

SQL HowTo

Алгоритмы на графах

Кирилл Боровиков / Компания «Тензор», технический директор / explain.tensor.ru, sbis.ru

Это не граф



... а барон

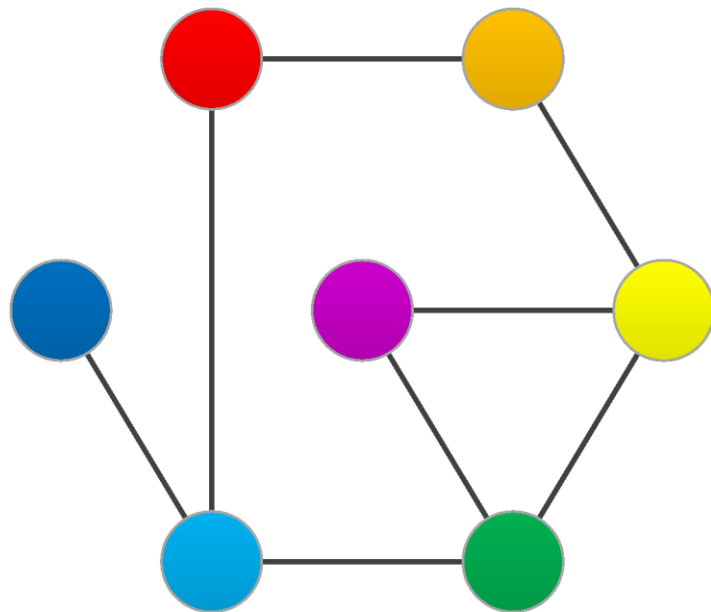
Это – граф



... да не тот



Это – граф

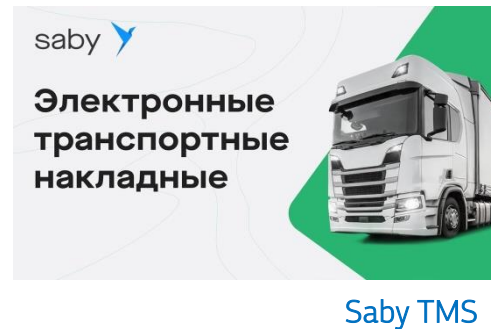
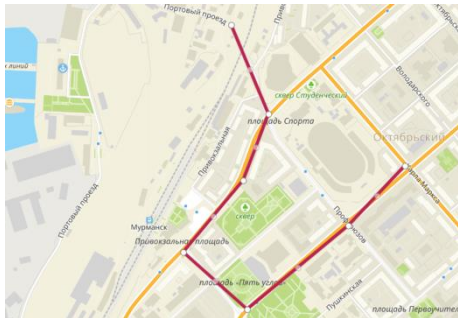
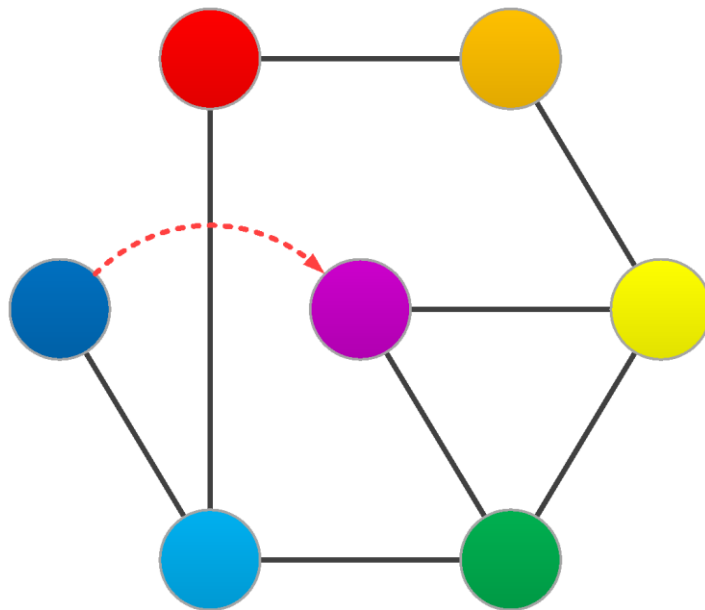


набор вершин, соединенных ребрами

Алгоритм Ли

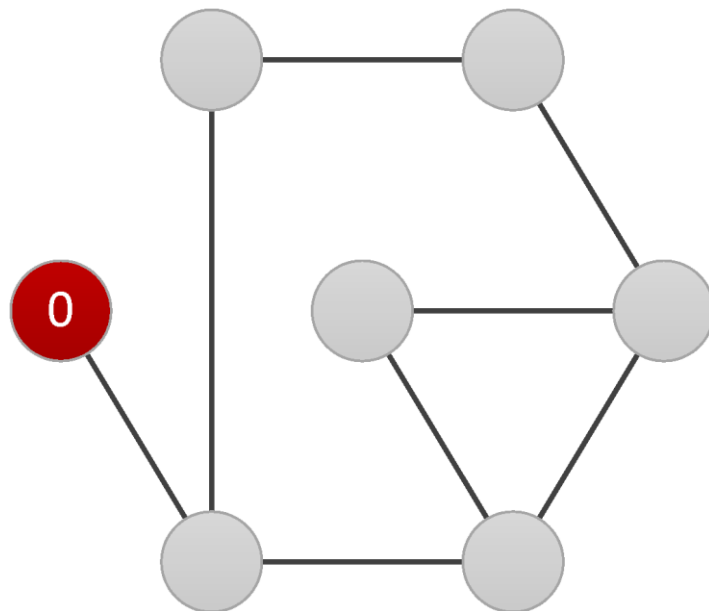
Поиск кратчайшего пути на планарном графе.

https://ru.wikipedia.org/wiki/Алгоритм_Ли



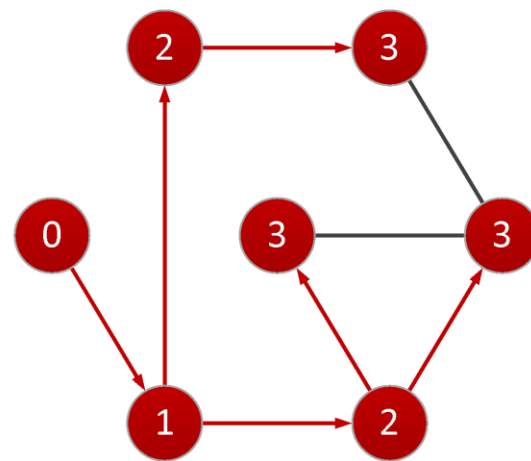
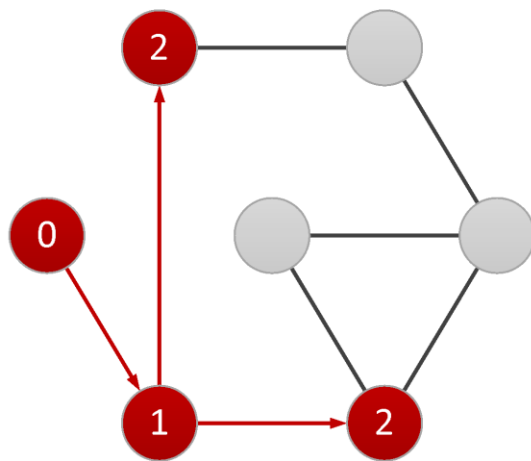
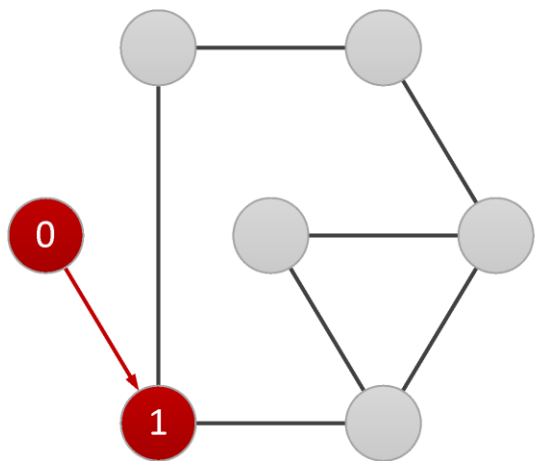
Алгоритм Ли

Шаг 1: инициализация волны



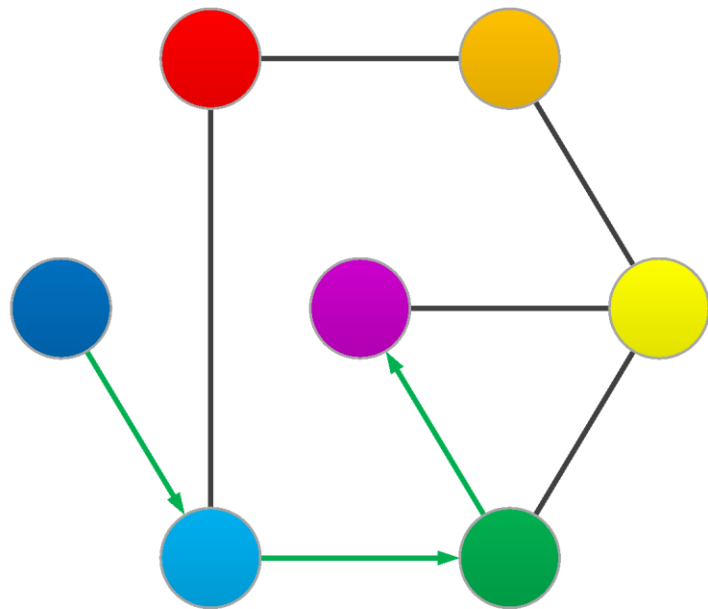
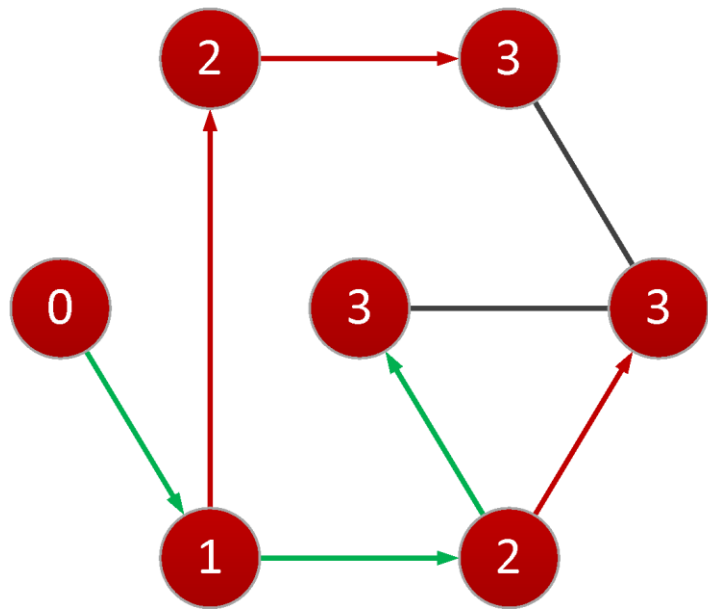
Алгоритм Ли

Шаг 2: распространение волны



Алгоритм Ли

Шаг 3: восстановление пути

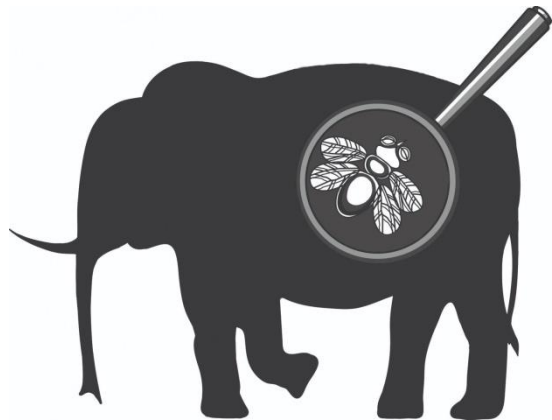


Из мухи – слона

«Правила игры очень просты: **надо построить цепочку слов от начального (МУХА) до конечного (СЛОН), на каждом шаге меняя только одну букву**. При этом могут использоваться только русские 4-буквенные нарицательные существительные в начальной форме: например, слова **БАЗА, НОЧЬ, САНИ** допускаются, а слова ЛИТЬ, ХОТЯ, РУКУ, НОЧИ, САНЯ, ОСЛО, АБВГ, ФЦНМ — нет.

Эта игра под названием «Дублеты» приобрела известность благодаря Льюису Кэрроллу — не только автору книг про Алису, но ещё и замечательному математику. В марте 1879 года он начал раз в неделю публиковать в журнале «Ярмарка тщеславия» по три задания в форме броских фраз: «Turn **POOR** into **RICH**» — «Преврати бедного в богатого», «Evolve **MAN** from **APE**» — «Выведи человека из обезьяны», «Make **TEA HOT**» — «Сделай чай горячим». В том же году он выпустил брошюру «Дублеты», подробно описал в ней правила и предложил читателям попрактиковаться на нескольких десятках примеров.»

Александр Пиперски, ["Из мухи — слона"](#), «Квантик» [№2, 2019](#) и [№3, 2019](#)



Из мухи – слона

Шаг 1: загружаем словарь (словарь Ефремовой, 51К существительных)

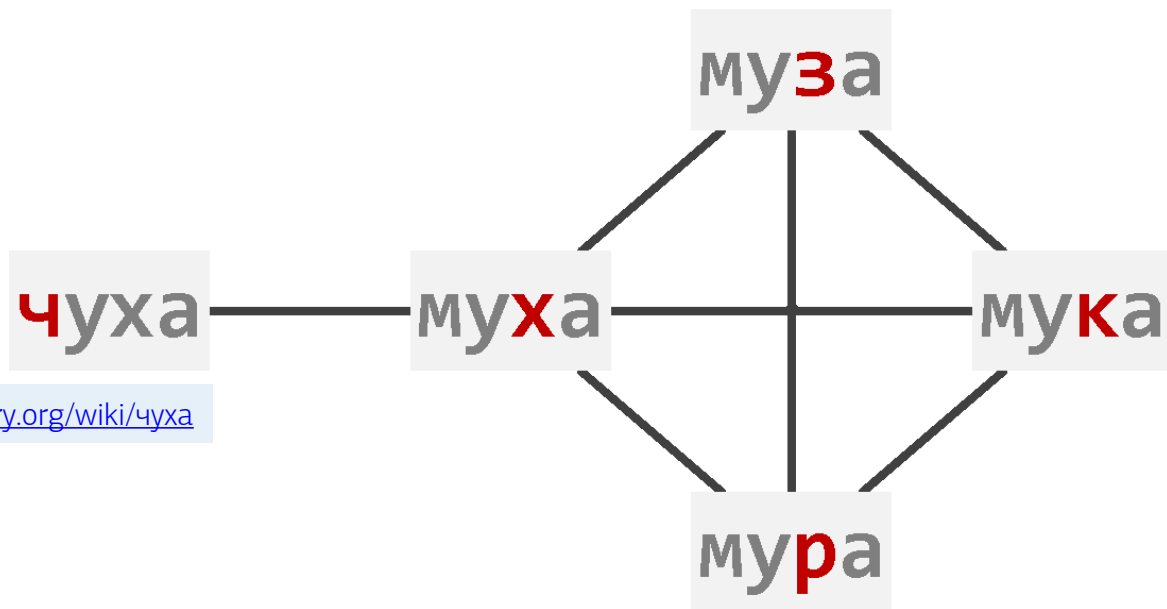
<https://harrix.dev/blog/2018/russian-nouns/>

```
CREATE TABLE dict AS
SELECT
    btrim(word) word    -- зачищаем пробелы с обеих сторон
FROM
    regexp_split_to_table(
абазур
абазурчик
абаз
абазин
-- ...
    $$, E'[\r\n]') word -- делим текст на строки
WHERE
    word !~ '\s*';      -- загружаем только непустые строки
```

Из мухи – слона

Шаг 1: загружаем словарь (словарь Ефремовой, 51K существительных)

<https://harrix.dev/blog/2018/russian-nouns/>



<https://ru.wiktionary.org/wiki/чуха>

Из мухи – слона

Шаг 2: формируем ребра графа

<https://postgrespro.ru/docs/postgresql/16/pgtrgm>

<https://postgrespro.ru/docs/postgresql/16/btree-gin>

<https://postgrespro.ru/docs/postgresql/16/textsearch-indexes>

```
CREATE EXTENSION pg_trgm;  
CREATE EXTENSION btree_gin;  
  
CREATE INDEX ON dict USING gin(  
    length(word)      -- для этого - btree_gin  
    , word gin_trgm_ops -- для этого - pg_trgm  
);
```

Из мухи – слона

Шаг 2.1: формируем наборы LIKE-шаблонов

```
CREATE TABLE pair AS
WITH src AS MATERIALIZED (
  SELECT
    *
  FROM
    dict
  , LATERAL (
    SELECT
      array_agg( -- 'муха' -> {'_уха','м_ха','му_а','мух_'}
        overlay(word PLACING '_' FROM i)
      ) patterns -- массив всех возможных LIKE-шаблонов замены одной буквы
    FROM
      generate_series(1, length(word)) i
  ) T
)
```

Из мухи – слона

Шаг 2.2: ищем «парные» слова по набору шаблонов

```
SELECT
  src.word src
, dst.word dst
FROM
  src
, LATERAL (
  SELECT
    word
  FROM
    dict
  WHERE
    length(dict.word) = length(src.word) AND -- ищем только в той же длине
    dict.word LIKE ANY(src.patterns) AND    -- одновременно по набору шаблонов
    dict.word <> src.word                    -- исключая исходное слово
) dst;
```

Из мухи – слона

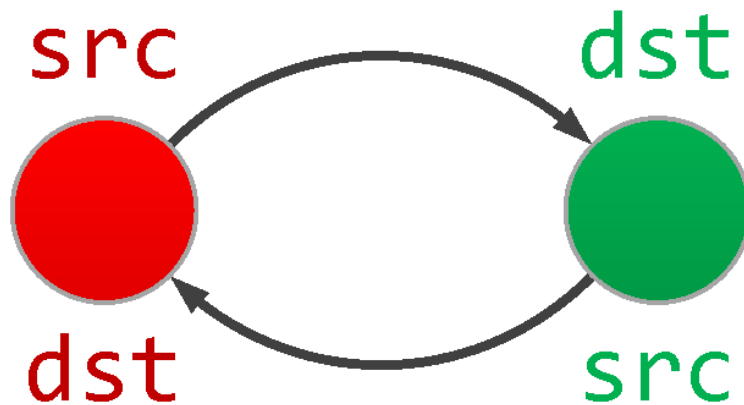
Шаг 2.2: ищем «парные» слова по набору шаблонов

| # | node, ms | io.rd, ms | io.wr, ms | tree, ms | rows | RRbF | Q | loops | |
|----------|------------|-----------|------------|------------|----------|---------|-------|--------|---|
| | 136.758 | 8.542 | 56'461.607 | 35'916 | 102'602 | | | | ▼ итоговые результаты (2.2MB = rows=35'916 x width=64) |
| 0 | 111.292 | | 56'461.607 | 35'916 ▼ | | | | | Nested Loop |
| 1 | | | | | | | | | CTE src |
| 2 | 43.648 | | 672.306 | 51'301 | | | | | -> Nested Loop |
| 3 | 13.046 | .198 | .016 | 51'301 | | | | | -> Seq Scan on dict dict_1 |
| 4 | 410.408 | | 615.612 | 51'301 | | 51'301 | | | -> Aggregate |
| 5 | 205.204 | | 461'709 ▼ | 51'301 | | 51'301 | | | -> Function Scan on generate_series i |
| 6 | 67.725 | | 740.031 | 51'301 | | | | | -> CTE Scan on src |
| 7 | 461.709 | 136.560 | 8.526 | 55'610.284 | 51'301 ▼ | 102'602 | 66.7% | 51'301 | -> Bitmap Heap Scan on dict |
| 8 | 55'148.575 | | 564'311 ▼ | 51'301 | | | | | -> Bitmap Index Scan on dict_length_word_idx (cost=0.00..0.13 rows=13 width=0) (actual time=1.075..1.075 rows=11 loops=51'301) Index Cond: ((length(word) = length(src.word)) AND (word ~ ANY (src.patterns))) Buffers: shared hit=10'558'976 |
| Planning | | | | | | | | | |
| | .246 | | | | | | | | Planning Time |
| | 107.390 | | 56'568.997 | | | | | | Execution Time |

Из мухи – слона

Шаг 2.3: индексируем ребра графа

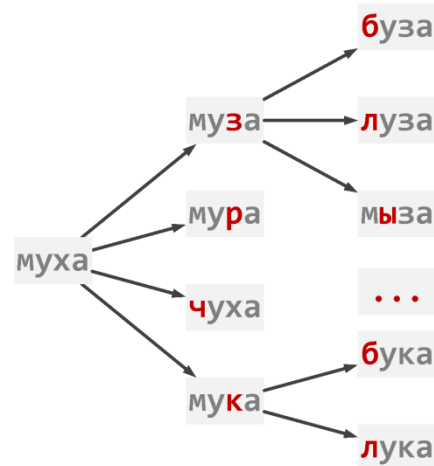
```
CREATE UNIQUE INDEX ON pair(src, dst);
```



Из мухи – слона

Шаг 3.1: запускаем «волну»

```
WITH RECURSIVE param(src, dst) AS (  
  VALUES('муха', 'слон')  
)  
, s2d AS (  
  SELECT  
    0 i                -- счетчик шага  
    , ARRAY[src] path  -- уже пройденный волной путь  
    , ARRAY[src] diff  -- "прирост" на новом шаге  
  FROM  
    param  
  UNION ALL  
    ...
```

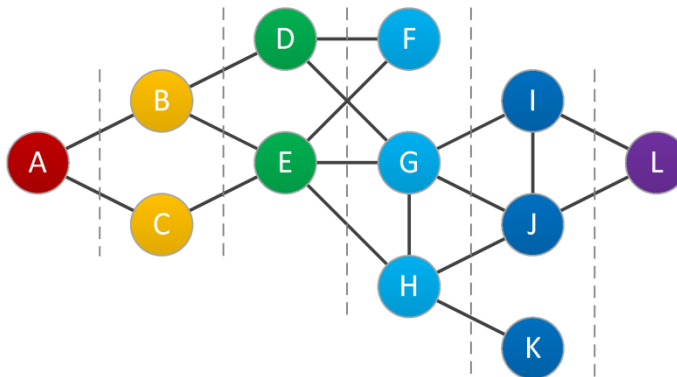


Из мухи – слона

Шаг 3.1: запускаем «волну»

```
...
UNION ALL
SELECT
    i + 1
  , T.path || dst$ -- добавляем узлы в пройденный путь
  , dst$
FROM
    s2d T
  , LATERAL (
    -- тут поиск узлов следующего шага волны -->
  ) X
  , param
WHERE
    dst <> ALL(path) AND -- останавливаемся, дойдя до цели
    i < 100              -- не более чем за 100 шагов
)
```

```
-- следующий шаг волны
SELECT
    array_agg(dst) dst$ -- сворачиваем в массив все достигнутые узлы
FROM
    pair
WHERE
    src = ANY(T.path) AND -- для всех оснований добавляем следствия,
    dst <> ALL(T.path)     -- которые еще не принадлежат пути
```



Из мухи – слона

Шаг 3.1: запускаем «волну»

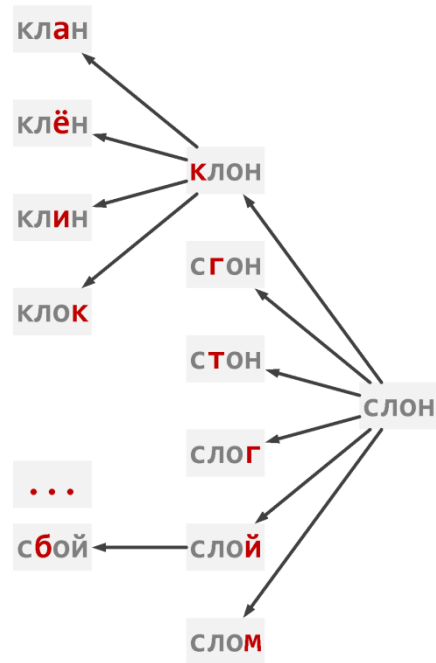
```
i | path | diff
0 | {муха}
  | {муха}
1 | {муха,муза,мука,мура,чуха}
  | {муза,мука,мура,чуха}
2 | {муха,муза,мука,мура,чуха,буза,луза,мыза,бука,лука,рука,сука,фука,щука,аура,бура,дура,кура,мера,сура,тура,фура,чоха,чума}
  | {буза,луза,мыза,бука,лука,рука,сука,фука,щука,аура,бура,дура,кура,мера,сура,тура,фура,чоха,чума}
...
11 | ...
```

Всего 11 шагов – и мы на месте!

Из мухи – слона

Шаг 3.2: запускаем «волну» обратно

```
, d2s AS (  
  SELECT  
    (SELECT max(i) FROM s2d) i -- уже знаем, сколько будет шагов  
    , ARRAY[dst] path          -- теперь начинаем «с конца»  
    , ARRAY[dst] diff  
  FROM  
    param  
  UNION ALL  
    ...
```

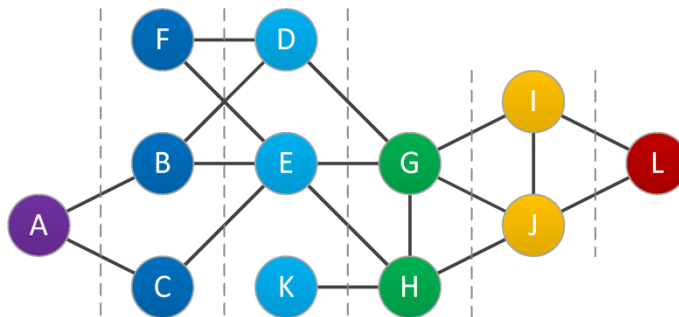


Из мухи – слона

Шаг 3.2: запускаем «волну» обратно

```
...
UNION ALL
SELECT
    i - 1          -- счетчик теперь идет «вниз»
  , T.path || dst$
  , dst$
FROM
    d2s T
  , LATERAL (
    -- тут поиск узлов следующего шага волны -->
  ) X
  , param
WHERE
    src <> ALL(path) AND -- останавливаемся, дойдя до начала
    i > 0
)
```

```
-- следующий шаг волны
SELECT
    array_agg(dst) dst$
FROM
    pair
WHERE
    src = ANY(T.path) AND
    dst <> ALL(T.path)
```



Из мухи – слона

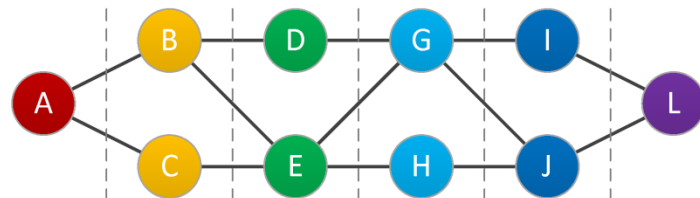
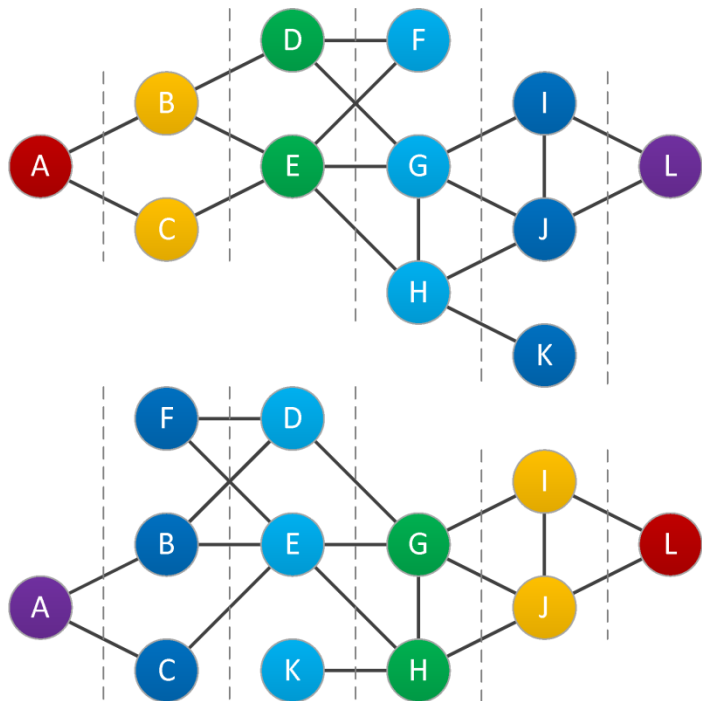
Шаг 3.2: запускаем «волну» обратно

```
i | path | diff
11 | {слон}
   | {слон}
10 | {слон, клон, сгон, слог, слой, слом, стон}
   | {клон, сгон, слог, слой, слом, стон}
 9 | {слон, клон, сгон, слог, слой, слом, стон, клан, клён, клин, клок, клоп, клот, крон, вгон, угон, смог, стог, сбой, стан, стен, стог, сток, стол, стоп}
   | {клан, клён, клин, клок, клоп, клот, крон, вгон, угон, смог, стог, сбой, стан, стен, стог, сток, стол, стоп}
...
0 | ...
```

... обратный отсчет

Из мухи – слона

Шаг 3.3: совмещаем «туда» и «обратно»



Из мухи – слона

Шаг 3.3: совмещаем «туда» и «обратно»

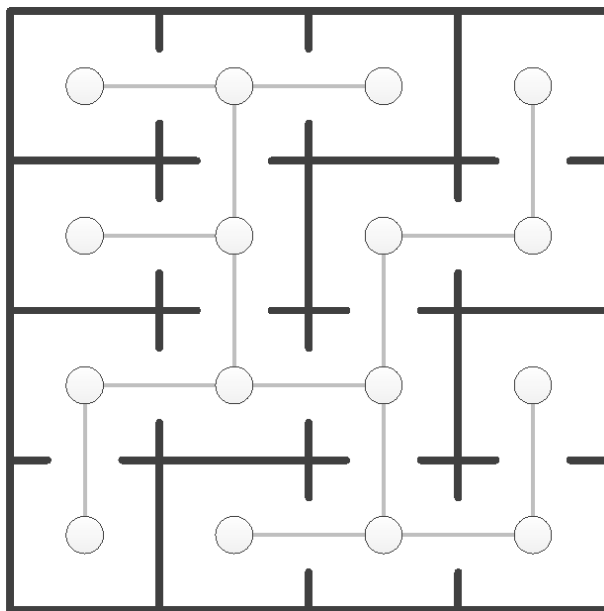
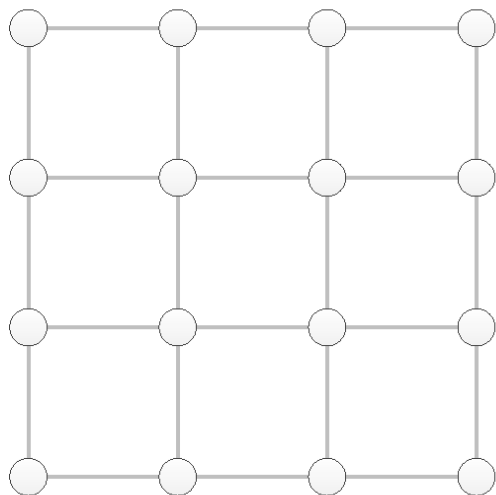
```
SELECT
  i
, unnest(ARRAY( -- потенциально, тут может оказаться несколько слов
    SELECT unnest(s2d.diff) -- развернули первый массив
  INTERSECT ALL
    SELECT unnest(d2s.diff) -- развернули второй массив
)) word
FROM
  s2d
JOIN
  d2s
  USING(i);
```

| i | word |
|----|------|
| 0 | муха |
| 1 | мура |
| 2 | фура |
| 3 | фара |
| 4 | фарт |
| 5 | фаут |
| 6 | паут |
| 7 | плут |
| 8 | плот |
| 9 | клот |
| 10 | клон |
| 11 | слон |

Алгоритм Прима

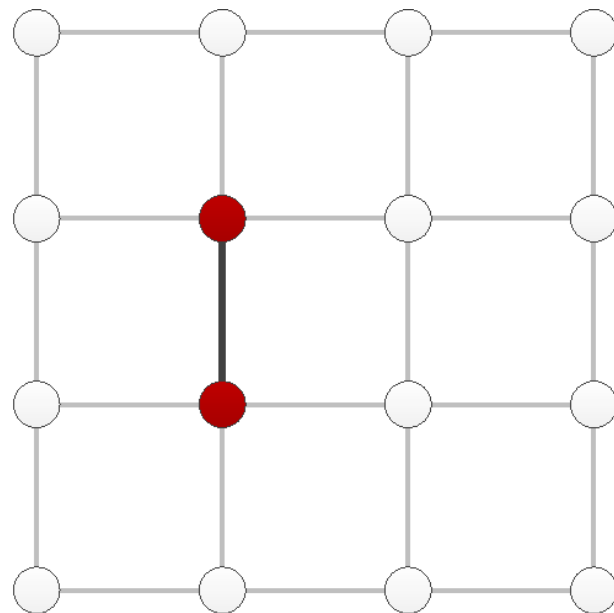
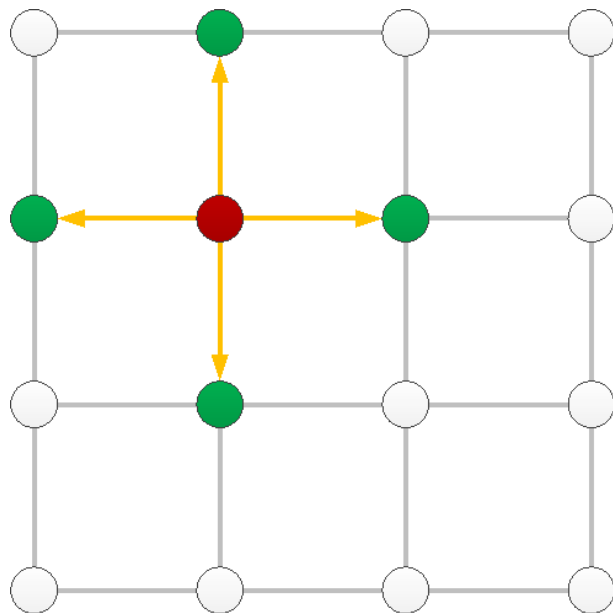
Построение минимального остовного дерева связного взвешенного графа.

https://ru.wikipedia.org/wiki/Алгоритм_Прима



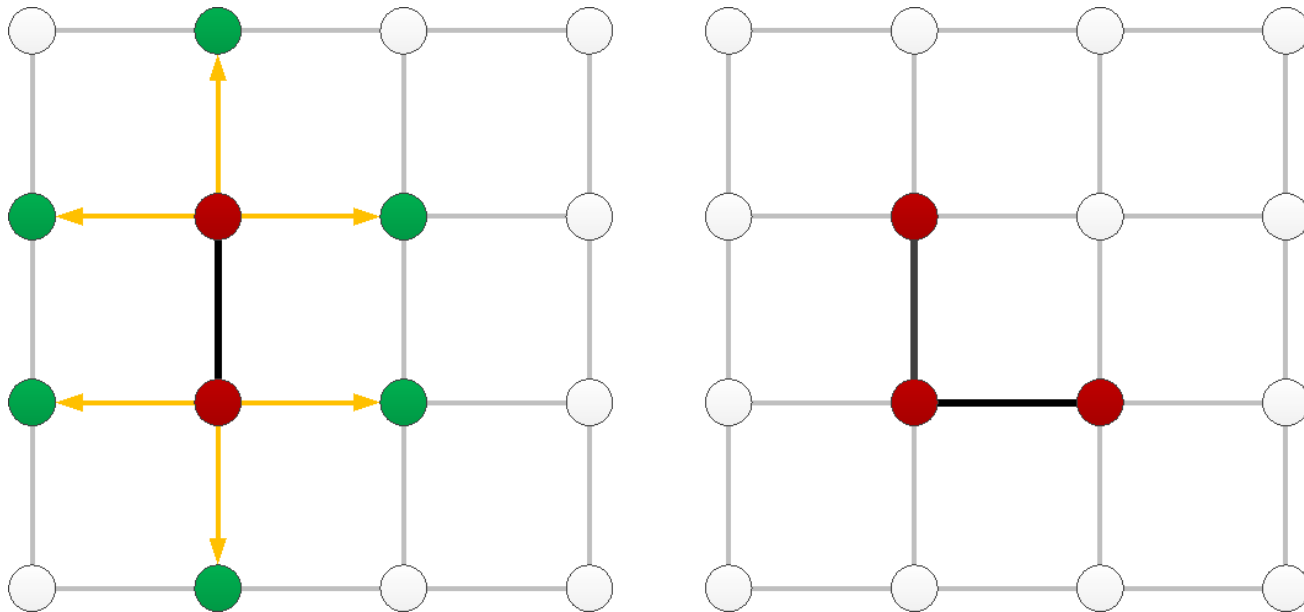
Алгоритм Прима

Шаг 1: стартуем с произвольной точки



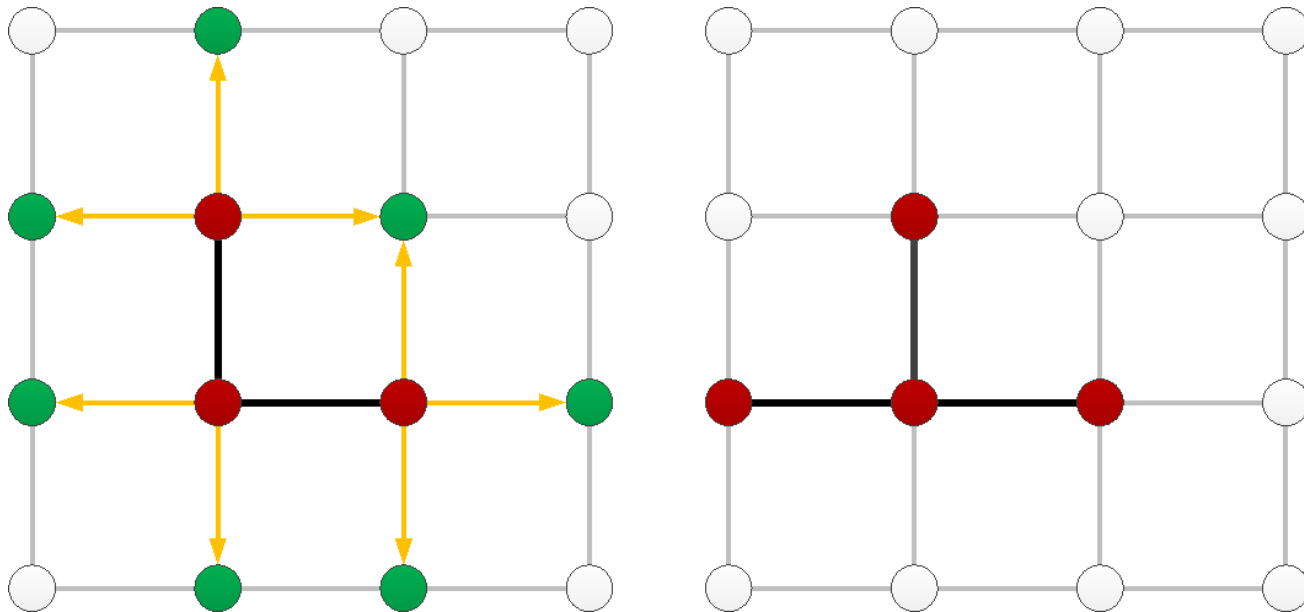
Алгоритм Прима

Шаг N: добавляем вершину из достижимых за минимальное расстояние



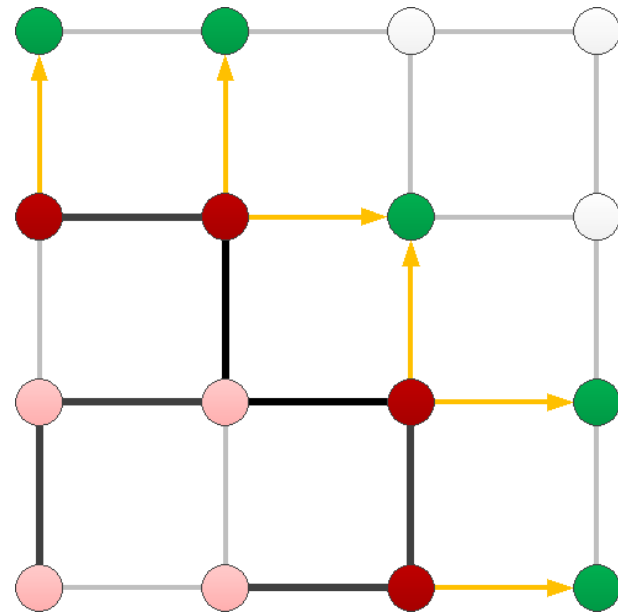
Алгоритм Прима

Шаг N: добавляем вершину из достижимых за минимальное расстояние



Алгоритм Прима

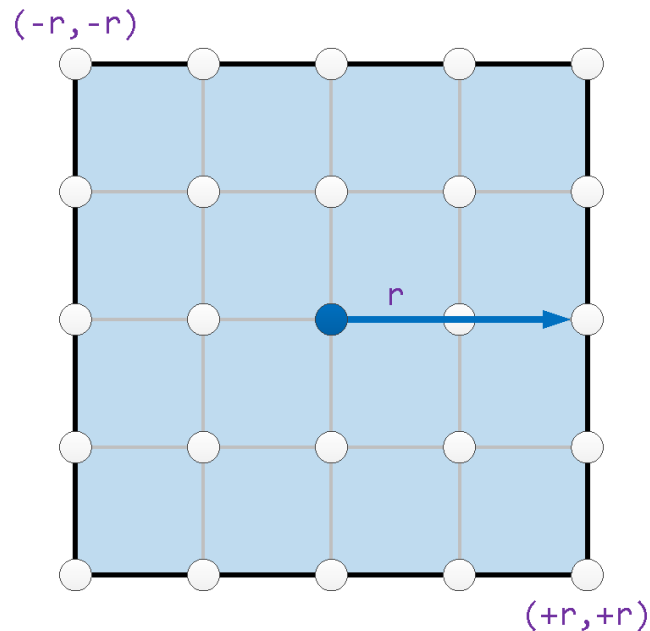
Шаг N: добавляем вершину из достижимых за минимальное расстояние



Строим лабиринт

Шаг 1: обозначим границы лабиринта

```
-- задаем "радиус" лабиринта
WITH RECURSIVE sz(r) AS (
  VALUES(2)
)
-- границы лабиринта
, box AS (
  SELECT
    box( -- прямоугольник по противоположным углам
      point(-r, -r)
      , point(+r, +r)
    )
  FROM
    sz
)
```

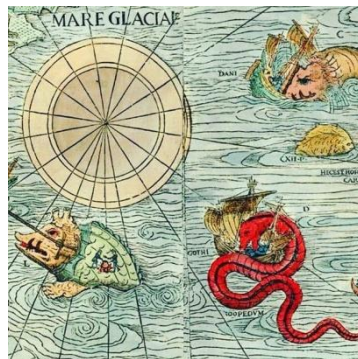


Строим лабиринт

Шаг 2: рекурсивный «цикл»

```
-- цикл выбора ребер
, T AS (
  SELECT
    ARRAY[p]::point[] points    -- уже достигнутые точки
  , ARRAY[p]::point[] wavefront -- фронт "волны"
  , NULL::lseg edge            -- выбранное ребро
  FROM
    SZ
  , point( -- случайная стартовая точка
    (random() * 2 * r)::integer - r
    , (random() * 2 * r)::integer - r
  ) p
  UNION ALL
    ...
)
```

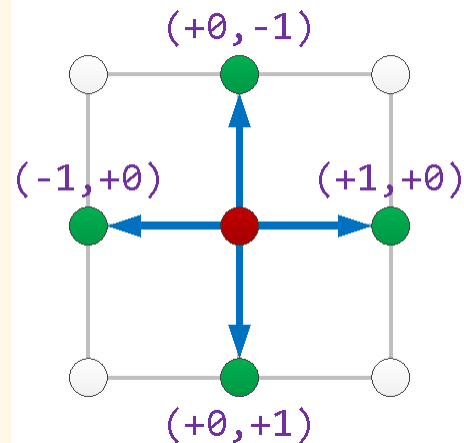
```
...
UNION ALL
  -- ... Hic sunt dracones
  WHERE
    array_length(T.points, 1) = 1 OR -- стартовый шаг
    T.edge IS NOT NULL -- продолжаем, пока можно выбрать ребро
)
```



Строим лабиринт

Шаг 2.1: ребра до «соседей»

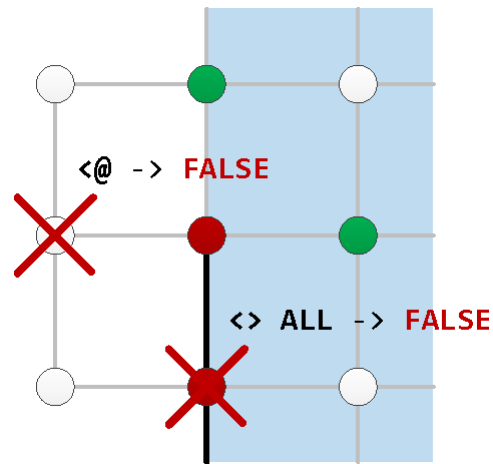
```
FROM
  unnest(T.wavefront) s -- для каждой точки фронта
, LATERAL (              -- формируем все 4 возможных ребра
  VALUES
    (lseg(s, point(s[0] - 1, s[1]    ))) -- лево
  , (lseg(s, point(s[0] + 1, s[1]    ))) -- право
  , (lseg(s, point(s[0],      s[1] - 1))) -- верх
  , (lseg(s, point(s[0],      s[1] + 1))) -- низ
) x(e)
```



Строим лабиринт

Шаг 2.2: проверка принадлежности лабиринту, но не дереву

```
, LATERAL (  
    SELECT e[1] d      -- получаем "целевые" точки из отрезков  
    ) Y  
WHERE  
    d <@ (TABLE box) AND -- точка должна находиться в границах box  
    d <> ALL(T.points)   -- и не должна быть достигнута ранее
```



Строим лабиринт

Шаг 2.3: формируем «фронт»

```
SELECT
  array_agg(e) edges -- все полученные ребра
, array_agg(DISTINCT s::text)::point[] wavefront -- новый фронт - все возможные "источники"
  предыдущего шага
```

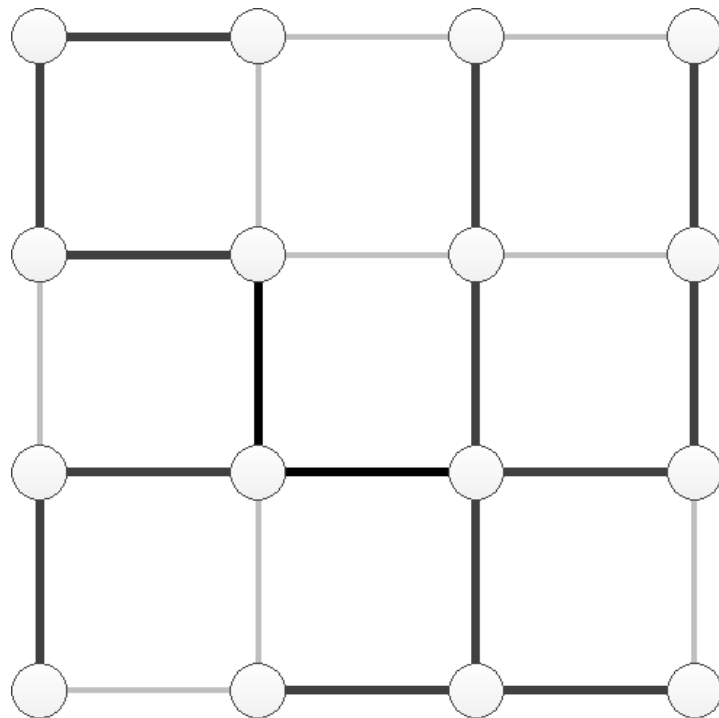
Строим лабиринт

Шаг 2.4: выбираем ребро

```
SELECT
  T.points || X.edge[1]    -- "хвост" выбранного ребра добавляем к достигнутым точкам
, X.wavefront || X.edge[1] -- ... и к фронту
, X.edge
FROM
  T
, LATERAL (
  SELECT
    Z.wavefront
  , Z.edges[(random() * (array_length(Z.edges, 1) - 1))::integer + 1] edge -- выбираем случайное ребро из набора
    -- (array_sample(Z.edges, 1))[1] в PostgreSQL 16+
  FROM
    (
    -- ...
    ) Z
  ) X
```

Строим лабиринт

```
-- цикл выбора ребер
, T AS (
  SELECT
    ARRAY[p::point[] points -- уже достигнутые точки
    , ARRAY[p::point[] wavefront -- фронт "волны"
    , NULL::lseg edge -- выбранное ребро
  FROM
    SZ
    , point( -- случайная стартовая точка
      (random() * 2 * r)::integer - r
      , (random() * 2 * r)::integer - r
    ) p
  UNION ALL
  SELECT
    T.points || x.edge[1] -- "хвост" выбранного ребра добавляем к достигнутым точкам
    , x.wavefront || x.edge[1] -- ... и к фронту
    , x.edge
  FROM
    T
    , LATERAL (
      SELECT
        Z.wavefront
        , Z.edges[(random() * (array_length(Z.edges, 1) - 1))::integer + 1] edge -- выбираем случайное ребро из набора
      FROM
        (
          SELECT
            array_agg(e) edges -- все полученные ребра
            , array_agg(DISTINCT s::text)::point[] wavefront -- новый фронт - все возможные "источники" предыдущего шага
          FROM
            unnest(T.wavefront) s -- для каждой точки фронта
            , LATERAL ( -- формируем все 4 возможных ребра
              VALUES
                (lseg(s, point(s[0] - 1, s[1]))),
                (lseg(s, point(s[0] + 1, s[1]))),
                (lseg(s, point(s[0], s[1] - 1))),
                (lseg(s, point(s[0], s[1] + 1)))
            ) X(e)
            , LATERAL ( -- получаем "целевые" точки
              SELECT e[1] d
            ) Y
          WHERE
            d <@ (TABLE box) AND -- целевая точка должна находиться в границах лабиринта
            d <> ALL(T.points) -- и не должна быть достигнута ранее
        ) Z
    ) X
  WHERE
    array_length(T.points, 1) = 1 OR -- стартовый шаг
    T.edge IS NOT NULL -- продолжаем, пока можно выбрать ребро
)
```



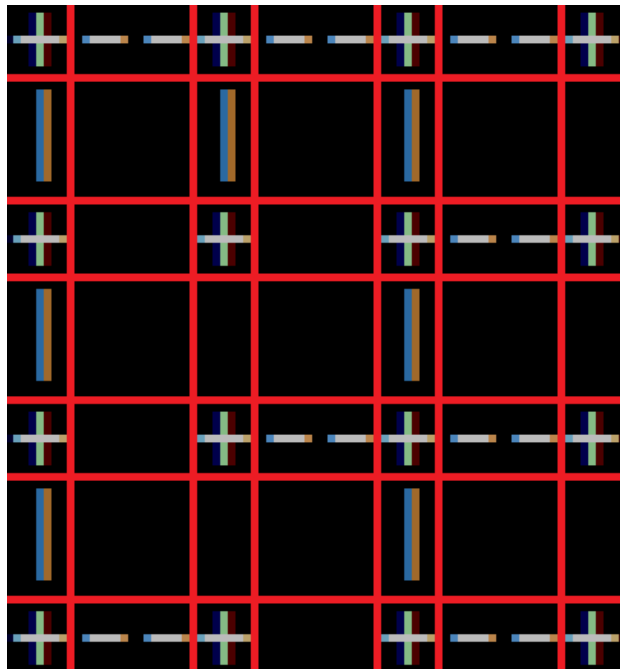
Строим лабиринт

Шаг 3: делаем «красиво»: удвоенная координатная сетка

| | | | | | | |
|----|----|----|----|----|----|----|
| N | eH | N | eH | N | eH | N |
| eV | | eV | | eV | | eV |
| N | eH | N | eH | N | eH | N |
| eV | | eV | | eV | | eV |
| N | eH | N | eH | N | eH | N |
| eV | | eV | | eV | | eV |
| N | eH | N | eH | N | eH | N |

Строим лабиринт

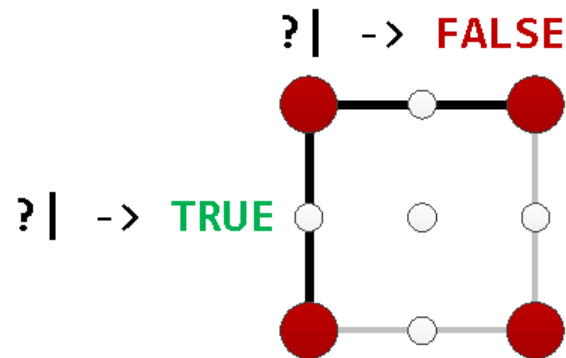
Шаг 3: делаем «красиво»: адаптация шрифта



Строим лабиринт

Шаг 3.1: готовим ребра

```
-- приводим ребра в удобный вид
, edges AS (
  SELECT
    point(edge) * point(2, 0) cx2 -- удваиваем координаты центра ребра
  , CASE
    WHEN ?| edge THEN '|' -- вертикальный отрезок
    ELSE '---'
  END v -- определяем его положение (вертикально/горизонтально)
  FROM
    T
  WHERE
    edge IS NOT NULL
)
```



Строим лабиринт

Шаг 3.2: заполняем сетку

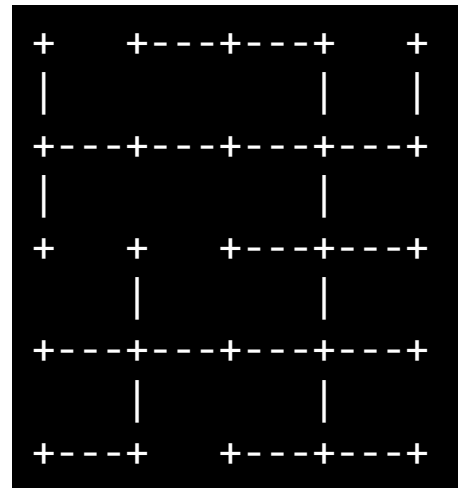
```
-- заполняем координатную сетку
, map AS (
  SELECT
    m.*
  , CASE
      WHEN x % 2 = 0 AND y % 2 = 0 THEN '+'
      WHEN x % 2 = 0 THEN coalesce(v, ' ')
      ELSE coalesce(v, ' ')
    END v
  FROM
    ( ... ) m -- удвоенная координатная сетка ->
  LEFT JOIN
    edges
    ON point(x, y) ~= cx2 -- point = point
)
```

```
( -- удвоенная координатная сетка
  SELECT
    x
  , y
  FROM
    sz
  , generate_series(-r * 2, r * 2) x
  , generate_series(-r * 2, r * 2) y
) m
```


Строим лабиринт

Шаг 3.3: собираем итоговый вывод

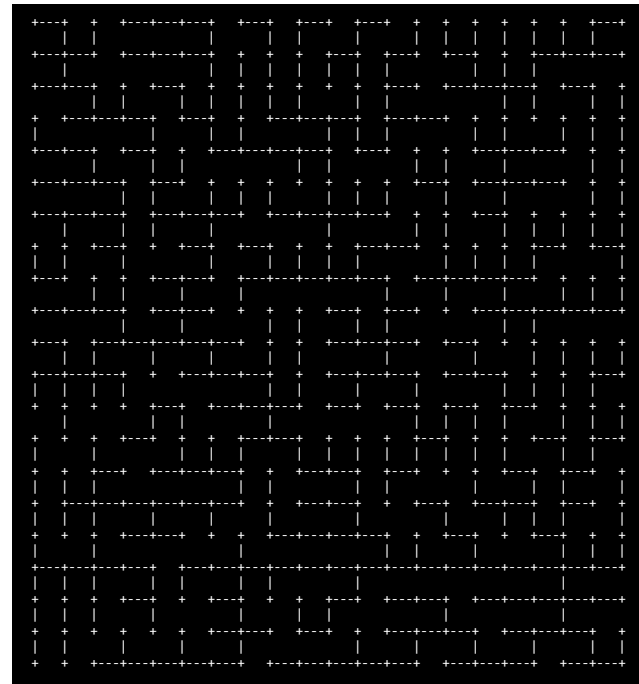
```
-- рисуем картинку
SELECT
    string_agg(v, '' ORDER BY x) maze
FROM
    map
GROUP BY
    y
ORDER BY
    y;
```



r = 2

Строим лабиринт

```
-- приводим ребра в удобный вид
, edges AS (
  SELECT
    point(edge) * point(2, 0) cx2 -- удваиваем координаты центра ребра
  , CASE
    WHEN ?| edge THEN '|' -- вертикальный отрезок
    ELSE '--'
  END v -- определяем его положение (вертикально/горизонтально)
  FROM
    T
  WHERE
    edge IS NOT NULL
)
-- заполняем координатную сетку
, map AS (
  SELECT
    m.*
  , CASE
    WHEN x % 2 = 0 AND y % 2 = 0 THEN '+'
    WHEN x % 2 = 0 THEN coalesce(v, '|')
    ELSE coalesce(v, '--')
  END v
  FROM
    ( -- удвоенная координатная сетка
      SELECT
        x
      , y
      FROM
        SZ
      , generate_series(-r * 2, r * 2) x
      , generate_series(-r * 2, r * 2) y
    ) m
  LEFT JOIN
    edges
    ON point(x, y) ~ cx2 -- point = point
)
-- рисуем картинку
SELECT
  string_agg(v, '' ORDER BY x) maze
FROM
  map
GROUP BY
  y
ORDER BY
  y;
```

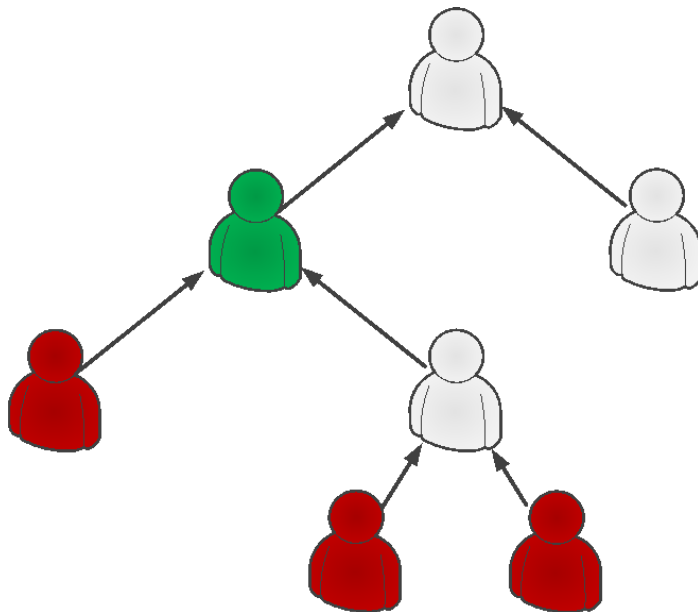


$r = 10$

Ближайший общий предок

Он же «наименьший» или «нижайший» (LCA, lowest (least) common ancestor).

https://ru.wikipedia.org/wiki/Наименьший_общий_предок

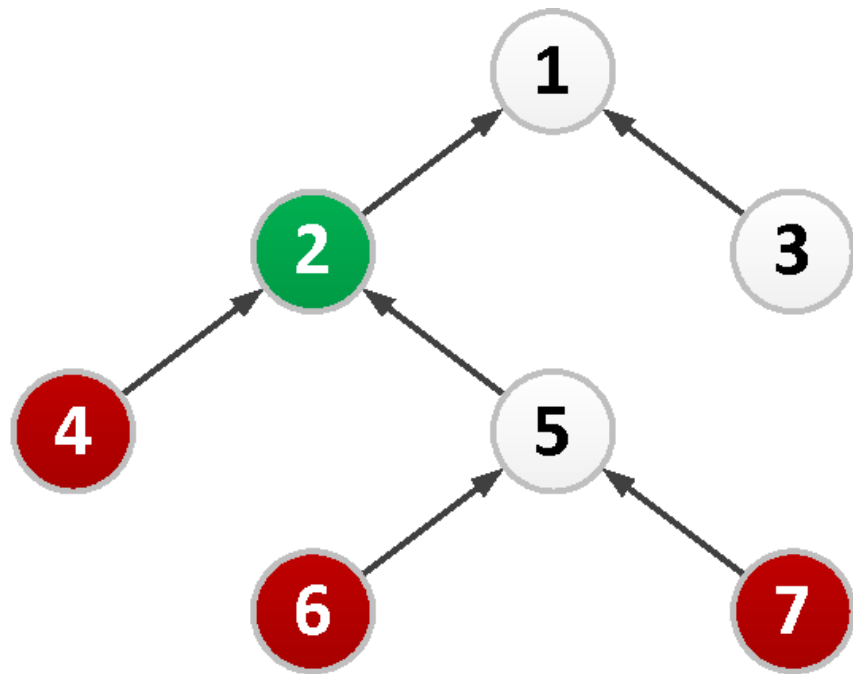


Saby Staff | Saby HRM

Ближайший общий предок

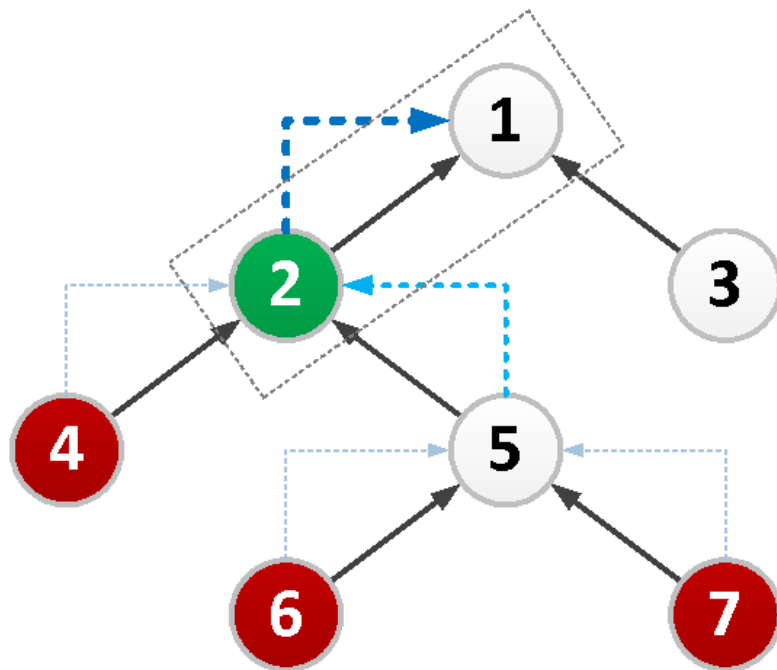
Шаг 1: исходное дерево

```
WITH RECURSIVE tree(id, pid) AS (  
  VALUES  
    (1, NULL)  
    , (2, 1)  
    , (3, 1)  
    , (4, 2)  
    , (5, 2)  
    , (6, 5)  
    , (7, 5)  
)
```



Ближайший общий предок

Наивный алгоритм: общий префикс всех путей



1-2-4

1-2-5-6

1-2-5-7

Ближайший общий предок

Наивный алгоритм: шаг 1 – генерируем все пути

```
, path AS (  
  SELECT  
    ARRAY[id] p  
  FROM  
    tree  
  WHERE  
    id = ANY('{4,6,7}'::integer[]) -- отбираем исходные узлы  
UNION ALL  
  SELECT  
    array_prepend(tree.pid, p) -- новый элемент добавляем в начало  
  FROM  
    path  
  , tree  
  WHERE  
    tree.id = path.p[1] AND -- "шагаем" от последнего добавленного узла  
    tree.pid IS NOT NULL -- выше корня не идем  
)
```

```
p  
{4}  
{6}  
{7}  
{2,4}  
{5,6}  
{5,7}  
{1,2,4}  
{2,5,6}  
{2,5,7}  
{1,2,5,6}  
{1,2,5,7}
```

Ближайший общий предок

Наивный алгоритм: шаг 2 – оставляем нужные пути

```
, path2root AS (  
  SELECT DISTINCT ON(p[l_n]) -- уникализируем по последнему (исходному) элементу  
    p  
  , l_n  
  FROM  
    path  
  , array_length(path.p, 1) l_n -- однократно вычисляем длину массива-пути  
  ORDER BY  
    p[l_n]  
  , l_n DESC -- оставляем самые длинные пути  
)
```

| p | | l_n |
|-----------|--|-----|
| {1,2,4} | | 3 |
| {1,2,5,6} | | 4 |
| {1,2,5,7} | | 4 |

Ближайший общий предок

Наивный алгоритм: шаг 3 – определяем длину общего префикса

```
, pos AS (  
  SELECT  
    i  
  FROM  
    path2root  
  , generate_series(  
    1  
    , (SELECT min(ln) FROM path2root) -- длина кратчайшего из путей  
  ) i -- перебираем все индексы до нее  
  GROUP BY  
    i -- группируем по длине префикса  
  HAVING  
    count(  
      DISTINCT p[:i] -- префикс из i первых элементов ...  
    ) = 1 -- ... единственный  
)
```

i
1
2

Ближайший общий предок

Наивный алгоритм: шаг 4 – получаем последний элемент префикса = LCA

```
SELECT  
  p[(SELECT max(i) FROM pos)] node  
FROM  
  path2root  
LIMIT 1;
```

node
2

Ближайший общий предок

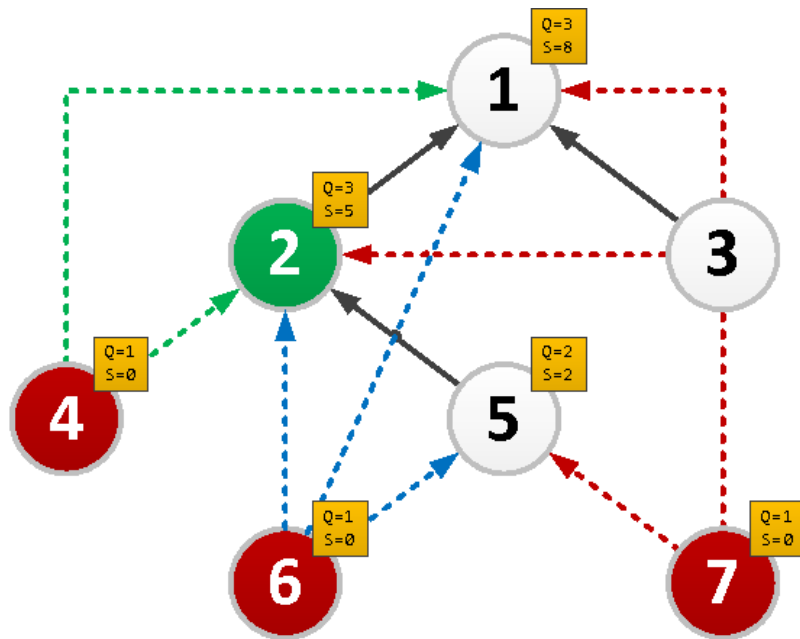
Наивный алгоритм: ... непросто

```
WITH RECURSIVE tree(id, pid) AS (  
  VALUES  
    (1, NULL)  
  , (2, 1)  
  , (3, 1)  
  , (4, 2)  
  , (5, 2)  
  , (6, 5)  
  , (7, 5)  
)  
, path AS (  
  SELECT  
    ARRAY[id] p  
  FROM  
    tree  
  WHERE  
    id = ANY('{4,6,7}'::integer[]) -- отбираем исходные узлы  
UNION ALL  
  SELECT  
    array_prepend(tree.pid, p) -- новый элемент добавляем в начало  
  FROM  
    path  
  , tree  
  WHERE  
    tree.id = path.p[1] AND -- "шагаем" от последнего добавленного узла  
    tree.pid IS NOT NULL -- выше корня не идем  
)
```

```
, path2root AS (  
  SELECT DISTINCT ON(p[ln]) -- уникализируем по последнему (исходному) элементу  
    p  
  , ln  
  FROM  
    path  
  , array_length(path.p, 1) ln -- однократно вычисляем длину массива-пути  
  ORDER BY  
    p[ln]  
  , ln DESC -- оставляем самые длинные пути  
)  
, pos AS (  
  SELECT  
    i  
  FROM  
    path2root  
  , generate_series(  
    1  
    , (SELECT min(ln) FROM path2root)  
  ) i -- перебираем все индексы до длины кратчайшего пути  
  GROUP BY  
    i -- группируем по длине префикса  
  HAVING  
    count(  
      DISTINCT p[:i] -- префикс из i первых элементов ...  
    ) = 1 -- ... единственный  
)  
  
SELECT  
  p[(SELECT max(i) FROM pos)] node  
FROM  
  path2root  
LIMIT 1;
```

Ближайший общий предок

Счетчик посещений: максимум путей при минимуме длины



Ближайший общий предок

Счетчик посещений: шаг 1 – считаем посещения каждого узла

```
, path AS (  
  SELECT  
    id src -- откуда вышли  
    , id dst -- докуда уже дошли  
    , 0 s   -- за сколько шагов  
  FROM  
    tree  
  WHERE  
    id = ANY('{4,6,7}'::integer[])  
UNION ALL  
  SELECT  
    path.src  
    , tree.pid  
    , s + 1 -- увеличиваем путь на 1 пройденное ребро  
  FROM  
    path  
    , tree  
  WHERE  
    tree.id = path.dst AND -- шагаем от последнего посещенного узла  
    tree.pid IS NOT NULL  
)
```

| src | dst | s |
|-----|-----|---|
| 4 | 4 | 0 |
| 6 | 6 | 0 |
| 7 | 7 | 0 |
| 4 | 2 | 1 |
| 6 | 5 | 1 |
| 7 | 5 | 1 |
| 4 | 1 | 2 |
| 6 | 2 | 2 |
| 7 | 2 | 2 |
| 6 | 1 | 3 |
| 7 | 1 | 3 |

Ближайший общий предок

Счетчик посещений: шаг 2 – считаем посещения каждого узла

```
SELECT
  dst                -- ищем среди всех достигнутых узлов
FROM
  path
GROUP BY
  1                  -- группируем по тому же dst
ORDER BY
  count(src) DESC    -- максимум количества узлов-источников
, sum(s)             -- минимум суммы пройденных путей
LIMIT 1;
```

dst
2

Ближайший общий предок

Счетчик посещений: подключаем PostgreSQL 14+

```
, path AS (  
  SELECT  
    id src  
  , id dst  
  FROM  
    tree  
  WHERE  
    id = ANY('{4,6,7}'::integer[])  
  UNION ALL  
  SELECT  
    path.src  
  , tree.pid  
  FROM  
    path  
  , tree  
  WHERE  
    tree.id = path.dst  
) SEARCH DEPTH FIRST BY dst SET path
```

| src | dst | path |
|-----|-----|--------------------------|
| 4 | 4 | {(4)} |
| 6 | 6 | {(6)} |
| 7 | 7 | {(7)} |
| 4 | 2 | {(4), (2)} |
| 6 | 5 | {(6), (5)} |
| 7 | 5 | {(7), (5)} |
| 4 | 1 | {(4), (2), (1)} |
| 6 | 2 | {(6), (5), (2)} |
| 7 | 2 | {(7), (5), (2)} |
| 4 | | {(4), (2), (1), ()} |
| 6 | 1 | {(6), (5), (2), (1)} |
| 7 | 1 | {(7), (5), (2), (1)} |
| 6 | | {(6), (5), (2), (1), ()} |
| 7 | | {(7), (5), (2), (1), ()} |

Ближайший общий предок

Счетчик посещений: подключаем PostgreSQL 14+

```
SELECT
    dst                -- ищем среди всех достигнутых узлов
FROM
    path
GROUP BY
    1                  -- группируем по тому же dst
ORDER BY
    count(src) DESC -- максимум количества узлов-источников
, sum(array_length(path, 1)) -- минимум суммы пройденных путей
LIMIT 1;
```

dst
2

Итоги

Алгоритм Ли – кратчайший путь в графе


Из мухи – слона

Алгоритм Прима – минимальное остовное дерево

Строим лабиринт (+ геометрические типы данных)

Ближайший предок в дереве

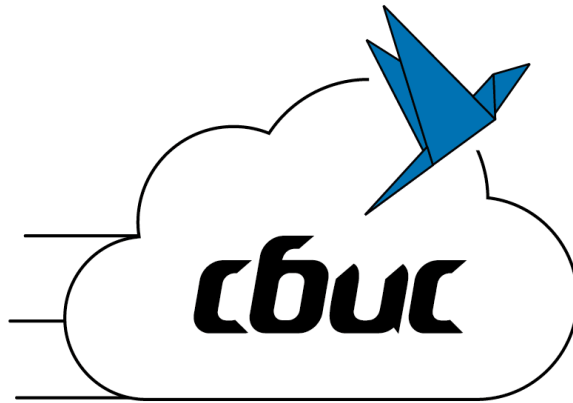
поиск в глубину (SEARCH DEPTH)



<https://explain.tensor.ru/>

55'610.284ms

[#7 BHS](#) | [#8 BIS](#) | [tilemap](#) | [piechart](#)



Спасибо за внимание!

Боровиков Кирилл

kilor@tensor.ru / <https://explain.tensor.ru>

sbis.ru / tensor.ru