📖 РУКОВОДСТВО: МОДЕЛЬ ВИЗУАЛЬНОГО ПРОГРАММИРОВАНИЯ UI для SOCRATES

Оглавление

ЧАСТЬ 1. Ответы на ключевые вопросы архитектуры

1. Какова модель управления? (Хаос или порядок?)

Модель управления в нашей системе — классическая Событийно-Ориентированная (Event-Driven Architecture), точно такая же, как в Windows Forms, iOS или веб-разработке.

В Blockly блоки не выполняются «сверху вниз всей кучей».

Каждая логическая цепочка начинается со «Шляпного блока-слушателя» (блок с круглой верхушкой и выемкой внутри):

- [🖱 При клике на элемент с ID ...]
- [🧠 Когда от Socrates AI пришел ответ]

Как это работает:

Когда веб-страница загружается, все эти цепочки регистрируются как отдельные изолированные обработчики событий и «засыпают».

- Пользователь нажал кнопку #btn_start? Просыпается только цепочка для этой кнопки.
- ИИ прислал ответ? Просыпается только цепочка ответа ИИ.

Они не мешают друг другу и работают независимо.

2. Как связываются элементы UI с кубиками? (Понятие «ID — это номер телефона»)

Связующим звеном между Дизайном и Логикой является ID элемента.

- В Дизайне вы даете элементу имя (например, кнопке даете ID: btn_send, а карточке — ID: result_card).
- В Логике кубики используют этот ID как «номер телефона», чтобы:
  - Слушать событие: [🖱 При клике на ID "btn_send”]
  - Считать данные (ВХОД): (🕹 Значение из поля с ID "input_text”)
  - Записать данные (ВЫХОД): [✍ Установить текст в ID "result_card" = ...]
  - Изменить состояние: [👁 Показать элемент с ID "result_card”]

ЧАСТЬ 2. Анатомия кубиков (Формы и их назначение)

Чтобы не путаться, кубики разделены по формам (как в пазлах):

1. ШЛЯПНЫЙ БЛОК (Событие / Триггер):

```
╔═══════════════════════════════════════════════════════╗
║ 🖱 При клике на ID "btn_send" ║
╠═══════════════════════════════════════════════════════╝
║ (Сюда вставляются действия, которые ║
║ должны произойти при нажатии) ║
╚═══════════════════════════════════════════════════════╝
```

2. БЛОК ДЕЙСТВИЯ (Команда / Выход):

```
├── [👁 Скрыть элемент с ID "panel_warning”] ──┤
```

├── [🧠 Отправить в ИИ текст: (...)] ──────────┤

3. КРУГЛЫЙ БЛОК ДАННЫХ (Значение / Вход):

(🕹️ Значение из поля с ID "user_prompt") ──> Вставляется в гнезда других блоков!

(💬 Текст ответа ИИ)

Создание своих виджетов ⭐:

- Соберите в Дизайне, например, карточку с кнопкой и статусом.
- Кликните по ней -> нажмите на появившуюся звездочку ⭐.
- Введите имя (например "Мой пульт ЧПУ").
- Откройте панель справа: там появилась категория ⭐ Мои виджеты, откуда этот готовый пульт можно добавлять куда угодно!

Живой визуальный отладчик ⚡:

- Откройте вкладку 🧩 Логика.
- Запустите сервер (желтая стрелка) и откройте страницу в браузере.
- Нажмите кнопку на веб-странице — и посмотрите на экран Студии: соответствующий кубик вспыхнет зеленым неоновым светом! Вы вживую видите работу логики!

ЧАСТЬ 3. Три практических примера (От простого к сложному)

ПРИМЕР 1: Локальная логика интерфейса (Без ИИ)

Задача: Врач нажимает на кнопку [Показать расширенную карту], открывается скрытая панель с анализами, а текст на кнопке меняется.

Шаг 1. В Дизайне (GrapesJS):

1. Вы перетаскиваете 🟢 Кнопку. Во вкладке ⚙️ Шестеренка задаете ID: btn_toggle. Текст: "Показать карту".
2. Вы перетаскиваете 📦 Карточку с шапкой. В ⚙️ Шестеренке задаете ID: medical_card.
3. Во вкладке 🎨 Стили для этой карточки ставите: Display = None (скрыта по умолчанию).

Шаг 2. В Логике (Blockly):

Собираем цепочку:

1. Берем блок: [🖱️ При клике на элемент с ID "btn_toggle"]
2. Внутрь него вставляем: [👁️ Переключить (Toggle) элемент с ID "medical_card"]

Результат: Нажатие кнопки мгновенно открывает/закрывает карточку прямо в браузере без обращения к ИИ.

ПРИМЕР 2: Диалог с ИИ (Ассистент / Чат)

Задача: Пользователь пишет вопрос в поле, нажимает кнопку -> вопрос улетает в Socrates AI -> ответ появляется в блоке диалога.

Шаг 1. В Дизайне (GrapesJS):

1. Большое текстовое поле -> задаем ID: user_question.
2. Кнопка действия -> задаем ID: btn_ask (текст: "Спросить").
3. Окно диалога с ИИ -> у него по умолчанию уже есть ID: ai_chat_messages.

Шаг 2. В Логике (Blockly):

Мы создаем ДВЕ независимые цепочки:

Цепочка А (Отправка вопроса):

1. Блок: [При клике на элемент с ID "btn_ask”]
2. Внутрь: [Добавить в Чат с ID "ai_chat_messages" текст: (Значение из поля с ID "user_question") Автор: Пользователь]
3. Внутрь: [Отправить в Socrates AI текст: (Значение из поля с ID "user_question") Режим: 0]
4. Внутрь: [Установить текст в ID "user_question" = "“] (очищаем поле ввода)

Цепочка Б (Прием ответа от ИИ):

1. Блок: [Когда от Socrates AI пришел ответ]
2. Внутрь: [Добавить в Чат с ID "ai_chat_messages" текст: (Текст ответа ИИ) Автор: Socrates AI]

Результат: Красивый полноценный чат. Сообщения пользователя уходят направо (синие), ответы ИИ приходят налево (белые), поле очищается, чат сам прокручивается.

ПРИМЕР 3: Промышленный пульт ЧПУ / Экзаменатор (С условиями «Если-То»)

Задача: Оператор выбирает режим работы из выпадающего списка. Если выбран "Аварийный тест" — запустить специальный SHP-скрипт и подсветить ошибку, иначе — отправить обычную команду.

Шаг 1. В Дизайне (GrapesJS):

1. Выпадающий список (Select) -> ID: select_mode.
2. Кнопка -> ID: btn_run.
3. Индикатор статуса -> ID: status_error.

Шаг 2. В Логике (Blockly):

1. Блок: [При клике на элемент с ID "btn_run”]
2. Внутрь вставляем блок условий из категории Условия:

- [Если (Значение из поля с ID "select_mode”) == "2”]
  То:
- [Выполнить SHP-скрипт на сервере: "Brain.TimerPause();”]
- [Показать элемент с ID "status_error”]
  Иначе:
- [Отправить в Socrates AI текст: "Запуск штатного цикла" Режим: 1]

Результат: Настоящая автоматизация! Логика ветвится в зависимости от того, что выбрал оператор на экране.

ЧАСТЬ 4. Памятка / Золотые правила для создателя

1. Правило осмысленных ID:

Никогда не оставляйте стандартные имена, если собираетесь управлять элементом. Давайте понятные имена: btn_start, input_patient_age, table_diagnostics.

2. Разделение экрана:

Во вкладке 🎨 Дизайн вы отвечаете только за «Как это выглядит» (красота, отступы, шрифты).

Во вкладке 🧩 Логика вы отвечаете только за «Как это ведет себя» (поведение, события, связь с нейросетью).

3. Независимость блоков:

Располагайте цепочки на сером холсте Blockly свободно. Они могут лежать рядом друг с другом в любом порядке — браузер сам поймет, какую цепочку, когда запускать!

1. Как расширить состав виджетов и кубиков управления (языка кубиков)?

Socratis использует стандарт описания этих элементов на JS введенный корпорацией Google. Материалы об этом широко представлены сети Интернет. Для добавления своих элементов достаточно разместить их JS- код в соответствующие свойства Системного Нейристора. Смотрите раздел: Свойства системного нейрона (System:System)

2. Как передать готовый продукт пользователю?

Встроенный компилятор создает папку переносимого приложения, которую нужно разместить на флэшке, или в не защищенной от перезаписи области, например в Мои документы, или в свободной для доступа зоне диска пользователя.

После того как вы скомпилировали готовое приложение (кнопка «ракета»), найдите папку C:\Users\Ваше имя\Documents\SocratesRun.

Скопируйте, и, если нужно — архивируйте ее.

Для запуска этого приложения с флэшки или из папки Документы пользователя достаточно запустить в ней файл:

Socrates.Studio.exe

Часть 1: Что такое Socrates

Socrates – интерактивный язык который можно рекурсивно применять при описании знаний и правил работы с ними, а так-же при описании архитектуру Графово-Символьной Нейронной Сети на любом уровне архитектуры. Он легко встраивается, и также может быть применим для других проектов – как отдельный интерпретатор и оркестратор вашей архитектуры. Язык, реализованный через нашу DLL, имеет богатый API для чисто программного вызовы всех его функций, включая вызов самого интерпретатора.

Все выражения Socrates представляют собой строки текста.

Socrates оперирует со следующими основными типами данных:

- Выражение Socrates – некоторый скрипт в форме текстовой строки
- Текст – строка или файл произвольного содержания на любом естественном или программном языке.
- Изображение – обычно файл, содержащий графическую структуру в одном из основных форматов графики.
- Речь – звуковой файл, содержащий речевое сообщение на одном из 63-х языков мира.
- GNN – вся текущая нейронная сеть (не в смысле LLM, а граф концептов)
- Нейрон – узел графа, контейнер, содержащий описание Концепта и его реализации
- Синапс – ребро графа, описывающее характер допустимого взаимодействия между Концептами

- Концепт – антологическое описание нейрона
- Сигнал – сложное (комплексное) сообщение, поступающее от одного концепта к другому

- Правило – некоторое выражение Socrates состоящее из Потерна и Вывода
- gnn_Контекст – состояние сети
- Функция \ Метод – в том-же применении что и в C#

Также все обычные типы: int, float, bool, string, array, list, dictionary и т. п.

Особенности синтаксиса и добавленные функции

Socrat-3 имеет синтаксис аналогичный C#, но со следующими упрощениями и особенностями:

- Описывать типы переменных не обязательно.
- Добавлены типы Rule, Rules, Neuron, Synapse, Attention, Message
- Объявление методов и функций, немного отличается от C#, и возможно только для глобального уровня в нейристоре Sectem свойство: _Global
- В нейристоре Sectem свойство: _Script - также может быть использовано для объявления, но оно будет использовано при запуске системы, после _Global

1. Объявление глобальных методов

```
void Name(аргументы)
{тело функции};
```

2. Объявление глобальной функций

```
result Name(аргументы)
{тело функции};
```

3. Объявление лямбда функций

```
Name = (аргументы) => {тело функции};
```

4. Объявление глобальных именованных правил

```
rule Name {образец => вывод};
```

5. Объявление лямбда правил

```
X = rule {образец => вывод};
```

6. Оператор return выражение;

кроме всех стандартных вариантов типа return; return value,

допускается добавление условия для return возвращающего результат выражения, с одним или двумя ветвями:

```
return (x==true):: выражение1; // при false return не выполняется
return (x==true):: выражение1:: выражение2; // при false возвращается результат 2
```

Пример: описание и применения функции Факториал:

```
result fct (n) {
return (n>1):: n * fct(n — 1):: 1;};
```

Применение:

```
x= fct(7);
```

7. Типы данных

Socrates-3 работает со следующими основными типами данных:

число (int, float, double) – автоматическое определение по формату записи, или явное указание типа.

bool – false, true, <0.7, >0.7, 0, 1

string – "произвольный текст в кавычках"

neuron – единица графовой памяти

synapse – единичная направленная связь от активного нейрона к нейрону приемнику

signal – сообщение, переданное по синапсу (например: параметр, фраза, формула, текстовое описание)

symbol – смысловой элемент сообщения (сигнала), например: число, слово

rule – правило действия (или преобразования сигнала), хранящееся в памяти нейрона.

* Нейрон хранит свои правила следующим образом:

key – должен содержать уникальное, краткое смысловое описания правила на любом языке (включая мат. Формулу)

value – должен содержать само правило реализующее описание в форме записи Socrates-3

success – автоматически регистрирует и выдает (по запросу) успешность выбора, указанного через key правила для нейрона

8. Конвейер функций

Конвейер функций позволяют последовательно применять разные функции или агенты к результату предыдущего вызова. Это делает многоэтапные вычисления более гибкими и наглядными.

Пример:

```
result double1(a) {
a * 2
}
result addFive(a) {
a + 5
}
double1(2)
|- addFive
|- double1; // 14
```

1. Конвейер функций с условиями

В конвейере функции могут иметь условия выполнения:

```
double1(2)
    |- (4>5) addFive // не выполняется
    |- double; // 4
```

9. Дополнительные параметры функции в конвейере.

Функциям конвейера можно передавать дополнительные параметры, по их имени:

```
|- func(argum_Name: 12)
```

10. Варианты конвейерных операций

Каждая операция конвейера после ее выполнения может передавать далее различные варианты сообщений.

|- f – строго передает только результат

|+ f – добавляет результат если он есть к тексту входного параметра спереди

(параметр _FCPInfo по умолчанию "Информация:\n")

|= f – передает результат если он есть, иначе передает входной параметр

|> f – выполняет действие и передает не измененный входной параметр

|- f(params) – передает функции дополнительные параметры (любое число, которое

принимает агент)

|- (condition) f ; – перед вычислением функции проверяет условие или пропускает ее

11. Декораторы @func

Декораторы позволяют изменять (декорировать) поведение уже существующих функций. Декоратор — это функция с как минимум одним аргументов, который представляет другую функцию. Во время применения декоратор должен находиться непосредственно на строку выше декорируемой функции и без «;» за ним. В данных примерах декорируется функция sum:

1. Пример 1: Декоратор без аргументов

```
result sum(a, b) {
a+b}
result decorator1(f) {
x=f.Eval();
f.ToString() + "=" + x; }
@decorator1
sum(3,5); // sum(3,5) = 8
```

2. Пример 2: Декоратор без аргументов с присваиванием

```
@decorator1
a =sum(3,5); // sum(3,5) = 8
```

3. Пример 3: Декоратор с аргументами

```
result decorator2(f, t) {
x=f.Eval();
t + f.ToString() + "=" + x; }
@decorator2("Result: ")
sum(3,5); // Result: sum(3,5) = 8
```

12. Встроенная коллекция текстов (InputText)

Встроенная коллекция текстов позволяет накапливать различные тексты, например введенные через Markdown прямо из скрипта, следующим образом:

```
#
текст
#End - доступен: InputTextx["_"];
# name text
текст
#End - доступен: InputTextx["name text"];
```

Базовый синтаксис C#

Все остальные типы аналогичны базовому C#

1. Нативные расширения

Вы можете подключить любые скомпилированные на C# DLL содержащие функции и классы расширения синтаксиса языка. И нужно просто поместить в папку, указанную в нейристоре System в свойстве _UsingDLLFolder, - по умолчанию /Plugins

2. Диалоговые операторы

В скриптах, особенно когда вы будете создавать собственные расширения \инструменты, вам понадобятся диалоговые запросы к пользователю Socratis.Studio

ВНИМАНИЕ: не пытайтесь применять эти операторы в целевом приложении для диалога с клиентом (за пределами Socratis.Studio). Они предназначены исключительно для расширения функций самой Socratis.Studio.

Для целевого приложения используйте только средства конструктора UI интерфейса.

1. Кнопки выбора

Вы можете создать запрос с любым количеством кнопок выбора вариантов:

```
result = ButtonDialog("Надпись","вопрос","111","222","333");
```

2. Запрос текста

Выводит сообщение и ждет ввода пользовательского текста:

```
result = InputDialog("Надпись","вопрос","по умолчанию");
result = InputDialog("Надпись”, “вопрос");
```

3. Выбор файла

```
selectedFile = PickFileDialog("Выберите файл проекта Socrates");
```

4. Описание файла

Позволяет описать файл и его данные.

Возвращает массив из 4 элементов: [Название, Категория, Источник, Описание]

3. Умный селектор текста SmartSwith

SmartSwith(script, key);

SmartSwith(script, key); вычисляет строковое выражение DSL относительно входного значения key. В зависимости от префикса script оператор может вернуть литеральное значение, значение глобальной переменной, результат выполнения скрипта или результат выбора одного из вариантов.

Результат имеет тип object?, поэтому может быть строкой, числом, логическим значением, объектом, null или пустой строкой.

Порядок обработки

Оператор проверяет выражение в следующем порядке:

1. null.
2. Скриптовый макрос @(...).
3. Точное сопоставление @swith:.
4. Семантическое сопоставление @like:.
5. Сопоставление по наличию текста @contains: или фактически распознаваемого варианта @contain:.
6. Скрипт с префиксом @ — символ @, за которым следует пробел.
7. Имя глобальной переменной, начинающееся с @.
8. Обычный текст, который возвращается без изменений.

Пробелы в начале выражения не влияют на распознавание префикса, поскольку перед проверкой используется Trim(). Однако само значение обычно не очищается от пробелов перед возвратом.

Поддерживаемые формы

Литеральное значение

Если script не начинается с одного из специальных префиксов, он возвращается как есть.

```
csharp
SmartSwith("Привет", "любой ключ");
```

Результат:

```
text
Привет
```

Аргумент key в этом случае не используется.

Если передать значение null, результатом также будет null:

```
csharp
SmartSwith(null, "key");
```

Результат:

```
text
null
```

1. Глобальная переменная

Форма:

```
text
@ИмяПеременной;
```

или:

```
text
@ИмяПеременной
```

После удаления символа @ имя передается в механизм поиска глобальных переменных.

Пример:

```
csharp
SmartSwith("@ApplicationName;", "ignored");
```

Если глобальная переменная ApplicationName имеет значение SmartSwitch DSL, результатом будет:

```
text
SmartSwitch DSL
```

Если переменная не найдена, оператор возвращает исходное имя без символа @. При наличии завершающей точки с запятой она сохраняется:

```
csharp
SmartSwith("@Unknown;", "ignored");
```

Фактический результат:

```
text
Unknown;
```

Таким образом, отсутствие переменной не приводит к исключению. Выражение превращается в обычный текст без маркера @.

2. Произвольный скрипт @(...)

Форма:

text

@(выражение)

Предназначена для вычисления произвольного DSL-скрипта.

Пример предполагаемого использования:

csharp

SmartSwith("@(2 + 3)", "ignored");

Предполагаемый результат:

text

5

Перед выполнением из выражения удаляются комментарии. Если скрипт не заканчивается точкой с запятой, она добавляется автоматически.

Внутренне выражение преобразуется примерно так:

text

@(2 + 3)

в:

text

2 + 3);

Затем оно передается интерпретатору скриптов.

Важно: текущая реализация удаляет последовательность @(, но не удаляет закрывающую скобку). Поэтому выражение @(2 + 3) может быть передано интерпретатору в синтаксически некорректном виде. Если интерпретатор не поддерживает такой формат, макрос необходимо исправить либо документировать без закрывающей скобки.

При невозможности вычисления выражения возвращается преобразованный текст скрипта, например:

text

2 + 3);

3. Скрипт с префиксом @

Форма:

text

@ выражение

В отличие от @(...), здесь после символа @ обязательно должен находиться пробел.

Пример:

csharp

SmartSwith("@ 2 + 3", "ignored");

Перед выполнением маркер @ удаляется:

text

2 + 3

Если отсутствует завершающая точка с запятой, оператор добавляет ее:

```
text
2 + 3;
```

Затем выражение передается в интерпретатор скриптов.

Этот вариант не имеет проблемы с лишней закрывающей скобкой и может использоваться для коротких выражений:

```
text
@ UserName;
@ Price * Count;
@ IsEnabled ? "Да" : "Нет";
```

Выбор значения по ключу

4. Точное совпадение: @swith:

Формат:

```
text
@swith: вариант1=> результат1; вариант2=> результат2; значениеПоУмолчанию
```

Каждая пара имеет вид:

```
text
условие=> результат
Последний элемент без => рассматривается как значение по умолчанию.
```

Пример:

```
csharp
SmartSwith(
"@switch: red=> Красный; green=> Зеленый; blue=> Синий; Неизвестный цвет",
"green");
```

Результат:

```
text
Зеленый
```

Алгоритм:

1. Выражение после @swith: разделяется по символу;.
2. Каждый непустой элемент разделяется по последовательности =>.
3. Левая часть сравнивается с key.
4. При сравнении у обеих сторон удаляются начальные и конечные пробелы.
5. При первом точном совпадении возвращается правая часть.
6. Если совпадений нет, возвращается значение без =>.
7. Если значения по умолчанию нет, возвращается пустая строка.

Пример со значением по умолчанию:

```
csharp
SmartSwith(
"@switch: admin=>Администратор; user=>Пользователь; Гость",
"operator");
```

Результат:

```
text
Гость
```

Сравнение чувствительно к регистру:

```
csharp
SmartSwith(
"@switch: yes=>Да; no=>Нет; Неизвестно",
"YES");
```

Результат:

```
text
Неизвестно
```

Поскольку в коде используется обычное сравнение строк, yes и YES считаются разными значениями.

Особенности синтаксиса

Правая часть не очищается от пробелов:

```
text
@swith: yes=> Да
```

может вернуть строку:

```
text
" Да"
```

с начальным пробелом.

Разделитель => должен встречаться ровно один раз. Конструкция с дополнительным разделителем:

```
text
@swith: a=>b=>c; default
```

не считается корректной парой и будет обработана как значение по умолчанию, фактически дав результат:

```
text
a
```

5. Сопоставление по наличию текста: @contains:

Предполагаемый формат:

```
text
@contains: фрагмент1=>результат1; фрагмент2=>результат2; значениеПоУмолчанию
```

Пример:

```
csharp
SmartSwith(
"@contains: домашний кот=>Животное; красный автомобиль=>Транспорт; Неизвестно",
"кот");
```

Смысл проверки в текущей логике — проверить, содержит ли левая часть условия значение key:

```
text
```

```
"домашний кот".Contains("кот")
```

Так как условие выполняется, результатом должно быть:

```
text
Животное
```

Другой пример:

```
csharp
SmartSwith(
"@contains: Минск является столицей Беларуси=>Беларусь; Нет совпадения",
"столицей");
```

Результат:

```
text
Беларусь
```

При проверке key предварительно очищается от пробелов:

```
text
" кот " → "кот"
```

Левая часть условия очищается от пробелов только при разборе пары, но сама проверка Contains выполняется над полученной строкой.

Если совпадений нет, используется значение по умолчанию:

```
csharp
SmartSwith(
"@contains: красный=>Красный; зеленый=>Зеленый; Другой цвет",
"синий");
```

Результат:

```
text
Другой цвет
```

Если значение по умолчанию отсутствует, результатом будет пустая строка.

6. Семантическое сопоставление: @like:

Формат:

```
text
@like: описание1=>результат1; описание2=>результат2; значениеПоУмолчанию
```

Для каждой пары вычисляется оценка смысловой близости между левой частью и key:

```
text
оценка = Relanker.GetScore(условие, key)
```

Предполагаемая цель — выбрать наиболее похожее условие.

Пример нормативного поведения:

```
csharp
SmartSwith(
"@like: столица Беларуси=>Минск; столица России=>Москва; Неизвестный город",
"главный город Беларуси");
```

Если модель семантической близости считает вариант столица Беларуси наиболее подходящим и его оценка не меньше 0.6, ожидаемый результат:

```
text
Минск
```

Порог 0.6 зафиксирован в реализации:

```
text
score < 0.6
```

Если максимальная оценка ниже порога, должно использоваться значение по умолчанию:

```
csharp
SmartSwith(
"@like: столица Беларуси=>Минск; столица России=>Москва; Неизвестный город",
"погода на завтра");
```

Ожидаемый результат:

```
text
Неизвестный город
```

При вычислении учитываются только элементы, распознанные как пары условие=>результат. Первый элемент без => немедленно считается значением по умолчанию, после чего дальнейший разбор прекращается.

Часть 2: Умная Гибридная Логика (Smart Hybrid Logic)

1. Введение

В классическом программировании логические выражения строго ограничены двумя состояниями: Истина (true) и Ложь (false). В реальном мире знания часто бывают неполными, вероятностными или зависят от контекста.

Наш скриптовый движок использует Умную Гибридную Логику. Она позволяет в рамках одного выражения без явного приведения типов объединять:

1. Четкую логику (bool): классические true / false.
2. Троичную логику Клини (TernaryLogic): добавляет состояние неопределенности Unknown.
3. Нечеткую логику Заде (double): степень уверенности от 0.0 до 1.0.
4. Семантическую логику (string): автоматическое вычисление смысла текста через нейросетевые эмбеддинги.

Движок сам определяет типы входных данных и выбирает правильный математический аппарат под капотом.

2. Правило доминантности (Как определяются типы)

Когда вы смешиваете разные типы в одном выражении (например, bool and double), система автоматически возвращает наиболее "богатый" тип данных:

1. Высший приоритет (double / string): Если в выражении есть нечеткое число или строка, результат будет нечетким числом [0.0 ... 1.0].
2. Средний приоритет (TernaryLogic): Если в выражении участвует константа Unknown, а чисел/строк нет, результат будет троичным (True, False или Unknown).
3. Низший приоритет (bool): Если все операнды строгие bool, результат останется строгим bool.

Никаких ошибок приведения типов (Type Casting Error)! Система всё сделает за вас.

3. Таблица операторов

Текстовый оператор	Классическое действие	Нечеткое/Семантическое действие
and	Строгое И	Выбирает минимум (Min). Для строк: семантическое пересечение.
or	Строгое ИЛИ	Выбирает максимум (Max).
not	Отрицание	Инверсия уверенности (1.0 - x).
xor	Исключающее ИЛИ	Нечеткий XOR.
implies	Следование (Если A, то B)	Степень семантического включения (например, понятие "Кот" входит в "Животное").
equiv	Эквивалентность	Авто-косинусное сходство. Насколько смысл текста A похож на текст B.

(Примечание: Доступны также операторы nand и nor)

4. Примеры использования (От простого к магии)

1. Классическая Булева логика

Если вы передаете bool, скрипт работает как обычный C#.

```
codeC#
var isValid = true and false;
// Результат: bool (false)
```

2. Троичная логика (Работа с неизвестностью)

В скриптах доступна глобальная константа Unknown (Неизвестно). Идеально для работы с неполными графами знаний.

```
codeC#
bool systemReady = true;
var decision = systemReady and Unknown;
// Результат: TernaryLogic (Unknown). Истина И Неизвестность = Неизвестность.
var badDecision = false and Unknown;
// Результат: TernaryLogic (False). Ложь перекрывает всё.
```

3. Нечеткая логика (Вероятности)

Работа с весами и уверенностью.

```
codeC#
double temperatureHot = 0.8;
double pressureHigh = 0.4;
var alarmLevel = temperatureHot and pressureHigh;
// Результат: double (0.4). Оператор 'and' работает как функция Min().
```

4. Семантическая логика (Магия смыслов)

Если операторы equiv или implies видят две строки, они автоматически измеряют их смысловую близость!

```
codeC#
string userRequest = "Хочу купить машину";
// Движок сам преобразует строки в векторы и найдет косинусное сходство!
var intention = userRequest equiv "Покупка автомобиля";
// Результат: double (например, 0.92)
var isAnimal = "Тигр" implies "Млекопитающее";
// Результат: double (например, 0.88 - высокая степень вхождения смысла)
```

5. Полный Гибрид (Смешивание всего)

Вы можете строить сложные правила принятия решений, смешивая датчики, текст и флаги в одной строке.

codeC#

bool isUserLogged = true;

string currentContext = "Жалоба на доставку";

// Вычисляется сходство строк (double), затем объединяется с логическим флагом (bool)

var shouldRedirectToSupport = isUserLogged and (currentContext equiv "Проблема с заказом");

// Результат: double (например, 0.85).

// Если бы isUserLogged был 'false', результат стал бы 0.0

6. Семантический выбор (SmSelect - выбор по смыслу)

Эта функция позволяет выбрать что-то по смыслу из списка представленных значений. Она работает очень быстро на нативном уровне.

object SmSelect(object source, string targetText, bool returnValue = false, double trigger = 0.5)

source должен быть одним из следующих значений:

Node (нейрон) — выбор идет из свойств

Engle (синапс) — выбор идет из свойств

List<string> - из значений

Dictionary<string, string> - выбор идет по ключам,

но выдача зависит от returnValue (key\value)

Dictionary<string, object>

Dictionary<string, int>

Dictionary<string, float>

Dictionary<string, double>

7. Семантическое множество (SmSelectTopN)

Эта функция позволяет выбрать TopN отсортированных в порядке убывания смысловой близости элементов из списка представленных значений. Она работает очень быстро на нативном уровне

public List<object> SmSelectTopN(object source, string tatgetText,bool returnValue = false, double trigger = 0.5,int topN = 3)

source должен быть одним из следующих значений:

Node (нейрон) — выбор идет из свойств

Engle (синапс) — выбор идет из свойств

List<string> - из значений

Dictionary<string, string> - выбор идет по ключам,

но выдача зависит от returnValue (key\value)

Dictionary<string, object>

Dictionary<string, int>

Dictionary<string, float>

Dictionary<string, double>

8. Глобальные функции помощники

Если автоматического приведения типов вам недостаточно, в скриптах всегда доступны встроенные конвертеры:

- Boolean(value, threshold = 0.7) — Принудительно отсекает нечеткость, возвращая строгий true или false (по умолчанию всё, что выше 0.7, считается Истиной).
- Пример: if (Boolean("Кот" equiv "Собака", 0.5)) { ... }
- Ternary(value) — Превращает любое число или логику в True, False или Unknown.
- Fuzzy(value) — Извлекает "сырое" нечеткое значение от 0.0 до 1.0 из любого объекта.

9. Гибридная логика: операторная форма (инфиксная запись)

В вашем микро-Рефал DSL все логические операторы доступны в естественной инфиксной форме (как в обычных языках программирования). Это позволяет строить выражения вида:

dsl

(x And y) Or (x1 fXor y1)

Чёткие операторы работают с булевыми значениями (true/false), нечёткие – со степенями истинности (double от 0.0 до 1.0). Гибридные выражения автоматически приводят нечёткие числа к булевым по порогу (0.7 по умолчанию), что позволяет смешивать оба типа.

1. Чёткие операторы (инфикс, результат – bool)

Оператор	Синтаксис	Тип операндов	Пример (вычисление)
И	a And b	bool	true And false → false
ИЛИ	a Or b	bool	true Or false → true
Исключающее ИЛИ	a Xor b	bool	true Xor true → false
Импликация	a Implies b	bool	false Implies true → true
Эквивалентность	a Equiv b	bool	true Equiv false → false
И НЕ	a Nand b	bool	true Nand true → false
ИЛИ НЕ	a Nor b	bool	true Nor false → false
Отрицание	Not a	bool	Not true → false
Вентиль	Gate a b	bool (возвращает a)	Gate true false → true

Примеры в DSL:

```
dsl
<let p = true>
<let q = false>
<if (p And (Not q)) then "выполнено">
<let r = (p Implies q) Xor (q Equiv p)> ;; r = false
```

2. Нечёткие операторы (инфикс, результат – double)

Все нечёткие операторы имеют префикс f перед названием. Работают с double. Реализация основана на стандартных t нормах (min) и t конормах (max) с производными операторами.

Оператор	Синтаксис	Формула	Пример при a=0.8, b=0.3
fИ	a fand b	min(a,b)	0.8 fand 0.3 → 0.3
fИЛИ	a for b	max(a,b)	0.8 for 0.3 → 0.8
fИскл. ИЛИ	a fxor b	(max-min) или (a xor b)	0.8 fxor 0.3 → 0.5
fИмпликация (Лукасевича)	a fimplies b	min(1, 1-a+b)	0.8 fimplies 0.3 → 0.5
fЭквивалентность	a fequiv b	1 - |a-b| (либо min двух импликаций)	0.8 fequiv 0.3 → 0.5
fИ НЕ	a fnand b	1 - min(a,b)	0.8 fnand 0.3 → 0.7
fИЛИ НЕ	a fnor b	1 - max(a,b)	0.8 fnor 0.3 → 0.2
fОтрицание	fnot a	1 - a	fnot 0.8 → 0.2

Примеры:

```
dsl
<let certainty = 0.8>
<let evidence = 0.3>
<let result = (certainty fand evidence) for (fnot evidence)>
;; result = max(min(0.8,0.3), 0.7) = max(0.3, 0.7) = 0.7
```

3. Гибридные выражения (смешивание чёткой и нечёткой логики)

В DSL разрешено использовать нечёткие значения (double) там, где ожидается bool, и наоборот. Приведение происходит автоматически по правилам:

- double → bool : true если значение > 0.7, иначе false.
- bool → double : true → 1.0, false → 0.0.

Это позволяет писать выражения, где, например, результат нечёткой операции fand используется в чётком And, а чёткий true – в нечётком for.

;; Гибридное И: левая часть – чёткая, правая – нечёткая (0.85 → true)

```
(true And Similarity("авто", "машина")) → true And true → true
```

;; Гибридное fИЛИ: чёткое true (1.0) for нечёткое значение 0.2

```
(true for 0.2) → max(1.0, 0.2) → 1.0
```

;; Сложное смешанное выражение

```
fuzzyA = 0.9;
fuzzyB = 0.4;
hybrid = (fuzzyA fand fuzzyB) Or (Similarity("бежать", "мчаться") fxor 0.6);
```

;; Подвыражения:

;; fand → 0.4 → в чётком Or приводится к false (0.4 <= 0.7)

fxor → Similarity (пусть 0.8) fxor 0.6 = max(0.8,0.6)-min(0.8,0.6)=0.2 → false

Выполнение скрипта

1. Общие скрипты

- Вычисление (скрипты вычисляющие что-либо).

2 * 5 -(Cos(x +y));

- Создание и изменение значения переменной.

A =2 * 5 -(Cos(x +y));

- Описание нового метода с параметрами или без.

void MeName(a,b)

{ x= a + b;}

- Описание новой функции с параметрами или без.

result MeName(a,b)

{ return a + b;}

- Описание конвейера функций.

X = f1(a,b)

|- f2

|- f3

|- f4;

2. Скрипты работающие с ГНС

Все команды ГНС должны начинаться с префикса GNN. Например:

Имя_проекта = GNN.Name; получает имя нейросети

Число_узлов = GNN.NodesCount; получает число нейронов (концептов)

GNN.ExecuteCommand(“скрипт”); создает или изменяет ГНС

3. Поисковые скрипты ГНС

Скрипты, которые ищут нейроны, связи, данные в нейронах, или группы нейронов.

Например (аналог GNN.Name через функции языка):

Имя_проекта = GNN.PropertyBag(“System:System”,”_projectName”);

4. Гибридные скрипты

Используют произвольную комбинацию алгоритмических, функциональных и поисковых команд:

Имя_проекта = GNN.PropertyBag(“System:System”,”_projectName”);

Пароль_проекта = GNN.PropertyBag(“System:System”,”_projectPassword”);

if(Пароль_проекта !=”” && AdminPassword != Пароль_проекта)

GNN.SystemPropertyVisible = false;

Else

GNN.SystemPropertyVisible = true;

5. Гибридные поисковые скрипты

Внутри команды GNN.FindNodesLogical(…); могут рекурсивно происходить вызовы функций и методом всего языка и GNN.

Гибридная (не четкая и когнитивная логика)

Socrates без специального описания типов комбинирует различные типы логики в выражениях. Причем допустимо получать результат:

- в четкой уверенности: true \ false
- в троичной уверенности true \ false \ susp
- в степени уверенности 0 … 1 float
- в автоматической оценке глубокого смыслового сходства (т. е. когда язык видит, что мы сравниваем смыслы A equiv B он автоматически применяет глубокое сравнение через встроенный компаратор, обученный на 1.9 миллиардах смыслов).

Пример скрипта с гибридным логическим выводом:

```
клиент = "врет, угрожает насилием";
сумма = 1000;
вид = "пиджак помятый, ботинки не чищены, руки грязные";
if(
клиент equiv "плохой человек"
&& сумма >= 500
&& (вид equiv "человек выглядит респектабельно") < 0.65
) return "не давать кредит";
```

Описание структуры сети

* Обычно сложные структуры нейронного графа описывают в удобном и наглядном графическом ИДЕ Socratis.Studio. Однако здесь мы рассмотрим форматы команд, которыми описывать и изменять структуру GNN практически с нуля.

Примечание: Все скриптовые команды не необходимо использовать префиксом GNN

1. Форматы базовых элементов

1.1. Формат Узла (Node)

Синтаксис: ([Schema :] Name [{ key=val; flag }])

Узел определяется в круглых скобках. Схема и свойства опциональны. Если схема не указана, используется Global. Строковые значения свойств могут заключаться в кавычки (обязательно, если внутри есть пробелы). Наличие ключа без значения (flag) интерпретируется как flag="true".

Пример синтаксиса	Описание
(Andrey)	Узел с именем Andrey в схеме по умолчанию.

(User:Andrey)	Узел Andrey в схеме User.
(DSL:Topic:Science)	Многоуровневая (иерархическая) схема DSL:Topic.
(File:Log {size=10; readonly})	Узел со свойствами. size равен "10", readonly равен "true".

1.2. Формат Ребра (Edge)

Синтаксис: -[Tag1, Tag2 [{ key=val; flag }]]->

Ребро определяется в квадратных скобках между дефисом и стрелкой. Ребро может содержать один или несколько тегов (разделенных запятой) и опциональный словарь свойств.

Пример синтаксиса	Описание
-[likes]->	Простая связь с одним тегом likes.
-[uses, depends]->	Мульти-теговая связь (ребро принадлежит сразу двум категориям).
-[route {dist=50; active}]->	Связь с тегом route и параметрами маршрута.

1.3. Ребра графа как синапсы ГНС

В Socrates ребра графа являются «трубопроводами» - синапсами проведения сообщений, имеющими некоторые обязательные свойства:

_weight - вес (проводимость) связи

_importance — важность связи

_protection — защита связи от удаления механизмом самообучения (минимальный вес != 0)

На кодовом уровне (C#; C#-script) синапсы имеют следующие важные свойства, дублирующие описанные выше:

float Weight

float Protection

int Importance

object PropertyBag(string propName, string retType = "string", object defaultValue = null, bool ignoreCase = false) - получение значения свойства с преобразованием типа

PropertySet(string propName, object value) — установка значения свойства

RemoveProperty(string key) — удаление свойства

1.4. Процессуальные типы синапсов (ребер графа)

С точки зрения активности нейронов в Зоне Внимания существует четыре типа синапсов:

Normal – обычные синапсы для передачи сообщений

Struct - структурные связи не передающие сообщения, описывают формальные отношения

Reflex – рефлексные синапсы (сильные). Моментально вызывают отклик нейрона.

Brake – анти-рефлексные синапсы. Моментально гасят нейрон в активной зоне.

2. API Поиска Узлов (FindNodes и FindNodesLogical)

Методы поиска принимают внутреннее содержимое скобок узла (без самих круглых скобок). Поддерживаются маски (Wildcards) * (любая строка) и ? (любой символ). Метод FindNodes использует сверхбыстрые индексы БД.

Запрос в API	Результат (что найдет)
FindNodes("*")	Возвращает абсолютно все узлы графа.
FindNodes("User:*")	Все узлы, принадлежащие схеме User.
FindNodes("*:Admin")	Узлы с именем Admin в любой схеме графа.
FindNodes("DSL:*:*")	Все узлы, схема которых начинается на DSL:.
FindNodes("* {active; error=404}")	Пересечение свойств: любые узлы, где есть флаг active и свойство error равно 404.
`FindNodes("User:A*	

2.1 Двухуровневый логический поиск

Для сложных математических и логических проверок используется метод FindNodesLogical с оператором WHERE.

- Логика работы: до WHERE выполняется быстрый индексный поиск. После WHERE вызывается скриптовый C#-интерпретатор для отобранных кандидатов.

2.2 Семантический просмотр свойств при поиске Like

вы можете комбинировать строгие фильтры и "размытый" нейросетевой поиск в одном запросе.

Примеры:

1. Простой семантический фильтр:

"Item:* WHERE Like('красный фрукт', 0.85)"

(Найдутся все узлы, чье описание похоже на "красный фрукт" с точностью 85%)

2. Комбинированный запрос (Цена + Смысл):

"Product:* {category=smartphones} WHERE Price < 50000 && Like('мощная камера и хороший экран', 75)"

(Здесь сначала сработает индекс по категории, потом отсеется цена, и только для оставшихся вызовется Like)

3. Использование переменных движка:

"Event:* WHERE Like(@SearchTopic, @MinScore)"

2.3 Встроенные переменные узла доступные в FindNodesLogical

_Id - ид узла

_Text - имя (текст)

_description - смысловое описание, на основе которого выбирается узел

_Schema - вся схема узла

_SchemaLeft - левая часть схемы (к какому множеству относится)

_SchemaRight - правая часть схемы (тип нейрона)

_Degree - общее число связей узла

_DegreeIn - число входящих связей узла

_DegreeOut - число исходящих связей узла

2.4 Использование общих функций языка в поиске

В правой части логического выражения при тестировании свойств может стоять любая функция языка:

```
Sensor:* WHERE _trigger >= fynctionName(...)
```

3. API Навигации и Траверсирования (ExecuteQuery)

Метод используется для поиска узлов на основе их связей. Возвращает список целевых узлов (Target), удовлетворяющих паттерну. Поддерживает маски * во всех трех блоках (Источник, Ребро, Цель).

Синтаксис: (SourceFilter) -[EdgeFilter]-> (TargetFilter)

Запрос в API	Описание
ExecuteQuery("(User:Andrey) -[likes]-> (*)")	Что любит Андрей? Вернет все узлы, на которые у Андрея есть связь likes.
ExecuteQuery("(*) -[depends]-> (Server:DB)")	Кто зависит от БД? Вернет все узлы, ссылающиеся на Server:DB связью depends.
ExecuteQuery("(User:*) -[friend {close}]-> (*:Admin)")	Кто близкий друг админов? Вернет узлы, проходящие через ребро со свойством close.
`ExecuteQuery("(Dog:*) -[likes	

4. API Создания и Модификации Графа

Эти методы интерпретируют DSL не как фильтр, а как инструкцию к созданию (Create/Update).

4.1. Создание Цепочек (ExecuteCommand)

Если узел не существует, он будет создан. Если существует — его свойства будут слиты с новыми, а недостающие связи добавлены.

Пакетная команда	Описание действий парсера
(User:Ivan)	Создаст узел Ivan в схеме User.
(User:Ivan) -[friend]-> (User:Petr)	Создаст обоих пользователей (если их нет) и проложит ребро от Ивана к Петру.
(A) -[x]-> (B) -[y]-> (C)	Цепное создание: создаст три узла и последовательно свяжет их (A ссылается на B, B ссылается на C).
(Dog:Sharik {hungry}) -[bites {force=10}]-> (Cat:Barsik)	Создаст узлы со свойствами и ребро с параметрами силы укуса.

4.2. Инстанцирование / Наследование (ExecuteCommand)

Ключевое слово NEW: и оператор FROM позволяют создать клон узла-шаблона со всеми его свойствами и исходящими связями.

Команда	Описание
NEW: (Dog:Sharik) FROM (Template:Dog)	Создает узел Dog:Sharik. Копирует в него все переменные из Template:Dog и дублирует все исходящие связи шаблона.
NEW: (Dog:Sharik {color="Black"}) FROM (Template:Dog)	Создает инстанс, но переопределяет или добавляет собственное свойство color.

4.3. Радикальное обновление состояния (ReplaceNodeFullState)

Используется, когда агенту нужно полностью "сменить контекст" (забыть старое, выучить новое).

В отличие от ExecuteCommand (который дополняет свойства), этот метод:

1. Очищает все текущие свойства корневого узла.

2. Удаляет все его исходящие связи.

3. Применяет новые свойства и связи из переданной команды.

Команда	Описание
ReplaceNodeFullState("(State:AI {status=idle})")	Узел State:AI теряет старые переменные, получает status=idle и лишается всех исходящих связей. Входящие связи сохраняются.
ReplaceNodeFullState("(Task:1 {done}) -[next]-> (Task:2)")	Задача 1 очищается от старых связей и свойств, помечается как done и прокладывает единственную новую связь к Задаче 2.

4.4. Поиск синапсов нейрона (ребер узла графа)

- GetOutgoingEdges(int nodeId) — исходящие связи (от кого-то к другим)
- GetIncomingEdges(int nodeId) — входящие связи (кто ссылается на этот узел)
- GetAllEdges(int nodeId) — вообще все связи узла

Синтаксис в скрипте (например):

```
GNN.GetOutgoingEdges( nodeId);
C#:
MainEngine.Instance.GetOutgoingEdges( nodeId);
```

5. Группы

Цель — уменьшить количество ребер, сохранить читаемость топологии и обеспечить удобную работу и обычных запросов, и механизмов распространения активации в графе.

В том числе автоматическое наследование свойств Паттернов и режимы коллективного взаимодействия.

Рассматриваются три практических способа: многократные однотипные строки в скрипте, паттерн «Хаб» (промежуточный узел группа) и генерация связей через программный цикл API.

Веер связей и паттерн «Хаб»

Прямой веер задается многострочным объявлением: одна и та же левая часть (A) повторяется, а правая (B1), (B2) и т.д. меняется, каждая связь описывается отдельной строкой с тем же типом ребра (например, manages). Для больших групп используется паттерн «Хаб»: создается узел группа (например, (Group:ProductionServers)), к нему подсоединяется управляющий узел через ребро manages, а все элементы группы (серверы) объявляют принадлежность через ребра Is-A. Тогда веер реализуется через два прыжка в запросах или в модели активации: A -[manages]-> Group и далее Server -*[Is-A]-> Group, что резко снижает число прямых связей «A→каждый Bi» и упрощает наследование общих свойств от узла группы.

Существует 3 базовых стандарта системных связей:

5.1 Наследование (Type-Of / Inheritance)

Тег: _IsA

Направление: От Экземпляра/Наследника -> К Шаблону/Родителю.

Смысл: Глубокая онтологическая связь. Указывает, что дочерний узел является конкретной реализацией родительского. Движок ГНС использует эту связь для "Сборки ДНК" — автоматического подтягивания свойств Variables от родителя к ребенку при активации.

Пример:

```
(Dog:Sharik) -[_IsA]-> (Template:Dog)
```

(Шарик наследует инстинкты и промпт концепта Собаки).

5.2 Групповое членство (Membership / Fan-out)

Тег: _InGroup

Направление: От Элемента -> К Узлу-Группе.

Смысл: Организационная связь. Элементы могут входить в сотни разных групп. Свойства группы не наследуются напрямую как инстинкты, но применяются как внешние правила (политики). Это позволяет делать массовые рассылки (Широковещание).

Пример:

```
(Server:Alpha) -[_InGroup]-> (Cluster:Production)
(User:Andrey) -[_InGroup {role="admin"}]-> (Department:IT)
```

(Обратите внимание: мы используем свойства ребра {role="admin"}, чтобы уточнить, кем именно элемент является в этой группе).

5.3 Композиция / Часть целого (Part-Of)

Тег: _PartOf

Направление: От Части -> К Целому.

Смысл: Жесткая физическая или логическая вложенность. Если удаляется Целое, движок должен (по логике сборки мусора) удалить и Части.

Пример:

```
(Component:Engine) -[_PartOf]-> (Car:BMW)
(Concept:Wheel) -[_PartOf]-> (Concept:Car)
```

5.4 Команды работ с группами

Получить все узлы, входящие в указанную группу.

(Ищет все входящие ребра _InGroup к узлу группы)

IEnumerable<Node> GetGroupMembers(this SimpleGraphDb db, int groupId)

Узнать, в каких группах состоит указанный узел.

(Ищет все исходящие ребра _InGroup от элемента)

IEnumerable<Node> GetNodeGroups(this SimpleGraphDb db, int nodeId)

Быстрое добавление узла в группу с возможностью указать роль (например: role="admin").

AddToGroup(this SimpleGraphDb db, int memberId, int groupId, Dictionary<string, string> edgeProps = null)

Быстрое удаление узла из группы.

RemoveFromGroup(this SimpleGraphDb db, int memberId, int groupId)

Собирает ВСЕХ предков узла вверх по графу (Genetic Tree), учитывая множественное наследование.

Используется перед вызовом LLM для сборки полного промпта свойств.

name="childNodeId">ID узла-наследника

"limit">Ограничитель: число (int - максимальная глубина) ИЛИ строка (имя или Схема:Имя целевого предка)

Список уникальных предков, отсортированный по степени близости (от родителей к прадедам)

List<Node> GetGeneticChain(this SimpleGraphDb db, int childNodeId, object limit = null)

Часть 3. Зона внимания

Внимание определяет сколько разных концептов одновременно может находиться в поле зрения ИИ системы. Обычно это число 10...300. Но если у вас мощное оборудование и требуется глубокий обзор, допустимо и больше.

1. Настройка внимания в проекте

При создании ГНС-проекта за внимание отвечают параметры концепта System:System:

```
// активная зона
    { "_AttentionMin", "20" },
    { "_AttentionMax", "100" }
```

2. Управление вниманием

Любой нейрон через скрипт, или просто скрипт может получить доступ к вниманию:

```
x = Attention;
Attention= X;
```

2.1. Ослабление связей для «недоживших» нейронов (LTD)

Вопрос: стоит ли ослаблять связи нейронов, которые остыли или были вытеснены, пропорционально их вкладу?

Да, это критически важно. В биологии это называется «синаптической депрессией».

Если синапсы подали сигналы, но нейрон так и не смог возбудиться до порога (остыл или был вытеснен), это означает, что данные сигналы были шумом или ложной тревогой.

Если не ослаблять такие связи, сеть быстро забьется «мусорными» путями, которые будут постоянно подогревать нейроны до около порогового состояния, мешая работе Зоны Внимания.

Как это правильно рассчитать:

Если общая сумма пришедшего подогрева была

E

E

(переменная Excitation), а конкретный импульс принес вес

W_i

Wi,

то его относительный вклад равен:

$P_i = \frac{W_i}{E}$

Pi=WiE

Мы должны уменьшить вес соответствующего ребра (Edge.Weight) пропорционально этому вкладу:

$Weight_{new} = Weight_{old} \times (1 - \beta \times P_i)$

Weightnew=Weightold×(1−β×Pi)

где

β

β

— коэффициент затухания (например, 0.05 или 5%). Если вес падает ниже предела (например, 0.05), связь удаляется из графа.

2.2. Автоматическое усиление при выстреле (LTP)

Вопрос: стоит ли усиливать связи сработавших нейронов? Не приведет ли это к хаосу?

Приведет к хаосу и лавинообразному взрыву активности, если не ввести жесткие предохранители.

В нейросетях без ограничений этот эффект называется Runaway Excitation (когда нейроны зацикливаются и начинают бесконечно возбуждать друг друга, аналог эпилепсии).

Однако автоподстройка необходима для самообучения. Чтобы избежать хаоса, нужно применить три правила:

1. Асимметрия обучения: Ослабление связей при неудаче должно быть чуть сильнее или происходить легче, чем их усиление при успехе (принцип «доверие трудно завоевать, но легко потерять»).

2. Мягкое насыщение (Soft Saturation): чем ближе вес к 1.0, тем труднее его увеличить. Формула:

$$Weight_{new} = Weight_{old} + \alpha \times Pi \times (1 - Weight_{old})$$

$$Weight_{new} = Weight_{old} + \alpha \times Pi \times (1 - Weight_{old})$$

3. Использование поля Protection: если у ребра высокий показатель Protection, оно должно медленнее остывать и медленнее разрушаться.

3. Принудительная активация нейрона

Позволяет программно (из скрипта) активировать любой нейрон по имени или id. Если нейрон был в зоне внимания, он будет удален.

Fire(neuron,input [, weight =10]);

4. Область временных знаний

ГНС может создавать нейроны и связи во Зоне Временных Знаний, которая на системном уровне просто кодируется префиксом схемы Temporary.*:*

Структуры, созданные в Temporary, ничем не отличаются от других структур.

Однако про каждом новом старте ГНС (Run) Temporary объекты и все их связи будут удалены полностью.

Для того чтобы не писать каждый раз шаблон схемы, можно использовать метод добавления:

```
TemporaryAddNode(string text, string schema = null, Dictionary<string, string> props = null, string temporaryName="Temporary")
```

Вы также можете удалить их программно в любой момент:

```
TemporaryClear();
```

Вы также можете добавлять свои собственные временные зоны, и удалять их самостоятельной в любой момент (но автоматически они не удаляются при старте):

```
TemporaryNew("гипотеза_о_Черной_материи");
TemporaryClear("гипотеза_о_Черной_материи");
```

Когда ГНС убедилась в полезности временных знаний, можно их сделать постоянными (одобрить):

```
int TemporaryCommit(string temporaryName = "Temporary")
```

Метод выдает число одобренных нейронов. Например:

```
одобрено = TemporaryCommit();
```

5. Свойства нейронов

Свойства нейронов позволяют гибко настраивать представляемые ими концепты и агенты

5.1. _synapseReception

Определяет, как будут обработаны сигналы входных связей при активации нейрона

rules - вычисление refal-правил, находящихся в свойствах нейрона

hot — самый горячий синапс

cold — самый холодный синапс

all - все сообщения скопом

first — первый активный синапс по времени

last - последний по времени

rnd - случайный синапс

shadow — только информация о типе самого горячего синапса

script — скрипт получит все сигналы — синапсы и должен выбрать

funcName(NeuralState state);

5.2. Свойства системного нейрона (System:System)

Этот нейрон никогда не попадает в Зону Внимания. Он хранить глобальные параметры всей ГНС, а связи с ним позволяют Вашему (будущему) умному процессу рефлексивно изучить и даже изменить базовую структуру ГНС.

_LoadScript - этот скрипт будет выполнен один раз при загрузке

Не добавляйте в него создания элементов, если состояние ГНС будет

сохранено потом

_RunScript - этот скрипт будет выполнен один раз при каждом запуске ГНС (Run)

// активная зона

_AttentionMin - минимальное (и начальное) число нейронов в Зоне Внимания

_AttentionMax - максимальное число нейронов в Зоне Внимания

// триггера уверенности

_triggerTerrible "0f" - ужасно

_triggerBad" "0.3f" - плохо

_triggerUncertain" "0.4f" - неуверенность

_triggerGood" "0.65f" - хорошо

_triggerExcellent" "0.8f" - отлично

Примечание: Соответствующие свойства в скрипте доступны:

float triggerTerrible

float triggerBad

float triggerUncertain

float triggerGood

float triggerExcellent

float ConfidenceAny - любое значение уверенности

// настройки

_echo_off - используется только UI. Отключает эхо ввода пользователя

_projectName - полное название ГНС на диске (без пути, но с расширением)

_inputTextChannel - имя сенсорного нейрона, принимающего текстовые сообщения

_inputImageChannel - имя сенсорного нейрона, принимающего графические сообщения

_fileKnowledgeChannel - имя сенсорного нейрона, принимающего текстовые файлы

_inputSpeechChannel - имя сенсорного нейрона, принимающего речевые сообщения

_outputChannel - имя Effector нейрона, выдающего текстовые ответы

_commandChannel - имя Effector нейрона, выдающего команды оборудованию

_errorOutChannel - имя Effector нейрона выдающего сообщения ошибок

_systemOutChannel - имя Effector нейрона, выдающего системные сообщения

_embeddingllmPath - краткое название, или Web маршрут для эмбеддинг модели

_rerankerllmPath - краткое название, или Web маршрут для реранкерной модели

_llmPath - краткое название, или Web маршрут для логической модели

_SpeecllmPath - краткое название, или Web маршрут для речевой модели

_UsingDLLFolder - все DLL из этой папки добавят функции языку Socrates-Refal

_NeuralNetworkDescription — краткое описание что делает и чем является ГНС

_UIBlocks - расширяемые блоки UI дизайнера

_UILogicBlocks - расширяемые блоки UI логического языка дизайнера

Примечание: Пользователь может сам писать новые виджеты и логически блоки для UI дизайнера на языке JS. Подробная документация есть в интернете или в справке самих блоков.

Примеры расширения:

// 3. ФАЙЛЫ И МЕДИА

bm.add('ctrl-file-picker', {

label: 'Выбор Файла', category: '🗀 Файлы и Медиа',

media: '<svg viewBox="0 0 24 24"><path d="M20 6h-8l-2-2H4c-1.1 0-1.99.9-1.99 2L2 18c0 1.1.9 2 2 2h16c1.1 0 2-.9 2-2V8c0-1.1-.9-2-2-2zm0 12H4V8h16v10z"/></svg>',

content: ' 🗀 Выберите файл для отправки в ИИ <input type="file" id="file_input" style="font-size: 12px; color: #64748b;" /> '

});

bm.add('media-image', {

label: 'Картинка / График', category: '🗀 Файлы и Медиа',

media: '<svg viewBox="0 0 24 24"><path d="M21 19V5c0-1.1-.9-2-2-2H5c-1.1 0-2 .9-2 2v14c0 1.1.9 2 2 2h14c1.1 0 2-.9 2-2zM8.5 13.5l2.5 3.01L14.5 12l4.5 6H5l3.5-4.5z"/></svg>',

content: ''

});

Виджеты:

// 4. ИИ И ГОЛОС

bm.add('ctrl-voice-btn', {

label: 'Голос 🎙️', category: '🧠 Голос и ИИ',

media: '<svg viewBox="0 0 24 24"><path d="M12 14c1.66 0 3-1.34 3-3V5c0-1.66-1.34-3-3-3S9 3.34 9 5v6c0 1.66 1.34 3 3 3zm5.91-3c-.49 0-.9.36-.98.85C16.52 14.2 14.47 16 12 16s-4.52-1.8-4.93-4.15c-.08-.49-.49-.85-.98-.85-.61 0-1.09.54-1 1.14.49 3 2.89 5.35 5.91 5.78V20c0 .55.45 1 1 1s1-.45 1-1v-2.08c3.02-.43 5.42-2.78 5.91-5.78.1-.6-.39-1.14-1-1.14z"/></svg>',

content: '<button id="btn_voice" style="display:inline-flex; align-items:center; gap:8px; padding:10px 20px; background:#0ea5e9; color:white; border:none; border-radius:25px; font-weight:bold; cursor:pointer; font-size:14px;">🎙️ Голосовой ввод</button>'

});

Логика:

window.gjsEditor = grapesjs.init({

container: '#' + containerId,

fromElement: false,

height: '100%',

width: 'auto',

storageManager: false,

styleManager: {

sectors: [

{ name: '🎨 Цвета, Фон и Границы', open: true, buildProps: ['background-color', 'color', 'border-radius', 'border', 'box-shadow'] },

```
{ name: '📏 Размеры и Отступы', open: false, buildProps: ['width', 'height', 'margin', 'padding', 'min-width', 'min-height'] },
{ name: '✍️ Типографика', open: false, buildProps: ['font-family', 'font-size', 'font-weight', 'text-align', 'line-height'] },
{ name: '📐 Макет (Flexbox / Позиция)', open: false, buildProps: ['display', 'flex-direction', 'justify-content', 'align-items', 'gap'] }
]
}
});
window.gjsEditor.Commands.add('jump-to-logic', {
run: function (editor) {
var selected = editor.getSelected();
if (!selected) return;
var id = selected.getId();
if (!id) { id = 'el_' + Math.random().toString(36).substr(2, 6); selected.setId(id); }
window.focusBlocklyOnElement(id);
}
});
```

5.3. Свойства нейрона памяти (Memory)

Нейроны памяти имеют уникальные не повторяющиеся ни где свойства:

_memoryType

concept - является уникальным концептом памяти

conceptGroup - объединяет все входные группы концептов памяти

Представителей группы входящими в эту группу связями

типа _InGroup (выберите галочку Struct в редакторе (w=0;i=100;))

allConcepts - объединяет все существующие концепты памяти

_source - описание источника

_persona - необязательное описание персоны,- если это персональный концепт

_guidance - необязательная инструкция

_content - всегда хранит последнюю копию своей памяти в читаемом формате

_description - краткое описание содержимого

События:

SuccessRememberEvent – хороший результат Memory

FailureRememberEvent – ничего не найдено Memory

5.4. Связи нейронов Memory

Для этого нейрона тип связей критичен и задает смысл операции над поступившими данными (по сути — тип команды к нему):

_recall - простой вопрос к концепту

_recalContextual - запрос контекстуального типа

_recalDetailed - запрос подробностей

_recalPersistent - настойчивый запрос

_recalCareful - запрос с аккуратным требование

_remember - запомнить новое состояние концепта

_guidance - запомнить руководство

_persona - запомнить персональную информацию концепта

_getPersonality - запрос о личных данных

_getGuidance - запрос о навыках

5.5. Свойства нейрона вводы (Sensor)

Нейроны ввода могут в некоторых режимах передавать информацию без явной связи — семантически выбранному нейрону (концепту). Но только таким, у которых задано свойство _description – должно в краткой, четкой форме описывать суть концепта.

_channel - подключает сенсор к одному из Типов каналов ввода:

- Message
- File
- MessageAndFile
- FileKnowledge
- MessageAndFileKnowledge
- Image
- Any

_messageType - настраивает тип сообщения:

ordinaryMessage - обычное текстовое сообщение

tempKnowing - временный файл для текущей работы

permanentKnowing - файл, который нужно запомнить

dataMessage - как правило используется датчиками и устройствами

command - команда управления для ГНС

_confidence - указывает минимальную уверенность, ниже которой синапс ничего не делает,

но посылает Событие

InputConfidenceFail.

Иначе выполняет функцию и посылается событие

InputConfidenceSuccess

Примечание: можно указывать числом 0f..1f, константами triggerTerrible

и т.п. А также ConfidenceAny

_processing - определяет способ обработки сообщения:

relay - ретрансляция по выходам

similarity — семантический выбор концепта приемника (по смыслу)

keywords - семантический выбор нескольких концептов приемников (по смыслу)

target -

rules -

_keyWords - задает максимально число ключевых слов (по умолчанию 5)

_condition - можно задать скрипт или условие, которое проверяет пригодность данного сенсора к входному сообщению. Для скрипта или функции доступен параметр value содержащий само сообщение

5.6. Групповой режим сенсоров

В нейроне System:System можно настраивать каналы ввода на групповой режим работы, что позволяет более тонко дифференцировать входную информацию.

Например:

```
{ "_inputTextChannel", "Global.Sensor:inputText*" }
```

Что обеспечит автоматический выбор из группы:

```
Global.Sensor:inputText1
Global.Sensor:inputText2
Global.Sensor:inputTextКнижка
```

И так далее. И уже в настройке каждого сенсора укажите на что он срабатывает

5.7. Свойства нейрона выводы (Effector)

Нейроны вывода не передают ничего по выходным связям. Для них выходные связи бессмысленны. Это конечные точки.

_channel - подключает эффектор к одному из каналов вывода:

_synapseReception — определяет способ выбора входного источника (см. выше)

5.8. Свойства системного нейрона (SystemCommand)

Командный нейрон не распространяет сигнал. Это конечная точка, предназначенная для выполнения скриптовых команд.

_method - должен содержать скрипт или ранее описанную функцию в формате

```
methodName(input,tag,state);
```

В момент активации доступны локальные данные:

input - входное сообщение

tag - имя синапса

state - весь объект активации нейрона:

```
public class NeuralState
{
public int NodeId;
public float Excitation;
public List<Impulse> Signals = new();
public DateTime LastActivity; }
public class Impulse
{
```

```
public int SourceId;
public string Text;
public float Weight;
public int Importance; // важность
public string Tag; } // Тег синапса, по которому пришел сигнал
```

5.9. Свойства универсального нейрона (Neuron)

Этот нейрон предназначен для описания типовых свойств и топологии ГНС.

_synapseReception - определяет способ обработки сигналов синапсов (см. выше)

_trigger" - порог срабатывания

_importance - важность

_processing - способ реагирования:

rule - Refal – перебор правил, не имеющих префикса _

similarity - на основе близкого смысла (ключей)

выдает записи из не имеющих префикса _

similarityscript - на основе близкого смысла (ключей)

запускает скрипт из не имеющих префикса _

script - запускает скрипт из _method

_method - описание скрипта (см. выше)

5.10. Свойства нейрона обучения (Conductor)

Этот нейрон предназначен для изменения свойств и топологии ГНС.

Он не помещается в активную зону, а срабатывает по своему событию — на которое настроен. Однако, он также может распространять свой собственный сигнал всем нейронам приемникам своего выхода (если такие есть). В этом случаи будет передаваться описание события «EventName; Target»

_trigger - success, failure, uncertainty

_importance

_synapseReception

_target - Тип нейрона

_synapse - Тип синапса (any или имя)

_event - Имя события

_processing

auto - работает внутренний механизм

script - работает скрипт:

_method - описание скрипта

methodName(input,tag,target,state);

(см. выше)

5.11. Свойства нейрона сборки (Assembler)

Этот нейрон собирает текущий шаг решения на основе заданной цели, памяти последних шагов, и полученных сообщений. Важной особенностью является работа с контекстом предыдущих шагов. В системе может быть несколько Ассемблеров — каждый для своей смысловой темы (типа заданий). Ассемблер может распространять сигнал обычным способом — через синапсы.

Цель задается одним или несколькими синапсами типа goal . Если одновременно пришло несколько целей, они будут конкурировать по весу и значимости.

_trigger" - порог срабатывания

_minWeight - порог для сообщений

_maxMessages - сколько сообщений учитывать

_guidance - инструкция

_waitComplete - ждать указанного числа сообщений

_temperature - собственная температура генератора смысла

_useContext - использовать контекст

_changeContext - изменять контекст

_history - число шагов истории для сохранения логики связанности

5.12. Нейрон обучения синапсов Modulator

_synapseTag - на какой тип синапсов воздействуем (tag)

_methodInfluence" - способ влияния

InferenceContext – на все связи указанного типа

OutNeurons - на выходные связи подключенных к его выходу нейронов

если тип связи * — то на все

EntryNeurons - на выходные связи подключенных к его выходу нейронов

если тип связи * — то на все

_expectedType - как интерпретировать входной сигнал

number - как число на входе

similar - как максимальное лингвистическое сходство с одним из значений с словаре

свойств нейрона. Например (1,черный кот; 2,белый кот). Имена свойств не имеют значения. Сравниваются только значения свойств.

Meaning - как максимальное смысловой сходство с одним из значений с словаре

свойств нейрона. Например (1,черный кот; 2,белый кот). Имена свойств не имеют значения. Сравниваются только значения свойств.

_factor - величина при успехе

_decay - величина при неудаче

_cooling - скорость восстановления

_event - имя события

_automaticDestruction - уничтожать обнулившийся синапс

Примечание: свойство _Protection != 0.00f защищает синапс от изменения нижеописанного уровня. Т.е. может быть и отрицательным значение.

_failureTrigger - триггер неудачи (обычно 0.45)

_successTrigger - триггер удачи (обычно 0.75)

_normal - нормальное состояние, к которому стремится затухание

5.13. Свойства нейрона Persona

Нейристор Persona – узкоспециализированный концепт, предназначенный для формирования ответа на основе LLM с поддержкой двух видов памяти: кратковременной и длительной (память фактов и заданий)

Также он поддерживает команду #clear которая, будучи переданная с параметрами (вес: >= _trigger, важность: 1) приводит к мгновенной очистки всех видов его памяти.

Свойства:

_longModel - false \ true – выбирает более мощную модель, если она загружена в главном нейристоре.

_trigger - величина терминального возбуждения для срабатывания

_persona - имя персоны, например «Собака Шарик, Петя, Катя»

_personalMemory – уникальное имя персональной длительной памяти. Если _persona – соответствует имени персоны. Если пусто или none – длительная память отсутствует. Длительная память сама извлекает факты.

_training - режим тренировки. Пока только none

_distribution - Режим раздачи результата: relay – всем выходам; semantic-relay – получателям описание которых (_description) соответствует смыслу ответа.

_guidance - инструкция

_maxTokens – максимальная длина ответа (обычно 512 токенов)

_contextLen – длина кратковременной памяти

timeOut - защита от зависания в минутах. Обычно 1 минута.

5.14. Системный контекст OutContext

Виртуальный (не исполняемый) нейрон, обеспечивающий настройки и связи с системным выходным контекстом (ассоциативный список результатов работы ассемблеров).

Он является основой для работы нейронов - Ассемблеров, нейронов самообучения и нейронов прогнозирования. Внутри него реализуются механизмы «отлавливания» различных видов совпадений и заранее заданной смысловой близости (за счет предварительно динамически настраиваемых алгоритмов структурного, и смыслового анализа).

Активность распространяется по выходным связям (если есть) с учетом их типа, а также широковещательными событиями, если они заданы.

_contextLen - длина контекста (по умолчанию 1024)

_eventPushResult — имя события (если требуется) возникающего при добавлении результата

_eventPushMining — имя события (если требуется) возникающего при повторении результата

_expectationDeadline — длина периода прогнозирования (в числе результатов)

_confidenceThreshold - порог мощности для создания ожидания

_matchesNumber - количество совпадений для срабатывания

_potentiationFactor - коэффициент автоматического обучения связей (по умолч. 0.04f)

_depressionFactor - коэффициент автоматического забывания связей (по умолч. 0.08f)

Выходные связи.

Информация передается присоединенным выходным нейронам с учетом имени связи:

push - добавление результата

mining - совпадение

5.15. Смысловой вектор свойств

При записи значения свойств можно установить для него смысловой вектор.

GNN.PropertySet(key, value, true);

5.16. Поиск свойств, отвечающих смыслу с учетом наследования

Для нейрона можно найти свойства, отвечающие заданному смыслу (если для них при записи или изменении задан режим вектора):

```
C#:
Task<List<(string key, string value, float Score)>> SemanticSearchProperties(
object Node, // нейрон, или его полное имя, или индекс
string findDescription, // что ищем
int topK1 = 5, // предварительный поиск(сколько выбрать?)
int topKOut = 1, // глубокий смысловой выбор (сколько?)
object limit = null // учитывать предков для уровня или имени)
```

5.17. Поиск наследуемых свойств по смыслу в MD формате

```
C#-script:
GetNodeGeneSemantic(
object node, // нейрон, или его полное имя, или индекс
string findDescription, // что ищем
int topK1 = 5, // предварительный поиск (сколько выбрать?)
int topKOut = 1, // глубокий смысловой выбор (сколько?)
object limit = null // учитывать предков для уровня или имени
)
C#:
await SemanticSearchPropertiesMarkdownAsync(
node,
findDescription,
topK1=10,
topKOut=3,
limit=null
)
```

5.18. Создание собственного прототипа нейрона userNeuron

Теперь вы можете добавлять собственные прототипы в конструктор.

Для каждого прототипа добавьте соответствующий код в _LoadScript скрипт системного нейрона.

Используйте следующий формат описания прототипа.

Обратите внимание, что:

- необходимо четко указывать один из четырех типов свойства

- значение свойства в […] является перечислением вариантов в редакторе.

- в перечисляемом свойстве по умолчанию устанавливается первое значение

- в скрипт передаются обычные параметры активации нейрона и ваши свойства, уже извлеченные по имени.

- все параметры и вами добавленные переменные - локальные

```
userNeuron ИМЯ( int _a1= 12345;
                float _ss = 0.12f;
                string _slon = африканский слон;
                bool _bb =[true,false]; - свойства строго по списку
                string _enumProp=[qqq,www,eee,rrr] ) - свойства строго по списку
{
// Место для описания скрипта (действующей части при активации)
  // Доступны системные параметры вызова
  nState – объект истории пребывания в Активной Зоне
  input - победивший синапсический сигнал
  tag - теп синапса
  signalType – тип поступившего сигнала
  // доступны все свойства
  (_a1, )ss, _slon, _bb, _enumProp)
// Опишите здесь скрипт действия и верните результат для рассылки
  return nul; // или верните null если рассылка не требуется};
```

1. Объект активации nState

```
/// <summary>
/// Локальная память активации нейрона (в активной зоне) - от первого синаптического импульса до порога возбуждения
/// </summary>
public class NeuralState{
public int NodeId; // нейрон, о котором идет речь
public float Excitation; // текущий суммарный "подогрев"
public List<Impulse> Signals = new(); // история пришедших синапсических сигналов
public DateTime LastActivity; // время начала активации
public Edge FireEdge { get; set; } // выбранный синапс после активации Нейрона
public NeuralState(int id){
```

```
NodeId = id;
LastActivity = DateTime.Now;}}
```

2. Объект синапсического импульса

```
/// <summary>
/// Временно регистрирует сигнал синапса в Зоне Внимания
/// </summary>
public class Impulse
{
public int SourceId; // идентификатор источника - если edge отсутствует (сообщение системное или от события)
public object Value; // само сообщение - любой объект
public int signalType= 0; // резервное поле для интерфейсных сообщений. Например: 0 сообщение,1 данные для анализа,2 информация к запоминанию
public float Weight; // импульс подогрева edge.Weight * Weight сообщения
public int Importance; // важность
public Edge edge; // связь пришедшего сигнала
public string Tag; // Тег синапса, по которому пришел сигнал
}
```

5.19. Callback-связь – обратный вызов терминальной функции

Callback-связь — это специальная направленная связь между двумя нейристами, предназначенная для адресного получения текущего состояния нейриста-источника без распространения его активности по обычным связям.

Нейрист-источник callback-связи не передаёт своё состояние по этой связи автоматически. Когда нейрист-получатель достигает терминальной функции и ему требуется дополнительная информация, он может выполнить callback-запрос к источнику. Поведение источника определяется исключительно его текущим состоянием:

1. Источник частично разогрет, то есть имеет накопленные сообщения, но ещё не достиг порога активации. В этом случае он принудительно выполняет терминальную функцию на имеющемся накопленном контексте и возвращает полученный результат вызывающему нейристу. При этом его накопление не сбрасывается, а сам источник продолжает обычное ожидание достижения порога.

2. Источник не разогрет и не находится в активной зоне. В этом случае терминальная функция не запускается. Источник возвращает своё последнее сохранённое состояние.

Полученный результат помещается в накопленный список сообщений вызывающего нейриста и может участвовать в его собственной терминальной функции.

Таким образом, callback представляет собой адресный запрос текущего состояния другого нейриста, а не дополнительный канал распространения возбуждения. Он не вызывает каскадного распространения активности по графу и не изменяет нормальный процесс накопления и активации нейриста-источника.

5.20. Обратное распространение ответа _isService

Некоторые нейристоры, например корень дерева «Животные», должны по смыслу своего назначения включить режим обратного распространения ответа по дереву:

_isService = true;

Это позволяет очень эффективно формировать работу с подсистемами — все внешние связи только с корнем системы \ или модуля, а элементы графа этой системы плотно связаны друг с другом.

Что это дает на практике?

При накоплении опыта или ручном формировании архитектуры ИИ, новые элементы и\или связи добавляются только в подсистеме. Например, новые «животные» и их связи. Но сама система не меняет связей с другими системами, хотя и изменяет поведение\реакции.

Для этого нужно:

1. Установить _isService=true только у корня системы (например, «Животные»).

2. Все выходные связи Корня, которые идут за пределы его подсистемы пометить типом _Service

3. Те конечные элементы подсистемы, которые должны формировать обратный ответ, не должны иметь выходных связей (вообще).

Инструменты

Инструменты, — это функции , обычно для общения ИИ с внешними программами, устройствами и сетями. Но могут быть и просто операции над данными.

Разница между обычными функциями и скриптами в том, что ИИ сам выбирает, когда и какой из доступных ему инструментов нужно вызвать (по смысловой надобности) и сам готовит для инструмента аргументы, которые ему требуются. Затем результат (если инструмент возвращает результат) ИИ сам использует в процессе работы.

Для этого инструмент кроме кода должен иметь смысловое описание, и описание каждого своего входного параметра (аргумента).

Сократ имеет несколько десятков встроенных инструментов, которые просто нужно перечислить в свойствах нейристора, который допускает вызов инструментов.

Для каждого нейристора или нейро-агента используйте минимально возможное количество необходимых ему инструментов. Желательно меньше 10. Иначе нейристор может начать путаться при вызовах.

5.21. Встроенные инструменты

Для добавления встроенного инструмента достаточно указать его имя в соответствующем свойстве нейристора, который поддерживает инструменты. Перечисляете инструменты через запятые и без пробелов. Пример: pdfsearch,pdfextract,pdfmerge

Встроенные инструменты:

pdfsearch — поиск текста и данных в PDF-документах

pdfextract — извлечение текста, изображений или содержимого из PDF

pdfmerge — объединение нескольких PDF-документов

pdfmetadata — получение или изменение метаданных PDF-документа

pdfpages — получение информации о страницах PDF-документа

pdftoimage — преобразование страниц PDF в изображения

pdfsplit — разделение PDF-документа на отдельные части

pdfunlock — снятие ограничений или пароля с PDF-документа

networkconnectivity — проверка доступности сетевого подключения

networkdns — выполнение DNS-запросов и разрешение доменных имён

networkinterfaces — получение информации о сетевых интерфейсах

networkping — проверка доступности узла по сети

networkportcheck — проверка доступности определённого сетевого порта

networkportscan — сканирование диапазона сетевых портов

5.22. Описание собственного инструмента

Вы можете добавить собственный инструмент (ы) описав их функцию в скрипте, или вызвав из скрипта метод внешнего плагина.

```
UserTool(
имя | - слитное краткое имя-описание функции, желательно на англ.
InputSchema | - JSON схема и описание параметров
Description - описание функции инструмента для ИИ
){
Ваш скрипт - все что должен сделать инструмент
return result; - что он должен вернуть для ИИ
}
Как описывать JSON - схему параметров, пример:
{
"type": "object",
"description": "Save an important finding or note during research.",
"properties": {
"category": {
"type": "string",
"description": "Category for the note (e.g., benefit, challenge, statistic, quote)."
},
"content": {
"type": "string",
"description": "The content of the note to save."
},
"source": {
"type": "string",
"description": "Source or reference for this information."
}},
"required": [
"content"]}
```

5.23. Диалог с расширением \ инструментом

Вы можете создавать удобные расширения редактора и других функций, не только используя выбор элементов графа мышкой, но и с элементами диалога (кнопки, текстовый ввод, выбор файла).

Для этого изучите раздел описания про диалоговые операторы DSL.

6. Навыки агента (Skills)

Agent Skills — это открытый стандарт для определения модульных, многократно используемых возможностей ИИ с помощью структурированных файлов инструкций. Навыки загружаются из папки, указанной в свойстве нейристора (агента) _skills – где должен быть указан путь к коренной папке, содержащей папки навыков.

Если первое слово _project – то путь начинается от текущей коренной папки проекта

Навыками могут пользоваться только нейристоры Agent и Persona.

Навыки позволяют агентам динамически приобретать специализированное поведение, экспертные знания в предметной области или возможности рабочих процессов без изменения кода. Навыки следуют шаблону поэтапного раскрытия с использованием файлов SKILL.md , которые объединяют метаданные YAML с инструкциями Markdown.

6.1. Что такое навыки агента?

Определение: Навыки агентов представляют собой стандартизированный подход к формированию возможностей агентов ИИ в виде переносимых, самодостаточных наборов инструкций. Каждый навык определяет, что может делать агент, как он должен себя вести, и какие инструменты или контекст ему необходимы, — всё это выражено в формате, понятном как людям, так и системам ИИ.

Стандарт Agent Skills возник из необходимости совместного использования и повторного применения моделей поведения агентов в различных системах, моделях и платформах. Вместо того чтобы жестко прописывать возможности агентов в логике приложения, навыки выносят эти определения в структурированные файлы, которые можно контролировать по версиям, совместно использовать и комбинировать.

6.2. Формат файла SKILL.md

Agent Skills используют простой, но мощный формат файлов, который объединяет в себе:

1. Метаданные YAML (Forwardmatter)

```
---
name: Code Review Assistant
description: Reviews code for bugs, security issues, and best practices
version: 1.0.0
author: LM-Kit
tags: [development, code-review, security]
tools:
- file_system
- code_analysis
```

```
context:
- programming_languages
- security_guidelines
---
```

2. Текст документа в формате Markdown (инструкции)

В теле документа содержатся инструкции на естественном языке, которые управляют поведением агента:

```
# Code Review Assistant
## Your Role
You are an expert code reviewer with deep knowledge of software engineering best practices, security vulnerabilities, and clean code principles.
## Guidelines
1. Focus on actionable feedback, not stylistic preferences
2. Prioritize security issues over minor optimizations
3. Explain the "why" behind each suggestion
## Review Process
- First, understand the code's purpose
- Identify potential bugs and edge cases
- Check for security vulnerabilities
- Suggest improvements with examples
```

6.3. Роль навыков агентов

1. Модульный подход к проектированию возможностей позволяет разделить «то, что знает агент» и «как агент работает». Это дает возможность заменять, обновлять или комбинировать навыки без изменения кода приложения.

2. Функция «Постепенное раскрытие информации» поддерживает поэтапное раскрытие информации, при котором базовые возможности всегда доступны, но расширенные функции могут быть разблокированы в зависимости от контекста или прав пользователя.

3. Пакет экспертных знаний в предметной области: специализированные знания (юридические, медицинские, финансовые, технические) в виде навыков, которые можно применять по мере необходимости, обеспечивая сосредоточенность и эффективность работы агентов.

4. Напишите один раз, используйте везде. Навык, написанный для одного приложения LM-Kit.NET, работает в любом другом, и навыки могут быть доступны для использования в разных командах, организациях или в более широком сообществе.

5. Навыки управления версиями и контроля — это текстовые файлы, которые органично вписываются в системы контроля версий, обеспечивая процессы проверки, отката и аудита изменений в поведении ИИ.

7. Краткая инструкция составления схемы

Схема параметров инструмента описывает, какие данные может передавать ИИ при его вызове.

- type — тип корневого объекта схемы. Для набора именованных параметров обычно используется object.
- description — краткое описание назначения инструмента или группы параметров.
- properties — список доступных параметров инструмента.
- Имя каждого параметра указывается внутри properties.
- type параметра определяет тип значения: string, integer, number, boolean, array или object.
- description параметра объясняет ИИ, когда и как его использовать.
- required содержит имена обязательных параметров.
- Параметры, не указанные в required, считаются необязательными.
- Для списков используется array, а тип элементов задаётся через items.
- Для вложенных структур используется object с собственным разделом properties.
- Примеры допустимых значений можно указывать в описании или через дополнительные ограничения, например enum, minimum и maximum.

8. Внешние коммуникации

8.1. COM порт \ USB-COM

Для настройки Эффекторов и Сенсоров на работу с виртуальным COM портом настройте номер порта и скорость передачи в свойстве _COM_Port нейристора System:System.

Например: 4:9600

Затем настройте имя групп для Сенсоров и Эффекторов в свойстве _COMChannel нейристора System:System. Например: Global.Sensor:COM*

Их свойства _channel

установите в состояние COM_PORT.

8.2. Агентные Tools для COM порт \ USB-COM

Агенты могут передавать текст в предварительно указанный в настройках проекта (см. выше) COM порт.

Для этого подключите к ним инструмент com_port_write.

Агенты не могут самостоятельно прослушивать порты. Воспользуйтесь Сенсорами.

Скриптовые операторы для прямой работы с COM портами

```
/// <summary>
/// Проверяет и подключает или отключает процесс
/// </summary>
bool CheckCOMProcess(bool conect)

/// <summary>
/// Список свободных портов
/// </summary>
List<string> GetFreePorts()

/// <summary>
/// Проверяет, можно ли открыть порт (существует ли он и не занят ли).
```

```
/// </summary>
bool IsPortAvailable(   portName)

/// <summary>
/// Подключение к COM-порту.
/// </summary>

bool Connect(  portName,    baudRate)

/// <summary>
/// Отключение от указанного порта.
/// </summary>
Disconnect(   portName)

/// <summary>
/// Закрыть все открытые порты.
/// </summary>
DisconnectAll()

/// <summary>
/// Отправка данных. Конфигурация задается строкой вида "4:9600" или "COM4:9600".
/// </summary>
  bool SendData(  configString,    message)

/// <summary>
/// Проверяет порт и, если в буфере есть законченное сообщение (\n или \r\n),
/// возвращает его и очищает буфер. Если сообщения нет — возвращает null.
/// </summary>
string ReadMessage(  configString)
```

9. Система управления интересами Interest

Это менеджер, который позволяет динамически менять весовую проводимость синапсов разных типов на основе запомненных интересов (сообщений определенного типа).

9.1. Системные события (_event of neuron)

События происходят, когда в системе случаются неординарные ситуации:

ErrorEvent – ошибка алгоритмов

SuccessEvent – хороший результат

UncertaintyEvent –неопределенность

FailureEvent – плохой результат

SuccessRememberEvent – хороший результат Memory

FailureRememberEvent – ничего не найдено Memory

Событие передает четыре параметра:

target - тип цели, к которой применялся результат

input – значение, которое передавалось

synapse – канал по которому передавалось значение

parametr – дополнительный параметр

ПРИМЕР: чтобы настроить кондуктор для создания нового концепта памяти при вводе текста, его параметры нужно настроить так:

_trigger failure

_target Memory

_synapse text_file

_event _inputTextChannel

Вы можете назначать любые типы событий и добавлять в них определенные нейроны при старте. Для этого в системном скрипте System:System {_RunScript} нужно написать:

1. SubscribeEvent(event_name, neuron);

Чтобы активировать любое событие по его имени из скрипта (в том числе и при срабатывании скрипта нейрона):

2. ShotBrainEvent(string event_name,object neuron,object parametr);

9.2. Системные типы синапсов (связи нейронов)

В зависимости от настройки своего _synapseReception параметра, большинство нейронов учитывают типы - (названия) синапса, по которому пришло сообщение.

Вы можете создавать любые названия типов. ГНС также сама может создавать человеко-читаемые и не читаемые названия для связей — на основе имеющихся концептов.

Однако существует несколько типов которые имеют определенный смысл.

Часть 4. Подсистема Агентов

1. Руководство пользователя: Визуальное проектирование графов AI-Агентов

1. Понятие AI- Агент

В ГНС Socrates концепты могут быть ИИ-агентами, и сложными подсистемами ИИ-агентов с собственной «умной» маршрутизацией взаимодействия. При этом с точки зрения Зоны внимания и остальной части ГНС «виден» только корневой агент подграфа. Как только он запускается в Зоне внимания, весь подграф начинает выполнять любые очень сложные действия как автоматическая подсистема.

При этом даже на крохотных моделях (2B) достигаются впечатляющие результаты.

Важно: Нельзя выстраивать внешние связи с подчиненными элементами Агентного графа — по сути он является описанием взаимодействия агентов (алгоритмом).

Что это даёт

- мы можем делегировать сложные, но хорошо отлаженные операции подсистемам агентов.
- при наличии соответствующих структур, Система может сама анализировать и собирать подграфы целевых агентов — подобно тому, как собираются обычные графы.

Важные уточнения

1. Агенты не могут быть связаны иными связями кроме _aCall

2. Вы должны понимать, что агентный граф любой сложности, с точки зрения Активной Зоны — только один корневой узел. Он является одновременно и входом, и выводом результата.

3. Исходя из п.п. №2 — никакие выходные связи от листьев агентного графа за его пределы (с другими типами нейристоров) не допустимы. А их создание приводит к непредсказуемым последствиям. Вплоть до падения всей ГНС.

2. Ручная сборка

В нашем редакторе вы можете визуально конструировать сложные рабочие процессы для ИИ: от простых ответов на вопросы до параллельной проверки фактов и условной маршрутизации.

Система сама соберет ваш визуальный граф в исполняемый код оркестратора и запустит его – как только активируется корневой Нейристор (узел) Агентной подструктуры. Все, что вам нужно — правильно настроить узлы (Nodes) и связи (Edges).

3. Свойства узла типа "Agent"

Каждый узел в графе управляется через словарь свойств (Variables). Свойства, начинающиеся с подчеркивания (_), отвечают за системную логику. Свойства без подчеркивания — это инструменты (Tools).

Основные (структурные) свойства

- _agentType — Определяет роль узла. Возможные значения:
- Agent — Реальный AI-помощник, который генерирует текст.
- Sequential — Менеджер. Запускает подчиненные узлы строго по очереди.
- Parallel — Менеджер. Запускает подчиненные узлы одновременно.
- Conditional — Менеджер-маршрутизатор. Направляет выполнение по одной из веток в зависимости от условия.

- _agent — Имя или "Личность" агента (например, TechExpert, Copywriter). Используется только если _agentType = Agent.
- _guidance — Системный промпт. Инструкция о том, как агент должен себя вести (например, "Ты строгий редактор, ищи стилистические ошибки").
- _planning — Стратегия мышления (none, ReAct, Reflection и др.). Выбирайте ReAct, если агенту нужно использовать инструменты (поиск в сети) и размышлять над результатом.

Свойства передачи данных (State & Context)

В сложных графах узлы обмениваются данными не напрямую, а через общее "Состояние" (State).

- _inputTemplate — Шаблон для формирования запроса к агенту. По умолчанию это просто {Input} (текст от предыдущего узла). Вы можете собирать сложные промпты, используя макросы:
  - {Input} — результат работы предыдущего узла.
  - {Original} — самый первый вопрос пользователя, с которым был запущен граф.
  - {State:ИмяКлюча} — вставка текста, сохраненного другим агентом.

Пример: Вопрос: {Original}. Черновик: {State:draft}. Оцени черновик.

- _saveStateKey — Имя ключа (ячейки памяти), куда узел сохранит свой ответ. Если указать draft, то следующий агент сможет прочитать этот текст через {State:draft}.
- _forwardInput — (true / false). Если true, узел проигнорирует свой собственный ответ при передаче данных дальше, и передаст следующему узлу то, что получил сам. Идеально для узлов-классификаторов.
- _maxTokens — Лимит длины передаваемых сообщений. 0 означает "использовать глобальный лимит движка 1024"

Свойства маршрутизатора (Для _agentType = Conditional)

_routeKey — Вычисляемое Имя ключа в State, значение которого определит, по какой связи (Edge) пойдет дальнейшее выполнение графа. В этом свойстве вы должны на основе входного значения, переданного агенту (передается неявно внутри процесса) вычислить имя ключа, соответствующего одному из ключей выходных связей агента. Или ветви default.

КАК ЭТО РАБОТАЕТ: Вы должны записать скрипт выбора или один из допустимых вариантов скрипта функции выбора. Смотрите раздел описания расширений DSL “SmartSwith”. При этом вы не должны использовать входной параметр key, поскольку это передается вашему скрипту автоматически. Возможный пример для выбора ключа:

```
@like:
столица Беларуси=>Минск;
столица России=>Москва;
Неизвестный город
```

Инструменты (Tools)

Любое свойство БЕЗ подчеркивания в начале воспринимается как инструмент.

websearch = true — дает агенту доступ к поиску в интернете.

datetime = true — дает агенту доступ к текущему времени.

1. Память агента (memory)

Агент может работать с векторной базой данных, если указать имя файла или путь к файлу:

_memoryName = file

2. Типы агентов

Тип агента назначается свойством _agentType

Обратите внимание, что некоторым агентам нужны инструменты, и память, а другим — инструкции.

Также — аналитические и программные агенты требую указание области действия Domain.

3. Назначение типов

Тип	Что он должен делать
agent	Универсальный агент: получает задачу, рассуждает и выдаёт ответ.
sequential	Последовательный конвейер: шаг 1 → шаг 2 → шаг 3.
parallel	Параллельный запуск нескольких агентов для разных мнений или частей задачи.
conditional	Выбор нужной ветки: например, вопрос о коде направить coder, а ошибку — debugger.
chat	Ведение обычного диалога с учётом истории сообщений.
assistent	Универсальный помощник: объясняет, отвечает, преобразует текст, помогает пользователю. Лучше назвать assistant.
toolWorker	Специализированный исполнитель инструментов: вызвать поиск, прочитать файл, выполнить команду, записать результат.
reActor	Агент по схеме ReAct: рассуждение → действие инструментом → наблюдение → следующее действие. Такая стратегия особенно предназначена для инструментальных агентов. docs.lm-kit
analyst	Аналитик: разбирает вопрос, сопоставляет факты, ищет закономерности, строит выводы.
coder	Генерирует или объясняет программный код.
debugger	Ищет причину ошибки, анализирует stack trace, логи и результаты тестов. Лучше назвать debugger, а не debuger.
editor	Редактирует текст или код по заданным правилам: исправляет, сокращает, форматирует, улучшает стиль.
planner	Составляет план выполнения сложной задачи, но обычно не выполняет его сам. Лучше назвать planner, а не plenner.
script	Выполняет скрипт или подготавливает команды для shell, PowerShell, Python и т. п.
function	Вызывает строго определённую функцию с параметрами и возвращает её результат. Это скорее исполнитель типизированной операции, чем свободный агент.

plugin	Работает через подключаемый внешний модуль: API, базу данных, Git, файловую систему или другой сервис.

4. Кому нужны инструменты

Инструмент требуется тогда, когда агент должен сделать что-то вне самой языковой модели: прочитать файл, выполнить код, отправить запрос, найти документ, вызвать API или изменить состояние системы.

Обязательно или почти обязательно инструменты нужны следующим типам:

- toolWorker — его основная задача состоит именно в вызове инструментов;
- reActor — без инструментов ReAct превращается почти в обычное рассуждение;
- script — нужен исполнитель скриптов или shell-команд;
- function — нужна зарегистрированная функция с входными параметрами;
- plugin — нужен сам подключённый плагин или адаптер;
- debugger — желательно иметь доступ к коду, логам, тестам и запуску программы;
- coder — инструменты нужны, если он должен читать или изменять файлы, компилировать и тестировать код;
- editor — инструменты нужны, если он редактирует реальные файлы, а не только возвращает исправленный текст.

Инструменты не обязательны, но могут быть полезны для:

- analyst — поиск по базе знаний, RAG, калькулятор, статистика;
- assistant — календарь, поиск, файлы, база данных;
- chat — обычно это просто диалог, но можно подключить поиск или память;
- planner — план можно составить без инструментов, а для автоматического выполнения понадобятся другие агенты;
- agent — универсальный тип; наличие инструментов зависит от задачи.

A sequential, parallel и conditional сами инструменты не используют. Они управляют другими узлами. Например:

conditional

├── analyst — анализ вопроса

├── coder — работа с кодом

└── chat — обычный ответ

sequential

├── planner — составить план

├── toolWorker — получить данные

├── analyst — обработать данные

└── editor — оформить результат

Таким образом:

agent, assistant, chat, analyst, coder, debugger, editor и planner - описывают роль агента

sequential, parallel и conditional — способ соединения агентов, а toolWorker, script, function и plugin — способ выполнения внешних операций.

Настройка связей (Edges)

Связи определяют направление потока данных и порядок выполнения.

Вес (Weight) — Для узлов Sequential и Parallel связи сортируются по весу. Узел с наибольшим весом связи выполнится первым.

_routeValue — Свойство связи. Используется только исходящими связями от Conditional узла. Указывает, при каком значении в State нужно переходить по этой связи (например, _routeValue = tech).

1. Программная сборка и вызов Агента

Сборка

Для чисто программной сборки Агентов воспользуйтесь командами языка:

```
targetId = neuron(name,type,shema)
{
property1= value1;
property2= value2;
……
};
BuildAgentCall(sourceId,targetId [, Weight =100] [,_routeValue] = null); // соединяет агента входящей связью
```

2. Программный запуск Агентного подграфа

Подграф можно запустить чисто программно и получить символьный ответ:

text = AgentExecute(node, input);

node – id или полное имя вершины подграфа.

3. Практические примеры сборки графов

Пример 1: Простой конвейер (Черновик -> Редактор)

Внимание! Полную инструкцию ищите в ПРИЛОЖЕНИЯХ (в конце документа)

Задача: Один агент пишет статью, второй ее редактирует.

Создаем узел-менеджер _agentType = Sequential.

Создаем Узел 1 (Писатель):

_agentType = Agent, _agent = Writer.

_guidance = Напиши короткий текст на заданную тему.

_saveStateKey = my_draft (он сохранит свой текст сюда).

Создаем Узел 2 (Редактор):

_agentType = Agent, _agent = Editor.

_guidance = Исправь ошибки в тексте.

_inputTemplate = Вот черновик: {State:my_draft}. Сделай его лучше.

Связи: Соединяем Менеджера с Узлом 1 (вес 10) и с Узлом 2 (вес 5).

Пример 2: Умная маршрутизация (Роутер)

Задача: если вопрос по коду — отвечает Программист. Если по продажам — Маркетолог.

Создаем Узел 1 (Классификатор):

_agentType = Agent

_guidance = Ответь одним словом: tech или biz.

_saveStateKey = category (сохраняем метку).

_forwardInput = true (ВАЖНО! Мы хотим, чтобы программист увидел сам вопрос, а не слово "tech").

Создаем Узел 2 (Роутер):

_agentType = Conditional

_routeKey = category (роутер будет смотреть на метку от классификатора).

Создаем Экспертов (Узел 3 и 4):

Узел 3 (TechExpert) и Узел 4 (BizExpert). Настраиваем им _guidance.

Связи:

Классификатор -> Роутер.

Роутер -> TechExpert (в свойствах связи указываем _routeValue = tech).

Роутер -> BizExpert (в свойствах связи указываем _routeValue = biz).

Пример 3: Сложный вложенный граф (Шедевр оркестровки)

Этот паттерн объединяет предыдущие. Поток: Классификация -> Маршрутизация к Эксперту -> Параллельная проверка фактов и стиля -> Финальная сборка.

Архитектура в визуальном редакторе:

Главный узел (Root): _agentType = Sequential. От него идут связи ко всем этапам.

Этап 1 (Классификатор): Описан в Примере 2. Сохраняет _saveStateKey = label, пробрасывает инпут (_forwardInput = true).

Этап 2 (Генерация черновика):

Здесь мы ставим Conditional узел (_routeKey = label).

От него идут связи к TechExpert и BizExpert.

ОБА эксперта должны иметь свойство _saveStateKey = expert_draft. Кто бы из них ни сработал, он запишет свой черновик в эту ячейку памяти.

Этап 3 (Ревьюеры):

Ставим узел Parallel.

К нему подключаем двух агентов: FactChecker и StyleChecker.

В их _inputTemplate пишем: Проверь этот текст: {State:expert_draft}.

Так как это Parallel узел, он соберет ответы обоих ревьюеров в один большой текстовый блок и передаст дальше.

Этап 4 (Финальный Редактор):

Ставим обычного Agent (Reviser).

_guidance = Собери финальный ответ.

_inputTemplate = Вопрос: {Original}. Черновик: {State:expert_draft}. Отзывы критиков: {Input}.

Как это работает:

Сборщик графа (Graph Assembler) начнет с Главного узла. Поймет, что это Sequential, и выстроит очередь. На втором этапе он наткнется на Conditional и вложит его внутрь очереди. На третьем вложит Parallel.

В итоге вы получаете сложнейшую логику, просто соединив кубики на экране и прописав ключи {State:...}!

4. Советы для профессионалов

1. Не делайте циклов в Менеджерах. Графы Sequential, Parallel и Conditional должны представлять собой дерево, направленное сверху вниз.

2. Используйте {Original}. Если на 5-м шаге цепочки агенту нужно вспомнить изначальный запрос пользователя, не пытайтесь передавать его через все узлы. Просто используйте макрос {Original} в шаблоне _inputTemplate.

3. Тестируйте инструменты. Если агенту не хватает информации, добавьте в узел переменную websearch = true и установите _planning = ReAct. Агент сам сходит в интернет перед тем, как дать ответ.

Синтаксис пакетного управления ГНС

1. Базовый синтаксис (Основы)

- Узел: (Schema:Name {key=val; flag})
- Ребро: -[Tag1, Tag2 {weight=1.0}]->
- Цепочка: (A) -[link]-> (B) -[link]-> (C)
- Принцип Upsert: если узел с таким Schema:Name уже есть, он не дублируется, а обновляется (добавляются новые свойства из {...}).

2. Динамика и Макросы (Препроцессор @)

Перед выполнением любой команды движок ищет символы @ и заменяет их значениями из C#-скрипт движка (ShpScript).

- @VarName: Простая подстановка переменной.
- @VarName[Default]: Подстановка с защитой (если переменной нет, используется Default).
- @(...): Вычисление любого C#-выражения "на лету".
- Пример: (Node {val=@(Math.Sin(45) * 100)})

3. Прототипы и Наследование (Ключевое слово FROM)

Используется в методе db.InstantiateChain(). Позволяет копировать "генетику" (свойства и связи) из существующих узлов-шаблонов.

- Синтаксис: (НовыйУзел FROM Прототип {переопределение_свойств})
- Ребро: -[Связь FROM ШаблонРебра]->

- Логика: сначала копируются все свойства прототипа, затем применяются свойства из фигурных скобок {...} (новые добавляются, старые перезаписываются).

4. Поиск и Маршрутизация (Пути / и \)

Используется в методе db.ExecutePath(). Позволяет ходить по связям как по файловой системе.

- /Tag: Шаг вперед по исходящей связи с тегом Tag.
- \Tag: Шаг назад по входящей связи с тегом Tag.
- =>Filter: Фильтрация текущего списка узлов.

5. 12 Практических Примеров

1. Простое создание с переменными

```
// sh='Net', name='Sensor1'
(@sh:@name {type="input"; active})
```

Результат: создаст узел Net:Sensor1 со свойствами type="input" и active="true".

2. Инстанцирование нейрона по шаблону

```
(Layer1:N1 FROM Models:ReluNeuron)
```

Результат: создаст Layer1:N1, скопировав все настройки (триггеры, веса) из Models:ReluNeuron.

3. Переопределение свойств прототипа

```
(Layer1:N2 FROM Models:ReluNeuron {threshold=0.8; _importance=100})
```

Результат: создаст нейрон, но заменит его порог на 0.8, даже если в шаблоне было 0.5.

4. Создание связи по прототипу (Синапс)

```
(A) -[syn1 FROM Models:ExcitatorySynapse {w=2.5}]-> (B)
```

Результат: Связь syn1 унаследует все параметры "Возбуждающего синапса", но получит индивидуальный вес 2.5.

5. Полностью динамическая сборка

```
(@layer:@id FROM @proto {@key=@val})
```

Результат: Все параметры (схема, имя, шаблон, свойства) подставятся из C# переменных.

6. Смешанная цепочка (Старый + Новый)

```
(ExistingNode) -[link]-> (NewNode FROM Models:BaseTemplate)
```

Результат: найдет уже существующий узел и пробросит от него связь к новому узлу, созданному по шаблону.

7. Использование вычислимых выражений

```
(Neuron {weight=@(Rand.NextDouble()); timestamp=@(DateTime.Now.Ticks)})
```

Результат: при каждом вызове будут генерироваться новые случайные числа и время.

8. Маршрутизация: Поиск "дедушки"

(User:Andrey) \child \child => User:*

Результат: найдет всех, чьим "внуком" является Андрей.

9. Сложная навигация (Кто охотится на моих врагов)

:Кот /враг /враг \охотится => Птица:

Результат: пройдет путь: Кот -> его Враги -> Враги врагов -> Кто на них охотится -> Фильтр "Птицы".

10. Логический поиск с WHERE

Sensor:* {type=temp} WHERE value > @(GlobalThreshold + 10)

Результат: найдет все датчики температуры, значение которых выше динамического порога.

11. Массовое создание в цикле (через C#)

// C# код

for(int i=0; i<3; i++)

db.InstantiateChain($"(Cell:{i}) -[step]-> (Cell:{i+1} FROM Models:Proto)");

Результат: создаст цепочку нейронов, где каждый следующий узел создается по шаблону.

12. "Умное" связывание (Side-effects)

(Task:Main) -[sub @(lastId = CreateSubTask(); "link")]-> (Task:@lastId)

Результат: Скрипт внутри @(...) выполнит функцию CreateSubTask(), сохранит результат в переменную lastId, и эта же переменная тут же подставится в имя целевого узла.

6. Рекомендации по использованию API:

1. Если нужно создать или обновить сложную структуру: используйте db.InstantiateChain(command).
2. Если нужно найти узлы по цепочке связей: используйте db.ExecutePath(query).
3. Если нужно найти узлы по математическим условиям: используйте db.FindNodesLogical(query).
4. Помните: все переменные @ берутся из MainEngine.Instance.ShpScript.Variables. Перед вызовом команд убедитесь, что вы положили туда нужные данные.

Markdown команды (#...)

Markdown команды позволяют легко описывать структуру ГНС прямо из скрипта, комбинируя с C# и макросам.

1. Несколько статей #...# подряд

Если нужно описать несколько статей Markdawn, то писать #End нужно только в конце.

#Tree

```
……
#Let
…..
#Tree
….
#End
```

2. Описание дерева (#Tree)

Позволяет описывать дерево (антологию), состоящее из описания узлов, их эмбеддинги и свойства.

Свойства могут описываться явно Ключ = Значение; и через переменные или функции, начинающиеся с символа @. $ - задает свободную (вне дерева) связь с любыми узлами

Пример: создание дерева:

```
#Tree
name ="Дерево";
result Свойство(a){
name + "-" + a;
} ;

#tree
ROOT:System:System // корень, от которого пойдет дерево
ROOTSCHEMA:System // схема по умолчанию
MONONAME:true // сливает все слова имени, делая заглавные буквы на стыках
AUTONAME: true // создает синоним (в свойствах) из описания в скобках
PROPVECTOR: true // Все значения свойств будут получать локальный эмбеддинг

## @name // узел дерева задан через переменную, а предок по умолчанию
[Это классификатор живых деревьев] // создает описание и эмбеддинг
<neuron> // переключает тип узла и для всех потомков
@ Свойство("место") = почва; // имя свойства задано через функцию
@ Свойство("ствол") = высокий;
$ включает > ветка, // устанавливается связь «включает» с двумя узлами.
корень; // если еще отсутствуют — создаются без свойств
## береза: @name (birch) // второй вариант имени (…), а предок задан явно, но переменной
@ Свойство("цвет ствола") = бело-черный;
## дуб // тот-же предок
@ Свойство("твердость") = высокая;
## сосна // тот-же предок
листья = иголки;
#End
```

3. Описание связей (#Link)

Позволяет описывать произвольные связи типа 1 к многим, или много к одному.

Секция заканчивается перед любой секцией начинающейся на #. Например, #End

Поддерживается прямая и косвенная адресация (через переменную или функцию)

Пример:

```
#Link
схема:имя > [tag1,tag2,..] схема1:имя1, схема2:имя2, … ; // 1 к 1; 1 к многим
схема1:имя1, схема2:имя2, … >[tag1,tag2,..] > схема:имя; // многие к 1
$схема; //схема для всех
$Tag:tag,…; //tag для всех (по молчанию -link)
имя > tag> имя1, имя2, … ;
$@схема; // схема для всех через переменную или функцию
$Tag: @tag,…;
@имя > @имя1, @имя2, … ; // параметры через переменную или функцию
#End
```

4. Получение унаследованных свойств (GetNodeGeneProps)

Метод GetNodeGeneProps(объект [, limit]); позволяет получить всю иерархию унаследованных свойств

нейрона в антологии, в формате MD.

Объект можно задавать полным именем, индексом(Id), ссылкой (Node).

Limit – число уровней, или имя предка, до которого собирать информацию.

5. Описание произвольного графа (#Let)

Смотри подробный синтаксис выше

```
#Let
(Net:A FROM Models:T1 {threshold=0.8; new_prop="hello"}) -
[link FROM Models:SynapseFast {weight=2.5}]->
(Net:B FROM Models:T2 {is_output=true});
(Proto) -[Models:InhibitorySynapse {_Weight=-1.5; _Importance=100}]-> (Proto);
#End
```

Часть 5. Для разработчиков, использующих DLL Socrates

Встроенная графовая БД

SocratesDSL GraphDB — это высокопроизводительная встраиваемая (in-memory) графовая база данных, ориентированная на хранение семантических сетей, контекста ИИ и динамических структур знаний.

Ключевые особенности версии 2.0:

- Property Graph Model: Узлы и ребра поддерживают произвольный набор свойств (key-value).
- Инвертированные индексы: Мгновенный поиск по свойствам и значениям
- (O(1) / O(Nmatches)) .
- Гибкий DSL: Мощный текстовый язык запросов с поддержкой Wildcards (*).
- AI-Oriented: Встроенные механизмы инстанцирования (прототипы), кластеризации смыслов (LPA) и очистки оперативной памяти.
- Визуальный редактор: Интерактивная визуализация с физическим движком.

1. Архитектура Данных

1.1. Узлы (Nodes)

Базовая единица информации.

- Id: int (Уникальный идентификатор).
- Schema: string (Пространство имен, напр. Global, User, DSL:Topic). Поддерживает иерархию через двоеточие.
- Text: string (Имя узла).
- Variables: Dictionary<string, string> (Свойства/Атрибуты).
- Системные: _ui_x, _ui_y (координаты), community_id (кластер), _prototype (родитель).

1.2. Ребра (Edges)

Направленные связи между узлами.

1. SourceId

→

TargetId.

2. Tags: HashSet<string> (Метки типа связи, напр. likes, causes).
3. Variables: Dictionary<string, string> (Свойства связи, напр. weight=0.9, since=2023).

1.3. Индексация и Производительность

Для работы с объемами до 100,000 узлов и 1,000,000 связей используются три уровня индексов:

- Text Index (_textIndex):
- Ключ: Schema:Text

→

Значение: NodeId.

- Позволяет находить конкретный узел за

O(1)

.

- Node Property Index (_nodePropIndex):
- Структура: PropName

→

PropValue

→

HashSet<NodeId>.

- Позволяет мгновенно находить узлы по запросам вида * {active=true}.
- Edge Property Index (_edgePropIndex):
- Структура: PropName

→

PropValue

→

HashSet<Edge>.

- Позволяет фильтровать связи без перебора всего графа.

ВАЖНО: Изменение свойств (Variables) напрямую запрещено, так как это рассинхронизирует индекс. Используйте API методы UpdateNodeProperty и UpdateEdgeProperty.

1.4. Скрипты

Команды могут быть написаны в одном скриптовом текста с комментариями, в обычном для C# формате внутри команды ExecuteScript(script):

```
ExecuteScript( @"
// Создаем персонажа с длинным описанием
(Hero:Geralt)
{ desc = "Ведьмак из Ривии. // этот текст не комментарий, т.к. в кавычках"; hp = 100; status = "active"
}
// Связь через перенос строки
(Hero:Geralt) -[friend_of {since="1234"}]-> (Bard:Dandelion)
// Команда NEW тоже может быть разбита
NEW: (Hero:Copy)
FROM (Hero:Geralt) ";
```

1.5. Шаблоны

В гБД реализована система Прототипного наследования (Инстанцирования) через метод Instantiate и парсер команд NEW: () FROM ().

Как это работает (Концепция):

В графовых базах данных шаблоном является любой обычный узел, у которого прописаны какие-то переменные (Variables) или связи (Edges). Когда вы создаете новый узел "из шаблона", гБД копирует все переменные и копирует все исходящие связи шаблона на новый узел.

Пример использования в коде (C#):

Создаем шаблон (Прототип):

```
codeC#
// Создаем узел в схеме System.Templates
var personTemplate = db.AddNode("PersonTemplate", "System.Templates");
// Добавляем свойства по умолчанию
db.UpdateNodeProperty(personTemplate.Id, "Age", "0");
db.UpdateNodeProperty(personTemplate.Id, "IsActive", "true");
db.UpdateNodeProperty(personTemplate.Id, "Type", "Human");
// Добавляем связь (шаблонный человек обязательно имеет связь с планетой Земля)
var earthNode = db.AddNode("Earth", "Planets");
db.AddEdge(personTemplate.Id, earthNode.Id, "lives_on");
```

Создаем инстанс (Новый узел по шаблону):

Вы используете встроенный метод Instantiate:

```
codeC#
// Формат: "НоваяСхема:ИмяНовогоУзла {Переопределенные свойства}", "ФильтрДляПоискаШаблона"
db.Instantiate(
newInstanceRaw: "Employees:JohnDoe {Age=35; Department=\"IT\"}",
templateFilter: "System.Templates:PersonTemplate"
);
```

Что произойдет под капотом Instantiate:

- Метод найдет PersonTemplate.
- Создаст узел JohnDoe в схеме Employees.
- Скопирует переменные шаблона: Type="Human", IsActive="true".
- Переопределит переменную из запроса: Age="35".
- Добавит новую переменную: Department="IT".
- Добавит системную переменную: _prototype="System.Templates:PersonTemplate".
- Скопирует связи: Узлу JohnDoe автоматически добавится связь [lives_on] -> (Planets:Earth).

Использование через скрипты (Командный парсер):

гБД умеет делать это напрямую из текста. Если вы введете команду:

```
NEW: (Employees:AliceSmith {Age=28}) FROM (System.Templates:PersonTemplate)
```

И передадите её в метод db.ExecuteCommand(...), произойдет ровно то же самое.

1.6. Язык запросов (DSL Syntax)

Текстовый формат для создания (ExecuteCommand) и поиска (FindNodes, ExecuteQuery).

1. Формат Узла

([Schema :] Name [{ key=val; flag }])

Пример	Описание
(User:Andrey)	Узел схемы User с именем Andrey.
(Server01)	Схема по умолчанию (Global).
(File:Log {size=10mb; readonly})	Узел со свойствами. Флаг readonly означает true.
(DSL:Topic:Science)	Иерархическая схема DSL:Topic.

2. Формат Ребра

-[Tag1, Tag2 [{ key=val }]]->

Пример	Описание
-[likes]->	Простая связь.
-[uses, depends]->	Связь с двумя тегами.
-[route {dist=50}]->	Связь со свойством dist.

3. Wildcards (Маски поиска)

Используются в фильтрах FindNodes.

Фильтр	Результат
*	Все узлы базы.
User:*	Все узлы схемы User.
*:Admin	Узлы с именем Admin в любой схеме.
DSL:*:*	Узлы, схема которых начинается на DSL.
* {active}	Любые узлы, имеющие свойство active.

1.7. API Разработчика (C#)

1. Основные операции

```
// Управление схемой по умолчанию
string oldSchema = db.DefaultSchema; db.DefaultSchema = "Temporary";
// ... делаем дела …
db.DefaultSchema = oldSchema;


// Создание (Upsert)
var node = db.AddNode("Name", "Schema", new Dictionary<string, string> {{"age","25"}});
```

Полная перезагрузка узла:

- Сохраняет ID и ВХОДЯЩИЕ связи.

- Удаляет старые СВОЙСТВА.
- Удаляет старые ИСХОДЯЩИЕ связи.
- Применяет новые свойства и создает новые исходящие связи из команды.
- Команда, например: "(Step:1) -[next]-> (Step:2)"</param>

```
var Node = ReplaceNodeFullState(string commandLine);
db.AddEdge(node1.Id, node2.Id, new[] {"link"}, new Dictionary<string, string> {{"w","1"}});


// Обновление (Безопасное)
db.UpdateNodeProperty(node.Id, "age", "26"); // Обновляет и индекс
db.UpdateEdgeProperty(edge, "w", "2");


// Поиск
var users = db.FindNodes("User:*");
var admins = db.FindNodes("* {role=admin}");


// Поиск пути (Traverse)
// Найти всех, на кого ссылаются активные узлы через связь 'follows'
var targets = db.ExecuteQuery("(* {active}) -[follows]-> (*)");


// Удаление
db.DeleteNode("User:Old"); // Удаляет узел и все его связи
db.DeleteEdge(edgeInstance);
```

2. AI-специфичные функции
- Инстанцирование (New Instance):

Создает новый узел, копируя свойства и исходящие связи из узла-шаблона.

```
codeC#
// DSL
db.ExecuteCommand("NEW: (Ctx:Session1) FROM (Template:Conversation)");
// C#
db.Instantiate("Ctx:Session1", "Template:Conversation");
```

- Очистка оперативной памяти:

Удаляет все узлы указанной схемы (например, временный контекст), корректно очищая связи с глобальной памятью.

```
codeC#
db.DeleteNodesBySchema("Operative:", prefixMatch: true);
```

- Кластеризация (Community Detection):

Алгоритм LPA (Label Propagation) для поиска смысловых групп.

```
codeC#


var detector = new CommunityDetector(db);
```

```
detector.DetectCommunitiesLPA(); // Записывает свойство 'community_id'
```

3. Глубокий семантический поиск узлов (версия ГНС)

Версия ГНС поддерживает свойство Узла

Description которое создает или выдает строку в словаре свойств _description, которая должна содержать краткое смысловое описание узла на любом естественном языке. Первоначально = null

При назначении или изменении этого свойства узлу создается смысловой вектор в N-мерном пространстве смыслов.

Далее мы можем найти нужное количество релевантных узлов по смысловому описанию:

```
/// Двухступенчатый семантический поиск:
/// Этап 1: Быстрый векторный поиск (Cosine Similarity) для отбора topK1 кандидатов.
/// Этап 2: Глубокое осмысление через Reranker для выдачи topKOut финальных результатов.
/// <param name="findDescription">Текстовый запрос (контекст)</param>
/// <param name="topK1">Количество узлов для грубого отбора (например, 100)</param>
/// <param name="topKOut">Количество финальных узлов после перекрестного анализа (например, 5)</param>
/// <returns>Список узлов и их финальный скор релевантности</returns>
public List<(Node Node, float Score)> SemanticSearch(string findDescription, int topK1 = 100, int topKOut = 5)
```

Часть 6. Встроенный микро-Refal (написание правил)

Микро-Рефал обрабатывает строку токенов (слов или концептов, разделенных пробелами), которая поступает в нейрон.

Интерпретатор берет переменные узла (Node.Variables) и пытается сопоставить их ключи (Паттерны) со входной строкой. Если совпадение найдено, он выполняет значение переменной (Действие).

Формат правила (в методе RuleExecute):

Паттерн => Действие

- Левая часть (Паттерн): описывает, как должна выглядеть входная строка, что пропустить, а что захватить в переменные.
- Правая часть (Действие): описывает, как собрать ответ из захваченных переменных, или какой скрипт запустить.

1. Левая часть: Анализ строки и пропуск элементов

Входная строка разбивается пробелами на массив токенов. Паттерн также разбивается пробелами. Интерпретатор идет слева направо.

1.1. Джокеры (Wildcards) для пропуска "мусора":

В асинхронной нейросети сообщения часто содержат шум. Для его обхода используются два символа:

- ? — Один любой токен. Безусловно пропускает ровно одно слово/токен.
- * — Сколько угодно токенов (Пропуск шума).
- Особенность вашей реализации: Звездочка реализует механизм Look-ahead (заглядывания вперед). Она проглатывает токены входной строки до тех пор, пока следующий за ней токен паттерна не совпадет с текущим словом.
- Если * стоит в самом конце паттерна — она проглатывает весь оставшийся «хвост» сообщения.

Пример:

- Вход: Кот съел черную мышь быстро
- Паттерн: Кот ? * мышь *
- Как сработает: Кот совпал

→

→

? проглотил съел

→

→

* проглотила черную (так как дальше идет мышь)

→

→

мышь совпала → * проглотила быстро. Итог: Совпадение (Success).

→

1.2. Умные переменные (Экстракторы)

Самая мощная часть вашего интерпретатора. Экстракторы не просто захватывают слова, они могут анализировать их смысл.

Синтаксис экстрактора строится через точки:

[Тег]Тип.Цель.[Порог].ИмяПеременной

(Элементы в квадратных скобках необязательны).

Виды типов (Type):

1. Строгий захват: s или t

Просто берет текущий токен и сохраняет его в переменную.

Синтаксис: s.ИмяПеременной (или t.ИмяПеременной — в вашем коде они работают одинаково).

Пример: Паттерн Привет s.Name для входа Привет Алиса сохранит в переменную Name значение Алиса.

2. Нечеткий текстовый поиск (Jaro-Winkler): n

Сравнивает токен с Целью по написанию (орфографии), допуская опечатки.

Синтаксис: n.Цель.ИмяПеременной (порог по умолчанию 0.75) или n.Цель.[Порог].ИмяПеременной.

Пример: n.Собака.0.8.Animal захватит слово Сабака (т.к. оно похоже на "Собака" больше чем на 80%) и сохранит его в Animal.

3. Семантический/Векторный поиск: v

Использует нейросетевые эмбеддинги (_graphEngine.VectorSimilarity). Сравнивает СМЫСЛ, а не буквы.

Синтаксис: v.Цель.ИмяПеременной или v.Цель.[Порог].ИмяПеременной.

Пример: Паттерн v.Транспорт.0.7.Target применительно к тексту Я купил велосипед. Токен велосипед по смыслу близок к Транспорт, порог пройден, в переменную Target запишется велосипед.

4. Проверка скриптом (Условный захват): x

Выполняет C# скрипт ShpScript для проверки текущего токена. Внутри скрипта токен доступен как value.

Синтаксис: x.[Скрипт].ИмяПеременной

Пример: x.[value.Length > 5].LongWord — захватит токен, только если в нем больше 5 символов.

1.3. Механизм Тегов (Тегирование синапсов)

В вашем коде есть потрясающая поддержка тегов, приходящих из синапсов (поля Tag в объекте Impulse).

Во входной строке они представлены форматом [ИмяТега]Текст.

Вы можете заставить паттерн реагировать только на сигналы с определенного синапса:

Синтаксис: [Тег]Тип...

Пример: Паттерн [Visual]s.Object совпадет с токеном [Visual]Кошка, запишет в Object слово Кошка. Но он проигнорирует токен [Audio]Кошка.

1.4. Динамическое вычисление Целей и Порогов

Ваш код умеет использовать ShpScript прямо внутри паттерна, если заключить выражение в [ ]. Это позволяет делать правила адаптивными!

v.[Global.CurrentContext].0.7.Topic — Смысловая цель берется не жестко из текста, а из переменной движка.

v.Опасность.[Node.Sensitivity].Threat — Порог чувствительности берется из свойства самого нейрона! Это позволяет модулировать возбудимость сети на лету.

2. Правая часть: Формирование ответа и Действия

Если Паттерн (левая часть) успешно сопоставился со строкой, выполняется Действие (правая часть).

2.1. Выполнение C# кода (Скрипты)

x.[Скрипт].Var — Выполняет ShpScript. Объект доступен в скрипте как переменная value.

Пример: x.[value.Excitation > 1.5].HotNode (Захватит только сильно возбужденный нейрон).

2.2. Логические операторы (| и !)

Делают правила короткими и мощными, избавляя от дублирования кода.

1. Оператор ИЛИ (|) - Обобщение категорий

Разделяет альтернативные проверки для одного и того же объекта.

Синтаксис: Условие1|Условие2|Условие3.Var

Пример: v.Опасность.0.8|v.Авария.0.8|prop.Tags.Alert.Target

(Если объект семантически похож на "Опасность" ИЛИ "Аварию", ИЛИ имеет тег "Alert" — записать в Target).

2. Оператор НЕ (!) - Торможение и фильтрация

Отвергает объект, если он соответствует условию. Если объект НЕ соответствует, его ID сохраняется в переменную.

Синтаксис: !Условие.Var

Пример: !prop.Schema.System.N1

(Захватить объект в N1, только если он НЕ является системным узлом).

Пример 2: !v.Животное.0.7.Item

(Захватить объект, только если по смыслу он НЕ является животным).

2.3. Неупорядоченные блоки { ... } (Асинхронный матчинг)

В Активной Зоне сигналы приходят параллельно. Если вам важен факт наличия группы объектов, но не важен порядок их поступления, оберните их в фигурные скобки.

(Важно: вокруг скобок должны быть пробелы)

Синтаксис: { ПаттернА ПаттернБ ПаттернВ }

Как работает: Движок проверит все возможные перестановки (биекцию) входящих объектов, чтобы удовлетворить условия внутри блока.

Пример: CoAct { type.Node.N1 type.Node.N2 }

(Правило найдет два узла независимо от того, какой из них в массиве нулевой, а какой первый).

2.4. Фильтрация по Синаптическим Тегам

Если вы хотите реагировать только на сигналы, пришедшие по определенным связям, укажите тег в квадратных скобках перед оператором.

Синтаксис: [Тег]Оператор...

Пример: [Visual]s.Object (Проигнорирует импульс от [Audio]-синапса, захватит только визуальный).

1. Подстановка переменных

В правой части вы можете выводить захваченные переменные.

Важно: согласно вашему коду result.Replace("s." + v.Key, v.Value), чтобы вывести переменную VarName, вы должны написать s.VarName (или e.VarName, t.VarName — движок заменяет все три префикса).

- Пример:
- Паттерн: * меня зовут s.Name
- Вход: Привет мир меня зовут Джон
- Действие: очень приятно s.Name
- Результат: очень приятно Джон

2. Выполнение скриптов в Действии

Если правая часть содержит круглые скобки (...), интерпретатор воспринимает содержимое скобок как SHScript (вызов метода SHScriptEvaluate).

В этот скрипт передается весь уже сформированный текст (с подставленными переменными) в качестве параметра value.

- Пример:
- Правило: v.Животное.0.7.Animal => (Memory.Save("Видел", "s.Animal"))
- Как работает: если во входе есть что-то похожее на животное (например, "корову"), переменная Animal станет "корову". Затем выполнится скрипт Memory.Save("Видел", "корову").

2.5. Практические примеры для вашей архитектуры (Нейрогенез)

Как это использовать в Конструкторе нейрогенеза, который мы обсуждали:

Задача: выявить, что событие (CoAct) связано с семантикой опасности, и создать узел.

Входная строка от Детектора:

CoAct 105 201 30 (где 105 - Гололед, 201 - Скорость, 30 - Авария)

Мы хотим, чтобы правило сработало, только если целевой узел (30) как-то связан со словом "Беда".

Как будет выглядеть Правило:

CoAct s.NodeA s.NodeB v.Беда.0.6.Target => CMD_CREATE_AND s.NodeA s.NodeB s.Target

Разбор работы этого правила:

1. Слово CoAct совпадает в точности.

2. Токен 105 захватывается как строка в NodeA (тип s).

3. Токен 201 захватывается как строка в NodeB (тип s).

4. Токен 30 (TargetId) передается в векторный анализатор v. Движок обращается к узлу 30, получает его текст (например, "Авария на дороге") и сравнивает векторно со словом "Беда".

5. Если порог 0.6 пройден — 30 сохраняется в Target.

6. В правой части переменные подставляются. Результат, который вернет интерпретатор: CMD_CREATE_AND 105 201 30.

Часть 7. API DLL Socrates

Часть 8. Советы и частые вопросы

Настройка ГНС

1. Ввод информации

1. Как отключить авто трансляцию сообщения в устройство вывода?

Нейрон: System:System" свойство "_echo_off" → false

2. Как программно добавить или изменить свойство нейрона?

В скрипте:

```
NeuronPropertySet( neuron, prop, value);
C#:
MainEngine.Instance.GraphDb.UpdateNodeProperty(int nodeId, string key, string newValue);
```

или

```
MainEngine.Instance.GraphDb.UpdateNodeProperty(Node node, string key, string newValue);
```

3. Как программно ввести информацию извне (например, от датчика)?

В скрипте:

```
SendMessage(string text, int mode,List<AttachedFile> attachFiles=null);
```

mode — 0; 1; 2; (ввод, проанализировать файл, запомнить файл)

```
C#:
MainEngine.Instance.SendMessage(string text, int mode,List<AttachedFile> attachFiles=null);
```

4. Как включить \ выключить векторизацию свойства Нейрона?

а) В графическом редакторе — активируйте \ дезактивируйте флажок в таблице свойств нейрона, против нужного свойства.

б) Программно:

```
GNN.UpdateNodeProperty(node, key, value, true\false);
```

Проверить:

```
x= GNN.IsVectorized(node, key);
x= node.IsVectorized(key);
```

2. Вывод информации

1. Как отключить вывод системных сообщений?

В конечном продукте вероятно не нужно показывать системные сообщения пользователю, а достаточно настроить нейронные цепи на правильное реагирование.

Нейрон: System:System" свойство "_systemOutChannel" → удалите все

2. Как включить автоматический вывод сообщений об ошибках?

Нейрон: System:System" свойство "_systemOutChannel" → auto

3. Как отключить вывод сообщений об ошибках?

В конечном продукте, вероятно, не нужно показывать ошибки пользователю, а достаточно настроить нейронные цепи на правильное реагирование.

Нейрон: System:System" свойство "_errorOutChannel" → удалите все

4. Как включить автоматический вывод сообщений об ошибках?

Нейрон: System:System" свойство "_errorOutChannel" → auto

3. Синапсические связи

1. Как создать рефлекс, анти-рефлекс?

Рефлексы и анти-рефлексы — связи моментального безусловного исполнения.

При этом анти-рефлексы моментально гасят и выталкивают нейроны из зоны внимания.

В графическом редакторе просто выберите соответствующую галочку.

Программно, это устанавливается значением _weight для синапса примерно в 2 раза больше, и соответственно в 2 раза меньше триггера нейрона цели (__trigger)

2. Как создать чисто информационную (пассивную) связь нейронов?

В графическом редакторе выберите галочку Struct.

Программно, это устанавливается значением _weight =0 и _importance = 100

4. Визуальный редактор связей

1. Как собрать визуальный модуль?

Визуальный модуль позволяет убрать (свернуть) лишние нейроны и их связи, относящиеся к некоторой условно выделенной вами группе нейронов. На работу это не влияет.

Для компоновки нейристоров в модуль необходимо выполнить такую последовательность:

1. Выбрать корень модуля (по вашему усмотрению_

2. Правой кнопкой по нему , выполнить «Создать модуль».

3. Чтобы свернуть \ развернуть модуль требуется двойной клик по нему.

4. Чтобы добавить нейристоры к модулю, разверните его, и правой кнопкой по добавляемому нейристору выполните «добавить в модуль».

5. Для выполнения обратных операций используйте правую кнопку по соответствующим графическим элементам.

ПРАВИЛА:

а) Выходные связи их модуля возможны от его любых элементов.

б) Входные связи к модулю, старайтесь создавать только к его корню, иначе смысл модуля размывается, а в свернутом состоянии такие связи не будут отображены.

ИИ Ассистент

Продукт имеет встроенного ИИ помощника, который доступен в Отладчике, а в платных версиях и в Редакторах графов и UI.

Его способности сильно зависят от качества подгруженной в ваш проект текущей ИИ модели.

Однако он может отвечать на вопросы по справочнику и помогать с проверкой и написанием кода. А в платных и расширенных версиях, способен создавать сложные нейронные и UI решения.

Встроенный RAG ассистента

Для своей работы ассистент опирается на предоставленные ему документы.

Вы можете управлять документами через окно Отладчика путем набора специальных команд.

ВАЖНО: В конечном продукте, пользователь также может пользоваться ИИ-ассистентом, и сам загружать в него новые документы.

Добавить документ ассистенту

```
#RAG+ Название_документа Путь_к_документу
```

Пример:

```
#RAG+ SocratesDoc C:\Users\100nout\OneDrive\Dokumentumok\SocratesDoc.txt
```

Удалить документ ассистента

```
#RAG- Название_документа
```

Пример:

```
#RAG- SocratesDoc
```

Получить список известных документов

```
#RAG?
```

Настройка Ассистента

Вы можете подобрать параметры работы Ассистента в соответствии с используемой моделью и типом документов\запросам к ним.

Параметр системы: AiAssystant записываются строкой с разделителем:

релевантность|режим рассуждений|число найденных частей|длина ответа в токенах| инструкция

Режимы RAG рассуждений (при поиске и сборке ответа):

_recall - Быстро найти достаточно релевантных фрагментов.

_recalDetailed - Найти больше связанной информации и дать подробное объяснение. _recalPersistent - Использовать и сохранять контекст между запросами. _recalCareful - Искать, перепроверять, сравнивать и осторожно формировать ответ

Пример настройки:

0.6|_recal|10|2048| Ты — ассистент по специальному C#-подобному скриптовому языку и нейристорной графовой Нейросети (подробности в справочнике).

Отвечай на русском языке и достаточно кратко.

Для вопросов по Графовой Нейро Сети и для написания скриптов используй подключённую справочную документацию. Не выдумывай отсутствующие типы, свойства, методы, события и параметры.

Если в источниках недостаточно данных, прямо напиши: «В доступных документах нет достаточных данных для ответа».

При написании скриптов обязательно соблюдай спецификацию языка из справочника. Никогда не используй классы, using, namespace и явные объявления типов.

Имена идентификаторов копируй точно.

Не повторяй один и тот же пункт или строку.

После завершения ответа остановись."