无人驾驶汽车系统入门（十六）——最短路径搜索之A*算法

原创 置顶 AdamShan 2018-04-14 22:35:03 12866 收藏 77 版权

分类专栏： 无人驾驶汽车专题 无人驾驶汽车系统入门 文章标签： 无人车 无人驾驶 路径规划 A*算法 最短路径搜索

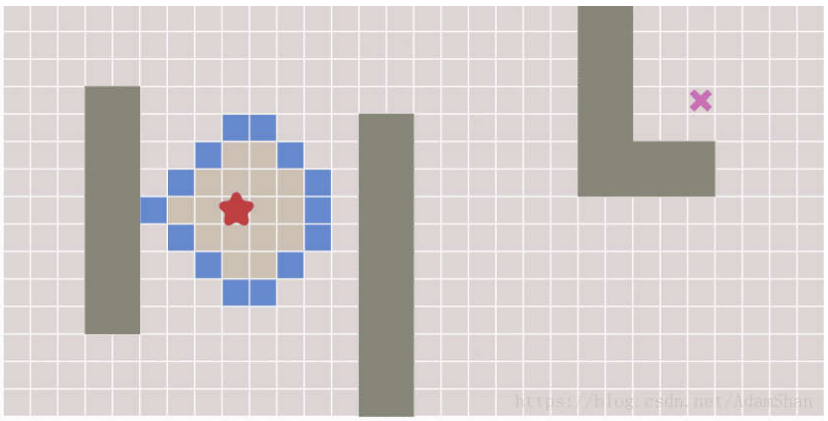
无人驾驶汽车系统入门（十六）——最短路径搜索之A*算法

> 路线规划中一个很核心的问题即最短路径的搜索，说到最短路径的搜索我们就不得不提A*算法，虽然原始的A*算法属于离散路径搜索算法（我们的世界是连续的），但是其使用启发式搜索函数的理念却影响着我们后面会介绍的连续路径搜索算法，所以在介绍连续路径搜索算法之前，理解基本的A*算法是很有必要的，本节我们从广度优先算法出发，一步步改良算法直到引出A*算法。

最短路径搜索是通过算法找到一张图从起点（start）到终点（goal）之间的最短路径（path），为了简化，我们这里使用方格图（该图可以简单地用二维数组来表示），如下动图所示，其中  代表起点，  代表终点。

广度优先算法（Breadth-First-Search, BFS）

广度优先算法实际上已经能够找到最短路径，BFS通过一种从起点开始不断**扩散**的方式来遍历整个图。可以证明，只要从起点开始的扩散过程**能够遍历到终点**，那么起点和终点之间**一定是连通的**，因此他们之间至少存在一条路径，而由于BFS从中心开始**呈放射状扩散**的特点，它所找到的这一条路径就是最短路径，下图演示了BFS的扩散过程：



其中**由全部蓝色方块组成的队列叫做frontier**（参考下面的BFS代码）

然而，BFS搜索最短路径实在太慢了，为了提高BFS的搜索效率，接下来我们从BFS一步步改良到A*算法（其中的代码主要用于表达思路，距离实际运行还缺部分support code）

BFS代码：

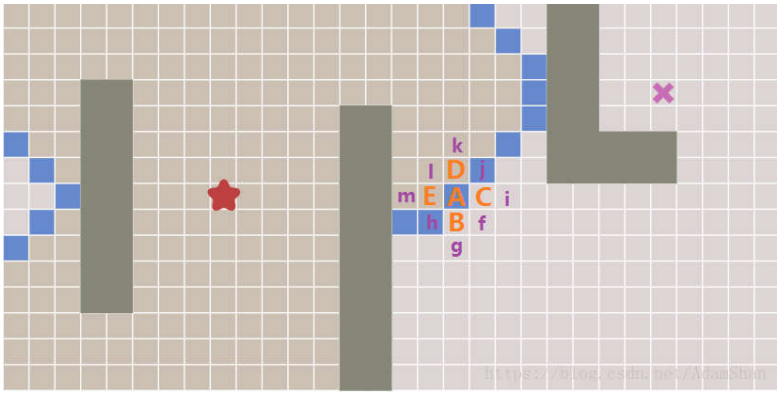
```
frontier = Queue()
frontier.put(start)
visited = {}
visited[start] = True

while not frontier.empty():
   current = frontier.get()
   for next in graph.neighbors(current):
      if next not in visited:
         frontier.put(next)
         visited[next] = True
```

所涉及到的主要数据结构:

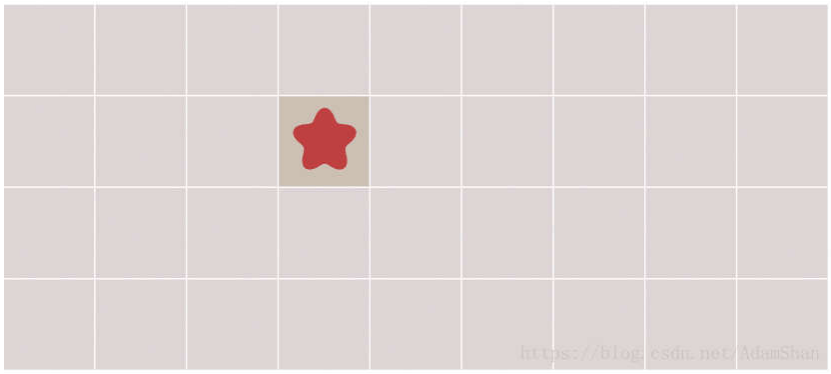
1. **graph**，要找到一张图中两点之间的path，我们需要一个最基本的graph数据结构。在本文中，我们只需要得到某一点的邻近点，在这里我们的代码调用graph.neighbors(current)，**该函数返回点current周围的所有邻近点构成的一个列表，由for循环可以遍历这个列表。**

2. **queue** 为了解释用队列的原因，请看下图，假设此时frontier为空，current当前是A点，它的neighbors将返回B、C、D、E四个点，在将这4个点都添加到frontier当中以后，下一轮while循环，frontier.get()将返回B点（根据FIFO原则，B点最早入队，应当最早出队），此时调用neighbors，返回A、f、g、h四个点，除了A点，其他3个点又被添加到frontier当中去。再到下一轮循环，此时frontier当中有C、D、f、g、h这几个点，由于队列的FIFO原则，frontier.get()将返回C点。**这样就保证了整个扩散过程是由近到远，由内而外的，这也是广度优先搜索的原则。**可以看到，frontier.get()从队列中取出一个元素（该元素将从队列中被删除）。而frontier.put()将current的邻近点又添加进去，整个过程不断重复，直到图中的所有点都被遍历一遍。

3. **visited列表**：接着上面的讨论，graph.neighbors(A)将返回B、C、D、E 4个点，随后这4个点被添加到frontier当中，下一轮graph.neighbors(B)将返回A、h、f、g四个点，而加入此时A再被添加到frontier当中就导致遍历陷入死循环，为了避免这种状况出现，我们需要将已经遍历过了的点添加到visited列表当中，之后在将点放入frontier之前，首先判断该点是否已经在visited列表当中。



下面的动图可以看到整个广度优先算法运行的详细过程：



其中绿色方框代表neighbors所返回的current点的邻近点，蓝色方块代表当前frontier队列中的点，由于队列先进先出(FIFO)的特点，**越早加入队列的点的序号（图中方块的数字）越小，因此越早被while not frontier.empty()循环中的 current = frontier.get()遍历到。**

找到路线

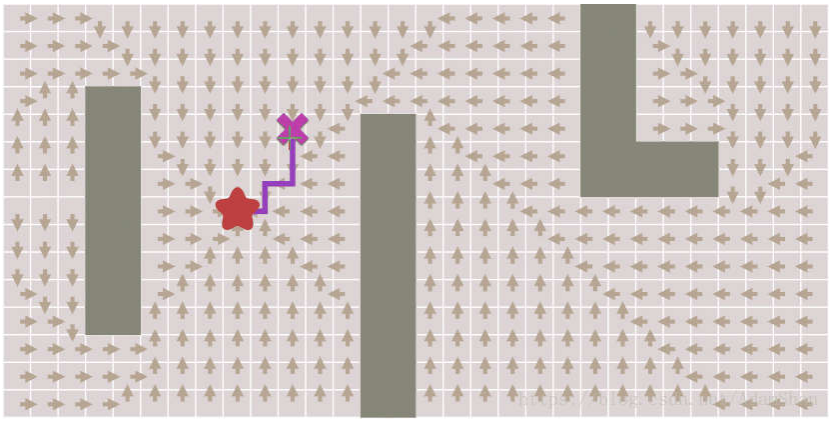
现在我们的算法能够对所有的点进行遍历，也就意味着**一定能够扩散到目标点**，因此从开始到终止点的路径是存在的。为了生成这一路径，我们需要对扩散的过程进行记录，保存每一个点的来源（该点由哪一个点扩散而来），最后通过这些记录进行回溯即可得出完整的路径。将visited数组改为came_from。比如从A扩散到B，则came_from[B]=A。有了这样的线索，我们就能够从终点回溯到起点。

```
frontier = Queue()
frontier.put()
came_from = {}
came_from[start] = None

while not frontier.empty():
   current = frontier.get()
   for next in graph.neighbors(current):
      if next not in came_from:
         frontier.put(next)
         came_from[next] = current
```

下面的算法通过came_from数组来生成path的算法，其中goal表示终点。

```
current = goal
path = []
while current != start
   path.append(current)
   current = came_from[current]
path.append(start)
path.reverse()
```

效果如下图，其中每个方块上的箭头指向它的来源点，注意观察随着终点的变化，如何通过箭头的回溯得到完整路径。



提前结束

目前我们的算法会遍历整个图的每一个点，回想我们最初的目标：找到从起点到终点之间的路径，只需要遍历到终点即可。为了减少无用功，我们设置终止条件：一旦遍历到goal以后，通过break让整个算法停止。

```
frontier = Queue()
frontier.put(start)
came_from = {}
came_from[start] = None

while not frontier.empty():
   current = frontier.get()

   if current == goal
      break

   for next in graph.neighbors(current):
      if next not in came_from:
         frontier.put(next)
         came_from[next] = current
```

扩散的方向性

到了这一步，整个BFS的思路已经完整了，但目前它的遍历方法仍然没有明确的目标，扩散朝着所有方向前进，十分蠢笨的遍历了以起点为中心的周围每一个方块，这不就是穷举吗？

在上面的算法运行过程中，frontier队列内部一般都会保持几个点（每次frontier.get()拿出来一个点，frontier.put()又放回去一个点）。而frontier.get()返回这些点中的哪一个决定了我们的**扩散向着哪一个方向进行**。之前这个函数只是根据queue默认的FIFO原则来进行，因此产生了**辐射状的扩散方式**，上文在介绍frontier的时候已经解释过这一点。

我们想到，能否让我们的扩散过程有**侧重方向**地进行呢？ 注意，其实我们始终清楚地知道起始点和终止点的坐标，**却浪费了这条有价值的信息**。在frontier.get()返回了frontier几个点当中的一个，为了“有方向”地进行扩散，我们让frontier返回那个看似距离终点最近的点。由于我们使用的是方格图，每个点都有(x,y)坐标，通过两点的(x,y)坐标就可以计算它们之间的距离，这里采用曼哈顿距离算法：

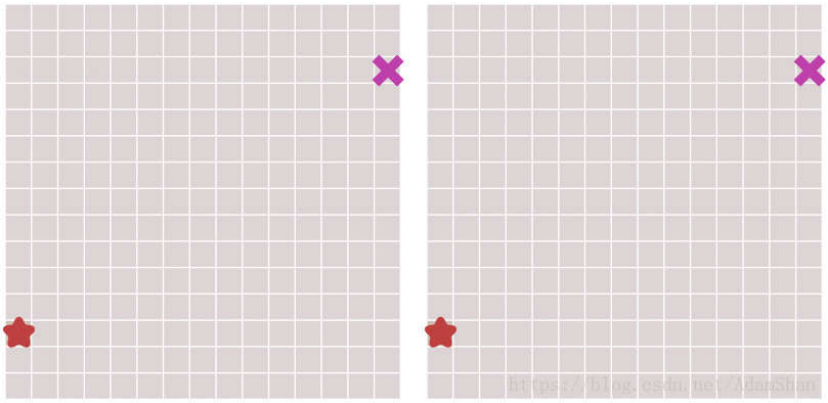
```
def heuristic(a, b):
   # 这种距离叫做曼哈顿距离（Manhattan）
   return abs(a.x - b.x) + abs(a.y - b.y)
```

启发式的搜索

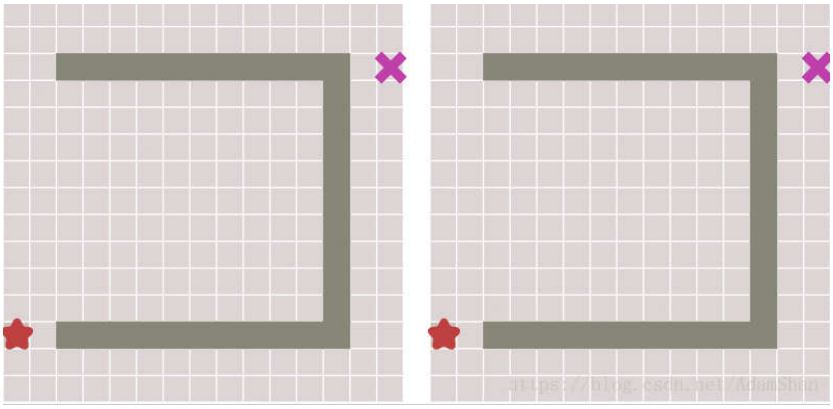
接下来我们改变原来队列的FIFO模式，给不同的点加入优先级，使用PriorityQueue，其中frontier.put(next,priority)的第二个参数越小，该点的优先级越高，可以知道，**距离终点的曼哈顿距离越小的点，会越早从frontier.get()当中返回**。

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}
came_from[start] = None

while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      if next not in came_from:
         priority = heuristic(goal, next)
         frontier.put(next, priority)
         came_from[next] = current
```

下面就是启发式搜索的效果，unbelievable!



到这里是不是游戏就结束了？ 这不就搞定啦，还要A*做什么？ 且慢，请看下图中出现的新问题：



可以看到，虽然启发式搜索比BFS更快得出结果，但它所生成的路径并不是最优的，其中出现了一些绕弯路的状况。

从起点到终点总会存在多条路径，之前我们通过visited（后来用came_from） 数组来避免重复遍历同一个点，然而这导致了先入为主地**将最早遍历路径当成最短路径**。为了兼顾效率和最短路径，我们来看Dijkstra算法，这种算法的主要思想是**从多条路径中选择最短的那一条**：我们记录每个点从起点遍历到它所花费的当前最少长度，当我们通过另外一条路径再次遍历到这个点的时候，由于该点已经被遍历过了（已经加入了came_from数组），我们此时不再直接跳过该点，而是比较一下目前的路径是否比该点最初遍历的路径花费更少，如果是这样，那就将该点纳入到新的路径当中去（修改该点在came_from中的值）。下面的代码可以看到这种变化，我们通过维护cost_so_far记录每个点到起点的当前最短路径花费（长度），并将这里的cost作为该点在PriorityQueue中的优先级。

Dijkstra:

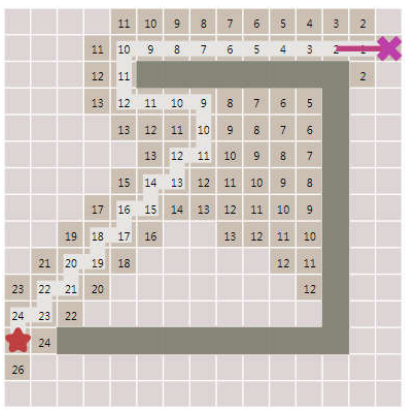
```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}
cost_so_far = {}
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + 1
      if next not in cost_so_far or new_cost < cost_so_far[next]:
         cost_so_far[next] = new_cost
         priority = new_cost
         frontier.put(next, priority)
         came_from[next] = current
```

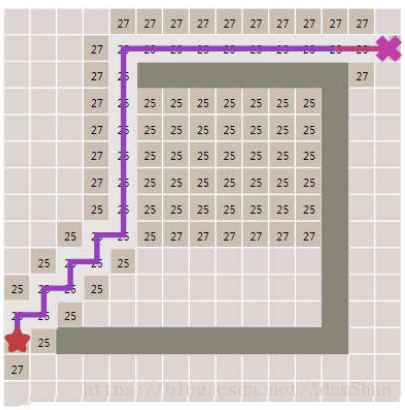
一方面，我们需要算法有方向地进行扩散（启发式），另一方面我们需要得到尽可能最短的路径，因此A*就诞生了， 它结合了Dijkstra和启发式算法的优点，以从起点到该点的距离加上该点到终点的估计距离之和作为该点在Queue中的优先级，下面是A*算法的代码：

A*算法

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}
cost_so_far = {}
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + 1
      if next not in cost_so_far or new_cost < cost_so_far[next]:
         cost_so_far[next] = new_cost
         priority = new_cost + heuristic(goal, next)
         frontier.put(next, priority)
         came_from[next] = current
```

下面的图展现了A*算法如何克服了启发式搜索遇到的问题：





这种A*算法用公式表示为： f(n)=g(n)+h(n),

也就指代这句代码：

```
1  priority = new_cost + heuristic(goal, next)
```

其中， f(n) 指当前n点的总代价(也就是priority，总代价越低，priority越小，优先级越高，越早被frontier.get()遍历到)，g(n) 指new_cost，从起点到n点已知的代价，h(n) 是从n点到终点所需代价的估算。

AdamShan 专家 图像处理 深度学习 TensorFlow

奔驰高级自动驾驶扫地僧，谷歌认证机器学习专家，兰州大学无人驾驶团队创始人，主攻深度学习，无人驾驶汽车方向，著有《无人驾驶原理与实践》一书。

优质评论可以帮助作者获得更高权重 评论

Liousvious： 我认为该博文应该注明参考文献或资料来源，毕竟尊重原作者的知识产权是一件很严肃且重要的事！ 2 年前 回复 11

hacker_G： 现在的很多创作者明明是绝大部分都是参考的,却不标注自己参考的哪些文章,而这些二次创作的作品的作者有些时候会省掉原文中的一些认为不重要的部分,而有些时候这些"不重要部分"却往往是我们这些读者所需要的,所以希望博主之后能够重视一下.另外,我看了一下您的其他文章,同样是很多不表明原文出处,不知道您是以怎样的心态去创作这些文章的,希望您今后能够把参考的文章也放上来,我们共同学习,谢谢 1 年前 回复 1

weixin_42621524： 博主你好，请问你用的是哪个软件编写的代码 4 月前 回复

西门吹瓶哥： 博主你好，if next not in cost_so_far or new_cost < cost_so_far[next]: 这里应该是if next not in came_from 这个列表吧? 3 年前 回复

uncle 。。： A*算法是怎么克服的 我还是没看懂。。 3 年前 回复

galaxy_v1 回复： 原文：https://www.redblobgames.com/pathfinding/a-star/introduction.html 3 年前 回复 3

相关推荐