人工智能导论 大作业一
兔子吃胡萝卜

自 66 林文镔 2016011503

目录

1. 需求分析

本次大作业需要实现一个具有路径搜索功能的图形应用程序。

该项目需要实现路径搜索算法，在给定的起点、终点以及地图下找到从起点到终点的路径，并实现搜索路径的可视化。路径搜索可提供多种搜索算法、不同节点扩展方式、不同启发函数的选项。

同时，为显示搜索算法的效果，需添加搜索路径代价与搜索计算时间的显示。

此外，该项目需要提供直观简洁的图形界面，便于用于设置搜索选项与执行搜索操作，并提供便捷的地图编辑、地图读写等功能。

2. 开发及编译环境

本项目使用 C++ 作为编程语言，基于 Qt 5.11.0 进行开发，使用 QtCreater 作为 IDE，编译器为 MinGW 5.3.0 32bit。若已安装 QtCreater，则使用 QtCreater 打开本项目中的.pro 文件即可打开本项目。

运行平台方面，本项目在 Window 10 下进行开发，理论上使用 Qt 开发的程序可跨平台运行，但本项目的运行试验仅在 Window 10 环境下进行，推荐使用 Window 系统运行本项目。

3. 程序操作方式

本项目实现的图形界面效果如下：

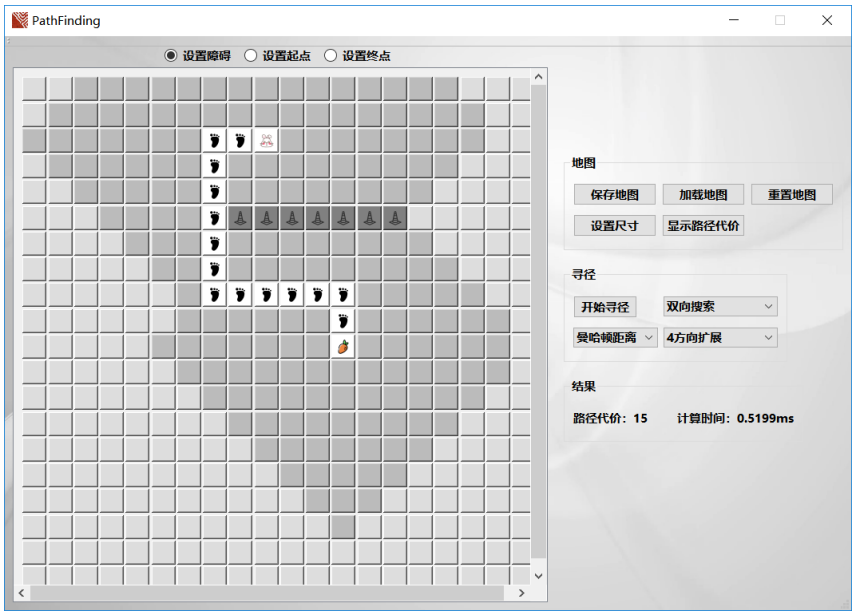


图 1: 用户界面效果

3.1 地图编辑与含义

3.1.1 地图编辑

界面的左侧为地图，地图上方有 3 个按钮，分别为“设置障碍”、“设置起点”和“设置终点”，选中其中之一后即进入相应的地图编辑模式。

进入编辑模式后，使用鼠标左键点击地图中的单元即可进行地图编辑。若使用鼠标右键点击地图，则可以设置该地图单元的路径代代价，所谓路径代价，即从相邻接到到该节点需花费的代价。路径代价相当于一个刻画地形的参数，其引入使得问题更有实际意义。

3.1.2 辅助功能

界面右侧最上方的一个控件组为其他地图相关的操作按钮，其功能与按钮上的文字对应，即“保存地图”、“加载地图”、“重置地图”、“设置尺寸”和“显示路径代价”。

本项目中地图将以 csv 文件格式保存，且仅能读取 csv 格式的地图文件。需要额外注意的是，待加载的 csv 文件路径中不能含有中文，否则无法加载。

重置地图则地图将被清空。

设置尺寸则可以在弹出的对话框中输入新的尺寸，尺寸的高度和宽度的范围均为 $[3, 99]$。

显示路径代价则可以显示或隐藏路径代价，而路径代价的显示并不实时更新，若重新编辑地图，则需重新点击按钮，以显示路径代价。

3.1.3 地图含义

地图中，浅灰色地图单元表示该单元可达；深灰色且带有“路障”图标的单元为不可达，即障碍物；白色背景“兔子”图标的单元为起点；白色障碍“胡萝卜”图标的单元为终点；白色背景“脚印”图标的单元为最终的搜索路径；中等灰度的单元表示曾被访问过。

3.2 寻径

界面右侧的处于中间位置的控件组为寻径相关的按钮，其中包括“开始寻径”的按钮，以及 3 个搜索选项。

右上方为搜索方式选项，其中含有 8 种搜索方式；左下方为启发函数选项，含有 3 中启发函数，启发函数仅对贪婪最佳优先搜索、A* 搜索和动态加权 A* 搜索有效；右下方为节点扩展方式选项，可选择 4 方向或 8 方向，4 方向依次为上、左、下、右，8 方向扩展则依次为上、左、下、右、左上、左下、右上、右下。

3.3 寻径结果

界面右侧下方的控件组为搜索结果，路径代价为从起点到终点所耗费的路径代价之和，计算时间仅包括运行寻径算法所耗费的时间，不计入路径在界面中的渲染等过程的时间。

4. 问题建模

本项目所解决的问题是一类路径搜索问题，需要在地图上找到从起点到终点的路径，路径搜索的目标为路径代价最小化，同时需要使搜索算法占用尽量小的时间和内存资源。

本项目中的路径搜索问题本质上仍然是状态空间搜索问题，可将问题以状态空间的语言进行建模。

状态：当前位置。对于本项目，当前位置可以是地图上任意一个可达的位置。设地图尺寸为 $H \times W$，其中有 K 的障碍物，则状态数为 $H \times W - K$。由于本项目仅需关注一个节点的位置，不需要进行复杂的排列组合，因此相对于国际象棋、围棋这类问题，本项目所需处理的问题在状态空间是很小的，故搜索时在内存基本不会成为搜索算法的限制。

初始状态：当前位置位于起点。

目标状态：当前位置位于终点。

后继函数：从当前位置出发，向当前位置的邻域移动（此处的邻域可以为上、下、左、右 4 个方向，也可以是上、下、左、右、左上、左下、右上、右下 8 个方向），且要求邻域可达，即目标邻域上不存在障碍物。

代价函数：即从当前位置出发，到相邻节点所需要付出的路径代价，此代价默认状态下为 1，也可为每个节点单独设置路径代价，程序中限制路径代价范围为 $[1, 9]$。

分支因子估计：由于本项目中的地图结构，地图节点的连边数量较多。而分支因子受到节点扩展方式、障碍物、地图边界以及搜索算法中复杂的剪枝所影响，很难准确地估计出平均意义下的分支因子。

在此，假设地图较大且不存在障碍物，进行简单的估计。因地图较大，忽略地图边界无法扩展的情况，在节点扩展方式为 4 邻域时，每个节点与 4 个节点形成连边，由于连边是两个节点共有的，总体上连边的数量约为节点数量的两倍，可近似将 2 作为分支因子的估计值。同理，节点扩展方式为 8 邻域时，分支因子的估计值为 4。

5. 程序架构

5.1 总体架构

本项目的系统流程图如下：

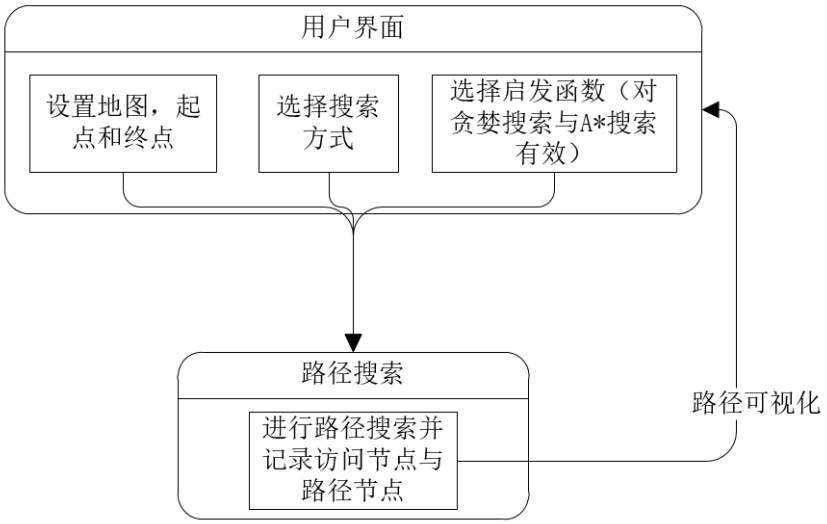


图 2: 系统流程图

5.2 节点定义

对于搜索中使用的节点，定义结构体如下：

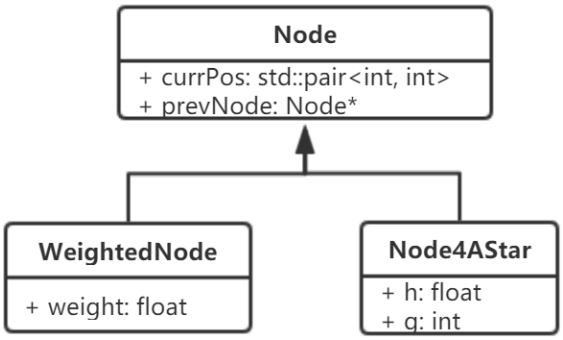


图 3: 节点结构体

节点中包含其节点位置，以及该节点的父节点，保存父节点可用于路径回溯，在完成搜索后，可沿父节点找到搜索路径。

基于 Node 结构体，派生了 WeightedNode 和 Node4AStar 两个结构体 WeightedNode 引入了节点权重，用于贪婪最佳优先搜索和一致代价搜索，Node4AStar 引入了路径代价 g 和启发函数 h 用于 A* 搜索和动态加权的 A* 搜索。

此外，本项目中针对 Node 节点定义了相等判定函数和哈希函数，其中相等判定函数比较节点位置是否相等，哈希函数值为 $1000 \times y + x$，其中 y 为行序号，x 为列序号，在本项目中限制的地图大小范围内，可保证该哈希函数不出现冲突。基于自定义的相等判定函数和哈希函数可使用 unordered_set 和 unordered_map 的数据结构，进行高效的节点管理。

对于 WeightedNode 和 Node4AStar 结构体，定义了大小比较函数，以便使用 priority_queue 的数据结构，获取最优节点。

5.3 搜索算法

本项目中，将各种搜索算法都定义成类，以便管理，其结构如下：

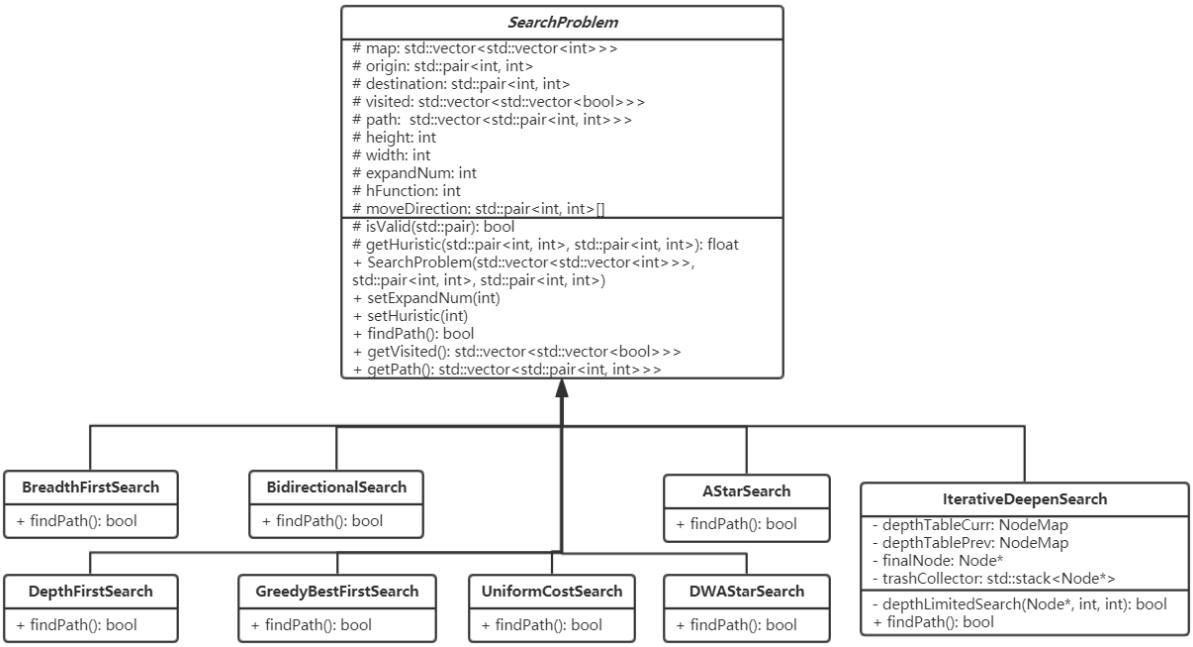


图 4: 搜索算法类架构

本项目中实现的 8 种搜索算法类均继承自抽象类 SearchProblem，该抽象类中包含了地图信息、起点终点、节点扩展方式、启发函数类型、访问节点表以及路径表等信息，提供通用的节点有效性判定函数、启发函数计算函数、扩展方式与启发函数的设置函数、路径以及访问节点表的获取函数等接口。其中唯一的纯虚函数为 findPath 函数，即路径搜索函数，各个具体的搜索算法类仅需重写此函数即可。

这样的架构，减少了代码的冗余，且结构清晰。在使用时，仅需实例化与选定的搜索算法相对应的类，并调用 findPath 函数即可。

5.4 用户界面

用户界面部分提供用于交互的操作按钮、搜索相关选项以及地图编辑按钮等。本项目中与图形界面有关的类有 MainWindow、DialogSetCost、DialogSetMapSize 和 Cell。其中，MainWindow 为主界面，用于放置部件，DialogSetCost 和 DialogSetMapSize 分别为设置路径代价和设置地图尺寸的对话框，Cell 类为地图的中的一个单元，MainWindow 中含有 Cell 对象的二维 vector。

Cell 类继承自 QPushButton，作为地图中的一个单元，通过重写 mousePressEvent 函数，可获得按钮鼠标左键和右键事件，左键用于设置障碍、起点和终点，右键用于设置路径代价。此外该类提供设置按钮显示效果的接口，供主界面调用。

6. 算法实现

本项目中，实现了宽度优先搜索、深度优先搜索、双向宽度优先搜索、迭代加深的深度优先搜索、贪婪最佳优先搜索、一致代价搜索、A* 搜索和动态加权的 A* 搜索，以下对这些搜索算法是实

现进行简要的说明。算法的效率对比等将在后续的部分介绍。

6.1 宽度优先搜索

宽度优先搜索利用 queue 的数据结构，利用其先进先出的特性，实现逐层遍历。由于本项目中的地图格点相对有限，闭节点表直接采用的 $h \times w$ 的 bool 类型矩阵进行维护，其中 h 和 w 分别为地图的高和宽，以下搜索算法若未特殊说明则同样采用此数据结构维护闭节点表。

6.2 深度优先搜索

深度优先搜搜利用 stack 的数据结构，利用其后进先出的特性，实现深度优先搜索。

6.3 双向宽度优先搜索

双向宽度优先搜索基于宽度优先搜索实现，在起点和终点两端同时进行宽度优先搜索，当两端访问的节点出现交汇时，则搜索完成。需要注意的是，节点扩展方式为 8 方向时，双向宽度优先搜索应优先拓展对角的节点，以保证两端的搜索尽可能早的出现交汇。

6.4 迭代加深的深度优先搜索

迭代加深的深度优先搜索通过将搜索的最大深度迭代增加，循环调用深度受限搜索实现。此算法在课堂中介绍相对简要，我个人认为而此算法在实现难度上，是本项目中难度最高的，故在此给出该算法的具体实现。

Algorithm 1 Iterative deepening depth-first search

Require:

map, origin, destination

Ensure:

path from origin to destination, visited node

```
1: function ITERATIVE DEEPENING DEPTH-FIRST SEARCH
2:     depthTablePrev = HashMap<node, depth>()
3:     depthTableCurr = HashMap<node, depth>()
4:     for i = 1; i<∞ ; ++i do
5:         depthTablePrev = depthTableCurr
6:         depthTableCurr.clear()
7:         if depthLimitedSearch(origin, 0, i) then
8:             return true
9:         else if depthTablePrev.size == depthTableCurr.size then
10:            return false
11:        end if
12:    end for
13:    return false
14: end function
15: function DEPTH-LIMITED DFS(node, currentDepth, maxDepth)
16:    if currentDepth > maxDepth then
17:        return false
18:    end if
19:    if node == destination then
20:        return true
21:    end if
22:    depthTableCurr[node] = currentDepth
23:    for each nextNode in the neighborhood of node do
24:        if isValid(nextNode) then
25:            if !depthTablePrev.find(nextNode)
26:                || depthTablePrev[nextNode] >= (currentDepth + 1) then
27:                if !depthTableCurr.find(nextNode)
28:                    || depthTableCurr[nextNode] > (currentDepth + 1) then
29:                    visited[currNode] = true
30:                    if depthLimitedSearch(nextNode, currentDepth+1, maxDepth) then
31:                        return true
32:                    end if
33:                end if
34:            end if
35:        end if
36:    end for
37:    return false
38: end function
```

以上算法中，深度受限搜索通过递归式的深度优先搜索实现，以便记录搜索的深度。

算法的核心在于深度表的维护，该表记录各个结点相对初始结点的深度。深度表的作用在于，在当前搜索深度已较高时，可避免对深度较低节点的访问，因此深度表可完全取代闭节点表，更重要的是，深度表实现了额外的剪枝。

此外，算法中在维护当前深度表 depthTableCurr 的同时，还使用了前一次搜索中留下的深度表 depthTablePrev，depthTablePrev 为搜索提供先验信息，尤其在当新深度表构建尚且不够完整时，这部分先验信息尤为重要。需要注意的是，对于 depthTablePrev，允许在后继节点的深度小于或等于 depthTablePrev 中深度时进行访问，而对于 depthTableCurr 后继节点的深度必须严格小于 depthTableCurr 中的深度。

当不存在合理的搜索路径时，若不断将深度限制提高，深度表将完全覆盖可达的区域。若深度限制提高前后深度表的大小不变，则不存在合理的搜索路径，因此，可以通过对比 depthTablePrev 和 depthTableCurr 的尺寸是否相等判断是否存在合理的搜索路径。

6.5 贪婪最佳优先搜索

贪婪最佳优先搜索，每次优先访问与目标节点最接近的节点。开节点表使用优先级队列维护，以保证 $O(log(n))$ 的访问效率。

6.6 一致代价搜索

一致代价搜索即使用迪杰斯特拉算法，优先访问路径代价最低的节点，同样使用了优先级队列。

6.7 A* 搜索

A* 搜索在算法实现上与贪婪最佳优先搜索类似，不同之处在于，节点访问的优先级综合考虑了路径代价与启发函数，即 $g(n) + h(n)$。

6.8 动态加权的 A* 搜索

动态加权的 A* 搜索在 A* 搜索的基础上引入了启发函数的权重，即节点的访问优先级函数为：

$$g(n) + (1 + \varepsilon w(n))h(n)$$

$$w(n) = \begin{cases} 1 - \dfrac{d(n)}{N} & d(n) \leq N \\ 0 & else \end{cases}$$

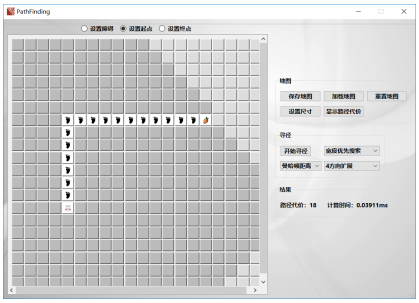
其中 N 为对路径代价的估计值，在本项目的算法实现中，该数值取 $3 \times h(origin)$，即 3 倍初始位置启发函数值，$d(n)$ 为当前累积路径代价，取 $\varepsilon = 0.5$。

7. 运行效果与实验

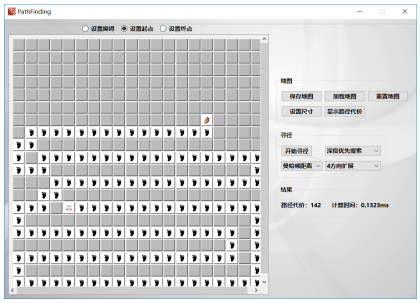
以下实验中使用的地图，均保存在 map 文件夹下，若需使用可直接加载。测试时，使用的电脑配置为 Windows 10 x64，Intel Core i7-6700HQ 2.6GHz。

7.1 无障碍物情况

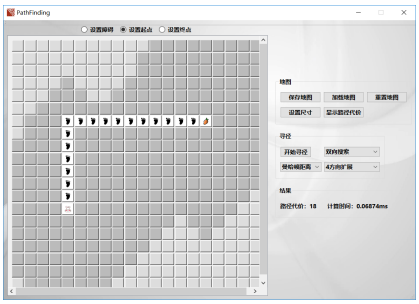
设置地图大小为 20×20，不添加障碍物，且地图上任意节点的路径代价均为 1，节点拓展方式选择 4 方向，启发函数选择曼哈顿距离，地图文件见 map/test1.csv。



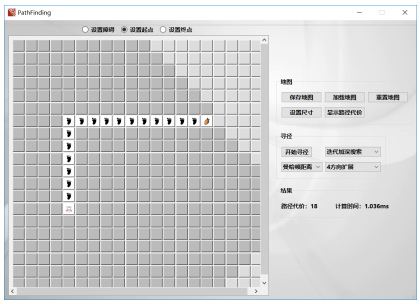
(a) 宽度优先搜索



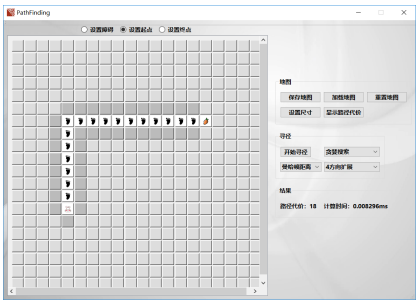
(b) 深度优先搜索



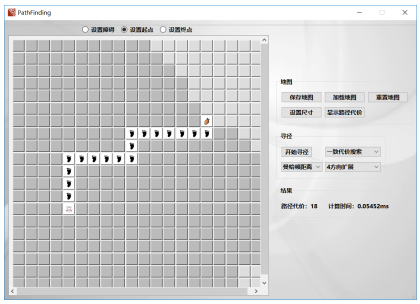
(c) 双向宽度优先搜索



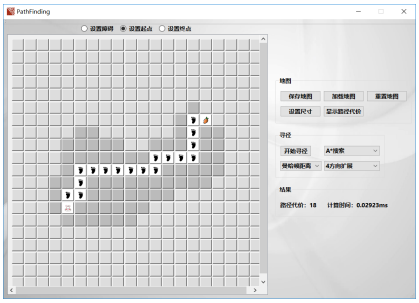
(d) 迭代加深的深度优先搜索



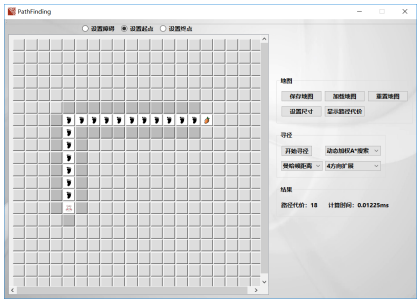
(e) 贪婪最佳优先搜索



(f) 一致代价搜索



(g) A* 搜索



(h) 动态加权的 A* 搜索

图 5: 无障碍物测试

从效果上看，宽度优先搜索逐层搜索，访问了深度小于目标节点深度的所有节点，迭代加深的深度优先搜索在访问结果上与宽度优先搜索时相同的，此外由于每个节点的路径代价相同，一致代价搜索退化为宽度优先搜索，仅在节点访问的先后顺序上有一定区别。

深度优先搜索的搜索路径饶了许多弯路，由于程序中对节点邻域的扩展顺序的固定的，因此深

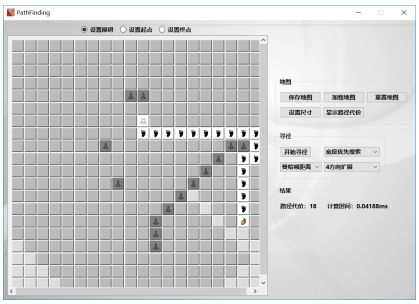
度优先搜索往往会沿固定的探索顺序，如图中所示的先向右探索到尽头，随后又向下探索到尽头。可见能够较快地达到目标节点具有较大的偶然性。

双向宽度优先搜索的搜索区域相当于面积较小的两个宽度优先区域的叠加，在一定程度上能够减少节点的访问量。

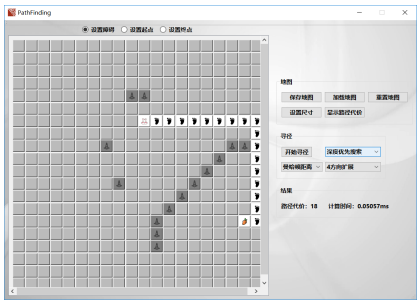
贪婪最佳优先搜索在无障碍物时表现出优良的特性，不对路径外的节点进行访问，直接找到通往终点的路径，而动态加权的 A* 算法，由于对启发函数增加了额外的权重，也表现出了贪婪的特性，而 A* 算法则对路径周围的节点进行的一定的访问，但总体节点访问数量仍然是不大的。

7.2 添加少量障碍物

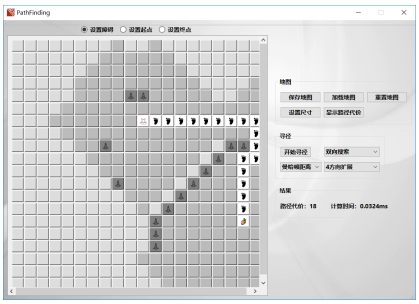
在无障碍物地图的基础上，添加少量量的障碍物，进行测试，地图文件见 map/test2.csv。



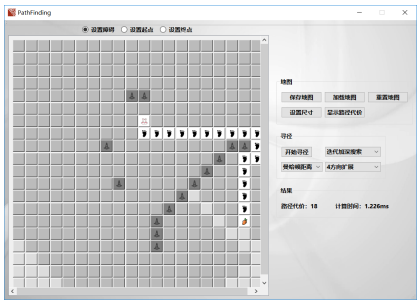
(a) 宽度优先搜索



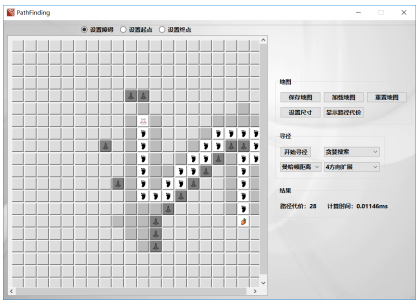
(b) 深度优先搜索



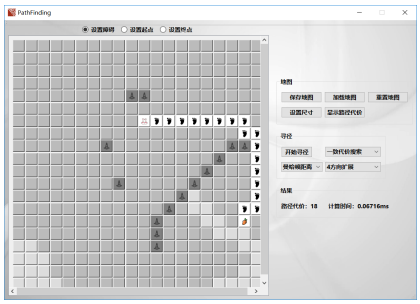
(c) 双向宽度优先搜索



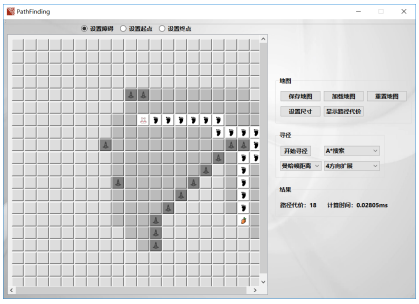
(d) 迭代加深的深度优先搜索



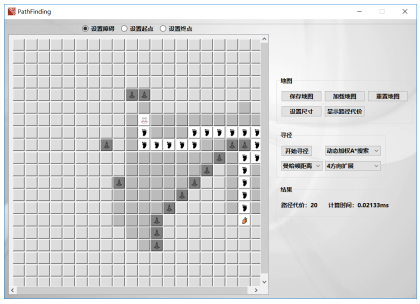
(e) 贪婪最佳优先搜索



(f) 一致代价搜索



(g) A* 搜索



(h) 动态加权的 A* 搜索

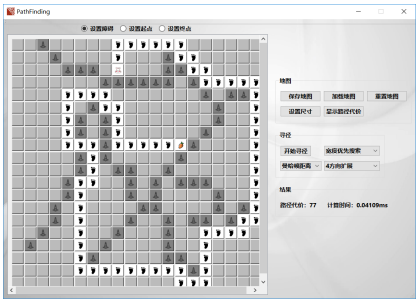
图 6: 少量障碍物

可见宽度优先搜索、迭代加深的深度优先搜索和一致代价搜索在路径搜索的结果上仍然是基本一致的。而双向宽度优先搜索和深度优先搜索的特性也与无障碍情况类似，以下主要分析贪婪最佳优先搜索、A* 搜索和动态加权的 A* 搜索在有障碍物情况下表现出的不同特性。

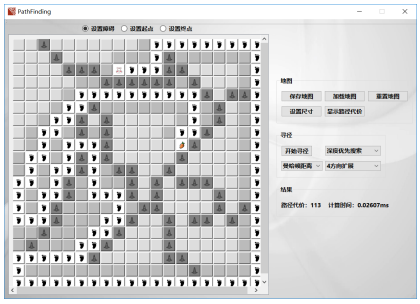
从路径代价上看，A* 搜索 < 动态加权的 A* 搜索 < 贪婪最佳优先搜索，而从访问的节点数量上看贪婪最佳优先搜索 < 动态加权的 A* 搜素 <A* 搜索。可见，A* 算法能保证搜索路径的最优性，而动态加权的 A* 算法路径由于增加了额外的权重，导致其启发函数不再具有一致性，因此不能保证搜索路径最优。从中可见，路径的最优性与搜索的效率不能两全，动态加权的 A* 算法是对贪婪最佳优先搜索和 A* 搜索的折中综合。

7.3 添加大量障碍物

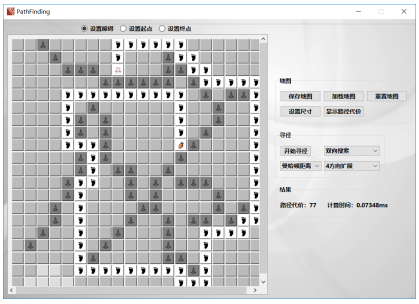
添加大量障碍物，再次进行测试，地图文件见 map/test3.csv。



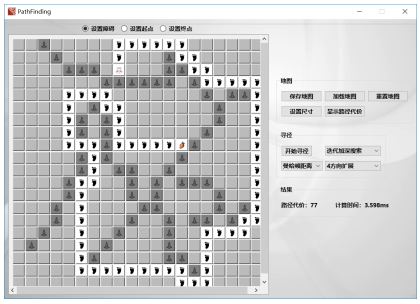
(a) 宽度优先搜索



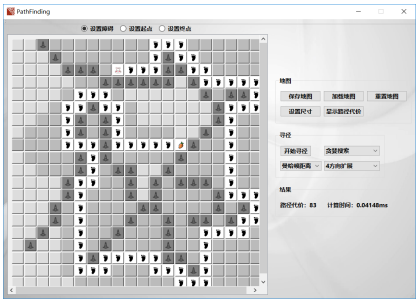
(b) 深度优先搜索



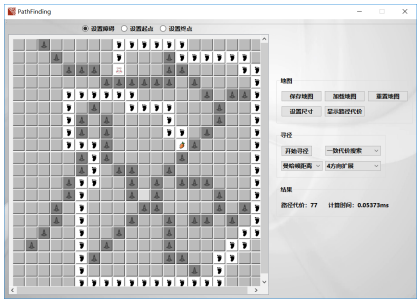
(c) 双向宽度优先搜索



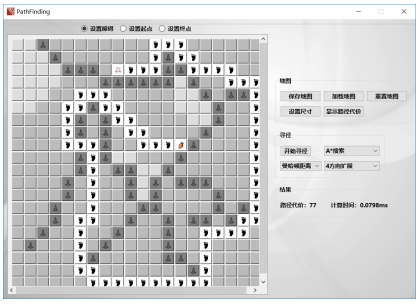
(d) 迭代加深的深度优先搜索



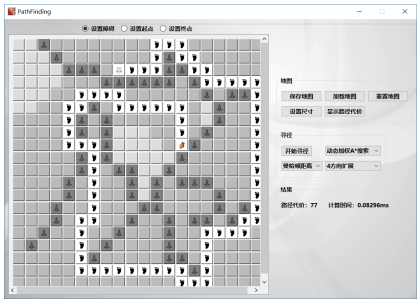
(e) 贪婪最佳优先搜索



(f) 一致代价搜索



(g) A* 搜索



(h) 动态加权的 A* 搜索

图 7: 大量障碍物

由以上测试结果可以看出，当地图中存在大量障碍物时，搜索算法几乎需要遍历所有可达节点。在测试中，深度优先搜索偶然地访问了较少的节点，但搜索路径仍然较长。此外，贪婪最佳优先搜索也减少了部分节点的访问，但也牺牲了路径的最优性。而 A* 算法与动态加权的 A* 算法搜索效果基本相同。

可以看出，当地图复杂度较高时，启发函数对于 A* 搜索和动态加权 A* 搜索的指导意义被削

弱了。

7.4 不存在搜索路径

添加障碍物将终点包围，使得合理的搜索路径不存在，测试算法的完整性与稳定性，由于路径无法找到，以下仅以迭代加深的深度优先搜索为例，展示路径不存在时，程序的运行效果，地图文件见 map/test4.csv。

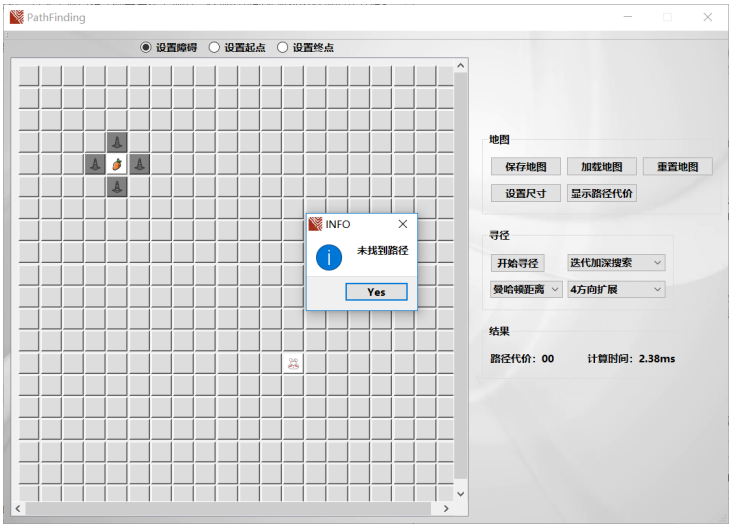
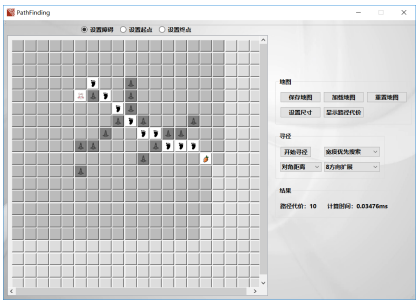


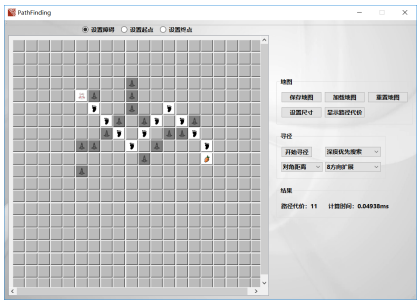
图 8: 未找到路径

7.5 不同的节点扩展方式

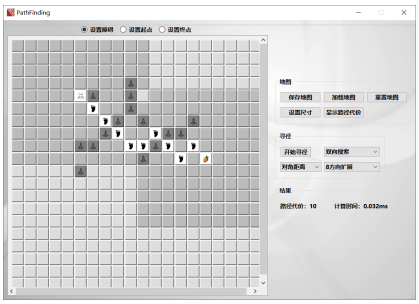
先前的测试中对节点邻域的扩展方式均为 4 方向扩展，以下选择邻域扩展方式为 8 方向，同时将启发函数更换为对角距离，进行测试，地图文件见 map/test5.csv。



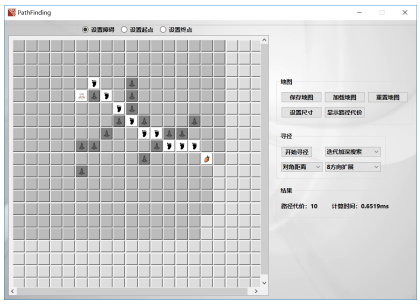
(a) 宽度优先搜索



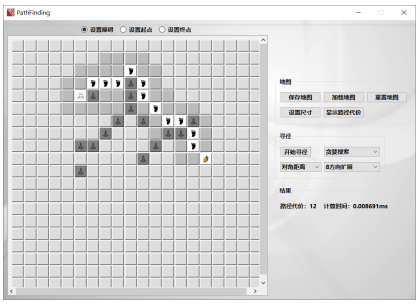
(b) 深度优先搜索



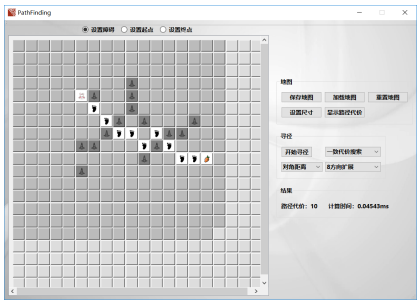
(c) 双向宽度优先搜索



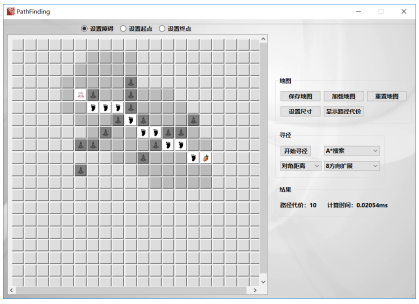
(d) 迭代加深的深度优先搜索



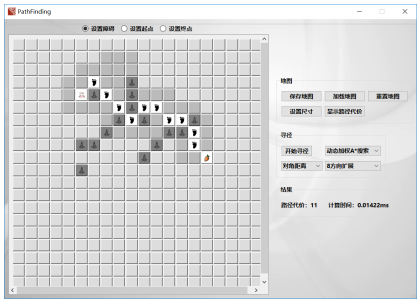
(e) 贪婪最佳优先搜索



(f) 一致代价搜索



(g) A* 搜索



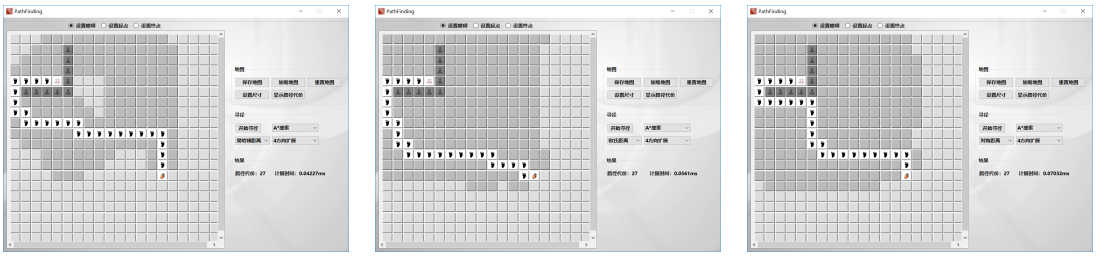
(h) 动态加权的 A* 搜索

图 9: 8 方向扩展

可见将扩展方式更换为 8 方向后，搜索时可使用较小的路径代价到达目标点，而各种搜索算法在特性上，并未表现出与先前的不同。

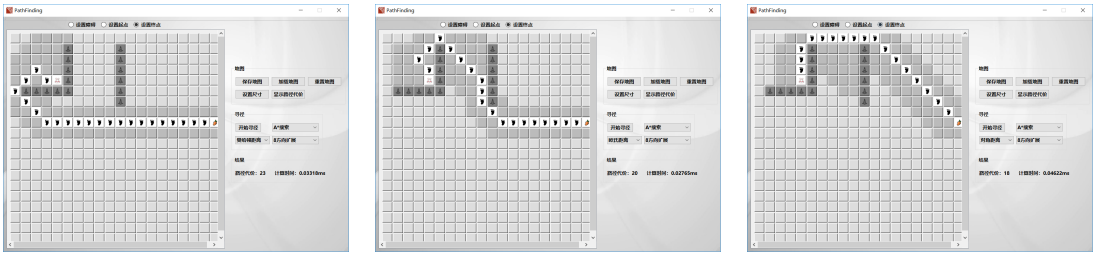
7.6 不同的启发函数

启发函数的不同对贪婪最佳优先搜索、A* 搜索和动态加权的 A* 搜索均会产生影响且在不同的邻域扩展方式下，启发函数将展现出不同的特性，由于更改启发函数对以上提到的 3 中搜索算法的影响是类似的，故以下以 A* 算法为例，进行实验，地图文件见 map/test6.csv 和 map/test7.csv。



(a) 曼哈顿距离　　(b) 欧氏距离　　(c) 对角距离

图 10: 4 方向扩展，不同启发函数下的 A* 搜索



(a) 曼哈顿距离　　(b) 欧氏距离　　(c) 对角距离

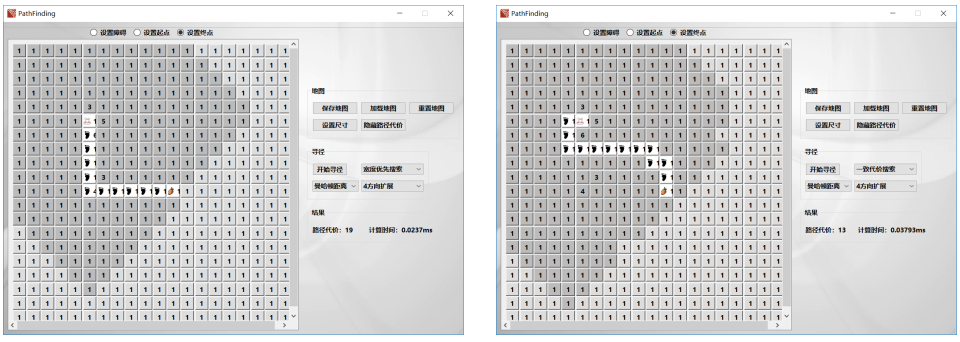
图 11: 8 方向扩展，不同启发函数下的 A* 搜索

当节点扩展方式为 4 邻域时，曼哈顿距离、欧式距离和对角距离都是对路径代价的乐观估计，都具有一致性，均能够保证搜索路径的最优性，而三种启发函数对 4 方向扩展的启发性依次下降，故在实验中表现为访问的节点数量依次增加。

当节点扩展方式为 8 邻域时，只有对角距离具有一致性，故只有使用对角距离时，才能保证找到最优距离，而使用曼哈顿距离或欧式距离，则使搜索具有更多的贪婪的特性，搜索时虽然都能较少访问的节点，但牺牲了最优性。

7.7　设置路径代价

由于宽度优先搜索、深度优先搜索、双向宽度优先搜索、迭代加深的深度优先搜索以及贪婪最佳优先搜索此 5 种算法均不考虑路径代价，故路径代价仅对一致代价搜索、A* 搜索和动态加权的 A* 搜索 3 算法产生影响。由于在先前的测试中，一致代价搜索与宽度优先搜索均为表现出差别，故此处对此两种算法进行实验，地图文件见 map/test8.csv。



(a) 宽度优先搜索　　(b) 一致代价搜索

图 12: 设置节点路径代价

当路径代价不全相等时，宽度优先搜索、双向宽度优先搜索以及迭代加深的深度优先搜索都无法保证最优性。

7.8 大型地图

使用尺寸较大的地图进行测试，能更好地反映算法效率。将地图大小设置为 99×99（本项目中地图大小上限值），设置起点为左上角，终点为右下角，不设置障碍，且节点路径代价均为 1，地图文件见 map/test9.csv。

由于不存在搜索路径时，每一种搜索算法都需遍历所有的可达节点，这也可在某种程度上反映出搜索算法维护自身搜索路径的效率，故将不存在搜索路径的大型地图也作为反映程序运行效率的测试。同样取起点为左上角，终点为右下角，仅在终点处添加障碍物，将其封锁，地图文件见 map/test10.csv。

最终得到的测试结果如下表：

搜索算法	4 方向扩展 (ms)	8 方向扩展 (ms)
宽度优先	0.985	1.176
深度优先	0.0321	0.0557
双向宽度优先	0.956	0.663
迭代加深的深度优先	297	144
贪婪最佳优先	0.0428	0.0439
一致代价	1.39	1.72
A*	0.0657	0.0732
动态加权 A*	0.0614	0.0944

表 1: 无障碍的大型地图搜索效率测试

搜索算法	4 方向扩展 (ms)	8 方向扩展 (ms)
宽度优先	0.978	1.14
深度优先	0.958	1.20
双向宽度优先	0.0292	0.0297
迭代加深的深度优先	294	146
贪婪最佳优先	1.56	1.86
一致代价	1.41	1.63
A*	4.06	7.43
动态加权 A*	3.98	7.15

表 2: 无搜索路径的大型地图搜索效率测试

需要说明的是，表中的用时测量为界面上显示的计算时间，且均进行了多次计算取平均，使用 4 方向扩展时启发函数为曼哈顿距离，使用 8 方向扩展时启发函数为对角距离。

此外，实际进行大地图测试时，程序运行的等待时间将出现较为明显的增长，而增长的时间主要是图形界面效果渲染等过程的时间，搜索算法的耗时虽然也有增长，但从计算时间上看，显然不是主要因素。

地图中无障碍时，深度优先搜索（具有偶然性）、贪婪最佳优先搜索、A* 和动态加权的 A* 搜索以及 8 方向扩展时的双向宽度优先搜索，由于访问节点数量较少，在效率上展现出较大的优势。

当无搜索路径时，除双向宽度优先搜索外的其余所有算法都需要遍历所有可行节点，这使得在无障碍情况时访问节点较少的算法失去了优势。

在本项目的 8 种搜索算法中，贪婪最佳优先搜索、一致代价搜索、A* 搜索和动态加权的 A* 搜索都使用了优先级队列，优先级队列虽然能够保证 $O(log(n))$ 级别的访问速度，但终究是比生成后继节点效率为常数级别的宽度优先搜索、深度优先搜索以及双向宽度优先搜索慢。尤其在无有效搜索路径时，4 种使用优先级队列的算法，搜索耗时均大于宽度优先搜索和深度优先搜索。

当节点邻域扩展方式由 4 方向变为 8 方向后，带来的是分支因子在一定程度上的增大，同时 8 邻域扩展中的对角扩展又使算法可在更小的深度到达更远的距离。故算法效率受到这两个因素的综合影响。使用优先级队列的 4 中算法以及宽度优先搜索、深度优先搜索受分支因子增大的影响更大，表现为耗时增加。而迭代加深的深度优先搜索受深度减小的影响更大，故耗时减少。

总体上，除迭代加深的深度优先搜索外的所有搜索算法在测试中的耗时都至少在几毫米量级，而迭代加深的深度优先搜索耗时则为百毫秒量级。理论上，对于较为平衡的树搜索而言，迭代加深的深度优先搜索的时间复杂度应宽度优先搜索的常数倍，但在测试中，迭代加深的宽度优先搜索效率为宽度优先搜索的数百倍。这是由于测试中每一层深度加深带来的节点增加数量并不是指数增长的，而是基本为线性增长，甚至会由于地图边界的限制而减小，故迭代加深的深度优先搜索在效率上存在明显的劣势。

理论上，以上算法在空间效率上，也存在一定区别，但由于本项目中状态数目并不大，各种算法的内存占用都不大，故很难进行比较分析。这也导致深度优先搜索、迭代加深的深度优先搜索等算法的优势无法体现。

8. 总结

1. 在本次大作业中，我对课堂上介绍的各种搜索算法进行了编程实现与效率分析，如老师所言，理解算法的捷径就是动手实践，在此过程中，我对这些搜索算法有了更深刻的理解。尤其对各个搜索算法之间的区别与联系方面，有了更加立体的印象。

2. 本次大作业的实现过程中，令我印象最为深刻同时也是耗费最多精力的是迭代加深的深度优先搜索的实现，虽然最终该方法的实现代码量并不大，但是在实现过程中进行了反复的 debug 与效率优化的尝试，至少实现了 3 种不同版本的实现方式。从中锻炼的自己的编程与思维能力，并且加深了对算法的理解。