Сопоставление синтаксических элементов в системах контроля версий[1]

Павленко Д. В., аспирант
Санкт-Петербургский государственный университет
dmit10@gmail.com

Описывается задача сопоставления синтаксических элементов файлов, хранящихся в системах контроля версий. Решение задачи сопоставления. позволит нагляднее описывать изменения хранимых файлов а также эффективнее решать проблему конфликтов.

1. Введение

В последние годы наблюдается массовая разработка программного обеспечения. В связи с этим актуальными являются задачи, связанные с облегчением процесса разработки для получения его большей производительности и управляемости. В качестве примера таких задач можно привести: задача отслеживания ошибок (bug tracking), задача статического анализа кода, задача резервного копирования и контроля версий программного продукта (version control) и др. В статье рассмотрим последнюю задачу поподробнее.

Задача контроля версий заключается в обеспечении надежного хранения и пересылки файлов исходного кода, их промежуточных состояний (называемых ревизиями или коммитами — терминология разнится в зависимости от используемой системы контроля версий), сравнении этих состояний и обнаружении изменений, обеспечении взаимодействия нескольких разработчиков, разграничении доступа. В настоящий момент наиболее популярными системами контроля версий (VCS) являются Subversion [1], Git [2] и Mercurial [3].

Все существующие системы контроля версий можно условно поделить на централизованные и децентрализованные. Централизованные системы (например, CVS, Subversion) характеризуются тем, что вся информация обо всех ревизиях (так называемый *репозиторий*) хранится на сервере, а на клиентской машине разработчика

[1] ©Д. В. Павленко, 2011

находится лишь "снимок" проекта определенной ревизии — так называемая *рабочая копия*. Для проведения большинства операций, в т.ч. для создания новой ревизии, клиентское приложение осуществляет запросы к серверу.

Рассмотрим подробнее систему контроля версий Subversion как наиболее популярную на сегодняшний день. Ее идея заключается в хранении снимков состояний файловой системы, называемых ревизиями. Ревизии нумеруются последовательными натуральными числами. Кроме содержимого файлов в ревизиях Subversion хранятся метаданные — так называемые свойства — произвольные пары ключ-значение, установленные на файлы или директории.

Работа пользователя с Subversion начинается со скачивания (операция checkout) последней ревизии, которая имеется на сервере. После работы с файлами пользователь создает новый снимок (операция commit) и отправляет на сервер, что происходит в процессе одной транзакции. Если, однако, от момента скачивания последней ревизии до создания снимка другой пользователь успел отправить на сервер свои изменения, то операция commit завершается с ошибкой. В этом случае пользователю необходимо снова скачать последнюю версию проекта с сервера (операция update) и снова повторить операцию commit.

В процессе операции update возможна такая ситуация, что скачиваемые изменения противоречат изменениям, сделанным локально. Например, пользователь A скачал последнюю версию проекта и локально добавил строку в файл file. После этого пользователь B скачал последнюю версию проекта, удалил из проекта file и зафиксировал это изменения операцией commit. В результате у пользователя A не последняя версия проекта, поэтому он не может зафиксировать свои локальные изменения, не выполнив операцию update. Однако при операции update окажется, что пользователи A и B выполнили противоречащие друг другу изменения к одному и тому же файлу. Такая ситуация называется конфликтом. В разобранном примере пользователю A приходится разрешать конфликт вручную, после чего он сможет выполнить commit.

Subversion предоставляет и другие подходы к осуществлению взаимодействия. Один из таких подходов — выставление так называемых блокировок (lock) на изменяемые файлы перед изменением. В этом случае никто, кроме установившего блокировку, не сможет внести изменения в заблокированный файл. Этот метод об-

ладает серьезными недостатками. Для того, чтобы снять блокировку, нужен доступ к серверу, но соединение может пропасть или установивший блокировку может забыть об этом. Хотя у блокировок есть время жизни, на этот срок изменить файл будет нельзя. Другой подход заключается в том, что каждый пользователь Subversion (или группа пользователей) создает отдельную ветку разработки. Это не решает проблему конфликтов, однако позволяет ее отсрочить, сделав ситуацию более предсказуемой. Т. е. те же самые конфликты возникнут, но не при скачивании последней версии, а при выполнении операции слияния веток разработки (merge). Для операции merge конфликты — это вполне ожидаемый побочный эффект. Последний подход в настоящее время пользуется гораздо большей популярностью, чем все остальные.

В последнее время набирают популярность так называемые децентрализованные системы контроля версий (например, Git и Mercurial). Их основное отличие заключается в том, что работа с ними начинается с полного клонирования всей истории проекта на клиентскую машину, в результате чего у пользователя локально появляется полноценный, полнофункциональный репозиторий, таким образом, все операции производятся локально. Популярность таких систем обеспечивается преимуществами:

- более высокая скорость операций;
- возможность создавать коммиты/ревизии вне зависимости от наличия соединения с сервером;
- отсутствие необходимости в системе резервного копирования, поскольку репозиторий может быть восстановлен из копии у любого из пользователей.

Кроме того, в этих системах, как правило, упрощены процессы создания и слияния ветвей разработки. Тем более актуальной остается проблема их слияния.

Ситуация, при которой образуются конфликты, очень часто встречается. Причиной тому: появление все более мощных инструментов разработки, возможности которых вышли далеко за пределы текстовых редакторов, в результате чего существенные изменения файлов производятся просто комбинацией клавиш (переформатирование кода, рефакторинги), иногда даже неявно (генерации, оптимизация и переупорядочивание import-деклараций в Java-средах).

Достаточно часто слияние может быть выполнено системами контроля версий автоматически (см рис. 1) — в тех случаях, когда изменения не пересекаются. В случаях, когда изменениям подвергаются одни и те же или соседние строки одного и того же файла, только человек может решить, какое из изменений выбрать.

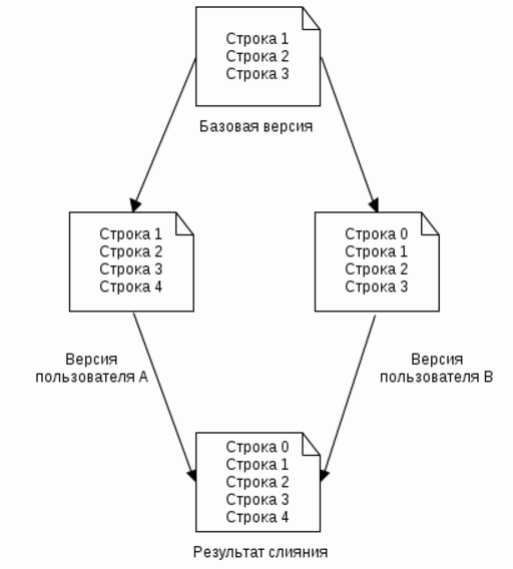

Рис. 1: Автоматическое слияние изменений файла.

Проведение слияния вручную — достаточно трудоемкая операция. Она производится вручную на основе трех состояний проекта: базовой версии (версии до изменений), версии после изменения первым пользователем и версии после изменения вторым пользователем.

Существующие системы контроля версий обычно оперируют строками, никак не пытаясь использовать возможную структуру содержимого файлов. Такой подход позволяет работать с текстовыми файлами любого содержания. Однако, зачастую, вследствие неиспользования информации о семантике сливаемых данных, возникают конфликты, которые могли бы теоретически также разрешаться автоматически. Далее рассмотрим такие случае подробнее.

В индустрии разработки программного обеспечения в последнее

время пользуется популярностью преобразование исходного кода, называемое *рефакторинг*. Рефакторинг — это преобразование кода программы, не меняющее ее функциональность. На сегодняшний день существуют несколько десятков видов рефакторингов.

Наиболее часто используемый рефакторинг — это переименование. В результате его выполнения переименовываются все вхождении имени переменной/класса/метода. Кроме того, переименование класса может спровоцировать переименование файла, в котором класс объявлен. Современные среды разработки проводят эту операцию автоматически, пользователю необходимо только указать новое имя переименовываемого синтаксического элемента. Таким образом, в результате лишь небольшого усилия со стороны разработчика могут быть изменены десятки строк исходного кода. Поэтому достаточно вероятна ситуация, что в результате автоматического слияния произойдут конфликты.

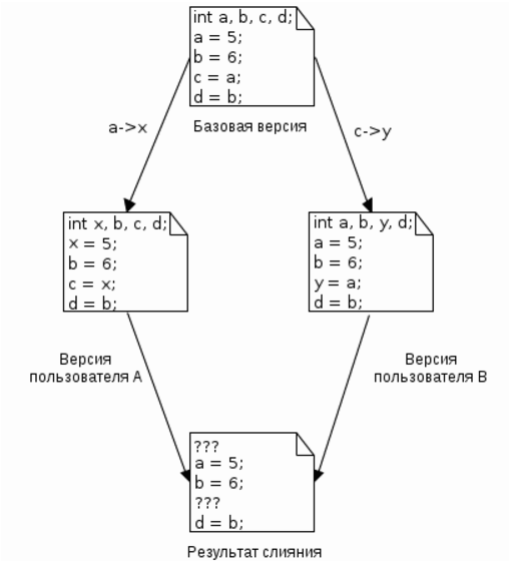

Рис. 2: Современные системы контроля версий не в состоянии провести автоматическое слияние изменений файла.

На рис. 2 продемонстрирована ситуация, при которой переименование переменной приводит к конфликту. Пользователь A пере-

именовал переменную a в x в своей ветви, пользователь B — переименовал c в y в своей ветви. В результате слияния образуются два конфликта, поскольку современные системы контроля версий на настоящий момент не позволяют использовать структуру содержимого файла. В то же время интуитивно понятно, что такое слияние вполне безопасно можно обработать автоматически и результатом такого слияния является файл следующего содержания:

```
int x, b, y, d;
x = 5;
b = 6;
x = y;
d = b;
```

Схожие проблемы вызывают и практически все другие виды рефакторингов, а также операция переформатирования кода (изменение отступов и расстановки символов перевода строки).

Кроме того, в некоторых языках программирования часто не имеет значения порядок некоторых синтаксических элементов. Так, например, в языке Java не важен порядок методов и import-деклараций. Современные системы разработки программного обеспечения также позволяют существенно менять этот порядок без значительных усилий со стороны пользователя. На практике обычно различный порядок import-деклараций является самой распространенной причиной конфликтов в Java-проектах.

Главной причиной того, что современные системы контроля версий не могут разрешать конфликты такого типа, можно назвать тот факт, что все существующие системы обрабатывают содержимое файлов на уровне строк, не используя структуру и семантику, хранящейся в них информации.

Кроме задачи слияния в системах, контроля версий очень важна задача вычисления изменений. Поскольку современные системы контроля версий рассматривают содержимое файлов на уровне строк, в некоторых случаях изменения могут быть определены некорректно. На рис. 3 показан типичный результат вычисления изменений, который демонстрирует недостатки подхода существующих систем контроля версий, не принимающих во внимание синтаксическую структуру файлов. На практике, вычисление изменений таким способом требует больше времени на визуальный анализ человеком, чтобы понять смысл изменений.

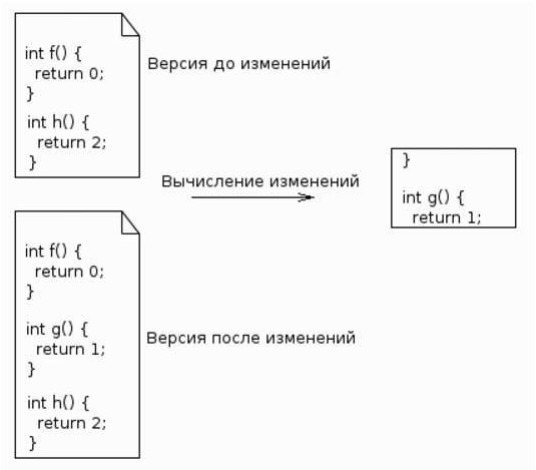

Рис. 3: Вычисление изменений содержимого без учета его синтаксической структуры.

В статье предлагается иной подход к решению задачи слияния для тех случаев, когда известно, что содержимое файлов представляет из себя текст программы на каком-либо языке программирования.

Для представления кода на каком-либо языке программирования часто используется структура, называемая *абстрактным синтаксическим деревом*. Это дерево из синтаксических элементов, построенное по некоторому набору заранее определенных для данного языка правил. Это дерево не включает в себя информацию о форматировании кода. Узлами дерева являются синтаксические элементы, а корнем — единица компиляции (как правило, файл, иногда проект целиком).

Чтобы обнаруживать изменения между двумя состояниями проекта с учетом семантики содержимого, вместо версионирования файлов как наборов строк, будем версионировать абстрактные синтаксические деревья и определять соответствия на уровне узлов этих деревьев.

2. Постановка задачи

Таким образом, формально основная задача может быть записана следующим образом. Даны два дерева T_1 и T_2. Каждую вершину v дерева T_2 нужно пометить меткой u из множества вершин дерева T_1 или особой меткой Λ таким образом, чтобы минимизировать некоторый функционал от такой разметки т.е. $F(u_{v_1}, ..., u_{v_n})$, где u_{v_i} — метка, присвоенная вершине v_i дерева T_2.

Вопрос выбора функционала F непростой и зависит от семантических особенностей сравниваемого содержимого. Однако для простоты воспользуемся моделью случайного марковского поля хорошо зарекомендовавшей себя в подходах к решению задачи сопоставления [4].

Случайное марковское поле — это графическая модель, в которой каждая переменная соответствует вершине некоторого графа, а условная вероятность принятия каждой переменной той или иной величины при известной совокупности значений всех прочих переменных равна условной вероятности принятия ею этого значения при известных значениях переменных, соответствующим соседним вершинам [5]. Иными словами значение каждой переменной зависит только от ее соседей. Формально это записывается так:

$$p(v = u|V \setminus v) = p(v = u|N(v)),$$

где V — множество всех переменных, $N(v)$ — множество переменных, соответствующих соседним вершинам вершины v. Это свойство называется свойством локальной марковости.

По теореме Хаммерсли-Клиффорда [6] локальная марковость равносильна тому, что совместная вероятность распределена по Гиббсу, то есть

$$p(u_1, ..., u_n) = \frac{1}{C} \prod_{v_i \in V} p_{v_i}(u_i) \prod_{(v_i, v_j) \in N} p_{v_i v_j}(u_i, u_j),$$

где C — некоторая константа, обеспечивающая

$$\sum_{u_1, ..., u_n} p(u_1, ..., u_n) = 1,$$

$p_{v_i}(u_i)$ и $p_{v_i v_j}(u_i, u_j)$ — некоторые функции, называемые потенциальными, N — множество пар соседних вершин.

Обычно в модели случайного марковского поля ставится задача максимизации вероятности $p(u_1, ..., u_n)$. В этом случае обычно рассматривают не саму вероятность, а логарифм от нее, взятый с противоположным знаком, и задача сводится к минимизации

$$F(u_1, ..., u_n) = -\ln\ p(u_1, ..., u_n) = .$$

$$= -C + \sum_{v_i \in V} f_{v_i}(u_i) + \sum_{(v_i, v_j) \in N} f_{v_i v_j}(u_i, u_j),$$

где

$$f_{v_i}(u_i) = -\ln\ p_{v_i}(u_i),$$

$$f_{v_i, v_j}(u_i, u_j) = -\ln\ p_{v_i v_j}(u_i, u_j).$$

Константу, не влияющую на оптимизацию, часто опускают и просто пишут

$$F(u_1, ..., u_n) = \sum_{v_i \in V} f_{v_i}(u_i) + \sum_{(v_i, v_j) \in N} f_{v_i v_j}(u_i, u_j).$$

Неформально в этой формуле функция $f_{v_i}(u_i)$ имеет смысл штрафа за неподходящее значение u_i переменной v_i. Функция $f_{v_i v_j}(u_i, u_j)$ имеет смысл штрафа за негладкость.

Применительно к задаче сопоставления синтаксических элементов переменные приобретают такой смысл: V — множество синтаксических элементов дерева T_2, U — объединение множества синтаксических элементов дерева T_1 с $\{\Lambda\}$; $f_{v_i}(u_i)$ — стоимость сопоставления элементу v_i дерева T_2 элемента u_i дерева T_1, если $u_i \neq \Lambda$, или стоимость несопоставления v_i ни одного элемента из T_1, если $u_i = \Lambda$; $f_{v_i v_j}(u_i, u_j)$ — штраф за сопоставление соседним элементам v_i и v_j дерева T_2 несоседних элементов дерева u_i и u_j T_1 (в случае $u_i = \Lambda$ или $u_j = \Lambda$ — просто некоторая константа). И тогда наилучшее сопоставление ищется как минимум функционала $F(u_1, ..., u_n)$.

Модель случайного марковского поля не накладывает ограничений на выбор функций $f_{v_i}(u_i)$ и $f_{v_i u_i}(u_i, u_j)$. Разумно было бы для каждого типа синтаксического элемента v_i (например, “объявление класса” или “вызов метода”) использовать свою функцию в зависимости от информации, содержащейся в том или ином синтаксическом элементе. Например, в синтаксическом элементе “объявление метода” содержится информация об имени метода, помимо этого

имеется возможность определить и другие его параметры, например, принадлежность к подмножеству методов называемых в программировании геттерами (getters). Например, вероятность того, что синтаксический элемент "доступ к полю класса" дерева T_1 соответствует в "вызову геттера" дерева T_2 выше, чем просто "вызову метода", поскольку современные системы разработки программного обеспечения предоставляют инструменты автоматической замены поля на геттеры. Таким образом, возникает еще одна задача — выбор функций $f_{v_i}(u_i)$ и $f_{v_i u_i}(u_i, u_j)$.

Для решения этой задачи можно выбрать некоторую параметрическую модель от набора параметров θ_i специфичных для данного синтаксического элемента v_i. И функции $f_{v_i}(u_i)$ и $f_{v_i u_i}(u_i, u_j)$ запишутся в виде:

$$f_{v_i}(u_i) = \hat{f}_{v_i}(u_i, \theta_i)$$

и

$$f_{v_i u_i}(u_i, u_j) = \hat{f}_{v_i u_i}(u_i, u_j, \theta_{ij}).$$

Примером такой параметрических модели являются нейронные сети, где подстраиваемый это набор всех весов нейронной сети.

В этом случае оптимизируемый функционал будет зависеть от θ — набора всех θ_i и θ_{ij}, записанных одним вектором:

$$\hat{F}(u_1, ..., u_n, \theta) = \sum_{v_i \in V} \hat{f}_{v_i}(u_i, \theta_i) + \sum_{(v_i, v_j) \in N} \hat{f}_{v_i v_j}(u_i, u_j, \theta_{ij}).$$

Если обозначить

$$\tilde{u}(\theta) = \arg\min_{(u_1, ..., u_n)} \hat{F}(u_1, ..., u_n, \theta),$$

то для данной конкретной задачи сопоставления оптимальные параметры выглядят можно вычислить как

$$\theta^* = \arg\min_{\theta} |\tilde{u}(\theta) - u^*|^2,$$

где u^* — правильное сопоставление, отмеченное вручную. Обычно в задачах подстройки параметров выбирают используют не один пример, а несколько. А оптимальные параметры вычисляют как минимум:

$$\theta^* = \arg\min_{\theta} |\tilde{u}^{(1)}(\theta) - u^{*(1)}|^2 + ... + |\tilde{u}^{(k)}(\theta) - u^{*(k)}|^2,$$

где $u^{*(m)}$ — правильное размеченное вручную решение m-ой задачи сопоставления, $\tilde{u}^{(m)}(\theta)$ — решение m-ой задачи при функциях $\hat{f}_{v_i}(u_i, \theta_i)$ и $\hat{f}_{v_i u_i}(u_i, u_j, \theta_{ij})$, параметризованных θ.

3. Методы решения

Задача оптимизации в случайном марковском поле обычно решается с помощью алгоритма распространения свидетельства [7]. Алгоритм основан на понятии *сообщения*. Сообщением $m_{ji}(u_i)$ от вершины v_j вершине v_i называется вектор, размерность которого равна количеству элементов множества всевозможных значений переменной v_j, а каждая компонента — некоторое число. Смысл этого числа можно определить неформально как оценка того, насколько вероятно, что вершина v_j имеет соответствующее значение.

Алгоритм распространения свидетельства состоит из этапов.

- Положить $m_{ji}(u_i)$ равной для всех значений u_i.
- Выполнить обновление сообщений до сходимости по формуле:

$$m_{j \to i}(u_j) \leftarrow \min_{u_i}(f_{v_i}(u_i) + f_{v_i v_j}(u_i, u_j) + \sum_{v_k \in N(v_j) \setminus v_i} m_{k \to j}(u_k))$$

- Вычислить оптимальные значения u_i^* по формуле:

$$u_i^* = \arg\min_{u_i}(f_{v_i}(u_i) + \sum_{v_j \in N(v_i)} m_{j \to i}(u_j)).$$

Таким образом, каждому синтаксическому элементу v_i дерева T_2 сопоставляется синтаксический элемент u_i^* дерева T_1 или символ Λ, т. е. ни одного элемента.

Для вычисления оптимальных параметров θ штрафных функций необходимо минимизировать функцию $\tilde{u}(\theta)$. Для одного вычисления этой функции в любой точке $\hat{\theta}$ необходимо решить задачу сопоставления со штрафными функциями параметризованными $\hat{\theta}$. Таким образом, при использовании алгоритма градиентного спуска для выполнения одной итерации алгоритма необходимо решить задачу оптимизации случайного марковского поля $2n$ раз, где n —

размерность θ, если использовать аппроксимацию градиента конечной разностью:

$$\nabla\tilde{u}(\theta) \approx \frac{1}{2}\begin{pmatrix}\tilde{u}(\theta+e_1)-\tilde{u}(\theta-e_1)\\ \cdots \\ \tilde{u}(\theta+e_n)-\tilde{u}(\theta-e_n)\end{pmatrix},$$

где e_l — единичные векторы.

Алгоритм градиентного спуска записывается в виде:

$$\hat{\theta}_m = \hat{\theta}_{m-1} - \alpha_m \nabla\tilde{u}(\hat{\theta}_{m-1}).$$

Поскольку каждое измерение $\tilde{u}(\theta)$ — относительно долгий процесс, разумно было бы вместо классического градиентного алгоритма использовать алгоритм, позволяющий производить меньшее количество измерений для выполнения итерации.

Например, таким свойством обладает рандомизированный алгоритм стохастической аппроксимации с пробным одновременным возмущением на входе [8, 9], который выглядит следующим образом:

$$\hat{\theta}_m = \hat{\theta}_{m-1} - \frac{\alpha_m}{2\beta_m}(\tilde{u}(\hat{\theta}_{m-1}+\beta_m\Delta_m) - \tilde{u}(\hat{\theta}_{m-1}-\beta_m\Delta_m))\Delta_m,$$

где α_m, β_m — некоторые последовательности, Δ_m — случайный вектор из $+1$ и -1.

Из формулы следует, что одна итерация алгоритма требует только два измерения $\tilde{u}(\theta)$, что существенно поскольку ее измерение является достаточно дорогим.

4. Пример

Для примера работы алгоритма рассмотрим простой искусственный язык, в котором файл представляет из себя набор объявлений функций. Каждая функция состоит из списка конструкций трех видов: объявление переменной, присваивания переменной выражения и возвращение значения. Для простоты, допустимыми выражениями являются переменная или сумма переменных.

Грамматика такого языка записывается в виде:

```
start: functionDefinition*;
variableAccess: ID;
```

```
constant: NUMBER;
atom: variableAccess | constant;
expression: atom | atom PLUS atom;
leftPart: ID;
functionName: ID;
arg: ID;
functionArguments: | arg (',' arg)*;
functionDefinition: functionName
  '(' functionArguments ')' '{' functionBody '}';
definitionStatement: DEF ID;
returnStatement: RETURN expression;
assignmentStatement: leftPart EQ expression;
statement: definitionStatement |
  assignmentStatement | returnStatement;
functionBody: (statement ';')*;
RETURN: 'return';
DEF: 'def';
PLUS: '+';
EQ: '=';
NUMBER: (DIGIT)+ ;
LETTER_OR_DIGIT: LETTER | DIGIT;
ID: LETTER (LETTER_OR_DIGIT)*;
WHITESPACE: ( '\t' | ' ' | '\r' | '\n' )+;
LETTER: 'a'..'z' | 'A'..'Z';
DIGIT: '0'..'9'.
```

Типичная программа на тестовом языке выглядит следующим образом:

```
sum(a, b) {
def c;
c = a + b;
return c;
}

main(k) {
def l;
l = k + k;
k = l;
return k;
```

```
}
```

Для демонстрации решения задачи сопоставления допустим, что содержимое файла изменилось на:

```
main(k) {
def l;
k = l;
return k;
}

sub(c, d) {
def e;
e = c + d;
return e;
}
```

Система контроля версий Git выдает следующий результат сравнения файлов.

```
-sum(a, b) {
-def c;
-c = a + b;
-return c;
-}
-
 main(k) {
 def l;
-l = k + k;
 k = l;
 return k;
 }

+sub(c, d) {
+def e;
+e = c + d;
+return e;
+}
```

Этот результат можно интерпретировать так: удалена функция “sum”, добавилась совершенно другая функция “sub”, и в функции “main” удалена строка.

На самом же деле в функции “main” удалена строка, но функция “sum” переименована в “sub”, а ее переменные переименованы соответственно “a” в “c”, “b” в “d”, “c” в “e”, а кроме того, функции переставлены местами. Это достаточно сложный пример сопоставления, поскольку переменная “c” присутствует, как в старой, так и новой функции, но имеет в них разный смысл.

Для сопоставления использовалась модель случайного марковского поля, построенного следующим образом: каждый узел абстрактного синтаксического дерева соответствует переменной случайного марковского поля; соседними вершинами в марковском поле являются родительский узел, дочерние узлы и так называемые узлы-братья (siblings), если порядок элементов этого типа важен для сопоставления. Наример, у узла, соответствующего выражению “e = c + d;” соседями являются родительский элемент (“sub”), дочерние (“e” и “c+d”) а также ближайшие братья (“def e;” и “return e;”). В нашем искусственном языке порядок функций не важен, поэтому “main” и “sub” соседними элементами случайного марковского поля не являются.

Штрафные функции определим так:

$$\begin{cases} 11, & u_i = \Lambda \\ 1000, & type(v_i) \neq type(u_i) \\ 0.3|pos(u_i) - pos(v_i)| & type(v_i) = type(u_i), val(v_i) = val(u_i) \\ 0.3|pos(u_i) - pos(v_i)| + 15.5 & type(v_i) = type(u_i), val(v_i) \neq val(u_i) \end{cases}$$

где $type$ — тип узла абстрактного синтаксического дерева (“функция”, “оператор присваивания” и т.д.), val — его строковое содержимое, например, (“sum”, “e”, “c”). Для узлов дерева, являющихся контейнерами, не содержащими никакого строкового значения (например, “e = c + d;”) определим val как пустую строку; pos — номер узла в среди потомков родительского узла, если таковой имеется, начиная с нуля, деленный на количество таких узлов; если родительского узла нет, то 0. Например, $pos(\text{“}e = c + d;\text{”}) = \frac{1}{3}$.

$$f_{v_i,v_j}(u_i, u_j) = \begin{cases} 4.7, & u_i = \Lambda || u_j = \Lambda, \\ 0, & u_i \in N(u_j), \\ 15, & u_i \notin N(u_j). \end{cases}$$

Первая штрафная функция оценивает несоответствие. Если сопоставляемые элементы разных типов, чего не может быть в нашем

языке (хотя может встречаться в языках типа Java), стоимость сопоставления большая. Если элемент ничему не сопоставлен, то штраф больше идеального сопоставления

$$type(v_i) = type(u_i), val(v_i) \neq val(u_i),$$

но меньше неверного сопставления (т.е. лучше ничего не сопоставить, чем сопоставить неверный элемент). Небольшая доля штрафа вносимая порядком элементов позволяет при прочих равных вариантах сопоставления отдать предпочтение вариантам с правильным порядком.

Вторая штрафная функция оценивает негладкость. Опять же, штраф за несопоставление – нечто среднее между идеальным сопоставлением (соседним элементам сопоставлены соседние) и неверным (соседним сопоставлены несоседние).

С использованием данных функций алгоритм распространения свидетельства дает верный результат сопоставления за две итерации для данного примера, несмотря на то, что в модели случайного марковского поля переименование каждого вхождения переменной не зависит от переименования другого вхождения той же переменной, хотя в реальной жизни это зависимые события.

Константы, фигурирующие в формулах получены достаточно произвольным образом. Алгоритм достаточно нечувствителен к небольшим изменениям параметров, составляющих штрафные функции, однако при их варьировании можно добиться желаемого поведения в зависимости от желаемых приоритетов. Например, сделав штраф за сопоставление функций нашего языка с разными именами более высоким, можно добиться запрета определения факта переименования.

5. Заключение

Таким образом, с помощью алгоритма распространения свидетельства может быть решена задача сопоставления синтаксических элементов. С помощью алгоритма стохастической оптимизации с пробным одновременным возмущением на входе можно подстроить параметры случайного марковского поля с тем, чтобы улучшить качество сопоставления для данного языка на основе примеров программ на рассматриваемом языке.

Сопоставление на уровне синтаксических элементов по сравнению с сопоставлением на уровне строк позволяет формировать описание изменение более удобочитаемым образом, ибо в этом случае учитывается семантика обрабатываемых данных. Более того, использование представления в виде абстрактного синтаксического дерева позволяет заранее отсеять неправдоподобные описания изменеий исходного кода, тем самым повышая вероятность правильного сопоставления.

Список литературы

[1] http://en.wikipedia.org/wiki/Subversion.

[2] http://en.wikipedia.org/wiki/Git.

[3] http://en.wikipedia.org/wiki/Mercurial.

[4] *Jian S., Heung-Yeung S., Nan-Ning Zh.* Stereo matching using belief propagation // IEEE Trans. on Pattern Analysis and Machine Intelligence. 2003. Vol. 25. P. 787–800.

[5] *Boykov Y., Veksler O., Zabih R.* Markov random fields with efficient approximations // In IEEE conf. on Computer Vision and Pattern Recognition (CVPR). 1998. P. 648–655.

[6] *Samson Ch.* Proof of Hammersley-Clifford Theorem. 2008.

[7] *Yedidia J.S., Freeman W.T., Weiss Y.* Understanding belief propagation and its generalizations // Exploring Artificial Intelligence in the New Millennium. Science & Technology Books. 2003. Chap. 8. P. 239–236.

[8] *Граничин О.Н.* Рандомизированные алгоритмы стохастической аппроксимации при произвольных помехах // Автоматика и телемеханика. 2002. № 2. С. 44–55.

[9] *Граничин О.Н.* Оптимальная скорость сходимости рандомизированных алгоритмов стохастической аппроксимации при произвольных помехах // Автоматика и телемеханика. 2003. № 2. С. 88–99.