Introduction to Algorithms

Science Honors Program (SHP)

Session 6

Benjamin Rubio and Christian Lim

Saturday, April 6, 2024

Slide deck in github

- You may get to the link by:
  - https://github.com/yongwhan/
  - => yongwhan.github.io
  - => columbia
  - => shp
  - => session 6 slide

Overview

- **Flows (con't)**
- Break #1 (5-minute)
- **Games**
- Break #2 (5-minute)
- **Ad Hoc**
  - Interactive
  - Constructive

CodeForces Columbia SHP Algorithms Group

- While I take the attendance, please join the following group: **https://codeforces.com/group/lfDmo9iEr5**

- We will be using them in the last portion of the session today!

Attendance

- Let's take a quick attendance before we begin!

Network Flow: Real-life Example

- We are given a directed graph, where each vertex represents a city and each directed edge represents a one-way road from one city to another.
- Suppose we want to send trucks from city s to city t and the capacity of each directed edge represents the number of trucks that can go on that road every hour.

Photo Credit

Network Flow: Real-life Example

- We are given a directed graph, where each vertex represents a city and each directed edge represents a one-way road from one city to another.
- Suppose we want to send trucks from city s to city t and the capacity of each directed edge represents the number of trucks that can go on that road every hour.

- **What is the maximum number of trucks that we can send from s to t every hour?**

Flow Network

- A **network** is a directed graph G with vertices V and edges E combined with a function c, which assigns each edge e a non-negative integer value, the **capacity** of e.
- Such a network is called a **flow network**, if we additionally label two vertices, one as **source** and the other as **sink**.

Flow Network

- A **flow** in a flow network is function f, that again assigns each edge e a non-negative integer value, namely the flow. The function has to fulfill the following conditions:
  - The flow of an edge cannot exceed the capacity: $f(e) \leq c(e)$.
  - The sum of the incoming flow of a vertex u has to be equal to the sum of the outgoing flow of u (except in the source and sink vertices): $\Sigma_v f((v, u)) = \Sigma_w f((u, w))$.
- The source vertex s only has outgoing flows.
- The sink vertex t only has incoming flows.
- $\Sigma_v f((v, t)) = \Sigma_w f((s, w))$.

Flow Network

- The value of the flow of a network is the sum of all the flows that get produced in the source s, or equivalently to the sum of all the flows that are consumed by the sink t.
- Formally, val(f), the **value** of a flow f, is defined as:

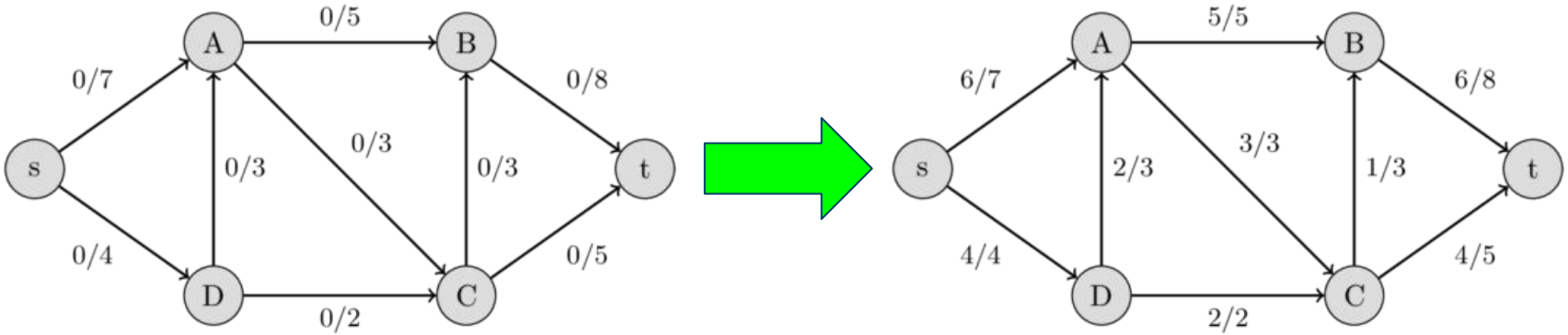
$$\Sigma_{e \text{ out of } s} f(e) - \Sigma_{e \text{ in to } s} f(e).$$

- A **maximum flow** is a flow with the maximum possible value of val(f).

- **Problem: Find this maximum flow of a flow network!**

Ford-Fulkerson Method: Example

- The first value of each edge represents the flow, which is initially 0, and the second value represents the capacity.



Ford-Fulkerson Method (1956)

- A **residual capacity** of an directed edge is the capacity minus the flow.
  - For each edge (u, v), we can create a reverse edge (v, u) with capacity 0 such that f((v, u)) = -f((u, v)).
  - This also defines the residual capacity for all the reversed edges.
- We can create a **residual network** from all these edges, which is just a network with the same vertices and edges, but we use the residual capacities as capacities.

Ford-Fulkerson Method (con't)

- We set the flow of each edge to 0.
- We look for an **augmenting path** from s to t.
  - An augmenting path is a simple path in the residual graph by always taking the edges whose residual capacity is **positive**.
- If such a path is found then we can increase the flow along these edges.
  - Let C be the smallest residual capacity of the edges in the path. Then we increase the flow by updating f((u, v)) += C and f((v, u)) -= C for every edge (u, v) in the path.
- Else, terminate! (A flow found is maximum).

Ford-Fulkerson Method: Augmenting Path?

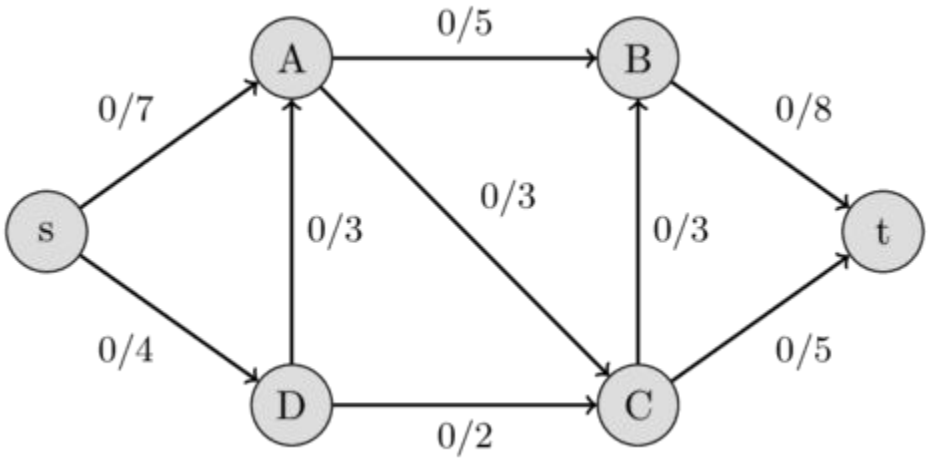
- Ford-Fulkerson method **does not specify** how to find an augmenting path. Possible approaches are using *Depth-First Search (DFS)* or *Breadth-First Search (BFS)*, both of which work in **O(E)**.

Ford-Fulkerson Method: Augmenting Path? (con't)

- **Integral** capacities
  - For each augmenting path, the flow of the network increases by at least 1. So, the complexity of Ford-Fulkerson is O(EF) where F is the maximum flow of the network (but, usually *much faster* in practice)!
- **Rational** capacities
  - The algorithm will terminate but the complexity is not bounded.
- **Irrational** capacities
  - The algorithm might never terminate and might not even converge to the maximum flow.

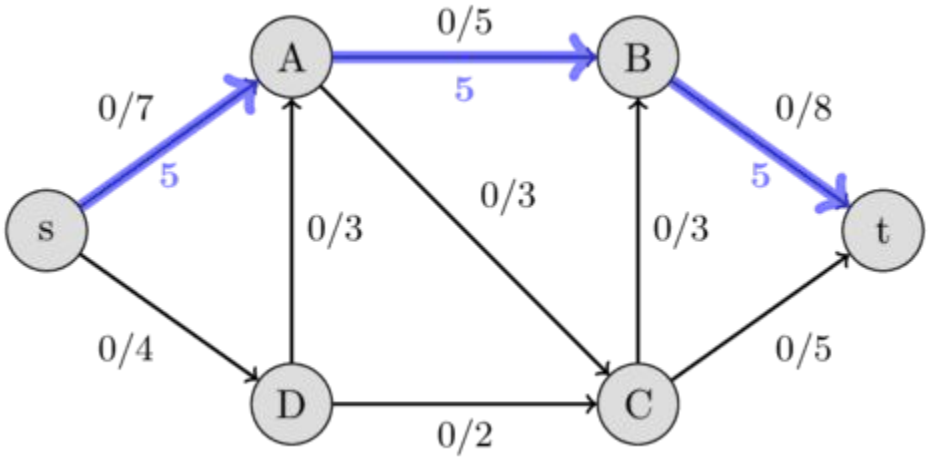
Ford-Fulkerson Method: Example

flow = 0



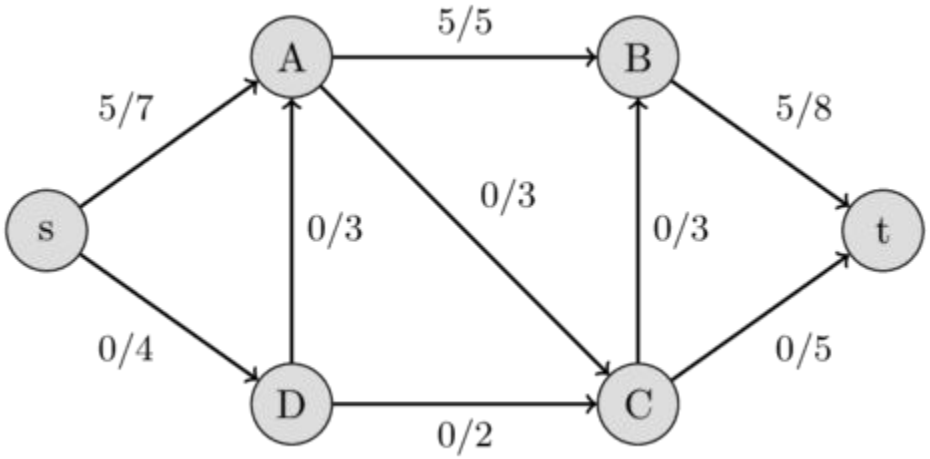
Ford-Fulkerson Method: Example

flow = 0



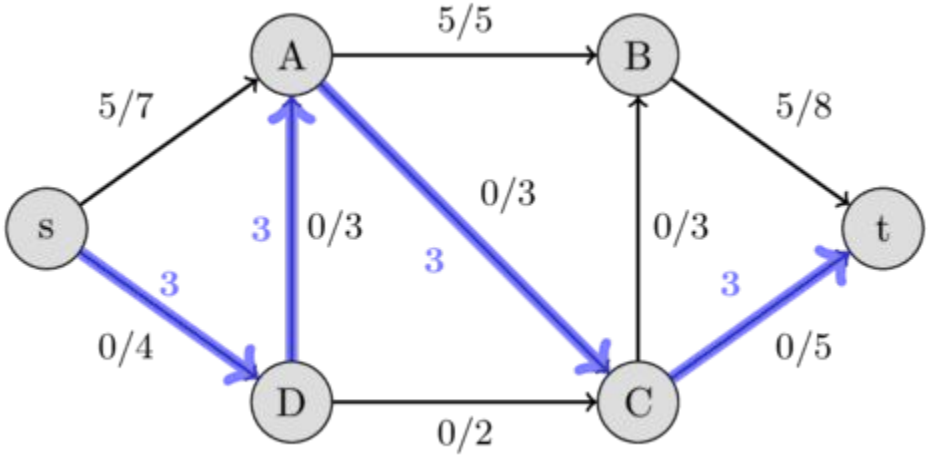
Ford-Fulkerson Method: Example

flow = 5



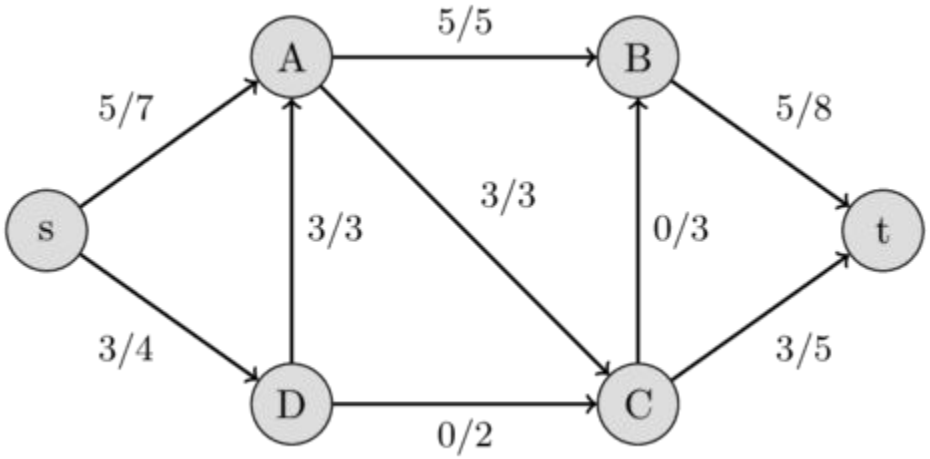
Ford-Fulkerson Method: Example

flow = 5



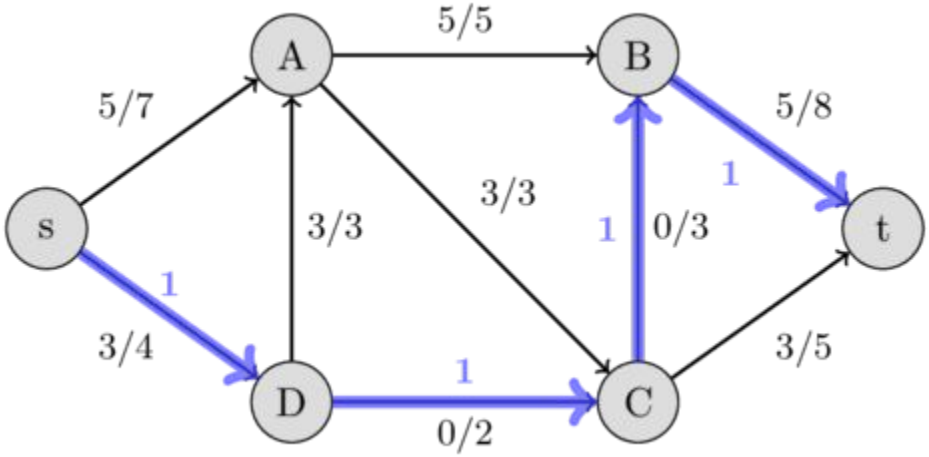
Ford-Fulkerson Method: Example

flow = 8



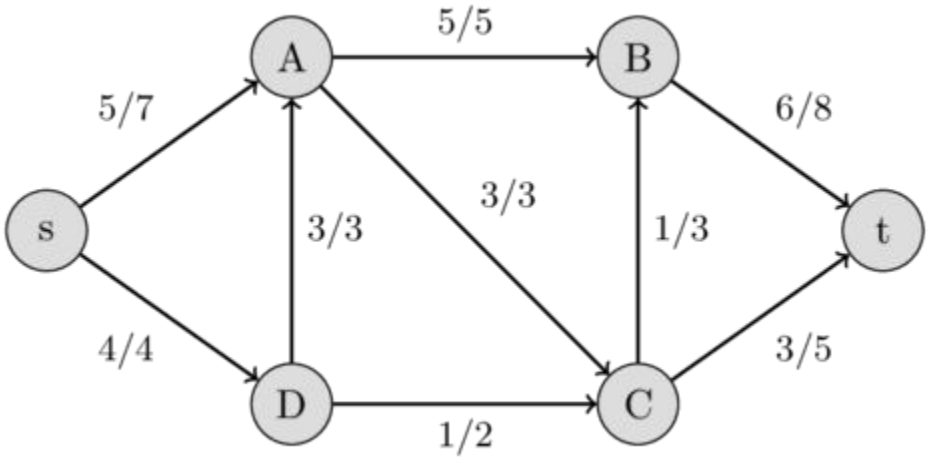
Ford-Fulkerson Method: Example

flow = 8



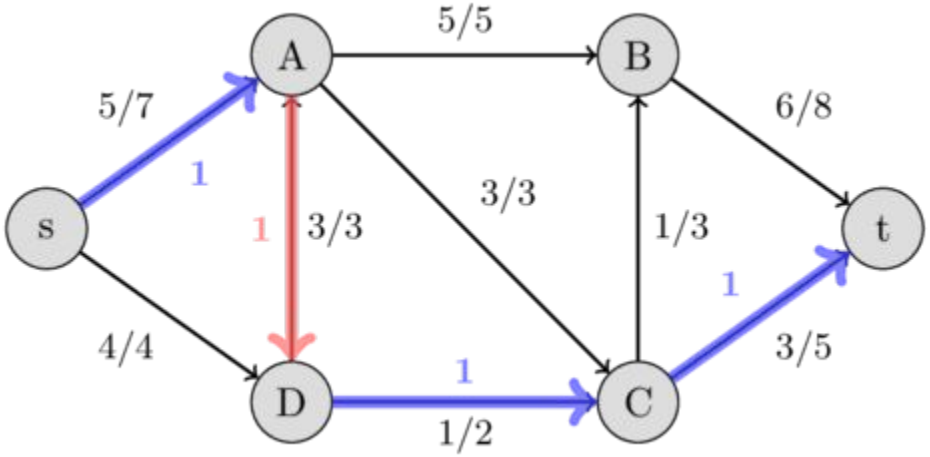
Ford-Fulkerson Method: Example

flow = 9



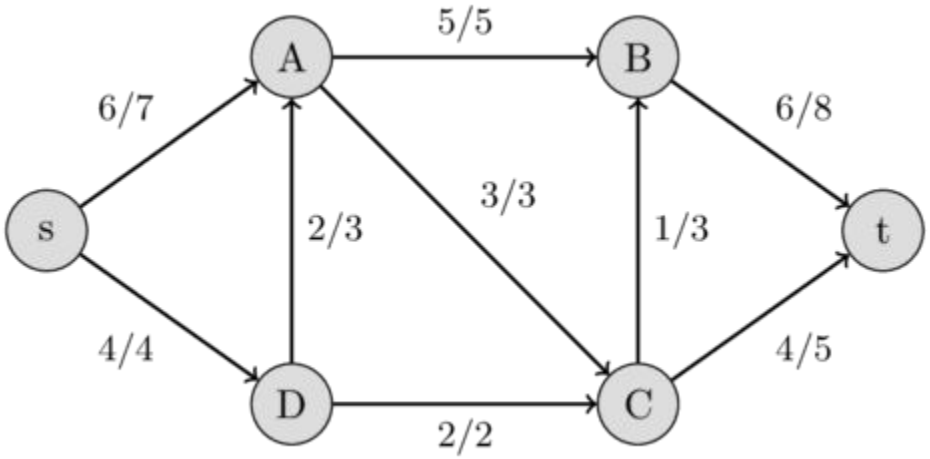
Ford-Fulkerson Method: Example

flow = 9



Ford-Fulkerson Method: Example

flow = **10**



Edmonds-Karp Algorithm (1972)

- **Edmonds-Karp algorithm** is just an implementation of the Ford-Fulkerson method that uses BFS for finding augmenting paths.

- First published by Yefim Dinitz in 1970; later, independently published by Jack Edmonds and Richard Karp in 1972.

- The complexity can be given *independently* of the maximum flow. The algorithm runs in **$O(VE^2)$** time (yes! even for irrational capacities).

Edmonds-Karp Algorithm (1972)

- **Intuitions**
  - Every time we find an augmenting path. one of the edges becomes saturated and the distance from the edge to s will be longer if it appears again in an augmenting path later.
  - The path length is bounded by V.

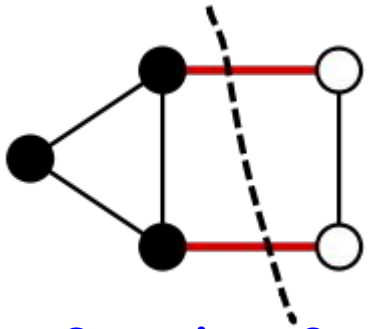
Integral Flow Theorem

- Suppose a given graph has each of its capacity integral. Then, there exists a maximum flow f where every flow value f(e) is an integer.

s-t cut and its capacity

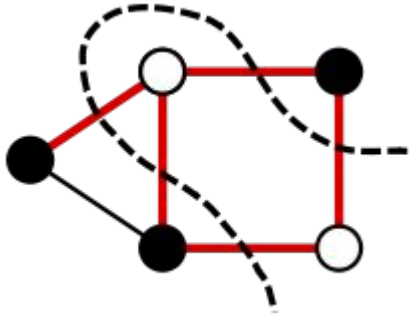
- An **s-t cut** is a partition (A, B) of the vertices with s ∈ A and t ∈ B.
- The capacity of a cut, **cap(A, B)**, is the sum of capacities of the edges from A to B.

Minimum Cut



Capacity = 2

Maximum Cut



Capacity = 5

Flow Value Lemma

- Let f be any flow and let (A, B) be any cut. Then,

$$\text{val}(f) = \Sigma_{\text{e out of A}} f(e) - \Sigma_{\text{e in to A}} f(e).$$

Flow Weak Duality Lemma

- Let f be any flow and let (A, B) be any cut. Then,

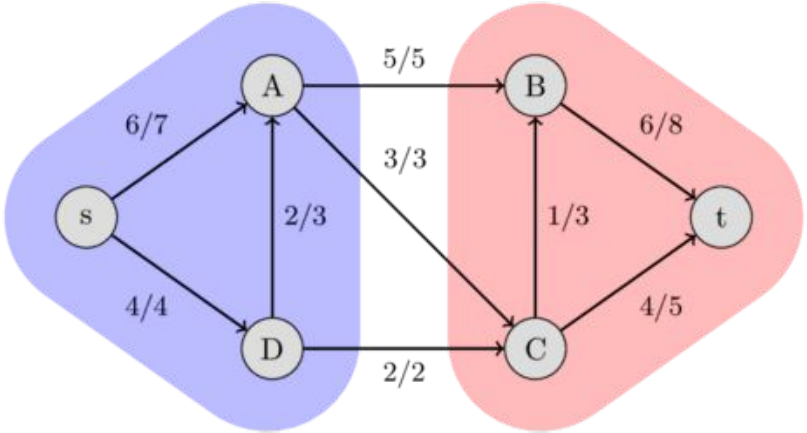
$$\text{val}(f) \leq \text{cap}(A, B).$$

Flow Certificate of Optimality

- Let f be a flow and let (A, B) be any cut. If val(f) = cap(A, B) then f is a maximum flow and (A, B) is a minimum cut.

Max-flow Min-cut Theorem

- The maximum value of an s-t flow is equal to the minimum capacity over all s-t cuts.



It is quite rare to actually use Ford Fulkerson in CP!

- But, without **Ford Fulkerson** and an intuition behind an **augmenting path**, it is quite difficult to understand **Dinic**, which is why we covered it first!

- Now, let's dive into **Dinic**!

Definitions

- A **residual network** G^R of network G is a network which contains two edges for each edge $(v, u) \in G$:
  - (v, u) with capacity $c^R_{vu} = c_{vu} - f_{vu}$
  - (u, v) with capacity $c^R_{uv} = f_{vu}$

Definitions

- A **blocking flow** of some network is such a flow that every path from s to t contains at least one edge which is saturated by this flow.
  - Note that a blocking flow is not necessarily maximal.

- A **layered network** of a network G is a network built in the following way:
  - For each vertex v we calculate level[v]: the shortest path (unweighted) from s to this vertex using only edges with positive capacity.
  - We keep only those edges (v, u) for which level[v]+1 = level[u].
  - Obviously, this network is acyclic.

Dinic's Algorithm

- The algorithm consists of several phases.
- On each phase:
  - Construct the layered network of the residual network of G.
  - Find an arbitrary blocking flow in the layered network and add it to the current flow.

Number of Phases

- The algorithm terminates in less than V phases.

Finding Blocking Flow

- In order to find the blocking flow on each iteration, we may simply try pushing flow with DFS (Depth-First Search) from s to t in the layered network while it can be pushed.
- In order to do it more efficiently, we must remove the edges which cannot be used to push anymore.
- We can keep a **pointer** in each vertex which points to the next edge which can be used.

Finding Blocking Flow (con't)

- A single DFS run takes O(k + V) time, where k is the number of pointer advances on this run.
- Over all runs, a number of pointer advances cannot exceed E.
- A total number of runs would not exceed E, as every run saturates at least one edge.
- So, a total running time of finding a blocking flow is **O(VE)**.

Dinic's Complexity

- Since there are less than V phases, the total time complexity is **$O(V^2E)$**.

Implementations in C++: *FlowEdge* struct

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

Implementations in C++: *Dinic* struct

```
struct Dinic {
    const long long flow_inf = 1e18;
    vector<FlowEdge> edges;
    vector<vector<int>> adj;
    int n, m = 0;
    int s, t;
    vector<int> level, ptr;
    queue<int> q;
    Dinic(int n, int s, int t) : n(n), s(s), t(t) {
        adj.resize(n); level.resize(n); ptr.resize(n);
    }
```

Implementations in C++: *Dinic* struct

```
    void add_edge(int v, int u, long long cap);
    bool bfs();
    long long dfs(int v, long long pushed);
    long long flow();
};
```

Implementations in C++: *flow* function

```
long long flow() {
  long long f = 0;
  while (true) {
    fill(level.begin(), level.end(), -1);
    level[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;
}
```

Implementations in C++: *bfs* function

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

Implementations in C++: *dfs* function

```
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], u = edges[id].u;
    if (level[v] + 1 != level[u] ||
        edges[id].cap - edges[id].flow < 1)
      continue;
```

Implementations in C++: *dfs* function

```
    long long tr = dfs(u,
      min(pushed, edges[id].cap - edges[id].flow));
    if (tr == 0) continue;
    edges[id].flow += tr;
    edges[id ^ 1].flow -= tr;
    return tr;
  }
  return 0;
}
```

Implementations in C++: *add_edge* function

```
void add_edge(int v, int u, long long cap) {
    edges.emplace_back(v, u, cap);
    edges.emplace_back(u, v, 0);
    adj[v].push_back(m);
    adj[u].push_back(m + 1);
    m += 2;
}
```

Here is a full implementation in one page

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

Practice Problems

- **Network Flow**
  - https://codeforces.com/contest/498/problem/c
  - https://codeforces.com/contest/1288/problem/f
  - https://codeforces.com/problemset/problem/1184/B2
  - https://codeforces.com/problemset/problem/1748/E
  - https://codeforces.com/problemset/problem/1252/L
  - https://codeforces.com/problemset/problem/491/C
  - https://codeforces.com/problemset/problem/717/G
  - https://codeforces.com/problemset/problem/1354/F
  - https://codeforces.com/problemset/problem/1572/D

References

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


BREAK #1

Games

- The Basics: Main Idea
- The Game of Nim
- Composite Games - Grundy Numbers

The Basics: Main Idea

- All terminal positions are losing.
- If a player is able to move to a losing position then he is in a winning position.
- If a player is able to move only to the winning positions then he is in a losing position.

The Basics: Implementation

```
boolean isWinning(position pos) {
    moves[] // possible positions to which
            // I can move from the position pos
    for (all x in moves)
        if (!isWinning(x)) return true;
    return false;
}
```

Exercise: LongLongNim

- https://community.topcoder.com/stat?c=problem_statement&pm=6856

Exercise: Solution Idea

- Iterate over the possible states and decide if they are winning for the second player.

Exercise #1: Solution Idea

- Iterate over the possible states and decide if they are winning for the second player.
- **Observation:** All moves remove at most 22 coins, therefore we only care about at most the previous 22 states.

Exercise #1: Solution Idea

- Iterate over the possible states and decide if they are winning for the second player.
- **Observation:** All moves remove at most 22 coins, therefore we only care about at most the previous 22 states.
- This is still not enough, complexity O(maxN * maxMove) can be too high.

Exercise #1: Solution Idea

- Iterate over the possible states and decide if they are winning for the second player.
- **Observation:** All moves remove at most 22 coins, therefore we only care about at most the previous 22 states.
- This is still not enough, complexity O(maxN * maxMove) can be too high.
- **Observation:** As we only care at most about the previous 22 states, there are 2^22 ~ (4 * 10^6) possible previous combinations, much lower than the maximum maxN. **At some point we repeat.**
- **Final Idea:** Take advantage of repetitions to skip cycles.

The Game of Nim: Problem Statement

- Problem Statement: There are n piles of coins. When it is a player's turn he chooses one pile and takes at least one coin from it. If someone is unable to move he **loses** (so the one who removes the last coin is the **winner**).

The Game of Nim: Main Idea

- Let n_1, ..., n_k be the pile sizes.
- Crucial observation: It is a **losing** position for the current player

if and only if

$$n_1 \text{ xor } \ldots \text{ xor } n_k = 0.$$

WHY?

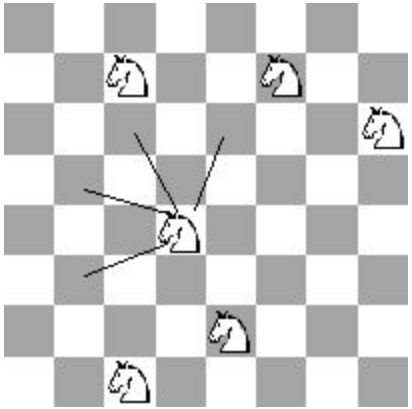
The Game of Nim: Main Idea

- From the losing positions we can move only to the winning ones:
  - if xor of the sizes of the piles is 0 then it will be changed after our move (at least one 1 will be changed to 0, so in that column will be odd number of 1s).

- From the winning positions it is possible to move to at least one losing:
  - if xor of the sizes of the piles is not 0 we can change it to 0 by finding the leftmost column where the number of 1s is odd, changing one of them to 0 and then by changing 0s or 1s on the right side of it to gain even number of 1s in every column.

Composite Games - Grundy Numbers

- Problem Statement: N x N chessboard with K knights on it. Unlike a knight in a traditional game of chess, these can move only as shown in the picture below (so the sum of coordinates is decreased in every move).



Composite Games - Grundy Numbers

```
int grundyNumber(position pos) {
    moves[] // possible positions to which
            // I can move from pos
    set s;
    for (all x in moves)
        insert into s grundyNumber(x);
    // return the smallest non-negative
    // integer not in the set s;
    int ret = 0;
    while (s.contains(ret)) ret++;
    return ret;
}
```

Composite Games - Grundy Numbers

- There can be more than one knight on the same square at the same time. Two players take turns moving and, when it is a player's, turn he chooses one of the knights and moves it. A player who is not able to make a move is declared the loser.
- This is the same as if we had K chessboards with exactly one knight on every chessboard. This is the ordinary sum of K games and it can be solved by using the **grundy numbers**. We assign grundy number to every subgame according to which size of the pile in the Game of Nim it is equivalent to. When we know how to play Nim we will be able to play this game as well.

Composite Games - Grundy Numbers

Why is the pile of Nim equivalent to the subgame if its size is equal to the grundy number of that subgame?

- If we decrease the size of the pile in Nim from A to B, we can move also in the subgame to the position with the grundy number B. (Our current position had grundy number A so it means we could move to positions with all smaller grundy numbers, otherwise the grundy number of our position would not be A.)
- If we are in the subgame at the position with grundy number 0, by moving from that we will get to a position with a grundy number higher than 0. Because of that, from such a position it is possible to move back to 0. By doing that we can nullify every move from the position from grundy number 0.

Composite Games - Grundy Numbers

Example of **Grundy Numbers** for the knight problem on a 8x8 board.

0	0	1	1	0	0	1	1
0	0	2	1	0	0	1	1
1	2	2	2	3	2	2	2
1	1	2	1	4	3	2	3
0	0	3	4	0	0	1	1
0	0	2	3	0	0	2	1
1	1	2	2	1	2	2	2
1	1	2	3	1	1	2	0

Practice Problems

- **Games**
  - https://community.topcoder.com/stat?c=problem_statement&pm=6239
  - https://community.topcoder.com/stat?c=problem_statement&pm=7424&rd=10662
  - https://community.topcoder.com/stat?c=problem_statement&pm=2987&rd=5862

References

- Game theory section from https://cp-algorithms.com/
- https://usaco.guide/adv/game-theory?lang=cpp

- *Winning Ways for Your Mathematical Plays*


BREAK #2

Interactive Problems

- The input data given to your program may not be predetermined but is built specifically for your solution.

- Jury writes a special program: **interactor**, such that its output is transferred to the input of your solution and the output of your program is sent to interactor's input.
  - Your solution and the interactor exchange the data and may decide on what to print based on the "history of communication".

Make sure to flush!

- When you write the solution for the interactive problem, it is important to keep in mind that if you output some data it is possible that this data is first placed to some internal buffer and may be not directly transferred to the interactor.
  - **You have to use flush operation each time you output some data.**
- Typically:
  - If you use “cout”, make sure to use **“endl”**; do not disable cin.tie/cout.tie.
  - If you use “printf”, make sure to use **“fflush(stdout)”**.

Warm-Up Problem: Guess the number

- In this problem there is some hidden number and you have to interactively guess it.
- You can make queries to the testing system. Each query is one integer from 1 to 1000000. There are two different responses:
  - “<” (without quotes), if (hidden number) < (your query);
  - “>=” (without quotes), if (hidden number) >= (your query);
- Print “! x” to provide x as the answer.
- Your program is allowed to make **no more than 25 queries** (not including printing the answer).

Warm-Up Problem: Guess the number (con't)

- **Input**: Use standard input to read the responses to the queries. The input will contain responses to your queries: "<" and ">=". When your program will guess the number print "! x", where x is the answer and terminate your program. The testing system will allow you to read the response on the query only after your program print the query for the system and perform flush operation.

Warm-Up Problem: Guess the number (con't)

- **Output**: To make the queries your program must use standard output. Your program must print the queries, one query per line. After printing each line your program must perform operation flush. The response to the query will be given in the input file after you flush output. In case your program guessed the number x, print "! x", where x is the answer and terminate your program.

Warm-Up Problem: Guess the number (con't)

- Any idea?

Warm-Up Problem: Guess the number (con't)

```
int main() {
  int l=1, r=1e6;
  while(l!=r) {
    int m=(l+r+1)/2;
    cout<<m<<endl;
    string s; cin>>s;
    if(s=="<") r=m-1;
    else l=m;
  }
  cout<<"! "<<l<<endl;
  return 0;
}
```

Exercise #1: [1700] Tree Diameter

- https://codeforces.com/problemset/problem/1146/C

Exercise #1: Solution Idea

- **BFS/DFS**

- The standard algorithm for finding the diameter of a tree is:
  - find the farthest distance from node 1;
  - find the farthest distance from that node.

- We can apply this idea in this problem!

Exercise #2: [2100] Guess The Maximums

- https://codeforces.com/problemset/problem/1363/D

Exercise #2: Solution Idea

- **Binary Search**

- maximum of the array is the password integer for all but at most 1 position.
- find the subset (if the maximum is in a subset) in which the maximum exists using **binary search**.
- query the answer for this subset separately.
- for all the subsets, the answer is the maximum for the whole array.

Exercise #3: [2200] In Search of Truth (Easy Version)

- https://codeforces.com/problemset/problem/1840/G1

Exercise #3: Solution Idea

- **Ad Hoc**

- Do “+ 1” query 999 times.
- Do “+ 1000” query 1000 times.

Constructive Problems

- Constructive problems are **ad-hoc**, so they are difficult to solve even if you have done lots of constructive exercises before.
- These problems are remarkable in that solutions rarely use some template algorithm (e.g., segment tree, dynamic programming, etc).
- The best predictor of your ability to solve constructive problems is going to be having a strong mathematical intuition. **It is much harder to train**.
- A common strategy for these problems is to **manually solve a few inputs** first. Then, build insight from that to construct a deterministic algorithm that provably works for all inputs.

Warm-Up Problem

- Let $1 \le N \le 100{,}000$ and $1 \le K \le \log_2(N)$ be fixed.

- Imagine performing a binary search on the values 1, ..., N.

- Give a value X, $1 \le X \le N$, which would be found after exactly K steps of the binary search.

Warm-Up Problem (con’t)

- Any idea?

Warm-Up Problem (con’t)

- Just do the usual/vanilla implementation of the binary search!

Exercise #1: [1600] Fixed Point Guessing

- https://codeforces.com/contest/1698/problem/D

Exercise #1: Solution Idea

- **binary search**!

Exercise #2: [2000] Matching vs Independent Set

- https://codeforces.com/problemset/problem/1198/C

Exercise #2: Solution Idea

- **Greedy**

- Let's try to take edges to matching greedily in some order.
- If we can add an edge to the matching (both endpoints are not covered), then we take it.
- **~~It is easy to see~~ that all vertices not covered by the matching form an independent set; otherwise, we would add an edge to the matching.**
- Either matching or independent set has size at least n.

Exercise #3: [2000] Game with modulo

- https://codeforces.com/problemset/problem/1103/B

Exercise #3: Solution Idea

- **powers of two and binary search**
- Let's ask (0, 1), (1, 2), (2, 4), (4, 8), ..., (2^{29}, 2^{30}). Let's find the first pair in this list with the answer “x”.
  - This pair exists and it will happen for the first pair (l_0,r_0) that satisfy the inequality $l_0 < a \le r_0$. We can find this pair using at most **31** questions.
- Now, if we ask (l_0, x) for some $l_0 < x \le r_0$, we will get the answer “y” if $x < a$ and the answer “x” otherwise. Binary search! Here we will use at most **29** questions.
- We need to ask at most 31+29 = 60 questions!

References

- **Interactive**
  - https://codeforces.com/blog/entry/45307
  - https://codeforces.com/problemset?order=BY_RATING_ASC&tags=interactive

- **Constructive**
  - https://codeforces.com/blog/entry/80317
  - https://codeforces.com/problemset/page/1?order=BY_RATING_ASC&tags=constructive+algorithms

Again, CodeForces Columbia SHP Algorithms Group

- Please join the following group: **https://codeforces.com/group/lfDmo9iEr5**

On April 6!

- On April 13, we will cover:
  - **Combinatorics**
  - **Number Theory**

Slide Deck

- You may **always** find the slide decks from:
  - https://github.com/yongwhan/yongwhan.github.io/blob/master/columbia/shp


THANK YOU