思路 1

https://www.cnblogs.com/longjiang-uestc/p/9539442.html

思路 2

https://blog.csdn.net/xclovecx1314/article/details/72915460

2017 华为软件挑战赛总结

这次比赛是去年做的，自己之前没有总结，现在才开始总结，很多东西快想不起来了，真是惭愧

赛题主要内容和目的

初赛题目和内容

- 给你一个流网络（边有容量和单位流量费用），已知有一些节点有流量需求（消费节点），现要选一些节点部署服务器（服务节点），给消费节点传输流量，使得在满足所有消费节点流量需求的条件下，最小化成本（服务器购买成本+线路流量费用）

- 服务器输出能力无上限，一个服务节点可以服务多个消费节点，一个消费节点也可以从多个服务节点获取流量
- 每台服务器的购买成本均相同
- 输出结果中每条边上的流量必须为整数

数据规模

- 网络节点数量不超过 1000，每个节点出度不超过 20，消费节点数量不超过 500
- 边容量和单位流量费用为[0,100]的整数，服务器与消费节点的带宽需求为[0,5000]的整数
- 非常好的一个 NP-hard 问题，数据规模和时限的设置比较合理，没有现成的模型可以立即套用
- 可以对整个比赛分为两个部分：服务器选址(启发式搜索) + 最小费用流(计算网络流的最小费用)
- 最小费用流可以认为是求解算法的底层基础设施，费用流越快，基本上可以搜索的空间就越大，可以实施的操作也越多，得到更优解的可能性也越大
- 费用流的速度决定了算法能力的上限
- SPFA 增广路算法--->原始—对偶算法（"zkw"算法）--->Cost Scaling--->增量式 zkw 算法--->网络单纯形

最小费用流问题

- SPFA 增广路算法--->原始—对偶算法（"zkw"算法）--->Cost Scaling--->增量式 zkw 算法--->网络单纯形

设施选址问题，搜索算法

- 采用局部搜索的方法(模拟退火)，从一个初始可行解出发，不断改进当前解，如果到了局部最优，尝试用一些策略跳出它，再去改进，如此循环
- 全局搜索(遗传算法和粒子群算法)，参数太多，更新速度较慢，大 Case 因为费用流的速度回降低，很容易遇到瓶颈
- 与邻居以及当前可行解的路由上的点进行启发式交换，可以进一步从质上提高解的质量，而且速度是遗传和粒子群所远不能比

复赛变化点

- 在初赛赛题的基础上，加入了服务器的差异化约束条件——服务器有 10 个档次
- 每个档次的服务器购买成本和容量都不同，每个网络节点的部署费用也可能不同。
- 也就是说，在一个网络节点部署一台服务器的成本 = 该服务器的购买成本 + 网络节点的部署成本

数据规模:

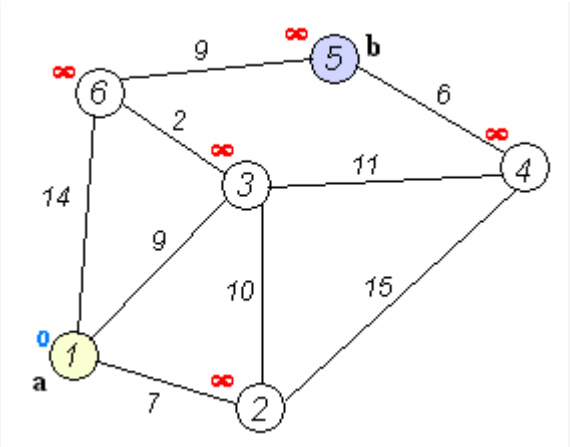
- 网络节点数量不超过 10000，每个节点的出度不超过 10000，消费节点数量不超过 10000
- 边容量与单位流量费用为[0，100]的整数

- 服务器档次不超过 10 个
- 服务器成本为[0，5000]的整数
- 服务器的输出能力、节点的部署成本与消费节点的流量需求为[0,10000]的整数
- NP-hard 的双层叠加: 除了要选择比较好的部署服务器的点集之外，还要确定服务器的档次，而一台服务器可选的档次就有 10 个之多，因而相当于面对两个 NP-hard 问题
- 30 个节点部署服务器，加入档次的条件后，即便确定了部署的服务器数量、在哪部署，还有 10^30 种可能
- 每次计算完费用流后，根据服务节点的输出流量确定相应的档次，可以有效避免成本的浪费
- 在搜索算法的前期，我们基本不考虑服务器档次，执行可行解中服务器的减少、增加和交换等操作的组合，和初赛算法类似；后期将档次的调整引入这些操作算子，并在算法的最后把降档和升档作为两个相对独立的操作过程又执行了一遍
- 只能用 C/C++/Java 自己实现算法（在我看来，理解原理是一回事，用高效的数据结构实现又是一回事。比如，多少人能手动把修正单纯形、割平面等算法撸出来）

ZKW 算法（算法复杂度是多少没有分析过）-- 最小费用可行流

- ZKW 算法详解
- 增广路径不机械使用源点的标号，应该是源点汇点标号之差

- 重赋权技术，即通过对每个节点合适的顶标，使 reduced cost 非负，这个顶标使用到汇点的距离
- 根据流量平衡条件，根据新的费用算出原费用，相当于一次 SPFA 操作
- 将所有负边强制满流，称为推流操作
- 什么是负圈?
    - 存在一个环(从某个结点出发又回到自己的路径)，而且这个环上所有路径之和是负数
    - 负权边，一条路径的和为负数

- 连续最短路径算法，弄懂 SPFA 算法和 Dijkstra 算法；
    - 

    - [dijkstra 算法讲解参考]
        - Dijkstra 算法的特点：以起点为中心向外层层扩展，直到扩展到终点为止；贪心算法，不能处理负边的情况

- 算法思想：设 G=(V 顶点，E 边)是一个带权有向图，把顶点集合分成两组，第一组为已经求出最短路径的顶点集合(用 S 表示，初始时 S 只有一个源点，以后每求得一条最短路径,就将其加到 S 中，只到全部顶点加入到 S 中，算法就结束)
- 第二组为其余未确定最短路径的顶点集合(用 U 表示)
- 按最短路径长度的递增次序依次把第二组的顶点加入 S 中;
- 在加入的过程中，总保持从源点 v 到 S 中各顶点的最短路径长度不大于从源点 v 到 U 中任何顶点的最短路径长度
- 每个顶点对应一个距离，S 中的顶点的距离就是从 v 到此顶点的最短路径长度，U 中的顶点的距离，是从 v 到此顶点**只包括 S 中的顶点为中间顶点**的当前最短路径长度

  - SPFA 可以处理负权边的情况:

    - SPFA 算法的精妙之处在于不是盲目的做松弛操作()，而是用一个队列保存当前做了松弛操作的结点
    - 只要队列不空，就可以继续从队列里面取点，做松弛操作

        - 初始时将源加入队列;每次从队列中取出一个元素，并对所有与他相邻的点进行松弛，若某个相邻的点松弛成功，如果该点没有在队列中，则将其入队。 直到队列为空时算法结束
        - 动态逼近法：设立一个先进先出的队列用来保存待优化的结点，优化时每次取出队首结点 u，并且用 u 点当前的最短路径估计值对离开 u 点所指向的结点 v 进行松弛操作，如果 v 点的最短路径估计值有所调整，且 v 点不在当前的队列中，就将 v 点放入队尾。这样不断从队列中取出结点来进行松弛操作，直至队列空为止

- 什么是拓扑排序?
    - 将有向图中的顶点以线性方式进行排序。即对于任何连接自顶点 u 到顶点 v 的有向边 uv，在最后的排序结果中，顶点 u 总是在顶点 v 的前面(打怪升级)
    - 一个有向图能被拓扑排序的充要条件就是它是一个有向无环图(DAG：Directed Acyclic Graph)
    - 拓扑排序的实现步骤:
        - 在有向图中选一个没有前驱的顶点并且输出

        - 从图中删除该顶点和所有以它为尾的弧(删除与它有关的边)
        - 重复上述两步，直达所有顶点输出，或者当前图中不存在无前驱的顶点为止，后者代表我们的有向图是由环的
        - 可以通过拓扑排序来判断一个图是否有环
    - 拓扑排序的实现算法
        - Kahn 算法
            - 使用一个栈保存入度为 0 的顶点，然后输出栈顶元素并将和栈顶元素有关的边删除，减少和栈顶元素有关的顶点的入度数量并且把入度减少到 0 的顶点也入栈
        - 基于 DFS 的拓扑排序算法
            - DFS 深度优先搜索，它每次都沿一条路径一直往下搜索，直到某个顶点灭有了出度时，就停止递归，往回走，DFS 很像 Kahn 算法的逆过程

- 定义 Di 为点 i 的距离标号(离起点的最短路径)，任何一个最短路算法保证，算法结束时对任意指向顶点 i 、从顶点 j 出发的边满足 Di <= Dj + cij (条件 1)，

且对于每个 i 存在一个 j 使得等号成立 (条件 2). 最短路径算法结束后，恰在最短路上的边满足 Di = Dj + cij

- 在最小费用流中的计算中，我们每次沿 Di = Dj + cij 的路径增广后，都不会破坏条件 1，但是可能破坏条件 2，使我们找不到每条边都满足 Di = Dj + cij 新的增广路，只好利用 Dijkstra，SPFA 等算法重新计算新的满足条件 2 的距离标号
- 什么是路径增广?
    - 路径增广是来自网络流的一种概念: 一定能找到一条路上每段路还允许流过流量的最小值 delta(容量 - 流量)；把这条路上每一段流量都加上这个 delta 值，一定可以保证这个流依然是可行流，这样我们可以得到更大的流，它的流量是之前的流量加上这个 delta，这条路就叫做增广路。
    - 增广路径是找出在残留网络中从源点到汇点的有向路径
    - 增广路径的残留容量是路径中任意边所形成的最小残留容量，我们可以沿着增广路径从源点到汇点发送额外的流
    - 流 x 是最大流当且仅当这个残留网络中不包含其他增广路径
- KM 算法中可以不断修改可行顶标，不断扩大可行子图
- 什么是 KM 算法?
    - KM 算法是对匈牙利算法的一种贪心扩展，为了高校求解二分图最佳完美匹配问题的一种算法

- Kuhn - Munkras 算法流程:
  - 初始化可行顶标的值
  - 用匈牙利算法寻找完备匹配
  - 若未找到完备匹配则修改可行顶标的值
  - 重复 2,3 直到找到相等子图的完备匹配为止，完美匹配是最大匹配
- KM 算法的核心部分即控制修改可行顶标的策略使得最终可到达一个完美匹配

- 什么是匈牙利算法?
  - 匈牙利算法是为了解决二分图的最大匹配(二分图的最大匹配也可以转换为一个网络流的问题)，二分图等价于不含**奇数条边的环**的图
  - 交错路
  - 增广路: 起点和终点都为未匹配点的交错路为增广路(这里的增广路和网络流中的增广路意义不同)
    - 增广路一定有奇数条边，而且未匹配边一定比匹配边多一条

    - 如果找到了一条增广路，那么将未匹配点与匹配边的身份调换,那么匹配的边数就多了一条，这样直到找不到增广路为止，那么整个图的匹配的边数一定最大，也就是找到了二分图的最大匹配

- ZKW 的核心就是在始终满足条件 1 的距离标号上不断修改，直到可以继续增广(满足条件 2)
- 初始标号为 0，不断增广，如果不能增广，修改标号继续增广，直到彻底不能增广(源点的标号已经被加到了正无穷)
- ZKW 的程序中，所有的 cost 均表示 reduced cost，即 cij = cij - Di + Dj；另外这个算法不能直接用于有任何负权边的图，更不能用于负权边的情况
- ZKW 算法只需要增广，改标号，不需要队列，BFS，SPFA 等复杂的操作
- ZKW 算法的精髓在于修改顶标，然后不断增广，不能增广了，在看能不能修改顶标，如果能就继续不断增广，如果不能的话就返回函数结果
- 实际增广是沿最短路进行的，时间复杂度与 SPFA 等连续最短路算法一致，但节约了 SPFA 或 Dijkstra 的运行时间，算法常数很小，速度较快
- Bellman-Ford 算法
- Dijkstra 的贪心算法的本质是如下条件要成立：如果存在某条路径 p,使得 p 是从顶点 u 到 v 的最短路径：
- p = u->v1->v2...->vn->v，则对于任意的 1<=k<=n，需要满足 d(u,vk) < d(u,v)。

- 很显然，这个条件要满足的话，那么需要图中无负权边才行
- Dijkstra 和 Bellman-Ford 的区别是:
    - 前者过于贪心，负权的情况他不考虑，也无能力考虑。
    - 后者反复调整，保证精确，顺便可以探知无解的情况。
- Floyd 算法:
    - Floyd-Warshall 算法（Floyd-Warshall algorithm）是解决任意两点间的最短路径的一种算法，可以正确处理有向图或负权的最短路径问题，同时也被用于计算有向图的传递闭包。Floyd-Warshall 算法的时间复杂度为 O(N3)，空间复杂度为 O(N2)
    - Floyd 算法是一个经典的动态规划算法;
    - 从任意节点 i 到任意节点 j 的最短路径不外乎 2 种可能，1 是直接从 i 到 j，2 是从 i 经过若干个节点 k 到 j;
    - 所以，我们假设 Dis(i,j)为节点 u 到节点 v 的最短路径的距离，对于每一个节点 k，我们检查 Dis(i,k) + Dis(k,j) < Dis(i,j)是否成立，如果成立，证明从 i 到 k 再到 j 的路径比 i 直接到 j 的路径短，我们便设置 Dis(i,j) = Dis(i,k) + Dis(k,j)，这样一来，当我们遍历完所有节点 k，Dis(i,j)中记录的便是 i 到 j 的最短路径的距离。
    - Floyd 算法描述:

- 从任意一条单边路径开始。所有两点之间的距离是边的权(求和)，如果两点之间没有边相连，则权为无穷大;
- 对于每一对顶点 u 和 v，看看是否存在一个顶点 w 使得从 u 到 w 再到 v 比已知的路径更短。如果是更新它;

- Floyd 算法过程矩阵的计算----十字交叉法

服务器选址算法

- 近邻算法的大致思路
- 第一步： 初始化一个 bool 型数组，有 800 个网络结点就声明大小为 800 的 bool 型 vector v, v[i] 为 false 表示不选择第 i 号网络结点放置服务器，v[i]为 true 表示选择 i 号网络结点放置服务器；可以随机初始化那些结点放置服务器，那些结点不放置服务器；也可以与消费结点相连的结点初始化为放置服务器；
- 第二步：循环迭代一次数组 v(这个相邻是 id 相邻，1,2,3,4,5…)，针对每一个结点会存在两种情况：选与不选

- 针对 v[i]之前本来是 true 的情况(也就是遍历的时候节点 i 已经被选择当做服务器)
  - 把 v[i]设置为 false(即从服务器集合中删去第 i 个网络结点)，再计算一次最小费用流；删掉一个结点会存在三种情况

        - 不能满足消费结点网络流的需求，把第 i 号节点的 v[i]设置为 true，并加入服务器集合中
        - 能满足消费结点的需求，但是总费用变得更高，把第 i 号节点的 v[i]设置为 true，并加入服务器集合中
        - 能满足消费结点的需求，而且总费用变得更低，只有这种情况才真正的删除这个第 i 号网络节点
    - 针对 v[i]之前本来是 false 的情况(第 i 号节点不是服务器)
        - 先把第 i 号节点加入服务器集合，v[i]设置为 true，并重新计算一次最小费用流；添加一个结点也会存在三种情况
            - 不满足消费节点网络流的需求，证明添加第 i 号节点之前和之后，服务器集合都是无解的，所以继续添加下一个结点知道有解
            - 能满足消费结点的需求，但费用更高，把第 i 号节点设置为 false，并从服务器集合中删除

        - 能满足消费结点的需求，而且费用更低，将全部最小费用更新为总费用值，并继续迭代

- 第三步：遍历服务器集合中的所有结点，与原来不是服务器结点的相邻结点进行交换(这里的相邻结点是指邻接表中的相邻，即在图中连接的相邻)
    - 存在两种情况： 造成更好的结果(满足消费节点的需求，并且总费用还降低了)和更坏的结果(总费用增高，或者是不满足消费结点的需求)
    - 如果结果变好则继续进行交换，如果结果变坏则还原成之前的情况

输入输出

- 输入输出要求都以文件的形式输入输出
    - 实现了两种不同的构图方式：邻接矩阵和邻接表
    - 输出利用 DFS 递归搜索所有有效的路径，并把其保存在一个二维数组中

https://blog.csdn.net/xclovecx1314/article/details/72915460

2017 年华为软件精英挑战赛初赛解题思路

题目链接： http://codecraft.huawei.com/

常规解题思路：网络流（最小费用最大流） + 启发式搜索算法 + 算法性能优化

1. 最大流最小费用算法

假设一个超级源点和超级汇点，其中将整个网络设置反向。超级源点和所有的消费节点相连，连接边的最大流量就是消费节点的消费流量；超级汇点和所有的网络节点相连，连接边费用为零，如果该网络节点被选择设置为服务器连接点，则连街边最大流为无穷大，否则为 0。

基于该假设可以选择使用景点的最小费用最大流求解。但是因为运行时间有限，可以采用改进的 spfa 算法进行运算，具体的实现思路和代码可以参考： https://github.com/lanyangyang2631546/20170308minimum-cost-flow-in-maximum-flow

后来还有效率更高的 zkw 算法： https://github.com/sanshanxiashi/MCMF_zkw ;

2. 启发式搜索算法

采用启发式搜索算法的目的就是寻找最优的网络节点挂载服务器。

大体看了一圈，貌似模拟退火的算法要比遗传算法效果好，效率高；但是我们采用的是兄弟实验室最新的成果：

论文 http://lamda.nju.edu.cn/yuy/GetFile.aspx?File=papers/aaai16-racos.pdf

代码 http://lamda.nju.edu.cn/yuy/

在 github 上有一段大规模数据的优化算法，适合用在大规模的数据上

3. 算法的性能优化

工程算法和学术算法不一致的地方就在于工程算法更加注重算法的性能优化，包括代码重构、循环剪枝等；

还有一个提升性能的方法就是选择合适的优化初始值，可以从直连节点进行比较筛

选得到。

如果仅仅有 1-3 点还是不够的，还需要从两个方面进行考虑：

一方面，需要对搜索进行有效剪枝，避免在局部最优陷入过长时间等等。

另一方面，应当充分利用网络信息进行启发式搜索，比如寻找相邻的消费节点，访问度数较低的节点合并入访问度数高的节点等等。