# Обход графа в глубину

# Depth-First Search (DFS)

# Обход графа в глубину (DFS)

*Напоминание:* задача обхода графа заключается в обходе (посещении) вершин и ребер графа в некотором порядке с учетом структуры графа.

*Идея DFS:* возьмем стартовую вершину $s$ и будем двигаться вглубь в произвольном направлении.

# Обход графа в глубину (DFS)

*Идея DFS:* возьмем стартовую вершину $s$ и будем двигаться вглубь в произвольном направлении.

Этот процесс может прерваться по двум причинам:

- Из очередной вершины нет ребер
- Все ребра ведут в уже просмотренные вершины

# Обход графа в глубину (DFS)

Алгоритм DFS:

1. Берем вершину, проходим из нее по ребру в произвольную еще не посещенную вершину.
2. Если из очередной вершины некуда идти, то возвращаемся в предыдущую и продолжаем обход.
3. Если вернулись в стартовую вершину и из нее некуда идти, то заканчиваем.

# Обход графа в глубину (DFS)

В процессе работы алгоритма для каждой вершины будем дополнительно хранить следующие величины:

- **Цвет:**
    - *Белый* - вершины еще не посещена.
    - *Серый* - из вершины осуществляется обход в глубину.
    - *Черный* - вершина была обнаружена и из нее нет доступных путей.
- **time_in:** момент первого обнаружения вершины
- **time_out:** момент завершения обхода из вершины

# Обход графа в глубину (DFS)

```
color = [WHITE, WHITE, ..., WHITE]
time_in = [inf, inf, ..., inf]
time_out = [inf, inf, ..., inf]
time = 0

def DfsVisit(G, v):  # G - граф, v - посещаемая вершина
  color[v] = GRAY
  time_in[v] = ++time

  for u in G.neighbors(v):
    if color[u] == WHITE:  # если вершина не посещена
      DfsVisit(G, u)

  color[v] = BLACK
  time_out[v] = ++time
```

Время работы - ?

# Обход графа в глубину (DFS)

```
color = [WHITE, WHITE, ..., WHITE]
time_in = [inf, inf, ..., inf]
time_out = [inf, inf, ...m inf]
time = 0

def DfsVisit(G, v):  # G - граф, v - посещаемая вершина
  color[v] = GRAY
  time_in[v] = ++time

  for u in G.neighbors(v):
    if color[u] == WHITE:  # если вершина не посещена
      DfsVisit(G, u)

  color[v] = BLACK
  time_out[v] = ++time
```

Время работы - $O(V+E)$ при использовании списков смежности, $O(V^2)$ для матрицы смежности

# Обход графа в глубину (DFS)

Теперь цветам можно придать иной смысл:

- *Белая вершина* - вершина, из которой еще не был запущен *DfsVisit*.
- *Серая вершина* - вершина, вызов *DfsVisit* от которой еще не завершился.
- *Черная вершина* - вершина, вызов *DfsVisit* от которой был завершен.

Заметим, что серые вершины всегда образуют путь (стек вызовов DfsVisit).

# Обход графа в глубину (DFS)

Если требуется посетить все вершины, а не только достижимые из данной, то дополнительно пишется следующий цикл:

```
def DFS(G):
  color = [WHITE, WHITE, ..., WHITE]
  time_in = [inf, inf, ..., inf]
  time_out = [inf, inf, ..., inf]
  time = 0

  for v in G.V:
    if color[v] == WHITE:
      DfsVisit(G, v)
```

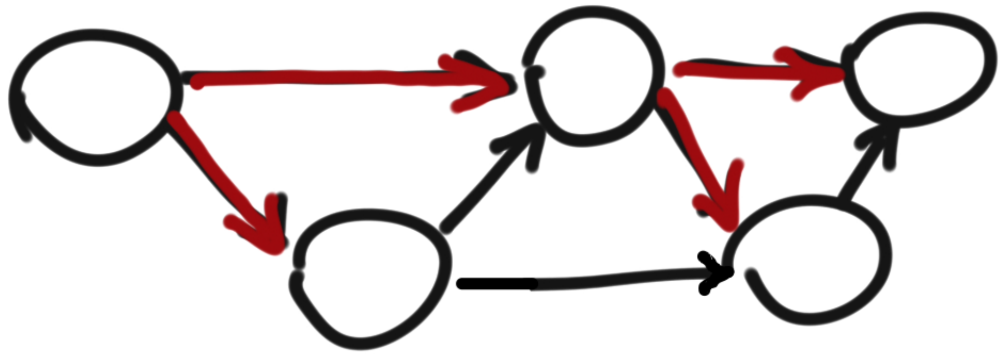
# Классификация ребер при обходе в глубину

# Дерево обхода в глубину

*Деревом в неориентированном графе* называется связный граф без циклов.

*Деревом в ориентированном графе* называется граф без циклов, в котором у всех вершин полустепень захода равна 1, кроме одной, у которой полустепень захода равна 0 (корень дерева).

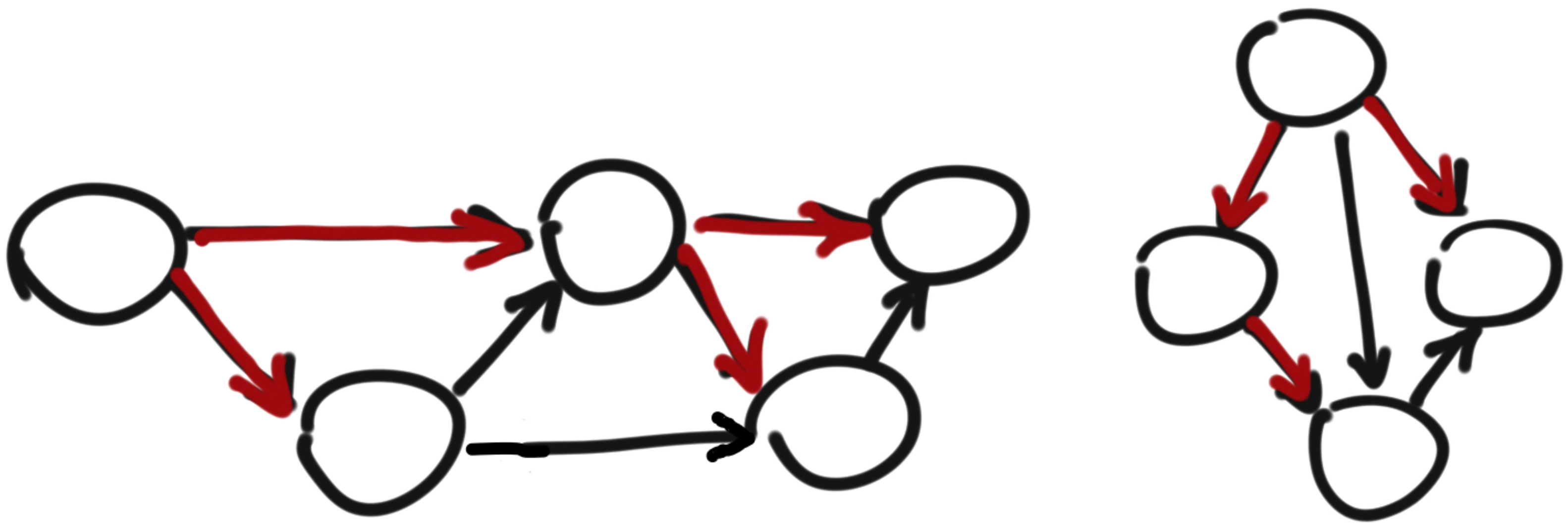
Заметим, что если оставить только посещенные ребра и ориентировать их в направлении обхода, то **DfsVisit образует дерево обхода**.



# Дерево обхода в глубину

Заметим, что если оставить только посещенные ребра и ориентировать их в направлении обхода, то **DfsVisit образует дерево обхода**.

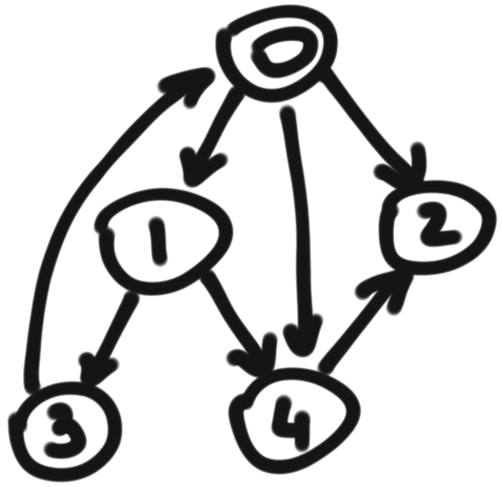
Деревья, которые получаются в результате обхода всего графа (процедура *DFS*), называются **лесом обхода в глубину**.



# Классификация ребер

После работы алгоритма *DFS* каждое ребро в графе можно отнести к одному из четырех классов:

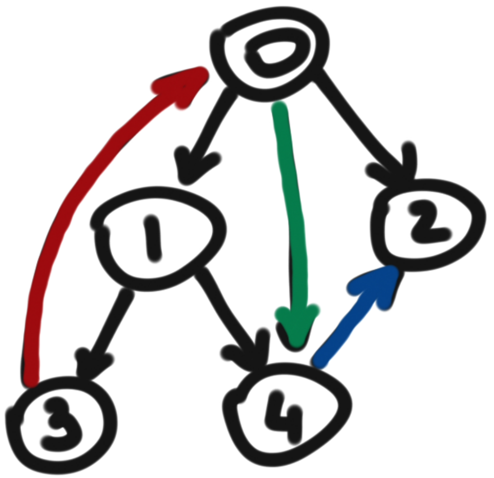
1. *Ребро дерева* - ребро, которое принадлежит какому-то дереву обхода.
2. *Обратное ребро* - ребро, ведущее из потомка в предка в некотором дереве.
3. *Прямое ребро* - ребро, ведущее из предка в потомка, не являющегося сыном.
4. *Перекрестное ребро* - ребро, соединяющее вершины не связанные отношением "предок-потомок" (все остальные ребра).



# Классификация ребер

После работы алгоритма *DFS* каждое ребро в графе можно отнести к одному из четырех классов:

1. *Ребро дерева* - ребро, которое принадлежит какому-то дереву обхода.
2. *Обратное ребро* - ребро, ведущее из потомка в предка в некотором дереве.
3. *Прямое ребро* - ребро, ведущее из предка в потомка, не являющегося сыном.
4. *Перекрестное ребро* - ребро, соединяющее вершины не связанные отношением "предок-потомок" (все остальные ребра).



# Классификация ребер

Как определить тип ребра во время работы *DFS*?

1. *Ребро дерева*:
2. *Обратное ребро*:
3. *Прямое ребро*:
4. *Перекрестное ребро*:

# Классификация ребер

Как определить тип ребра во время работы *DFS*?

1. *Ребро дерева*: ребро ведущее в белую вершину.
2. *Обратное ребро*: ребро ведущее в серую вершину.
3. *Прямое ребро*: ребро $vu$ ведущее в черную вершину, у которого *time_in[v]* < *time_in[u]*
4. *Перекрестное ребро*: ребро $vu$ ведущее в черную вершину, у которого *time_in[v]* > *time_in[u]*

**Упражнение.** *Доказать, что в неориентированных графах есть только древесные и обратные ребра.*

# Корректность

# Корректность

Докажем несколько утверждений о DFS

**Утверждение.** *Ни в один момент времени не может быть ребра из черной вершины в белую*.

# Корректность

Докажем несколько утверждений о DFS

**Утверждение.** *Ни в один момент времени не может быть ребра из черной вершины в белую*.

*Доказательство.* Пусть ребро $vu$ ведет из черной вершины ($v$) в белую ($u$). Рассмотрим момент, когда $v$ стала черной. Этому предшествовал цикл по соседям вершины $v$. В этом цикле встречалась белая вершина $u$. Значит она была пропущена этим циклом. Противоречие с построенным алгоритмом. ■

**Упражнение.** *Покажите, что все остальные варианты возможны.*

# Корректность

**Теорема (лемма о белых путях).**
*Вершина* $u$ *будет посещена в процессе вызова* $DfsVisit(G, v) \Leftrightarrow$ *в момент вызова* $DfsVisit(G, v)$ *существует путь из* $v$ *в* $u$*, состоящий только из белых вершин.*

# Корректность

**Теорема (лемма о белых путях).**
*Вершина $u$ будет посещена в процессе вызова $DfsVisit(G, v)$ $\Leftrightarrow$ в момент вызова $DfsVisit(G, v)$ существует путь из $v$ в $u$, состоящий только из белых вершин.*

*Доказательство.*
$\Rightarrow$ Тривиально. Посещаются только белые вершины, а значит искомый путь - последовательность вызовов приведшая в $u$.

$\Leftarrow$ Рассмотрим какой-то белый путь из $v$ в $u$: $v, v_1, v_2, ..., u$.
В момент завершения $DfsVisit(G, v)$ вершина $v$ - черная, а на рассматриваемом пути нет серых вершин (все порожденные рекурсивные вызовы завершились). По **утверждению 1** на этом пути нет черно-белых переходов. Значит все вершины на этом пути стали черными, то есть были посещены. $\blacksquare$

# Применения DFS

# Компоненты связности и достижимые вершины

- Из **леммы о белых путях** следует, что при запуске $DfsVisit(G, v)$, когда все вершины изначально белые, обход посетит вершины достижимые из $v$ и только их

*Компонентой связности неориентированного графа* называется максимальный по включению связный подграф.

- Запуск $DfsVisit(G, v)$ на "белом" графе находит компоненту связности, которой принадлежит вершина $v$.

# Проверка на ацикличность

**Теорема (критерий ацикличности).**

*В графе $G$ есть цикл $\Leftrightarrow$ при обходе в глубину было найдена серая вершина.*

# Проверка на ацикличность

**Теорема (критерий ацикличности).**
*В графе $G$ есть цикл $\Leftrightarrow$ при обходе в глубину было найдена серая вершина.*

*Доказательство.*
$\Leftarrow$ Серые вершины - индикатор обратных ребер. Обратное ребро ведет в предка в дереве обхода $\Rightarrow$ в дереве обхода есть цикл $\Rightarrow$ в графе есть цикл.

$\Rightarrow$ Рассмотрим цикл $v_0, v_1, ..., v_n = v_0$. Пусть (без ограничения общности) $v_0$ - первая вершина на этом цикле, которая была посещена во время обхода. Тогда по **лемме о белых путях** в процессе посещения $v_0$ была посещена и $v_{n-1}$, из которой вело ребро в серую вершину $v_0$. $\blacksquare$

# Проверка на ацикличность

```
def HasCylcle(G):
  color = [WHITE, WHITE, ..., WHITE]
  for v in G.V:
    if color[c] == WHITE:
      if HasCycleDfs(G, v):
        return True
  return False

def HasCycleDfs(G, v):
  color[v] = GRAY
  for u in G.neighbors(v):
    if color[u] == GRAY:
      return True
    if color[u] == WHITE:
      if HasCycleDfs(G, u):
        return True
  color[v] = BLACK
  return False
```

# Проверка на ацикличность

**Замечание 1.** Восстановить цикл можно с помощью хранения *parent*.

**Замечание 2.** Неориентированный граф представляется с помощью хранения ребер, ведущих в обе стороны. В чем проблема?

# Проверка на ацикличность

**Замечание 1.** Восстановить цикл можно с помощью хранения *parent*.

**Замечание 2.** Неориентированный граф представляется с помощью хранения ребер, ведущих в обе стороны. В этом случае одно ребро всегда будет давать цикл. Чтобы этого не происходило, нужно отдельно рассмотреть случай, когда сосед - родитель в дереве обхода.

**Замечание 3.** В случае кратных ребер пункт 2 не применим, так как в родителя может вести другое ребро. Что делать?

# Проверка на ацикличность

**Замечание 1.** Восстановить цикл можно с помощью хранения *parent*.

**Замечание 2.** Неориентированный граф представляется с помощью хранения ребер, ведущих в обе стороны. В этом случае одно ребро всегда будет давать цикл. Чтобы этого не происходило, нужно отдельно рассмотреть случай, когда сосед - родитель в дереве обхода.

**Замечание 3.** В случае кратных ребер пункт 2 не применим, так как в родителя может вести другое ребро. Но если в графе есть кратные ребра, то в нем гарантированно есть цикл.

# Слабая и сильная связность

*Ориентированный граф слабо связен*, если его неориентированная версия (замена ориентированных ребер на неориентированные) связна.

- Проверить слабую связность и найти компоненты слабой связности так же просто, как и для обычной связности

*Ориентированный граф сильно связен*, если из любой вершины есть путь до любой другой.

- Найти компоненты сильной связности можно тоже с помощью *DFS*. Этот алгоритм мы рассмотрим в следующий раз.