寻路算法在各游戏场景中的应用与改进

软件工程 陈飞飞 指导教师 张惠娟

【摘要】 人工智能在现在的游戏中占有重要地位，而寻路算法则是人工智能的基础与核心。本论文对寻路算法进行了系统的讨论研究，首先介绍了寻路相关的基础知识，并对解析式算法代表Dijkstra和启发式算法代表A\*进行研究分析。其次对使用广泛的A\*算法进行了深度的改进研究，同时对BFS无法找到最短路径问题给出解决方案，提出了优先队列的BFS，对DFS寻路无方向性给出了初步解决方案，提出了方向指导的DFS。再次对游戏实际问题进行了分析，探讨了启发式函数的设计。最后，配以相应的算法测试系统，对改进过的算法和原有算法进行实验对比，通过五个实验场景的测试，得出各算法适用性以及效率的结论。

【关键词】 人工智能 寻路 A\*

【Abstract】 Artificial Intelligence plays an important role in the game now , and the pathfinding algorithm is the foundation and core of artificial intelligence.This thesis makes a system discussion about the pathfinding algorithm, first introduced the basics of pathfinding, and analysis the representative Dijkstra algorithm of analytic algorithm and the representative A\* of heuristic algorithm. Depth study of the improvement is followed by the use of a wide range of A\* algorithm. Give a solution on the BFS which is unable to find the shortest path problem, raised the priority queue of BFS. Also give a preliminary solution on the DFS which has no direction, put forward the DFS of the direction guidance. Then analysis the practical problems of the game and explore the design of the heuristic function. Finally, together with the corresponding algorithm testing system, make a experimental comparison with the improved algorithm and original ones. After five experimental test of the scene, the thesis comes to the conclusion of the applicability and efficiency of each algorithm.

【Keywords】 AI PathFinding A\*

目录

[1 引言 2](#_Toc324680117)

[1.1 研究背景 2](#_Toc324680118)

[1.2 研究意义 3](#_Toc324680119)

[1.3 研究现状 3](#_Toc324680120)

[1.4 研究内容与论文组织安排 4](#_Toc324680121)

[1.4.1 论文研究内容 4](#_Toc324680122)

[1.4.2 论文组织安排 4](#_Toc324680123)

[2 寻路算法基础 5](#_Toc324680124)

[2.1 寻路问题简介 5](#_Toc324680125)

[2.1.1 非负权边的图 6](#_Toc324680126)

[2.1.2 有向权图 6](#_Toc324680127)

[2.2 几种寻路算法 7](#_Toc324680128)

[2.2.1 Dijkstra算法 7](#_Toc324680129)

[2.2.2 A\*算法 9](#_Toc324680130)

[2.2.3 BFS算法 10](#_Toc324680131)

[3 游戏场景中寻路算法改进研究 12](#_Toc324680132)

[3.1 寻路环境设计 12](#_Toc324680133)

[3.2 A\*算法的研究及改进 13](#_Toc324680134)

[3.2.1 传统A\*算法代码分析 13](#_Toc324680135)

[3.2.2 A\*算法初步改进 15](#_Toc324680136)

[3.2.3 A\*算法进一步改进 16](#_Toc324680137)

[3.2.4 使用二叉堆 19](#_Toc324680138)

[3.3 BFS的改进研究 24](#_Toc324680139)

[3.4 DFS的改进研究 27](#_Toc324680140)

[3.5 实际问题的改进研究 29](#_Toc324680141)

[3.5.1 对于八方向的地图寻路 29](#_Toc324680142)

[3.5.2 对于不同消耗地图的研究 31](#_Toc324680143)

[4 寻路算法测试系统设计实现 35](#_Toc324680144)

[4.1 需求分析 35](#_Toc324680145)

[4.2 测试系统的设计实现 35](#_Toc324680146)

[4.2.1 对核心功能的支持 35](#_Toc324680147)

[4.2.2 多种算法支持 35](#_Toc324680148)

[4.2.3 不同场景的设置 37](#_Toc324680149)

[4.2.4 测试过程与结果的展示 37](#_Toc324680150)

[5 寻路算法测试系统测试 38](#_Toc324680151)

[5.1 无障碍物地图测试 38](#_Toc324680152)

[5.2 不同消耗的地图测试 40](#_Toc324680153)

[5.3 一般障碍物的地图测试 41](#_Toc324680154)

[5.4 内凹型障碍物地图测试 43](#_Toc324680155)

[5.5 迷宫型地图的测试 45](#_Toc324680156)

[6 总结与展望 46](#_Toc324680157)

[6.1 总结 46](#_Toc324680158)

[6.2 展望 46](#_Toc324680159)

1. 引言
   1. 研究背景

一直以来，人工智能在游戏中都扮演着重要的角色。作为人工智能领域非常基础而且核心的部分，寻路算法一直是游戏开发者们热衷讨论的话题。不管是可以准确避开障碍物行进的怪物，还是列队跨过桥梁的军队，它们的行为和现实的人类是非常相似的，这些“智能”的行为会让玩家感到游戏中的角色是“真实的”，它们和人类一样拥有“思维”和“眼睛”，它们可以看见眼前的障碍，并且通过逻辑判断后精巧地绕开。而赋予它们这些“智能”的上帝就是寻路算法。

由于游戏的类型多种多样，产生了各种适用于不同类型游戏的寻路算法。而且不同的游戏规模、不同的路径条件、不同的寻路要求，导致同一种寻路算法发展出许多“变种”，以适应游戏的需求。为了深入理解各种寻路算法在游戏中的应用与表现，以便未来可以在游戏编写中灵活运用，本文进行了综合性的讨论与研究。

* 1. 研究意义

从电脑游戏诞生开始到现在这几十年中，软硬件水平都在以惊人的速度不断地提高着，使得游戏也不断地发展完善。而这些游戏元素的提升主要表现在以下几个方面：画面效果趋于真实、绚丽，声音更加逼真；游戏交互更加符合人类感受；游戏创意更加新奇、震撼。随着游戏引擎的不断升级换代，现在大部分游戏的画面都越来越华丽，这也引发了一个问题：玩家对于这种华丽的外观已经视觉疲劳了，如何更进一步吸引玩家的眼球呢？

玩过《星际争霸》的人都知道，这款游戏的画面在现在看来是十分粗糙的，但是直到现在，热衷于玩它的人数量依然庞大，在《星际争霸2》出售之前，国际上依然有众多它的比赛项目，为什么一款十几年前的游戏依然受到大家的追捧？其很大一部分原因就是由于该游戏的人工智能做的非常完善，不仅表现在玩家与电脑对战的时候，在玩家之间的对战上，人工智能也表现地淋漓尽致：单位之间的相互协同，编队移动以及精准的寻路等。

玩家需要的是一个不通过作弊手段而达到很高水平的一个对手，怎样让电脑像玩家一样，它们也不知道玩家开始建造基地的位置，也不知道迷雾中玩家部分的编制，却要采取正确地应对措施与玩家对抗，这明显取决于人工智能部分编写的好坏。由此可见，人工智能是一款游戏是否耐玩的关键性部分。而通过资料显示，早期8位计算机时代或者更早的时候，我们只有1-2%的CPU时间用于运行人工智能部分，而现在则提高到了10-35%的CPU时间。更多的CPU时间意味着更复杂的人工智能得以展示，这对人工智能的发展非常有利。

在很多需要人工智能的游戏中，寻路都是非常基础而且重要的部分。就像前面提到的 《星际争霸》，每一次的单位移动，寻路算法都在运行。而寻路算法的效率将会直接影响到玩家的游戏感受，试想一下没有人口上限的《红色警戒2》，当玩家控制超过200个单位的大部队移向敌方基地时，如果寻路算法低效，那么游戏将很可能处于死机的状态。

那么，如何编写出一个高效且符合要求的寻路算法就成为本论文的研究核心了。

* 1. 研究现状

一个好的寻路算法，应该满足以下几个条件：

高效：单位对于挑选去目的地的路径必须是迅速的，通常不能让玩家感觉到游戏由于正在由于寻路而帧数下降或是暂停，这是无法忍受的；

可靠：由于游戏中存在着各式各样的复杂情况（随游戏的复杂程度而定），单位必须要处理好各种情况下的寻路不会出现异常；

智能：这里指的是单位选择的路径的过程是智能的，有的时候影响路径的因素不仅仅是路程的长短，这个时候程序就要做出权衡，以选择符合整体要求的路径；又有的时候单位需要像自然人一样探索性的寻找路径，而不是一下子就找到了最短路径。

这里，基本的寻路算法像是Dijkstra算法在一般情况下是无法满足上述要求的，故衍生出很多其他寻路算法，它们的侧重点各有不同。像是Dijkstra这种属于解析算法，搜索过程没有对于目标的提示，盲目地搜索，缺点是显而易见的，搜索范围增大后耗费时间提升巨大。如果可以提示算法朝着目标运行会怎么样呢？

由此产生了新的类型：启发式算法，这种算法通常使用一个估计函数h(n)（又称启发式函数），用于指导算法朝着预想的目标搜索，如果h(n)是可接受的，那么启发式算法一定能找到最佳解。作为启发式算法的代表，A\*算法在1968年首先由Peter Hart, Nils Nilsson和Bertram Raphael提出，它具有能迅速找到最优解的优点而被广泛使用，当然前提是估计函数h(n)是可接受的。在一些复杂的路径状况下，糟糕的h(n)设计往往适得其反，使得A\*算法不仅效率直线下降，甚至最终得到错误的解。所以如何设计h(n)成为A\*算法的一大难点。

对于A\*算法中数据的组织结构的不同，又产生了形如二叉堆A\*算法以及IDA\*算法等等，前者追求的是最快的搜索速度，特别是大规模搜索，后者则追求更少的内存占用。

目前，A\*算法及其变种是游戏业界的使用标准。

* 1. 研究内容与论文组织安排
     1. 论文研究内容

本文首先对于主流的几种寻路算法进行了一一概括与比较，得出它们的时间复杂度，总结出它们适用的场景。然后对于以下几种场景进行试验对比：无障碍物场景、有障碍物场景以及多单位障碍物场景。并且对于特定场景中的算法进行了一定的改进研究。

论文首先对Dijkstra算法进行分析研究，作为其他算法的基础标准。之所以选择这个算法作为标准，是因为它相对于其他算法来说，时间复杂度较高，一定规模的节点下消耗时间较多，容易对比出其他算法的效率提高；还有一点就是它一定能找到最短路径，有些算法可以在很短时间内找出次佳解，而不是最佳解，比如h(n）设计存在问题的A\*算法，这是Dijkstra算法可以作为最佳解的标准。但是Dijkstra却不能应对负权边的路径搜索。

论文其次对于深度优先搜索(DFS)和广度优先搜索(BFS)进行了研究，深度优先搜索不适合寻找最短路径甚至是最少步数，它适用于迷宫出路的搜索或者是连通图的寻路。而广度优先搜索同样无法找到最短路径，却可以找到最少步数的路径。如果使用优先队列来对广度优先搜索加以修改的话，可以找到非负权值的最短路径。论文同样讨论了优先队列的广度优先搜索。

论文对于A\*算法做了一些比较细致的改进研究，主要如下：

1. 节点组织结构的改进

为了解决A\*算法在开启列表、关闭列表规模变大时效率降低明显的问题，对开启列表以及关闭列表的节点排列方式做了改进，采用了插入排序的方式，使得每次取出f(n)最少的节点即列表中第一个节点；由于插入排序的时间复杂度为o(n^2)，所以规模变大后依然无法接受，故又采用了二分查找来改进插入的过程以及判断后来发现的节点是否存在于开启关闭列表中。

1. 节点存储容器的改进

一开始存储节点的列表使用了标准库的list容器，便于插入删除元素。但是由于改进采用了二分查找，需要对列表中节点进行随机访问，故改用vector代替list。但是由于vector采用了线性存储结构，所以插入删除元素将会引发该元素后面的元素整体移动，降低效率。然而观察发现，当算法不断进行，所找到的节点越来越靠近终点，此时找到的节点处于最短路径的可能性越来越高，所以节点的f(n)值就会越来越小，所以将vector中的节点按f值从大到小排列，这样能保证新插入的节点尽量是处于列表后面的位置，从而使vector插入时引发的整体移动降低。

论文在后面考虑了实际的游戏中寻路的各种状况。首先，我们一般考虑的寻路算法都是在理想状况下进行测试的，比如地图除了无法通过的障碍物之外其他的区域移动消耗都是一样的，有的甚至没有障碍物设定，这种情况下对于A\*算法，使用最通用的曼哈顿函数作为启发式函数即可。然而对于像是自然界那样的地形，有耗费高的山地和耗费低的平原以及无法通过的河流，这时曼哈顿函数可能会引发错误，产生错误的路径，论文将会对这种状况作出研究讨论。

* + 1. 论文组织安排

论文共分为六章，详细地研究了寻路算法以及其在游戏场景中的具体应用与相关改进，并给出了实验数据予以论证改进的成功性。

1. 第一章，引言

介绍了人工智能领域中寻路算法的相关背景，阐述了游戏中人工智能的应用，指出人工智能的现状，得出其作用将越来越重要，从而给出寻路算法的研究意义。同样概括了本论文对于寻路算法的相关研究。

1. 第二章，寻路算法基础

该章节介绍了寻路问题，介绍了两大类寻路算法：解析算法以及启发式算法。并分别介绍了其代表性的算法：Dijkstra算法和A\*寻路算法，对算法进行数学分析，并指出这些原有算法的适用性和不足之处。

1. 第三章，游戏场景中寻路算法改进研究

该章节综合了现实游戏中寻路可能出现的状况进行分析，对上述提到的标准算法进行改进，主要目的就是提高寻路效率，降低内存与时间的开销以及对寻路结果进行优化。并对启发函数的设计等方面进行讨论研究。

1. 第四章，寻路算法测试系统设计实现

该章节介绍了寻路算法测试系统的设计，该系统设计目标是可以自由切换待测试的算法，对多种不同的场景进行统一测试，可以得出算法消耗的时间以及探索过的节点和得出的路径。系统必须要具备很高的灵活性，可以随意配置场景和算法，最后通过图形界面将寻路过程呈现出来。

1. 第五章，寻路算法测试系统测试

该章展现上述系统的实现，通过一定的代码说明系统的架构。通过控制变量法进行多次实验，并把结果截图呈现出来，分析制表，得出数据。主要是对不同的算法进行比较实验以及同一种算法的改进对比实验。

1. 第六章，总结与展望

该章是对论文的总结，概括论文的研究成果，并且指出函待改进的部分以及今后还需要做的努力。

1. 寻路算法基础
   1. 寻路问题简介

寻路一般指的就是找到某点到目标点之间的路径。寻路是以一张地图为基础的，而地图则是由一系列的节点以及节点之间的路径构成的。为了使问题简单化，游戏中采用的方法就是把一整张地图分割成许多单位大小的地图块，把这些地图块都看作是寻路中的节点，把相连的地图块看作它们中存在路径，这样看似复杂的地图场景，就变成了无向图(或是有向图)。

寻路在各种不同的游戏层次，要求不尽相同。对于迷宫，我们只寻求最快找到出路，是不是最短路径无所谓；对于很多即时策略游戏，我们追求的是路径短与速度快的双重要求。权衡游戏对于寻路的要求，找到适合游戏的寻路算法是很重要的，好比为一款即时战略游戏的每个单位都配备Dijkstra算法，这显然是不可接受的。有时可以将游戏的寻路划分层次，结合不同的寻路算法也是很多游戏的选择，比如在大地图层次，先移动到目标场景，节点较少，使用Dijkstra算法找到正确的路径，在达到目的地后转为真正的场景层次，使用A\*算法快速找到场景内某目标点的路径。

但是这些算法无法直接使用游戏层的数据，这些数据必须被放到一个特定的结构中才能使算法很好工作，而这个结构就是上述提到的非负权边的图。

* + 1. 非负权边的图

图是一种被广泛使用的数学结构，前面已经提到，图是由一定数量的节点以及节点之间的边组成。一般地，我们为两个节点之间的边增加一个数值，来代表通过两点之间的路径所产生的消耗，这样的图就称为带权值的图。当这个数值是非负的时候，该图就称为非负权边的图。

一个简单的例子，如下图：

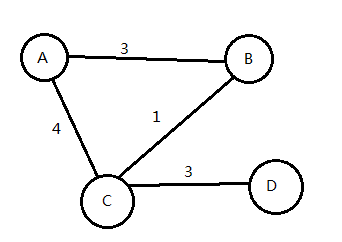


图4.1 无向非负权图

这里我们假定玩家位于A点的位置，当从A移向B时，玩家将会消耗3点的行动力，而从A移向C时，玩家将会消耗4点的行动力。图中边上标的数字就是权值。

如果图中存在负的权值边，那么该图就是带有负权边的图，比如上图BC之间的边权值由1改为-1。这时从A点经过B点移动到C点的消耗值将会变成2，如果再回到B点，消耗将变成1，如此在B点和C点之间来回往复，消耗将不断降低，这个时候将不存在到达C的最短路径(即消耗最低的路径)。上述Dijkstra和A\*算法必须使用非负权边的图的原因就是这个，具体解释将会在这些算法部分提到。

考虑这样一种情况，A点是山脚，而B点是山顶，那么从A到B和从B到A的消耗应该是明显的不同，上山要比下山消耗更多；另一种情况，A点是一个高台，B点是地面，从A跳到B后，将无法再爬上A，此时从B到A的边是不存在的。为了解决这两种情况，就有了有向图。

* + 1. 有向权图

顾名思义，有向图的边是有方向的，边AB(指从A到B)并不等于边BA，如下图：

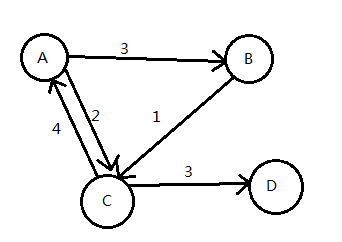


图4.2 有向权图

A点到C点消耗为2，而C点到A点消耗为4，D点无法达到C点。

有向图中出现负权边跟无向图稍稍有些不同，描述是这样的：当有向图中存在权值总和为负的循环时，此图将不存在最短路径。比如上图，如果C点到A点的权值改为-5时，从A到B到C再到A点的消耗值就变为-1，此时不存在A点到任何点的最短路径，因为A可以经过无限多个循环来创造更低的消耗再移动到目标点。Bellman-Ford 算法是处理这种带负权边的图上最短路径的寻路算法，它能检测负值循环的存在，除非源点无法达到那些循环。这种算法也会在后面具体介绍。

* 1. 几种寻路算法

提到寻路算法，不得不说Dijkstra算法和A\*算法，这两种算法作为两大类算法：解析式算法和启发式算法的代表，被广泛地应用。同样的，其他算法也被应用于各种场合，比如Bellman-Ford算法就被应用于存在负权边的图中。

* + 1. Dijkstra算法

Dijkstra算法是荷兰计算机科学家Edsger Dijkstra于1956年提出的，可以解决非负权边的图中单源最短路径问题。

1. 算法描述

从源点开始，X点的距离定义为从源点到X点的距离；

(1)为所有点的距离赋值，源点的距离为0，其他店的距离为无穷大；

(2)标记所有点为未访问状态。设定当前点为源点，将所有未访问点放入到未访问集中，除了源点；

(3)从当前点开始，找到其所有未访问的邻居点，并计算它们的暂定距离。例如：当前点A的距离为5，A有一条边连接B，其长度为4，那么B的距离就为9，意思是从源点起，某条经过A的路径达到B的消耗为9。如果存在另外一条路径，其达到B的消耗小于9，那么将B的距离改为较小的那个。此时，尽管邻居点被计算了一遍，但是它们仍然保持未访问状态；

(4)当计算完当前点的所有邻居点的距离之后，标记源点为已访问状态，从未访问集中删除，已访问的点将不会再检查，它的距离也是最小的和不再改变的；

(5)如果目标点已经被标记已访问状态，或者所有点的暂定路径中最小值是无穷大，那么算法终止；

(6)将具有最小距离的未访问点作为当前点，跳转到(3)；

1. 伪代码

从算法描述可以看出，Dijkstra算法是从源点开始不断地计算当前点周围的点，呈现一种发散型的图示，直到找到目标节点。代码实现需要借助2个数组，cost[]用于保存各节点的距离，previous[]保存达到该点的上一个节点，用于回溯生成的路径。集合Q用于存储所有未访问状态的节点，伪代码如下：

**function** Dijkstra(*Graph*, *source, target*):

**for each** vertex v in *Graph*:

cost[v] := infinity;

previous[v] := undefined;

push v in Q;

**end for;**

cost[*source*] := 0;

**while** Q is not empty:

temp := getLeastCost(Q); //get the least cost vertex in Q and remove it from Q

**if** cost[temp] = infinity:

**break**;

**end if**;

**if** temp = target:

**return**;//find the target

**for each** vertex n in getNeighbor(temp)://getNeighbor() return all the neighbor vertexes of temp not remove from Q

w := distance(n,temp);//get the distance from temp to n

**if** cost[n] > cost[temp] + w:

cost[n] := cost[temp] + w;

previous[n] := temp;

**end if**;

**end for**;

**end while**;

**return**;//not find the target

**end** Dijkstra.

通过返回目标节点和previous[]数组可以回溯出路径，代码如下：

**function** path(*target*):

S := empty sequence;

u := target;

**while** previous[u] is defined:

insert u at the front of S;

u := previous[u];

**end while**;

**return** S;

**end** path.

1. 时间复杂度

计算Dijkstra算法的运行时间上界，首先假设graph中存在E条边以及V个顶点，getLeastCost的运行时间为,distance的运行时间为，观察伪代码得知时间上界为,一般地，采用list存储Q中节点时，仅仅是线性扫描Q，此时的上界 = 。所以没有经过任何改进的Dijkstra算法的时间复杂度为，

可见节点数增加以后效率直线下降，不适合大量节点的搜索。

* + 1. A\*算法

Peter Hart、Nils Nilsson和Bertram Raphael在1968年首先提出了A\*算法，作为Dijkstra的延展。A\*算法通过使用启发式函数，有效地减少了寻路过程中检查的节点数，提高了效率。

1. 算法描述

A\*与Dijkstra最直接的不同就在于Dijkstra使用当前节点到源点的距离来判定下一个检查的节点，而A\*使用估计函数与当前距离结合来判定下一个该检查的节点。

首先A\*用于判定节点的函数称之为f(x)，它是由两部分构成：

(1)当前节点与源点之间的距离值，记为g(x)；

(2)当前点到目标点之间距离的估计值，记为h(x)；

h(x)一般采用比较常用的曼哈顿方法来判断，就是当前点与目标点之间的横纵坐标的差的绝对值之和。举例来说，假如A点的坐标为(3,4)，B点的坐标为(5,7)，C点的坐标为(1,9),A为起点，C为终点，假定目前路径经过B点，那么B点的g(B)就是|5-3|+|7-4|=5,是实际已经走过的距离，而h(B)是|1-5|+|9-7|=6，是估计从B到C需要走的距离，那么此时B点的f(B)=g(B)+h(B)=11。

h(x)必须是可接受的，也就是说，它不能对到目标点的距离估计过高，否则将导致错误的结果。算法的其他部分与Dijkstra相差不多，所以说Dijkstra就是A\*算法中h(x)为0的特殊情况。

1. 伪代码

A\*算法使用2个列表：开启列表和关闭列表。开启列表中存储着待检测的节点，而关闭列表中保存着已经检测过的节点，并且以后将不会检测它，用openset和closeset表示。伪代码如下：

**function** Astar(*source, goal*)

closeset := the empty set;

openset := {*source*};

gScore[*source*] := 0;

hScore[*source*] := getHScore(*source*, *goal*);// this function return the h(x) of the node

fScore[*source*] := gScore[*source*] + hScore[*source*];

**while** openset is not empty:

current := getLeastF(openset);//this function return the least f(x) in the openset and remove it from openset

**if** current = *goal*:

**return**;

**end if**;

push current in closeset;

**for each** vertex v in getNeighbor(current)://getNeighbor return all neighbors of the node not in the closeset

tempGScore := gScore[current] + distance(current,v);//distance return the distance between the two nodes

**if** v not in openset:

push v in openset;

hScore[v] := getHScore(v, *goal*);

tempGood := **true**;

**else if** tempGScore < gScore[v]:

tempGood := **true**;

**else**

tempGood := **false**;

**end if**;

**if** tempGood = **true**:

previous[v] := current;

gScore[v] := tempGScore;

fScore[v] := tempGScore + hScore[v];

**end if**;

**end for**;

**end while**;

**return**;

**end** Astar.

tempGood用于控制是否更新节点的g(x)和f(x)，同样地，通过previous[]可以回溯出路径，此函数与Dijkstra相同，故不再列出。

1. 时间复杂度

A\*算法的时间复杂度依赖于h(x)的实现，最差情况下，节点的扩展是指数增长的，但是当搜索空间是树形结构，只有一个目标点的时候，并且h(x)满足关系式：，此时节点扩展呈现多项式增长，这里是最优的启发函数，是x到目标点的精确消耗。

如果采用线性表结构作为开启列表与关闭列表的容器，且不对存入的节点做任何排序，那么每次检查节点时getLeastF的运行时间为，并且检查节点是否存在于开启关闭列表中同样需要，故传统的A\*算法还是有很多可改进的地方。

* + 1. BFS算法

作为另一个比较经典的搜索算法，BFS同样也被广泛地应用。BFS，其全称为breadth-first search，即广度优先搜索，顾名思义，它应用于树形结构，并且通过不断搜索父节点的所有子节点而得名。

1. 算法描述

举例说明，如下图：

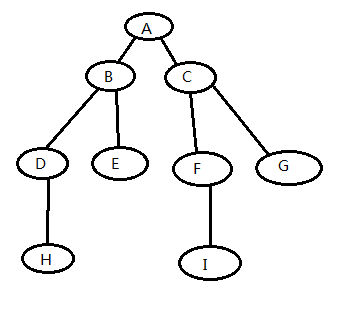


图2.3 BFS搜索实例

假定A为源节点，I为目标节点，那么从A开始，搜索A的子节点B和C，发现B和C都不是目标节点，那么继续搜索B的子节点D和E，仍然不是，继续搜索C的子节点F和G，下一步搜索D的子节点H，E没有子节点所以跳过，搜索F的子节点I，找到目标，结束。

为了实现每次检查处于树中同一层的节点的子节点后再继续下一层搜索，BFS采用队列这种先进先出的数据结构。

(1)将根节点入队列，也就是源节点；

(2)将一个节点出队列，并检查它：如果它是目标节点，那么退出搜索返回结果；否则将该节点的所有直接子节点入队列；

(3)如果队列为空，那么所有的节点都被检查过了，退出搜索返回空；

(4)如果队列不为空，跳转(2)

1. 伪代码

算法需要借助visited[]数组来记录哪些节点被检查过了，哪些还没有，Q为队列，s为根节点，t为目标节点，伪代码如下：

**function** BFS(*Graph*, *s*, *t*)

**for each** vertex v in *Graph*:

visited[v] := **false**;

**end for**;

Q := the empty queue;

enqueue *s* onto Q;

visited[*s*] := **true**;

**while** Q is not empty:

current := dequeue Q;

**if** current = *t*:

**return**;

**end if**;

**for each** child n in getChild(*Graph*, current):

**if** visited[n] = **false**:

visited[n] := **true**;

enqueue n onto Q;

previous[n] := current;

**end if**;

**end for**;

**end while**;

**end** BFS.

enqueue是将节点入队列操作，dequeue是将节点出队列操作，最后根据previous[]可回溯出路径。

1. 时间复杂度

假定图中有节点V个，边E条，那么在最坏情况下，所有的节点和边都被扩展检查到，那么时间复杂度为，考虑到V个节点最多可能存在条边，故介于和之间。由此看出，BFS的时间复杂度要好于Dijkstra算法，但是该算法并不一定能找到最短路径，除非所有的边消耗都一样，也就是图中所有的权值都一样，否则BFS只能找到步数最少的路径。通过一定的修改，可以将BFS改为可查找最短路径的算法，这将会在下一章介绍。

1. 游戏场景中寻路算法改进研究

上一章介绍了几种代表性的寻路算法，并且对其算法的运行与时间复杂度都进行了一定的说明，本章将要就A\*算法以及BFS、DFS算法配以实际的代码说明，进行改进，使其达到提高效率、降低运行时间上界和成功寻路的目的。并且将对游戏中一些寻路问题进行研究讨论。

* 1. 寻路环境设计

在真正的算法改进之前，需要先对寻路环境进行设计，以方便寻路算法的代码实现。我们首先假定寻路的地图是矩形的，由很多个小方块组成，每个方块代表寻路中的一个节点。这样设计是为了寻路的方便，而且不会对算法产生不利影响。寻路是八方向的，也就是说图中每个小方块都可以向周围八个方块扩展，如下图：

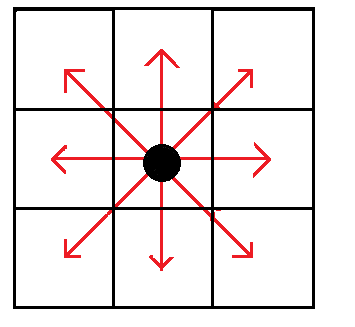


图3.1 八方向扩展实例

把每个小方块看作一个节点，其位于矩形的第几行第几列就是节点的坐标；有时地图中会存在障碍物，所以对于每个节点有个布尔变量控制其是否可以通过；又由于另外维护previous数组不如直接把其与节点结合方便，所以又定义一个指针变量存储其前一个节点的指针，这就是节点包含的基本信息，结构体如下：

struct Node

{

int postionX,postionY;

bool achiveble;

Node\* previous;

};

typedef Node \*pNode;

关于权值，这里采用高度图的设计思想，即赋予每个小方块一个高度值，从一个方块移动到另一个方块的耗费就是这两个方块的高度差。这种设计的优点就是非常方便，易于管理，只要定义一个二维数组即可，缺点是对于A到B和B到A权值不同的寻路时一个高度图就无法满足了，但是现在暂时不考虑这种情况。从上图可以看出，一次斜向移动其实比一次横竖向移动距离要长，故对于斜向移动，耗费值=两点高度差\*1.4，这里模仿的是正方形对角线长度为边长的倍，更加贴近真实。

* 1. A\*算法的研究及改进

由于传统的A\*算法的运行效率不是特别理想，并且A\*算法的应用非常广泛，故对A\*算法进行了改进研究。

* + 1. 传统A\*算法代码分析

由于需要对开启列表插入删除操作，故采用了std::list这种数据结构，节点的存储采用的是顺序存储，即新加入的节点存储在list的末尾。前面说过，算法最耗时的部分，也就是找出f(x)最小的节点和判断节点是否存在于列表中这两部分。由于采用顺序存储，故这两部分的实现非常简单，顺序比较列表中节点即可。取最小f(x)代码如下：

list<pNode>::iterator current = openList.begin();

for (list<pNode>::iterator it=openList.begin();it!=openList.end();it++)

{

if(fScore[(\*current)->postionX][(\*current)->postionY]>fScore[(\*it)->postionX][(\*it)->postionY])

{

current = it;

}

}

判断节点函数如下：

bool findNode(pNode node,list<pNode> nodeList)

{

if(nodeList.empty())

return false;

for (list<pNode>::iterator it=nodeList.begin();it!=nodeList.end();it++)

{

if(node->postionX==(\*it)->postionX && node->postionY==(\*it)->postionY)

return true;

}

return false;

}

由代码易知时间复杂度是。最好情况是，最差情况是，平均时间复杂度为。

使用算法测试系统实验结果如下：

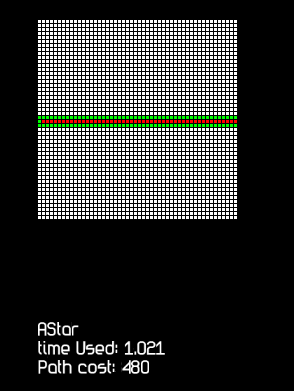


图3.2 传统A\*算法实验结果

此系统为研究算法配套测试系统，主要用于测试算法结果是否正确以及耗费时间，详细说明在第五章会说到。图中白色代表可移动区域，黑色小方块代表不可通过区域(该图中没有)，起点位于左边，终点在右边，绿色是算法扩展到的点，也就是检查过的点，红色为从起点到终点的结果路径。由图可知，传统A\*算法寻路耗费了1.021秒的时间。

* + 1. A\*算法初步改进

首先针对上述问题，提出改进措施，最容易想到的就是把顺序存储改为插入排序，按照f(x)由小到大排列，这样取f(x)最小的点就是开启列表中的首元素。插入排序代码如下：

void insert\_sort(list<pNode>& nodeSet, pNode node,int fScore[width][length])

{

list<pNode>::iterator temp = nodeSet.end();

for (list<pNode>::reverse\_iterator rit=nodeSet.rbegin();rit!=nodeSet.rend();rit++)

{

if(fScore[(\*rit)->postionX][(\*rit)->postionY] > fScore[node->postionX][node->postionY])

temp--;

else

break;

}

nodeSet.insert(temp,node);

}

这样只要从openList中取出第一个元素即可，代码如下：

pNode removeLeastNode(list<pNode>& nodeSet)

{

if(nodeSet.empty())

return NULL;

pNode leastNode = nodeSet.front();

nodeSet.pop\_front();

return leastNode;

}

这段代码结合了移除首元素的部分，由于使用的是std::list，所以插入删除元素并不会造成后面元素的向前向后移动，效率非常高。有一点需要注意，就是当节点发生更新时，也就是当发现从A到B比从C到B更加节省消耗时，就需要把B的前置节点给为A，且gScore和fScore都需要改变。此时由于fScore发生了改变，那么该节点处于开启列表中的位置就需要更新，所以需要一个更新操作，遍历开启列表，找到该节点位于开启列表的位置，拿出来，然后找到现在适合新fScore的位置，再插入。

由于插入排序仅仅被使用在了寻找f(x)最小的地方，且又增加了更新操作，故这次改进对于运行时间并没有明显的改善。经过测试，由于电脑性能的波动，发现有时运行时间甚至比传统A\*算法要低，如下图：

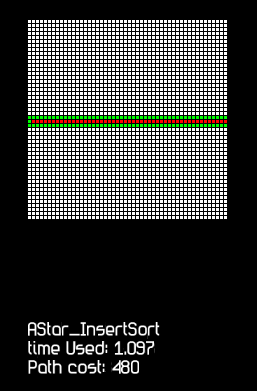


图3.3 使用插入排序的A\*算法实验结果

此次改进效果不明显，但是对后面的进一步改进给出了启发，那就是对于列表中节点的搜索也不能仅仅通过顺序遍历来实现，需要利用已有的排列顺序来降低时间消耗。

* + 1. A\*算法进一步改进

前面提到插入排序虽然利用list实现，降低了运行时间上界，但是时间复杂度依然保持在了。通过更加效率的方法——二分插入，可以明显降低时间复杂度。但是有一点需要注意，二分插入会对列表进行随机访问，而std::list不支持随机访问，故选择std::vector来存储节点。vector采用线性存储结构，故对其进行插入删除操作会造成元素的整体移动，但是研究发现vector由于其设计原因，每次分配空间时都会扩展两倍大小空间，所以插入新元素时大多数情况下不需要申请空间，故插入位置后元素的移动不会降低态太多，可以接受。删除操作只发生在取fScore最小节点时，所以将vector中节点按从大到小排列，每次取节点都发生在vector的结尾处，这样就不会造成其它节点的前向移动了。但是为了最大限度降低元素的整体移动，如果新插入节点的fScore与列表中已有节点的fScore相同，那么新插入节点将会放在相同节点的后面，这样能保证相同的节点不发生向后移动。二分插入代码如下：

void BinaryInsert(vector<pNode>& nodeList,pNode node,int score[width][length])

{

int min = 0,max = nodeList.size()-1;

int mid = -1;

while (min <= max)

{

mid = (min+max)>>1;

if(score[nodeList[mid]->postionX][nodeList[mid]->postionY] == score[node->postionX][node->postionY])

{

int temp = mid;

while(score[nodeList[temp]->postionX][nodeList[temp]->postionY] == score[node->postionX][node->postionY])

{

if(temp < nodeList.size()-1)

temp++;

else

break;

}

nodeList.insert(nodeList.begin()+temp,node);

return;

}

if(score[nodeList[mid]->postionX][nodeList[mid]->postionY] > score[node->postionX][node->postionY])

min = mid+1;

else

max = mid - 1;

}

if(mid==-1)

{

nodeList.push\_back(node);

return;

}

if(score[nodeList[mid]->postionX][nodeList[mid]->postionY] > score[node->postionX][node->postionY])

nodeList.insert(nodeList.begin()+mid+1,node);

else

nodeList.insert(nodeList.begin()+mid,node);

}

如果mid等于-1，说明插入位置位于列表的结尾，故直接使用push\_back即可。由插入节点使用二分插入自然联想到判断节点是否存在于列表中同样可以使用二分查找的方法来提高效率。首先根据fScore查找，如果列表中不存在这样的fScore则表示没有该节点，如果存在，那么还应该比较位置，因为列表中可能存在不止一个fScore相同的节点。由于使用二分查找来定位，故最终查到的点相连的前面和后面都有可能存在和该点fScore相同的点，这时需要一一对比位置信息来判断。其他部分代码与二分插入大体相同，下面是前向后向查找代码：

if(score[nodeList[mid]->postionX][nodeList[mid]->postionY] == score[node->postionX][node->postionY])

{

int temp = mid;

// look forward

while(score[nodeList[temp]->postionX][nodeList[temp]->postionY] == score[node->postionX][node->postionY])

{

if(nodeList[temp]->postionX==node->postionX && nodeList[temp]->postionY==node->postionY)

return temp;

if(temp > 0)

temp--;

else

break;

}

// look backward

if(temp < nodeList.size()-1)

temp = mid+1;

else

return -1;

while(score[nodeList[temp]->postionX][nodeList[temp]->postionY] == score[node->postionX][node->postionY])

{

if(nodeList[temp]->postionX==node->postionX && nodeList[temp]->postionY==node->postionY)

return temp;

if(temp < nodeList.size()-1)

temp++;

else

break;

}

return -1;

}

节点的fScore发生更新时，对于vector和list的操作也不同，我们需要先移除并保存将要更新的节点，然后再查找新插入位置，并插入之。

使用二分插入和二分查找，时间复杂度由以前的变为现在的，这次改进后，运行结果如下图：

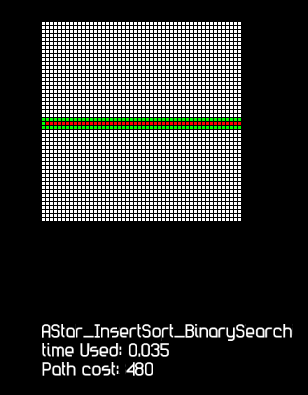


图3.4 进一步改进的A\*算法实验结果

从图中可以看出，这次改进后，运行时间得到了明显的降低，从原来的1秒多下降到了只要0.035秒，效率提升了接近30倍，所以这次的改进还是比较成功的，但是效率能否进一步提升呢，因为vector这种容器虽然设计良好，但是线性的存储结构还是无法适应大量的中间插入删除操作，尤其是发生在vector头部的插入删除。另外，我们不需要一个完整的排好序的列表，我们仅仅需要fScore最小的元素排列在我们知道的位置，那么其他地方就不需要花费很多时间进行排序了。由这些改进点，提出更好的改进方法，那就是使用全新的容器——二叉堆。

* + 1. 使用二叉堆

1. 什么是二叉堆

二叉堆是这样一种数据结构，堆中最大或者最小的元素处于堆顶，就像二叉树一样，堆顶就是根节点，它有左右两个子节点，子节点又可以有自己的子节点，其他元素按照某种关系处于这些子节点的位置。举例来说，有一组数字：3、8、1、4、7、2、9，现在按照二叉堆的方式来排列它们，选择最小的值排列在堆顶，那么1就处于堆顶位置。上面说得某种关系就是每个节点的子节点都和它们的父节点相等或者略高，那么这组数字的一种可能排列就是：

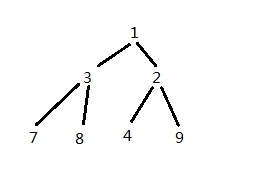


图3.5 二叉堆排列

1的两个子节点3和2都大于1,3的两个子节点7和8都大于3,2的两个子节点4和9都大于2，所以这种排列是符合二叉堆的。排列的方式不止一种，只要符合二叉堆的要求就行。一般我们使用一维数组来存储这个排列，数组中堆顶元素处于数组下标为1的位置，而不是数组第一个位置，原因是这样有利于后面的元素排列。任何元素的左子节点的位置都是处于一维数组中它们父节点元素位置乘以2的位置，右子节点则是左子节点位置加1。那么上图二叉堆的一维数组排列是：

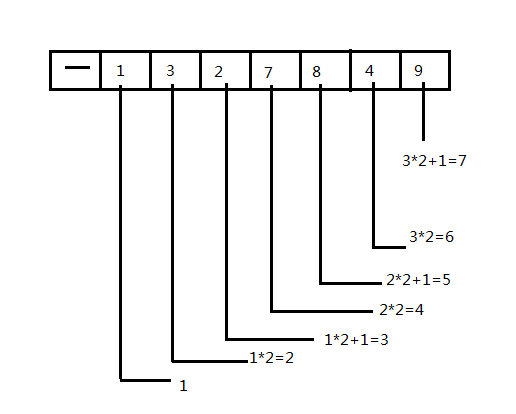


图3.6 二叉堆的一维数组排列

为了方便使用，实验系统将二叉堆封装成一个类BinaryHeap，同样为了便于操作，类使用了布尔类型的二维数组isInHeap[width][length]来记录节点是否存在于二叉堆中。

1. 二叉堆的操作

要把二叉堆作为节点的容器，就需要知道二叉堆的操作方法：插入、删除以及更新。

1. 插入

当有新元素需要插入到二叉堆时，首先把它放到数组的末尾位置，然后把它和它的父节点比较大小，也就是当前位置的二分之一位置的元素，如果新元素更大那么就将它放在数组的末尾，如果新元素比其父节点要小，那么就交换这两个元素。依次重复该操作，直到它不再比其父节点更小为止或者它已经处于数组顶端。

举例来说：往1、3、2、7、8、4、9中插入元素2，首先将2放到数组尾部，变成1、3、2、7、8、4、9、2，它的位置现在是8(数组的0位置被占用，不存储元素)，那么它的父节点就是位置4的7，2比7小，故交换它们，变成了1、3、2、2、8、4、9、7。继续比较，它的父节点现在是位置2的3,2比3小，故交换它们，变成了1、2、2、3、8、4、9、7。继续跟它的父节点比较，现在父节点是1,2比1大，比较停止。

实验系统使用fScore作为比较的依据，故二叉堆需要传递fScore作为参数，伪代码如下：

**function** push(*node*, *fScore*):

push *node* to the back of heap;//heap represents the array

isInHeap[*node*] := **true**;//record node has been push into the heap

num := heap.size()-1;//the position of node

**while** num<>1:

harfPosition := num/2;

**if** *fscore*[num] < *fscore*[harfPosition]:

swap(heap[num],heap[harfPostion]);

**else**

**break**;

**end if**;

**end while**;

**end** push.

从代码可以看出，插入操作最好情况是不用swap操作，而最差情况是执行次swap操作，平均时间复杂度就是，T是swap操作所占用的时间，那么降低swap所用时间将会对插入操作的效率起很大作用。这里交换的是两个节点，如果节点占用的内存空间很大，那么将会非常费时，但是只交换节点的指针的话，交换的内存就只有四个字节，效率就会提升不少。故heap中存储的都是节点的指针信息，而不是真正的节点。

1. 删除

从堆中把节点删除类似队列的pop操作，但是内部还需要一些变换。首先删除位置是1的节点，然后将数组中最后一个元素移动位置1中；接着比较此元素和它的两个子节点比较，如果它比两个子节点都小，就保持不变，否则，就把它和较小的子节点交换位置，不断重复比较交换的步骤，直到它达到数组的末尾或者它比两个子节点都小为止。

举例来说：数组1、2、2、3、8、4、9、7，对它进行删除操作，1被移除，数组末尾的7被移到了1的位置，变成了7、2、2、3、8、4、9。现在拿7和它的两个子节点2和2比较，很明显7更大，而它的两个子节点同样大，故取第一个子节点和7交换，变成了2、7、2、3、8、4、9。现在继续拿7和它两个子节点3和8比较，7比它的左子节点小，故交换之，变成了2、3、2、7、8、4、9，现在位置为4的7已经没有子节点了，故操作停止。

删除操作还需要清楚该节点在堆中的记录，由于需要比较fScore，故fScore作为参数传递，伪代码如下：

**function** pop(*fScore*):

**if** heap.size() -1 < 0:

**return**;

**end if**;

isInHeap[1] := **false**;

heap[1] := heap's last element;

pop heap's last element;

fatherIndex := 1;

**while true**:

childIndex := fatherIndex;

**if** 2\*chidIndex+1 <= heap.size()-1://both children exist

**if** *fScore*[childIndex] > *fScore*[childIndex\*2]://father's fScore greater than left child

fatherIndex := childIndex\*2;

**end if**;

**if** *fScore*[fatherIndex] > *fScore*[childIndex\*2+1]://father or left child's fScore greater than right child

fatherIndex := childIndex\*2+1;

**end if**;

**else if** 2\*childIndex <= heap.size()-1://only left child exists

**if** *fScore*[childIndex] > *fScore*[childIndex\*2]:

fatherIndex := childIndex\*2;

**end if**;

**end if**;

**if** fatherIndex <> childIndex://father's fScore greather than one or both children

swap(heap[fatherIndex], heap[childIndex]);

**else**

**break**;

**end if**;

**end while**;

**end** pop.

这里fatherIndex代表的是父节点将要交换到的位置，父子节点一共只有三种情况：没有子节点、有两个子节点、只有左子节点，而不可能出现只有右子节点的情况，这是由于插入操作决定的。删除操作最好情况下不需要进行swap操作，最坏情况下需要进行次swap，平均时间复杂度为。同样地，堆存储节点的指针以提高swap效率。

1. 更新

跟使用排过序的列表一样，如果列表中节点的fScore发生了变化，则需要更新其在列表中的位置。先是查找该点在堆中的位置，这个很不幸，只能通过依次遍历的方法来查找，由于堆中元素不是完全有序地排列，故不能使用二分查找。查找到目标后，替换成新的fScore，再像插入元素一样，依次和父节点比较并交换。因为fScore发生更新只可能是由于其fScore变小，所以只要跟父节点比较即可。

举例来说：2、3、2、7、8、4、9这个数组，当发现7变成了1时，首先遍历该数组，找到发生改变的7的位置，替换成1，然后,和它的父节点3比较，1比3要小，故交换它们，变成了2、1、2、3、8、4、9，此时继续拿1和它的父节点2比较，1比2小，故交换它们，变成了1、2、2、3、8、4、9，此时1处于堆顶，故操作停止。

同样地，需要fScore作为参数，伪代码如下：

**function** update(*fScore*):

index := 1;//record the node's position which changed

**for** index **from** 1 **to** heap.size()-1:

**if** heap[index] = node:

**break**;

**end if**;

**end for**;

**while** index<>1:

harfPosition := index/2;

**if** *fScore*[index] < *fScore*[harfPosition]:

swap(heap[index],heap[harfPosition]);

**else**

**break**;

**end if**;

**end while**;

**end** update.

由代码可知，其实二叉堆的更新操作使用vector的二分A\*算法要高效，因为它们都需要遍历一段节点，当找到目标后，二分A\*需要对vector进行一次删除插入操作，而且是发生在vector的中部，而相应地，二叉堆A\*只需要进行一次swap操作。

1. 在A\*算法中应用二叉堆

前面已经解决了二叉堆的插入删除以及更新操作，那么现在只需要将A\*算法的开启列表的容器更换成二叉堆即可。在原来需要调用BinaryInsert、removeLeastNode以及updateNode的地方改成调用二叉堆的函数push、pop以及update。在判断节点是否存在于开启列表中时，现在不需要再去列表中遍历查找，而是仅仅调用二叉堆的函数find来查看相应的布尔值即可：

bool find(const pNode& node)

{

return isInHeap[node->postionX][node->postionY];

}

由此启发，在查看扩展出的邻居节点是否存在关闭列表中时，也可以通过这种增加空间消耗来降低时间消耗的方法，定义一个布尔类型的二维数组isInCloseList，初始化为全false，当有节点被放入关闭列表时，数组相应的布尔值被改写成true。在查询节点时只要查询相应的布尔值即可。

使用二叉堆A\*算法的运行结果如下图：

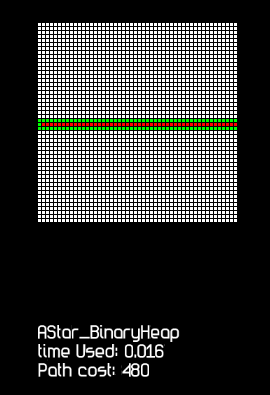


图3.7 二叉堆A\*算法实验结果

由图中可以看出，使用二叉堆后，运行时间几乎又降低了一倍，由0.035秒下降到了0.016秒。虽然使用二叉堆时需要手动实现复杂的插入删除更新操作，但是这对于效率地提升来说是值得的。当查询节点变多时，二叉堆A\*的效率提升地更多，而当地图中节点很少，比如只有不到100个时，使用二叉堆的意义不大，因为节点太少，二叉堆的优势无法体现。

对于A\*算法的改进就提这么多，接下来就前面提出的BFS算法，如何让它能查找最短路径给出解决办法。

* 1. BFS的改进研究

在第二章时介绍BFS时说过，只有地图中所有路径的消耗(权值)都一样，BFS才能找到最短的路径，否则只能找到步数最少的路径。例如下图：

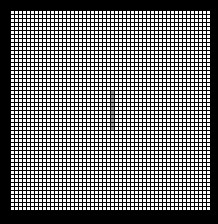


图3.8 BFS地图实例

地图中间灰色区域的高度为100，而其他白色区域的高度都为0，起点和终点跟以前一样保持不变，此时地图上的消耗就是不同的，如果是BFS的话，它只会寻找最少步数的路径，按照四方向的移动规则，结果如下图：

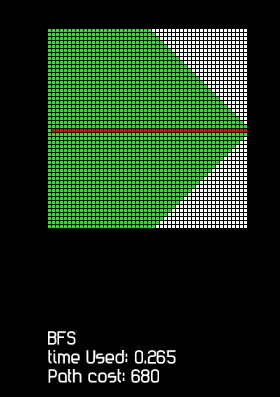


图3.9 不同消耗下BFS寻路结果

可见BFS寻路下，路径无视了中间消耗较大的方块，直接穿过，从而形成了移动步数最少的直线路径。但这不是寻路需要的，故要对BFS进行修改，办法就是将消耗变成寻路的因子，让它能决定下一个待检查的节点。换句话说，就是把消耗最少的那个节点放到队列中的最前端。std中的优先队列priority\_queue正好可以满足这个要求。

优先队列是这样一种队列，它会把优先级最高的元素排列在队列头以便于取出。为了让优先队列应用于BFS算法，需要为节点定义优先级，而最简单的方法就是将节点的距离的负数设为优先级。这样，从起点起到目前为止总消耗最低的那个节点的优先级将会最高，会在下一次被检测到。

整个算法看起来有点像是Dijkstra算法，但是两者不一样，前者需要在寻路的过程中不断地往队列中插入新的节点、更新优先级并且移除检测过的节点，而Dijkstra则是不断地移除节点的遍历操作。

关于优先队列的更新操作，有一点需要注意，那就是不能直接更改优先队列里的优先级，因为优先队列里的节点都是事先按照各自的优先级排序的，如果后面更改了某节点的优先级，队列并不能察觉到这一改变，节点的位置也就不能得到更新，而且使用std::priority\_queue时进行这种改变还会发生invaild heap异常。这个时候就需要取出原队列中的节点，并按照最新的优先级重新放入新的优先队列中，虽然这样比较低效，但是这是唯一有效的方法，相关代码如下：

priority\_queue<priorityNode> BFS\_Priority::updateQueue(int newPriority,priorityNode pronode,priority\_queue<priorityNode>& origQueue)

{

if(pronode.node->postionX==origQueue.top().node->postionX && pronode.node->postionY==origQueue.top().node->postionY)

{

pronode.node->priority = newPriority;

return origQueue;

}

priority\_queue<priorityNode> newQueue;

while (!origQueue.empty())

{

priorityNode temp = origQueue.top();

origQueue.pop();

if(temp.node->postionX!= pronode.node->postionX || temp.node->postionY!=pronode.node->postionY)

newQueue.push(temp);

else

{

temp.node->priority = newPriority;

newQueue.push(temp);

}

}

return newQueue;

}

函数一开始的比较是为了当队列头元素的优先级发生变化时避免重构优先队列，从而提高效率。经此修改之后，使用优先队列的BFS将能在不同消耗的地图上查找最短路径，结果如下：

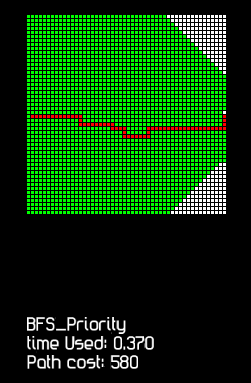


图3.10 使用优先队列的BFS实验结果

由图可以看出，路径绕开了中间消耗高的那段，从而使路径消耗降低了100。时间使用上虽然上升了一些，但是以效率地小幅下降为代价换来的正确寻路是值得的。

* 1. DFS的改进研究

将BFS使用的队列改成堆之后，就形成了DFS算法。它是深度优先算法，与BFS相对应，它是把每个节点的子节点都遍历完以后再去遍历另一个节点。它的伪代码如下：

**function** DFS(*Graph*, *start*):

visited[*start*] := **true**

**for each** neighbor in getNeighbor(*Graph*, start):

**if** visited[neighbor] = **false**:

previous[neighbor] := start;

DFS(*Graph*, neighbor);

**end if**;

**end for**;

**end** DFS.

在八方向寻路的地图中，每个节点相当于有八个子节点，在这样相互连通的地图中进行寻路，DFS有个非常大的缺陷，就是一旦一开始向错误的方向扩展，那么除非走到死路，否则将会将错误的地方都探索一遍，得出的路径也就非常的冗长，举例来说，如果getNeighbor函数优先返回当前节点上面的节点，那么起点终点位置不变，用DFS进行寻路的结果如下图所示：

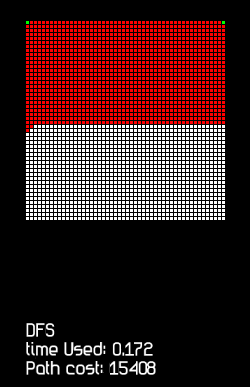


图3.11 DFS寻路结果

DFS最终得到的路径几乎走遍了扩展的节点，使最终路径非常长，这并不是想要的结果。如果在寻路过程中能指引DFS的寻路方向，那么就可以少走很多弯路，最直接的办法就是实时更改getNeighbor返回的节点的顺序。思路是当目标节点位于当前点右下角时，getNeighbor优先返回右下角的邻居，其次返回右边和下边的邻居，最后返回左上角的邻居，这样就能对DFS的寻路方向进行指引。

首先定义代表方向的标志位

enum

{

EAST = 0x00000001,

WEST = 0x00000010,

NORTH = 0x00000100,

SOUTH = 0x00001000,

EASTNORTH = 0x00010000,

EASTSOUTH = 0x00100000,

WESTNORTH = 0x01000000,

WESTSOUTH = 0x10000000,

};

其次计算当前点坐标与终点的位置关系，返回相应的方向，getNeighbor再根据方向对八个邻居进行排序，这就是有方向指引的DFS，其寻路结果如下图：

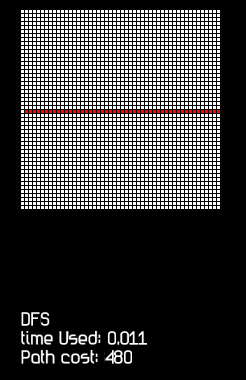


图3.12 有方向指引的DFS寻路结果

由此可见，此次寻路扩展的节点一点都没有浪费，甚至比二叉堆的A\*算法还要迅速。

* 1. 实际问题的改进研究

前面对算法进行了改进研究，现在将就游戏中实际出现的问题进行分析研究。

* + 1. 对于八方向的地图寻路

当一张地图可以进行八方向寻路时，如果仅仅使用曼哈顿距离的A\*算法将很有可能得不到最短路径，比如下面两张图，使用优先队列的BFS可以得到最短路径，而使用曼哈顿距离的二叉堆A\*算法则没有得到最短路径。

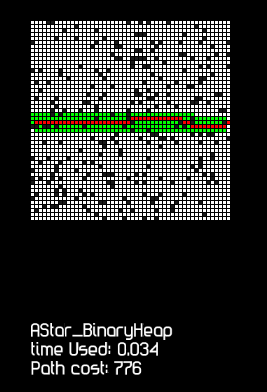
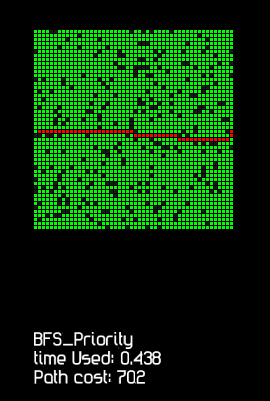


图3.13 图3.14 点阵图寻路对比

这是由于传统的曼哈顿只计算了横纵距离和，这在四方向寻路中是正确的，而在八方向寻路中由于斜向移动的存在，且一次斜向移动的消耗按照实际应该等于1.4倍的一次横向或者纵向移动，h(x)就必须要体现出对斜向移动的鼓励，因为一次斜向移动的效果等于一次横向加上一次纵向移动，而消耗要少。比如现在处于点(1,2)，而目标点是(5,4)，如果是按照最小消耗来预判的话，应该是先斜向移动到(3,4)，再横向移动到(5,4)，假定一次横纵向移动消耗为10，这样预计消耗为2\*14+2\*10=48。如果按照传统的曼哈顿距离来计算的话，就是总共4次横向移动加上2次纵向移动，总消耗为60。故这影响了后续节点的判断。

对于八方向寻路，需要把曼哈顿距离也改成增加斜向移动部分。当前点与终点之间的水平距离记为w，竖直距离记为h，斜向部分就是当前点与终点之间的水平竖直距离中较小的那个，记为L，当前点朝着目标点移动了L个斜向单位后，接下来只要再移动个单位的直向单位即可。如下图：

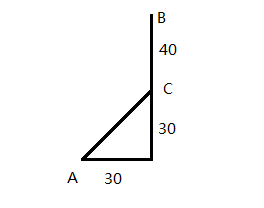


图3.15 八方向移动h(x)预判图示

从A到B，先斜向移动到C，再移动到B才是正确的预判路径。

* + 1. 对于不同消耗地图的研究

对于不同消耗的地图，比如战棋类游戏中的地图，单位移动到平原上消耗要比移动到山地中低，此时需要对h(x)函数进行重新设计使其在找到最短路径与减少扩展点数之间进行平衡。

前面的测试中，绝大部分都是采用最简单的统一权值地图，所以小方块的高度都是一样的，也就是说移动过程中消耗是线性递增的。A\*算法中x点到终点的h(x)必须要小于等于x点到终点的实际消耗才能保证A\*算法一定能找到最短路径，而当h(x)大于实际消耗时查找的节点数减少但是不一定能找到最短路径。启发式函数h(x)越大，扩展节点越少，越能快速地找到目标，但是对于消耗的判断力就越低，h(x)越小，扩展节点就越多，查询越慢速，但是查找最短路径的准确率就越高，h(x)=0，就转变成了Dijkstra算法。考虑下面的地图，所有的方块的高度都是随机的，从0到200不等，首先采用八方向的曼哈顿方法作为h(x)来寻路，结果如下图：

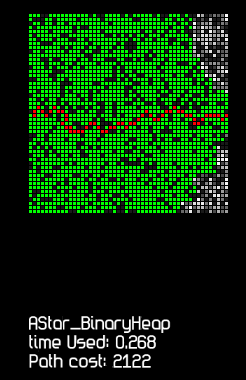


图3.16 全随机地图下A\*寻路

可以看出，此次寻路扩展了几乎地图上的所有节点，这是由于使用了八方向的曼哈顿距离作为h(x)，它远小于节点到终点的实际消耗，但是这样也保证它找到了最短路径。观察一下时间，这次寻路使用了0.268秒，如此多节点的扩展使其效率下降了10倍左右。如果人为地将h(x)扩大5倍，再寻路一次，结果如下图：

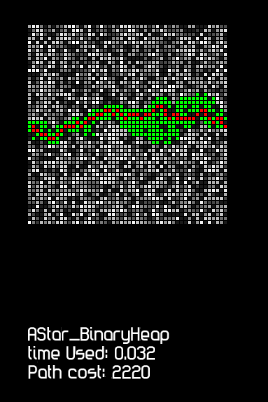


图3.17 全随机地图下A\*寻路二

很明显地，扩展的节点数减少了很多，几乎没有什么浪费就找到了路径，而且寻路时间相比刚才也下降了很多，但是这却不是最短路径。如何使A\*能找到最短路径而又扩展节点最少，这就需要研究寻路的地图，找出其权值变化的特点，从而设计出符合该地图的h(x)，但是上图采用的是全随机地图，故几乎无法找到规律，而正常的游戏场景不会出现这样的地图。考虑这样一张地图，左半部分是平原地图，中间出现了高原，后半部分又是平原，高原的高度为100，平原高度为0，如下图所示：

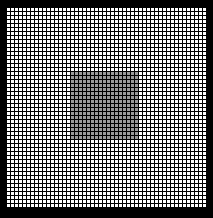


图3.18 设想的游戏地形

起点和终点依然保持不变。此地图的权值变化规律还是比较明显的，横向坐标在时，消耗值增加为110(高度差为100，而固定消耗值为10)，而在其他地方，消耗值都为10，如果是沿着直线从起点到终点，那么总消耗是680，按照八方向的曼哈顿距离，起点的h(x)是440，相比680要小220。但是680并不一定是最终的最佳路径消耗，事先通过寻路得知最佳路径是绕过中间高原的路径，其消耗为544，那么为了减少扩展点又能得到最短路径，就需要是新的h(x)在各节点中符合544消耗的最佳路径的预判值。但是一开始是无法得知最佳路径的，所以可以采用靠近法，为原有的h(x)增加一个因子，最简单的方法就是形如，由于本项目为了效率采用了整型的h(x)值，故取λ=2。结果如下图：

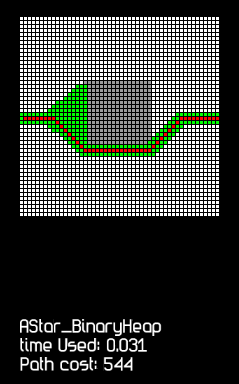
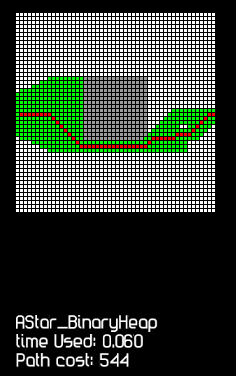


图3.19 λ=1的寻路结果 图3.20 λ=2的寻路结果

可以看出，采用新的h(x)后，扩展点数确实要比原来少了很多，而且路径更加的平滑，但是此方法有一定风险，因为起点处h(x)值大于544，从而可能无法得到最短路径。使用分段函数来设计h(x)效果会更好，但是地形再复杂一点的话，这就会大大增加h(x)的工作量，对于需要实时计算寻路信息的游戏来说不合适。有时，游戏响应时间会比是否是最佳结果更加重要，这个时候，适当的增加h(x)的值非常有用，比如再让λ变大，只要得到的路径还处于可接受范围，这就是求解次佳解的过程。

讨论完算法改进之后，现在要做的就是设计专用的测试系统，为了达到对这些算法一一对比测试的目的。

1. 寻路算法测试系统设计实现
   1. 需求分析

作为一个算法测试系统，应该具备以下功能：

(1)最主要的功能就是可以对算法进行测试并且给出测试结果;

(2)因为测试的算法有多种，那么如何灵活地将各种算法应用于同一个寻路场景中也是非常重要的，这就牵扯到将算法的输入输出统一化的问题；

(3)场景也不能是唯一不变的，可以自由定义不同的寻路场景也是应当具备的功能；

(4)应该直观地表现寻路的过程以及最终的结果，通过直观的图形界面体现出来；

* 1. 测试系统的设计实现

系统将围绕上述要求逐条进行完善设计，首先定义一个原型系统，该系统只包含一部分，那就是寻路算法的运行，具体描述如下：原型系统只拥有静态的地形图，即地形数据定义在静态数组之中，整个系统只能运行一次，属于命令行程序，算法被定义成全局函数被程序主函数所调用，无图形结果。

明显地，该原型系统无法达到上述功能要求，接下来对该原型系统进行一步步改进使其成为最终算法测试系统。系统开发工具为：Microsoft Visual Studio 2010以及第三方游戏引擎HGE。

* + 1. 对核心功能的支持

核心功能就是对算法进行测试并给出测试结果，原型系统可以调用算法的全局函数，对定义好的静态地形数据进行测试，但是由于命令行程序的局限性，无法观察到寻路的过程和结果，或者通过文本展现的测试结果过于简单，不能形象描述寻路过程。

此处需要将原型系统改为图形界面程序，同时为了方便，使用图像引擎作为测试过程与结果的绘制工具。这里使用HGE可以有效地生成图形化的程序，然后HGE默认的系统架构具有很大的问题，那就是功能模块杂糅在一起，系统使用的全局变量过多，项目规模变大以后很难维护。这里采取的措施是定义一个Application类，该类为单件，也就是全局只能有一个实现。Application类主要有五大功能模块，第一个是系统初始化模块，该模块用于初始化图形化程序部分、部分私有成员的初始化操作以及载入资源；第二个是算法配置模块，用于系统测试算法参数配置，比如设置寻路的起点与终点等信息；第三个是算法运行模块，此模块用于算法测试，并且将结果输出到下一个模块结果展示模块；第四个结果展示模块主要用于不断接受测试模块传递的各种数据，并且绘制在屏幕上，并对测试结果进行一定程度的分析；最后一个模块是清理模块，用于资源的释放等操作。

* + 1. 多种算法支持

本系统对以下算法进行测试：Dijkstra算法、传统A\*算法、使用插入排序的A\*算法、使用二分插入的A\*算法、使用二叉堆的A\*算法、BFS算法、DFS算法和使用优先队列的BFS算法。由于测试算法较多，而程序的图形显示大小有限，且为了系统的灵活性考虑，没有让系统定死必须同时运行所有算法，而是采用了前期配置的方法，允许在测试前自由配置此次测试需要使用的算法，配置的算法种类与数量都可以自由设定。

为了实现这一目标，有两种方法：一种是在Application类中将所有算法封装成成员函数，并且定义一组标志变量，和算法一一对应，当某算法被配置时，标志变量被设置，在算法运行模块中，先检测标志变量，然后调用相应的算法进行检测。这样的实现思路比较简单，但是缺点是要让Application类来处理算法数据并且保存测试结果，对于不同的算法需要定义不同的成员变量来保存其测试结果，如果配置了几个同样的算法，那么还需要将这些变量定义成数组。但是系统事先不知道将会有几种算法被配置，每种算法有几个被配置，所以这样的设定会造成Application类拥有大量的额外处理代码，与其设计目的不符。

另外一种方法是将算法封装成类，包括算法的运行，结果的处理都是类的成员函数，Application只要调用它们相应的函数即可。为了满足动态配置算法的要求，将所有的算法定义成同一个算法基类Algorithm的继承类，然后在Application中定义一个Algorithm类型的指针数组。当Application在算法配置模块每接收到一个算法被配置时，该数组就扩展一次大小并根据新配置的算法类型生成一个该算法的指针存入数组中。

明显采用第二种方法更加符合面向对象的程序设计要求，也符合系统设计要求。但是为了系统的性能考虑，对于可配置算法数量进行了限定，在初始化时即生成了指针数组的大小：

algorithms = new Algorithm\*[\_Algorithm\_Count];

识别算法类型需要定义的枚举类型如下：

enum

{

\_Dijkstra,

\_Astar,

\_Astar\_BinaryHeap,

\_Astar\_InsertSort,

\_Astar\_InsertSort\_BinarySearch,

\_BFS,

\_BFS\_Priority,

\_DFS,

\_Algorithms,

\_Algorithm\_Count = \_Algorithms

};

配置算法时根据算法类型进行算法配置的实现如下：

inline void setTestAlgorithm(int algorithm)

{

switch(algorithm)

{

case \_Dijkstra:

algorithms[numOfTest] = new Dijkstra();

algorithms[numOfTest]->setType(\_Dijkstra);

numOfTest++;

break;

case \_Astar:

algorithms[numOfTest] = new AStar();

algorithms[numOfTest]->setType(\_Astar);

numOfTest++;

break;

...

default:

break;

}

}

其中numOfTest代表系统中已经配置算法的数量，setType函数用于将每个被配置的算法标记类型，因为所有的算法都是由基类Algorithm指针强转而成，所以它们无法知道自己的类型，不定义标记的话，就无法正确地调用属于子类的函数。

不同算法子类所共同的函数定义在基类Algorithm中，比如地形的初始化，资源的初始化以及释放，绘制寻路结果，计算消耗，设定算法类型等。算法的运行过程定义在成员函数run()之中，而基类的run()则是简单的为空，它被所有子类算法重定义。子类算法寻路时使用的地图信息以及存储的节点信息全部都继承自基类，为了展示寻路的过程，子类同样将曾经扩展的节点信息存储在父类的list类型的成员变量history中。

当算法测试模块开始运行时，只要根据numOfTest的值进行for循环，每次对algorithms中存储的算法指针进行调用getType以判断它是哪种算法，从而对其进行强转，调用相应的子类函数，部分实现如下：

for (int i=0;i<numOfTest;i++)

{

switch(algorithms[i]->getType())

{

case \_Dijkstra:

((Dijkstra\*)algorithms[i])->run();

break;

case \_Astar:

((AStar\*)algorithms[i])->run();

break;

...

}

* + 1. 不同场景的设置

不同的寻路地图，算法的变现是有所不同的，比如DFS在完全无障碍物的地图上寻路也是非常低效的，而在迷宫中它又能快速找到出口。所以系统必须能够对算法测试场景进行配置以进行不同的场景测试。

整个地图被限定为矩形，有两个参数width和length控制其大小，这其实不影响场景的设计，因为可以在矩形内部对小方块进行设置以展现不同形状的地图。系统默认带有两种模式：Chaos和Ideal，Chaos的地图完全是随机的，没有任何规律，这种地图的路径消耗一般很大，适用于算法求次佳解的范围；Ideal则是由大部分的高度为0的白色小方块以及部分无法通过的黑色小方块组成，这种地图适用于一般寻路算法的效率对比。在算法配置模块进行场景模式配置后，Application类根据不同的模式，在生成地图高度图时进行不同的操作，Chaos模式下，每个小方块的高度仅仅是随机产生的，Ideal模式下则可以通过手工设置或是通过二维数组赋予，由于系统没有办法知道将要测试的场景，所以手工录入场景的工作还是必须的。寻路的起点和终点同样是通过算法配置模块进行配置的。

* + 1. 测试过程与结果的展示

算法在运行过程中是如何扩展节点的，这方面信息对于研究算法也是非常重要的，所以在算法运行过程中，相应的信息将会被输送到结果展示模块以显示出节点的扩展过程。为了计算算法消耗的时间，在算法run()运行前使用系统的clock()函数计时，并且在一次算法运行结束后立即使用clock()，计算两者差值得到运行时间，这种计时方法也限定了每次只能有一个算法在运行，不能使用类似于多线程那样的多算法同时运行。

算法运行后，简单地通过终点来回溯得到路径，并且根据路径计算消耗，把这些信息连同节点扩展历史信息传送到展示模块显示出来，扩展的节点使用绿色显示，而路径则使用红色表示。显示路径代码如下：

void Algorithm::renderHisAndPath()

{

if(!history.empty())

{

list<int>::iterator it = history.begin();

block[(\*it)/length][(\*it)%length]->SetColor(0xFF00FF00);

history.pop\_front();

}

if(history.empty() && !path.empty())

{

pNode tempNode = path.front();

block[tempNode->postionX][tempNode->postionY]->SetColor(0xFFFF0000);

path.pop\_front();

}

}

其中history中存储的是被扩展的节点的序号，而path则是回溯产生的路径。

1. 寻路算法测试系统测试

根据上一章设计实现的测试系统，本章将要进行以下几种测试：

(1)无障碍物的地图，对所有算法进行测试；

(2)不同消耗的地图，对所有算法进行测试；

(3)一般障碍物的地图，对所有算法进行测试；

(4)内凹型障碍物的地图，对所有算法进行测试；

(5)迷宫型地图，对所有算法进行测试；

根据以上测试，对算法的适用性、效率进行分析。测试标准是对于每种测试地图大小可能会改变但是同一种测试对所有算法来说地图是一样的，每种测试进行三次，观察消耗的时间以及总消耗，取平均值。

* 1. 无障碍物地图测试

地图大小为50\*50，起点(2,2)，终点(46,30)，测试目的是检查各算法是否能找到最短路径以及算法的效率。部分算法的结果图如下：

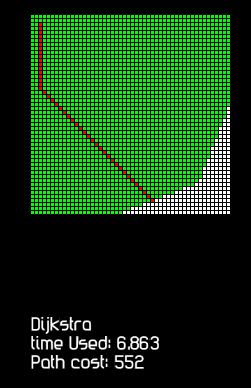
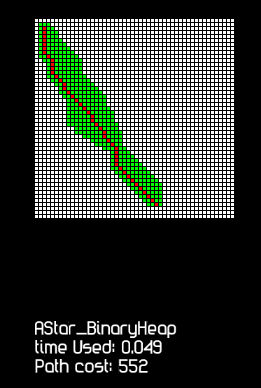


图5.1 二叉堆寻路结果 图5.2 Dijkstra寻路结果

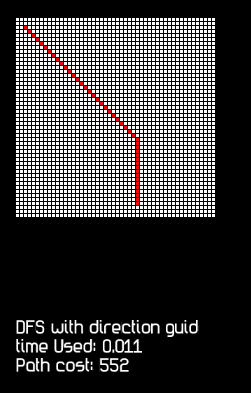


图5.3 带方向指导的DFS寻路结果

所有算法的运行结果表格如下：

表5.1 无障碍地图测试结果

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 算法名称 | 第一次时间(秒) | 第二次 | 第三次 | 平均 | 路径消耗 | 是否最佳 |  |
| Dijkstra | 6.927 | 6.884 | 6.906 | 6.906 | 552 | 是 |  |
| A\* | 10.196 | 10.205 | 10.178 | 10.193 | 552 | 是 |  |
| 插入排序A\* | 10.365 | 10.324 | 10.317 | 10.335 | 552 | 是 |  |
| 二分插入A\* | 0.383 | 0.381 | 0.381 | 0.382 | 552 | 是 |  |
| 二叉堆A\* | 0.057 | 0.045 | 0.051 | 0.051 | 552 | 是 |  |
| DFS | 0.309 | 0.307 | 0.315 | 0.310 | 28032 | 否 |  |
| 方向指导DFS | 0.011 | 0.010 | 0.011 | 0.011 | 552 | 是 |  |
| BFS | 0.426 | 0.465 | 0.400 | 0.430 | 552 | 是 |  |
| 优先队列BFS | 0.399 | 0.400 | 0.384 | 0.394 | 552 | 是 |  |

由表可知，除了DFS算法外，其他算法都可以计算出最短路径，但是Dijkstra、A\*和插入排序的A\*算法由于耗时过长而不适合该类型的寻路，最佳选择是方向指导的DFS，耗时仅0.011秒。

* 1. 不同消耗的地图测试

地图为50\*50，起点(2,2)，终点(46,30)，地图上部区域高度为0，中间高度为100，下部高度为50，如下图所示：

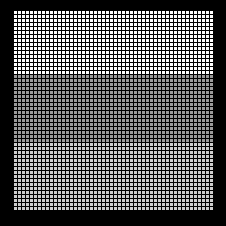


图5.4 不同消耗的地图

本次测试目的是检查在具有不同消耗的地图上各种算法是否能找到最短路径以及效率如何，A\*算法使用的为八方向曼哈顿距离。测试结果如下表：

表5.2 不同消耗地图测试结果

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 算法名称 | 第一次时间(秒) | 第二次 | 第三次 | 平均 | 路径消耗 | 是否最佳 |  |
| Dijkstra | 6.898 | 6.915 | 6.861 | 6.891 | 702 | 是 |  |
| A\* | 58.876 | 69.037 | 66.316 | 64.743 | 702 | 是 |  |
| 插入排序A\* | 20.102 | 20.079 | 20.149 | 20.11 | 702 | 是 |  |
| 二分插入A\* | 2.853 | 2.843 | 2.610 | 2.769 | 702 | 是 |  |
| 二叉堆A\* | 0.743 | 0.851 | 0.863 | 0.819 | 702 | 是 |  |
| DFS | 0.312 | 0.318 | 0.325 | 0.318 | 34302 | 否 |  |
| 方向指导DFS | 0.010 | 0.011 | 0.011 | 0.011 | 742 | 否 |  |
| BFS | 0.304 | 0.303 | 0.594 | 0.400 | 722 | 否 |  |
| 优先队列BFS | 0.535 | 0.571 | 0.554 | 0.553 | 702 | 是 |  |

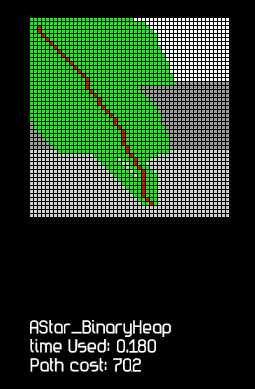


图5.5 二叉堆A\*算法寻路结果

由表可知，DFS、方向指导的DFS以及BFS都没有找到最短路径，而Dijkstra、A\*以及插入排序的A\*虽然找到的最短路径，但是由于寻路时间过长同样不适合该种类型的地图，最佳寻路算法为优先队列的BFS。

* 1. 一般障碍物的地图测试

地图为50\*50，起点(2,2)，终点(46,30)，地图中间有一块无法通过的矩形，如下图所示：

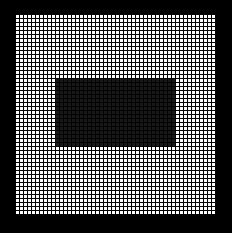


图5.6 一般障碍物地图

本次测试目的是检查在具有障碍物的地图上各种算法是否能找到最短路径以及效率如何，A\*算法使用的为八方向曼哈顿距离。测试结果如下表：

表5.3 一般障碍物地图测试结果

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 算法名称 | 第一次时间(秒) | 第二次 | 第三次 | 平均 | 路径消耗 | 是否最佳 |  |
| Dijkstra | 6.117 | 6.126 | 6.120 | 6.121 | 594 | 是 |  |
| A\* | 4.099 | 4.104 | 4.109 | 4.104 | 594 | 是 |  |
| 插入排序A\* | 4.064 | 4.083 | 4.068 | 4.072 | 594 | 是 |  |
| 二分插入A\* | 0.487 | 0.486 | 0.485 | 0.486 | 594 | 是 |  |
| 二叉堆A\* | 0.083 | 0.081 | 0.079 | 0.081 | 594 | 是 |  |
| DFS | 0.245 | 0.253 | 0.240 | 0.246 | 21276 | 否 |  |
| 方向指导DFS | 0.028 | 0.025 | 0.027 | 0.027 | 2106 | 否 |  |
| BFS | 0.233 | 0.244 | 0.231 | 0.236 | 618 | 否 |  |
| 优先队列BFS | 0.423 | 0.389 | 0.400 | 0.404 | 594 | 是 |  |

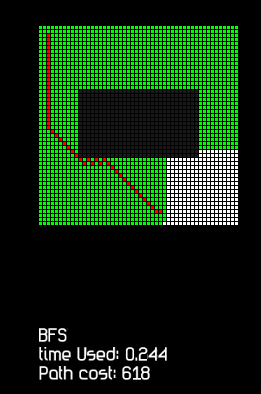


图5.7 BFS寻路结果

由表可以看出，A\*、插入排序A\*以及Dijkstra算法的运行时间比无障碍物要少，这是由于存在障碍物后，可扩展的点数变少所致，并不意味着这三种算法适用于一般障碍物的地图寻路。DFS、方向指导的DFS、BFS依然没有找到最短路径，而且方向指导的DFS路径相比上次消耗变得很大，可见这种算法波动性较大。二叉堆A\*是该类型地图的最佳算法。

* 1. 内凹型障碍物地图测试

地图为50\*50，起点(2,2)，终点(26,15)，终点处于一个凹进去的障碍物内，如下图所示：

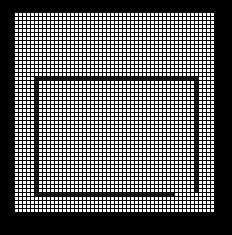


图5.8 内凹型障碍物地图

本次测试目的是检查在内凹型障碍物的地图上各种算法是否能找到最短路径以及效率如何，A\*算法使用的为八方向曼哈顿距离。测试结果如下表：

表5.4 内凹型障碍物的地图测试结果

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 算法名称 | 第一次时间(秒) | 第二次 | 第三次 | 平均 | 路径消耗 | 是否最佳 |  |
| Dijkstra | 6.855 | 6.795 | 6.768 | 6.806 | 1132 | 是 |  |
| A\* | 107.000 | 109.234 | 105.764 | 107.332 | 1132 | 是 |  |
| 插入排序A\* | 123.667 | 122.654 | 123.812 | 123.378 | 1132 | 是 |  |
| 二分插入A\* | 3.582 | 3.521 | 3.525 | 3.543 | 1132 | 是 |  |
| 二叉堆A\* | 0.211 | 0.209 | 0.209 | 0.210 | 1132 | 是 |  |
| DFS | 0.188 | 0.185 | 0.193 | 0.189 | 16348 | 否 |  |
| 方向指导DFS | 0.057 | 0.060 | 0.059 | 0.059 | 5166 | 否 |  |
| BFS | 0.381 | 0.376 | 0.382 | 0.380 | 1326 | 否 |  |
| 优先队列BFS | 0.352 | 0.343 | 0.360 | 0.352 | 1132 | 是 |  |

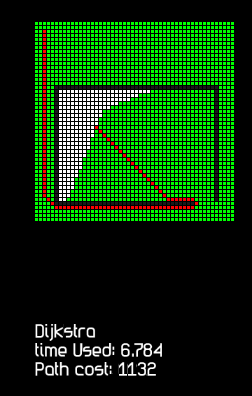


图5.9 Dijkstra寻路结果

由表可知，A\*和插入排序的A\*消耗时间非常长，可见它们完全不适合扩展节点多的寻路。DFS、BFS以及方向指导的DFS依然不能找到最短路径。二分插入的A\*表现一般，优先队列的BFS和二叉堆的A\*表现良好，其中二叉堆A\*为最佳算法。

* 1. 迷宫型地图的测试

地图为10\*10，起点(0,0)，终点(9,9)，由于是迷宫型地图，为了限定路线，这次采用四方向寻路，如下图：

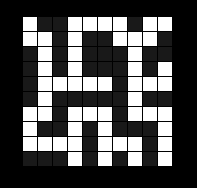


图5.10 迷宫型地图

本次测试目的是检查在迷宫型地图上各种算法找到路径的效率如何，A\*算法使用的为传统曼哈顿距离。测试结果如下表：

表5.5 迷宫型地图测试结果

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 算法名称 | 第一次时间(秒) | 第二次 | 第三次 | 平均 | 路径消耗 | 是否最佳 |  |
| Dijkstra | 0.016 | 0.016 | 0.015 | 0.016 | 260 | 是 |  |
| A\* | 0.032 | 0.032 | 0.033 | 0.032 | 260 | 是 |  |
| 插入排序A\* | 0.038 | 0.038 | 0.038 | 0.038 | 260 | 是 |  |
| 二分插入A\* | 0.018 | 0.017 | 0.017 | 0.017 | 260 | 是 |  |
| 二叉堆A\* | 0.011 | 0.011 | 0.011 | 0.011 | 260 | 是 |  |
| DFS | 0.005 | 0.005 | 0.005 | 0.005 | 260 | 是 |  |
| 方向指导DFS | 0.005 | 0.005 | 0.006 | 0.005 | 260 | 是 |  |
| BFS | 0.006 | 0.007 | 0.007 | 0.007 | 260 | 是 |  |
| 优先队列BFS | 0.006 | 0.006 | 0.006 | 0.006 | 260 | 是 |  |

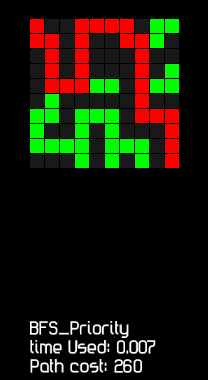


图5.10 优先队列的BFS寻路结果

由于本次试验的迷宫地图只有一条唯一出路，所以所有能找到出路的算法的消耗值都是一样的，故只要比较消耗的时间即可。由表可知，DFS、方向指导的DFS、BFS以及优先队列的BFS在迷宫方面的效率很高，属于迷宫型地图的最佳算法。二叉堆A\*算法只有当迷宫的范围增大后才能发挥优势，其他算法表现一般。

1. 总结与展望
   1. 总结

论文先介绍了寻路问题的相关内容，包括有权无权图以及有向无向图，也较为详细地Dijkstra算法、传统A\*算法和BFS算法。

其次论文对A\*算法进行了改进研究，从将开启列表中元素用插入排序进行排列到使用二分插入来排序，又研究了什么是二叉堆，使用二叉堆来实现A\*算法。同样地研究了使用有限队列的BFS，解决BFS无法找到最短路径问题。对DFS在无障碍物的地图上寻路低效问题给出了初步的研究，提出了方向指导的DFS。接着，论文就游戏中实际问题，比如八方向寻路的启发式函数设计、不同消耗地图的启发式函数设计进行了研究讨论。

为了验证算法改进的成果，论文设计了一个算法测试系统，该系统可以自由地配置寻路算法对不同的场景进行寻路测试，并且呈现寻路的过程和结果，给出寻路的时间和消耗。并且在最后论文对于五种不同的地形进行了系统的测试，对比了9种寻路算法，给出了测试结论。

* 1. 展望

论文对于复杂地图中A\*算法的启发式函数设计没有做出深入的研究工作，还不能提出解决方案，这是今后要做的工作。对于游戏寻路中的一些其他常见问题如平滑路径问题、多单位调度问题、动态地图寻路问题、多点寻路问题等没有做出研究工作，还需要学习研究。

实验系统还有一定的缺陷，不能通过程序外部来对寻路场景进行配置，不能在程序运行中切换寻路算法，不能在程序运行中切换场景，使得该系统还是具有一定的局限性，接下来将要完善它。

关于寻路算法这一领域，还有很多内容值得去学习研究，这对于今后的学习工作将会有很大的帮助，特别是如果今天参与游戏开发工作。更大的一方面，人工智能的研究现在越来越流行，游戏中的AI设计占游戏可玩性比例越来越高，学好人工智能，对游戏开发是非常重要的，也是做好游戏开发所必备的条件，加深这方面的研究将势在必行。

主要参考文献

1. 何国辉,陈家琪.游戏开发中智能路径搜索算法的研究[J].计算机工程与设计,2006,27(13):2334-2337页.
2. Hart P.E.,Nilsson,N.J. and Raphael B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths"[J]. Systems Science and Cybernetics,1968,4(2):100-107页.
3. Pearl,Judea.Heuristics:Intelligent Search Strategies for Computer Problem Solving.Addison-Wesley[M].Reading,Massachuset:Addison-Wesley.1984:1-103页.
4. Nilsson,N.J.Principles of Artificial Intelligence[M].Palo Alto,California: Tioga Publishing Company,1980.53-81页.
5. 李庆华,戴光明.基于演化计算的最短路避障径算法设计[J].小型微型计算机系统,2005,26(3):108-112页.
6. 王士同.启发式图搜索算法A\*的改进[J].计算机学报,1991,(3):192-194页.
7. 邹亮,徐建闽.A\*算法改进及其在动态最短路径问题中的应用[J].深圳大学学报理工版,2007,24(1):32-35页.
8. 王志和,凌云.Dijkstra 最短路径算法的优化及其实现[J].微计算机信息,2007,23(11):275-277页.
9. Steve Rabin.AI game programming wisdom[M].北京:清华大学出版社,2005:143-156页.
10. 王一剑.人工智能在游戏开发中的应用[D].上海:同济大学,2009.
11. Alex J.Champandard.AI Game Development[M].北京:中国环境科学出版社,2004:202-232页.
12. 樊莉.人工智能中的A\*算法应用及编程[J].微机发展,2003,13(5):33-35页.

谢辞

在论文即将完成之际，我衷心地感谢指导教授张惠娟教授。她在我撰写论文期间给予了我莫大的帮助，不辞辛苦为我解答在论文方面的疑惑，多次对我的论文提出了修改建议，对我的论文完成功不可没。张老师的严谨的治学之风深深的影响到了我，是我受益匪浅。

其次要感谢软院的同学们，是他们的好学好问给了我动力，使我可以持之以恒地进行论文研究撰写工作。

最后要感谢我的家人，感谢他们对我无私的关爱。