# Алгоритмические проблемы в системах распознавания речи

Мальковский Николай Владимирович
Старший исследователь
Группа STT

ТИНЬКОФФ

ТИНЬКОФФ

# Sequence to sequence

- Пусть $\mathcal{A}, \mathcal{B}$ – некоторые множества, $\mathcal{A}^*, \mathcal{B}^*$ – множество последовательностей над $\mathcal{A}, \mathcal{B}$, некоторая функция $f : \mathcal{A}^* \to \mathcal{B}^*$

ТИНЬКОФФ

# Sequence to sequence

- Пусть $\mathcal{A}, \mathcal{B}$ – некоторые множества, $\mathcal{A}^*, \mathcal{B}^*$ – множество последовательностей над $\mathcal{A}, \mathcal{B}$, некоторая функция $f : \mathcal{A}^* \to \mathcal{B}^*$
  - Хотим уметь моделировать $f$ в виде некоторого распределения $P(a^*, b^*)$, $a^* \in \mathcal{A}^*, b^* \in \mathcal{B}^*$

ТИНЬКОФФ

# Sequence to sequence

- Пусть $\mathcal{A}, \mathcal{B}$ – некоторые множества, $\mathcal{A}^*, \mathcal{B}^*$ – множество последовательностей над $\mathcal{A}, \mathcal{B}$, некоторая функция $f : \mathcal{A}^* \to \mathcal{B}^*$
  - Хотим уметь моделировать $f$ в виде некоторого распределения $P(a^*, b^*)$, $a^* \in \mathcal{A}^*, b^* \in \mathcal{B}^*$
  - Хотим дополнительно для фиксированной входной последовательности $a^* \in \mathcal{A}^*$ уметь быстро находить

$$\text{argmax}_{b^* \in \mathcal{B}^*} P(b^*|a^*)$$

ТИНЬКОФФ

# Sequence to sequence

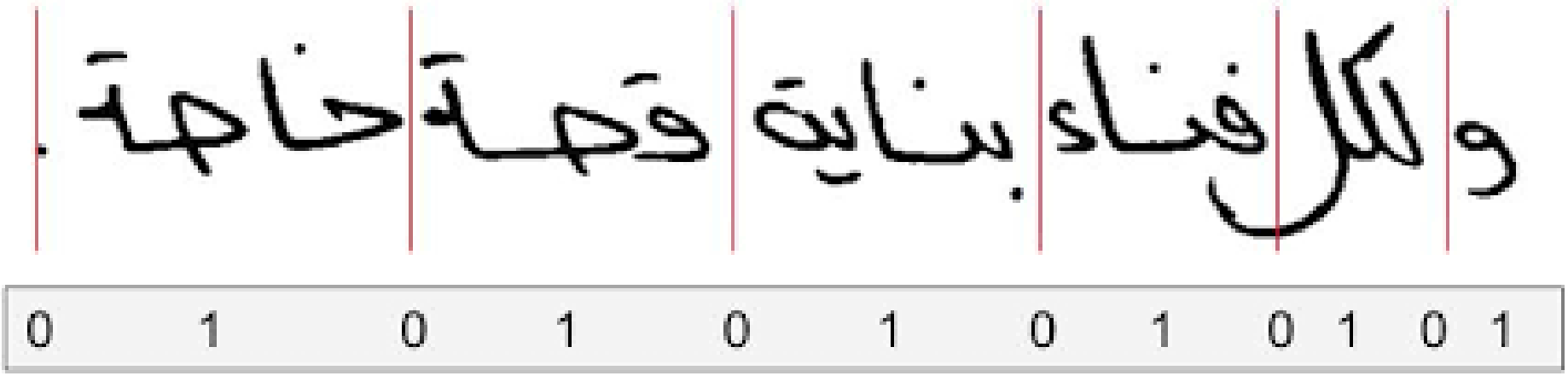
- Пусть $\mathcal{A}, \mathcal{B}$ – некоторые множества, $\mathcal{A}^*, \mathcal{B}^*$ – множество последовательностей над $\mathcal{A}, \mathcal{B}$, некоторая функция $f : \mathcal{A}^* \to \mathcal{B}^*$
  - Хотим уметь моделировать $f$ в виде некоторого распределения $P(a^*, b^*)$, $a^* \in \mathcal{A}^*, b^* \in \mathcal{B}^*$
  - Хотим дополнительно для фиксированной входной последовательности $a^* \in \mathcal{A}^*$ уметь быстро находить

$$\text{argmax}_{b^* \in \mathcal{B}^*} P(b^*|a^*)$$

- ASR, TTS, machine translation, ...

ТИНЬКОФФ

# Sequence to sequence




ТИНЬКОФФ

# Seq-2-seq в ASR

- Традиционный подход: как-то смоделировать отдельно $P_a(b \mid i, a^*)$ – акустическую модель ($i$ – номер фрейма) и $P_b(b^*)$ – языковую модель

ТИНЬКОФФ

# Seq-2-seq в ASR

- Традиционный подход: как-то смоделировать отдельно $P_a(b \mid i, a^*)$ – акустическую модель ($i$ – номер фрейма) и $P_b(b^*)$ – языковую модель
- Умение их комбинировать и быстро искать argmax – сложная алгоритмическая задача

ТИНЬКОФФ

# Seq-2-seq в ASR

- Традиционный подход: как-то смоделировать отдельно $P_a(b \mid i, a^*)$ – акустическую модель ($i$ – номер фрейма) и $P_b(b^*)$ – языковую модель
- Умение их комбинировать и быстро искать argmax – сложная алгоритмическая задача
- Обучающие данные имеют вид пар $(a^*, b^*)$, как на таких данных обучать $P_a(b \mid i, a^*)$?

ТИНЬКОФФ

# Sequence to sequence


ТИНЬКОФФ

# Dynamic Time Wrapping

- Идея: делаем несколько эталонных записей, при распознавании просто сравниваем с тем, что есть

# Dynamic Time Wrapping

- Идея: делаем несколько эталонных записей, при распознавании просто сравниваем с тем, что есть
- Как сравнивать? Гипотеза: если на записях говорят одно и тоже, то участки одной из них можно вырезать/ускорить/замедлить так, чтобы получить хорошее пофреймовое соответствие

ТИНЬКОФФ

# Dynamic Time Wrapping

- Идея: делаем несколько эталонных записей, при распознавании просто сравниваем с тем, что есть
- Как сравнивать? Гипотеза: если на записях говорят одно и тоже, то участки одной из них можно вырезать/ускорить/замедлить так, чтобы получить хорошее пофреймовое соответствие
- Формально – хотим найти монотонное выравнивание такое, что максимизирует величину

$$\max_{j_1 \le \dots \le j_m} \prod_{i=1}^{n} S(a^*[i], \hat{a}^*[j_i])$$

для некоторой пофреймовой функции схожести $S(\cdot, \cdot)$

ТИНЬКОФФ

# Dynamic Time Wrapping

- Идея: делаем несколько эталонных записей, при распознавании просто сравниваем с тем, что есть
- Как сравнивать? Гипотеза: если на записях говорят одно и тоже, то участки одной из них можно вырезать/ускорить/замедлить так, чтобы получить хорошее пофреймовое соответствие
- Формально – хотим найти монотонное выравнивание такое, что максимизирует величину

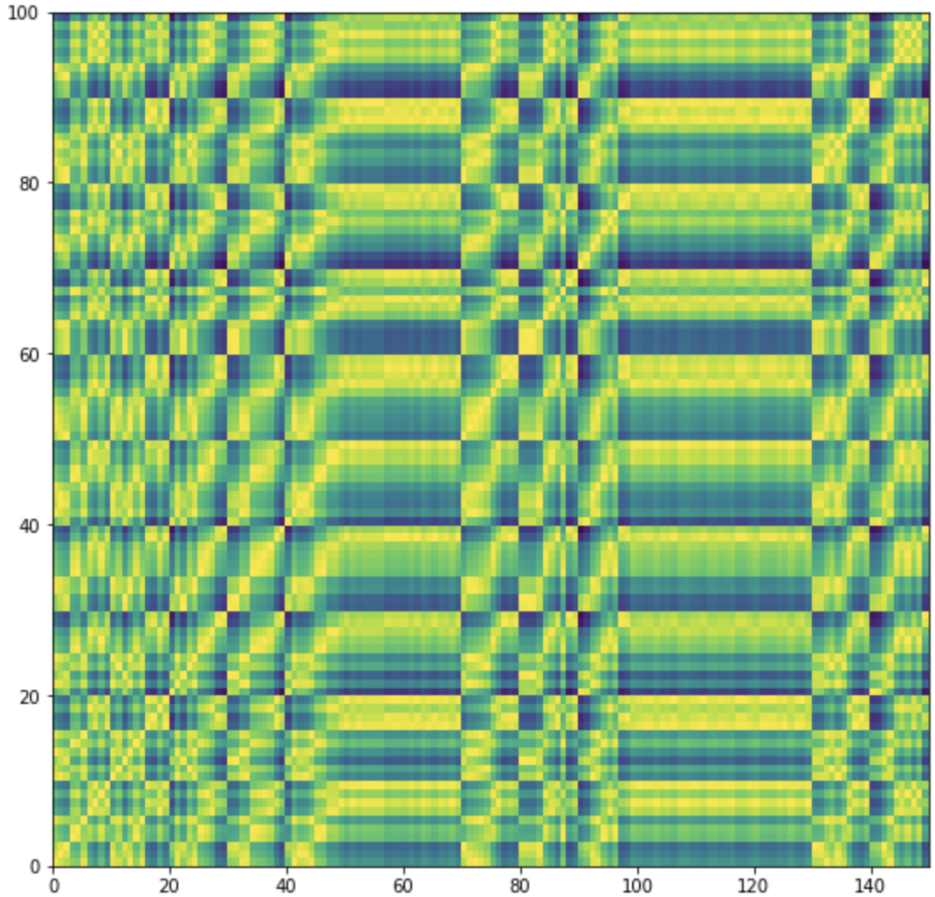
$$\max_{j_1 \leq \dots \leq j_m} \prod_{i=1}^{n} S(a^*[i], \hat{a}^*[j_i])$$

  для некоторой пофреймовой функции схожести $S(\cdot, \cdot)$
- Можно вычислить эффективно с помощью динамического программирования за $\mathcal{O}(nm)$, где $n, m$ – длины записей. Этот алгоритм принято называть **Dynamic Time Wrapping**

ТИНЬКОФФ

# Dynamic time wrapping

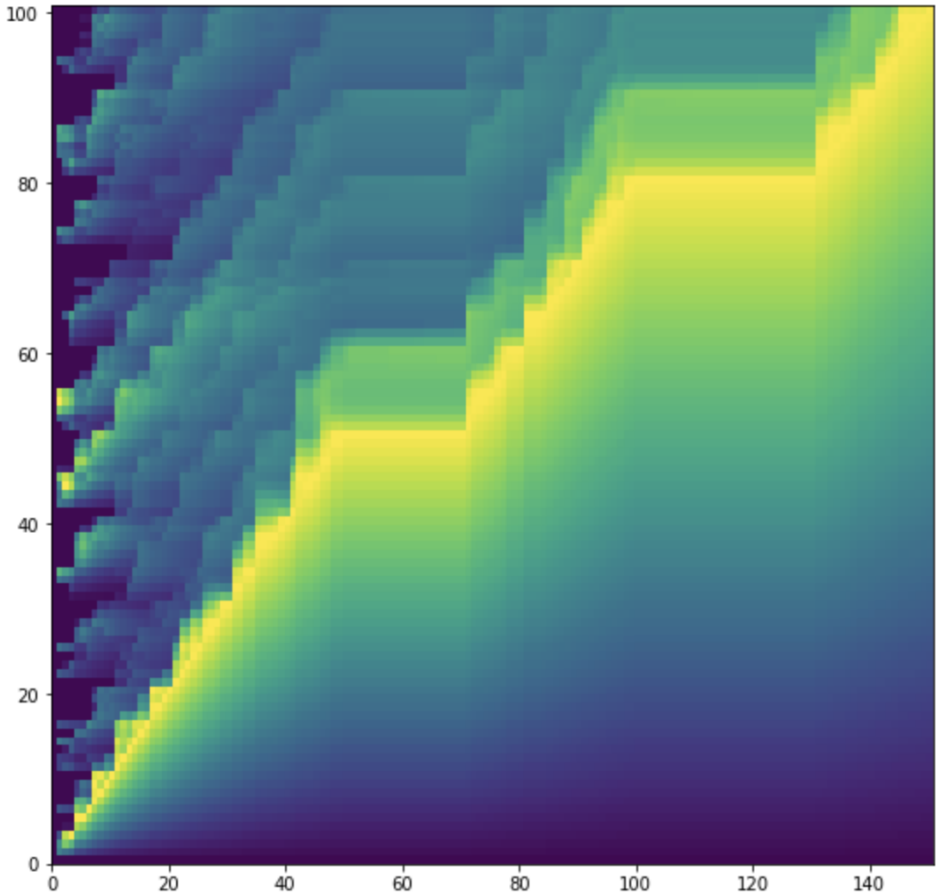
Матрица схожести фреймов




ТИНЬКОФФ

# Dynamic time wrapping

Выравнивание DTW




ТИНЬКОФФ

# Dynamic time wrapping

- Выравнивание последовательностей длин $n, m$ требует $\mathcal{O}(nm)$ вычислений и $\mathcal{O}(\min(n, m))$ дополнительной памяти
- Большинство промежуточных вычислений нерелевантны, нужны только для гарантии оптимальности
- Очень схоже с задачей о кратчайших путях, где есть в том числе и точные эвристики, позволяющие добиться $o(nm)$

ТИНЬКОФФ

# Пример с расстоянием Левенштейна

Базовый код для вычисления расстояния Левенштейна двух строк $s, t$ за $\mathcal{O}(nm)$ времени и памяти

```
def levenstein(i, j, s, t, cache):
    if i == 0 or j == 0:
        return max(i, j)
    if (i, j) in cache:
        return cache[(i, j)]

    cache[(i, j)] = min(
        levenstein(i - 1, j - 1, s, t, cache)
            + (0 if s[i - 1] == t[j - 1] else 1),
        levenstein(i - 1, j, s, t, cache) + 1,
        levenstein(i, j - 1, s, t, cache) + 1)
    return cache[(i, j)]

# levenstein(len(s), len(t), s, t, dict())
```

ТИНЬКОФФ

# Расстоянием Левенштейна

Допустим мы предполагаем, что расстояние не больше $k$, следующий код считает расстояние Левенштейна точно только если оно меньше $k$ за $\mathcal{O}((m+n)k)$ времени и памяти

```
def levenstein(i, j, s, t, k, cache):
    if i == 0 or j == 0:
        return max(i, j)
    if abs(i - j) >= k:
        return k
    if (i, j) in cache:
        return cache[(i, j)]

    cache[(i, j)] = min(
        levenstein(i - 1, j - 1, s, t, k, cache)
            + 0 if s[i - 1] == t[j - 1] else 1,
        levenstein(i - 1, j, s, t, k, cache) + 1
        levenstein(i, j - 1, s, t, k, cache))
    return cache[(i, j)]
```

ТИНЬКОФФ

# Расстоянием Левенштейна

"Онлайн" вариант расстояния Левенштейна за $\mathcal{O}(nm)$ времени и $\mathcal{O}(m)$ памяти.

```
def levenstein_online(s, t, states = None):
    distances = [i for i in range(len(t) + 1)]

    for k, c in enumerate(s):
        # initialization with deletion of current symbol
        new_distances = [d + 1 for d in distances]
        # Subs or corrects
        for i, d in enumerate(new_distances[1:]):
            distance_sub = distances[i] + (0 if c == t[i] else 1)
            new_distances[i + 1] = min(distance_sub, d)
        # Insertions
        for i, d in enumerate(new_distances[:-1]):
            new_distances[i + 1] = min(new_distances[i + 1],
                                       new_distances[i] + 1)

        distances = new_distances
```

ТИНЬКОФФ

# Расстояние Левенштейна

Усеченный онлайн вариант: $\mathcal{O}(nk)$ времени и $\mathcal{O}(k)$ памяти

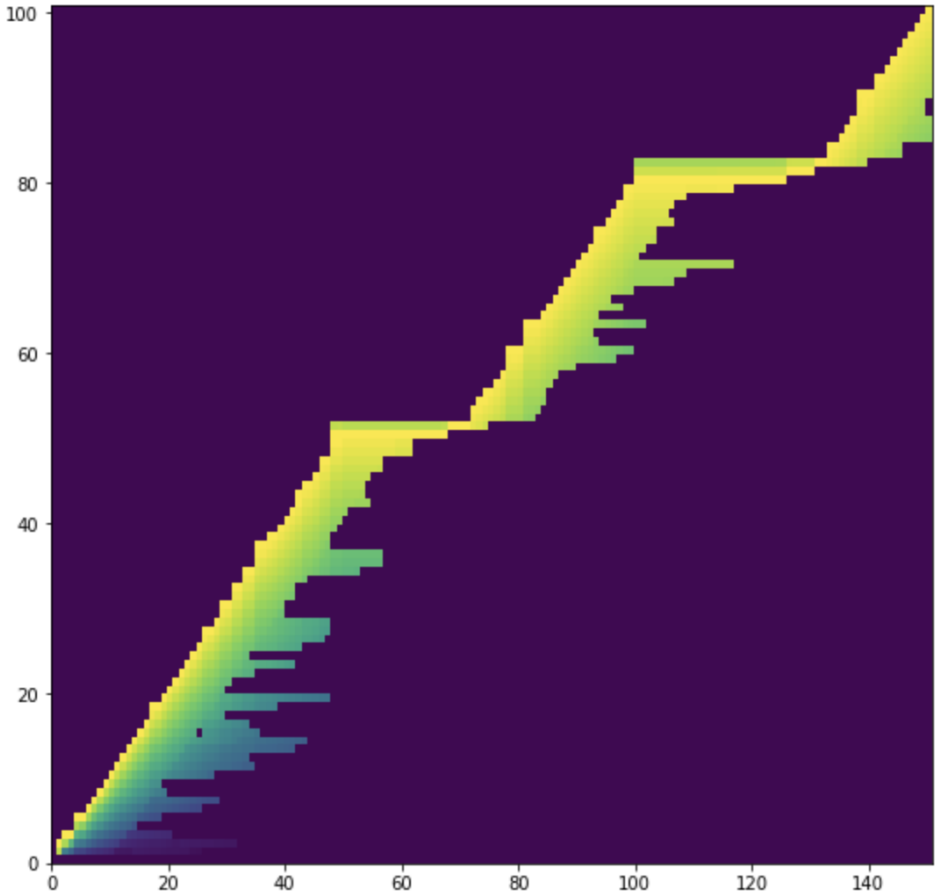
```
def levenstein(s, t, k):
    # each entry is a position in t -> distance dictionary pair
    entries = {i: i for i in range(len(t))}

    for c in s:
        new_entries = {pos: d + 1 for pos, d in entries.items()}
        for pos, d in  entries.items():
            if pos >= len(t):
                continue
            distance_sub = d + (0 if c == t[pos] else 1)
            if pos + 1 not in new_entries:
                new_entries[pos + 1] = distance_sub
            new_entries[pos + 1] = min(distance_sub, new_entries[pos + 1])
        prev_pos = -1
        for pos, d in enumerate(new_entries.items()):
            if prev_pos == -1:
                prev_pos = pos
                continue
            new_entries[pos] = min(new_entries[pos],
                                   new_entries[prev_pos] + pos - prev_pos)
        entries = GetTopKEntries(new_entries, k)

    return entries[len(t)]
```

ТИНЬКОФФ

# Усеченный DTW




ТИНЬКОФФ

# Усеченный DTW

- Последний вариант можно встретить в литературе под названиями *token passing algorithm* и *[Viterbi] beam search*
- Алгоритм может быть разбит на несколько шагов, на каждом из которых по очереди рассматривается очередой символ первой последовательности; это дает возможность расширять эту последовательность без полного пересчета
- Можно обобщить таким образом, что вместо второй последовательности используется более сложная структура
- Мы научились считать расстояние Левенштейна/выравнивание, а как его восстановить? (об этом чуть позже)

ТИНЬКОФФ

# Связь с задачей о кратчайших путях

Хотим найти в графе путь минимальной суммарной длины от $s$ в $t$

ТИНЬКОФФ

# Связь с задачей о кратчайших путях

Хотим найти в графе путь минимальной суммарной длины от $s$ в $t$

- Дейкстра: перебираем вершины в порядке увеличения $d(s, v)$, останавливаемся когда дойдем до $t$

ТИНЬКОФФ

# Связь с задачей о кратчайших путях

Хотим найти в графе путь минимальной суммарной длины от $s$ в $t$

- Дейкстра: перебираем вершины в порядке увеличения $d(s, v)$, останавливаемся когда дойдем до $t$
- $A^*$: перебираем вершины в порядке увеличения $d(s, v) + h(v, t)$, где $h(u, v) \leq d(u, v)$ – вспомогательная эвристика

ТИНЬКОФФ

# Связь с задачей о кратчайших путях

Хотим найти в графе путь минимальной суммарной длины от $s$ в $t$

- Дейкстра: перебираем вершины в порядке увеличения $d(s, v)$, останавливаемся когда дойдем до $t$
- $A^*$: перебираем вершины в порядке увеличения $d(s, v) + h(v, t)$, где $h(u, v) \leq d(u, v)$ – вспомогательная эвристика
- Идеальная ситуация: если $h(u, v) = d(u, v)$ и минимальный путь единственен, то $A^*$ просматривает только вершины на минимальном пути $s \to t$

ТИНЬКОФФ

# Связь с задачей о кратчайших путях

Хотим найти в графе путь минимальной суммарной длины от $s$ в $t$

- Дейкстра: перебираем вершины в порядке увеличения $d(s, v)$, останавливаемся когда дойдем до $t$
- $A^*$: перебираем вершины в порядке увеличения $d(s, v) + h(v, t)$, где $h(u, v) \leq d(u, v)$ – вспомогательная эвристика
- Идеальная ситуация: если $h(u, v) = d(u, v)$ и минимальный путь единственен, то $A^*$ просматривает только вершины на минимальном пути $s \to t$

ТИНЬКОФФ

# Связь с задачей о кратчайших путях

Хотим найти в графе путь минимальной суммарной длины от $s$ в $t$

- Дейкстра: перебираем вершины в порядке увеличения $d(s, v)$, останавливаемся когда дойдем до $t$
- $A^*$: перебираем вершины в порядке увеличения $d(s, v) + h(v, t)$, где $h(u, v) \leq d(u, v)$ – вспомогательная эвристика
- Идеальная ситуация: если $h(u, v) = d(u, v)$ и минимальный путь единственен, то $A^*$ просматривает только вершины на минимальном пути $s \to t$

Практические проблемы beam search:

- Чувствителен к скачкообразному поведению: если стоимость выравнивания размазана неравномерно, то мы рискуем потерять терять много хороших гипотез где-то посередине
- На практике используется очень много различных техник “сглаживания”, которые сделаны наподобие введении вспомогательной эвристики как в $A^*$

ТИНЬКОФФ

ТИНЬКОФФ

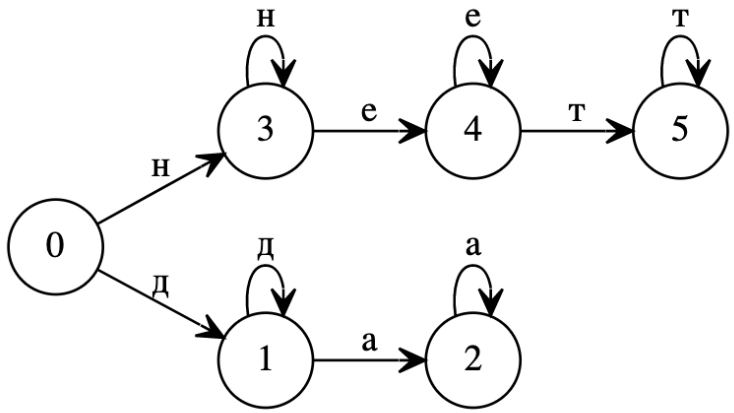
# Гибридные модели

Предположим, что у нас есть хорошая модель, которая умеет выдавать правдоподобия соответствия букв кириллицы фрейму звука (aka акустическая модель), рассмотрим следующий граф

ТИНЬКОФФ

# Гибридные модели

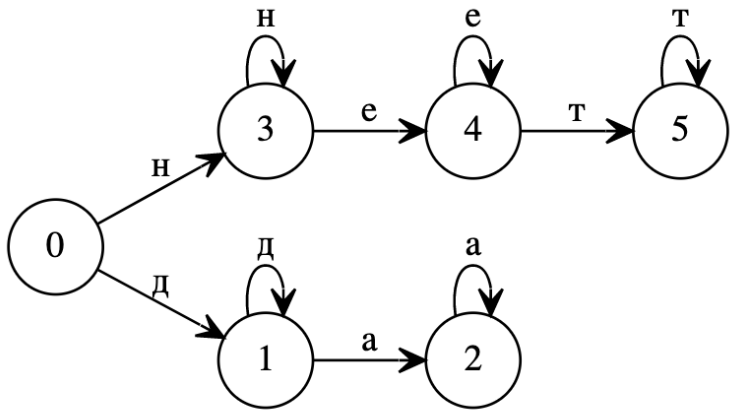
Предположим, что у нас есть хорошая модель, которая умеет выдавать правдоподобия соответствия букв кириллицы фрейму звука (aka акустическая модель), рассмотрим следующий граф




ТИНЬКОФФ

# Гибридные модели

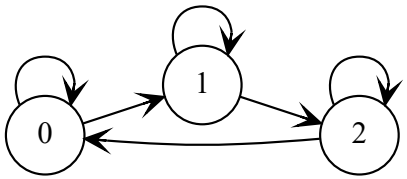
Предположим, что у нас есть хорошая модель, которая умеет выдавать правдоподобия соответствия букв кириллицы фрейму звука (aka акустическая модель), рассмотрим следующий граф

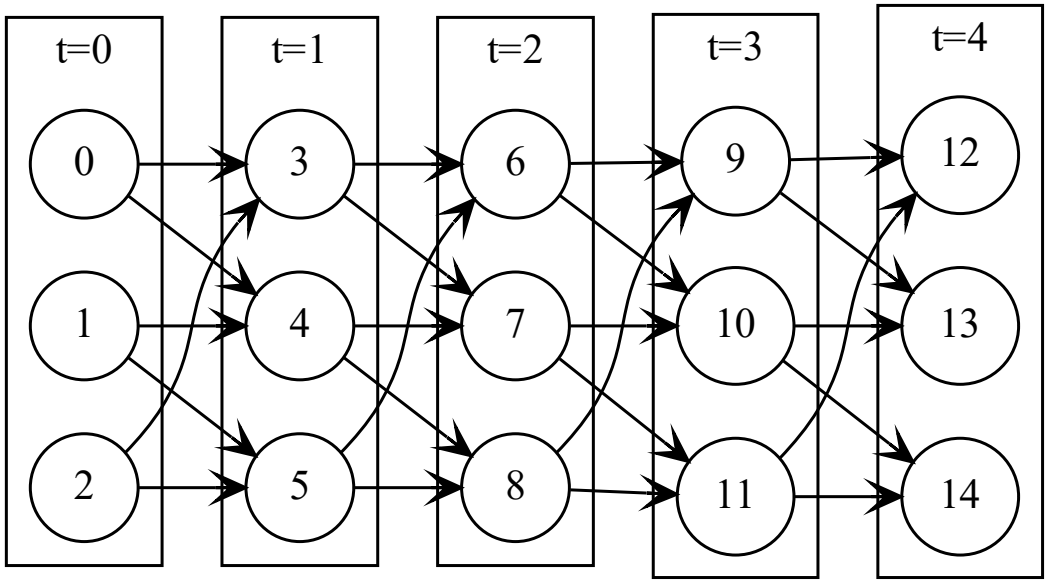



Можно проделать процедуру, подобную DTW, но вместо эталонной последовательности использовать этот граф, а весто функции схожести $S$ имеющуюся у нас акустическую модель, в данном конкретном случае это поможет оценить что более правдоподобно по произнесению “да” или “нет”

ТИНЬКОФФ

# Выравнивание в HMM




ТИНЬКОФФ

# Гибридные модели

- Такую конструкцию обычно называют “гибридной” моделью

ТИНЬКОФФ

# Гибридные модели

- Такую конструкцию обычно называют “гибридной” моделью
- Скрытая марковская модель (HMM) отвечает за структурную связь между различными акутическими единицами

ТИНЬКОФФ

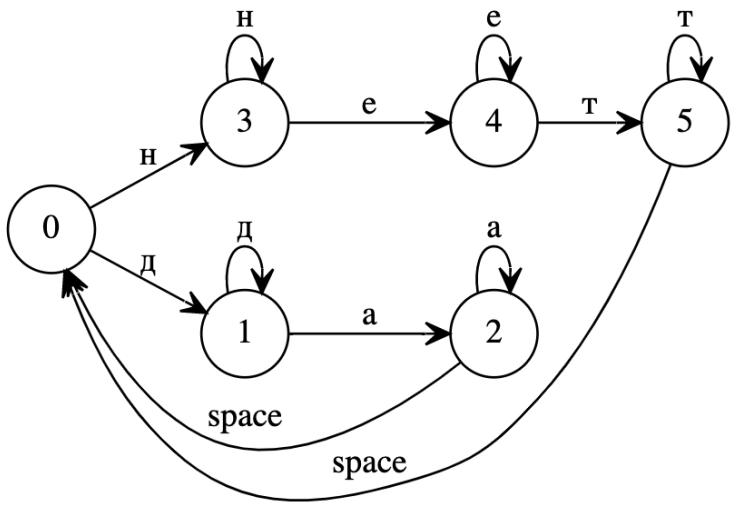
# Гибридные модели

- Такую конструкцию обычно называют “гибридной” моделью
- Скрытая марковская модель (HMM) отвечает за структурную связь между различными акутическими единицами
- Акустическая модель отвечает за соответствия фреймам звука акустическим единицам (раньше использовались *гауссовы смеси* (GMM), сейчас в основном нейронные сети (DNN))

ТИНЬКОФФ

# Минимальная часть HMM

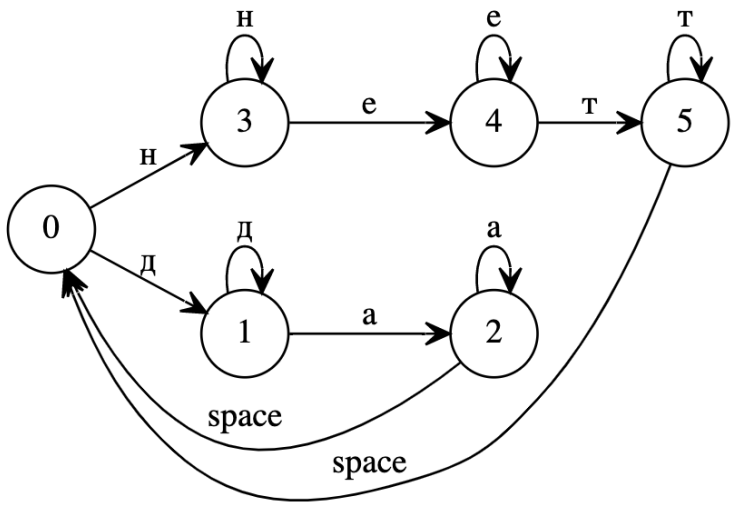
- Зацикленный лексикон




ТИНЬКОФФ

# Минимальная часть HMM

- Зацикленный лексикон

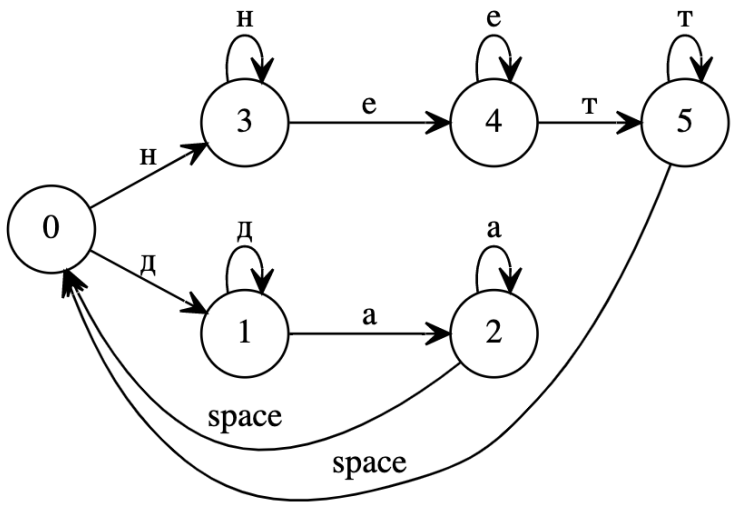



- Грамматика или языковая модель, описывающие последовательность слов, которые мы ожидаем услышать

ТИНЬКОФФ

# Минимальная часть HMM

- Зацикленный лексикон




- Грамматика или языковая модель, описывающие последовательность слов, которые мы ожидаем услышать
- Для представления можно использовать оптимизированные WFST (weighted finite state transducer)

ТИНЬКОФФ

# Схожесть с системами для игр

- Leela chess zero:
  - Monte-Carlo-Tree-Search: алгоритм усеченного поиска, используемый в пошаговых играх
  - Policy network: нейронная сеть, выдающая вероятность выигрыша для текущей позиции и рассматриваемого хода
  - Value network: нейронная сеть, выдающая выроятность выиграша для текущий позиции
- Stockfish:
  - Alpha-Beta-Search: другой алгоритм поиска, используемый в пошаговых играх
  - Position evaluation: Value network в Stockfish 14, более простые статистические модели и априорные знания в Stockfish $\leq 13$

ТИНЬКОФФ

# Обучение акустики: Baum-Welsh

Гибридные системы обычно используют специальный EM-алгоритм для HMM:

- Expectation step: построить выравнивание распознав текущей моделью
- Maximization step: переучить акустику с построенным выравниванием

ТИНЬКОФФ

# Обучение акустики: Baum-Welsh

Гибридные системы обычно используют специальный EM-алгоритм для HMM:

- Expectation step: построить выравнивание распознав текущей моделью
- Maximization step: переучить акустику с построенным выравниванием

Это алгоритм *Баума-Велша*. Транскрипция аудио используется для построения HMM под каждую конкретную аудиозапись например для “да нет наверное” самая простая HMM выглядила бы как-то так


ТИНЬКОФФ

# Множественное выравнивание vs одиночное

В алгоритме Баума-Велша построение выравнивания в E-шаге обычно происходит одним из двух способов:

# Множественное выравнивание vs одиночное

В алгоритме Баума-Велша построение выравнивания в E-шаге обычно происходит одним из двух способов:

- Одиночное: взять одно наиболее вероятное выравнивание

ТИНЬКОФФ

# Множественное выравнивание vs одиночное

В алгоритме Баума-Велша построение выравнивания в E-шаге обычно происходит одним из двух способов:

- Одиночное: взять одно наиболее вероятное выравнивание
  - \+ Можно найти с помощью beam search

# Множественное выравнивание vs одиночное

В алгоритме Баума-Велша построение выравнивания в E-шаге обычно происходит одним из двух способов:

- Одиночное: взять одно наиболее вероятное выравнивание
  - \+ Можно найти с помощью beam search
  - \- Обычно требует промежуточные стадии: flat start -> monophone GMM -> triphone GMM

ТИНЬКОФФ

# Множественное выравнивание vs одиночное

В алгоритме Баума-Велша построение выравнивания в E-шаге обычно происходит одним из двух способов:

- Одиночное: взять одно наиболее вероятное выравнивание
  - \+ Можно найти с помощью beam search
  - \- Обычно требует промежуточные стадии: flat start -> monophone GMM -> triphone GMM
- Множественное: взять все возможные выравнивания

ТИНЬКОФФ

# Множественное выравнивание vs одиночное

В алгоритме Баума-Велша построение выравнивания в E-шаге обычно происходит одним из двух способов:

- Одиночное: взять одно наиболее вероятное выравнивание
  - \+ Можно найти с помощью beam search
  - \- Обычно требует промежуточные стадии: flat start -> monophone GMM -> triphone GMM
- Множественное: взять все возможные выравнивания
  - \+ Вычисляется с помощью динамического программирования, его промежуточные значения могут быть использованы для backpropagation

ТИНЬКОФФ

# Множественное выравнивание vs одиночное

В алгоритме Баума-Велша построение выравнивания в E-шаге обычно происходит одним из двух способов:

- Одиночное: взять одно наиболее вероятное выравнивание
  - \+ Можно найти с помощью beam search
  - \- Обычно требует промежуточные стадии: flat start -> monophone GMM -> triphone GMM
- Множественное: взять все возможные выравнивания
  - \+ Вычисляется с помощью динамического программирования, его промежуточные значения могут быть использованы для backpropagation
  - \+ Можно обучать с нуля
    Hadian H. et al. End-to-end Speech Recognition Using Lattice-free MMI //Interspeech. – 2018. – С. 12-16.

ТИНЬКОФФ

# Множественное выравнивание vs одиночное

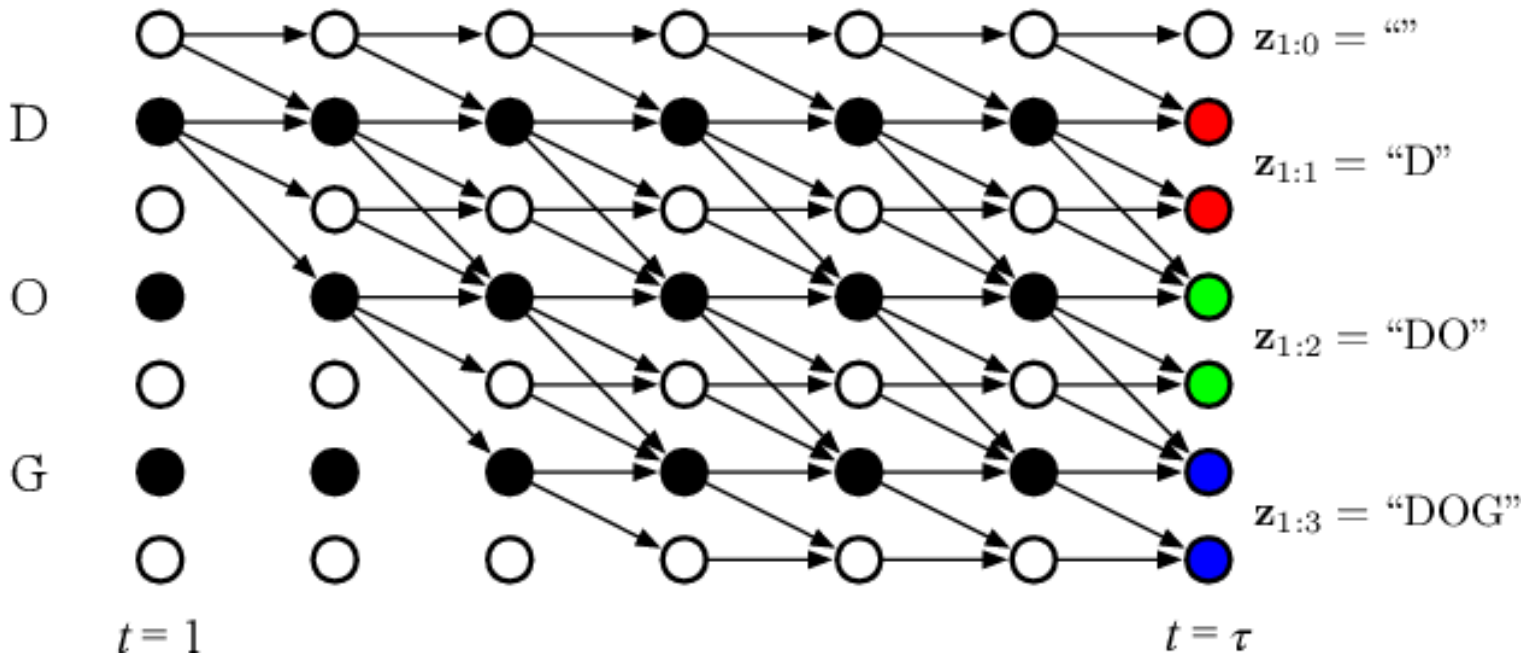
В алгоритме Баума-Велша построение выравнивания в E-шаге обычно происходит одним из двух способов:

- Одиночное: взять одно наиболее вероятное выравнивание
  - \+ Можно найти с помощью beam search
  - \- Обычно требует промежуточные стадии: flat start -> monophone GMM -> triphone GMM
- Множественное: взять все возможные выравнивания
  - \+ Вычисляется с помощью динамического программирования, его промежуточные значения могут быть использованы для backpropagation
  - \+ Можно обучать с нуля
    Hadian H. et al. End-to-end Speech Recognition Using Lattice-free MMI //Interspeech. – 2018. – С. 12-16.
  - \- Возникает квадратичная сложность по длине аудио

ТИНЬКОФФ

# Обучение акустики: CTC

Connectionist Temporal Classification (CTC): частный случай Баума-Велша с упрощенной HMM




Zeyer A. et al. CTC in the context of generalized full-sum HMM training //INTERSPEECH. – 2017. – С. 944-948.

ТИНЬКОФФ

1. Выравнивание и алгоритмы для его построения
2. Гибридные HMM модели
3. Структуры и алгоритмы для онлайн распознавания
4. Множественные результаты распознавания

ТИНЬКОФФ

# Восстановление гипотез в beam search

- Мы научились делать beam search с ограниченной памятью, но как найти итоговое выравнивание или результат распознавания?

ТИНЬКОФФ

# Восстановление гипотез в beam search

- Мы научились делать beam search с ограниченной памятью, но как найти итоговое выравнивание или результат распознавания?
- Простой вариант: храним всю историю вычислений

ТИНЬКОФФ

# Восстановление гипотез в beam search

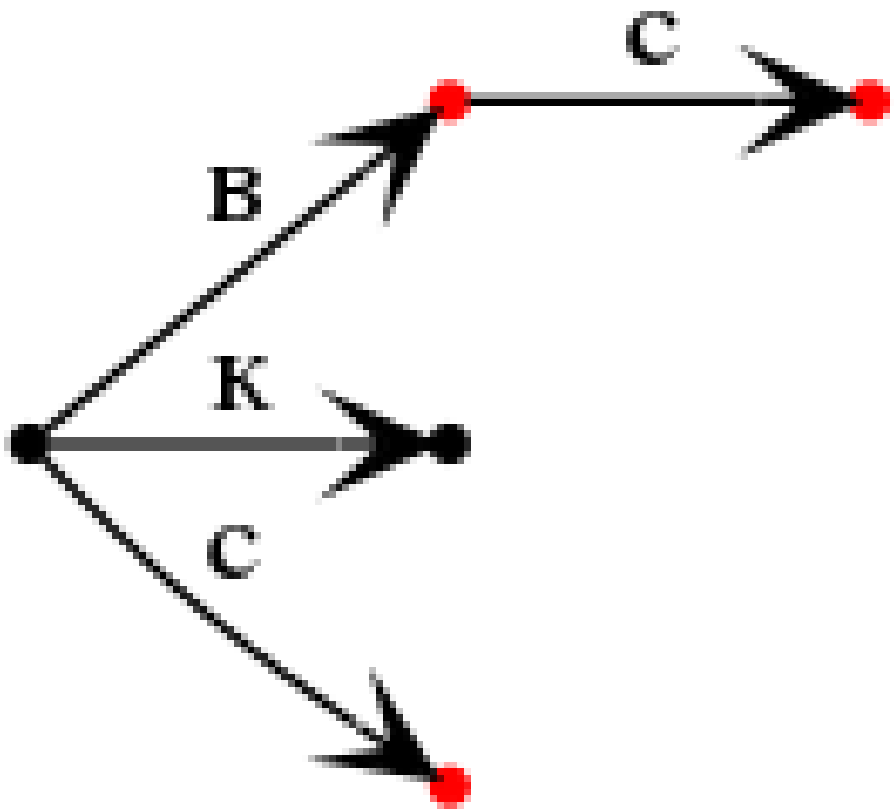
- Мы научились делать beam search с ограниченной памятью, но как найти итоговое выравнивание или результат распознавания?
- Простой вариант: храним всю историю вычислений
- Чуть сложнее: запоминаем только нужную

ТИНЬКОФФ

# Дерево гипотез

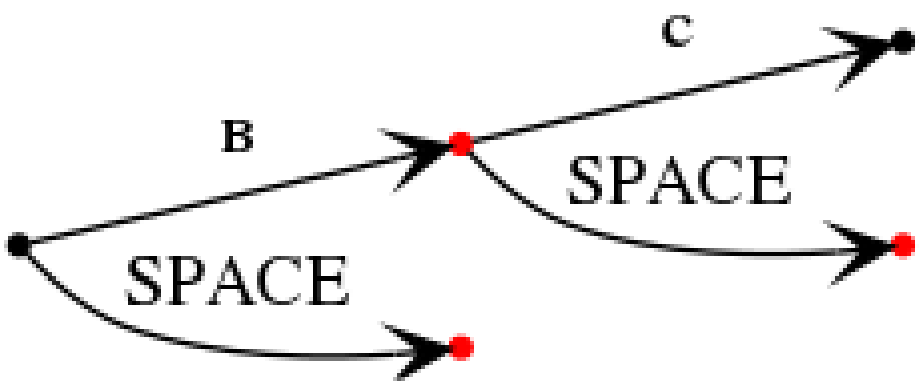
КОФФ

# Дерево гипотез



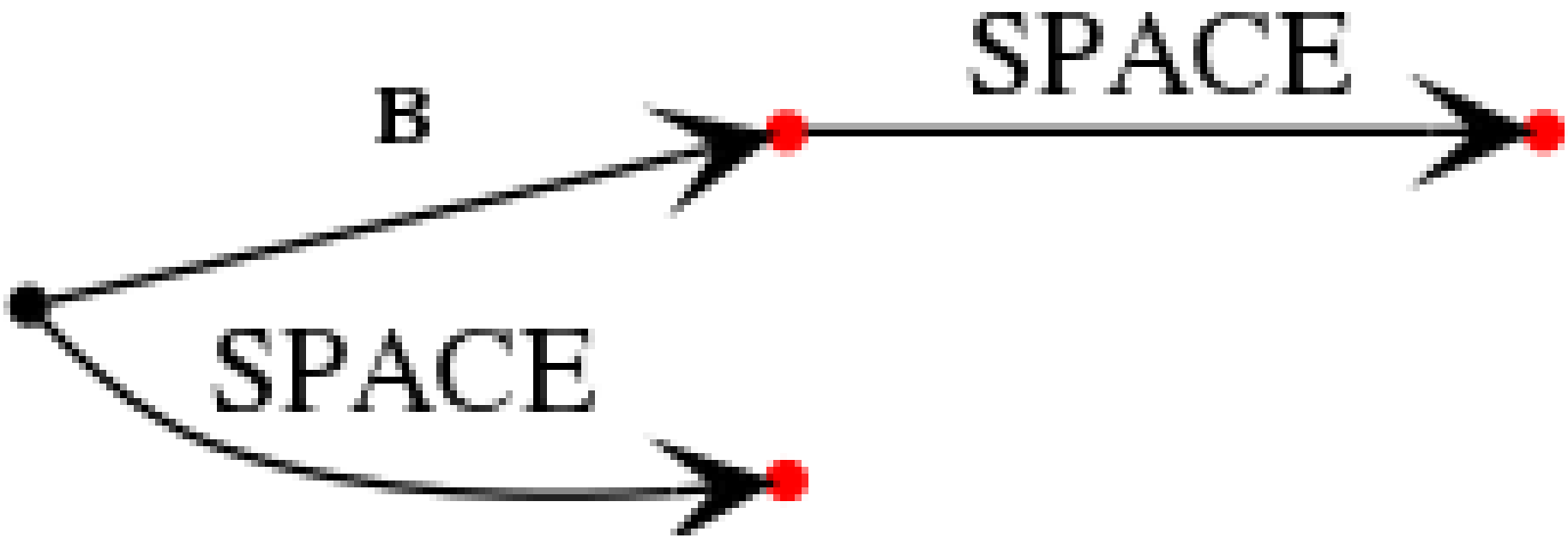

ТИНЬКОФФ

# Дерево гипотез



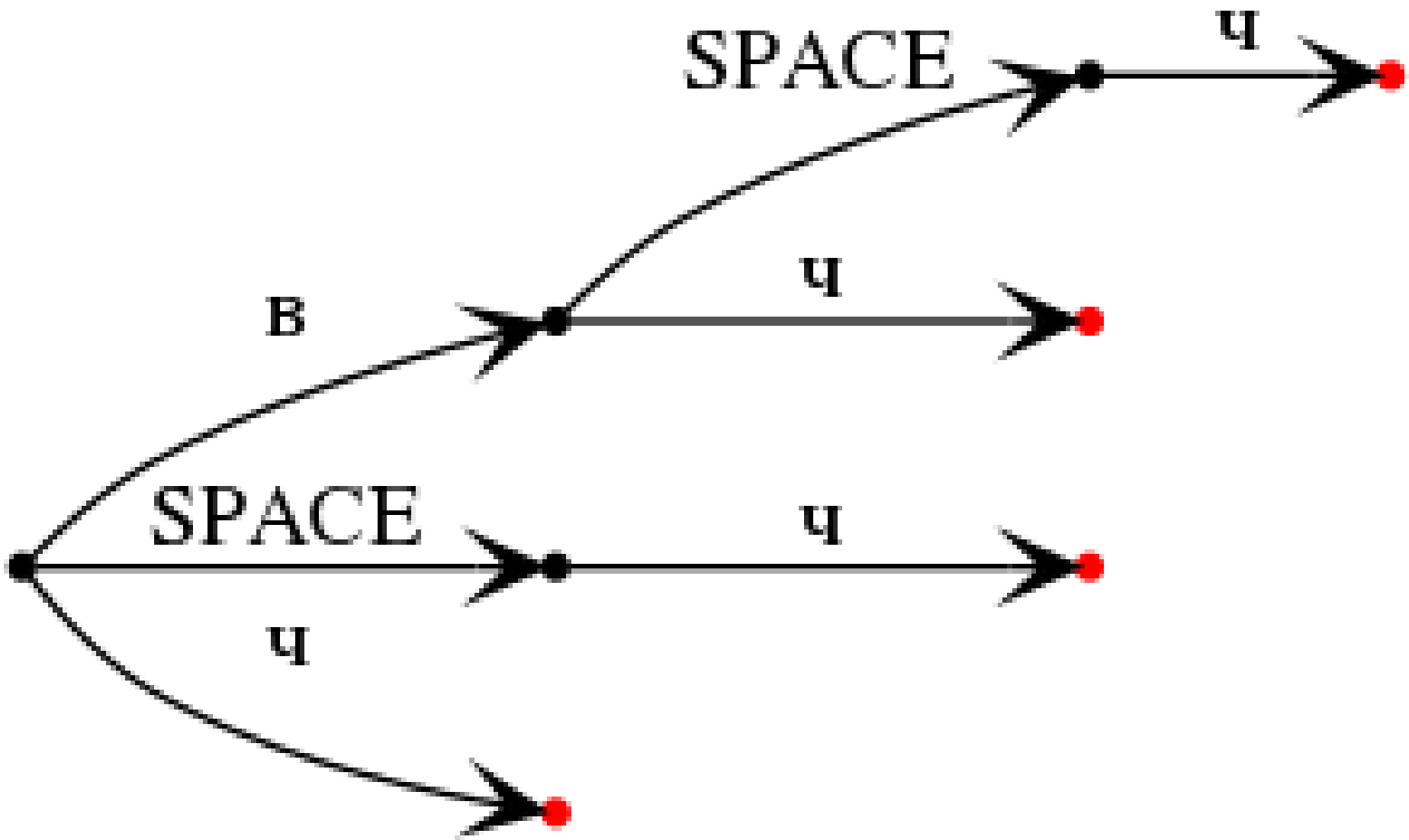

ТИНЬКОФФ

# Дерево гипотез

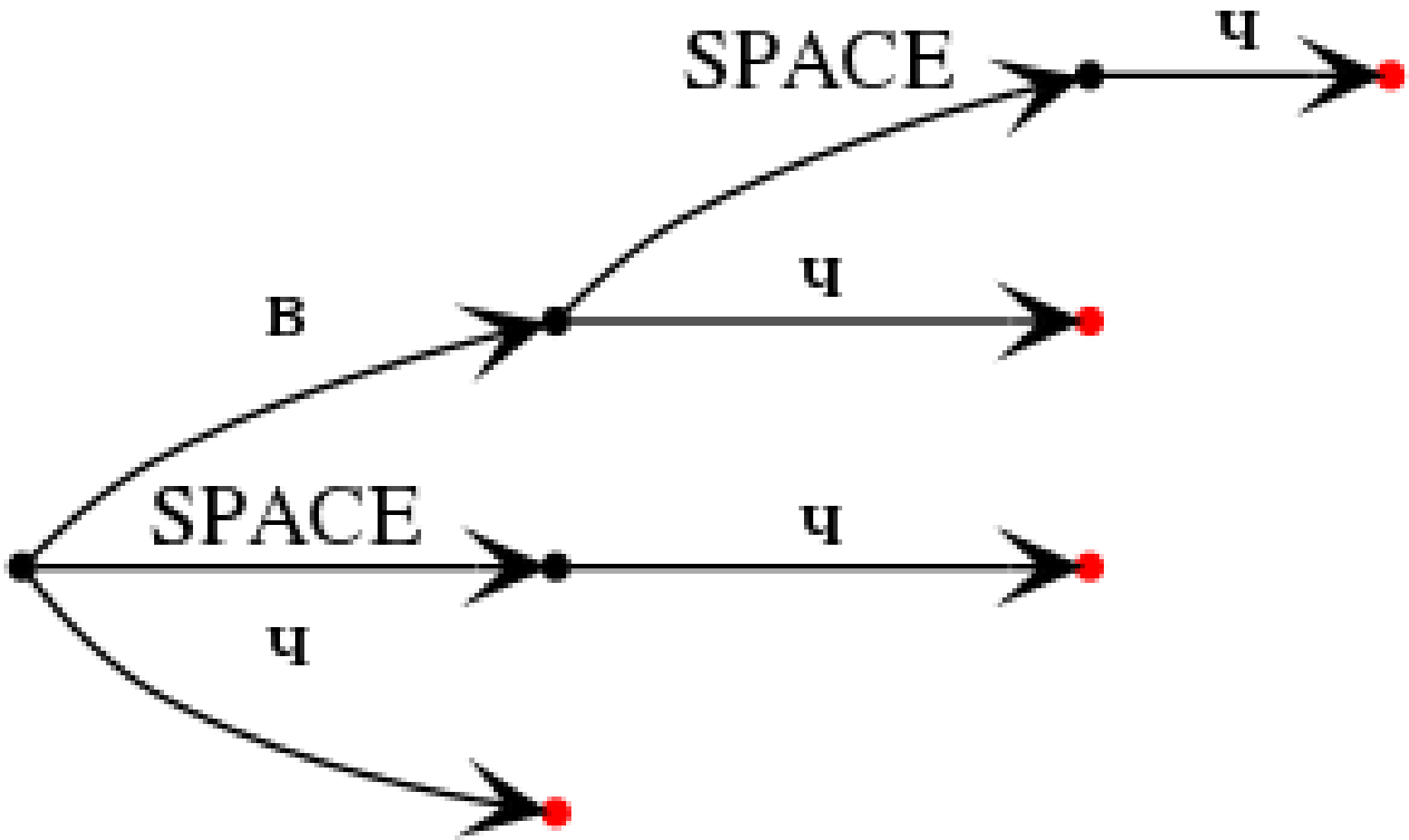



ТИНЬКОФФ

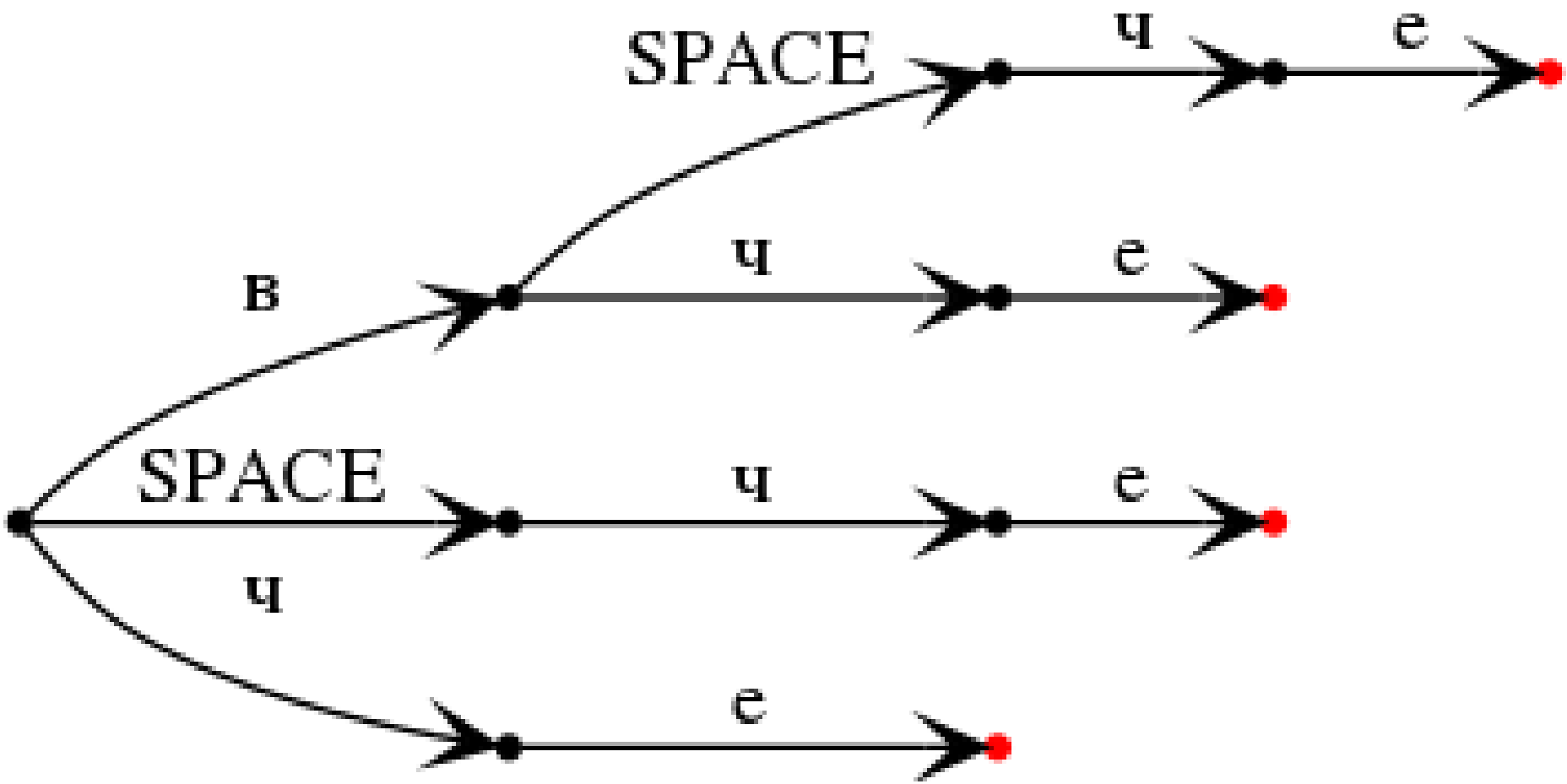
# Дерево гипотез



# Дерево гипотез

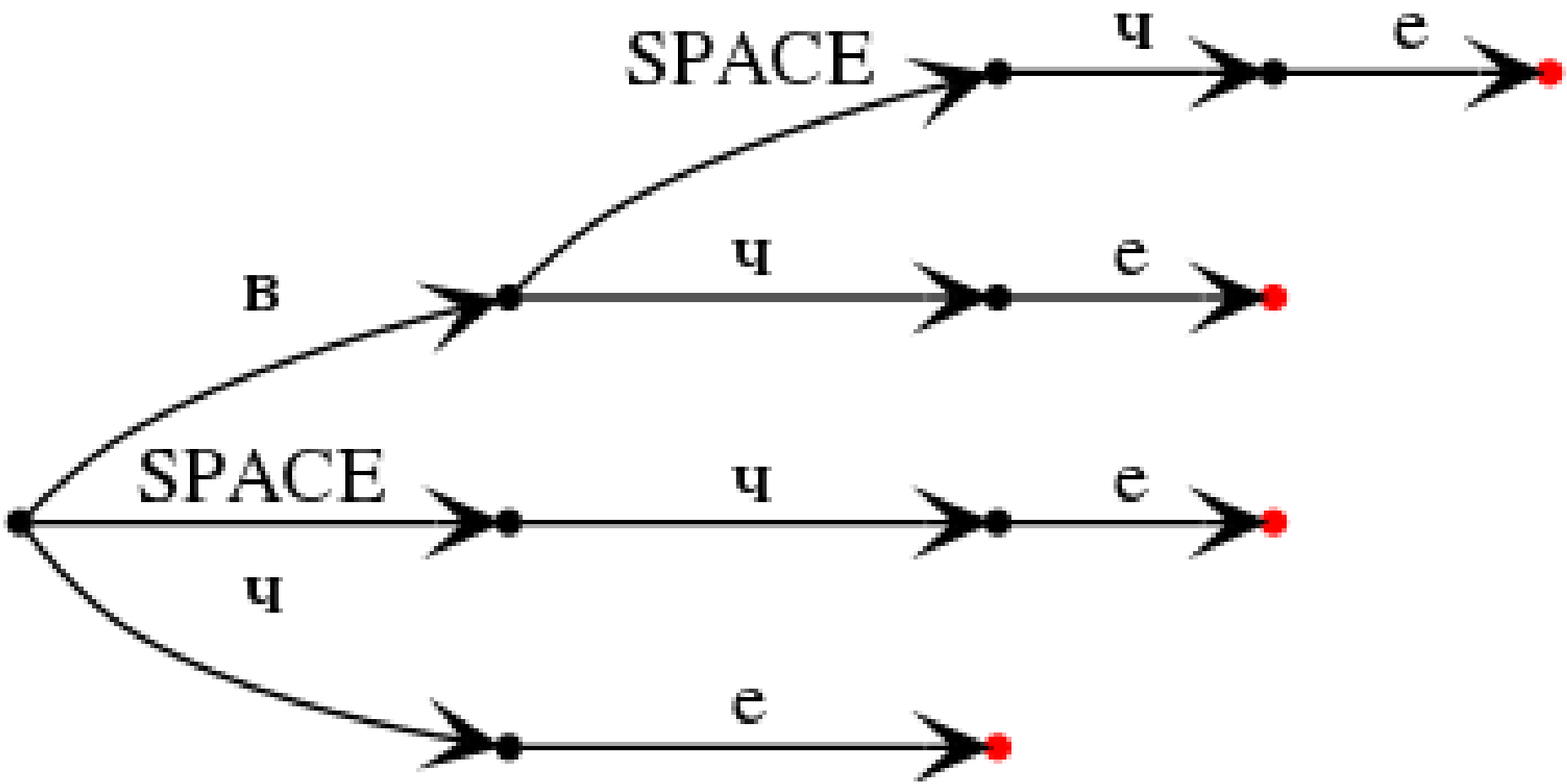


# Дерево гипотез



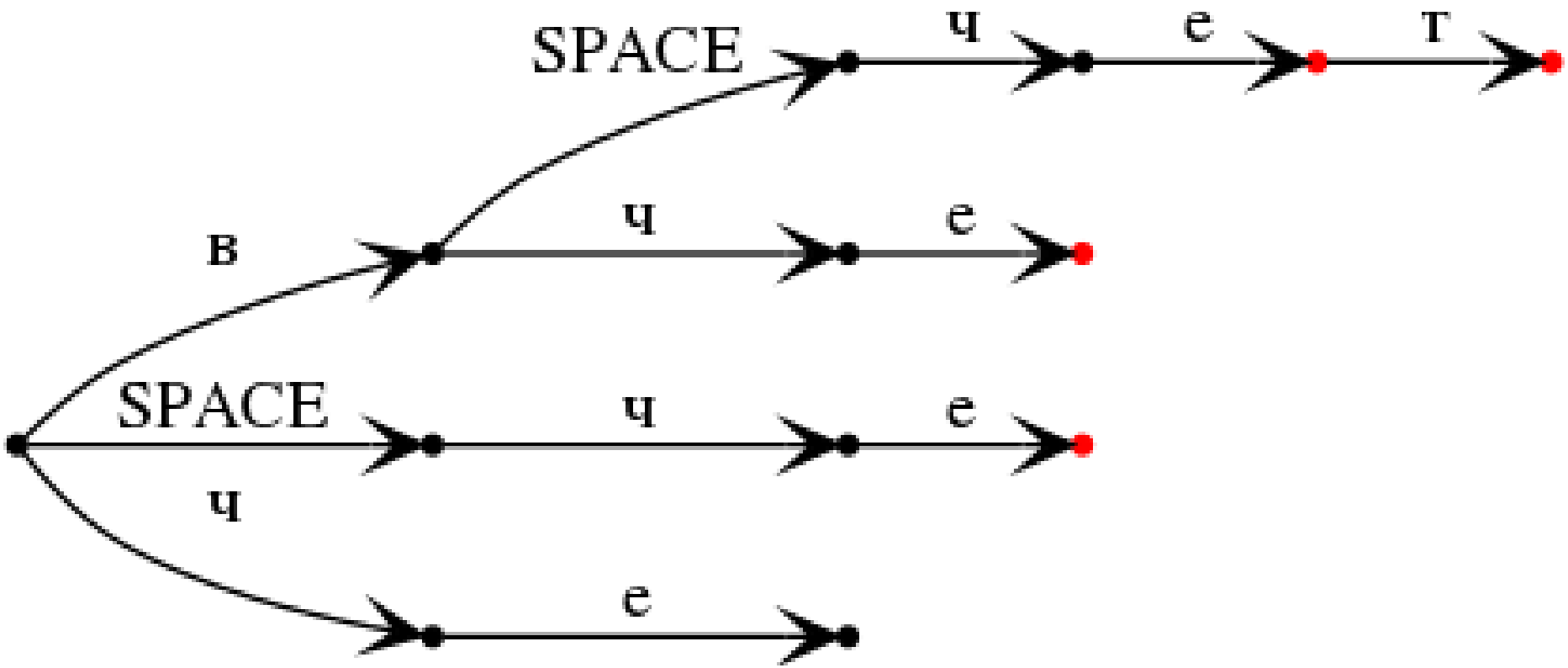

ТИНЬКОФФ

# Дерево гипотез



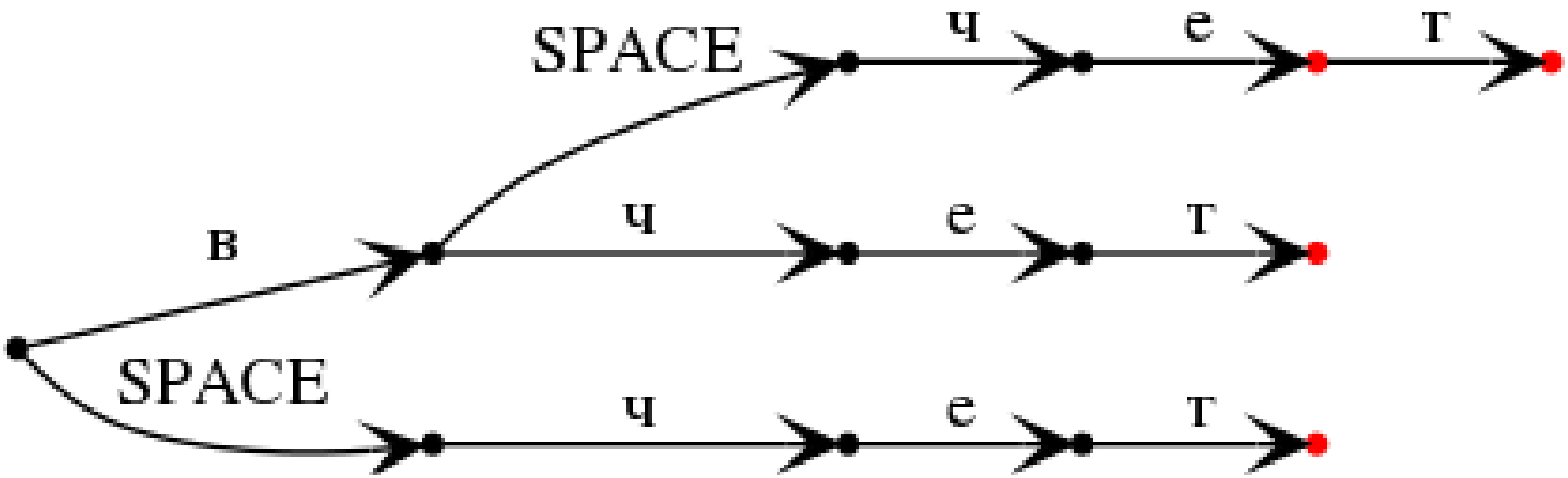

ТИНЬКОФФ

# Дерево гипотез



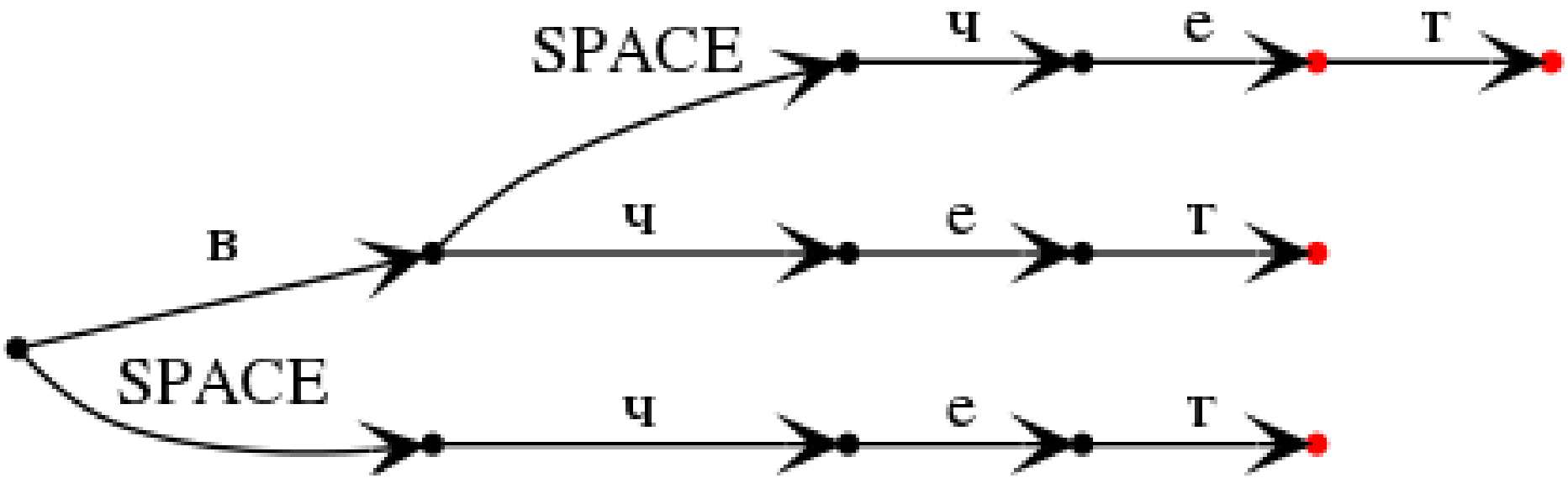

ТИНЬКОФФ

# Дерево гипотез



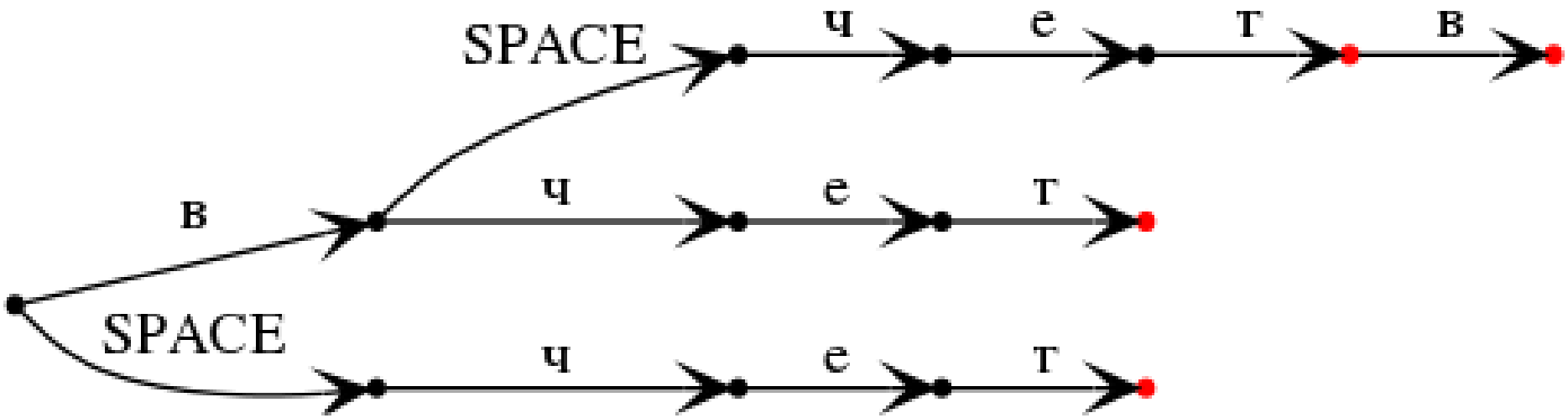

ТИНЬКОФФ

# Дерево гипотез



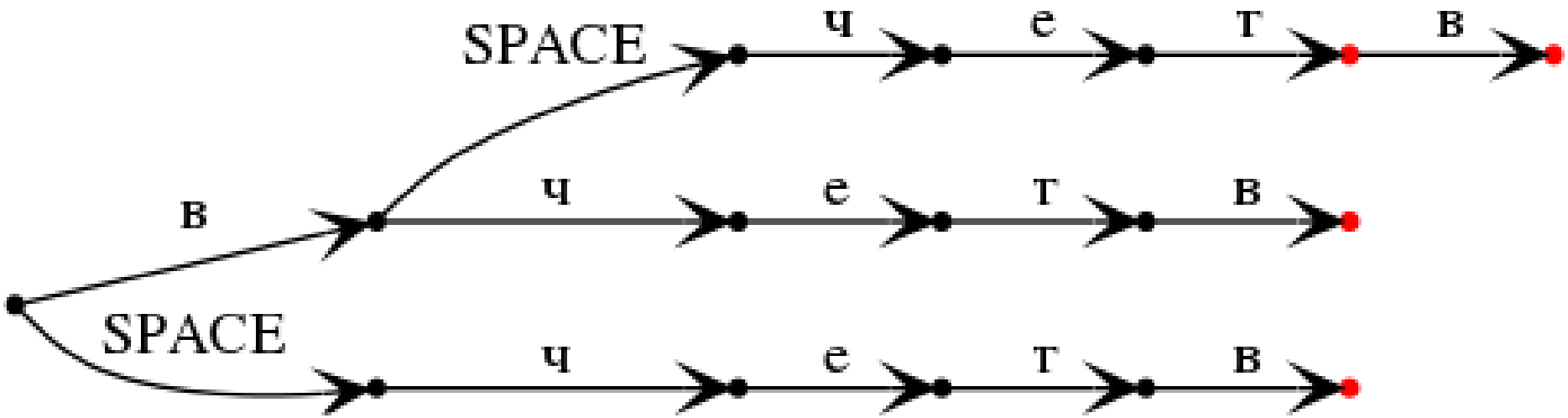

ТИНЬКОФФ

# Дерево гипотез



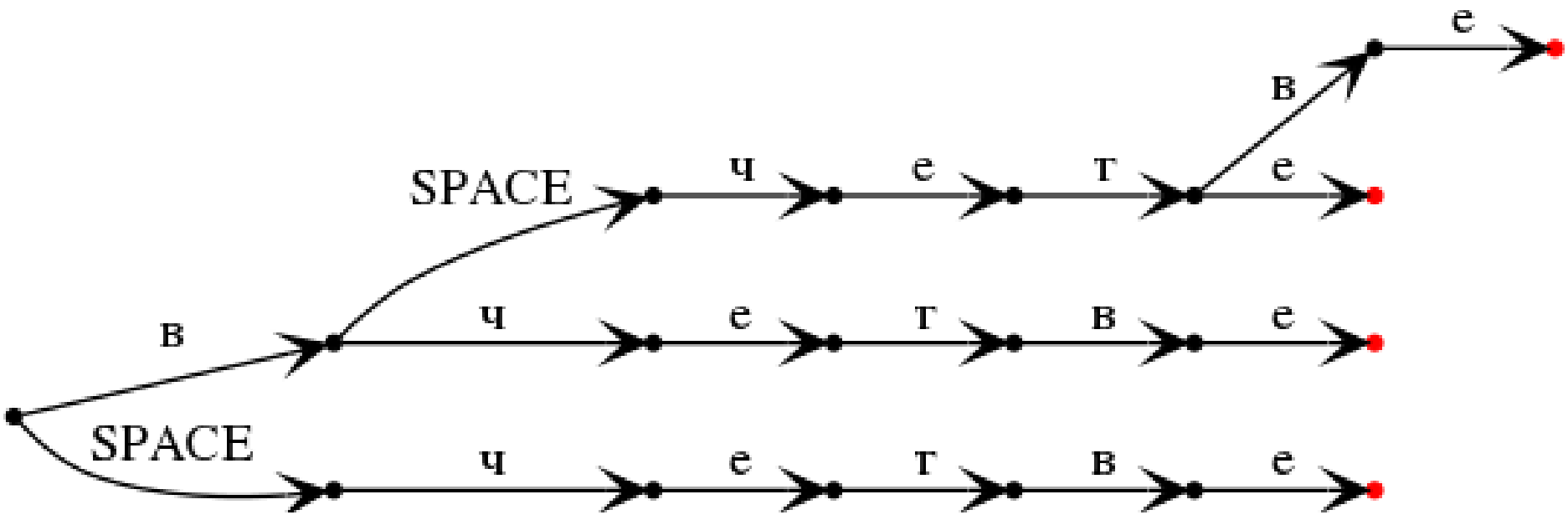

ТИНЬКОФФ

# Дерево гипотез

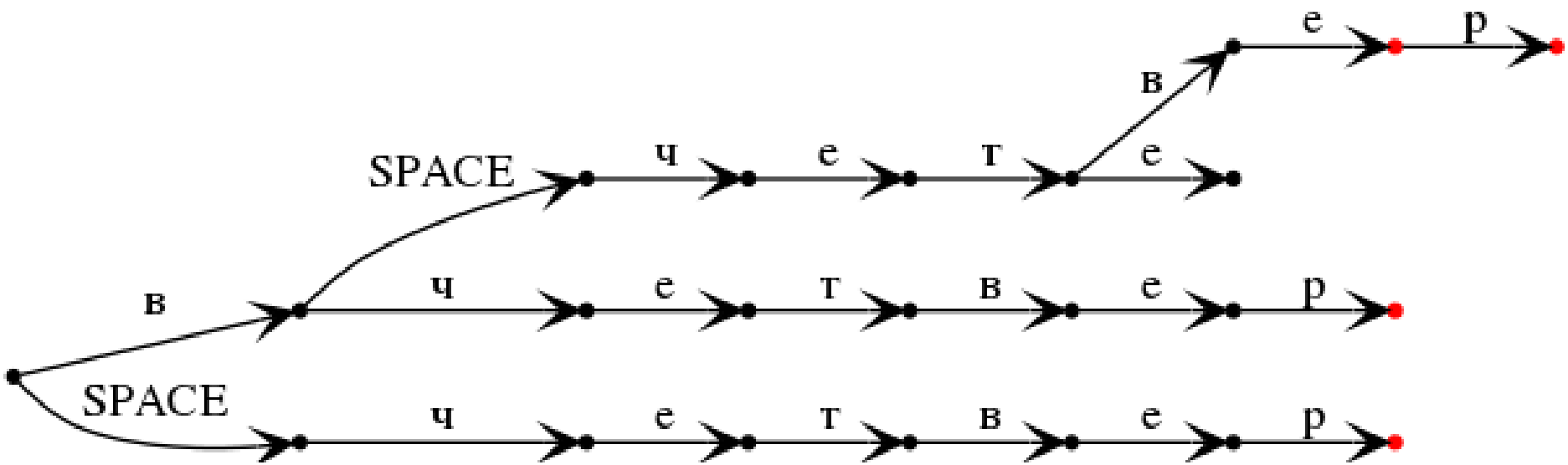



ТИНЬКОФФ

# Дерево гипотез

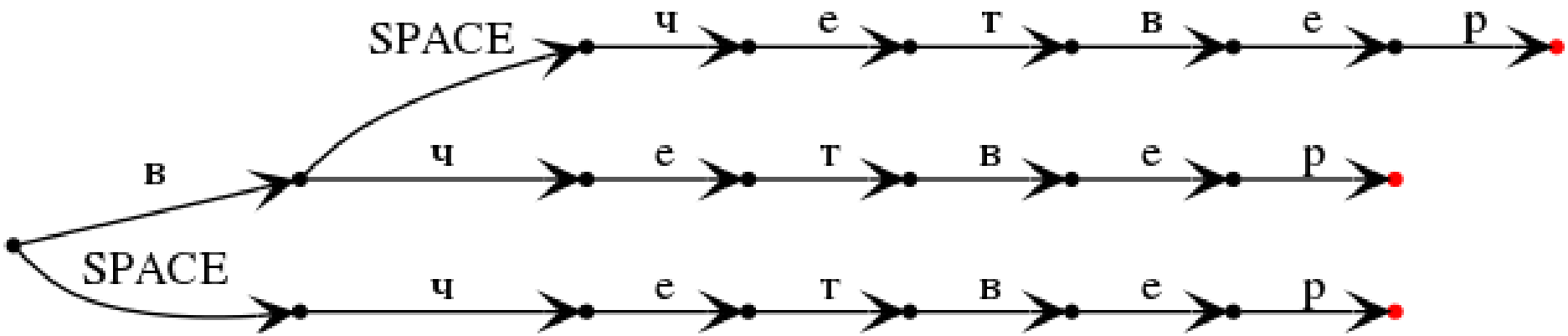


# Дерево гипотез



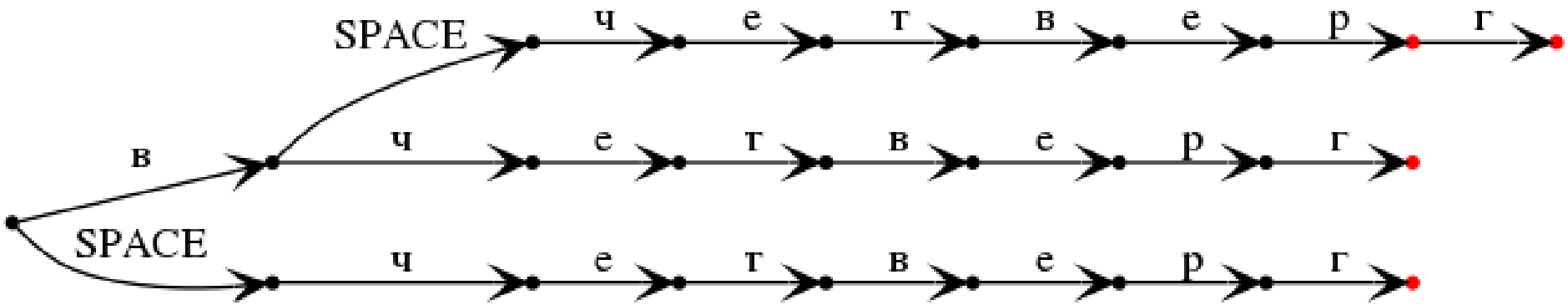

ТИНЬКОФФ

# Дерево гипотез



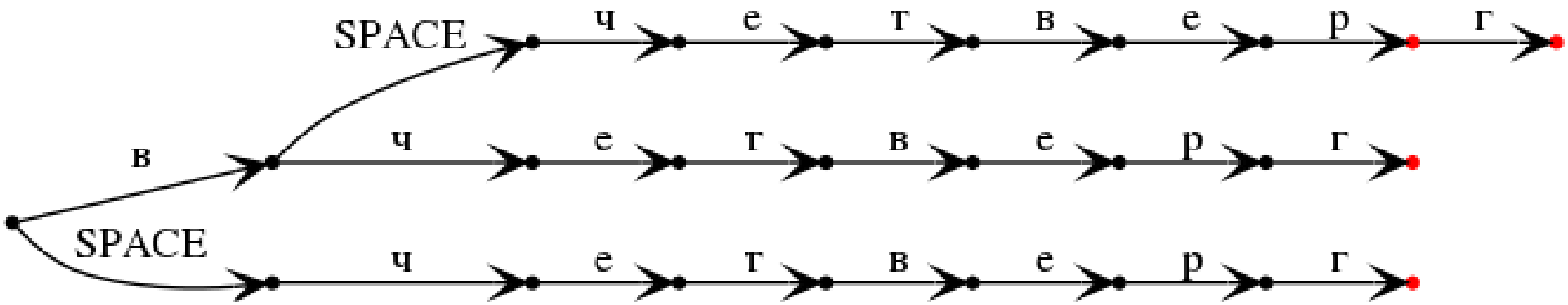

ТИНЬКОФФ

# Дерево гипотез



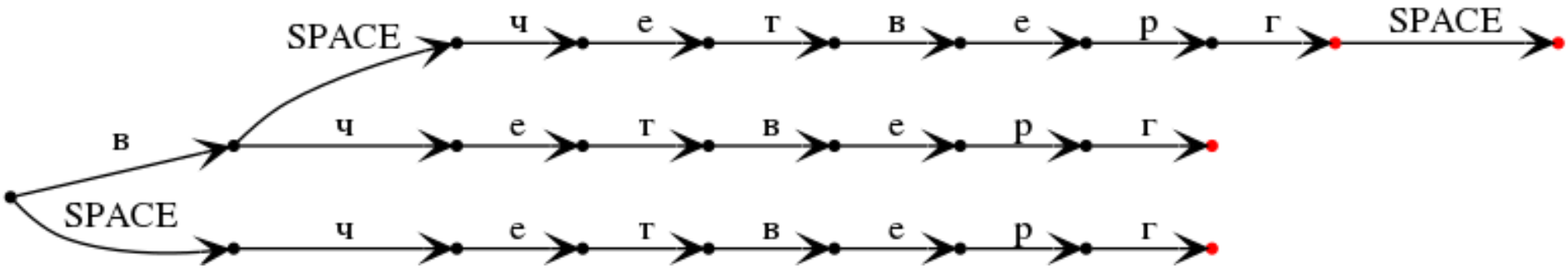

ТИНЬКОФФ

# Дерево гипотез



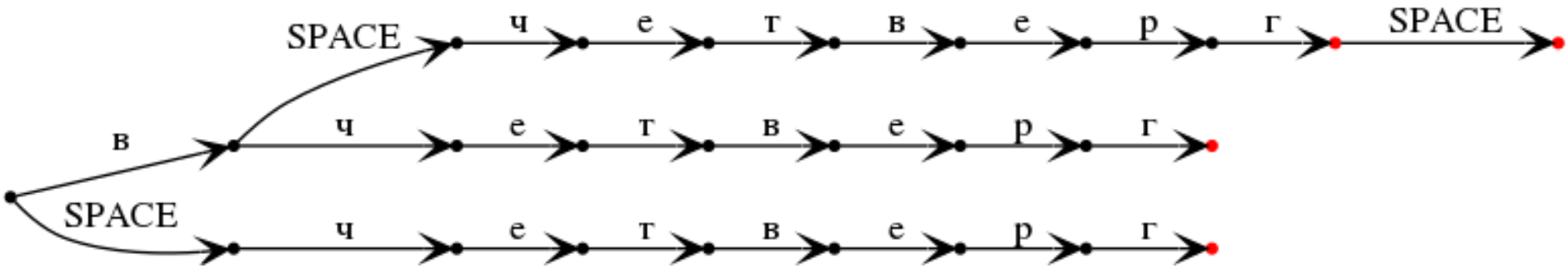

ТИНЬКОФФ

# Дерево гипотез



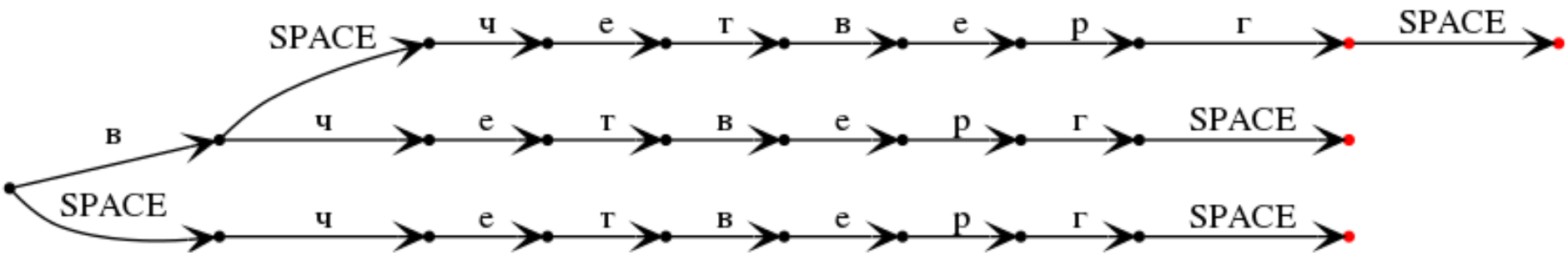

ТИНЬКОФФ

# Дерево гипотез



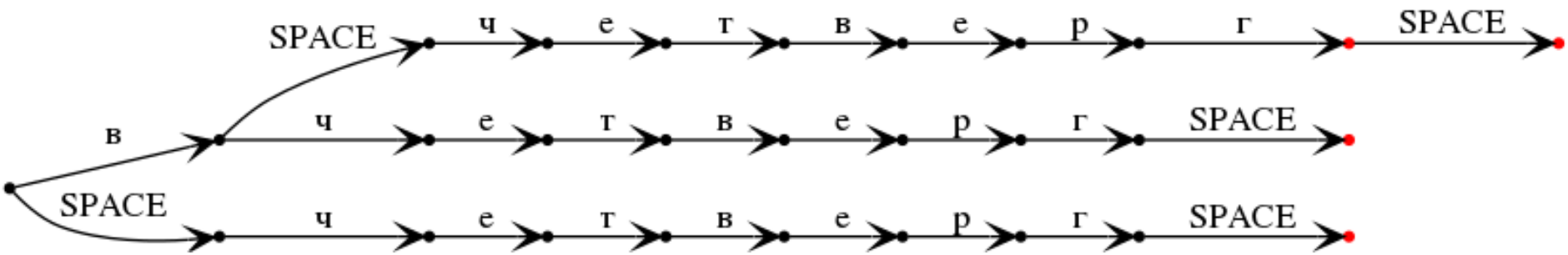

ТИНЬКОФФ

# Дерево гипотез



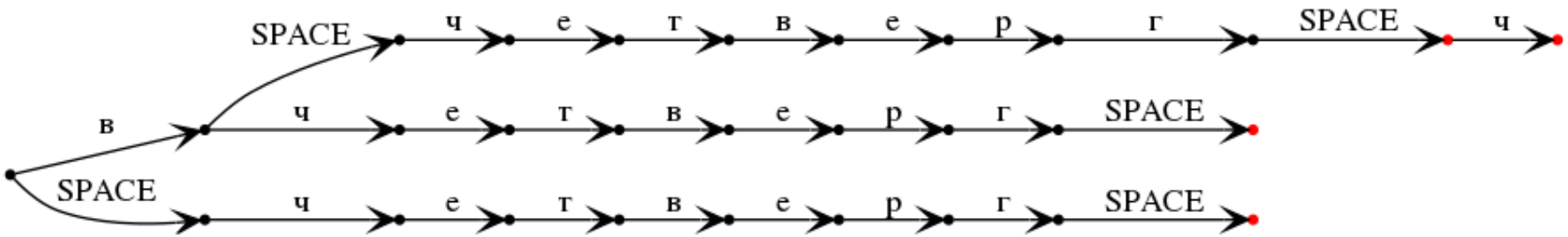

ТИНЬКОФФ

# Дерево гипотез



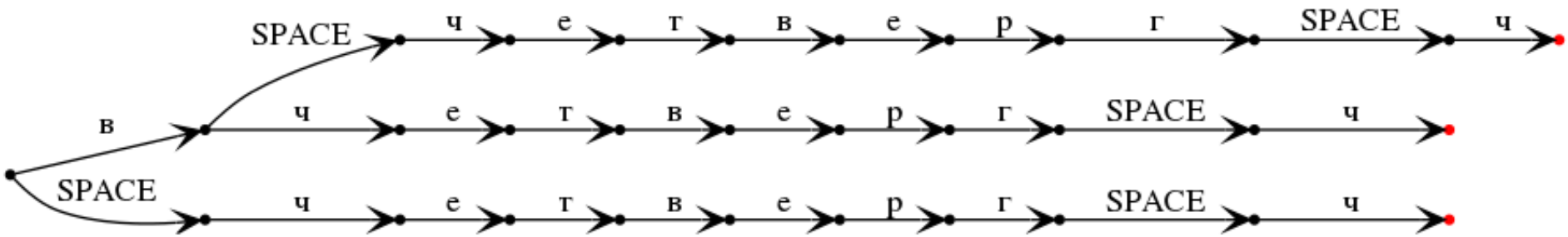

ТИНЬКОФФ

# Дерево гипотез



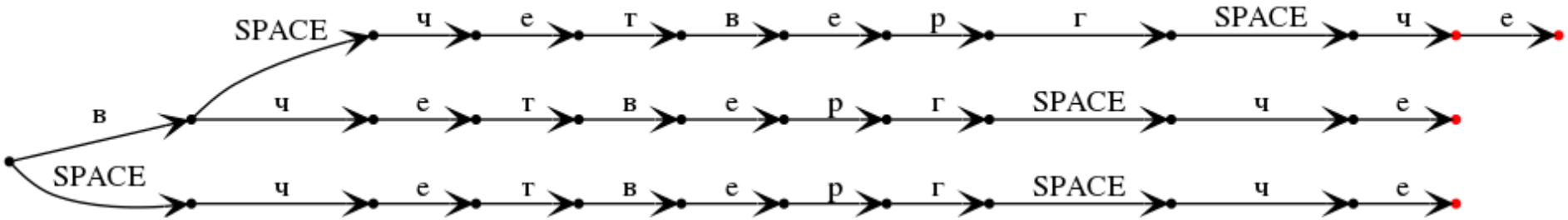

ТИНЬКОФФ

# Дерево гипотез



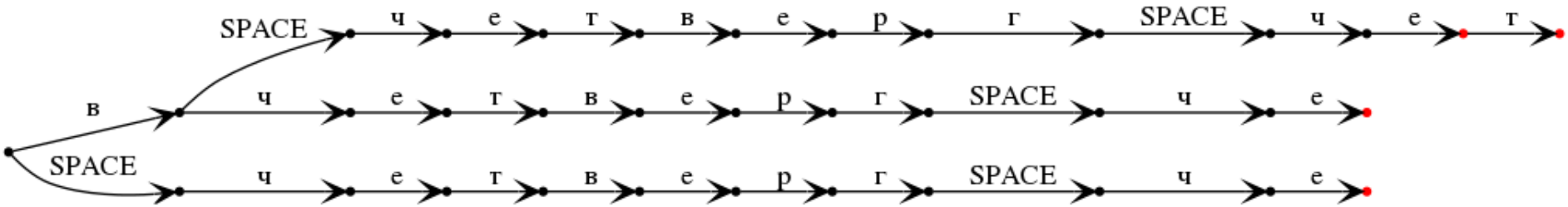

ТИНЬКОФФ

# Дерево гипотез



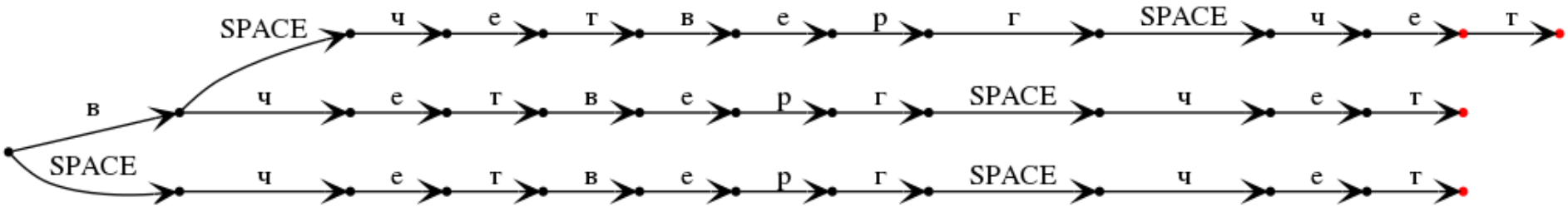

ТИНЬКОФФ

# Дерево гипотез



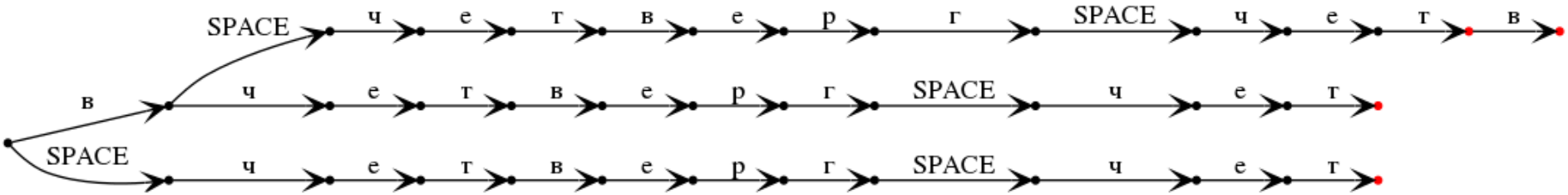

ТИНЬКОФФ

# Дерево гипотез



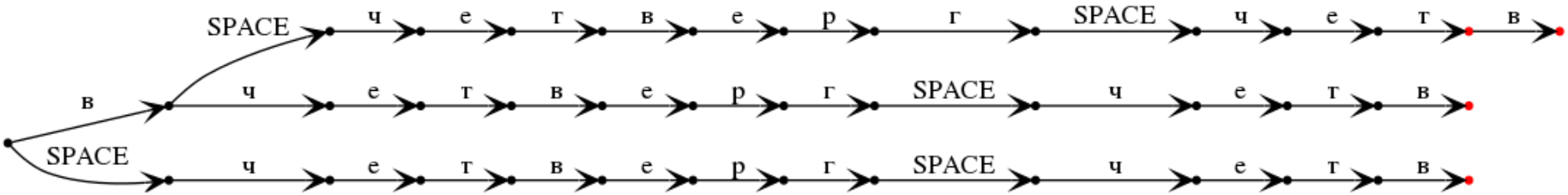

ТИНЬКОФФ

# Дерево гипотез



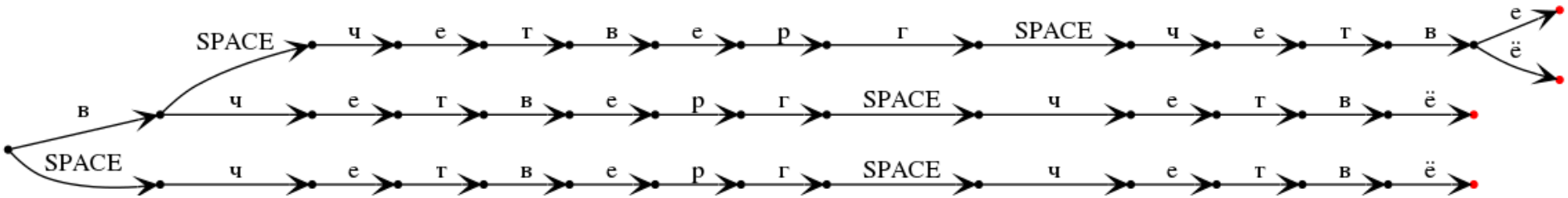

ТИНЬКОФФ

# Дерево гипотез



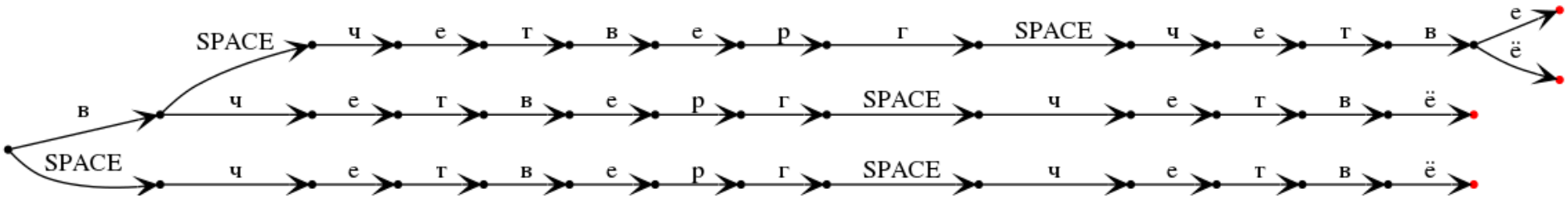

ТИНЬКОФФ

# Дерево гипотез



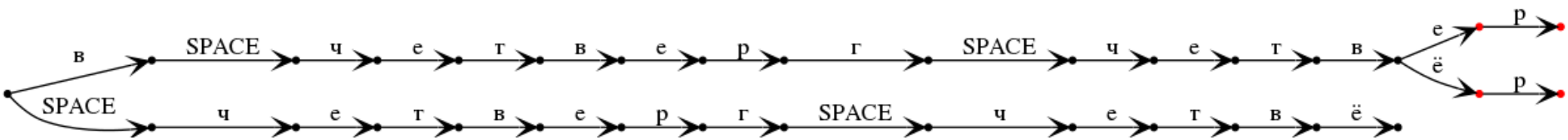

ТИНЬКОФФ

# Дерево гипотез



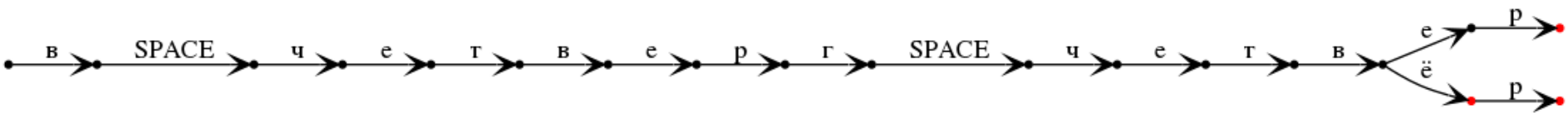

ТИНЬКОФФ

# Дерево гипотез



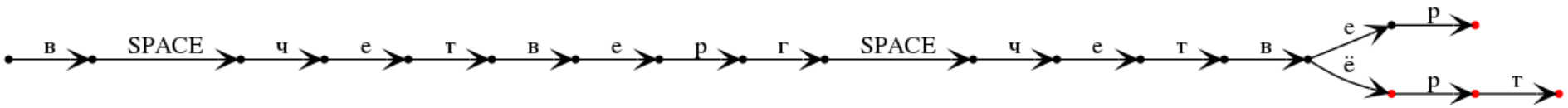

ТИНЬКОФФ

# Дерево гипотез



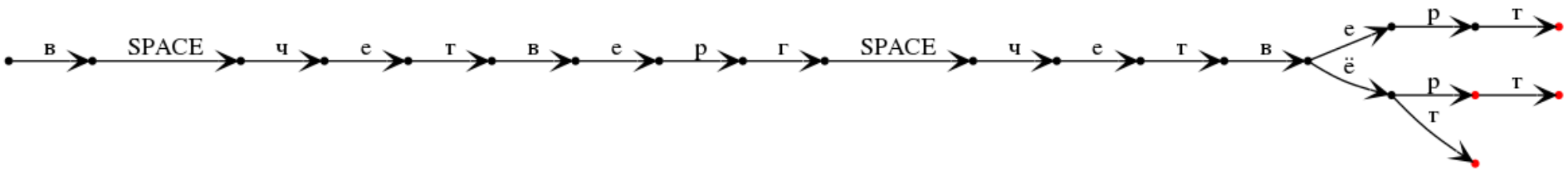

ТИНЬКОФФ

# Дерево гипотез



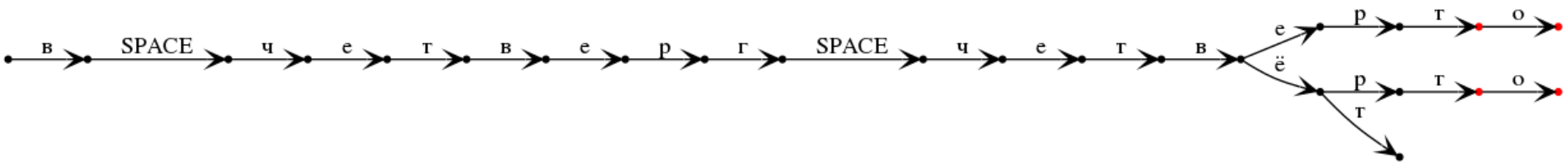

ТИНЬКОФФ

# Дерево гипотез



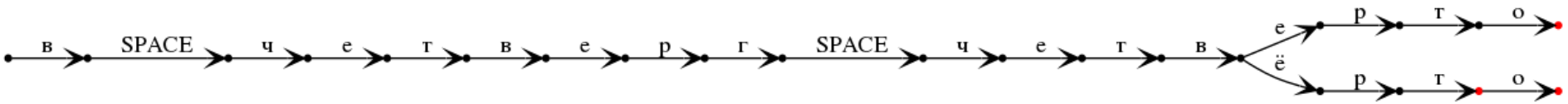

ТИНЬКОФФ

# Дерево гипотез




ТИНЬКОФФ

# Дерево гипотез




ТИНЬКОФФ

# Дерево гипотез


ТИНЬКОФФ

# Дерево гипотез


ТИНЬКОФФ

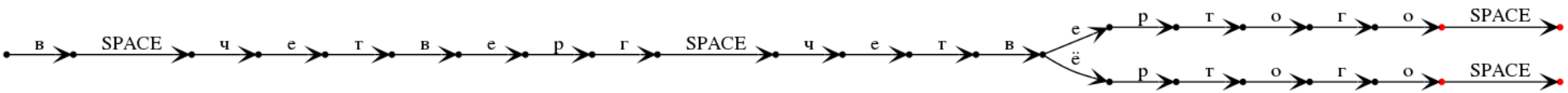
# Дерево гипотез


ТИНЬКОФФ

# Дерево гипотез

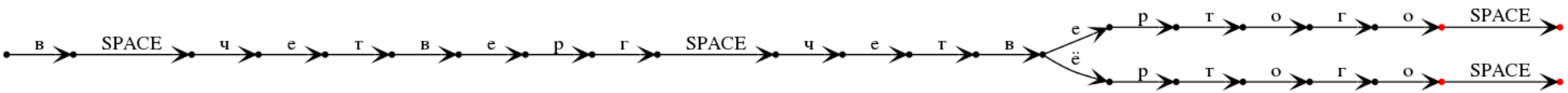

ТИНЬКОФФ

# Дерево гипотез



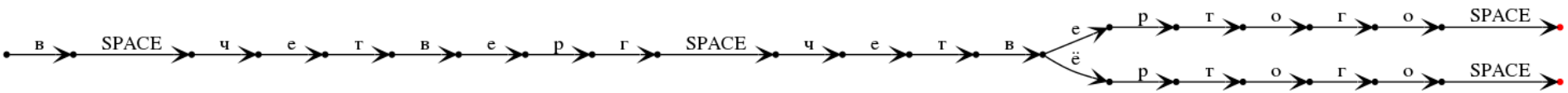

ТИНЬКОФФ

# Дерево гипотез



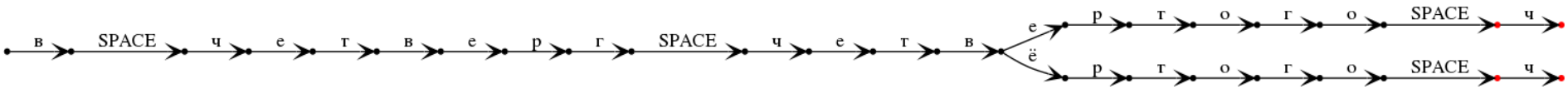

ТИНЬКОФФ

# Дерево гипотез



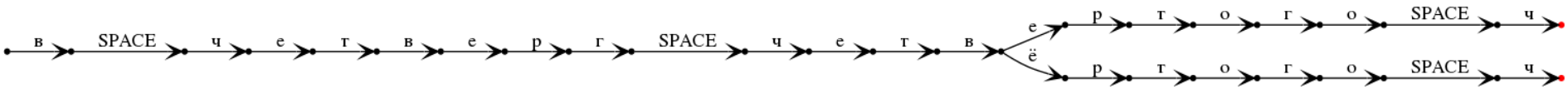

ТИНЬКОФФ

# Дерево гипотез



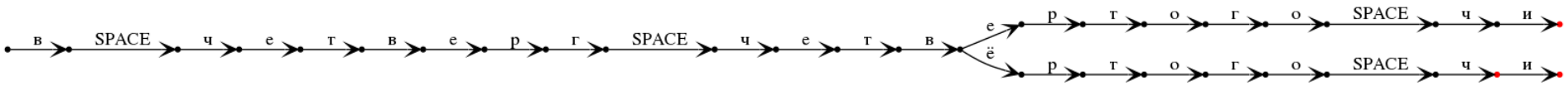

ТИНЬКОФФ

# Дерево гипотез



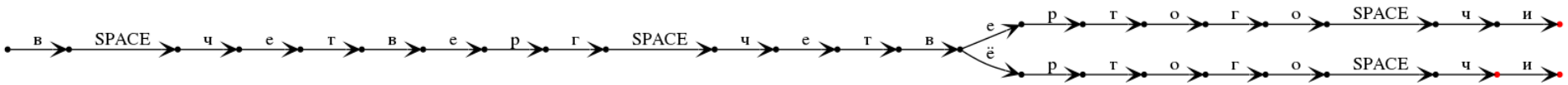

ТИНЬКОФФ

# Дерево гипотез



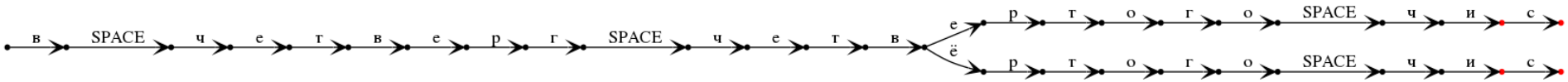

ТИНЬКОФФ

# Дерево гипотез



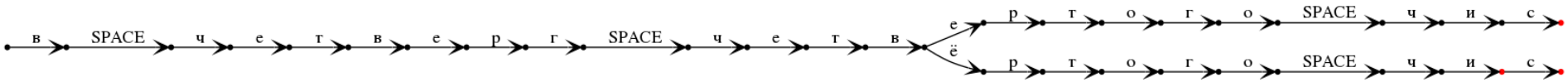

ТИНЬКОФФ

# Дерево гипотез



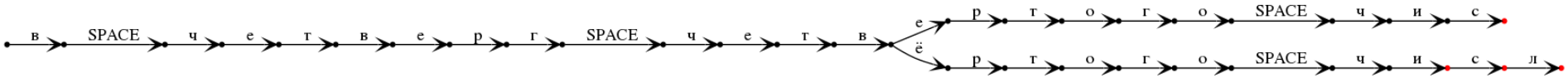

ТИНЬКОФФ

# Дерево гипотез



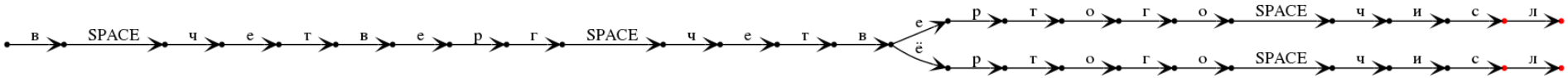

ТИНЬКОФФ

# Дерево гипотез




ТИНЬКОФФ

# Дерево гипотез




ТИНЬКОФФ

# Элемент дерева гипотез

```
class Entry:
    def __init__(self, parent, token):
        self.parent = parent
        self.token = token
        self.next = WeakValueDictionary()

    def prolong(self, token):
        if token not in self.next:
            new_item = Entry(self, token)
            self.next[token] = new_item
            return new_item
        else:
            return self.next[token]

    def backtrace(self):
        result = []
        cur = self
        while cur.token != None:
            result.append(cur.token)
            cur = cur.parent

        return list(reversed(result))
```

ТИНЬКОФФ

# Сохранение лог-вероятностей

```
class Entry:
    def __init__(self, parent, token, acoustic_logprob):
        self.parent = parent
        self.token = token
        self.logprob = acoustic_logprob
        self.next = WeakValueDictionary()

    def prolong(self, token, acoustic_logprob):
        if token not in self.next:
            new_item = Entry(self, token, acoustic_logprob)
            self.next[token] = new_item
            return new_item
        else:
            return self.next[token]
```

ТИНЬКОФФ

# Использование языковой модели (N-gram, rnn)

```
class Entry:
    def __init__(self, parent, token, lm_state, acoustic_logprob):
        self.parent = parent
        self.token = token
        self.lm_state = lm_state
        self.logprob = acoustic_logprob
        self.next = WeakValueDictionary()

    def prolong(self, token, acoustic_logprob, language_model):
        if token not in self.next:
            new_lm_state, lm_logprob =
                self.language_model.query(lm_state, token)
            new_item = Entry(self, token,
                new_lm_state, acoustic_logprob + lm_logprob)
            self.next[token] = new_item
            return new_item
        else:
            return self.next[token]
```

ТИНЬКОФФ

# Использование языковой модели (N-gram, rnn)

```
class Entry:
    def __init__(self, parent, token, lm_state, acoustic_logprob):
        self.parent = parent
        self.token = token
        self.lm_state = lm_state
        self.logprob = acoustic_logprob
        self.next = WeakValueDictionary()

    def prolong(self, token, acoustic_logprob, language_model):
        if token not in self.next:
            new_lm_state, lm_logprob =
                self.language_model.query(lm_state, token)
            new_item = Entry(self, token,
                new_lm_state, acoustic_logprob + lm_logprob)
            self.next[token] = new_item
            return new_item
        else:
            return self.next[token]
```

Работает для $N$-грамной и однонаправленной RNN языковых моделях, RNN-transducer делает что-то подобное

ТИНЬКОФФ

# Использование языковой модели (attention-based)

```
class Entry:
    def __init__(self, parent, token, acoustic_logprob):
        self.parent = parent
        self.token = token
        self.logprob = acoustic_logprob
        self.next = WeakValueDictionary()

    def prolong(self, token, acoustic_logprob, language_model):
        if token not in self.next:
            lm_logprob = self.language_model.query(self.backtrace(), token)
            new_item = Entry(self, token,
                acoustic_logprob + lm_logprob))
            self.next[token] = new_item
            return new_item
        else:
            return self.next[token]
```

ТИНЬКОФФ

# Использование языковой модели (attention-based)

```
class Entry:
    def __init__(self, parent, token, acoustic_logprob):
        self.parent = parent
        self.token = token
        self.logprob = acoustic_logprob
        self.next = WeakValueDictionary()

    def prolong(self, token, acoustic_logprob, language_model):
        if token not in self.next:
            lm_logprob = self.language_model.query(self.backtrace(), token)
            new_item = Entry(self, token,
                acoustic_logprob + lm_logprob))
            self.next[token] = new_item
            return new_item
        else:
            return self.next[token]
```

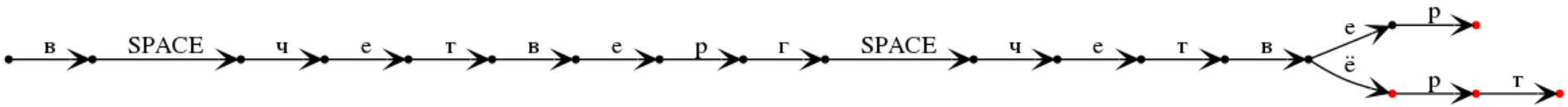
При использовании трансформера необходимо сделать backtrace() (хранить историю непосредственно дороже), что создает проблему квадратичной сложности

Beck E., Schlüter R., Ney H. LVCSR with Transformer Language Models //INTERSPEECH. – 2020. – С. 1798-1802.

ТИНЬКОФФ

# Фиксированная часть распознавания

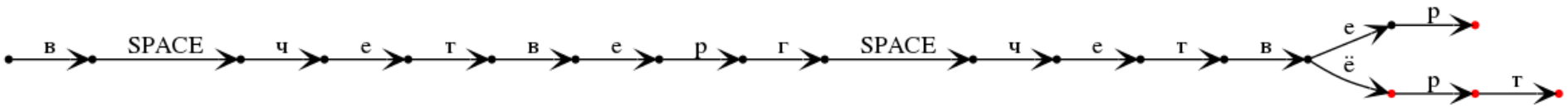
Начиная с какого-то момента “в четверг четв” является фиксированной частью, т.е. содержится во любой текущей гипотезе, а значит будет содержаться и в любых будущих гипотезах. Такую часть можно отрезать и выдать, чтобы не хранить лишнюю информацию в движке распознавания – улучшает стабильность распознавания в real-time системах.




ТИНЬКОФФ

# Фиксированная часть распознавания

Начиная с какого-то момента “в четверг четв” является фиксированной частью, т.е. содержится во любой текущей гипотезе, а значит будет содержаться и в любых будущих гипотезах. Такую часть можно отрезать и выдать, чтобы не хранить лишнюю информацию в движке распознавания – улучшает стабильность распознавания в real-time системах.




По сути нужно найти наименьшего общего предка всех (LCA) всех текущих гипотез: можно либо пройти от корня к листам до первого разветвления, либо в случае отсутствия прямых переходов пройти в обратном порядке и найти последний элемент, на который указывает больше одной ссылки, это и будет первая развилка.

ТИНЬКОФФ

# Промежуточные гипотезы

Для визуального эффекта, а иногда и для принятия решений, хорошей практикой является выдача промежуточных гипотез по мере обработки фреймов звука. Разделение на фиксированную и нефиксированную части помогает для обработки длинных длинных гипотез: например обозначив в круглых скобках фиксированную часть, следующая последовательность

- мама
- (мама) мыла
- мыла раму
- (мыла) папу
- (раму мылом)

эквивалентна гипотезам

- мама
- мама мыла
- мама мыла раму
- мама мыла папу
- мама мыла раму мылом

ТИНЬКОФФ

ТИНЬКОФФ

# Множественные гипотезы

Во многих случаях оказывается полезным выдавать не одну лучшую гипотезу, а сразу несколько:

- Можно сделать “рескоринг” более хорошей языковой моделью поверх этих гипотез (лучше качество)

ТИНЬКОФФ

# Множественные гипотезы

Во многих случаях оказывается полезным выдавать не одну лучшую гипотезу, а сразу несколько:

- Можно сделать “рескоринг” более хорошей языковой моделью поверх этих гипотез (лучше качество)
- Можно сделать MBR-декодерование (лучше качество)

ТИНЬКОФФ

# Множественные гипотезы

Во многих случаях оказывается полезным выдавать не одну лучшую гипотезу, а сразу несколько:

- Можно сделать “рескоринг” более хорошей языковой моделью поверх этих гипотез (лучше качество)
- Можно сделать MBR-декодерование (лучше качество)
- Вычислять пословный конфиденс

ТИНЬКОФФ

# Множественные гипотезы

Во многих случаях оказывается полезным выдавать не одну лучшую гипотезу, а сразу несколько:

- Можно сделать “рескоринг” более хорошей языковой моделью поверх этих гипотез (лучше качество)
- Можно сделать MBR-декодерование (лучше качество)
- Вычислять пословный конфиденс
- Поиск ключевых слов

ТИНЬКОФФ

# MBR-декодирование

| Sentence | Model Prob |
|---|---|
| A B C | 0.4 |
| A D X | 0.3 |
| A D Y | 0.3 |

(a) Modeled probabilities

| Decoded Sentence | Expected Errors | |
|---|---|---|
| | Sent | Word |
| A B C | 0.6✓ | 1.2 |
| A D X | 0.7 | 1.1 |
| A D Y | 0.7 | 1.1 |
| A D C | 1.0 | 1.0 ✓ |

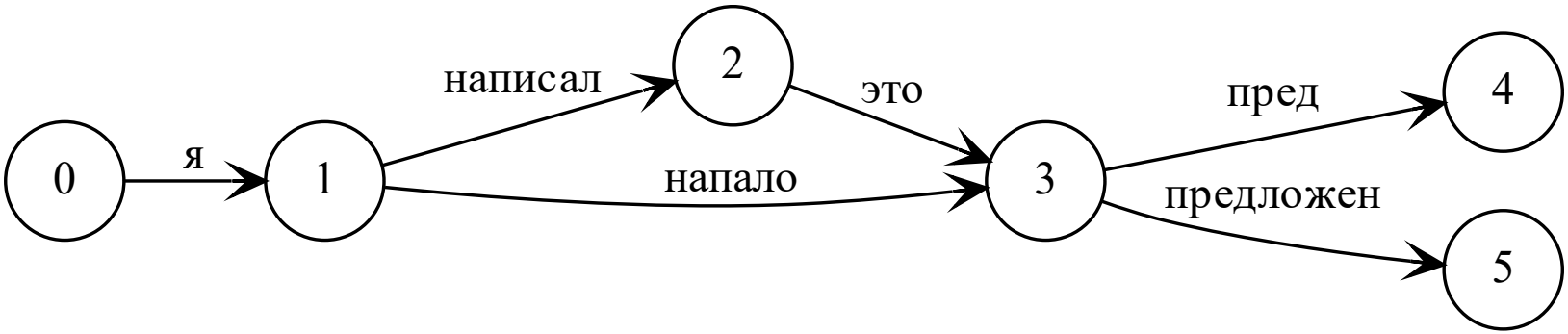
(b) Expected errors

Fig. 1. Example of MBR estimate differing from MAP estimate.

ТИНЬКОФФ

# Lattice

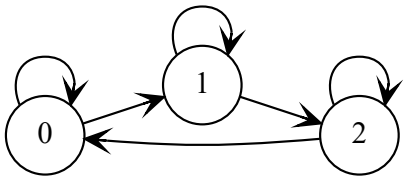
Lattice – это граф, в котором каждому пути соответствует гипотезы. Такой способ представления **гораздо** более компактный, чем *n*-best, т.е. просто список из нескольких непосредственно представленных гипотез.

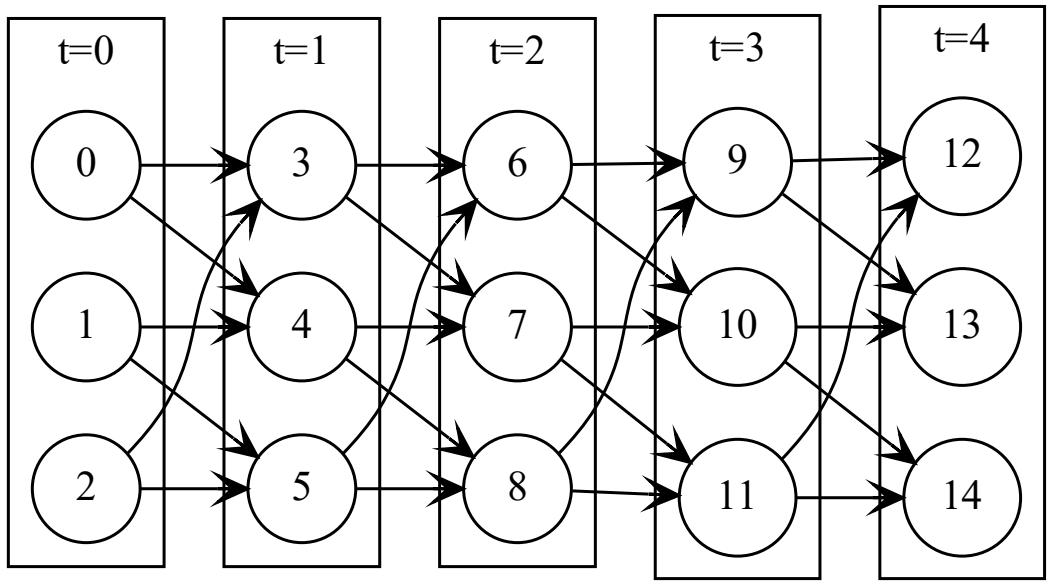



ТИНЬКОФФ

# Lattice в HMM

Одно из канонических определений латтиса (использующихся например в калди): lattice – это все возможные пути раскрытого во времени графа (правого), содержащего гипотезы, лог-правдоподобие которых отличается от лучшей гипотезы не хуже, чем на фиксированную константу (параметр латтиса)




ТИНЬКОФФ

# Forward-backward метод

Пример: хотим найти в lattice точки сочленения – такие вершины, удаление которых нарушит связность начальной вершины $s$ и конечной вершины $t$

ТИНЬКОФФ

# Forward-backward метод

Пример: хотим найти в lattice точки сочленения – такие вершины, удаление которых нарушит связность начальной вершины $s$ и конечной вершины $t$

Решение через подсчет количества путей:

- Пусть $\alpha(v)$ – количество различных путей из $s$ в $v$

# Forward-backward метод

Пример: хотим найти в lattice точки сочленения – такие вершины, удаление которых нарушит связность начальной вершины $s$ и конечной вершины $t$

Решение через подсчет количества путей:

- Пусть $\alpha(v)$ – количество различных путей из $s$ в $v$
- Пусть $\beta(v)$ – количество различны путей из $v$ в $t$

ТИНЬКОФФ

# Forward-backward метод

Пример: хотим найти в lattice точки сочленения – такие вершины, удаление которых нарушит связность начальной вершины $s$ и конечной вершины $t$

Решение через подсчет количества путей:

- Пусть $\alpha(v)$ – количество различных путей из $s$ в $v$
- Пусть $\beta(v)$ – количество различны путей из $v$ в $t$
- По построению $\alpha(s) = \beta(t) = 1$ и $\alpha(t) = \beta(s)$

ТИНЬКОФФ

# Forward-backward метод

Пример: хотим найти в lattice точки сочленения – такие вершины, удаление которых нарушит связность начальной вершины $s$ и конечной вершины $t$
Решение через подсчет количества путей:

- Пусть $\alpha(v)$ – количество различных путей из $s$ в $v$
- Пусть $\beta(v)$ – количество различны путей из $v$ в $t$
- По построению $\alpha(s) = \beta(t) = 1$ и $\alpha(t) = \beta(s)$
- $v$ является точкой сочленения если $\alpha(v)\beta(v) = \alpha(t) = \beta(s)$

ТИНЬКОФФ

# Forward-backward метод

```
def forward_paths(lattice):
    forward = [(1 if node == lattice.start else 1) for node in lattice.nodes]
    for node in top_sort(lattice.nodes):
        for arc in lattice.arcs[node]:
            forward[arc.nextstate] += forward[node]
    return forward

def backward_paths(lattice):
    backward = [(1 if node == lattice.end else 1) for node in lattice.nodes]
    for node in reversed(top_sort(lattice.nodes)):
        for arc in lattice.arcs[node]:
            backward[node] += backward[arc.nextstate]
    return backward

def find_cutpoints(lattice):
    forward = forward_paths(lattice)
    backward = backward_paths(lattice)
    return [node for node in lattice.nodes
            if forward[node] * backward[node] == backward[lattice.start]]
```

ТИНЬКОФФ

# Lattice $\rightarrow$ n-best

Есть множественный результат в виде lattice, хотим извлечь из него $n$ лучших путей из $s$ в $t$. Стандартное решение – модификация $A^*$, сложность $\mathcal{O}(n|E|\ell)$, $E$ – переходы в lattice, $\ell$ - сложность операции извлечения минимума используемой структуры данных

```
def backward_paths(lattice):
    backward = [(0 if node == lattice.end else float("inf"))
                for node in lattice.nodes]
    for node in reversed(top_sort(lattice.nodes)):
        for arc in lattice.arcs[node]:
            backward[node] = min(backward[node],
                                 backward[arc.nextstate] + arc.weight)
    return backward
```

ТИНЬКОФФ

# Lattice → n-best

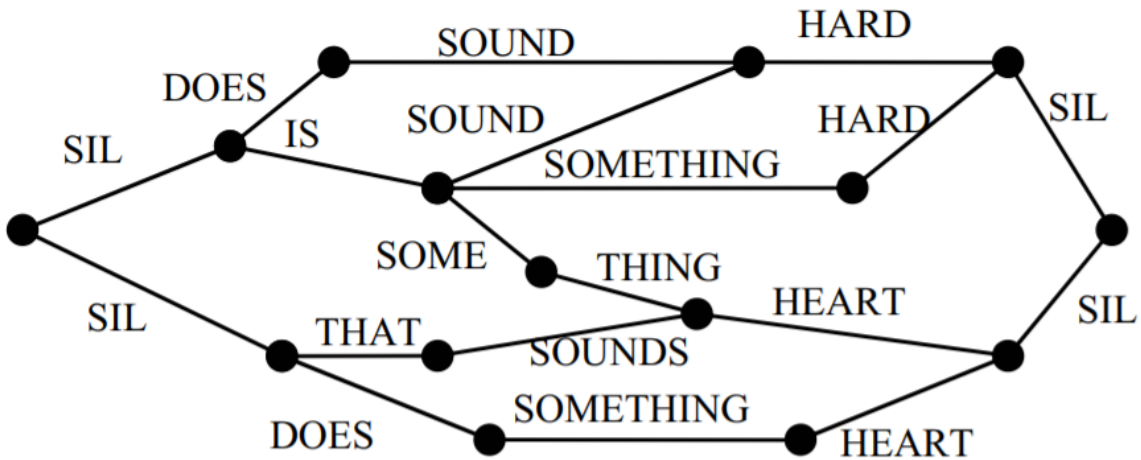
```
class BestPathTreeNode:
    def __init__(self, node, forward_weight, last_arc):
        self.node = node
        self.forward_weight = forward_weight
        self.last_arc = last_arc

def LatticeToNBest(lattice, n):
    backward = backward_paths(lattice)
    q = PriorityQueue()
    q.put(backward[lattice.start], BestPathTreeNode(lattice.start, 0, None))
    result = []
    while n > 0:
        current_node = q.pop()
        if current_node.node == lattice.end:
            result.append(current.node)
            n -= 1
            continue
        for arc in lattice.arcs(current_node.node):
            q.put(current_node.forward_weight + arc.weight
                  + backward[arc.nextstate],
                  BestPathTreeNode(arc.nextstate,
                                   current_node.forward_weight + arc.weight,
                                   arc))
    return result
```

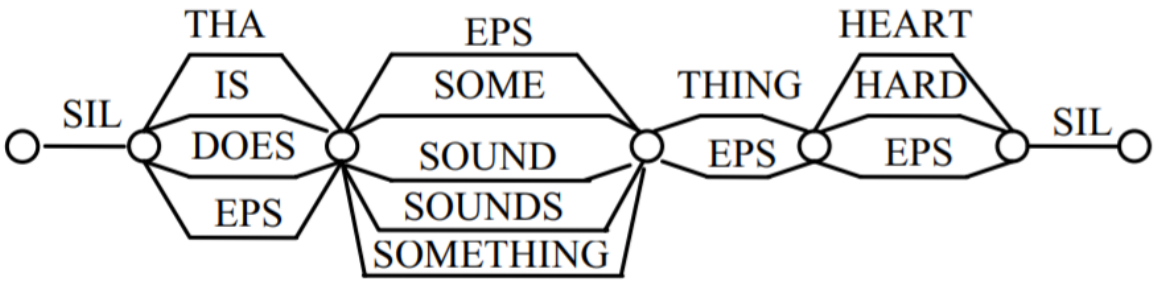
ТИНЬКОФФ

# Lattice и CN

*Confusion network* (CN) – упрощенная структура, которая возникает при MBR-декодировании, позволяет вычислять эффективный словный конфиденс, немного улучшает WER, да и просто с ней проще работать. К сожалению, эффективное построение – сложная алгоритмическая задача




(a) Initial lattice




(b) Confusion Network

# CN как объединение выходов нескольких систем

Предположим, что три системы выдали следующие результаты распознавания

- мама мыла раму мылом
- мама рыла яму мылом
- мыла раму шилом

ТИНЬКОФФ

# CN как объединение выходов нескольких систем

Предположим, что три системы выдали следующие результаты распознавания

- мама мыла раму мылом
- мама рыла яму мылом
- мыла раму шилом

если оптимально выровнять эти гипотезы, то получится

| | | | |
|---|---|---|---|
| мама | мыла | раму | мылом |
| мама | рыла | яму | мылом |
| **** | мыла | раму | шилом |

ТИНЬКОФФ

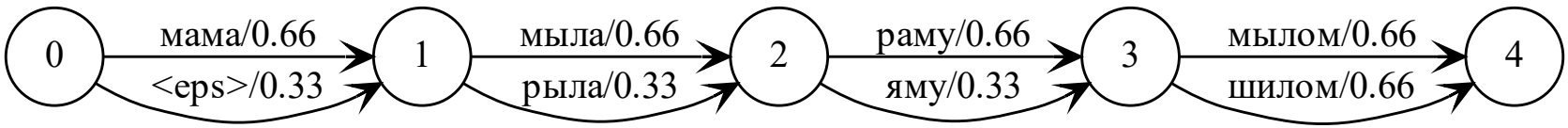
# CN как объединение выходов нескольких систем

Предположим, что три системы выдали следующие результаты распознавания

- мама мыла раму мылом
- мама рыла яму мылом
- мыла раму шилом

если оптимально выровнять эти гипотезы, то получится

| | | | |
|---|---|---|---|
| мама | мыла | раму | мылом |
| мама | рыла | яму | мылом |
| **** | мыла | раму | шилом |




ТИНЬКОФФ