数据结构 课程大作业 实验报告

14307130078 张博洋

一、程序亮点

1. **采用自己编写的 R 树作为索引的数据结构**（仅支持插入和查询）
2. **可以图形化显示程序内部数据结构**（显示 R 树矩形和最短路树）
3. **极高的画图效率**（可采用 R 树辅助画图）
4. **较低的内存占用**（载入上海地图后，占用内存约 400MB）
5. **出租车数据不全部读入内存**（先扫描建立索引然后按需读取）
6. **可载入其他地区的 OSM 地图**（没有硬编码的数组大小）
7. **模块化的源代码组织方式，重要语句带有注释**（约 20 个 C++ 类）
8. **可在运行时根据配置文件调整参数**（多数参数不是硬编码在代码中的）

二、程序使用说明

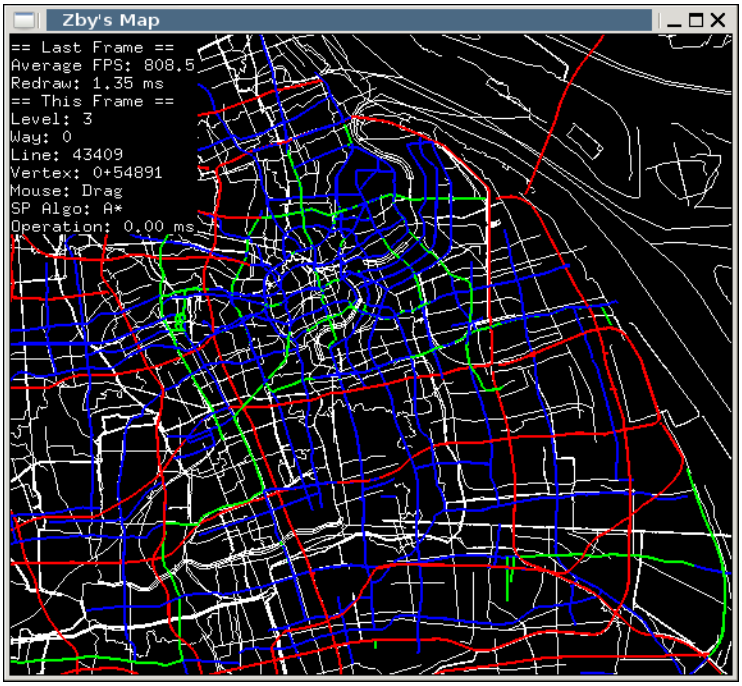
1. 系统要求

Windows 或 Linux 操作系统

内存 512MB 以上

支持 OpenGL 1.5 的显卡

2. 界面说明



打开程序后，大约需要 20 秒来加载数据，其间会有信息显示在程序的终端窗口。程序主界面打开后，位于左上角的白底黑字的是信息栏，剩余部分为地图。

3. 地图可视化

主界面打开后，可以通过鼠标拖拽或键盘上的方向键来移动地图视野，可以通过鼠标滚轮或键盘上的“+-”、“PageDown”、“PageUp”键来缩放地图视野。此外，可以按 F2 键来显示 R 树中的矩形（再次按可关闭显示）。

4．查询地图对象的信息

按键盘上的 Tab 键可以把鼠标模式从拖拽模式切换为选择模式，此时可以通过鼠标左键选择点，鼠标右键选择路，当前选中的点或路的简略信息会显示在信息栏中。按 F11 可以查看当前选中的点的详细信息（如经纬度、名称等），按 F12 可以查看当前选中的路的详细信息（如长度、面积、标签等）。再次按 Tab 键可将鼠标模式切换回拖拽模式。

5．最短路的查询

先将鼠标移至起点位置，按 A 键设置当前鼠标位置为起点，此时应有红色方框标记出起点。再讲鼠标移至终点位置，按 S 键设置当前鼠标位置为终点，此时应有橘色方框标记出终点。按 F5 来计算最短路，计算完毕后最短路会以红色标注在地图上，信息栏中也会显示最短路的简略信息。若要查看详细信息，可以按 F7 键。若要切换算法，需先按“~”（即数字 1 左侧的键），当前选定的算法会标注在信息栏中的“SP Algo”位置，再按 F5 在使用新的算法计算最短路。此外，还可以按 F6 键来显示最短路树（再次按可关闭显示）。操作完毕后可按 D 键清除地图上的标记。

6．范围查询与近邻查询

范围查询与近邻查询用到了多边形顶点列表，它的使用方法如下：按 Z 键可以把鼠标当前位置作为多边形顶点追加到列表中，按 X 键可以清除多边形顶点列表。当列表中的顶点数目大于等于 2 时，程序会在界面上画出多边形。F4 键的查询范围会根据当前列表中顶点的个数变化。当顶点数目小于等于 1 时，查询范围为当前地图视野；当顶点数目恰为 2 时，查询范围是以第一个顶点为圆心，两顶点直接距离为半径的园；当顶点数目大于等于 3 时，查询范围是列表中的顶点组成的多边形。在使用 Z 键添加完顶点后，按 F4 会弹出输入框，此时输入要查询的兴趣点类型（如“food”、“supermarket”、“transport”、“money”等，具体的兴趣点定义在配置文件 config.txt 中以 MAPTAG 开头的项中），输入完毕后按回车即可执行查询。查询结果的简略信息会显示在信息栏中，也可通过 F8 键查看查询结果的详细信息。此时按 1234567890 数字键可以居中显示结果中的点，按 QWERTYUIOP 键可以居中显示查询结果中的路。浏览完毕后，可以按 C 键清除地图上的查询结果。

7．按名称查询地图对象

按 F3 键并输入要查询的名称（如“复旦”）即可查询地图上所有名字中含有给定字符串的地图对象。之后的操作方法类似范围查询和近邻查询。

8．出租车轨迹查询

按“/”键会弹出输入窗口，此时输入出租车编号并按回车即可执行查询。查询到的出租车轨迹会以黄绿两色显示在地图上，黄色表明该段为空驶状态，绿色表明该段为载客状态。同时路径的信息（如空驶率、行驶路程等）会显示在信息栏中。按“;”可以将路径起点居中显示。按“<”或“>”键可以将上一路径点或下一路径点居中显示。按“M”键输入路径点编号并回车可以居中显示对应路径点。操作完毕后，可以按 L 键清除地图上的查询结果。

9．按键功能表

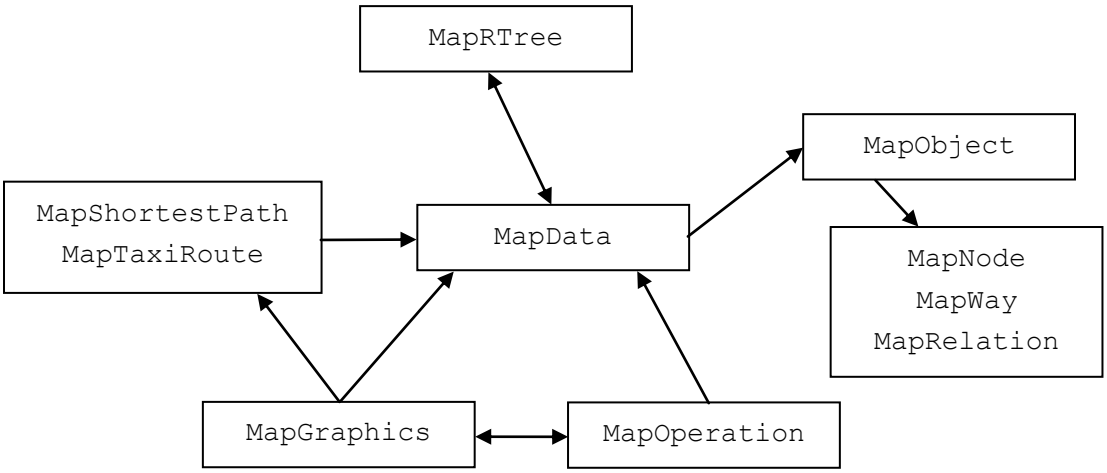
按键	功能说明
方向键、鼠标拖动	移动题图视野，鼠标拖动仅在鼠标模式为拖拽模式有效
+ 或 = 或 PageDown	放大地图视野
- 或 _ 或 PageUp	缩小地图视野
| 或 \	切换信息栏显示
ESC	清除所有地图上的标记
F1	重置地图的视图
F2	显示 R 树的矩形结构
F3	按名称查询地图对象
F4	按兴趣点类型查询地图对象
F5	计算最短路
F6	显示最短路树
F7	显示最短路详细信息
F8	显示上次查询地图对象的详细信息
F9	居中显示当前选定的点（Node）
F10	居中显示当前选定的路（Way）
F11	显示当前选定的点的详细信息
F12	显示当前选定的路的详细信息
` 或 ~（1 左侧按键）	切换最短路算法
Tab	切换当前鼠标模式（在拖拽模式和选择模式之间切换）
1234567890	选择并居中对应数字编号的点
SHIFT+1234567890	将当前选定的点设置为对应数字编号的点
QWERTYUIOP	选择并居中对应字母编号的路
SHIFT+QWERTYUIOP	将当前选定的路设置为对应字母编号的路
A	标记当前鼠标位置为最短路起点
S	标记当前鼠标位置为最短路终点
D	清除地图上的最短路标记
Z	添加当前鼠标位置作为多边形顶点
X	清除所有多边形定点
C	清除查询结果
' 或 "	查询出租车路径数据的索引
/ 或 ?	载入指定的出租车路径数据
: 或 ;	居中显示出租车路径起点
> 或 .	居中显示下一出租车路径点
< 或 ,	居中显示上一出租车路径点
M	居中显示给定的出租车路径点
L	清除出租车路径

三、程序源代码说明

1．简介

程序代码约有 4300 行，大小约 120K，全部为手工编写。采用模块化的设计，共有 20 余个 C++ 类，分别负责不同的功能。重要的语句与变量定义带有注释。另外程序可以跨平台地在 Linux 上或 Windows 上编译运行。

2．源代码组织结构示意图



3．重要的类的说明

类名	功能说明
MapData	程序的核心，保存地图数据，提供部分操作数据的函数
MapRTree	R 树的模板类，可以索引并管理 MapNode、MapLine 数据
MapObject	地图数据对象，是 MapNode、MapWay、MapRelation 的基类
MapNode	地图数据中 Node 型数据对应的类，也是最短路中的顶点
MapWay	地图数据中 Way 型数据对应的类，含有 MapNode 的有序列表
MapLine	表示 Way 型地图数据中的线段，也是最短路中的边
MapRelation	地图数据中 Relation 型数据对应的类，只预留了位置，尚未实现
MapGraphics	绘制地图的类，使用 OpenGL 与 freeglut 画图
MapOperation	处理用户操作的类，与 MapGraphics 一起实现用户交互
MapShortestPath	最短路算法类，实现了 A*、Dijkstra、SPFA 等算法
MapDict	字典类，负责管理地图对象的名字，采用了后缀数组思想
MapVector	点类 MapPoint 和向量类 MapVector，实现基础计算几何算法
MapRect	矩形类，实现了有关矩形的功能

4．使用的开源库及用途

库名称	用途
libxml2	读取 XML 格式的地图文件
freeglut, glew	OpenGL 绘图的框架库
wxWidgets	绘制图形界面

四、程序中用到的数据结构

1．R 树

本程序中用到了 R 树，对应的类是 MapRTree。它是一个模板类，可以索引并管理带有 get_rect()方法的地图数据对象。它采用的是平方级的节点分裂算法，每个节点 M 值上限为 15，下限为 7。此实现仅支持插入和查询两种操作，即：插入对象、查询所有与矩形范围相交的数据对象。在本程序中，R 树被用于做范围查询和近邻查询，也可以用来辅助画图。

（1）范围查询和近邻查询

这两种查询的算法如下：先求出要查询的范围的包围矩形，在 R 树中查询该矩形，然后在查询结果中暴力过筛，选择出符合要求的结果。因此查询效率与所查询的范围的包围矩形密切相关。

（2）辅助画图

鉴于直接将所有地图数据对象画图会导致画图速度较慢，一种自然想到的优化方法就是只绘制视野范围内的地图对象。这就需要 R 树来帮忙了，每次画图时用当前视野矩形查询 R 树，只绘制查询到的结果，这样以来便可以大大减小要绘制的地图对象个数，加快地图绘制速度。

在程序设计之初，采用 R 树而不是 K-D 树的原因之一就是为了辅助画图。使用 K-D 树画图有一种缺陷：当两点在视野范围之外，而两点之间的线段穿过了视野时，K-D 树无法发现这种情况，会把该线段舍弃，因此会出现地图放大过程中某条路突然消失的情况。而 R 树没有此缺陷，R 树中的对象均以矩形形式存在，只要把所有线段放入 R 树管理，无论视野范围多小，它都会与视野范围相交，因此 R 树的查询结果一定不会丢失线段。

但是，使用 R 树辅助画图并不能显著提高画图效率。因为经过实验后发现，画图效率的瓶颈在于 CPU 与显卡之间的数据交换。如果每次绘图都查询 R 树，则每次都需要向显卡更新顶点信息，并不会将效率提高太多。一种更高级的优化是，将所有地图对象的顶点数据在程序开始执行时就一次性地提交到显卡，之后每次绘图都直接绘制所有图形。这样虽然看起来显卡要画的东西变多了，实际上由于减少了效率瓶颈，再加上显卡内部的优化，可以大大地提高绘图效率。

2．出租车路径数据的索引

出租车路径数据规模较大，将所有数据一次性读入到内存中的代价较大。我采用的方法是：用常数的空间先扫描一遍整个文件，统计出每辆出租车信息的开始位置，建立一个索引列表。当查询出租车路径时，先会在索引中查询到对应出租车信息的开始位置，再使用 fseek 函数将文件指针定位到该位置，然后读取该辆出租车的路径数据。这样常驻在内存中的数据只有一张索引表，和一辆出租车的路径数据。在不影响功能的情况下，大大地减小了内存占用。

五、性能分析

1．R 树的性能分析

（1）定性分析

根据 R 树的原始论文，R 树的节点分裂算法有多种实现。由于节点分裂仅在插入时发生，因此节点分裂算法的复杂度直接关系到插入操作的时间。另外由于节点分裂会对树的形态造成影响，因此节点分裂结果的优劣会间接地关系到查询操作的时间。

对于本程序来讲，多种节点分裂算法中，综合性能较优的是平方级的节点分裂算法。它通过损耗一部分插入操作的速度，换取较为优秀的查询速度。因为对于本程序来说，插入操作仅在预处理时发生，插入操作的速度只会影响到预处理速度。而之后的用户查询才是重头戏，需要查询速度越快越好。因此平方级节点分裂算法很好地符合本程序的使用场景的需求。

（2）实验分析

通过实验可以大致地了解 R 树的查询速度。以下的实验数据是将上海地图分为许多个小块后，分别进行查询而得到的查询耗时数据，每种分块方案取结果数量前 4 大的数据。

分块个数	查询时间（毫秒）/结果数量（个）			
1*1=1	14.56ms/534294			
2*2=4	5.91ms/234505	2.95ms/112359	2.80ms/108838	1.94ms/79505
3*3=9	4.71ms/179992	2.59ms/97886	1.56ms/61296	1.31ms/58898
4*4=16	3.41ms/131442	1.73ms/67656	1.32ms/53669	0.91ms/44589
5*5=25	2.62ms/101127	1.76ms/72522	1.01ms/47258	0.90ms/42867
6*6=36	2.23ms/90408	0.89ms/44720	0.81ms/40549	0.72ms/37789
7*7=49	1.32ms/56779	0.92ms/46279	0.84ms/40628	0.59ms/34278
8*8=64	1.10ms/49809	0.81ms/38889	0.50ms/30509	0.36ms/26141
9*9=81	1.07ms/49530	0.73ms/37575	0.66ms/35936	0.41ms/26211
10*10=100	0.71ms/36498	0.41ms/26947	0.43ms/26679	0.39ms/24569
11*11=121	0.70ms/34986	0.54ms/32668	0.42ms/25627	0.31ms/21214
12*12=144	0.68ms/35718	0.39ms/24331	0.25ms/17672	0.24ms/15821
13*13=169	0.48ms/28215	0.30ms/20387	0.31ms/19528	0.28ms/19079
14*14=196	0.54ms/30845	0.29ms/19470	0.27ms/19137	0.25ms/17951
15*15=225	0.40ms/25997	0.41ms/24482	0.29ms/19327	0.29ms/18886
16*16=256	0.27ms/20106	0.31ms/18683	0.23ms/16582	0.24ms/14657
17*17=289	0.37ms/23107	0.27ms/19086	0.22ms/16553	0.22ms/16130
18*18=324	0.23ms/16216	0.23ms/15376	0.26ms/15121	0.24ms/14196
19*19=361	0.34ms/20904	0.27ms/18305	0.21ms/15038	0.16ms/10314
20*20=400	0.22ms/13772	0.22ms/13406	0.19ms/13401	0.19ms/13015

可以看出，在查询得到的结果较多的时候，R 树大致速度与结果数量成近似的线性关系。

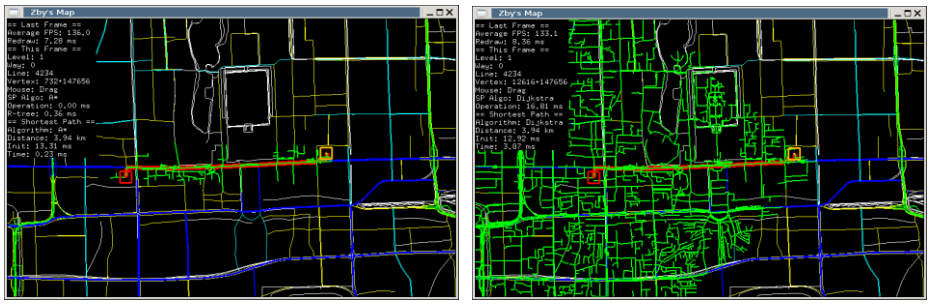
2. 最短路算法的性能分析

本程序中实现了三种最短路算法：A*、Dijkstra、SPFA。还有一个对比算法是 BFS（直接从源点广搜，搜到终点为止，因此结果并不是最短路）。

（1）定性分析

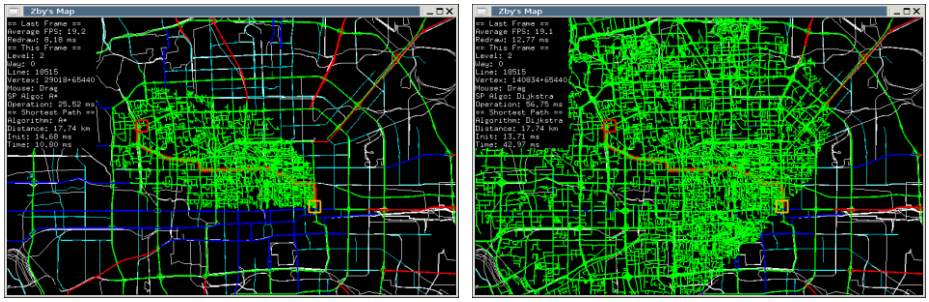
以北京地图为例，分析 A*与 Dijkstra 直接的速度差异。

A*算法十分占优的情形



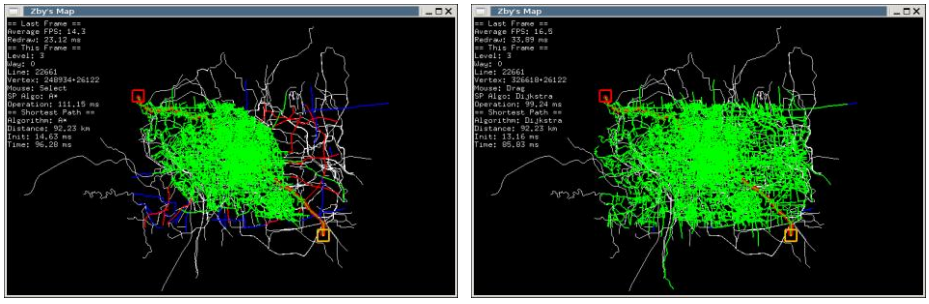
算法结束时，计算出的最短路树用绿色标注在图上。其中左侧为 A*算法，右侧为 Dijkstra 算法。此例子中，A*算法耗时约 0.2ms，Dijkstra 算法耗时 3.9ms。速度上 A*比 Dijkstra 快上十倍。通过最短路树可以看出，在这种走直线的情况下，A*算法涉及到的顶点和边明显少于 Dijkstra 算法。

A*算法比较占优的情形



一般情况下，A*的最短路树呈水滴状，而 Dijkstra 呈菱形状。此例子中，A*算法耗时约 11ms，Dijkstra 算法耗时 43ms。速度上 A*比 Dijkstra 快上四倍。虽然 A*算法涉及到的顶点和边明显少于 Dijkstra 算法，但是由于计算估价函数也需要不小的代价，因此 A*算法的优势不如之前十倍那样明显。

A*算法不占优的情形



此例子中，A*算法耗时约 96ms，Dijkstra 算法耗时 86ms。速度上 A*反而比 Dijkstra 慢约 10ms。这是因为在这种从一个边角到另一个边角的最短路上，A*的估价函数对速度的增加已经不足以抵消计算估价函数所需的时间了。因此即使 A*算法涉及到的顶点和边少于 Dijkstra 算法，但是 A*的总体执行速度比 Dijkstra 慢了不少。

（2）实验分析

通过实验可以测试平均情况下的最短路效率。程序随机从上海地图中挑选两个顶点作为起点和终点，然后用四种不同的最短路算法进行计算，记录它们的运行时间，重复 40 次。

A*	Dijkstra	SPFA	BFS
31.2ms/1.0000x	111.3ms/0.2801x	705.9ms/0.0442x	40.4ms/0.7713x
0.8ms/1.0000x	9.7ms/0.0804x	10.3ms/0.0751x	3.1ms/0.2509x
12.2ms/1.0000x	102.3ms/0.1197x	902.7ms/0.0136x	39.7ms/0.3084x
0.6ms/1.0000x	8.4ms/0.0684x	9.1ms/0.0632x	1.9ms/0.3003x
71.5ms/1.0000x	118.7ms/0.6017x	1273.8ms/0.0561x	45.7ms/1.5643x
2.8ms/1.0000x	11.8ms/0.2400x	18.6ms/0.1530x	3.5ms/0.8033x
5.0ms/1.0000x	27.9ms/0.1805x	213.6ms/0.0236x	12.2ms/0.4149x
36.2ms/1.0000x	115.9ms/0.3126x	1391.4ms/0.0260x	47.6ms/0.7619x
2.3ms/1.0000x	13.3ms/0.1735x	102.1ms/0.0226x	6.9ms/0.3362x
16.2ms/1.0000x	115.1ms/0.1407x	983.9ms/0.0165x	33.1ms/0.4896x
1.8ms/1.0000x	13.5ms/0.1332x	15.7ms/0.1148x	3.0ms/0.6076x
33.8ms/1.0000x	116.0ms/0.2910x	1237.7ms/0.0273x	46.2ms/0.7299x
25.0ms/1.0000x	63.4ms/0.3936x	550.7ms/0.0453x	29.6ms/0.8427x
0.8ms/1.0000x	7.2ms/0.1048x	10.7ms/0.0703x	1.6ms/0.4818x
1.5ms/1.0000x	13.1ms/0.1113x	7.7ms/0.1905x	2.5ms/0.5897x
41.9ms/1.0000x	110.8ms/0.3785x	719.4ms/0.0583x	37.1ms/1.1292x
18.5ms/1.0000x	70.0ms/0.2651x	942.0ms/0.0197x	26.4ms/0.7030x
4.6ms/1.0000x	42.8ms/0.1072x	394.1ms/0.0117x	23.4ms/0.1966x
27.3ms/1.0000x	106.1ms/0.2575x	1896.0ms/0.0144x	31.5ms/0.8676x
7.6ms/1.0000x	81.4ms/0.0933x	725.0ms/0.0105x	28.1ms/0.2708x
17.5ms/1.0000x	93.7ms/0.1872x	857.5ms/0.0205x	27.0ms/0.6489x
20.9ms/1.0000x	79.3ms/0.2634x	775.9ms/0.0269x	43.7ms/0.4781x
20.7ms/1.0000x	67.7ms/0.3055x	641.7ms/0.0322x	35.8ms/0.5780x
6.7ms/1.0000x	31.7ms/0.2123x	261.6ms/0.0257x	8.9ms/0.7592x
26.0ms/1.0000x	63.5ms/0.4099x	881.4ms/0.0295x	25.2ms/1.0339x
7.3ms/1.0000x	32.9ms/0.2210x	126.9ms/0.0572x	17.0ms/0.4284x
9.9ms/1.0000x	72.3ms/0.1370x	1157.2ms/0.0086x	45.5ms/0.2180x
7.3ms/1.0000x	71.9ms/0.1016x	647.3ms/0.0113x	28.1ms/0.2605x
1.9ms/1.0000x	18.2ms/0.1026x	90.8ms/0.0205x	9.5ms/0.1957x
15.7ms/1.0000x	84.1ms/0.1865x	1200.6ms/0.0131x	33.5ms/0.4680x
7.3ms/1.0000x	47.7ms/0.1522x	335.2ms/0.0217x	13.9ms/0.5239x
17.6ms/1.0000x	64.0ms/0.2744x	845.5ms/0.0208x	29.9ms/0.5878x
2.3ms/1.0000x	31.5ms/0.0730x	164.4ms/0.0140x	14.9ms/0.1547x
0.8ms/1.0000x	13.5ms/0.0578x	17.8ms/0.0437x	5.9ms/0.1320x
24.0ms/1.0000x	80.1ms/0.2995x	927.8ms/0.0259x	42.2ms/0.5689x
27.2ms/1.0000x	94.9ms/0.2871x	976.8ms/0.0279x	39.3ms/0.6929x
9.6ms/1.0000x	65.9ms/0.1455x	412.8ms/0.0232x	29.9ms/0.3205x
3.8ms/1.0000x	15.2ms/0.2488x	42.6ms/0.0886x	8.9ms/0.4220x
77.0ms/1.0000x	115.5ms/0.6661x	1370.2ms/0.0562x	45.3ms/1.6974x
8.1ms/1.0000x	68.3ms/0.1184x	644.5ms/0.0125x	25.0ms/0.3236x
16.3ms/1.0000x	61.8ms/0.2643x	612.2ms/0.0267x	24.8ms/0.6579x

表格最后一行是这 40 组数据的平均值。从平均值上可以看出，在一般情况下，A*的速度大约是 Dijkstra 的 4 倍。SPFA 的速度与 A*和 Dijkstra 不可同日而语，耗时大约是 A*的 40 倍。