LR(1)-анализатор языков, заданных контекстно-свободными грамматиками

Опишем в вольном стиле части парсера

1. Грамматика

Все начинается с задания грамматики. В парсере поддержка грамматики реализована в виде набора классов, главный из которых - Grammar, содержит `vector<Rule>` из правил. Само правило - `Token` левой части (`prefix`) и `vector<Token> suffix` правой части. Класс `Token` хранит id токена, т.е. хеш строки, заданной по определенным правилам (нетерминал - строка капсом, терминал - иначе). При этом если `id < Token::border`, то это нетерминал, иначе - терминал. От класса наследуется `LRGrammar` - расширение грамматики для LR, в котором еще дополнительно реализован `First` (о нем далее) и метод его построения. Я решил вынести First, т.к. по сути он не связан ни с каким из состояний, создается только на этапе предобработки и нужен для построения главных частей парсера LR, а не для парсинга. Заметим, что использование хешей под токены позволяет экономить память и время работы (O(1) на сравнения).

2. LRParser

Сам класс и некоторый набор других классов имплементирует логику построения и работы анализатора. Интерфейс, вынесенный в public - два метода: `fit(Grammar g)` - предобработка грамматики и построение таблиц и `predict(std::string word)` - метод, запускающий парсинг слова.

Сам `LR(k), k=1` парсер представляет из себя модифицированную версию алгоритма *перенос-свертка*, позволяющую рассматривать более обширный класс грамматик за счет *предпросмотра* в процессе парсинга следующего за текущим символом символа. Этот трюк позволяет избежать `reduce-shift` конфликтов в некоторых не `LR(0)` грамматиках. Чем больше k, тем больший класс грамматик алгоритм сможет обработать без конфликтов.

- Предобработка: построение First

  Сначала строится `First` по грамматике, поданной на вход, неформальное определение этой абстракции - первые k=1 символов, которые можно вывести из нетерминала во всевозможных правилах. Реализован через `unordered_map` `unordered_set`-ов для амортизированно-быстрого доступа к нужным токенам (заточен скорее под грамматики большого размера, для малых можно было ограничиться и аналогами на деревьях). `First` строится путем раскрытия первого токена правой части правила, если это нетерминал. Так раскрываем до тех пор, пока множество First не перестанет меняться (транзитивное замыкание - позволяет корректно обрабатывать рекурсивные случаи объявления правил грамматики). Асимптотика построения: для каждого нетерминала добавим не больше `term_cnt` (количество терминалов в грамматике) токенов, поэтому грубая оценка: $O(|P| \cdot term\_cnt)$, где $|P|$ - количество правил в грамматике.

Набор таблиц парсера - вектор пар `<State, automata_item_type>`, хранит как сами состояния, так и переходы по токенам из i-го состояния в id других состояний, тем самым это как-бы неявный автомат.

- Предобработка: построение таблиц (a.k.a ДКА над множествами ситуаций)

  Таблица, она же `State` - класс, который хранит сет `Item`-ов, а `Item` содержит `Grammar::Rule` правило и сет `lookahead` - предпросмотр для состояния.

1. В начале о том, как добавляется конкретная новая таблица. Таблица может быть создана из `kernel` - начальный набор ситуаций, полученный либо на этапе инициализации построения ($S' \to \cdot S$), либо с переходом из предыдущей таблицы. В любом случае, нужно *замкнуть* `kernel` через `State::closure(&grammar)`, это делается путем создания и добавления новых ситуаций (с левой частью `A`, правой - всеми правыми частями правил `G[A]`, где `G[A]` - все правила грамматики с левой частью A), если в `kernel` точка стоит перед нетерминалом `A` с соответствующими `lookahead`. При построении используется все тот же принцип транизитивного замыкания, но реализован эффективнее, в виде очереди с отслеживанием дубликатов.

Асимптотика замыкания одного состояния: количество ситуаций зависит от ядра, могут добавиться ситуации с одинаковыми правилами, но разными позициями точек + разными $lookahead$, по которым идет итерация при добавлении ситуаций, поэтому - $O(|P| \cdot |\Sigma| \cdot \Gamma)$, где Γ - суммарная длина правых частей всех правил. Это же оценка на количество ситуаций в одном состоянии.

2. Новые таблицы возникают при построении парсера в функции `buildAutomata`, которая рекурсивно строит новые состояния и связывает переходами по токенам. Кратко опишем суть:

- получаем на вход id текущего состояния, составляем ядра по всем возможным переходам (тем токенам, перед которыми стоят точки в ситуациях состояния), запоминаем токены-переходы.
- создаем новое состояния из ядер, замыкаем их (дубликаты отслеживаются). Связываем их переходами по токенам-переходам.
- рекурсивно идем в созданные состояния

Асимптотика: каждое состояние может породить какое-то ядро, т.е. набор ситуаций, а их не более Γ, $states\_cnt = n = \Gamma$. Из каждого состояния могут идти ребра в другие состояния по $\Sigma + N$ токенам. Не забываем про время построения замыкания. В оценке не учитывается то, что построение рекурсивное. Итого: $O(n^2 \cdot (\Sigma + N) \cdot closure\_time)$.

3. Последний шаг - построение action таблицы - вектор $unordered\_map$-ов, который для каждого id состояния для каждого токена-перехода говорит, что парсеру делать - `reduce(i), shift(j), acc, reject`, i - номер правила, j - номер состояния. Асимптотика: $O(|P| \cdot |\Sigma| \cdot \Gamma \cdot (\Sigma + N) \cdot \Sigma)$, что видно из функции `build_action()`.

- Парсинг: `parse(word)`

  Заводится стек, в котором хранится путь, по которому мы шли во время работы - чередующиеся токены и номера состояний в автомате. Каждая итерация - это либо reduce, либо shift согласно таблице action. Выбор происходит однозначно, если грамматика является `LR(1)` грамматикой и за $O(1)$ времени, поэтому все работает за $O(|word|)$.

Как видим, предобработку можно загрубить до $O(n^2 \cdot (\Sigma + N) \cdot closure\_time)$, но волноваться не стоит, так как это предобработка! Сам парсинг работает за линию, и это замечательно.