

**算法设计与分析课程报告**

姓 名： 郑召作 学 号： 20161001451

学 院： 计算机学院 专 业： 信息安全

小组成员： 诸葛子音 、潘冠男 、王伟

选题名称： 静态路网寻路问题——A\*算法

指导教师： 王茂才

2018 年 7 月

目录

[一、课堂研讨 2](#_Toc519440825)

[1. 选题介绍 2](#_Toc519440826)

[2．选题分析 2](#_Toc519440827)

[2.1问题分析 2](#_Toc519440828)

[2.2输入与输出分析 2](#_Toc519440829)

[2.3问题难点分析 3](#_Toc519440830)

[3．算法选择 4](#_Toc519440831)

[3.1 求解本问题常用的算法介绍与分析 4](#_Toc519440832)

[3.2 本报告采用的算法思想 4](#_Toc519440833)

[3.3 本算法与其它常用算法的分析比较 5](#_Toc519440834)

[4、代码实现 6](#_Toc519440835)

[5、实现结果 7](#_Toc519440836)

[6、性能分析 8](#_Toc519440837)

[7、总结与展望 9](#_Toc519440838)

[8、本人参与的工作 9](#_Toc519440839)

[9、源代码 12](#_Toc519440840)

[二、上机实习 23](#_Toc519440841)

[第一题 23](#_Toc519440842)

[1. 实现结果及代码测试 23](#_Toc519440843)

[2. 实现中遇到的主要困难及解决方法 23](#_Toc519440844)

[3. 本次实习的心得与体会 23](#_Toc519440845)

[4. 源代码 24](#_Toc519440846)

[第二题 27](#_Toc519440847)

[1. 实现结果及代码测试 27](#_Toc519440848)

[2. 实现中遇到的主要困难及解决方法 27](#_Toc519440849)

[3. 本次实习的心得与体会 27](#_Toc519440850)

[4. 源代码 28](#_Toc519440851)

[第三题 30](#_Toc519440852)

[1. 实现结果及代码测试 30](#_Toc519440853)

[2. 实现中遇到的主要困难及解决方法 31](#_Toc519440854)

[3. 本次实习的心得与体会 31](#_Toc519440855)

[4. 源代码 31](#_Toc519440856)

# **一、课堂研讨**

# 1. 选题介绍

“如何从一个地点到达另外一个地点，并且在这个过程中所需要的花费最少”这是我们在日常生活中经常会遇到的路径规划问题。

关于路径搜索及其规划问题，我们在之前的学习中曾经接触过。其中，我们接触最多的是有名的最短路径算法——Dijkstra算法（迪杰斯特拉算法）和弗洛伊德算法。当然，BFS（广度优先搜索算法）也能够解决最优路径的选取问题。

因为这些算法在之前的有些课程中都有过详细的介绍以及系统的学习，不能够体现本次研究的创新性。所以，在本次研讨中我们决定采用另外一种之前从未接触的路径搜索算法作为本次算法的研究对象，最终我们选取的算法就是A\*（A-Star)算法。

A\*算法是一种静态路网中求解最短路径最有效的直接搜索方法，也是解决许多搜索问题的有效算法。所以在本次课堂研讨中我们决定采用这个算法，希望能够通过这次学习加深我们对寻路算法的理解。

# 2．选题分析

## 2.1问题分析

在处理路径规划问题时，我们往往要寻找最短且最优的路径。这类问题在处理时大致会有两种不同的情况：一是地图中没有明显的障碍物，可以实现直线路径的规划的情况；二是地图中有或凹或凸的障碍物，规划的路径可能无法径直到达的情况。

## 2.2输入与输出分析

在处理第一种情况的问题时，我们首先选择的算法大多是Dijkstra算法或者BFS算法。在问题处理过程中，Dijkstra算法通过图结构进行路径搜索和规划，首先从物体所在的起始位置开始，将图中所有的其余结点的权重值赋值为无限大，意为不可以直接到达。算法开始后，依次遍历搜索起点的每个相临结点，并根据不同路径所拥有的权重值更新路径权重，最后将权重值最小的相邻结点放入路径集合中。用这种方法不断的迭代扩展，直到达到目的结点停止搜索。可以将算法描述为下面几步：

1. 从物体所在的初始点开始，访问图中的结点。
2. 迭代检查待检查结点集中的结点。
3. 把和该结点最靠近的尚未检查的结点加入待检查结点集。
4. 结点集从初始结点向外扩展，直到到达目标结点。
5. 保证所有边都有一个非负的权重值，得到一条可找到的最短路径。

通过这几步的不断迭代调用，最终总会在该图中找到一个已存在的权重值和最小的路径，这个算法在不考虑障碍物影响的情况下，往往具有不错的效果，能够得到一个考虑实际权重的较为高效的最短路径。

BFS处理这类问题时，我们需要引入一个启发式节点函数，这可以帮助算法更加快速的找到所需要的路径。在处理过程上，BFS算法在运行流程上与Dijkstra算法类似，它的特殊之处在于BFS算法能够评估任意节点到目标点的代价，这被称之为启发式的代价评估。也正是因为这个特点，使得BFS算法在处理无障碍路径搜索规划问题时能够有更加快速的运行效果。

BFS算法的运行特点如下：

1. 不选择离初始结点最近的结点，而选择离目标最近的结点
2. 不能保证找到一条最短路径
3. 采用启发式搜索函数，使得运行效率要高于Dijkstra算法。

对于BFS算法的启发式搜索可以进行形象的进行理解，比如说，如果目标位于出发点的南方，BFS将趋向于导向南方的路径。这就使得BFS算法够更加快速的得到目的地位于南方路径。

这便是我们在处理无凹凸障碍路径搜索问题时最常用的两种方法，这种地图中的路径可以根据起始点和终点直接规划出一个当前的最优路径，不用考虑障碍的影响，所以最终的路径甚至可能是直线。不过在实际应用中，大多数的情况经常是在地图中存在或凹或凸的障碍，使得路径的搜索要考虑如何避开障碍，并使得路径之和最短。

## 2.3问题难点分析

当这两种算法用于这种情况时，对于Dijkstra算法，由于存在障碍物影响，导致每次迭代搜索最优路径时，计算次数大大增加，从而增加了运行时间。但是即使需要的时间增加，但却是能够找到一个较为合理的最短路径。而对于BFS算法，在这种考虑地图障碍物的情况下，BFS算法的运行时间仍然是要快于Dijkstra算法的，但是在路径的选择上，BFS算法所选择的路径明显不是最优的（此处的不是最优是指在与Dijkstra算法寻找的路径相比时）。

为了获得兼具速率快与路径优两个特点的路径搜索算法，我们决定尝试A\*算法。A\*算法在算法设计上就对时间与路径进行了双重考虑，能够在保证最优搜索时间的前提下，得到一个最小权重的路径。它具备了Dijkstra算法与BFS算法二者的优点，使得它在处理无障碍地图最短路径搜索时能够A\*能用启发式函数引导它自己，所以能够有像BFS路径算法一样的速度，也能够获得类似Dijkstra算法结果一样的最短路径。在处理有障碍地图的路径搜索问题时，要确保选择的算法能够有较强的灵活性，能够根据地图环境随时调整自己的搜索方式。

基于上述的这些考虑以及需求，我们得出了本次研究的研究重点。于是，本次研究的重点便是在考虑障碍分布的情况下，对我们所选择的A\*算法与其余算法的进行效率分析。在此过程中我们需要对多个算法在相同的地图上进行路径搜索，并对运行时间和最后得出的路径进行分析。

# 3．算法选择

## 3.1 求解本问题常用的算法介绍与分析

通过算法分析，我们已经得知在处理有障碍路径搜索问题时可以采用的方法有Dijkstra算法、BFS算法以及A\*算法。不过，Dijkstra算法和BFS算法在处理有障碍路径搜索问题时都有各自的缺陷。正是因为这些算法存在这些较为明显的不完美的问题，使我们决定去尝试新的较为优秀的路径搜索算法——A\*算法。首先对于Dijkstra算法，由于其算法的设计思想，导致其在寻找最优路径时需要遍历图中的每个结点并进行大量复杂的运行分析，这就会导致程序的运行时间大大增加，降低了寻路算法的效率与性能。而对于BFS算法，该算法在设计中加入了一个启发式的节点函数，这个函数可以在程序寻找路径时提供一个简单而准确的方向，大大增加程序寻找最优路径的速度。BFS算法虽然在一定程度上解决的Dijkstra算法的运行速度问题，但它又产生了新的问题。BFS算法虽然在速度方面比Dijkstra算法更具优势，但是其最终寻找到的路径往往不是唯一的，也就是说通过BFS算法搜索而得到的路径往往不是最优的，这就为BFS算法带来了一个巨大的缺陷。

## 3.2 本报告采用的算法思想

我们在本次研究中采用的算法主要为 A\*算法。经过我们查阅资料得知，A\*算法在路径搜索算法中是非常受欢迎的选择，它的使用相当灵活，这个特点使得它能够在各种各样的情形之中如鱼得水。而且，除此之外，A\*算法还能够潜在地搜索地图中的一个很大的区域，这一点和其他众多的图搜索算法是一样的。在解决具体问题时，A\*算法和Dijkstra算法一样，最终得到的地图路径都是一个较为优秀的最短路径，而且由于A\*算法也使用了启发式函数引导自身搜索过程，使得A\*算法的处理时间也像BFS算法一样，只需要花费较少的处理时间。特别是在处理简单的地图路径搜索时，A\*算法和BFS算法一样快。

A\*算法是一种在静态路网中求解最短路径最有效的直接搜索方法，也是在解决许多其他问题时常用的启发式搜索算法。我们称A\*算法是一个最有效的直接搜索算法是因为在A\*算法之后还出现了其他的一些搜索算法，但他们是预搜索算法，比如说，ALT算法、CH算法以及HL算法等等，这些算法在搜索和查询效率上是A\*算法的数千倍甚至是上万倍，所以说A\*算法只是目前最有效的直接搜索方法。在其之外还存在的其他许许多多更加高效的路径搜索算法。

A\*算法在算法设计上，将Dijkstra算法与BFS算法的信息模块结合起来，有效地利用Dijkstra算法和BFS算法的优点，从而能够获得一个兼顾速度与最优路径的较为优秀的算法。

A\*算法的算法计算公式如下:

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

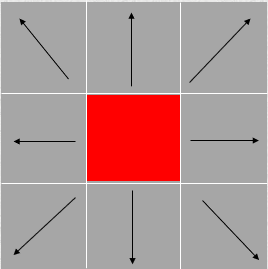
在这个公式中，f(n) 是从初始状态经由状态n到目标状态的代价估计，g(n) 是在状态空间中从初始状态到状态n的实际代价，h(n) 是从状态n到目标状态的最佳路径的估计代价。

上述的这些变量的意义在实际使用中还会发生些许的变化，在实际问题中，g(n)表示从初始结点到任意结点n的代价，而h(n)表示从结点n到目标点的启发式评估代价。当从初始点向目标点移动时，A\*算法根据内在机制将g(n)和h(n)进行权衡。当每次进行主循环时，该算法会对f(n)的最小的结点进行检查，此时的f(n)就是根据上面的计算公式而得出。

A\*算法的执行流程主要是通过起始点开始，不断地检查相邻方格的位置，以此向外扩展直到找到路径最终的目标位置。而在这个过程中，又可以这一个大的流程简单大致地分为四小步：

1. 确定起点位置，并将它作为待处理点存入一个“开启列表”，这是一个待检查方格的列表。
2. 从该起始点开始，寻找周围所有可到达的或者可通过的方格，并把他们加入“开启列表”，加入“开启列表”后，将起始点的方格作为这些方格的父方格。
3. 从“开启列表”中删除起始点，并把它加入到一个“关闭列表”，该列表中保存所有不需要再次进行检查的方格。
4. 之后不断地重复二、三过程，直到最终到达目的地为止。

在搜索路径时，A\*算法可以根据八个方向来规划最短路径，如图一所示。



图一

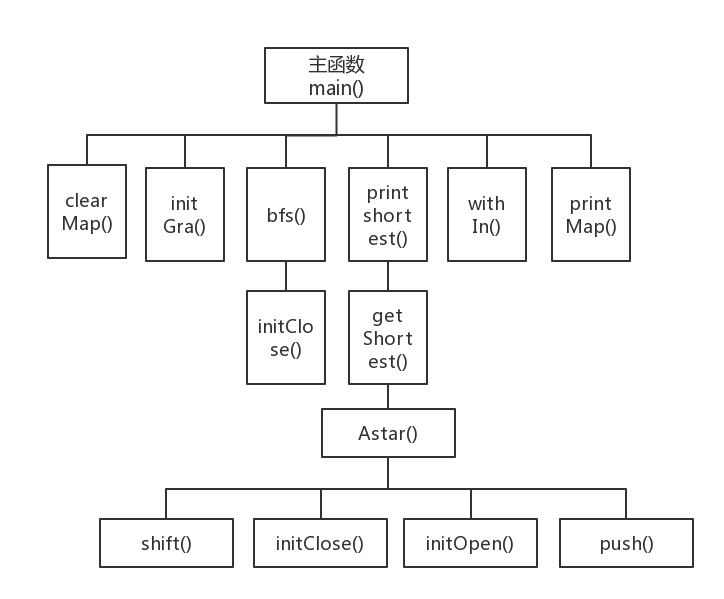
在确定起点后，我们通过F值评估来决定之后选择的方格位置。F评估的关键等式为：F=G+H。其中，G表示从起点A，沿着产生的路径，移动到网格上指定方格的移动耗费；H表示从网格的某个方格到终点B的预估移动耗费。这个过程之所以称为启发式搜索，是因为在有障碍的地图上，障碍的不确定性使得该算法的路径搜索需要不断地试探和猜测，最终才可以确定一条较为合理的路径。

## 3.3 本算法与其它常用算法的分析比较

在处理有障碍路径搜索问题时可以采用的方法有Dijkstra 算法、BFS 算法以及A\*算法。首先对于 Dijkstra 算法，由于其算法的设计思想，导致其在寻找最优路径时需要遍历图中的每个结点并进行大量复杂的运行分析，这就会导致程序的运行时间大大增加，降低了寻路算法的效率与性能。而对于 BFS 算法，该算法在设计中加入了一个启发式的节点函数，这个函数可以在程序寻找路径时提供一个简单而准确的方向，大大增加程序寻找最优路径的速度。BFS 算法虽然在一定程度上解决的 Dijkstra 算法的运行速度问题，但它又产生了新的问题。BFS 算法虽然在速度方面比 Dijkstra 算法更具优势，但是其最终寻找到的路径往往不是唯一的，也就是说通过 BFS 算法搜索而得到的路径往往不是最优的，这就为 BFS 算法带来了一个巨大的缺陷。

# 4、代码实现

我们在代码实现时采用了C语言，代码的函数调用关系如图二所示，从上到下表示的是函数的层次调用深度，在下面的函数表示层次更深一些。



图二

A\*算法的算法文字描述如下：

1. 把起点加入 open list
2. 遍历 open list ，查找 F 值最小的节点，把它作为当前要处理的节点
3. 把这个节点移到 close list
4. 对当前方格的 8个相邻方格的每一个方格进行如下判断。
5. 对当前方格的 8个相邻方格的每一个方格进行如下判断。
6. 如果它不在 open list 中，把它加入 open list，并且把当前方格设置为它的父亲，记录该方格的 F，G和H值。
7. 如果它已经在 open list 中，用 G 值判断当前路径是否更佳，若更佳，将其父亲格设为当前格。
8. 当open list为空时或终点已加入open list时，跳出循环。
9. 保存路径。从终点开始，每个方格沿着父节点移动直至起点，就是最佳路径。

我们对算法的文字描述转换为了代码语言：

while(OPEN!=NULL) {

从OPEN表中取f(n)最小的节点n;

if(n节点==目标节点)

break;

for(当前节点n的每个子节点X) {

计算f(X);

if(XinOPEN)

if(新的f(X)<OPEN中的f(X)) {

把n设置为X的父亲;

更新OPEN表中的f(n);

}

if(XinCLOSE)

continue;

if(Xnotinboth) {

把n设置为X的父亲;

求f(X);

并将X插入OPEN表中;//还没有排序

}

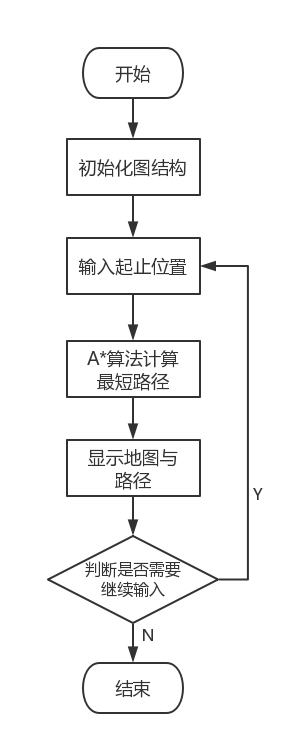
}//endfor

将n节点插入CLOSE表中;

按照f(n)将OPEN表中的节点排序;//实际上是比较OPEN表内节点f的大小，从最小路径的节点向下进行。

}//endwhile(OPEN!=NULL)

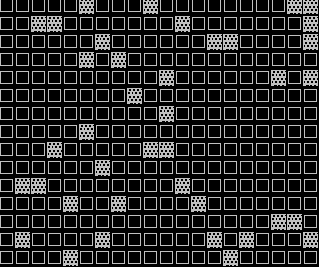
在代码的实际实现过程中我们将整个过程分为了五大步骤，这五大步骤的先后执行关系我们通过下面的流程图详细的展现出来。首先初始化图结构，之后进行起止位置的输入，输入位置后就可以通过A\*算法来进行计算该地图中的最短路径。

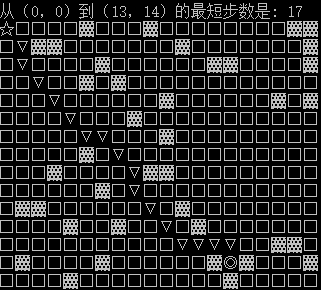


图三

# 5、实现结果

我们对我们的代码以及编写的算法进行了多次多方面的测试，每次测试都能够得到一个较为优秀的最短路径。同时，我们还将A\*算法与其他算法进行了比较，最后得出A\*比起其他算法较为优秀的结论。

A\*算法执行后的界面如下图所示。

  
 图四 图五

其中图四是程序初始地图，图五表示的是程序执行结束后所找到的最优路径，图中的△表示的是路径，空心方格表示可通过的路径方格，实心方格表示不可通过。

# 6、性能分析

在完成算法后，我们对A\*算法、Dijkstra算法以及BFS算法在相同地图上的不同性能进行了分析，最终得到以下结果。

A\*算法更关注于点到点的最短路径(包括具体路径)。当距离估计与实际值越接近，估价函数取得就越好，可以取两节点间曼哈顿距离作为距离估计，能保证最短路的搜索向终点的方向进行。在这一点上，A\*算法明显优于迪杰斯特拉算法的毫无方向的四周搜索。所以，A\*算法其实是一种最好优先的算法。由于在一些问题求解时，我们希望能够求解出状态空间搜索的最短路径，也就是用最快的方法求解问题，而A\*算法就是干这种事情的,它的时间复杂度趋近于O(n)。

不过，A\*算法的最终搜索结果还是受到h(n)的影响。如果我们以d(n)表达状态n到目标状态的距离，那么h(n)的选取大致有如下三种情况。h(n)< d(n)时，在这种情况下，搜索的点数多，搜索范围大，效率低，但能得到最优解。而当h(n)=d(n)，即距离估计h(n)等于最短距离时，那么搜索将严格沿着最短路径进行，此时的搜索效率是最高的。当h(n)>d(n)时，搜索的点数少，搜索范围小，效率高，但此时不能保证能够得到最优解。

Dijkstra算法是计算源点到其他所有点的最短路径长度，Dijkstra算法的实质是广度优先搜索，是一种发散式的搜索，所以空间复杂度和时间复杂度都比较高。时间复杂度为T(n)=O(n2)，而且有时候不能求得最短路径。

广度优先算法虽然效率较高但不是最优解。即宽度优先虽然是完备的（如果分支因子有限的话），在任何情况下广度优先算法都能找到一个解。但是，它找到的第一个解并非最优的。此外，它还存在一个最坏的情况，当目标结点是第d层的最后一个被扩展的结点时，它将耗费大量的时间。最终我们求得BFS算法的宽度优先时间复杂度为：O(b\*(d+1))（b为分支因子，d为 深度）；空间复杂度为：0(n)，n为所存储的节点的个数。

# 7、总结与展望

在本次算法的研究与实现的过程中，我们实现了多种不同的最短路径算法，其中我们重点研究了A\*路径算法。我们将其他多种算法与A\*算法进行比较，我们比较了Dijkstra与A\*算法，BFS算法与A\*算法以及贪心算法与A\*算法，经过比较我们发现，在相同的障碍物环境下，A\*算法比起其他的有更高的效率以及更快的速度。

在这四种比较中，我们重点分析了Dijkstra算法与A\*算法之间的不同点，我们将其总结为以下四点：

1.Dijkstra算法计算源点到其他所有点的最短路径长度，A\*关注点到点的最短路径(包括具体路径)。

2.Dijkstra算法建立在较为抽象的图论层面，A\*算法可以更轻松地用在诸如游戏地图寻路中。

3.Dijkstra算法的实质是广度优先搜索，是一种发散式的搜索，所以空间复杂度和时间复杂度都比较高。对路径上的当前点，A\*算法不但记录其到源点的代价，还计算当前点到目标点的期望代价，是一种启发式算法，也可以认为是一种深度优先的算法。

4.由第一点，当目标点很多时，A\*算法会带入大量重复数据和复杂的估价函数，所以如果不要求获得具体路径而只比较路径长度时，Dijkstra算法会成为更好的选择。

在本次研究路径算法的过程中，我们认识到路径算法是一种非常重要的算法，我们生活中的许多方面都会用到这个算法，比如说导航系统，游戏路径识别以及工程路径规划，在这些方面路径算法都起到了举足轻重的作用。我们本次研究的A\*算法只是一种比较基础的路径算法，不过这次研究为我们奠定了一个不错的基础，使我们未来能够根据自己的想法设计出更加优秀的路径搜索算法。

在未来我们期望可以可以通过改变h函数来改变路径搜索过程,或者我们可以将机器学习技术引入到该算法中，借助机器学习来寻找最合适的G值，提高路径搜索算法的速度以及效率。

# 8、本人参与的工作

本人负责代码的所有编写以及作为组长负责任务的分配。

在研究这个问题的时候，我将这个问题抽象化处理，着重深入探讨这个问题需要解决的是什么？首先从起点到终点有很多条路径，本文需要的是找到一条最好的路径，在这里，我们评判路径好坏的标准是路径的长短。在寻找路径的过程中，难免有障碍物，这就极大地增加了路径搜索的难度，经过翻阅资料，我发现了很多算法，比如Dijkstra算法、最佳优先搜索算法、A\*算法。

最后经过小组的讨论之后，我们提出使用启发式算法A\*来解决这个问题：

在查阅相关文献资料的时候，得知A\*算法作为一种有序搜索算法，其特点在与对评估函数的定义上，因此我觉得对于我们对于一般的有序搜索，总是选择f值最小的节点作为扩展节点。因此，f是根据需要找到的一条最小代价路径的观点来估算节点的，所以，可考虑每个节点n的估计函数值为两个分量组成：从起始节点到节点n的代价以及从节点n的代价到达目标节点的代价，即表达为公式之后为F(n)=G(n)+H(n)。

在整个编写代码过程之中，首先，先在纸上构思，使用怎样的存储结构来表示障碍物和通路，怎么编写A\*算法，然后在纸上画了大致的流程图，弄清楚框架之后，我才开始进行代码的编码。首先用代码把基本的框架完成，再把队列的操作、地图的初始化等实现，最后完成了功能的实现比如地图的打印、路径是否最短判断、地图的重新构造、地图的清屏、A\*算法等等，一开始思路不是很明确，在知网上面看了很多文献，然后慢慢出来了。

编码过程中首当其冲的出现的问题是该使用怎样的存储方式来对图进行表示呢？常见的对图的表示方法有数组存储和链表存储，而且这两种表示法也各有自己的优缺点，数组可以方便地实现对其中某个元素的存取，但对图中的插入和删除操作却很困难，而链表则利于插入和删除，但对某个特定元素的定位却需借助于搜索。而A\*算法则需要快速插入和删除所求得的最优值以及可以对当前结点以下结点的操作。当然本题只需对图的表示进行标注，数组是个很不错的选择。

我采用的方法是通过固定地形的地图的路径来建立路径点系统。路径点通常是道路或地牢通道的转折点。可以预先设定路径点。如果两个路径点的连线没有障碍物的话它们被视为相邻的。可以保存这些相邻信息在某种表中，当openlist增加新项时使用。然后记录G值（可能用两个结点间的直线距离）和H值（可能使用从节点到目标的直线距离）。其它的都想往常一样处理。

在编写代码时，为了方便对物体运动的操作，我使用了一个结构体数组来存储每一个操作对X，Y值的影响，比如向东操作的值为{ 0, 1 }，含义是X轴向右平移一个单位，Y轴不操作，因为在代码中，地图为二维平面，对其进行操作的时候，水平方向的操作（X轴）是低维坐标，而垂直方向上是高维坐标，我在编写代码时，未加思考，就轻易将判定，导致X、Y的操作混淆，造成了和预期完全不一样的结果，在单步调试之后，发现了错误并及时修改了八个操作的值。

然后在对Openlist进行存储时，我使用的是队列数组进行存储，使用队列能够最方便地返回到上一步。队列作为路径的Openlist的存储方式，可以有效的利用资源。如果原来的G值比较大，重新赋予G值、父节点、F值，并对Openlist中的元素进行重新选择，同时对Close表中的元素进行插入，但是如果原来的G值比较小，则不用管。即原来的G值比较大的话，证明找到了更优的路径，应该更换。

对整个模型框架进行了大致的编码完成之后，我才对h值的确认进行编码，主要是为了想通过对比比较来寻找最适合的h函数，因为在小组讨论的时候，我们一直对h值的函数保佑争议，产生了很大的分歧，常见的h函数有我们可以发现，A\*算法成功与否的关键在于估价函数的正确选择，从理论上说，一个完全正确的估价函数是可以非常迅速地得到问题的正确解答，但一般完全正确的估价函数是得不到的，因而A\*算法不能保证它每次都得到正确解答。一个不理想的估价函数可能会使它工作得很慢，甚至会给出错误的解答。为了提高解答的正确性，我们可以适当地降低估价函数的值，从而使之进行更多的搜索，但这是以降低它的速度为代价的，因而我们可以根据实际对解答的速度和正确性的要求而设计出不同的方案，使之更具弹性。

在上面中，提到的估计函数，F(n)为节点n的股价函数，G(n)为状态空间中从出事结点到n结点的实际代价,H(n)为从n到目标结点最佳路径的估计代价。在这里主要是H(n)体现了搜索的启发信息，因为G(n)是已知的。详细点说，G(n)代表了搜索的广度的优先趋势。h(n)是从n到目标节点最佳路径的估计代价, 体现了搜索的启发信息。如果当本模型的h(n) 始终为0的话，那么A\*算法就和Dijkstra算法一样。由于g(n)是已知的，说的详细一点，g(n)代表了广度优先搜索的趋势，如果当h(n)>>g(n)时，可以忽略g(n)，提高效率。

h(n)取值常见有曼哈顿距离、对角线距离、欧几里得距离及其变形、Breaking Ties等等，本文所选取的曼哈顿距离，不是因为其简单方便，而是针对本课题而言，曼哈顿距离能找到最优解，在编写代码过程中，我针对了曼哈顿距离、欧几里得距离以及比较少见的余弦距离一一进行了比较，比较结果如下表所示，但是这并不意味着，在其他情况下，曼哈顿距离比其他距离更有效更合理。曼哈顿距离是两点在垂直方向上的距离加上水平方向上的距离，即D（i，j）=|Xi-Xj|+|Yi-Yj|。而欧几里得距离则是对两点的在两个方向差的平方和来开根号，而余弦距离，又叫余弦相似度，用向量空间中两个向量夹角的余弦值作为衡量两个个体间差异的大小，相比距离度量，余弦相似度更加注重两个向量在方向上的差异，而非距离或长度上。

表中数据为从起点出发到终点的路径长度：

|  |  |  |
| --- | --- | --- |
| 距离函数 | 图1结果 | 图2结果 |
| 曼哈顿距离 | 10步 | 13步 |
| 欧几里得距离 | 15步 | 12步 |
| 余弦距离 | 14步 | 17步 |

表1：距离公式比较

上表中的实验结果这和我预期的差距很大，我本以为欧几里得距离能够比曼哈顿距离更切合本模型，可能是由于样本数据太少的原因。在面对这样的问题，我提出使用数据挖掘的算法来计算出正确的h值，通过随机化选择起始点，对每一次寻找路径使用三种距离算法并记录每一种距离算法的路径长度，通过对数据进行统计学习，能够找到最适合的距离，但是由于编写语言是C++，而本人暂时只会使用Python机器学习算法的编写，因此，这只是本人的一个展望，并没有投入到代码的编写。

虽然不是由我来对该算法的时间、空间复杂度来进行测量，但是在进行编码过程中深有体会，对于规模这么小的迷宫算法，A\*算法和其他算法的差距不是很明显，但是当规模突然增大时，A\*算法相比较于其他算法有很明显的优势，比如我们在数据结构中学习的迪杰斯特拉算法，它本身的实质还是广度优先算法，空间复杂度和时间复杂度都很高，而且不一定能求出最短路径来。因为每次访问openlist 时，都需要找到具有最小的F值方格，有几种做法可以做到这个。当你通过遍历整个open list来找到具有最小F值的方格时，可以随意保存路径元素。这个很简单，但对于很长的路径会很慢。比较号的方法可以通过维护一个排好序的表来改进，每次当你需要找到具有最小F值的方格时，仅取出表的第一项即可。这是我写代码的过程中，使用的第一个方法。对于小地图，这可以很好的工作，但这不是最快的方案。追求速度的A\*程序员使用了叫做二叉堆的东西，我的程序里没有使用了这个。但是以我的以往的编程经验，这种方法在多数场合下会快2—3倍，对于更长的路径速度成几何级数增长(10倍甚至更快)。

最后一个问题是平滑处理，这是我在课堂答辩完之后，在听机器学习公开课的过程，所思考到的。平滑处理是数据挖掘和机器学习当中为了帮助解决噪声数据而提出的一种方法，常见解决办法是二则化。在A\*算法中，会给我寻找到一条花费最小，代价最少的路径，但是他不会提供一条最平滑的路径，比如当一个方格既可以先向下再向右，也可以直接走右下方格，由于没有进行平滑处理，因此我认为可以惩罚那些改变方向的方格，把它的G 值增加一个额外的开销，另外一个选择是，可以在遍历路径的过程中，查找那些使用相邻的方格来进行代替是路径更为平滑

另外在查阅资料的过程中，了解到了一种改进算法，A~\*算法，它通过筛选跳点来进行扩展，直到生成最终路径，扩展过程中使用跳点代替A~\*算法中大量可能被添加到OpenList和ClosedList的不必要节点，从而减少计算量。可以在寻路过程中扩展更少的节点，寻路速度更快，且加速效果随环境地图的增大更加明显。

在代码编写完成之后，我对于整个算法有了更深入的理解。作为一种启发式算法的经典算法，和其他启发式算法（如蚁群算法、神经网络等）一样，实质是它对可接受的花费（计算时间与空间）下给出待解决组合优化问题每一个实例的一个可行解，而此可行解与最优解的偏离程度不能被估计。A\*算法不仅在图的研究上有很大作用，在如今发展的机器学习、人工智能等领域也有很多成就，常见的比如有家庭机器人寻路。

# 9、源代码

#include <stdio.h>

#include <math.h>

#define \_CRT\_SECURE\_NO\_WARNINGS

#define MaxLength 100 //用于优先队列（Open表）的数组

#define Height 15 //地图高度

#define Width 20 //地图宽度

#define Reachable 0 //可以到达的结点

#define Bar 1 //障碍物

#define Pass 2 //需要走的步数

#define Source 3 //起点

#define Destination 4 //终点

#define Sequential 0 //顺序遍历

#define NoSolution 2 //无解决方案

#define Infinity 0xfffffff

#define East (1 << 0)

#define South\_East (1 << 1)

#define South (1 << 2)

#define South\_West (1 << 3)

#define West (1 << 4)

#define North\_West (1 << 5)

#define North (1 << 6)

#define North\_East (1 << 7)

typedef struct

{

signed char x, y;

} Point;

const Point dir[8] =

{

{ 0, 1 }, // East

{ 1, 1 }, // South\_East

{ 1, 0 }, // South

{ 1, -1 }, // South\_West

{ 0, -1 }, // West

{ -1, -1 }, // North\_West

{ -1, 0 }, // North

{ -1, 1 } // North\_East

};

unsigned char within(int x, int y)

{

return (x >= 0 && y >= 0

&& x < Height && y < Width);

}

typedef struct

{

int x, y;

unsigned char reachable, sur, value;

} MapNode;

typedef struct Close

{

MapNode \*cur;

char vis;

struct Close \*from;

float F, G;

int H;

} Close;

typedef struct //优先队列（Open表）

{

int length; //当前队列的长度

Close\* Array[MaxLength]; //评价结点的指针

} Open;

static MapNode graph[Height][Width];

static int srcX, srcY, dstX, dstY; //起始点、终点

static Close close[Height][Width];

// 优先队列基本操作

void initOpen(Open \*q) //优先队列初始化

{

q->length = 0; // 队内元素数初始为0

}

void push(Open \*q, Close cls[Height][Width], int x, int y, float g)

{ //向优先队列（Open表）中添加元素

Close \*t;

int i, mintag;

cls[x][y].G = g; //所添加节点的坐标

cls[x][y].F = cls[x][y].G + cls[x][y].H;

q->Array[q->length++] = &(cls[x][y]);

mintag = q->length - 1;

for (i = 0; i < q->length - 1; i++)

{

if (q->Array[i]->F < q->Array[mintag]->F)

{

mintag = i;

}

}

t = q->Array[q->length - 1];

q->Array[q->length - 1] = q->Array[mintag];

q->Array[mintag] = t; //将评价函数值最小节点置于队头

}

Close\* shift(Open \*q)

{

return q->Array[--q->length];

}

// 地图初始化操作

void initClose(Close cls[Height][Width], int sx, int sy, int dx, int dy)

{ // 地图Close表初始化配置

int i, j;

for (i = 0; i < Height; i++)

{

for (j = 0; j < Width; j++)

{

cls[i][j].cur = &graph[i][j]; // Close表所指节点

cls[i][j].vis = !graph[i][j].reachable; // 是否被访问

cls[i][j].from = NULL; // 所来节点

cls[i][j].G = cls[i][j].F = 0;

cls[i][j].H = abs(dx - i) + abs(dy - j); // 评价函数值

}

}

cls[sx][sy].F = cls[sx][sy].H; //起始点评价初始值

// cls[sy][sy].G = 0; //移步花费代价值

cls[dx][dy].G = Infinity;

}

void initGraph(const int map[Height][Width], int sx, int sy, int dx, int dy)

{ //地图发生变化时重新构造地

int i, j;

srcX = sx; //起点X坐标

srcY = sy; //起点Y坐标

dstX = dx; //终点X坐标

dstY = dy; //终点Y坐标

for (i = 0; i < Height; i++)

{

for (j = 0; j < Width; j++)

{

graph[i][j].x = i; //地图坐标X

graph[i][j].y = j; //地图坐标Y

graph[i][j].value = map[i][j];

graph[i][j].reachable = (graph[i][j].value == Reachable); // 节点可到达性

graph[i][j].sur = 0; //邻接节点个数

if (!graph[i][j].reachable)

{

continue;

}

if (j > 0)

{

if (graph[i][j - 1].reachable) // left节点可以到达

{

graph[i][j].sur |= West;

graph[i][j - 1].sur |= East;

}

if (i > 0)

{

if (graph[i - 1][j - 1].reachable

&& graph[i - 1][j].reachable

&& graph[i][j - 1].reachable) // up-left节点可以到达

{

graph[i][j].sur |= North\_West;

graph[i - 1][j - 1].sur |= South\_East;

}

}

}

if (i > 0)

{

if (graph[i - 1][j].reachable) // up节点可以到达

{

graph[i][j].sur |= North;

graph[i - 1][j].sur |= South;

}

if (j < Width - 1)

{

if (graph[i - 1][j + 1].reachable

&& graph[i - 1][j].reachable

&& map[i][j + 1] == Reachable) // up-right节点可以到达

{

graph[i][j].sur |= North\_East;

graph[i - 1][j + 1].sur |= South\_West;

}

}

}

}

}

}

int bfs()

{

int times = 0;

int i, curX, curY, surX, surY;

unsigned char f = 0, r = 1;

Close \*p;

Close \*q[MaxLength] = { &close[srcX][srcY] };

initClose(close, srcX, srcY, dstX, dstY);

close[srcX][srcY].vis = 1;

while (r != f)

{

p = q[f];

f = (f + 1) % MaxLength;

curX = p->cur->x;

curY = p->cur->y;

for (i = 0; i < 8; i++)

{

if (!(p->cur->sur & (1 << i)))

{

continue;

}

surX = curX + dir[i].x;

surY = curY + dir[i].y;

if (!close[surX][surY].vis)

{

close[surX][surY].from = p;

close[surX][surY].vis = 1;

close[surX][surY].G = p->G + 1;

q[r] = &close[surX][surY];

r = (r + 1) % MaxLength;

}

}

times++;

}

return times;

}

int astar()

{ // A\*算法遍历

//int times = 0;

int i, curX, curY, surX, surY;

float surG;

Open q; //Open表

Close \*p;

initOpen(&q);

initClose(close, srcX, srcY, dstX, dstY);

close[srcX][srcY].vis = 1;

push(&q, close, srcX, srcY, 0);

while (q.length)

{ //times++;

p = shift(&q);

curX = p->cur->x;

curY = p->cur->y;

if (!p->H)

{

return Sequential;

}

for (i = 0; i < 8; i++)

{

if (!(p->cur->sur & (1 << i)))

{

continue;

}

surX = curX + dir[i].x;

surY = curY + dir[i].y;

if (!close[surX][surY].vis)

{

close[surX][surY].vis = 1;

close[surX][surY].from = p;

surG = p->G + sqrt((curX - surX) \* (curX - surX) + (curY - surY) \* (curY - surY));

push(&q, close, surX, surY, surG);

}

}

}

return NoSolution; //无结果

}

const int map[Height][Width] = {

{ 0,0,0,0,0,1,0,0,0,1,0,0,0,0,1,0,0,0,1,1 },

{ 0,0,1,1,0,0,1,1,0,0,0,1,0,0,0,0,0,1,0,1 },

{ 1,0,1,0,0,0,1,0,0,0,0,0,0,1,1,0,0,0,0,1 },

{ 0,0,1,0,0,1,0,1,0,1,0,0,0,1,0,1,0,1,0,0 },

{ 0,1,0,1,0,0,1,0,0,0,1,0,0,1,0,0,0,1,0,1 },

{ 0,0,0,1,0,1,0,0,1,0,0,1,0,1,0,1,0,0,0,0 },

{ 0,1,0,0,0,0,1,0,1,0,1,0,0,0,1,0,0,0,1,0 },

{ 0,1,0,0,1,1,0,0,0,0,0,0,0,1,0,1,0,1,0,0 },

{ 0,1,0,1,0,0,0,0,0,1,1,0,1,0,0,0,1,0,0,0 },

{ 0,0,0,0,1,0,1,0,1,1,0,1,0,0,1,1,0,0,0,0 },

{ 0,1,1,0,0,0,0,1,0,1,1,1,0,1,0,0,0,1,0,0 },

{ 0,0,0,0,1,0,0,1,0,1,0,0,1,0,0,1,0,0,0,0 },

{ 0,0,0,0,0,1,0,0,0,0,1,1,0,0,0,0,0,1,1,0 }, //12

{ 0,1,0,0,0,0,1,0,0,0,0,0,0,1,0,1,0,0,0,1 },

{ 0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0 }

};

const char Symbol[5][3] = { "□", "▓", "▽", "☆", "◎" };

void printMap()

{

int i, j;

for (i = 0; i < Height; i++)

{

for (j = 0; j < Width; j++)

{

printf("%s", Symbol[graph[i][j].value]);

}

puts("");

}

puts("");

}

Close\* getShortest()

{ // 获取最短路径

int result = astar();

Close \*p, \*t, \*q = NULL;

switch (result)

{

case Sequential: //顺序最近

p = &(close[dstX][dstY]);

while (p) //转置路径

{

t = p->from;

p->from = q;

q = p;

p = t;

}

close[srcX][srcY].from = q->from;

return &(close[srcX][srcY]);

case NoSolution:

return NULL;

}

return NULL;

}

static Close \*start;

static int shortestep;

int printShortest()

{

Close \*p;

int step = 0;

p = getShortest();

start = p;

if (!p)

{

return 0;

}

else

{

while (p->from)

{

graph[p->cur->x][p->cur->y].value = Pass;

printf("（%d，%d）→\n", p->cur->x, p->cur->y);

p = p->from;

step++;

}

printf("（%d，%d）\n", p->cur->x, p->cur->y);

graph[srcX][srcY].value = Source;

graph[dstX][dstY].value = Destination;

return step;

}

}

void clearMap()

{ // Clear Map Marks of Steps

Close \*p = start;

while (p)

{

graph[p->cur->x][p->cur->y].value = Reachable;

p = p->from;

}

graph[srcX][srcY].value = map[srcX][srcY];

graph[dstX][dstY].value = map[dstX][dstY];

}

void printDepth()

{

int i, j;

for (i = 0; i < Height; i++)

{

for (j = 0; j < Width; j++)

{

if (map[i][j])

{

printf("%s ", Symbol[graph[i][j].value]);

}

else

{

printf("%2.0lf ", close[i][j].G);

}

}

puts("");

}

puts("");

}

void printSur()

{

int i, j;

for (i = 0; i < Height; i++)

{

for (j = 0; j < Width; j++)

{

printf("%02x ", graph[i][j].sur);

}

puts("");

}

puts("");

}

void printH()

{

int i, j;

for (i = 0; i < Height; i++)

{

for (j = 0; j < Width; j++)

{

printf("%02d ", close[i][j].H);

}

puts("");

}

puts("");

}

int main(int argc, const char \*\*argv)

{

initGraph(map, 0, 0, 0, 0);

printMap();

while (scanf\_s("%d %d %d %d", &srcX, &srcY, &dstX, &dstY) != EOF)

{

if (within(srcX, srcY) && within(dstX, dstY))

{

if (shortestep = printShortest())

{

printf("从（%d，%d）到（%d，%d）的最短步数是: %d\n",

srcX, srcY, dstX, dstY, shortestep);

printMap();

clearMap();

bfs();

//printDepth();

puts((shortestep == close[dstX][dstY].G) ? "正确" : "错误");

clearMap();

}

else

{

printf("从（%d，%d）不可到达（%d，%d）\n",

srcX, srcY, dstX, dstY);

}

}

else

{

puts("输入错误！");

}

}

return (0);

}

# **二、上机实习**

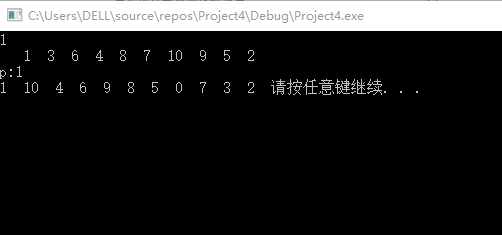
# 第一题

## 1. 实现结果及代码测试

数据输入：

A[1] = 1; A[2] = 3; A[3] = 6; A[4] = 4; A[5] = 8; A[6] = 7; A[7] = 10; A[8] = 9; A[9] = 5; A[10] = 2;

数据输出：



## 2. 实现中遇到的主要困难及解决方法

开始编码的时候，一开始是使用python实现，因此在编程过程中，由于python是没有指针和引用传递的，因此在编写函数mergesort时，导致p传递不出来，最后无法实现递归，想了很多办法，比如使用函数的返回值传递出来，但是又引出了函数参数确定的问题，最后还是用C语言实现。

在进行改进的归并算法之前，我先实现的是经典的归并算法，在对其进行分析的时候，在函数参数传递过程中，我使用的指针必须初始化为空，最后遇到了堆栈溢出的问题，在对其进行断点分析之后，发现传递过去后的指针的内容发生了变化，因此使用了无需初始化的引用参数传递。

## 3. 本次实习的心得与体会

对于我们计算机作业的学生来说，数据结构和算法设计都是非常重要的课程，在整个设计的过程中，让我们认真的独立思考，当然在遇到困难的时候，也可以寻找同学请求帮助，比如解决这道题时，我在指针问题上面遇到了难题，之后是同学再帮我分析找出了解决方案。通过课程设计让我们不断的发现自己的不足从而去改善，这是一种学习的态度，不仅仅是在这次的课程设计中，在以后的无论生活还是学习方面都应该注意和努力改善。

## 4. 源代码

#include<stdio.h>

#include <stdlib.h>

#include <iostream>

using namespace std;

int A[11];

int LINK[11];

void InsertSort(int A[], int Link[], int low, int high, int&p) {

int i, k, small = low, max = low;

int b;

for (i = low; i <= high; i++) {

if (A[small] >= A[i])

small = i;

if (A[max] <= A[i])

max = i;

}

p = small;

for (i = low; i <= high; i++) {

b = max;

for (int j = low; j <= high; j++) {

if (A[i] < A[j] && A[j]<A[b])

b = j;

}

Link[i] = b;

}

Link[max] = 0;

}

void merge(int q, int r, int &p) {

int i, j, k;

i = q;

j = r;

k = 0;

while (i != 0 && j != 0) {

if (A[i] <= A[j]){

LINK[k] = i;

k = i;

i = LINK[i];

}

else {

LINK[k] = j;

k = j;

j = LINK[j];

}

}

if (i == 0)

LINK[k] = j;

else

LINK[k] = i;

p = LINK[0];

}

void mergesort(int low,int high,int &p) {

if (high - low + 1<3){

InsertSort(A,LINK, low, high, p);

}

else {

int mid = (low + high) / 2;

int q, r;

mergesort(low, mid, q);

mergesort(mid + 1, high, r);

merge(q, r, p);

}

}

int main() {

A[1] = 1;

A[2] = 3;

A[3] = 6;

A[4] = 4;

A[5] = 8;

A[6] = 7;

A[7] = 10;

A[8] = 9;

A[9] = 5;

A[10] = 2;

cout << A[1];

for (int i = 0; i <= 10; i++)

LINK[i] = 0;

int p;

mergesort(1, 10, p);

cout << "\n ";

for (int i = 1; i <= 10; i++) {

cout << A[i] << " ";

}

cout << "\n";

cout << "p:" << p<<"\n";

for (int i = 0; i <= 10; i++) {

cout <<LINK[i] << " ";

}

system("pause");

}

# 第二题

## 实现结果及代码测试

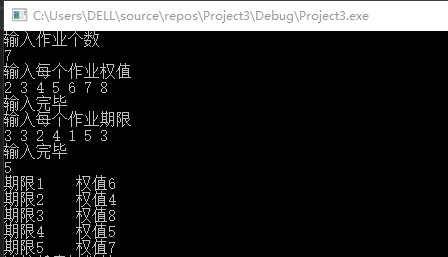
输 入：

作业个数：7

作业权限：2 3 4 5 6 7 8

作业期限：3 3 2 4 1 5 3

实验结果：



## 实现中遇到的主要困难及解决方法

在这道题上，我主要花时间在sort排序上面，这个排序使用的是快速排序方法，虽然在数据结构中有学习到，但是并没有进行编码过，主要是因为快速排序使用了分治的思想。之前循环判断时没有判断没考虑到查找到末尾，导致出现了数组越界，最后重写了while循环判断条件，改正之后排序结果就正确了。

## 3. 本次实习的心得与体会

调试程序的能力得到了提高。在对这种相对以前写的程序复杂的多的程序进行调试时，会比以往困难的多，当程序写到一定规模的时候，会很容易的写到后面忘了前面程序的思路，导致会花很长时间来重新理清一下思维，这让我懂得了及时添加注释的重要性。而且，在进行调试，寻找bug的过程中，让我更加的熟练使用断点以及单步调试，使自己解决问题的能力更上一层楼。

## 4. 源代码

#include <stdio.h>

#include <iostream>

#define MAX 100

using namespace std;

int JS(int date[], int J[],int n,int k) {

date[0] = 0;

J[0] = 0;

J[1] = 1;

k = 1;

int r;

for (int i = 2; i <= n; i++) {

r = k;

while (date[J[r]] > date[i] && date[J[r]] != r)

r = r - 1;

if (date[J[r]] <= date[i] && date[i] > r) {

for (int j = k; j > r ; j--) {

J[j + 1] = J[j];

}

J[r + 1] = i;

k++;

}

}

return k;

}

void sort(int p[], int date[],int n) {

for (int i = 1; i <= n; i++) {

int k = i - 1;

p[0] = p[i];

date[0] = date[i];

while (p[0] > p[k]) {

p[k + 1] = p[k];

date[k + 1] = date[k];

k--;

}

p[k + 1] = p[0];

date[k + 1] = date[0];

}

}

int main() {

int date[MAX];

int J[MAX];

int p[MAX];

int num;

cout << "输入作业个数" << endl;

cin >> num;

cout << "输入每个作业权值" << endl;

for (int i = 1; i <= num; i++) {

cin >> p[i];

}

cout << "输入完毕" << endl;

cout << "输入每个作业期限" << endl;

for (int i = 1; i <= num; i++) {

cin >> date[i];

}

cout << "输入完毕" << endl;

sort(p, date, num);

int work=JS(date, J, num,num);

cout << work<<endl;

for (int i = 1; i <= work; i++)

cout <<"期限" <<date[J[i]] << " 权值" << p[J[i]] << endl;

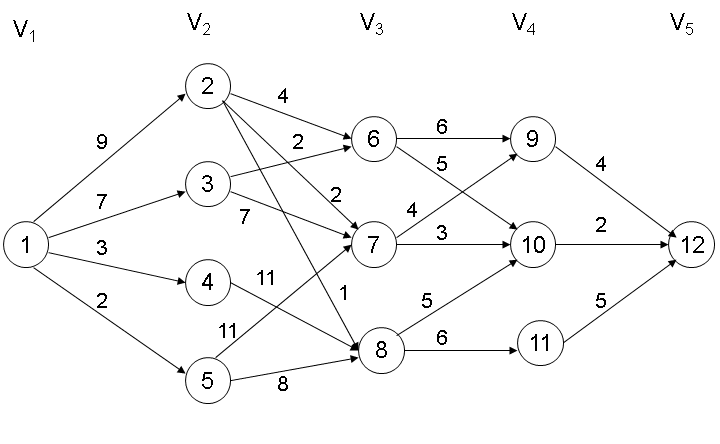
system("pause");

}

# 第三题

## 实现结果及代码测试

数据图：



数据输入：

weight[0][1] = 9; weight[0][2] = 7; weight[0][3] = 3; weight[0][4] = 2; weight[1][5] = 4; weight[1][6] = 2; weight[1][7] = 1; weight[2][5] = 2; weight[2][6] = 7; weight[3][7] = 11; weight[4][6] = 11; weight[4][7] = 8;weight[5][8] = 6; weight[5][9] = 5; weight[6][8] = 4; weight[6][9] = 3; weight[7][9] = 5; weight[7][10] = 6;weight[8][11] = 4; weight[9][11] = 2; weight[10][11] = 5;

实验结果：

打印出的路径：



## 实现中遇到的主要困难及解决方法

在对搜索路径进行编码的过程中，出现了不可预期的错误，之后在寻找bug的过程中，让我更加的熟练使用断点以及单步调试，使自己解决问题的能力更上一层楼。

## 3. 本次实习的心得与体会

通过课程设计让我们不断的发现自己的不足从而去改善，这是一种学习的态度，不仅仅是在这次的课程设计中，在以后的无论生活还是学习方面都应该注意和努力改善。本道题是使用了动态规划思想，通过对一个问题进行分解来寻找到子问题的最优解。做完这道题，加深了我对动态规划的掌握能力。

## 4. 源代码

#include "iostream"

#include "stdlib.h"

#define MAX 333

#define K 5

using namespace std;

typedef struct Node {

int i;

int quan;

struct Node \*next;

}Node;

int main() {

int weight[12][12];

Node\* L[12];

int s;

int path[K];

for (s = 0; s < 12; s++)

for (int j = 0; j < 12; j++)

weight[s][j] = MAX;

weight[0][1] = 9; weight[0][2] = 7; weight[0][3] = 3; weight[0][4] = 2;

weight[1][5] = 4; weight[1][6] = 2; weight[1][7] = 1; weight[2][5] = 2; weight[2][6] = 7; weight[3][7] = 11; weight[4][6] = 11; weight[4][7] = 8;

weight[5][8] = 6; weight[5][9] = 5; weight[6][8] = 4; weight[6][9] = 3; weight[7][9] = 5; weight[7][10] = 6;

weight[8][11] = 4; weight[9][11] = 2; weight[10][11] = 5;

for (int i = 0; i < 12; i++)

L[i] = (Node \*)malloc(sizeof(Node));

Node \*q ,\*p;

for (int i = 0; i < 12; i++)

{

q = L[i];

for (int j = 0; j < 12; j++)

if (weight[i][j] > 0 && weight[i][j] < MAX)

{

p = (Node \*)malloc(sizeof(Node));

p->i = j;

p->quan = weight[i][j];

q->next = p;

q = p;

}

q->next = NULL;

}

int i, j, maxlen, temp, v[12], d[12];

for (i = 0; i < 12; i++)

v[i] = 0;

for (i = 10; i >= 0; i--)

{

for (maxlen = MAX, j = i + 1; j <12; j++)

if (weight[i][j] > 0 && weight[i][j] + v[j] < maxlen)

{

maxlen = weight[i][j] + v[j];

temp = j;

}

v[i] = maxlen;

d[i] = temp;

}

path[0] = 0;

path[K - 1] = 11;

for (i = 1; i <= K - 2; i++)

path[i] = d[path[i - 1]];

for (i = 0; i < K; i++)

printf("%d ", path[i] + 1);

printf("\n");

system("pause");

return 0;}