**电子科技大学**

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

**课程结业论文**



论文题目 A\*算法在寻路中的运用

专 业 信息安全

学 号 2017060201009

作者姓名 蒋程

指导教师

**A \* 算法在寻路中的运用**

蒋程 20170602009

平时做的OJ并不多，寻路的题目也只是仅仅有所了解。所以早期对于迷宫寻路的相关技术栈仅局限在暴力搜索的层面。而且DFS或者BFS也没有较多的使用，只是说知道该这么做罢了。后来在写一个迷宫小游戏的时候，无意间听说了A \* (A star) 寻路算法，并且在游戏里有广泛的使用，故这里对其进行一下简单的研究。

**一、什么是A\* 算法**

**A\*搜索算法**，俗称**A星算法**。这是一种在图形平面上，有多个节点的路径，求出最低通过成本的算法。常用于游戏中的NPC的移动计算，或网络游戏的BOT的移动计算上。（摘于wiki）

该算法综合了[BFS](https://zh.wikipedia.org/wiki/%E5%B9%BF%E5%BA%A6%E4%BC%98%E5%85%88%E6%90%9C%E7%B4%A2)和[Dijkstra算法](https://zh.wikipedia.org/wiki/Dijkstra%E7%AE%97%E6%B3%95)的优点：在进行启发式搜索提高算法效率的同时，可以保证找到一条最优路径（基于评估函数）。

**二、A\* 算法的原理**

A\* 算法和其他的图搜索算法一样，用于寻找图中的最短通路。但是它是[启发式](https://zh.wikipedia.org/zh/%E5%90%AF%E5%8F%91%E5%BC%8F%E6%90%9C%E7%B4%A2)的，就像贪心算法一样，它每次行走后会评估此点到终点的代价，从而实现快速的搜索。

**以下为核心公式**：

* F = G +H
* F：方块的总的移动代价
* G：开始点到当前方块的移动代价
* H：当前方块到结束点的预估移动代价

**G值的计算：**

比如我们在某一点，那么将会有八个方向可以移动。当往上下左右移动时，代价为10；而当往左上、左下、右上、右下、移动的时候，代价为14（10√2），这个是计算G值的基本公式。

当然如果是在游戏的运用中，可能会考虑地形等因素，所以G的值会比实际上更加复杂。比如平地和丘陵，那么显然两者的代价是不同的，但是这里我们只讨论平面下的寻路，所以所有的格子代价均为移动代价。

* 当前节点的G值 = 父节点的G值 + 父节点到当前节点的移动代价

**H值的预估：**

这里有一个问题，就是如果我们只知道当前的点和终点，但是不知道这两点之间的路径，那么如何精确地计算两者间的H值呢？显然是不可能的，所以我们只能对H的值进行预估。这也是A\*被称为启发式算法的原因，即在移动的过程中预估H的值从而降低周弯路的可能性，以便找到更短的路径。而相对的，BFS就是穷举处所有的可能性，然后从中挑选处一条最短的路径。

常用的H预估方法：

* [曼哈顿距离](https://zh.wikipedia.org/wiki/%E6%9B%BC%E5%93%88%E9%A0%93%E8%B7%9D%E9%9B%A2)
* [欧几里得距离](https://zh.wikipedia.org/wiki/%E6%AC%A7%E5%87%A0%E9%87%8C%E5%BE%97%E8%B7%9D%E7%A6%BB)
* 对角线距离

**三、A\* 的实现步骤**

**需要用到的名词**：

* 开放列表：记录没有检查过的节点
* 关闭列表：记录已经检查过的节点

**步骤**：

1. 把开始节点A放入开放列表
2. 找出从A点可达的相邻节点，并放入开放列表
3. 把A点从开放列表移除，并加入到关闭列表（因为A已经被检查过了）

**下一步计算如何移动**：

1. 计算A周围所有节点的F值
2. 找出F值最小的节点p（有多个的话，就随机选择一个）
3. 将p从开始列表中删除，并放到关闭列表里面
4. 检查P点所有相邻且可达的节点，如果这些节点不在开启列表中，便将他们加入进去并计算他们的F
   * 设置新加入的节点的最小F值节点为其父节点
   * 如果某个相邻节点D已经在开启列表，检查如果用新的路径(就是前面提到的，经过P的路径)到达它，G值是否会更低, 如果更低, 那就把它的父节点改前面选到的节点，然后重新计算它的F值和G值。
   * 如果新的G值比较高, 就说明经过P再到达此点是不明智的，这时什么的都不做。

**重复搜索判定工作：**

* 直到终点节点加入到开启列表中的时候，搜索完成。此时得到的路径为一条通路但是不一定是最好的。
* 当终点节点加入到关闭列表的时候，搜索完成。此时得到的路径为最佳路径，但是计算量相对更大。
* 当没有找到目标节点（终点），并且开启列表为空的时候，那没有通路，此时搜索结束。

**四、相关代码：**

这里贴出迷宫小游戏的A\* 寻路的核心算法

*/\*\**

*\* 关键的寻路函数*

*\*/*

**override** **fun** findPath(callBack: PathFindCallBack) {

mHandler.post {

**var** endNode: PathNode? = **null**

**val** startNode = PathNode(startY, startX)

**val** openList = ArrayList<PathNode>()

**val** closeList = ArrayList<PathNode>()

openList.add(startNode)

**var** tryCount = 0

**val** startTime = System.currentTimeMillis()

*//A\*的思想和BFS相似，故这里需要一个列表来放已经遍历了的节点，这里就是openList*

**while** (openList.isNotEmpty()) {

tryCount++

**val** curMinFNode = getMinFPathNode(openList)

openList.remove(curMinFNode)

closeList.add(curMinFNode)

**if** (curMinFNode.y == endY && curMinFNode.x == endX) {

endNode = curMinFNode

endNode.count = tryCount

endNode.timeCost = System.currentTimeMillis() - startTime

**break**

} **else** {

handleChildNode(curMinFNode, curMinFNode.y, curMinFNode.x + 1, openList)

handleChildNode(curMinFNode, curMinFNode.y + 1, curMinFNode.x, openList)

handleChildNode(curMinFNode, curMinFNode.y, curMinFNode.x - 1, openList)

handleChildNode(curMinFNode, curMinFNode.y - 1, curMinFNode.x, openList)

}

}

callBack.notify(endNode, STYLE\_A\_STAR)

}

}

*/\*\**

*\* 遍历四周的节点，并计算F值，并将其添加到openList里面*

*\*/*

**private** **fun** handleChildNode(curMinFNode: PathNode, y: Int, x: Int, openList: ArrayList<PathNode>) {

**val** child: PathNode

**if** (isPath(y, x)) {

child = PathNode(y, x)

**if** (!isExistList(child, closeRouteMap) && !isExistList(child, openRouteMap)) {

child.F = getF(y, x)

child.parent = curMinFNode

openList.add(child)

openRouteMap[y][x] = **true**

}

}

}

**private** **fun** getF(y: Int, x: Int): Int {

**val** g = Math.abs(startX - y) + Math.abs(startX - x)

**val** h = Math.abs(endX - y) + Math.abs(endY - x)

**return** g + h

}

*/\*\**

*\* 找出F值最小的节点P*

*\*/*

**private** **fun** getMinFPathNode(openList: ArrayList<PathNode>): PathNode {

**var** pos = 0

**var** minF = openList[pos].F

**for** (i **in** openList.indices) {

**if** (openList[i].F < minF) {

minF = openList[i].F

pos = i

}

}

**return** openList[pos]

}

**五、测试**

由于迷宫使用的Prim算法生成的，写的不是很好，随机性比较差，所以只能较为勉强的反应测试结果。

**20\*20的小迷宫:**

* **DFS 耗时**

可以看见DFS算是比较慢的了。当然这里为了测试的效果，DFS使用得比较粗暴，并没有怎么优化。



* **BFS耗时**

BFS相对于DFS就要快了不少。



* **A\* 耗时**

A\* 比BFS 甚至比BFS还要快一点，显然启发式函数起到了一定的作用。

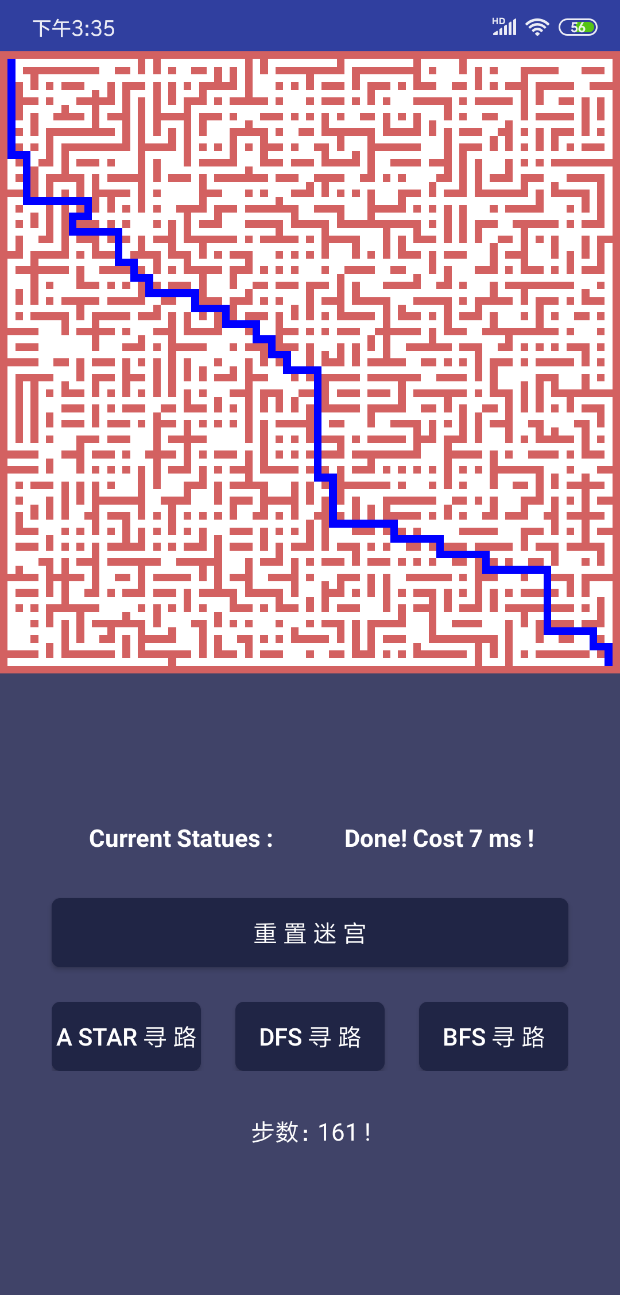


**80 \* 80 的迷宫**

* **DFS直接炸了，故这里放BFS和A\* 的图**
* **BFS耗时**

嗯？？？居然比20 \* 20 还快，应该是多次测试手机频率上去了的原因。

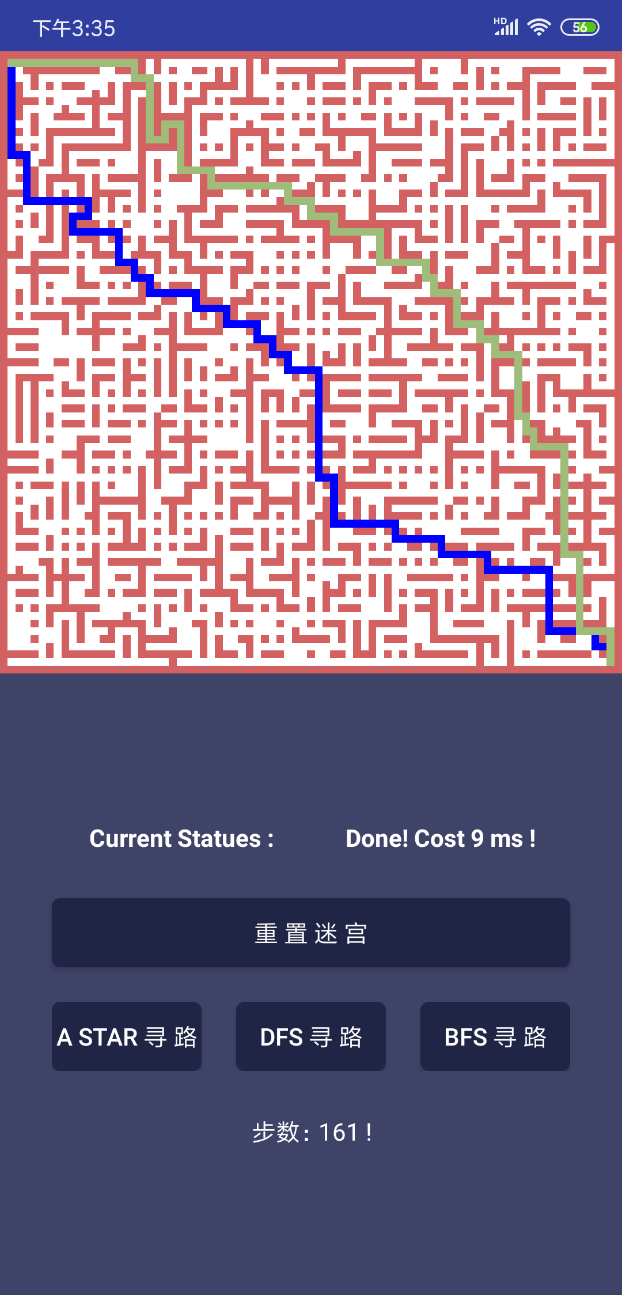
不过确实这种迷宫对于BFS是有优势的(基本不存在障碍和弯路，一条斜线走到底)



* **A \* 耗时**

**A\* 比BFS稍慢一点。**

不过也在情理之中，因为地图生成得比较烂的原因，A\*的启发式算法带来的时间上的减少并没有足够多的收益。外加由于每一次需要额外寻找F值最小的节点，所以需要的时间更长（后面测试了一下优先队列，结果发现时间和BFS差不多，看迷宫生成算法果然写得不太好）。



**六、A\* 算法的相关优化**

A\* 的算法并不难实现，但是在实际使用中的运用远不止上述代码般简单，往往还需要对细节进行优化，以提升算法速度，降低时间复杂度。

* 对于存放节点的开放列表openList，可以使用二叉树构建小顶堆来作为一个优先队列，这样堆顶上放的一直是F值最小的节点，避免了每次在寻找F值最小的节点时都去遍历整个列表。
* 可以使用一个二维的布尔值数组来记录哪些节点已经被加入到了开放列表了，从而降低去遍历开放列表验证节点是否被遍历的时间
* 采用布兰森汉姆算法预先判断两点是否可以直接通行，可通行就直接返回两点的直线路径，不可直接通行再采用A星算法寻路，提高寻路效率