图

概念

- 由顶点和边组成，1个顶点可以连接0条或多条边，1条边有两个顶点。顶点的集合是非空的，边的集合可以空
- 可以分为有向图和无向图以及混合图

有向图

- 有向图的边有明确的行进方向，当从一个顶点经过n条边回到该顶点的有向图称作有向有环图，否则为有向无环图
- 出度： 有x条边以该顶点为起点
- 入度： 有x条边以该顶点为终点

无向图

- 边没有明确的行进方向

有权图

边有权值的图

图的实现方式

- 顶点类value用于保存顶点的信息，inEdges集合用于保存终点是该顶点的所有边，outEdges集合用于保存起点是该顶点的所有边。

```
private class Vertex<V: Hashable, E: Comparable & Numeric>: Hashable,C
ustomStringConvertible {
        var value: V?
        var inEdges = Set<Edge<V, E>>()
        var outEdges = Set<Edge<V, E>>()
 }
```

- 边类from用于保存该边的起点的顶点，to用于保存该边的终点的顶点，weight保存该边的权值

```
private class Edge<V: Hashable, E: Comparable & Numeric>: Hashable,Cus
tomStringConvertible,Comparable {

        var from: Vertex<V, E>
        var to: Vertex<V, E>
        var weight: E?
 }
```

- 通过一个集合保存所有的边，顶点使用map保存，每个顶点的名字为key，实例对象为value

```
    class ListGraph<V: Hashable, E: Comparable & Numeric>: Graph<V, E> {
    /// 保存图中所有的边
    private var edges = Set<Edge<V, E>>()

    /// 保存图中所有的顶点
    private var vertices = [V: Vertex<V, E>]()
   }
```

图的遍历

有两种遍历方式，分为深度优先遍历和广度优先遍历。

深度优先搜索

从一个顶点起步，一直寻找可以通向的下一个顶点，直到下一个顶点是空，返回上一层重复寻找。类似于二叉树的前序遍历

```
override func dfs(begin: V, visitor: (V) -> ()) {
        let beginVertex = vertices[begin]
        guard beginVertex != nil else { return }
        var visitedVertex = Set<Vertex<V, E>>() // 记录已经遍历过的节点
        if beginVertex!.value != nil {
            visitedVertex.insert(beginVertex!)
            visitor(beginVertex!.value!)
        }
        dfs(begin: beginVertex!, visitor: visitor, visitedVertex: &visite
    }
```

广度优先搜索

从一个顶点起步，寻找所有可以通向的顶点，在所有可通向的顶点依次按此步骤处理，直到遍历完成。类似于二叉树的层序遍历。

```
override func bfs(begin: V, visitor: (V) -> ()) {
        let beginVertex = vertices[begin]
        guard beginVertex != nil else { return }
        var visitedVertex = Set<Vertex<V, E>>() // 记录已经遍历过的节点
        var array = [Vertex<V, E>]() // 当作队列
        array.append(beginVertex!) // 入队
        visitedVertex.insert(beginVertex!)

        while !array.isEmpty {
            let head = array[0]
            array.remove(at: 0) // 出队
            visitor(head.value!)
            for edge in head.outEdges {
                if visitedVertex.contains(edge.to) { continue }
                array.append(edge.to)
                visitedVertex.insert(edge.to)
            }
        }
    }
```

拓扑排序

将图中的所有顶点按照依赖顺序进行排序，某顶点的前驱顶点一定排在该顶点和后继节点前。若顶点间存在互相依赖的行为则无法拓扑排序。**（可能不唯一）**

- 前驱节点：前驱节点工作结束后才会执行
- 后继节点：某节点工作结束后才会执行后继节点

实现步骤：将图中所有入度为0的顶点放入结果集，然后把该顶点移除，直至没有入度为0的顶点。如果结果集中的顶点数量和图中顶点数量相同则排序完成，否则证明有依赖环。

```
func topologicalSort() -> [V] {
        var result = [V]()
        // 顶点：入度的map，模拟有几条入度边
        var tempMap = [Vertex<V, E>: Int]()
        // 队列放顶点
        var queue = [Vertex<V, E>]()
        vertices.forEach { (key, vertex) in
            if vertex.inEdges.count == 0 {
                queue.append(vertex)
            }else {
                tempMap[vertex] = vertex.inEdges.count
            }
        }

        while !queue.isEmpty {
            // 出队
            let vertex = queue[0]
            result.append(vertex.value!)
            queue.remove(at: 0)
            // 更新出度的节点
            vertex.outEdges.forEach { (edge) in
                let toVertex = edge.to // 顶点指向的点
                let newCount = tempMap[toVertex]! - 1 // 模拟删除入度
                if newCount == 0 {
                    queue.append(toVertex) // 度为0入队
                }else {
                    tempMap[toVertex] = newCount // 更新入度
                }
            }
        }
        return result
    }
```

最小生成树

- **适用于有权连通图**，有n-1条边把所有n条顶点全部连通的树中，总权值最小的一颗。**（可能不唯一）**

prim实现最小生成树

- 选中图中任意一个顶点A，将该顶点的出边中最短的一条选中，找到该边的终点B，将B的出边汇合到之前的边的集合中，再次寻找最短边,如果最短边的终点已经在之前的找到的顶点集合中，则放弃此边。直到将所有顶点选中。

```
private func prim() -> Set<EdgeInfo<V, E>> {
        guard vertices.count > 0 else { return Set() }
        // 已经被添加过的顶点集合
        var addedVertices = Set<Vertex<V, E>>()
        // 返回结果集合
        var edgeInfos = Set<EdgeInfo<V, E>>()
        let vertex = vertices.first!.value
        var array = [Edge<V, E>]()
        addedVertices.insert(vertex)
        // 将第一个顶点的出边建堆
        vertex.outEdges.forEach { (edge) in
            array.append(edge)
        }
        let minHeap = Heap(elements: &array, rule: .SmallTop)
        // 最小生成树顶点是图顶点的数量-1
        while !minHeap.isEmpty() && addedVertices.count < vertices.count
            // 拿出最小边
            let edge = minHeap.remove()!
            if addedVertices.contains(edge.to) { continue }
            edgeInfos.insert(edge.convertToEdgeInfo())
            addedVertices.insert(edge.to)
            // 将该边的顶点的所有出边加入堆中
            edge.to.outEdges.forEach { (e) in
                minHeap.add(element: e)
            }
        }

        return edgeInfos
    }
```

kruskul实现最小生成树

- 将图中所有边从小到大依次取出最小边和该边的两个顶点放入生成树中，从第三次开始需要判断该边的两个顶点和生成树是否存在环，若存在则放弃该边(可以使用并查集查询头尾两个顶点是否在同一集合)。直到生成树中存在了比顶点少1条边为止。

```
private func kruskal() -> Set<EdgeInfo<V, E>> {
        guard vertices.count > 0 else { return Set() }
        // 返回结果集合
        var edgeInfos = Set<EdgeInfo<V, E>>()
        let unionFind = UnionFindObj<Vertex<V, E>>()
        var edgeList = [Edge<V, E>]()
        // 所有边加入堆
        edges.forEach { (edge) in
            edgeList.append(edge)
        }
        let minHeap = Heap(elements: &edgeList, rule: .SmallTop)
        vertices.forEach { (key,vertex) in
            // 每条边自成一个并查集集合
            unionFind.makeSet(e: vertex)
        }

        while !minHeap.isEmpty() && edgeInfos.count < vertices.count - 1
            let e = minHeap.remove()!
            let v1 = e.to
            let v2 = e.from
            // 如果最小边的头尾在同一集合证明该边和树形成了环
            if unionFind.isSame(e1: v1, e2: v2) { continue }
            // 合并成同一集合
            unionFind.union(e1: v1, e2: v2)
            edgeInfos.insert(e.convertToEdgeInfo())
        }


        return edgeInfos
    }
```

最短路径

- 指两个顶点之间权值之和最小的路径**（可能不唯一）。 计算最短路径的图中不能有负权环。**

dijkstra计算最短路径

- 不能用负权边
- 将所有顶点看作石头，边看作绳子。拉起目标起点的石头后会逐渐将所有的石头拉起，直至拉起目标终点的石头，拉起的顺序就是最短路径（需要是拉起最终目标的相关石头）

```
private func dijkstra(_ begin: V) -> [V: PathInfo<V, E>] {
        guard vertices[begin] != nil else { return [:] }
        let beginVertex = vertices[begin]!
        // 被选中的最短路径map
        var selectedPaths = [V: PathInfo<V, E>]()

        // 保存所有待选择的路径
        var paths = [Vertex<V, E>: PathInfo<V, E>]()

        // 将目标的直连的顶点放入待选择paths中
        beginVertex.outEdges.forEach { (edge) in
            // 起始点都是直连线，edge的weight就是path的weight
            var pathInfo = PathInfo<V, E>(weight: edge.weight!)
            pathInfo.edgeInfos.append(edge.convertToEdgeInfo())
            paths[edge.to] = pathInfo
        }
        while !paths.isEmpty {
            // 返回最短路径元组（顶点，pathInfo）
            let minPath = getMinPath(paths: paths)
            selectedPaths[minPath.0.value!] = minPath.1
            // 已选好的顶点和路径信息从待选paths中移除
            paths.removeValue(forKey: minPath.0)

            // 选出最短路径后，纳刀最短路径的出度的边进行松弛操作
            for edge in minPath.0.outEdges {
                // 出度的点已经保存在已经选中的最短路径内，跳过
                if selectedPaths.keys.contains(edge.to.value!) { continue
                relaxDijkstra(minPath: minPath, paths: &paths, edge: edge
            }
        }
        // 移除自己
        selectedPaths.removeValue(forKey: begin)
        return selectedPaths
    }

    /// 从paths中选出一个最小的路径
    /// - Parameter paths: <#paths description#>
    /// - Returns: <#description#>
    private func getMinPath(paths: [Vertex<V, E>: PathInfo<V, E>]) -> (Ve
        var minTuples: (Vertex<V, E>, PathInfo<V, E>)?
        paths.forEach { (key,value) in
            if minTuples == nil || value.weight < minTuples!.1.weight {
                minTuples = (key, value)
            }
        }
        return minTuples!
    }
```

松弛操作: 更新两个顶点的最短路径

```
/// 松弛
    /// - Parameters:
    ///   - minPath: <#minPath description#>
    ///   - paths: <#paths description#>
    ///   - edge: <#edge description#>
    private func relaxDijkstra(minPath: (Vertex<V, E>, PathInfo<V, E>), p
        // 选出的最短边和出度的边相加算出新的可选路径权重
        let newWeight = minPath.1.weight + edge.weight!
        // 拿出原顶点保存的路径权重
        let oldWeight = paths[edge.to]?.weight
        // 对比
        if oldWeight == nil || newWeight < oldWeight! {
            var newPathInfo = PathInfo<V, E>(weight: newWeight)
            newPathInfo.edgeInfos.append(contentsOf: minPath.1.edgeInfos
            newPathInfo.edgeInfos.append(edge.convertToEdgeInfo())
            paths[edge.to] = newPathInfo
        }
    }
```

bellmanFord计算最短路径

- 可以有负权边，且可以检测出是否有负权环
- 原理：对所有边进行 v - 1 次松弛操作。即可算出目标起点到其他顶点的最短路径。

```
private func bellmanFord(_ begin: V) -> [V: PathInfo<V, E>] {
        guard vertices[begin] != nil else { return [:] }
        // 被选中的最短路径map
        var selectedPaths = [V: PathInfo<V, E>]()

        selectedPaths[begin] = PathInfo(weight: 0)

        for _ in 0..<vertices.count - 1 {
            for edge in edges {
                let fromPath = selectedPaths[edge.from.value!]
                if fromPath == nil { continue }
                relaxBellmanFord(edge: edge, fromPath: fromPath!, paths:
            }
        }

        for edge in edges {
            let fromPath = selectedPaths[edge.from.value!]
            if fromPath == nil { continue }
            if (relaxBellmanFord(edge: edge, fromPath: fromPath!, paths:
                print("有负权环")
                selectedPaths.removeAll()
```

```
                break
            }
        }
        // 移除自己
        selectedPaths.removeValue(forKey: begin)
        return selectedPaths
    }

    /// 松弛
    /// - Parameters:
    ///   - edge: 需要进行松弛的边
    ///   - fromPath: edge的from的最短路径信息
    ///   - paths: 存放着其他点的最短路径信息
    /// - Returns: <#description#>
    @discardableResult
    private func relaxBellmanFord(edge: Edge<V, E>, fromPath: PathInfo<V
        // 选出的最短边和出度的边相加算出新的可选路径权重
        let newWeight = fromPath.weight + edge.weight!
        // 拿出原顶点保存的路径权重
        let oldWeight = paths[edge.to.value!]?.weight
        // 对比
        if oldWeight == nil || oldWeight == 0 || newWeight < oldWeight!
            var newPathInfo = PathInfo<V, E>(weight: newWeight)
            newPathInfo.edgeInfos.append(contentsOf: fromPath.edgeInfos)
            newPathInfo.edgeInfos.append(edge.convertToEdgeInfo())
            paths[edge.to.value!] = newPathInfo
            return true
        }
        return false
    }
```