https://cp-algorithms.com/graph/edmonds_karp.html

Максимален поток – FordFulkerson и Edmondс-Karp

Алгоритъмт Edmonds-Karp е имплементация на метода на Ford-Fulkerson за изчисляване на максимален поток в потокова мрежа.

❖ Понятия (потокова мрежа, поток и максимален поток)

Мрежа е насочен граф G с множество върхове V и множество ребра E, комбиниран с функция C, която присвоява на всяко ребро **e** ∈ E неотрицателно цяло число – **капацитет** на реброто.
Такава мрежа се нарича **потокова мрежа**, **ако допълнително маркираме два върха**, единият за **източник (s),** а другият за **приемник (t)**.
Поток в потокова мрежа е функция f, която присвоява на всяко ребро **e** неотрицателно цяло число, а именно поток.

Функцията f (поток) трябва да отговаря на следните две условия:

Условие 1: Потокът на реброто не може да надвишава капацитета му:

$$f(e) \leq c(e)$$

Условие 2: За всеки връх, с изключение на върховете s и t, сумата от входящия във върха поток трябва да е равна на сумата на изходящия от него поток:

$$\sum_{(v,u)\in E} f((v,u)) = \sum_{(u,v)\in E} f((u,v))$$

Източникът **s** има само изходящ поток, а приемникът **t** - само входящ поток.
Лесно е да се види, че е в сила следното уравнение:

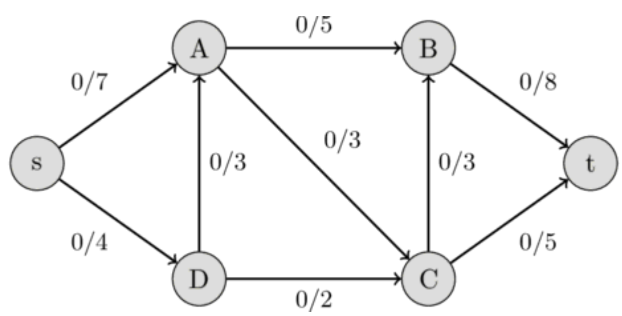
$$\sum_{(s,u)\in E} f((s,u)) = \sum_{(u,t)\in E} f((u,t))$$

❖ Добра аналогия за потокова мрежа е визуализация, при което представяме ребрата като водопроводи:

Капацитетът на ребро е максималното количество вода, което може да през него за секунда, а **потокът** на ребро е количеството вода за секунда, което тече в момента през тръбата.
През тръбата не може да преминава повече вода, отколкото е нейният капацитет(условие 1).

Върховете действат като кръстовища, където водата излиза от някои тръби и се разпределя по някакъв начин към други тръби. Това е и Условие 2 за потока - във всяко кръстовище цялата постъпваща вода се разпределя към другите тръби. Не може магически да изчезне или да се появи. От източника s е излиза цялата вода и тя може да се оттича само към приемника,

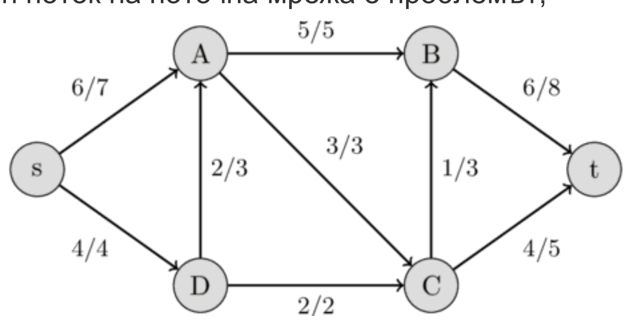
Фигурите изобразяват потокова мрежа.
За всяко ребро първата стойност е **поток**, втората - **капацитет**.
Първоначално потокът в мрежата е 0 ==========>



Стойността на потока в мрежа е сумата от всички потоци, които се произвеждат в източник **s** или еквивалентното - потоците, които се консумират от приемника **t**.

Максимален поток е поток с максималната възможна стойност. Намирането на този максимален поток на поточна мрежа е проблемът, който искаме да решим.

При визуализацията с водопроводи проблемът може да се формулира така:
колко вода можем да прокараме през тръбите от източника **s** до приемника **t**?
Изображениеъо вдясно ====================== >
показва максималния поток в поточната мрежа.



I. Метод на Форд-Фулкерсон

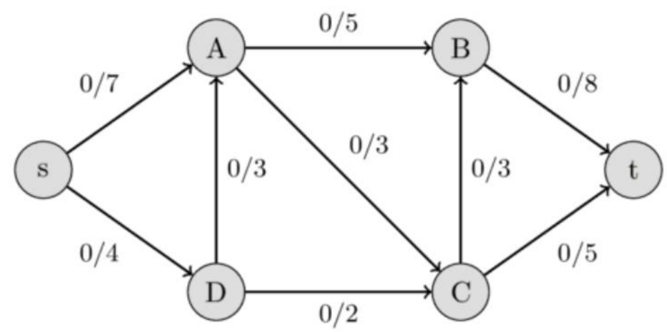
остатъчен капацитет на насочено ребро **е капацитетът минус потока**.
Трябва да се отбележи, че ако има поток през някакво насочено ребро (u, v), тогава обратното ребро има капацитет 0 и можем да определим потока през него като $f(v,u)= -f(u,v)$,
Това определя и остатъчните капацитети за всички обратни ребра. От всички тези ребра можем да създадем **остатъчна мрежа**, която е мрежа със същите върхове и ребра, но с използване на остатъчните капацитети като капацитети.

Методът на Форд-Фулкерсън работи по следния начин:
Първо задаваме потока на всяко ребро на нула. След това търсим **увеличаващ път** от **s** към **t.**

Увеличаващият път е прост път в остатъчния граф, т.е. по ребрата на който остатъчният капацитет е положителен. Намерен ли е такъв път, тогава можем да го добавим за увеличаване на потока по тези ребра. Продължаваме да търсим увеличаващи пътища и увеличаваме потока, докато намираме увеличаващ път. Когато приключи процесът, потокът е максимален.

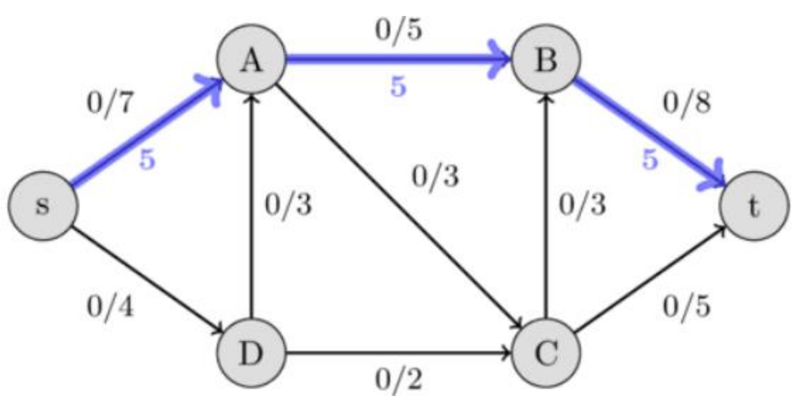
Как увеличаваме потока по увеличаващ път?

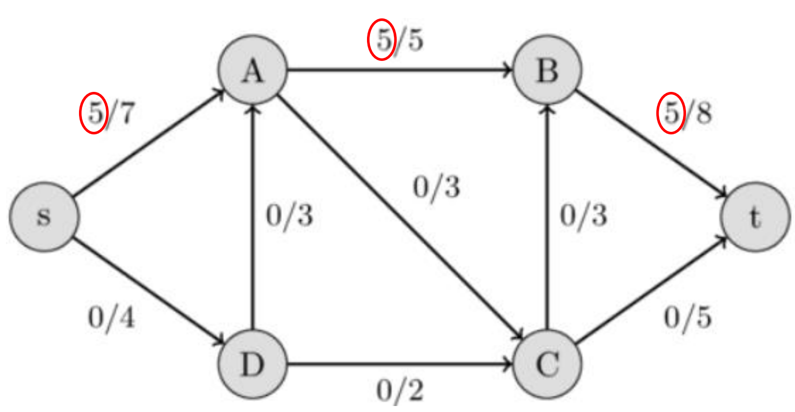
Първоначално започваме с поток 0 =============>
Намираме най-малкия остатъчен капацитет C
по ребрата от пътя s → t,
След това увеличаваме потока, като за всяко ребро (u,v) от пътя s → t актуализираме f(u,v) += C и f(v,u) –= C



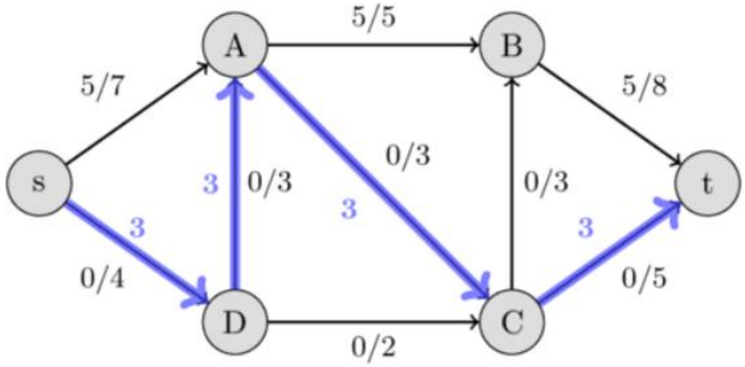
1. Намираме пътя **s – A – B - t** и остатъчните капацитети **7, 5 и 8**.

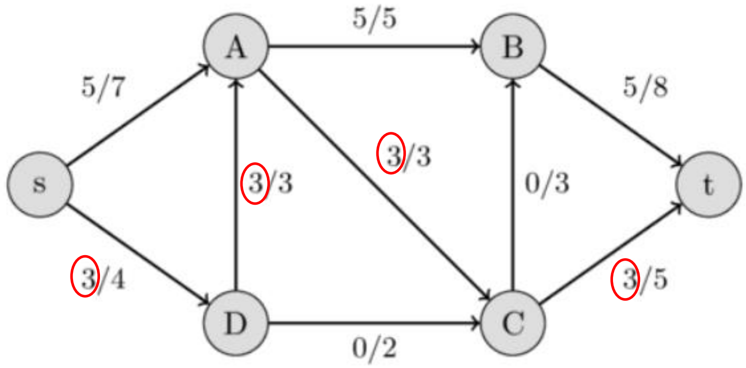
Техният минимум е 5, следователно можем да увеличим потока по този път с 5.
Това дава поток 5 за мрежата.



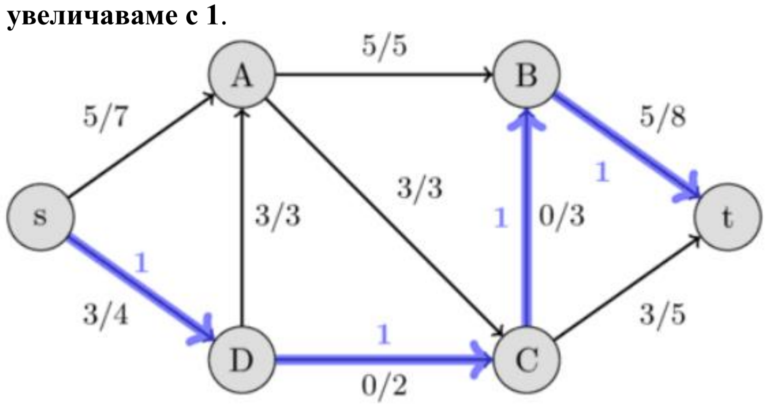


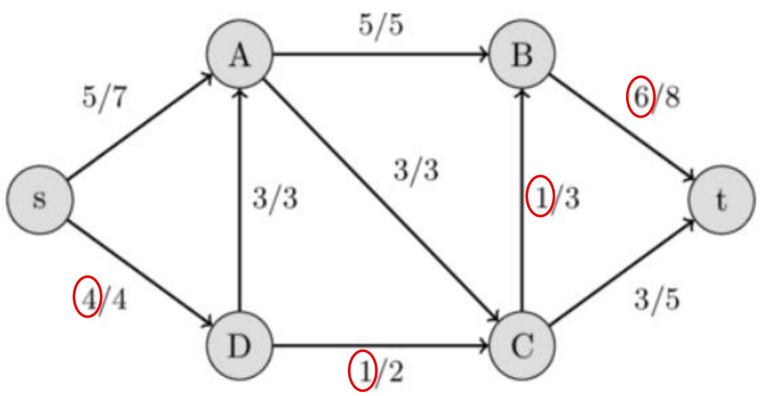
2. Отново търсим увеличаващ път и намираме пътя **s – D – A – C – t**
с остатъчните капацитети **4**, **3**, **3** и **5**. Следователно **можем да увеличим потока с 3**
и така получаваме поток **8** за мрежата.





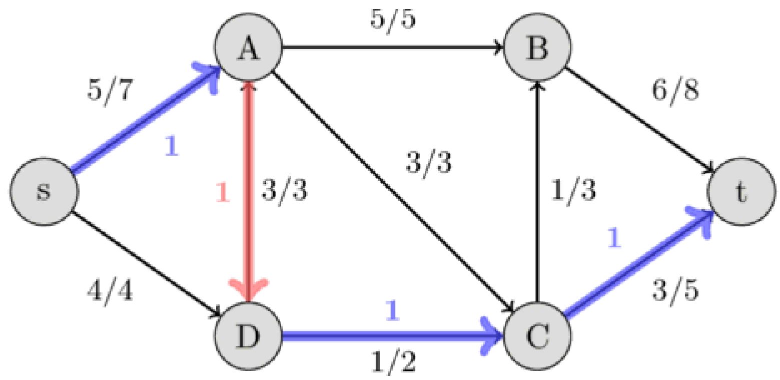
3. Сега намираме пътя **s – D – C – B – t** с остатъчните капацитети **1** (4-3)**, 2, 3 и 3** (8-5)
Минималният е 1 ==> **увеличаваме с 1**.



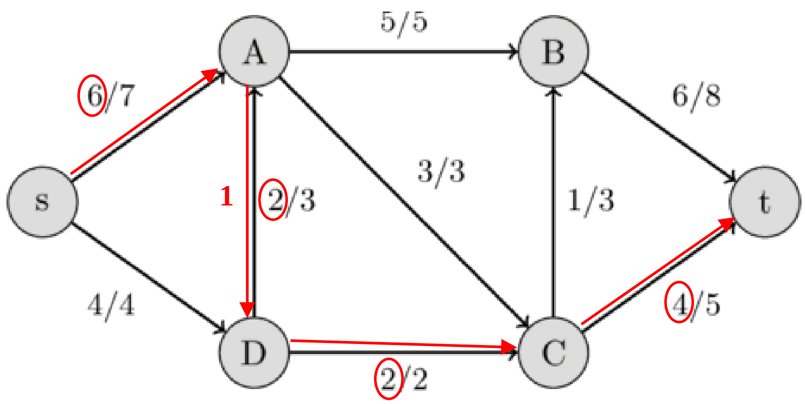


4. След това намираме увеличаващия се път **s – A – D – C – t**
с остатъчните капацитети **2, 3, 1 и 2**.

Можем да увеличим с 1, **но този път е много интересен** - включва **обратното ребро (A-D)**
В оригиналната мрежа за потоци не ни е позволено да изпращаме никакъв поток от A до D.
Но **тъй като вече имаме поток 3 от D до A, е възможно**.



Идеята е следната: Вместо да изпращаме поток 3 от D до A, изпращаме само 2
и компенсираме това, като изпращаме **допълнителен поток поток 1 от s до A**,
което ни позволява да изпратим допълнителен поток 1 по пътя D – C – t



Вече е **невъзможно** да се намери увеличаващ път между s и t, следователно този поток
е максимално възможният. **Намерихме максималния поток. Той е 10**. ☺

Трябва да се отбележи, че **методът на Форд-Фулкерсон не уточнява метод за намиране на увеличаващия се път**.
Възможните подходи използват DFS или BFS, които работят за време O(E).

Ако всички капацитети на мрежата са цели числа, тогава за всеки увеличаващ път потокът на мрежата се увеличава поне с 1 (за повече подробности вижте Теорема за интегрален поток).
Следователно сложността на Ford-Fulkerson е O(E.F), където F е максималният поток на мрежата.

При **рационални капацитети**, алгоритъмът също ще прекрати, но сложността не е ограничена.
При **ирационален капацитет**, алгоритъмът никога няма да се прекрати и дори не може да се доближи до максималния поток.

II. Алгоритъм на Edmonds-Karp

Алгоритъмът на Edmonds-Karp е просто имплементация на метода на Ford-Fulkerson, който **използва BFS за намиране на увеличаващи се пътища**.
Публикуван за първи път от Юфим Диниц през 1970 г., а по-късно през 1972 г. независимо е публикуван и от Джак Едмондс и Ричард Карп.
Сложността може да се даде независимо от максималния поток.
Алгоритъмът работи за време $O(E^2.V)$ дори за ирационални капацитети.
Идеята е, че всеки път, когато намерим увеличаващ се път, едно от ребрата става наситено, а разстоянието от реброто до **s** ще бъде по-дълго, ако се появи по-късно отново в увеличаващ се път. Дължината на простите пътища е ограничена от броя V на върховете.

❖ Имплементация

Матрицата `capacity` съхранява капацитета за всяка двойка върхове.

`adj` е **свързан списък на ненасочения граф,** тъй като трябва да използваме и обратните ребра, когато търсим увеличаващи пътища.

Функцията `maxflow` ще върне стойността на максималния поток.

Всъщност по време на алгоритъма матрицата `capacity` ще съхранява остатъчния капацитет на мрежата, а стойността на потока във всяко ребро няма да бъде запазена.
Но лесно може да се разшири реализацията чрез използване на допълнителна матрица, която да съхрани и потока и да върне стойността му.

```
int n;
vector<vector<int>> capacity;
vector<vector<int>> adj;

int bfs(int s, int t, vector<int>& parent) // търси увеличаващи пътища
{
    fill(parent.begin(), parent.end(), -1);
    parent[s] = -2;
    queue<pair<int, int>> q;
    q.push({s, INF});

    while (!q.empty())
    {
        int cur = q.front().first;
        int flow = q.front().second; q.pop();

        for (int next : adj[cur])
        {
            if (parent[next] == -1 && capacity[cur][next])
            {
                parent[next] = cur;
                int new_flow = min(flow, capacity[cur][next]);
                if (next == t)  return new_flow;
                q.push({next, new_flow});
            }
        }
    }
    return 0;
}
```

```
int maxflow(int s, int t)
{
    int flow = 0;
    vector<int> parent(n);
    int new_flow;

    while (new_flow = bfs(s, t, parent)) /// докато намира път
    {
        flow += new_flow;
        int cur = t;                      /// започва от t (приемника)
        while (cur != s)                  /// докато не стигне до s
        {
            int prev = parent[cur];
            capacity[prev][cur] -= new_flow;
            capacity[cur][prev] += new_flow;
            cur = prev;
        }
    }
    return flow;
}
```

❖ Теорема за интегрален поток

Теоремата просто казва, че ако всеки капацитет в мрежата е цяло число, тогава и потокът във всяко ребро ще бъде цяло число в максималния поток.

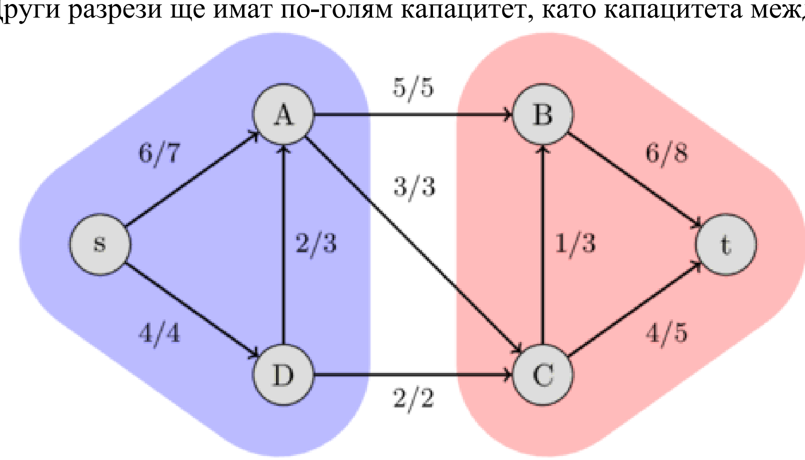
❖ Max-flow min-cut теорема (максимален поток с минимален разрез)

s-t-cut е разделяне на върховете на поточна мрежа на две множества, така че едното включва източника s, а другото - приемника t.

Капацитетът на s-t-cut е сумата от капацитетите на ребрата от източника към приемника. Очевидно не можем да изпратим повече поток от s към t, отколкото капацитета на който и да е s-t-разрез. Следователно максималният поток е ограничен от минималния капацитет на срязване.

Теоремата за максимален поток с минимален разрез отива по-надалеч. Тя казва, че капацитетът на максималния поток трябва да е = на капацитета на минималния разрез.

На следващото изображение можете да видите минималния разрез на поточната мрежа, който използвахме по-рано. Тя показва, че капацитетът на рязането {s,A,D} и {B,C,t} е 5+3+2=10, което е равно на максималния поток, който намерихме.
Други разрези ще имат по-голям капацитет, като капацитета между {s,A} и {B,C,D,t} е 4+3+5=12,



Минимален разрез може да се намери след изчисление на максимален поток по метода на Форд-Фулкерсон.

Един възможен минимален разрез е следният:
множество от всички върхове, до които може да се достигне s в остатъчния граф (използвайки ребра с положителен остатъчен капацитет) и множеството от всички останали върхове. Този дял може лесно да се намери с DFS, като се започне от s.

❖ Проблеми от практиката

- Codeforces - масив и операции
- Codeforces - червено-син граф

https://cp-algorithms.com/graph/dinic.html

III. АЛГОРИТЪМ НА ДИНИЦ ЗА МАКСИМАЛЕН ПОТОК -

Съдържание

Проблемът с максималния поток е разгледан е статия Максимален поток - Ford-Fulkerson и Edmonds-Karp, чийто алгоритъм е открит от Юфим Диниц през 1970г.

Алгоритъмът на Dinic решава проблема за максималния поток за време $O(E.V^2)$

❖ Определения

остатъчна мрежа G_R на мрежа G - съдържа по две ребра за всяко ребро (v,u) $\in$ **G**:

- (v,u) с капацитет (пропускателна способност) $C^R_{vu} = C_{vu} - f_{vu}$
- (u,v) с капацитет (пропускателна способност) $C^R_{uv} = f_{vu}$

Следва да се отбележи, че с такава дефиниция в остатъчната мрежа могат да се появят множество ребра, ако първоначалната мрежа има и ребро (v,u) и (u,v)..

Остатъчното ребро може интуитивно да се разбере като мярка за това с колко може да се увеличи потокът по някакво ребро. В действителност, ако по ребро (v,u) тече поток f_{vu} с капацитет C_{vu}, тогава по него потенциално могат да преминат още $C_{vu} - f_{vu}$ единици поток, а в обратната посока могат да се пропуснат f_{vu} единици поток, което ще отмени потока в първоначалната посока.

блокиращ поток на мрежа е такъв поток, **който** за всеки път от s към t **съдържа поне едно ребро, наситено от този поток**. С други думи, ако потокът насити поне едно ребро в дадената мрежа, няма път от s → t, по който безпрепятствено да се увеличи потокът.

Блокиращият поток не е непремепо максимален! Теоремата на Форд-Фулкерсон говори за това, че потокът ще е максимален тогава и само тогава, когато в остатъчната мрежа не е намерен път s→t.

Блокиращият поток не потвърждава за съществуване на път по ребрата, които се появяват в остат.мрежа.

многослойна мрежа на мрежата G е мрежа, изградена по следния начин:

Определят се най-кратките пътища от s до всички останали върхове

и се включват само онези ребра (v,u) на изходната мрежа, които водят от едно ниво към друго по-голямо ниво, т.е. тези, за които lev[v] + 1 = lev [u].

Ще означим разстоянието от s до връх v като lev[v] (ниво на връх v спрямо s)

lev[v] е най-краткият (непретеглен) път от **s до v**, изминат само по ребра с положителен капацитет.

lev[v] е най-краткият път, защото многослойната мрежа включва само ребра (v,u) от изходната мрежа, които водят от едно ниво към друго по-голямо ниво, т.е. lev[v] + 1 = lev [u]

(т.е. разликата в разстоянията не може да надвишава 1, следователно това са най-късите разстояния.

Запазването само на ребрата (v,u), за които lev[v] + 1 = lev [u], игнорира всички ребра, които са изцяло вътре в нивата, както и тези, водещи обратно към предишни нива.

Това води до очевидния извод, че **многослойната мрежа е ациклична и всеки път s→t в нея е най-краткият път в изходната**.

Построяването на многослойната мрежа е много лесно:

- пуска се обхождане в ширина по ребрата на дадената мрежа, отчитайки за всеки връх v неговото lev[v] и вкарване на всички подходящи ребра в многослойната мрежа.

❖ Алгоритъм

– състои се от няколко фази. На всяка фаза:

1. Първо се посторояваа остатъчната мрежа G_R на дадената **G**
2. После чрез bfs се изгражда многослойна мрежа, спрямо остатъчната G_R
3. Търси се произволен блокиращ поток в многослойната мрежа и ако се намери, се добавя към текущия поток, а ако не се намери, процесът приключва.

❖ Доказателство за коректност

Когато алгоритъмът прекрати работа, той е намерил максималния поток, защото:
Алгоритъмът се прекратява, когато не може да намери блокиращ поток в многослойната мрежа.
Това означава, че в многослойната мрежа няма път s → t, което означава, че остатъчната мрежа няма път от s към t, а това означава, че потокът е максимален.

❖ Брой фази

Алгоритъмът приключва с **по-малко от V фази**. За да го докажем, ще докажем 2 леми.

Лема 1. Разстоянията от s до всеки връх не намаляват след всяка итерация,

т.е. $lev_{i+1}[v] \geq lev_i[v]$

Доказателство. Нека сме на фаза i и връх v.
Дължината на всеки най-кратък път P от s до v в остатъчната мрежа GR_{i+1} е = $lev_{i+1}[v]$.
Забележете, че GRi+1 може да съдържа само ребра от GR_i и техни обратни ребра.

Ако в пътя P няма обратни ребра за GR_i, тогава $lev_{i+1}[v] \geq lev_i[v]$, защото P също е път в GRi,
Сега да предположим, че P има поне едно обратно ребро и първото такова е (u,w).

Тогава $lev_{i+1}[u] \geq lev_i[u]$ (поради първия случай).
Реброто (u,w) не принадлежи на GR_i, така че за реброто (w,u), повлияно от блокиращия поток при предишната итерация, означава, че $lev_i[u] = lev_i[w] +1$, както и $lev_{i+1}[w] = lev_{i+1}[u]+1$.

От тези две уравнения и $lev_{i+1}[u] \geq lev_i[u]$ получаваме $lev_{i+1}[w] \geq lev_i[w] +2$,
Сега можем да използваме същата идея за останалата част от пътя.

Лема 2. Разстоянията от s до всеки връх се увеличават след всяка итерация,

т.е. $lev_{i+1}[t] > lev_i[t]$

Доказателство. От предишната лема знаем, че $lev_{i+1}[t] \geq lev_i[t]$

Допускаме, че $lev_{i+1}[t] = lev_i[t]$. Понеже знаем, че остатъчната мрежа GRi+1 може да съдържа само ребра от GR_i и обратни ребра за ребра от GR_i, това означава, че има най-кратък път в GR_i, който не е блокиран от блокиращия поток, което е противоречие.

От тези две леми **заключаваме, че има по-малко от V фази**,
защото lev[t] се увеличава, но не може да стане по-голямо от V-1.

❖ Намиране на блокиращ поток

За да намерим блокиращия поток на всяка итерация, може просто да опитаме да прекарваме потока с DFS от s към t в слоестата мрежа, докато той може да минава.
За да го направим по-бързо, трябва **да премахнем ребрата**, които вече не могат да се използват. **За целта** можем да пазим **показалец във всеки връх, който сочи към следващото ребро, което може да се използва**. Всеки показалец може да бъде преместен най-много E пъти, така че всяка фаза работи за O(E.V).

❖ Сложност

Има по-малко от V фази, така че общата сложност е $O(E.V^2)$

❖ Единични мрежи

Единична мрежа е мрежа, при което всички ребра имат единичен капацитет и за всеки връх, с изключение на s и t входящото или изходящиото ребро е уникално. Точно такъв е случаят с мрежата, която изграждаме, за да разрешим проблема с максималното съвпадение с потоците.

В единичните мрежи алгоритъмът на Dinic работи за време O (E. $\sqrt{V}$ **)**
Нека докажем това:
Първо, всяка фаза работи за време O(E), защото всяко ребро ще бъде разгледано най-много веднъж.
Второ, да предположим, че вече е имало $\sqrt{V}$ фази, след които са намерени всички увеличаващи пътища с дължина $\leq \sqrt{V}$

Нека f текущият поток, а f* е търсеният максимален поток.
Разликата им (f*- f) е поток в остатъчната мрежа G_R. Той има стойност |f*| - |f| и за всяко ребро е 0 или 1.
Можем да декомпозираме тази стойност на |f*| - |f| на брой пътища от s до t и евентуално цикли.
Тъй като мрежата е единична, тези пътища не могат да имат общи върхове,
затова **общият брой върхове в тази мрежа** е $\geq$ (|f*| - |f|)**.** $\sqrt{V}$, но той също е $\leq$ V,
така че определено ще намерим максималния поток за $\sqrt{V}$ итерации.

❖ Имплементация

```
struct FlowEdge
{
    int v, u;
    cap, flow = 0;
    FlowEdg(int v, int u, long long cap) : v(v), u(u), cap(cap) {}
};

struct Dinic
{
    const long long flow_inf = 1e18;
    vector<FlowEdge> edg;         /// списък ребра
    vector<vector<int>> adj;     /// свързан списък на неориентирания граф
    int n, m = 0;                /// n - брой върхове, m - брой ребра
    int s, t;
    vector<int> lev, ptr;
    queue<int> q;

    Dinic(int n, int s, int t) : n(n), s(s), t(t)
    {
        adj.resize(n);
        lev.resize(n);
        ptr.resize(n);
    }

    void add_edge(int v, int u, long long cap)
    {
        edg.emplace_back(v, u, cap);   // право ребро
         edg.emplace_back(u, v, 0);    // обратно ребро
        adj[v].push_back(m);           // № на право ребро (от връх v)
         adj[u].push_back(m + 1); // № на обратно ребро (от връх u)
        m += 2;
    }
```

```
    bool bfs()
    {
        fill(lev.begin(), lev.end(), -1);
        lev[s] = 0;
        q.push(s);

        while (!q.empty())
        {
            int v = q.front(); q.pop();
            for (int id : adj[v])
            {
                if (edg[id].cap - edg[id].flow < 1) continue;
                if (lev[edg[id].u] != -1) continue;
                lev[edg[id].u] = lev[v] + 1;
                q.push(edg[id].u);
            }
        }
        return lev[t] != -1;
    }

    long long dfs(int v, long long pushed)
    {
        if (pushed == 0) return 0;
        if (v == t) return pushed;
        for (int& cid = ptr[v]; cid < (int)adj[v].size(); cid++)
        {
            int id = adj[v][cid];
            int u = edg[id].u;
            if (lev[v] + 1 != lev[u] || edg[id].cap - edg[id].flow < 1)
                                continue;
            long long tr;
            tr = dfs(u, min(pushed, edg[id].cap - edg[id].flow));
            if (tr == 0) continue;
            edg[id].flow += tr;
            edg[id ^ 1].flow -= tr;
            return tr;
        }
        return 0;
    }

    long long flow()
    {   long long f = 0;
        while (bfs())/// while (true)
        {   // fill(lev.begin(), lev.end(), -1);
            // lev[s] = 0;
            // q.push(s);
            /// if (!bfs()) break;
            fill(ptr.begin(), ptr.end(), 0);
            while (long long pushed = dfs(s, flow_inf))
                { f += pushed;  }
        }
        return f;
    }
};                                               // (в) 2014-2020 превод от http://github.com/e-maxx-eng
```

https://cp-algorithms.com/graph/min_cost_flow.html

IV. MINIMUM-COST FLOW - SUCCESSIVE SHORTEST PATH ALGORITHM

(Минимален разход за поток - алгоритъм за последователен най-кратък път)

Съдържание

Дадена е мрежа G състояща се от n върха и m ребра. За всяко ребро (най-общо казано, ориентирани ребра, вижте по-долу) са дадени капацитетът (неотрицателно цяло число) и цената за единица поток по това ребро (цяло число). Маркирани са и източникът s и приемникът t са маркирани.

За дадена стойност K трябва да намерим поток от това количество и измежду всички потоци от това количество трябва да изберем потока с най-ниска цена. Тази задача се нарича **проблем с минимални разходи**.

Понякога задачата се дава малко по-различно: иска се да се намери максималният поток и сред всички максимални потоци да намерим този с най-малко разходи. Това се нарича **проблем с максималния разход** на **минимални разходи**.

И двата проблема могат да бъдат решени ефективно с алгоритъма на последователни по-кратки пътища.

❖ алгоритъм

Този алгоритъм е много подобен на Edmonds-Karp за изчисляване на максималния поток.

Най-прост случай

Първо разглеждаме само най-простия случай, когато графиката е ориентирана и има най-много един ребро между която и да е двойка върхове (например, ако (i, j) тогава е ребро в графиката (j, i) също не може да участва в него).

Позволявам $U_{i j}$ да бъде капацитетът на реброто (i, j) ако това ребро съществува. И нека $C_{i j}$ бъде цената за единица поток по това ребро (i, j)

И накрая нека $F_{i, j}$ бъде потока по реброто (i, j), Първоначално всички стойности на потока са нула.

Ние **модифицираме** мрежата, както следва: за всяко ребро(i, j) добавяме **обратния** ребро (j, i)към мрежата с капацитет $U_{j i}= 0$ и разходите $C_{j i}= -C_{i j}$, Тъй като, според нашите ограничения, ръба(j, i) не беше в мрежата преди, ние все още имаме мрежа, която не е мултиграф (графика с множество ребра). Освен това винаги ще поддържаме състоянието $F_{j i}= -F_{i j}$ вярно по време на стъпките на алгоритъма.

Определяме **остатъчната мрежа** за някакъв фиксиран поток F както следва (точно както в алгоритъма на Ford-Fulkerson): остатъчната мрежа съдържа само ненаситени ребра (т.е. ребра, в които $F_{i j}<U_{i j}$), и остатъчният капацитет на всеки такъв ребро е $R_{i j}=U_{i j}-F_{i j}$,

Сега можем да говорим за **алгоритмите** за изчисляване на потока с минимални разходи. При всяка итерация на алгоритъма намираме най-краткия път в остатъчната графика от s да се t, Противно на Edmonds-Karp, ние търсим най-краткия път по отношение на цената на пътя, вместо броя на ребрате. Ако вече не съществува път, алгоритъмът се прекратява и потокът F е желаният. Ако бъде намерен път, увеличаваме потока по него възможно най-много (т.е. намираме минималния остатъчен капацитет R от пътя и увеличете потока по него и намалете обратните ребра със същото количество). Ако в даден момент дебитът достигне стойността K, след това спираме алгоритъма (обърнете внимание, че при последната итерация на алгоритъма е необходимо да увеличите потока само с такова количество, така че крайната стойност на потока да не надмине K).

Не е трудно да се види, че ако се поставим K до безкрайност, тогава алгоритъмът ще намери максималния поток с минимални разходи. Така че и двете вариации на проблема могат да бъдат решени от един и същ алгоритъм.

Ненасочени графи / мултиграфи

Случаят с неориентиран граф или мултиграф не се различава концептуално от горния алгоритъм. Алгоритъмът ще работи и върху тези графи. Но малко по-трудно е да се приложи.

Едно **ненасочено** ребро (i, j) всъщност е същото като две ориентирани ребра (i, j) и (j, i) със същия капацитет и стойности. Тъй като гореописаният алгоритъм на потока с минимални разходи генерира обратно ребро за всяко насочено ребро, ненасочените ребра се разделят на по 4 насочени ребра и всъщност получаваме **мултиграф**.

Как да се справим с множеството ребра?
Първо - потокът за всяко от множеството ребра трябва да се поддържа отделно.
Второ - при търсене на най-краткия път е необходимо да се вземе предвид, че е важно кое от множеството ребра се използва в пътя. По този начин вместо обичайния масив с предшественици, ние допълнително трябва да съхраняваме номерата на ребрата, от който сме дошли, заедно с прародителите им.
Трето - тъй като потокът се увеличава по определено ребро, е необходимо да се намали потока по обратното ребро. Тъй като имаме множество ребра, трябва да съхраняваме номера на обратното ребро на всяко ребро.
Няма други препятствия с насочени графи или мултиграфии.

Сложност

Аналогично на анализа на алгоритъма на Edmonds-Karp получаваме следната оценка: $O(n, m) \cdot T(n, m)$, където $T(n,m)$ е времето, необходимо за намиране на най-краткия път в граф с n върха и m ребра.

Ако това търсене се извърши с [алгоритъма на Dijkstra](), тогава сложността на алгоритъма с минимални разходи ще стане $O(n^3m)$.
Ние обаче работим с ребра с отрицателна цена. Така че Dijkstra е неприложим, ако не е модифициран.
Но вместо това, можем да използваме [алгоритъма на Bellman-Ford](). С него сложността става $O(n^2m^2)$

❖ Имплементация

Ето една реализация, използваща [алгоритъма SPFA(Shortest Path Faster Algorithm)] за най-простия случай.

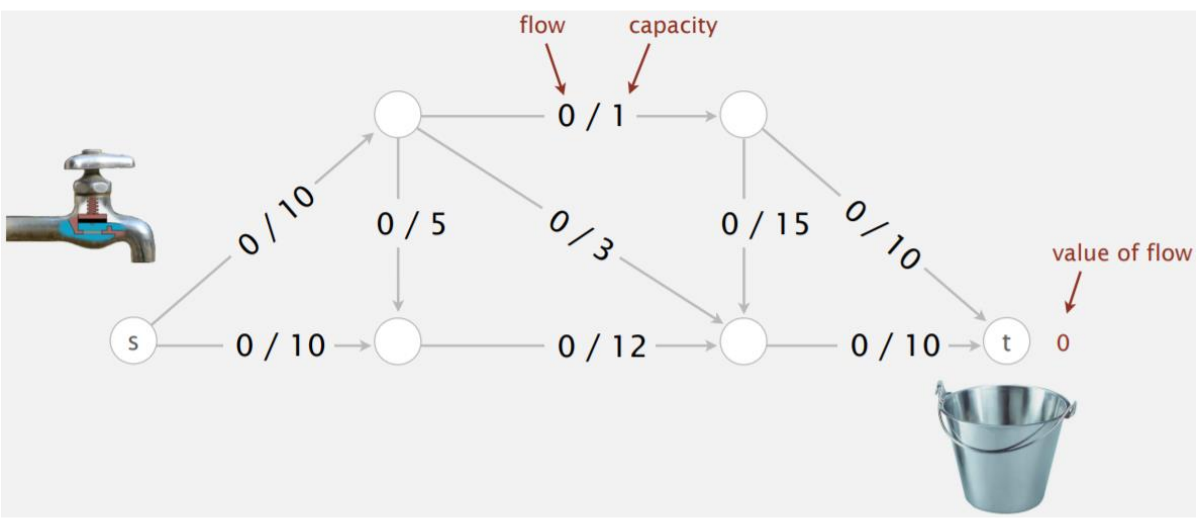
```
struct Edge
{
    int from, to, capacity, cost;
};
vector<vector<int>> adj, cost, capacity;

const int INF = 1e9;

int min_cost_flow(int N, vector<Edge> edges, int K, int s, int t)
{
    adj.assign(N, vector<int>());
    cost.assign(N, vector<int>(N, 0));
    capacity.assign(N, vector<int>(N, 0));
    for (Edge e : edges)
    {
        adj[e.from].push_back(e.to);
        adj[e.to].push_back(e.from);
        cost[e.from][e.to] = e.cost;
        cost[e.to][e.from] = -e.cost;
        capacity[e.from][e.to] = e.capacity;
    }
    int flow = 0;
    int cost = 0;
    vector<int> d, p;
    while (flow < K)
    {
        shortest_paths(N, s, d, p);
        if (d[t] == INF) break;

        // find max flow on that path
        int f = K - flow;
        int cur = t;
        while (cur != s)
        {
            f = min(f, capacity[p[cur]][cur]);
            cur = p[cur];
        }

        // apply flow
        flow += f;
        cost += f*d[t];
        cur = t;
        while (cur != s)
        {
            capacity[p[cur]][cur] -= f;
            capacity[cur][p[cur]] += f;
            cur = p[cur];
        }
    }

    if (flow < K) return -1;
    else return cost;
}
```

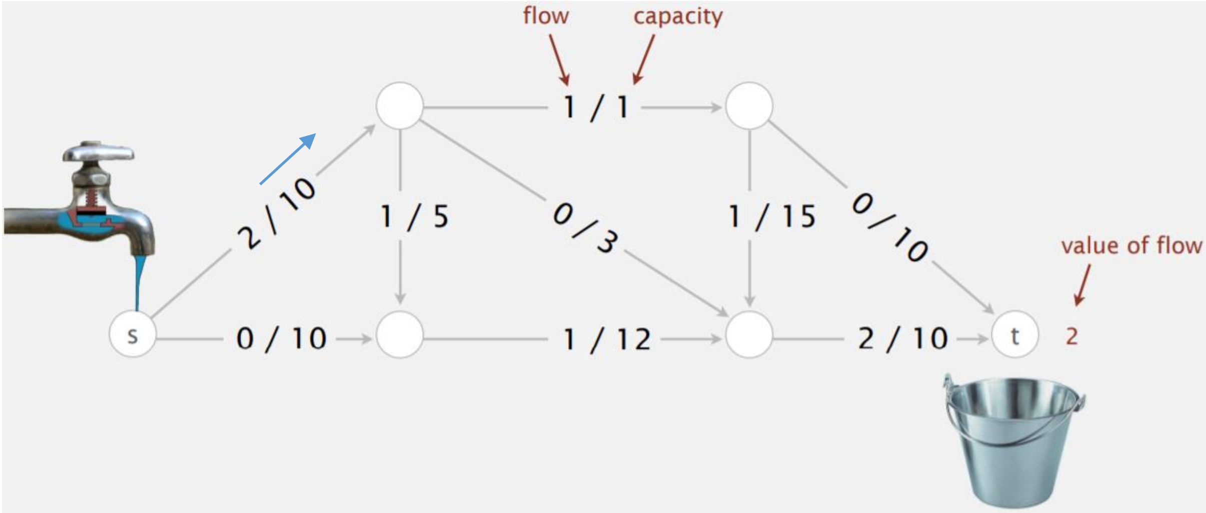
```
void shortest_paths(int n, int v0, vector<int>& d, vector<int>& p)
{
    d.assign(n, INF);
    d[v0] = 0;
    vector<bool> inq(n, false);
    queue<int> q;
    q.push(v0);
    p.assign(n, -1);

    while (!q.empty())
    {
        int u = q.front();
        q.pop();
        inq[u] = false;
        for (int v : adj[u])
        {
            if (capacity[u][v] > 0 && d[v] > d[u] + cost[u][v])
            {
                d[v] = d[u] + cost[u][v];
                p[v] = u;
                if (!inq[v])
                {
                    inq[v] = true;
                    q.push(v);
                }
            }
        }
     }
}
```

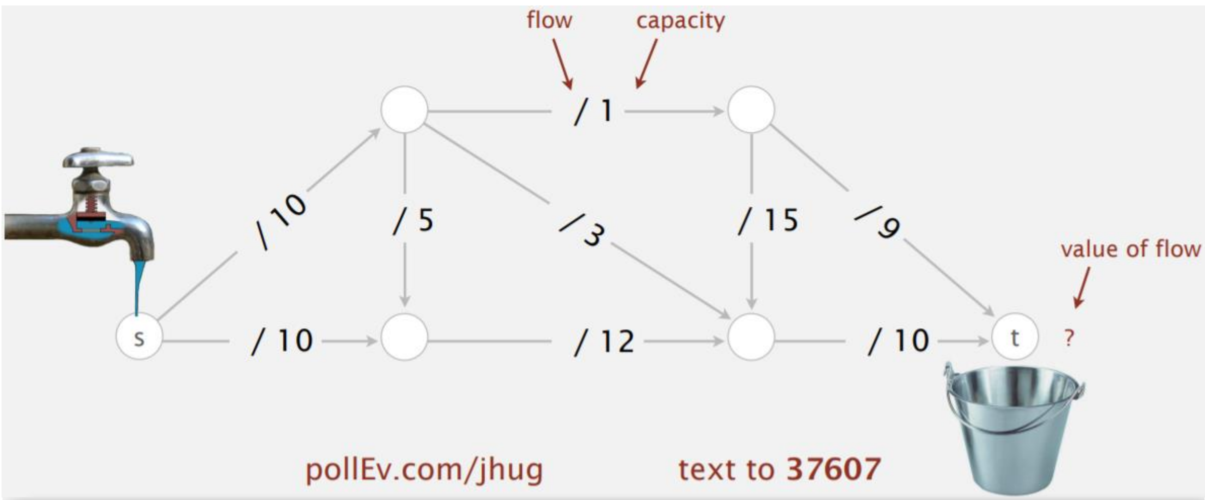
(в) 2014-2020 превод от http://gith

- https://algs4.cs.princeton.edu/lectures/keynote/64MaxFlow.pdf

V.



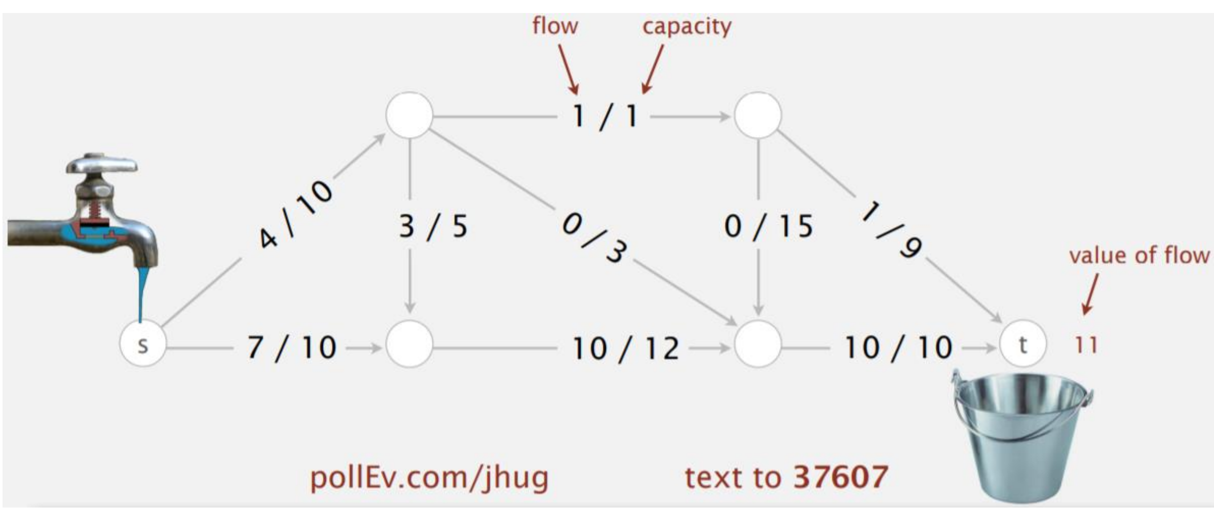




Q: What is the value of the max flow?

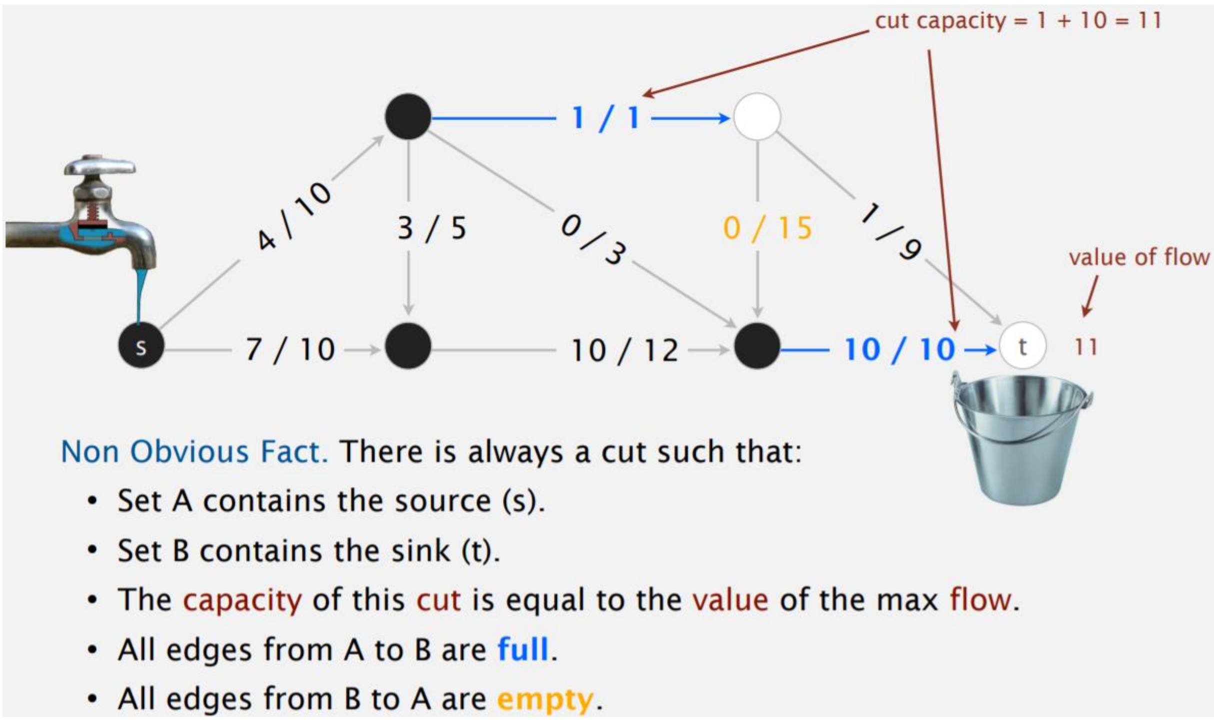
A. 20	[191071]	D. 11	[170148]
B. 19	[175058]	E. 10	[170215]
C. 16	[170059]		

Extra: Find a cut whose total capacity equals the max flow.



Q: What is the value of the max flow?
D. 11

Extra: Find a cut whose total capacity equals the max flow.



❖ https://algs4.cs.princeton.edu/lectures/keynote/64MaxFlow.pdf