链路补全以及排序

1 功能概述

链路补全与排序旨在根据提供的标签序列输出连通、有序的标签序列，并最后生成逻辑正确的 match 语句。标签，也叫做实体类型，是指图谱中的节点，在链路补全、排序功能中，整个图谱的概念框架可以看作是一个图，标签可以看作图中的顶点，由于不同标签之间的关系的指向方向对链路补全、排序不造成影响，因此图谱可以看作一个无向图。

链路补全的功能在与补全给定标签序列中缺失的部分标签。由于用户概念模糊或者混淆，给定的标签序列中的各个标签之内可能不是连通的，为了生成完整的 match 语句，需要对原标签序列中补充部分标签以保证新的标签序列构成一条连通的路径。例如系统识别出用户的输入包括两种实体标签:

[“基金经理人”，“股票”]

而两者之间不连通，因此需要补充” 基金” 实体标签来完善这个标签序列:

[“基金经理人”，“股票”，“基金”]

链路补全后的排序功能保证 match 语句的逻辑正确。由于系统只能识别用户输入的自然语言包含哪些标签，并不能识别它们的顺序，而 match 语句中要求相邻的两个实体类型中有关系存在，如果不存在，则数据库系统会对两种类型的数据做笛卡尔乘积，其造成的开销在我们的系统中是不能接受的。因此，我们现需要保证排序后的标签序列中，数组中相邻的两个标签之间一定存在关系。例如系统经过补全，得到的标签序列是

[“基金经理人”，“股票”，“基金”]

在这个序列中，“基金经理人” 与 “股票” 之间没有联系，因此我们需要对其调整顺序为:

[“基金经理人”，“基金”，“股票”]

这是一个可行的序列，因为 “基金经理人” 与 “基金” 之间有 “管理” 关系，“基金” 与 “股票” 之间有 “持有” 关系。

2 算法设计

对于链路补全与排序，整体思路是使用一个路径搜索算法同时完成补全与排序，该算法反复搜索直到搜索的路径之中完全包含给定的标签序列输出当前路径。但由于给定的标签集合中可能会有重复标签，因此我设计了两种大体上相同，细节上有差异的算法以应对不同情况。两种算法中涉及到的路径搜素算法均为 A* 搜索，搜索的起点可能是给定的标签序列中的每个标签，遍历所有的搜索起点并得到对应的序列，最后求出任以长度最短的路径即可。

Algorithm 1 是两种情况共用的整体框架，两者的区别在于其中的 A* 算法，分别在 2.1 和 2.2 会提到。

Algorithm 1: Link Completion & Rearrangement

Data: 标签序列 T, 邻接矩阵 G
Result: 连通且有序的标签序列 T
Initialize: $shortest\_path = Null, shortest\_length = inf$;
for $t \in T$ **do**
    $path = AstarSearch(t, T, G)$;
    if $path \neq Null \wedge path.length < shortest\_length$ **then**
        $shortest\_length = path.length$;
        $shortest\_path = path$;
    end
end
Return $shortest\_path$;

2.1 无重复标签

本节的 A* 算法 (2) 是最基本的 A* 算法，不能展开已经遍历过的节点，启发式估计函数估计的是给定标签序列中还未被遍历到的标签个数，容易证明这个估计是 admissible 的（估计出的代价总是低于真实代价）。

Algorithm 2: A* search without duplication

Data: 搜索起点 t, 标签序列 T, 邻接矩阵 G
Result: 连通且有序的标签序列 T
Initialize: 优先级队列 Q;
$Q.append((0, t, []))$;
while Q *is not empty* **do**
    $f, current\_node, path = Q.pop()$;
    $path.append(current\_node)$;
    $h = (T \setminus path).length$;
    $g = path.length$;
    if $h = 0$ **then**
        Return $path$;
    end
    for $next\_node \in current\_node.neighbors()$ **do**
        if $next\_node \notin path$ **then**
            $Q.append(g + h, next\_node, path)$;
        end
    end
end
Return $Null$;

2.2 有重复标签

本节的 A* 算法 (3) 在 2.1 的基础上，修改了启发式估计函数的设计，以及分支展开的条件，使得搜索的路径可以包含重复的顶点。启发式函数估计与 2.1 类似，但由于有重复值，因此我用计数的方式来统计给定标签中，哪些标签还要遍历几次，并对这些次数求和作为估计值。另外，在邻节点展开时，由于路径可重复，所以任意邻节点都需要展开搜索，因此不需要任何条件。

Algorithm 3: A* search with duplication

Data: 搜索起点 t, 标签序列 T, 邻接矩阵 G

Result: 连通且有序的标签序列 T

Initialize: 优先级队列 Q, 计数器 C;

$Q.append((0, t, []))$;

while Q *is not empty* **do**

  $f, current\_node, path = Q.pop()$;

  $path.append(current\_node)$;

  $h = sum(max(T.count(e) - path.count(e), 0)\ for\ e \in T)$;

  $g = path.length$;

  if $h = 0$ **then**

    Return $path$;

  end

  for $next\_node \in current\_node.neighbors()$ **do**

    $Q.append(g + h, next\_node, path)$;

  end

end

Return $Null$;

2.3 时间复杂度分析

A* 算法的时间复杂度与分支大小、搜索长度有关，假设标签序列长度为 n，我们可以确定搜索长度为 O(n)，分支大小为图谱每个节点的平均度数，约为 3。

因此，2.1 中 A* 算法 (2) 的时间复杂度上界是 $O(3^n)$，总共开销是 $O(n \cdot 3^n)$。

但对于 2.2 的算法 (3) 而言，由于改变了分支进入的条件，不属于给定的标签序列的节点都可以展开，因此时间复杂度与图谱所设计的标签类型数量有关。在我们的图谱里，共有 14 个标签，因此除去当前节点以外，分支大小上界为 13，A* 算法的时间复杂度上界为 $O(13^n)$，总共开销是 $O(n \cdot 13^n)$。理论上而言，这个算法在无解的情况下会陷入死循环，但图谱里的所有节点都是相通的，因此无论任何输入都是有解的，不会陷入死循环。

对于实际应用而言，最极端的 5 跳查询下，n=6，2.1 算法的运行耗时低于 1ms，2.2 算法的运行耗时在 2ms 左右，因此其时间开销几乎可以忽略不计。