Санкт-Петербургский государственный университет
информационных технологий, механики и оптики

Кафедра компьютерных технологий

И. Р. Ахметов

Разработка визуализатора алгоритма Укконена построения суффиксных деревьев на основе технологии *Vizi*

Программирование с явным выделением состояний

Проектная документация

Санкт-Петербург

2005

Содержание

Введение 4

1. Анализ литературы 5

2. Описание алгоритма 5
   2.1. Неявные суффиксные деревья 5
   2.2. Общее описание алгоритма Укконена 6
      2.2.1. Общее описание алгоритма Укконена 6
      2.2.2. Правила продолжения суффиксов 6
   2.3. Реализация и ускорение 7
      2.3.1. Суффиксные связи: первое ускорение реализации 7
      2.3.2. Переходы по суффиксным связям при построении T_{i+1} 8
      2.3.3. Алгоритм отдельного продолжения 8

3. Преобразованная программа 10

4. Описание интерфейса визуализатора 10

5. Описание конфигурации визуализатора 12

Заключение. 13

Приложение 1. XML-описание визуализатора 15
   Ukkonen.xml. 15
   Ukkonen-Algorithm.xml. 15
   Ukkonen-Configuration.xml 27

Приложение 2. Сгенерированные исходные коды визуализатора 30

Приложение 3. Исходные коды визуализатора 49
   ru.ifmo.vizi.ukkonen.UkkonenVisualizer 49
   ru.ifmo.vizi.ukkonen.suffixtree.Edge 55
   ru.ifmo.vizi.ukkonen.suffixtree.NodeShape. 60
   ru.ifmo.vizi.ukkonen.suffixtree.Options 62
   ru.ifmo.vizi.ukkonen.suffixtree.SuffixLink 62
   ru.ifmo.vizi.ukkonen.suffixtree.SuffixTree 64
   ru.ifmo.vizi.ukkonen.suffixtree.SuffixTreeData 71
   ru.ifmo.vizi.ukkonen.suffixtree.SuffixTreeLayout 76
   ru.ifmo.vizi.ukkonen.suffixtree.SuffixTreeNode 78
   ru.ifmo.vizi.ukkonen.suffixtree.SuffixTreeState 84
   ru.ifmo.vizi.ukkonen.ui.HintedCheckbox 90
   ru.ifmo.vizi.ukkonen.ui.HintedChoice 91

ru.ifmo.vizi.ukkonen.ui.HintedTextField 92
ru.ifmo.vizi.ukkonen.widgets.StringVisualizer 93
ru.ifmo.vizi.ukkonen.widgets.VerticalStringVisualizer 98

Введение

Суффиксное дерево — это структура данных, позволяющая представить строку в виде, удобном в ряде задач при ее последующей обработке. В частности, эта структура данных в настоящее время очень широко используется в вычислительной биологии.

Описывая, как строить суффиксное дерево для произвольной строки, будем работать с общей строкой S длины m. Суффиксным деревом T для m-символьной строки S называется ориентированное дерево с корнем, имеющее ровно m листьев, занумерованных от 1 до m. Каждая внутренняя вершина, отличная от корня, имеет не меньше двух детей, а каждая дуга помечена непустой подстрокой строки S (дуговой меткой). Никакие две дуги, выходящие из одной и той же вершины, не могут иметь пометок, начинающихся с одного и того же символа. Главная особенность суффиксного дерева заключается в том, что для каждого листа i конкатенация меток дуг на пути от корня к листу i в точности составляет суффикс строки S, который начинается в позиции i (то есть этот путь произносит $S[i..m]$). Однако если у строки один суффикс является префиксов другого, то для нее не существует суффиксного дерева, удовлетворяющего этому определению. Во избежание этой трудности предположим, что последний символ S нигде больше в строку не входит. Для того, чтобы обеспечить это на практике, мы можем добавить в конец S какой-либо символ, не входящий в основной алфавит. В качестве такого завершающего символа мы будем использовать $\$$.

Алгоритм непосредственного построения суффиксного дерева для строки S может быть сформулирован следующим образом. Поместим сначала в дерево простую дугу для суффикса $S[1..m]\$$ (для всей строки), а затем будем последовательно вставлять в растущее дерево суффиксы $S[i..m]$ для i от 2 до m. В обычном предположении ограниченного алфавита этот наивный метод строит суффиксное дерево для строки S длины m за время $O(m^2)$.

Предметом рассмотрения настоящей работы является алгоритм построения суффиксного дерева за линейное от длины входа время, который предложил Эско Укконен в [1]. Главное достоинство алгоритма Укконена — в простоте его описания, доказательства и анализа быстродействия. Это связано с тем, что алгоритм можно представить как простой, но не эффективный метод, который, однако, за несколько шагов может быть улучшен так, что будет обладать линейной оценкой сложности.

Для наглядности рассмотрим промежуточный вариант, работающий за время $O(m^2)$. Реализация выполнена при помощи технологии *Vizi*, основанной на использовании конечных автоматов, и представляет собой Java-апплет, пошагово визуализирующий работу алгоритма на разных входных данных.

1. Анализ литературы

Первый алгоритм для конструирования суффиксных деревьев за линейное время был предложен Вайнером [2] в 1973 г. Другой алгоритм, более экономный по памяти, был предложен Мак-Крейгом [3] несколько лет спустя. В 1995 г. Укконен [1] разработал принципиально иной алгоритм построения суффиксных деревьев, который в оптимальной реализации работает за линейное время и обладает всеми преимуществами алгоритма Мак-Крейга, но допускает более простое истолкование.

В книге [4] приведена общая информация по строковым алгоритмам, суффиксным деревьям и их многочисленным применениям в вычислительной биологии, а также подробно описаны алгоритмы Укконена и Вайнера.

2. Описание алгоритма

2.1. Неявные суффиксные деревья

Алгоритм Укконена строит последовательность *неявных* суффиксных деревьев, последнее из которых преобразуется в настоящее суффиксное дерево строки S.

> **Определение.** *Неявное суффиксное дерево* строки S — это дерево, которое получено из суффиксного дерева строки $S\$$ удалением всех вхождений терминального символа \$ из меток дуг дерева, удалением после этого дуг без меток, а также удалением после этого вершин, имеющих меньше двух детей.

Неявное суффиксное дерево префикса $S[1..i]$ строки S получается аналогично из суффиксного дерева для $S[1..i]\$$ удалением символов \$, дуг и вершин, как описано выше.

> **Определение.** Обозначим через T_i неявное суффиксное дерево строки $S[1..i]$, где i изменяется от 1 до m.

Хотя неявное суффиксное дерево может иметь листья не для всех суффиксов, в нем закодированы все суффиксы S — каждый «произносится» символами какого-либо пути от корня этого неявного суффиксного дерева. Однако, если этот путь не кончается листом, то не будет маркера, обозначающего конец пути. Таким образом, неявные суффиксные деревья сами по себе несколько менее информативны, чем обычные суффиксные деревья. Будем использовать неявные деревья указанного типа как вспомогательные в алгоритме Укконена для того, чтобы получить настоящее суффиксное дерево для S.

2.2. Общее описание алгоритма Укконена

Алгоритм Укконена строит неявное суффиксное дерево T_i для каждого префикса $S[1..i]$ строки S, начиная с T_1 и увеличивая i на единицу до тех пор, пока не будет построено дерево T_m. Суффиксное дерево для S получается из T_m, и вся работа требует времени $O(m)$.

2.2.1. Общее описание алгоритма Укконена

Алгоритм Укконена делится на *m фаз*. В фазе $i+1$ дерево T_{i+1} строится из T_i. Каждая фаза $i+1$ в свою очередь делится на $i+1$ *продолжений*, по одному на каждый из $i+1$ суффиксов $S[1..i+1]$. В *продолжении* с номером j фазы $i+1$ алгоритм сначала находит конец пути из корня, помеченного подстрокой $S[j..i]$. Затем он продолжает эту подстроку, добавляя к ее концу символ $S(i+1)$, если только $S(i+1)$ еще не существует. Таким образом, в фазе $i+1$ сначала помещается в дерево строка $S[1..i+1]$, а за ней строки $S[2..i+1]$, $S[3..i+1]$, ... (соответственно, в продолжениях 1, 2, 3, ...). Продолжение $i+1$ фазы $i+1$ продолжает *пустой* суффикс строки $S[1..i]$ — включает в дерево строку из одного символа $S(i+1)$ (если только ее там не было). Дерево T_1 состоит из одной дуги, помеченной символом $S(1)$. Формальное описание приведено в алгоритме 1.

Общее описание алгоритма Укконена
Построить дерево T_1.
for i from 1 to $m-1$ do begin {фаза $i+1$}
  for j from 1 to $i+1$ do begin {продолжение j}
    найти в текущем дереве конец пути из корня с меткой $S[j..i]$.
    если нужно, продолжить путь, добавив символ $S(i+1)$, обеспечив
      появление строки $S[j..i+1]$ в дереве.
  end;
end;

Алгоритм 1. Общее описание алгоритма Укконена

2.2.2. Правила продолжения суффиксов

Для того, чтобы превратить изложенное выше общее описание алгоритма Укконена в конкретный алгоритм, необходимо точно указать, как выполнять *продолжение суффикса*. Пусть $S[j..i] = \beta$ — суффикс $S[1..i]$. В продолжении с номером j, когда алгоритм находит конец суффикса β в текущем дереве, алгоритм продолжает β для того, чтобы обеспечить присутствие суффикса $\beta S(i+1)$ в дереве. Алгоритм действует по одному из следующих трех правил.

Правило 1. В текущем дереве путь β кончается в листе. Это значит, что путь от корня с меткой β доходит до конца некоторой «листовой» дуги (дуги, входящей в лист). При изменении дерева следует добавить к концу метки этой листовой дуги символ $S(i+1)$.

Правило 2. Ни один путь из конца строки β не начинается символом $S(i+1)$, но, по крайней мере, один начинающийся оттуда путь имеется.

В этом случае должна быть создана новая листовая дуга, начинающаяся в конце β и помеченная символом $S(i+1)$. При этом, если β кончается внутри дуги, должна быть создана новая вершина. Листу в конце новой листовой дуги сопоставляется номер j.

Правило 3. Некоторый путь из конца строки β начинается символом $S(i+1)$. В этом случае строка $\beta S(i+1)$ уже имеется в текущем дереве, и поэтому ничего не требуется делать.

2.3. Реализация и ускорение

Ключевым моментом в реализации алгоритма Укконена является определение концов всех $i+1$ суффиксов $S[1..i]$. Действуя наивно, можно найти конец любого суффикса β за время $O(|\beta|)$, двигаясь от корня текущего дерева. При этом подходе продолжение j в фазе $i+1$ займет время $O(i+1-j)$, T_{i+1} будет создано из T_i за $O(i^2)$ и T_m получится за время $O(m^3)$.

Сведем $O(m^3)$ к $O(m^2)$ с помощью нескольких наблюдений и приемов реализации. Каждый прием сам по себе выглядит как полезная эвристика для ускорения алгоритма, и, действуя по отдельности, они не могут уменьшить оценку времени для наихудшего случая. Результат достигается только при совместном их применении. Наиболее существенный элемент ускорения — это использование *суффиксных связей.*

2.3.1. Суффиксные связи: первое ускорение реализации

> **Определение.** Пусть $x\alpha$ обозначает произвольную строку, где x — ее первый символ, а α — оставшаяся подстрока (возможно, пустая). Если для внутренней вершины v с путевой меткой $x\alpha$ существует другая вершина $s(v)$ с путевой меткой α, то указатель из v в $s(v)$ называется *суффиксной связью.*

Хотя из определения суффиксных связей не следует, что из каждой внутренней вершины неявного суффиксного дерева выходит суффиксная связь, на самом деле так и будет.

Теорема. В алгоритме Укконена любая вновь созданная внутренняя вершина будет иметь суффиксную связь с концом следующего продолжения.

Таким образом, суффиксные связи будут исходить из всех внутренних вершин изменяющегося дерева, кроме внутренней вершины, введенной последней. Она получит свою суффиксную связь в конце следующего продолжения. Покажем, как суффиксные связи используются для ускорения вычислений.

2.3.2. Переходы по суффиксным связям при построении T_{i+1}

Заметим, что конец полной строки $S[1..i]$ должен быть в листе дерева T_i, так как $S[1..i]$ — самая длинная строка в нем. Следовательно, конец этого суффикса найти легко. Представим строку $S[1..i]$ в виде $x\alpha$, где x — один символ, а α — оставшаяся подстрока (возможно, пустая), и пусть $(v, 1)$ — дуга дерева, входящая в лист 1. Опишем подробно второе продолжение. Пусть γ обозначает дуговую метку дуги $(v, 1)$. Для того, чтобы найти конец α, требуется пройти вверх от листа 1 до вершины v, далее по суффиксной связи из v в $s(v)$, а затем от $s(v)$ — вниз по пути (который может состоять больше чем из одной дуги) с меткой γ. Конец этого пути и является концом α. В конце пути α дерево изменяется по правилам продолжения суффика.

При продолжении любой строки $S[j..i]$ до $S[j..i+1]$ при $j > 2$ следуем той же общей идее. Начиная в конце строки $S[j-1..i]$ в текущем дереве, поднимаемся вверх не больше чем на одну вершину, попадая либо в корень, либо в вершину v, из которой выходит суффиксная связь. Далее, пусть γ — дуговая метка этой дуги; в случае, если v не корень, проходим суффиксную связь из v в $s(v)$ и спускаемся по дереву из $s(v)$ по пути, помеченному γ, к концу $S[j..i]$. И наконец, продолжаем суффикс до $S[j..i+1]$ по правилам продолжения.

2.3.3. Алгоритм отдельного продолжения

Собирая вместе все эти фрагменты, реализующие использование суффиксных связей, можно записать продолжение $j \geq 2$ фазы $i+1$ (алгоритм 2).

Использование суффиксных связей дает очевидное практическое улучшение за счет сокращения передвижений от корня в каждом продолжении, которые выполняются в наивном алгоритме. Однако это не улучшит оценку времени выполнения в наихудшем случае. Введем особый прием, который уменьшит оценку для алгоритма до $O(m^2)$

На шаге 2 продолжения $j+1$ алгоритм идет *вниз* из вершины $s(v)$ по пути с меткой γ. При буквальной реализации прохождение вдоль γ занимает время, пропорциональное $|\gamma|$, *числу символов* в этом пути. Однако простой прием, именуемый *скачком по счетчику*, уменьшит время перехода до приблизительно пропорционального *числу вершин* в пути. Отсюда будет следовать, что время на все проходы вниз

Алгоритм отдельного продолжения

begin

1. Найти в конце строки $S[j-1..i]$ или выше его первую вершину v, которая либо имеет исходящую суффиксную связь, либо является корнем. Для этого необходимо пройти вверх от конца $S[j-1..i]$ в текущем дереве не более чем на одну дугу. Пусть γ обозначает строку между v и концом $S[j-1..i]$ (возможно, пустую).

2. Если v не корень, пройти суффиксную связь из v в вершину $s(v)$ и спуститься из $s(v)$ по пути строки γ. Если v — корень, пройти по пути для $S[j..i]$ из корня (как в наивном алгоритме).

3. Обеспечить вхождение строки $S[j..i]S(i+1)$ в дерево, используя правила продолжения.

4. Если в продолжении $j-1$ была создана новая внутренняя вершина ω (по правилу продолжения 2), то строка α должна кончаться в вершине $s(\omega)$, конце суффиксной связи из ω. Создать эту суффиксную связь $(\omega, s(\omega))$.

end.

Алгоритм 2. Алгоритм отдельного продолжения

за фазу не превзойдет $O(m)$.

Пусть g обозначает длину γ. Напомним, что никакие две метки дуг, выходящих из $s(u)$, не могут начинаться с одного и того же символа, так что первый символ γ должен быть начальным только у одной дуги, выходящей из $s(v)$. Пусть g' — число символов в этой дуге. Если $g' < g$, то алгоритму не требуется просматривать остальные символы дуги — он просто перепрыгивает к ее концу. Затем g полагается равным $g - g'$, $h = g' + 1$, и просматриваются выходящие дуги для того, чтобы найти среди них правильное продолжение (с начальным символом, равным символу h строки γ). По достижении дуги, в которой g не превосходит g', алгоритм переходит к символу g на дуге и завершается, поскольку путь γ из $s(v)$ кончается на этой дуге, строго в g символах вниз по метке.

Эффект от использования скачков по счетчику будет в обеспечении перехода от вершины пути γ к другой вершине за *константное* время. Полное время прохода по пути становится при этом пропорциональным числу *вершин*, а не символов в нем.

Теорема. При использовании скачков по счетчику любая фаза алгоритма Укконена занимает время $O(m)$.

Всего у нас m фаз, так что верна теорема, приведенная ниже.

Теорема. Суффиксные связи в алгоритме Укконена обеспечивают время работы

$O(m^2)$.

Заметим, что временная граница для алгоритма $O(m^2)$ получена умножением временной оценки отдельной фазы $O(m)$ на m (число фаз). Такое грубое умножение было необходимо, так как временной анализ проводился для отдельной фазы. После применения некоторых изменений в реализации, которые позволят провести временной анализ, выходящий за границы отдельных фаз, возможно достичь оценки времени работы $O(m)$. Необходимые приемы описаны в работе [4].

3. Преобразованная программа

Приложение 1 содержит описывающий логику визуализатора XML-файл, в котором используются только следующие базовые конструкции:

- последовательные шаги визуализации;
- операторы ветвления (*if-then-else*);
- циклы с предусловием (*while*).

В таком виде программа может быть преобразована в систему конечных автоматов. Технология *Vizi* генерирует на его основе программу на языке программирования Java, работающую под управлением конечных автоматов, код которой приведен в приложении 2.

Всего в программе два автомата — прямого и обратного хода алгоритма. В обоих автоматах по 55 состояний, которые одинаковы для обоих автоматов. Сравнительно большое количество состояний в автоматах связано с подробной визуализацией алгоритма.

4. Описание интерфейса визуализатора

Внешний вид визуализатора представлен на рис. 1. В верхней части окна находится текущее суффиксное дерево. Его листья помечены соответствующими им номерами (в полном суффиксном дереве лист с номером i соотвествует префиксу $S[i..m]$). Ребра дерева помечены соответствующими строковыми метками. Ребра, по которым требуется подняться или спуститься на текущем шагу визуализации, выделяются цветом. Справа показана часть строки, конец которой необходимо найти в дереве на текущем шаге. Внизу — панель, в которой показана вся строка. В ней отдельно выделена часть строки, которую необходимо вставить в дерево на текущем продолжении. Панель можно убрать, сбросив флажок в элементе управления «Строка».

Ниже расположена область, где отображаются пояснения к тому, что делает алгоритм, и элементы управления. Апплет имеет следующие элементы управления:

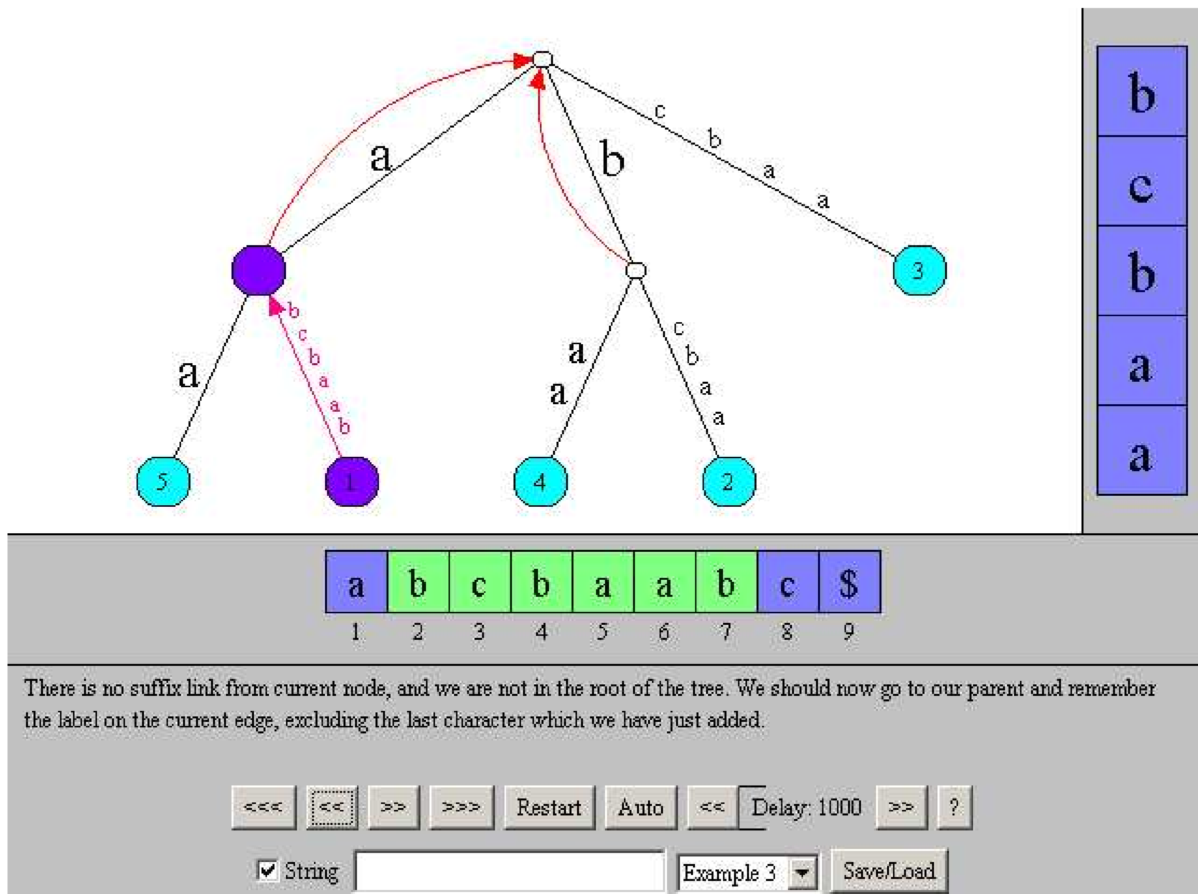

Рис. 1. Внешний вид визуализатора

1. Кнопка «>>» служит для перехода к следующему шагу визуализации.

2. Кнопка «<<» служит для перехода к предыдущему шагу визуализации.

3. Кнопки «>>>» служит для перехода к следующему продолжению.

4. Кнопка «<<<» служит для перехода к предыдущему продолжению.

5. Кнопка «Рестарт» позволяет запустить алгоритм с самого начала.

6. Кнопка «Авто» запускает визуализацию в автоматическом режиме. В этом режиме визуализации апплет самостоятельно переходит к следующему шагу по истечении определенной паузы.

7. Улемент управления «Задержка» регулирует паузу между шагами в автоматическом режиме.

8. Кнопка «?» показывает диалоговое окно с информацией об визуализаторе.

9. Флажок «Строка» позволяет убрать панель с текущей строкой.

10. Текстовое поле позволяет ввести произвольную строку, которая будет использоваться для построения суффиксного дерева.

11. Выпадающее меню служит для выбора одного из предопределенных примеров входных данных.

12. Кнопка «Сохранить/Загрузить» вызывает диалоговое окно сохранения и загрузки состояния визуализатора.

5. Описание конфигурации визуализатора

Параметры визуализатора хранятся в файле `Ukkonen-Configuration.xml`. Благодаря этому можно без труда изменять цветовую гамму и вид элементов управления, а также некоторые другие настройки.

Таблица, приведенная ниже, содержит описание нестандартных параметров визуализатора.

Параметр	Описание
`string-visualizer-height`	Высота панели со строкой.
`string-visualizer-width`	Ширина панели со строкой.
`string-visualizer-background`	Цвет фона панели со строкой.
`string-visualizer`	Стили панели со строкой.
`-style0`	Базовый стиль.
`-style1`	Стиль обычной ячейки.
`-style2`	Стиль подсвеченной ячейки.
`-style3`	Стиль выделенной ячейки.
`-style4`	Стиль для отображения чисел.
`SaveLoadDialog`	Параметры диалога сохранения/загрузки.
`-CommentPane-lines`	Количество строк в панели с комментарием.
`-columns`	Ширина текстовой области.
`-rows`	Высота текстовой области.
`examples-num`	Количество примеров.
`example-1-title`	Заголовок первого примера.
`example-1`	Строка первого примера.
`example-2-title`	Заголовок второго примера.
`example-2`	Строка второго примера.
...	...
`examples-hint`	Подсказка к списку примеров.
`suffix-tree`	Стили суффиксного дерева.
`-style0`	Стиль листьев.
`-style1`	Стиль выделенных листьев.
`-style2`	Стиль внутренних узлов.

`-style3`	Стиль выделенных внутренних узлов.
`-style4`	Стиль ребер дерева.
`-style5`	Стиль выделенных ребер дерева.
`-style6`	Стиль суффиксных ссылок.
`-style7`	Стиль выделенных суффиксных ссылок.
`string-field-hint`	Подсказка к полю ввода строки.

Полностью файл `Ukkonen-Configuration.xml` приведен в приложении 1.

Заключение

Применение технологии *Vizi*, предназначенной для формализации описания логики работы на основе автоматного подхода к созданию программного обеспечения, упростило создание визуализаторов по сравнению с традиционным подходом.

Список литературы

[1] Ukkonen E. *On-line construction of suffix-trees* // Algorithmica. 1995. Vol. 14. P. 249–260.

[2] Weiner P. *Linear pattern matching algorithms* // Proc. of the 14th IEEE Symp. on Switching and Automata Theory. 1973. P. 1–11.

[3] McCreight E. M. *A space-economical suffix tree construction algorithm* // J. ACM. 1976. Vol. 23. P. 58–66.

[4] Gusfield D. *Algorithms on strings, trees and sequences: computer science and computational biology* // New York: Cambridge University Press, 1997. [Русск. пер.: Гасфилд Д. *Строки, деревья и последовательности в алгоритмах. Информатика и вычислительная биология.* СПб.: Невский Диалект, 2003.]

Приложение 1. XML-описание визуализатора

Описание разбито на три файла:

- `Ukkonen.xml` — содержит общую информацию о визуализаторе и ссылки на два следующих файла;
- `Ukkonen-Algorithm.xml` — описывает структуру автоматов, реализующих алгоритм;
- `Ukkonen-Configuration.xml` — содержит параметры визуализатора.

Ukkonen.xml

```
<?xml version="1.0" encoding="WINDOWS-1251"?>

<!--
    "Ukkonen" visualizer description (example)
-->

<!DOCTYPE visualizer PUBLIC
    "-//IFMO Vizi//Visualizer description"
    "http://ips.ifmo.ru/vizi/dtd/visualizer.dtd"
[
    <!ENTITY algorithm SYSTEM "Ukkonen-Algorithm.xml">
    <!ENTITY configuration SYSTEM "Ukkonen-Configuration.xml">
]>

<visualizer
    id="Ukkonen"
    package="ru.ifmo.vizi.ukkonen"
    main-class="UkkonenVisualizer"

    preferred-width="640"
    preferred-height="480"

    name-ru="Алгоритм Укконена"
    name-en="Ukkonen algorithm"

    author-ru="Игорь Ахметов"
    author-en="Igor Ahmetov"
    author-email="ahmetov@rain.ifmo.ru"

    supervisor-ru="Георгий Корнеев"
    supervisor-en="Georgiy Korneev"
    supervisor-email="kgeorgiy@rain.ifmo.ru"

    copyright-ru="Copyright \u00A9 Кафедра КТ, СПб ГИТМО (ТУ), 2004"
    copyright-en="Copyright \u00A9 Computer Technologies Department, SPb IFMO, 2004"
>
    &algorithm;
    &configuration;
</visualizer>
```

Ukkonen-Algorithm.xml

```
<?xml version="1.0" encoding="WINDOWS-1251"?>

<!--
    "Ukkonen" algoritm description
-->

<algorithm>
    <import>ru.ifmo.vizi.ukkonen.SuffixTree.SuffixTreeState</import>
    <data>
        <variable description="Counter variable for phase loop">int i;</variable>
```

```
        <variable description="Counter variable for extension loop">int j;</variable>
        <variable description="Applet instance">UkkonenVisualizer visualizer;</variable>
        <toString>
            return visualizer.getSuffixTree().toString();
        </toString>
    </data>

    <auto id="Main" description="Builds suffix tree">
        <start
            comment-ru="Введите строку из строчных и заглавных букв
                        для построения суффиксного дерева или выберете
                        один из примеров."
            comment-en="Enter a string for the suffix tree consisting of
                        lowercase or capital letters or choose one of the examples."
        >
            <draw/>
        </start>
        <step
            id="Description1"
            description="Description of the algorithm"
            comment-en="Ukkonen algorithm builds suffix trees T(i) for each prefix
                        S[1..i] of the input string S. Let m be length of the string S.
                        Algorithm is divided into m phases. During phase i+1
                        algorithm builds tree T(i+1) from T(i). Each phase i+1 itself
                        is divided into i+1 extensions, one extension for each of the i+1
                        suffixes S[1..i+1]."
            comment-ru="Алгоритм Укконена строит неявное суффиксное дерево T(i) для
                каждого
                        префикса S[1..i] входной строки S. Пусть m - длина S. Алгоритм
                        делится на m фаз. В фазе i+1 дерево T(i+1) строится из T(i).
                            Каждая
                        фаза i+1 сама делится на i+1 продолжений, по одному на каждый из
                            i+1
                        суффиксов S[1..i+1]."
        >
            <direct/>
        </step>
        <step
            id="Description2"
            description="Description of the algorithm"
            comment-en="During extension j of phase i+1 the algorithm finds end of the
                path
                        from root of the tree with label S[j..i]. Then it extends this
                            substring
                        by appending to its end symbol S[i+1], if S[i+1] doesn''t exists
                            yet.
                        So, in phase i+1 substrings S[1..i+1], S[2..i+1], ... are
                            inserted into
                        the tree."
            comment-ru="В продолжении j фазы i+1 алгоритм сначала находит конец пути из
                корня,
                        помеченного подстрокой S[j..i]. Затем он продолжает эту подстроку
                            , добавляя
                        к ее концу символ S[i+1], если только S[i+1] еще не существует.
                            Итак, в фазе
                        i+1 сначала помещается в дерево строка S[1..i+1], а за ней строки
                             S[2..i+1],
                        S[3..i+1], ..."
        >
            <direct/>
        </step>
        <step
            id="Description3"
            description="Description of the algorithm"
            comment-en="Let xa be an arbitrary string, where x is its first symbol,
                        and a is the remaining substring. If for some inner node v with
                            path label xa
                        there is another node s(v) with path label a, then pointer from
                        v to s(v) is called a suffix link."
            comment-ru="Пусть xa обозначает произвольную строку, где x - ее первый символ
                , a
                        a - оставшаяся подстрока. Если для внутреннего узла v с
                        путевой меткой xa существует другой узел s(v) с путевой меткой a
                            , то указатель
                        из v в s(v) называется суффиксной связью."
        >
```

```
        <direct/>
    </step>
    <step
        id="Description4"
        description="Description of the algorithm"
        comment-en="During work of the algorithm we support following invariance:
                    for each inner node there is a suffix link. Due to the suffix
                        links
                    we can reach asymptotic estimation of the working time of the
                    algorithm O(n^2)."
        comment-ru="Во время работы алгоритма поддерживается следующий инвариант:
                    у каждого внутреннего узла существует суффиксная связь.
                    Благодаря этому приему достигается асимптотическая
                    оценка времени работы алгоритма O(n^2)."
    >
        <direct/>
    </step>
    <step
        id="AddSentinelCharacter"
        description="Adding of the sentinel character"
        level="1"
        comment-en="To build correct suffix tree we need to add
                    to the string some character that it doesn''t already contain.
                    Let it be ''{0}''."
        comment-ru="Чтобы построить корректное суффиксное дерево мы должны
                    добавить к нашей строке какой-нибудь символ, который она еще
                    не содержит. Добавим к строке символ ''{0}''."
        comment-args="d.visualizer.SENTINEL"
    >
        <draw>
            d.visualizer.updateTree();
        </draw>
        <direct>
            d.visualizer.addSentinelCharacter();
        </direct>
        <reverse>
            d.visualizer.removeSentinelCharacter();
        </reverse>
    </step>
    <step
        id="FirstPhase"
        description="First phase"
        level="1"
        comment-en="We insert into the tree the first character of the string and
            remember the resulting leaf
                    (we will use it at the beginning of each phase)."
        comment-ru="Вставим в дерево первый символ строки и запомним получившийся
            лист
                    (мы будем начинать работу в нем в начале каждой фазы)."
    >
        <draw>
            d.visualizer.updateTree();
        </draw>
        <direct>
            d.visualizer.nextStep();
        </direct>
        <reverse>
            d.visualizer.previousStep();
        </reverse>
    </step>
    <step
        id="PhaseLoopInit"
        description="Phase loop counter initialization"
        level="-1"
    >
        <direct>
            d.i = 1;
        </direct>
    </step>
    <while
        id="PhaseLoop"
        description="Phase loop"
        test="d.i &lt; d.visualizer.getString().length()"
        level="-1"
    >
        <step
```

```
    id="PhaseBegin"
    description="Begin of the phase"
    level="1"
    comment-en="Now we are in phase {0}. During this phase we extend the tree
         so it contains
                substrings from ''{2}'' to ''{3}'' (extensions 1..{1}))."
    comment-ru="Переходим к фазе {0}. Во время этой фазы мы расширяем дерево
        так,
                чтобы оно содержало подстроки от ''{2}'' до ''{3}'' (
                    продолжения
                1..{1})."
    comment-args="new Integer(d.i), new Integer(d.i + 1),
                  d.visualizer.getString().substring(0, d.i + 1),
                  d.visualizer.getString().substring(d.i, d.i + 1)"
    >
    <draw>
        d.visualizer.updateTree();
    </draw>
    <direct>
        d.visualizer.nextStep();
    </direct>
    <reverse>
        d.visualizer.previousStep();
    </reverse>
</step>

<step
    id="SelectLeaf"
    description="Selecting leaf as current node"
    comment-en="We choose the leaf of the tree with number 1 as current node.
        "
    comment-ru="Выберем в качестве текущего узла лист дерева с номером 1"
>
    <draw>
        d.visualizer.updateTree();
    </draw>
    <direct>
        d.visualizer.nextStep();
    </direct>
    <reverse>
        d.visualizer.previousStep();
    </reverse>
</step>
<step
    id="FirstExtension"
    description="First extension"
    level="1"
    comment-en="We append character ''{0}'' to the edge that leads to the
                current node to insert prefix ''{1}'' into the tree
                (first extension)."
    comment-ru="Вставим префикс ''{1}'' в дерево, добавив символ ''{0}''
                к ребру, ведущему в текущий узел (первое продолжение)."
    comment-args="new Character(d.visualizer.getString().charAt(d.i)),
                  d.visualizer.getString().substring(0, d.i + 1)"
>
    <draw>
        d.visualizer.updateTree();
    </draw>
    <direct>
        d.visualizer.nextStep();
    </direct>
    <reverse>
        d.visualizer.previousStep();
    </reverse>
</step>
<step
    id="ExtensionLoopInit"
    description="Initialization of the extension loop"
    level="-1"
>
    <direct>
        d.j = 1;
    </direct>
</step>
<while
    id="ExtensionLoop"
```

```
        description="Цикл"
        test="d.j &lt;= d.i"
        level="-1"
    >
        <step
            id="ExtensionBegin"
            description="Extension begin"
            level="1"
            comment-en="Now we should insert into the tree substring ''{0}''
                        (extension number {1})."
            comment-ru="Теперь мы должны вставить в дерево подстроку ''{0}''
                        (продолжение номер {1})."
            comment-args="d.visualizer.getString().substring(d.j, d.i + 1),
                          new Integer(d.j + 1)"
        >
            <draw>
                d.visualizer.updateTree();
            </draw>
            <direct>
                d.visualizer.nextStep();
            </direct>
            <reverse>
                d.visualizer.previousStep();
            </reverse>
        </step>
        <if
            id="IfHasnotSuffixLink1"
            description="Checks if current node has suffix link"
            level="-1"
            test="!d.visualizer.getState().getCurrentNode().hasSuffixLink() &amp
                ;&amp;
                  !d.visualizer.getState().getCurrentNode().isRoot()"

        >
            <then>
                <if
                    id="IfGoingUpFromLeaf"
                    description="Checks if we go up from leaf"
                    level="-1"
                    test="d.visualizer.getState().getCurrentNode().isLeaf()"
                >
                    <then>
                        <step
                            id="NoSuffixLink0"
                            description="There is no suffix link from current
                                node"
                            comment-en="There is no suffix link from current node
                                , and we are not
                                        in the root of the tree. We should now go
                                             to our parent and
                                        remember the label on the current edge,
                                            excluding the last
                                        character which we have just added."
                            comment-ru="У текущей вершины нет суффиксной связи, и
                                 мы находимся не в
                                        корне суффиксного дерева. Поднимемся на
                                            одну вершину вверх,
                                        запомнив строку на текущем ребре за
                                            исключением ее последнего
                                        символа, который мы только что добавили в
                                             дерево."
                        >
                            <draw>
                                d.visualizer.updateTree();
                            </draw>
                            <direct>
                                d.visualizer.nextStep();
                            </direct>
                            <reverse>
                                d.visualizer.previousStep();
                            </reverse>
                        </step>
                    </then>
                    <else>
                        <step
                            id="NoSuffixLink1"
```

```
                    description="There is no suffix link from current
                        node"
                    comment-en="There is no suffix link from current node
                        , and we are not
                                in the root of the tree. We should now go
                                     to our parent and
                                remember the label on the current edge."
                    comment-ru="У текущей вершины нет суффиксной связи, и
                         мы находимся не в
                                корне суффиксного дерева. Поднимемся на
                                    одну вершину вверх,
                                запомнив строку на текущем ребре."
                >
                    <draw>
                        d.visualizer.updateTree();
                    </draw>
                    <direct>
                        d.visualizer.nextStep();
                    </direct>
                    <reverse>
                        d.visualizer.previousStep();
                    </reverse>
                </step>
            </else>
        </if>
    </then>
</if>
<if
    id="IfHasnotSuffixLink2"
    description="Checks if current node has suffix link"
    level="-1"
    test="!d.visualizer.getState().getCurrentNode().hasSuffixLink()"
>
    <then>
        <if
            id="RootAddChar"
            description="We are in root node"
            level="-1"
            test="d.j == d.i"
        >
            <then>
                <step
                    id="NoSuffixLink3"
                    description="Root node"
                    comment-en="We are in root node. Now we should insert
                         symbol ''{0}''
                                into the tree."
                    comment-ru="Мы находимся в корне дерева. Теперь мы
                        должны вставить в
                                дерево символ ''{0}''."
                    comment-args="d.visualizer.getString().substring(d.i
                        , d.i + 1)"
                >
                    <draw>
                        d.visualizer.updateTree();
                    </draw>
                    <direct>
                        d.visualizer.nextStep();
                    </direct>
                    <reverse>
                        d.visualizer.previousStep();
                    </reverse>
                </step>
            </then>
            <else>
                <step
                    id="NoSuffixLink2"
                    description="Root node"
                    comment-en="We are in root node. Now we should find
                        an end of the
                                substring ''{0}'' and append character
                                     ''{1}'' to it."
                    comment-ru="Мы находимся в корне дерева. Теперь мы
                        должны найти конец подстроки ''{0}''
                                и добавить к ее концу символ ''{1}''."
```

```
                    comment-args="d.visualizer.getString().substring(d.j
                        , d.i),
                                  d.visualizer.getString().substring(d.i
                                      , d.i + 1)"
                >
                    <draw>
                        d.visualizer.updateTree();
                    </draw>
                    <direct>
                        d.visualizer.nextStep();
                    </direct>
                    <reverse>
                        d.visualizer.previousStep();
                    </reverse>
                </step>
            </else>
        </if>
    </then>
    <else>
        <step
            id="FoundSuffixLink"
            description="Suffix link found"
            comment-en="There is a suffix link from this node. We should
                follow it now."
            comment-ru="Есть суффиксная связь, ведущая из текущей вершины
                . Пройдем ней."
        >
            <draw>
                d.visualizer.updateTree();
            </draw>
            <direct>
                d.visualizer.nextStep();
            </direct>
            <reverse>
                d.visualizer.previousStep();
            </reverse>
        </step>
        <if
            id="ActiveSubstringLength"
            description="Checks length of the active substring."
            level="-1"
            test="d.visualizer.getState().getActiveSubstring().length()
                 != 0"
        >
            <then>
                <step
                    id="SuffixLinkEnd"
                    description="We've passed the suffix link."
                    comment-en="Now we should find an end from current
                        node of the
                                substring ''{0}'' which we passed when we
                                     were searching for the
                                suffix link and append character ''{1}''
                                    to it."
                    comment-ru="Мы находимся к корне дерева. Теперь мы
                        должны найти конец подстроки
                                ''{0}'', которую мы прошли, когда искали
                                    суффиксную связь,
                                и добавить к ее концу символ ''{1}''."
                    comment-args="d.visualizer.getString().substring(d.j
                        , d.i),
                                  d.visualizer.getString().substring(d.i
                                      , d.i + 1)"
                >
                    <draw>
                        d.visualizer.updateTree();
                    </draw>
                    <direct>
                        d.visualizer.nextStep();
                    </direct>
                    <reverse>
                        d.visualizer.previousStep();
                    </reverse>
                </step>
            </then>
        </if>
```

```
    </else>
</if>

<while
    id="DescentLoop"
    description="Descent"
    test="d.visualizer.getNextState().getSpecialState() ==
          SuffixTreeState.STATE_DESCENT"
    level="-1">
    <if
        id="IsActiveSubstringZeroLength"
        description="Checks if the active substrng if of zero lenght"
        test="d.visualizer.getNextState().getActiveSubstring().length()
             == 0"
        level="-1"
    >
        <then>
            <step
                id="DescentStep1"
                description="Descent step"
                comment-en="Going down through the edge with label
                     ''{0}''.
                            Now we should insert into the suffix tree
                                symbol ''{1}''."
                comment-ru="Проходим по ребру с меткой ''{0}''. Теперь мы
                     должны вставить в дерево
                            символ ''{1}''."
                comment-args="d.visualizer.getState().getCurrentNode().
                    getString(),
                              d.visualizer.getString().substring(d.i, d.i
                                   + 1)"
            >
                <draw>
                    d.visualizer.updateTree();
                </draw>
                <direct>
                    d.visualizer.nextStep();
                </direct>
                <reverse>
                    d.visualizer.previousStep();
                </reverse>
            </step>
        </then>
        <else>
            <step
                id="DescentStep2"
                description="Descent step"
                comment-en="Going down through the edge with label
                     ''{0}''.
                            Now we search the end of substring ''{1}''."
                comment-ru="Проходим по ребру с меткой ''{0}''. Теперь
                    начиная с его конца мы должны найти
                            конец строки ''{1}''"
                comment-args="d.visualizer.getState().getCurrentNode().
                    getString(),
                              d.visualizer.getState().getActiveSubstring
                                  ()"
            >
                <draw>
                    d.visualizer.updateTree();
                </draw>
                <direct>
                    d.visualizer.nextStep();
                </direct>
                <reverse>
                    d.visualizer.previousStep();
                </reverse>
            </step>
        </else>
    </if>
</while>

<if
    id="AppendingChar"
    description="Checks where we need to append new character."
    level="-1"
```

```
    test="d.visualizer.getNextState().getSpecialState() ==
          SuffixTreeState.STATE_ADD_CHAR_LEAF"
>
    <then>
        <step>
            id="AppendCharLeaf"
            description="We''re in leaf."
            comment-en="We''re in leaf, so we just append character
                 ''{0}'' to the
                        label on the edge between this leaf and its
                            parent."
            comment-ru="Мы находимся в листе, поэтому просто добавляем
                символ ''{0}'' к метке на
                        ребре между этим листом и его родителем."
            comment-args="d.visualizer.getString().substring(d.i, d.i
                 + 1)"
        >
            <draw>
                d.visualizer.updateTree();
            </draw>
            <direct>
                d.visualizer.nextStep();
            </direct>
            <reverse>
                d.visualizer.previousStep();
            </reverse>
        </step>
    </then>
    <else>
        <if
            id="AddCharNewLeaf"
            description="Has new leaf been added."
            level="-1"
            test="d.visualizer.getNextState().getSpecialState() ==
                SuffixTreeState.STATE_ADD_CHAR_NEW_LEAF"
        >
            <then>
                <step>
                    id="AddNewLeaf"
                    description="Added new leaf."
                    comment-en="There is no edge from current node that
                        starts with
                                symbol ''{0}'', so we have to create new
                                    leaf with
                                label ''{0}'' on the edge between this
                                    leaf and current node."
                    comment-ru="Из текущего узла не идет ни одного ребра
                        , начинающегося на
                                символ ''{0}'', поэтому мы должны создать
                                     новый лист дерева с
                                меткой ''{0}'' на ребре между этим листом
                                     и текущей вершиной."
                    comment-args="d.visualizer.getString().substring(d.i
                        , d.i + 1)"
                >
                    <draw>
                        d.visualizer.updateTree();
                    </draw>
                    <direct>
                        d.visualizer.nextStep();
                    </direct>
                    <reverse>
                        d.visualizer.previousStep();
                    </reverse>
                </step>
            </then>
            <else>
                <if
                   id="AddCharSplitEdge"
                    description="Has edge been split."
                    level="-1"
                    test="d.visualizer.getNextState().getSpecialState()
                         ==
                        SuffixTreeState.STATE_ADD_CHAR_SPLIT_EDGE"
                >
                    <then>
```

```
<step
    id="SplittingEdge"
    description="We have to split an edge."
    comment-en="We have to split the label on
        this edge into labels ''{1}''
                and ''{2}'' by creating a new
                    inner node
                to be able to add new edge with
                    symbol ''{0}'' on it to this
                inner node."
    comment-ru="Нам необходимо разбить строку на
        этом ребро на части с
                ''{1}'' и ''{2}'' создав новый
                внутренний узел, чтобы добавить к
                     этому узлу новое
                ребро с символом ''{0}'' на нем."
    comment-args="d.visualizer.getString().
        substring(d.i, d.i + 1),
                  d.visualizer.getState().
                      getActiveSubstring(),
                  d.visualizer.getState().
                      getCurrentNode().getString
                      ().substring(d.visualizer.
                      getState().
                      getActiveSubstring().length
                      (), d.visualizer.getState()
                      .getCurrentNode().getString
                      ().length())"
>
    <draw>
        d.visualizer.updateTree();
    </draw>
    <direct>
        d.visualizer.nextStep();
    </direct>
    <reverse>
        d.visualizer.previousStep();
    </reverse>
</step>
<step
    id="SplitEdgeAddLeaf"
    description="New leaf added."
    comment-en="After splitting the edge we can
        add new leaf to the tree with
                label ''{0}'' between this leaf
                    and current node."
    comment-ru="После разбиения ребра мы можем
        добавить к дереву новый лист
                с меткой ''{0}'' между этим
                    листом и текущим узлом."
    comment-args="d.visualizer.getString().
        substring(d.i, d.i + 1)"
>
    <draw>
        d.visualizer.updateTree();
    </draw>
    <direct>
        d.visualizer.nextStep();
    </direct>
    <reverse>
        d.visualizer.previousStep();
    </reverse>
</step>
<step
    id="SuffixLinkOnNextStep"
    description="We should create a new suffix
        link during next step."
    comment-en="We have created a new inner node
        of the suffix tree,
                and during next extension we
                should create a suffix link for
                    it."
    comment-ru="Мы создали новый внутренний узел
        суффиксного дерева,
                и во время следующего продолжения
                     шага мы должны
```

```
                                построить для этого узла
                                правильную суффиксную связь."
                    >
                        <draw>
                            d.visualizer.updateTree();
                        </draw>
                        <direct>
                            d.visualizer.nextStep();
                        </direct>
                        <reverse>
                            d.visualizer.previousStep();
                        </reverse>
                    </step>
                </then>
                <else>
                    <step
                        id="AlreadyInTree"
                        description="Our substring is already in the
                            tree."
                        comment-en="Our substring is already in the
                            tree, nothing to be done."
                        comment-ru="Наша подстрока уже содержится в
                            дереве, ничего делать не надо."
                    >
                        <draw>
                            d.visualizer.updateTree();
                        </draw>
                        <direct>
                            d.visualizer.nextStep();
                        </direct>
                        <reverse>
                            d.visualizer.previousStep();
                        </reverse>
                    </step>
                </else>
            </if>
        </else>
    </if>
</else>
</if>

<if
    id="IfLinkHasBeenCreated"
    description="Checks if suffix link has been created"
    level="-1"
    test="d.visualizer.getNextState().getSpecialState() ==
          SuffixTreeState.STATE_LINK_ADDED"
>
    <then>
        <step
            id="LinkHasBeenCreated"
            description="A suffix link has been created in the tree."
            comment-en="We''ve created a new inner node during the
                previous step,
                        and now we should add a suffix link to the tree
                            from this inner node
                        to the current node to keep the tree
                        correct."
            comment-ru="На предыдущем шаге мы добавили к дереву новый
                внутренний узел,
                        и теперь мы должны добавить к дереву новую
                            суффиксную связь, ведущую
                        от этого внутреннего узла к текущему узлу."
        >
            <draw>
                d.visualizer.updateTree();
            </draw>
            <direct>
                d.visualizer.nextStep();
            </direct>
            <reverse>
                d.visualizer.previousStep();
            </reverse>
        </step>
    </then>
</if>
```

```
        <step
            id="ExtensionEnd"
            description="Extension end"
            comment-en="Substring ''{0}'' is in the tree now, continuing to next
                extension."
            comment-ru="Подстрока ''{0}'' в дереве, переходим к следующему
                продолжению."
            comment-args="d.visualizer.getString().substring(d.j, d.i + 1)"
        >
            <draw>
                d.visualizer.updateTree();
            </draw>
            <direct>
                d.visualizer.nextStep();
            </direct>
            <reverse>
                d.visualizer.previousStep();
            </reverse>
        </step>
        <step
            id="ExtensionLoopIncrement"
            description="Extension loop increment"
            level="-1"
        >
            <direct>
                d.j++;
            </direct>
            <reverse>
                d.j--;
            </reverse>
        </step>
    </while>
    <step
        id="PhaseEnd"
        description="Phase end"
        comment-en="End of the phase {0}."
        comment-ru="Конец фазы {0}."
        comment-args="new Integer(d.i)"
    >
        <draw>
            d.visualizer.updateTree();
        </draw>
        <direct>
            d.visualizer.nextStep();
        </direct>
        <reverse>
            d.visualizer.previousStep();
            d.j = d.i + 1;
        </reverse>
    </step>
    <step
        id="PhaseLoopIncrement"
        description="Increment of the phase loop counter"
        level="-1"
    >
        <direct>
            d.i++;
        </direct>
        <reverse>
            d.i--;
        </reverse>
    </step>
</while>
<step
    id="Finish"
    description="End of the algorithm"
    level="-1"
>
    <direct>
        d.visualizer.nextStep();
    </direct>
    <reverse>
        d.visualizer.previousStep();
    </reverse>
</step>
```

```
        <finish
            comment-ru="Суффиксное дерево построено"
            comment-en="Suffix tree has been built."
        >
            <draw>
                d.visualizer.updateTree();
            </draw>
        </finish>
    </auto>
</algorithm>
```

Ukkonen-Configuration.xml

```
<?xml version="1.0" encoding="WINDOWS-1251"?>

<!--
    "Ukkonen" visualizer configuration (example).
-->

<configuration>
    <property
        description = "Comment pane height"
        param       = "comment-height"
        value       = "60"
    />
    <message
        description = 'Comment for "Step" parameter in the output file'
        param       = "StepComment"
        message-ru  = "Номер шага"
        message-en  = "Current step"
    />
    <styleset
        description = "String visualizer style set"
        param       = "string-visualizer"
    >
        <style
            description     = "Visualizer style"
            text-color      = "000000"
            text-align      = "0.5"
            border-color    = "000000"
            border-status   = "true"
            fill-color      = "c0c0c0"
            fill-status     = "true"
            aspect-status   = "false"
            padding         = "0.2"
        >
            <font
                face        = "Serif"
                size        = "12"
                style       = "plain"
            />
        </style>
        <style
            description     = "Ordinary cell"
            fill-color      = "8080ff"
        />
        <style
            description     = "Highlighted cell"
            fill-color      = "80ff80"
        />
        <style
            description     = "Selected cell"
            fill-color      = "ffff00"
        />
        <style
            description     = "Numbers"
            fill-status     = "false"
            border-status   = "false"
        />
    </styleset>
    <group
        description = "Save/Load dialog configuration"
        param       = "SaveLoadDialog"
    >
        <property
```

```
        description = "Height of the comment pane"
        param       = "CommentPane-lines"
        value       = "2"
    />
    <property
        description = "Width of the text area"
        param       = "columns"
        value       = "40"
    />
    <property
        description = "Height of the text area"
        param       = "rows"
        value       = "7"
    />
</group>

<property
    description = "Max string that can be fit into cell"
    param       = "cell-max-string"
    value       = "X"
/>

<property
    description = "Number of examples"
    param       = "examples-num"
    value       = "4"
/>
<message
    description = "First example title"
    param       = "example-1-title"
    message-ru  = "Пример 1"
    message-en  = "Example 1"
/>
<property
    description = "First example"
    param       = "example-1"
    value       = "hello"
/>
<message
    description = "Second example title"
    param       = "example-2-title"
    message-ru  = "Пример 2"
    message-en  = "Example 2"
/>
<property
    description = "Second example"
    param       = "example-2"
    value       = "axabxac"
/>
<message
    description = "Third example title"
    param       = "example-3-title"
    message-ru  = "Пример 3"
    message-en  = "Example 3"
/>
<property
    description = "Third example"
    param       = "example-3"
    value       = "abcbaabc"
/>
<message
    description = "Fourth example title"
    param       = "example-4-title"
    message-ru  = "Пример 4"
    message-en  = "Example 4"
/>
<property
    description = "Fourth example"
    param       = "example-4"
    value       = "abcdefabcuvw"
/>

<message
    description = "Hint for examples choice"
    param       = "examples-hint"
    message-ru  = "Примеры"
```

```
    message-en  = "Examples"
/>

<property
    description = "Height of the string visualizer"
    param       = "string-visualizer-height"
    value       = "0.2"
/>
<property
    description = "Width of the vertical string visualizer"
    param       = "string-visualizer-width"
    value       = "0.1"
/>

<styleset
    description = "Suffix tree styleset"
    param       = "suffix-tree"
>
    <style
        description     = "Leaf node"
        text-color      = "000000"
        text-align      = "0.5"
        border-color    = "000000"
        border-status   = "true"
        fill-color      = "00ffff"
        fill-status     = "true"
        aspect-status   = "false"
        padding         = "0.2"
    >
        <font
            face        = "Serif"
            size        = "12"
            style       = "plain"
        />
    </style>
    <style
        description     = "Selected leaf node style"
        fill-color      = "8000ff"
    />
    <style
        description     = "Node style"
        fill-color      = "ffffff"
    />
    <style
        description     = "Selected node style"
        fill-color      = "8000ff"
    />
    <style
        description     = "Edge style"
        fill-color      = "000000"
        border-color        = "000000"
    />
    <style
        description     = "Selected edge style"
        border-color    = "ff0080"
        fill-color      = "ff0080"
    />
    <style
        description     = "Suffix link style"
        fill-color      = "ff0000"
        border-color    = "ff0000"
    />
    <style
        description     = "Selected suffix link style"
        fill-color      = "ff00ff"
        border-color    = "ff00ff"
    />
</styleset>

<message
    description = 'Comment for "step" parameter in the output file'
    param       = "save-step-comment"
    message-ru  = "Номер шага"
    message-en  = "Current step"
/>
<message
```

```
        description = 'Comment for "string" parameter in the output file'
        param       = "save-string-comment"
        message-ru  = "Строка суффиксного дерева"
        message-en  = "Suffix tree string"
    />

    <message
        description = "Commect for text field for entering suffix tree string"
        param       = "string-field-hint"
        message-ru  = "Введите строку для построения суффиксного дерева"
        message-en  = "Enter a string for the suffix tree"
    />

    <property
        description = "String visualizer background"
        param       = "string-visualizer-background"
        value       = "c0c0c0"
    />

    <message
        description = "Title of the checkbox that controls string visibility"
        param       = "string-checkbox"
        message-en  = "String"
        message-ru  = "Строка"
    />
    <message
        description = "Hint of the checkbox that controls string visibility"
        param       = "string-checkbox-hint"
        message-en  = "Show string of the suffix tree"
        message-ru  = "Показать строку суффиксного дерева"
    />
</configuration>
```

Приложение 2. Сгенерированные исходные коды визуализатора

```
package ru.ifmo.vizi.ukkonen;

import ru.ifmo.vizi.ukkonen.SuffixTree.SuffixTreeState;
import ru.ifmo.vizi.base.auto.*;
import java.util.Locale;

public final class Ukkonen extends BaseAutomataWithListener {
    /**
      * Модель данных.
      */
    public final Data d = new Data();

    /**
      * Конструктор для языка
      */
    public Ukkonen(Locale locale) {
        super("ru.ifmo.vizi.ukkonen.Comments", locale);
        init(new Main(), d);
    }

    /**
      * Данные.
      */
    public final class Data {
        /**
          * Counter variable for phase loop.
          */
        public int i;

        /**
          * Counter variable for extension loop.
          */
        public int j;

        /**
          * Applet instance.
          */
```

```
    public UkkonenVisualizer visualizer;

    public String toString() {
        return visualizer.getSuffixTree().toString();
    }
}

/**
  * Builds suffix tree.
  */
private final class Main extends BaseAutomata implements Automata {
    /**
      * Начальное состояние автомата.
      */
    private final int START_STATE = 0;

    /**
      * Конечное состояние автомата.
      */
    private final int END_STATE = 56;

    /**
      * Конструктор.
      */
    public Main() {
        super(
            "Main",
            0, // Номер начального состояния
            56, // Номер конечного состояния
            new String[]{
                "Начальное состояние",
                "Description of the algorithm",
                "Description of the algorithm",
                "Description of the algorithm",
                "Description of the algorithm",
                "Adding of the sentinel character",
                "First phase",
                "Phase loop counter initialization",
                "Phase loop",
                "Begin of the phase",
                "Selecting leaf as current node",
                "First extension",
                "Initialization of the extension loop",
                "Цикл",
                "Extension begin",
                "Checks if current node has suffix link",
                "Checks if current node has suffix link (окончание)",
                "Checks if we go up from leaf",
                "Checks if we go up from leaf (окончание)",
                "There is no suffix link from current node",
                "There is no suffix link from current node",
                "Checks if current node has suffix link",
                "Checks if current node has suffix link (окончание)",
                "We are in root node",
                "We are in root node (окончание)",
                "Root node",
                "Root node",
                "Suffix link found",
                "Checks length of the active substring.",
                "Checks length of the active substring. (окончание)",
                "We've passed the suffix link.",
                "Descent",
                "Checks if the active substrng if of zero lenght",
                "Checks if the active substrng if of zero lenght (окончание)",
                "Descent step",
                "Descent step",
                "Checks where we need to append new character.",
                "Checks where we need to append new character. (окончание)",
                "We''re in leaf.",
                "Has new leaf been added.",
                "Has new leaf been added. (окончание)",
                "Added new leaf.",
                "Has edge been split.",
                "Has edge been split. (окончание)",
                "We have to split an edge.",
                "New leaf added.",
```

```
            "We should create a new suffix link during next step.",
            "Our substring is already in the tree.",
            "Checks if suffix link has been created",
            "Checks if suffix link has been created (окончание)",
            "A suffix link has been created in the tree.",
            "Extension end",
            "Extension loop increment",
            "Phase end",
            "Increment of the phase loop counter",
            "End of the algorithm",
            "Конечное состояние"
        }, new int[]{
            Integer.MAX_VALUE, // Начальное состояние,
            0, // Description of the algorithm
            0, // Description of the algorithm
            0, // Description of the algorithm
            0, // Description of the algorithm
            1, // Adding of the sentinel character
            1, // First phase
            -1, // Phase loop counter initialization
            -1, // Phase loop
            1, // Begin of the phase
            0, // Selecting leaf as current node
            1, // First extension
            -1, // Initialization of the extension loop
            -1, // Цикл
            1, // Extension begin
            -1, // Checks if current node has suffix link
            -1, // Checks if current node has suffix link (окончание)
            -1, // Checks if we go up from leaf
            -1, // Checks if we go up from leaf (окончание)
            0, // There is no suffix link from current node
            0, // There is no suffix link from current node
            -1, // Checks if current node has suffix link
            -1, // Checks if current node has suffix link (окончание)
            -1, // We are in root node
            -1, // We are in root node (окончание)
            0, // Root node
            0, // Root node
            0, // Suffix link found
            -1, // Checks length of the active substring.
            -1, // Checks length of the active substring. (окончание)
            0, // We've passed the suffix link.
            -1, // Descent
            -1, // Checks if the active substrng if of zero lenght
            -1, // Checks if the active substrng if of zero lenght (окончание)
            0, // Descent step
            0, // Descent step
            -1, // Checks where we need to append new character.
            -1, // Checks where we need to append new character. (окончание)
            0, // We''re in leaf.
            -1, // Has new leaf been added.
            -1, // Has new leaf been added. (окончание)
            0, // Added new leaf.
            -1, // Has edge been split.
            -1, // Has edge been split. (окончание)
            0, // We have to split an edge.
            0, // New leaf added.
            0, // We should create a new suffix link during next step.
            0, // Our substring is already in the tree.
            -1, // Checks if suffix link has been created
            -1, // Checks if suffix link has been created (окончание)
            0, // A suffix link has been created in the tree.
            0, // Extension end
            -1, // Extension loop increment
            0, // Phase end
            -1, // Increment of the phase loop counter
            -1, // End of the algorithm
            Integer.MAX_VALUE, // Конечное состояние
        }
    );
}

/**
  * Сделать один шаг автомата в перед.
  */
```

```
protected void doStepForward(int level) {
    // Переход в следующее состояние
    switch (state) {
        case START_STATE: { // Начальное состояние
            state = 1; // Description of the algorithm
            break;
        }
        case 1: { // Description of the algorithm
            state = 2; // Description of the algorithm
            break;
        }
        case 2: { // Description of the algorithm
            state = 3; // Description of the algorithm
            break;
        }
        case 3: { // Description of the algorithm
            state = 4; // Description of the algorithm
            break;
        }
        case 4: { // Description of the algorithm
            state = 5; // Adding of the sentinel character
            break;
        }
        case 5: { // Adding of the sentinel character
            state = 6; // First phase
            break;
        }
        case 6: { // First phase
            state = 7; // Phase loop counter initialization
            break;
        }
        case 7: { // Phase loop counter initialization
            stack.pushBoolean(false);
            state = 8; // Phase loop
            break;
        }
        case 8: { // Phase loop
            if (d.i < d.visualizer.getString().length()) {
                state = 9; // Begin of the phase
            } else {
                state = 55; // End of the algorithm
            }
            break;
        }
        case 9: { // Begin of the phase
            state = 10; // Selecting leaf as current node
            break;
        }
        case 10: { // Selecting leaf as current node
            state = 11; // First extension
            break;
        }
        case 11: { // First extension
            state = 12; // Initialization of the extension loop
            break;
        }
        case 12: { // Initialization of the extension loop
            stack.pushBoolean(false);
            state = 13; // Цикл
            break;
        }
        case 13: { // Цикл
            if (d.j <= d.i) {
                state = 14; // Extension begin
            } else {
                state = 53; // Phase end
            }
            break;
        }
        case 14: { // Extension begin
            state = 15; // Checks if current node has suffix link
            break;
        }
        case 15: { // Checks if current node has suffix link
            if (!d.visualizer.getState().getCurrentNode().hasSuffixLink() &&
                                              !d.visualizer.getState().getCurrentNode
```

```
        ().isRoot()) {
            state = 17; // Checks if we go up from leaf
        } else {
            stack.pushBoolean(false);
            state = 16; // Checks if current node has suffix link (окончание)
        }
        break;
    }
    case 16: { // Checks if current node has suffix link (окончание)
        state = 21; // Checks if current node has suffix link
        break;
    }
    case 17: { // Checks if we go up from leaf
        if (d.visualizer.getState().getCurrentNode().isLeaf()) {
            state = 19; // There is no suffix link from current node
        } else {
            state = 20; // There is no suffix link from current node
        }
        break;
    }
    case 18: { // Checks if we go up from leaf (окончание)
        stack.pushBoolean(true);
        state = 16; // Checks if current node has suffix link (окончание)
        break;
    }
    case 19: { // There is no suffix link from current node
        stack.pushBoolean(true);
        state = 18; // Checks if we go up from leaf (окончание)
        break;
    }
    case 20: { // There is no suffix link from current node
        stack.pushBoolean(false);
        state = 18; // Checks if we go up from leaf (окончание)
        break;
    }
    case 21: { // Checks if current node has suffix link
        if (!d.visualizer.getState().getCurrentNode().hasSuffixLink()) {
            state = 23; // We are in root node
        } else {
            state = 27; // Suffix link found
        }
        break;
    }
    case 22: { // Checks if current node has suffix link (окончание)
        stack.pushBoolean(false);
        state = 31; // Descent
        break;
    }
    case 23: { // We are in root node
        if (d.j == d.i) {
            state = 25; // Root node
        } else {
            state = 26; // Root node
        }
        break;
    }
    case 24: { // We are in root node (окончание)
        stack.pushBoolean(true);
        state = 22; // Checks if current node has suffix link (окончание)
        break;
    }
    case 25: { // Root node
        stack.pushBoolean(true);
        state = 24; // We are in root node (окончание)
        break;
    }
    case 26: { // Root node
        stack.pushBoolean(false);
        state = 24; // We are in root node (окончание)
        break;
    }
    case 27: { // Suffix link found
        state = 28; // Checks length of the active substring.
        break;
    }
    case 28: { // Checks length of the active substring.
```

```
    if (d.visualizer.getState().getActiveSubstring().length() != 0) {
        state = 30; // We've passed the suffix link.
    } else {
        stack.pushBoolean(false);
        state = 29; // Checks length of the active substring. (окончание)
    }
    break;
}
case 29: { // Checks length of the active substring. (окончание)
    stack.pushBoolean(false);
    state = 22; // Checks if current node has suffix link (окончание)
    break;
}
case 30: { // We've passed the suffix link.
    stack.pushBoolean(true);
    state = 29; // Checks length of the active substring. (окончание)
    break;
}
case 31: { // Descent
    if (d.visualizer.getNextState().getSpecialState() ==
                                SuffixTreeState.STATE_DESCENT) {
        state = 32; // Checks if the active substrng if of zero lenght
    } else {
        state = 36; // Checks where we need to append new character.
    }
    break;
}
case 32: { // Checks if the active substrng if of zero lenght
    if (d.visualizer.getNextState().getActiveSubstring().length() == 0) {
        state = 34; // Descent step
    } else {
        state = 35; // Descent step
    }
    break;
}
case 33: { // Checks if the active substrng if of zero lenght (окончание)
    stack.pushBoolean(true);
    state = 31; // Descent
    break;
}
case 34: { // Descent step
    stack.pushBoolean(true);
    state = 33; // Checks if the active substrng if of zero lenght (
        окончание)
    break;
}
case 35: { // Descent step
    stack.pushBoolean(false);
    state = 33; // Checks if the active substrng if of zero lenght (
        окончание)
    break;
}
case 36: { // Checks where we need to append new character.
    if (d.visualizer.getNextState().getSpecialState() ==
                                SuffixTreeState.STATE_ADD_CHAR_LEAF) {
        state = 38; // We''re in leaf.
    } else {
        state = 39; // Has new leaf been added.
    }
    break;
}
case 37: { // Checks where we need to append new character. (окончание)
    state = 48; // Checks if suffix link has been created
    break;
}
case 38: { // We''re in leaf.
    stack.pushBoolean(true);
    state = 37; // Checks where we need to append new character. (
        окончание)
    break;
}
case 39: { // Has new leaf been added.
    if (d.visualizer.getNextState().getSpecialState() ==
                                      SuffixTreeState.
        STATE_ADD_CHAR_NEW_LEAF) {
        state = 41; // Added new leaf.
```

```
        } else {
            state = 42; // Has edge been split.
        }
        break;
    }
    case 40: { // Has new leaf been added. (окончание)
        stack.pushBoolean(false);
        state = 37; // Checks where we need to append new character. (
            окончание)
        break;
    }
    case 41: { // Added new leaf.
        stack.pushBoolean(true);
        state = 40; // Has new leaf been added. (окончание)
        break;
    }
    case 42: { // Has edge been split.
        if (d.visualizer.getNextState().getSpecialState() ==
                                                    SuffixTreeState.
            STATE_ADD_CHAR_SPLIT_EDGE) {
            state = 44; // We have to split an edge.
        } else {
            state = 47; // Our substring is already in the tree.
        }
        break;
    }
    case 43: { // Has edge been split. (окончание)
        stack.pushBoolean(false);
        state = 40; // Has new leaf been added. (окончание)
        break;
    }
    case 44: { // We have to split an edge.
        state = 45; // New leaf added.
        break;
    }
    case 45: { // New leaf added.
        state = 46; // We should create a new suffix link during next step.
        break;
    }
    case 46: { // We should create a new suffix link during next step.
        stack.pushBoolean(true);
        state = 43; // Has edge been split. (окончание)
        break;
    }
    case 47: { // Our substring is already in the tree.
        stack.pushBoolean(false);
        state = 43; // Has edge been split. (окончание)
        break;
    }
    case 48: { // Checks if suffix link has been created
        if (d.visualizer.getNextState().getSpecialState() ==
                                SuffixTreeState.STATE_LINK_ADDED) {
            state = 50; // A suffix link has been created in the tree.
        } else {
            stack.pushBoolean(false);
            state = 49; // Checks if suffix link has been created (окончание)
        }
        break;
    }
    case 49: { // Checks if suffix link has been created (окончание)
        state = 51; // Extension end
        break;
    }
    case 50: { // A suffix link has been created in the tree.
        stack.pushBoolean(true);
        state = 49; // Checks if suffix link has been created (окончание)
        break;
    }
    case 51: { // Extension end
        state = 52; // Extension loop increment
        break;
    }
    case 52: { // Extension loop increment
        stack.pushBoolean(true);
        state = 13; // Цикл
        break;
```

```
        }
        case 53: { // Phase end
            state = 54; // Increment of the phase loop counter
            break;
        }
        case 54: { // Increment of the phase loop counter
            stack.pushBoolean(true);
            state = 8; // Phase loop
            break;
        }
        case 55: { // End of the algorithm
            state = END_STATE;
            break;
        }
    }

    // Действие в текущем состоянии
    switch (state) {
        case 1: { // Description of the algorithm
            break;
        }
        case 2: { // Description of the algorithm
            break;
        }
        case 3: { // Description of the algorithm
            break;
        }
        case 4: { // Description of the algorithm
            break;
        }
        case 5: { // Adding of the sentinel character
            d.visualizer.addSentinelCharacter();
            break;
        }
        case 6: { // First phase
            d.visualizer.nextStep();
            break;
        }
        case 7: { // Phase loop counter initialization
            d.i = 1;
            break;
        }
        case 8: { // Phase loop
            break;
        }
        case 9: { // Begin of the phase
            d.visualizer.nextStep();
            break;
        }
        case 10: { // Selecting leaf as current node
            d.visualizer.nextStep();
            break;
        }
        case 11: { // First extension
            d.visualizer.nextStep();
            break;
        }
        case 12: { // Initialization of the extension loop
            d.j = 1;
            break;
        }
        case 13: { // Цикл
            break;
        }
        case 14: { // Extension begin
            d.visualizer.nextStep();
            break;
        }
        case 15: { // Checks if current node has suffix link
            break;
        }
        case 16: { // Checks if current node has suffix link (окончание)
            break;
        }
        case 17: { // Checks if we go up from leaf
            break;
```

```
}
case 18: { // Checks if we go up from leaf (окончание)
    break;
}
case 19: { // There is no suffix link from current node
    d.visualizer.nextStep();
    break;
}
case 20: { // There is no suffix link from current node
    d.visualizer.nextStep();
    break;
}
case 21: { // Checks if current node has suffix link
    break;
}
case 22: { // Checks if current node has suffix link (окончание)
    break;
}
case 23: { // We are in root node
    break;
}
case 24: { // We are in root node (окончание)
    break;
}
case 25: { // Root node
    d.visualizer.nextStep();
    break;
}
case 26: { // Root node
    d.visualizer.nextStep();
    break;
}
case 27: { // Suffix link found
    d.visualizer.nextStep();
    break;
}
case 28: { // Checks length of the active substring.
    break;
}
case 29: { // Checks length of the active substring. (окончание)
    break;
}
case 30: { // We've passed the suffix link.
    d.visualizer.nextStep();
    break;
}
case 31: { // Descent
    break;
}
case 32: { // Checks if the active substrng if of zero lenght
    break;
}
case 33: { // Checks if the active substrng if of zero lenght (окончание)
    break;
}
case 34: { // Descent step
    d.visualizer.nextStep();
    break;
}
case 35: { // Descent step
    d.visualizer.nextStep();
    break;
}
case 36: { // Checks where we need to append new character.
    break;
}
case 37: { // Checks where we need to append new character. (окончание)
    break;
}
case 38: { // We''re in leaf.
    d.visualizer.nextStep();
    break;
}
case 39: { // Has new leaf been added.
    break;
}
```

```
            case 40: { // Has new leaf been added. (окончание)
                break;
            }
            case 41: { // Added new leaf.
                d.visualizer.nextStep();
                break;
            }
            case 42: { // Has edge been split.
                break;
            }
            case 43: { // Has edge been split. (окончание)
                break;
            }
            case 44: { // We have to split an edge.
                d.visualizer.nextStep();
                break;
            }
            case 45: { // New leaf added.
                d.visualizer.nextStep();
                break;
            }
            case 46: { // We should create a new suffix link during next step.
                d.visualizer.nextStep();
                break;
            }
            case 47: { // Our substring is already in the tree.
                d.visualizer.nextStep();
                break;
            }
            case 48: { // Checks if suffix link has been created
                break;
            }
            case 49: { // Checks if suffix link has been created (окончание)
                break;
            }
            case 50: { // A suffix link has been created in the tree.
                d.visualizer.nextStep();
                break;
            }
            case 51: { // Extension end
                d.visualizer.nextStep();
                break;
            }
            case 52: { // Extension loop increment
                d.j++;
                break;
            }
            case 53: { // Phase end
                d.visualizer.nextStep();
                break;
            }
            case 54: { // Increment of the phase loop counter
                d.i++;
                break;
            }
            case 55: { // End of the algorithm
                d.visualizer.nextStep();
                break;
            }
        }
    }

    /**
     * Сделать один шаг автомата назад.
     */
    protected void doStepBackward(int level) {
        // Обращение действия в текущем состоянии
        switch (state) {
            case 1: { // Description of the algorithm
                break;
            }
            case 2: { // Description of the algorithm
                break;
            }
            case 3: { // Description of the algorithm
                break;
```

```
}
case 4: { // Description of the algorithm
    break;
}
case 5: { // Adding of the sentinel character
    d.visualizer.removeSentinelCharacter();
    break;
}
case 6: { // First phase
    d.visualizer.previousStep();
    break;
}
case 7: { // Phase loop counter initialization
    break;
}
case 8: { // Phase loop
    break;
}
case 9: { // Begin of the phase
    d.visualizer.previousStep();
    break;
}
case 10: { // Selecting leaf as current node
    d.visualizer.previousStep();
    break;
}
case 11: { // First extension
    d.visualizer.previousStep();
    break;
}
case 12: { // Initialization of the extension loop
    break;
}
case 13: { // Цикл
    break;
}
case 14: { // Extension begin
    d.visualizer.previousStep();
    break;
}
case 15: { // Checks if current node has suffix link
    break;
}
case 16: { // Checks if current node has suffix link (окончание)
    break;
}
case 17: { // Checks if we go up from leaf
    break;
}
case 18: { // Checks if we go up from leaf (окончание)
    break;
}
case 19: { // There is no suffix link from current node
    d.visualizer.previousStep();
    break;
}
case 20: { // There is no suffix link from current node
    d.visualizer.previousStep();
    break;
}
case 21: { // Checks if current node has suffix link
    break;
}
case 22: { // Checks if current node has suffix link (окончание)
    break;
}
case 23: { // We are in root node
    break;
}
case 24: { // We are in root node (окончание)
    break;
}
case 25: { // Root node
    d.visualizer.previousStep();
    break;
}
```

```
case 26: { // Root node
    d.visualizer.previousStep();
    break;
}
case 27: { // Suffix link found
    d.visualizer.previousStep();
    break;
}
case 28: { // Checks length of the active substring.
    break;
}
case 29: { // Checks length of the active substring. (окончание)
    break;
}
case 30: { // We've passed the suffix link.
    d.visualizer.previousStep();
    break;
}
case 31: { // Descent
    break;
}
case 32: { // Checks if the active substrng if of zero lenght
    break;
}
case 33: { // Checks if the active substrng if of zero lenght (окончание)
    break;
}
case 34: { // Descent step
    d.visualizer.previousStep();
    break;
}
case 35: { // Descent step
    d.visualizer.previousStep();
    break;
}
case 36: { // Checks where we need to append new character.
    break;
}
case 37: { // Checks where we need to append new character. (окончание)
    break;
}
case 38: { // We''re in leaf.
    d.visualizer.previousStep();
    break;
}
case 39: { // Has new leaf been added.
    break;
}
case 40: { // Has new leaf been added. (окончание)
    break;
}
case 41: { // Added new leaf.
    d.visualizer.previousStep();
    break;
}
case 42: { // Has edge been split.
    break;
}
case 43: { // Has edge been split. (окончание)
    break;
}
case 44: { // We have to split an edge.
    d.visualizer.previousStep();
    break;
}
case 45: { // New leaf added.
    d.visualizer.previousStep();
    break;
}
case 46: { // We should create a new suffix link during next step.
    d.visualizer.previousStep();
    break;
}
case 47: { // Our substring is already in the tree.
    d.visualizer.previousStep();
    break;
```

```
        }
        case 48: { // Checks if suffix link has been created
            break;
        }
        case 49: { // Checks if suffix link has been created (окончание)
            break;
        }
        case 50: { // A suffix link has been created in the tree.
            d.visualizer.previousStep();
            break;
        }
        case 51: { // Extension end
            d.visualizer.previousStep();
            break;
        }
        case 52: { // Extension loop increment
            d.j--;
            break;
        }
        case 53: { // Phase end
            d.visualizer.previousStep();
            d.j = d.i + 1;
            break;
        }
        case 54: { // Increment of the phase loop counter
            d.i--;
            break;
        }
        case 55: { // End of the algorithm
            d.visualizer.previousStep();
            break;
        }
    }

    // Переход в предыдущее состояние
    switch (state) {
        case 1: { // Description of the algorithm
            state = START_STATE;
            break;
        }
        case 2: { // Description of the algorithm
            state = 1; // Description of the algorithm
            break;
        }
        case 3: { // Description of the algorithm
            state = 2; // Description of the algorithm
            break;
        }
        case 4: { // Description of the algorithm
            state = 3; // Description of the algorithm
            break;
        }
        case 5: { // Adding of the sentinel character
            state = 4; // Description of the algorithm
            break;
        }
        case 6: { // First phase
            state = 5; // Adding of the sentinel character
            break;
        }
        case 7: { // Phase loop counter initialization
            state = 6; // First phase
            break;
        }
        case 8: { // Phase loop
            if (stack.popBoolean()) {
                state = 54; // Increment of the phase loop counter
            } else {
                state = 7; // Phase loop counter initialization
            }
            break;
        }
        case 9: { // Begin of the phase
            state = 8; // Phase loop
            break;
        }
```

```
case 10: { // Selecting leaf as current node
    state = 9; // Begin of the phase
    break;
}
case 11: { // First extension
    state = 10; // Selecting leaf as current node
    break;
}
case 12: { // Initialization of the extension loop
    state = 11; // First extension
    break;
}
case 13: { // Цикл
    if (stack.popBoolean()) {
        state = 52; // Extension loop increment
    } else {
        state = 12; // Initialization of the extension loop
    }
    break;
}
case 14: { // Extension begin
    state = 13; // Цикл
    break;
}
case 15: { // Checks if current node has suffix link
    state = 14; // Extension begin
    break;
}
case 16: { // Checks if current node has suffix link (окончание)
    if (stack.popBoolean()) {
        state = 18; // Checks if we go up from leaf (окончание)
    } else {
        state = 15; // Checks if current node has suffix link
    }
    break;
}
case 17: { // Checks if we go up from leaf
    state = 15; // Checks if current node has suffix link
    break;
}
case 18: { // Checks if we go up from leaf (окончание)
    if (stack.popBoolean()) {
        state = 19; // There is no suffix link from current node
    } else {
        state = 20; // There is no suffix link from current node
    }
    break;
}
case 19: { // There is no suffix link from current node
    state = 17; // Checks if we go up from leaf
    break;
}
case 20: { // There is no suffix link from current node
    state = 17; // Checks if we go up from leaf
    break;
}
case 21: { // Checks if current node has suffix link
    state = 16; // Checks if current node has suffix link (окончание)
    break;
}
case 22: { // Checks if current node has suffix link (окончание)
    if (stack.popBoolean()) {
        state = 24; // We are in root node (окончание)
    } else {
        state = 29; // Checks length of the active substring. (окончание)
    }
    break;
}
case 23: { // We are in root node
    state = 21; // Checks if current node has suffix link
    break;
}
case 24: { // We are in root node (окончание)
    if (stack.popBoolean()) {
        state = 25; // Root node
    } else {
```

```
            state = 26; // Root node
        }
        break;
    }
    case 25: { // Root node
        state = 23; // We are in root node
        break;
    }
    case 26: { // Root node
        state = 23; // We are in root node
        break;
    }
    case 27: { // Suffix link found
        state = 21; // Checks if current node has suffix link
        break;
    }
    case 28: { // Checks length of the active substring.
        state = 27; // Suffix link found
        break;
    }
    case 29: { // Checks length of the active substring. (окончание)
        if (stack.popBoolean()) {
            state = 30; // We've passed the suffix link.
        } else {
            state = 28; // Checks length of the active substring.
        }
        break;
    }
    case 30: { // We've passed the suffix link.
        state = 28; // Checks length of the active substring.
        break;
    }
    case 31: { // Descent
        if (stack.popBoolean()) {
            state = 33; // Checks if the active substrng if of zero lenght (
                окончание)
        } else {
            state = 22; // Checks if current node has suffix link (окончание)
        }
        break;
    }
    case 32: { // Checks if the active substrng if of zero lenght
        state = 31; // Descent
        break;
    }
    case 33: { // Checks if the active substrng if of zero lenght (окончание)
        if (stack.popBoolean()) {
            state = 34; // Descent step
        } else {
            state = 35; // Descent step
        }
        break;
    }
    case 34: { // Descent step
        state = 32; // Checks if the active substrng if of zero lenght
        break;
    }
    case 35: { // Descent step
        state = 32; // Checks if the active substrng if of zero lenght
        break;
    }
    case 36: { // Checks where we need to append new character.
        state = 31; // Descent
        break;
    }
    case 37: { // Checks where we need to append new character. (окончание)
        if (stack.popBoolean()) {
            state = 38; // We''re in leaf.
        } else {
            state = 40; // Has new leaf been added. (окончание)
        }
        break;
    }
    case 38: { // We''re in leaf.
        state = 36; // Checks where we need to append new character.
        break;
```

```
}
case 39: { // Has new leaf been added.
    state = 36; // Checks where we need to append new character.
    break;
}
case 40: { // Has new leaf been added. (окончание)
    if (stack.popBoolean()) {
        state = 41; // Added new leaf.
    } else {
        state = 43; // Has edge been split. (окончание)
    }
    break;
}
case 41: { // Added new leaf.
    state = 39; // Has new leaf been added.
    break;
}
case 42: { // Has edge been split.
    state = 39; // Has new leaf been added.
    break;
}
case 43: { // Has edge been split. (окончание)
    if (stack.popBoolean()) {
        state = 46; // We should create a new suffix link during next
            step.
    } else {
        state = 47; // Our substring is already in the tree.
    }
    break;
}
case 44: { // We have to split an edge.
    state = 42; // Has edge been split.
    break;
}
case 45: { // New leaf added.
    state = 44; // We have to split an edge.
    break;
}
case 46: { // We should create a new suffix link during next step.
    state = 45; // New leaf added.
    break;
}
case 47: { // Our substring is already in the tree.
    state = 42; // Has edge been split.
    break;
}
case 48: { // Checks if suffix link has been created
    state = 37; // Checks where we need to append new character. (
        окончание)
    break;
}
case 49: { // Checks if suffix link has been created (окончание)
    if (stack.popBoolean()) {
        state = 50; // A suffix link has been created in the tree.
    } else {
        state = 48; // Checks if suffix link has been created
    }
    break;
}
case 50: { // A suffix link has been created in the tree.
    state = 48; // Checks if suffix link has been created
    break;
}
case 51: { // Extension end
    state = 49; // Checks if suffix link has been created (окончание)
    break;
}
case 52: { // Extension loop increment
    state = 51; // Extension end
    break;
}
case 53: { // Phase end
    state = 13; // Цикл
    break;
}
case 54: { // Increment of the phase loop counter
```

```
                state = 53; // Phase end
                break;
            }
            case 55: { // End of the algorithm
                state = 8; // Phase loop
                break;
            }
            case END_STATE: { // Начальное состояние
                state = 55; // End of the algorithm
                break;
            }
        }
    }

    /**
     * Комментарий к текущему состоянию
     */
    public String getComment() {
        String comment = "";
        Object[] args = null;
        // Выбор комментария
        switch (state) {
            case START_STATE: { // Начальное состояние
                comment = Ukkonen.this.getComment("Main.START_STATE");
                break;
            }
            case 1: { // Description of the algorithm
                comment = Ukkonen.this.getComment("Main.Description1");
                break;
            }
            case 2: { // Description of the algorithm
                comment = Ukkonen.this.getComment("Main.Description2");
                break;
            }
            case 3: { // Description of the algorithm
                comment = Ukkonen.this.getComment("Main.Description3");
                break;
            }
            case 4: { // Description of the algorithm
                comment = Ukkonen.this.getComment("Main.Description4");
                break;
            }
            case 5: { // Adding of the sentinel character
                comment = Ukkonen.this.getComment("Main.AddSentinelCharacter");
                args = new Object[]{d.visualizer.SENTINEL};
                break;
            }
            case 6: { // First phase
                comment = Ukkonen.this.getComment("Main.FirstPhase");
                break;
            }
            case 9: { // Begin of the phase
                comment = Ukkonen.this.getComment("Main.PhaseBegin");
                args = new Object[]{new Integer(d.i), new Integer(d.i + 1),
                                                 d.visualizer.getString().substring
                    (0, d.i + 1),                                     d.visualizer.
                    getString().substring(d.i, d.i + 1)};
                break;
            }
            case 10: { // Selecting leaf as current node
                comment = Ukkonen.this.getComment("Main.SelectLeaf");
                break;
            }
            case 11: { // First extension
                comment = Ukkonen.this.getComment("Main.FirstExtension");
                args = new Object[]{new Character(d.visualizer.getString().charAt(d.i
                    )),                                  d.visualizer.getString().
                    substring(0, d.i + 1)};
                break;
            }
            case 14: { // Extension begin
                comment = Ukkonen.this.getComment("Main.ExtensionBegin");
                args = new Object[]{d.visualizer.getString().substring(d.j, d.i + 1)
                    ,                                        new Integer(d.j + 1)};
                break;
            }
```

```
case 19: { // There is no suffix link from current node
    comment = Ukkonen.this.getComment("Main.NoSuffixLink0");
    break;
}
case 20: { // There is no suffix link from current node
    comment = Ukkonen.this.getComment("Main.NoSuffixLink1");
    break;
}
case 25: { // Root node
    comment = Ukkonen.this.getComment("Main.NoSuffixLink3");
    args = new Object[]{d.visualizer.getString().substring(d.i, d.i + 1)
        };
    break;
}
case 26: { // Root node
    comment = Ukkonen.this.getComment("Main.NoSuffixLink2");
    args = new Object[]{d.visualizer.getString().substring(d.j, d.i),
                                                          d.visualizer.
        getString().substring(d.i, d.i + 1)};
    break;
}
case 27: { // Suffix link found
    comment = Ukkonen.this.getComment("Main.FoundSuffixLink");
    break;
}
case 30: { // We've passed the suffix link.
    comment = Ukkonen.this.getComment("Main.SuffixLinkEnd");
    args = new Object[]{d.visualizer.getString().substring(d.j, d.i),
                                                          d.visualizer.
        getString().substring(d.i, d.i + 1)};
    break;
}
case 34: { // Descent step
    comment = Ukkonen.this.getComment("Main.DescentStep1");
    args = new Object[]{d.visualizer.getState().getCurrentNode().
        getString(),                                               d.
        visualizer.getString().substring(d.i, d.i + 1)};
    break;
}
case 35: { // Descent step
    comment = Ukkonen.this.getComment("Main.DescentStep2");
    args = new Object[]{d.visualizer.getState().getCurrentNode().
        getString(),                                               d.
        visualizer.getState().getActiveSubstring()};
    break;
}
case 38: { // We''re in leaf.
    comment = Ukkonen.this.getComment("Main.AppendCharLeaf");
    args = new Object[]{d.visualizer.getString().substring(d.i, d.i + 1)
        };
    break;
}
case 41: { // Added new leaf.
    comment = Ukkonen.this.getComment("Main.AddNewLeaf");
    args = new Object[]{d.visualizer.getString().substring(d.i, d.i + 1)
        };
    break;
}
case 44: { // We have to split an edge.
    comment = Ukkonen.this.getComment("Main.SplittingEdge");
    args = new Object[]{d.visualizer.getString().substring(d.i, d.i + 1)
        ,                                                            d.
        visualizer.getState().getActiveSubstring(),
                                                                   d.
        visualizer.getState().getCurrentNode().getString().substring(d.
        visualizer.getState().getActiveSubstring().length(), d.visualizer
        .getState().getCurrentNode().getString().length())};
    break;
}
case 45: { // New leaf added.
    comment = Ukkonen.this.getComment("Main.SplitEdgeAddLeaf");
    args = new Object[]{d.visualizer.getString().substring(d.i, d.i + 1)
        };
    break;
}
case 46: { // We should create a new suffix link during next step.
```

```
                comment = Ukkonen.this.getComment("Main.SuffixLinkOnNextStep");
                break;
            }
            case 47: { // Our substring is already in the tree.
                comment = Ukkonen.this.getComment("Main.AlreadyInTree");
                break;
            }
            case 50: { // A suffix link has been created in the tree.
                comment = Ukkonen.this.getComment("Main.LinkHasBeenCreated");
                break;
            }
            case 51: { // Extension end
                comment = Ukkonen.this.getComment("Main.ExtensionEnd");
                args = new Object[]{d.visualizer.getString().substring(d.j, d.i + 1)
                    };
                break;
            }
            case 53: { // Phase end
                comment = Ukkonen.this.getComment("Main.PhaseEnd");
                args = new Object[]{new Integer(d.i)};
                break;
            }
            case END_STATE: { // Конечное состояние
                comment = Ukkonen.this.getComment("Main.END_STATE");
                break;
            }
        }

        return java.text.MessageFormat.format(comment, args);
    }

    /**
     * Выполняет действия по отрисовке состояния
     */
    public void drawState() {
        switch (state) {
            case START_STATE: { // Начальное состояние
                break;
            }
            case 5: { // Adding of the sentinel character
                d.visualizer.updateTree();
                break;
            }
            case 6: { // First phase
                d.visualizer.updateTree();
                break;
            }
            case 9: { // Begin of the phase
                d.visualizer.updateTree();
                break;
            }
            case 10: { // Selecting leaf as current node
                d.visualizer.updateTree();
                break;
            }
            case 11: { // First extension
                d.visualizer.updateTree();
                break;
            }
            case 14: { // Extension begin
                d.visualizer.updateTree();
                break;
            }
            case 19: { // There is no suffix link from current node
                d.visualizer.updateTree();
                break;
            }
            case 20: { // There is no suffix link from current node
                d.visualizer.updateTree();
                break;
            }
            case 25: { // Root node
                d.visualizer.updateTree();
                break;
            }
            case 26: { // Root node
```

```
                d.visualizer.updateTree();
                break;
            }
            case 27: { // Suffix link found
                d.visualizer.updateTree();
                break;
            }
            case 30: { // We've passed the suffix link.
                d.visualizer.updateTree();
                break;
            }
            case 34: { // Descent step
                d.visualizer.updateTree();
                break;
            }
            case 35: { // Descent step
                d.visualizer.updateTree();
                break;
            }
            case 38: { // We''re in leaf.
                d.visualizer.updateTree();
                break;
            }
            case 41: { // Added new leaf.
                d.visualizer.updateTree();
                break;
            }
            case 44: { // We have to split an edge.
                d.visualizer.updateTree();
                break;
            }
            case 45: { // New leaf added.
                d.visualizer.updateTree();
                break;
            }
            case 46: { // We should create a new suffix link during next step.
                d.visualizer.updateTree();
                break;
            }
            case 47: { // Our substring is already in the tree.
                d.visualizer.updateTree();
                break;
            }
            case 50: { // A suffix link has been created in the tree.
                d.visualizer.updateTree();
                break;
            }
            case 51: { // Extension end
                d.visualizer.updateTree();
                break;
            }
            case 53: { // Phase end
                d.visualizer.updateTree();
                break;
            }
            case END_STATE: { // Конечное состояние
                d.visualizer.updateTree();
                break;
            }
        }
    }
  }
}
```

Приложение 3. Исходные коды визуализатора

ru.ifmo.vizi.ukkonen.UkkonenVisualizer

```
package ru.ifmo.vizi.ukkonen;

import ru.ifmo.vizi.base.ui.*;
import ru.ifmo.vizi.base.*;
import ru.ifmo.vizi.base.widgets.ShapeStyle;
```

```java
import ru.ifmo.vizi.ukkonen.widgets.StringVisualizer;
import ru.ifmo.vizi.ukkonen.widgets.VerticalStringVisualizer;
import ru.ifmo.vizi.ukkonen.ui.HintedChoice;
import ru.ifmo.vizi.ukkonen.ui.HintedTextField;
import ru.ifmo.vizi.ukkonen.ui.HintedCheckbox;
import ru.ifmo.vizi.ukkonen.SuffixTree.SuffixTree;
import ru.ifmo.vizi.ukkonen.SuffixTree.SuffixTreeState;

import java.awt.*;
import java.awt.event.*;
import java.util.Vector;

/**
 * Ukkonen applet.
 *
 * @author  Igor Ahmetov
 * @version $Id$
 */
public final class UkkonenVisualizer extends Base {
    /**
     * Suffix tree sentinel.
     */
    public static final String SENTINEL = "$";

    /**
     * Ukkonen automata instance.
     */
    private final Ukkonen auto;

    /**
     * Ukkonen automata data.
     */
    private final Ukkonen.Data data;

    /**
     * Save/load dialog.
     */
    private SaveLoadDialog saveLoadDialog;

    /**
     * Predefined examples.
     */
    private Vector examples = new Vector();

    /**
     * String visualizator.
     */
    private StringVisualizer stringVis;

    /**
     * Active string visualizer.
     */
    protected VerticalStringVisualizer activeStringVis;


    /**
     * Suffix tree.
     */
    private SuffixTree suffixTree;

    /**
     *
     */
    protected ClientAdapter clientAdapter;

    /**
     * Boolean variable that is <code>true</code> if the string field has focus.
     */
    protected boolean stringFieldHasFocus = false;

    /**
     * Component with focus.
     */
    protected Component focusComponent;
```

```java
/**
 * Creates a new Ukkonen visualizer.
 *
 * @param parameters visualizer parameters.
 */
public UkkonenVisualizer(VisualizerParameters parameters) {
    super(parameters);
    auto = new Ukkonen(locale);
    data = auto.d;
    data.visualizer = this;

    clientPane.addComponentListener(
            clientAdapter = new ClientAdapter(config.getDouble("string-visualizer-
                height", 0.2),
                                config.getDouble("string-visualizer-width", 0.1))
    );

    stringVis = new StringVisualizer(
            this, ShapeStyle.loadStyleSet(config, "string-visualizer")
    );
    activeStringVis = new VerticalStringVisualizer(
            this, ShapeStyle.loadStyleSet(config, "string-visualizer"), false
    );
    suffixTree = new SuffixTree(this, "", ShapeStyle.loadStyleSet(config, "suffix-
        tree"));

    clientPane.setLayout(new BorderLayout());
    clientPane.add(suffixTree, BorderLayout.CENTER);
    clientPane.add(activeStringVis, BorderLayout.EAST);
    clientPane.add(stringVis, BorderLayout.SOUTH);

    createInterface(auto);
}

/**
 * This method creates panel with visualizer controls.
 *
 * @return controls pane.
 */
public Component createControlsPane() {
    Panel panel = new Panel(new BorderLayout());
    Panel bottomPanel = new Panel(new FlowLayout());

    panel.add(new AutoControlsPane(config, auto, forefather, true), BorderLayout.
        CENTER);

    HintedCheckbox stringCheckbox = new HintedCheckbox(config, "string-checkbox");
    stringCheckbox.setState(true);
    stringCheckbox.addItemListener(
            new ItemListener() {
                public void itemStateChanged(ItemEvent e) {
                    stringVis.setVisible(e.getStateChange() == ItemEvent.SELECTED);
                    clientPane.validate();
                    updateTree();
                }
            }
    );
    bottomPanel.add(stringCheckbox);

    HintedTextField stringField = new HintedTextField(config, "string-field", "", 24)
        ;
    stringField.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    setString(((HintedTextField) e.getSource()).getText());
                }
            }
    );
    stringField.addKeyListener(
            new KeyAdapter() {
                public void keyPressed(KeyEvent e) {
                    char ch = e.getKeyChar();
                    if (!(Character.isLetter(ch)
                            || Character.getType(ch) == Character.CONTROL
                            || Character.getType(ch) == Character.FORMAT
                            || Character.getType(ch) == Character.MODIFIER_LETTER
```

```
                    || Character.getType(ch) == Character.MODIFIER_SYMBOL
                    || Character.getType(ch) == Character.UNASSIGNED))
                e.consume();

            }
        }
);
stringField.addFocusListener(
        new FocusListener() {
            public void focusGained(FocusEvent e) {
                stringFieldHasFocus = true;
            }

            public void focusLost(FocusEvent e) {
                stringFieldHasFocus = false;
            }
        }
);
bottomPanel.add(stringField);

final HintedChoice examplesChoice = new HintedChoice(config, "examples");
examplesChoice.addItemListener(new ItemListener() {
    public void itemStateChanged(ItemEvent e) {
        setString((String) examples.elementAt(examplesChoice.getSelectedIndex()))
            ;
    }
});

int num = config.getInteger("examples-num", 0);
for (int i = 0; i < num; i++) {
    examplesChoice.add(config.getParameter("example-" + (i + 1) + "-title", "
        Example " + i));
    examples.addElement(config.getParameter("example-" + (i + 1), ""));
}
setString((String) examples.elementAt(0));

bottomPanel.add(examplesChoice);

if (config.getBoolean("button-ShowSaveLoad")) {
    bottomPanel.add(new HintedButton(config, "button-SaveLoad") {
        protected void click() {
            saveLoadDialog.center();
            StringBuffer buffer = new StringBuffer();
            buffer.append("/* ").append(
                    config.getParameter("save-string-comment")
            ).append(" */\n");

            String str = getString();
            buffer.append("string = " + str.substring(0, str.length() - 1));

            buffer.append("\n/* ").append(
                    config.getParameter("save-step-comment")
            ).append(" */\n");
            buffer.append("step = ").append(auto.getStep());
            saveLoadDialog.show(buffer.toString());
        }
    });
}
panel.add(bottomPanel, BorderLayout.SOUTH);

saveLoadDialog = new SaveLoadDialog(config, forefather) {
    public boolean load(String text) throws Exception {
        SmartTokenizer tokenizer = new SmartTokenizer(text, config);
        tokenizer.expect("string");
        tokenizer.expect("=");
        setString(tokenizer.nextWord());

        tokenizer.expect("step");
        tokenizer.expect("=");
        rewind(tokenizer.nextInt());

        tokenizer.expectEOF();

        return true;
    }
};
```

```java
        return panel;
    }

    /**
     * Sets new string for the suffix tree.
     *
     * @param string new string.
     */
    public void setString(String string) {
        rewind(0);
        suffixTree.setString(string + SENTINEL);
        stringVis.setString(string);
        //activeStringVis.setVisible(false);
    }

    /**
     * Rewinds algorithm to the specified step.
     *
     * @param step spet of the algorith to rewind to.
     */
    private void rewind(int step) {
        auto.toStart();

        while (!auto.isAtEnd() && auto.getStep() < step) {
            auto.stepForward(0);
        }
    }

    /**
     * Returns suffix tree string.
     *
     * @return suffix tree string.
     */
    public String getString() {
        return suffixTree.getString();
    }

    /**
     * Performs next step.
     */
    public void nextStep() {
        suffixTree.nextStep();
    }

    /**
     * Returns to previous step.
     */
    public void previousStep() {
        suffixTree.previousStep();
    }

    /**
     * Updates string visualizers.
     */
    protected void updateStrings() {
        stringVis.highlight(suffixTree.getExtension(), suffixTree.getPhase() + 2);

        SuffixTreeState state = suffixTree.getState();
        String str = state.getActiveSubstring();
        if (state.getSpecialState() == SuffixTreeState.STATE_DESCENT) {
            str = state.getCurrentNode().getString() + str;
        }

        activeStringVis.setString(str, false);
        if (state.getSpecialState() == SuffixTreeState.STATE_DESCENT) {
                activeStringVis.highlight(0, state.getCurrentNode().getString().length())
                    ;
        }
        /*if (str.length() == 0) {
            if (activeStringVis.isVisible()) {
                activeStringVis.setVisible(false);
                clientPane.validate();
            }
        } else {
            boolean visible = activeStringVis.isVisible();
```

```java
        activeStringVis.setVisible(true);

        if (!visible) clientPane.validate();
    }*/
}
/**
 * Returns suffix tree.
 *
 * @return current suffix tree.
 */
public SuffixTree getSuffixTree() {
    return suffixTree;
}

/**
 * Selects part of the string.
 *
 * @param beginIndex begin index.
 * @param endIndex end index.
 */
public void stringSelect(int beginIndex, int endIndex) {
    stringVis.select(beginIndex, endIndex);
}

/**
 * Updates the suffix tree.
 */
public void updateTree() {
    suffixTree.updateTree();
    updateStrings();
}

/**
 * Adds sentinel character to string visualizer string.
 */
public void addSentinelCharacter() {
    stringVis.setString(suffixTree.getString());
}

/**
 * Returns current algorithm state.
 *
 * @return current algorithm state.
 */
public SuffixTreeState getState() {
    return suffixTree.getState();
}

/**
 * Returns next algorithm state.
 * @return next algorithm state.
 */
public SuffixTreeState getNextState() {
    return suffixTree.getNextState();
}

/**
 * Removes sentinel character from string visualizer string.
 */
public void removeSentinelCharacter() {
    String str = suffixTree.getString();
    stringVis.setString(str.substring(0, str.length() - 1));
}

/**
 * Validates this container.
 */
public void resizeStringVisualizers() {
        if (clientPane.getSize().height != 0) {
            clientAdapter.setStringVisualizerHeight(
                    (double) stringVis.getSize().height / clientPane.getSize().height
            );
    }
    if (clientPane.getSize().width != 0) {
            clientAdapter.setStringVisualizerWidth(
```

```
                        (double) activeStringVis.getSize().width / clientPane.getSize().
                            width
                );
            }
        }

        /**
         * Returns some information about the applet.
         *
         * @return string containing short description of the applet.
         */
        public String getAppletInfo() {
            return "Visualizer of the Ukkonen Algorithm. Igor Ahmetov, CTD IFMO, 2004";
        }


        /**
         * Component adapter that handles resizing of the client pane.
         */
        class ClientAdapter extends ComponentAdapter {
            /**
             * Height of the string visualizer.
             */
            protected double stringVisHeight;

            /**
             * Width of the vertical string visualizer.
             */
            protected double stringVisWidth;

            /**
             * Default constructor.
             *
             * @param height height of the string visualizer
             */
            public ClientAdapter(double height, double width) {
                this.stringVisHeight = height;
                this.stringVisWidth = width;
            }

            /**
             * Sets height of the horizontal string visualizer.
             *
             * @param height new height of the visualizer.
             */
            public void setStringVisualizerHeight(double height) {
                    stringVisHeight = height;
            }

            /**
             * Sets width of the vertical string visualizer.
             *
             * @param width new width of the visualizer.
             */
            public void setStringVisualizerWidth(double width) {
                    stringVisWidth = width;
            }

            /**
             * Deals with resizing of the clientPane.
             * @param e component event
             */
            public void componentResized(ComponentEvent e) {
                stringVis.setSize(200,
                        (int) (e.getComponent().getSize().height * stringVisHeight));
                stringVis.resize();
                activeStringVis.setSize(
                        (int) (e.getComponent().getSize().width * stringVisWidth), 200);
                activeStringVis.resize();
                e.getComponent().validate();
            }
        }
    }
```

ru.ifmo.vizi.ukkonen.suffixtree.Edge

```java
package ru.ifmo.vizi.ukkonen.SuffixTree;

import ru.ifmo.vizi.base.widgets.Shape;
import ru.ifmo.vizi.base.widgets.ShapeStyle;

import java.awt.*;

/**
 * Edge class.
 *
 * @author  Igor Ahmetov
 * @version $Id$
 */

public class Edge extends Shape {
    static public double arcHeight = 0.15;

    /**
     * Arrow height.
     */
    static public double arrowHeight = 10;

    /**
     * Arrow width.
     */
    static public double arrowWidth = 4;

    /**
     * Beginning of the edge.
     */
    protected NodeShape from;

    /**
     * End of the edge.
     */
    protected NodeShape to;

    /**
     * Stores <code>true</code> if this edge has acquired the edge lock.
     */
    protected boolean hasLock = false;

    protected boolean hasArrow;

    protected boolean hasReverseArrow;

    /**
     * Default constructor.
     *
     * @param styleSet edge style set.
     * @param from beginning of the edge.
     * @param to end of the edge.
     */
    public Edge(ShapeStyle[] styleSet, NodeShape from, NodeShape to) {
        super(styleSet);

        this.from = from;
        this.to = to;
    }

    /**
     * Returns source node of this edge.
     *
     * @return source node of this edge.
     */
    public NodeShape getSourceNodeShape() {
        return from;
    }

    /**
     * Returns target node of this edge.
     *
     * @return target node of this edge.
     */
    public NodeShape getTargetNodeShape() {
        return to;
```

```java
}

/**
 * Checks if the edge contains the specified point.
 *
 * @param x x coordinate of the point.
 * @param y y coordinate of the point.
 * @return <code>true</code> if the edge contains the
 *         specified point.
 */
public boolean containsPoint(int x, int y) {
    Rectangle fromBounds = from.getBounds();
    Rectangle toBounds = to.getBounds();
    int x1 = fromBounds.x + fromBounds.width / 2;
    int y1 = fromBounds.y + fromBounds.height / 2;
    int x2 = toBounds.x + toBounds.width / 2;
    int y2 = toBounds.y + toBounds.height / 2;

    int a = y1 - y2, b = x2 - x1, c = -a * x1 - b * y1;
    double d = Math.abs(a * x + b * y + c) / Math.sqrt(a * a + b * b);

    return (d <= 8 && x >= Math.min(x1, x2) - d && x <= Math.max(x1, x2) + d
            && y >= Math.min(y1, y2) - d && y <= Math.max(y1, y2) + d);
}

/**
 * Checks if this <code>Edge</code> object has the edge lock.
 *
 * @return <code>true</code> if this edge has the lock.
 */
public boolean hasLock() {
    return hasLock;
}

/**
 * Acquires the edge lock.
 *
 * @return <code>true</code> if the lock has been acquired.
 */
public boolean acquireLock() {
    if (hasLock) return false;

    SuffixTree tree = (SuffixTree) getParent();
    if (tree == null) return false;
    hasLock = tree.acquireLock(this);
    return hasLock;
}

/**
 * Releases the edge lock.
 *
 * @return <code>true</code> if the lock has been released.
 */
public boolean releaseLock() {
    if (!hasLock) return false;

    SuffixTree tree = (SuffixTree) getParent();
    if (tree == null) return false;
    tree.releaseLock(this);
    hasLock = false;
    return true;
}

protected Dimension fit(Dimension size) {
    return size;
}

public void setArrow(boolean hasArrow) {
    this.hasArrow =  hasArrow;
}

public void setReverseArrow(boolean hasReverseArrow) {
    this.hasReverseArrow = hasReverseArrow;
}

protected double getAngle(double x, double y) {
```

```
        double a = (x == 0) ? (y >= 0 ? Math.PI / 2 : Math.PI * 3 / 2) :
                Math.atan(y / x);
        if (x < 0) a += Math.PI;
        return a;
    }


    /**
     * Paints this edge.
     *
     * @param g graphics context for painting.
     */
    public void paint(Graphics g) {
        if (look == null || g == null || to == null || from == null) return;

        Rectangle fromBounds = from.getBounds();
        if (fromBounds == null) return;

        Rectangle toBounds = to.getBounds();
        if (toBounds == null) return;

        int x1 = fromBounds.x + fromBounds.width / 2;
        int y1 = fromBounds.y + fromBounds.height / 2;
        int x2 = toBounds.x + toBounds.width / 2;
        int y2 = toBounds.y + toBounds.height / 2;

        if (look.getBorderStatus(style)) {
            g.setColor(look.getBorderColor(style));
            g.drawLine(x1, y1, x2, y2);
        }

        if (hasArrow) {
            double angle = getAngle(x2 - x1, y2 - y1) + Math.PI;
            double r = Math.min(toBounds.width / 2, toBounds.height / 2);

            int x[] = new int[3], y[] = new int[3];
            x[0] = x2 + (int) Math.round(r * Math.cos(angle));
            y[0] = y2 + (int) Math.round(r * Math.sin(angle));
            x[1] = x[0] + (int) Math.round(arrowHeight * Math.cos(angle) + arrowWidth *
                Math.sin(angle));
            y[1] = y[0] + (int) Math.round(arrowHeight * Math.sin(angle) - arrowWidth *
                Math.cos(angle));
            x[2] = x[0] + (int) Math.round(arrowHeight * Math.cos(angle) - arrowWidth *
                Math.sin(angle));
            y[2] = y[0] + (int) Math.round(arrowHeight * Math.sin(angle) + arrowWidth *
                Math.cos(angle));

            if (look.getFillStatus(style)) {
                g.setColor(look.getFillColor(style));
                g.fillPolygon(x, y, 3);
            }
            if (look.getBorderStatus(style)) {
                g.setColor(look.getBorderColor(style));
                g.drawPolygon(x, y, 3);
            }
        }

        if (hasReverseArrow) {
            double angle = getAngle(x1 - x2, y1 - y2) + Math.PI;
            double r = Math.min(fromBounds.width / 2, fromBounds.height / 2);

            int x[] = new int[3], y[] = new int[3];
            x[0] = x1 + (int) Math.round(r * Math.cos(angle));
            y[0] = y1 + (int) Math.round(r * Math.sin(angle));
            x[1] = x[0] + (int) Math.round(arrowHeight * Math.cos(angle) + arrowWidth *
                Math.sin(angle));
            y[1] = y[0] + (int) Math.round(arrowHeight * Math.sin(angle) - arrowWidth *
                Math.cos(angle));
            x[2] = x[0] + (int) Math.round(arrowHeight * Math.cos(angle) - arrowWidth *
                Math.sin(angle));
            y[2] = y[0] + (int) Math.round(arrowHeight * Math.sin(angle) + arrowWidth *
                Math.cos(angle));

            if (look.getFillStatus(style)) {
                g.setColor(look.getFillColor(style));
                g.fillPolygon(x, y, 3);
```

```java
            }
            if (look.getBorderStatus(style)) {
                g.setColor(look.getBorderColor(style));
                g.drawPolygon(x, y, 3);
            }
        }

        if (to.getNode() == null) return;
        String str = to.getNode().getString();
        if (str == null) return;

        int edgeWidth = Math.abs(x1 - x2);
        int edgeHeight = Math.abs(y1 - y2);

        int fontSize = Math.min(30, Math.max(edgeHeight / (str.length() + 3),
                                    edgeWidth / 18));

        if (fontSize < 12 && to.getNode().getString().length() > 7) {
                String nodeString = to.getNode().getString();
                int len = nodeString.length();

            str = "" + nodeString.charAt(0) + nodeString.charAt(1) + ".." +
                        nodeString.charAt(len - 2) + nodeString.charAt(len - 1);

            fontSize = Math.min(30, Math.max(edgeHeight / (str.length() + 3),
                                    edgeWidth / 18));
        }

        if (fontSize < 8) return;

        if (getFont() == null) return;
        Font font = new Font(getFont().getName(), getFont().getStyle(), fontSize);
        if (font == null) return;
        g.setFont(font);

        for (int i = 0; i < str.length(); i++) {
            double newx, newy;
            double k = 0.025;
            double x = x1 + (x2 - x1) * (i + 2) / (str.length() + 3);
            double y = y1 + (y2 - y1) * (i + 2) / (str.length() + 3);

            int a = x2 - x1, b = y2 - y1;

            Container parent = getParent();
            if (parent == null || parent.getSize() == null) return;
            int size = Math.min(parent.getSize().width,
                        parent.getSize().height);

            FontMetrics fm = g.getFontMetrics();
            if (fm == null) return;
            int charHeight = fm.getAscent();
            int charWidth = fm.charWidth(str.charAt(i));

            if (x2 >= x1) {
                newx = x + k * b / Math.sqrt(a * a + b * b) * size;
                newy = y - k * a / Math.sqrt(a * a + b * b) * size;
                g.drawString(str.substring(i, i + 1),
                                (int) Math.round(newx), (int)Math.round(newy + (double)
                                    charHeight / 3));
            } else {
                newx = x - k * b / Math.sqrt(a * a + b * b) * size;
                newy = y + k * a / Math.sqrt(a * a + b * b) * size;
                g.drawString(str.substring(i, i + 1),
                                (int) Math.round(newx - charWidth),
                        (int) Math.round(newy + (double) charHeight / 3));
            }
        }
    }

    /**
     * Paints hint of this edge on specified <code>Graphics</code> object.
     *
     * @param g graphics context for painting.
     */
    public void paintHint(Graphics g) {
        if (g == null) return;
```

```
        if (hasLock) {
            if (getSize() == null || getParent() == null || getFont() == null) return;
            Font font = new Font(getFont().getName(), getFont().getStyle(),
                                 (int) Math.round(0.08 * getParent().getSize().height));
            if (font == null) return;
            g.setFont(font);

            if (to.getNode() == null) return;
            String str = to.getNode().getString();

            FontMetrics fm = g.getFontMetrics();
            if (fm == null) return;
            int charHeight = fm.getAscent();
            int stringWidth = fm.stringWidth(str);

            Point mousePoint = ((SuffixTree) getParent()).getMousePoint();
            if (mousePoint == null) return;

            int x = mousePoint.x, y = mousePoint.y;
            int width = stringWidth + 10, height = charHeight + 5;

            if (y - height < 0) {
                x -= width;
                y += height;
            } else
                if (x + width > getSize().width)
                    x -= width;

            g.clearRect(x, y - height, width, height);
            g.drawLine(x, y, x, y - height);
            g.drawLine(x, y - height, x + width, y - height);
            g.drawLine(x + width, y - height, x + width, y);
            g.drawLine(x + width, y, x, y);

            g.drawString(str, x + 5, y - 5);
        }
    }
}
```

ru.ifmo.vizi.ukkonen.suffixtree.NodeShape

```
package ru.ifmo.vizi.ukkonen.SuffixTree;

import ru.ifmo.vizi.base.widgets.Ellipse;
import ru.ifmo.vizi.base.widgets.ShapeStyle;

import java.awt.*;

/**
 * Component that represents suffix tree node.
 *
 * @author       Igor Ahmetov
 * @version      $Id$
 */
public class NodeShape extends Ellipse {
    /**
     * Parent container.
     */
    protected SuffixTree tree;

    /**
     * Corresponding node in the suffix tree.
     */
    protected SuffixTreeNode node;

    /**
     * Depth of this node in the suffix tree.
     */
    protected int depth;

    /**
     * Horizontal position of this node in the suffix tree.
     */
    protected int hpos;
```

```java
/**
 * Constructs new <code>NodeShape</code> object.
 *
 * @param styleSet node style set.
 * @param tree parent container.
 * @param node corresponding node in the suffix tree.
 */
public NodeShape(ShapeStyle[] styleSet, SuffixTree tree, SuffixTreeNode node) {
    this(styleSet, tree, node, "");
}

/**
 * Constructs new <code>NodeShape</code> object with
 * specified message on it.
 *
 * @param styleSet node style set.
 * @param tree parent container.
 * @param node corresponding node in the suffix tree.
 * @param message message on the node.
 */
public NodeShape(ShapeStyle[] styleSet, SuffixTree tree, SuffixTreeNode node,
                 String message) {
    super(styleSet, message);

    this.tree = tree;
    this.node = node;
}

/**
 * Returns component that represents this nodes parent in the suffix tree.
 *
 * @return component.
 */
public NodeShape getParentShape() {
    return tree.getNodeShape(node.getParent());
}

/**
 * Returns component that represents this nodes leftmost child in the suffix tree.
 *
 * @return component.
 */
public NodeShape getLeftChildShape() {
    return tree.getNodeShape(node.getLeftChild());
}

/**
 * Returns component that represents this nodes right sibling in the suffix tree.
 *
 * @return component.
 */
public NodeShape getRightSiblingShape() {
    return tree.getNodeShape(node.getRightSibling());
}

/**
 * Returns tree node associated with this shape.
 *
 * @return tree node.
 */
public SuffixTreeNode getNode() {
    return node;
}

/**
 * Returns depth in the suffix tree of this node.
 *
 * @return depth in the suffix tree.
 */
public int getDepth() {
    return depth;
}

/**
 * Returns horizontal position of this node.
 *
```

```
     * @return horizontal position of this node.
     */
    public int getHorizontalPos() {
        return hpos;
    }

    /**
     * Checks if the node contains the specified point.
     *
     * @param point point to check.
     * @return <code>true</code>, if node contains the specified point.
     */
    public boolean containsPoint(Point point) {
        int x = getLocation().x + getSize().width / 2;
        int y = getLocation().y + getSize().height / 2;
        int r = Math.min(getSize().width / 2, getSize().height / 2);

        return (point.x - x) * (point.x - x) +
               (point.y - y) * (point.y - y) <= r * r;
    }

    /**
     * Calculates depth of the subtree with this node as root
     * assuming that this node has depth <code>curDepth</code>
     *
     * @param curDepth depth of this node.
     */
    public void calculateDepth(int curDepth) {
        depth = curDepth;

        NodeShape node = getLeftChildShape();

        while (node != null) {
            node.calculateDepth(depth + 1);
            node = node.getRightSiblingShape();
        }
    }

    public void calculateHorizontalPos() {
        hpos = tree.addElementAtDepth(depth);

        NodeShape node = getLeftChildShape();

        while (node != null) {
            node.calculateHorizontalPos();
            node = node.getRightSiblingShape();
        }
    }
}
```

ru.ifmo.vizi.ukkonen.suffixtree.Options

```
package ru.ifmo.vizi.ukkonen.SuffixTree;

public class Options {
    public final static boolean DEBUG = false;
}
```

ru.ifmo.vizi.ukkonen.suffixtree.SuffixLink

```
package ru.ifmo.vizi.ukkonen.SuffixTree;

import java.lang.Math;
import ru.ifmo.vizi.base.widgets.Shape;
import ru.ifmo.vizi.base.widgets.ShapeStyle;

import java.awt.*;

/**
 * Suffix links class.
 *
 * @author  Igor Ahmetov
 * @version $Id$
 */
```

```java
public class SuffixLink extends Shape {
    static public double arcHeight = 0.15;

    /**
     * Arrow height.
     */
    static public double arrowHeight = 10;

    /**
     * Arrow width.
     */
    static public double arrowWidth = 4;

    /**
     * Beginning of the suffix link.
     */
    protected NodeShape from;

    /**
     * End of the suffix link.
     */
    protected NodeShape to;

    /**
     * Default constructor.
     *
     * @param styleSet style set.
     * @param from beginning of the suffix link.
     * @param to end of the suffix link.
     */
    public SuffixLink(ShapeStyle[] styleSet, NodeShape from, NodeShape to) {
        super(styleSet);

        this.from = from;
        this.to = to;
    }

    /**
     * Returns square of the argument.
     *
     * @param x argument.
     * @return square of <code>x</code>.
     */
    protected double sqr(double x) {
        return x * x;
    }

    /**
     * Returns source node of this suffix link.
     *
     * @return source node of this suffix link.
     */
    public NodeShape getSourceNodeShape() {
        return from;
    }

    /**
     * Returns target node of this suffix link.
     *
     * @return target node of this suffix link.
     */
    public NodeShape getTargetNodeShape() {
        return to;
    }

    protected double getAngle(double x, double y) {
        double a = (x == 0) ? (y >= 0 ? Math.PI / 2 : Math.PI * 3 / 2) :
                Math.atan(y / x);
        if (x < 0) a += Math.PI;
        return a;
    }

    protected Dimension fit(Dimension size) {
        Rectangle fromBounds = from.getBounds();
        Rectangle toBounds = to.getBounds();
```

```
        int x1 = fromBounds.x + fromBounds.width / 2;
        int y1 = fromBounds.y + fromBounds.height / 2;
        int x2 = toBounds.x + toBounds.width / 2;
        int y2 = toBounds.y + toBounds.height / 2;

        return new Dimension(Math.abs(x1 - x2), Math.abs(y1 - y2));
    }

    public void paint(Graphics g) {
        Rectangle fromBounds = from.getBounds();
        Rectangle toBounds = to.getBounds();
        int x1 = fromBounds.x + fromBounds.width / 2;
        int y1 = fromBounds.y + fromBounds.height / 2;
        int x2 = toBounds.x + toBounds.width / 2;
        int y2 = toBounds.y + toBounds.height / 2;

        double a = y1 - y2, b = x2 - x1;
        double angle = getAngle(a, b);
        double l = Math.sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
        double arc = arcHeight * l;
        double r = (4 * sqr(arc) + sqr(l)) / (8 * arc);
        double xPos = (x1 + x2) / 2 + (r - arc) * Math.cos(angle);
        double yPos = (y1 + y2) / 2 + (r - arc) * Math.sin(angle);
        double startAngle = getAngle(x1 - xPos, y1 - yPos) / Math.PI * 180;
        double endAngle = getAngle(x2 - xPos, y2 - yPos) / Math.PI * 180;
        if (endAngle < startAngle) endAngle += 360;

        if (look.getBorderStatus(style)) {
            g.setColor(look.getBorderColor(style));
            g.drawArc((int) Math.round(xPos - r), (int) Math.round(yPos - r),
                    (int) Math.round(2 * r), (int) Math.round(2 * r),
                    -(int) Math.round(startAngle), -(int) Math.round((endAngle -
                        startAngle)));
        }

        angle = getAngle(x2 - xPos, y2 - yPos) - Math.PI / 2;

        r = toBounds.width / 2;

        int x[] = new int[3], y[] = new int[3];
        x[0] = x2 + (int) Math.round(r * Math.cos(angle));
        y[0] = y2 + (int) Math.round(r * Math.sin(angle));
        x[1] = x[0] + (int) Math.round(arrowHeight * Math.cos(angle) + arrowWidth * Math.
            sin(angle));
        y[1] = y[0] + (int) Math.round(arrowHeight * Math.sin(angle) - arrowWidth * Math.
            cos(angle));
        x[2] = x[0] + (int) Math.round(arrowHeight * Math.cos(angle) - arrowWidth * Math.
            sin(angle));
        y[2] = y[0] + (int) Math.round(arrowHeight * Math.sin(angle) + arrowWidth * Math.
            cos(angle));

        if (look.getFillStatus(style)) {
            g.setColor(look.getFillColor(style));
            g.fillPolygon(x, y, 3);
        }

        if (look.getBorderStatus(style)) {
            g.setColor(look.getBorderColor(style));
            g.drawPolygon(x, y, 3);
        }
    }
}
```

ru.ifmo.vizi.ukkonen.suffixtree.SuffixTree

```
package ru.ifmo.vizi.ukkonen.SuffixTree;

import ru.ifmo.vizi.base.widgets.ShapeStyle;
import ru.ifmo.vizi.ukkonen.UkkonenVisualizer;

import java.awt.*;
import java.awt.event.MouseMotionAdapter;
import java.awt.event.MouseEvent;
import java.util.Hashtable;
import java.util.Vector;
```

```java
/**
 * Suffix tree visualizer.
 *
 * @author  Igor Ahmetov
 * @version     $Id$
 */
public class SuffixTree extends Panel {
        /**
         * Suffix tree.
         */
        protected SuffixTreeData tree;

    /**
     * Shape styles.
     */
    protected ShapeStyle[] styleSet;

    /**
     * Hashtable for storing nodes of the tree.
     */
    protected Hashtable nodes = new Hashtable();

    /**
     * Current algorithm state.
     */
    protected SuffixTreeState state;

    /**
     * Stores number of elements on different levels.
     */
    protected Vector elemCount = new Vector();

    /**
     * Edge lock.
     */
    protected boolean edgeLock = false;

    /**
     * Edge that has acquired the lock.
     */
    protected Edge lockedEdge;

    /**
     * Used for double-buffering.
     */
    protected Image buffer;

    /**
     * Stores edges component of this tree.
     */
    protected Vector edges = new Vector();

    /**
     * Stores mouse cursor coordinates.
     */
    protected Point mousePoint = new Point(0, 0);

    /**
     * Parent applet.
     */
    protected UkkonenVisualizer applet;

        /**
         * Default constructor.
         *
         * @param string        string of suffix tree.
         */
        public SuffixTree(UkkonenVisualizer applet,
                     String string, ShapeStyle[] styleSet) {
                super(new SuffixTreeLayout());

        this.applet = applet;
        this.styleSet = styleSet;
                initTree(string);
```

```
        addMouseMotionListener(new TreeMouseMotionAdapter());
        }

    /**
     * Returns current string.
     */
    public String getString() {
        return tree.getString();
    }

    /**
     * Sets new string.
     */
    public void setString(String string) {
        removeAll();
        initTree(string);
        repaint();
    }

    /**
     * Initializes tree.
     *
     * @param string suffix tree string.
     */
    private void initTree(String string) {
        tree = new SuffixTreeData(string);
        state = tree.getState();
    }

    /**
     * Performs next step.
     */
    public void nextStep() {
        if (Options.DEBUG) System.out.println("SuffixTree.nextStep()");
        tree.nextStep();
        state = tree.getState();
        if (Options.DEBUG) System.out.println("New state:\n" + state);
    }

    /**
     * Returns to previous step.
     */
    public void previousStep() {
        if (Options.DEBUG) System.out.println("SuffixTree.previousStep()");
        tree.previousStep();
        state = tree.getState();
        if (Options.DEBUG) System.out.println("New state:\n" + state);
    }

    /**
     * Returns current algorithm state.
     *
     * @return current algorithm state.
     */
    public SuffixTreeState getState() {
        return state;
    }

    /**
     * Returns algorithm state during next step.
     *
     * @return algorithm state during next step.
     */
    public SuffixTreeState getNextState() {
        return tree.getNextState();
    }

    /**
     * Returns current extension.
     *
     * @return current extension.
     */
    public int getExtension() {
        return state.getExtension();
    }
```

```java
/**
 * Returns current phase.
 *
 * @return current phase.
 */
public int getPhase() {
    return state.getPhase();
}

/**
 * Returns string representation of this tree.
 *
 * @return string representation of this tree.
 */
public String toString() {
    return "Suffix tree:\n" + tree.toString() +
            "Current state:\n" + state.toString();
}

/**
 * Returns <code>NodeShape</code> object for given node.
 *
 * @param node suffix tree node.
 * @return <code>NodeShape</code> component for <code>node</code>.
 */
public NodeShape getNodeShape(SuffixTreeNode node) {
    if (node == null) return null;
    return (NodeShape) nodes.get(node);
}

/**
 * Updates tree components.
 */
public void updateTree() {
    if (Options.DEBUG) System.out.println("SuffixTree.updateTree");
    removeAll();
    nodes.clear();

    edges.removeAllElements();

    edgeLock = false;
    lockedEdge = null;

    updateNode(state.getRoot());
    updateLinks(state.getRoot());

    elemCount.removeAllElements();
    if (state.getRoot() != null) {
        getNodeShape(state.getRoot()).calculateDepth(1);
        getNodeShape(state.getRoot()).calculateHorizontalPos();
    }

    validate();

    for (int i = 0; i < edges.size(); i++) {
        Edge edge = (Edge) edges.elementAt(i);
        if (edge.containsPoint(mousePoint.x, mousePoint.y))
            edge.acquireLock();
    }
    repaint();

    if (Options.DEBUG) System.out.println("Components updated");
}

/**
 * Updates subtree with give node as its root.
 *
 * @param node root of the subtree.
 */
protected void updateNode(SuffixTreeNode node) {
    if (node == null) return;

    NodeShape shape;
    if (node.isLeaf()) {
        shape = new NodeShape(styleSet, this, node,
                String.valueOf(node.getLeafNumber()));
```

```
    } else {
        shape = new NodeShape(styleSet, this, node);
    }

    nodes.put(node, shape);

    int nodeStyle = state.getNodeStyle(shape.getNode());
    shape.setStyle(nodeStyle);
    add(shape);

    SuffixTreeNode child = node.getLeftChild();
    while (child != null) {
        updateNode(child);

        Edge edge = new Edge(styleSet, shape, getNodeShape(child));
        int edgeStyle = state.getEdgeStyle(edge.getTargetNodeShape().getNode());
        edge.setStyle(edgeStyle);

            edge.setArrow(state.hasArrow(edge.getTargetNodeShape().getNode()));
            edge.setReverseArrow(state.hasReverseArrow(edge.getTargetNodeShape().
                getNode()));

        add(edge);
        edges.addElement(edge);

        child = child.getRightSibling();
    }
}

/**
 * Updates links of the subtree with give node as its root.
 *
 * @param node root of the subtree.
 */
protected void updateLinks(SuffixTreeNode node) {
    if (node == null) return;

    if (node.getSuffixLink() != null) {
        SuffixLink link = new SuffixLink(styleSet, getNodeShape(node),
                getNodeShape(node.getSuffixLink()));

        int linkStyle = state.getLinkStyle(link.getSourceNodeShape().getNode());
        link.setStyle(linkStyle);
        add(link);
    }

    SuffixTreeNode child = node.getLeftChild();
    while (child != null) {
        updateLinks(child);
        child = child.getRightSibling();
    }
}

/**
 * Checks if the node corresponding to the specified <code>NodeShape</code>
 * object is selected in the current algorithm state.
 *
 * @param node target node.
 * @return <code>true</code> if <code>node</code> is selected,
 *         <code>false</code> otherwise.
 */
public boolean isNodeSelected(NodeShape node) {
    return state.isNodeSelected(node.getNode());
}

/**
 * Adds new element at given depth.
 *
 * @param depth depth of the new element.
 * @return element position.
 */
protected int addElementAtDepth(int depth) {
    if (elemCount.size() < depth) {
        elemCount.addElement(new Integer(1));
        return 1;
    } else {
```

```
            int pos = ((Integer) elemCount.elementAt(depth - 1)).intValue();
            elemCount.setElementAt(new Integer(pos + 1), depth - 1);
            return pos + 1;
        }
    }

    /**
     * Returns number of elements at given depth.
     * @param depth depth.
     * @return number of elements.
     */
    public int getElementCountAtDepth(int depth) {
        return ((Integer) elemCount.elementAt(depth - 1)).intValue();
    }

    /**
     * Tries to acquire edge lock.
     *
     * @param edge edge that wants to acquire lock.
     * @return <code>true</code> if lock has been succesfully acquired.
     */
    public boolean acquireLock(Edge edge) {
        if (!edgeLock) {
            edgeLock = true;
            lockedEdge = edge;
            return true;
        } else
            return false;
    }

    /**
     * Releases edge lock.
     *
     * @param edge edge that has acquired the lock.
     */
    public void releaseLock(Edge edge) {
        if (edge == lockedEdge) {
            lockedEdge = null;
            edgeLock = false;
        }
    }

    /**
     * Paints the suffix tree.
     *
     * @param g {@link Graphics} on which to paint.
     */
    public void paint(Graphics g) {
        int width = getSize().width;
        int height = getSize().height;

        if (width > 0 && height > 0) {
            if (buffer == null) {
                buffer = this.createImage(width, height);
            }

            Graphics bg = buffer.getGraphics();
            bg.clearRect(0, 0, width, height);

            super.paint(bg);
            if (lockedEdge != null)
                lockedEdge.paintHint(bg);

            bg.dispose();

            if (buffer != null && g != null)
                g.drawImage(buffer, 0, 0, null);
        }
    }

    /**
     * Invalidates this component.
     */
    public void invalidate() {
        super.invalidate();
        buffer = null;
```

```java
}

/**
 * Updates the component.
 *
 * @param g {@link Graphics} on which to paint.
 */
public void update(Graphics g) {
    paint(g);
}

/**
 * Returns mouse cursor point.
 *
 * @return mouse cursor point.
 */
public Point getMousePoint() {
    return mousePoint;
}

/**
 * Moves specified component on top of this container.
 *
 * @param comp the component to be moved.
 */
protected void moveToTop(Component comp) {
    remove(comp);
    add(comp, 0);
}

/**
 * Class for handling mouse motion events for the edge.
 */
class TreeMouseMotionAdapter extends MouseMotionAdapter {
    /**
     * Handles mouse motion events.
     *
     * @param e <code>MouseEvent</code> object.
     */
    public void mouseMoved(MouseEvent e) {
        mousePoint = e.getPoint();
        for (int i = 0; i < edges.size(); i++) {
            Edge edge = (Edge) edges.elementAt(i);

            if (edge.containsPoint(e.getX(), e.getY())
                && !edge.getSourceNodeShape().containsPoint(e.getPoint())
                && !edge.getTargetNodeShape().containsPoint(e.getPoint())) {
                if (edge.acquireLock()) {
                    SuffixTreeNode node = edge.getTargetNodeShape().getNode();
                    applet.stringSelect(node.getBeginIndex(),
                                        node.getEndIndex());
                }
                if (edge.hasLock()) {
                    moveToTop(edge);
                    moveToTop(edge.getSourceNodeShape());
                    moveToTop(edge.getTargetNodeShape());
                    repaint();
                }
            } else {
                if (edge.releaseLock()) {
                    applet.stringSelect(-1, -1);
                    repaint();
                }
            }
        }
    }

    /**
     * Handles mouse dragging events.
     *
     * @param e <code>MouseEvent</code> object.
     */
    public void mouseDragged(MouseEvent e) {
        mousePoint = e.getPoint();
        for (int i = 0; i < edges.size(); i++) {
            Edge edge = (Edge) edges.elementAt(i);
```

```
                if (edge.containsPoint(e.getX(), e.getY())
                    && !edge.getSourceNodeShape().containsPoint(e.getPoint())
                    && !edge.getTargetNodeShape().containsPoint(e.getPoint())) {
                    if (edge.acquireLock()) {
                        SuffixTreeNode node = edge.getTargetNodeShape().getNode();
                        applet.stringSelect(node.getBeginIndex(),
                                            node.getEndIndex());
                    }
                    if (edge.hasLock()) {
                        moveToTop(edge);
                        moveToTop(edge.getSourceNodeShape());
                        moveToTop(edge.getTargetNodeShape());
                        repaint();
                    }
                } else {
                    if (edge.releaseLock()) {
                        applet.stringSelect(-1, -1);
                        repaint();
                    }
                }
            }
        }
    }
}
```

ru.ifmo.vizi.ukkonen.suffixtree.SuffixTreeData

```
package ru.ifmo.vizi.ukkonen.SuffixTree;

import java.util.Vector;

/**
 * Stores suffix tree.
 *
 * @author      Igor Ahmetov
 * @version     $Id$
 */
public class SuffixTreeData {
    /**
     * Debug flag.
     */
    public final static boolean DEBUG = true;

        /**
         * String data for this tree.
         */
        protected String string;

        /**
         * Root node.
         */
        protected SuffixTreeNode root;

    /**
     * States of the algorithm.
     */
    protected Vector states;

    /**
     * Current step.
     */
    protected int step;

    /**
     * Current extension.
     */
    protected int extension;

    /**
     * Current phase;
     */
    protected int phase;

        /**
```

```java
     * Default constructor.
     *
     * @param string        string for suffix tree.
     */
    public SuffixTreeData(String string) {
            this.string = string;
    states = new Vector();

    buildTree();
    }

/**
 * Builds tree for given string.
 */
private void buildTree() {
    if (Options.DEBUG) System.out.println("SuffixTreeData.buildTree()");

    SuffixTreeState state;
    states.addElement(
            new SuffixTreeState(-2, 0, null, null)
    );

    root = new SuffixTreeNode(this, null, null, null, null, -1, -1);

    if (string.length() == 0) return;

    root.addChar(0);
    SuffixTreeNode leaf = root.getLeftChild();

    states.addElement(
        new SuffixTreeState(-1, 0,  root, root)
    );

    for (int i = 0; i < string.length() - 1; i++) {
        phase = i;
        extension = 0;

        states.addElement(
            new SuffixTreeState(i, 0,  root, root)
        );

        SuffixTreeNode node = leaf;
        SuffixTreeNode lastNode = null;

        String activeSubstring = "";

        state = new SuffixTreeState(i, 0, root, node);
        state.selectNode(leaf);
        states.addElement(state);

        node.addChar(i + 1);
        int curpos = i + 1;

        state = new SuffixTreeState(i, 0, root, node);
        state.selectNode(leaf);
        state.selectEdge(leaf);
        states.addElement(state);

        for (int j = 1; j <= i + 1; j++) {
            extension = j;

            state =  new SuffixTreeState(i, j, root, node);
            state.selectNode(node);
            states.addElement(state);

            int len = 0;
            SuffixTreeNode link = node.getSuffixLink();

            if (link == null && !node.isRoot()) {
                len = node.getLength();
                activeSubstring = node.getString();
                if (node.isLeaf()) {
                    activeSubstring = activeSubstring.substring(0,
                            activeSubstring.length() - 1);
                }
```

```
        state = new SuffixTreeState(i, j, root, node.getParent());
        state.selectNode(node);
        state.selectNode(node.getParent());
        state.addReverseArrow(node);
        state.selectEdge(node);
        state.setActiveSubstring(activeSubstring);
        states.addElement(state);

        curpos = curpos - len;
        node = node.getParent();
    }

    if (node.getSuffixLink() != null) {
        state = new SuffixTreeState(i, j, root, node);
        state.selectLink(node);
        state.selectNode(node);
        state.selectNode(node.getSuffixLink());
        state.setActiveSubstring(activeSubstring);
        states.addElement(state);

        node = node.getSuffixLink();

        if (activeSubstring.length() != 0) {
            state = new SuffixTreeState(i, j, root, node);
            state.selectNode(node);
            state.setActiveSubstring(activeSubstring);
            states.addElement(state);
        }

        if (node.isRoot()) {
            curpos = j - 1;
            activeSubstring = getString().substring(j, i + 1);
        }
    } else {
        curpos = j - 1;
        activeSubstring = getString().substring(j, i + 1);

        state = new SuffixTreeState(i, j, root, node);
        state.selectNode(root);
        state.setActiveSubstring(activeSubstring);
        states.addElement(state);
    }

    while (true) {
        SuffixTreeNode child = node.getChild(curpos + 1);

        if (curpos == i) {
            node.addChar(i + 1);
            if (node.isLeaf()) {
                curpos++;

                state = new SuffixTreeState(i, j, root, node);
                state.selectNode(node);
                state.selectEdge(node);
                state.setSpecialState(SuffixTreeState.STATE_ADD_CHAR_LEAF);
                states.addElement(state);
            } else if (child == null) {
                state = new SuffixTreeState(i, j, root, node);
                state.selectNode(node);
                SuffixTreeNode newChild = node.getChild(curpos + 1);
                state.selectNode(newChild);
                state.selectEdge(newChild);
                state.setSpecialState(SuffixTreeState.STATE_ADD_CHAR_NEW_LEAF
                    );
                states.addElement(state);
            } else {
                state = new SuffixTreeState(i, j, root, node);
                state.selectNode(node);
                state.selectEdge(child);
                states.addElement(state);
            }

            if (lastNode != null) {
                lastNode.setSuffixLink(node);

                state = new SuffixTreeState(i, j, root, node);
```

```
            state.selectNode(lastNode);
            state.selectLink(lastNode);
            state.selectNode(node);
            state.setSpecialState(SuffixTreeState.STATE_LINK_ADDED);
            states.addElement(state);
        }
        lastNode = null;
        activeSubstring = "";
        break;
    }

    if (curpos + child.getLength() > i) {
        if (string.charAt(i - curpos + child.getBeginIndex()) ==
                string.charAt(i + 1)) {
            activeSubstring = string.substring(child.getBeginIndex(),
                    child.getBeginIndex() + i - curpos);

            state = new SuffixTreeState(i, j, root, node);
            state.setActiveSubstring(activeSubstring);
            state.selectNode(node);
            state.selectEdge(child);
            states.addElement(state);
            break;
        }

        state = new SuffixTreeState(i, j, root, child);
        state.selectNode(node);
        state.selectNode(child);
        state.selectEdge(child);
        state.setActiveSubstring(activeSubstring);
        state.setSpecialState(SuffixTreeState.STATE_ADD_CHAR_SPLIT_EDGE);
        states.addElement(state);

        SuffixTreeNode leftSibling = child.getLeftSibling();
        if (leftSibling != null)
            leftSibling.setRightSibling(child.getRightSibling());
        else
            node.setLeftChild(child.getRightSibling());
        child.setRightSibling(null);

        node.setLeftChild(
                new SuffixTreeNode(this, node, child, node.getLeftChild()
                    ,
                        null, child.getBeginIndex(),
                        i - curpos + child.getBeginIndex())
        );
        child.setBeginIndex(i - curpos + child.getBeginIndex());
        child.setParent(node.getLeftChild());

        curpos = i;
        node = node.getLeftChild();
        node.addChar(i + 1);

        state = new SuffixTreeState(i, j, root, node);
        state.selectNode(node);
        SuffixTreeNode newChild = node.getChild(i + 1);
        state.selectNode(newChild);
        state.selectEdge(newChild);
        states.addElement(state);

        state = new SuffixTreeState(i, j, root, node);
        state.selectNode(node);
        states.addElement(state);

        if (lastNode != null) {
            lastNode.setSuffixLink(node);

            state = new SuffixTreeState(i, j, root, node);
            state.selectNode(lastNode);
            state.selectLink(lastNode);
            state.selectNode(node);
            state.setSpecialState(SuffixTreeState.STATE_LINK_ADDED);
            states.addElement(state);
        }

        lastNode = node;
```

```
                        activeSubstring = "";

                        break;
                    }

                    state = new SuffixTreeState(i, j, root, child);
                    state.selectNode(node);
                    state.selectNode(child);
                    state.selectEdge(child);
                    state.addArrow(child);

                    activeSubstring = activeSubstring.substring(child.getLength(),
                            activeSubstring.length());
                    state.setActiveSubstring(activeSubstring);
                    state.setSpecialState(SuffixTreeState.STATE_DESCENT);
                    states.addElement(state);

                    node = child;
                    curpos += child.getLength();
                }

                state = new SuffixTreeState(i, j, root, node);
                state.selectNode(node);
                states.addElement(state);
            }
            states.addElement(new SuffixTreeState(-2, -2, root, node));
        }
        states.addElement(
            new SuffixTreeState(-2, -2, root, root)
        );
        if (Options.DEBUG) {
            System.out.println("SuffixTreeData.buildTree() finished");
            System.out.println("Generated " + states.size() + " states");
        }
    }

        /**
         * Returns string of this tree.
     *
     * @return string of this tree.
     */
        public String getString() {
                return string;
        }

        /**
         * Returns root node of this tree.
         *
         * @return      root node of this tree.
         */
        public SuffixTreeNode getRoot() {
        return root;
        }

    /**
     * Returns current algorithm state.
     *
     * @return current state.
     */
    public SuffixTreeState getState() {
        return (SuffixTreeState) states.elementAt(step);
    }

    /**
     * Returns algorithm state during next step.
     *
     * @return algorithm state during next step.
     */
    public SuffixTreeState getNextState() {
        return (SuffixTreeState) states.elementAt(step + 1);
    }

    /**
     * Performs next step.
     */
    public void nextStep() {
```

```
        step++;
    }

    /**
     * Returns to previous step.
     */
    public void previousStep() {
        step--;
    }

    /**
     * Returns string representation of this tree.
     *
     * @return string representation of this tree.
     */
    public String toString() {
        return root.toString();
    }

    int getExtension() {
        return extension;
    }

    int getPhase() {
        return phase;
    }
}
```

ru.ifmo.vizi.ukkonen.suffixtree.SuffixTreeLayout

```
package ru.ifmo.vizi.ukkonen.SuffixTree;

import java.awt.*;
import java.util.Hashtable;

/**
 * Suffix tree layout.
 *
 * @author      Igor Ahmetov
 * @version     $Id$
 */
public class SuffixTreeLayout implements LayoutManager {
    /**
     * Horizontal gap.
     */
    protected double hgap = 0.06;

    /**
     * Vertical gap.
     */
    protected double vgap = 0.1;

    /**
     * Node size.
     */
    protected double nodeSize = 0.04;

    /**
     * Components that have been layed out.
     */
    protected Hashtable components = new Hashtable();

        /**
         * Default constructor.
         */
        public SuffixTreeLayout() {
        }

        /**
     * Adds the specified component to the layout.
     *
     * @param name the name of the component
     * @param comp the component to be added
     */
    public void addLayoutComponent(String name, Component comp) {
```

```java
}

/**
 * Removes the specified component from the layout.
 *
 * @param comp the component to remove
 */
public void removeLayoutComponent(Component comp) {
}

/**
 * Returns the preferred dimensions for this layout given the
 * <i>visible</i> components in the specified target container.
 * @param target the component which needs to be laid out
 * @return    the preferred dimensions to lay out the
 *            subcomponents of the specified container
 */
public Dimension preferredLayoutSize(Container target) {
    synchronized (target.getTreeLock()) {
        return new Dimension(400, 400);
            }
}

/**
 * Returns the minimum dimensions needed to layout the <i>visible</i>
 * components contained in the specified target container.
 * @param target the component which needs to be laid out
 * @return    the minimum dimensions to lay out the
 *            subcomponents of the specified container
 */
public Dimension minimumLayoutSize(Container target) {
        synchronized (target.getTreeLock()) {
        return new Dimension(0, 0);
            }
}

/**
 * Layouts node <code>node</code> of tree <code>tree</code>, assuming
 * that depth of the tree is <code>depth</code>.
 *
 * @param tree suffix tree.
 * @param node node component to layout.
 * @param depth depth of the tree.
 * @param hoffset horizontal offset in the container.
 * @param width width of the container.
 * @param voffset vertical offset in the container.
 * @param height height of the container.
 */
protected void layoutNode(SuffixTree tree, NodeShape node, int depth,
                         int hoffset, int width,
                         int voffset, int height) {
    if (node == null) return;

    double hpos = 0.5;
    double vpos = (double) (node.getDepth() - 1) / (depth - 1);

    if (Options.DEBUG) System.out.println("hpos: " + hpos + "; vpos: " + vpos);

    double size = Math.min(tree.getSize().width * nodeSize,
            tree.getSize().height * nodeSize);

    if (node.isVisible()) {
        if (node.getMessage().length() != 0 || tree.isNodeSelected(node)) {
            node.setBounds((int) (hoffset + width * hpos) - (int) (size * 1.25),
                    (int) (voffset + height * vpos) - (int) (size * 1.25),
                    (int) (size * 1.25) * 2 + 1, (int) (size * 1.25) * 2 + 1);
            node.adjustFontSize();
        } else {
            node.setBounds((int) (hoffset + width * hpos)- (int) size / 2,
                    (int) (voffset + height * vpos) - (int) size / 2,
                    (int) size / 2 * 2 + 1, (int) size / 2 * 2 + 1);
        }
        components.put(node, new Object());
    }

    int totalWidth = node.getNode().getWidth();
```

```
        int currentWidth = 0;

        NodeShape child = node.getLeftChildShape();

        while (child != null) {
            int nodeWidth = child.getNode().getWidth();
            layoutNode(tree, child, depth,
                    hoffset + (int) ((double) width * currentWidth / totalWidth),
                    (int) ((double) width * nodeWidth / totalWidth), voffset, height);

            currentWidth += nodeWidth;
            child = child.getRightSiblingShape();
        }
    }

    /**
     * Lays out the container. This method lets each component take
     * its preferred size by reshaping the components in the
     * target container in order to satisfy the alignment of
     * this <code>SuffixTreeLayout</code> object.
     *
     * @param target    the specified component being laid out
     */
    public void layoutContainer(Container target) {
        synchronized (target.getTreeLock()) {
            if (Options.DEBUG) System.out.println("SuffixTreeLayout.layoutContainer()");
                        Insets insets = target.getInsets();
            SuffixTree tree = (SuffixTree) target;
            Dimension size = target.getSize();

            components.clear();

                        int width = size.width -
                    (insets.left + insets.right + (int) (size.width * hgap * 2));
            int height = size.height -
                    (insets.top + insets.bottom + (int) (size.height * vgap * 2));

                        int nmembers = target.getComponentCount();

            if (Options.DEBUG) System.out.println("Laying out " + nmembers + " components
                ");

            SuffixTreeNode root = tree.getState().getRoot();

            if (root != null) {
                int treeDepth = root.getDepth();

                layoutNode(tree, tree.getNodeShape(root), treeDepth,
                        insets.left + (int) (size.width * hgap), width,
                        insets.top + (int) (size.height * vgap), height);
            }

            for (int i = 0 ; i < nmembers ; i++) {
                            Component m = target.getComponent(i);
                        if (m.isVisible() && !components.containsKey(m)) {
                    m.setBounds(0, 0, size.width, size.height);
                            }
                        }
        }
    }
}
```

ru.ifmo.vizi.ukkonen.suffixtree.SuffixTreeNode

```
package ru.ifmo.vizi.ukkonen.SuffixTree;

/**
 * Represents suffix tree node.
 *
 * @author                Igor Ahmetov
 * @version      $Id$
 */
public class SuffixTreeNode {
    /**
     * Parent node of this node.
```

```
 */
protected SuffixTreeNode parent;

    /**
     * Leftmost child of this node.
     */
    protected SuffixTreeNode leftChild;

    /**
     * Right sibling of this node.
     */
    protected SuffixTreeNode rightSibling;

/**
 * Suffix link.
 */
protected SuffixTreeNode suffixLink;

    /**
     * Start position of this node's substring.
     */
    protected int beginIndex;

    /**
     * End position of this node's substring.
     */
    protected int endIndex;

    /**
     * Suffix tree to which this node belongs.
     */
    protected SuffixTreeData tree;

/**
 * Leaf number of this node.
 */
protected int leafNumber;

    /**
     * Creates an inner node of a suffix tree..
     *
     * @param tree  suffix tree to which this node belongs
 * @param parent    parent node of this node.
     * @param leftChild     leftmost child of this node.
     * @param rightSibling  right sibling of this node.
 * @param suffixLink    suffix link of this node.
     * @param beginIndex    start position of substring for this node.
     * @param endIndex      end position of substring for this node.
     */
    public SuffixTreeNode(SuffixTreeData tree,
                          SuffixTreeNode parent, SuffixTreeNode leftChild,
                          SuffixTreeNode rightSibling, SuffixTreeNode suffixLink,
                          int beginIndex, int endIndex) {
            this.beginIndex = beginIndex;
            this.endIndex = endIndex;

            this.parent = parent;
    this.leftChild = leftChild;
            this.rightSibling = rightSibling;
    this.suffixLink = suffixLink;

            this.tree = tree;
    }

/**
     * Constructs leaf node of a suffix tree.
     *
     * @param tree  suffix tree to which this node belongs
 * @param parent    parent node of this node.
 * @param rightSibling  right sibling of this node.
 * @param leafNumber    leaf number of this node.
     * @param beginIndex    start position of substring for this node.
     * @param endIndex      end position of substring for this node.
     */
    public SuffixTreeNode(SuffixTreeData tree,
                          SuffixTreeNode parent, SuffixTreeNode rightSibling,
```

```
                         int leafNumber,
                         int beginIndex, int endIndex) {
              this.beginIndex = beginIndex;
              this.endIndex = endIndex;

              this.parent = parent;
    this.leafNumber = leafNumber;

    this.rightSibling = rightSibling;

              this.tree = tree;
    }

/**
 * Returnes leaf number of this node if this node is a leaf.
 *
 * @return leaf number of this node.
 */
public int getLeafNumber() {
    return leafNumber;
}

/**
 * Returns begin index of this node.
 *
 * @return begin index of this node.
 */
public int getBeginIndex() {
    return beginIndex;
}

/**
 * Sets begin index of this node.
 *
 * @param beginIndex new begin index of this node.
 */
public void setBeginIndex(int beginIndex) {
    this.beginIndex = beginIndex;
}

/**
 * Returns end index node of this node.
 *
 * @return end index of this node.
 */
public int getEndIndex() {
    return endIndex;
}

/**
 * Sets end index of this node.
 *
 * @param endIndex new parent for this node.
 */
public void setParent(int endIndex) {
    this.endIndex = endIndex;
}

/**
 * Returns parent node of this node.
 *
 * @return parent node of this node.
 */
public SuffixTreeNode getParent() {
    return parent;
}

/**
 * Sets parent of this node.
 *
 * @param parent new parent for this node.
 */
public void setParent(SuffixTreeNode parent) {
    this.parent = parent;
}
```

```java
/**
 * Returns leftmost child of this node.
 *
 * @return leftmost child of this node or <code>null</code> if none.
 */
public SuffixTreeNode getLeftChild() {
    return leftChild;
}

/**
 * Sets leftmost child of this node.
 *
 * @param leftChild new leftmost child of this node.
 */
public void setLeftChild(SuffixTreeNode leftChild) {
    this.leftChild = leftChild;
}

/**
 * Returns right sibling of this node.
 *
 * @return right sibling of this node or <code>null</code> if none.
 */
public SuffixTreeNode getRightSibling() {
    return rightSibling;
}

/**
 * Sets right sibling of this node.
 *
 * @param rightSibling new right sibling of this node.
 */
public void setRightSibling(SuffixTreeNode rightSibling) {
    this.rightSibling = rightSibling;
}

/**
 * Returns suffix link of this node.
 *
 * @return suffix link of this node or <code>null</code> if none.
 */
public SuffixTreeNode getSuffixLink() {
    return suffixLink;
}

/**
 * Sets suffix link of this node.
 *
 * @param suffixLink new suffix link of this node.
 */
public void setSuffixLink(SuffixTreeNode suffixLink) {
    this.suffixLink = suffixLink;
}

/**
 * Returns left sibling of this node.
 *
 * @return left sibling of this node or <code>null</code> if none.
 */
public SuffixTreeNode getLeftSibling() {
    SuffixTreeNode node = parent.getLeftChild();
    if (node == this)
        return null;
    while (node.getRightSibling() != this) {
        node = node.getRightSibling();
    }
    return node;
}

/**
 * Returns rightmost child of this node.
 *
 * @return rightmost child of this node or <code>null</code> if none.
 */
public SuffixTreeNode getRightChild() {
    SuffixTreeNode node = getLeftChild();
```

```
    if (node == null)
        return null;

    while (node.getRightSibling() != null) {
        node = node.getRightSibling();
    }
    return node;
}

/**
 * Assures that there is an extension from this node that
 * starts woth given character and creates new node if there isn't one.
 *
 * @param index index of the character.
 */
public void addChar(int index) {
    SuffixTreeNode node = getChild(index);
    if (node != null) return;

    if (leftChild == null) {
        if (parent == null) {
            leftChild = new SuffixTreeNode(tree, this, null, tree.getExtension() + 1,
                index, index + 1);
        } else {
            endIndex++;
        }
    } else {
        leftChild = new SuffixTreeNode(tree, this, leftChild, tree.getExtension()
             + 1,
                    index, index + 1);
    }
}


/**
 * Returns depth of the subtree with root as this node.
 *
 * @return depth of the subtree with root as this node.
 */
public int getDepth() {
    int depth = 0;
    SuffixTreeNode node = leftChild;

    while (node != null) {
        depth = Math.max(depth, node.getDepth());
        node = node.getRightSibling();
    }

    return depth + 1;
}

/**
 * Returns width of the subtree with root as this node.
 *
 * @return width of the subtree with root as this node.
 */
public int getWidth() {
    int width = 0;
    SuffixTreeNode node = leftChild;

    while (node != null) {
        width = width + node.getWidth();
        node = node.getRightSibling();
    }

    if (width == 0)
        return 1;
    else
        return width;
}

/**
 * Returns substring on this node.
 *
 * @return  substring on this node.
 */
```

```java
public String getString() {
    if (parent == null)
        return "root";
    else
        return tree.getString().substring(beginIndex, endIndex);
}

/**
 * Returns length of the label on this node.
 *
 * @return length of the label on this node.
 */
public int getLength() {
    return endIndex - beginIndex;
}

/**
 * Returns tree to which this node belongs.
 *
 * @return tree to which this node belongs.
 */
public SuffixTreeData getTree() {
    return tree;
}

/**
 * Returns length of the label between root node and this node.
 *
 * @return length of the label between root node and this node.
 */
public int getLabelLength() {
    if (parent == null)
        return getLength();
    else
        return getLength() + parent.getLabelLength();
}

/**
 * Returns <code>true</code> if this node is root node of the tree.
 *
 * @return <code>true</code> if this node is root node of the tree.
 */
public boolean isRoot() {
    return parent == null;
}

/**
 * Returns <code>true</code> if this node is a leaf.
 *
 * @return <code>true</code> if this node is a leaf.
 */
public boolean isLeaf() {
    return leftChild == null;
}

/**
 * Returns first char of this node.
 *
 * @return first char of this node.
 */
public char getFirstChar() {
    return tree.getString().charAt(beginIndex);
}

/**
 * Returns child node that starts with given character.
 *
 * @param index index of the character.
 * @return child node or <code>null</code> if none.
 */
public SuffixTreeNode getChild(int index) {
    SuffixTreeNode node = leftChild;
    while (node != null) {
        if (tree.getString().charAt(index) == node.getFirstChar()) {
            return node;
        }
```

```
            node = node.getRightSibling();
        }
        return null;
    }

    /**
     * Returns child node that starts with given character.
     *
     * @param ch character.
     * @return child node or <code>null</code> if none.
     */
    public SuffixTreeNode getChild(char ch) {
        SuffixTreeNode node = leftChild;
        while (node != null) {
            if (ch == node.getFirstChar()) {
                return node;
            }
            node = node.getRightSibling();
        }
        return null;
    }

    /**
     * Returns string representation of this node.
     *
     * @return string representation of this node.
     */
    public String toString() {
        String string = "Node: ";
        if (parent == null)
            string += "root; ";
        else
            string += getString() + "; parent: " + parent.getString() + "; ";

        if (suffixLink != null)
            string += "link: " + suffixLink.getString() + ";";
        string += "\nChildren: ";

        SuffixTreeNode node = leftChild;
        while (node != null) {
            string += node.getString() + " ";
            node = node.getRightSibling();
        }
        string += "\n\n";

        node = leftChild;
        while (node != null) {
            string += node.toString();
            node = node.getRightSibling();
        }

        return string;
    }

    /**
     * Checks if this node has suffix link.
     *
     * @return <code>true</code> if this node has suffix link.
     */
    public boolean hasSuffixLink() {
        return suffixLink != null;
    }
}
```

ru.ifmo.vizi.ukkonen.suffixtree.SuffixTreeState

```
package ru.ifmo.vizi.ukkonen.SuffixTree;

import java.util.Hashtable;
import java.util.Vector;

/**
 * Represents current state of the Ukkonen algorithm.
 *
 * @author      Igor Ahmetov
```

```
 * @version      $Id$
 */
public class SuffixTreeState {
    /**
     * Style of leaf nodes.
     */
    public final static int LEAF_STYLE = 0;

    /**
     * Style of selected leaf nodes.
     */
    public final static int SELECTED_LEAF_STYLE = 1;

    /**
     * Style of inner nodes.
     */
    public final static int NODE_STYLE = 2;

    /**
     * Style of selected inner nodes.
     */
    public final static int SELECTED_NODE_STYLE = 3;

    /**
     * Style of edges.
     */
    public final static int EDGE_STYLE = 4;

    /**
     * Style of selected edges.
     */
    public final static int SELECTED_EDGE_STYLE = 5;

    /**
     * Style of suffix links.
     */
    public final static int LINK_STYLE = 6;

    /**
     * Style of selected suffix links.
     */
    public final static int SELECTED_LINK_STYLE = 7;

    /**
     * Default state.
     */
    public final static int STATE_NONE = 0;

    /**
     * Performing descent in the tree.
     */
    public final static int STATE_DESCENT = 1;

    /**
     * A char was added to the leaf label.
     */
    public final static int STATE_ADD_CHAR_LEAF = 2;

    /**
     * Edge was splitted.
     */
    public final static int STATE_ADD_CHAR_SPLIT_EDGE = 3;

    /**
     * New leaf was created during adding of a char to the tree.
     */
    public final static int STATE_ADD_CHAR_NEW_LEAF = 4;

    /**
     * Suffix link has been added to the tree.
     */
    public final static int STATE_LINK_ADDED = 5;

    /**
     * Phase of the algorithm.
     */
```

```java
protected int phase;

/**
 * Extension number.
 */
protected int extension;

/**
 * State of the algorithm.
 */
protected int state;

/**
 * Root node.
 */
protected SuffixTreeNode root;

/**
 * Hashtable for matching nodes.
 */
protected Hashtable map;

/**
 * Selected suffix links.
 */
protected Vector selectedLinks = new Vector();

/**
 * Selected edges.
 */
protected Vector selectedEdges = new Vector();

/**
 * Selected nodes.
 */
protected Vector selectedNodes = new Vector();

/**
 * Edges with arrows.
 */
protected Vector arrowEdges = new Vector();

protected Vector reverseArrowEdges = new Vector();

/**
 * Current suffix tree node.
 */
protected SuffixTreeNode currentNode;

/**
 * Active substring.
 */
protected String activeSubstring = "";

/**
 * Default constructor.
 *
 * @param phase phase of the algorithm.
 * @param extension extension number.
 * @param root root node of the tree.
 * @param currentNode current suffix tree node.
 */
public SuffixTreeState(int phase, int extension, SuffixTreeNode root,
                       SuffixTreeNode currentNode) {
    this.phase = phase;
    this.extension = extension;

    map = new Hashtable();

    this.root = copyTree(root);
    copyLinks(root);

    if (currentNode != null)
        this.currentNode = (SuffixTreeNode) map.get(currentNode);

    if (Options.DEBUG) {
```

```
            System.out.println("SuffixTreeState()");
            System.out.println("Stored state:\n" + this.root);
        }
    }

    /**
     * Adds given node to the list of selected nodes.
     *
     * @param node target node.
     */
    public void selectNode(SuffixTreeNode node) {
        selectedNodes.addElement(map.get(node));
    }

    /**
     * Adds the suffix link from a given node to the list of selected links.
     *
     * @param node source of the link.
     */
    public void selectLink(SuffixTreeNode node) {
        selectedLinks.addElement(map.get(node));
    }

    /**
     * Adds edge from the given node to its parent to the list of selected edges.
     *
     * @param node end of the edge.
     */
    public void selectEdge(SuffixTreeNode node) {
        selectedEdges.addElement(map.get(node));
    }

    /**
     * Adds arrow to the edge.
     *
     * @param edge.
     */
    public void addArrow(SuffixTreeNode node) {
        arrowEdges.addElement(map.get(node));
    }

    public void addReverseArrow(SuffixTreeNode node) {
        reverseArrowEdges.addElement(map.get(node));
    }

    /**
     * Copies suffix tree without suffix links.
     *
     * @param node root of the tree to copy.
     * @return root of the new tree.
     */
    protected SuffixTreeNode copyTree(SuffixTreeNode node) {
        if (node == null) return null;

        SuffixTreeNode newNode;

        if (node.getParent() == null) {
            if (node.isLeaf()) {
                newNode = new SuffixTreeNode(node.getTree(),
                        null, copyTree(node.getRightSibling()), node.getLeafNumber(),
                        node.getBeginIndex(), node.getEndIndex());
            } else {
                newNode = new SuffixTreeNode(node.getTree(),
                        null, null, copyTree(node.getRightSibling()), null,
                        node.getBeginIndex(), node.getEndIndex());
            }
        } else {
            if (node.isLeaf()) {
                newNode = new SuffixTreeNode(node.getTree(),
                        (SuffixTreeNode) map.get(node.getParent()),
                        copyTree(node.getRightSibling()), node.getLeafNumber(),
                        node.getBeginIndex(), node.getEndIndex());
            } else {
                newNode = new SuffixTreeNode(node.getTree(),
                        (SuffixTreeNode) map.get(node.getParent()),
                        null, copyTree(node.getRightSibling()), null,
```

```java
                        node.getBeginIndex(), node.getEndIndex());
            }
        }

        map.put(node, newNode);
        newNode.setLeftChild(copyTree(node.getLeftChild()));

        return newNode;
    }

    /**
     * Corrects links.
     *
     * @param node root of the old tree.
     */
    protected void copyLinks(SuffixTreeNode node) {
        if (node == null) return;

        SuffixTreeNode newNode = (SuffixTreeNode) map.get(node);
        if (node.getSuffixLink() != null)
            newNode.setSuffixLink((SuffixTreeNode) map.get(node.getSuffixLink()));

        copyLinks(node.getLeftChild());
        copyLinks(node.getRightSibling());
    }

    /**
     * Returns current algorithm phase.
     *
     * @return algorithm phase.
     */
    public int getPhase() {
        return phase;
    }

    /**
     * Returns current extension number.
     *
     * @return current extension number.
     */
    public int getExtension() {
        return extension;
    }

    /**
     * Returns current algorithm special state.
     *
     * @return algorithm state.
     */
    public int getSpecialState() {
        return state;
    }

    /**
     * Sets current algorithm special state.
     *
     * @param state new state.
     */
    public void setSpecialState(int state) {
        this.state = state;
    }

    /**
     * Returns suffix tree root.
     *
     * @return suffix tree root.
     */
    public SuffixTreeNode getRoot() {
        return root;
    }

    /**
     * Retuns current node.
     *
     * @return current node.
     */
```

```java
public SuffixTreeNode getCurrentNode() {
    return currentNode;
}

/**
 * Returns string representation of this <code>SuffixTreeState</code> object
 *
 * @return string representation of this object.
 */
public String toString() {
    if (root != null)
        return root.toString();
    else
        return "null\n";
}

/**
 * Returns style of the node <code>node</code>.
 *
 * @param node target.
 * @return style of the node.
 */
public int getNodeStyle(SuffixTreeNode node) {
    if (node.isLeaf()) {
        if (selectedNodes.contains(node))
            return SELECTED_LEAF_STYLE;
        else
            return LEAF_STYLE;
    } else {
        if (selectedNodes.contains(node))
            return SELECTED_NODE_STYLE;
        else
            return NODE_STYLE;
    }
}

/**
 * Returns style of the suffix link from node <code>node</code>.
 *
 * @param node target.
 * @return style of the link from specified node.
 */
public int getLinkStyle(SuffixTreeNode node) {
    if (selectedLinks.contains(node))
        return SELECTED_LINK_STYLE;
    else
        return LINK_STYLE;
}

/**
 * Returns style of the edge between node <code>node</code> and its parent.
 *
 * @param node target.
 * @return style of the edge between specified node and its parent.
 */
public int getEdgeStyle(SuffixTreeNode node) {
    if (selectedEdges.contains(node))
        return SELECTED_EDGE_STYLE;
    else
        return EDGE_STYLE;
}

/**
 * Returns <code>true</code> if given node is selected.
 *
 * @param node node to check for selection.
 * @return <code>true</code> if specified node is selected.
 */
public boolean isNodeSelected(SuffixTreeNode node) {
    return selectedNodes.contains(node);
}

/**
 * Returns <code>true</code> if given edge contains arrow.
 *
 * @param node check edge from this node to its parent.
```

```
     * @return <code>true</code> if specified edge contains arrow.
     */
    public boolean hasArrow(SuffixTreeNode node) {
        return arrowEdges.contains(node);
    }

    public boolean hasReverseArrow(SuffixTreeNode node) {
        return reverseArrowEdges.contains(node);
    }

    /**
     * Sets active substring.
     *
     * @param substring new active substring.
     */
    public void setActiveSubstring(String substring) {
        activeSubstring = substring;
    }

    /**
     * Returns active substring of this state.
     *
     * @return active substring.
     */
    public String getActiveSubstring() {
        return activeSubstring;
    }
}
```

ru.ifmo.vizi.ukkonen.ui.HintedCheckbox

```
package ru.ifmo.vizi.ukkonen.ui;

import ru.ifmo.vizi.base.Configuration;
import ru.ifmo.vizi.base.ui.Hinter;

import java.awt.*;
import java.awt.event.ComponentEvent;

/**
 * Simple checkbox with hint.
 * <p>
 * Used configuration parameters:
 * <table border="1">
 *      <tr>
 *          <th>Name</th>
 *          <th>Description</th>
 *      </th>
 *      <tr>
 *          <td><b>name</b></td>
 *          <td>Checkbox caption</td>
 *      </tr>
 *      <tr>
 *          <td><b>name</b>-hint</td>
 *          <td>Checkbox hint</td>
 *      </tr>
 * </table>
 *
 * @author  Igor Ahmetov
 * @version $Id: 0.0$
 */

public class HintedCheckbox extends Checkbox {
    /**
     * Creates a new hinted checkbox without caption and hint.
     */
    protected HintedCheckbox() {
        enableEvents(AWTEvent.ACTION_EVENT_MASK | AWTEvent.COMPONENT_EVENT_MASK);
    }

    /**
     * Creates a new hinted checkbox with caption specified in parameter
     * <link>name</link> and hint specified in parameter <link>name</link>-hint.
     *
     * @param config applet configuration.
```

```
     * @param name checkbox name.
     */
    public HintedCheckbox(Configuration config, String name) {
        this();

        setLabel(config.getParameter(name));
        setHint(config.getParameter(name + "-hint"));
    }

    /**
     * Hint to show.
     */
    private String hint;

    /**
     * Sets new hint for this checkbox.
     *
     * @param hint hint to set.
     */
    public void setHint(String hint) {
        this.hint = hint;
        applyHint();
    }

    /**
     * Applies hint, if needed.
     */
    private void applyHint() {
        if (isVisible()) {
            Hinter.applyHint(this, hint);
        } else {
            Hinter.applyHint(this, null);
        }
    }

    /**
     * Invoked when component state changes.
     *
     * @param e the component event
     */
    protected final void processComponentEvent(ComponentEvent e) {
        super.processComponentEvent(e);
        applyHint();
    }
}
```

ru.ifmo.vizi.ukkonen.ui.HintedChoice

```
package ru.ifmo.vizi.ukkonen.ui;

import ru.ifmo.vizi.base.Configuration;
import ru.ifmo.vizi.base.ui.Hinter;

import java.awt.*;
import java.awt.event.ComponentEvent;

/**
 * Simple choice with hint.
 * <p>
 * Used configuration parameters:
 * <table border="1">
 *      <tr>
 *          <th>Name</th>
 *          <th>Description</th>
 *      </th>
 *      <tr>
 *          <td><b>name</b>-hint</td>
 *          <td>Choice hint</td>
 *      </tr>
 * </table>
 *
 * @author  Igor Ahmetov
 * @version $Id: 0.0$
 */
```

```java
public class HintedChoice extends Choice {

    /**
     * Creates a new hinted choice without caption and hint.
     */
    protected HintedChoice() {
        enableEvents(AWTEvent.ACTION_EVENT_MASK | AWTEvent.COMPONENT_EVENT_MASK);
    }

    /**
     * Creates a new hinted choice with hint specified in parameter <link>name</link>-
         hint.
     *
     * @param config applet configuration.
     * @param name choice name.
     */
    public HintedChoice(Configuration config, String name) {
        this();

        setHint(config.getParameter(name + "-hint"));
    }

    /**
     * Hint to show.
     */
    private String hint;

    /**
     * Sets new hint for this choice.
     *
     * @param hint hint to set.
     */
    public void setHint(String hint) {
        this.hint = hint;
        applyHint();
    }

    /**
     * Applies hint, if needed.
     */
    private void applyHint() {
        if (isVisible()) {
            Hinter.applyHint(this, hint);
        } else {
            Hinter.applyHint(this, null);
        }
    }

    /**
     * Invoked when component state changes.
     *
     * @param e the component event
     */
    protected final void processComponentEvent(ComponentEvent e) {
        super.processComponentEvent(e);
        applyHint();
    }
}
```

ru.ifmo.vizi.ukkonen.ui.HintedTextField

```java
package ru.ifmo.vizi.ukkonen.ui;

import ru.ifmo.vizi.base.Configuration;
import ru.ifmo.vizi.base.ui.Hinter;

import java.awt.*;
import java.awt.event.ComponentEvent;

/**
 * Simple textfield with hint.
 *
 * @author  Igor Ahmetov
 * @version $Id$
 */
```

```java
public class HintedTextField extends TextField {
    /**
     * Creates a new hinted textfiled without hint,
     * initialized with the specified text to be displayed,
     * and wide enough to hold the specified number of columns.
     *
     * @param string    the text to be displayed.
     * @param columns   the number of columns.
     */
    protected HintedTextField(String string, int columns) {
        super(string, columns);
        enableEvents(AWTEvent.ACTION_EVENT_MASK | AWTEvent.COMPONENT_EVENT_MASK);
    }

    /**
     * Creates a new hinted textfield with hint specified in parameter
     * <link>name</link>-hint, initialized with the specified text
     * to be displayed, and wide enough to hold the specified number of
     * columns.
     *
     * @param config applet configuration.
     * @param name text field name.
     */
    public HintedTextField(Configuration config, String name,
                           String text, int columns) {
        this(text, columns);

        setHint(config.getParameter(name + "-hint"));
    }

    /**
     * Hint to show.
     */
    private String hint;

    /**
     * Sets new hint for this text field..
     *
     * @param hint hint to set.
     */
    public void setHint(String hint) {
        this.hint = hint;
        applyHint();
    }

    /**
     * Applies hint, if needed.
     */
    private void applyHint() {
        if (isVisible()) {
            Hinter.applyHint(this, hint);
        } else {
            Hinter.applyHint(this, null);
        }
    }

    /**
     * Invoked when component state changes.
     *
     * @param e the component event
     */
    protected final void processComponentEvent(ComponentEvent e) {
        super.processComponentEvent(e);
        applyHint();
    }
}
```

ru.ifmo.vizi.ukkonen.widgets.StringVisualizer

```java
package ru.ifmo.vizi.ukkonen.widgets;

import ru.ifmo.vizi.base.widgets.Rect;
import ru.ifmo.vizi.base.widgets.ShapeStyle;
import ru.ifmo.vizi.ukkonen.UkkonenVisualizer;
```

```java
import java.util.Vector;
import java.awt.*;
import java.awt.event.*;

/**
 * Class for string visualization.
 *
 * @author  Igor Ahmetov
 */

public class StringVisualizer extends Panel {
    /**
     * Panel style.
     */
    protected final static int VISUALIZER_STYLE = 0;
    /**
     * Default style for character cells.
     */
    protected final static int DEFAULT_STYLE = 1;

    /**
     * Selected style.
     */
    protected final static int HIGHLIGHT_STYLE = 2;

    /**
     * Highlight style for character cells.
     */
    protected final static int SELECT_STYLE = 3;

    /**
     * Numbers style.
     */
    protected final static int NUMBERS_STYLE = 4;

    /**
     * Height of the component, used for resizing with mouse.
     */
    protected int height;

    /**
     * New height of the component, used for resizing with mouse.
     */
    protected int newHeight;

    /**
     * Is <code>true</code> when the component is being resized.
     */
    protected boolean resizing = false;

    /**
     * Used for holding the elements.
     */
    protected Vector cells;

    /**
     * Used for holding components that represent numbers.
     */
    protected Vector numbers;

    /**
     * Current string.
     */
    protected String string;

    /**
     * Cells styleset.
     */
    protected ShapeStyle[] styleSet;

    /**
     * Used in double-buffering.
     */
    protected Image buffer;
```

```java
/**
 * Begin index of the highlighted part of the string.
 */
protected int highlightBeginIndex;

/**
 * End index of the highlighted part of the string.
 */
protected int highlightEndIndex;

/**
 * Begin index of the selected part of the string.
 */
protected int selectBeginIndex;

/**
 * End index of the selected part of the string.
  */
protected int selectEndIndex;

/**
 * Parent applet.
 */
protected UkkonenVisualizer applet;

/**
 * Default constructor.
 *
 * @param styleSet cells styleset.
 */
public StringVisualizer(UkkonenVisualizer applet, ShapeStyle[] styleSet) {
    setLayout(null);

    this.styleSet = styleSet;
    this.applet = applet;

    this.addComponentListener(new ComponentAdapter() {
        public void componentResized(ComponentEvent e) {
            StringVisualizer.this.resize();
        }
    });

    setString("");

    addMouseMotionListener(new StringMouseMotionAdapter());
    addMouseListener(new StringMouseAdapter());
}

/**
 * Sets string.
 *
 * @param string new string.
 */
public void setString(String string) {
    this.string = string;
    removeAll();

    cells = new Vector();
    numbers = new Vector();
    for (int i = 0; i < string.length(); i++) {
            Rect elem = new Rect(styleSet, string.substring(i, i + 1));
        elem.setStyle(DEFAULT_STYLE);
            cells.addElement(elem);

        Rect number = new Rect(styleSet, String.valueOf(i + 1));
        number.setStyle(NUMBERS_STYLE);
        numbers.addElement(number);

            add(elem);
        add(number);
    }

    resize();
    repaint();
}
```

```java
/**
 * Returnes current string.
 *
 * @return string current string.
 */
public String getString() {
    return string;
}

/**
 * Higlights cells from <code>beginIndex</code> to
 * <code>endIndex - 1</code> (inclusive).
 *
 * @param beginIndex
 * @param endIndex
 */
public void highlight(int beginIndex, int endIndex) {
    highlightBeginIndex = beginIndex;
    highlightEndIndex = endIndex;

    for (int i = 0; i < cells.size(); i++) {
        ((Rect) cells.elementAt(i)).setStyle(DEFAULT_STYLE);
    }
    for (int i = Math.max(0, beginIndex); i < Math.min(endIndex, cells.size()); i++)
         {
        ((Rect) cells.elementAt(i)).setStyle(HIGHLIGHT_STYLE);
    }
    for (int i = Math.max(0, selectBeginIndex);
         i < Math.min(selectEndIndex, cells.size()); i++) {
        ((Rect) cells.elementAt(i)).setStyle(SELECT_STYLE);
    }

    repaint();
}

public void select(int beginIndex, int endIndex) {
    selectBeginIndex = beginIndex;
    selectEndIndex = endIndex;

    for (int i = 0; i < cells.size(); i++) {
        ((Rect) cells.elementAt(i)).setStyle(DEFAULT_STYLE);
    }
    for (int i = Math.max(0, highlightBeginIndex);
         i < Math.min(highlightEndIndex, cells.size()); i++) {
        ((Rect) cells.elementAt(i)).setStyle(HIGHLIGHT_STYLE);
    }
    for (int i = Math.max(0, beginIndex); i < Math.min(endIndex, cells.size()); i++)
         {
        ((Rect) cells.elementAt(i)).setStyle(SELECT_STYLE);
    }

    repaint();
}

/**
 * Resizes the component. Must be called every time after the
 * component has been resized.
 */
public void resize() {
    int n = cells.size();
    Dimension size = getSize();

    int vskip = (int) Math.round((double) size.height / 8);
    int hskip = 10;
    int width = (int) Math.min(Math.round((double) (size.width - 2 * hskip) / n),
            Math.round((double) (size.height - 2 * vskip) * 5 / 8));
    int numHeight =  (int) Math.round((double) 3 / 5 *  width);

    int hgap = (size.width - width * n) / 2;

    for (int i = 0; i < n; i++) {
        Rect rect = (Rect) cells.elementAt(i);
        rect.setBounds(hgap + i * width, vskip, width + 1, width + 1);
        rect.adjustFontSize();

        rect = (Rect) numbers.elementAt(i);
```

```java
            rect.setBounds(hgap + i * width, vskip + width, width + 1, numHeight + 1);
            rect.adjustFontSize();
        }
    }

    /**
     * Paints the string tree.
     *
     * @param g {@link Graphics} on which to paint.
     */
    public void paint(Graphics g) {
        int width = getSize().width;
        int height = getSize().height;

        if (width > 0 && height > 0) {
            if (buffer == null) {
                buffer = this.createImage(width, height);
            }

            Graphics bg = buffer.getGraphics();

            if (styleSet[VISUALIZER_STYLE].getFillStatus()) {
                bg.setColor(styleSet[VISUALIZER_STYLE].getFillColor());
            }
            bg.fillRect(0, 0, width, height);

            super.paint(bg);

            if (styleSet[VISUALIZER_STYLE].getBorderStatus()) {
                bg.setColor(styleSet[VISUALIZER_STYLE].getBorderColor());
                bg.drawLine(0, 0, width - 1, 0);
            }

            bg.dispose();

            g.drawImage(buffer, 0, 0, null);
        }
    }

    /**
     * Invalidates this component.
     */
    public void invalidate() {
        super.invalidate();
        buffer = null;
    }

    /**
     * Updates the component.
     *
     * @param g {@link Graphics} on which to paint.
     */
    public void update(Graphics g) {
        paint(g);
    }

    /**
     * Supporting class for handling mouse motion events for the string visualizer.
     *
     * @author  Igor Ahmetov
     * @version $Id$
     */
    class StringMouseMotionAdapter extends MouseMotionAdapter {
        /**
         * Handles mouse dragging.
         *
         * @param e mouse event.
         */
        public void mouseDragged(MouseEvent e) {
            if (resizing) {
                if (height - e.getY() > 5 && height - e.getY() < getParent().getSize().
                    height - 5)
                    newHeight = height - e.getY();
            }
        }
```

```
        /**
         * Handles mouse movement.
         *
         * @param e mouse event.
         */
        public void mouseMoved(MouseEvent e) {
            if (e.getY() < 10) {
                setCursor(new Cursor(Cursor.N_RESIZE_CURSOR));
            } else {
                setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
            }
        }
    }

    /**
     * Supporting class for handling mouse events for the string visualizer.
     *
     * @author  Igor Ahmetov
     * @version $Id$
     */
    class StringMouseAdapter extends MouseAdapter {
        /**
         * Handles mouse clicks.
         *
         * @param e mouse event.
         */
        public void mousePressed(MouseEvent e) {
            if (e.getY() < 10) {
                resizing = true;
                newHeight = height = getSize().height;
            }
        }

        /**
         * Handles mouse clicks.
         * @param e mouse event.
         */
        public void mouseReleased(MouseEvent e) {
            if (resizing) {
                resizing = false;
                setSize(getSize().width, newHeight);
                applet.resizeStringVisualizers();
                getParent().validate();
            }
        }
    }
}
```

ru.ifmo.vizi.ukkonen.widgets.VerticalStringVisualizer

```
package ru.ifmo.vizi.ukkonen.widgets;

import ru.ifmo.vizi.base.widgets.Rect;
import ru.ifmo.vizi.base.widgets.ShapeStyle;
import ru.ifmo.vizi.ukkonen.UkkonenVisualizer;

import java.util.Vector;
import java.awt.*;
import java.awt.event.*;

/**
 * Class for string visualization.
 *
 * @author  Igor Ahmetov
 */

public class VerticalStringVisualizer extends Panel {
    /**
     * Panel style.
     */
    protected final static int VISUALIZER_STYLE = 0;
    /**
     * Default style for character cells.
     */
    protected final static int DEFAULT_STYLE = 1;
```

```java
/**
 * Selected style.
 */
protected final static int HIGHLIGHT_STYLE = 2;

/**
 * Highlight style for character cells.
 */
protected final static int SELECT_STYLE = 3;

/**
 * Numbers style.
 */
protected final static int NUMBERS_STYLE = 4;

/**
 * Width of the component, used for resizing with mouse.
 */
protected int width;

/**
 * New width of the component, used for resizing with mouse.
 */
protected int newWidth;

/**
 * Is <code>true</code> when the component is being resized.
 */
protected boolean resizing = false;

/**
 * Used for holding the elements.
 */
protected Vector cells;

/**
 * Used for holding components that represent numbers.
 */
protected Vector numbers;

/**
 * Current string.
 */
protected String string;

/**
 * Cells styleset.
 */
protected ShapeStyle[] styleSet;

/**
 * Used in double-buffering.
 */
protected Image buffer;

/**
 * Begin index of the highlighted part of the string.
 */
protected int highlightBeginIndex;

/**
 * End index of the highlighted part of the string.
 */
protected int highlightEndIndex;

/**
 * Begin index of the selected part of the string.
 */
protected int selectBeginIndex;

/**
 * End index of the selected part of the string.
  */
protected int selectEndIndex;
```

```java
/**
 * Parent applet.
 */
protected UkkonenVisualizer applet;

/**
 * Default constructor.
 *
 * @param styleSet cells styleset.
 * @param showNumbers must be <code>true</code> if we need to show the numbers of the
 *     cells.
 */
public VerticalStringVisualizer(UkkonenVisualizer applet, ShapeStyle[] styleSet,
    boolean showNumbers) {
    setLayout(null);

    this.styleSet = styleSet;
    this.applet = applet;

    this.addComponentListener(new ComponentAdapter() {
        public void componentResized(ComponentEvent e) {
            VerticalStringVisualizer.this.resize();
        }
    });

    setString("", showNumbers);

    addMouseMotionListener(new StringMouseMotionAdapter());
    addMouseListener(new StringMouseAdapter());
}

/**
 * Sets string.
 *
 * @param string new string.
 * @param showNumbers must be <code>true</code> if we need to show the numbers.
 */
public void setString(String string, boolean showNumbers) {
    this.string = string;
    removeAll();

    cells = new Vector();
    numbers = new Vector();
    for (int i = 0; i < string.length(); i++) {
            Rect elem = new Rect(styleSet, string.substring(i, i + 1));
        elem.setStyle(DEFAULT_STYLE);
            cells.addElement(elem);

        add(elem);

        if (showNumbers) {
            Rect number = new Rect(styleSet, String.valueOf(i + 1));
            number.setStyle(NUMBERS_STYLE);
            numbers.addElement(number);

            add(number);
        }
    }

    resize();
    repaint();
}

/**
 * Returnes current string.
 *
 * @return string current string.
 */
public String getString() {
    return string;
}

/**
 * Higlights cells from <code>beginIndex</code> to
 * <code>endIndex - 1</code> (inclusive).
 *
```

```java
 * @param beginIndex
 * @param endIndex
 */
public void highlight(int beginIndex, int endIndex) {
    highlightBeginIndex = beginIndex;
    highlightEndIndex = endIndex;

    for (int i = 0; i < cells.size(); i++) {
        ((Rect) cells.elementAt(i)).setStyle(DEFAULT_STYLE);
    }
    for (int i = Math.max(0, beginIndex); i < Math.min(endIndex, cells.size()); i++)
         {
        ((Rect) cells.elementAt(i)).setStyle(HIGHLIGHT_STYLE);
    }
    for (int i = Math.max(0, selectBeginIndex);
         i < Math.min(selectEndIndex, cells.size()); i++) {
        ((Rect) cells.elementAt(i)).setStyle(SELECT_STYLE);
    }

    repaint();
}

public void select(int beginIndex, int endIndex) {
    selectBeginIndex = beginIndex;
    selectEndIndex = endIndex;

    for (int i = 0; i < cells.size(); i++) {
        ((Rect) cells.elementAt(i)).setStyle(DEFAULT_STYLE);
    }
    for (int i = Math.max(0, highlightBeginIndex);
         i < Math.min(highlightEndIndex, cells.size()); i++) {
        ((Rect) cells.elementAt(i)).setStyle(HIGHLIGHT_STYLE);
    }
    for (int i = Math.max(0, beginIndex); i < Math.min(endIndex, cells.size()); i++)
         {
        ((Rect) cells.elementAt(i)).setStyle(SELECT_STYLE);
    }

    repaint();
}

/**
 * Resizes the component. Must be called every time after the
 * component has been resized.
 */
public void resize() {
    int n = cells.size();
    Dimension size = getSize();

    int hskip = (int) Math.round((double) size.width / 8);
    int vskip = 10;

    int height, numWidth = 0;

    if (numbers.size() > 0) {
        height = (int) Math.min(Math.round((double) (size.height - 2 * vskip) / n),
                Math.round((double) (size.width - 2 * hskip) * 5 / 8));
        numWidth =  (int) Math.round((double) 3 / 5 *  height);
    } else {
        height = (int) Math.min(Math.round((double) (size.height - 2 * vskip) / n),
                Math.round((double) (size.width - 2 * hskip)));
    }

    int vgap = (size.height - height * n) / 2;

    for (int i = 0; i < n; i++) {
        Rect rect = (Rect) cells.elementAt(i);
        rect.setBounds(hskip, vgap + i * height, height + 1, height + 1);
        rect.adjustFontSize();

        if (numbers.size() > 0) {
            rect = (Rect) numbers.elementAt(i);
            rect.setBounds(hskip + width, vgap + i * height, height + 1, numWidth
                 + 1);
            rect.adjustFontSize();
        }
```

```
    }
}

/**
 * Paints the string tree.
 *
 * @param g {@link java.awt.Graphics} on which to paint.
 */
public void paint(Graphics g) {
    int width = getSize().width;
    int height = getSize().height;

    if (width > 0 && height > 0) {
        if (buffer == null) {
            buffer = this.createImage(width, height);
        }

        Graphics bg = buffer.getGraphics();

        if (styleSet[VISUALIZER_STYLE].getFillStatus()) {
            bg.setColor(styleSet[VISUALIZER_STYLE].getFillColor());
        }
        bg.fillRect(0, 0, width, height);

        super.paint(bg);

        if (styleSet[VISUALIZER_STYLE].getBorderStatus()) {
            bg.setColor(styleSet[VISUALIZER_STYLE].getBorderColor());
            bg.drawLine(0, 0, 0, height);
        }

        bg.dispose();

        g.drawImage(buffer, 0, 0, null);
    }
}

/**
 * Invalidates this component.
 */
public void invalidate() {
    super.invalidate();
    buffer = null;
}

/**
 * Updates the component.
 *
 * @param g {@link Graphics} on which to paint.
 */
public void update(Graphics g) {
    paint(g);
}

/**
 * Supporting class for handling mouse motion events for the string visualizer.
 *
 * @author  Igor Ahmetov
 * @version $Id$
 */
class StringMouseMotionAdapter extends MouseMotionAdapter {
    /**
     * Handles mouse dragging.
     *
     * @param e mouse event.
     */
    public void mouseDragged(MouseEvent e) {
        if (resizing) {
            if (width - e.getX() > 5 && width - e.getX() < getParent().getSize().
                width - 5)
                newWidth = width - e.getX();
        }
    }

    /**
     * Handles mouse movement.
```

```java
         *
         * @param e mouse event.
         */
        public void mouseMoved(MouseEvent e) {
            if (e.getX() < 10) {
                setCursor(new Cursor(Cursor.E_RESIZE_CURSOR));
            } else {
                setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
            }
        }
    }

    /**
     * Supporting class for handling mouse events for the string visualizer.
     *
     * @author  Igor Ahmetov
     * @version $Id$
     */
    class StringMouseAdapter extends MouseAdapter {
        /**
         * Handles mouse clicks.
         *
         * @param e mouse event.
         */
        public void mousePressed(MouseEvent e) {
            if (e.getX() < 10) {
                resizing = true;
                newWidth = width = getSize().width;
            }
        }

        /**
         * Handles mouse clicks.
         * @param e mouse event.
         */
        public void mouseReleased(MouseEvent e) {
            if (resizing) {
                resizing = false;
                setSize(newWidth, getSize().height);
                applet.resizeStringVisualizers();
                getParent().validate();
            }
        }
    }
}
```