PPI网络的构建及最短路径算法

1.Motivation

随着分子生物学研究进入以蛋白质组学为标志的后基因组时代，蛋白质相互作用成为蛋白质组学研究的一个重要主题，然而，**蛋白质相互作用是极其复杂的。**

在常见的数据结构中，在**线性表**中，数据元素之间是被串起来的，仅有线性关系，每个数据元素只有一个直接前驱和一个直接后继。在树形结构中，数据元素之间有着明显的层次关系，并且每一层上的数据元素可能和下一层中多个元素相关，但只能和上一层中一个元素相关。而图是一种较线性表和树更加复杂的数据结构。在**图形结构**中，结点之间的关系可以是任意的，图中任意两个数据元素之间都可能相关。因此一个蛋白质-蛋白质相互作用数据集可以被建模为一个无向网络，可以加权或非加权。

其中，在**蛋白质-蛋白质相互作用网络**中，两个蛋白质之间的最短路径可以作为它们的生物学亲缘关系的指示，这一鉴定相似性的方法因为摆脱复杂的实验推断而广受关注。

因此在本项目中，研究员通过算法分析**蛋白质-蛋白质相互作用网络中的所有对最短路径**来推断**生物学相似性。**

2.Solution

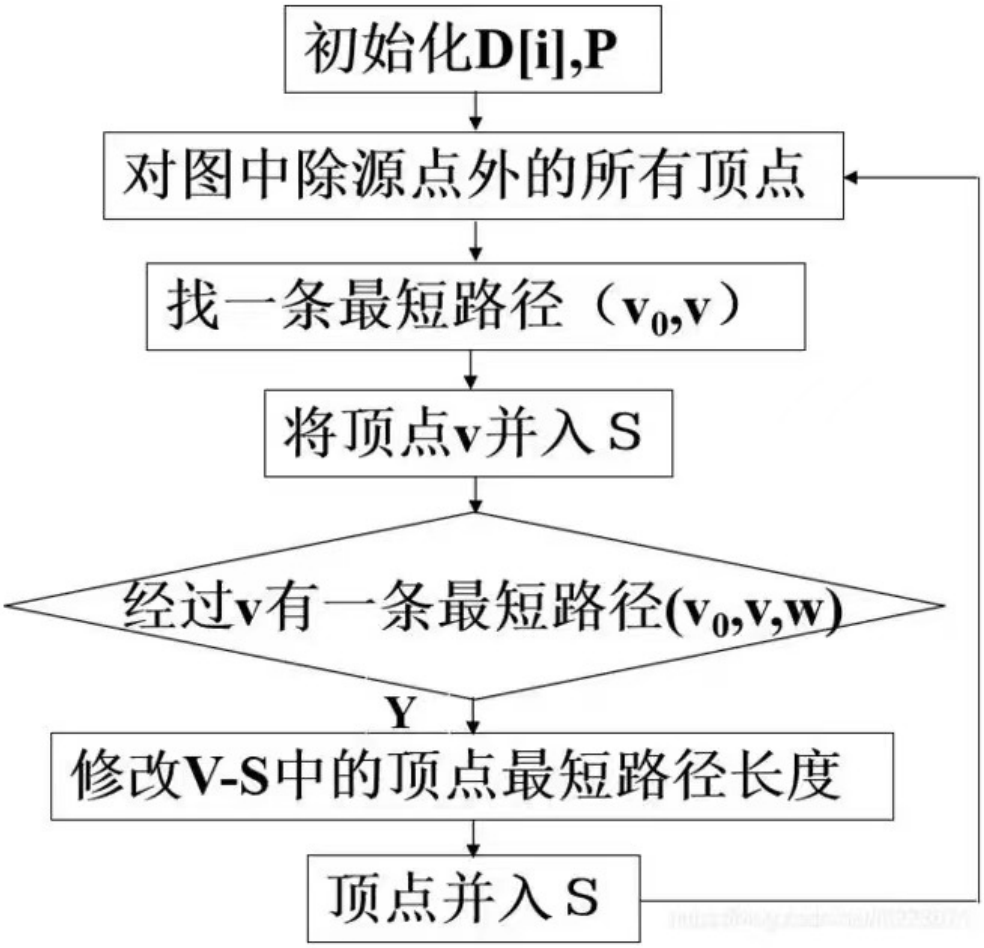
现有的最短路径算法主要有**Floyd**、**Dijkstra**、**Bellman-ford**、**DFS**四种种。下面将具体介绍其算法思想以及优缺点。

2.1 Floyd 算法

2.1.1算法流程分析

Floyd-Warshall算法是一种在具有正或负边缘权重（但没有负周期）的加权图中找到最短路径的算法。算法的单个执行将找到所有顶点对之间的最短路径的长度（加权）。该算法也称为该算法也称为Floyd算法，Roy-Warshall算法，Roy-Floyd算法或WFI算法，是一个有三个for循环嵌套的动态规划的例子。

核心思路为通过一个图的权值矩阵求出它的每两点间的最短路径矩阵。从图的带权n×n邻接矩阵开始，迭代地进行n次更新，即由矩阵D(0)=A，按一个公式，构造出矩阵D(1)；又用同样地公式由D(1)构造出D(2)；……；最后又用同样的公式由D(n-1)构造出矩阵D(n)。矩阵D(n)的i行j列元素便是i号顶点到j号顶点的最短路径长度，称D(n)为图的距离矩阵，同时还可引入一个后继节点矩阵path来记录两点间的最短路径。采用松弛操作，对在i和j之间的所有其他点进行一次松弛。



伪代码如图：

Algorithm 1: Floyd

Data: a copy of the adjacency matrix of Protein protein interaction network $*g$

Result: the shortest way between each two proteins in adjacency matrix

```
for k ← 0 to MAX do
    for i ← 0 to MAX do
        for j ← 0 to MAX do
            if g[i][j] > g[i][k] + g[k][j] then
                g[i][j] = g[i][k] + g[k][j]
            end
            ;
        end
    end
end
```

2.1.2 优缺点分析

优点：容易理解，代码容易编写；动态规划法，稠密图效果最佳，边权可正可负。

缺点：该方法时间复杂度和空间复杂度都是O(n^3)，数据较大时计算耗时较长不适合。

比较Dijkstra与Floyd

1. Dijkstra不能处理负权图，Floyd能处理负权图；
2. Dijkstra处理单源最短路径，Floyd处理多源最短路径；
3. Dijkstra时间复杂度为O（n^2^），Flyod时间复杂度为O（n^3^） 空间复杂度为O（n ^2^）。
4. 总结：如果是单源点正权图，就用Dijkstra
   如果是任意两个点之间的最短路径或者是负权图，就用Floyd

2.1.3 改进思路

1. 利用矩阵对称性进行改进。

```
int i,j,k,t;
for(k = 1; k <= N; k++){
    for(i = 1; i <= N; i++){
        t = A[i][k];
        for(j = 1; j <= i; j++){
            A[i][j] = min(A[i][j], t+A[k][j]);
            A[j][i] = A[i][j];
        }
    }
}
```

2. 只用矩阵的下三角部分

```
int i,j,k,t;
for(k = 1; k <= N; k++){
    for(i = 1; i <= N; i++){
       if(k != i){
            t = (k<i) ? A[i][k] : A[k][i];
            for(j = 1; j <= min(k,i); j++){
                A[i][j] = min(A[i][j], t+A[k][j]);
```

```
            }
            for(j = k+1; j <= i; j++){
                A[i][j] = min(A[i][j], t+A[j][k]);
            }
        }
    }
}
```

3.跳过不存在的路径

```
    int i,j,k,t;
    for(k = 1; k <= N; k++){
        for(i = 1; i <= N; i++){
            if(k != i){
                t = (k<i) ? A[i][k] : A[k][i];
                if(t == Inf)  continue;
                for(j = 1; j <= min(k,i); j++){
                    A[i][j] = min(A[i][j], t+A[k][j]);
                }
                for(j = k+1; j <= i; j++){
                    A[i][j] = min(A[i][j], t+A[j][k]);
                }
            }
        }
    }
```

4. 避免调用大量函数（min）

```
    int i,j,k,t;
    for(k = 1; k <= N; k++){
        for(i = 1; i <= N; i++){
            if(k != i){
                t = (k<i) ? A[i][k] : A[k][i];
                if(t == Inf)  continue;
                for(j = 1; j <= ((k<i)?k:i); j++){
                    if(t+A[k][j] < A[i][j])  A[i][j] = t + A[k][j];
                }
                for(j = k+1; j <= i; j++){
                    if(t+A[j][k] < A[i][j])  A[i][j] = t + A[j][k];
                }
            }
        }
    }
```

2.2 Dijkstra算法

2.2.1 DijA简介

从起始点开始，采用贪心算法的策略，每次遍历到始点距离最近且未访问过的顶点的邻接节点，直到扩展到终点为止。

2.2.2 代码实现

```
void dij(int n,int origin,int graph[][])
{
  int count = 0,i,j;           //count是已求出最短路径的顶点数目
    visit[0] = 1;
    count++;
    for (i = 1; i < n; i++)    //初始化
    {
        dis[i] = graph[origin][i];
    }
    while (count < n)
    {
        int min = INF, target_index;
        for (i = 1; i < n; i++)
        {
            if (visit[i] == 0 && min > dis[i])     //找到距离源点最短的target_index
            {
                min = dis[i];
                target_index = i;
            }
        }
        visit[target_index] = 1;
        count++;
        for (i = 1; i < n; i++)
        {
            if (visit[i] == 0 && dis[target_index] +
            graph[target_index][i] < dis[i])            //更新
            {
                dis[i] = dis[target_index] + graph[target_index][i];
            }
        }
    }
}
void getdij(int n,int origin,int graph[][],char prodic[][])
{
    int i,j;
    dij(n,origin,graph[1000][1000]);
    printf("\n\n");
    for (i = 1; i < n; i++)
    {
        if (dis[i] == INF)
        {
          printf("There is no interaction between protein %s and protein %s \n",
          prodic[origin],prodic[i]);
        }
        else
            printf("Shortest path between protein %s and protein %s is %d
            \n",prodic[origin],prodic[i],dis[i]);

    }
}
```

2.2.3 特点

在一类特殊问题下，有着足够好的准确率与运行速度。当数据形式理想时，时间复杂度可以进一步降低。

对于存在负权边的图，相较于SPFA，大部分情况下只能得到局部最优解，无法得到全局最优解。

2.3 Bellman-Ford算法

2.3.1 SPFA简介

Bellman-Ford Algorithm是针对于传统路径生成方法中，由于"负权边的存在，无法很好发挥作用的问题所创造的一种算法。

其特点为，所有候补节点并非平权，在选择可能路径上有一定的倾向性。但最终所有路径都会遍历，同DijA相比，时间复杂度会略高一些，但不会出现由于负权边导致的错误答案等问题。

若网络中，顶点数为n，边数为m；针对每个顶点都进行一轮更新，每次更新复杂度为O(m)，一共需要更新n次，那么算法复杂度为O(mn).

2.3.2 代码实现

```
def BFA(graph, source):
    dst = {}
    p = {}
#呜呜，还是Python好，可以直接用无穷大保留字
    for v in graph:
        dst[v] = float('inf')
        p[v] = None
    dst[source] = 0
#迭代迭代
    for i in range(len( graph ) - 1):
        for u in graph:
            for v in graph[u]:
                if dst[v] > graph[u][v] + dst[u]:
                    dst[v] = graph[u][v] + dst[u]
                    p[v] = u
#判断有无哈密顿回路
    for u in graph:
        for v in graph[u]:
            if dst[v] > dst[u] + graph[u][v]:
                return None, None

    return dst, p
```

2.3.3 补充说明

无论在更为鲁棒的SPFA中，还是在不包括DFS/BFS的其他路径生成算法中，对于输入的图网络都要保证一点，即所有边权的加和不应小于0，否则最终会进入死循环，无法给出合适的结果。一个简单的例子是，针对各边权均为负数的哈密顿回路，每次循环，都会使路径长度变短，最终会得到"在回路中循环无数遍，即为最短路径"这种明显不合适的结论。

2.4 DFS算法

2.4.1 DFS 算法介绍

DFS即Depth First Search，是一种用于遍历或搜索树或图的算法。 沿着树的深度遍历树的节点，尽可能深的搜索树的分支。其过程简要来说是对每一个可能的分支路径深入到不能再深入为止，而且每个节点只能访问一次。而DFS算法用于搜索最小路径时，其核心思路为1.使用一个数组充当递归工作栈和另一个数组用来标记节点是否被访问过，搜索从起始点开始2.读取一个节点之后的操作为按序读取第一个未被经过的节点，将读取的节点在数组中标记为1，并将被读取节点推入栈中。3.边界条件为读取到的节点为目标节点或已走路径长度大于现有最小路径长或没有更多的未经过节点，第一种情况下将最小路径长更新为当前已走路径长。遇到边界情况则将读取到的节点推出栈。其时间复杂度为O（n**2）

2.4.2 实现

```
//DFS算法核心递归函数
void dfs(int cur, float dst, int final,float matrix[21][21]){
    int n = num;
    if (minPath < dst) return;
    if (cur == final){
        int j;
        if (minPath > dst){
            minPath = dst;
            for (j = 0; j < num; j++){
                answercpy[j] = answer[j];
            }
        }
        return;
    }
    else{
        int i;
        for (i = 0; i < num; i++){
            if (matrix[cur][i] != inf&& visited[i] == 0){
                visited[i] = 1;
                answer[hope] = i;
                hope++;
                dfs(i, dst + matrix[cur][i], final,matrix);
                visited[i] = 0;
                hope--;
                answer[hope] = -1;
            }
        }
        return;
    }
}

//DFS算法函数，输入起始protein序号，终止protein序号，蛋白质数量，邻接矩阵，蛋白序列，输出最短路径及其长度
float DFS(int start, int des,int npro,float matrix[21][21],char prodic[200][5]){
    int i;
    num = npro;
    memset(answercpy, -1, sizeof(int)*num);
    memset(answer, -1, sizeof(int)*num);
    minPath = 999999;
    hope = 1;
    answer[0] = start;
    dfs(start, 0, des,matrix);
```

```
    printf("the path is:");
    for (i = 0; i < num; i++){
        if (answercpy[i] != -1){
            printf("%d->", answercpy[i]);
        }
    }
    printf("end");
    printf("minPath=%f\n", minPath);
    return minPath;
}
```

2.4.3 比较DFS与BFS

1. DFS 是靠递归的堆栈记录走过的路径，要找到最短路径，需要把图中所有路径都探索完才能对比出最短的路径有多长。而 BFS 借助队列做到一次一步「齐头并进」，是可以在不遍历完整棵树的条件下找到最短距离
2. BFS 可以找到最短距离，但是空间复杂度高，而 DFS 的空间复杂度较低。 DFS空间复杂度一般为O(logN) BFS的空间复杂度为O(N) N为节点个数。

3.Improvement

3.1 图结构的存储

在常见的图网络问题中，存储图结构的方式多为以下两种：

矩阵形式（NumPy,etc）

创造一个矩阵，其横纵坐标都设置为图中点的编号。例如，在无向图的矩阵中，（3，5）位置的分数就代表了点3与点5之间的距离，或者有无邻接。并且（3，5）与（5，3）位置的分数应该为相同的。

在有向图中，（3，5）代表着从3出发、终点为5，（5，3）则反之。由于是有向图，则沿主对角线对称的点打分不一定相同。

张量形式（networkx,etc）

将图的信息存储为三个张量：u,v,w。对于其中的连接关系i，有：u[i]为出发点，v[i]为终点，而w[i]则代表从u[i]到v[i]这条有向边的权重，或距离。

可以看出，当图的结构非常大时，二者都会非常臃肿，极大影响运行速度。而同矩阵相比，张量形式的特点为：不会存储不连通的关系。networkx偏爱张量形式的存储，是因为这避免了矩阵稀疏的问题。

但，图的点之间连通度较小时，就不能使用矩阵了吗？尤其是面对邻接矩阵这种对于图聚类至关重要的数据，也只能转为张量再行处理吗？

针对存储的问题，我们也有解决方案。

3.1.1 使用SciPy

对于Sparse矩阵进行压缩处理。清除掉无用数据之后，还能一定程度上保持矩阵的形状，便于存储与运算处理。

3.1.2 借助DC策略

借助分治策略，将原先的系数矩阵划分为若干子矩阵，在保证这些子矩阵为Dense矩阵之后，对子矩阵进行操作。

之后再对若干子矩阵操作后的结果进行操作，这样借助矩阵投影与分治策略来减少存储压力。

形象的描述就是：海洋上的诸多岛屿，岛屿内部四通八达，但岛屿之间籍由少数桥梁连接。如果想从A岛的某一处到达C岛，那么分别计算在每个岛屿内部到达桥梁的最短距离，之后再去将这些路径拼起来即可。岛屿就对应着子矩阵，桥梁是稀疏连接，海洋则是不连通的那些无效内容。

3.2 图聚类

图聚类是从传统聚类脱胎而来的聚类方式。虽然常见于城市交通规划设计（借助POI、路网等数据，分割城市生活圈）、图像处理（使用图的形式存储图像内容，再进行聚类操作，进行图像分割）等，但是同样可以用来处理PPI相关的任务。

如，networkx这一数据库，就提供了很多有用的工具与方法，比如基于邻接矩阵、NLP等信息进行聚类活动。基于邻接矩阵进行聚类是一种常规的使用方式，某种意义上可以看成上文所述的“寻找子空间”任务的强化版。下面对于借助NLP的图聚类方式进行介绍。

NLP，Natural Language Processing，即自然语言处理。原本是用来分割语义、构建图谱的一种ML方法。但是，随着应用面的不断扩大，NLP已经可以视为“针对无法量化的数据的万能工具”了。较为夸张的断言为，一切数据量足够大的非欧数据都可以借助NLP来进行量化。

在networkx中，城市道路是"单行道还是双行道"，“是否允许掉头”这样看不出什么特点的分类数据，借助内置的NLP工具，依然可以成为多维聚类的维度之一，或无法进行正则化，而是若干项结合起来成为一个维度。聚类的结果，也可以帮助进行预测活动。

而对于我们的PPI工作而言，与NLP关系最为密切的莫过于蛋白质的功能与结构这些难以量化的内容。但是可以借助networkx工具，选择合适的特征，来尝试进行图聚类。

虽然我们未进行这种操作，但是一个猜想是：对于泛素E1、E2、E3三个家族而言，如果选择适度低的粒度，他们会被归为三类；而如果选择适度高的粒度，这三类又会合为一类，即泛素相关蛋白。

而对于更多的未知类型的蛋白质而言，相信借助聚类结果，我们会从中“挖掘”出更多的“数据”，如功能群、结构群等，以及更多我们未知的蛋白质分类，为生命科学的研究找到全新且独特的视角！