基于图论的路径规划算法

计科 171 白天浩 10117103

摘　要: 路径规划是一个非常有现实意义的算法问题，在诸多领域都有广泛应用。路径规划算法对时间和空间复杂度的要求非常严格，因为移动者必须在移动之前提前知晓下一步的路径。本文考察了这种算法的谱系，从最经典古老的算法到最近非常高效的算法。路劲规划算法大致可以分为两类：实时的和非实时的，区别在于是否在踏出第一步之前就计算完全部路径。本文分析各类算法的优缺点，最终得出 A*算法是一种比较优秀的算法这一结论。而后介绍了 A*算法实时的和非实时的改进版本。最后通过亲自写代码实验进一步加强了对这一系列算法的了解，通过实践避免了纸上谈兵的局限性。
关键词: 路径规划;图论;寻路算法;A*算法;

Path planning algorithms based on graph theory

Abstract: Path planning is a very practical algorithm problem, which is widely used in many fields. The path planning algorithm has very strict requirements on time and space complexity, because the mover must know the next path in advance. This paper examines the pedigree of this algorithm, from the most classic and ancient algorithm to the most recent highly efficient algorithm. Road planning algorithms can be roughly divided into two categories: real-time and non-real-time. The difference lies in whether the full path is calculated before the first step is taken. This paper analyzes the advantages and disadvantages of various algorithms, and finally concludes that the A * algorithm is a relatively good algorithm. Then it introduces real-time and non-real-time improved versions of the A * algorithm. Finally, by writing code experiments in person, the understanding of this series of algorithms is further strengthened, and the limitations of talking on paper are avoided through practice.

Key words: path planning; graph theory; path-finding algorithms, astar algorithm (a* algorithm)

在地图导航，卫星系统，移动机器人，游戏 AI 等众多领域中，都需要在短时间内找出两点间最近的路径，路径规划（path planning）算法显得尤为重要。本文将以计算机游戏中的寻路算法为例，研究各类算法, 通过分析效率低下的算法来克服它们的缺点，通过分析高效的算法来了解其高效的原因。

在 2D 计算机游戏中，玩家和 AI 在大多数情况下都需要在平面上进行移动，需要调用寻路算法的场景非常普遍，同一时间会有很多 AI 需要调用寻路算法，因此寻路算法的效率与正确性至关重要。而在 3D 游戏中的移动也需要参考 2D 的坐标点，通常 3D 游戏不止一层（比如说有海、地、空三层），在多层的地图上寻路其实也就是将问题归结为开始点到转换点和转换点到目标点的若干个 2D 导航问题。

寻路算法与数据结构中图的算法关系非常紧密。大多数地图都可以看作是一个非常规则的图，所以 Dijkstra 算法等图中的经典算法在寻路中也有着广泛应用。

1 经典算法

在寻路算法中最重要的是躲避障碍物的问题。躲避障碍物的效果直接影响了算法的效果。

1.1 短视算法

短视算法的实现非常简单，可以用以下伪代码表示[1]：

```
while not at the goal
pick a direction to move toward the goal
if that direction is clear for movement
move there
else
pick another direction according to an avoidance strategy
```

短视算法采取的策略是“走一步算一步”，碰到障碍物无法前进再选择另一条路。这种算法的要求很低，仅仅需要实体与目标的相对位置即可。虽然它实现方便简单但寻路的正确性很低。

下面介绍几种短视算法中规避障碍物的方法：

1) 随机移动（Movement in a random direction）

随机移动指的是在实体遇见障碍物无法前进时，向可以行走的地方随机走一步，以求找到一条可以通行的道路。这种移动方式对小的、凸的障碍物比较有效，但一旦面对大的、凹的障碍物，则很容易卡住或是在障碍物上浪费很多时间。

这种算法的缺点很明显，即寻路成功率与正确性低，但它的效率非常高，时间复杂度与空间复杂度都为O(1)。

2) 沿着障碍物移动（Tracing around the obstacle）

这种算法的思想针对大型的障碍物。类似于盲人的行动模式，该算法在碰到障碍物时，沿着障碍物的边缘走下去，到某一个点时结束，回到正常的寻路模式上。

这种算法最关键的是结束边缘行走的启发函数。一种很典型的启发函数是在前进的方向正好是面向目标的方向时结束，但这种方法已经被证明了拥有无限循环的可能性[1]。

同样地，这种算法的时间复杂度与空间复杂度也为O(1)。

3) 鲁棒追踪（Robust tracing）

这种算法是前一种的改良版。正如它名字所示，它具有更强的鲁棒性。它来源于机器人的移动策略：当被阻挡时，设定一条当前位置到目标位置的直线，沿着障碍物移动直到再一次与直线相交。相比较 2)，这种方法在追踪时耗费的时间更多，但它可以保证找到一条可行的路径[1]。

当然还有一种 2)与 3)结合的算法，在没有检测到死循环时使用 2)，否则则用 3)。

1.2 预先规划算法

预先规划算法是一种“目光长远”的算法，在实体踏出第一步之前就已经计算好了整个路线。这些算法与图论中的算法是紧密相连的，游戏地图可以看作是一种顶点作矩形状排列的特殊的图。如果存在一条可能的路径，这些算法不会将实体引向死胡同，但这些算法实现复杂，时间与空间复杂度高，如何在取得全局最优与复杂度低之间做权衡就是本节的要点。

假设地图为一个边长为 n 的正方形，则顶点数(vertex)为n^2,边数(edge)为$2n(n-1)$。

1) 深度优先算法（Depth First Search）

在深度优先搜索中，你可以从图中的某个节点开始，继续深入和深入图形，同时可以找到尚未到达的新节点(或直到找到解决方案)。每当 DFS 运行完毕，它会回溯到最新的点，在那里它可以做出不同的选择，然后从那里探索。

伪代码[2]：

```
Create Start Node with Current Position
Add Start Node to Stack
While Stack Not Empty
    Get Last Node From Stack call Node “N”
    If N is Goal Then Found and Exit Loop
```

Else
Mark N Node as Visited
Expand a reachable Node from N call Node “Next N”
DFS with Next N (recursive)

时间复杂度[4]：

最坏情况：搜索完整个地图才找到目标点，$O(n^2)$

平均情况：搜索过半个地图之后找到，$O\left(\frac{n^2}{2}\right) = O(n^2)$

空间复杂度[4]：

最坏情况：考虑地图中可能情况下最长的一条路径，$O(n^2)$

平均情况：平均最大深度$d = \frac{\sqrt{2}}{2}n$，$O\left(\frac{\sqrt{2}}{2}n\right) = O(n)$

2) 迭代深化算法(Iterative Deepening)

迭代深化搜索实质是限定下界的深度优先搜索。它给出一个深度的上限，在迭代中不断增加这个上限。

伪代码[2]：

Create Start Node with Current Position
Add Start Node to Stack
While Stack Not Empty
Get Last Node From Stack call Node “N”
If N is Goal Then Found and Exit Loop
If N Cost > Max Depth Then Exit Loop
Else
Mark N Node as Visited
Expand a reachable Node from N call Node “Next N”
DFS with Next N (recursive)
If Not Found Then DFS with Max Depth = Max Depth + 1

时间复杂度：

最坏情况[3]：搜索完整个地图才找到目标点，$O(V) = O(n^2)$

平均情况：平均距离$\frac{\sqrt{2}}{2}n$，$O\left(\frac{n^2}{2}\right) = O(n^2)$，虽然这里和普通的 dfs 算法相同，但在 kort 等人的研究中指出，普通 dfs 常常会带有一个大于 1 的常数 c，因此实际效果是这种算法比较优秀。

空间复杂度[4]：

最坏情况：考虑地图中可能情况下最长的一条路径，$O(n^2)$

平均情况：平均最大深度$\mathrm{d}=\frac{\sqrt{2}}{2}n$，$\mathrm{O}\left(\frac{\sqrt{2}}{2}n\right)=\mathrm{O}(\mathrm{n})$

3) 广度优先算法(Breadth First Search)

伪代码[2]：

```
Create Start Node with Current Position
Add Start Node to Queue
While Queue Not Empty
    Get Best Node in Same Depth From Queue call Node “N”
    If N is Goal Then Found and Exit Loop
    Else
    Mark N Node as Visited
Expand each reachable Node from N call Node “Next N”
```

时间复杂度[4]：

最坏情况：搜索完整个地图才找到目标点，$\mathrm{O}(\mathrm{n}^2)$

平均情况：搜索过半个地图之后找到，$\mathrm{O}\left(\frac{\mathrm{n}^2}{2}\right)=\mathrm{O}(\mathrm{n}^2)$

空间复杂度[4]：

最坏情况：考虑地图中可能情况下最长的一条路径，$\mathrm{O}(\mathrm{n}^2)$

平均情况：广度优先需要存储节点非常多，$\mathrm{O}\left(\frac{n^2}{2}\right)=\mathrm{O}(n^2)$

广度优先搜索在实践中时间复杂度上相比深度优先搜索来说存在一些优势，但因为它需要占用的内存太多，缺点过于致命，所以一般不做使用。

4) Dijkstra 算法(Dijkstra’s Algorithm)

在地图内的每个区块移动消耗不同时，Dijkstra 算法可以非常方便的找出从地图上某个起始区块到其他所有可达区块的最短路径。Dijkstra 算法的消耗很大，但其最大优点在于它可以确保找到一条最短的路径。

伪代码[2]：

```
Create Start Node with Current Position
Add Start Node to Queue
While Queue Not Empty
    Get Best Node in Same Depth From Queue call Node “N”
    If N is Goal Then Found and Exit Loop
    Else
    Mark N Node as Visited
    Expand each reachable Node from N call Node “Next N”
If “Next N” is Diagonal Direction Then Cost = Cost + 0.4
```

时间复杂度[4]：

最坏情况：搜索完整个地图才找到目标点，$O(n^2)$

平均情况：$O(n^2)$

空间复杂度[4]：

最坏情况：考虑地图中可能情况下最长的一条路径，$O(n^2)$

平均情况：$O(2n^2) = O(n^2)$

5) 最佳优先算法（Best First Search）

最佳优先搜索，是一种启发式搜索算法（Heuristic Algorithm）,我们也可以将它看做广度优先搜索算法的一种改进；最佳优先搜索算法在广度优先搜索的基础上，用启发估价函数对将要被遍历到的点进行估价，然后选择代价小的进行遍历，直到找到目标节点或者遍历完所有点，算法结束。

当我们在状态空间中搜索的时候，最简单的方法就是穷举，在之前文章提及到的广度优先搜索和深度优先搜索都属于穷举类型的搜索，这种搜索方法有一个很大的缺点，就是在状态空间十分大的时候效率非常的差，因为需要穷举的状态太多了。而启发式搜索就是对状态空间中的每个搜索的位置（如图中的节点）进行一个评估，然后选出最好的位置。而在启发估价中使用到的函数我们称之为启发估价函数。

伪代码[2]：

```
Create Start Node with Current Position
Add Start Node to Queue
While Queue Not Empty
    Sort Node Queue by Cost Value in Ascending
    Get First Node From Queue call Node “N”
    If N is Goal Then Found and Exit Loop
    Else
    Mark N Node as Visited
    Expand each reachable Node from N call Node “Next N”
```

6) Astar 算法

Astar 算法综合了最良优先搜索和 Dijkstra 算法的优点：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）。

在此算法中，如果以$g(n)$表示从起点到任意顶点n的实际距离，$h(n)$表示任意顶点n到目标顶点的估算距离（根据所采用的评估函数的不同而变化），那么 A*算法的估算函数为：

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

这个公式遵循以下特性：

1. 如果$g(n)$为 0，即只计算任意顶点n到目标的评估函数$h(n)$，而不计算起点到顶点n的距离，则算法转化为使用贪心策略的最良优先搜索，速度最快，但可能得不出最优解；
2. 如果$h(n)$不大于顶点n到目标顶点的实际距离，则一定可以求出最优解，而且$h(n)$越小，需要计算的节点越多，算法效率越低，常见的评估函数有——欧几里得距离、曼哈顿距离、切比雪夫距离；
3. 如果$h(n)$为 0，即只需求出起点到任意顶点n的最短路径$g(n)$，而不计算任何评估函数$h(n)$，则转化为单源最短路径问题，即 Dijkstra 算法，此时需要计算最多的顶点；

伪代码[2]：

```
Create Start Node with Current Position
Add Start Node to Queue
While Queue Not Empty
    Sort Node Queue by f(N) Value in Ascending
    Get First Node From Queue call Node “N”
    If N is Goal Then Found and Exit Loop
    Else
    Mark N Node as Visited
    Expand each reachable Node from N call Node “Next N”
    f(Next N) = g(Next N) + h(Next N)
```

时间复杂度：

最坏情况：搜索完整个地图才找到目标点，$O(n^2)$

平均情况：与启发函数$h(n)$的取值相关

空间复杂度：

最坏情况：考虑地图中可能情况下最长的一条路径，$O(n^2)$

平均情况：与启发函数$h(n)$的取值相关

Astar 算法是一种非常灵活的算法，在实际应用中也被经常使用。通过修改启发函数$h(n)$可以得到不同的效果。

但 Astar 也有它的缺点，其中最致命的是它占用的内存很多，因为需要存放所有周边的节点。

1.3 算法比较

上方考察了很多经典的算法，它们在纯理论上性能差别并不明显，但考虑到在实际应用上地图一般比实体和目标点的距离大很多，所以不同的算法差距也会变得非常大。接下来通过考察它们在实际应用中的表现，得出一个较为初步的结论。

在 Khantanapoka 等的研究中[2]，用了四张地图进行实验，其中第一张没有障碍物而另外三

张都有。

TABLE I
ONE LAYER OF GROUND : OUTDOOR

Algorithm	Map 1# have not barrier.		Map 2#	
	Take time(ms.)	Expand Node	Take time(ms.)	Expand Node
Dijkstra	109	783	109	643
BFS	46	133	31	123
DFS	63	397	47	217
IDA	15	133	16	113
A*	32	133	32	126
IDA*	31	133	31	126
Breadth FS	125	783	94	643

Algorithm	Map 3#		Map 4#	
	Take time(ms.)	Expand Node	Take time(ms.)	Expand Node
Dijkstra	94	557	109	582
BFS	31	115	31	328
DFS	47	220	63	271
IDA	32	282	16	128
A*	32	108	31	129
IDA*	16	113	31	127
Breadth FS	94	555	94	582

[Dijkstra] Dijkstra's algorithm, [BFS] Best First Search, [DFS] Depth First Search,[IDA] Iterative Deepening algorithm, [A*] A-Star Algorithm (A*),[IDA*] Iterative Deepening A* algorithm, [Breadth FS] Breadth First Search [***DepthD A****] Depth Direction A*

Figure 1

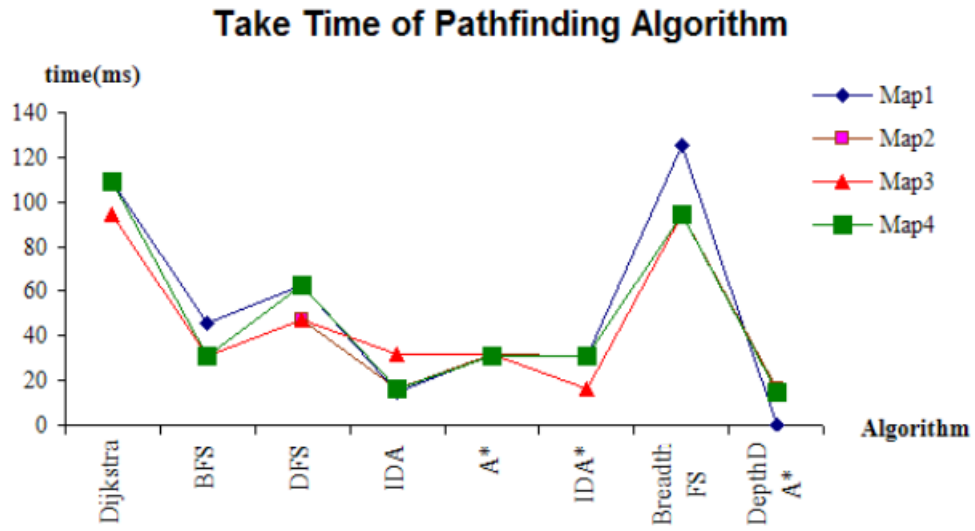


Figure 2

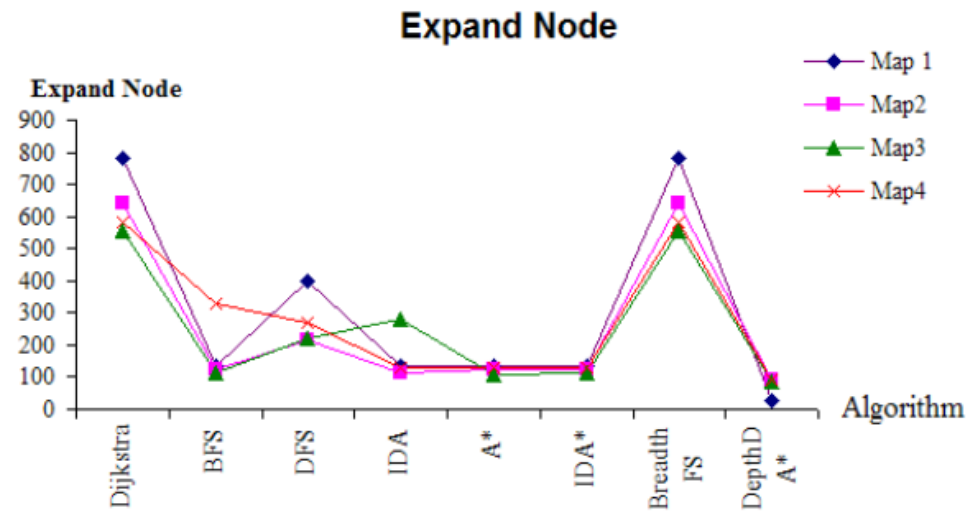


Figure 3

可以看出在上述地图中，dijkstra 算法和广度优先算法的效率很低，但它们可以保证找到最短的路径（这一点 dfs 无法保证）。DFS 和它的变形 IDA 在效率上比较优秀，但不够稳定。IDA 很成功地把内存占用降低了。让人惊讶的是 IDA*并没有比 A*在内存占用方面有更好的表现，不过这两者还是在经典算法中取得了毫无疑问的领先。

小结

本节考察了各种各样的经典算法，其中短视算法在如今算力的飞速进步下已经显得过时；

而在七种比较经典的预先规划算法的比较中，A*算法适用范围广，可以根据需求灵活改变$h(n)$，时间与空间复杂度低，在效率与正确性中取得了很好的平衡，成为了这些算法中的首选。当然作为一种古老的算法它也有着很多需要改进的地方，接下来，本文将主要研究近年来 A*算法变体的最新进展。

2 Non-Real-Time A*

在第二部分中，介绍的是非实时的 A*算法变体。非实时指的是算法在实体踏出第一步之前就规划好了全部的路径，而非在走每一步时实时计算下一步。这些改进算法大多是基于传统A*算法，对计算效率与占用内存做出了一定的优化。

1) **Depth Direction A***[2]

伪代码：

```
Create Start Node with Current Position
Create A Linear Function by m = (Start Y – Goal Y) / (Start X – Goal X)
Add Start Node to Queue
While Queue Not Empty
    Sort Node Queue by f(N) Value in Ascending
    Get First Node From Queue call Node “N”
    If N is Goal Then Found and Exit Loop
    Else
    Mark N Node as Visited
    Expand a Node by Linear Function call Node “Next N”
    If Found Obstructions Then
    f(Next N) = g(Next N) + h(Next N)
    adjust Algorithm Style to A-Star Algorithm
If Around Current Position <> 1 Then call Node “Next N”
```

这种算法的核心思想是两种搜索模式的切换：一种是经典的 A*算法，另一种是深度方向算法。深度方向算法首先要算出一个 m，即目标到开始点的斜率。如果没有碰到障碍物，就沿着开始点到目标点连成的直线前进，一旦碰到了障碍物，就切换为A*算法。这种算法在障碍物较少的地图上非常高效，可以长时间调用深度方向算法，如果在障碍物较多的地图上，就变成了 A*算法，效率也不会非常低下。

事实上，在实际的地图上应用，这种算法的表现优秀（见 Figure2，Figure3）。

2) **Improved A***[11]

伪代码：

1. *To establish a search graph G formed by the initial node s. To push the initial node s into table OPEN, OPEN=(s). CLOSED=(), at this time, table CLOSED is an empty table, so f(s)=0+h(s) can be expressed.*

2. *To repeat the following process until the destination node is searched out. If the table OPEN is empty, that is, OPEN=(), then quit, that is, failed to find out the destination.*
3. *To select and mark the node of the smallest f value without previous setting from the table OPEN, remove it from the table OPEN and put it into the table CLOSED.*
4. *If the best node is the destination node, that is, the target destination has been reached, the search has successfully obtained a solution and the algorithm is over. The best path from the initial node to node n is gotten by tracking the pointer of the main chain.*
5. *The node n will be expanded if the best node n is not the destination node. The node set M=(m) is generated which consisted of all the successor nodes of node n that are not its ancestors. Node m is added as a successor to the node n into the search graph G.*
6. *Node m is added to the table OPEN if m appeared neither in the list OPEN nor in the table CLOSED.*
7. *Node k is removed from the table OPEN and the node m is added to the table OPEN if m has a duplicate node k in the table OPEN and g (m) <g (k).*
8. *To see whether the successor node is in the table CLOSED. If it is not in table OPEN, the operations as follows are carried out if the successor node has duplicate node in the table CLOSED.*
   a) *To change node k into node m in the table CLOSED;*
   b) *To modify the successor g, f value in the table OPEN and CLOSED which contains node k in the sequence of the successor elements.*
9. *To resort the nodes from small to large in the OPEN table according to the value of f.*
10. *To jump to the step(2) and to cycle.*

这种算法针对 A*算法的两个缺点进行了改进：针对速度慢的缺点对评估函数$f(n)$进行了加权处理；针对在未知环境中可能卡死的缺点设置了手动的搜索标记。

A. 加权处理

启发函数$h(n)$通常是两点间的欧几里得距离，而它常常比两点间的实际最短距离要短。虽然它保证了算法的可接收性，但在效率上做出了很多妥协。

加权处理是根据以下引理[12]保证了算法的可靠性：在搜索结束之前，A*算法中的每一步中一定存在一个在 OPEN 表中的节点 n*，有以下性质：

1) n*在最佳路径上
2) A*已经找到了达到 n*的最佳路径
3) $f(n^*) \le f(n0)$

加权后的公式：

$$f_w(n) = (1-w)g(n) + wh(n), w \in (0.5,0.9)$$

通过加权可以得到更高的计算效率，并且能在存在路径的情况下找到最优解。

B. 手动标记

传统的 A*算法可能会在一些狭窄的或是弧形的障碍物中卡死，所以在算法中需要去预判并避免这些情况。

手动标记算法并不是需要程序员一个个去标记会造成卡死的点。在算法

开始时，当实体走如一些会卡死的地形时，这些点会被标记成潜在的障碍，加入一个手动标记集合中。标记完后实体会沿路返回，不会再陷入这些陷阱。在之后的搜索中，手动标记集合中的节点不会被放进 OPEN 集合中，因此不会被纳入路径当中，实体就不会再一次陷入这些陷阱。

C. 可行路径的优化

在搜索中第 *n* 步遇到障碍时，会在扩展节点中选出第 *n+1* 个节点。如果评估函数$f(n)$的选择不合适，导致扩展结点中的一些$f(n)$拥有相同的最小值时，会随机选择其中之一。但在再下一步，*n+2* 节点并一定是最优的选择。所以需要用这个算法来解决这种问题。

在手动标记算法中，会不断地将走不通的死路标记为障碍物而不断地选出更好的路径。优化算法需要在搜索结束后比较这一次和上一次搜索的路径，如果一致，则结束搜索，如果不一致，则继续。

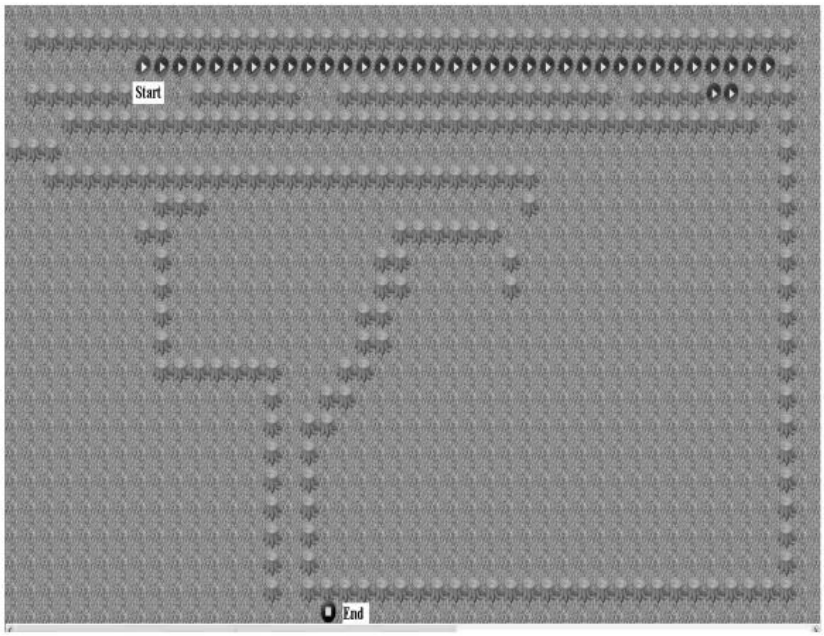
算法步骤：

1) *To set end node label i=1, the length n of table CLOSED is solved, to make the node label j=n, to make vector empty;*
2) *If i=n-1, to go to Step 6). Otherwise, to compare the node i with the node j. If i and j are different, to go to Step If i and j are the same, to go to Step 4);*
3) *If j=i+1, then to extract the node i, merge into vector and turn to Step5), else j--, transfer to Step2);*
4) *To extract the nodes i and merge into vector, i= j, j= n, to turn to Step2);*
5) *i++, j = n, to turn to Step2);*
6) *When node extraction is completed, the node sequence stored in the vector shall be the feasible path.*

实验结果

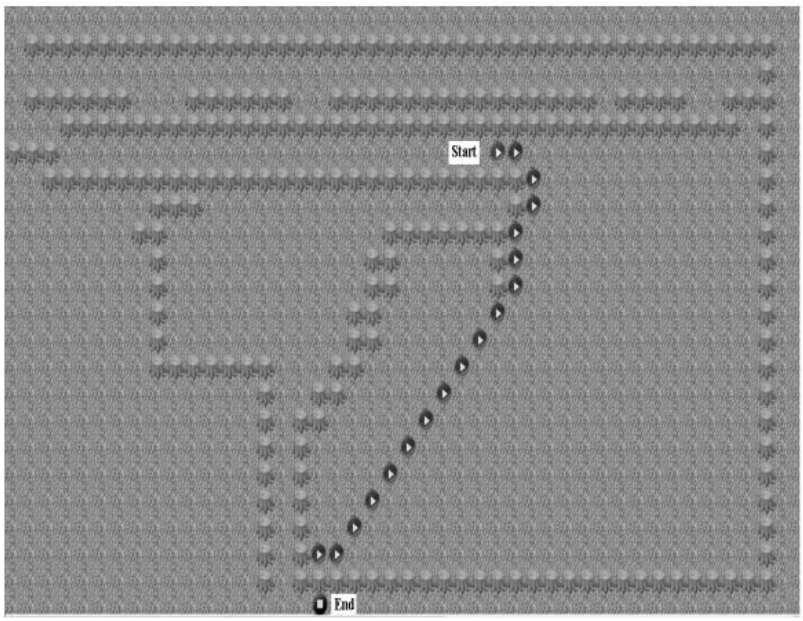
根据 Figure4、Figure5 可以看出传统 A*算法具有的一些局限性，会在长条形与弧形区域卡死而找不到真正正确的路径。在 Figure6 中可以看到 Improved A*成功绕过了障碍物，到达了终点。

在运算速度上，Table1 也展示了 Improved A*相比较传统 A*取得了不错的进步。



(a) Caught in strip-type traps using traditional A * algorithm

Figure 4



(b) Caught in arc-type traps using traditional A * algorithm

Figure 5

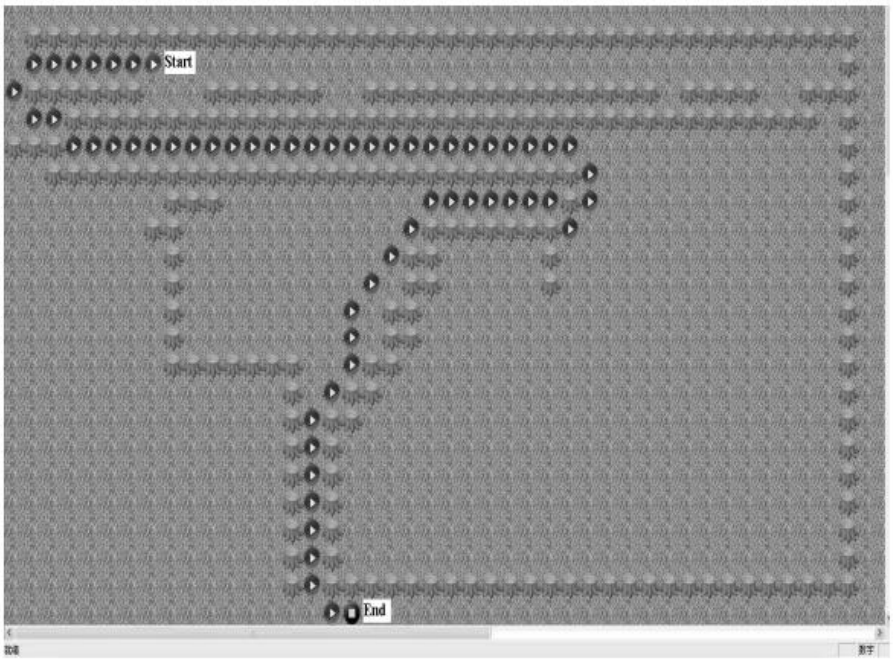


Figure 6

TABLE I. SEARCH EFFICIENCY COMPARISON TABLE

Type	Search times	Search cost (S)	Process time (S)
Traditional A*	200	4.359	7.879
Improved A*	80	2.823	3.601

Table 1

3 Real-Time A*

在 Non-Real-Time A*算法中，需要在移动者移动之前提前算出整条路径。这种算法在单移动者的情况下高效而可行，但如果移动者不止一个，那其他的移动者也会被看作“移动的障碍物”，从而改变地图，有可能让提前规划好的路径变得不再可行。而如果在每一步踏出之前都调用 Non-Real-Time A*算法，那么会导致庞大的计算量并占用指数级的内存空间。因此，在多移动者的场景之中，需要用到 Real-Time A*算法。

1) Minimin Lookahead Search[14]

在所有边的代价都是相同的情况下，Minimin Lookahead Search 算法从当前位置搜

索一片固定深度的范围，对所有搜索的边界应用启发函数，以求在受限的视野中找到最优的路径。找到了最优的搜索边界后，从边界反向追溯到眼前的一步并走出，之后再不断调用这个算法。没有直接朝着边界前进的理由是踏出下一步后可能会出现更好的边界可供选择。

在边的代价不同的情况下，则需要采用 A*的代价函数$f(n) = g(n) + h(n)$。有两种策略可以选择：一是前进固定的步数，二是前进固定的估算函数$g(n)$。算法采取了第一种，因为计算量取决于步数而不是实际行走的代价。如果在视野内发现了目标点，则$h(n)$为 0，寻路结束；相反，如果碰到了死路，则将$h(n)$赋值为无穷大，保证了这条路径不会被选择。

Minimin Lookahead Search 主要有以下两种实现：

a) Time-limited A*

Time-limited A*采取的策略是限制 A*搜索的时间，在规定时间结束时不论是否找到最佳路径，都给出下一步的方向。这种算法的缺点和 A*类似，都会占用指数级的空间。另外，这种算法可能会在一次搜索的迭代中强行打断，从而给出一个临时的结果，具有很高的随机性。

b) Threshold-limited IDA*

Threshold-limited A*采取的策略是调用 IDA*固定次数，即 threshold，在迭代过常数次之后，给出最后的结果。这种策略的问题是 threshold 的值如何设定。

2) Alpha Pruning[14]

Prune 是修剪的意思，该算法的核心思想就是减少需要检查的“分支”，但能做出同 Minimin Lookahead Search 一样的决定。

在 Korf 的论文中证明了如果不单单对边界节点应用代价函数估算，而对中间节点应用，以达成修剪的效果，只要代价函数$h(n)$是单调的。在实际应用中，无论是曼哈顿距离还是欧几里得距离，这些函数都是单调的。

Alpha Pruning 的实现：在变量α中存放着最小的$f(n)$，一旦碰到的分支的$f(n)$不比α小，则直接终结这个分支的搜索。因为$f(n)$是单调的，所以边界节点的$f(n)$一定大于或等于中间节点的$f(n)$。如果碰到了更小的α，则把α赋值为更小的那个。

实验结果

见 Figure7，左侧的直线为没有修剪的搜索算法，右侧为 Alpha Pruning 算法，可以看见在相同的搜索视野中，Alpha Pruning 明显生成的节点更少。

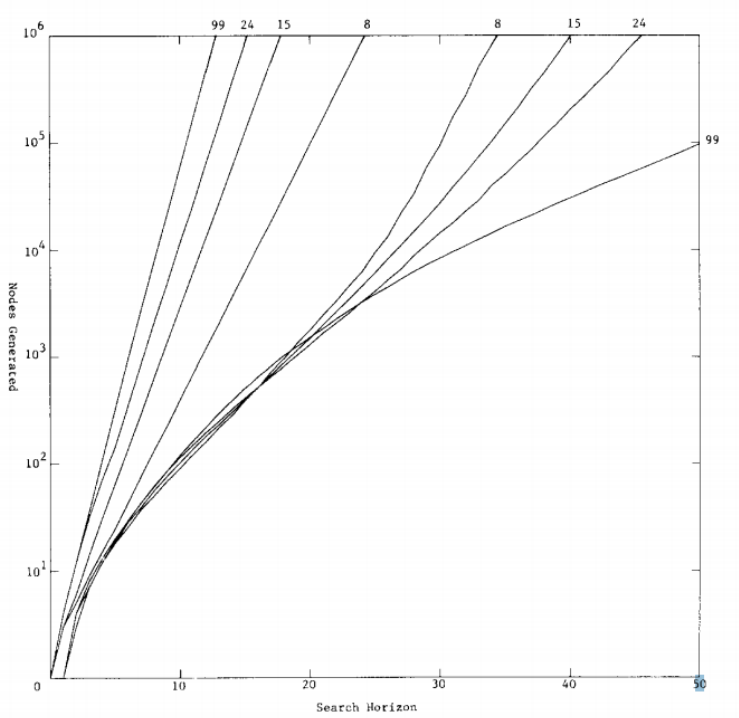


Figure 7

3) (L)RTA*[14]

LRTA*全称 Learning Real-time A*，是 Korf 在 1990 年提出的一种让移动者在知道到达目标的全部路径之前就移动的实时启发算法，也是实时 A*算法的先驱。LRTA*算法受到国际象棋程序的启发，在国际象棋中，每一步都要在规定的时间内下完，对手下的棋也会影响到之后的判断，总的可能位置约有 10^{43} 个。因此在这里用一种从第一手之前就预测完接下来所有的落子是不现实的。象棋算法面对一种受限的搜索视野，从而只能做出次优的决策。

RTA*

在了解 LRTA*前，首先要了解它的前身 RTA*。RTA*的发明是为了解决一个 Minimin Lookahead Search 存在的问题：因为首先的搜索视野，所以很可能会遇到死路或障碍时不断返回到之前访问过的节点，陷入死循环。

在 RTA*中，每个节点也存在一个代价函数$f(n) = g(n) + h(n)$。但不同于 A*，RTA*中的$g(n)$是从当前位置到节点n的实际距离，而不是从初始位置到节点n的距离。通过维护一张线性表，每一步中更新$g(n)$的值，避免了陷入死循环。

但这种简单的算法存在以下问题：

1. 维护线性表并在每一步中更新需要耗费很多时间。
2. 还不清楚如何去更新$g(n)$的值。
3. 还不清楚如何找到一条前往从 OPEN 表中选出的目标节点的路径。

针对以上缺点，Kort 提出了一种维护哈希表的算法。在算法的每一轮中，对每一个不在表中的节点计算启发函数$h(n)$，对在表中的节点则直接使用表中的值，然后在所有候选节点中选出$f(n)$最小的一个，执行移动。同一时间，将第二小的状态与$f(n)$的值记录在哈希表中，代表着返回到这个状态估算出的$h(n)$。

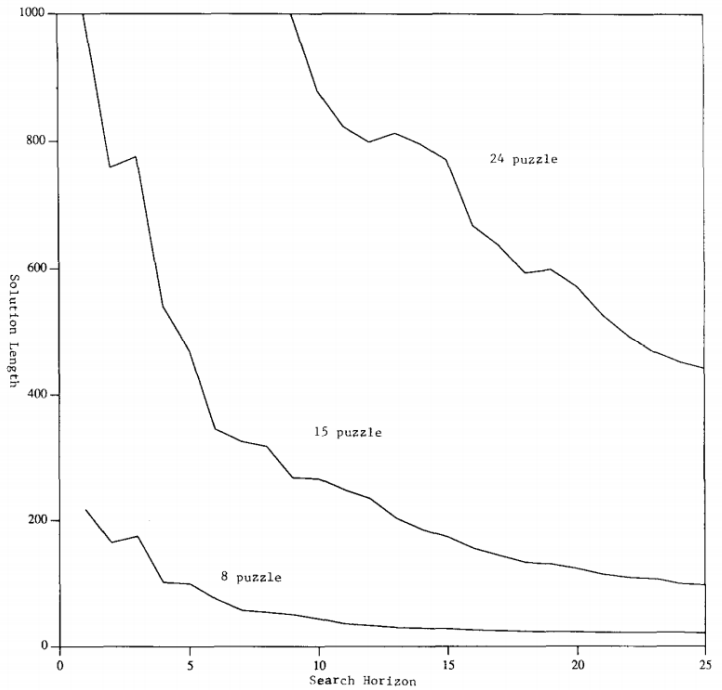


Figure 8

Learning RTA*

Learning RTA*的发明是为了解决 RTA*不能应对多问题的场景。因为在 RTA*中，哈希表中存放着第二优的节点，代表着对从选中的最优节点返回到第二优节点的代价的准确估计。但是，如果事实证明选择的最优节点是正确的，存储的第二优节点就会造成膨胀的值，让其他节点的值显得小了。这些膨胀的值会导致在随后的寻路中走向错误的道路。

解决这个问题只需要简单地将存储第二小的值变成存储最小的值，这就变成了 LRTA*。LRTA*的核心思想是随着移动者的移动，越来越多的信息被获取了，这就是一个“学习”的过程，因此可以做出更明智的判断。

在一个 N 为边长的正方形中，有 N^2 个节点。在实验中，平均学习时间是$O(n^3)$，在学习完成之后，平均的寻路时间降低到了线性时间 2N-2。

4 代码及实验

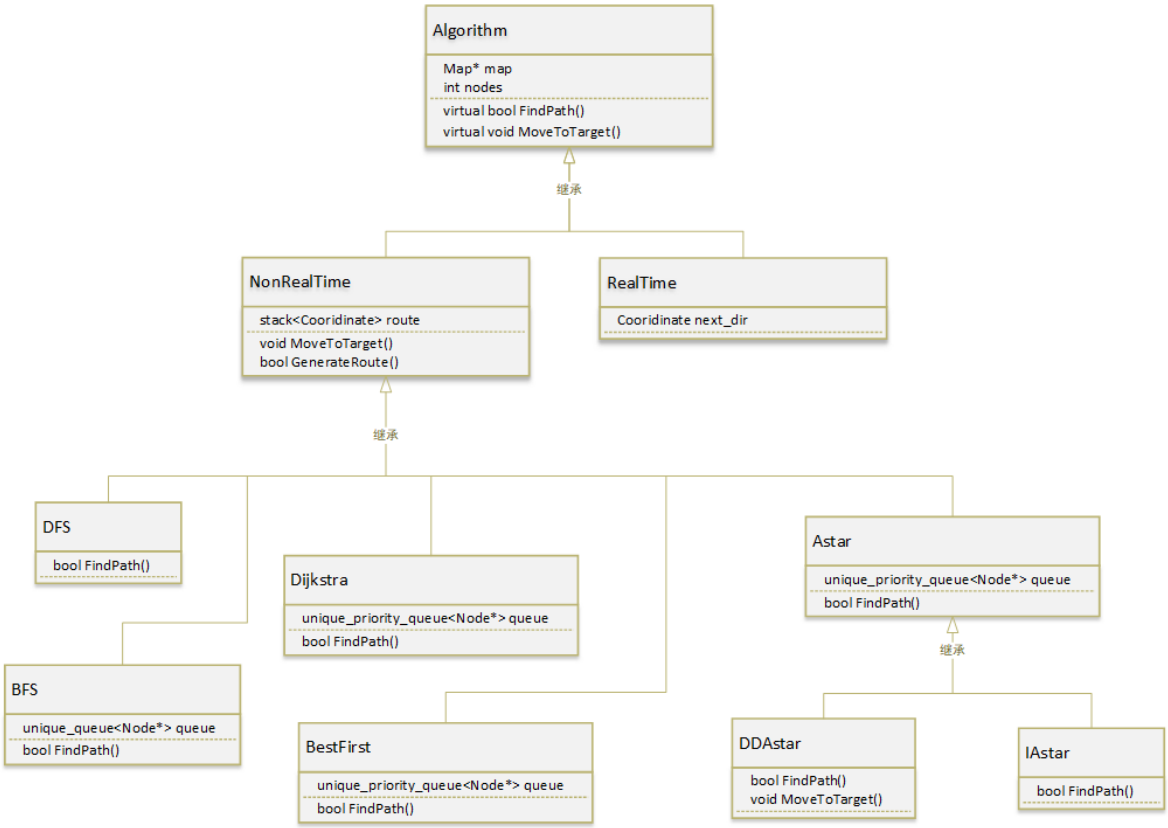
4.1 代码结构

最原始的基类为 Algorithm 抽象类，其中有一个 map 类的指针，nodes 节点数的计数值。成员函数有虚函数 FindPath，若成功找到路径则返回 true，否则返回 false，虚函数 MoveToTarget，令移动者从开始点一步步走向目标点。

Algorithm 类有两个子类，NonRealTime 和 RealTime。

NonRealTime 有 DFS、BFS、Dijkstra、BestFirst、Astar 几个子类，因为算法的不同，这些子类都有各自的数据结构。

Astar 有两个子类，DDAstar 和 IAstar，都是 Astar 的改进版本。



4.2 地图介绍

实验中准备了六张地图，其中两张 8*8，两张 16*16，两张 32*32，每组中各有一张稀疏型，一张密集型（见 Figure9）。红色为起点，蓝色为终点，+为障碍物。从左到右，从上到下分别为 map1，map2，map3，map4，map5，map6。

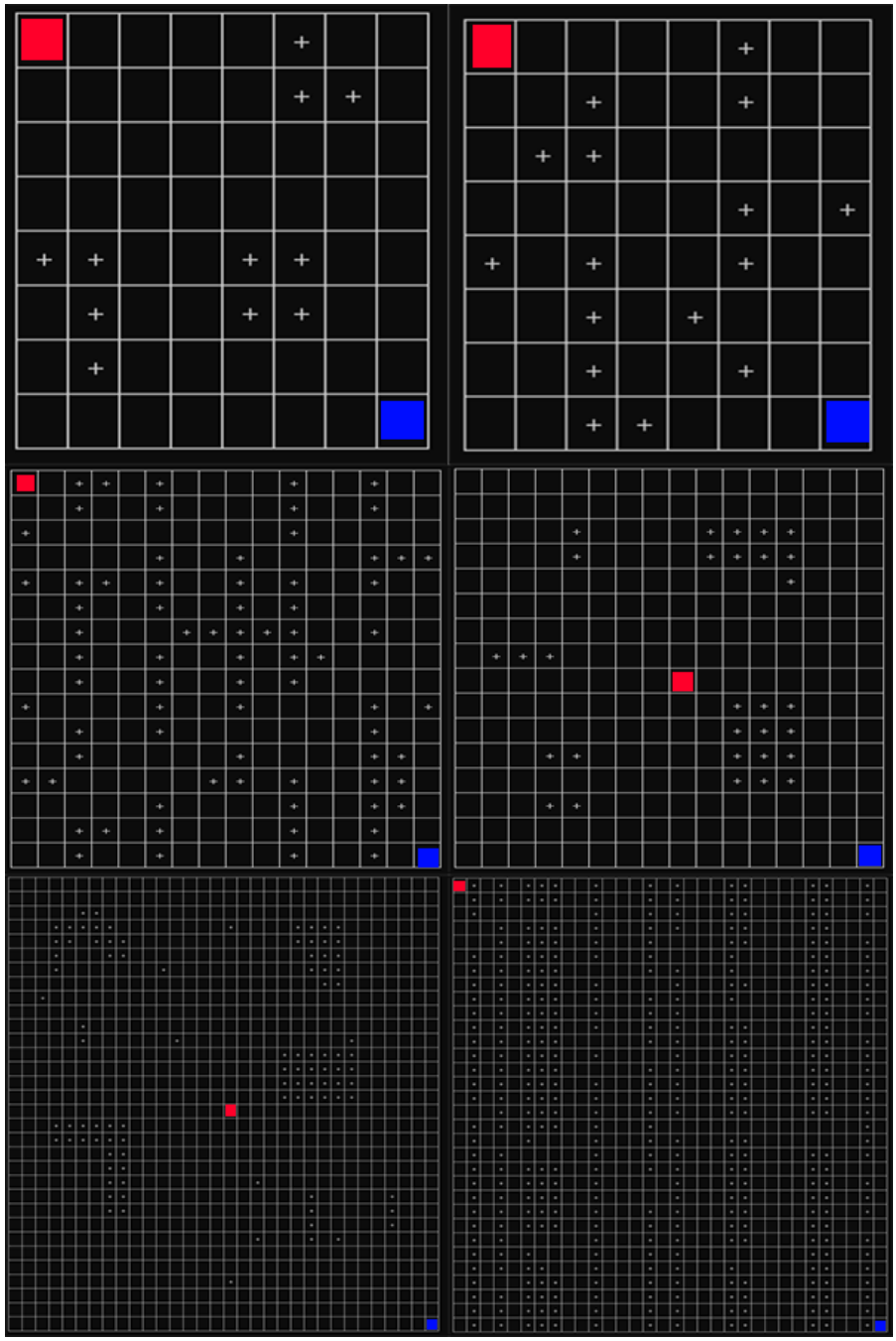


Figure 9

在这六张地图中，分别用 DFS、BFS、Dijkstra、Astar、BestFirst、DepthDirectonAstar、ImprovedAstar 算法进行了实验。消耗的时间见 Figure10，创建的节点数见 Figure11，生成路径的长度见 Figure12。

4.3 时间消耗

可以看出没有使用启发函数的算法：DFS、BFS、Dijkstra 算法消耗的时间非常多。特别是在 map5，地图大而空旷的情况下，BFS 需要非常多的时间去遍历整个地图，从而搜索到目标节点。Dijkstra 算法在理论上说是会比较慢的，但在代码中使用了用堆实现的优先队列，从而快速地选出$g(n)$最小的节点。

而使用了启发函数的剩下的算法则表现都非常优异。DDA 和 IA 是对 Astar 的改进版，在一些比较空旷的地形（如 map5）相比 Astar 来说会更快，但是放在比较密集的地形（如 map6），它们自身算法的复杂性反而延缓了搜索的速度。

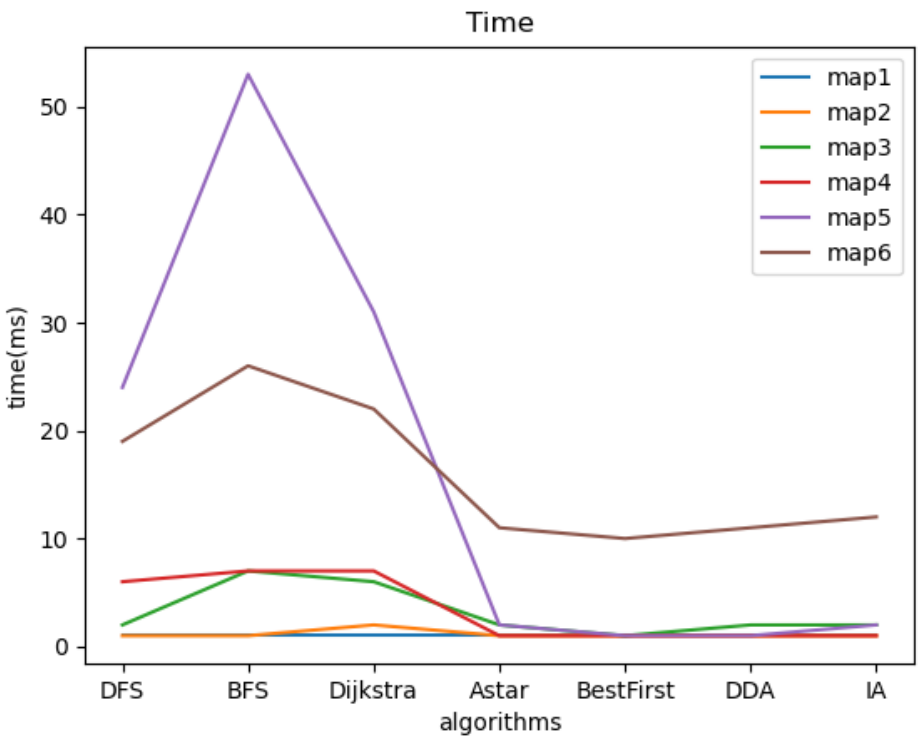


Figure 10

4.4 展开的节点

与理论分析不同的是，DFS 反而比 BFS 展开了更多的节点。一方面，游戏中使用的地图与常规图论中的图不同，会有很多障碍物，这些节点不会被 BFS 展开。另一方面，在 map5 这杨空旷的地图上 DFS 算法非常盲目，在一个方向上搜寻到底，如果运气好则会很快找到，运气不好会不断地将节点压入栈中。

因为 Dijkstra 算法和它之后的算法都可以使用优先队列进行优化，所以占用的内存空间都比较小。

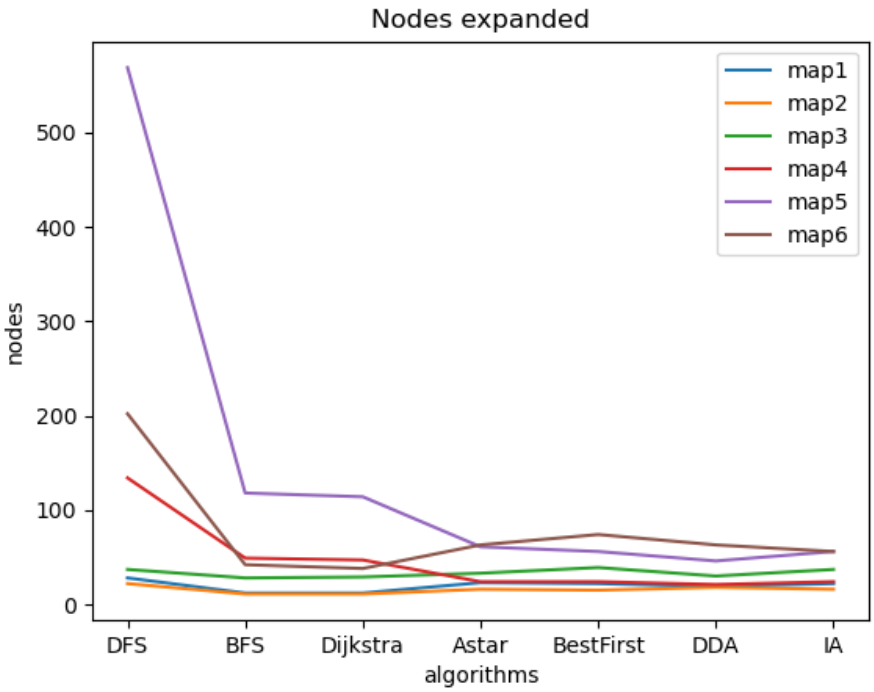


Figure 11

4.5 路径长度

DFS 因为其盲目性，在运气不佳的时候（map5）会走一条非常曲折漫长的路径，而在运气好的时候（map3）则比 BFS 的路径短不少。

BestFirst 算法是一种贪婪算法，在很多情况下不能得出最优解，在 map6 就计算出了一条比较长的路径。

Dijkstra 算法保证了能找到最短路径。在上文中提到了 Astar 算法中启发函数$f(n)$如果不大于实际距离，也可以保证得到最短路径。但在本次实验中为了保证计算效率，启发函数用了曼哈顿距离以避免欧几里得距离的浮点运算，所以有的时候不能保证路径最短，但可以看出，Astar 算出的路径长度与 Dijkstra 算法的没有很明显的差异，属于可接受的范围。

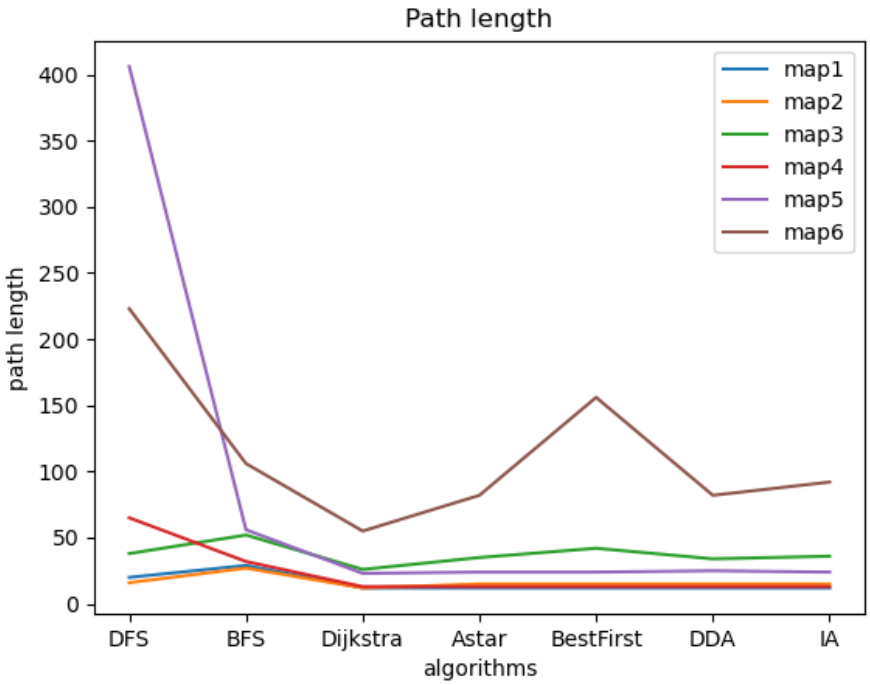


Figure 12

References:

[1] Stout B. Smart moves: Intelligent pathfinding[J]. Game developer magazine, 1996, 10: 28-35.

[2] Khantanapoka K, Chinnasarn K. Pathfinding of 2D & 3D game real-time strategy with depth direction A∗ algorithm for multi-layer[C]//2009 Eighth international symposium on natural language processing. IEEE, 2009: 184-188.

[3] Depth-First Search (DFS). Brilliant.org. Retrieved 12:41, April 4, 2020, from https://brilliant.org/wiki/depth-first-search-dfs/

[4] Korf R E. Depth-first iterative-deepening: An optimal admissible tree search[J]. Artificial intelligence, 1985, 27(1): 97-109.

[5] Korf R E, Reid M, Edelkamp S. Time complexity of iterative-deepening-A∗[J]. Artificial Intelligence, 2001, 129(1-2): 199-218.

[6] Hart, P.E., Nilsson, N.J., Raphael, B., 1968. A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics SSC4 (2), 100–107.

[7] Bell M G H. Hyperstar: A multi-path Astar algorithm for risk averse vehicle navigation[J]. Transportation Research Part B: Methodological, 2009, 43(1): 97-107.

[8] Eppstein, D., 1998. Finding the k-shortest paths. SIAM Journal of Computing 28 (2), 652–673.

[9] Dinic, E.A., 1970. Algorithm for solution of a problem of maximum flow in a network with power estimation. Soviet Mathematics Doklady 11, 248–264.

[10] Spiess, H., Florian, M., 1989. Optimal strategies: a new assignment model for transit networks. Transportation Resea rch Part B 23 (2), 83–102.

[11] Yao J, Lin C, Xie X, et al. Path planning for virtual human motion using improved A* star algorithm[C]//2010 Seventh international conference on information technology: new generations. IEEE, 2010: 1154-1158.

[12] GAO Qingji, YU Yongsheng, HU Dandan, "feasible path search and optimization Based on an improved A * algorithm", China Civil Aviation College Journal, 2005,23 (4): 42-44.

[13] Lawrence R, Bulitko V. Database-driven real-time heuristic search in video-game pathfinding[J]. IEEE Transactions on Computational Intelligence and AI in Games, 2012, 5(3): 227-241.

[14] Korf R E. Real-time heuristic search[J]. Artificial intelligence, 1990, 42(2-3): 189-211.