Содержание

1. Графы и базовый поиск в глубину

1.1 Определения

Определение 1. *Граф* G – пара множеств вершин и рёбер $\langle V, E\rangle$. V – множество вершин, E – множество ребер (пар вершин).

- Вершины ещё иногда называют *узлами.*
- Если направление ребёр не имеет значение, граф *неориентированный* (неорграф).
- Если направление ребёр имеет значение, граф *ориентированный* (орграф).
- Если ребру дополнительно сопоставлен вес, то граф называют *взвешенным.*
- Рёбра в орграфе ещё называют *дугами* и у ребра вводят понятие *начало* и *конец.*
- Если E – мультимножество, то могут быть равные рёбра, их называют *кратными.*
- Иногда, чтобы подчеркнуть, что E – мультимножество, говорят *мультиграф.*
- Для ребра $e = (a,b)$, говорят, что a *инцидентно* вершине b.
- *Степень* вершины a в неорграфе $\deg v$ – количество инцидентных ей рёбер.
- В орграфе определяют ещё входящую и исходящую степени: $\deg v = \deg_{in} v + \deg_{out} v$.
- Два ребра с общей вершиной называют *смежными.*
- Две вершины, соединённых ребром тоже называют *смежными.*
- Вершину степени ноль называют *изолированной.*
- Вершину степени один называют *висячей* или *листом.*
- Ребро (a,a) называют *петлёй.*
- *Простым* будем называть граф без петель и кратных рёбер.

Определение 2. *Путь* – чередующаяся последовательность вершин и рёбер, в которой соседние элементы инцидентны, а крайние – вершины. В орграфе направление всех рёбер от i к $i + 1$.

- Путь *вершинно простой* или просто простой, если все вершины в нём различны.
- Путь *рёберно простой*, если все рёбра в нём различны.

- Пути можно рассматривать и в неорграфах и в орграфах. Если в графе нет кратных рёбер, обычно путь задают только последовательностью вершин.

Замечание. Иногда отдельно вводят понятие маршрута, цепи, простой цепи. Мы, чтобы не захламлять лексикон, ими пользоваться не будем.

- *Цикл* – путь с равными концами. Циклы тоже бывают вершинно и рёберно простыми.
- *Ацикличный граф* – граф без циклов.
- *Дерево* – ацикличный связный неорграф.

1.2 Хранение графа

Будем обозначать $|V| = n$, $|E| = m$. Иногда сами V и E будут обозначать размеры.

Список ребер

Можно просто хранить рёбра: `pair<int,int> edges[m];` Чтобы в будущем удобно обрабатывать и взвешенные графы, и графы с потоком:

```
struct Edge {
   int from, to, weight;
};
Edge edges[m];
```

Матрица смежности

Можно для каждой пары вершин хранить наличие ребра, или количество рёбер, или вес...
`bool c[n][n];` для простого невзвешенного графа. n^2 бит памяти.
`int c[n][n];` для простого взвешенного графа или незвешенного мультиграфа. $\mathcal{O}(n^2)$ памяти.
`vector<int> c[n][n];` для взвешенного мультиграфа придётся хранить список всех весов всех рёбер между парой вершин.
`vector<vector<bool>> c(n, vector<bool>(n));` – чтобы первый способ правда весил n^2 бит. Константа времени работы увеличится (нужно достать определённый бит 32-битного числа).

Списки смежности

Можно для каждой вершины хранить список инцидентных ей рёбер: `vector<Edge> c[n];` Чтобы списки смежности умели быстро удалять, заменяем `vector` на `set/unordered\_set`.

	adjacent	get	all	add	del	memory
Список ребер	$\mathcal{O}(E)$	$\mathcal{O}(E)$	$\mathcal{O}(E)$	$\mathcal{O}(1)$	$\mathcal{O}(E)$	$\mathcal{O}(E)$
Матрица смежности	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(V^2)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(V^2)$
Списки смежности (vector)	$\mathcal{O}(deg)$	$\mathcal{O}(deg)$	$\mathcal{O}(V+E)$	$\mathcal{O}(1)$	$\mathcal{O}(deg)$	$\mathcal{O}(E)$
Списки смежности (hashTable)	$\mathcal{O}(deg)$	$\mathcal{O}(1)$	$\mathcal{O}(V+E)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(E)$

Список ребер

Сравнение способов хранения

Основных действий, которых нам нужно будет проделывать с графом не так много:

1. `adjacent(v)` – перебрать все инцидентные v ребра.
2. `get(a,b)` – посмотреть на наличие/вес ребра между a и b.
3. `all` - просмотреть все рёбра графа.
4. `add(a,b)`–добавить ребро в граф.
5. `del(a,b)` – удалить ребро из графа.

Ещё важно оценить дополнительную память.

Единственные плюсы первого способа – не нужна доппамять; в таком виде удобно хранить граф в файле (чтобы добавить одно ребро, допишем его в конец файла). Если матрица смежности уж слишком велика, можно хранить хеш-таблицу $\langle a,b\rangle \to c[a,b]$. В большинстве задач граф хранят списками смежности (иногда с `set` вместо `vector`).

Пример задачи, которую хорошо решает матрица смежности: даны граф и последовательность вершин в нём, проверить, что она – простой путь.

Пример задачи, которую хорошо решают списки смежности: пометить все вершины, смежные с v.

Пример задачи, где нужна сила обеих структур: даны две смежные вершины, найти третью, чтобы получился треугольник.

Мультисписок

Рёбра, смежные с v, лежат в односвязном списке `head[v]`, `next[head[v]]`, `next[next[head[v]]]`...

Все перечисленные элементы – номера рёбер.

По номеру ребра e можем хранить любую информацию про него, например, куда оно ведёт.

```
struct MultiList {
    struct Edge { int next, to; };
```

```
    vector<int> head; // first edge for every vertex
    vector <Edge > es; // all edges
    int e; // amount of edges
    MultiList(int n, int m) :
        head(n, -1), // -1 is a sign of the enf of the list
        e s ( m ) , // max amount of edges
        e = 0 { } // there is no edges at the start
    void addEdge(int a, int b) { // one directed edge
        es[e] = {head[a], b}, head[a] = e++;
    }
    void adjacent(int v) { // all connected with v edges
    for (int e = head[v]; e != -1; e = es[e].next)
        cout << es[e].next << " "; // or anything else
    }
};
```

По сути эти те же «списки смежности», но более аккуратно сохранённые.

Пусть $V = E = 10^6$, граф случайный. Оценим память. На 64-битной машине `vector<vector<int>> g` будет в итоге весить $X = ?_1 = 3 \cdot 8 + ?_2$ мегабайта (сам по себе вектор – 3 указателя). Можно подумать, что $?_2 = E \cdot 4$, но нет, в векторе size $\neq$ capacity $\Rightarrow$ нужно проводить эксперимент. На двух экспериментах видим, что $?_2 \approx E \cdot 12$ и $E \cdot 16$. Итого $X \approx 36 - 38$ мегабайт. А мультисписок $12 = 3 \cdot 4$ мегабайта. Итого разница в ≈ 3 раза. По скорости создания/удаления (выделить память, положить все элементы, освободить память) мультисписок будет в ≈ 10 раз быстрее.

1.3 Поиск в глубину

Поиск в глубину = *depth-first-search* = dfs.

Задача: пометить все вершины, достижимые из a.

Решение: рекурсивно вызываемся от всех соседей a.

```
vector<vector<int>> g(n);
void dfs(int v) {
    used[v] = true;
    for (int u : g[v])
        p[u] = v, dfs(u);
}

dfs(v);
```

Время работы $\mathcal{O}(E)$, так как по каждому ребру dfs проходит не более одного раза.

Для восстановления пути из вершины v в вершину u достаточно хранить массив предков, где для каждой вершины, хранится номер вершины, из которой мы пришли. У стартовой $p[v] = -1$.

Немного его модифицируем, а именно будем сохранять для каждой вершины, в какой момент мы в неё вошли и в какой вышли — соответствующие массивы будем называть tin и $tout$.
Как их заполнить: заведем таймер, отвечающий за «время» на текущем состоянии обхода, и будем инкрементировать его каждый раз, когда заходим в новую вершину:

```
    std::vector<int> tin(n), tout(n);
    int t = 0; // timer

    void dfs(int v) {
        tin[v] = t++;
        for (int u : g[v])
            if (!used[u])
                dfs(u);
        tout[v] = t;
}
```

Полезные свойства этих массивов:

1. Вершина u является предком $v_v \in [tin_u, tout_u)$. Эту проверку можно делать за константу.
2. Два полуинтервала $[tin_u, tout_u)$ и $[tin_v, tout_v)$ либо не пересекаются, либо один вложен в другой.
3. В массиве tin есть все числа от 0 до $(n-1)$, причём у каждой вершины свой номер.
4. Размер поддерева вершины v (включая саму вершину) равен $(tout_v - tin_v)$.
5. Если ввести нумерацию вершин, соответствующую tin-ам, то индексы любого поддерева всегда будут каким-то промежутком в этой нумерации.

1.4 Компоненты связности

Определение 3. Две вершины u и v называются *связанными* (adjacent), если в графе G существует путь из u в v (обозначение: $u \rightsquigarrow v$).

Определение 4. *Компонентой связности неориентированного графа* называется подмножество вершин, достижимых из какой-то заданной вершины.

Теорема 1. *Связанность — отношение эквивалентности.*

Доказательство. Рефлексивность: $\forall a \in V \;\; a \rightsquigarrow a$.
Симметричность: $a \rightsquigarrow b \Rightarrow b \rightsquigarrow a$ (в силу неориентированности).
Транзитивность: $a \rightsquigarrow b \land b \rightsquigarrow c \Rightarrow a \rightsquigarrow c$
Действительно, сначала пройдем от a до b, затем от b до c, что и означает существования пути $a \rightsquigarrow c$. □

Поиск компонент связности

Для решения задачи модифицируем обход в глубину так, чтобы запустившись от вершины какой-то компоненты, от пометил все вершины этой компоненты — то есть все достижимые вершины — заданным номером этой компоненты.

```
void dfs(int v, int col) {
    used[v] = col;
    for (int u : g[v])
        if (!used[v])
            p[u] = v, dfs(u, col);
}

int comp = 0;
for (int v = 0; v < n; v++)
    if (!used[v])
        dfs(v, ++comp);
```

Итоговая асимптотика составит $\mathcal{O}(V+E)$, потому что такой алгоритм не будет запускаться от одной и той же вершины дважды, и каждое ребро будет просмотрено ровно два раза (с одного конца и с другого).

1.5 Классификация рёбер

После dfs(v) остаётся дерево с корнем в v. Отец вершины u – та, из которой мы пришли в v. Пусть все вершины достижимы из v. Рёбра разбились на следующие классы:

1. Древесные: принадлежат дереву.
2. Прямые: идут вниз по дереву.
3. Обратные: идут вверх по дереву.
4. Перекрёстные: идут между разными поддеревьями.

Рёбра можно классифицировать относительно любого корневого дерева, но именно относителмно дерева, полученного dfs в неорграфе, нет перекрёстных рёбер.

Лемма 1. Относительно дерева dfs неорграфа нет перекрёстных рёбер.

Доказательство. Если есть перекрёстное ребро $a \to b$, есть и $b \to a$ (граф неориентированный). Пусть $tin_a < tin)b$. $a \to b$ перекрёстное $\Rightarrow tin_b > tout_a$. Противоречие с тем, что dfs пытался пройти по ребру $a \to b$. □

1.6 Топологическая сортировка

Определение 5. Топологической сортировкой орграфа называется сопоставление вершинам номеров $ind[v] : \forall (a \to b) \ ind\_a < ind\_b$.

Лемма 2. Топологическая сортировка существует тогда и только тогда, когда граф ацикличен.

Доказательство. Если есть цикл, то рассмотрим неравенства по циклу и получим противоречие.

Если цикла нет, то сещтвует вершина нулевой входящей степени, сопоставим ей $ind[v] = 0$, удалим ее из графа, граф останется ациклинчым, по индукции нумеруем оставшиеся вершины. □

В процессе доказательства получили нерекурсивный алгоритм топологической сортировки за $\mathcal{O}(V + E)$: поддерживаем входящие степени и очереди вершин нулевой входящей степени. Итенрация:

```
    v = q.pop();
    topsort.push_back(v);
    for (int u : g[v])
        if (--dex[u] == 0)
            q.push(u);
;
```

Алгоритм топологической сортировки

dfs умеет сортировать вершины по времени входа и времени выхода.

```
    void dfs(int v) {
        in_time_order.push_back(v);
        ...
        out_time_order.push_back(v);
    }
```

Топологический порядок вершин записан в `reverse(out_time_order)`.

Доказательство. Пусть есть ребро $a \to b$, тогда мы сперва выйдем из b и тоолько затем из a. □

2. Введение в теорию сложности

2.1 Основные классы

Алгоритмы позволяют для какой-то задачи сказать, за сколько она решается: дана задача $A \to$ решим ее за $\mathcal{O}(2^n)$

Теория сложности же позволяет сказать, что для какой-то задачи не существует алгоритма, решающего ее за какую-то асимптотику: дана задача $A \to$ не решается быстрее, чем за $\mathcal{O}(n \log n)$.

Алгоритмически не разрешимые задачи

Существуют ли неразрешимые задачи (то есть для которых нет решающих их алгоритмов)? На самом деле, таких задач больше, чем алгоритмов.

Рассмотрим следующие задачи: $A : \begin{cases} \text{input: } n \in \mathbb{N} \\ \text{output: } true/false \end{cases}$.

Любую такую задачу можно задать подмножеством натуральных чисел, на которых ответ true $A \subseteq \mathbb{N}$.

Задач по крайней мере столько, сколько множеств натуральных чисел — $|2^N|$. Алгоритмов счётное число (то есть $|N|$), ведь их всех можно пронумеровать: сначала выпишем все однобуквенные, затем все двухбуквенные, и так далее.

$|2^n| > |N|$ — тут можно либо сослаться на общую теорему Кантора ($2^A > |A|$), либо вспомнить школьные доказательства того, что $|2N| = |R|$ и $|R| > |N|$. Так что на самом деле «почти все» задачи неразрешимы.

Пример 1. Неразрешимая задача: задача остановки HALTING: *Дана программа, остановится ли она когда-нибудь на данном входе?*

$$HALTING : \begin{cases} \text{input: код программы и вход для этой программы} \\ \text{output: остановится ли запуск?} \end{cases}.$$

Теорема 2. *Halting problem алгоритмически не разрешима.*

Доказательство. От противного. Пусть есть алгоритм terminates(code, x), всегда останавливающийся, и возвращающий true только если code(x) останавливается. Рассмотрим программу:

```
def invert(code):
    if terminates(code, code): while (true)
```

Запустим invert(invert), что может случится:

1. Он зависнет $\Rightarrow$ terminate(invert, invert)$=false \Rightarrow$ invert не зависает ?!

2. Он завершится $\Rightarrow$ terminate(invert, invert)$=true \Rightarrow$ invert зависает ?!

Противоречие. Значит, такого terminate не существует. □

Теорема 3. ***Теорема Успенского-Райса***

Любое нетривиальное свойство программ неразрешимо (то есть нет алгоритма, которые бы его проверял).

Поясним, что это значит. Будем говорить, что программы A и B эквивалентны ($A \sim B$), если для каждого входа A либо они обе зависают, либо обе останавливаются и печатают одно и тот же ответ (время работы и память при этом могут отличаться).

Определение 6. *Свойство программы* – это такой предикат $P(code)$ который для любых эквивалентных программ $\mathcal{A}$ и $\mathcal{B}$ выдаёт одно и то же: $\mathcal{A} \sim \lfloor \Rightarrow P(\mathcal{A}) = P(\mathcal{B})$. «Нетривиальное» означает, что хотя бы одна программа ему удовлетворяет, но не все программы.

Теорема 4. ***Теорема Гёделя о неполноте***
Если формальная система S непротиворечива, то в ней невыводимы обе формулы B и $!B$; иначе говоря, если система S непротиворечива, то она неполна, и B служит примером неразрешимой формулы.

Один из способов доказательства этой теоремы – через неразрешимость.

2.2 Decision/search problem

Определение 7. Если в задаче ответ – true/false, то это *decision problem* (задача распознавания). Иначе это *search problem* (задача поиска).

Пример 2.

1. Decision: проверить, есть ли x в массиве a.
2. Search: Найти позицию x в массиве a.
3. Decision: Проверить, есть ли путь из a в b в графе G.
4. Search: Найти сам путь.
5. Decision: Проверить, есть ли в графе клика размера хотя бы k.
6. Search: Найти максимальный размер клики (или саму клику).

Замечание. Decision problem f можно задавать, как язык (множество входов) $L = \{x : f(x) = true\}$.

2.3 DTime, P, EXP (классы для decision задач)

Определение 8. DTime$[f(n)]$ – множество задач распознавания, для которых $C > 0$ и детерминированный алгоритм, работающий на всех входах не более чем $C \cdot f(n)$, где n – длина входа.

Пример 3. IS_SORTED $\in$ DTime[n]
IS_SORTED $\in$ DTime[n^2]

Определение 9. $\mathrm{P} = \bigcup\limits_{k>0} \mathrm{DTime}[n^k]$. Т.е. задачи, имеющие полиномиальное решение.

Определение 10. $\mathrm{EXP} = \bigcup\limits_{k>0} \mathrm{DTime}[2^{n^k}]$. Т.е. задачи, имеющие экспоненциальное решение.

Пример 4. KNAPSACK: n предметов, рюкзак размера w, можно ли уложить $\geq k$ предметов?
Умеем решать за $\mathcal{O}(2^n \cdot n)$, за $\mathcal{O}(nw)$ (не полиномиальное решение!).
Длина входа: $\mathcal{O}(n \log n + W \log W + k \log n + n \log W) = |x|$.

Теорема 5. ***Об иерархии по времени***
$DTime[f(n)] \subsetneq DTime[f(n) \log^2 f(n)]$.

Доказательство. $\subset$ понятно, почему.
Но почему не $\subseteq$? Значит, существует задача, которая решает за $\mathcal{O}(f(n) \log^2 f(n))$ и не решается за $\mathcal{O}(f(n))$ шагов.
Задача: дана программа и вход. Завершится ли она на этом входе за $f(n) \log f(n)$ шагов?

□

Следствие 1. *P$\neq$ EXP.*

Доказательство. $\mathrm{P} \subseteq \mathrm{DTime}[2^n] \subsetneq \mathrm{DTime}[2^{2n}] \subseteq \mathrm{EXP}$. □

2.4 NP (non-deterministic polynomial)

Задача $\to$ да + сертификат / нет.

Определение 11. $NP = \{L : \exists$ алгоритм M, работающий за полином от $|x|$, $\forall x\ (\exists y :\ M(x,y) = 1) \Leftrightarrow (x \in L)\}$.

Неформально: «NP – класс языков $L : \forall x \in L$, если нам дадут подсказку $y(x)$, то мы за полином сможем убедиться, что $x \in L$».
Ещё более неформально: «NP ˘ класс задач, к которым ответ можно проверить за полином».
Подсказку y так же называют *свидетелем/сертификатом* того, что x лежит в L.

Пример 5. Примеры NP-задач:

1. HAMPATH = $\{G \mid G$ – неорграф, в котором есть гамильтонов путь$\}$.

   Подсказка y – путь. M получает вход $x = G$, подсказку y проверяет, что y прост, $|y| = n$ и $\forall (e \in y)\ e \in G$.

2. k-CLIQUE – проверить наличие в графе клики размером k.

   Подсказка y – клика.

3. IS-SORTED – отсортирован ли массив? Она даже лежит в P.

Замечание. P $\subseteq$ NP (можно взять пустую подсказку).

Определение 12. coNP$= \{L \mid \overline{L} \in$NP$\}$

Определение 13. coNP$= \{L : \exists M$, работающий за полином от $|x|, \forall x\ (\exists y\ M(x,y) = 0) \Leftrightarrow (x \in L)\}$.

Определение 14. coNP$= \{L : \exists M$, работающий за полином от $|x|, \forall x\ (\exists y\ M(x,y) = 1) \Leftrightarrow (x \notin L)\}$.

Пример 6. Пример coNP задачи:
PRIME – является ли число простым. Подсказкой является делитель.
На самом деле PRIME $\in$ P, но этого мы пока не умеем понимать.

Замечание. Вопрос P $=$ NP или P $\neq$ NP остаётся открытым. Предполагают, что $\neq$.

2.5 NP-hard, NP-complete

Определение 15. $\exists$ *полиномиальное сведение (по Карпу)* задачи A к задаче B $(A \leq_P B) \Leftrightarrow \exists$ алгоритм f, работающий за полином, $(x \in A) \Leftrightarrow (f(x) \in B)$.

Замечание. f работает за полином $\Rightarrow |f(x)|$ полиномиально ограничена $|x|$.

Определение 16. $\exists$ *сведение по Куку* задачи A к задаче B $(A \leq_C B) \Leftrightarrow \exists M$, решающий A, работающий за полином, которому разрешено обращаться за $\mathcal{O}(1)$ к решению/оракулу B.

Ещё говорят «задача A сводится к задаче B».
В обоих сведениях мы решаем задачу A, используя уже готовое решение задачи B.
Другими словами доказываем, что «A не сложнее B». Различие в том, что в первом случае решением B можно воспользоваться только один раз (и инвертировать ответ нельзя), во втором случае – полином раз.

Определение 17. NP-hard $=$ NPh $= \{L : \forall A \in$ NP $\Rightarrow A \leq_P L\}$.

NP-трудные задачи – класс задач, которые не проще любой задачи из класса NP.

Определение 18. NP-complete $=$ NPc $=$ NPh $\cap$ NP.

NP-полные задачи – самые сложные задачи в классе NP.

Если мы решим хотя бы одну из NPc за полином, то решим все из NP за полином. Хорошая новость: все NP-полные по определению сводятся друг к другу за полином.

Замечание. Когда хотите выразить мысль, что задача трудная в смысле решения за полином (например, поиск гамильтонова пути), неверно говорить «это NP задача» (любая из P тоже в NP) и странно говорить «задача NP-полна» (в этом случае вы имеете в виду сразу, что и трудная, и в NP). Логично сказать «задача NP-трудна».

Лемма 3. $A \leq_P B, B \in P \Rightarrow A \in P$.

Доказательство. Сведение f работает за n^s, B решается за $n^t \Rightarrow A$ решается за n^{st}. □

Лемма 4. $A \leq_P B, A \in$NPh $\Rightarrow B \in$ NPh.

Доказательство. $\forall L \in$NP $(\exists f : L$ сводится к A функцией $f(x)) \vee (A \leq_P B$ функцией $g(x)) \Rightarrow L$ сводится к B функцией $g(f(x))$ за полином. □

NP-полные задачи существуют!

Приведём простую и очень важную теорему. На экзамене доказательство можно сформулировать в одно предложение, здесь же оно для понимания расписано максимально подробно.

Определение 19. BH = BOUNDED-HALTING: вход $x = \langle \underbrace{11...1}_{k}, Mx\rangle$, проверить, $\exists$ ли такой $y : M(x,y)$ остановится за k шагов и вернёт $true$.

Теорема 6. *BH = BOUNDED-HALTING* $\in$ *NPc.*

Доказательство.

1. BH $\in$ NP

   Подсказка – такой y. Алгоритм – моделирование k шагов M за $\mathcal{O}((k))$.

   Важно, что если бы число k было записано, используя $\log_2 k$ бит, моделирование работало бы за экспоненту от длины входа, и нельзя было бы сказать «задача лежит в NP».

2. BH $\in$ NPh. То есть нужно доказать, что любой язык из NP к ней сводится.

   $L \in$ NP: $\exists y : \ A(x,y) = 1 \Leftrightarrow x \in L$

   A – полиномиальный алгоритм $\Rightarrow \exists P(x)$, ограничивающий время работы A. Программа A всегда отрабатывает за $P(|x|)$, если ее запустить с ограничением $P(|x|)$, то ничего не поменяется.

   Рассмотрим $f(x) = (\underbrace{11...1}_{P(|x|)}, A, x)$. Получили полиномиальное сведение:

   $x \in L \Leftrightarrow \exists y : A(x,y) = 1 \Leftrightarrow f(x) \in$ BH.

□

2.6 Сведения, новые NP-полные задачи

Началось всё с того, что в 1972-м Карп опубликовал список из 21 полной задачи, и дерево сведений. Кстати, в его работе все сведения крайне лаконичны. Итак, приступим:
Чтобы доказать, что $B \in$ NPh, нужно взять любую $A \in$ NPh и свести A к B полиномиально. Пока такая задача A у нас одна – BH. На самом деле их очень много.
Чтобы доказать, что $B \in$ NPc, нужно ещё не забыть проверить, что $B \in$ NP.
Во всех теоремах ниже эта проверка очевидна, мы проведём её только в доказательстве первой.

Теорема 7. *BH* $\leq_p$ *CIRCUIT-SAT* $\leq_p$ *SAT* $\leq_p$ *3-SAT* $\leq_p$ *k-INDEPENDENT* $\leq_p$ *k-CLIQUE*

BH$= \{(A,x,1^k) \mid A(x)\}$

Определение 20. CIRCUIT-SAT. Дана схема, состоящая из входов, выхода, гейтов AND, OR, NOT. Проверить, существует ли набор значений на входах, дающий *true* на выходе.

Теорема 8. *CIRCUIT-SAT* $\in$ *NPc.*

Доказательство. $\{C_{\text{схема}} \mid \exists \vec{x} : C(\vec{x}) = 1\}$, $\vec{x} = (x_1,...,x_n)$

- $\in$ NP

  Подсказка – набор значений на входах $\Rightarrow$ CIRCUIT-SAT NP.

- $\in$ NPh

  Сводим BH к CIRCUIT-SAT $\Rightarrow$ нам даны программа A, время выполнения k, вход x'.

  $f(A,x,\underbrace{1..1}_{k}) = C\ (A,x,1^k) \in \text{BH} \Leftrightarrow C \in \text{CIRCUIT-SAT}$

  За время k программа обратится не более чем к k ячейкам памяти.

  Обозначим за $s_{i,j}$ состояние true/false j-й ячейки памяти в момент времени i. $s_{o,i}$ – вход, $s_{t,potput}$ – выход, $\forall i \in [1,k)$ $s_{i,j}$ зависит от $\mathcal{O}(1)$ переменных $(i1)$-го слоя. f запишем в КНФ, чтобы получить гейты вида AND, OR, NOT. Размер КНф $- \mathcal{O}(1)$ (КНФ –частный случай схемы). Размер схемы вырастет в константу раз. Получили $\mathcal{O}(tn)$ булевых гейтов $\Rightarrow$ по (A,x,k) за полином построили вход к CIRCUIT-SAT.

□

Теорема 9. *SAT* $\in$ *NPc.*

Доказательство. TODO □

Теорема 10. *3-SAT* $\in$ *NPc.*

Доказательство. Пусть есть клоз $(x_1 \vee x_2 \vee ... \vee x_n)$, $n \geq 4$. Введём новую переменную w и заменим его на $(x_1 \vee x_2 \vee w) \wedge (x_1 \vee x_2 \vee ... \vee x_n \vee \overline{w})$.
$\phi(x_1,...,x_n) \to \psi(x_1,...,x_n, w)$. Хотим доказать, что ϕ выполнима $\Leftrightarrow \psi$ выполнима.

1. ϕ выполнима $\Rightarrow \psi$ выполнима.

   Если $x_i = 1$, то одна часть формулы равна 1, а вторую можно сделать равна 1, если w (или $\overline{w}$) сделать равной 1.

2. ψ выполнима $\Rightarrow \phi$ выполнима.

   Посмотрим на скобку, в которой $w = 0$. Значит, там есть $x_i = 1 \Rightarrow \phi(...) = 1$.

□

Определение 21. k-INDEPENDENT: $(G,k) \to \exists$ существует независимое множество размера $\geq k$?

Теорема 11. *k-INDEPENDENT $\in$ NPc.*

Доказательство. Сведем: 3-SAT $\leq_p$ k-INDEPENDENT
Наша формула – m клозов $(l_{i1} \vee l_{i2} \vee l_{i3})$, где l_{ij} – литералы.
Построим граф из ровно $3m$ вершин – l_{ij}. $\forall i$ добавим треугольник (l_{i1},l_{i2},l_{i3}) (итого $3m$ рёбер).
В любое независимое множество входит максимум одна вершина из каждого треугольника.
$\forall k = 1...n$ соединим все вершины $l_{ij} = x_k$ со всеми вершинами$l_{ij} = \overline{x}_k$.
Теперь $\forall k = 1...n$ в независимое множество нельзя одновременно включить x_k и $\overline{x}_k$.
Итог: $\exists$ независимое размера $m \Rightarrow$ у 3-SAT было решение. □

Теорема 12. *k-CLIQUE $\in$ NPc.*

Доказательство. Сведем: k-CLIQUE $\leq_p$ k-INDEPENDENT.
$(G,k) \leftrightarrow (\overline{G},k)$, $\overline{G} = (V,\overline{E})$
$(vu) \in E \Leftrightarrow (vu) \notin \overline{E}$ □

TODO

1. GI, (G,H)

   $T = 2^{\mathcal{O}(\log^3 n)} = n^{\mathcal{O}(\log^2 n)}$

2.7 Задачи поиска

Определение 22. $\overline{NP}, \overline{NPc}, \overline{NPh}^{\smile}$.

Сведение задач минимизации, максимизации к decision задачам

Пусть мы умеем проверять, есть ли в графе клика размера k. Чтобы найти размер максимальной клики, достаточно применить бинпоиск по ответу. Это общая техника, применимая для максимизации/минимизации численной характеристики.
MAX-CLIQUE$\to$ k-CLIQUE

Сведение search задач к decision задачам

Последовательно фиксируются биты (части) подсказки y.

Пример 7.

1. 3-SAT

   $\varphi(x_1,..,x_n)$, φ невыполнима $\Rightarrow +$

   Рассмотрим $x_n = 0$.

   $\varphi(x_1,..,x_{n-1},0)$ невыполнима $\Rightarrow x_n = 1$.

   $\varphi(x_1,..,x_{n-1},0)$ выполнима $\Rightarrow x_n = 0$.

   И перейдем к следующей переменной.

2. k-INDEPENDENT

   $G' = G \setminus \{v\}$

   G' $-$ есть k-IS $\Rightarrow G := G'$

   Иначе IS$=$ $\{v\}\cup$ IS $(G\{u \mid g[v]\})$ (соседи v)

Решение NP-трудных задач

Если встретилась задача, которую не удается быстро решить, то:

1. Поискать ее в списке трудных.

2. Свести трудную к ней.

3. Свести задачу к SAT (существуют SAT-solver'ы, которые могут ее решить).