基于原论文给出的伪代码实现自然语言依赖解析

小组成员及分工

姓名	学号	工作
冯浩	21307155	共同完成论文阅读、读书报告、代码、实验报告（报告多一点）
刘俊杰	21307174	共同完成论文阅读、读书报告、代码、实验报告（代码多一点）

对于依赖解析以及集合演算模型的认识

依赖解析

依赖解析是自然语言处理中的重要技术，用于理解句子中的语法结构及词汇之间的关系。通过解析句子中的每个词，确定词汇之间的依赖关系，从而构建出一个依赖树。每个词作为一个节点，依赖关系则是节点之间的有向边。
在论文中提到的依赖解析模型中，词汇区（LEX）和其他大脑区域（如主语区、动词区）是解析过程的核心。每个词汇在词汇区有一个固定的神经元集合，称为 xw。这些集合的放电会执行特定的短程序，称为词汇的动作（αw），其通过去抑制和抑制命令影响其他区域的神经元活动。解析过程中，系统依次处理每个词汇，通过激活其对应的集合并执行其预指令和后指令，构建词汇之间的依赖关系。
解析过程主要分为以下步骤：

1. 初始化：激活词汇区（LEX）、主语区（SUBJ）和动词区（VERB）。
2. 逐词处理：遍历句子中的每个词汇，激活其在词汇区的集合，并执行其预指令。
3. 投射操作：在预指令执行完毕后，进行投射操作，将当前激活的集合投射到其他区域。
4. 后指令执行：投射操作后，执行该词汇的后指令，进一步调整神经元活动。
5. 依赖关系提取：在解析完成后，通过读取解析树，提取词汇之间的依赖关系。

集合演算

集合演算（Assembly Calculus，简称 AC）是由 Papadimitriou 等人 于 2020 年提出的一种计算模型，旨在通过模拟大脑中神经元和突触的活动 来执行认知功能。AC 的核心思想是“集合”（Assembly），它是由一组高度 连接的兴奋性神经元组成的集合，这些神经元通过突触相互连接，并在同一 大脑区域内共同工作。
AC 描述了一个动态系统，系统中有被广泛证实的以下部分和属性：
a）脑区域间存在随机连接的神经元；
b）神经元输入的简单线性模型；
c）每个区域内的抑制，使得前 k 个激活最强的神经元被激发；
d）一种简单的 Hebbian 可塑性模型，即随着神经元的激发，突触的强度增加。
这个动态系统的更新是通过一系列精确定义的步骤完成的，每个步骤都基于前一个时间步的状态来计算下一个时间步的状态。除了神经元的自然激活和突触权重的调整外，还可以通过高级命令来控制特定神经元或纤维的活动。
集合是动态系统中的一个关键新兴属性，它由一组特殊的 k 个神经元组成，这些神经元位于同一大脑区域内，并且以密集的方式相互连接。而投影操作允许集合在不同大脑区域之间传递其活动。

项目设计

数据对象

Area类：表示大脑中的一个区域，设置以下属性和方法：

1. 类成员变量：
   name: 区域的名称，作为识别标签。
   n: 该区域中的神经元数量。
   k: 该区域激活时触发的神经元数量。
   beta: 激活参数的默认值。
   beta_by_stimulus: 从刺激名称到对应 beta 值的映射。
   beta_by_area: 从区域名称到对应 beta 值的映射。
   w: 曾经在该区域中触发的神经元数量。
   saved_w: 每轮支持大小的列表。
   winners: winner列表，由先前的动作构成的集合，表示的是自上次调用 `.project()`以来的值。
   saved_winners: 所有winners的列表，每轮一个子列表。
   num_first_winners: 首次保存的winners的数量。
   fixed_assembly: 表示该区域的winners集合是否被冻结。
   explicit: 表示该区域是否完全模拟。
2. 类函数 构造函数 **init**: 初始化区域对象。
   _update_winners: 更新赢家集合。
   update_beta_by_stimulus: 更新特定刺激的 beta 值。
   update_area_beta: 更新特定区域的 beta 值。
   fix_assembly: 固定赢家集合。
   unfix_assembly: 取消固定赢家集合。
   get_num_ever_fired: 获取该区域中曾经激活的神经元数量。

```
class Area {
public:
    // 默认构造函数：初始化所有成员变量
    Area() : name(""), n(0), k(0), beta(0.05), w(0), _new_w(0),
num_first_winners(-1), fixed_assembly(false), explicitArea(false),
num_ever_fired(0) {}

    // 参数化构造函数：初始化成员变量，并根据参数进行赋值
    Area(const std::string& name, int n, int k, double beta = 0.05, int w = 0,
bool explicitArea = false)
        : name(name), n(n), k(k), beta(beta), w(w), _new_w(0),
num_first_winners(-1), fixed_assembly(false), explicitArea(explicitArea),
num_ever_fired(0) {}

    // 更新赢家 (winners) 的函数
    void update_winners() {
        winners = _new_winners; // 更新赢家列表
        if (!explicitArea) { // 如果不是显式区域
            w = _new_w;
        }
    }

    // 根据刺激更新 beta 值的函数
```

```
    void update_beta_by_stimulus(const std::string& name, double new_beta) {
        beta_by_stimulus[name] = new_beta; // 更新刺激的 beta 值
    }

    // 更新区域 beta 值的函数
    void update_area_beta(const std::string& name, double new_beta) {
        beta_by_area[name] = new_beta; // 更新区域的 beta 值
    }

    // 固定集合状态的函数
    void fix_assembly() {
        if (winners.empty()) { // 如果赢家列表为空
            throw std::runtime_error("Area " + name + " does not have assembly;
cannot fix."); // 抛出异常
        }
        fixed_assembly = true; // 固定集合状态
    }

    // 取消固定集合状态的函数
    void unfix_assembly() {
        fixed_assembly = false; // 取消固定集合状态
    }

    // 获取曾经激活过的神经元数量的函数
    int getNumEverFired() const {
        if (explicitArea) {
            return num_ever_fired;
        } else {
            return w;
        }
    }

    // 成员变量
    std::string name; // 区域名称
    int n; // 总神经元数
    int k; // 激活神经元数
    double beta;
    std::unordered_map<std::string, double> beta_by_stimulus; // 刺激对应的 beta 值
映射
    std::unordered_map<std::string, double> beta_by_area; // 区域对应的 beta 值映射
    int w;
    int _new_w;
    std::vector<int> saved_w;
    std::vector<int> winners; // 当前赢家列表
    std::vector<int> _new_winners; // 更新后的赢家列表
    std::vector<std::vector<int>> saved_winners; // 保存的赢家列表
    int num_first_winners;
    bool fixed_assembly; // 集合状态是否固定
    bool explicitArea; // 是否是显式区域
    int num_ever_fired;
    std::vector<bool> ever_fired;
};
```

Brain类：包含多个神经区域、刺激信号和连接体的相关信息和操作。

1. 类成员变量
   area_by_name：用于存储每个区域的名称及其对应的 Area 对象。
   stimulus_size_by_name：用于存储每种刺激信号的名称及其size特征。
   connectomes_by_stimulus：存储每种刺激信号名称对应的区域激活向量。
   connectomes：存储从源区域到目标区域的连接信息。
   p：神经元之间的连接概率。
   save_size 和 save_winners：用于控制是否保存区域大小和winners信息的布尔变量。
   disable_plasticity：控制是否禁用可塑性的布尔变量。
   _rng：随机数生成器。
   _uniform_dist：均匀分布的随机数生成器。
   _use_normal_ppf：是否使用正态分布的布尔变量。

```
// 存储不同区域的名称和对应的 Area 对象
    std::unordered_map<std::string, Area> area_by_name;

    // 存储每个刺激名称及其大小
    std::unordered_map<std::string, int> stimulus_size_by_name;

    // 存储每个刺激对应的激活向量，按区域名称映射
    std::unordered_map<std::string, std::unordered_map<std::string,
std::vector<float>>> connectomes_by_stimulus;

    // 存储区域之间的连接，从源区域名称到目标区域名称到连接对象的映射
    std::unordered_map<std::string, std::unordered_map<std::string, Connectome>>
connectomes;

    // 连接概率
    double p;
    bool save_size;
    bool save_winners;
    bool disable_plasticity;
    std::mt19937 _rng;
    std::uniform_real_distribution<float> _uniform_dist;
    bool _use_normal_ppf;
```

2. 类成员函数： Brain()：初始化大脑对象，包括设置概率参数、是否保存区域大小和winner信息等。

```
Brain(double p, bool save_size = true, bool save_winners = false, int seed = 0)
        : p(p), save_size(save_size), save_winners(save_winners),
disable_plasticity(false), _rng(seed), _use_normal_ppf(false) {}

    void add_area(const std::string& area_name, int n, int k, float beta) {
    // 向 area_by_name 添加新的区域
    area_by_name[area_name] = Area(area_name, n, k, beta);
    // 创建一个新的 connectomes 映射用于存储连接信息
    std::unordered_map<std::string, Connectome> new_connectomes;
```

```
    // 遍历所有已存在的区域
    for (auto& kv : area_by_name) {
        const std::string& other_area_name = kv.first;
        // 获取当前遍历区域的大小（如果是显式区域）
        int other_area_size = area_by_name[other_area_name].explicitArea ?
area_by_name[other_area_name].n : 0;
        // 初始化 new_connectomes 中当前遍历区域的连接矩阵大小
        new_connectomes[other_area_name].col = other_area_size;
        new_connectomes[other_area_name].row = 0;
        // 如果当前遍历区域不是新添加的区域
        if (other_area_name != area_name) {
            // 初始化 connectomes 映射中从当前遍历区域到新添加区域的连接矩阵大小
            connectomes[other_area_name][area_name].con.resize(other_area_size,
std::vector<float>(0));
            connectomes[other_area_name][area_name].row = other_area_size;
            connectomes[other_area_name][area_name].col = 0;
        }
        // 设置当前遍历区域到新添加区域的 beta 值
        area_by_name[other_area_name].beta_by_area[area_name] =
area_by_name[other_area_name].beta;
        area_by_name[area_name].beta_by_area[other_area_name] = beta;
    }
    // 将新的 connectomes 映射添加到 connectomes 中
    connectomes[area_name] = new_connectomes;
    }
```

add_explicit_area：添加明确指定的区域，可以设置自定义的连接概率。

```
void add_explicit_area(const std::string& area_name,
                       int n, int k, float beta,
                       float custom_inner_p = -1,
                       float custom_out_p = -1,
                       float custom_in_p = -1) {
    // 创建一个显式区域，并将其添加到 area_by_name 映射中
    area_by_name[area_name] = Area(area_name, n, k, beta, n, true);
    // 初始化显式区域的 ever_fired 和 num_ever_fired 属性
    area_by_name[area_name].ever_fired = std::vector<bool>(n, false);
    area_by_name[area_name].num_ever_fired = 0;

    // 根据参数或默认值设置 inner_p、in_p 和 out_p
    float inner_p = (custom_inner_p != -1) ? custom_inner_p : p;
    float in_p = (custom_in_p != -1) ? custom_in_p : p;
    float out_p = (custom_out_p != -1) ? custom_out_p : p;
    // 创建一个新的 connectomes 映射，用于存储连接信息
    std::unordered_map<std::string, Connectome> new_connectomes;
    // 遍历所有已存在的区域
    for (auto& kv : area_by_name) {
        const std::string& other_area_name = kv.first;
        Area& other_area = kv.second;
        // 如果是当前新添加的区域
        if (other_area_name == area_name) {
            // 初始化新添加区域到自身的连接矩阵
```

```
            new_connectomes[other_area_name].con = std::vector<std::vector<float>>
(n, std::vector<float>(n));
            new_connectomes[other_area_name].col = n;
            new_connectomes[other_area_name].row = n;
            // 使用二项分布填充连接矩阵
            std::binomial_distribution<int> distribution(1, inner_p);
            for (int i = 0; i < n; ++i) {
                for (int j = 0; j < n; ++j) {
                    new_connectomes[other_area_name].con[i][j] =
distribution(_rng);
                }
            }
        }
        else {
            // 如果当前遍历的区域也是显式区域
            if (other_area.explicitArea) {
                int other_n = other_area.n;
                // 初始化新添加区域到当前遍历区域的连接矩阵
                new_connectomes[other_area_name].con =
std::vector<std::vector<float>>(n, std::vector<float>(other_n));
                new_connectomes[other_area_name].row = n;
                new_connectomes[other_area_name].col = other_n;
                // 使用二项分布填充连接矩阵
                std::binomial_distribution<int> distribution(1, out_p);
                for (int i = 0; i < n; ++i) {
                    for (int j = 0; j < other_n; ++j) {
                        new_connectomes[other_area_name].con[i][j] =
distribution(_rng);
                    }
                }
                // 初始化当前遍历区域到新添加区域的连接矩阵
                connectomes[other_area_name][area_name].con =
std::vector<std::vector<float>>(other_n, std::vector<float>(n));
                distribution = std::binomial_distribution<int>(1, in_p);
                for (int i = 0; i < other_n; ++i) {
                    for (int j = 0; j < n; ++j) {
                        connectomes[other_area_name][area_name].con[i][j] =
distribution(_rng);
                    }
                }
            }
            else {
                // 如果当前遍历的区域不是显式区域，清空其连接信息
                new_connectomes[other_area_name].con.clear();
                connectomes[other_area_name][area_name].con.clear();
            }
        }
        // 设置其他区域到新添加区域的 beta 值
        other_area.beta_by_area[area_name] = other_area.beta;
        // 设置新添加区域到其他区域的 beta 值
        area_by_name[area_name].beta_by_area[other_area_name] = beta;
    }
    // 将新的 connectomes 映射添加到 connectomes 中
    connectomes[area_name] = new_connectomes;
```

```
    }
```

update_plasticity：更新从一个区域到另一个区域的可塑性参数。

```
void update_plasticity(const std::string& from_area, const std::string& to_area,
double new_beta) {
    area_by_name[to_area].beta_by_area[from_area] = new_beta;
}
```

update_plasticities：批量更新区域之间和刺激信号到区域的可塑性参数，调用上一个更新函数。

```
void update_plasticities(const std::unordered_map<std::string,
std::vector<std::pair<std::string, float>>>& area_update_map = {}) {
    // Update plasticities from area to area
    for (const auto& kv : area_update_map) {
        const std::string& to_area = kv.first;
        for (const auto& update_rule : kv.second) {
            const std::string& from_area = update_rule.first;
            double new_beta = update_rule.second;
            update_plasticity(from_area, to_area, new_beta);
        }
        }
}
```

activate：激活指定区域的某个索引范围内的神经元。

```
void activate(const std::string& area_name, int index) {
    // 激活指定区域的一个特定索引处的神经元集合（一个集合）
    Area& area = area_by_name[area_name];
    int k = area.k;
    int assembly_start = k * index;
    area.winners.clear();
    for (int i = assembly_start; i < assembly_start + k; ++i) {
        area.winners.push_back(i); // 将集合中的神经元添加到获胜神经元列表中
    }
    area.fix_assembly();
}
```

project：将输入的刺激信号和其他区域的活动投射到目标区域，并更新目标区域的状态。

```
void project(const std::unordered_map<std::string, std::vector<std::string>>&
areas_by_stim,
```

```
        const std::unordered_map<std::string, std::vector<std::string>>&
dst_areas_by_src_area)
    {
        // area_in: 映射，从源区域到投影到它的目标区域的列表
        std::unordered_map<std::string, std::vector<std::string>> area_in;

        for (auto& pair : dst_areas_by_src_area) {
            std::string from_area_name = pair.first;
            vector<string> tmp =  pair.second;
            if (area_by_name.find(from_area_name) == area_by_name.end()) {
                throw std::invalid_argument(from_area_name + " not in
brain.area_by_name");
            }
            for (std::string& to_area_name : tmp) {
                if (area_by_name.find(to_area_name) == area_by_name.end()) {
                    throw std::invalid_argument("Not in brain.area_by_name: " +
to_area_name);
                }
            area_in[to_area_name].push_back(from_area_name);
            }
        }
        // 需要更新的目标区域名称集合
        std::set<std::string> to_update_area_names;
        for (const auto& pair : area_in) {
            to_update_area_names.insert(pair.first);
        }
         // 第一次遍历：将源区域投影到目标区域，并更新第一组获胜者
        for (auto& area_name : to_update_area_names) {
            Area& area = area_by_name[area_name];
            int num_first_winners = project_into(area, area_in[area_name]);
            area.num_first_winners = num_first_winners;
            if (save_winners) {
                area.saved_winners.push_back(area._new_winners);
            }
        }
         // 第二次遍历：更新获胜者并保存区域大小
        for (auto& area_name : to_update_area_names) {
            Area& area = area_by_name[area_name];
            area.update_winners();
            if (save_size) {
                area.saved_w.push_back(area.w);
            }
        }
    }
```

project_into：执行投射操作，包括从刺激信号和其他区域计算输入，并确定新的获胜者。

```
int project_into(Area& target_area, const vector<string>& from_areas) {

        int num_inputs_processed;
        int processed_input_count;
```

```
        // If projecting from area with no assembly, throw an error.
        for (const auto& from_area_name : from_areas) {
            auto& from_area = area_by_name[from_area_name];
            if (from_area.winners.empty() || from_area.w == 0) {
                throw std::runtime_error("Projecting from area with no assembly: "
+ from_area_name);
            }
        }

        vector<vector<int>> inputs_by_first_winner_index;
        string target_area_name = target_area.name;

        if (target_area.fixed_assembly) {
            target_area._new_winners = target_area.winners;
            target_area._new_w = target_area.w;
            num_inputs_processed = 0;
        }
        else {

            vector<int> cumulative_winners_per_source_area;
            int total_source_areas;
            int total_winners;
            vector<float> previous_winners_input(target_area.w, 0.0f);
            vector<float> all_potential_winner_inputs;


             for (const auto& from_area_name : from_areas) {
                auto& connectome = connectomes[from_area_name]
[target_area_name].con;
                auto& from_area = area_by_name[from_area_name];
                for (int w : from_area.winners) {
                    for (size_t i = 0; i < target_area.w; ++i) {
                        previous_winners_input[i] += connectome[w][i];
                    }
                }
            }

            if (!target_area.explicitArea) {
                float effective_neurons, quantile_value, alpha_value, mean,
stddev, lower_bound, upper_bound;
                vector<float> potential_new_winner_inputs(target_area.k);
                total_source_areas = 0;
                total_winners = 0;
                cumulative_winners_per_source_area.push_back(total_winners);
                for (const auto& from_area_name : from_areas) {
                    int active_winners =
area_by_name[from_area_name].winners.size(); // effective_k
                    total_source_areas += 1;
                    total_winners += active_winners;
                    cumulative_winners_per_source_area.push_back(total_winners);
                }

                effective_neurons = target_area.n - target_area.w;
                if (effective_neurons <= target_area.k) {
```

```
                    cout << "Remaining size of area " << target_area_name << " too
small to sample k new winners." << endl;
                    return -1;
                }

                quantile_value = (effective_neurons - target_area.k) /
effective_neurons;
                alpha_value = BinomQuantile(total_winners, p, quantile_value);
                mean = total_winners * p;
                stddev = sqrt(total_winners * p * (1.0 - p));
                lower_bound = (alpha_value - mean) / stddev;

                for (auto& input : potential_new_winner_inputs)
                    input = min<float>(total_winners,
round(TruncatedNorm(lower_bound, _rng) * stddev + mean));

                all_potential_winner_inputs.resize(previous_winners_input.size() +
potential_new_winner_inputs.size());
                copy(previous_winners_input.begin(), previous_winners_input.end(),
all_potential_winner_inputs.begin());
                copy(potential_new_winner_inputs.begin(),
potential_new_winner_inputs.end(), all_potential_winner_inputs.begin() +
previous_winners_input.size());
            }
            else {
                all_potential_winner_inputs = previous_winners_input;
            }

            std::vector<int> new_winner_indices =
getNLargestIndices(all_potential_winner_inputs, target_area.k);
            vector<float> initial_winner_inputs;
            num_inputs_processed = 0;
            if (!target_area.explicitArea) {
                int new_winner_indices_len = new_winner_indices.size();
                for (int i = 0; i < new_winner_indices_len; i++) {
                    if (new_winner_indices[i] >= target_area.w) {

initial_winner_inputs.push_back(all_potential_winner_inputs[new_winner_indices[i]]
);
                        new_winner_indices[i] = target_area.w +
num_inputs_processed;
                        ++num_inputs_processed;
                    }
                }
            }

            target_area._new_winners = new_winner_indices;
            target_area._new_w = target_area.w + num_inputs_processed;
            inputs_by_first_winner_index = vector<vector<int>>
(num_inputs_processed, vector<int>());

            for (auto i = 0; i < num_inputs_processed; i++) {
                vector<int> input_indices = uniqueRandomChoices(total_winners,
int(initial_winner_inputs[i]),_rng);
```

```
                vector<int> connections_per_source_area(total_source_areas, 0);
                for (auto j = 0; j < total_source_areas; j++) {
                    for (const auto& winner : input_indices) {
                        if (cumulative_winners_per_source_area[j + 1] > winner &&
winner >= cumulative_winners_per_source_area[j])
                            connections_per_source_area[j] += 1;
                    }
                }
                inputs_by_first_winner_index[i] = connections_per_source_area;
            }
        }

        processed_input_count = 0;
        for(auto&from_area_name : from_areas) {
            int&  from_area_w = area_by_name[from_area_name].w;
            vector<int>& from_area_winners = area_by_name[from_area_name].winners;
            Connectome& the_connectomes = connectomes[from_area_name]
[target_area_name];

            the_connectomes.colpadding(num_inputs_processed, 0);

            for (int i = 0; i < num_inputs_processed; ++i) {
                int total_in = inputs_by_first_winner_index[i]
[processed_input_count];
                vector<int>sample_indices = random_choice(
total_in,from_area_winners);
                for(auto& j : sample_indices)
                    the_connectomes.con[j][target_area.w + i] = 1.0;

                for (int j = 0; j <  from_area_w; ++j)
                    if (find(from_area_winners.begin(), from_area_winners.end(),
j) == from_area_winners.end())
                        the_connectomes.con[j][target_area.w + i] =
std::binomial_distribution<int>(1, p)(_rng);
            }

            float area_to_area_beta = disable_plasticity ? 0 :
target_area.beta_by_area[from_area_name]; // area_to_area_beta

            for (const auto& i : target_area._new_winners)
                for (const auto& j : from_area_winners)
                    the_connectomes.con[j][i] *= 1.0 + area_to_area_beta;
            processed_input_count++;
        }

        for (auto& kv : area_by_name) {
            const std::string& other_area_name = kv.first;
            Area& other_area = kv.second;
            if (find(from_areas.begin(), from_areas.end(), other_area_name) ==
from_areas.end()) {
                connectomes[other_area_name]
[target_area_name].colpadding(num_inputs_processed, 0);
                for (int i = 0; i < connectomes[other_area_name]
[target_area_name].row; ++i) {
```

```
                    for (int j = target_area.w; j < target_area._new_w; ++j) {
                        connectomes[other_area_name][target_area_name].con[i][j] =
std::binomial_distribution<int>(1, p)(_rng);
                    }
                }
            }

            connectomes[target_area_name]
[other_area_name].rowpadding(num_inputs_processed, 0);
            for (int i = target_area.w; i < target_area._new_w; ++i) {
                for (int j = 0; j < connectomes[target_area_name]
[other_area_name].col; ++j) {
                    connectomes[target_area_name][other_area_name].con[i][j] =
std::binomial_distribution<int>(1, p)(_rng);
                }
            }
        }
        return num_inputs_processed;
    }
```

Rule和Rules类

1. Rule类成员变量包括索引（index）、规则名称（rule_name）、动作（action）、区域（area）、区域1（area1）和区域2（area2）。
2. Rules类成员变量包括索引（index）、前置规则集合（PRE_RULES）、后置规则集合（POST_RULES）。除了构造函数外还有函数add_PreRule 和 add_PostRule 分别用于向前置规则集合和后置规则集合中添加规则。

```
class Rule {
public:
    Rule(int index, std::string rule_name, std::string action, std::string area,
         std::string area1, std::string area2);
    int index;
    std::string rule_name;
    std::string action;
    std::string area;
    std::string area1;
    std::string area2;
};

// Rule 类构造函数
Rule::Rule(int index, std::string rule_name, std::string action, std::string area,
           std::string area1, std::string area2)
    : index(index), rule_name(rule_name), action(action),
      area(area), area1(area1), area2(area2) {}

// Rules 类
class Rules {
public:
    Rules(int index);
    Rules(){};
```

```
    Rules(int index, std::vector<Rule> PRE_RULES, std::vector<Rule> POST_RULES)
        : index(index), PRE_RULES(PRE_RULES), POST_RULES(POST_RULES) {};
    void add_PreRule(int index, std::string rule_name, std::string action,
std::string area,
                     std::string area1 = "", std::string area2 = "");
    void add_PostRule(int index, std::string rule_name, std::string action,
std::string area,
                      std::string area1 = "", std::string area2 = "");
    int index;
    std::vector<Rule> PRE_RULES;
    std::vector<Rule> POST_RULES;
};
```

对于Rules有一些规则生成函数:
generic_noun, generic_trans_verb, generic_intrans_verb, generic_copula, generic_adverb, generic_determinant, generic_adjective, generic_preposition 分别生成名词、及物动词、不及物动词、连词、副词、冠词、形容词和介词的通用规则集合。
这些函数根据传入的索引参数生成相应的规则集合对象，并调用 add_PreRule 和 add_PostRule 方法添加规则到集合中。

```
// 创建针对不同词性的通用规则的函数
Rules generic_noun(int index) {
    Rules noun_rules(index);

    noun_rules.add_PreRule(0, "FiberRule", DISINHIBIT, "", LEX, SUBJ);
    noun_rules.add_PreRule(0, "FiberRule", DISINHIBIT, "", LEX, OBJ);
    noun_rules.add_PreRule(0, "FiberRule", DISINHIBIT, "", LEX, PREP_P);
    noun_rules.add_PreRule(0, "FiberRule", DISINHIBIT, "", DET, SUBJ);
    noun_rules.add_PreRule(0, "FiberRule", DISINHIBIT, "", DET, OBJ);
    noun_rules.add_PreRule(0, "FiberRule", DISINHIBIT, "", DET, PREP_P);
    noun_rules.add_PreRule(0, "FiberRule", DISINHIBIT, "", ADJ, SUBJ);
    noun_rules.add_PreRule(0, "FiberRule", DISINHIBIT, "", ADJ, OBJ);
    noun_rules.add_PreRule(0, "FiberRule", DISINHIBIT, "", ADJ, PREP_P);
    noun_rules.add_PreRule(0, "FiberRule", DISINHIBIT, "", VERB, OBJ);
    noun_rules.add_PreRule(0, "FiberRule", DISINHIBIT, "", PREP_P, PREP);
    noun_rules.add_PreRule(0, "FiberRule", DISINHIBIT, "", PREP_P, SUBJ);
    noun_rules.add_PreRule(0, "FiberRule", DISINHIBIT, "", PREP_P, OBJ);

    noun_rules.add_PostRule(0, "AreaRule", INHIBIT, DET, "", "");
    noun_rules.add_PostRule(0, "AreaRule", INHIBIT, ADJ, "", "");
    noun_rules.add_PostRule(0, "AreaRule", INHIBIT, PREP_P, "", "");
    noun_rules.add_PostRule(0, "AreaRule", INHIBIT, PREP, "", "");
    noun_rules.add_PostRule(0, "FiberRule", INHIBIT, "", LEX, SUBJ);
    noun_rules.add_PostRule(0, "FiberRule", INHIBIT, "", LEX, OBJ);
    noun_rules.add_PostRule(0, "FiberRule", INHIBIT, "", LEX, PREP_P);
    noun_rules.add_PostRule(0, "FiberRule", INHIBIT, "", ADJ, SUBJ);
    noun_rules.add_PostRule(0, "FiberRule", INHIBIT, "", ADJ, OBJ);
    noun_rules.add_PostRule(0, "FiberRule", INHIBIT, "", ADJ, PREP_P);
    noun_rules.add_PostRule(0, "FiberRule", INHIBIT, "", DET, SUBJ);
    noun_rules.add_PostRule(0, "FiberRule", INHIBIT, "", DET, OBJ);
    noun_rules.add_PostRule(0, "FiberRule", INHIBIT, "", DET, PREP_P);
```

```
    noun_rules.add_PostRule(0, "FiberRule", INHIBIT, "", VERB, OBJ);
    noun_rules.add_PostRule(0, "FiberRule", INHIBIT, "", PREP_P, PREP);
    noun_rules.add_PostRule(0, "FiberRule", INHIBIT, "", PREP_P, VERB);
    noun_rules.add_PostRule(1, "FiberRule", DISINHIBIT, "", LEX, SUBJ);
    noun_rules.add_PostRule(1, "FiberRule", DISINHIBIT, "", LEX, OBJ);
    noun_rules.add_PostRule(1, "FiberRule", DISINHIBIT, "", DET, SUBJ);
    noun_rules.add_PostRule(1, "FiberRule", DISINHIBIT, "", DET, OBJ);
    noun_rules.add_PostRule(1, "FiberRule", DISINHIBIT, "", ADJ, SUBJ);
    noun_rules.add_PostRule(1, "FiberRule", DISINHIBIT, "", ADJ, OBJ);
    noun_rules.add_PostRule(0, "FiberRule", INHIBIT, "", PREP_P, SUBJ);
    noun_rules.add_PostRule(0, "FiberRule", INHIBIT, "", PREP_P, OBJ);
    noun_rules.add_PostRule(0, "FiberRule", INHIBIT, "", VERB, ADJ);

    return noun_rules;
}

Rules generic_trans_verb(int index){
    Rules verb_rules(index);

    verb_rules.add_PreRule(0, "FiberRule", DISINHIBIT, "", LEX, VERB);
    verb_rules.add_PreRule(0, "FiberRule", DISINHIBIT, "", VERB, SUBJ);
    verb_rules.add_PreRule(0, "FiberRule", DISINHIBIT, "", VERB, ADVERB);
    verb_rules.add_PreRule(1, "AreaRule", "", DISINHIBIT, ADVERB);


    verb_rules.add_PostRule(0, "FiberRule", INHIBIT, "", LEX, VERB);
    verb_rules.add_PostRule(0, "AreaRule", "", DISINHIBIT, OBJ);
    verb_rules.add_PostRule(0, "AreaRule", "", INHIBIT, SUBJ);
    verb_rules.add_PostRule(0, "AreaRule", "", INHIBIT, ADVERB);
    verb_rules.add_PostRule(0, "FiberRule", DISINHIBIT, "", PREP_P, VERB);

    return verb_rules;
}

Rules generic_intrans_verb(int index){
    Rules verb_rules(index);

    verb_rules.add_PreRule(0, "FiberRule", DISINHIBIT, "", LEX, VERB);
    verb_rules.add_PreRule(0, "FiberRule", DISINHIBIT, "", VERB, SUBJ);
    verb_rules.add_PreRule(0, "FiberRule", DISINHIBIT, "", VERB, ADVERB);
    verb_rules.add_PreRule(1, "AreaRule", "", DISINHIBIT, ADVERB);


    verb_rules.add_PostRule(0, "FiberRule", INHIBIT, "", LEX, VERB);
    verb_rules.add_PostRule(0, "AreaRule", "", INHIBIT, SUBJ);
    verb_rules.add_PostRule(0, "AreaRule", "", INHIBIT, ADVERB);
    verb_rules.add_PostRule(0, "FiberRule", DISINHIBIT, "", PREP_P, VERB);

    return verb_rules;
}

Rules generic_copula(int index){
    Rules copula_rules(index);
```

```
    copula_rules.add_PreRule(0, "FiberRule", DISINHIBIT, "", LEX, VERB);
    copula_rules.add_PreRule(0, "FiberRule", DISINHIBIT, "", VERB, SUBJ);

    copula_rules.add_PostRule(0, "FiberRule", INHIBIT, "", LEX, VERB);
    copula_rules.add_PostRule(0, "AreaRule", "", INHIBIT, OBJ);
    copula_rules.add_PostRule(0, "AreaRule", "", INHIBIT, SUBJ);
    copula_rules.add_PostRule(0, "FiberRule", DISINHIBIT, "", ADJ, VERB);

    return copula_rules;
}

Rules generic_adverb(int index){
    Rules adverb_rules(index);

    adverb_rules.add_PreRule(0, "AreaRule", DISINHIBIT, ADVERB);
    adverb_rules.add_PreRule(0, "FiberRule", DISINHIBIT, "", LEX,ADVERB);

    adverb_rules.add_PostRule(0, "FiberRule", INHIBIT, "", LEX, ADVERB);
    adverb_rules.add_PostRule(1, "AreaRule", "", INHIBIT, ADVERB);


    return adverb_rules;
}

Rules generic_determinant(int index) {
    Rules determinant_rules(index);

    determinant_rules.add_PreRule(0, "AreaRule", DISINHIBIT, DET);
    determinant_rules.add_PreRule(0, "FiberRule", DISINHIBIT, "", LEX, DET);

    determinant_rules.add_PostRule(0, "FiberRule", INHIBIT, "", LEX, DET);
    determinant_rules.add_PostRule(0, "FiberRule", INHIBIT, "", VERB, ADJ);

    return determinant_rules;
}

Rules generic_adjective(int index) {
    Rules adjective_rules(index);

    adjective_rules.add_PreRule(0, "AreaRule", DISINHIBIT, ADJ);
    adjective_rules.add_PreRule(0, "FiberRule", DISINHIBIT, "", LEX, ADJ);

    adjective_rules.add_PostRule(0, "FiberRule", INHIBIT, "", LEX, ADJ);
    adjective_rules.add_PostRule(0, "FiberRule", INHIBIT, "", VERB, ADJ);

    return adjective_rules;
}

Rules generic_preposition(int index) {
    Rules preposition_rules(index);

    preposition_rules.add_PreRule(0, "AreaRule", DISINHIBIT, PREP);
    preposition_rules.add_PreRule(0, "FiberRule", DISINHIBIT, "", LEX, PREP);
```

```
    preposition_rules.add_PostRule(0, "FiberRule", INHIBIT, "", LEX, PREP);
    preposition_rules.add_PostRule(0, "AreaRule", "", DISINHIBIT, PREP_P);
    preposition_rules.add_PostRule(1, "FiberRule", INHIBIT, "", LEX, SUBJ);
    preposition_rules.add_PostRule(1, "FiberRule", INHIBIT, "", LEX, OBJ);
    preposition_rules.add_PostRule(1, "FiberRule", INHIBIT, "", DET, SUBJ);
    preposition_rules.add_PostRule(1, "FiberRule", INHIBIT, "", DET, OBJ);
    preposition_rules.add_PostRule(1, "FiberRule", INHIBIT, "", ADJ, SUBJ);
    preposition_rules.add_PostRule(1, "FiberRule", INHIBIT, "", ADJ, OBJ);

    return preposition_rules;
}
```

ParserBrain类

ParserBrain 类继承自 Brain 类，并扩展了其功能，增加了对词素、区域状态、纤维状态和规则的管理和操作能力。通过类里面的数据结构和函数，可以实现对神经网络模型中不同区域和连接的详细管理和解析。

1. 类中的成员变量：
   lexeme_dict：存储词素（词的基本单位）的名称及其对应的规则。
   all_areas：存储所有区域的名称。
   recurrent_areas：存储所有重复（循环）区域的名称。
   initial_areas：存储所有初始区域的名称。
   readout_rules：存储每个区域的读取规则。
   fiber_states：存储每对区域之间的纤维连接状态（激活或抑制）。
   area_states：存储每个区域的状态（激活或抑制）。
   activated_fibers：存储每个区域激活的纤维。

```
std::unordered_map<std::string, Rules> lexeme_dict;
std::vector<std::string> all_areas;
std::vector<std::string> recurrent_areas;
std::vector<std::string> initial_areas;
std::unordered_map<std::string, vector<std::string>> readout_rules;

std::unordered_map<std::string, std::unordered_map<std::string, std::set<int>>>
fiber_states;
std::unordered_map<std::string, std::set<int>> area_states;
std::unordered_map<std::string, std::set<std::string>> activated_fibers;
```

2. 构造函数和成员函数：
   （1） 构造函数初始化 ParserBrain 对象，并调用 initialize_states 函数来初始化状态。

```
ParserBrain(double p,
                std::unordered_map<std::string, Rules> lexeme_dict = {},
                std::vector<std::string> all_areas = {},
                std::vector<std::string> recurrent_areas = {},
                std::vector<std::string> initial_areas = {},
                std::unordered_map<std::string, vector<std::string>> readout_rules
= {})
```

```
        : Brain(p),
          lexeme_dict(lexeme_dict),
          all_areas(all_areas),
          recurrent_areas(recurrent_areas),
          initial_areas(initial_areas),
          readout_rules(readout_rules) {
        initialize_states();
    }
```

（2） initialize_states函数为每对区域初始化纤维状态，并为每个区域初始化其状态，用于初始化 fiber_states 和 area_states。

```
// 初始化状态
void initialize_states() {
    for (const auto& from_area : all_areas) {
        //fiber_states[from_area] = unordered_map<string, set<int>>();
        for (const auto& to_area : all_areas) {
            fiber_states[from_area][to_area].insert(0);
        }
    }
    for (const auto& area : all_areas) {
        area_states[area].insert(0);
    }
    for (const auto& area : initial_areas) {
        area_states[area].erase(0);
    }
}
```

（3） applyFiberRule函数根据规则的动作（INHIBIT 或 DISINHIBIT），更新 fiber_states，用来根据规则应用纤维状态的抑制或去抑制。

```
// 应用Fiber规则
void applyFiberRule(const Rule& rule) {
    if (rule.action == INHIBIT) {
        fiber_states[rule.area1][rule.area2].insert(rule.index);
        fiber_states[rule.area2][rule.area1].insert(rule.index);
    } else if (rule.action == DISINHIBIT) {

        fiber_states[rule.area1][rule.area2].erase(rule.index);
        fiber_states[rule.area2][rule.area1].erase(rule.index);
    }
}
```

（4） applyAreaRule函数根据规则的动作（INHIBIT 或 DISINHIBIT），更新 area_states，用来根据规则应用区域状态的抑制或去抑制。

```
// 应用Area规则
void applyAreaRule(const Rule& rule) {
    if (rule.action == INHIBIT) {
        area_states[rule.area].insert(rule.index);
    } else if (rule.action == DISINHIBIT) {
        area_states[rule.area].erase(rule.index);
    }
}
```

(5) applyRule函数根据规则类型调用 applyFiberRule 或 applyAreaRule，用于应用指定的规则。

```
// 应用规则
bool applyRule(const Rule& rule) {
    if (rule.rule_name=="FiberRule") {
        applyFiberRule(rule);
        return true;
    }
    if (rule.rule_name=="AreaRule") {
        applyAreaRule(rule);
        return true;
    }
    return false;
}
```

(6) parse_project函数获取投射映射，记住激活的纤维，并调用 project 函数进行投射。

```
void parse_project() {
    auto project_map = getProjectMap();
    remember_fibers(project_map);
    project({}, project_map);
}
```

(7) remember_fibers函数根据project_map更新 activated_fibers。

```
void remember_fibers(const unordered_map<std::string, std::vector<std::string>>&
project_map) {
    for (const auto& [from_area, to_areas] : project_map) {
        activated_fibers[from_area].insert(to_areas.begin(),to_areas.end());
    }
}
```

(8) recurrent函数检查 recurrent_areas 集合中是否包含指定区域。

```
bool recurrent(const string& area) const {
    return find(recurrent_areas.begin(),recurrent_areas.end(),area) !=
recurrent_areas.end();
}
```

（9） getProjectMap函数根据区域状态和纤维状态生成投射映射，并返回得到的投射映射。

```
// 获取投影
virtual unordered_map<std::string, std::vector<std::string>> getProjectMap() {
    unordered_map<std::string, std::vector<std::string>> proj_map;
    for (const auto& area1 : all_areas) {
        if (area_states[area1].empty()) {
            for (const auto& area2 : all_areas) {
                if (area1 == "LEX" && area2 == "LEX") continue;
                if (area_states[area2].empty()) {
                    if (fiber_states[area1][area2].empty()) {
                        if (!area_by_name[area1].winners.empty()) {
                            proj_map[area1].push_back(area2);
                        }
                        if (!area_by_name[area2].winners.empty()) {
                            proj_map[area2].push_back(area2);
                        }
                    }
                }
            }
        }
    }
    return proj_map;
}
```

（10） activateWord函数根据词素字典中的词索引和区域的大小，激活对应的神经元。用来激活指定区域中的一个词。

```
// 激活单词
void activateWord(const string& area_name, const string& word) {
    Area& area = area_by_name[area_name];
    int k = area.k;
    int assembly_start = lexeme_dict[word].index * k;
    std::vector<int> result_vector(k);
    std::iota(result_vector.begin(), result_vector.end(), assembly_start);
    area.winners = result_vector;
    area.fix_assembly();
}
```

（11） activateIndex函数根据索引计算神经元范围，并激活对应的神经元。

```
void activateIndex(const string& area_name, int index) {
    Area& area = area_by_name[area_name];
    int k = area.k;
    int assembly_start = index * k;
    std::vector<int> result_vector(k);
    std::iota(result_vector.begin(), result_vector.end(), assembly_start);
    area.winners = result_vector;
    area.fix_assembly();
}
```

（12） interpretAssemblyAsString函数调用 getWord 函数。

```
string interpretAssemblyAsString(const string& area_name) {
    return getWord(area_name, 0.7);
}
```

（13） getWord函数根据区域中的winners神经元与词素字典中的词的重叠度，返回最匹配的词，如果没有返回空字符串。

```
virtual string getWord(const string& area_name, double min_overlap = 0.7) {
    const Area& area = area_by_name.at(area_name);
    if (area.winners.empty()) {
        throw runtime_error("Cannot get word because no assembly in " +
area_name);
    }
    set<int> winners(area.winners.begin(), area.winners.end());
    int area_k = area.k;
    float threshold = min_overlap * area_k;
    for (const auto& [word, lexeme] : lexeme_dict) {
        int word_index = lexeme.index;
        set<int> word_assembly;
        for (int i = word_index * area_k; i < (word_index + 1) * area_k; ++i) {
            word_assembly.insert(i);
        }
        vector<int> intersection;
        set_intersection(winners.begin(), winners.end(), word_assembly.begin(),
word_assembly.end(), back_inserter(intersection));
        if (intersection.size() >= threshold) {
            return word;
        }
    }
    return "<NON-WORD>";
}
```

（14） getActivatedFibers函数根据 readout_rules 过滤 activated_fibers，返回满足读取规则的激活纤维，从而获取需要的激活的纤维。

```
// 获取纤维
unordered_map<string, set<string>> getActivatedFibers() {
    unordered_map<string, set<string>> pruned_activated_fibers;
    for (const auto& [from_area, to_areas] : activated_fibers) {
        for (const auto& to_area : to_areas) {
            for(int i=0;i<readout_rules[from_area].size();i++){
                if(readout_rules[from_area][i]==to_area) {
                    pruned_activated_fibers[from_area].insert(to_area);
                    break;
                    }
            }
            // 下面三行被上面三行替代
            // if (readout_rules[from_area].count(to_area)) {
            //     pruned_activated_fibers[from_area].push_back(to_area);
            // }
        }
    }
    return pruned_activated_fibers;
}
```

EnglishParserBrain类

EnglishParserBrain 类继承自 ParserBrain，扩展了其功能以适应处理英语句子的解析任务。该类初始化了多个特定的语言区域，并设置了不同的可塑性值。

1. 类中的成员变量
   verbose：控制是否要输出详细信息
2. 构造函数和其他成员函数
   （1） 构造函数EnglishParserBrain调用基类 ParserBrain 的构造函数，并初始化了多个语言相关的区域，如 LEX、SUBJ、OBJ 等。还设置了不同区域之间的可塑性值。
   （2） getProjectMap该函数重载了基类的 getProjectMap 函数，首先调用基类的同名函数获取投射映射。然后检查 LEX 区域是否投射到多个区域，如果是，则抛出异常。
   （3） getWord该函数重载了基类的 getWord 函数，首先调用基类的同名函数获取单词。如果没有找到匹配的单词且区域名为 DET，则检查 DET 区域的获胜神经元是否与 nodet_assembly 有足够的重叠，如果是，则返回 。

```
class EnglishParserBrain : public ParserBrain {
public:
    EnglishParserBrain(double p, int non_LEX_n = 2000, int non_LEX_k = 100, int
LEX_k = 20,
                       double default_beta = 0.2, double LEX_beta = 1.0, double
recurrent_beta = 0.05,
                       double interarea_beta = 0.5, bool verbose = false)
        : ParserBrain(p, LEXEME_DICT, AREAS, RECURRENT_AREAS, {"LEX", "SUBJ",
"VERB"}, ENGLISH_READOUT_RULES),
          verbose(verbose) {

        int LEX_n = LEX_SIZE * LEX_k;
        add_explicit_area("LEX", LEX_n, LEX_k, default_beta);
```

```
        int DET_k = LEX_k;
        add_area("SUBJ", non_LEX_n, non_LEX_k, default_beta);
        add_area("OBJ", non_LEX_n, non_LEX_k, default_beta);
        add_area("VERB", non_LEX_n, non_LEX_k, default_beta);
        add_area("ADJ", non_LEX_n, non_LEX_k, default_beta);
        add_area("PREP", non_LEX_n, non_LEX_k, default_beta);
        add_area("PREP_P", non_LEX_n, non_LEX_k, default_beta);
        add_area("DET", non_LEX_n, non_LEX_k, default_beta);
        //add_area("DET", non_LEX_n, DET_k, default_beta);
        add_area("ADVERB", non_LEX_n, non_LEX_k, default_beta);

        // 初始化自定义可塑性
        std::unordered_map<std::string, std::vector<std::pair<std::string,
float>>> custom_plasticities;
        for (const auto& area : RECURRENT_AREAS) {
            custom_plasticities["LEX"].emplace_back(area, LEX_beta);
            custom_plasticities[area].emplace_back("LEX", LEX_beta);
            custom_plasticities[area].emplace_back(area, recurrent_beta);
            for (const auto& other_area : RECURRENT_AREAS) {
                if (other_area != area) {
                    custom_plasticities[area].emplace_back(other_area,
interarea_beta);
                }
            }
        }
        update_plasticities(custom_plasticities);
    }

    unordered_map<std::string, std::vector<std::string>> getProjectMap()override
    {
        unordered_map<std::string, std::vector<std::string>> proj_map =
ParserBrain::getProjectMap();
        if (proj_map.find("LEX") != proj_map.end() && proj_map["LEX"].size() > 2)
{
            throw std::runtime_error("Got that LEX projecting into many areas: " +
std::to_string(proj_map["LEX"].size()));
        }
        return proj_map;
    }

    std::string getWord(const std::string& area_name, double min_overlap =
0.7)override{
        std::string word = ParserBrain::getWord(area_name, min_overlap);
        if (!word.empty()) {
            return word;
        }
        return "<NON-WORD>";
    }

private:
    bool verbose;

};
```

```
```

其他函数

（1） read_out 函数：读取指定区域的词汇及其依赖关系。首先获取 area 投射到的区域，然后投射到这些区域并获取词汇，最后递归读取非 LEX 区域的依赖关系。

```
// 读取输出函数 read_out
void read_out(const string& area, const unordered_map<string, vector<string>>&
mapping, EnglishParserBrain& b, vector<vector<string>>& dependencies) {
    // 获取要读取的目标区域列表
    auto& to_areas = mapping.at(area);

    // 读取当前区域到目标区域的映射
    unordered_map<string, vector<string>> a1;
    for (auto& to : to_areas) {
        a1[area].push_back(to);
    }

    // 项目映射到目标区域
    b.project({}, a1);
    auto this_word = b.getWord("LEX");

    // 遍历每一个目标区域
    for (const auto& to_area : to_areas) {
        if (to_area == "LEX") continue;

        // 准备项目映射
        unordered_map<string, vector<string>> a3;
        unordered_map<string, vector<string>> a2{{to_area, {"LEX"}}};

        // 项目映射到目标区域
        b.project(a3, a2);
        auto other_word = b.getWord("LEX");
        dependencies.push_back({this_word, other_word, to_area});
    }

    // 递归读取输出，处理非 "LEX" 的目标区域
    for (const auto& to_area : to_areas) {
        if (to_area != "LEX") {
            read_out(to_area, mapping, b, dependencies);
        }
    }
}
```

（2） parseHelper 函数：辅助解析句子，parse函数的辅助函数。主要过程是：先分割句子成一个个单词并激活每个单词，应用预处理规则。然后进行多轮投射和解析，应用后处理规则。最后读取依赖关系。

```
// 解析辅助函数 parseHelper
int parseHelper(EnglishParserBrain& b, const string& sentence, float p, int LEX_k,
```

```
int project_rounds, bool verbose, bool debug,
    unordered_map<string, Rules>& lexeme_dict,
    const vector<string>& all_areas, const vector<string>& explicit_areas,
    ReadoutMethod readout_method, const unordered_map<string, vector<string>>&
readout_rules) {

    // 解析句子获取单词列表
    string w = "";
    vector<string> words;
    for (int i = 0; i < sentence.length(); i++) {
        if (sentence[i] == ' ') {
            words.push_back(w);
            w = "";
        } else {
            w += sentence[i];
        }
    }
    words.push_back(w);

    // 对每一个单词进行处理
    for (const string& word : words) {
        Rules& lexeme = lexeme_dict.at(word);
        b.activateWord("LEX", word);
        if (verbose) {
            cout << "Activated word: " << word << endl;
            // 输出激活的单词
            for (auto& it : b.area_by_name["LEX"].winners) {
                cout << it << " ";
            }
            cout << endl;
        }

        // 应用预规则
        for (Rule& rule : lexeme.PRE_RULES) {
            b.applyRule(rule);
        }

        // 获取项目映射
        auto proj_map = b.getProjectMap();
        for (const auto& [area, value] : proj_map) {
            // 如果 "LEX" -> area 不在映射中，则修复组装
            if (find(proj_map["LEX"].begin(), proj_map["LEX"].end(), area) ==
proj_map["LEX"].end()) {
                b.area_by_name[area].fix_assembly();
                if (verbose) {
                    cout << "FIXED assembly bc not LEX->this area in: " << area <<
endl;
                }
            } else if (area != "LEX") {
                // 否则，解除组装并清空赢家
                b.area_by_name[area].unfix_assembly();
                b.area_by_name[area].winners.clear();
                if (verbose) {
                    cout << "ERASED assembly because LEX->this area in " << area
```

```
<< endl;
                }
            }
        }

        // 进行多轮项目映射
        for (int i = 0; i < project_rounds; ++i) {
            b.parse_project();
        }

        // 应用后规则
        for (Rule& rule : lexeme.POST_RULES) {
            b.applyRule(rule);
        }
    }

    // 禁用可塑性
    b.disable_plasticity = true;
    for (const auto& area : all_areas) {
        b.area_by_name[area].unfix_assembly();
    }

    // 读取依赖关系
    vector<vector<string>> dependencies;

    if (readout_method == ReadoutMethod::FIBER_READOUT) {
        unordered_map<string, set<string>> activated_fibers =
b.getActivatedFibers();
        if (verbose) {
            cout << "Got activated fibers for readout:" << endl;
            // 输出激活的纤维
            for (auto& it : activated_fibers) {
                cout << it.first << ": ";
                for (auto& x : it.second) {
                    cout << x << " ";
                }
                cout << endl;
            }
        }

        // 读取输出
        read_out("VERB", set_map_to_vector_map(activated_fibers), b,
dependencies);
        cout << "-------------------" << endl;
        cout << "Got dependencies: " << endl;
        // 输出依赖关系
        for (auto& it : dependencies) {
            for (auto& x : it) cout << x << " ";
            cout << endl;
        }
        cout << "-------------------" << endl;
    }
    return words.size();
}
```

```
```

（3） parse 函数：解析输入的句子。首先创建并初始化 EnglishParserBrain 对象，然后调用 parseHelper 函数进行具体的解析工作。

```
// 解析函数 parse
int parse(const std::string& sentence = "big people bite the big dogs quickly",
const std::string& language = "English",
           double p = 0.1, int LEX_k = 20, int project_rounds = 20, bool verbose =
false,
           bool debug = false, ReadoutMethod readout_method =
ReadoutMethod::FIBER_READOUT) {

    std::unordered_map<std::string, Rules> lexeme_dict;
    std::vector<std::string> all_areas;
    std::vector<std::string> explicit_areas;
    std::unordered_map<std::string, std::vector<std::string>> readout_rules;

    EnglishParserBrain brain = EnglishParserBrain(p,2000,100,
LEX_k,0.2,1.0,0.05,0.5, verbose);

    lexeme_dict = LEXEME_DICT;
    all_areas = AREAS;
    explicit_areas = EXPLICIT_AREAS;
    readout_rules = ENGLISH_READOUT_RULES;

    if (language != "English"){
        cout<<"Not English?"<<endl;
        return -1;
    }
    return parseHelper(brain, sentence, p, LEX_k, project_rounds, verbose, debug,
                lexeme_dict, all_areas, explicit_areas, readout_method,
readout_rules);
}
```

提升代码性能和保证线程安全

多线程并行方式：因为可以独立处理每个单词，所以我们考虑使用多线程OpenMP来并行化词汇激活和投射步骤，所以我们主要是在brain.h 的project_into里加了 #pragma omp parallel for实现多线程加速，每个线程可以处理循环中的不同部分。

#pragma omp atomic: 这个指令确保紧随其后的操作是原子的。也就是说，这个操作在多个线程之间不会相互干扰，是一个不可分割的整体。对于语句previous_winners_input[i] += connectome[w][i]:对previous_winners_input[i] 进行累加操作，确保这个累加操作是原子的。这样，多个线程可以安全地对previous_winners_input[i] 进行累加，而不会导致数据不一致。

```
int project_into(Area& target_area, const vector<string>& from_areas) {

        int num_inputs_processed;
        int processed_input_count;
```

```
        // If projecting from area with no assembly, throw an error.
        for (const auto& from_area_name : from_areas) {
            auto& from_area = area_by_name[from_area_name];
            if (from_area.winners.empty() || from_area.w == 0) {
                throw std::runtime_error("Projecting from area with no assembly: "
+ from_area_name);
            }
        }

        vector<vector<int>> inputs_by_first_winner_index;
        string target_area_name = target_area.name;

        if (target_area.fixed_assembly) {
            target_area._new_winners = target_area.winners;
            target_area._new_w = target_area.w;
            num_inputs_processed = 0;
        }
        else {

            vector<int> cumulative_winners_per_source_area;
            int total_source_areas;
            int total_winners;
            vector<float> previous_winners_input(target_area.w, 0.0f);
            vector<float> all_potential_winner_inputs;

                #pragma omp parallel for
             for (const auto& from_area_name : from_areas) {
                auto& connectome = connectomes[from_area_name]
[target_area_name].con;
                auto& from_area = area_by_name[from_area_name];
                for (int w : from_area.winners) {
                    for (size_t i = 0; i < target_area.w; ++i) {
                        #pragma omp atomic
                        previous_winners_input[i] += connectome[w][i];
                    }
                }
            }

            if (!target_area.explicitArea) {
                float effective_neurons, quantile_value, alpha_value, mean,
stddev, lower_bound, upper_bound;
                vector<float> potential_new_winner_inputs(target_area.k);
                total_source_areas = 0;
                total_winners = 0;
                cumulative_winners_per_source_area.push_back(total_winners);

                for (const auto& from_area_name : from_areas) {
                    int active_winners =
area_by_name[from_area_name].winners.size(); // effective_k
                    total_source_areas += 1;
                    total_winners += active_winners;
                    cumulative_winners_per_source_area.push_back(total_winners);
                }
```

```
                effective_neurons = target_area.n - target_area.w;
                if (effective_neurons <= target_area.k) {
                    cout << "Remaining size of area " << target_area_name << " too
small to sample k new winners." << endl;
                    return -1;
                }

                quantile_value = (effective_neurons - target_area.k) /
effective_neurons;
                alpha_value = BinomQuantile(total_winners, p, quantile_value);
                mean = total_winners * p;
                stddev = sqrt(total_winners * p * (1.0 - p));
                lower_bound = (alpha_value - mean) / stddev;
                #pragma omp parallel for
                for (auto& input : potential_new_winner_inputs)
                    input = min<float>(total_winners,
round(TruncatedNorm(lower_bound, _rng) * stddev + mean));

                all_potential_winner_inputs.resize(previous_winners_input.size() +
potential_new_winner_inputs.size());
                copy(previous_winners_input.begin(), previous_winners_input.end(),
all_potential_winner_inputs.begin());
                copy(potential_new_winner_inputs.begin(),
potential_new_winner_inputs.end(), all_potential_winner_inputs.begin() +
previous_winners_input.size());
            }
            else {
                all_potential_winner_inputs = previous_winners_input;
            }

            std::vector<int> new_winner_indices =
getNLargestIndices(all_potential_winner_inputs, target_area.k);
            vector<float> initial_winner_inputs;
            num_inputs_processed = 0;
            if (!target_area.explicitArea) {
                int new_winner_indices_len = new_winner_indices.size();
                for (int i = 0; i < new_winner_indices_len; i++) {
                    if (new_winner_indices[i] >= target_area.w) {

initial_winner_inputs.push_back(all_potential_winner_inputs[new_winner_indices[i]]
);
                        new_winner_indices[i] = target_area.w +
num_inputs_processed;
                        ++num_inputs_processed;
                    }
                }
            }

            target_area._new_winners = new_winner_indices;
            target_area._new_w = target_area.w + num_inputs_processed;
            inputs_by_first_winner_index = vector<vector<int>>
(num_inputs_processed, vector<int>());
```

```
            for (auto i = 0; i < num_inputs_processed; i++) {
                vector<int> input_indices = uniqueRandomChoices(total_winners,
int(initial_winner_inputs[i]),_rng);
                vector<int> connections_per_source_area(total_source_areas, 0);
                for (auto j = 0; j < total_source_areas; j++) {
                    #pragma omp parallel for
                    for (const auto& winner : input_indices) {
                        if (cumulative_winners_per_source_area[j + 1] > winner &&
winner >= cumulative_winners_per_source_area[j])
                            #pragma omp atomic
                            connections_per_source_area[j] += 1;
                    }
                }
                inputs_by_first_winner_index[i] = connections_per_source_area;
            }
        }

        processed_input_count = 0;
        for(auto&from_area_name : from_areas) {
            int&  from_area_w = area_by_name[from_area_name].w;
            vector<int>& from_area_winners = area_by_name[from_area_name].winners;
            Connectome& the_connectomes = connectomes[from_area_name]
[target_area_name];

            the_connectomes.colpadding(num_inputs_processed, 0);
             #pragma omp parallel for
            for (int i = 0; i < num_inputs_processed; ++i) {
                int total_in = inputs_by_first_winner_index[i]
[processed_input_count];
                vector<int>sample_indices = random_choice(
total_in,from_area_winners);
                for(auto& j : sample_indices)
                    the_connectomes.con[j][target_area.w + i] = 1.0;

                for (int j = 0; j <  from_area_w; ++j)
                    if (find(from_area_winners.begin(), from_area_winners.end(),
j) == from_area_winners.end())
                        the_connectomes.con[j][target_area.w + i] =
std::binomial_distribution<int>(1, p)(_rng);
            }

            float area_to_area_beta = disable_plasticity ? 0 :
target_area.beta_by_area[from_area_name]; // area_to_area_beta
             #pragma omp parallel for
            for (const auto& i : target_area._new_winners)
                for (const auto& j : from_area_winners)
                     #pragma omp atomic
                    the_connectomes.con[j][i] *= 1.0 + area_to_area_beta;
            processed_input_count++;
        }

        for (auto& kv : area_by_name) {
            const std::string& other_area_name = kv.first;
            Area& other_area = kv.second;
```

```
            if (find(from_areas.begin(), from_areas.end(), other_area_name) ==
from_areas.end()) {
                connectomes[other_area_name]
[target_area_name].colpadding(num_inputs_processed, 0);
                for (int i = 0; i < connectomes[other_area_name]
[target_area_name].row; ++i) {
                    for (int j = target_area.w; j < target_area._new_w; ++j) {
                        connectomes[other_area_name][target_area_name].con[i][j] =
std::binomial_distribution<int>(1, p)(_rng);
                    }
                }
            }

            connectomes[target_area_name]
[other_area_name].rowpadding(num_inputs_processed, 0);
            #pragma omp parallel for
            for (int i = target_area.w; i < target_area._new_w; ++i) {
                for (int j = 0; j < connectomes[target_area_name]
[other_area_name].col; ++j) {
                    connectomes[target_area_name][other_area_name].con[i][j] =
std::binomial_distribution<int>(1, p)(_rng);
                }
            }
        }
        return num_inputs_processed;
    }
```

结果展示

```
// Test case for parsing sentences
TEST(ParserTest, BasicSentences1) {
    verify_parse("dogs are big", "are big ADJ are dogs SUBJ ");
}
TEST(ParserTest, BasicSentences2) {
    verify_parse("cats are bad", "are bad ADJ are cats SUBJ ");
}
TEST(ParserTest, BasicSentences3) {
    verify_parse("the big dogs chase the bad cats quickly", "chase quickly ADVERB chase <NON-WORD> SUBJ <NON-WORD> big ADJ <NON-WORD> the DET ");
}
TEST(ParserTest, BasicSentences4) {
    verify_parse("big people bite the big dogs quickly", "bite quickly ADVERB bite <NON-WORD> SUBJ <NON-WORD> big ADJ <NON-WORD> the DET ");
}
TEST(ParserTest, BasicSentences5) {
    verify_parse("people run", "run people SUBJ ");
}
```

我们使用给定单词集里的单词组成合法句子，然后用GitHub上面的python代码跑出一个正确的结果，将这个正确的结果与我们代码对句子的解析结果进行比较。

```
[==========] Running 5 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 5 tests from ParserTest
[ RUN      ] ParserTest.BasicSentences1
-------------------
Got dependencies:
are big ADJ
are dogs SUBJ
```

```
-------------------
Sentence: "dogs are big"
Elapsed time: 0.0366834 seconds
Time per word: 0.00305695 seconds/word
[       OK ] ParserTest.BasicSentences1 (36 ms)
[ RUN      ] ParserTest.BasicSentences2
-------------------
Got dependencies:
are bad ADJ
are cats SUBJ
-------------------
Sentence: "cats are bad"
Elapsed time: 0.0347248 seconds
Time per word: 0.00289373 seconds/word
[       OK ] ParserTest.BasicSentences2 (34 ms)
[ RUN      ] ParserTest.BasicSentences3
-------------------
Got dependencies:
chase quickly ADVERB
chase <NON-WORD> SUBJ
<NON-WORD> big ADJ
<NON-WORD> the DET
-------------------
Sentence: "the big dogs chase the bad cats quickly"
Elapsed time: 0.113197 seconds
Time per word: 0.00290249 seconds/word
[       OK ] ParserTest.BasicSentences3 (113 ms)
[ RUN      ] ParserTest.BasicSentences4
-------------------
Got dependencies:
bite quickly ADVERB
bite <NON-WORD> SUBJ
<NON-WORD> big ADJ
<NON-WORD> the DET
-------------------
Sentence: "big people bite the big dogs quickly"
Elapsed time: 0.0909198 seconds
Time per word: 0.00252555 seconds/word
[       OK ] ParserTest.BasicSentences4 (91 ms)
[ RUN      ] ParserTest.BasicSentences5
-------------------
```

```
Got dependencies:
run people SUBJ
```

```
--------------------
Sentence: "big people bite the big dogs quickly"
Elapsed time: 0.0909198 seconds
Time per word: 0.00252555 seconds/word
[       OK ] ParserTest.BasicSentences4 (91 ms)
[ RUN      ] ParserTest.BasicSentences5
--------------------
Got dependencies:
run people SUBJ
--------------------
Sentence: "people run"
Elapsed time: 0.0216472 seconds
Time per word: 0.00216472 seconds/word
[       OK ] ParserTest.BasicSentences5 (22 ms)
[----------] 5 tests from ParserTest (299 ms total)

[----------] Global test environment tear-down
[==========] 5 tests from 1 test suite ran. (299 ms total)
[  PASSED  ] 5 tests.

D:\VisualStudio\project\Project1\x64\Release\Project1.exe (进程 56280)已退出，代码为 0。
按任意键关闭此窗口. . .
```

可以看到我们的解析器可以正确解析规定单词的正确语句，且解析时长/句子单词数大约为2ms。还可以在test文件中增加测试语句和相应的正确解析（如果发现不同会显示我们的解析结果）。