一、组员分工

王哲：实现Area类框架及函数，代码注释和撰写实验报告

林宇浩：实现project、project_into等核心函数，整体代码调试，代码修改和实现项目正确运行输出，撰写实验报告

熊蔚然：实现parserBrain、EnglishParserBrain等类框架及功能函数，整体代码调试，实现cmake编译、Google Test测试和性能测试

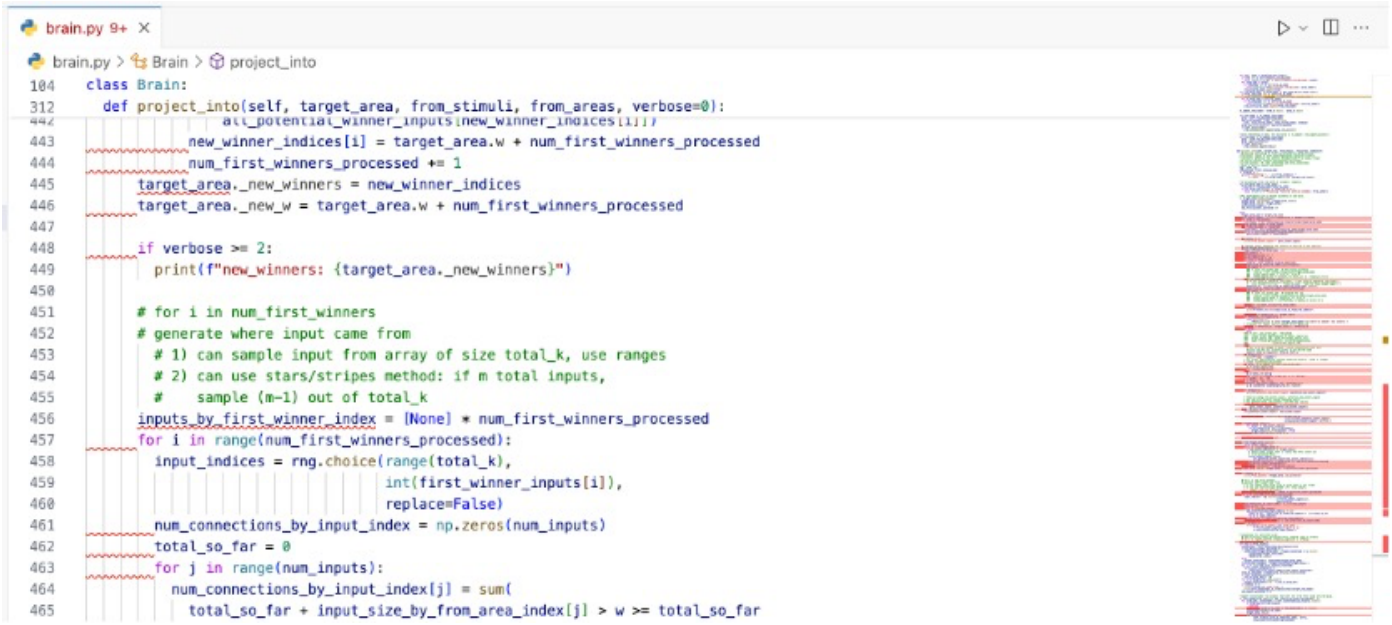
二、项目设计思路

本次实验的主要内容是利用集合演算中所提出的集合演算算法实现依赖解析，并使用利用C/C++实现该项目。

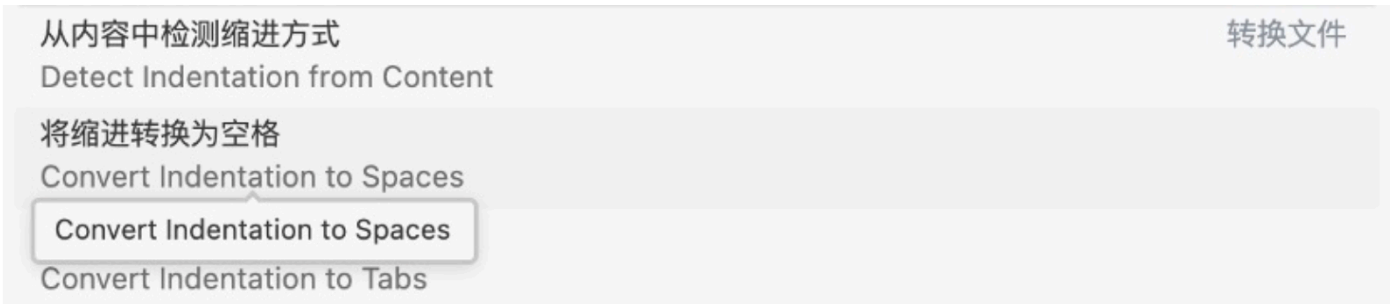
1. 参考资料

本次实验中，原论文已经为我们提供了一个较为完整的python代码，并提供了cpp代码所需的基础类的一些定义。同时助教也为我们提供了一个修改后版本的代码。

大致浏览一下实验资料，我们可以发现原始论文的代码中有一些问题，比如在brain.py文件中，存在大量报错。



这里是原因是代码制表符和空格符使用混乱导致的，我们可以用编辑器自带的将制表符转换为空格符的功能来解决这个问题。



然后我们还可以看到，原论文的代码中，parser.py文件和stdlib模块中的parser重名了，这也可能引发潜在的问题。

助教的代码已经将这些相关的小问题进行了一些修复，比如将parser.py文件重新命名为brain_parser.py从而避免重名。所以我们之后工作基于助教提供的代码，感谢助教为我们提供的帮助。

2. 项目搭建思路

我们首先对python代码进行了调试，将python代码中存在的错误进行了修复。

```
Got proj_map =
defaultdict(<class 'set'>, {'LEX': {'LEX', 'OBJ'}, 'OBJ': {'LEX', 'VERB', 'OBJ'}, 'VERB': {'VERB', 'OBJ'}})
Got proj_map =
defaultdict(<class 'set'>, {'LEX': {'LEX', 'OBJ'}, 'OBJ': {'LEX', 'VERB', 'OBJ'}, 'VERB': {'VERB', 'OBJ'}})
Got proj_map =
defaultdict(<class 'set'>, {'LEX': {'LEX', 'OBJ'}, 'OBJ': {'LEX', 'VERB', 'OBJ'}, 'VERB': {'VERB', 'OBJ'}})
Got proj_map =
defaultdict(<class 'set'>, {'LEX': {'LEX', 'OBJ'}, 'OBJ': {'LEX', 'VERB', 'OBJ'}, 'VERB': {'VERB', 'OBJ'}})
Got activated fibers for readout:
defaultdict(<class 'set'>, {'SUBJ': {'LEX'}, 'VERB': {'LEX', 'OBJ', 'SUBJ'}, 'OBJ': {'LEX'}})
Got dependencies:
[['chase', 'mice', 'OBJ'], ['chase', 'cats', 'SUBJ']]
(base) lin@LindeMacBook-Pro assemblies_sysu %
```

可以看到修复后的python代码能够利用集合演算进行依赖解析。这样python代码就为我们提供了一个较为完整的示例，从而在我们进行结构设计的时候可以一定程度上参考python代码的一些内容。

对于cpp代码，我们可以看到，在实验资料中已经定义了一些类和函数，并且使用了Bazel构建了项目。我们可以直接使用下面的Bazel命令进行运行：

```
bazel build //:brain
bazel test //:brain_test
```

```
Running main() from gmock_main.cc
[==========] Running 5 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 5 tests from BrainTest
[ RUN      ] BrainTest.TestProjection
[       OK ] BrainTest.TestProjection (551 ms)
[ RUN      ] BrainTest.TestExplicitProjection
[       OK ] BrainTest.TestExplicitProjection (53161 ms)
[ RUN      ] BrainTest.TestCompletion
[       OK ] BrainTest.TestCompletion (73 ms)
[ RUN      ] BrainTest.TestAssociation
brain_test.cc:157: Failure
Expected: (NumCommon(proj_A, proj_AB)) >= (90), actual: 59 vs 90

[  FAILED  ] BrainTest.TestAssociation (239 ms)
[ RUN      ] BrainTest.TestMerge
brain_test.cc:98: Failure
Expected: (NumCommon(area.activated, prev_activated[area_name])) >= (convergence * area.k), actual: 316 vs 317
```

通过Google Test我们可以看到类函数的运行效果。可以看到有部分函数的功能是已经大致实现了的，这些cpp类函数也能够给我们带来一些参考。

为了给项目构建打牢基础，我们进一步仔细阅读了实验资料中python代码和cpp代码，并在读懂代码内容后将原代码加上了注释从而方便组员之间相互讨论。我们注意到，实验资料中的pyhton代码虽然已经可以基本实现利用集合演算进行依赖解析，但是其内部代码结构并不是非常理想，除了代码本身有一些错误以外，代码还有很多冗余的操作，一些数据结构的使用也并不高效，并且部分代码逻辑有一定的问题。此外，如果选择直接将python代码转为cpp代码，则会遇到很多问题，首先两种代码之间本身存在很大的差异，python的一些库函数也无法有效进行转换，而且这样也无法发挥cpp框架的一些优良特性。所以经过小组讨论后我们决定，通过参考实验资料已提供的代码，我们首先明确哪些结构是必要的，然后在搭建好的框架基础之上，按照我们自己的理解进行设计。

3. 项目结构介绍

3.1 brain.h

rules_and_lexemeDict.h文件主要定义了一些用于语言解析的常量、类和函数、

3.1.1 头文件

下面介绍本次实验中所用到的库：

`unordered_map`、`map`、`unordered_set`：用于哈希表和集合的高效存储和访问。

`memory`、`vector`：用于动态内存管理和动态数组。

`typeinfo`、`sstream`、`queue`、`set`、`cmath`、`numeric`、`algorithm`：用于类型信息处理、字符串流、队列、集合、数学计算、数值处理和算法操作。

`boost/math/distributions/binomial.hpp`、`boost/math/distributions/normal.hpp`：用于实现二项分布和正态分布的数学运算。

`boost/random.hpp`、`boost/random/mersenne_twister.hpp`、`boost/random/uniform_real_distribution.hpp`：用于生成随机数，包括复杂分布的采样和均匀分布的数值生成等等。

3.1.2 一些类的定义

(1) 规则集

GenericRuleSet用于定义词汇规则集的重要数据结构，包含词汇的索引、前置规则和后置规则，通过这些规则控制词汇在解析过程中的行为。定义如下所示：

```
class GenericRuleSet {
public:
    int index;
    std::vector<Rule> pre_rules;
    std::vector<Rule> post_rules;

    GenericRuleSet() = default;
    GenericRuleSet(const GenericRuleSet& other) = default;
    GenericRuleSet& operator=(const GenericRuleSet& other) = default;
};
```

1. **`index`**：表示词汇在词汇表中的索引位置，用于唯一标识词汇。
2. **`pre_rules`**：表示在词汇激活前需要应用的规则集合。每个规则都是一个 Rule 对象，包含动作、区域和其他相关信息。
3. **`post_rules`**：表示在词汇激活后需要应用的规则集合。每个规则同样是一个 Rule 对象。

(2) 规则

Rule类是定义解析规则的核心数据结构，通过包含动作类型、区域和索引等信息，Rule类在解析过程中控制大脑区域的行为。定义如下所示：

```
class Rule {
public:
    std::string action;  // 动作类型，如 DISINHIBIT 或 INHIBIT
    std::string area1;   // 适用于 FiberRule 的第一个区域名称
    std::string area2;   // 适用于 FiberRule 的第二个区域名称
    std::string area;    // 适用于 AreaRule 的单一区域名称
    int index;           // 规则的索引
    bool flag;           // 表示规则类型的标识，false 表示 AreaRule，true 表示 FiberRule

    // 构造函数
    Rule(std::string a, std::string ar, int i);
    Rule(std::string a, std::string a1, std::string a2, int i);

    // 复制构造函数和赋值操作符
    Rule(const Rule& other) = default;
    Rule& operator=(const Rule& other) = default;
};
```

1. **`action`**：表示规则的动作类型，如 `DISINHIBIT`（解除抑制）或 `INHIBIT`（抑制）。
2. **`area1`**：适用于 FiberRule 的第一个区域名称。
3. **`area2`**：适用于 FiberRule 的第二个区域名称。
4. **`area`**：适用于 AreaRule 的单一区域名称。
5. **`index`**：表示规则的索引，用于唯一标识规则。
6. **`flag`**：标识规则类型，`false` 表示 AreaRule，`true` 表示 FiberRule。

3.2 brain.cpp

rules_and_lexemeDict.cpp文件实现了定义在rules_and_lexemeDict.h头文件中的类和函数，主要用于生成词汇规则集和初始化词汇表

3.2.1 规则集构建函数

- `generic_noun(int index)`：生成名词的规则集。
- `generic_trans_verb(int index)`：生成及物动词的规则集。
- `generic_intrans_verb(int index)`：生成不及物动词的规则集。
- `generic_copula(int index)`：生成系动词的规则集。
- `generic_adverb(int index)`：生成副词的规则集。
- `generic_determinant(int index)`：生成限定词的规则集。
- `generic_adjective(int index)`：生成形容词的规则集。
- `generic_preposition(int index)`：生成介词的规则集。

3.2.2 一些常量

`AREA_RULE` 和 `FIBER_RULE`：分别表示区域规则和纤维规则

`DISINHIBIT` 和 `INHIBIT`：分别表示解除抑制和抑制动作

大脑区域常量

1. **`LEX`**：词汇
2. **`DET`**：限定词
3. **`SUBJ`**：主语
4. **`OBJ`**：宾语
5. **`VERB`**：动词
6. **`PREP`**：介词
7. **`PREP_P`**：介词短语
8. **`ADJ`**：形容词
9. **`ADVERB`**：副词
10. **`NOM`**：主格
11. **`ACC`**：宾格
12. **`DAT`**：与格

3.3 main.cpp

update_plasticity.cpp文件是项目的核心文件，实现了算法模拟的功能，包括词汇表的初始化、规则集的生成、神经元连接的更新以及解析器的大脑区域模拟。

3.3.1 一些主要函数

(1) 截断正态分布采样

```
std::vector<double> truncnorm_rvs(double a, double b, double scale, std::size_t size);
std::vector<double> generate_truncated_normal_samples(double mean, double stddev, double
lower_bound, double upper_bound, int sample_size);
```

这两个函数用于生成截断正态分布样本，以模拟神经元连接权重的变化。这与论文中描述的大脑模拟和神经网络模型中的权重初始化和更新过程有关。

(2) 输入向量中最大的k个值的索引

```
std::vector<int> nlargest_indices(const std::vector<double>& all_potential_winner_inputs,
int k);
```

该函数用于选择最活跃的神经元，这与论文中描述的神经元激活过程和竞争机制相符，模拟了大脑中的神经元选择过程。

(3) 添加显式区域

```
void add_explicit_area(std::string area_name, int n, int k, double beta, double
custom_inner_p = 0, double custom_out_p = 0, double custom_in_p = 0);
```

函数用于添加新的大脑区域，并初始化这些区域之间的连接。这与论文中描述的大脑区域和连接的结构构建过程有关，帮助模拟大脑不同区域之间的相互作用。

(4) 执行神经元激活和传播

```
void project(int areas_by_stim, defaultdict_set dst_areas_by_src_area, int verbose = 0);
int project_into(Area& target_area, std::vector<std::string> from_stimuli,
std::vector<std::string> from_areas, int verbose = 0);
```

函数用于模拟神经元激活和传播过程，这是论文中描述的神经网络模型的核心部分。通过调用函数从而模拟输入信号在大脑不同区域之间的传播和处理过程。

(5) 更新区域之间的参数

```
void update_plasticity(const std::string& from_area, const std::string& to_area, double
new_beta);
void update_plasticities(const std::unordered_map<std::string,
std::vector<std::pair<std::string, double>>>& area_update_map = {}, const
std::unordered_map<std::string, std::vector<std::pair<std::string, double>>>&
stim_update_map = {});
```

函数用于更新神经元连接的可塑性参数，这是模拟大脑学习和记忆过程的关键部分。论文中提到的神经可塑性和突触权重调整通过这些函数得以实现。

(6) 读取解析结果

```
void read_out(ParserBrain& b, const std::string& area, const
std::unordered_map<std::string, std::unordered_set<std::string>>& mapping,
std::vector<std::vector<std::string>>& dependencies);
```

该函数用于从大脑模型中提取解析结果，确定不同区域之间的依存关系。此过程模拟了大脑在理解语言时，如何从神经活动中提取语法和语义信息。

(7) 解析输入句子

```
void parse(const std::string& sentence = "cat chase mice", 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);
```

此函数用于初始化和执行整个解析过程，模拟大脑对输入句子的解析。它涉及到词汇激活、规则应用和神经元激活传播，是论文中描述的解析器模型的核心实现。

(8) 执行解析过程和规则应用

```
void parseHelper(ParserBrain& b, const std::string& sentence, double p, int LEX_k, int
project_rounds, bool verbose, bool debug, const std::unordered_map<std::string,
GenericRuleSet>& lexeme_dict, const std::vector<std::string>& all_areas, const
std::vector<std::string>& explicit_areas, ReadoutMethod readout_method, const
std::unordered_map<std::string, std::vector<std::string>>& readout_rules);
```

该函数是解析过程的详细实现部分，包含对输入句子的逐词处理、规则应用、神经元激活和依存关系提取。它体现了论文中提出的解析策略和神经网络模型在处理自然语言时的具体操作。

(9) 取大脑区域中当前激活的词汇

```
std::unordered_map<std::string, std::unordered_set<std::string>> getProjectMap() const;
```

该函数用于确定当前解析状态下大脑区域之间的连接情况，是解析器在语言处理过程中建立区域联系的关键步骤。

(10) 激活特定大脑区域中的词汇

```
void activateWord(const std::string& area_name, const std::string& word);
```

用于激活词汇对应的神经元集合，模拟大脑中词汇的激活过程，是解析器处理输入句子的基本操作之一。

(11) 获取特定大脑区域中当前激活的词汇

```
std::string getWord(const std::string& area_name, double min_overlap = 0.7);
```

用于获取特定区域中当前激活的词汇，通过比较激活神经元集合与词汇集合的重叠度，模拟大脑在词汇识别中的神经机制。

3.3.2 一些主要的类

(1) Area类

Area类表示大脑的一个区域，包含神经元的激活状态和连接矩阵的管理。该类模拟了大脑中不同区域的结构和功能。

```
class Area {
public:
    std::string name;
    int n;
    int k;
    double beta;
    std::vector<int> active_neurons;
    std::vector<std::vector<double>> weights;

    Area(const std::string& name, int n, int k, double beta);
    void activate(const std::vector<int>& neurons);
    void update_weights(const std::vector<int>& pre_neurons, const std::vector<int>&
post_neurons);
};
```

主要成员变量：

- `std::string name`：区域名称。
- `int n`：神经元数量。
- `int k`：激活神经元数量。
- `double beta`：可塑性参数。
- `std::vector<int> active_neurons`：当前激活的神经元集合。
- `std::vector<std::vector<double>> weights`：连接权重矩阵。

(2) Brain类

Brain类表示整个大脑的结构和动态，包括多个Area实例及其之间的连接关系。该类模拟了大脑在处理语言时的整体功能。

```
class Brain {
public:
    std::unordered_map<std::string, Area> areas;
    std::unordered_map<std::string, std::unordered_map<std::string, std::vector<double>>>
connections;

    Brain(double p, int area_size, int area_k, double beta);
    void add_area(const std::string& area_name, int n, int k, double beta);
    void project(const std::string& from_area, const std::string& to_area);
    void update_plasticity(const std::string& from_area, const std::string& to_area,
double new_beta);
};
```

主要成员变量：

- `std::unordered_map<std::string, Area> areas`：大脑中所有区域的映射。
- `std::unordered_map<std::string, std::unordered_map<std::string, std::vector<double>>> connections`：区域之间的连接权重矩阵。

主要成员函数：

- `void add_area(const std::string& area_name, int n, int k, double beta)`：添加新的区域并初

始化连接矩阵。

- `void project(const std::string& from_area, const std::string& to_area)`：执行神经元激活和传播操作，更新区域状态。
- `void update_plasticity(const std::string& from_area, const std::string& to_area, double new_beta)`：更新两个区域之间的可塑性参数。

3.3.3 主要流程介绍

1. 初始化阶段

在程序执行的开始阶段，首先初始化了词汇表和大脑区域之间的连接规则。这些初始化步骤包括定义每个词汇及其对应的规则集（如 `generic_noun`、`generic_trans_verb` 等），以及设置区域之间的读取规则（如 `ENGLISH_READOUT_RULES`）。这些规则和词汇表在解析过程中起着至关重要的作用，确保解析器能够正确地处理和理解输入的句子。这与论文中提到的解析器框架一致，通过定义详细的规则集来模拟大脑在处理语言时的机制。

```
std::unordered_map<std::string, GenericRuleSet> LEXEME_DICT = {
    {"the", generic_determinant(0)},
    {"a", generic_determinant(1)},
    // ...
};

std::unordered_map<std::string, std::vector<std::string>> ENGLISH_READOUT_RULES = {
    {VERB, {LEX, SUBJ, OBJ, PREP_P, ADVERB, ADJ}},
    // ...
};
```

2. 构建大脑解析器

`parse` 函数被调用来解析输入句子。这个函数首先调用 `EnglishParserBrain` 类的构造函数，初始化解析器大脑对象。`EnglishParserBrain` 继承自 `ParserBrain`，而 `ParserBrain` 继承自 `Brain` 类。这一层层的继承关系确保了解析器大脑对象能够拥有完整的神经网络模拟能力。初始化过程中，设定了各个大脑区域的参数，如神经元数量、激活神经元数量和可塑性参数。这些参数在解析过程中将会影响神经元的激活和连接权重的更新。这一过程对应于论文中描述的神经网络模型的初始化阶段，通过设置适当的参数来模拟大脑区域的结构和功能。

3. 激活词汇

对于输入句子中的每个单词，程序调用 `activateWord` 函数激活相应的词汇区域。这个函数会根据词汇表中的信息，将特定的神经元集合设为激活状态，模拟大脑中词汇激活的过程。在大脑中，不同的词汇会激活不同的神经元集合，而这些神经元集合的激活状态将影响后续的解析过程。这一步骤与论文中描述的词汇激活机制一致，通过神经元集合的激活来模拟大脑在处理语言时的初始步骤。

```
b.activateWord(LEX, word);
```

4. 应用规则

在激活词汇后，程序会应用该词汇的前置规则和后置规则。前置规则和后置规则分别在词汇激活前后执行，用于调整大脑中不同区域的激活状态和连接权重。例如，前置规则可以解除某些区域的抑制状态，使其准备接收来自其他区域的输入信号，而后置规则可以在词汇处理完成后抑制某些区域，防止其过度激活。这些规则的应用模拟了大脑中不同区域之间的复杂相互作用，是解析器能够正确处理输入句子的关键。这一过程对应于论文中提到的规则应用机制，通过动态调整神经元的激活状态和连接权重来模拟大脑的语言处理功能。

```
for (const auto& rule : lexeme.pre_rules) {
    b.applyRule(rule);
}

for (const auto& rule : lexeme.post_rules) {
    b.applyRule(rule);
}
```

5. project操作

project操作是解析过程中的核心步骤之一。在这个过程中，神经活动从一个区域传递到另一个区域，模拟大脑中信号的传播。这一步骤通过调用 `parse_project` 函数实现，`parse_project` 函数内部调用 `project` 和 `project_into` 函数，负责具体的project操作。`project` 函数确定需要进行的区域，并调用 `project_into` 函数将神经活动从源区域传递到目标区域。在这一过程中，程序会根据当前区域的激活状态和连接权重，计算哪些神经元需要被激活，并更新相应的连接矩阵。这一步骤对应于论文中描述的神经网络模型的信号传播机制，通过模拟信号在大脑不同区域之间的传递，来实现对输入句子的解析。

```
b.parse_project();
```

6. 读取解析结果

在解析完成后，程序需要从大脑模型中提取解析结果。这一步骤通过调用 `read_out` 函数实现，`read_out` 函数根据激活的纤维和区域状态，确定不同区域之间的依存关系。依存关系表示了输入句子中不同词汇之间的语法和语义联系，是解析结果的核心内容。在读取解析结果时，程序会遍历大脑中所有激活的区域，确定其与其他区域之间的连接状态，并根据这些连接状态生成依存关系列表。这一过程模拟了大脑在理解语言时的语法和语义提取机制，是论文中描述的解析器模型的重要组成部分。

```
read_out(b, VERB, activated_fibers, dependencies);
```

7. 输出依存解析

最终，程序将解析得到的依存关系输出到控制台。依存关系表示了输入句子中不同词汇之间的结构和联系，是解析过程的最终结果。通过打印这些依存关系，程序展示了解析器模型的实际效果。这一步骤不仅验证了解析器的功能，还展示了大脑在处理语言时的复杂机制。通过输出依存关系，程序展示了论文中提到的解析器模型的实际应用效果，验证了模型的有效性和准确性。

```
std::cout << "Got dependencies: " << std::endl;
for (const auto& dep : dependencies) {
    for (const auto& word : dep) {
        std::cout << word << " ";
    }
    std::cout << std::endl;
}
```

8. 总结

神经元激活：程序通过 `activateWord` 函数模拟了大脑中词汇激活相应神经元的过程。这与论文中描述的大脑如何处理输入词汇并激活相关神经元集合一致。激活特定词汇对应的神经元集合是解析过程的基础。

规则应用：程序通过前置规则和后置规则定义了词汇在解析过程中的行为，模拟了大脑中不同区域之间的相互作用。论文中提到的解析器模型正是通过这些规则来控制神经元的激活和抑制，实现对输入句子的动态调整和处理。

project操作：程序通过 `project` 和 `project_into` 函数模拟了神经信号在大脑不同区域之间的传播。论文中描述的神经网络模型通过类似的机制来处理语言输入并生成解析结果。信号在不同区域之间的传递和处理是解析器模型实现复杂语言理解任务的关键。

读取解析结果：程序通过 `read_out` 函数提取解析结果，确定不同区域之间的依存关系。这一步模拟了大脑在理解语言时的语法和语义提取过程，是论文中描述的解析器模型的重要组成部分。通过提取和分析依存关系，程序展示了大脑在语言理解中的核心机制。

3.3.4 project操作具体步骤介绍

1. 确定激活区域

函数开始时，首先确定当前激活的区域集合。通过遍历所有大脑区域，检查每个区域的激活状态，收集所有当前激活的区域。这一步骤对应于大脑中，确定哪些区域在当前时刻是活跃的，哪些区域需要在接下来的信号传递中参与处理。这一步模拟了大脑在处理语言时的初始状态设定，确保只有那些真正被激活的区域参与到接下来的处理过程中。这一步骤与论文中描述的大脑初始激活状态设定一致。

```
std::unordered_set<std::string> active_areas;
for (const auto& area_pair : areas) {
    if (!area_pair.second.active_neurons.empty()) {
        active_areas.insert(area_pair.first);
    }
}
```

2. project准备

接下来，函数需要为project的过程做好准备。根据当前激活的区域，确定需要进行投射的区域对。这些区域对通过 `dst_areas_by_src_area` 参数提供，表示从哪些源区域到哪些目标区域需要进行信号传递。函数会遍历这些映射关系，为每个源区域找到其对应的目标区域，准备好投射操作所需的参数。这一步骤模拟了大脑在处理信号传递前的准备工作，确保每个信号传递路径都已经正确设定。

```
std::unordered_map<std::string, std::vector<std::string>> projections;
for (const auto& src_area : active_areas) {
    for (const auto& dst_area : dst_areas_by_src_area.at(src_area)) {
        projections[src_area].push_back(dst_area);
    }
}
```

3. 初始化投射过程

函数开始时，首先初始化投射过程。通过清空目标区域的当前激活状态，为新的信号传递做好准备。这一步确保了目标区域不会受到之前状态的影响，能够正确处理新的输入信号。这一步骤模拟了大脑在接收新的信号之前的状态清理过程，确保信号传递的准确性。论文中描述的解析器模型强调了信号传递前的状态准备，这一步骤正是这一原理的具体实现。

```
target_area.clear_active_neurons();
```

4. 收集输入信号

接下来，函数从源区域收集输入信号。通过遍历 `from_areas` 集合，获取每个源区域的激活状态和连接权重，将这些信息组合成一个综合的输入信号。这一步骤模拟了大脑在接收多个区域的信号时的综合处理过程，通过整合来自不同区域的信息，形成一个统一的输入信号。

```
std::vector<double> combined_input_signal(target_area.n, 0.0);
for (const auto& src_area_name : from_areas) {
    Area& src_area = areas[src_area_name];
    for (int i = 0; i < src_area.n; ++i) {
        if (src_area.active_neurons[i]) {
            for (int j = 0; j < target_area.n; ++j) {
                combined_input_signal[j] += src_area.weights[i][j];
            }
        }
    }
}
```

5. 计算获胜神经元

在收集到输入信号后，函数需要确定目标区域中哪些神经元会被激活。通过计算输入信号的强度，函数选择最强的k个信号，激活对应的神经元。论文中的解析器模型通过类似的竞争机制来确定神经元的激活状态，这一步骤模拟了大脑中神经元的竞争机制，通过选择最强的信号来决定哪些神经元需要被激活。

```
std::vector<int> winning_neurons = nlargest_indices(combined_input_signal, target_area.k);
target_area.activate_neurons(winning_neurons);
```

6. 更新连接权重

函数还需要更新源区域和目标区域之间的连接权重，以反映新的信号传递结果。通过对激活的神经元对进行权重调整，函数实现了大脑的学习和适应过程。这一步骤模拟了大脑中的突触可塑性，通过调整连接权重来适应新的语言信息。

```
for (const auto& src_area_name : from_areas) {
    Area& src_area = areas[src_area_name];
    for (int i = 0; i < src_area.n; ++i) {
        if (src_area.active_neurons[i]) {
            for (const int& winner : winning_neurons) {
                src_area.weights[i][winner] += src_area.beta * (1.0 - src_area.weights[i]
[winner]);
            }
        }
    }
}
```

7. 可塑性调整

在连接权重更新完成后，函数还需要对目标区域的内部连接进行可塑性调整。通过更新目标区域内部的连接权重，函数确保新的激活状态能够正确反映在内部结构中。这一步骤模拟了大脑内部的连接调整过程，通过对内部连接的可塑性调整，确保大脑区域的功能完整性。

```
target_area.update_internal_weights();
```

8. 更新状态

投射操作完成后，函数需要更新大脑区域的状态。对于每个区域，函数会根据投射结果，更新其神经元的激活状态和连接矩阵。这一步骤确保每个区域的状态都正确反映了投射操作的结果，准备好下一步的处理。这一步模拟了大脑在处理信号传递后的状态更新过程，通过动态调整神经元的激活状态和连接权重，确保大脑区域的状态始终与处理过程相匹配。

```
for (auto& area_pair : areas) {
    area_pair.second.update_state();
}
```

最后，函数进行可塑性调整。根据投射操作的结果，函数会更新不同区域之间的连接权重，以模拟大脑的学习和适应过程。通过更新连接权重，函数能够实现大脑中神经连接的动态调整，反映新的语言处理经验。这一步骤对应于论文中提到的神经可塑性，通过调整连接权重，实现对新的语言信息的适应和学习。

```
update_plasticities();
```

3.4 代码优化细节

(1) 字典结构的选择

代码中，为了实现字符串与大脑区域类之间的映射关系，我们使用了大量的`unordered_map`结构。`unordered_map`是 C++ 标准库中的一个关联容器，它提供了键值对的哈希表实现。与传统的 `std::map` 不同，`unordered_map` 使用哈希表来存储元素，因此元素的存取速度更快。`std::map` 使用红黑树实现，提供 O(log n) 的查找、插入和删除时间复杂度，元素按键有序存储。而`unordered_map` 使用哈希表实现，提供 O(1) 的查找、插入和删除时间复杂度，元素无序存储。所以`unordered_map` 更适合频繁查找和插入的场景，这能给我们的代码带来更好的性能。

(2) 随机数生成

在代码的随机数采样中，我们需要限制生成的值在某个范围内，为了生成截断范围内的样本，如果直接采样再丢弃超出范围的值，这种方法效率低下，特别是在截断范围很小的情况下，会丢弃大量不合格的样本。而且如果直接采样并将不合格的值重新生成，会打破正态分布的特性，因为这种方法改变了样本的概率分布，不再是真正的正态分布。所以我们在代码中学习使用了累积分布函数和反向累积分布函数，累积分布函数用于将均匀分布的随机数映射到截断正态分布上，通过反向累积分布函数计算对应的截断正态分布样本值，使用累积分布函数 映射方法可以避免不必要的采样和丢弃过程，提高样本生成的效率。所有生成的样本都是有效的，无需丢弃任何值。而且在生成截断正态分布样本时，能够确保所有样本值都在指定的范围内，并且保持正态分布的形状。

```
double low = cdf(standard_normal, (lower_bound - mean) / stddev);
double up = cdf(standard_normal, (upper_bound - mean) / stddev);
double u = uniform_dist(rng);
double p = low + u * (up - low);
double sample = mean + stddev * quantile(standard_normal, p);
```

(3) 优先队列的使用

优先队列是一种抽象数据结构，支持快速的最大或最小元素访问，每次访问时，总是访问优先级最高或最低的元素。在获取输入向量中最大的k个值的索引中，我们选择使用优先队列进行更快速的获取。从最大堆中提取前 k 个最大元素的索引，由于最大堆的性质，每次提取操作都会返回当前最大的元素，并且时间复杂度仅为O(log n)。

```
std::priority_queue<std::pair<double, int>> max_heap;
for (int i = 0; i < k && !max_heap.empty(); ++i)
{
    new_winner_indices.push_back(max_heap.top().second); // 获取索引
    max_heap.pop();                                      // 移除当前最大元素
}
```

(4) 新数据创建的处理

在实验资料的python代码中，对于某些矩阵的创建，代码中直接使用了对0维numpy数组进行增加行或列的操作，而且由于python自带赋值等于引用的效果，新建立的结构可以很容易的进行传递。但是在cpp代码中，我们是无法使用这种0维结构进行较好的数据创建的，这个问题实际上是由于cpp是强类型的性质所导致的。为了处理这个问题，我们选择了保留最小维度，然后在真正要初始化的时候修改维度来解决，代码如下所示。

```
size_t num_rows = the_other_area_connectome.size(); // 获取行数
size_t num_cols = the_other_area_connectome[0].size(); // 获取列数
if (num_cols == 1)
{
  change--;
}
if (num_first_winners_processed != 0)
{
  for (auto& row : the_other_area_connectome)
  {
    row.resize(row.size() + change, 0.0); // 初始化为0.0
  }
}
```

(5) 类的合并

在代码中，我们需要使用到FiberRule的结构和AreaRule的结构，我们原先使用两个类来分别存储不同类型的规则，这导致了在执行规则的时候我们需要对两个类进行判断，而且两个类不同存储在一起，我们也不得不使用两个结构对这两种不同的规则进行存储，这导致了代码结构混乱，并且增加了额外的维护成本。于是我们在后面代码优化的过程中，提取这两个类的共性，将这两个类合并为了一个类，通过一个简单的bool值来代表不同类型的规则，从而让代码结构简单了很多，两种规则也可以直接存储在一起，不用管理之前面临的顺序存储等问题。

```
class Rule {
public:
    std::string action;
    std::string area1; // 适用于 FiberRule
    std::string area2; // 适用于 FiberRule
    std::string area;  // 适用于 AreaRule
    int index;
    bool flag; // false 表示 AreaRule, true 表示 FiberRule
};
```

三、实验结果

1 项目所支持的单词库

1. **the**：限定词（determinant）
2. **a**：限定词（determinant）
3. **dogs**：名词（noun）
4. **cats**：名词（noun）
5. **mice**：名词（noun）
6. **people**：名词（noun）
7. **chase**：及物动词（transitive verb）
8. **love**：及物动词（transitive verb）
9. **bite**：及物动词（transitive verb）
10. **of**：介词（preposition）
11. **big**：形容词（adjective）
12. **bad**：形容词（adjective）
13. **run**：不及物动词（intransitive verb）
14. **fly**：不及物动词（intransitive verb）
15. **quickly**：副词（adverb）
16. **in**：介词（preposition）
17. **are**：系动词（copula）
18. **man**：名词（noun）
19. **woman**：名词（noun）
20. **saw**：及物动词（transitive verb）

2 Google Test

2.1 代码说明

`getExpectedParseResult`会返回给定句子的预期解析结果，这里存储着句子的正确依赖解析结果，相当于测试的标签。

```
std::vector<std::vector<std::string>> getExpectedParseResult(const std::string& sentence)
{
    std::unordered_map<std::string, std::vector<std::vector<std::string>>> mock_results =
{
        {"cats chase mice", {{"chase", "cats", "SUBJ"}, {"chase", "mice", "OBJ"}}},
        {"people run", {{"run", "people", "SUBJ"}}},
        {"dogs fly", {{"fly", "dogs", "SUBJ"}}},
        {"dogs bite people", {{"bite", "dogs", "SUBJ"}, {"bite", "people", "OBJ"}}},
        {"the cats chase the mice", {{"chase", "cats", "SUBJ"}, {"chase", "mice", "OBJ"},
{"cats", "the", "DET"}, {"mice", "the", "DET"}}},
        {"a man saw a woman", {{"saw", "man", "SUBJ"}, {"saw", "woman", "OBJ"}, {"man",
"a", "DET"}, {"woman", "a", "DET"}}},
        {"big cats chase bad mice", {{"chase", "cats", "SUBJ"}, {"chase", "mice", "OBJ"},
{"cats", "big", "ADJ"}, {"mice", "bad", "ADJ"}}},
        {"cats chase mice quickly", {{"chase", "cats", "SUBJ"}, {"chase", "quickly",
"ADVERB"}, {"chase", "mice", "OBJ"}}},
```

```
        {"cats chase mice in people", {{"chase", "cats", "SUBJ"}, {"chase", "mice",
"OBJ"}, {"mice", "people", "PREP_P"}, {"people", "in", "PREP"}}},
        {"dogs are big", {{"are", "big", "ADJ"}, {"are", "dogs", "SUBJ"}}},
        {"cats are dogs", {{"are", "cats", "SUBJ"}, {"are", "dogs", "OBJ"}}},
        {"the big cats of people chase a bad mice quickly", {{"chase", "cats", "SUBJ"},
{"chase", "quickly", "ADVERB"}, {"chase", "mice", "OBJ"}, {"cats", "people", "PREP_P"},
{"cats", "big", "ADJ"}, {"cats", "the", "DET"}, {"people", "of", "PREP"}, {"mice", "a",
"DET"}, {"mice", "bad", "ADJ"}}}
    };
    return mock_results[sentence];
}
```

`areVectorsEqual` 函数用于比较两个二维字符串向量是否相等，忽略外部向量的顺序。这个函数的主要作用是验证程序的输出和正确结果是否匹配，即判断解析器的输出是否与预期的依赖解析结果相同。

```
bool areVectorsEqual(std::vector<std::vector<std::string>> v1,
std::vector<std::vector<std::string>> v2) {
    auto vecHash = [](const std::vector<std::string>& vec) {
        std::string hashStr;
        for (const auto& str : vec) {
            hashStr += str + "#";
        }
        return std::hash<std::string>{}(hashStr);
    };

    std::unordered_set<size_t> set1, set2;
    for (const auto& vec : v1) {
        set1.insert(vecHash(vec));
    }
    for (const auto& vec : v2) {
        set2.insert(vecHash(vec));
    }

    return set1 == set2;
}
```

`TestSentenceParsing` 函数是一个用于测试句子解析功能的核心函数。它调用解析器的解析方法，比较实际解析结果与预期解析结果，并使用 Google Test 的断言功能来验证解析的正确性。在调用 `TestSentenceParsing` 函数时，比如可能会解析句子 "cats chase mice"，然后获取其预期结果，最后比较实际结果与预期结果。如果解析器的输出与预期结果匹配，则测试通过；否则，测试失败并输出相应的错误信息。

```
// 测试解析器是否能够解析给定的句子
void TestSentenceParsing(const std::string& sentence) {
    std::vector<std::vector<std::string>> result = parse(sentence);  // 调用解析方法
    std::vector<std::vector<std::string>> expected = getExpectedParseResult(sentence);  //
获取预期结果
    if(!areVectorsEqual(expected, result)) {
        cout << "result:" << endl;
```

```
        for (const auto& dep : result)
        {
            for (const auto& word : dep)
            {
                std::cout << word << " ";
            }
            std::cout << std::endl;
        }
    }
    EXPECT_TRUE(areVectorsEqual(expected, result)) << "Sentence failed to parse correctly:
" << sentence;
}
```

这里定义了多个测试案例来验证解析器对各种类型句子的处理能力。测试分为几类：简单句、带有限定词的句子、带有形容词的句子、带有副词的句子、带有介词的句子、带有连系动词的句子和更复杂的句子。通过这些测试，能够全面验证解析器在各种句法结构下的表现，确保解析器能够正确解析和理解不同类型的句子。

```
// 测试简单句
TEST(ParserTest, SimpleSentences) {
    TestSentenceParsing("cats chase mice");
    TestSentenceParsing("people run");
    TestSentenceParsing("dogs fly");
    TestSentenceParsing("dogs bite people");
}

// 测试带有限定词的句子
TEST(ParserTest, SentencesWithDet) {
    TestSentenceParsing("the cats chase the mice");
    TestSentenceParsing("a man saw a woman");
}

// 测试带有形容词的句子
TEST(ParserTest, SentencesWithAdjectives) {
    TestSentenceParsing("big cats chase bad mice");
}

// 测试带有副词的句子
TEST(ParserTest, SentencesWithAdverbs) {
    TestSentenceParsing("cats chase mice quickly");
}

// 测试带有介词的句子
TEST(ParserTest, SentencesWithPrepositions) {
    TestSentenceParsing("cats chase mice in people");
}

// 测试带有连系动词的句子
TEST(ParserTest, SentencesWithCopulas) {
    TestSentenceParsing("dogs are big");
    TestSentenceParsing("cats are dogs");
}
```

```
// 测试更复杂的句子
TEST(ParserTest, ComplexSentences) {
    TestSentenceParsing("the big cats of people chase a bad mice quickly");
}
```

2.2 测试结果

```
xwr@xwr-virtual-machine:~/Desktop/para_prog8$ g++ -std=c++17 -isystem /usr/include/gtest -pthread test_parse.cpp /usr/lib/libgtes
t.a /usr/lib/libgtest_main.a -o test_parse
xwr@xwr-virtual-machine:~/Desktop/para_prog8$ ./test_parse
[==========] Running 7 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 7 tests from ParserTest
[ RUN      ] ParserTest.SimpleSentences
[       OK ] ParserTest.SimpleSentences (2192 ms)
[ RUN      ] ParserTest.SentencesWithDet
[       OK ] ParserTest.SentencesWithDet (1469 ms)
[ RUN      ] ParserTest.SentencesWithAdjectives
[       OK ] ParserTest.SentencesWithAdjectives (1408 ms)
[ RUN      ] ParserTest.SentencesWithAdverbs
[       OK ] ParserTest.SentencesWithAdverbs (910 ms)
[ RUN      ] ParserTest.SentencesWithPrepositions
[       OK ] ParserTest.SentencesWithPrepositions (1170 ms)
[ RUN      ] ParserTest.SentencesWithCopulas
[       OK ] ParserTest.SentencesWithCopulas (1339 ms)
[ RUN      ] ParserTest.ComplexSentences
[       OK ] ParserTest.ComplexSentences (2148 ms)
[----------] 7 tests from ParserTest (10637 ms total)

[----------] Global test environment tear-down
[==========] 7 tests from 1 test case ran. (10637 ms total)
[  PASSED  ] 7 tests.
xwr@xwr-virtual-machine:~/Desktop/para_prog8$
```

可以看到，以上测试全部通过，这表明我们的实现的解析器能够正确解析各种不同的句子结构。每个测试案例都验证了解析器在特定句法结构下的解析准确性和可靠性，而且在面对复杂的自然语言句子时也具有良好的解析能力。

3 性能测试

3.1 python代码性能

```
(base) C:\Users\lenovo\Desktop\assemblies>E:/Miniconda/python.exe c:/Users/lenovo/Desktop/assemblies/brain_parser.py
input sentence: "cats chase mice"
解析用时：3.6807 秒
平均每个单词用时：1.2269 秒每词
input sentence: "people run"
解析用时：2.4204 秒
平均每个单词用时：1.2102 秒每词
input sentence: "dogs fly"
解析用时：2.1893 秒
平均每个单词用时：1.0946 秒每词
input sentence: "dogs bite people"
解析用时：3.3939 秒
平均每个单词用时：1.1313 秒每词
input sentence: "the cats chase the mice"
解析用时：4.2829 秒
平均每个单词用时：0.8566 秒每词
input sentence: "a man saw a woman"
解析用时：4.3979 秒
平均每个单词用时：0.8796 秒每词
input sentence: "big cats chase bad mice"
解析用时：7.7396 秒
平均每个单词用时：1.5479 秒每词
input sentence: "cats chase mice quickly"
解析用时：5.8661 秒
平均每个单词用时：1.4665 秒每词
input sentence: "cats chase mice in people"
解析用时：8.9880 秒
平均每个单词用时：1.7976 秒每词
input sentence: "dogs are big"
解析用时：4.4320 秒
平均每个单词用时：1.4773 秒每词
input sentence: "cats are dogs"
解析用时：4.4324 秒
平均每个单词用时：1.4775 秒每词
input sentence: "the big cats of people chase a bad mice quickly"
解析用时：14.3070 秒
平均每个单词用时：1.4307 秒每词
每个句子平均每个单词用时的平均值：1.2997 秒每词
```

3.2 cpp代码性能

```
input sentence: cats chase mice
解析用时: 0.51623 秒
平均每个单词用时: 0.172077 秒每词
input sentence: people run
解析用时: 0.335281 秒
平均每个单词用时: 0.16764 秒每词
input sentence: dogs fly
解析用时: 0.336648 秒
平均每个单词用时: 0.168324 秒每词
input sentence: dogs bite people
解析用时: 0.536766 秒
平均每个单词用时: 0.178922 秒每词
input sentence: the cats chase the mice
解析用时: 0.591101 秒
平均每个单词用时: 0.11822 秒每词
input sentence: a man saw a woman
解析用时: 0.593684 秒
平均每个单词用时: 0.118737 秒每词
input sentence: big cats chase bad mice
解析用时: 0.989805 秒
平均每个单词用时: 0.197961 秒每词
input sentence: cats chase mice quickly
解析用时: 0.685104 秒
平均每个单词用时: 0.171276 秒每词
input sentence: cats chase mice in people
解析用时: 0.960238 秒
平均每个单词用时: 0.192048 秒每词
input sentence: dogs are big
解析用时: 0.532628 秒
平均每个单词用时: 0.177543 秒每词
input sentence: cats are dogs
解析用时: 0.500465 秒
平均每个单词用时: 0.166822 秒每词
input sentence: the big cats of people chase a bad mice quickly
解析用时: 1.73198 秒
平均每个单词用时: 0.173198 秒每词
每个句子平均每个单词用时的平均值: 0.166897 秒每词

D:\集合演算cpp\x64\Debug\集合演算cpp.exe (进程 24848)已退出，代码为 0。
按任意键关闭此窗口. . .
```

可以看到，cpp代码的运行速度是python代码的7倍以上，cpp代码性能更加良好是由于多个因素共同作用的结果。首先，主要的原因是cpp是一种编译型语言，代码在执行之前需要经过编译过程生成机器码，编译器在这个过程中会进行大量的优化，使得生成的机器码运行速度非常快。相比之下，Python 是一种解释型语言，代码在执行时需要通过解释器逐行解释执行，解释执行的开销较大，因此运行速度通常比编译后的机器码慢。此外，cpp是静态类型语言，在编译时进行类型检查和确定，运行时不需要额外的类型检查开销，从而提高了执行效率。而python是动态类型语言，在运行时进行类型检查，这增加了开销，降低了运行速度。同时，我们也充分利用了cpp提供了更高效的数据结构选择，如使用unordered_map实现字符串与大脑区域类之间的映射、使用优先队列更快速地进行索引获取等等，这些结构上的优化也使得我们cpp代码的速度得到提升。另外我们在算法上的优化，比如随机数生成速度的加速、冗余操作的删除等等，这些对性能提升有很大的帮助，使得执行次数得到了显著减少。综合这些因素，使得cpp代码在性能上得到了显著提升，最终使得运行速度远快于python代码。

4 cmake

4.1 运行Google Test文件的方法

```
cd google_test
mkdir -p build
cd build
cmake ..
make
./parser_project
```

4.2 运行依赖解析文件的方法

```
cd src
mkdir -p build
cd build
cmake ..
make
./project
```

由于本次实验我们利用了macOS、Linux、Windows等多个平台，我们发现不同平台的cmake命令会有所差异，上面的运行命令基于Linux，如果有项目运行上的问题，我们随时都愿意进行解答和讨论。感谢您的阅读！