

# 数据结构课程设计

## 算法设计与综合应用

2019年8月31日

姓名：魏宇翔

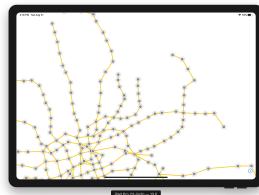
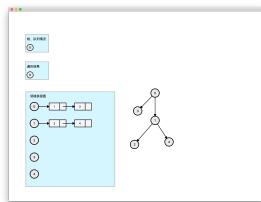
学号：1750222

## 任务综述

在数据结构理论课的基础上，进行复杂、功能全面、有友好交互界面的程序设计，分别完成算法实现题以及综合应用题。

## 内容实现概述

- 使用 C++ Qt 框架完成算法实现题中对相关图算法的图形演示。
- 使用声明式 UI 框架 SwiftUI 完成综合应用题：地铁换乘系统。



## Github

### \* 算法实现

▷ <https://github.com/UniverseFly/Graph-DataStructureCourseDesign>

### \* 综合应用

▷ <https://github.com/UniverseFly/SwiftUI-SubwayTransferSystem>

---

# 目录

数据结构课程设计	1
目录	2
算法实现设计说明	4
题目	4
软件功能	4
用户交互	4
算法功能	5
设计思想	5
Model	5
View	6
Controller	6
逻辑结构与物理结构	7
顶点	7
图	8
开发平台	8
概述	8
运行环境	8
系统的运行结果分析说明	9
开发工具	9
调试	9
版本控制系统	10
正确性	11
稳定性	12
容错能力	13
操作说明	14
综合应用设计说明	15
题目	15
软件功能	15

---

SwiftUI	15
用户交互	17
设计思想	18
数据结构	19
算法设计	19
App 设计	19
逻辑结构与物理结构	19
站点	19
地铁图	20
开发平台	21
概述	21
运行环境	21
系统运行结果分析说明	22
开发工具	22
数据获取	23
正确性	23
稳定性	25
容错能力	25
版本控制系统	26
操作说明	26
实践总结	27
所做的工作	27
总结与收获	27
参考资料	28

# 算法实现设计说明

## 题目

用邻接矩阵的方式确定一个图，完成：

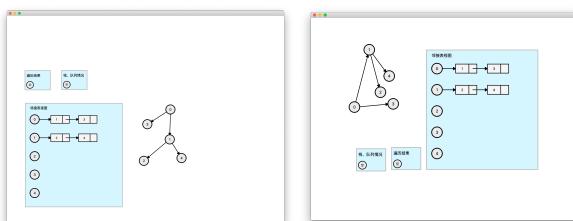
1. 建立并显示出它的邻接链表；
2. 以递归和非递归的方式进行深度优先遍历，显示遍历结果，并随时显示栈的出、入情况；
3. 对该图进行广度优先遍历，显示遍历结果，并随时显示队列的出、入情况。

## 软件功能

软件功能分为两部分，用户交互与算法功能。用户交互体现了用户使用该软件时能进行的操作；算法功能则体现了该软件的实用价值。用户交互采用 C++ Qt 框架，算法功能采用C++14，使用大量 C++ 特性。

## 用户交互

首先，软件有灵活的界面，画面上的几乎所有显示的元素都可由用户用鼠标拖动，使用户根据需求改变画面。



其次，该软件有菜单项供用户操作，一切操作都在菜单项中，没有更多多余的选项，使用户几乎不需要任何学习成本。

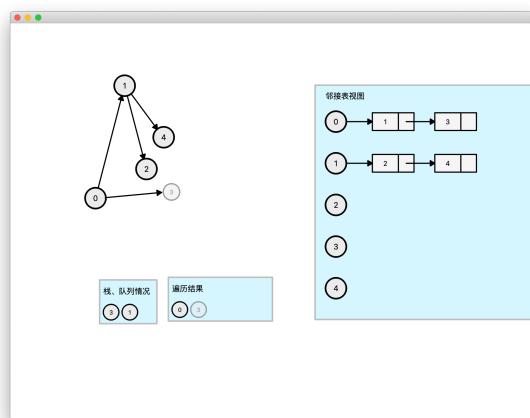


最后，菜单中有丰富的快捷键，使软件的操作简化。（放大与缩小的操作还能使用 Ctrl/Command + 滚轮进行，体验自然。）



## 算法功能

该软件在功能实现上完成了题目要求，实时显示一个有向图对应的邻接链表，并完成了深度（递归与非递归）、广度优先遍历的结果显示，同时显示结果以及栈、队列信息。显示过程中包含动画效果，对所显示的图进行实时渲染。



## 设计思想

该程序的设计大体分为三个块。首先是算法本身的设计实现，包括题目所有要求的内部算法实现，以及内部的数据结构，不包含任何图形界面；其次是图形显示的设计，它代表了图形显示，而不包含任何算法；最后是二者的整合，实现图形与内部数据结构的同步。该设计思路遵循了一种设计模式 (Design Pattern)：MVC (Model View Controller) <sup>[1]</sup>。

## Model

数据结构与算法模型采用 C++ 编写，大量使用 C++ 特性如模板 (template)，auto，范围 for 循环、C++11 引入的智能指针，基本标准库数据结构如 std::vector，高级标准库数据结构如 std::deque、std::unordered\_set。

图的定义，它是个顶点存放 Value 类型值的模板：

```
template<typename Value> struct Graph;
```

代码简单易懂、有详细注释，思路较为清晰，一切 C++ 相关的知识均参考自《C++ Primer》<sup>[2]</sup> 以及官网<sup>[3]</sup>。而算法的设计参考自《算法导论》<sup>[4]</sup>。整个模型分为两个文件：

Graph.h: 图的数据结构,

GraphAlgorithms.h: 深度优先遍历等图算法。

图的数据结构中包含了所有的图操作, 例如添加顶点等操作。而图算法通过图操作返回遍历结果、栈或队列的每次更新后的结果, 供视图显示。结果类型为 SearchResult, 是自定义类型, 能够表达所有视图所需要的信息:

```
struct SearchResult {  
    // 记录了某种遍历时的下标顺序。  
    std::vector<int> indexOrder;  
    // 记录了栈、队列的实时信息。  
    std::vector<std::deque<int>> containerCondition;  
};
```

通过下标顺序以及栈、队列的实时信息, 返回给视图层进行渲染就十分容易。

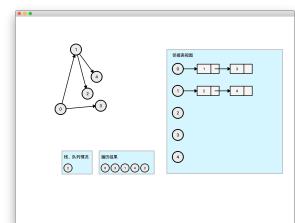
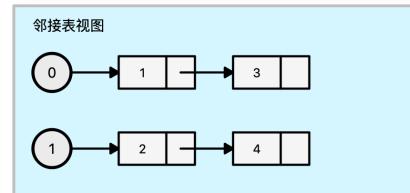
## View

视图层使用 Qt 作为开发框架, 主要运用了 Qt 的 Graphics View Framework<sup>[5]</sup>, 该框架能够做到对组件的组合。

程序中, 图形显示的思路就是组合。邻接表是由上到下关于每个顶点的链接显示, 这样就可以将整个视图作为许多个顶点的链的组合, 达到了从小到大搭建视图的效果。

同样对于栈以及队列, 可以看成许多个顶点的组合, 并添加外框。

整个程序采用组合思路的好处在于组件的重复利用, 该程序中的顶点在 4 个模块中达到复用: 图的显示、邻接表的显示、遍历结果、栈或队列的实时更新。如此一来, 大的图形就可以拆分为一个个小的部分, 降低思维与编写代码的负担。



## Controller

控制器是协调视图和模型的枢纽, 有了控制器即能达到视图与模型的分离和同步。它的主要任务有两点。首先是使用 Qt 中的类对模型进行再封装, 组织接口提供给视图层显示。其次是对模型与视图的同步, 例如用户选择清空顶点后, 控制器要负责将视图层的顶点清除, 同时将模型层的顶点清除。

```

void clearGraph() {
    if (isAnimating) {
        QMessageBox::warning(this, "操作失败", "请等待动画结束后操作");
        return;
    }
    model.reset();
    graphObject.reset();
    adjointListGraph.resetFromRaw({});
}

```

当用户点击清空顶点时上述代码触发。代码中进行了 3 处 reset, model.reset() 将模型中的顶点全部删除, graphObject.reset() 将显示图中的顶点删除, 更新视图, 最后一行代码则清空邻接表视图。如此达到模型与视图同步。

添加顶点	⌘V
添加弧	⌘A
清空顶点	⌘R

## 逻辑结构与物理结构

下面将讨论 Model 中图模型的数据结构, 而关于 View 和 Controller 的结果表示可参考源代码。

### 顶点

```

template<typename Value>
struct Graph<Value>::Vertex {
private:
    friend struct Graph<Value>;

    Value value;
    // 邻接表, 不会产生循环引用, 用智能指针自动销毁,
    // `std::unordered_set<int>` 代表顶点在图中所处的下标而非值
    std::shared_ptr<std::unordered_set<int>> adjointList;

    // ...
}

```

首先, 每个顶点包含了它所需存储的值, 这里将它定义为模板类型 Value。同时, 由于整个图采用了邻接矩阵的方式来存储, 每个顶点都会有衍生出去的邻接表, 在这里将邻接表定义为指向无序集合的智能指针。

智能指针的作用是根据引用计数自动销毁堆中创建的变量<sup>[2]</sup>, 避免内存泄漏。无序集合采用了散列存储的方式使元素唯一且插入删除达到  $O(1)$ 。(可参

考 cppreference 中对 `std::unordered_set` 的介绍<sup>[2]</sup> 该集合的作用是存储所邻接顶点在图结构中顶点的下标。 (见下文)

## 图

```
/// Value should be hashable
template<typename Value>
struct Graph {
private:
    // `Graph<Value>::Vertex`, 定义在下方。
    struct Vertex;
    std::vector<Vertex> vertices;
    // 每个顶点值到其所在下标的映射。
    std::unordered_map<Value, int> valueToIndex;

    // ...
}
```

图是一个模板类型，它可以用任何类型 `hashable` 的顶点值来实例化。其中存储了两个部分。首先是顶点的 `vector`<sup>[2]</sup>，是一个顺序存储的线性表，可以动态插入删除元素。其次是一个无序映射，它从给定的顶点值映射到它在 `vector` 中的下标，以优化直接通过值（而非下标）来对图进行操作的情况（该应用的操作全部基于下标）。

通过顶点形成的动态数组以及每个顶点链接出去的邻接表，一个无向图的数据结构可以完整表示。

## 开发平台

### 概述

- ▶ 操作系统: macOS Mojave 10.14.6
- ▶ 编程语言: C++14
- ▶ 开发框架: Qt 5.13.0
- ▶ 编译器: clang-1001.0.46.4
- ▶ IDE: CLion 2019.2 (Build with CMake)

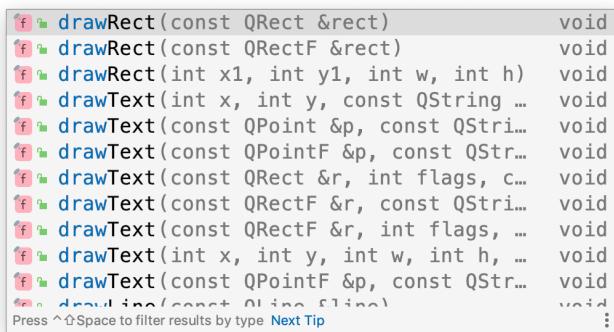
### 运行环境

上述环境可以直接运行可执行文件。若非上述环境则或需重新编译整个项目，需要装有 Qt 环境以及支持 C++14 且能够编译 Qt 库的编译器如 MinGW、Clang。

## 系统的运行结果分析说明

### 开发工具

起初我使用 Qt Creator 作为开发工具，它的好处在于环境已配置，打开即可使用，且整合了 Qt Designer 方便制作界面。然而它的代码提示以及界面友好度都大不如 JetBrains 开发的 CLion。CLion 的代码提示功能使我可以根据提示猜测代码功能，在开发过程中大大减少了阅读文档的频率。



配置 CLion 也只需几行 CMake 代码连接 Qt 类库，十分方便。唯一的不足在于对外部工具 Qt Designer 等的使用会较为麻烦，但总体体验十分良好。

### 调试

调试手段有两种，一种是通过设置断点查看变量信息，第二种是通过 `std::cout`、 `qDebug` 语句在 CLion 集成的控制台输出结果。



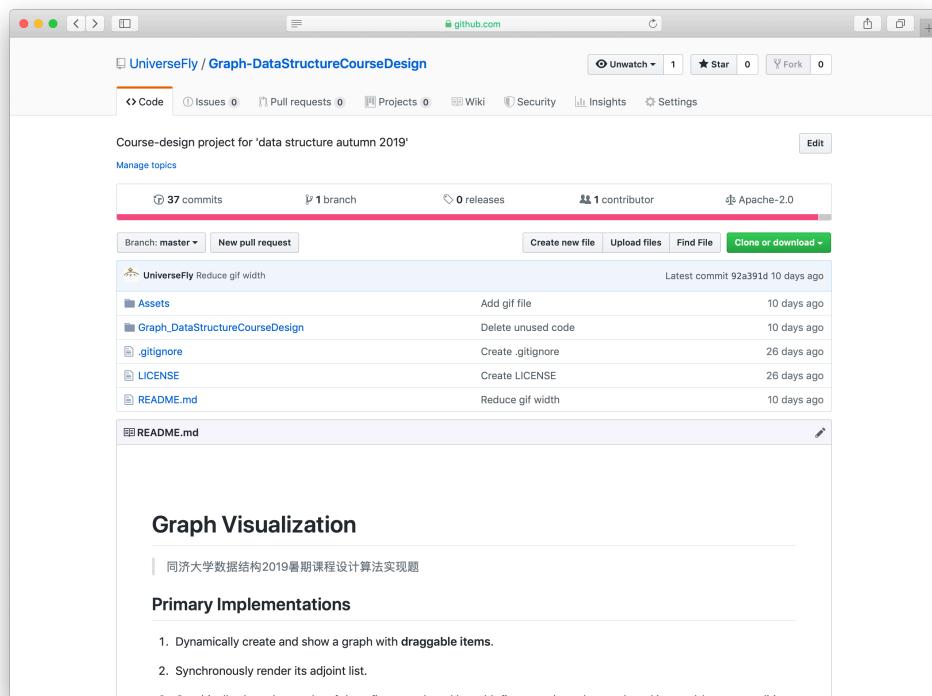
## 版本控制系统

本次开发中使用了版本控制系统 Git 来保存代码的历史记录。同时使用代码托管平台 GitHub 将整个 Git 仓库存储到该网络平台中，使代码“有迹可循”。

Git 的使用有两方面，最传统的方式是通过 Mac 自带的 Terminal，同时 CLion 也集成了 Git 和控制台可以方便使用。

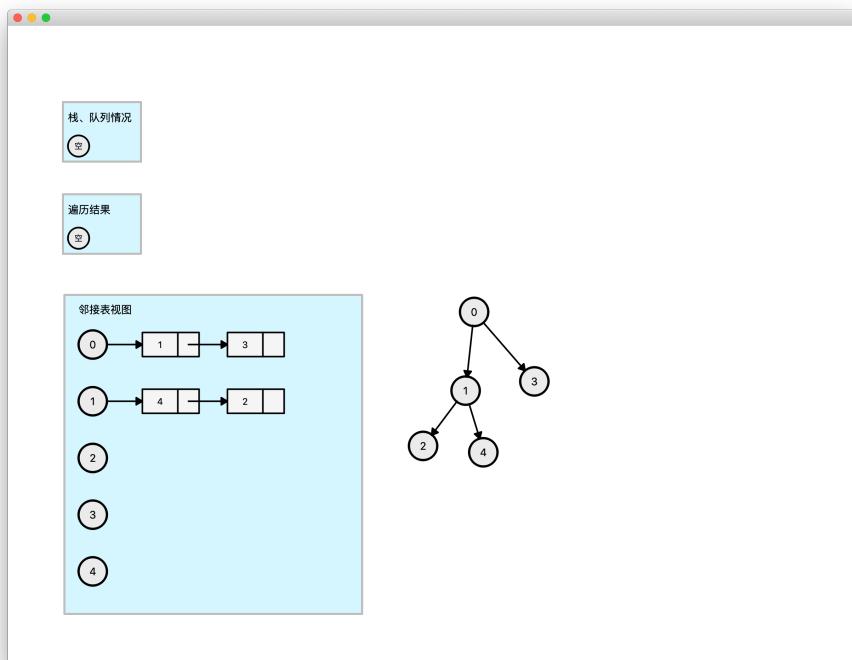


Github地址在首页已有说明，也可以点击此处前往。

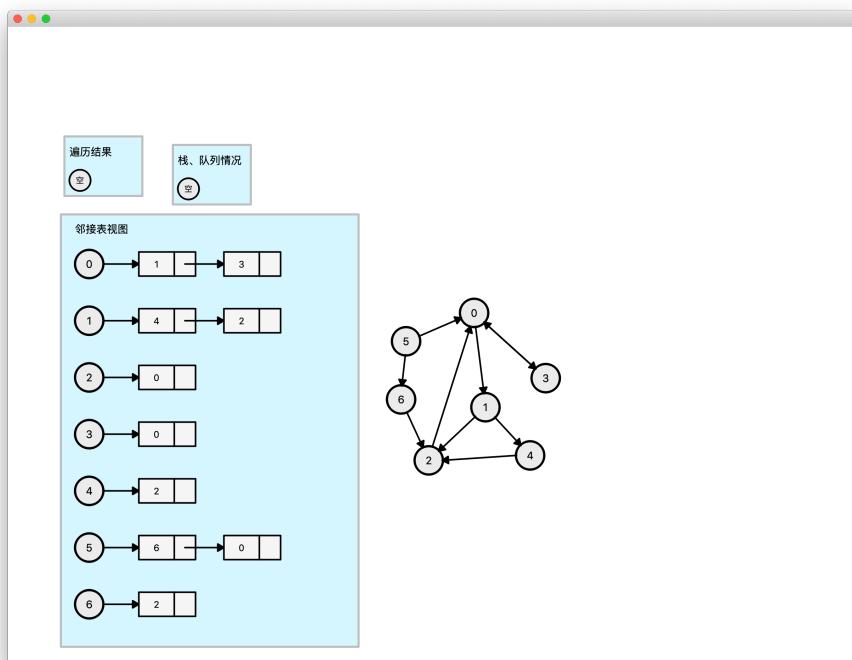


## 正确性

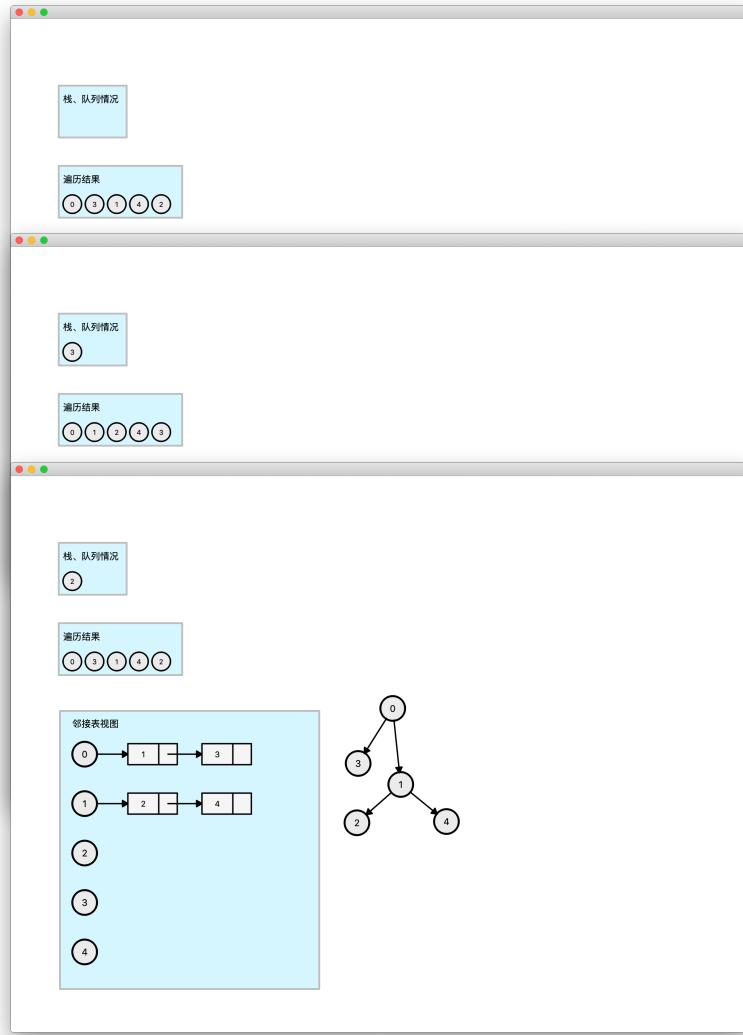
首先是邻接表的显示，本项目中的图均是有向图，下述案例中邻接表显示正确，0号顶点邻接1和3，1号顶点邻接2和4，没有更多的邻接情况。



补充案例：



在上述第一个案例的基础上对从 0 号顶点开始分别进行深度优先遍历递归、深度优先遍历非递归、广度优先遍历，得到如下结果（栈队列情况实时更新因此需要实际演示）。



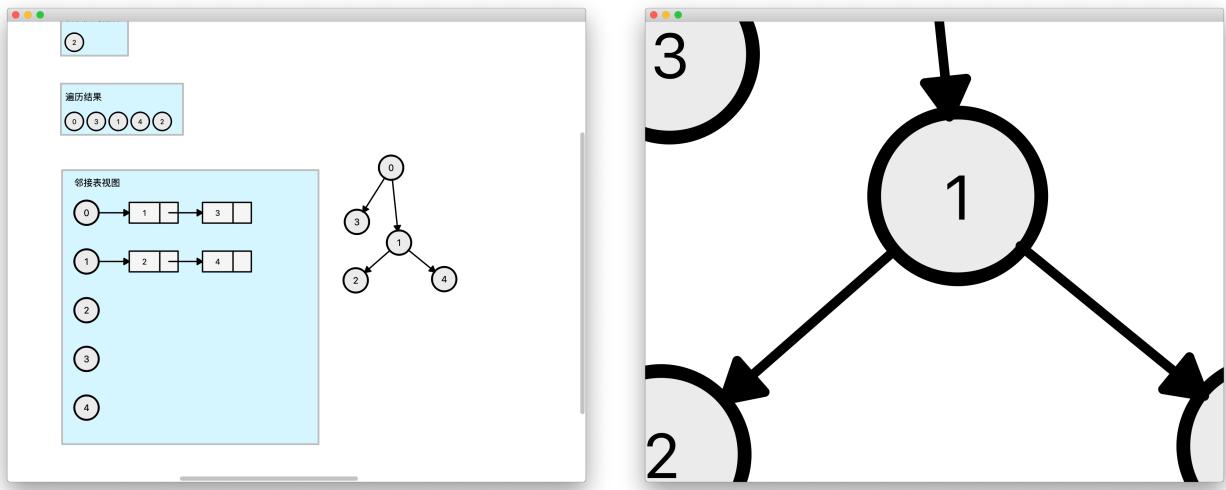
这三组结果是合理的。之所以深度优先搜索递归和非递归产生了不同结果是因为邻接顶点采用的数据结构是 `std::unordered_set`，所以一个顶点的邻接顶点集合是无序的，这和深度优先并不矛盾。

## 稳定性

图形界面的大小可以有限度的调整，可以通过触控版、鼠标滚轮进行移动。同时整个界面可以放大缩小，灵活适应用户需求。

放大   
 缩小

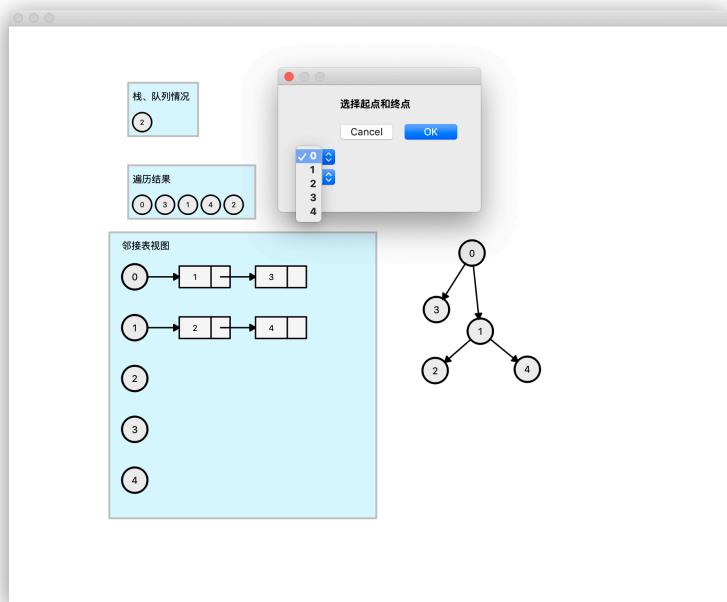
同时，在部件的拖拽，窗口大小改变的过程中，程序保持良好流畅度，且测试中暂未发生过应用崩溃现象。动画保持流畅。



## 容错能力

在产生一些异常用户输入时，程序会阻止该输入。

首先，例如在用户选择添加弧时，程序从选择的角度阻止了异常输入，不允许超出范围的顶点选择。



其次，当用户添加了已经添加的弧或添加了自己到自己的顶点时会弹出提示信息。

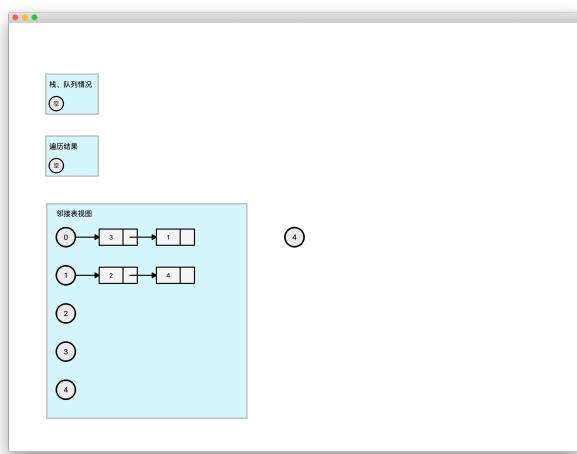


当用户清空顶点后，无法再继续添加弧、进行遍历。

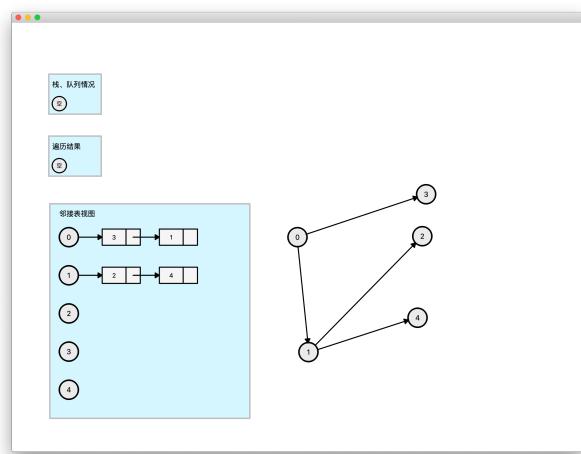


## 操作说明

本项目的操作均集成在菜单上，上手十分简单。值得注意的是，进入程序后会预先加载已经录入的样例图，这些顶点是重合在一起的，需要通过鼠标拖动将其分开，顶点可拖动这一特点也卸载了帮助菜单中。



刚进入界面



拖动顶点后



# 综合应用设计说明

## 题目

上海的地铁交通网络已经基本成型，建成的地铁十多条，站点上百个，现需建立一个换乘指南打印系统，通过输入起点站和终点站，打印出地铁换乘指南，指南内容包括起点站、换乘站、终点站。

1. 图形化显示地铁网络结构，能动态添加地铁线路和地铁站点。
2. 根据输入起点站和终点站，显示地铁换乘指南。
3. 通过图形界面显示乘车路径。

## 软件功能

在综合应用题中，编程语言以及开发框架都与算法实现题不同。编程语言改为 Swift，而开发框架改为 WWDC 2019<sup>①</sup> 中发布的 SwiftUI。

在算法实现题中已经基本实现了图的数据结构以及各种操作，在这一经验的基础上第二题在算法功能的实现上只需将代码进行编程语言上的迁移，Swift 是一个极具表达性、现代化、语法优美的静态类型语言，因此迁移工作很简单，代码量也相应减少。

在迁移代码的基础上，还增加了对最短路径算法的编写，这一算法返回一系列坐标对来表示一条真实的最短路径。

整个程序的重点，就是 SwiftUI。

## SwiftUI

SwiftUI 是一种声明式的 UI 框架，声明式使得代码与界面达到了同步，举个例子如下，上面是代码（很简单），下面是界面。

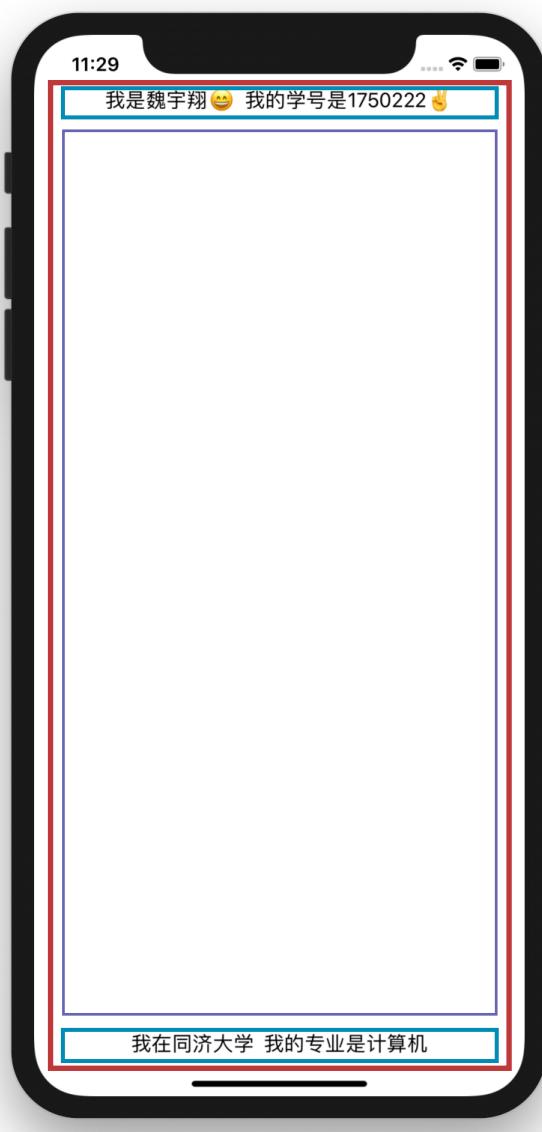
VStack 中的 V 代表 Vertical，即垂直。

HStack 中的 H 代表 Horizontal，即水平。

Spacer 则代表一个很大的空白区域。

下面的代码分别将两组文字水平显示，再将他们组织到垂直区域，其中包含一个 Spacer（相应颜色的代码与相应颜色的框选区域吻合，框选区域为手动添加为了演示效果）。

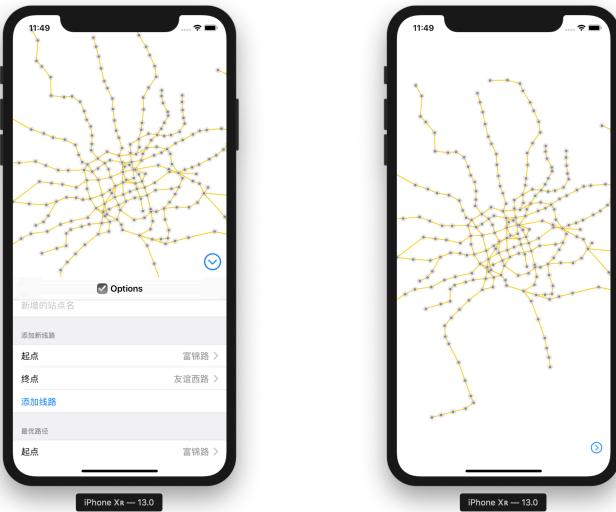
```
struct ContentView: View {
    var body: some View {
        VStack {
            HStack {
                Text("我是魏宇翔😊")
                Text("我的学号是1750222✌️")
            }
            Spacer()
            HStack {
                Text("我在同济大学")
                Text("我的专业是计算机")
            }
        }
    }
}
```



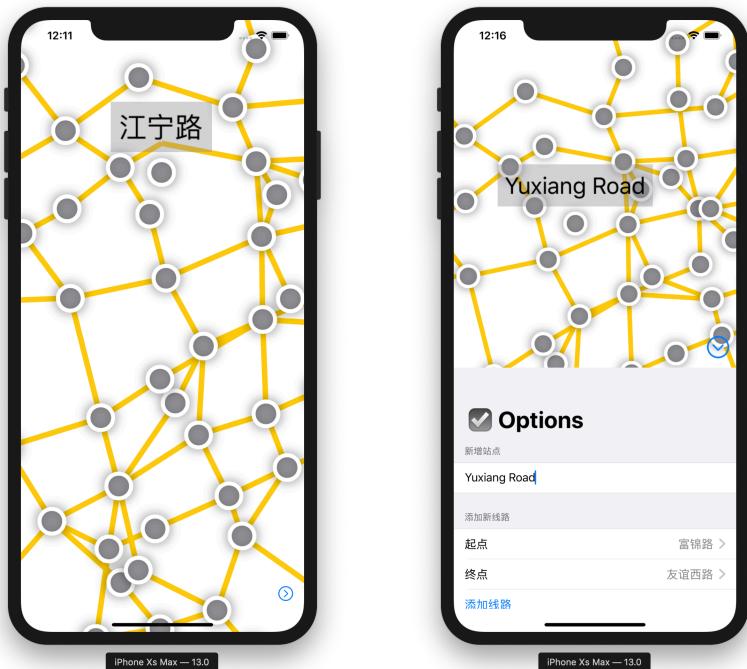
iPhone XR — 13.0

## 用户交互

首先，该 app 将所有的操作（除了通过手势的一些操作）都集成在了一个悬浮菜单栏中，用户可以自行摸索，简单易懂。悬浮菜单栏可以自由显示、隐藏，这使屏幕利用率大大增高。

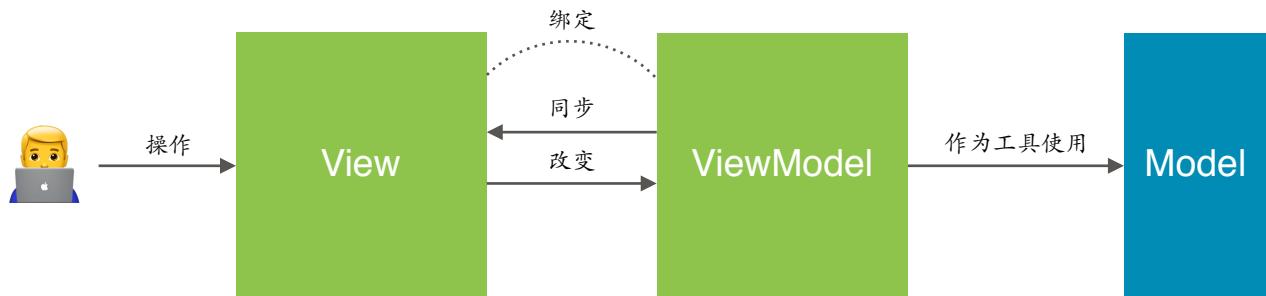


由于手机的特殊性，手势操作对用户交互的体验提升很大。该 app 包含了地图二指捏合缩放手势，单手拖拽移动地图手势。同时，长按一个顶点会显示或隐藏该顶点代表的站点名。将地图稍微放大，手指从任意站点开始拖拽，拖拽的目的地就是新增的站点目的地，而新增的站点名在菜单栏中。



## 设计思想

该 app 的设计思路与算法实现题的设计思路有所差别，并不采用 MVC 架构，而采用的是 MVVM (Model View ViewModel) 设计模式。它的数据通信方式如下所示。



其中 Model 是整个数据模型，作为工具。ViewModel 则是整个视图的抽象表示，它代表了整个视图，同时通过绑定将其绑定到视图中的组件中。用户直接操作视图，视图的改变自动改变绑定到自己的 ViewModel 中的相应数据，ViewModel 的改变则继续对 View 进行同步。

SwiftUI 作为声明式 UI 提供了一切 MVVM 所需要的工具如 `@Binding`、`@State`、`@ObservedObject` 等 attributes。通过声明式语法快速构建 View，并将 ViewModel 通过这些 attributes 绑定到 View 中，一个 app 就完成了。为了更好的阐述这一理念，下面举个例子。

```
struct ContentView: View {
    @State var toggleIsOn = false

    var body: some View {
        VStack {
            Toggle(isOn: $toggleIsOn) { Text("显示文字") }
                .padding()
            if toggleIsOn { Text("我真难。。计算机好累") }
        }
    }
}
```



---

这段代码的 ViewModel 就是 `@State var toggleIsOn: Bool` 这个布尔变量，它没有 Model。这个变量通过 Toggle 中设计的某个 Binding 绑定到了视图上，视图上切换键（toggle）的改变直接会改变被绑定数据，即 `toggleIsOn` 这个变量。切换键绿色的时候该变量为 `true`，切换键灰色时该变量为 `false`。这个变量的真假直接影响文字是否显示，从而同步给这个 View。这就是 MVVM 架构的数据流。（SwiftUI 中 `@State` 与 `@ObservedObject` 理念相同，但 `@ObservedObject` 更适合用于完整的较大的 ViewModel，而 `@State` 适合于布尔变量等小型数据。）

## 数据结构

与算法实现题相同，仍然采用邻接表。采用邻接表的理由首先是地铁线路属于稀疏图，几乎站点之间除非前后两站不会直连，同时考虑到手机 app 对存储的要求采用了邻接表而非邻接矩阵。

## 算法设计

算法采用了没有优化的 Dijkstra 算法，复杂度为  $O(n^2)$ 。设计思路无他，参考算法导论<sup>44</sup>。

## App 设计

整个 app 的 Model 就是邻接表表示的图；ViewModel 则是使用 Model 并对视图进行抽象。在此主要讨论 View 的设计。

SwiftUI 中的一切东西都是一个 View，更准确的说它们都遵循 View 这个协议，Swift 是个面向协议的编程语言<sup>45</sup>。这些视图均有着高度组合性，自定义视图只需要遵循该协议并实现其中 `var body: some View { get }` 即可。因此整个 app 的设计思路就是不同板块的组合：显示界面、选择菜单、地铁站、地铁线路……

## 逻辑结构与物理结构

与算法设计题相同，此处仅讨论 Model 的数据结构。

## 站点

```

/// Model for a station
struct Station {
    var name: String
    var position: CGPoint

    static func distance(_ lhs: Self, _ rhs: Self) -> Double {
        let deltaX = lhs.position.x - rhs.position.x
        let deltaY = lhs.position.y - rhs.position.y
        return sqrt(Double(deltaX * deltaX + deltaY * deltaY))
    }
}

extension Station: Codable {}

```

一个站点有两个信息，名字和位置，因此它有这两个属性。同时，定义了一个静态方法计算两点间的距离。最后的 `extension` 使站点遵循 `Codable` 协议，方便外部 JSON 数据的导入。

## 地铁图

```

/// Data model for subway graph
struct SubwayGraph: Codable {
    /// 邻接边对应的顶点下标和距离
    struct ArcNode: Hashable, Codable {
        var index: Int
        var distance: Double
    }

    private(set) var vertices = [Station]()
    private(set) var arcs = [Int: Set<ArcNode>]()

    /// 站点名到它所在下标的 map
    private(set) var stationToIndex = [String: Int]()

    /// 起点与终点的坐标对
    struct StationPair: Codable {
        var start: CGPoint
        var destination: CGPoint
    }

    /// 所有线路的起始和终点坐标
    private(set) var subwayLines: [StationPair] = []
}

// ...

```

地铁图的数据结构与算法实现题中图的数据结构相似，不同点在于，首先在邻接顶点的属性中增加了 `distance` 表示距离，其次添加了 `subwayLines` 这个属性来表示所有连接好的边的起始与目的顶点，方便视图显示。

## 开发平台

### 概述

- ▷ 操作系统：macOS Mojave 10.14.6
- ▷ 编程语言：Swift 5.1
- ▷ 开发框架：SwiftUI
- ▷ 编译器：Apple Swift
- ▷ IDE：Xcode Version 11.0 beta 6 (11M392R)

### 运行环境

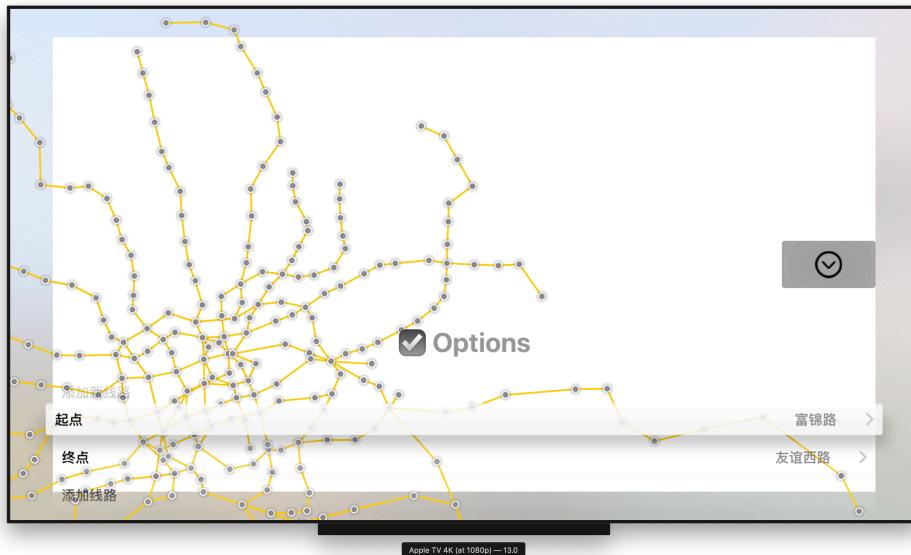
上述环境可以构建并运行该 app，通过 Xcode Simulator。可以模拟出各种 Apple 设备。



其它 macOS 环境下能否直接运行可执行程序至 simulator 未知。

发行版 iOS 13、发行版 Xcode 11 放出后可以直接 build 到 iPhone、iPad 上。若有 iOS 13 beta 版的设备也可以直接使用上述环境 build。

SwiftUI 跨自家平台，因此该 app 稍作修改即可变成 macOS、tvOS、watchOS 的 app，下面是源码未经修改直接添加到新的 tvOS 项目后的结果。因为 SwiftUI 的布局系统适应不同的 OS，因此显示效果正常（但有些功能如拖动手势无法使用）。



## 系统运行结果分析说明

### 开发工具

开发工具采用了 Xcode 的 beta 版本。由于 SwiftUI 的年龄才不到 3 个月，正式投入生产还需要一段时间，因此相关工具链也只有测试版才支持。同时，工具链中包含的 Swift 5.1 版本的编译器也未正式发行。若对 Swift 感兴趣可以参考 [Swift 官网](#)。

总体来说测试版的使用还是很不错的，很少发生崩溃现象。同时 Swift 5.1 的新特性加上 SwiftUI 也使代码能够更加放飞。

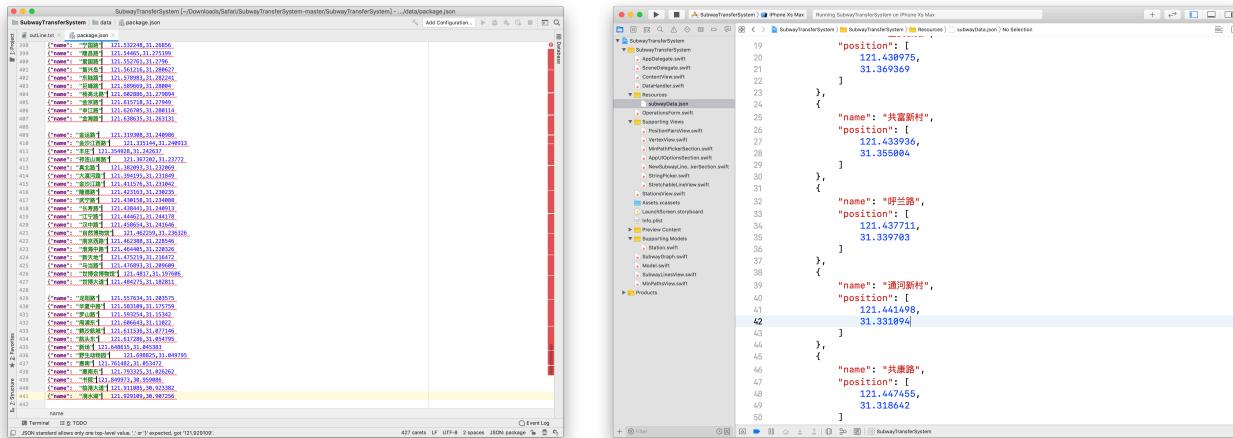
```

6 // Copyright © 2019 Tongji University. All rights reserved.
7 //
8
9 import SwiftUI
10
11 /// 每个地铁站点的节点
12 struct VertexView: View {
13     var name: String
14     @State private var showDetail = false
15
16     var body: some View {
17         VStack(spacing: 5) {
18             if (showDetail) {
19                 Text(name).padding(5).background(Color.gray.opacity(0.4))
20             }
21             Circle()
22                 .foregroundColor(Color.gray)
23                 .overlay(Circle().stroke(Color.white, lineWidth: 2.5))
24                 .shadow(radius: 2.5)
25                 .frame(width: 15, height: 15)
26         }
27         .animation(.easeInOut(duration: 1.3))
28         .transition(.scale)
29         .onLongPressGesture {
30             withAnimation { self.showDetail.toggle() }
31         }
32     }
33 }

```

## 数据获取

整个上海地铁站的数据获取主要通过该链接，经过刻骨铭心的人工操作  
将它转换为 JSON 文件。



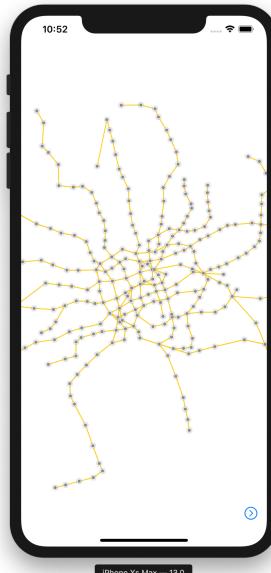
这个 JSON 文件最终的操作用于初始化每条地铁线的所有起点到终点的站点数组。

```
let stationGroups: [[Station]] = load("subwayData.json")
```

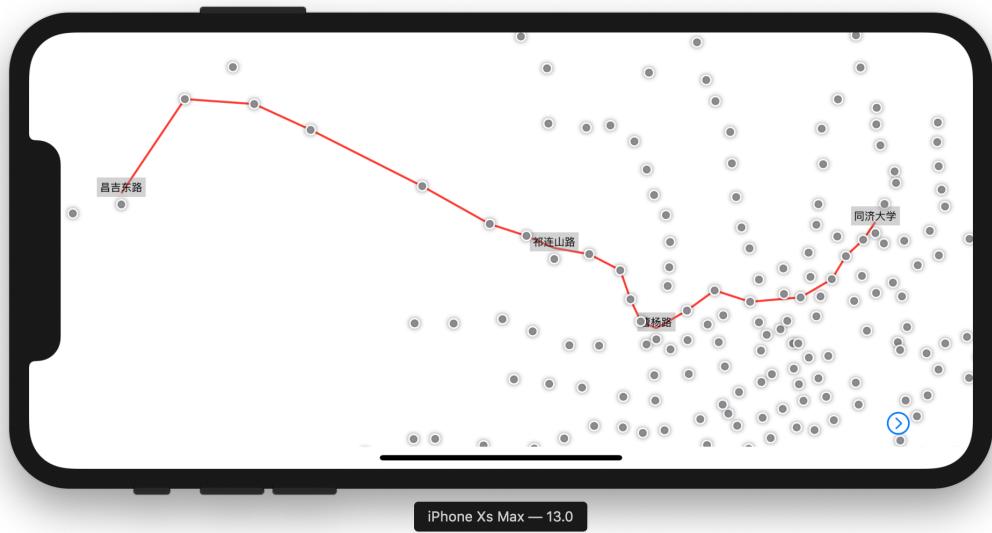
通过这个数组再导入到 ViewModel 中。具体可以参照源文件 DataHandler.swift。

## 正确性

地铁线路图经过经纬度到平面坐标的转换能基本达到预期效果。



现在查询从嘉定校区到四平校区的路线



可以看到图中清晰的红色线路，长按顶点显示站点名，发现线路符合预期。

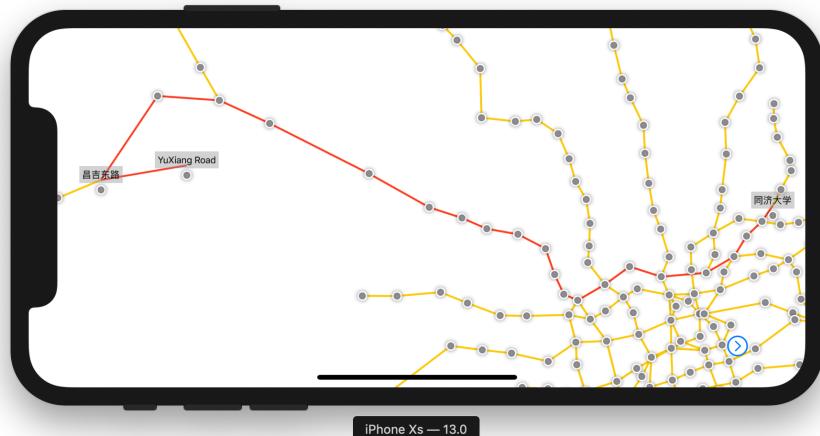
动态添加顶点通过滑动手势，首先在菜单中选择新增的站点名，再拖动到所要添加的位置，可以成功准确添加。



在此基础上添加 YuXiang Road 到昌吉东路的线路（动态添加线路是单向添加）。

添加完线路后再观察 YuXiang Road 到同济大学站的最短路线，发现结果正常。

此处的地铁黄色线路的显示是刻意为之，实际可以通过一个 toggle 切换是否显示。最短路径也是如此，均有很赞的动画效果。



## 稳定性

emmm 😅，还可以。10% 左右的概率 app 会在中途莫名 crash。但总体来说稳定性还是可观的。

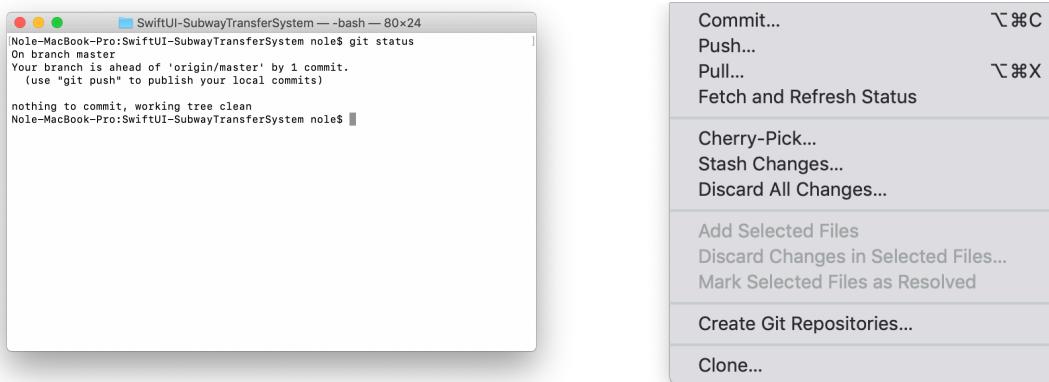
## 容错能力

从输入选择角度避免了异常输入。同时若用户选择了重名的顶点添加则不会成功，因为采用数据结构就是根据站点名来区分站点的 Dictionary，若名字相同则站点无法区分。还有，如果添加了同样的线路，因为邻接点用了集合的形式，所以集合中插入同样的值不会改变集合，因此在数据结构的角度上就避免了重复添加线路的情况。

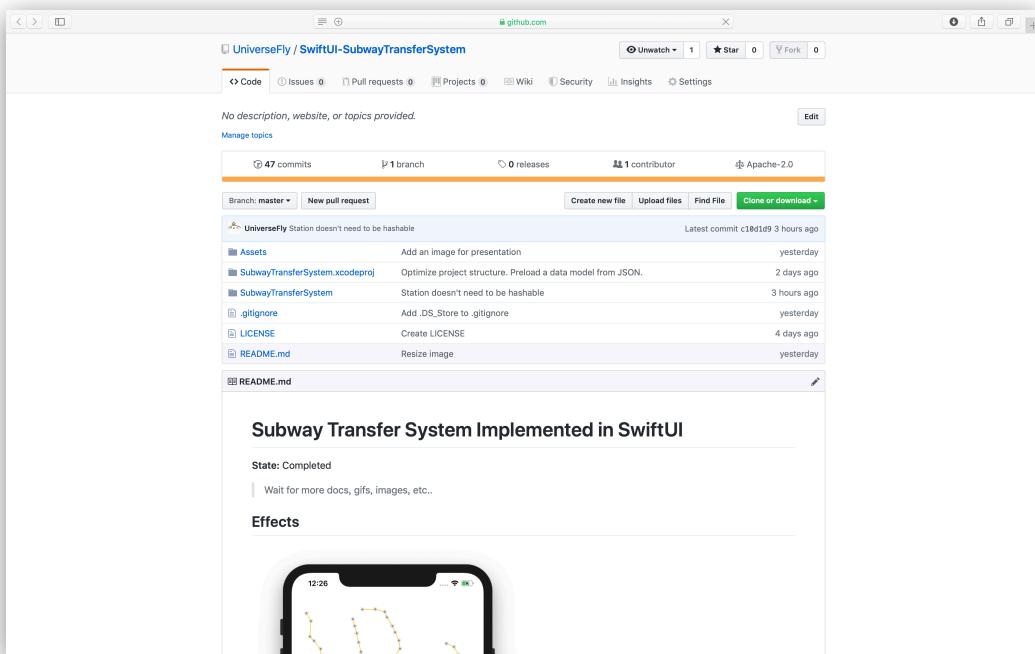
## 版本控制系统

本次开发中使用了仍然使用了 Git<sup>[6]</sup> 与 GitHub。

Git 的使用同样有两方面，Terminal、Xcode 集成。



Github地址在首页已有说明，也可以点击此处前往。



## 操作说明

操作很简单，通过悬浮菜单栏中的提示完全可以上手操作，但需要有运行环境。值得注意的是手势操作，二指捏合放开能够缩放，地图稍微放大后长按站点可以显示站点名，从一个已有站点开始拖动至某个新位置来添加新站点。

该 app 实际还有诸多不足，需要更多知识储备弥补。

# 实践总结

## 所做的工作

学习 Qt、学习 Git、学习 Swift、学习 SwiftUI、学习算法。

设计代码结构、优化代码结构、重构代码。

上手写代码。

不断调试。

完成项目。

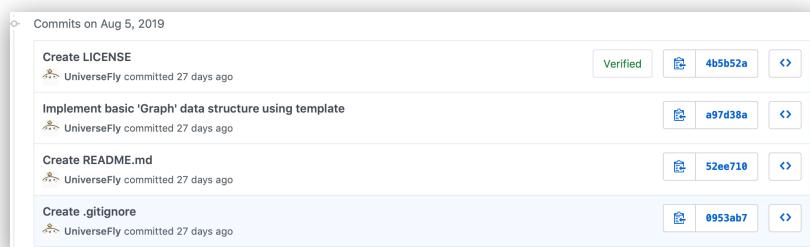
## 总结与收获

这次课程设计正好抽到了最难的题（自认为）。算法设计题画了许多的时间，由于不熟悉 Qt 需要很多方面的学习。好在 C++ 底子还算可以，学习 Qt 的路虽然艰难但收获不小。阅读了 Qt 官网上大量的文档、搜索了许多网上的教学资料，总体来说还是十分辛苦，没有什么空闲的时间。

对于 Swift 和 SwiftUI 纯粹是因为兴趣抱着试一试的想法在最后的不到 10 天内开工，写代码的体验比起 Qt 提升了好多档次。声明式 UI 真香。通过 SwiftUI 我也知道了 Flutter、Reactive Native 这两个声明式 UI 的祖先，对 Flutter 略有兴趣。

个人认为，学习的良性过程应该是：学习、实践、总结、巩固、实践。在学习 Qt 的路上我记了一些笔记，将知识存储起来（SwiftUI 暂时没记，打算重新再学一遍）。不过由于时间赶，这两个 UI 框架的基础并不扎实，很多代码也未必正确，但通过 Stack Overflow、CSDN、SegmentFault、Swift Forums 等论坛的帮助，及时解决问题完成项目。但我觉得扎实的知识基础更加重要，因此这两个框架日后还会重新学习巩固。

最后，本以为只有两道题目很简单，但我错了。一共花了将近 30 天。



---

# 参考资料

- [1] <https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html>
- [2] Stanley B. Lippman, Josee Lajoie, Barbara E. Moo, C++ Primer 中文版, 电子工业出版社, 2013
- [3] <https://zh.cppreference.com>
- [4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms Third Edition, 机械工业出版社, 2013
- [5] <https://doc.qt.io/qt-5/graphicsview.html#>
- [6] <https://git-scm.com/book/zh/v2>
- [7] <https://developer.apple.com/wwdc19/>
- [8] <https://developer.apple.com/videos/play/wwdc2015/408/>