Содержание

Monday, May 28, 2018 18:22

Введение

1. NP-полнота

Поиск с возвращением

1. Идея поиска с возвращением (backtracking)
2. Задача о ферзях. Усовершенствования
3. Оценка сложности выполнения: метод Монте-Карло
4. Другие способы программирования поиска с возвращением: рекурсия и использование макросредств

Графы и структуры данных

1. Графы: определения и примеры. Упорядоченный граф
2. Представления графов: матрица инцидентности, матрица смежности, список пар, структура смежности (списки инцидентности)
3. Преобразования представлений

Топологическая сортировка

1. Поиск в глубину
2. Нахождение циклов
3. Алгоритм топологической сортировки:
    - уборкой истоков
    - нумерацией шагов обхода
4. Построение и свойства остовных деревьев при поиске в глубину и в ширину. Поиск в глубину и топологическая сортировка

Остовные деревья графа

1. Задача о связности графа и остовный лес
2. Минимальное остовное дерево. Теорема “о минимальном ребре”:
    - Жадный алгоритм (Краскал)
    - Алгоритм “ближайшего соседа” (Ярник, Прим, Дейкстра)
    - Алгоритм Борувки (O(m*log n))

Кратчайшие пути в графе. Дейкстра

1. Кратчайшие пути от фиксированной вершины
2. Случай неотрицательных весов: алгоритм Дейкстры

Задачи связности

1. Связные компоненты
2. Нахождение компонент двусвязности: точки сочленения графа и их свойства в глубинном остовном дереве
3. Алгоритм нахождения сильно связных компонент (Косарайю)

Алгоритм Кнута-Морриса-Пратта

1. Основные определения
2. Задача точного поиска образца в строке
3. Наивный алгоритм
4. КМП

Алгоритм Ахо-Корасик

1. Задача точного поиска набора образцов
2. Trie
3. Задача о словаре
4. Алгоритм Ахо-Корасик

Суффиксные деревья

1. Суффиксное дерево (СД)
2. Применения СД
3. Наивный алгритм построения СД
4. Алгоритм Укконена

Алгоритм Рабина-Карпа

1. Идея использования хешей для решения задачи точного поиска образца в строке
2. Полиномиальные хеши для строк
3. Алгоритм быстрого вычисления всех хешей текста длины образца
4. Алгоритм Рабина-Карпа

Кратчайшие пути в графе. A*

1. Эвристические алгоритмы
2. Алгоритм A*
3. Эвристические функции

Потоки в графах

1. Алгоритм Форда-Фалкерсона

Раскраска графов

1. Алгоритм полного перебора
2. Перебор с учётом выбора только из 2 цветов
3. Перебор подмножеств размера $\Leftarrow n/3$
4. Вероятностный алгоритм. Сведение к задаче выполнимости
5. Применение раскраски на практике

Клики графа

1. Полные подграфы, клики
2. Применения и сложность задачи построения клик графа
3. Алгоритм нахождения клик на основе поиска с возвращением.

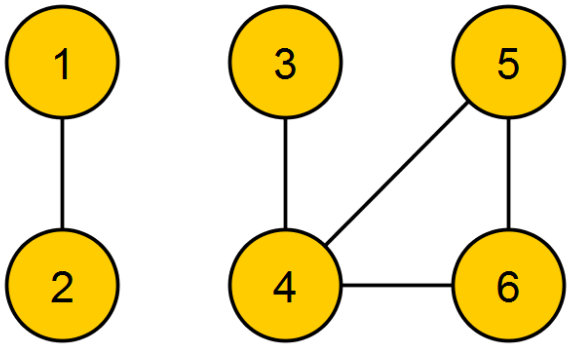
Паросочетания в графах

1. Понятия вершинных и рёберных покрытий
2. Теорема Галлаи
3. Алгоритмы поиска паросочетания в двудольном графе (через максимальный поток и поиск дополняющего пути)

Алгоритмы на графах

2 марта 2018 г. 12:25

Граф - множества E, V, Q
Простейший граф - **неориентированный невзвешенный граф**



Эти графы применяются в том, что касается изоморфизма разных вещей - анализ молекул и т.п.

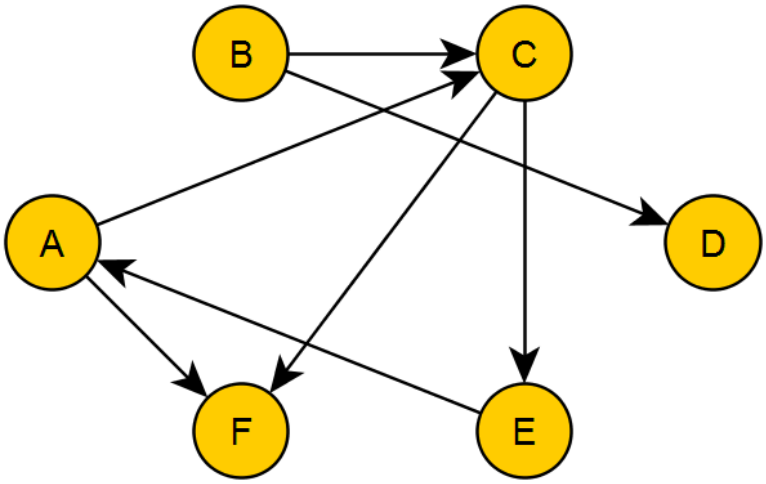
Простейший вариант обхода графа - **в глубину**. Один шаг - посетить любую инцидентную вершину данной, если она не посещена. Применить данную процедуру к каждой посещенной вершине.

```
explore (v){
     visited(v) = true;
     foreach(inc(v, w) == true){
          explore(w);
     }
}
```

В результате применения алгоритма обхода в глубину получается **остовное дерево**

Неориентированные взвешенные графы используются в системах распознавания.

Ориентированные графы применяются при хранении данных, в картографии.



Основные способы представления графа - **матрица инцидентности:**

	A	B	…
A	0	1	…
B	0	0	…
…	…	…	…

Список инцидентности:

```
N
A: B,…
B: …
```

- В матрице быстрее доступ к отдельным элементам, кроме того, матрицы обеспечивают возможность применения линейной алгебры.
- Список занимает намного меньше места, поэтому можно обрабатывать большие графы. Также в списке из-за отсутствия ненужных элементов и локальность памяти быстрее получение инцидентных вершин к данной. В современных системах им отдается предпочтение

Классы задач. Алгоритмы с возвратом

18 февраля 2018 г. 21:37

Классы задач

Полиномиальные задачи (P) - задачи, решающиеся за полиномиальное время, т.е. $t = P_n(a)$, где $a -$ количество элементов
Неполиномиальные задачи (NP) - задачи, решение которых можно проверить за полиномиальное время.
NP-полная задача (NPC) - задача из класса NP, к которой можно свести любую другую задачу этого класса за полиномиальное время
Сильная NP-полная задача (NPCS) - задача, которая

1. Имеет максимальные границы сверху для длины входа
2. Принадлежит классу NP
3. Является NP-полной

Верхняя граница сложности - O

1. $O(1) -$ не зависит от входных данных
2. $O(\log(n)) -$ например, быстрое возведение в степень. Характерно, когда алгоритм разбивается на несколько.
3. $O(n)$ - линейная сложность
4. $O(n\log(n)) -$ например, быстрое преобразование Фурье
5. $O(n^2), O(n^3) \dots, O(n^6) -$ полиномиальная сложность
6. $O(2^n) -$ экспоненциальная
7. $O(n!) -$ факториальная

1-5 считаются "**хорошими**", а 6-7 - "**плохими**".

Бэктрэкинг

- поиск с возвратом. Алгоритм запоминает свои состояния в точках ветвления, и придя в "тупик", возвращается в прошлое состояние и делает другой выбор. Пример - поиск пути в лабиринте.
Такие алгоритмы **детерминированы**, т.е. проверяют все варианты. Это очень неэффективно, если вариантов много - в таких случаях логичнее использовать **эвристические** алгоритмы.
Алгоритмы с возвратом обычно рекурсивны, значит - они используют стек и могут его переполнить, так как расширить стек сложно. Поэтому часто рекурсию нужно "развязать" в линейную структуру

Задача о ферзях

Задача - расставить n ферзей на доске $n \times n$ так, чтобы два ферзя не стояли на одной диагонали или прямой.
Нетрудно посчитать, что вариантов расставить 8 фигур на такой доске - $C_{64}^{8} = 4426165368$
Если учесть, что фигуры не могут стоять на одной вертикали или

горизонтали **(Задача о ладьях),** то количество вариантов перебора существенно сократится - всего $8^8 = 16777216$
Если же учесть атаки по диагоналям, число таких позиций станет $8! = 40320$
Такую задачу можно решить поиском с возвратом - первый ферзь ставится на первую горизонталь, каждый следующий ставится так, чтобы его не били. Если свободного места не осталось, нужно сделать шаг назад и переставить ранее установленного ферзя

Метод Монте-Карло

Задача - рассчитать площадь фигуры, вписанной в квадрат.
Метод состоит в следующем - равномерно распределяются точки и считаются точки, попавшие в фигуру. В таком случае

$$\text{Площадь} = \frac{\text{Сколько попало}}{\text{Сколько всего}} \cdot \text{Площадь квадрата}$$

Сложность алгоритма зависит от:

- Сложности генерации случайных (или псевдослучайных) чисел
- Сложности оценки попадания в фигуру

Рекурсивное программирование поиска с возвратом

Пусть A — множество состояний, $a \in A$. $B \subseteq A$ — множество терминальных состояний (может быть и пустым), $b \in B$.
$\varphi: A \to A$ - рекурсивная функция. Если $\exists b: \varphi(a) = b$, то решение существует, иначе $B = \emptyset$ и решения нет
Рекурсивная функция в общем случае будет выглядеть так:

```
function Rec(a)
    //Сделать действие
    if (a принадлежит B) //Если решение найдено
        return a
    if (тупик)
        return 0
    for (a2 - все варианты следующих шагов из a)
        b := Rec(a2)
        if (b != 0)
            return b
    return 0
```

Граф. Основные определения

Monday, May 28, 2018 22:16

Графы. Некоторые определения

Граф - $G = (V, E)$ - пара множеств $V = \{v_1, \dots, v_p\} \neq \emptyset$ - непустое конечное множество вершин и $E = \{\{v_i, v_j\} \mid v_i, v_j \in V\}$ - множество неупорядоченных пар вершин - **ребер**.

- Вершины и ребра графа - его **элементы**.
- Число вершин графа - **порядок.**
- Ребро (v, v) - **петля**.

Ребро - $e = (v, u) \in E$. $v, u -$ **концы** ребра, ребро e **инцидентно** вершинам v, u.

Степень вершины - количество инцидентных ребер.
Изолированная вершина - вершина степени 0, **висячая** - вершина степени 1.

Путь из A в B - непрерывная последовательность ребер, начавшаяся в A и заканчивающаяся в B. Если начало и конец совпадают, путь **- циклический,** иначе - **открытый**. Если две вершины в пути не совпадают, путь **простой**. Простой открытый путь - **цепь**, замкнутый - **цикл**.

Граф - **полный**, если любые две его вершины смежны.
Граф - **связен**, если между любыми двумя вершинами существует путь.

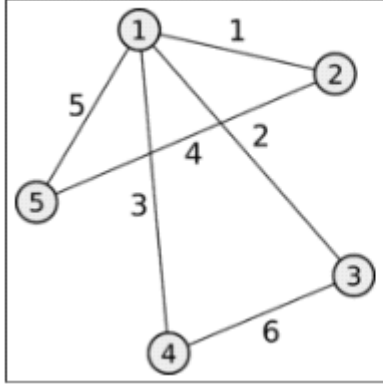
Мост - ребро, удаление которого увеличивает число компонентов связности
Остов - связный подграф, проходящий по всем вершинам

Упорядоченный граф - граф, в котором ребра, выходящие из каждой вершины, пронумерованы

Представления графов

Матрица инцидентности

Матрицей инцидентности для неориентированного графа называется матрица $I(|V| \times |E|)$, для которой $I_{i,j} = 1$, если вершина v_i инцидентна ребру e_j в противном случае $I_{i,j} = 0$
Для ориентированного графа $I_{i,j} = -1$, если $v_i -$ начало дуги e_j, и $I_{i,j} = 1$, если $v_i -$ конец дуги e_j
Пример

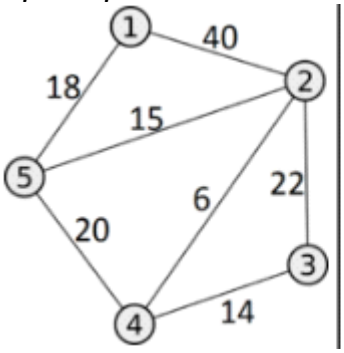
Граф	Матрица инцидентности
	$\begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$

Здесь строки соответствуют вершинам от 1 до 5, а столбцы - рёбрам от 1 до 6. Например, из вершины 4 исходит ребро 3 и 6.

Матрица смежности

Матрицей смежности $A = \|a_{i,j}\|$ графа $G = (V, E)$ называется матрица $A(|V| \times |V|)$, в которой $a_{i,j}$ – количество рёбер (или вес для взвешенного графа), соединяющих вершины v_i и v_j

Пример



$$\begin{pmatrix} 0 & 40 & \infty & \infty & 18 \\ 40 & 0 & 22 & 6 & 15 \\ \infty & 22 & 0 & 14 & \infty \\ \infty & 6 & 14 & 0 & 20 \\ 18 & 15 & \infty & 20 & 0 \end{pmatrix}$$

Список пар

Списком пар графа $G = (V, E)$ называется список из элементов вида (a_{i1}, a_{i2}), где $a_{i1}, a_{i2} \in V$ – такие вершины, для которых существует ребро (дуга) $e_i \in E$.

Список инцидентности

Список инцидентности графа $G = (V, E)$ называется список из элементов вида (a_i, b_i), где $a_i \in V$, а $b_i = < e_{ij} >$ – список всех вершин e_{ij}, инцидентных a_i

Сравнение методов

Преимущества матриц в том, что они позволяют очень быстро проверить, если ли ребро между двумя элементами, но зато они требуют очень много памяти - $O(|V|^2)$ или $O(|V| \cdot |E|)$. Логично применять матрицы, когда число ребёр в графе сопоставимо с числом вершин. Если же граф **разрежен** $|E| \ll |V^2|$, то логичнее использовать

списки - они более эффективны по памяти, но менее эффективны по скорости доступа

	Матрица инцидентности	Матрица смежности	Список пар	Список инцидентности
Объем памяти	$O(\|V\| \cdot \|E\|)$	$O(\|V\|^2)$	$O(E)$	$O(V + E)$
Скорость доступа	$O(1)$	$O(1)$	$O(E)$	$\mathrm{O(V + E)}$

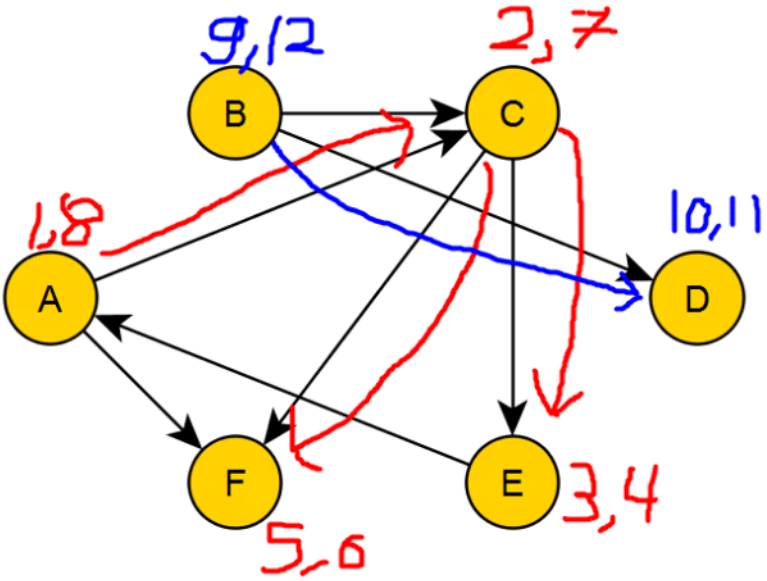
Сложность преобразования представлений графа равна сложности обхода той или иной структуры, т.е. равна объему памяти. Алгоритмы преобразования очевидны.

Обход в глубину. Топологическая сортировка

9 марта 2018 г. 12:19

Обход в глубину для ориентированного графа

Пример



1. Посещена вершина A
    1. Посещена вершина C
        1. Посещена вершина E
        2. Закончена вершина E
        3. Посещена вершина F
        4. Закончена вершина F
    2. Закончена вершина C
2. Закончена вершина A
3. Посещена вершина B
    1. Посещена вершина D
    2. Закончена вершина D
4. Закончена вершина B

Алгоритм на псевдокоде:

```
function doDfs(G[n]: Graph): // функция принимает граф G с количеством вершин n и выполняет обход в глубину во всем графе
  visited = array[n, false]  // создаём массив посещённых вершины длины n, заполненный false изначально

  function dfs(u: int):
    visited[u] = true
    for v: (u, v) in G
      if not visited[v]
        dfs(v)
    for i = 1 to n
      if not visited[i]
        dfs(i)
```

Сложность алгоритма - $O(|V| + |E|)$

Использование поиска в глубину для поиска циклов

Нужно модифицировать алгоритм - пусть он при входе в вершину красит её в серый цвет, а при выходе - в чёрный. Тогда, если мы попали в серую вершину - найден цикл. Посещённые вершины можно сохранять в стеке.

Алгоритм на псевдокоде:

```
func dfs(v: vertex):  // v — вершина, в которой мы сейчас находимся
   color[v] = grey
   for (u: vu ∈ E)
      if (color[u] == white)
         dfs(v)
      if (color[u] == grey)
         print()   // вывод ответа
   color[v] = black
```

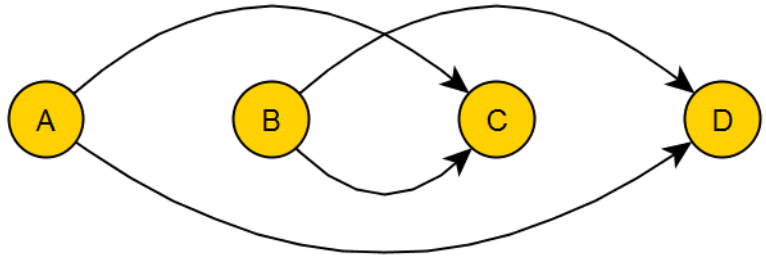
Использование для построения остовного дерева

Для построения остовного дерева годятся различные алгоритмы обхода в графа - например, поиск в ширину или в глубину.

В каждом случае нужно запоминать уже посещённые вершины и не посещать их во второй раз.

Топологическая сортировка

Топологически сортированный граф - граф, у которого из вершины с меньшим номером всегда ребра идут в вершину с большим номером.



Если в графе есть цикл, топологическая сортировка невозможна

Топологическая сортировка уборкой истоков

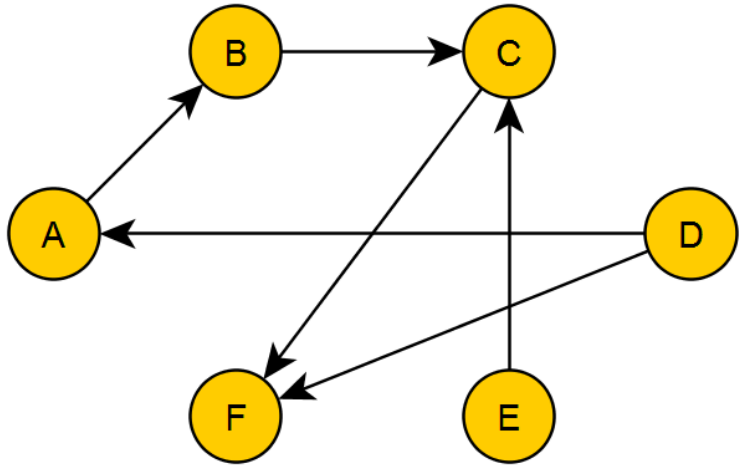
1. Считается количество входящих вершин.
2. Берется вершина с минимальным количеством входящих
3. Эта вершина удаляется
4. Веса пересчитываются
5. Обратно на шаг 2, если в графе есть ребра

На псевдокоде

```
G = (V,E) - граф
L - Пустой список с отсортированными элементами
S - Список элементов без входящих ребер
while (S ≠ ∅)
      for (n: n ∈ S)
            L ← n
            for (m: e = nm ∈ E)
                  delete e
```

```
                    if ({e = nm | e ∈ E} = ∅)
                        S ← m
if (E != ∅)
    return error
else
    return L
```

Пример



A	B	C	D	E	F
1	1	2	0	0	2

1. D:Удалить D, удалить (D->A, D->F).

A	B	C	E	F
0	1	2	0	1

2. A: Удалить A, удалить (A->B)

B	C	E	F
0	2	0	1

3. B: Удалить B, удалить (B->C)

C	E	F
1	0	1

4. E: Удалить E, удалить (E->C)

C	F
0	1

5. C: Удалить C, удалить (C->F)

F
0

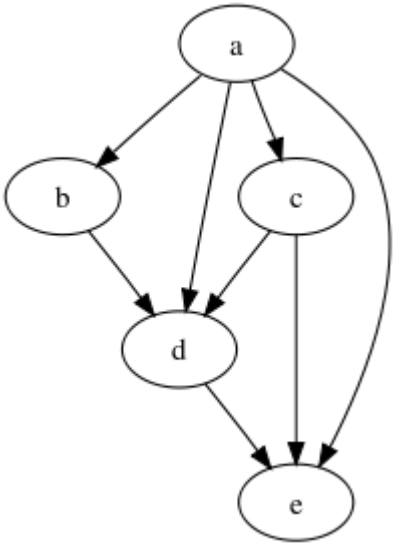
6. F: Удалить F

Алгоритм Тарьяна - топологическая сортировка нумерацией шагов обхода

Топологическую сортировку можно выполнить и с помощью обхода в глубину, отсортировав при этом вершины по времени выхода. Алгоритм при этом похож на поиск цикла в графе.

- Из каждой вершины проводится поиск в глубину
- При входе вершина красится в серый цвет, при выходе - в чёрный. Если вошли в серую вершину, в алгоритме цикл и сортировка невозможна

Пример



Шаг	Текущая	Белые	Стек (серые)	Выход (чёрные)
0	—	a, b, c, d, e	—	—
1	c	a, b, d, e	c	—
2	d	a, b, e	c, d	—
3	e	a, b	c, d, e	—
4	d	a, b	c, d	e
5	c	a, b	c	d, e
6	—	a, b	—	c, d, e
7	d	a, b	—	c, d, e
8	e	a, b	—	c, d, e
9	a	b	a	c, d, e
10	b	—	a, b	c, d, e
11	a	—	a	b, c, d, e
12	—	—	—	a, b, c, d, e
13	b	—	—	a, b, c, d, e

Применение

При запуске любого приложения загрузчик смотрит зависимости приложения и строит граф зависимости. Для адекватного решения этой задачи применяется топологическая сортировка.

Другой вариант топологической сортировки - поиск в глубину с запоминанием времени обработки.

Остовное дерево

9 марта 2018 г. 11:45

Поиск минимального остовного дерева

Дан граф неориентированный граф $G = (V, E)$; $w(u, v)$ − весовая функция. Нужно построить минимальное остовное дерево для G

Остовное дерево G - связный ациклический подграф графа G, в который входят все его вершины

Минимальное остовное дерево - остовное дерево с минимальным суммарным весом рёбер.

Остовный лес

- Не обязательно связный ациклический подграф, включающий все вершины
- Объединение остовных деревьев для каждой компоненты связности

Теорема о минимальном ребре

Разрез графа $G = (V, E)$ - разбиение V на два непересекающихся подмножества $S, T = V \backslash S$ - обозначается как $< S, T >$

Ребро $(u, v) \in E$ **пересекает** разрез $< S, T >$, если $u \in S, v \in T$.

$G' -$ подграф минимального остовного дерева G. Ребро $(u, v) \notin G' -$ **безопасное**, если при добавлении его в G' $G' \cup \{(u, v)\} -$ тоже подграф минимального остовного дерева G.

Рассмотрим неориентированный взвешенный граф $G = (V, E)$ с весовой функций $\omega: E \to \mathbb{R}$. Пусть $G' = (V, E') -$ подграф некоторого минимального остовного дерева G, $< S, T >$ - такой разрез G, что ни одно ребро из E' его не пересекает, а $e = (u, v) \notin E' -$ минимальное из ребер, пересекающих $< S, T >$. Тогда $e -$ безопасное ребро для G'

Алгоритм Борувки

- Минимальное ребро цикла принадлежит остовному дереву
- **Компонента связности графа** - это подграф, в котором можно найти из любой вершину в любую.
  Минимальное ребро, которое соединяет эти компоненты связности, обязательно попадает в остовное дерево графа.

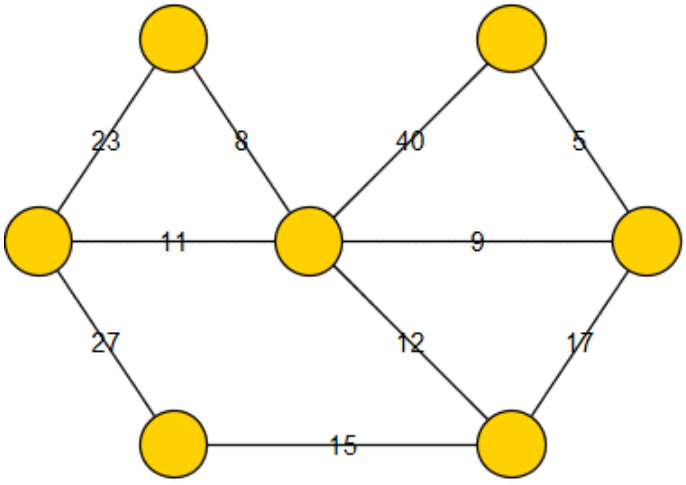
Принцип работы:

1. Изначально каждая вершина графа G— тривиальное дерево, а ребра не принадлежат никакому дереву
2. Для каждого дерева T найдём минимальное инцидентое ему ребро, добавим к остовному дереву все такие рёбра
3. Повторить шаг 2, пока в графе не останется одно дерево T

Алгоритм на псевдокоде

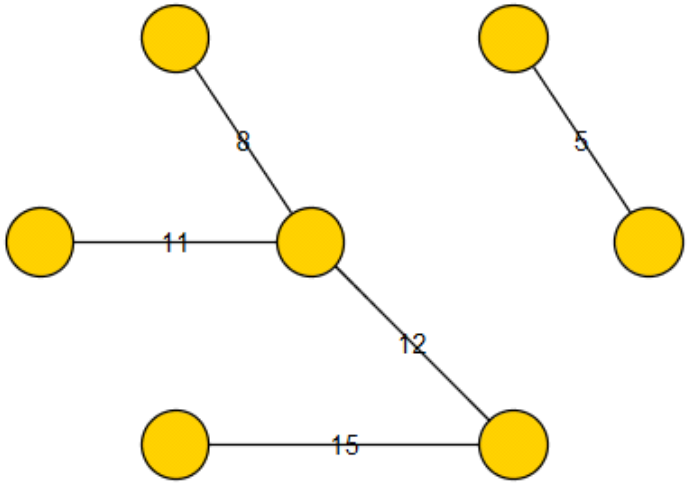
```
  // GG – исходный граф
  // ww – весовая функция
  function boruvkaMST()
      while T.size < n–1
          for k ∈ Component // Component – множество компонент
связности в T. Для каждой компоненты связности вес минимального ребра
= ∞
              w(minEdge[k])=∞
          findComp(T)                              // Разбиваем граф
T на компоненты связности обычным dfs-ом.
          for (u,v) ∈ E
              if u.comp≠v.comp
                  if w(minEdge[u.comp])<w(u,v)
                      minEdge[u.comp]=(u,v)
                  if w(minEdge[v.comp])<w(u,v)
                      minEdge[v.comp]=(u,v)
          for k ∈ Component
              T.addEdge(minEdge[k])  // Добавляем ребро, если его не
```

Пример

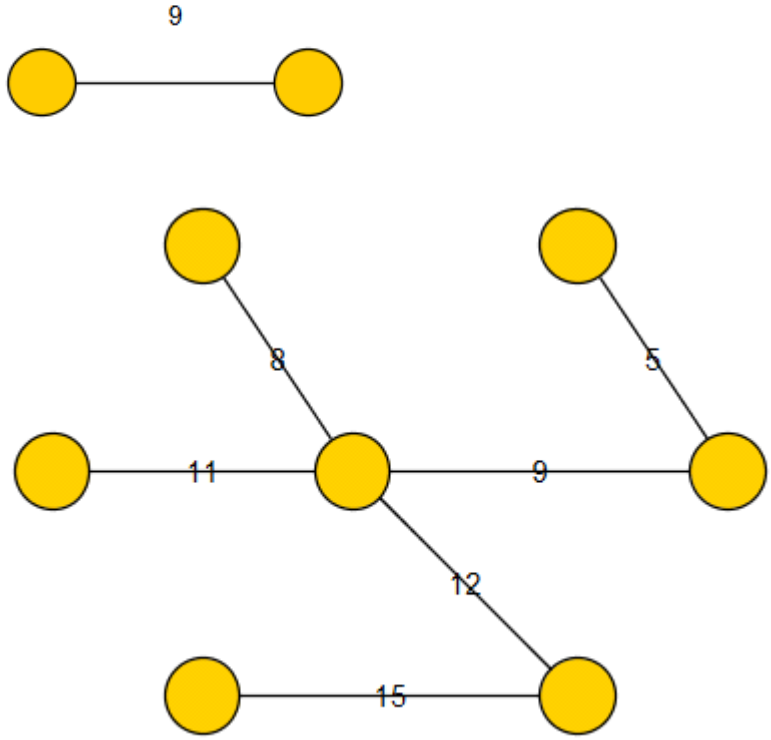


Шаг 1:

К остовному дереву добавляются все минимальные ребра, инцидентные каждой вершине



Шаг 2:
Компоненты связности "стягиваются" в одну вершину. Для соединения выбирается минимальное ребро. Это ребро добавляется к дереву.



Шаги повторяются, пока не останется одна вершина

В худшем случае на каждом шаге количество вершин каждый раз в два раза, т.е. каждая вершина стягивается только с соседней. Поэтому сложность $O(|E| \cdot \log|V|)$

Алгоритм Прима
Берется произвольная вершина.
На каждом шаге к множеству вершин минимального остовного дерева добавляется минимальное ребро, которое соединяет наше множество с остальными вершинами графа.

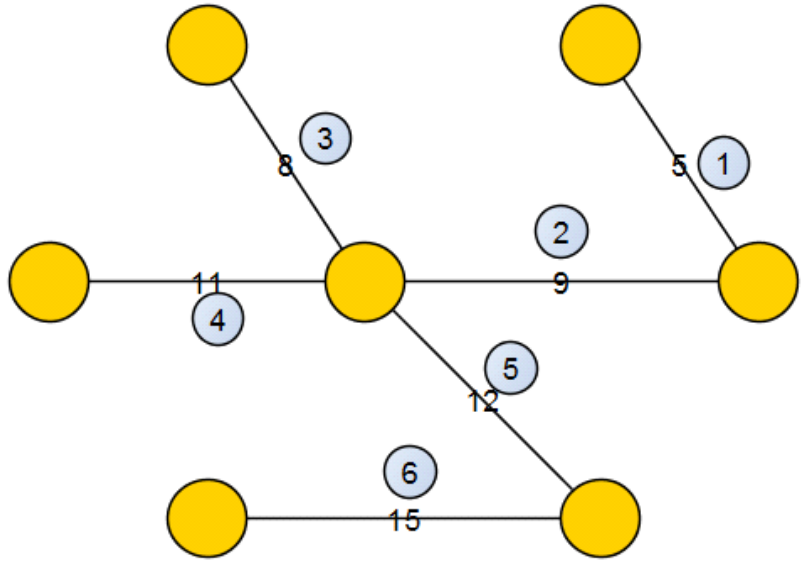
Псевдокод

```
T ← {} //Множество ребер остовного дерева
for (i ∈ V)
    d[i] ← ∞ //Расстояния до i-й вершины дерева
    p[i] ← nil //Предок i-й вершины
d[1] ← 1

Q ← V
v ← Q.Extract_min

while (Q ≠ ∅)
    for (uv ∈ E)
        if (u ∈ Q and w(v,u) < d[u])
            d[u] ← w(v,u)
            p[u] ← v
```

```
        v ← Q.Extract_min
    T ← T + (p[v], v)
```

Пример

В кружках около вершины - номер шага



<u>Жадный алгоритм (Краскала)</u>

Жадный алгоритм - на каждом шаге делается лучшая для данного момента операция.

Идея алгоритма в следующем - в множество E' остовного дерева $G' = (V, E')$ графа $G = (V, E)$ в порядке невозрастания весов добавляются рёбра.

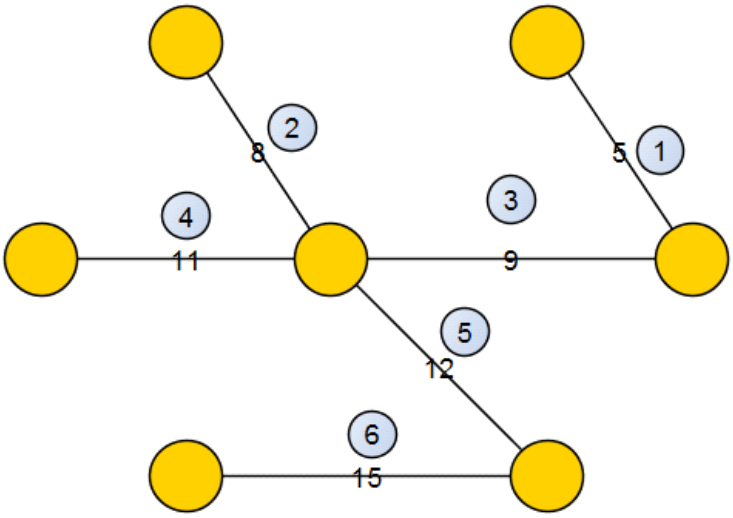
- Если очередное ребро соединяет вершины одной компоненты связности G', то добавление его создаст цикл
- Если же оно соединяет вершины разных компонент, то по теореме о минимальном ребре оно безопасно и может быть включено в граф

<u>Псевдокод</u>

```
G = (V,E)  //исходный граф
G' = (V',E') //результат алгоритма
V' ← V
E' ← ∅
for (uv ∈ E ordered by w(u,v))
    if (Component(u) ≠ Component(v))
        E' ← uv
```

Пример

В кружках около вершины - номер шага



Применение алгоритмов

Применяется в САПР при разводке печатных плат, чтобы определить минимальное количество дорожек. Это уменьшает затраты материала и вероятности ошибок.

Кроме того, многие протоколы в телекоммуникационных сетях используют минимальное остовное дерево, например STP (Spine Tree Protocol). Это используется для обхода циклов в сетях и уменьшения нагрузки на сеть. При широковещательной посылке пакетов это также используется.

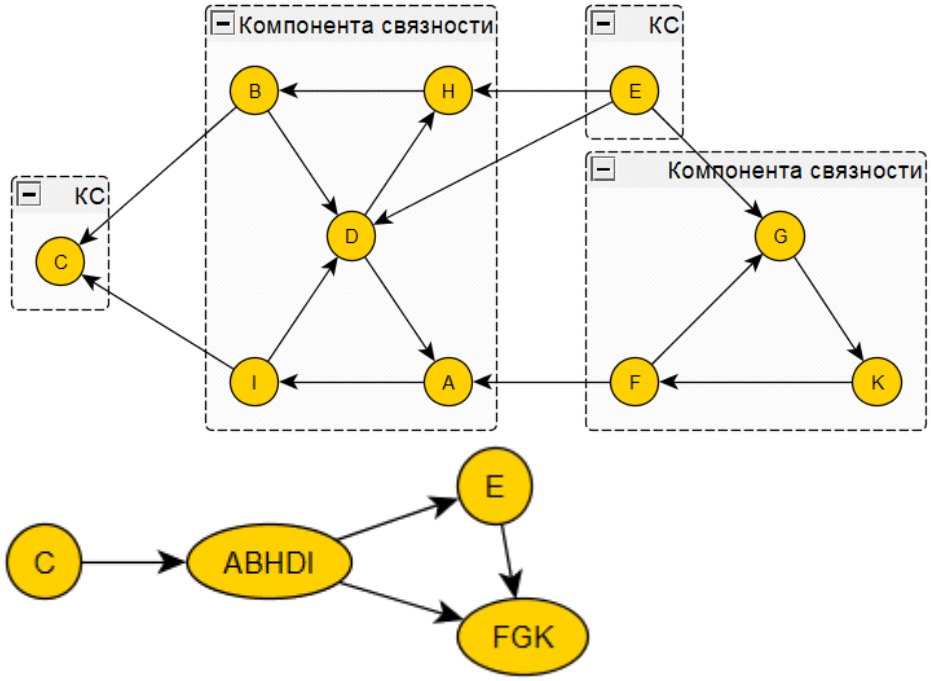
Компоненты связности

Tuesday, May 29, 2018 20:56

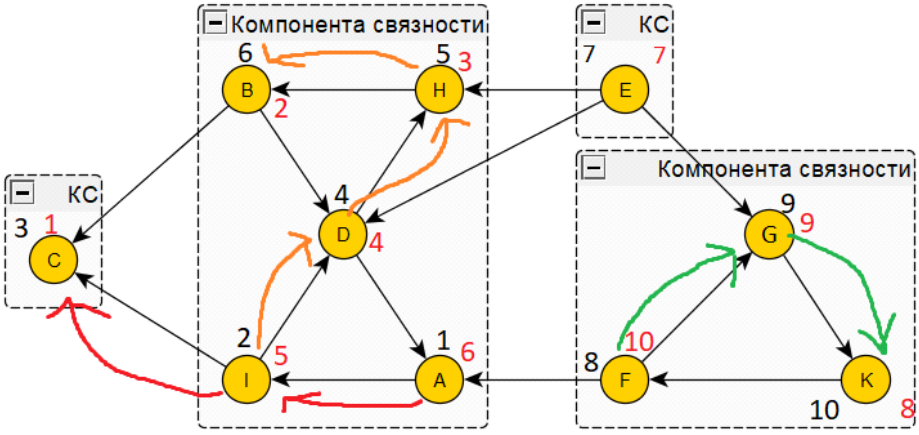
Компоненты связности

Дан граф $G = (V, E)$. Нужно найти число компонент связности графа.
Компонента связности G - максимальный по включению связный подграф графа G.

Алгоритм Косарайю



Любая компонента связности из нескольких вершин содержит циклы.
Сток - та часть графа, в которую входят вершины, но не выходят из неё. Из **истока** выходят вершины, но не входят. Найти исток можно поиском в глубину. Чтобы найти сток, нужно транспонировать граф и найти исток.



Таким образом, поиск происходит в три этапа:

1. Транспонируется граф
2. В графе делается поиск в глубину, для каждой вершины запоминается время окончания обработки:

CBHDIAEKGF

3. Производится обход с конца списка и поиск компонент связности.
   CBHDIAEKGF

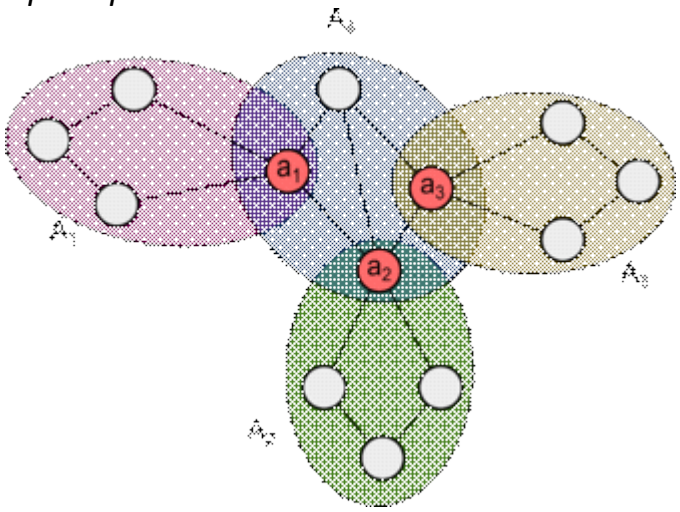
Компоненты двусвязности

Два ребра графа называются **вершинно двусвязными**, если существуют вершинно непересекающиеся пути, соединяющие их концы.

Компоненты вершинной двусвязности графа - блоки - такие подграфы графа, множества рёбер которых - классы эквивалентности вершинной двусвязности, а множества вершин - множества всевозможных концов ребер и соответствующих классов

Точка сочленения графа - вершины, принадлежащие двум блокам графа. Это такая вершина, при которой увеличивается число компонент связности.

Пример



Здесь A_1, A_2, A_3, A_4 - компоненты двусвязности, а a_1, a_2, a_3 — точки сочленения

Теорема. Пусть $G' = (V, E')$ - остовное дерево графа $G = (V, E)$, построенное методом поиска в глубину от $r \in V$. В этом случае $v \in V$ — точка сочленения G тогда и только тогда, когда верно одно из двух

- $v = r, \exists va \in E', \exists vb \in E' : a \neq b$ — это вершина, имеющая по крайней мере двух сыновей в G'
- $v \neq r, \exists vw \in E'$: ни w, ни какой-либо из потомков w в G' не связан ребром ни с одним предком v в G

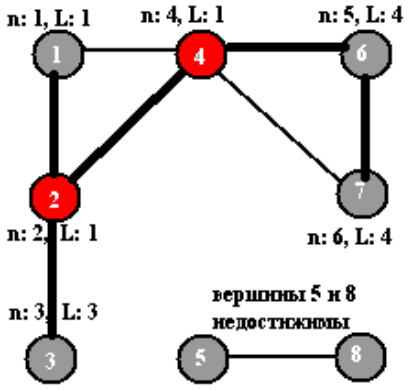
Поиск компонент двусвязности поиском в глубину

Псевдокод

```
function doDfs(G[n]: Graph):
    visited = array[n, false]
```

```
number = array[n, 0] //Номер вершины в порядке обхода
L = array[n] // Минимальный номер вершины, которая связана обратным ребром с данной или с каким-либо из потомков
i = 0
function dfs(u: int, prev: int):
        number[u] = i++
        L[u] = n[u]
        visited[u] = true
        for v: (u, v) ∈ G
                if not visited[v]
                        dfs(v, u)
                        if L[v] < L[u]
                                L[v] = L[u]
                else
                        if (number[v] < number[u]) and (v ≠ prev)
                                if (number[v] < L[u])
                                        L[u] = number[v]
for i = 1 to n
        if not visited[i]
                dfs(i, -1)
```

Пример работы



Для начальной вершины L = n = 1. Для вершины 2 $L = 1$, т.к. из вершины 4 - потомка 2 в остовном дереве - можно вернутся в 1. Аналогично L[6] = L[7] = 4, т.к. из вершины 7 - потомка 6 - можно вернутся в 4.

С помощью этого алгоритма можно найти точки сочленения - согласно теореме, это такие вершины, для потомков которых значение L будет больше или равно номеру самой вершины.

Алгоритм Дейкстры

16 марта 2018 г. 12:04

Постановка задачи

Задача о кратчайшем пути - задача поиска кратчайшего пути от заданной вершины до заданной или до всех остальных.
Существует множество алгоритмов поиска кратчайшего пути:

- **Алгоритм Дейкстры** находит кратчайший путь от одной из вершин графа до всех остальных. Алгоритм работает только для графов без рёбер отрицательного вес
- **Алгоритм Форда-Беллмана** - в отличие от алгоритма Дейкстры, способен корректно обработать отрицательный вес
- **Алгоритм A*** - эвристический алгоритм поиска

Алгоритм Дейкстры (случай неотрицательных весов)

На каждом шаге существует множество уже обработанных вершин и еще не обработанных.

Псевдокод

Инициализация:

```
for (v ∈ V)
    DIST[v] = ∞
    PREV[v] = ∅
DIST[v' ∈  V] = 0  //стартовая вершина
```

Первый шаг:

```
H ← MakeQueue() //формирование очереди с приоритетами для вершины
v'. Все инцидентные вершины попадают сюда
```

Один шаг

```
while (H ≠ ∅)
    v ← min(H) //из очереди с приоритетами выбирается минимальный
    DIST[v]
    for (vu ∈ E) //для каждого инцидентного ребра
        if (DIST[u] > DIST[v] + w(v,u) //если расстояние до
        вершины больше, чем то, по которому мы проходим - условие
        релаксации
            DIST[u] ← DIST[V] + w(v,u)
            PREV[u] ←
            UpdatePriorities(H)
```

Сложность алгоритма

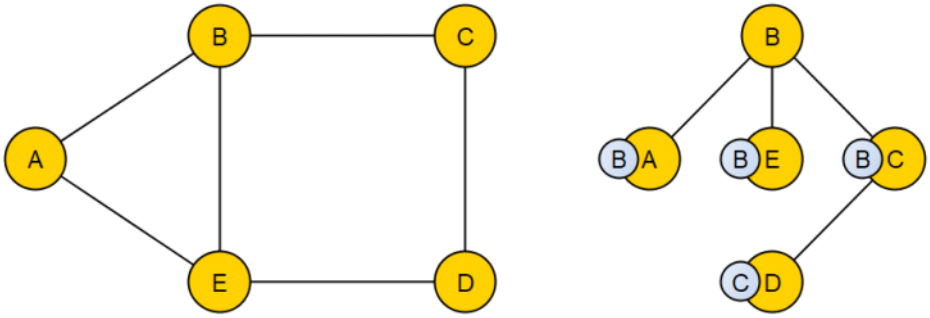
Худший случай - каждый путь содержит в себе все остальные (v вершин). Если каждый такой путь будет хранится к каждой вершине, память будет v^2. Для оптимизации в каждой вершине хранится не весь путь, а только предыдущую вершину, из которой можно попасть в текущую.

Общая сложность алгоритма:
$O(v^2)$ при работе на массиве

$O(\ln(v))$ при работе на куче.
Если количество ребер небольшое, выгоднее использовать реализацию на куче, если же ребер намного больше, чем вершин, лучше использовать работу на массиве.

Применение алгоритма
Одно из применений алгоритма - маршрутизация. Например, алгоритм OSPF (Open Shortest Pass First). Каждый маршрутизатор строит некоторый граф и использует алгоритм Дейкстры в чистом виде.

Пример
Граф невзвешенный



1. Инициализация:
   Берется стартовая вершина, она помещается в множестве рассмотренных вершин. Формируются две структуры данных:
   - Расстояние от этой вершины до всех остальных

A	B	C	D	E

   - Массив из предыдущих вершин, которые нужно пройти, чтобы дойти до следующей вершине

2. Посещена вершина B.

A	B	C	D	E
	0			

3. Ведется обход графа в ширину - просматриваются все ребра, инцидентные данной вершине, из них выбирается минимальное и добавляется в множество.
   Посещена вершина A

A	B	C	D	E
1	0			

4. Посещена вершина C

A	B	C	D	E
1	0	1		

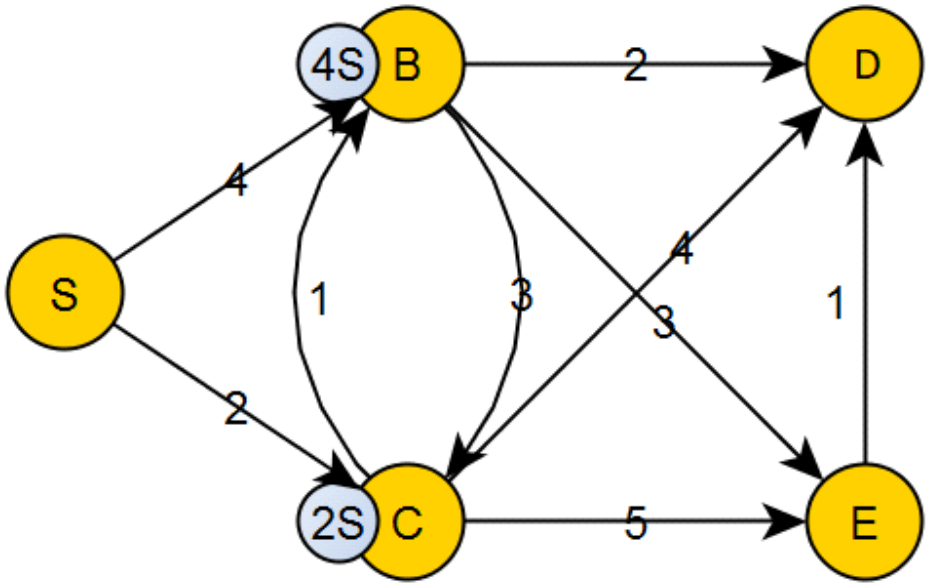
5. Посещена вершина E

A	B	C	D	E
1	0	1		1

6. Посещена вершина D:

A	B	C	D	E
1	0	1	2	1

Пример для взвешенного графа



На данном шаге:

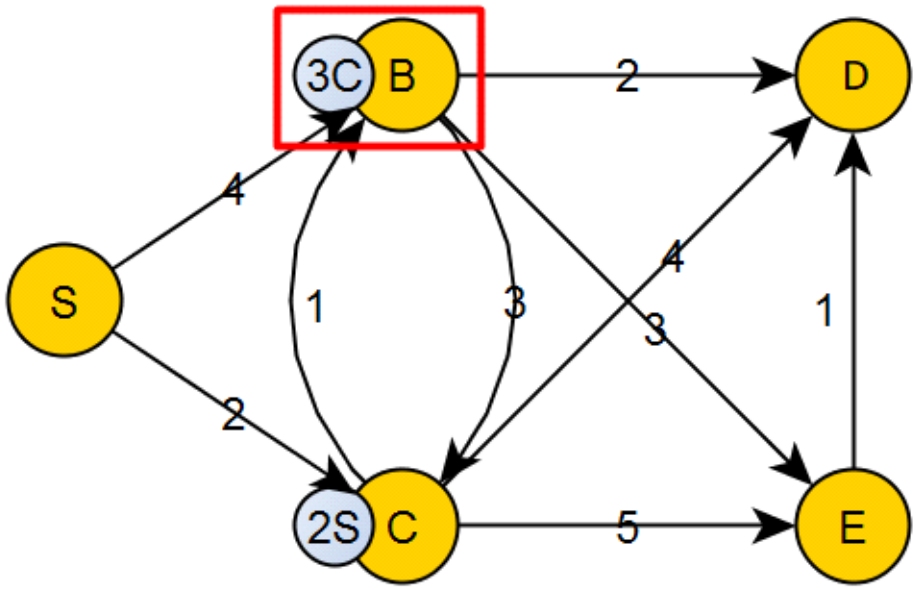
S	B	C	D	E
0	4	2		

Из очереди с приоритетами берется минимальное ребро.
При обработке каждой новой вершины происходит пересчёт весов в массиве - **релаксация**. Если найденный путь до вершины меньше чем тот, что мы нашли ранее, путь уменьшается.

В данном случае идём от C к B

S	B	C	D	E
0	3	2		

…



S	B	C	D	E
0	3	2	6	7

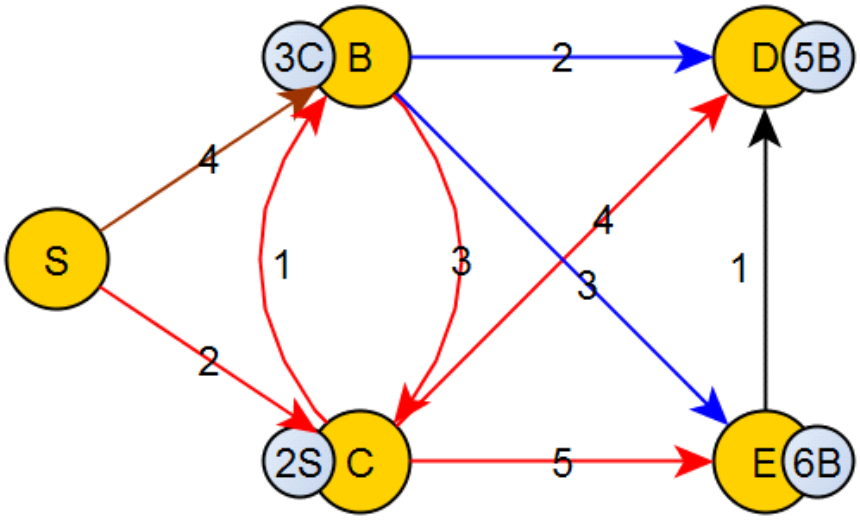
Берется вершина B. Из неё путь в D занимает 2 - релаксируется вершина D.

S	B	C	D	E
0	3	2	5	7

Путь из B в E занимает 3. Релаксируется E

S	B	C	D	E
0	3	2	5	6

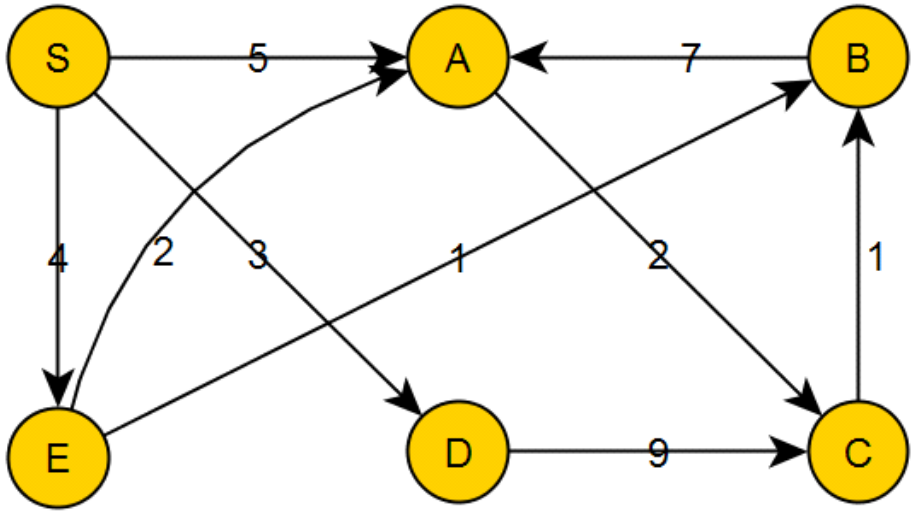
Из вершины B в C переход занимает больше, поэтому релаксации не происходит



Вершина B исключается из очереди.
Осталось обработать вершину E. Из неё путь в D занимает 1, поэтому

релаксации также не происходит.

Еще пример



Инициализация:
DIST:

S	A	B	C	D	E
∞	∞	∞	∞	∞	∞

PREV:

S	A	B	C	D	E
Ø	Ø	Ø	Ø	Ø	Ø

H - очередь с приоритетами

1. Берем S и делаем первую очередь приоритетов

S	A	B	C	D	E
0	∞	∞	∞	∞	∞
S	A	B	C	D	E
Ø	Ø	Ø	Ø	Ø	Ø

Берется путь из S в A

S	A	B	C	D	E
0	5	∞	∞	∞	∞
S	A	B	C	D	E
Ø	*S*	Ø	Ø	Ø	Ø

H:

(5,A)

Берется путь из S в D

S	A	B	C	D	E
0	5	∞	∞	3	∞
S	A	B	C	D	E
∅	S	∅	∅	S	∅

H:

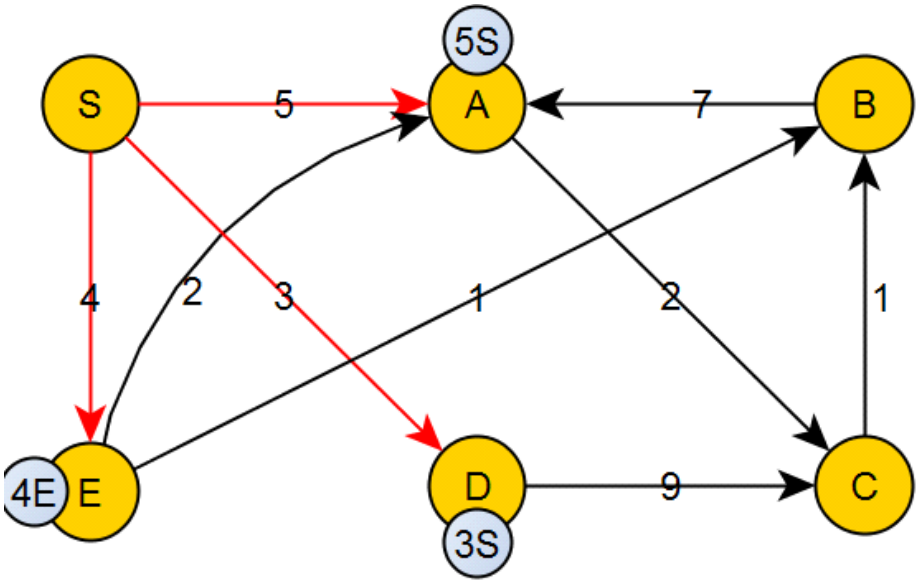
(3,D)
(5,A)

Берется путь из S в E

S	A	B	C	D	E
0	5	∞	∞	3	4
S	A	B	C	D	E
∅	S	∅	∅	S	S

H:

(3,D)
(4,E)
(5,A)



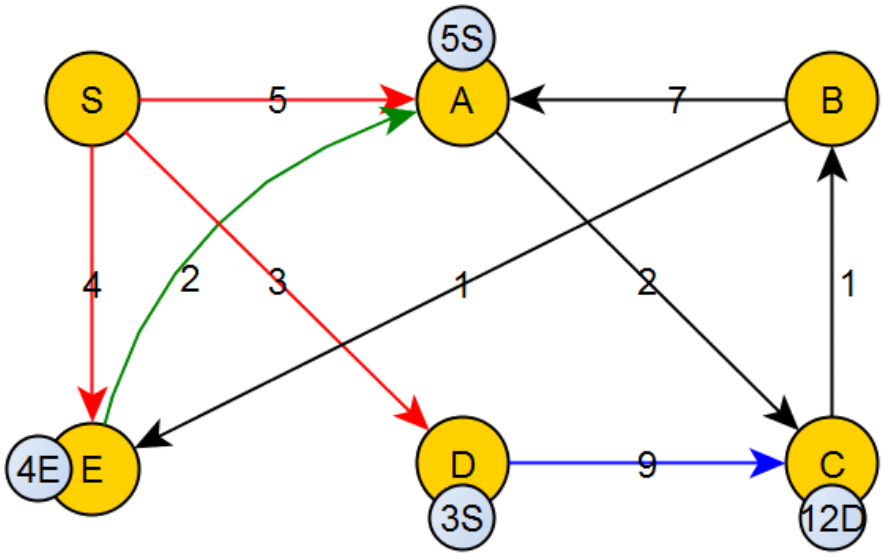
2. Из очереди приоритетов берется и удаляется вершина D.
   Берутся инцидентные ей вершины и производится релаксация:
   Берется ребро DC
   $DIST[C] < DIST[D] + W(DC)$
   $\infty < 3 + 9 \Rightarrow DIST[C] \coloneqq 12; PREV[C] \coloneqq D$

S	A	B	C	D	E
0	5	∞	12	3	4
S	A	B	C	D	E
∅	S	∅	D	S	S

   В очередь добавляется C
   H:

(4,E)
(5,A)
(12,C)

   Обработка D закончена
3. Берется вершина E. При обработке E никакой релаксации не происходит

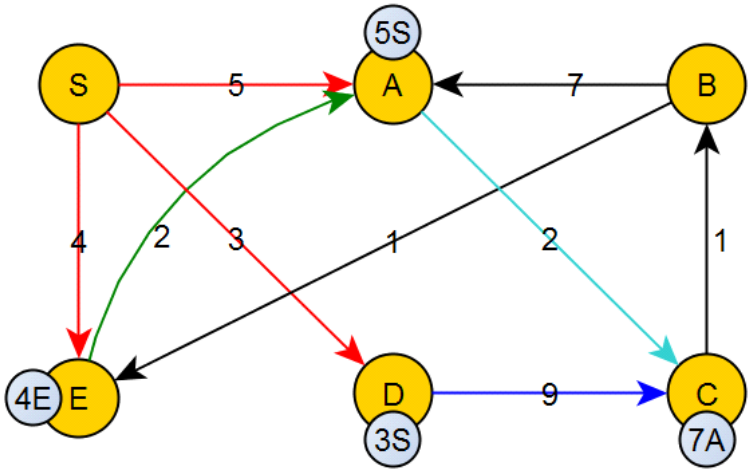


4. Берется вершина A.
   $DIST[C] < DIST[A] + W(AC)$
   $9 < 5 + 2 \Rightarrow DIST[C] = 7; PREV[C] \coloneqq A$

S	A	B	C	D	E
0	5	∞	7	3	4
S	A	B	C	D	E
∅	S	∅	A	S	S

   H:

(7,C)



5. Берется вершина C

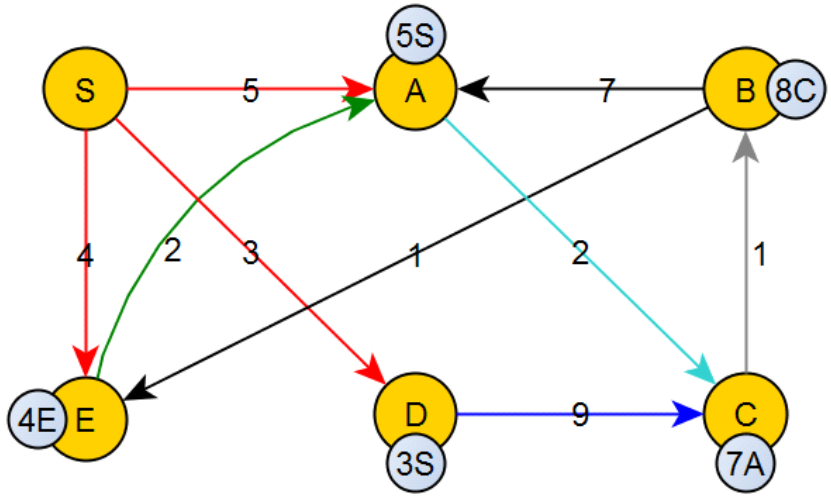
$DIST[B] < DIST[C] + W(CB)$

$\infty < 7 + 1 \Rightarrow DIST[C] := 8; PREV[B] := C$

S	A	B	C	D	E
0	5	8	7	3	4
S	A	B	C	D	E
∅	S	C	A	S	S

H:

(8,B)

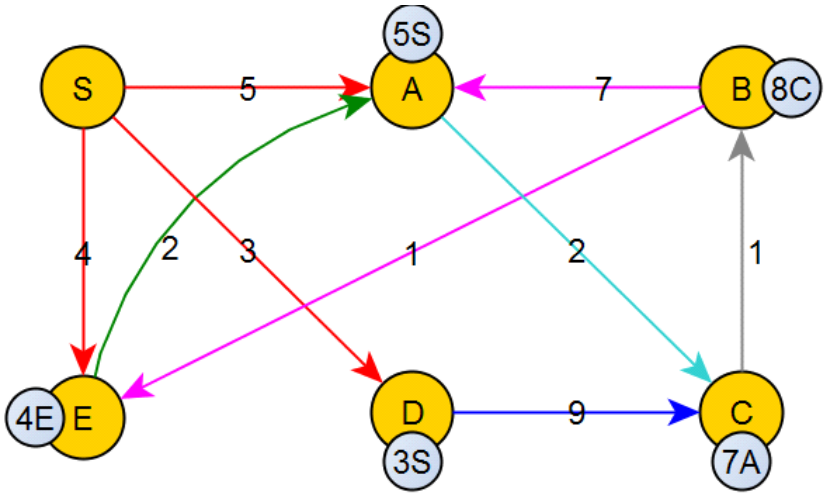


6. Берется вершина B

$DIST[E] < DIST[B] + W(BE)$
$4 < 8 + 1$
$DIST[A] < DIST[B] + W(BA)$
$5 < 8 + 7$
Релаксации не происходит

S	A	B	C	D	E
0	5	8	7	3	4
S	A	B	C	D	E
∅	S	C	A	S	S

H:



7. Очередь приоритетов пуста. Работа алгоритма закончена.

Поиск потоков в графе

30 марта 2018 г. 12:27

Сеть - ориентированный граф $G = (V, E)$, у которого есть взвешенные ребра. У каждого ребра (u, v) есть сопоставленная ему пропускная способность $c(u, v)$ - capacity. Граф имеет один сток и исток

Поток - подграф графа, который имеет пропускную способность, не больше, чем пропускная способность этого графа. Поток в графе - один из возможных путей распространения вещества от истока к стоку.
Формальное определение поток - функция $f: V \times V \to \mathbb{R}$,

- Пропускные способности не нарушены: $\forall u, v \in V: f(u, v) \leq c(u, v)$
- Кососимметричность: $\forall u, v \in V: f(u, v) = -f(v, u)$
- Сохранение потока: $\forall u \in V$

Задача - найти максимальный поток в графе.

<u>Метод Форда-Фалкерсона</u>

- Есть сеть G и поток f. Будем называть **остаточность пропускной способностью** (u, v) разность пропускной способности потока и величины потока
- **Остаточной сетью**, которая порождена потоком, мы будем называть такую сеть, порожденную из тех же вершин, ребра которой будут иметь только остаточную пропускную способность.
- **Дополняющий путь** в сети - обычный путь из истока в сток в остаточной сети.

Если в графе есть хотя бы один дополняющий путь, текущий построенный поток не максимален.

- **Пропускная способность пути** - минимальная из всех пропускных способностей рёбер
- **Остаточная пропускная способность дополняющего пути** - наибольший поток, который мы можем пропустить по этому пути.
- **Разрез** графа - разделение графа на две части

Поток через любой разрез совпадает с величиной самого потока.

- **Пропускная способность разреза** - суммарная пропускная способность ребер - мостов, соединяющих разрез.
- **Минимальный разрез** - разрез с минимальной пропускной способностью.

Следующие утверждения равносильны:

- Поток f максимален
- Остаточная сеть не содержит дополняющих путей
- Существует такой разрез, для которого пропускная способность потока равна пропускной способности разреза.

Алгоритм на псевдокоде

```
Func F-B(G, s, t) //G = (V,E) - входная сеть; s - исток; t - сток
for (uv ∈ E) //Инициализация
    f(uv) ← 0
    f(vu) ← 0
while ((p = G.FindPath(s,t)) ≠ ∅) //Найти путь в остаточной сети поиском в ширину
    c(p) ← min(c(uv): (uv) ∈ p) //Установка пропуской способности пути
    for ((uv) ∈ p) //Формирование остаточной сети
        f(uv) ← f(uv) + c(p)
        f(vu) ← f(uv)
```

Критерий существования дополняющего пути - существование входных ребер в сток.

Эвристические алгоритмы. A*

9 апреля 2018 г. 13:50

Эвристические алгоритмы

Эвристический алгоритм - алгоритм решения задачи, правильность которого для всех возможных случаев не доказана, но про который точно известно, что он дает достаточно хорошее решение в большинстве случаев.
Эвристика - это не полностью математически обоснованный, но при этом практически полезный алгоритм. Особенности:

- Не гарантирует нахождение лучшего решения
- Не гарантирует нахождение решения, даже если оно заведомо существует
- Может давать совершенно неверный результат в определенных случаях.

Алгоритм поиска "A-звездочка" относится к эвристическим методам поиска на графе с положительными (>0) весами рёбер, который находит маршрут с наименьшей стоимостью от одной вершины в другой. Алгоритм был описан в 1968 году.
В отличие от алгоритма Дейкстры, использует эвристическую функцию.

Идея алгоритма: A* пошагово просматривает все пути, ведущие от начальной вершины к конечной, пока не найдет минимальный путь. Как и все эвристические алгоритмы поиска, алгоритм сначала просматривает те маршруты и те ребра, которые кажутся ведущими к цели. От **жадного алгоритма** его отличает то, что при выборе вершины он учитывает весь путь до неё.
В начале работы просматриваются узлы, смежные с начальным. Выбирается тот, который имеет минимальное значение f(x), после чего узел раскрывается. В начале работы алгоритм оперирует с множеством нераскрытых вершин.
Затем $f(x) = h(x) + g(x)$ − к эвристической функции прибавляется путь до текущей вершины.
Структура Вирта - удобная структура для хранения планарных графов. Удобнее, чем списки смежности.

Алгоритм на псевдокоде:

```
Function A*(start, end){
    closedset = {empty}; //Множество закрытых вершин
    openset = {start}; //Множество открытых вершин
    fromset = {empty}; //Множнество пройденных вершин
    G(start) = cost(start, start) = 0; //Стоимость пути до вершины
    F(start) = G(start) + H(start, end); //H - эвристическая функция,
дающая априорную оценку
    while (openset != {empty}){
        curr = minF(openset); //Текущая вершина - вершина с минимальным
```

```
значение пути от начала
        if (curr == end) //Конец
            return path(fromset, start, end);
        remove(curr, openset);
        add(curr, closedset);
        foreach(p - сосед curr){
            if (p в closedset) continue;
            tentative_g = G(curr) + cost(curr, p);
            if (p не в openset){
                add(p, openset);
                tentative_better = true;
            }
            else{
                if (tentative_g < H(p))
                    tentative_better = true;
                else
                    tentative_better = false;
            }
            if (tentative_better == true){
                fromset(p) = curr;
                G(p) = tentative_g;
                F(p) = G(P) + H(p, end);
            }
        }
    }
    return false; //Не нашли
}

Function path(fromset, start, end){ //Восстановление маршрута по
множеству
    pathset={empty};
    curr = end;
    add(curr, pathset);
    if (curr!=start){
        curr = fromset(curr);
        add(curr, pathset);
    }
    return reverse(pathset);
}
```

Эвристическая функция

Доказано, что A* всегда дает лучший маршрут для конкретной эвристической функции.

Под **эвристической функцией** понимается аппарат, позволяющий априорно выбрать тот элемент, который быстрее приведет к решению задачи. Для A* используется несколько эвристических функций:

- **Манхэттенское расстояние**
  При движении по сетке по четырем направлениям
  $h(u) = |ux - goal\ x| + |uy\ - goal\ y|$
- **Расстояние Чебышева**
  При движении по 9 направлениям по сетке
  $h(u) = \max(|ux - goal.x|, |uy - goal.y|)$
- Для планарного графа
  $h(u) = \sqrt{(ux - goal\ x)^2 + (uy - goal\ y)^2}$

Требования к эвристической функции:

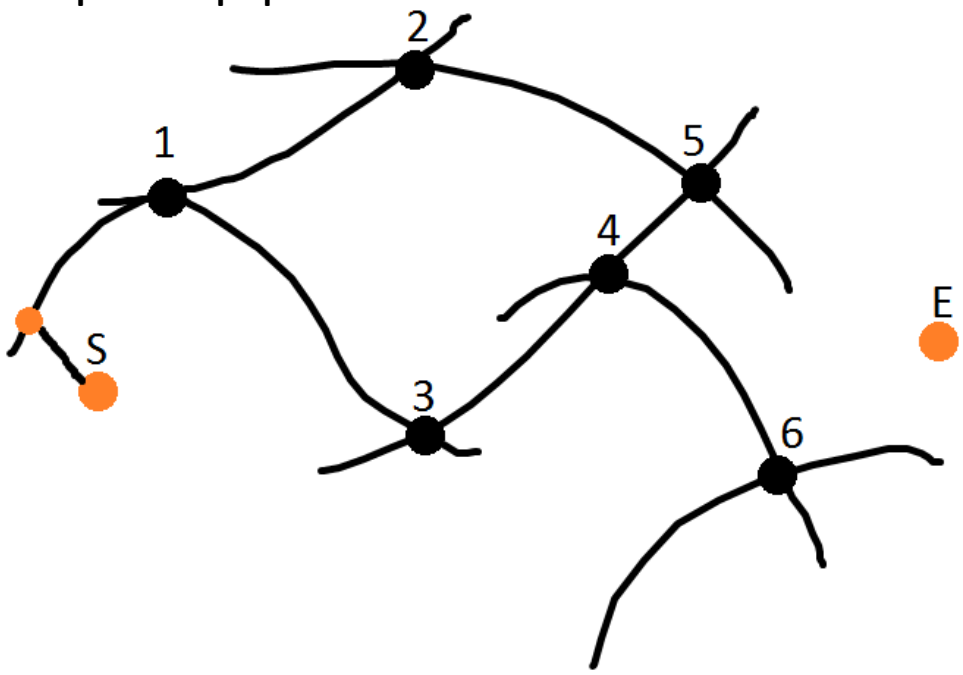
- Монотонность

- Согласование единиц измерения с неэвристической функцией

Свойства A*

1. **Допустимость**. Если решение существует, с помощью этого алгоритма оно будет найдено
2. **Оптимальность**. Найденное решение всегда оптимально.
3. **Эффективность**. Не существует в данный момент алгоритмов, который находят решение быстрее с применением той же эвристической функции. A* в ходе решения раскрывает минимальное количество вершин.

Построение графа



Вершины - пересечение дорог. Для нахождения пересечения линий используется **алгоритм заметающей линии (прямой)**.
Ребро - длина дороги, а не прямое расстояние между вершинами.
При расчёте маршрута самая трудоемкая операция - cost. Поэтому эти данные нужно подготавливать и структурировать заранее. Скорость алгоритма зависит от реализации контейнеров.

Нужно рассмотреть три случая для рассмотрения положения начальной и конечной точки:

1. Точка попала в никуда
   Один из способов решения - построение кратчайшего расстояния до ближайшего ребра.
   Другой способ - расширяющаяся окружность (тоже эвристический алгоритм). Точки пересечения окружности с ребрами - начальные точки алгоритма.
2. Точка попала на ребро
3. Точка попала на вершину

Эта операция тоже трудоемкая. Кроме того, структуры графа должны

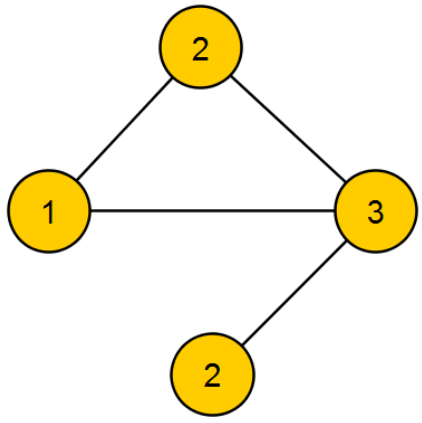
динамически изменятся. Каждое решение не всегда оптимально.

Раскраски графа

Friday, May 11, 2018 12:11

Раскраска графа

Хроматическое число графа - минимальное количество цветов покрасок. Соседей вершины нельзя красить в тот же цвет.



Для такого графа хроматическое число - 3.

Полный граф - такой граф, у которого есть путь из каждой вершины в каждую. Очевидно, что хроматическое число полного графа совпадает с количеством вершин.

(Совсем) наивный алгоритм построения три-раскраски

Пусть дан граф и дана его раскраска. Чтобы проверить её корректность, нужно обойти все ребра и проверить, если одинаковые цвета на концах. Значит, сложность проверки - линейная.

Самый простой вариант - генерация всех возможных три-раскрасок и проверка каждой на корректность. Сложность такого алгоритма - $O(3^n)$, т.к. это число возможных три-раскрасок графа.

Степенную сложность степени не убрать никак, на данный момент самый эффективный алгоритм - $O(1.232^n)$.

Алгоритм 1

Пусть есть некоторая вершина. Мы красим её в первый цвет. Для инцидентных вершин выбор идет не из 3-х цветов, а из 2-х. Сложность такого алгоритма - $O(2^n)$

Другими словами, используется знание, что инцидентные данной вершины нельзя красить в цвет данной

Алгоритм 2

Два соображения:

- Если граф уже раскрашен, то вершины разделяются на множества - первого, второго и третьего цвета. Очевидно, что внутри этих множеств не существует ребер. Если есть n вершин и 3 множества, очевидно, что одно из этих множеств будет иметь мощность $\leq \frac{n}{3}$

вершин.

- Пусть нам заранее известно одно из этих множеств. Если это так, то все оставшиеся вершины будут красится гораздо проще - с линейной сложностью.

Значит, нужно найти такое множество. Способов выбрать такое множество:

$C_n^0 + C_n^1 + C_n^2 + \cdots + C_n^{\frac{n}{3}} \leq n \cdot C_n^{\frac{n}{3}} \leq 1{,}9^n$

- **радиус шара Хемминга**

Значит, сложность алгоритма - $O(1{,}9^n)$

<u>Вероятностный алгоритм (Two-List Coloring)</u>

Посмотрим все цвета для вершин, в которые можно их покрасить. Можно для каждой вершины случайным образом выкинуть один цвет и красить из оставшихся. В таком случае каждая вершина может быть покрасить в один из двух цветов.

Такую задачу можно свести к **задаче выполнимости булевых формул (2SAT)**.

Сведение такое:

Пусть каждый цвет обозначается переменной a_1, a_2, a_3. Нужно покрасить все вершины в какие-то цвета. Это значит, что нужно выбрать хотя бы (и только) один из этих цветов. Т.е. для одной вершины верно следующее

$(a_1 \vee a_2 \vee a_3) \wedge (\overline{a_1} \vee \overline{a_2}) \wedge (\overline{a_1} \vee \overline{a_3}) \wedge (\overline{a_2} \vee \overline{a_3})$

1-й дизъюнкт - вершину нужно покрасить

2,3,4-е - вершину можно покрасить только в один цвет

Ещё нужно учесть ограничение три-раскраски, т.е. добавить ограничение на неодинаковость цвета инцидентных вершин.

$\wedge \left(\overline{a_1} \vee \overline{b_1}\right) \wedge \left(\overline{a_2} \vee \overline{b_2}\right) \wedge (\overline{a_3} \vee \overline{b_3})$

Вычеркнув неиспользуемые цвета получаем задачу 2SAT, где каждый дизъюнкт содержит не больше чем два литерала. Эту задачу можно решить за полиномиальное время.

Задачу 2SAT возможно свести к графу, и решением будет поиск компонент связности

Если найдено решение задачи, то граф точно может быть раскрашен, но обратное неверно - возможно, мы вычеркнули нужные цвета для раскраски. Вероятность того, что раскраска выживет после вычеркивания - $\left(\frac{2}{3}\right)^n$. Это очень мало.

Но если прогнать алгоритм $\left(\frac{3}{2}\right)^n$ раз, то вероятность ошибки - $\frac{1}{e}$.

Если после этого прогнать ещё 100 раз, то вероятность ошибки - $\frac{1}{e^{100}}$

Сложность алгоритма - $O(1.5^n)$

Алгоритм 4

Мы можем перебирать множества инцидентных вершин достаточно эффективно: если одна вершина попала в первое множество, то инцидентные гарантированно туда не попадут. Алгоритм, основанный на этой идее, дает сложность $O(1.45^n)$

Применение

1. Задача планирования. Пусть есть список заказов, которые начинаются и заканчиваются в определенное время. Нужно понять, какое минимальное количество ресурсов нужно выделить для решения этих заказов.
   Сводится так: пересечение заказов по времени - ребро графа. Хроматическое число графа и есть ответ.
2. Есть карта - отображение информации, которое предполагает, что между объектами есть границы (например - политическая карта мира). Задача - определить минимальное число цветов раскрасить карту, чтобы были видны границы между объектами. Задача очевидным образов сводится к раскраске графа.
3. Задача оптимального распределения переменных по регистрам.

Клики графа

Friday, May 18, 2018 12:13

Клики графа

Клика графа - максимальный полный подграф данного графа
Полный подграф - такое подмножество вершин, любые две из которых соединены ребром

Здесь клика - ABC

Наивный поиск клики графа

Решается алгоритмом с возвратов:

1. Берется произвольная вершина и добавляется в клику
2. В граф добавляется одна вершина, инцидентная всем вершинам клики
3. Если ещё остались вершины, вернутся на шаг 2.
4. Если не осталось, удалить соответствующую вершину и вернутся на шаг 2

Сложность алгоритма

- Добавить вершину ничего не стоит
- Для проверки инцидентности нужно перебрать все ребра, которые выходят из вершины, худший случай - $O(|E|)$
- Количество проверок клики из одной вершины - все возможные размещения вершин - $O(|V|!)$
- Это нужно сделать из каждой вершины графа - $O(|V|)$

Итого $O(|E| \cdot |V| \cdot |V|!)$

Оптимизации алгоритма

При таком алгоритме каждая клика встретится несколько раз - для любой клики размера k она будет найдена $k!$ раз.

- Возможно запоминать вершины, которые не подошли для клики размера n-1 и не проверять их при построении клики размера n
- Если уже был обработан полный подграф, то уже не имеет смысла заходить вглубь в него во второй раз.

Алгоритм Брона-Кербоша

Алгоритм использует то, что всякая клика - по определению полный подграф. Значит, можно на каждом этапе отсечь вершины, которые заведомо не приведут к построению клики.

Алгоритм на псевдокоде

```
//Алгоритм использует 3 подмножества для вершин графа:
//compsub - на каждом шаге рекурсии содержит полный подграф для данного шага
//candidates - множество вершин, которые могут увеличить compsub
```

```
//not - множество вершин, которые уже использовались для расширения
compsub
Procedure extend(candidates, not):
    while (candidates ≠ ∅) and ({u ∈ not | ∀v ∈ candidates: ∃uv} =  ∅})
    //пока candidates не пусто и not не содержит вершины,соединённой со
    всеми вершинами из candidates,
        v ← candidates //Выбор вершины из candidates
        compsub ← v
        new_candidates ← candidates /{u ∈ candidates | ∄uv}
        new_not ← not /{u ∈ not | ∄uv}
        //Формируем new_candidates и new_not, удаляя из candidates и
        not вершины, не соединённые с v
        if (new_candidates = ∅) and (new_not = ∅)
            cliques ← compsub //Это клика
        else
            extend(new_candidates, new_not)
        compsub ← compsub/v
        candidates ← candidates/v
        not ← v
        //Удаляем v из compsub и candidates, и помещаем в not
```

Сложность алгоритма - линейная относительно количества клик. Худший случай - $O\left(3^{\frac{|V|}{3}}\right)$

Применение:

- Задача резервирования и проверки надежности
- Проектирование электрический сетей и микросхем
- Поиск сильно связанных групп людей в социальных сетях
- Системы точной идентификации личности в машинном зрении
- Анализ данных в биоинформатике - поиск связанных групп генов и т.п.

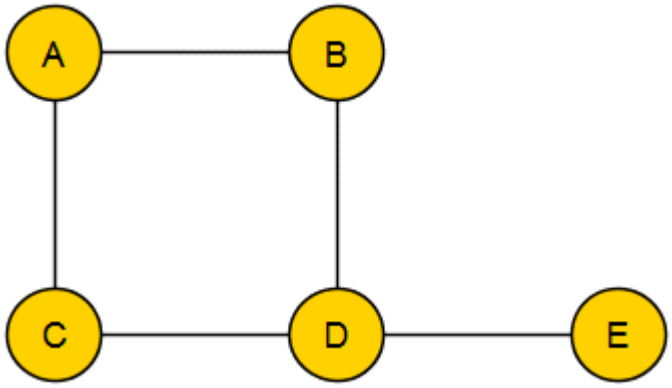
Поиск клик графа не применим как алгоритм реального времени из-за сложности.

Покрытия графа

Friday, May 18, 2018 12:34

Покрытие графа

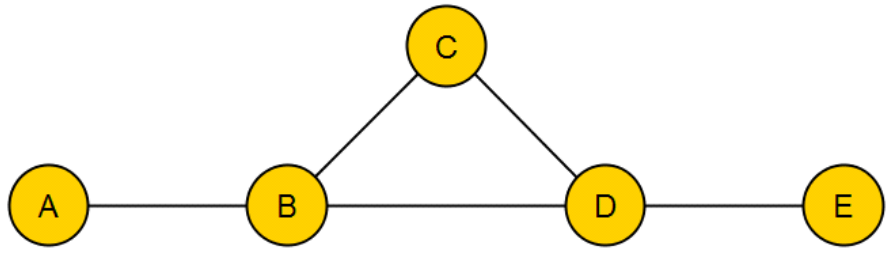
I. Максимальное вершинное независимое множество - максимальное подмножество всех вершин графа, никакая из которых не имеет ни с какой другой общих ребер.



Для данного графа — это множество $\{C, B, E\}$ и $\lambda(G) = 3$
Полиномиального алгоритма нахождения такого множества пока, и решается она перебором.

II. Минимальное вершинное покрытие - набор вершин, таких, что любое ребро инцидентно какой-нибудь из этих вершин
Для графа из предыдущего примера это $\{A, D\}$ и $\beta(G) = 2$
Теорема. $\alpha(\mathrm{G}) + \beta(\mathrm{G}) = \mathrm{n}$
α - дополнение β до G

III. Максимальное паросочетание - множество попарно несмежных ребер данного графа



$\alpha'(\mathrm{G}) = 2$, например $\{BC, DE\}$

IV. Реберное покрытие графа - минимальный набор ребер, такой, что любая вершина этого графа входит хотя бы в одно ребро данного множества.
Для предыдущего графа это
$\beta'(G) = 3$, например - $\{AB, BC, DE\}$

Теорема Галлаи. $\alpha'(G) + \beta'(G) = n$

Часть доказательства
Максимальное паросочетание покрывает $2\alpha'(G)$ вершин. Количество непокрытых вершин - $|o| = n - 2\alpha'(G)$
Пусть из каждой непокрытой вершины пущено одно ребро. Этих ребер тоже $|o|$ штук. Этого достаточно, чтобы покрыть все вершины
Набор $n - 2\alpha'(G) + \alpha'(G) = n - \alpha'(G) \geq \beta'(G)$
Осталось доказать в обратную сторону.

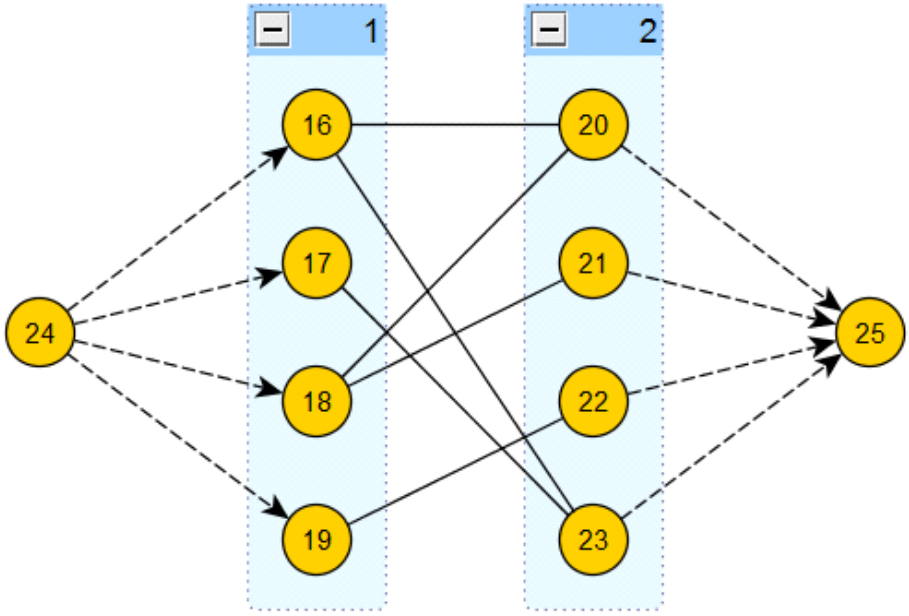
Двудольный граф - такой граф, вершины которого можно разделить на два множества, таких, что все ребра будут проходить между множествами.

Теорема Кёнига. Для двудольных графов размерность минимального вершинного покрытия совпадает с размерностью максимального паросочетания, т.е. $\alpha'(\mathrm{G}) = \beta(\mathrm{G})$.

Алгоритм поиска максимального паросочетания в двудольном графе

Для поиска максимального паросочетания в двудольном графе существуют хорошие алгоритмы. Это важно, т.к. к двудольным графам сводится множество задач - например 2SAT.

Алгоритм I
Задачу поиска максимального паросочетания можно свести к поиску максимального потока:



Сложность такого алгоритма - полиномиальная. Это гораздо проще, чем для задачи в общем виде.

Алгоритм II. Задача об устойчивых браках

Пара $\{A, b\}$ называется **неустойчивой**, если:

1. В паросочетании уже есть пары $\{A, a\}$ и $\{B, b\}$

2. A предпочитает b элементу a
3. b предпочитает A элементу B

Стоит задача найти полное устойчивое паросочетание между элементами двух множеств размера n, имеющими свои предпочтения (т.е. каждый элемент из множества α может отсортировать свои предпочтения из элементов множества β, и наоборот).
По такому определению строится двудольный граф, где доли - α, β

Описание алгоритма:

1. $\forall a \in \alpha$ делают предложение элементам β
2. $\forall b \in \beta$ отвечают на наилучшее предложение "может быть", отказывая остальные
3. $\forall a \in \alpha$, получившие отказ, обращаются к следующим элементам β из своих список предпочтений
4. Если $b \in \beta$ пришло предложение от $a \in \alpha$ лучше предыдущего, то b отвергает предыдущее предложение и отвечает a "может быть"
5. Если у α не исчерпался список предложений, повторить шаги 1-4.
6. $\forall b \in \beta$ отвечают согласием на существующие в данный момент "может быть"

Чтобы свести эту задачу к поиску максимального паросочетания, достаточно сказать, что элементы левой доли предпочитают инцидентные им элементы правой доли в лексикографическом порядке.

Применение алгоритмов

- Сведение из 2SAT

Алгоритмы на строках

9 апреля 2018 г. 13:50

Алгоритм Кнута-Морриса-Пратта

13 апреля 2018 г. 11:49

Алгоритм Кнута-Морриса-Пратта

Алфавит - множество символов Σ
Строка - последовательность символов из алфавита S
Подстрока - $S[i:j]$ — с i —го по j — й символы
Длина строки - число символов в ней
Σ^k — множество всех строк длины k из этого алфавита
$\Sigma^* = \bigcup_{k=0}^{\infty} \Sigma^k$ - множество всех строк
$\varepsilon = \Sigma^0$ — пустая строка
$\Sigma^+ = \Sigma^* \backslash \Sigma^0$
$x[y - x -$ **префикс** y, если $\exists \omega \in \Sigma^+ : y = x\omega$
$x]y - x -$ **суффикс** y, если $\exists \omega \in \Sigma^+ : y = \omega x$
Собственный суффикс - суффикс, не совпадающий со всей строкой
x - **бордер** y, если $\exists \omega, \gamma \in \Sigma^+ : x\omega = \gamma x = y$

$$\begin{cases} x[z \\ y[z \\ |x| < |y| \end{cases} \Rightarrow x[y$$

Постановка задачи

Дана строка T и образец P. Нужно найти все позиции, начиная с которых P входит в T.

Префикс-функция

Префикс-функция - $\pi(S, i): \Sigma^* \to \mathbb{N}$, где S $\in \Sigma^+$ - строка, i $\in [2, |S|]$ - длина префикса в S. Определяет длину наибольшего префикса подстроки $S[1:i]$, который одновременно является её собственным суффиксом (т.е. случай $\pi(S, i) = |S|$ не рассматривается).
Часто результат записывается в виде вектора из $\pi(S, i)$, где i пробегает по всем возможным значениям

Пример: строка abcabcd

```
0 - a
0 - ab
0 - abc
1 - abca
2 - abcab
3 - abcabc
0 - abcabcd
```

В таком случае $\pi(abcabcd) = [0,0,0,1,2,3,0]$

Наивный алгоритм

Можно вычислить префикс-функцию по определению, т.е. сравнивая префиксы и суффиксы для всех подстрок.
Здесь и далее обозначения псевдокода:
p - массив со значениями префикс-функции, соответственно $p[i]$ – префикс-функция от $i-$ й подстроки
$s-$ строка, $s[i]-i-$ й символ. Нумерация с 0.
На псевдокоде:

```
int[] prefixFunction(string s):
    int[] p = int[s.length]
    fill(p, 0)
    for i = 0 to s.length - 1
       for k = 0 to i - 1
          if s[0..k] == s[i - k..i]
             p[i] = k
    return p
```

На языке C++ это будет выглядеть так:

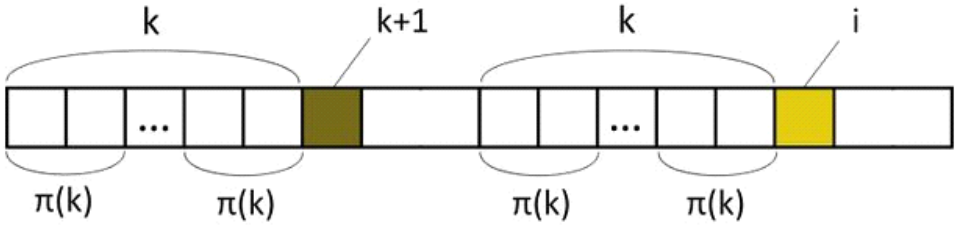
```
vector<int> prefix_function (string s) {
      int n = (int) s.length();
      vector<int> p (n);
      for (int i=0; i<n; ++i)
            for (int k=0; k<=i; ++k)
                  if (s.substr(0,k) == s.substr(i-k+1,k))
                        p[i] = k;
      return p;
}
```

Сложность такого алгоритма - итераций цикла $O(n^2)$, сравнение - $O(n)$, в итоге $O(n^3)$.

Эффективный алгоритм

Префикс-функция используется в алгоритме КМП, поэтому её эффективное вычисление важно для работы алгоритма. Для повышения эффективности используются следующие закономерности:
Во-первых, $p[i+1] \leq p[i]+1$. Это очевидно - если рассмотреть суффикс, заканчивающийся на позиции $i+1$ и убрать один символ, получится суффикс длины $p[i+1]-1$.
Во-вторых, нужно избавится от сравнений строк. Т.е. уже вычислено $p[i]$:

- Если $s[i+1] = s[p[i]+1]]$, то $p[i+1] = p[i]+1$.
- Если это неверно, нужно найти подстроку меньшей длины. Это тоже нужно сделать максимально эффективно - сразу перейти к бордеру наибольшей длины.
  Подберем такое k, что $k = p[i]-1$. Исходное $k - p[i-1]$. Если $s[k+1] \neq s[i]$, то следующее значение - $p[k]$. Если $k = 0$, то при $s[i] = s[1]\ p[i] \coloneqq 1$; иначе $p[i] = 0$.



Теперь на префикс-функция будет выглядеть следующим образом:

На псевдокоде:

```
int[] prefixFunction(string s):
  p[0] = 0
  for i = 1 to s.length - 1
     k = p[i - 1]
     while k > 0 and s[i] != s[k]
        k = p[k - 1]
     if s[i] == s[k]
        k++
     p[i] = k
  return p
```

На языке C++:

```
vector<int> prefix_function (string s) {
    int n = (int) s.length();
    vector<int> p (n);
    for (int i=1; i<n; ++i) {
        int j = p[i-1];
        while (j > 0 && s[i] != s[j])
            j = p[j-1];
        if (s[i] == s[j])  ++j;
        p[i] = j;
    }
    return p;
}
```

Теперь время работы алгоритма - $O(n)$.

Алгоритм Кнута-Морриса-Пратта

Дана строка T и образец P. Нужно найти все позиции, начиная с которых P входит в T. Для этого строится строка $S = P\#T$, где $\# \notin \Sigma$.

На этой строке вычисляется значение префикс-функции. Если для какого-то $i: p[i] = |P|$, то это - вхождение. Алгоритм выглядит следующим образом:

На псевдокоде:

```
int[] kmp(string P, string T):
  int pl = P.length
  int tl = T.length
  int[] answer
  int[] p = prefixFunction(P + "#" + T)
  int count = 0
  for i = 0 .. tl - 1
    if p[pl + i + 1] == pl
      answer[count++] = i - pl + 1
  return answer
```

На языке C++:

```
vector<int> kmp(string P, string T){
    int pl = P.length();
    int tl = T.length();
    vector<int> answer;
    vector<int> p = prefix_function(P + "#" + T);
    for (int i=0; i<tl; i++){
        if (p[pl+i+1] == pl)
            answer.push_back(i - pl + 1);
    }
    return answer;
}
```

Время работы алгоритма оценивается как $O(P + T)$.

Алгоритм Ахо-Корасик

13 апреля 2018 г. 11:50

Алгоритм Ахо-Корасик

Визуализация алгоритма:
https://github.com/SqrtMinusOne/AHO_CORASICK_ALGORITHM

Постановка задачи

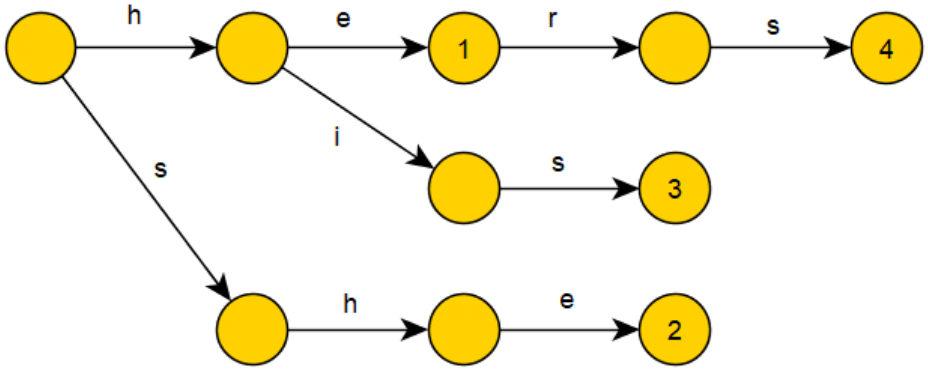
Есть множество шаблонов $\Phi = \{p_1, \dots, , p_n\}$ и один текст $|T| = n$.
Суммарная длина всех образцов - m.
Нужно найти, какие образцы из данного множества встречаются в этом тексте. Применение алгоритма Кнута-Морриса-Пратта неэффективно, т.к. алгоритм нужно будет запустить для каждого шаблона.

Задача о словаре

Пусть есть словарь. Задача - как можно более эффективно хранить в нём слова. Структура данных, подходящая для этой цели - **бор**.
Если после этого стоит задача найти в тексте все слова из словаря, то рассматриваемый алгоритм как раз эффективно решает эту задачу.

Построение бора

Бор (Trie) - дерево с корнем в некоторой вершине, каждое ребро которого подписано некоторой буквой.
Пусть дан набор строк - $\{he, she, his, hers\}$



У дерева каждая дуга имеет пометку в виде одного символа, и не может быть двух дуг из одной вершины, которые имеют одинаковую пометку.

Элемент бора на псевдокоде:

```
struct Node:
    Node son[k]                         // массив сыновей
    Node go[k]                          // массив переходов (запоминаем
переходы в ленивой рекурсии), используемый для вычисления суффиксных
ссылок
    Node parent                         // вершина родитель
    Node suffLink                       // суффиксная ссылка (вычисляем
в ленивой рекурсии)
```

```
    Node up                             // сжатая суффиксная ссылка
    char charToParent                   // символ, ведущий к родителю
    bool isLeaf                         // флаг, является ли вершина
терминалом
    vector<int> leafPatternNumber       // номера строк, за которые
отвечает терминал
```

Добавление слова в бор на псевдокоде:

```
fun addString(string s, int patternNumber):
    Node cur = root
    for i = 0 to s.length - 1
        char c = s[i]
        if cur.son[c] == 0
            cur.son[c] = Node
            /* здесь также нужно обнулить указатели на переходы и
сыновей */
            cur.son[c].suffLink = 0
            cur.son[c].up = 0
            cur.son[c].parent = cur
            cur.son[c].charToParent = c
            cur.son[c].isLeaf = false
        cur = cur.son[c]
    cur.isLeaf = true
    cur.leafPatternNumber.pushBack(patternNumber)
```

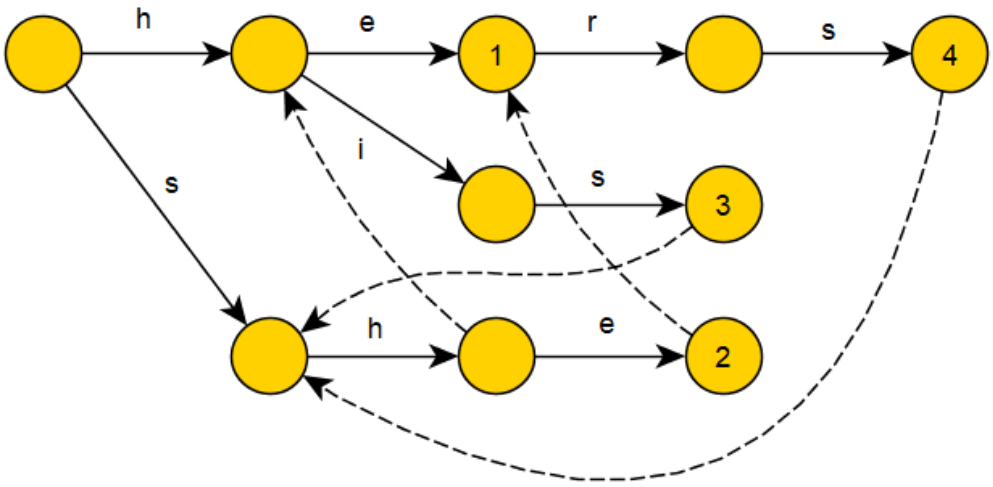
Пусть $[u]$ — слово, приводящие в вершину u в боре. Узлы бора можно понимать как состояния автомата, корень - как начальное состояние. Соответственно, узлы бора, в которых заканчиваются строки, становятся терминальными.

Переходы по бору

Заведем несколько функций для перехода по бору:
Во-первых, нужно определить функцию перехода "наверх":
$parent(u)$ — возвращает родителя вершины u

Суффиксные ссылки

$\pi(u) = \delta(\pi(parent(u)), c)$ — **суффиксная ссылка**, и существует переход из $parent(u)$ в u по символу c
Суффиксная ссылка для вершины u — вершина, в которой оканчивается наидлиннейший собственный суффикс строки, соответствующей вершине u. Суффикс может быть и нулевой длины - т.е. суффиксная ссылка может вести в корень.
Другими словами, $\pi(u) = v$, если $[v]$ — максимальный суффикс $[u]$, $[u] \neq [v]$.
Суффиксные ссылки для вышеуказанного примера:



Если ссылка не обозначена, значит, она ведёт в корень.

Вычисление суффиксной ссылки на псевдокоде

```
Node getSuffLink(Node v):
    if v.suffLink == null                          // если суффиксная
ссылка ещё не вычислена
       if v == root or v.parent == root
            v.suffLink = root
       else
            v.suffLink = getLink(getSuffLink(v.parent),
v.charToParent)
    return v.suffLink
```

В данном случае и далее используется **ленивая рекурсия** - ссылки не вычисляются, если не нужны, но вычисленные ссылки запоминаются. Это позволяет экономить ресурсы.

Переход по бору

$$\delta(u,c) = \begin{cases} v & \text{если } v \text{ — сын символа } c \\ root & \text{если } u \text{ — корень и } c \text{ — не сын } u \\ \delta(\pi(u),c) & \text{иначе} \end{cases}$$

Переход осуществляется по текущей вершине u и символу c.

Функция перехода на псевдокоде

```
Node getLink(Node v, char c):
   if v.go[c] == null  // если переход по символу c ещё не вычислен
        if v.son[c]
            v.go[c] = v.son[c]
        else if v == root
            v.go[c] = root
        else
            v.go[c] = getLink(getSuffLink(v), c)
    return v.go[c]
```

Сжатые суффиксные ссылки

При построении автомата может возникнуть ситуация, что ветвление

есть не на каждом символе. Тогда можно использовать **сжатые суффиксные ссылки**:

$$up(u) = \begin{cases} \pi(u) & \text{если } u - \text{терминальный} \\ \emptyset & \text{если } u \text{ — корень} \\ up(\pi(u)) & \text{иначе} \end{cases}$$

$up(u)$ — ближайшее допускающее состояние (терминал) перехода по суффиксным ссылкам.

Вычисление сжатых суффиксных ссылок на псевдокоде:

```
Node getUp(Node v):
    if v.up == null // если сжатая суффиксная ссылка ещё не вычислена
        if getSuffLink(v).isLeaf
            v.up = getSuffLink(v)
        else if getSuffLink(v) == root
            v.up = root
        else
            v.up = getUp(getSuffLink(v))
    return v.up
```

В результате вышеописанных действий построен конечный детерминированный автомат.

Использование автомата

В общих чертах, получившийся автомат нужно использовать следующим образом: по очереди просматривать символы текста, для каждого символа c осуществляя переход по $\delta(u, c)$, где δ — текущее состояние. Оказавшись в новом состоянии, отметить по сжатым суффиксным ссылкам строки, которые встретились, и, если требуется, позицию.

Использование автомата на псевдокоде:

```
fun processText(string t):
    Node cur = root
    for i = 0 to t.length - 1
        char c = t[i] - 'a'
        cur = getLink(cur, c)
        /* В этом месте кода должен выполняться переход по сжатой
суффиксной ссылке getUp(cur). Для вершины,
           обнаруженной по ней тоже ставим, что она найдена, затем
повторяем для её сжатой суффиксной ссылки
           и так до корня. */
```

Суффиксные деревья

Friday, April 20, 2018 11:48

Можно почитать в Гасфилде

Суффиксные деревья

Суффиксный бор

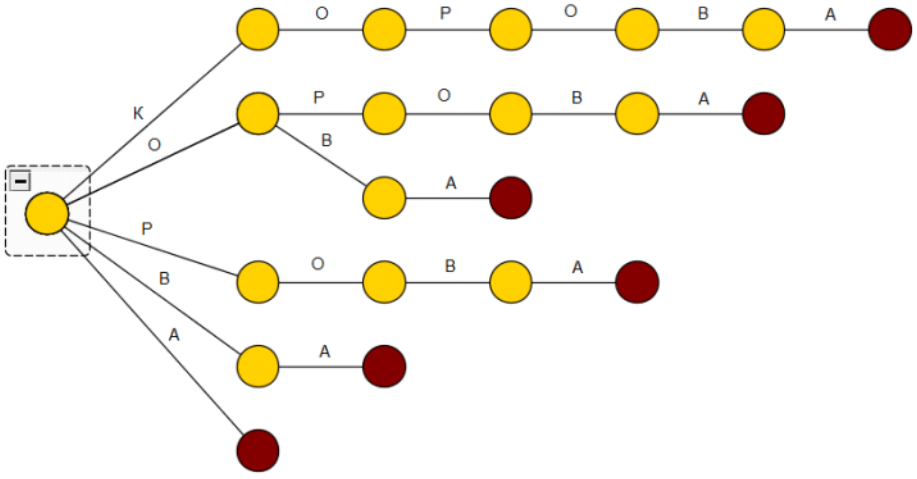
Суффиксный бор - бор, содержащий все суффиксы данной строки.

Пример 1

КОРОВА

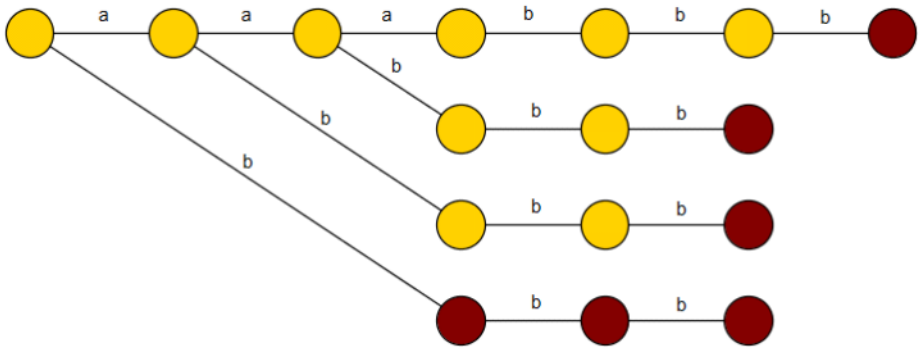
Суффиксы:

КОРОВА
ОРОВА
РОВА
ОВА
ВА
А



Пример 2

aaabbb



Чтобы построить суффиксный бор для строки, достаточно добавить все суффиксы в бор.

Такой суффиксный бор можно использовать для поиска подстроки в строке за линейное время. Чтобы построить его наивным способом,

требуется $O(n^2)$ операций.

Суффиксный бор неэффективен по памяти - если хранить в каждом элементе массив перехода размера $|\Sigma|$, где Σ — алфавит, то потребуется $O(n^2 \cdot |\Sigma|)$ памяти.
Если учесть, что число ветвлений не превышает число суффиксов, то число вершин, из которых идёт более одного перехода - $O(n)$. Поэтому в неветвящихся вершинах можно хранить только символ перехода и ребёнка, чтобы занять $O(n^2 + n|\Sigma|)$ памяти.

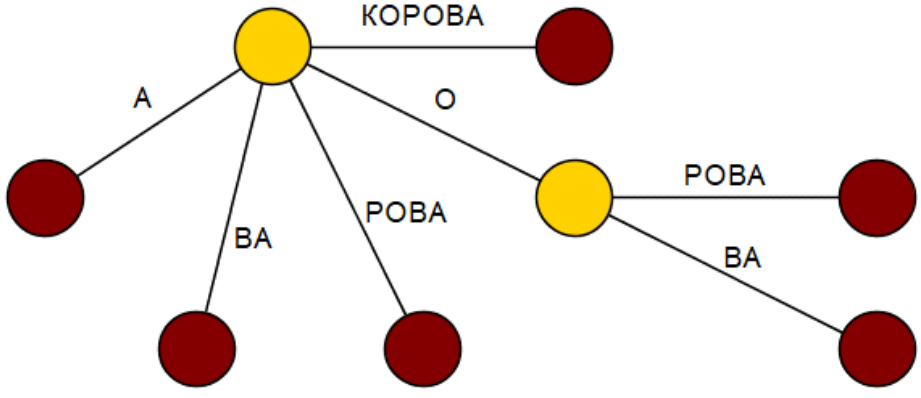
Суффиксное дерево

(Сжатое) суффиксное дерево - это оптимизация суффиксного бора, требующая линейное количество памяти. Свойства суффиксного дерева для строки $|s| = n$

- Каждая вершина имеет не меньше двух детей
- Каждое ребро помечено непустой подстрокой строки s
  Сжатие дуговых веток - вместо того, чтобы хранить всю строчку, хранятся номера начала и конца. Это сокращает время обращения к данным.
- Никакие два ребра, выходящие из одной вершины, не могут иметь пометок, начинающихся с одного и того же символа
- Дерево должно содержать все суффиксы s, причем каждый суффикс заканчивается в листе и нигде более.

Не для каждой строки можно построить суффиксное дерево. Если в строке есть суффикс, который является префиксом другого суффикса, то суффикс заканчивается не в листе и суффиксное дерево построить невозможно. Чтобы это обойти, нужно добавить в конец строки символ, которого нет нигде в строке. Так, в слове "КОРОВА" этот символ - А.

Пример



Индукцией по n можно доказать, что число листьев дерева - n, а число внутренних вершин - меньше n. Представим дерево как двумерный массив размера $|V| \times |E|$, где $|V|$ — число вершин в дереве, $|\Sigma|$ —

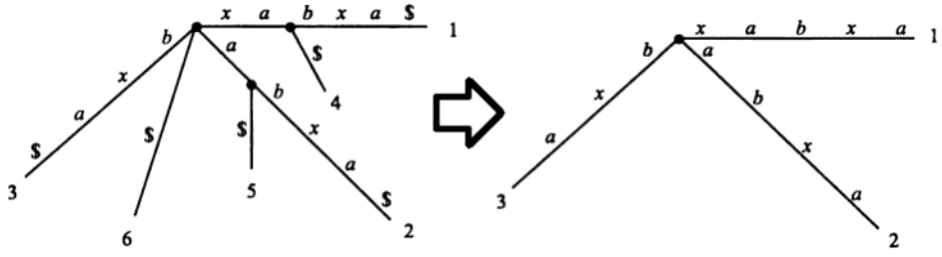
мощность алфавита. Для любого суффиксного дерева верна предыдущая лемма (у каждой вершины, по определению, не менее двух детей), значит, $|V| = O(2n)$. Каждая $a[i][j]$ ячейка содержит информацию о том, в какую вершину ведет ребро из i-ой вершины по j-ому символу и индексы l, r начала и конца подстроки, записанной на данном переходе. Итак, дерево занимает $O(n|\Sigma|)$ памяти.

Поиск наибольшей подстроки в S1 и S2

Кнут считал, что эту задачу невозможно решить за линейное время. Это оказалось неверным.
Сначала нужно добавить в конец строк два символа - например # и $
КОРА$ ОРАТЬ#
После чего построить суффиксное дерево для строки.

Эффективное построение суффиксного дерева (Укконена)

Сложность построения дерева описанным способом - $1 + 2 + \cdots + \mathrm{m} = \mathrm{O}(\mathrm{m}^2)$. Это неэффективно.
Неявное суффиксное дерево строки - дерево, полученное из суффиксного дерева удалением всех вхождений терминального символа, удалением после этого дуг без меток и удаление вершин, имеющих меньше двух детей.
Пример:
Дерево было построено для строки xabxa$, где $ был добавлен как терминальный символ



Хотя неявное суффиксное дерево может иметь листья не для всех суффиксов, в нем закодированы все суффиксы S.

Алгоритм Укконена строит неявное суффиксное дерево T_i для каждого префикса $s[1 \dots i]$ строки S, начиная с 1 и заканчивая T_m, где $m = |S|$.
Алгоритм делится на m **фаз**. В фазе $i + 1$ дерево T_{i+1} строится из T_i.
Каждая фаза $i + 1$ делится на $i + 1$ **продолжений**. В продолжении j фазы $i + 1$ алгоритм сначала находит конец пути из корня, помеченного подстрокой $S[j..i]$. Затем он продолжает строку, добавляя к её концу символ $S(i + 1)$, если $S(i + 1)$ ещё не существует.
Таким образом, в фазе $i + 1$ в дерево помещается строка $S(1 \dots i + 1)$, а за ней строки $S[2 \dots i + 1], S[3 \dots i + 1]$ в продолжениях 1,2,3, … .
Продолжение $i + 1$ фазы $i + 1$ продолжает **пустой** суффикс строки $S[1 \dots i]$, т.е. включает в дерево строку из одного символа $S(i + 1)$ (опять же, если

её там не было)

Алгоритм на псевдокоде

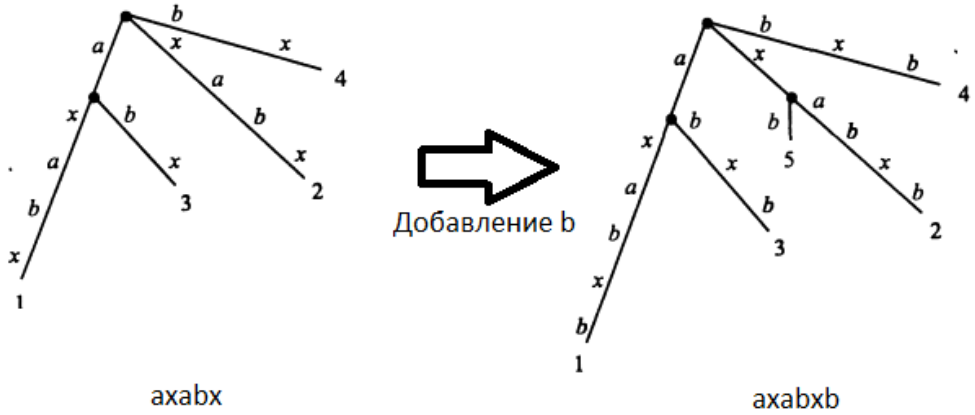
```
Построить дерево T1
for (i = 1...m-1) //Фаза i+1
    for (j = 1...i+1) //Продолжение j
        Найти в текущем дереве конец пути из корня с меткой S[j...i]
        Если нужно, продолжить путь, добавить символ S(ш+1), обеспечив
            Появление строки S[j...i] в дереве
```

Продолжение суффикса

Пусть $S[j \dots i] = \beta -$ суффикс $S[1 \dots i]$. В продолжении j, когда алгоритм находит конец β в текущем дереве, он продолжает β, чтобы обеспечить присутствие суффикса $\beta S(i+1)$ в корне. Он действует по одному из следущих правил:

1. Если путь β кончает в листе, то это значит, что путь от корня с меткой β доходит до конца некоторой дуги, входящей в лист. В этом случае нужно добавить к концу метки листовой дуги символ $S(i+1)$
2. Ни один путь из конца строки β не начинается символов $S(i+1)$, но по крайней мере один начинающийся путь оттуда 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.
3. Если некоторый путь из конца строки β начинается символом $S(i+1)$, то эта строка уже есть в дереве и ничего не надо делать

Пример



Фаза 6. Суффиксы:

- axabxb - в конец добавлен b
- xabxb - в конец добавлен b
- abxb - в конец добавлен b
- bxb - в конец добавлен b
- xb - уже есть
- b - уже есть

Сложность алгоритма в таком виде:

- Поиск конца суффикса $\beta - O(|\beta|)$
- Продолжение j в фазе $i+1$ - $O(i-j+1)$

- Построение $T_i - O(i^2)$

Итого $O(m^3)$.

Улучшение алгоритма

Суффиксные связи

Пусть $x\alpha -$ произвольная строка, где $x -$ её первый символ, а $\alpha -$ оставшаяся подстрока. Если для внутренней вершины v с её путевой меткой $x\alpha$ существует другая вершина $s(v)$ с путевой меткой α, то указатель из v в $s(v)$ называют **суффиксной связью.** Обозначается так: $(v,\ s(v))$

Если α пуста, то суффиксная связь идёт в корневую вершину.

Можно доказать, что из каждой внутренней вершины неявного суффиксного дерева выходит суффиксная связь.

Переходы по суффиксным связям

В фазе $i+1$ алгоритм находит место суффикса $S[j \dots i]$ строки $S[1 \dots i]$ в продолжении j и $j = 1 \dots i+1$. При наивном подходе строка $S[j \dots i]$ просматривается вдоль всего пути от корня в текущем дереве. Суффиксные связи могут сократить это движение и каждое его продолжение.

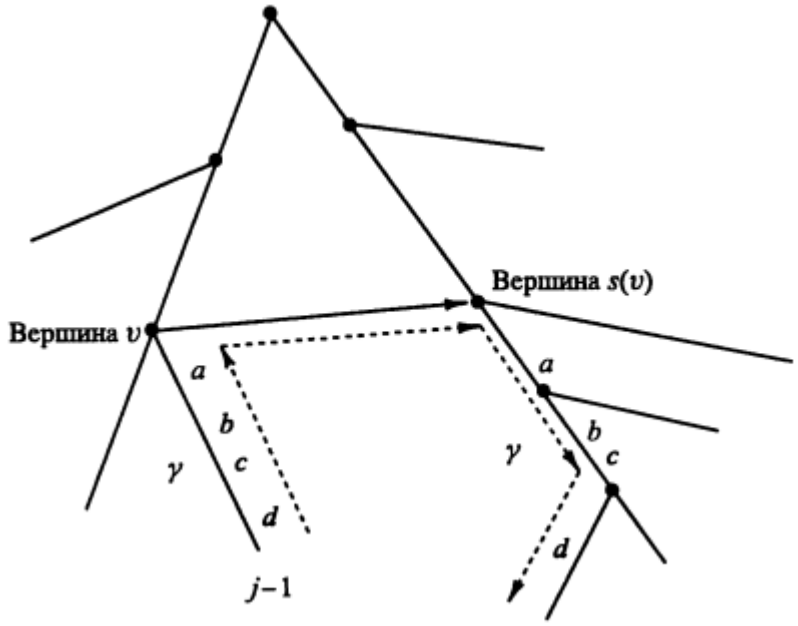
Рассмотрим первые два продолжения $(j = 1,2)$ фазы $i+1$.

Первое продолжение - особый случай: конец полной строки $S[1 \dots i]$ должен быть в листе дерева T_i, т.к. $S[1 \dots i] -$ самая длинная строка в нем. Значит, при конструировании деревьев мы можем всё время поддерживать указатель на лист, соответствующей текущей полной строке и достраивать продолжение по правилу 1, чтобы строить его за константное время.

Представим строку $S[1 \dots i]$ в виде $x\alpha$ и пусть $(v, l) -$ дуга дерева, входящая в лист l. Следующее действие алгоритма - нахождение конца $S[2 \dots i]$. В этом месте v является либо корнем, либо внутренней вершиной T_i. Если это корень, то для нахождения конца алгоритм спускается по дереву пути, помеченному α. А если это внутренняя вершина, то v имеет суффиксную связь с $s(v)$. Поскольку $s(v)$ имеет префикс строки α, то конец строки α находится в поддереве $s(v)$. Значит, при поиске конца α не нужно проходить от корня, достаточно пройти путь от вершины $s(v)$. Таким образом, для второго продолжения нужно пройти вверх от листа l до вершины v, далее по суффиксной связи из v в $s(v)$, после чего от $s(v) -$ вниз по пути и в конце пути изменить дерево по правилам продолжения суффикса.

В общем случае идея такая же: при продолжении любой строки $S[j \dots i]$ до $S[j \dots i+1]$ мы поднимаемся вверх, попадая либо в корень, либо в вершину v, из которой выходит суффиксная связь. Если $v -$ не корень, то

пусть $\gamma -$ дуговая метка.



Алгоритм отдельного продолжения

Продолжение $j \geq 2$ фазы $i + 1$

1. Найти в конце строки $S[j-1 \dots i]$ или выше его первую вершину v, которая либо имеет исходящую суффиксную связь, либо является корнем. Для этого нужно пройти вверх до конца $S[j-1 \dots i]$. Пусть $\gamma -$ строка между v и концом $S[j-1 \dots i]$
2. Если v не корень, пройти суффиксную связь из v в вершину $s(v)$ и спустится из $s(v)$ по пути γ.
   Если $v -$ корень, пройти, как в наивном алгоритме
3. С помощью правил продолжения, обеспечить вхождение строки $S[j \dots i]S(i+1)$ в дерево
4. Если в продолжении $j-1$ была создана новая внутренняя вершина w, то строка α должна кончаться в вершине $s(w)$. в конце суффиксной связи из w. Нужно создать суффиксную связь $\big(w, s(w)\big)$

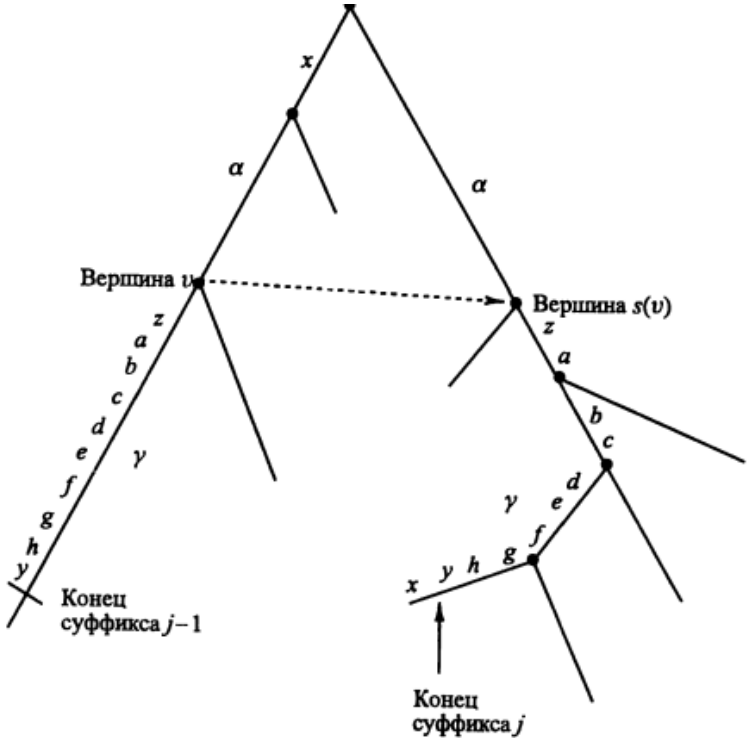
Это ещё не улучшает скорость работы алгоритма.

<u>Скачок по счётчику</u>

На шаге 2 алгоритм идёт в вниз из вершины $s(v)$ по пути с меткой γ. Это прохождение занимает время $|\gamma|$ и всё время, затрачиваемое на это - $O(m)$.

Пусть $g = |\gamma|$. Пусть $g' -$ число символов в дуге, исходящей по нужному символу. Если $g' < g$, то можно просто перепрыгнуть в конец. Затем $g := g - g'; h = g' + 1$ и просматриваются выходящие дуги, чтобы найти среди них правильно продолжение - с начальным символом h.

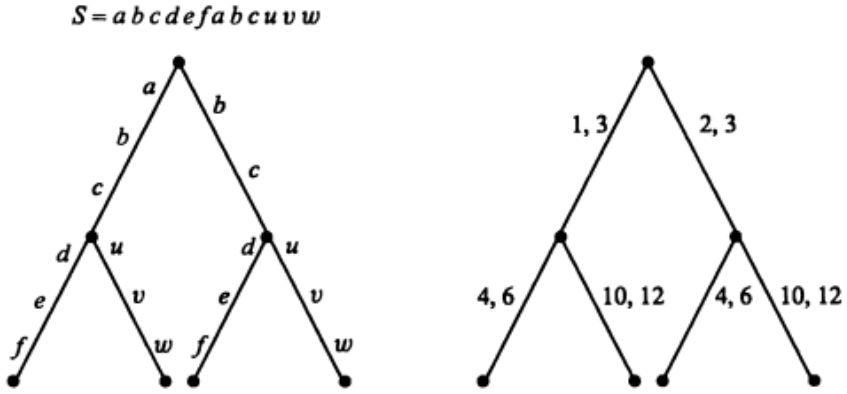
При идентификации следующей дуги пути, он сравнивает текущее значение g с числом символов на этой дуге g'. Если $g \geq g'$, алгоритм перескакивает к концу дуги.



Это обеспечивает переход за время, пропорциональное числу вершин, а не символов.

Можно доказать, что вершинная глубина v не более чем на единицу превосходит $s(v)$. Теперь суффиксные связи обеспечивают время работы $O(m^2)$.

Сжатие дуговых ссылок позволяет хранить дерево в памяти за $O(m)$.



Ещё оптимизации

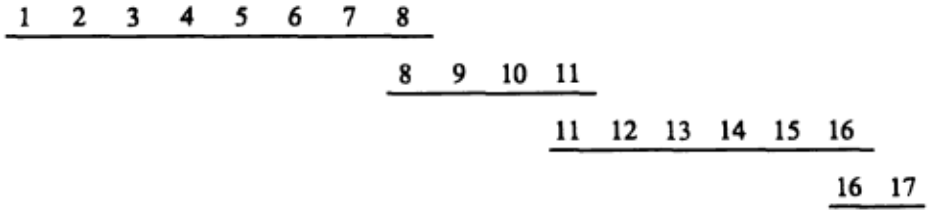
1. После первого использования правила 3 заканчивать работу с продолжением
2. Если в какой-то момент будет создан лист с меткой j, то он останется листом во всех последующих итерациях продолжения. Из такого наблюдения следует следующая оптимизация:
   В фазе $i+1$ запоминать $e = i+1-$ символ текущего конца. В результате, когда алгоритм узнает, что в продолжениях с 1 до j_i будет применено правило 1, в первых j_i продолжениях не потребуется никакой дополнительной работы.

Итог: Алгоритм одной фазы

1. Увеличить индекс e до $i + 1$
2. Явно вычислить последовательные продолжения от $j_i + 1$ до достижения первого продолжения j^*, где применяется правило 3, или до конца этой фазы.
3. Приготовляясь к следующей фазе, положить j_{i+1} равным $j^* - 1$

Шаг 3 устанавливает корректное значение j_{i+1}, т.к. исходная последовательность продолжений, в которых используются правила 1 и 2, должна заканчиваться в точке, где применяется правило 3.

Теперь время линейное - $O(m)$



Получение суффиксного дерева из неявного суффиксного дерева
Для этого достаточно добавить к строке терминальный символ и снова запустить алгоритм Укконена.

Алгоритм Рабина-Карпа

Friday, April 27, 2018 11:50

Алгоритм Рабина-Карпа

Постановка задачи

Есть текст длиной n, есть шаблон длиной m. q — число вхождений, p — простое число. Нужно найти все вхождения шаблона в текст.
Наивный алгоритм сравнивает посимвольно символы начиная с первого. Если закончился шаблон или получено несовпадение, алгоритм сдвигается на 1.

Посимвольное сравнение не очень эффективно. Строке нужно сопоставить число, которое бы характеризовало её уникальным образом - например, с помощью хэш-функции.

В таком случае, нужно посчитать хэш-функцию для шаблона и для первых m символов текста. Если хэш-сумма не совпадает, то строки точно не совпадают. Если же совпадают, тогда нужно выполнить посимвольную проверку. Вычисление хэш-функции для шаблона - $O(m)$. Проблема в том, что в худшем случае данный алгоритм работает даже хуже, чем наивный алгоритм.

Хэширование

Допустим, стоит задача написать телефонную книжку. На вход поступает 10-значный номер, нужно сказать, кому принадлежит номер.

1. Если номер преобразуется в индекс, то задачу можно решить очень быстро, но длина массива в таком случае - 10^{10}.
2. Можно хранить только те номера, которые нужны - например, в списке. В этом случае поиск по книжке (и другие операции тоже) будет равен количеству номеров. Это эффективно по памяти, но не по производительности.

Выделяем массив длины m - пусть будет не более 100 номеров. Нужно взять хэш-функцию, которая сопоставит номеру число от 0 до 100. Пример - деление по модулю. Проблема в том, что для очень большого количества объектов произойдут **коллизии** - совпадение хэш-функций для разных объектов. Разрешить коллизии можно разными способами:

1. Сделать массив из списков. Каждый элемент будет содержать объекты, содержащие одинаковые хэш-суммы.
   В худшем случае сложность - максимальная длина списка.
2. Переход в первую незанятую ячейку
3. Открытая адресация - под одинаковые хэш-суммы может быть выделено несколько позиций

$\varphi: X \to \mathbb{Z}_m$ - хэш-функция
Таким образом, к хэш-функции предъявляются следующие требования:

- Простота расчёта
- Минимальное количество коллизий

  $\forall x, y \in X, x \neq y: P\big(\varphi(x) = \varphi(y)\big) \leq \frac{1}{m}$
- Детерминированность - одинаковые значения для одного объекта

В хэш-функции обычно используется элемент рандомизации - база для хэша берется случайной.

Полиномиальный хеш для строк
При хэшировании строки необходимо учитывать не только разные символы, но и разные позиции символов.

$$\varphi(0 \dots n-1) = \sum_{i=0}^{n-1} p^i s[i] \; mod \; r$$

$p \in [1 \dots r-1]$ – простое число
Модуль необходим, чтобы решить проблему переполнения.
Для работы алгоритма нужно считать хеш подстроки. Далее операции в кольце вычетов по модулю r.

$$\varphi(0 \dots j) = \sum_{i=0}^{j} p^i s[i] =$$
$$= s[0] + ps[1] + \cdots + p^{i-1}s[i-1] + p^i s[i] + \cdots p^{j-1}s[j-1] + p^j s[j] =$$
$$= \big(s[0] + ps[1] + \cdots + p^{i-1}s[i-1]\big) + \big(p^i s[i] + \cdots p^{j-1}s[j-1] + p^j s[j]\big)$$
$$=$$
$$= \big(s[0] + ps[1] + \cdots + p^{i-1}s[i-1]\big)$$
$$+ p^i\big(s[i] + \cdots + p^{(j-i-1)}s[j-1] + p^{j-i}s[j]\big) =$$
$$\varphi(0 \dots i-1) + p^i \varphi(i \dots j)$$

Отсюда

$$\varphi(i \dots j) = \left(\frac{1}{p^i}\right)\big(\varphi(0 \dots j) - \varphi(0 \dots i-1)\big)$$

Вероятность коллизии такого хеша:

$$P\big(\varphi(s_1) = \varphi(s_2)\big) = \frac{\max(|S_1|, |S_2|)}{P} = \frac{L}{P}$$

Если $p \gg mL$, то будет $O\left(\frac{1}{m}\right)$.

Вычислять многочлен эффективнее по схеме Горнера, чем с помощью прямого умножения. Время вычисления будет пропорционально длине строки.

Алгоритм для быстрого вычисления хешей

Время работы в таком случае всё равно будет пропорционально n^2, и выигрыша не будет. Чтобы всё же обеспечить выигрыш во времени, нужно найти зависимость между соседними хэшами. Если хэш считается как сумма, это просто: вычесть слагаемое для первого символа, домножить на x и прибавить слагаемое для последнего.
$\varphi(i+1 \dots i+m-1) = \varphi(i \dots i+m-1) - p^{m-1}s[i]$
$\varphi(i+1 \dots i+m) = p \cdot \varphi(i+1 \dots i+m-1) + s[i+m]$
Результат:
$\varphi(i+1 \dots i+m) = p \cdot \varphi(i \dots i+m-1) - p^{m}s[i] + s[i+m]$

Алгоритм Рабина-Карпа
Пусть текст - s, шаблон - a. $n = |s|, m = |a|$. Алгоритм начинается с подсчёта $\varphi(s[0 \dots m-1])$ и $\varphi(a[0 \dots m-1])$.
Для $i \in [0 \dots n-m]$ вычисляется $\varphi(s[i \dots i+m-1])$. Если он оказывается равным хешу образца, то нужно явно сравнить строки.