算法说明文档：层次化内存分配与域字典构建算法

1. 概述

本算法旨在解决层次化数据结构中的内存分配问题，并在此基础上构建域字典，以优化存储和访问效率。算法核心是通过宽度优先搜索遍历树形结构，动态分配内存块，并记录每个节点的内存占用情况。同时，利用深度优先搜索构建域字典，以便于后续查询操作。

2. 算法目标

- **内存分配**：为每个节点分配最优的内存位置，减少碎片，提升内存使用效率。
- **拓扑排序**：确定节点间的拓扑顺序，用于指导内存分配顺序。
- **域字典构建**：根据节点间的关系，生成一个映射关系，用于快速定位相关节点集合。

3. 数据结构定义

- **Node结构体**：包含节点ID、宽度、层级、子节点、兄弟节点以及起始位置信息。

```
struct Node {
        int id;
        int width;
        int level;  // 节点拓扑序，用level表示
        int left;  // 子节点信息
        int right; // 兄弟节点信息
        int pt[3];  // 邻接指针
        int pt2[3]; // 后接指针
    };
```

- 私有数据结构：

```
    unordered_map<int, int> idx_list;
    unordered_map<int, Node> g; // 存节点
    unordered_map<int, vector<int>> son_map; //存儿子 <fa_id,vector<son_id>>
    unordered_map<int, vector<int>> fa_map; //存父亲 <son_id,vector<fa_id>>
    unordered_map<int, pair<int, int>> mem;      // 存储内存分配情况 <id,<start_loc,width>
    map<pair<int, string>, vector<int>> dic;    // 存储域字典     <<index,dic_key>,vector<id>>
    map<string, vector<int>> ddic;  // only for test
    unordered_map<int, vector<int>> pri_map;    // 存储各node的拓扑序  <level,vector<id>>
    unordered_map<int, int> inDegree; // 记录每个节点的入度  <id,in_degree>
    queue<Node> q;
    unordered_map<int, set<int>> sset;  // to get common set
```

4. 算法流程

4.1 初始化阶段

- **读取输入文件**：解析输入文件，构建节点之间的父子关系和宽度信息。
- **确定根节点**：通过计算节点的入度，找出没有父节点的根节点。
- **初始化队列**：将根节点及其直接子节点放入队列，开始层次遍历。

4.2 内存分配阶段

- **层次遍历**：按照节点的层级进行广度优先搜索，分配内存并更新节点的层级信息。
- **优先队列**：使用宽度作为优先级，保证较宽的节点优先分配内存。
- **内存槽位计算**：根据节点宽度，确定内存起始位置，确保连续且符合特定对齐要求。

- **更新内存映射**：记录每个节点的内存占用区间。

Algorithm 1 Memory Allocation

Input: a priority queue as pq to store Nodes; a bool variable as $is\_continue$ to be a flag

// Kahn
for node i in g **do**
  for node j in son_map_i **do**
    $level_j = \max(level_j , level_i + 1)$
    $inDegree_j$ -= 1
    if $inDegree_i = 0$ **then**
      push i to Q
    end if
  end for
end for
// Memory allocation core
while $is\_continue$ = True **do**
  if pq is empty **then**
    Push next_level nodes into pq
  end if
  if $next\_level > level_{max}$ **then**
    $is\_continue$ = False
  end if
  while pq is not empty **do**
    for each node $top \in pq$ **do**
      select pt_{max} and $pt2_{max}$ from fa_map_{top}
      if $Mem_{top} = pt_{max}$ is legal **then**
        $Mem_{top} = pt_{max}$
      else
        $Mem_{top} = pt2_{max}$
      end if
      Update pt_{top} and $pt2_{top}$
    end for
  end while
end while
return Mem

4.3 域字典构建阶段

- **深度遍历**：针对每个节点，执行深度优先搜索，构建节点间的关系字典。
- **集合交集**：在遍历过程中，计算子节点内存区域的交集，用于形成域字典的键值。

- **字典记录**：将计算出的交集与节点ID关联，存储到域字典中。

Algorithm 2 Domain dictionary construction

Input: an array as is_visit to record whether a node has been visited

// warm-up
push all nodes into Q in topological order
// Domain dictionary construction core
while Q is not empty **do**
  top = dequeue(Q)
  if $is\_visit_{top} = true$ **then**
    continue
  end if
  use Algorithm 3 to decide $extern\_len_{top}$
  Update s based on $extern\_len_{top}$
  use Algorithm 4 to construct domain dictionary Dic
end while
return Dic

Algorithm 3 Extern_len Decision

Input: a node as top; a set as $sset$ to store the maximum contribution length of the current node

Merge legality check of top
if legal **then**
  Update $sset_{top}$
  return $sset_{top}$
else
  for each node $i \in son\_map_{top}$ **do**
    use Algorithm 3 to decide $extern\_len_i$
  end for
  get common set from children's $sset$
  Update $sset_{top}$
  return $sset_{top}^{max}$
end if

Algorithm 4 Construction core

Input: a node as top; an int as now_len to denote used length; an int as $extern\_len$ to denote maximum length; a string as s to denote dictionary key; an array as is_visit record whether a node has been visited

Find the max index $idx_{max}$from fa_map_{top}
if s is existed in Dic **and** $idx_s >= idx_{max}$ **then**
  put top into existed Dic_s
else
  put top into newly Dic_s
end if
if $now\_len + width_{top}$ ¡ $extern\_len$ **then**
  for each node $iinson\_map_{top}$ **do**
    use Algorithm 4 to construct domain dictionary Dic
  end for
end if

5. 输出处理

- **输出内存分配信息**：将每个节点的内存分配信息输出至CSV文件，包括节点ID、起始位置和长度。
- **输出域字典信息**：将构建的域字典信息输出至另一个CSV文件，便于后续查询使用。

6. 关键函数说明

- **init(path)**：从指定路径读取输入文件，初始化节点信息。
- **get_root()**：确定根节点并初始化层次遍历。
- **mem_alloc()**：执行内存分配，采用宽度优先策略。
- **get_output1(path1)**：输出节点内存分配详情。
- **get_output2(path2)**：输出域字典信息。
- **dfs(Node top)**：深度优先搜索，用于计算当前节点的最大可贡献长度。
- **ddfs(Node top, int now_len, ...)**：辅助函数，根据最大可贡献长度构建域字典。

9. 测试与验证

- 本算法经过测试于赛道一获得**1516**分，赛道二获得**9435**分，分别位列第**6**名和第**9**名，取得了不错的成果。

10. 不足与后续

- 本算法在内存分配阶段倾向于尽可能的复用后半段内存即4B容器，一方面是因为4B是复用效率最高的容器，另一方面便于后续域字典构建时的优化；然而，当数据量足够大的时候，可能会导致4B容器不够用而崩溃，尽管当前测试数据并没有那么大的数据量，后续优化方案如下：
    - 通过检测最长链路长度是否大于4B容器总容量来决定是否全部放在4B容器中，若足够装下则使用本算法，若不足够装下则按照层次分别使用4B，2B，1B容器。
    - 本算法已经提前留好了2B容器和1B容器的相关指针与接口，迫于时间压力只实现了当前算法，后续会继续跟进。
- 本算法在域字典构建阶段只会将通“独生节点”与其父亲进行合并，并没有考虑多父亲节点的情况。事实上，在多父亲合并上也有尝试，但是迫于时间压力，一方面没来得及解决节点“不重不漏”的情况（已验证会有bug），另一方面即便是这个有bug的版本依旧可以跑分（已实现）并且分数和当前算法相同（确

认是不同版本的跑分），可能是第一阶段内存分配的影响或是测试数据的问题。【当前提交的最终版是无bug版本,尽管有bug版本逻辑更完善并且测试数据并没有测试出bug】