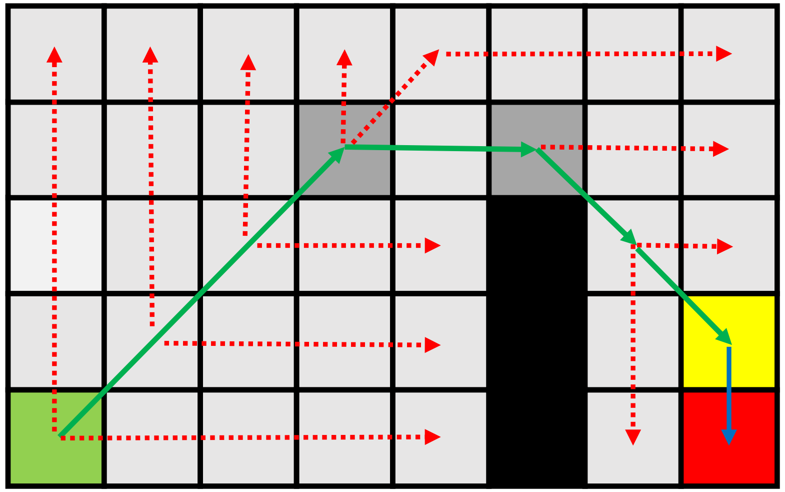
- Finds nothing.
- 沿对角线移动。

Jumping Example



- 水平和垂直扩展
- 查找目标，与查找具有强制邻居的节点同样权重
- 将此节点添加到open list
- 完成当前节点的扩展（没有自然邻居）
- 将其从open list 中弹出

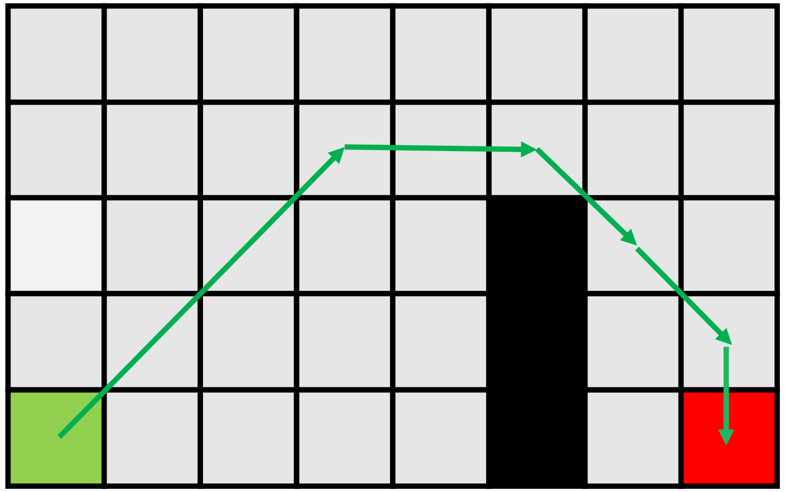
Jumping Example



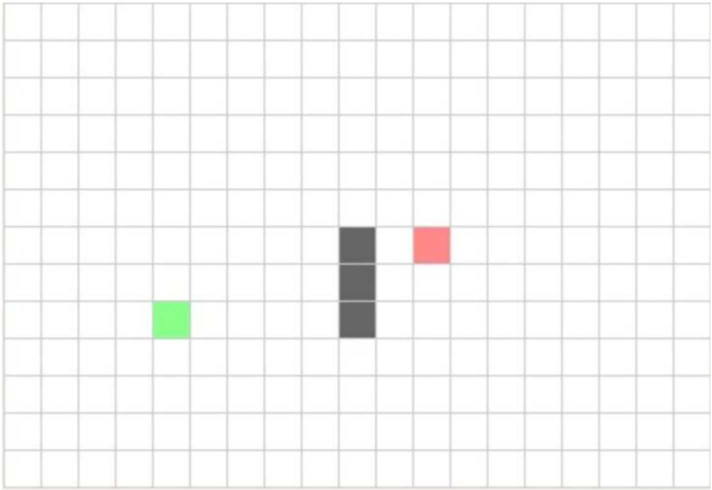
- 检查open list 中的“new best”节点。
- 水平扩展（无位置），垂直扩展（找到目标）。
- 结束。

Jumping Example

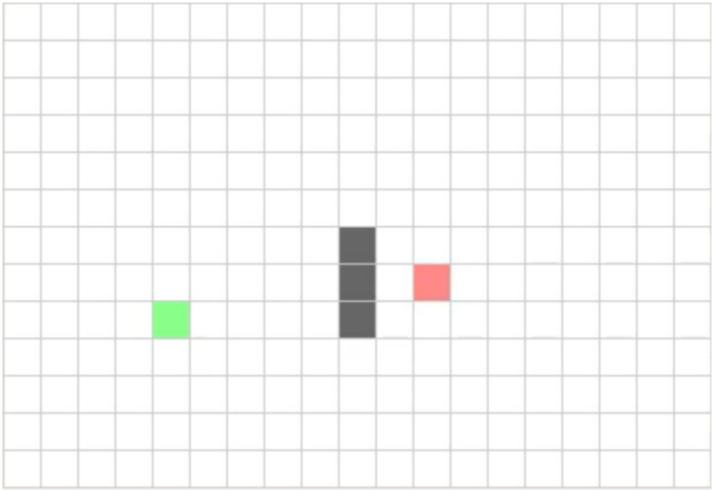
Final Path



Jumping Example



(1)



(2)

Thanks:

https://zerowidth.com/2013/a-visual-explanation-of-jump-point-search.html

3D JPS

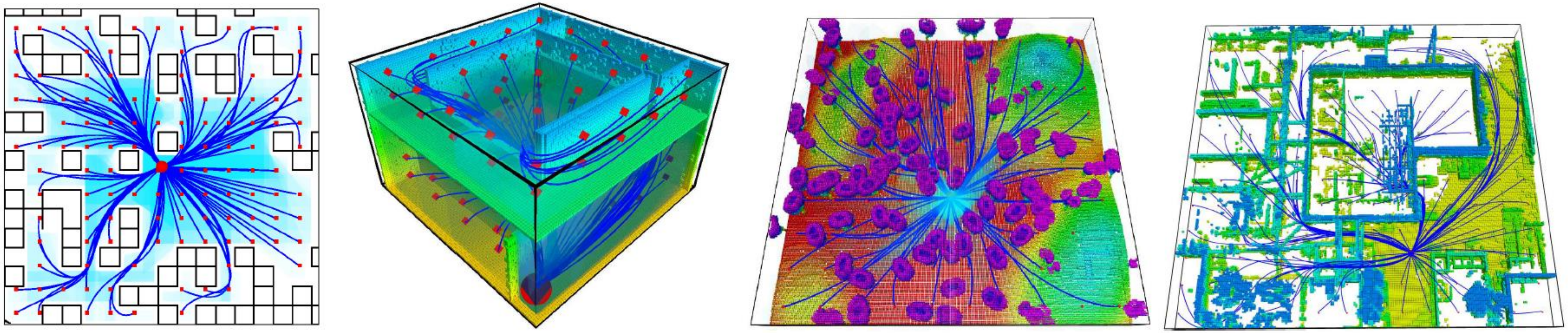


Table 1. Trajectory Generation Run Time (sec)

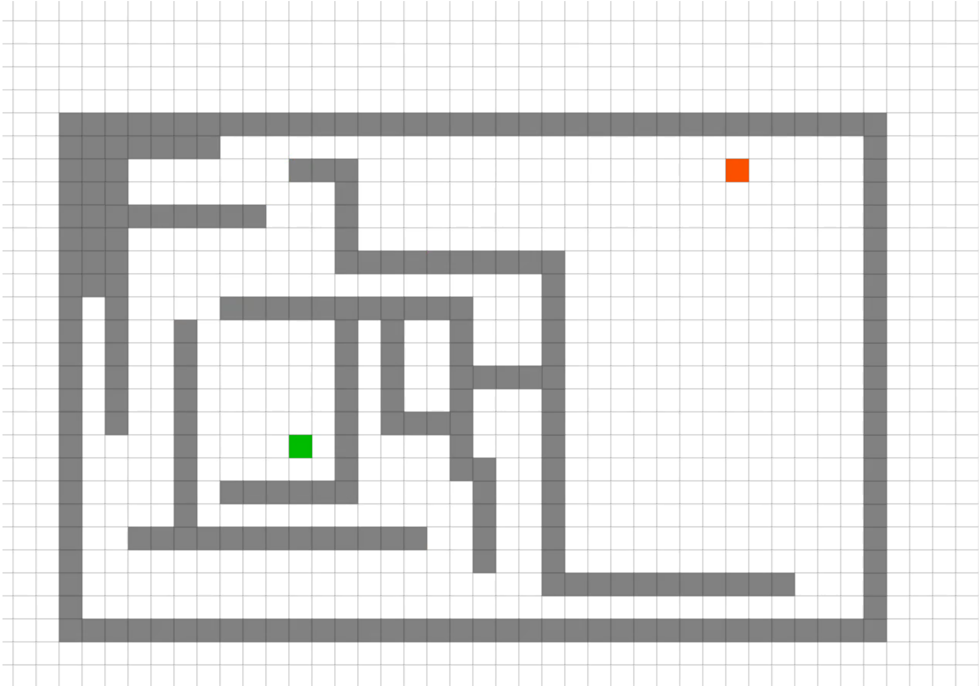
Map	Size	# of Cells	# of Trajs	Time (s)	Path Planning A*	Path Planning JPS	Convex Decomp	Traj Opt	Replan (JPS)
Random Blocks	$40 \times 40 \times 1$	1.4×10^6	130	Avg	0.57	0.034	0.0021	0.028	0.065
				Std	1.26	0.034	0.0028	0.022	0.051
				Max	9.98	0.19	0.020	0.099	0.27
Multiple Floors	$10 \times 10 \times 6$	5.9×10^5	147	Avg	6.12	0.039	0.0064	0.082	0.13
				Std	15.77	0.046	0.0038	0.041	0.081
				Max	84.56	0.22	0.021	0.23	0.45
The Forest	$50 \times 50 \times 6$	1.8×10^6	89	Avg	0.65	0.033	0.0039	0.055	0.094
				Std	1.57	0.044	0.0024	0.031	0.068
				Max	7.78	0.20	0.010	0.12	0.30
Outdoor Buildings	$100 \times 110 \times 7$	6.2×10^5	127	Avg	0.54	0.028	0.0066	0.099	0.14
				Std	1.46	0.045	0.0053	0.064	0.10
				Max	10.96	0.27	0.027	0.24	0.47

Planning Dynamically Feasible Trajectories for Quadrotors using Safe Flight Corridors in 3-D Complex Environments, Sikang Liu, RAL 2017

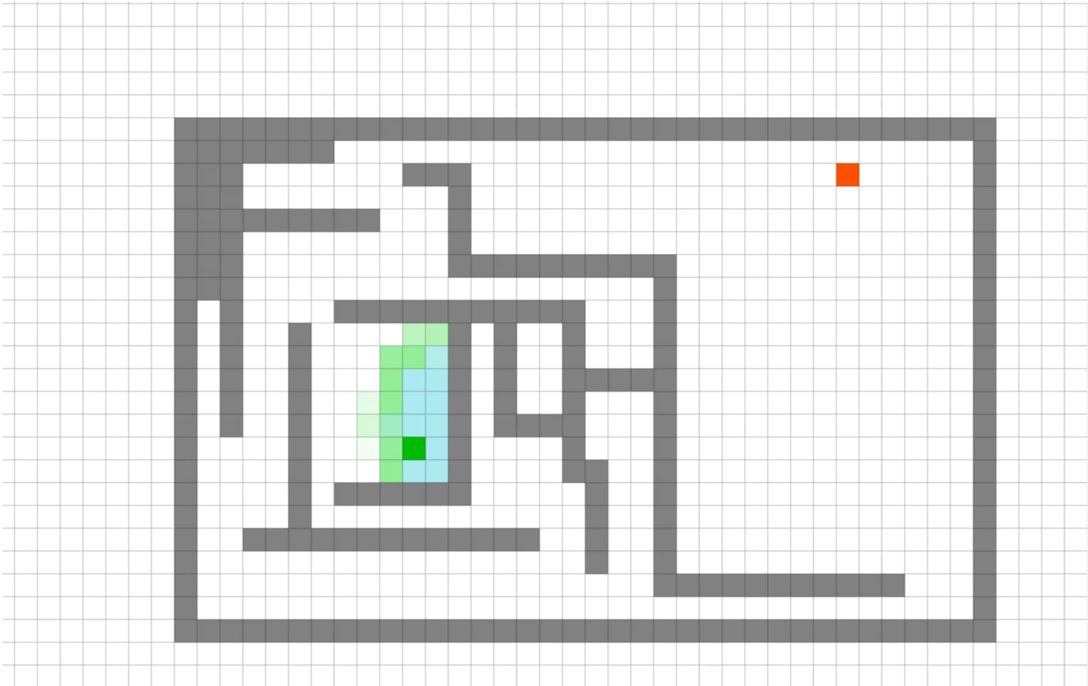
https://github.com/KumarRobotics/jps3d

JPS 总是更好?

迷宫般的环境



JPS

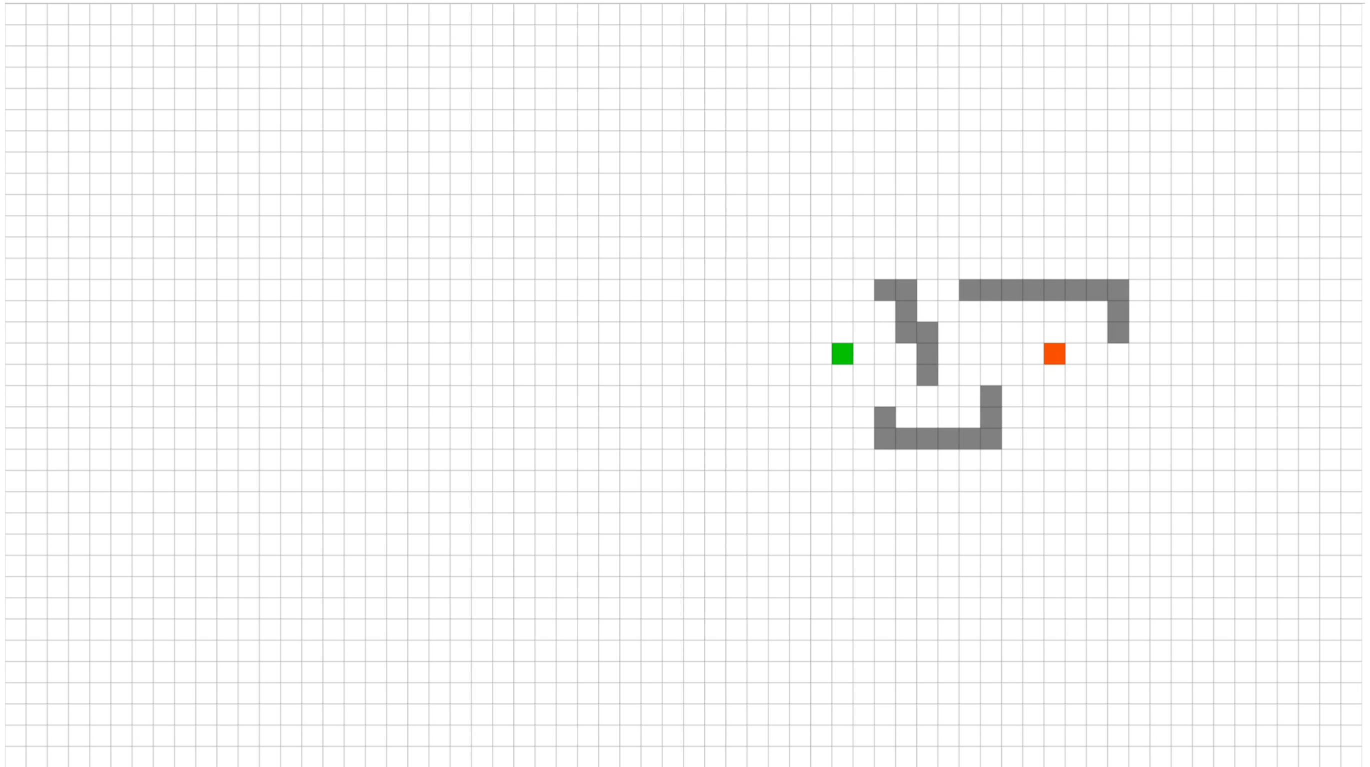


A*

Do more tests by yourself!

Thanks: http://qiao.github.io/PathFinding.js/visual/

JPS 总是更好?



- 这是一个说“不”的简单例子
- 这种情况通常发生在机器人导航中。
- FOV有限的机器人，但具有全局地图/大型局部地图。

- 大多数时候，尤其是在复杂的环境中，JPS更好，但远不是“总是”。为什么？
- JPS减少了Open List中的节点数量，但增加了状态查询的数量。
- 可以尝试JPS。
- JPS的限制：仅适用于统一网格地图。

实例介绍 1：Fast Marching in Distance Field

□ 前端路径搜索：获得时间索引最小路径

- **模拟波传播**

$$|\nabla T(x)| = \frac{1}{f(x)}$$

T 是到达时间的函数， x 是位置， $f(x)$ 表示不同位置 x的速度

- **速度场**

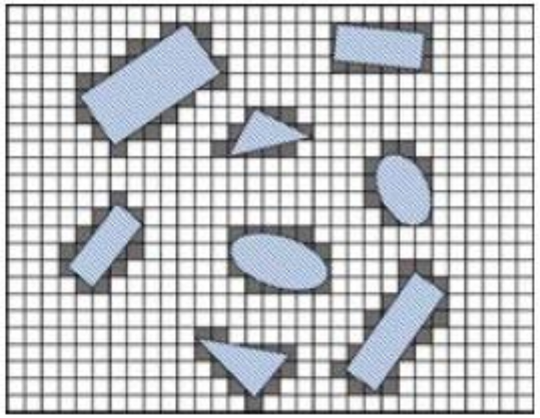
$$f(d) = \begin{cases} v_m \cdot (\tanh(d - e) + 1)/2, & 0 \le d \\ 0, & d < 0 \end{cases}$$

v_m 为最大速度, d 为 ESDF 中位置 x 处的距离值

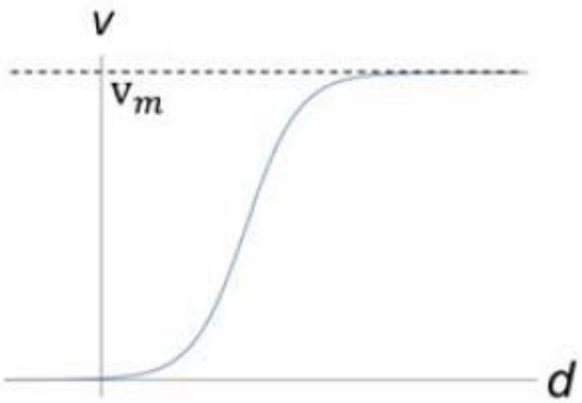
- **启发函数：到达目标的最短时间**

$$h(x) = d^*(x)/v_m$$

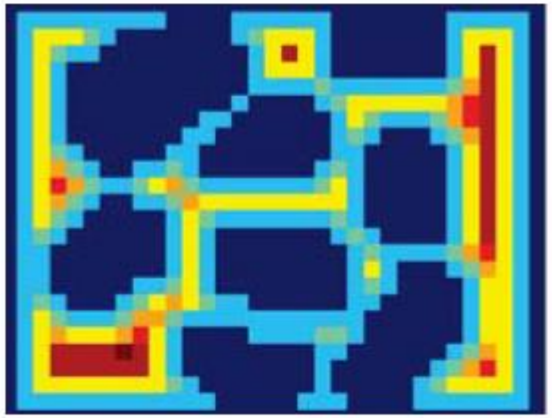
$d^*(x)$ 表示 x 到目标点的欧氏距离



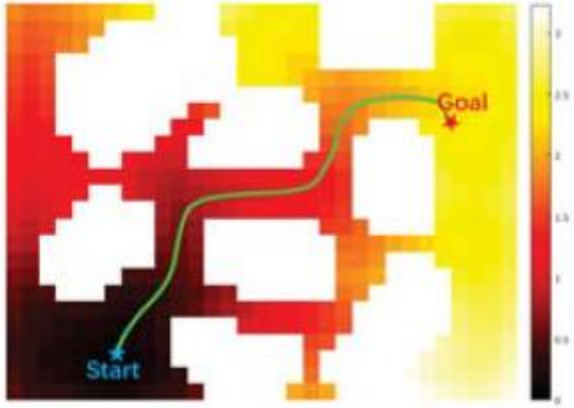
占据栅格地图



速度函数 $v=f(d)$

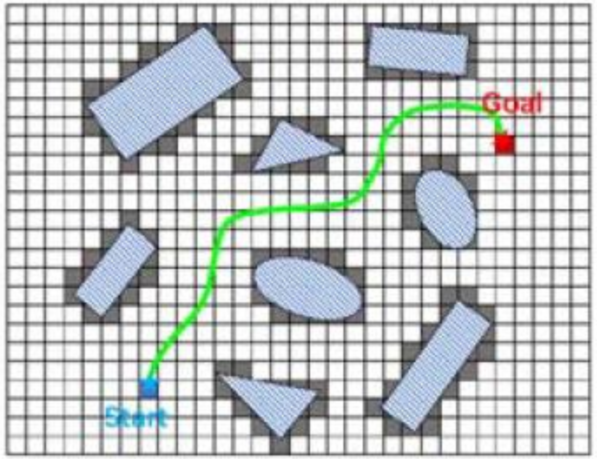


速度场

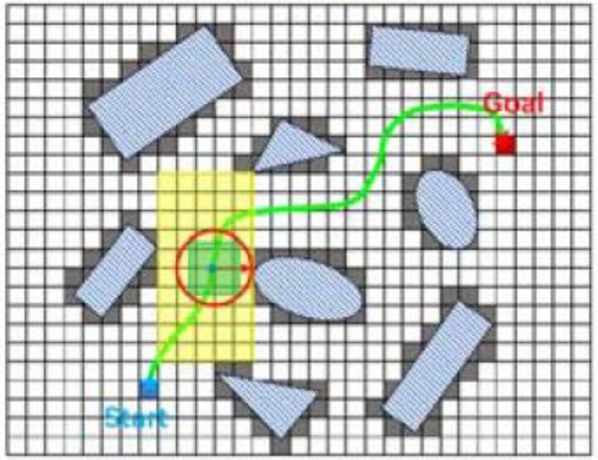


每个点到达的时间

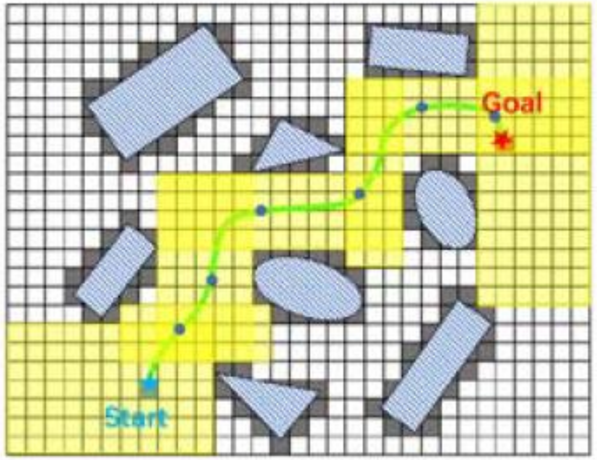
实例介绍 1：Fast Marching in Distance Field



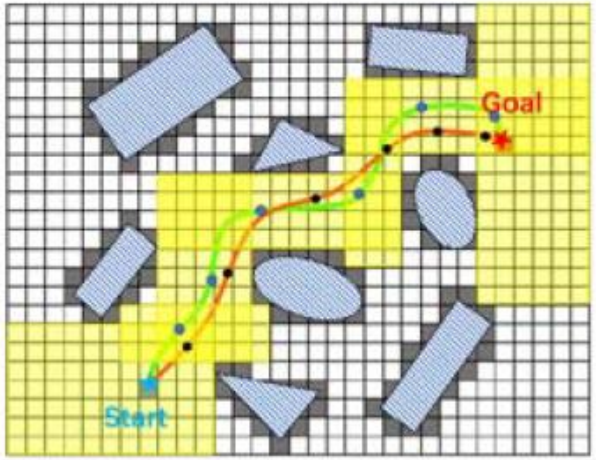
前端搜索的路径



飞行走廊初始化



飞行走廊修剪



优化后的轨迹

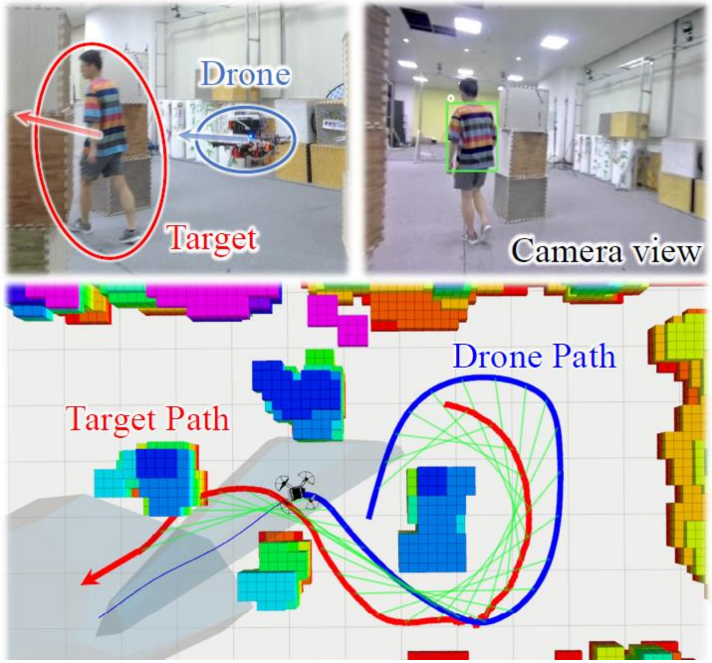
☐ 后端轨迹优化

轨迹表示：贝塞尔曲线

$$B_j(t) = c_j^0 b_n^0(t) + c_j^1 b_n^1(t) + \cdots + c_j^n b_n^n(t) = \sum_{i=0}^{n} c_j^i b_n^i(t),$$

路点约束 | 连续性约束 | 安全性约束 | 运动可行性约束

实例介绍 2：Elastic Tracker



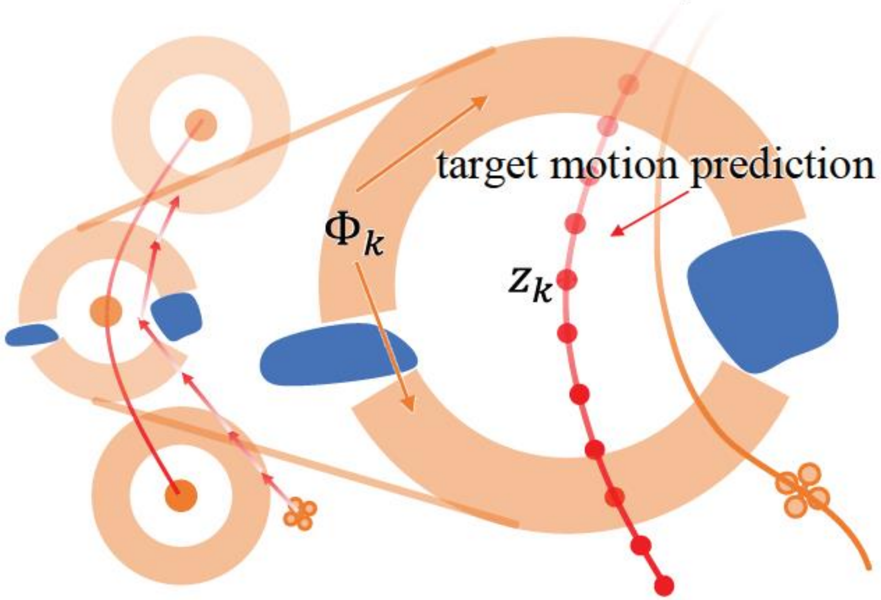
- 为每一个目标预测位置 z_k 定义一个无遮挡观察区域 Φ_k；
- 使用A*找到一条路径依次经过 $\Phi_1, \Phi_2, \ldots \Phi_k$ 区域

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

$$h^k(n) = \sqrt{\left\{d_{xy}^k(n) - d_d\right\}^2 + \left\{d_z^k(n)\right\}^2}$$

d_{xy}^k 和 d_z^k 是与目标预测位置点 z_k 的距离的水平和竖直方向分量;

d_d 是无人机和目标的期望距离;



□ 前端路径搜索：

- 对目标未来 T_p 时间内的轨迹 z_k 做出预测

$$\mathcal{T} = \left\{t_k \in [0, T_p] \middle| t_k \to z_k, 0 < k \le M_{\mathcal{T}}\right\}$$

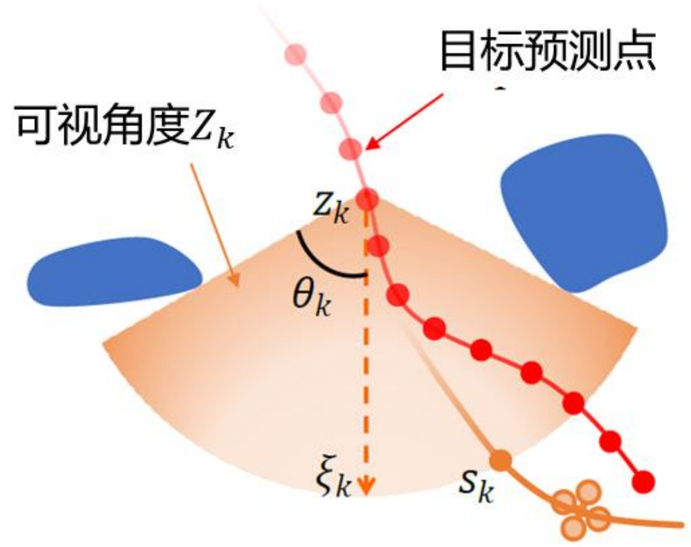
$M_{\mathcal{T}}$ 为预测的被跟踪目标的未来位置的个数，

$\mathcal{T}$ 为与目标未来每个位置相对应的时刻

实例介绍 2：Elastic Tracker

□ 后端轨迹优化：

- **可视性 Visibility 约束：**

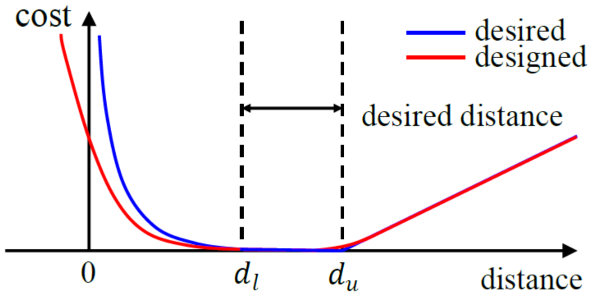


$$\mathcal{V}_k = \{ x \in \mathbb{R}^3 \mid \langle x - z_k, \xi_k \rangle \le \theta_k \}$$

$\mathcal{V}_k$：扇形可视区域；z_k：目标预测位置

s_k：可视点；ξ_k：扇形的角平分线向量

- **距离约束：**



$$\min_{p(t),T} \quad \mathcal{J}_o = \rho T + \int_0^T \| p^{(3)}(t) \|^2 \, dt$$

$$\text{s.t.} \quad p^{[s-1]}(0) = \bar{p}_o, p^{[s-1]}(T) = \bar{p}_f$$

$$\| p^{(1)}(t) \| \le v_m, \| p^{(2)}(t) \| \le a_m, \forall t \in [0, T]$$

$$p(t) \in \mathcal{P}, \forall t \in [0, T]$$

$$p(t_k) \in \mathcal{V}_k, \forall t \in [0, T]$$

$$d_l \le \| p(t_k) - z_k \| \le d_u, \forall t_k \in \mathcal{T}$$

$$T \ge T_p$$

Thanks for Listening!

Flying Autonomous Robotics（FAR）

FASTLAB

Field Autonomous System & compuTing