北京工业大学

2017 - 2018 学年 第1学期

信息学部 计算机学院

|  |  |  |  |
| --- | --- | --- | --- |
| 课程名称： | 数据结构课程设计 | | |
| 报告性质： | 实验报告 | | |
| 学号： | 15143103 | 姓名： | 刘方瑞 |
| 任课教师： | 王众 | 课程性质： | 学科基础必修课 |
| 学分： | 2 | 学时： | 60 |
| 班级： | 150743 | 成绩： |  |
| 教师评语： |  | | |

|  |  |  |
| --- | --- | --- |
| 需求分析 | 根据题中需求，提供功能划分说明 |  |
| 设计 | 逻辑结构、存储结构设计、算法描述 |  |
| 使用说明 | 界面是否友好 |  |
| 总结 | 是否感悟有收获 |  |
| 摘要 | 考察文字抽象能力 |  |
| 格式 | 是否有目录、页号 |  |

2017年 月 日

目录

[一、 需求分析 - 2 -](#_Toc499558525)

[1.1 概述 - 2 -](#_Toc499558526)

[1.2 应用场景 - 2 -](#_Toc499558527)

[1.3 算法模型 - 2 -](#_Toc499558528)

[1.4 架构思路 - 3 -](#_Toc499558529)

[二、 数据结构设计 - 4 -](#_Toc499558530)

[2.1 数据结构模型 - 4 -](#_Toc499558531)

[2.1.1 逻辑模型 - 4 -](#_Toc499558532)

[2.1.2 存储模型 - 5 -](#_Toc499558533)

[2.2 算法思路 - 5 -](#_Toc499558534)

[2.2.1 概述 - 5 -](#_Toc499558535)

[2.2.2 遍历路径树 - 6 -](#_Toc499558536)

[2.2.3 重复排除 - 7 -](#_Toc499558537)

[2.3 算法特点 - 8 -](#_Toc499558538)

[三、 详细设计 - 8 -](#_Toc499558539)

[3.1 架构设计 - 8 -](#_Toc499558540)

[3.1.1 Web前端 - 8 -](#_Toc499558541)

[3.1.2 后端与flask - 9 -](#_Toc499558542)

[3.1.3 算法模块 - 10 -](#_Toc499558543)

[3.2 算法实现 - 11 -](#_Toc499558544)

[3.2.1 Floyd矩阵与距离更新 - 11 -](#_Toc499558545)

[3.2.2 路径树的生成与选路策略 - 12 -](#_Toc499558546)

[3.2.3 路径输出 - 14 -](#_Toc499558547)

[3.3 算法分析 - 15 -](#_Toc499558548)

[3.3.1 时间复杂度 - 15 -](#_Toc499558549)

[3.3.2 空间复杂度 - 15 -](#_Toc499558550)

[3.3.3 可行性分析 - 15 -](#_Toc499558551)

[四、 测试 - 15 -](#_Toc499558552)

[4.1 模块测试 - 15 -](#_Toc499558553)

[4.1.1 前端测试 - 15 -](#_Toc499558554)

[4.1.2 算法模块测试 - 15 -](#_Toc499558555)

[4.2 整体测试 - 16 -](#_Toc499558556)

[五、 总结与提高 - 16 -](#_Toc499558557)

[5.1 总结与提高 - 16 -](#_Toc499558558)

[5.2 写给自己的话 - 16 -](#_Toc499558559)

[六、 附录 - 17 -](#_Toc499558560)

[6.1 算法API手册 - 17 -](#_Toc499558561)

[6.2 服务器API手册 - 17 -](#_Toc499558562)

[6.3 web交互手册 - 17 -](#_Toc499558563)

1. 需求分析
   1. 概述

本次数据结构课程设计的目标是完成一个校园导游系统。假定学校有十个场所，而任意两个场所之间的距离各不相同。系统可以在用户选定的场所中规划出最优的游览路线，并且对于其中的一些路径点能够考虑用户对时间的要求，最终回到起点。而这个规划的路线应该尽可能得短，同时还要满足用户对于时间上的要求。系统在给出路径规划的同时，应该给出路径的时间消耗信息和路径经过的具体点。

对于这个问题我们可以使用数据结构中与图相关的算法来求解得到。由于涉及到不同类型的策略，所以我们不能完全“照搬”图中求解单源最短路径的方法。在这次我们解决实际问题的过程当中混合了不同的求解方法，而这些子算法中都是运用数据结构的具体知识实现的。

* 1. 应用场景

首先我们的目标用户是参观大学的普通人群，因此我们在准备验证用的终端程序的同时，还要向用户提供界面友好的接口，来提供可视化的结果，给用户更直观的反馈。

由于现在移动设备的增加，如果继续使用客户端/服务器架构来提供服务，会增加用户的使用成本。对于地图线路规划的这种轻量级服务，下载客户端会成为用户使用服务的累赘。为此我们选择了浏览器/服务器的架构来提供服务。

浏览器/服务器架构可以支持多种操作系统，可以在几近所有的设备上运行。由于互联网服务在快速发展，通讯成本在迅速下降。这样使得将服务部署到互联网与部署到本地存储几乎无异。同时，浏览器/服务器结构可以很好地保护知识产权。所有的算法核心代码全部都在远程的服务器中，用户无需存储任何核心算法代码。因此也就没有任何被侵权的可能性了。

* 1. 算法模型

我们在课程设计中使用的核心算法是Floyd算法。

对比Dijkstra算法，Floyd的时间复杂度也许并没有很大优势。然而由于我们需要随机从图中取出最短路径，而Floyd恰好在一次更新邻接关系和距离以后能够得到所有点之间两两的关系。因此比每次都要单独求解的Dijkstra算法更加方便。在地图不变化的前提条件下，甚至可以直接存储更新以后的Floyd邻接矩阵和距离矩阵，大大减少了计算量，将O(n3)的时间复杂度直接下降至O(n)[[1]](#footnote-1)。

我们在算法中使用矩阵存储图。因为Floyd算法与矩阵更加友好，而且矩阵更加便于存取。矩阵中分两种顶点，一种是可见顶点，这种顶点是可以被用户选中作为路径点的；而另一种是不可见顶点，这种顶点比较特殊：不可见顶点对于用户是不可见的，而对于整个规划系统是可见的。这些顶点一般都是多个顶点的中转点，类似十字路口等的顶点。我们在将图抽象成邻接矩阵的时候，会将邻接矩阵一并计算在内，这样可以方便在添加顶点后的矩阵设计，也可以更规整地设计整个抽象地图与实际地图的映射关系。

由于在需求中规定需要考虑部分顶点的时间限制要求，那么我们就需要考虑在规定的时间中我们是否能够完成我们规划的路径。考虑到给用户提供最大的自由度，我们选择遍历所有可能的路径点来选择最优的路径解。我们如此抽象问题：假设用户给定的具有时间限制的顶点都是按照时间顺序排列好的。而这些顶点中带有的时间限制仅仅是单纯的从上一个出发点到这个顶点过程中能够消耗的最大时间。我们就针对这个顶点构建一棵树。如图 1所示。



图 1 遍历路径树

我们将这种图的结构称为遍历路径树[[2]](#footnote-2)。树的起点是上一个出发顶点，所有叶子结点是所有可能的到达顶点，也就是所有的叶子顶点的下一个路径点都是带有时间要求的这个节点本身。不考虑剪枝情况且将目的顶点考虑在内的遍历路径树，如图 2所示。



图 2 完全遍历路径树

其中顶点0是上一个出发点，顶点5是具有时间要求的顶点。中间的顶点都是可能的路径点。当然我们在构建这样一棵伪树的时候，是必须根据路径长度进行剪枝的。

选择遍历的方法是因为我们可以从中获益：用户可以选择自己不同时间关键点[[3]](#footnote-3)之间需要途径的节点个数。从遍历路径树的角度而言，就是可以通过限制树高就可以完成这个功能。同时我们在对完全遍历路径树[[4]](#footnote-4)剪枝的过程中就完成了对时间限制因素的考量。同时我们可以对路径树中的每层节点进行升序排序，并且按照最左优先的原则选择路径，就可以得到我们期望的最短（且最深）的路径。

这一方法最大的特点就是灵活，并且可以根据用户的不同需求求解不同策略下的最优路径解。

* 1. 架构思路

本次课程设计采用了浏览器/服务器的架构设计。因此我们需要将整个应用划分为前端服务页面、后端服务应用和算法服务三个部分。

前端服务页面主要由动态网页框架与网页脚本两部分组成。动态网页框架主要定义服务页面的布局与外观，用于给用户一个良好的可视化界面。网页框架主要将内容分为三层，分别用于显示内容，更新内容和监听用户行为来作出反应。网页脚本主要负责对已经收到的用户行为做出相应，检查和组装数据，并把合法数据发送给服务器。

后端服务应用由软路由和中间件组成。我们通过软路由将网页分为多个独立可服务的页面。用过对于每个可路由的地址我们都编写相应的服务处理程序来响应所有合法的请求。中间件的主要功能是将请求的信息转化为能够被算法服务解析的数据结构，例如数组等。

算法服务时整个应用的核心。我们主要将算法应用根据功能划分为了三个部分。

首先是单源最短路径求解模块。这个模块主要负责使用Floyd算法更新并存储链接状态矩阵组[[5]](#footnote-5)和计算单源最短路径。这个模块是我们整个算法服务的核心。我们之后的模块都是以此为基础的。

在这之上的是多路求解模块，该模块负责接收两组数据，分别是包含普通路径点的无序集合[[6]](#footnote-6)N和包含时间关键点有序序列K。我们的算法主要是对于给定的N与K，规划一条最优路径规划。模块最终的输出是一组由二元组构成的路径序列。其中每一个二元组都是对一条边的描述。

第三个模块就是路径树处理模块。该模块负责对遍历路径树进行构建、剪枝、以及选路的操作。我们的路径树是由链式存储的，因此对于每一次多路求解的迭代，都需要动态地构建并对路径树剪枝。该模块输入是路径树的起点、终点以及一个中继顶点集合，而这些中继顶点都是由普通路径点构成的。

以上就是本次课程设计的架构设计思路，具体每个模块的设计细节会在第三章中具体介绍。

1. 数据结构设计
   1. 数据结构模型
      1. 逻辑模型

在此次数据结构课程设计中，我们一共运用了以下数据结构：队列、n叉树和图。

首先队列是我们在程序设计当中最常用的数据结构。由于我们的算法中涉及到迭代的算法，队列是我们判定迭代结束的重要条件。我们利用队列先进先出的特性，对于固定顺序的时间关键点分别求解最短路径。

N叉树用于我们构建遍历路径树的过程中。我们使用一棵N叉树来存储遍历路径树。对于一个长度为k的普通路径点集合，路径树第一层的树枝顶点数为k个，因为遍历路径树会穷举所有的可能性。对于路径树的第二层树枝节点，我们会安排k-1个顶点。以此类推，当我们完成遍历路径树的穷举时，一共会生成k！条路径。当然这是没有剪枝的情况，通过剪枝，我们会得到更加精炼的结果。

由于我们的数据规模较小，因此在面对k!条路径并不是很困难，因此我们就选择了遍历路径树来解决这个问题。因为路径树可以给我们更灵活的选择，并且在这种数据规模下并不会影响算法的效率。但是如果数据量扩大十倍，也许我们就需要更加高效率的算法来实现选路的问题了。

接下来是图结构。由于本次数据结构课程设计的主要操作对象是地图，因此图作为基本的载体，是我们主要操作的对象。对于一张图G，拥有边集E与点集V，记作G<V,E>。

这就是我们使用的数据结构逻辑模型的简单介绍。

* + 1. 存储模型

我们使用数组来存储队列。在使用数组存储队列的时候需要注意一点，就是数组是一个固定长度的结构，如果我们向数组里面插入的元素超出了缓冲区的长度，我们就需要更新数组的容器，也就是创建一个更大的数组，并将数组的内容拷贝过去，更新缓冲区长度的值，并释放掉原来的数组。这样我们就完成了一个变长的数组队列。

我们使用链式存储树。我们使用了链式方法存储树，也就是说树的子节点都是用指针存储的。如果想要访问树的子节点，仅需要访问指针指向的结构体就可以完成。链式存储有几点好处，我们可以更加动态地创建树的结构，并且更加形象地完成剪枝的动作。

我们使用矩阵存储图。由于我们使用了Floyd算法作为我们的单源最短路径求解方法，所以最好的选择就是使用矩阵存储图。矩阵存储图访问速度更快一些，并且会摆脱指针的痛苦。我们知道图有三种存储方法，邻接矩阵是一种，剩下的两种[[7]](#footnote-7)都需要维护指针对应的表结构。这样可以让我们随机读取边的信息，可以有效提高我们算法执行的性能。

* 1. 算法思路
     1. 概述

在课程设计中我们主要是围绕这如何将时间关键点纳入到路径规划算法当中。我们会阐述之前的一些设计思路，并详细解释为什么选择现有的算法。

首先要考虑时间关键点的问题，就要考虑时间关键点为什么出现。经过仔细阅读课程设计说明，并向老师请教核实以后，我认为，我们要规划的路径要在每对时间关键点之间插入任意数量的路径点。然而选择多少点插入，并且选择那些点插入都是需要算法给出的。对于这样的多自由度的问题，我们就需要制定策略来规划具体路径了。由于对于这样的问题求解全局最优解也许会比较繁琐，考虑到运筹学中的部分知识，我们从局部最优解考虑此类问题或许是一个比较好的办法。

于是我们想到了直接向规划好的路径进行插入的方法。然而这种方法很难预期下一步得到的新路径的消耗是否能够满足时间关键点的要求。因此其实也相当于在遍历整个路径的可能性。而且这样的方法很难有条理地理解路径的走向。我们需要一个在生成路径的同时完成插入路径点的算法。因此我们引入了遍历路径树来解决这个问题。

我们将问题抽象成两个层次的子问题：宏观的关键点层次[[8]](#footnote-8)的路径求解问题和低层次的图中顶点的路径求解问题。由于我们已经将图中任意两个顶点之间的链接状态进行了求解，因此我们可以对任意的两点快速得到最短路径，甚至这些路径可能由多个顶点构成。于是我们可以先完成在关键点层次的路径求解，通过求解出宏观路径而得到低层次的也就是具体在图中的路径，如图 3所示。



图 3 左图是关键点层次路径规划，右图是顶点层次路径规划

然而我们在抽象问题的过程中发现了会出现在规划完关键点层次的宏观路径以后，再重新求得实际路径的时候，会出现关键点在得到的实际路径中重复出现中出现。这样就浪费了我们的路程消耗，不过我们最终解决了这个问题。详细设计请参照2.2.4。

* + 1. 遍历路径树

正如我们在1.4所讨论的，遍历路径树是我们选择时间关键点之间路径点策略的工具。遍历路径树可以灵活地把我们的策略与具体的数据结构联结起来。这赋予了算法其他方法不可比拟的灵活性，在部署的时候可以根据需求灵活地更改策略，也可以让用户自定义策略，完成不同目标的路径规划任务。我们认为遍历路径树的主要操作涉及构建、剪枝、以及排序。

首先我们来抽象地描述一下我们的规划算法：

算法 1：假设由起始时间关键点S，目的时间关键点D，以及由普通路径点组成的集合G={N1, N2, N3,… …, Nk} 。在时间关键点S与时间关键点D之间，有时间要求L。我们的是找出在满足时间要求L所有可能的P⊂G，也就是所有满足时间要求的G的子集。之后从子集中选出符合目前策略的集合Pr 。最终我们使用路径函数H(S, D, Pr)将整个集合转化为路径R。这样得到的路径R就是局部最优解。对于处于时间关键点中所有的有序对<S, D>求解路径R，就得到了该问题的一个全局解。

其中对于路径树的生成方法，已经给出了一个比较清晰的描述：遍历路径树的目标就是穷举在所有可能的P⊂G，也就是G的子集[[9]](#footnote-9)。如果我们能够构建这样一个结构，就相当于把所有的解都进行了遍历，就是定义了我们求解的解空间的范围。

细心的你可能会发现，我们这里构建所谓的“遍历路径树”并不是一棵树，而是个严格意义上的图。在这里我们需要重新说明一下，虽然我们将遍历路径树称为树，然而它其实不是真正的树。我们在构建的是一个非平面图，但是它并不平凡，是因为如果我们抛去目的时间关键点D不看，那么整个我们构建的图的结构就是一棵树。如图 4所示。



图 4遍历路径树的立体表示

这样的一个结构为什么能够遍历所有的G的子集呢？思考这样一个问题，如果我们将元素的所有有序排列全部找到了，那么对于每一个单独的元素来说，它之前（包含它本身）构成的集合都能构成一个唯一的G的子集。这里可能描述的有些抽象，还可以参考一下图 1帮助理解。

而且我们之所以把这样的结构称作遍历路径树，是因为我们实际上是使用树的结构来实现对这样一个非平面图的存储。由于涉及到具体实现过程，所以在此不再赘述，请您参见3.2.2关于路径树生成的详细设计。

之后我们来详细探讨有关遍历路径树的剪枝方法。我们通过上述生成算法得到的仅仅是一个没有考虑到时间限制的一个简单遍历。在我们的整个解空间里还充斥着不符合条件限制的解。那么我们就需要将解空间进一步缩小，以得到最精简的可行解空间。

换言之，遍历路径树的剪枝也就是在解集P中选取所有满足时间要求的子集PL。如果按照完全的集合操作，那么我们的时间成本会直线上升。这里又体现了使用路径树的一个优点。我们仅仅需要剪短一条边，就可以去掉后续所有的不可行解，如图 5所示。



图 5 遍历路径树的剪枝

我们将从起点到该普通路径点的时间消耗以数值的形式存储，并将其与时间限制做比较。如果时间消耗超出了要求，就将该普通路径点从树中剪下，与此同时，其子节点也将一并被舍弃。例如图 5中红色标出的连接，是在该限制条件下的剪枝结果。

我们在完成了对路径树的剪枝以后，就需要对整个路径树进行排序。这样是方便我们根据策略选择具体的路径。如果我们将所有节点的孩子节点进行升序排序，那么我们就可以使用全局最小的策略选择路径，并根据用户给定的深度限制，得出该策略下的局部最优解。

* + 1. 重复排除

在2.2.1中我们曾经提到了在 从宏观路径提取实际路径的过程中，肯能会重复经过一些点。我们的任务和目标就是尽可能多地去掉重复的路径点，给用户提供一个更经济的路径规划。我们在算法 1的基础上又做了修正，得到了可以排除重复的算法设计。

算法 对于一个确定的迭代步t，我们从有序时间关键点集 F(t) 中挑选前两个时间关键点K1(t)、K2(t)。如果普通路径点集G（t-1）={N1, N2, N3,… …, Nk} 中已经包含上一迭代得到的实际路径S（t-1）中的元素Nd, 则从中剔除并得到普通路径点集G(t)。在时间关键点K1(t)、K2(t)之间，有时间要求L（t）。我们的是找出在满足时间要求L（t）所有可能的P(t)⊂G(t)，也就是所有满足时间要求的G(t)的子集。之后从子集中选出符合目前策略的集合Pr(t) 。同时将Pr(t)中元素从G(t)中剔除。最终我们使用路径函数H(K1(t),K2(t)，Pr(t))将整个集合转化为路径R(t)。这样得到的路径R(t) 就是局部最优解。之后将局部最优解R(t)与上一状态已经连接的实际路径S(t-1)进行连接的到S（t）。将S（t）中与G(t)和F(t)重复的元素从S（t）中剔除。如此迭代，直到G与F全部为空。

这样一以来，我们在每一次算法迭代当中都将单步的实际路径与我们现在的未分配顶点点做比较，如果有重复的顶点，那么我们就将其从未分配顶点中剔除出去。

如此一来，我们就构建好了我们的核心算法。我们算法的主要过程就是基于算法 2的，根据不同应用场景的不同，我们还会对路径树进行其他特殊的修建。不过总体思路上并不会影响核心算法过程。

* 1. 算法特点

首先我们将路径规划的问题从单源最短路径抽象为关键点层级的路径规划问题。也就是说如果我们知道了图中每个节点两两之间的最短路径，那么我们就可以将路径用关键点描述，因为任意两点之间的最短路径已经是固定的了。在这基础上，我们又将问题上升到以时间关键点为分界，分步求解路径的层面。第一，这可以大大简化我们可能生成的路径树的树高；第二，我们也可以方便的根据时间关键点的顺序以及限制分步求解局部最优最后得到全局最优解。因此本算法拥有很高的可读性，也很容易理解。对于后期扩展又很好的兼容性。

其次我们使用了遍历路径树来生成我们的局部最优解。这种结构赋予了本算法很高的灵活性，并且能够根据用户的需求动态地定制所需策略，从而满足不同场景下的需求。这种灵活性体现在两方面：第一，我们可以让用户设置中转节点的限制，也就是用户可以选择中转的最多节点数量，同时也可以不选择；第二，我们可以提供不同的选路策略，根据不同给定的策略公式，我们可以生成多种路径。比如一种策略就是，我们要选择单位中转距离最短的路径，那么我们的策略公式就是总距离/节点个数，那么我们通过对该策略求解最小，就能得到该策略下的最优解。

1. 详细设计
   1. 架构设计
      1. Web前端

Web前端作为与用户直接操作的界面，应该给用户一个更加方便、美观的界面。由于我们在矩阵中操作的图上的点都是抽象的路径点，而用户实际操作的路径点应该是显示在地图上的路径点。那么我们前端的作用主要就是建立从抽象路径点到具有实际语义路径点的联系，并向后端发送数据，进行后一步的处理。

我们可以将网页样式按照网页内容分为三层，分别为option层、map\_mask层和base层，如图 6所示。



图 6 样式设计

其中option是选项层。由于在最上层，因此这一层用来处理用户的操作。map\_mask层用来使用SVG[[10]](#footnote-10)在地图上渲染路径和路径点。最底层的base层是用来显示地图数据以及其他与交互无关的信息内容。

在option层，我们设计了一个能够与用户交互的选项窗口。用户在右键点击网页上路径点的时候可以呼出该窗口。在这个窗口上，用户可以对选择的路径点进行参数的设定。选项窗口如图 7所示。



图 7 选项窗口设计（颜色仅供参考）

之后，我们在侧栏设计了一个输入内容显示框。如图 8所示。



图 8 输入内容显示边栏（颜色仅供参考）

以上就是我们所有关于前端设计的具体内容。

* + 1. 后端与flask

我们使用了flask[[11]](#footnote-11)作为我们的后端实现框架。后端主要负责处理HTTP请求，对于请求的虚拟路径进行应用服务的路由与相应处理。我们的服务内容大致分为三部分。

首先，输入服务负责收集用户的输入。用户在/index页面的所有有效输入都会被记录到我们的路径点队列中。其次是输出路径服务。该服务负责将计算好的路径格式化后动态的插入到设计好的网页模板中，并将页面返回给用户以显示路径。第三部分就是错误处理服务。该服务用于处理各种非法路由，或者是非法输入等错误信息，并将错误信息与提示显示给用户。如图 9所示



图 9 服务后端结构

* + 1. 算法模块

我们最终选择了算法 2来完成我们的算法模块。我们将算法模块的程序执行过程描述成了流程框图，如图 10所示。



图 10 算法模块流程图

* 1. 算法实现

以下代码使用python实现。

* + 1. Floyd矩阵与距离更新

更新Floyd矩阵内容：

def SGT\_Floyd\_Update(self):

"""

# Smallest Generated Tree

#   Algorithm of Floyd

#   ------------------------------------------------

#   paras:

#       NULL

"""

dist = self.dist\_map.copy()

path = self.neigh\_map.copy()

for j in range(dist.shape[0]):

for i in range(dist.shape[0]):

for k in range(dist.shape[0]):

#   avoid fake shortest path

if dist[i][j] + dist[j][k] < dist[i][k]:

dist[i][k] = dist[i][j] + dist[j][k]

path[i][k] = path[j][k]

return dist, path

回溯最短路径

def SSSP\_Floyd(self, node\_src, node\_dst):

"""

#   Single Source Shortest Path

#   Algorithm of Floyd

#   ------------------------------------------------

#   Feature:

#       Find SSSP with a SGT struct

#   method to get the shortest source for the

#   specific source node of this graph

#   ------------------------------------------------

#   paras:

#       node\_src:   name(num) of the source node

#       node\_dst:   name(num) of the destination node

#   get the full path

"""

node\_cur = node\_dst

temp\_path = []

while node\_cur != node\_src:

temp\_path.insert(0, int(self.path[node\_src][node\_cur]))

node\_cur = int(self.path[node\_src][node\_cur])

temp\_path.append(node\_dst)

print self.path[node\_src][node\_dst]

return self.dist[node\_src][node\_dst], temp\_path

* + 1. 路径树的生成与选路策略

1. 生成路径树

def CreateTree(self, parent, child\_list):

"""

# Tree Creation

# --------------------------------------------

# paras:

# parent: parent node

# child\_list: child node list

# lastleaf: last leaf to be fill

"""

if child\_list == []:

return

return\_child\_list = []

for n in child\_list:

temp\_child\_list = copy.deepcopy(child\_list)

#print parent, "\t", parent.toString()

for k in temp\_child\_list:

if k.data==n.data:

temp\_child\_list.remove(k)

#print temp\_child\_list

p = TreeNode(n.data)

p.parent = parent

parent.AddChild(p)

#print parent.child

self.CreateTree(p, temp\_child\_list)

self.NodeTree = parent

return parent

1. 更新路径树消耗值，方便剪枝

def UpdateCost(self, root, dist\_matrix, num=1):

"""

# update the cost

# --------------------------------------------

# paras:

# node: root node in sub tree

# num: count num start

"""

for i in root.child:

i.cost = root.cost + dist\_matrix[i.data][root.data]

i.depth = num

self.UpdateCost(i, dist\_matrix, num+1)

1. 对路径树进行剪枝

def ReduceTree(self, root, dist\_matrix, cost\_limit, depth, num=1):

"""

# cut the branch which mismatch the limit of cost

# --------------------------------------------

# paras:

# node: root node in sub tree

# num: count num start

"""

for i in root.child[:]:

total\_cost = i.cost + dist\_matrix[self.lastleaf.data][i.data]

print "to ", i.data, " path length is ", total\_cost,

" limit is : ", cost\_limit

if total\_cost > cost\_limit or num >= depth:

root.child.remove(i)

else:

self.ReduceTree(i, dist\_matrix,

cost\_limit, num=num+1, depth=depth)

1. 使用迭代算法遍历路径树得到最短路径

def FindPath(self, node):

"""

# Shortest Path finder

# --------------------------------------------

# paras:

# node: root node in sub tree

"""

stack = []

min\_cost = self.INFINITE

min\_node = node

# Push the Item into a stack

if node.child == []:

min\_node = node

for i in range(0,len(node.child)):

stack.append(node.child[i])

# while stack is not empty

while(stack):

# pop a item

cur\_node = stack[-1]

del stack[-1]

if cur\_node.child==[]:

if cur\_node.cost < min\_cost:

min\_node = cur\_node

min\_cost = min\_node.cost

continue

for i in range(0,len(cur\_node.child)):

stack.append(cur\_node.child[i])

return min\_node

* + 1. 路径输出

1. 输出路径的部分代码

# concenate the path pieces into continuos path sequence

mid\_path = []

for n in macro\_path:

for m in n[1:]:

mid\_path.append(m)

mid\_path.insert(0, macro\_path[0][0])

print "Macro Path is :\t", mid\_path

# get list of single paths

# the point pair which describe the path

micro\_path\_p = []

while len(mid\_path)>1:

point\_src = mid\_path[0]

point\_dst = mid\_path[1]

micro\_path\_p.append(self.singlepath\_list[node\_src.\_type\_].SSSP\_Floyd(point\_src, point\_dst)[1])

del mid\_path[0]

# get micro path

micro\_path = []

for n in micro\_path\_p:

for m in n[1:]:

micro\_path.append(m)

micro\_path.insert(0, micro\_path\_p[0][0])

tag\_path = []

while len(micro\_path)>1:

tag\_temp = [micro\_path[0],micro\_path[1]]

tag\_path.append(tag\_temp)

del micro\_path[0]

* 1. 算法分析
     1. 时间复杂度
     2. 空间复杂度
     3. 可行性分析

1. 测试
   1. 模块测试
      1. 前端测试



* + 1. 算法模块测试





* 1. 整体测试

1. 总结与提高
   1. 总结与提高
   2. 写给自己的话
2. 附录
   1. 算法API手册
   2. 服务器API手册
   3. web交互手册

1. 由于Floyd求解路径时涉及到回朔(back trace)路径，因此查询复杂度与回溯复杂度为O(n+1)=O(n)。 [↑](#footnote-ref-1)
2. 遍历路径树：通过遍历所有可能的顶点形成的图结构，后文会简称为路径树。实际上并不是逻辑上的树结构，我们会在2.2.2以及3.2.2中详细探讨这个问题。 [↑](#footnote-ref-2)
3. 时间关键点：具有时间限制要求的顶点 [↑](#footnote-ref-3)
4. 完全遍历路径树：没有经过时间限制剪枝的遍历路径树 [↑](#footnote-ref-4)
5. 链接状态矩阵组由邻接矩阵和距离矩阵组成 [↑](#footnote-ref-5)
6. 普通路径点是不带有时间限制的顶点 [↑](#footnote-ref-6)
7. 这两种表分别为邻接表以及十字链表。邻接表是将与此顶点相邻的节点标识符以链表形式存储起来。十字链表可以理解为将邻接矩阵以链式存储的结果。两种方法都需要维护指针，尤其是后者。 [↑](#footnote-ref-7)
8. 关键点指的是时间关键点与普通路径点组成的问题所涉及的抽象层次。 [↑](#footnote-ref-8)
9. 这里之所以与上文描述不符，是因为生成算法未包含剪枝算法。为了区分开来，我们在这里分开定义。 [↑](#footnote-ref-9)
10. SVG: Scalable Vector Graphics（可伸缩矢量图形）用于在网页上渲染以xml格式存储的矢量图形或动画 [↑](#footnote-ref-10)
11. flask: http://flask.pocoo.org/ [↑](#footnote-ref-11)