实验原理

A*算法

A*算法通过估价函数来计算每个节点的优先级。

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

- f(n)是节点 n 的综合优先级。当我们选择下一个要遍历的节点时，我们总会选取综合优先级最高（值最小）的节点。
- g(n)是节点 n 距离起点的代价。
- h(n)是节点 n 距离终点的预计代价，这也就是 A*算法的启发函数。

A*算法在运算过程中，每次从优先队列中选取 f(n)值最小（优先级最高）的节点作为下一个待遍历的节点。另外，A*算法使用两个集合来表示待遍历的节点，与已经遍历过的节点，这通常称之为 open_set 和 close_set。

A*算法的步骤

A*算法基本上与广度优先算法相同，但是在扩展出一个结点后，要计算它的估价函数，并根据估价函数对待扩展的结点排序，从而保证每次扩展的结点都是估价函数最小的结点。

A*算法的步骤如下：

1）建立一个队列，计算初始结点的估价函数 f，并将初始结点入队，设置队列头和尾指针。

2）取出队列头（队列头指针所指）的结点，如果该结点是目标结点，则输出路径，程序结束。否则对结点进行扩展。

3）检查扩展出的新结点是否与队列中的结点重复，若与不能再扩展的结点重复（位于队列头指针之前），则将它抛弃；若新结点与待扩展的结点重复（位于队列头指针之后），则比较两个结点的估价函数中 g 的大小，保留较小 g 值的结点。跳至第五步。

4）如果扩展出的新结点与队列中的结点不重复，则按照它的估价函数 f 大小将它插入队列中的头结点后待扩展结点的适当位置，使它们按从小到大的顺序排列，最后更新队列尾指针。

5）如果队列头的结点还可以扩展，直接返回第二步。否则将队列头指针指向下一结点，再返回第二步。

A*算法伪代码

```
* 初始化 open_set 和 close_set；
* 将起点加入 open_set 中，并设置优先级为 0（优先级最高）；
* 如果 open_set 不为空，则从 open_set 中选取优先级最高的节点 n：
   * 如果节点 n 为终点，则：
      * 从终点开始逐步追踪 parent 节点，一直达到起点；
      * 返回找到的结果路径，算法结束；
```

* 如果节点 n 不是终点，则：
  * 将节点 n 从 open_set 中删除，并加入 close_set 中；
  * 遍历节点 n 所有的邻近节点：
    * 如果邻近节点 m 在 close_set 中，则：
      * 跳过，选取下一个邻近节点
    * 如果邻近节点 m 也不在 open_set 中，则：
      * 设置节点 m 的 parent 为节点 n
      * 计算节点 m 的优先级
      * 将节点 m 加入 open_set 中

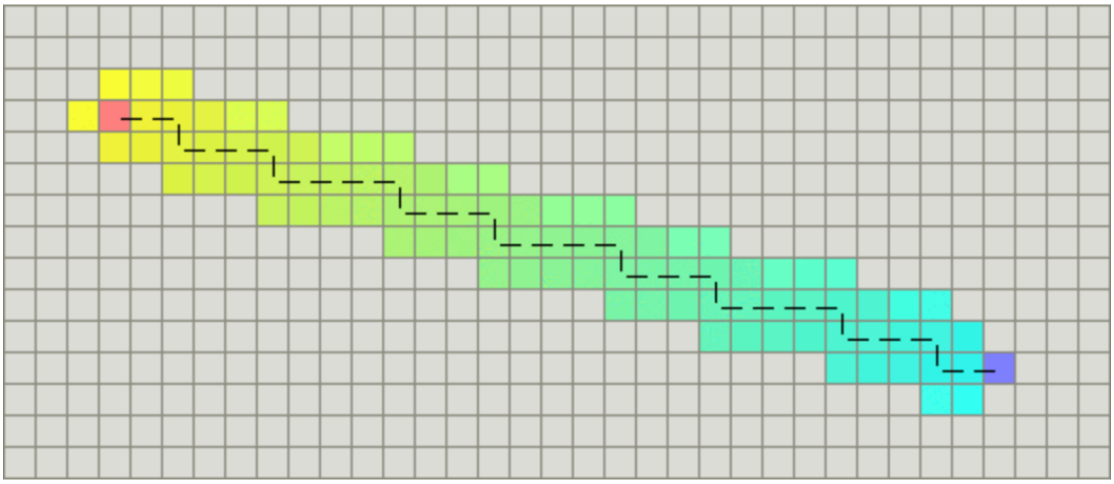
启发函数

上面已经提到，启发函数会影响 A*算法的行为。在极端情况下，当启发函数 h(n)始终为 0，则将由 g(n)决定节点的优先级，此时算法就退化成了 Dijkstra 算法。如果 h(n)始终小于等于节点 n 到终点的代价，则 A*算法保证一定能够找到最短路径。但是当 h(n)的值越小，算法将遍历越多的节点，也就导致算法越慢。如果 h(n)完全等于节点 n 到终点的代价，则 A*算法将找到最佳路径，并且速度很快。可惜的是，并非所有场景下都能做到这一点。因为在没有达到终点之前，我们很难确切算出距离终点还有多远。如果 h(n)的值比节点 n 到终点的代价要大，则 A*算法不能保证找到最短路径，不过此时会很快。在另外一个极端情况下，如果 h(n)相较于 g(n)大很多，则此时只有 h(n)产生效果，这也就变成了最佳优先搜索。由上面这些信息我们可以知道，通过调节启发函数我们可以控制算法的速度和精确度。因为在一些情况，我们可能未必需要最短路径，而是希望能够尽快找到一个路径即可。这也是 A*算法比较灵活的地方。

对于网格形式的图，有以下这些启发函数可以使用：

- 如果图形中只允许朝上下左右四个方向移动，则可以使用**曼哈顿距离**（Manhattan distance）。
- 如果图形中允许朝八个方向移动，则可以使用**对角距离**。
- 如果图形中允许朝任何方向移动，则可以使用**欧几里得距离**（Euclidean distance）。

1. **曼哈顿距离**：如果图形中只允许朝上下左右四个方向移动，则启发函数可以使用曼哈顿距离，它的计算方法如下图所示：



计算曼哈顿距离的函数如下，这里的 D 是指两个相邻节点之间的移动代价，通常是一个固定的常数。

```
function heuristic(node) =
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * (dx + dy)
```

2. **对角距离**： 如果图形中允许斜着朝邻近的节点移动，则启发函数可以使用对角距离。它的计算方法如下：

计算对角距离的函数如下，这里的 D2 指的是两个斜着相邻节点之间的移动代价。如果所有节点都正方形，则其值就是$\sqrt{2}$*D。

```
function heuristic(node) =
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)
```

3. **欧几里得距离**

如果图形中允许朝任意方向移动，则可以使用欧几里得距离。欧几里得距离是指两个节点之间的直线距离，$\sqrt{(x_2-x_1)^2-(y_2-y_1)^2}$

```
function heuristic(node) =
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * sqrt(dx * dx + dy * dy)
```

N 拼图问题

N 拼图或滑动拼图是一种流行的拼图，由 N 个拼图块组成，其中 N 可以是 8、15、24 等等。在我们的示例中，N = 8。拼图分为 sqrt(N+1) 行和 sqrt(N+1) 列。例如，15 拼图将有 4 行和 4 列，而 8 拼图将有 3 行和 3 列。拼图由 N 个拼图块和一个可以移动拼图块的空白处组成。拼图的起始和目标配置（也称为状态）已提供。拼图可以通过在单个空白处逐个移动拼图块来解决，从而实现目标配置。

八数码问题: 在 3×3 的九宫格棋盘上，摆有 8 个刻有 1～8 数码的将牌。棋盘中有一个空格，允许紧邻空格的某一将牌可以移到空格中，这样通过平移将牌可以将某一将牌

布局变换为另一布局。针对给定的一种初始布局或结构（目标状态），问如何移动将牌，实现从初始状态到目标状态的转变。

Initial State

1	2	3
8		4
7	6	5

Goal State

2	8	1
	4	3
7	6	5

解答谜题的规则。

我们可以想象移动空白处的瓷砖，而不是移动空白处的瓷砖，基本上就是将瓷砖与空白处交换。空白处只能向四个方向移动，即：

1. 向上
2. 向下
3. 向右 或
4. 向左

空位不能对角移动，并且每次只能迈出一步（即每次将空位移动一个位置）。

八数码问题的 A*算法的估价函数

估价函数中，主要是计算 h，对于不同的问题，h 有不同的含义。八数码问题的一个状态实际上是数字 0~8 的一个排列，用一个数组 p[9]来存储它，数组中每个元素的下标，就是该数在排列中的位置。例如，在一个状态中，p[3]=7，则数字 7 的位置是 3。如果目标状态数字 3 的位置是 8，那么数字 7 对目标状态的偏移距离就是 3，因为它要移动 3 步才可以回到目标状态的位置。

八数码问题中，每个数字可以有 9 个不同的位置，因此，在任意状态中的每个数字和目标状态中同一数字的相对距离就有 9*9 种，可以先将这些相对距离算出来，用一个矩阵存储，这样只要知道两个状态中同一个数字的位置，就可查出它们的相对距离，也就是该数字的偏移距离：

	0	1	2	3	4	5	6	7	8
0	0	1	2	1	2	3	2	3	4
1	1	0	1	2	1	2	3	2	3
2	2	1	0	3	2	1	4	3	2
3	1	2	3	0	1	2	1	2	3
4	2	1	2	1	0	1	2	1	2
5	3	2	1	2	1	0	3	2	1

6　2 3 4 1 2 3 0 1 2

7　3 2 3 2 1 2 1 0 1

8　4 3 2 3 2 1 2 1 0

例如在一个状态中，数字 8 的位置是 3，在另一状态中位置是 7，那么从矩阵的 3 行 7 列可找到 2，它就是 8 在两个状态中的偏移距离。估价函数中的 h 就是全体数字偏移距离之和。显然，要计算两个不同状态中同一数字的偏移距离，需要知道该数字在每个状态中的位置，这就要对数组 p[9]进行扫描。由于状态发生变化，个数字的位置也要变化，所以每次计算 h 都沿线扫描数组，以确定每个数字在数组中的位置。为了简化计算，这里用一个数组存储状态中各个数字的位置，并让它在状态改变时随着变化，这样就不必在每次计算 h 时，再去扫描状态数组。