# Combinatorial Optimization

Dr. R.A. Sitters (SBE) - coordinator
Prof. dr. J.A.S. Gromicho (SBE)

meetings outside the course times are on appointment only

VU VRIJE UNIVERSITEIT AMSTERDAM

LOOKING FURTHER

# Rene Sitters

Dr. René Sitters is the coordinator of the Combinatorial Optimization course.

Specialized in Combinatorial Optimization problems: algorithms and complexity.

Winner of:

- Gijs de Leve prize (for the best PhD dissertation on the mathematics of OR)
- Veni Innovational Research NWO

# Joaquim Gromicho

Professor of Applied Optimization in Operations Research at the SEB.

Scientific & education Officer at ORTEC, an international company specialized in mathematical optimization.

Editor in Chief of STAtOR, the ‘glossy’ of the Netherlands Society of Statistics and Operations Research.

Member of the coordinating committee of the EURO working group on the Practice of OR.

# Today's course

Exposition of the structure of the course and the grading.

All this, and more, can be found on the course manual (on canvas).

Description of the bibliography.

Introduction to combinatorial optimization with the focus on theorems, proofs and algorithms.

Today's course considers only graph problems, therefore a gentle introduction to graphs is also given.

Important learning objectives:

- Proving optimality of algorithms and analyzing their computational effort.
- Understanding that small differences in related algorithms influence their computational effort.
- Understand that implementation details influence computational effort.

# First part: theory in period 4

| Days | theme | lecture | tutorial |
|---|---|---|---|
| February 3 and 6 | Introduction, graphs, paths and trees, flows | Gromicho | Sitters |
| February 10 and 13 | Complexity classes and reductions | Gromicho | Sitters |
| February 17 and 20 | Dynamic Programming | Gromicho | Gromicho |
| February 24 and 27 | Scheduling | Sitters | Sitters |
| March 2 and 5 | Approximation algorithms | Sitters | Sitters |
| March 9 and 12 | Material relevant for the case | Gromicho | Sitters |

# Course bibliography: lectures

Course notes by René **Sitters**. These are all relevant! Will be available per week.

Book ‘Algorithms’ by **Dasgupta**, Papadimitriou, and Vazirani ISBN-13: 978-0073523408, last free version can be found on Blackboard

Book ‘Scheduling Theory, Algorithms, and Systems’ by **Pinedo**, ISBN-13: 978-0130281388, instructions how to access online via the VU library on blackboard.

Material relevant for the case. Will be available toward the end of the first block.

# The main book

Algorithms

*Sanjoy Dasgupta, Christos Papadimitriou & Umesh Vazirani*

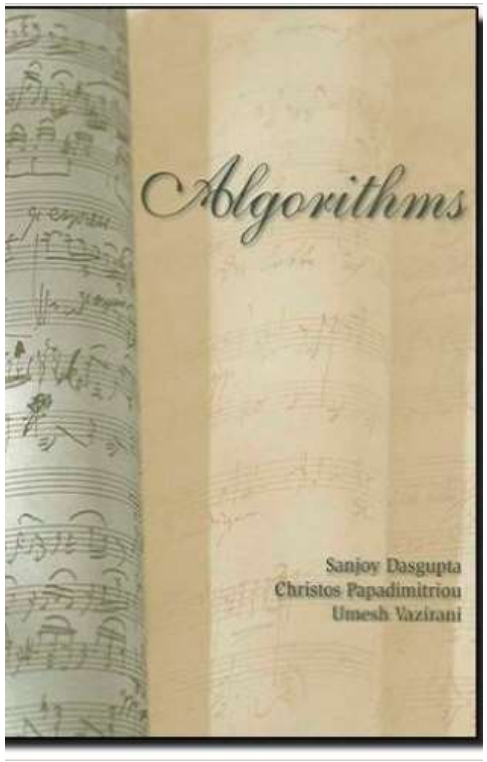
McGraw-Hill Education

336 pages
September 13, 2006

ISBN: 0073523402

Price: usually less than € 50

Was long time free online, still easy to find



# Optional: a classical book

Combinatorial Optimization: Algorithms and Complexity

*Christos H. Papadimitriou*

*&*

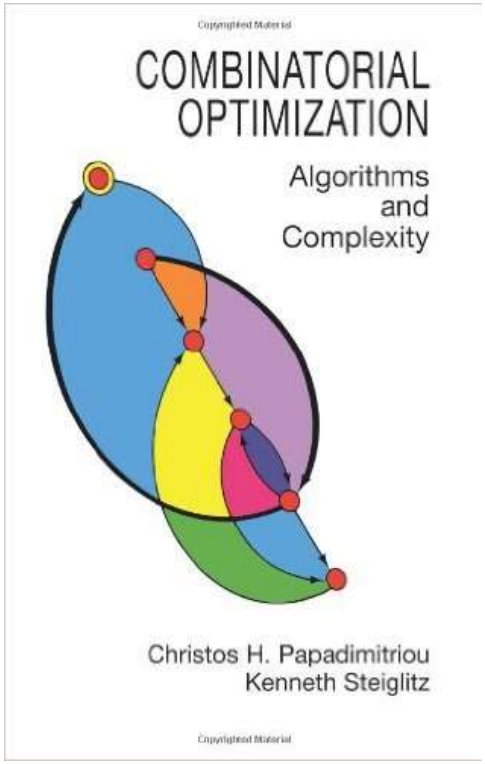
*Kenneth Steiglitz*

Dover Publications

512 pages
July 7, 1998

ISBN: 0486402584

Price: usually less than € 20



# Period 4

| Days | theme | lecture | tutorial |
|---|---|---|---|
| February 3 and 6 | Introduction, graphs, paths and trees, flows | Gromicho | Sitters |
| February 10 and 13 | Complexity classes and reductions | Gromicho | Sitters |
| February 17 and 20 | Dynamic Programming | Gromicho | Gromicho |
| February 24 and 27 | Scheduling | Sitters | Sitters |
| March 2 and 5 | Approximation algorithms | Sitters | Sitters |
| March 9 and 12 | Material relevant for the case | Gromicho | Sitters |

The **exam** is at the end of period 4!

March 27, 12:15-14:30

***The roster wrongly shows an exam on June 2***

# Block 5

| Days | theme | lecturer | type |
|---|---|---|---|
| March 31 | Introducing the case | Gromicho Sitters | Lecture |
| April 7 | Q&A | Gromicho | Tutorial |
| April 14 | Advise | Gromicho | Tutorial |
| April 21 | Advise | Gromicho | Tutorial |
| April 28 | Advise | Sitters | Tutorial |
| May 5 | | | |
| May 12 | Presentations and discussion | Gromicho Sitters | Assessment |

# Grades: exam + case


**Minimum 5 for each part.**
**The resit is only for the exam!**
**Case is compulsory!!!**

# *Algorithms* by Dasgupta et al

Chapter 0, Running time

Sections 4.1 - 4.4 Dijkstra's algorithm

Sections 5.1.1 - 5.1.3 Minimum Spanning Tree

Chapter 6  Dynamic Programming

Sections 7.1-7.4  Linear programming and Flows

Chapter 8 NP-completeness

Sections  9.2 - 9.3

# Course bibliography: tutorials

The answers to all exercises will be provided.

Some extra material will be used and placed on blackboard.

Tutorial exercises are part of the exam material.

Questions from the slides are intended to motivate discussion and reflection.

# Today's course

Main source: notes om Graphs by René Sitters

Additionally, from the book Algorithms:

- Preliminaries from chapter 0
- Paths in graphs including Dijkstra in chapter 4

# What is combinatorial optimization?

Combinatorial optimization is ‘just’ to choose the best (optimum) from a finite number of alternatives.

Seems easier than it really is… since finite does not mean attainable.

It is common to confuse combinatorial optimization with integer programming, especially with binary variables.

A stronger relation exists between combinatorial optimization and combinatorial decision problems: does a given instance have a specific type of solution?

# The focus of the course

Highlighting some areas such as graphs, networks, routing, packing, scheduling, …

Identifying important problems in those areas.
Distinguishing ‘easy’ from ‘difficult’ problems.

Be able to prove that a ‘difficult’ problem is indeed difficult.

Presenting mathematical models for these problems.
Studying the important properties of these models.
Explaining solution procedures for each problem, emphasizing on computational efficiency.

# How difficult is a combinatorial optimization problem?

We will learn that combinatorial optimization problems may differ much in the effort needed to solve them.

We will understand that there are mainly two types of combinatorial optimization problems:

- Those that we can solve efficiently
- Those that are as hard to solve as the hardest combinatorial optimization problems

# Typical structure of a ‘topic’

A problem description

A proof of hardness in case the problem is indeed hard

Algorithms, including analysis of the effort

- Exact (optimal)
- Heuristic (sometimes approximation) if the problem is hard

# Effort of algorithms

Running time is not a property of the problem, is a characteristic of algorithms and their implementation details!

Effort is measured by counting ‘elementary operations’ as a function of the ‘input size’.

These concepts will be defined later in detail, important for now is understanding that differences between computers or languages should not matter.

We start with a simple example.

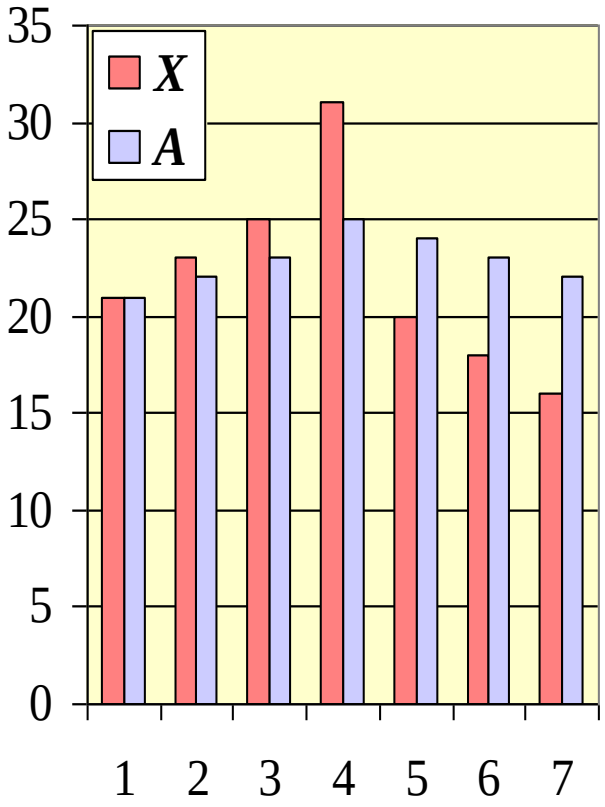
# Computing Prefix Averages

Let us illustrate computational effort analysis with two algorithms for prefix averages

The $i^{th}$ prefix average of an array $\boldsymbol{X}$ is average of the first $\boldsymbol{i}$ elements of $\boldsymbol{X}$:

$$\boldsymbol{A_i} = \frac{\boldsymbol{X_1} + \boldsymbol{X_2} + \ldots + \boldsymbol{X_i}}{i}$$

Computing the array $\boldsymbol{A}$ of prefix averages of another array $\boldsymbol{X}$ has applications to financial analysis



# Prefix Averages version 1

The following implementation computes prefix averages by applying the definition

```
def PrefixAverages1( X ):
    # Input array X of n integers
    # Output array A of prefix averages of X
                                # operations
    n = len(X)                  # O(1)
    A = [0]*n                   # O(n)
    for i in range(n):          # O(n)
        s = 0                   # O(n)
        for j in range(i+1):    # O(1+2+3+ ... + n) <- highest
            s = s + X[j]        # O(1+2+3+ ... + n) <- highest
        A[i] = s / (i+1)        # O(n)

    return A                    # O(1)
```

# Prefix Averages version 2

The following implementation computes prefix averages in less time by keeping a running sum

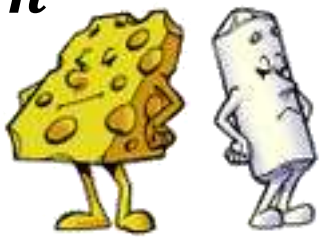
```
def PrefixAverages2( X ):
    # Input array X of n integers
    # Output array A of prefix averages of X
                                # operations
    n = len(X)                  # O(1)
    A = [0]*n                   # O(n)
    s = 0                       # O(1)
    for i in range(n):          # O(n)
        s = s + X[i]            # O(n) <- highest
        A[i] = s / (i+1)        # O(n)

    return A                    # O(1)
```

# How does the effort change with $n$?

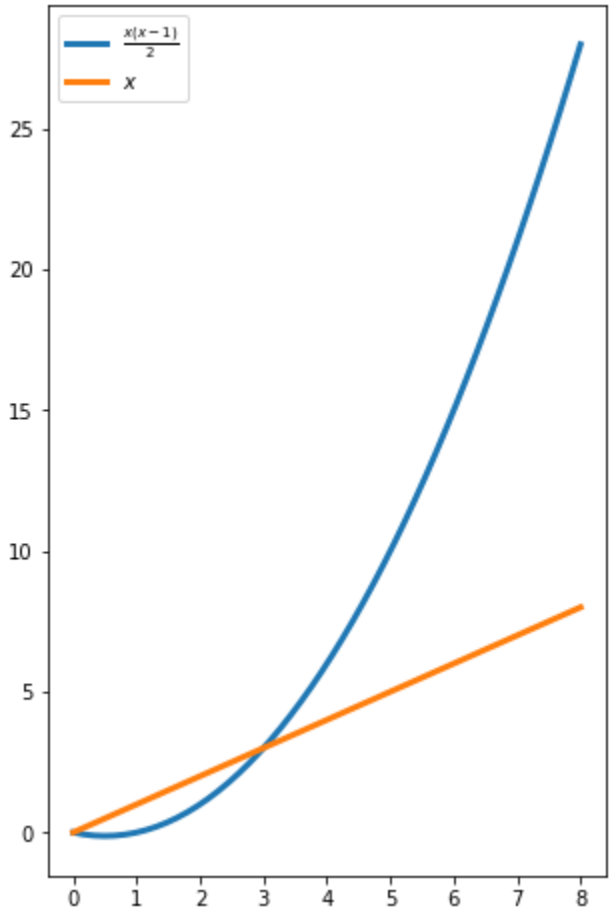
The running time of ***PrefixAverages1*** is dominated by

$$1 + 2 + \ldots + n$$

The sum of the first $n$ integers is $\frac{n(n+1)}{2}$



The running time of ***PrefixAverages2*** is dominated by $n$



# Order of

We say that a function $f$ is of order of a function $g$ and write:

$$f(n) = O(g(n))$$

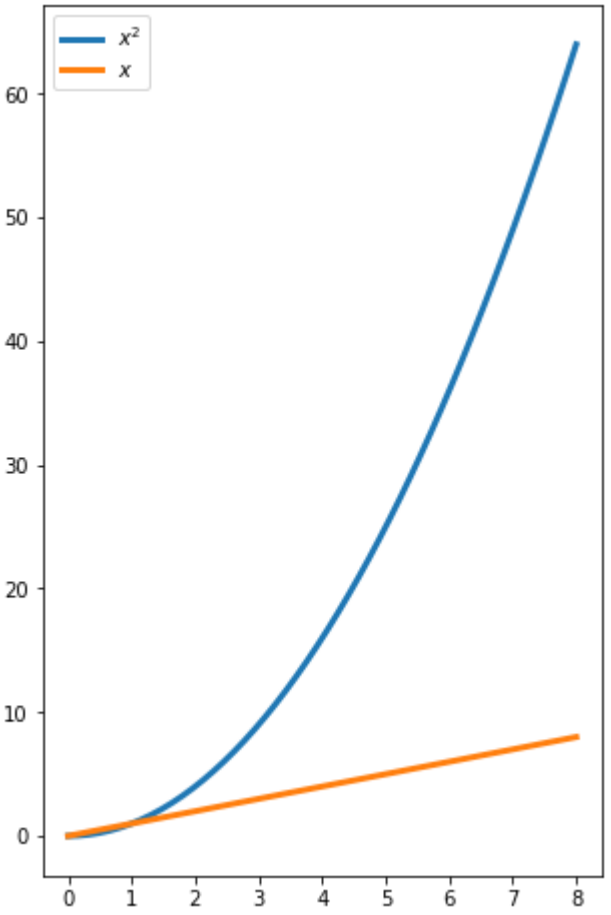
if there is a constant c and a number k such that

$$f(n) \leq cg(n) \quad \text{for all} \quad n \geq k.$$

# Prefix averages: conclusion

Understanding that PrefixAverages1 is $O(n^2)$ and PrefixAverages2 is $O(n)$ should become more or less 'automatic'.

Concluding that PrefixAverages2 is much more efficient and scales much better with the growth of the input should then be immediate.



# Effort is an implementation issue!

It is common to say

*this problem can be solved in* $O(g(n))$

for some function $g$, but it is not always obvious how to achieve the specified effort.

As we just saw, a roughly described algorithm leaves so many details to the implementation that the factual runtime may change the nature of the function $g$!

We will see this again at the end of today's course when studying max flow algorithms.

# Complementary material

# What is this course all about?

Models
Theorems
Proofs
Methods
Algorithms
Implementations
Computational effort
Computational complexity

# Start with ‘decision’ problems

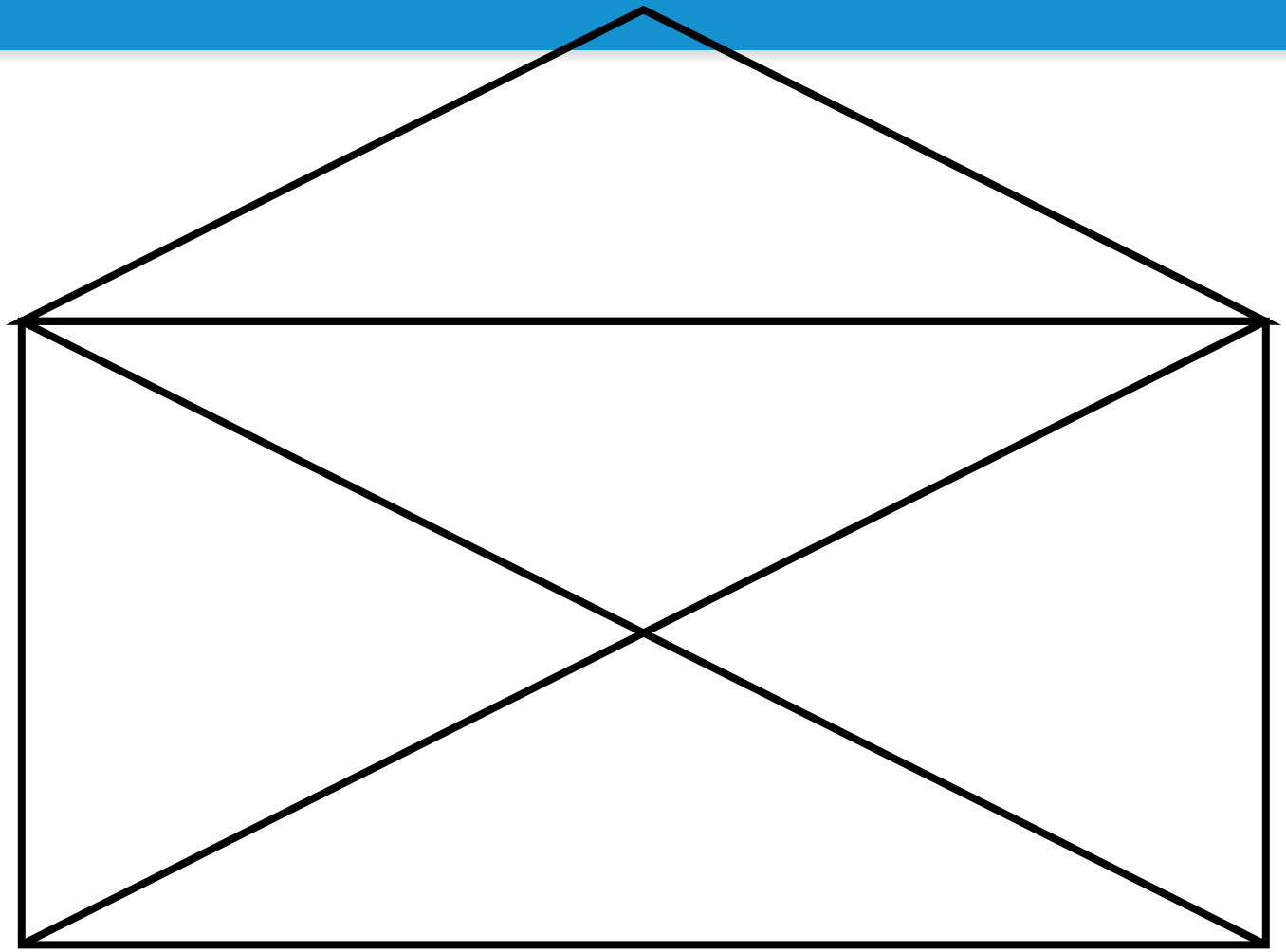
Is this number prime?
Can I reach that location from this one using the available roads?
Can I finish my tasks in less than a given amount of time?
Can I draw this graph without lifting the pen and without retracing lines?

# Let us illustrate a few things

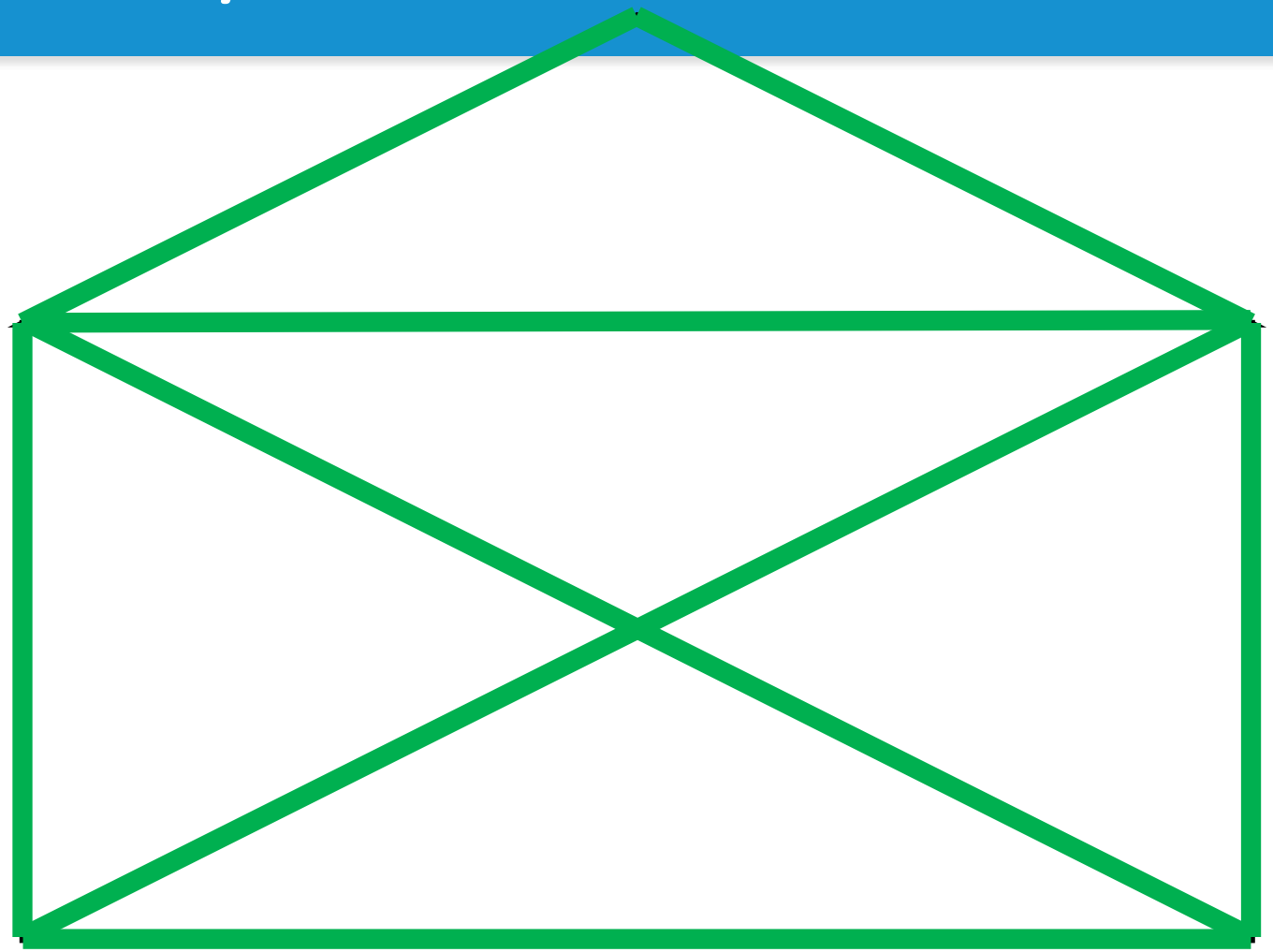
We will now see how a theorem, and in particular its proof, may relate to:

- Answering a decision problem: yes or no
- Understanding the effort of making the decision
- Leading to an algorithm to find an yes answer if it exists
- Determining the effort of finding a solution

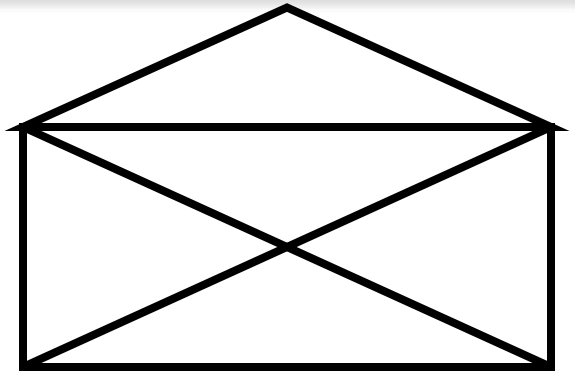
# Do you know this puzzle?



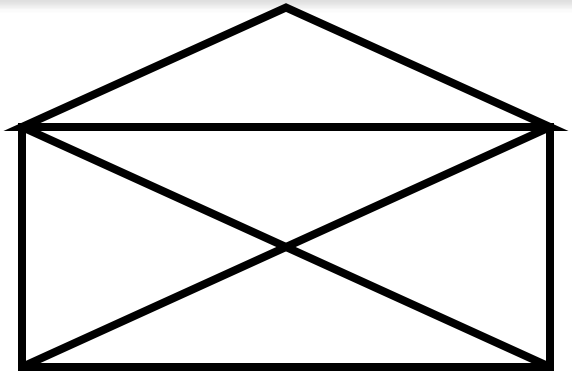
# Answer is **yes**



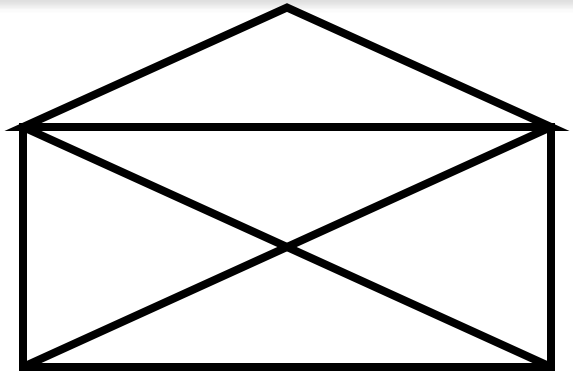
# What if you get the following cases?

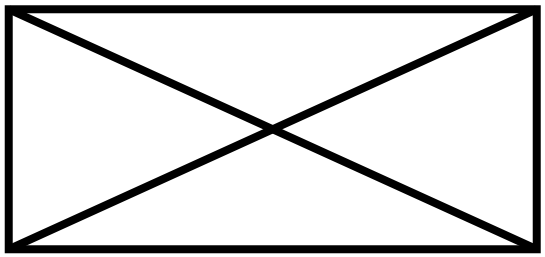


# What if you get the following cases?

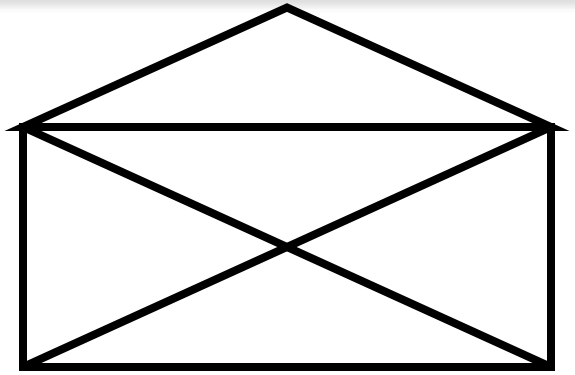


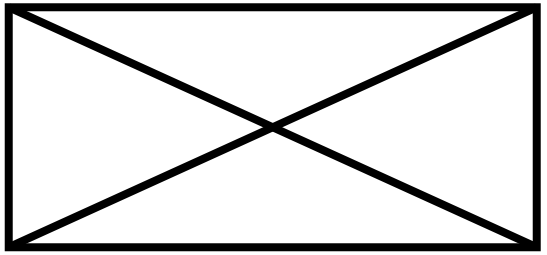
# What if you get the following cases?

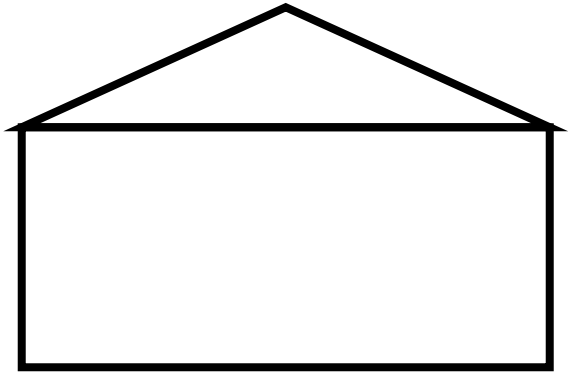




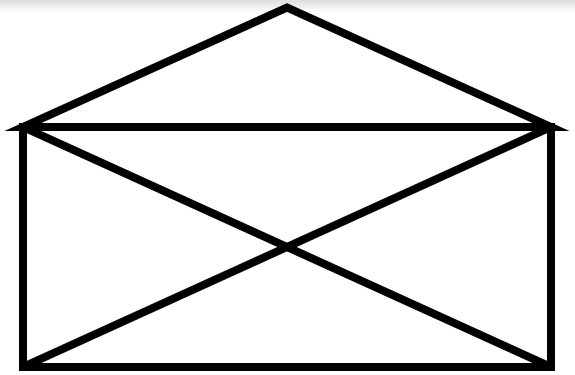
# What if you get the following cases?

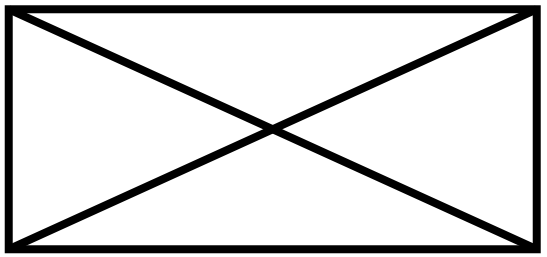


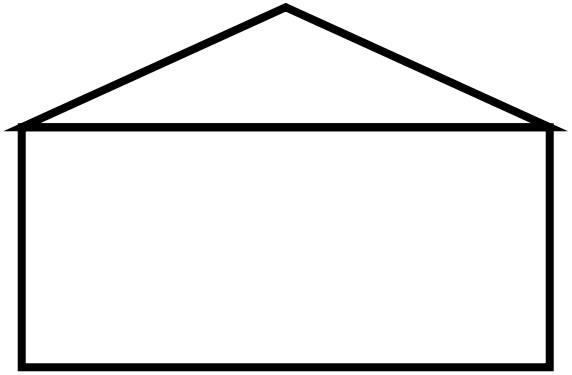




# What if you get the following cases?

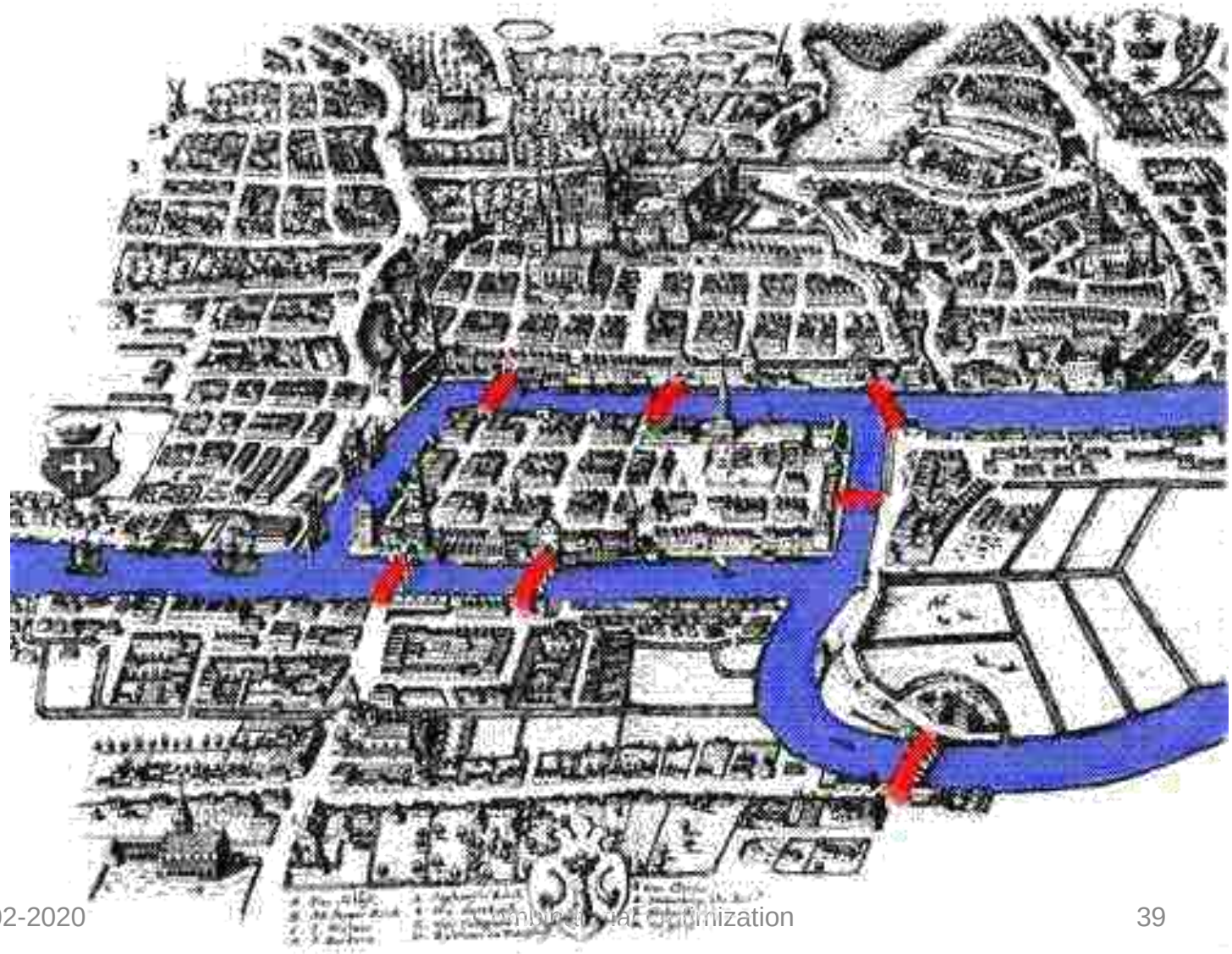




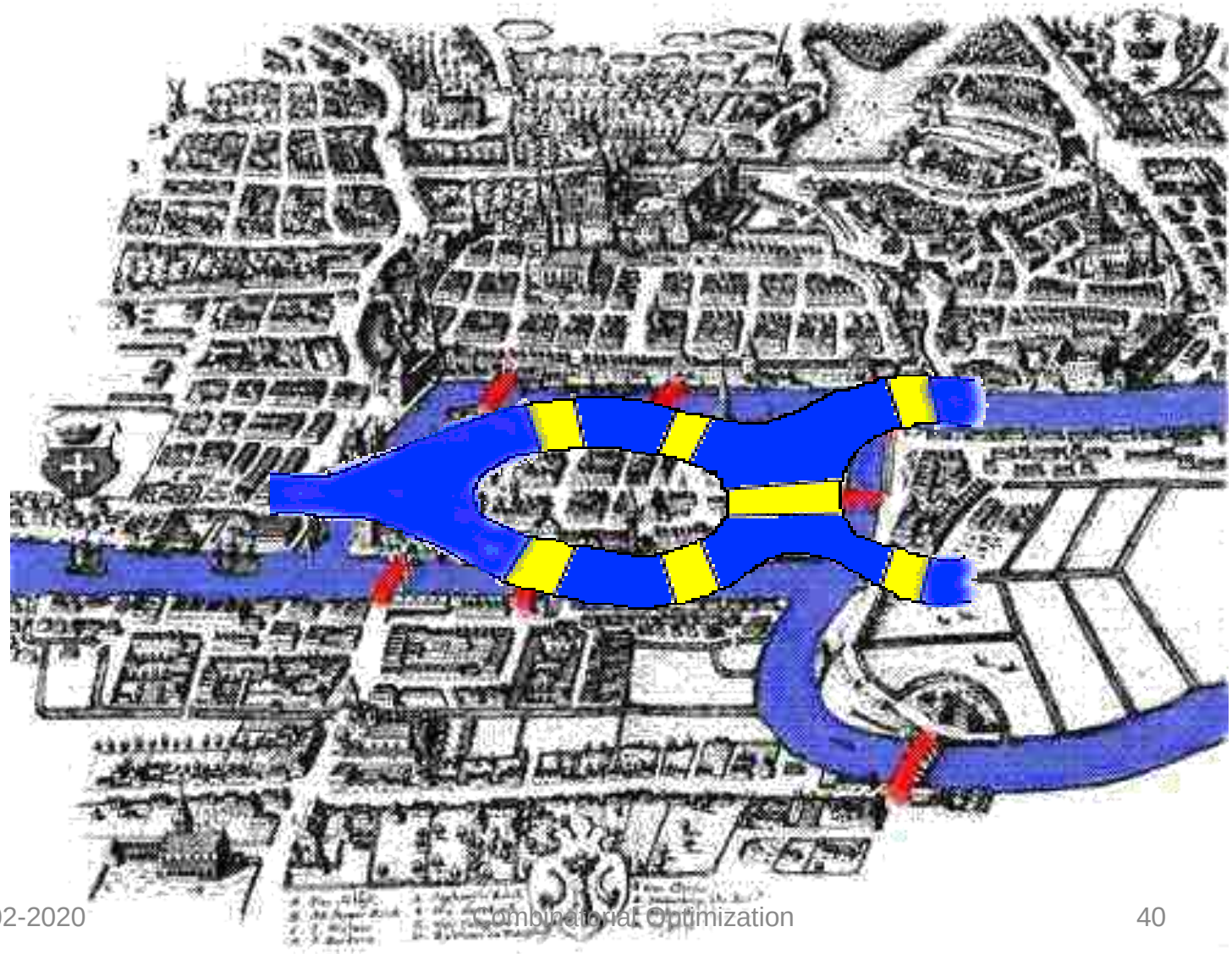


# Let us go back in time…

# The 7 bridges of Königsberg in the XVIII$^{th}$ century

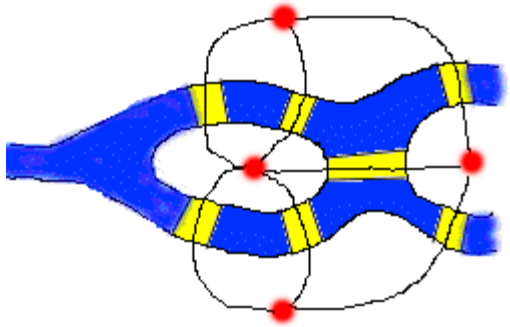


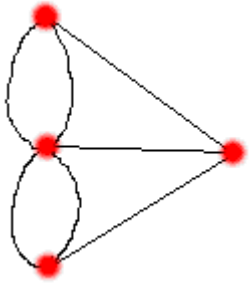
# A scheme: simplifies experimentation

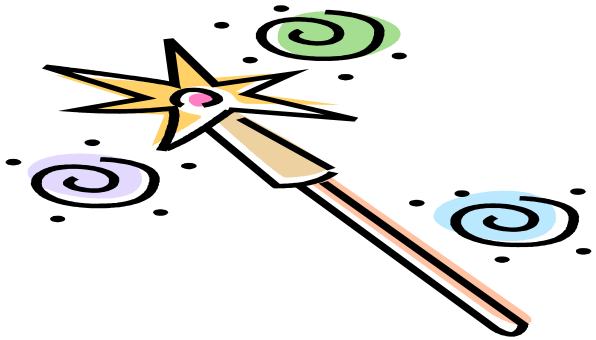


# A scheme: simplifies experimentation

# Higher abstraction: a model!



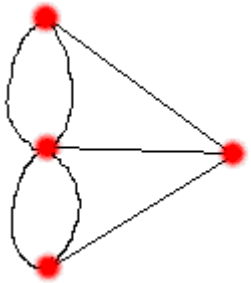


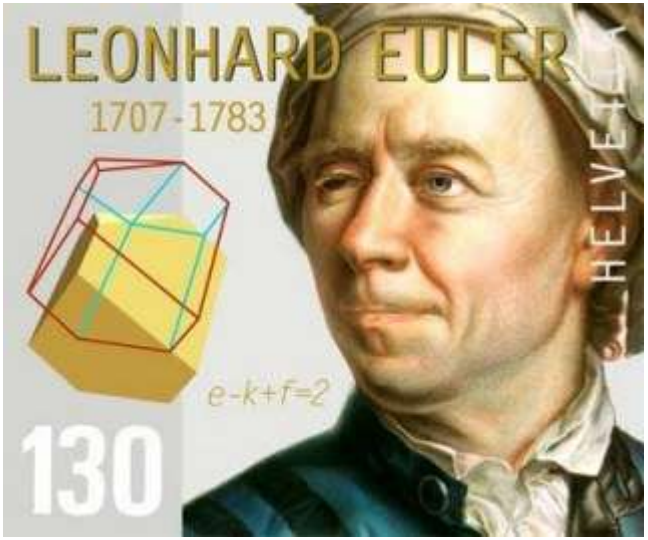


# Our first model: a graph (see notes!)

- Some points: nodes or vertices.
- Some lines connecting pairs of points: edges or arcs (in case they can only be traversed in one way).
- The number of edges at each node is called the degree of that node.
- In case of arcs then we distinguish in-degree and out-degree.
- Two nodes are connected if one can move from the first to the second node using edges or arcs and maybe intermediate nodes.

# The first graph theorem of Euler (1736)






*A connected (multi)graph has an Eulerian path if and only if the number of nodes with odd degree is 0 or 2.*

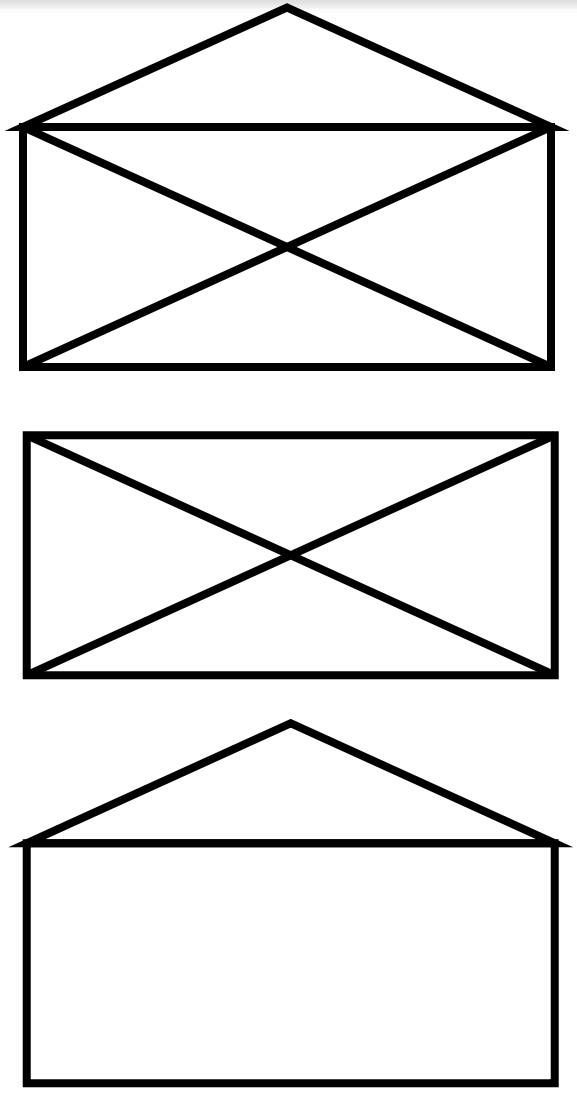
*May the number be 0 then the path is a cycle: it starts and finishes on the same node.*

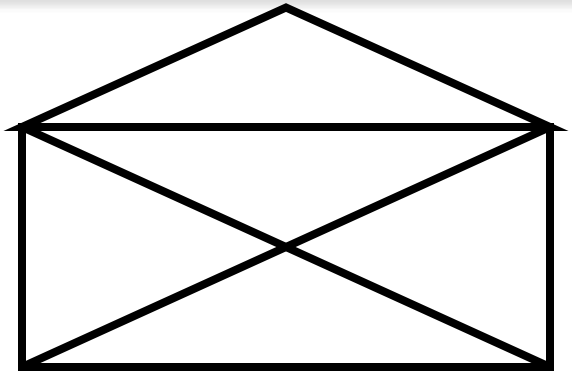
# Remarks

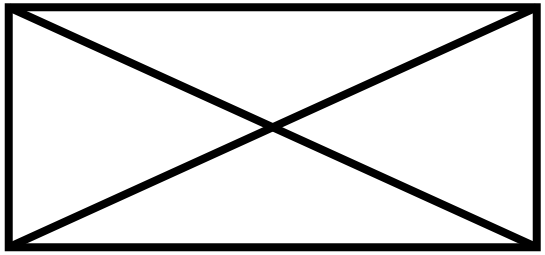
The theorem gives a certificate for a graph to be Eulerian.

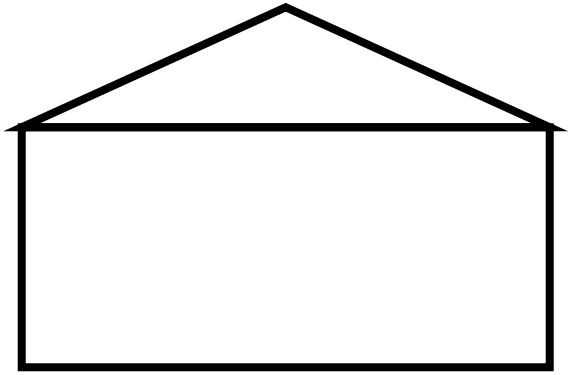
This certificate is very easy (and in fact ***efficient***) to check.

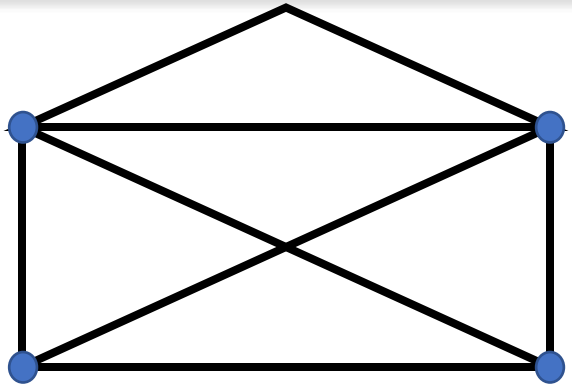
In itself, the certificate does not disclose nor requires an actual solution to the problem.

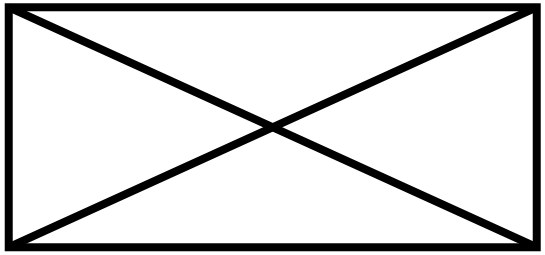


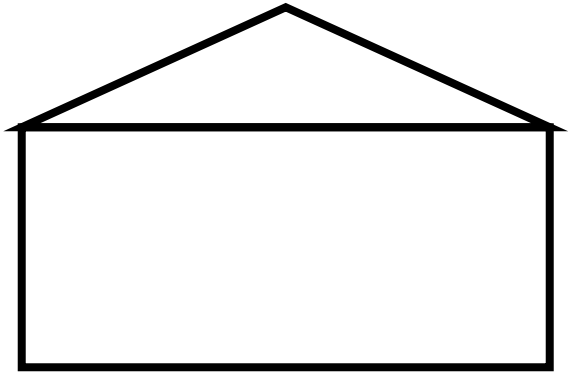




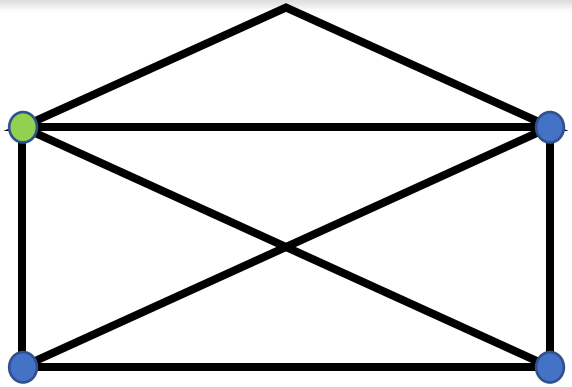


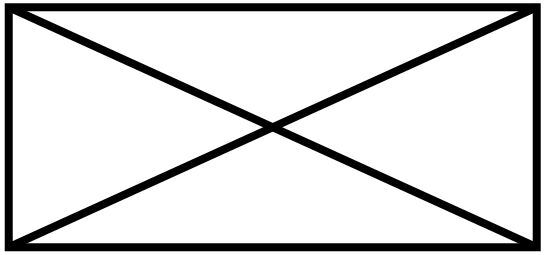


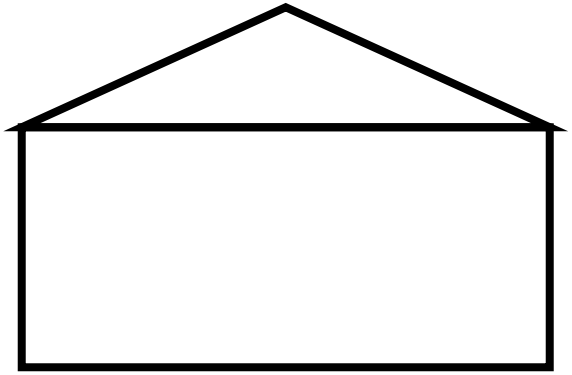


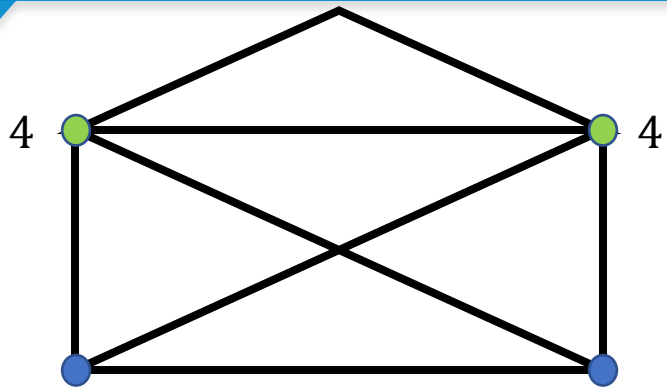


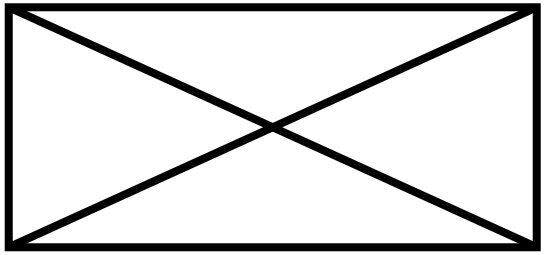
4

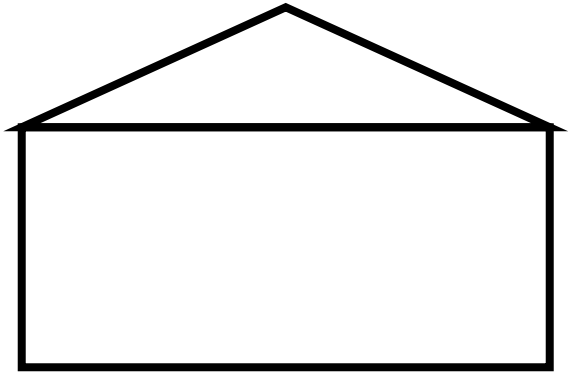


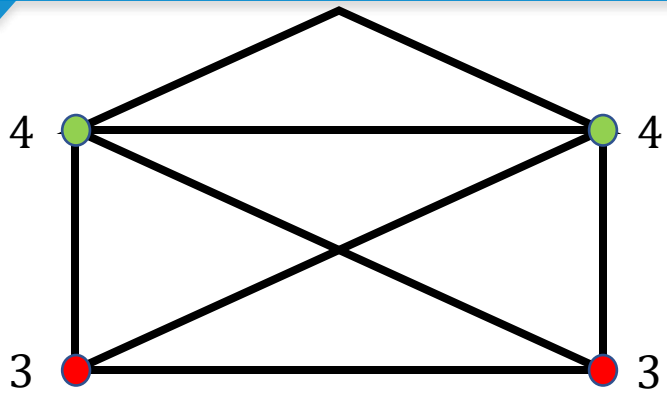


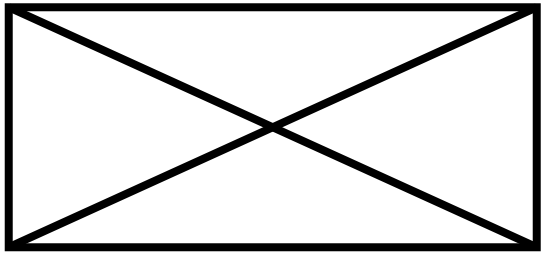


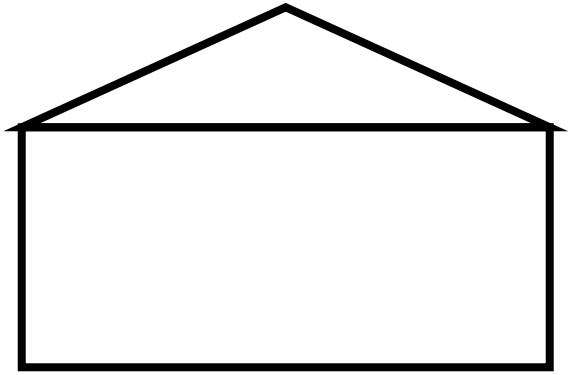

4
4

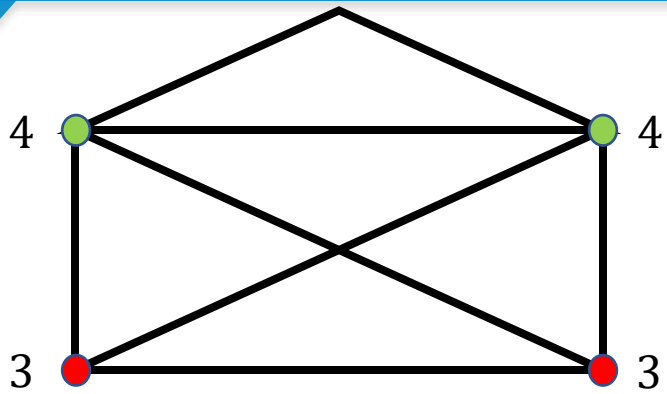


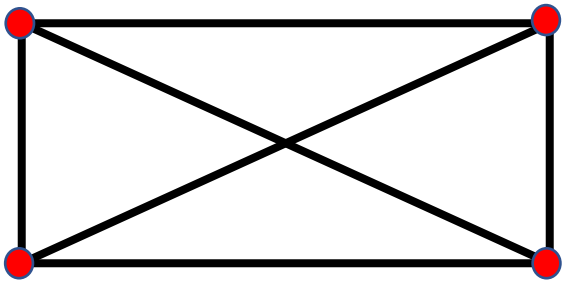


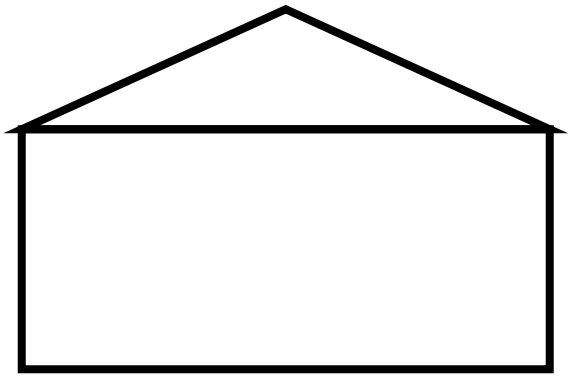

4
4
3
3

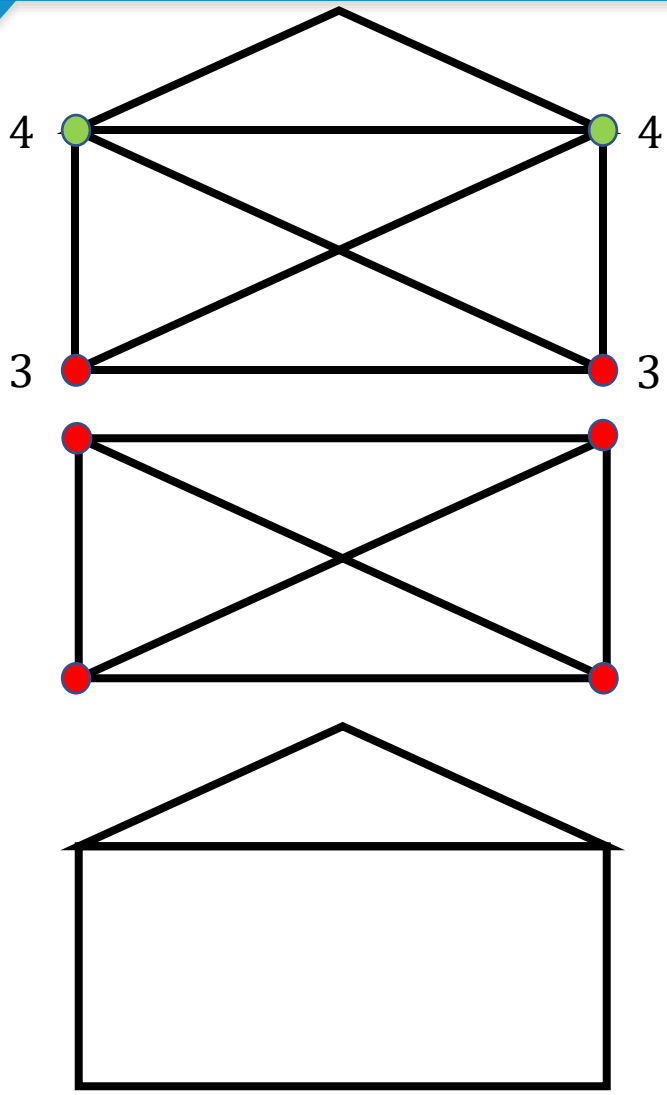


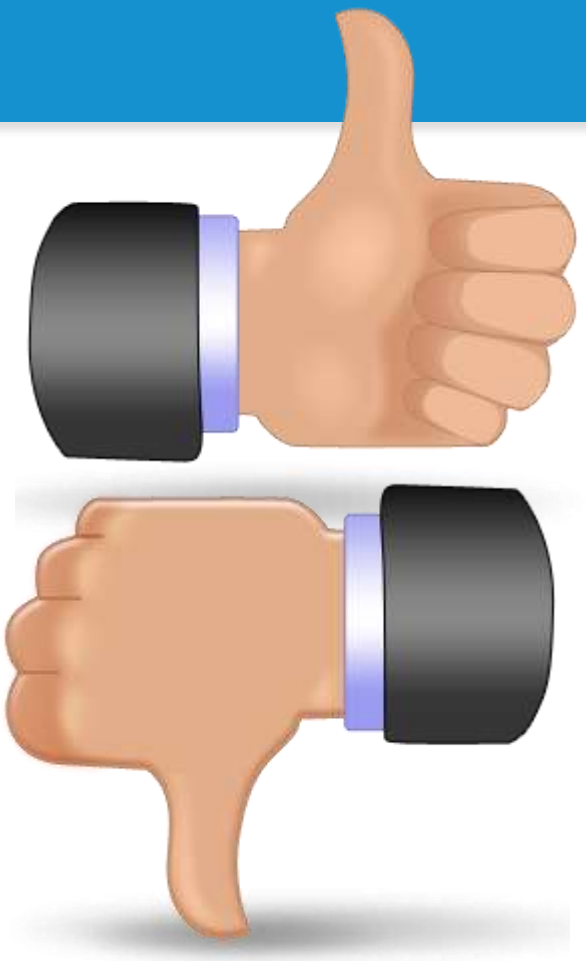


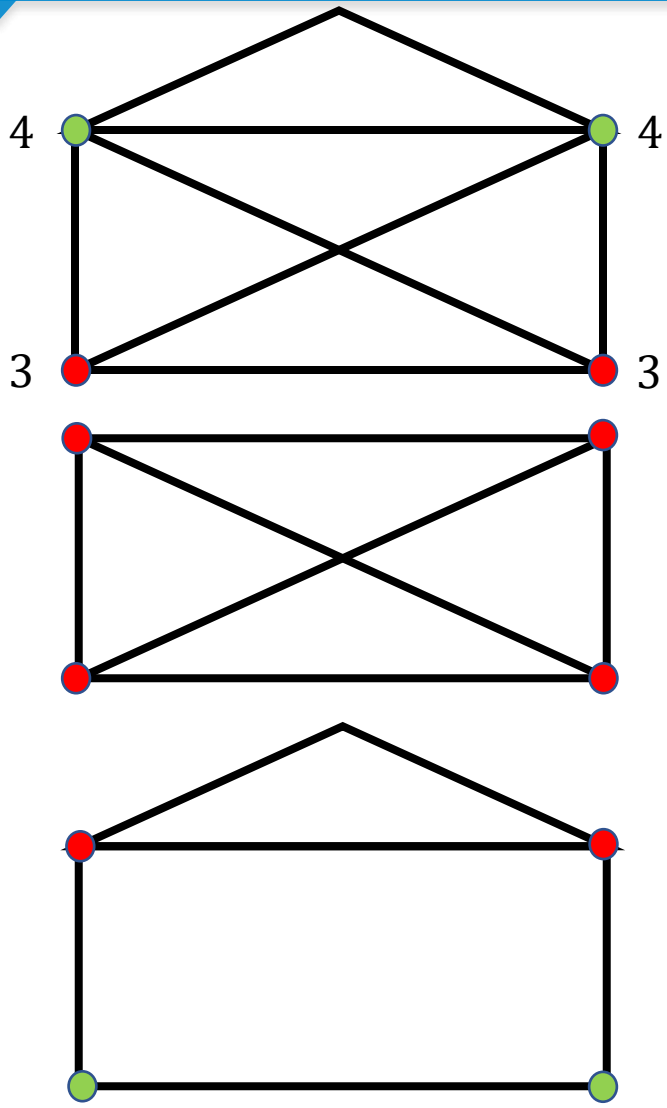

4
4
3
3

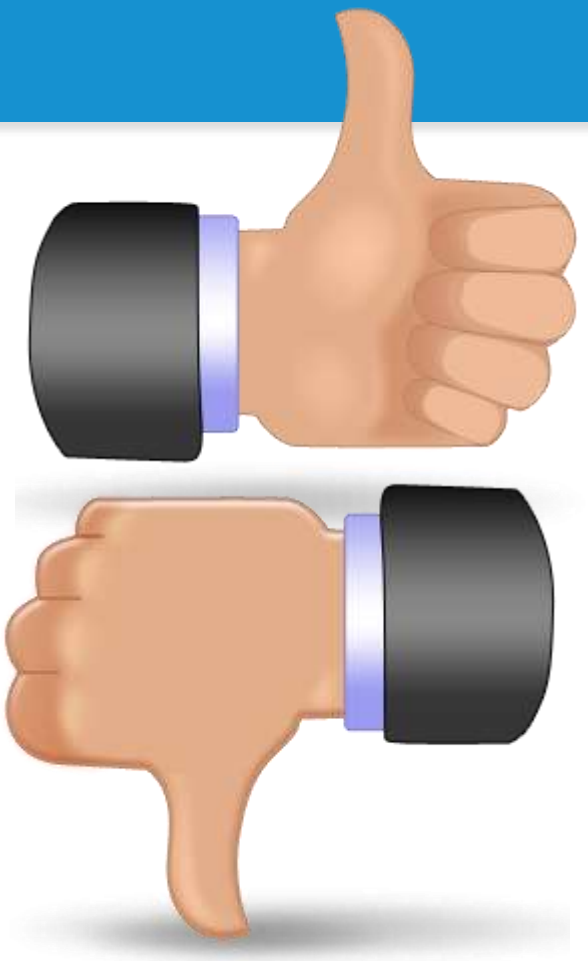


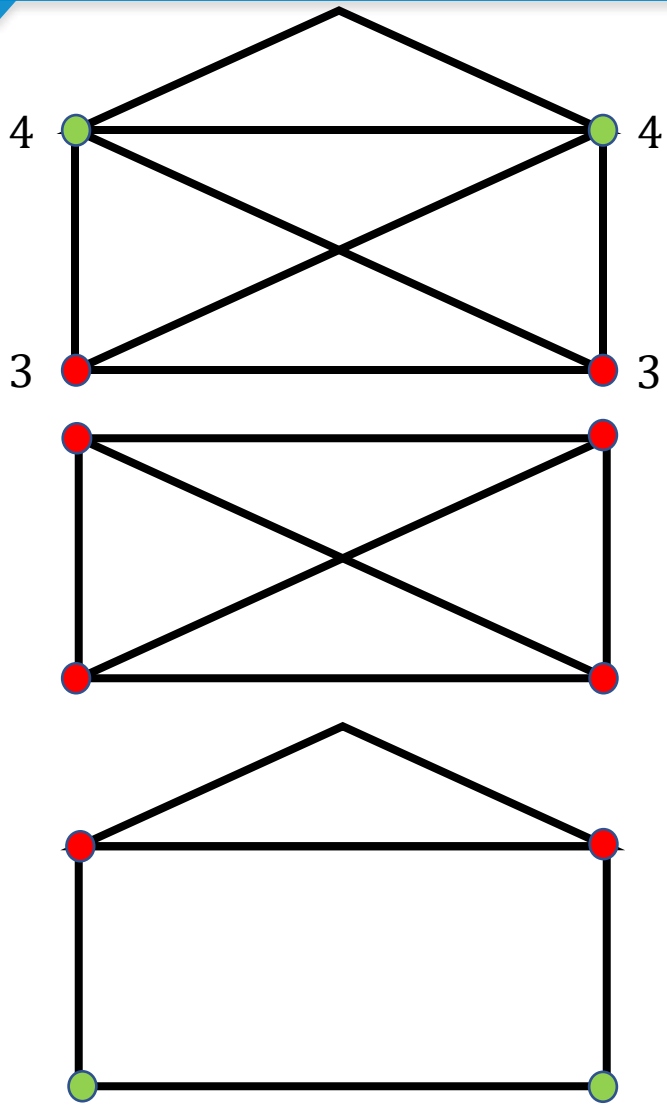


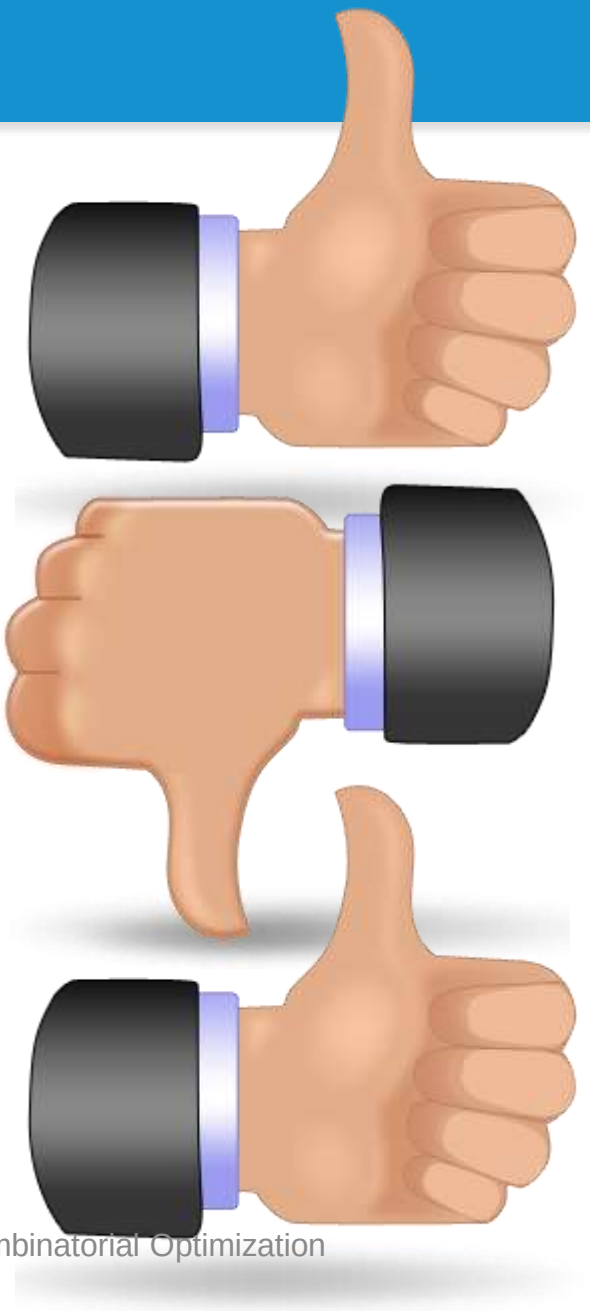

4
4
3
3

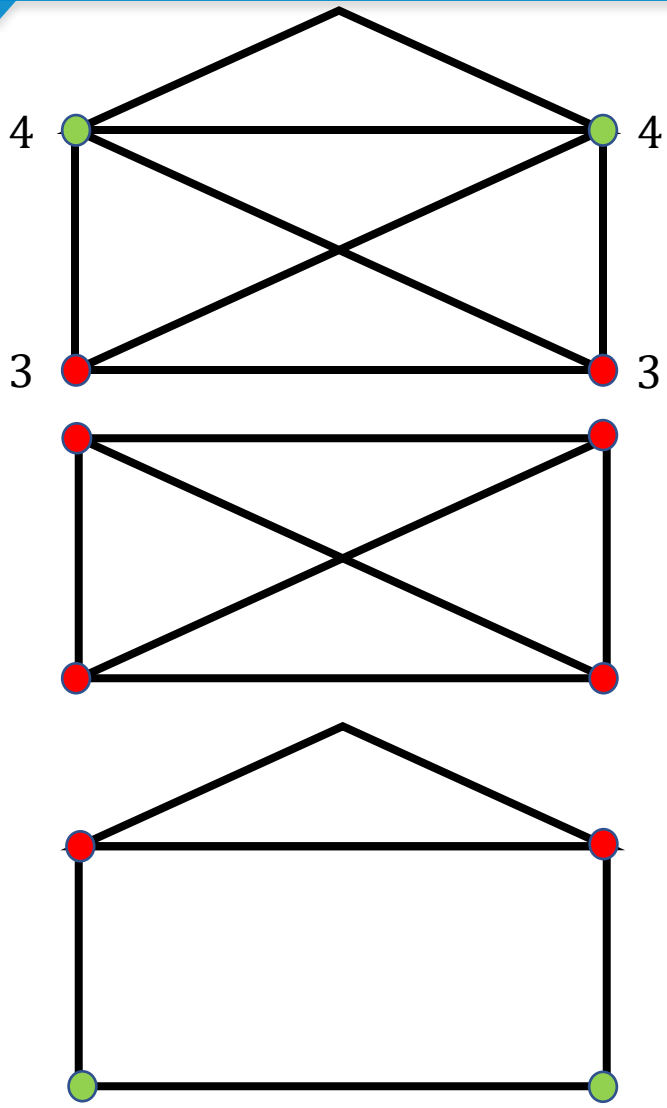


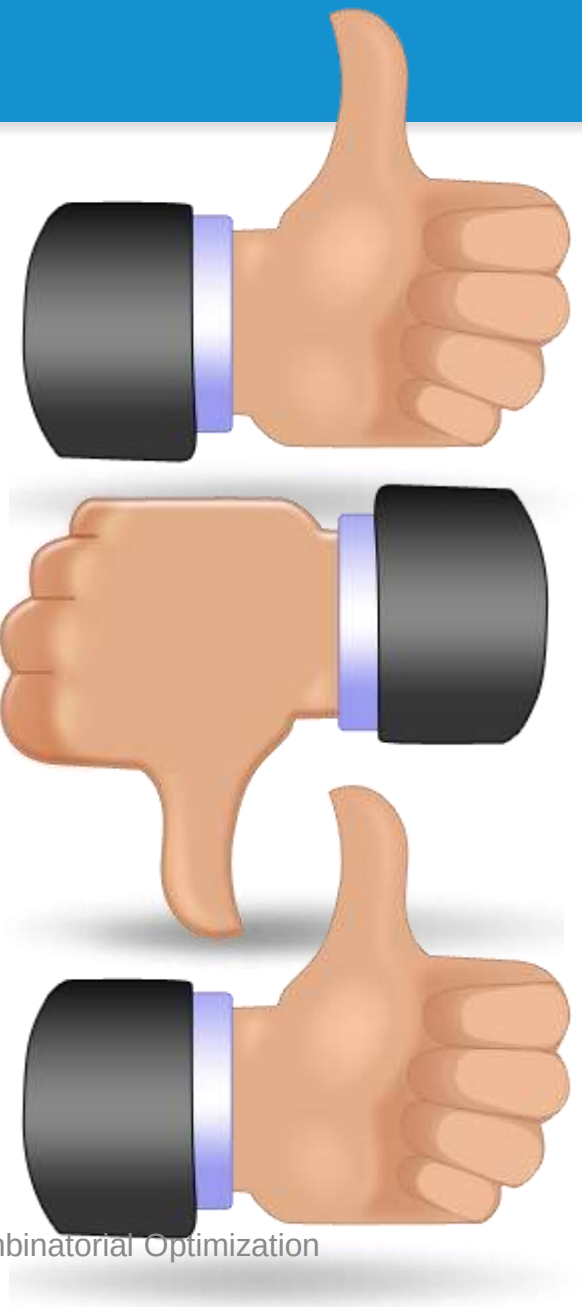

4
4
3
3



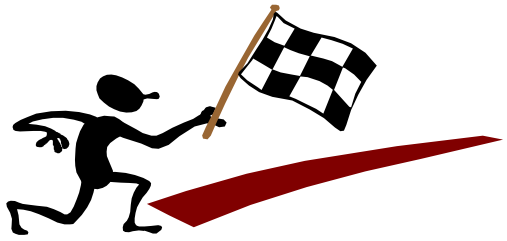

4
4
3
3

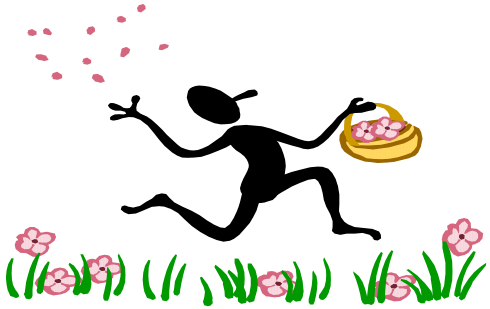


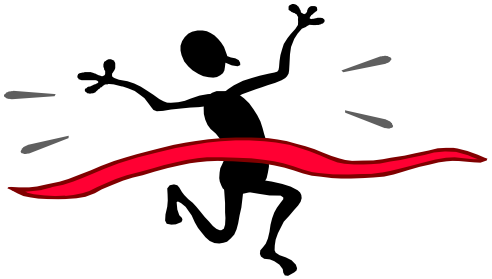

4
4
3
3

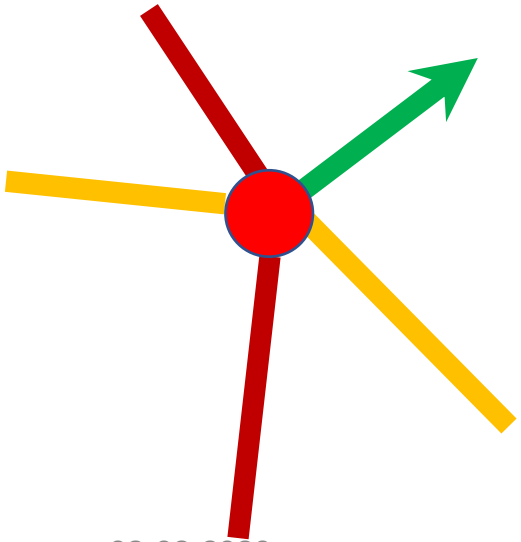


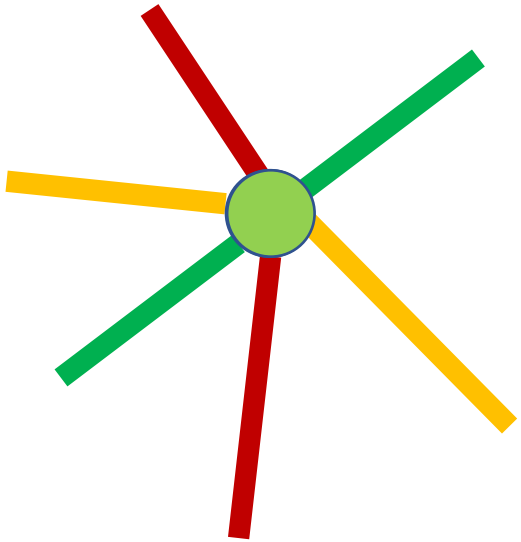
# A theorem must be proven

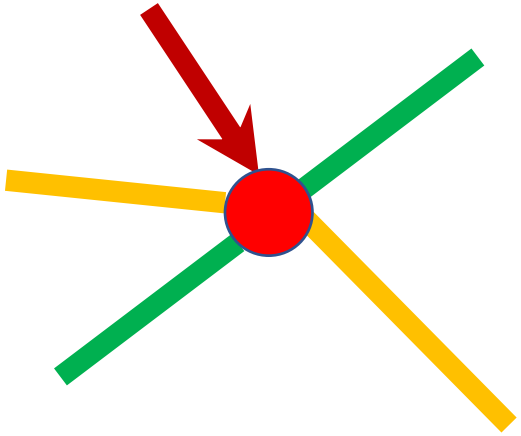


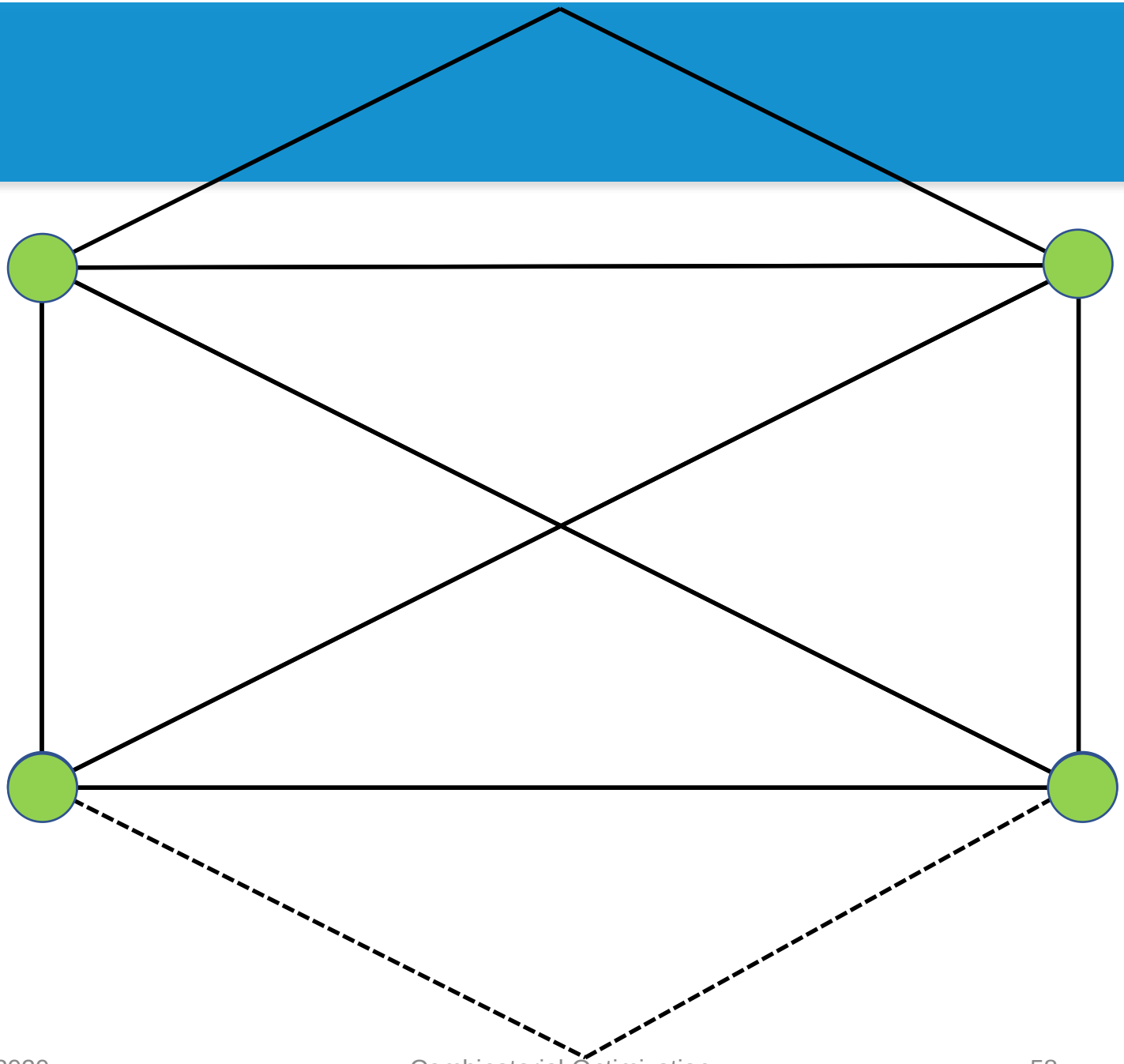


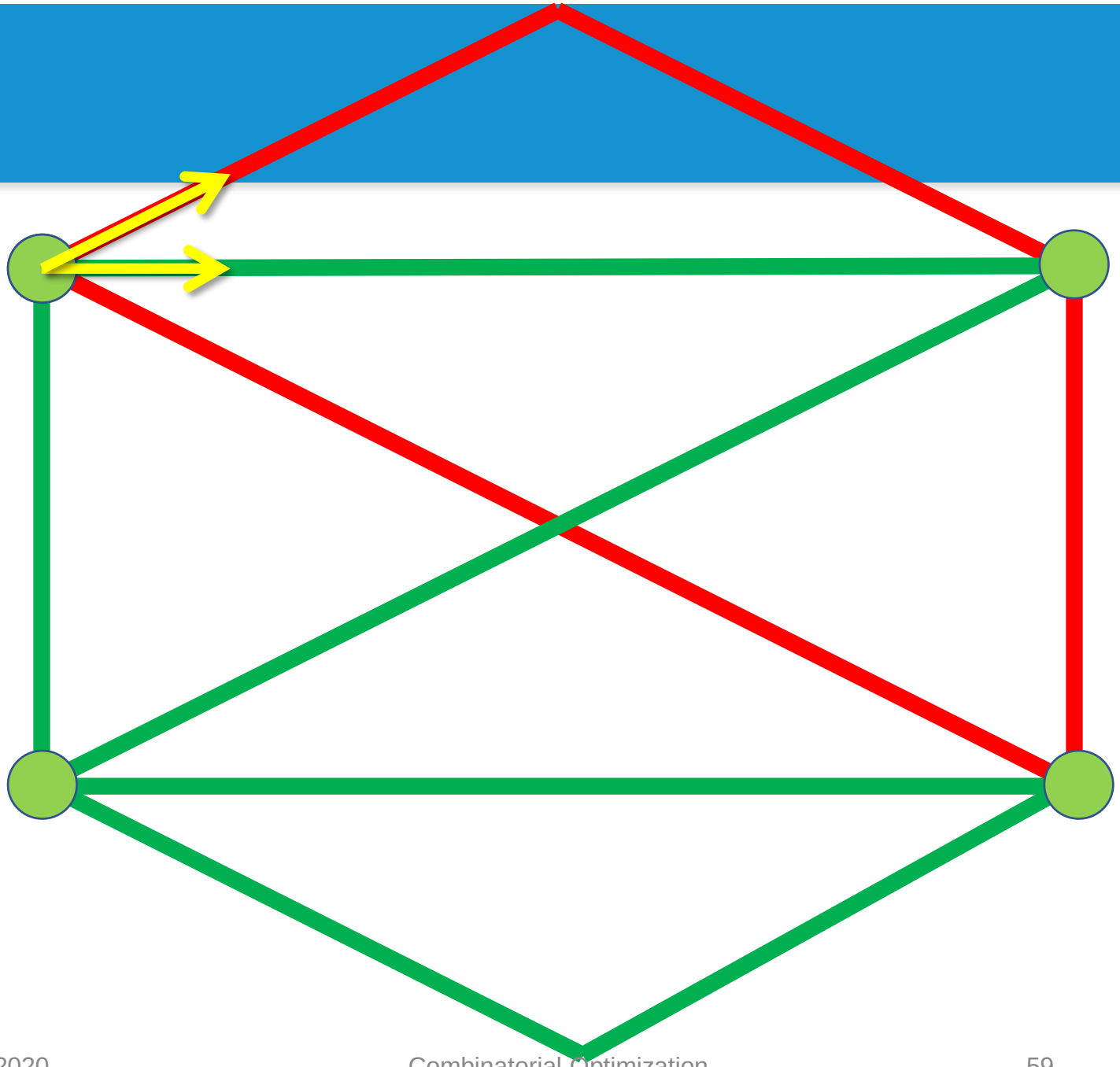


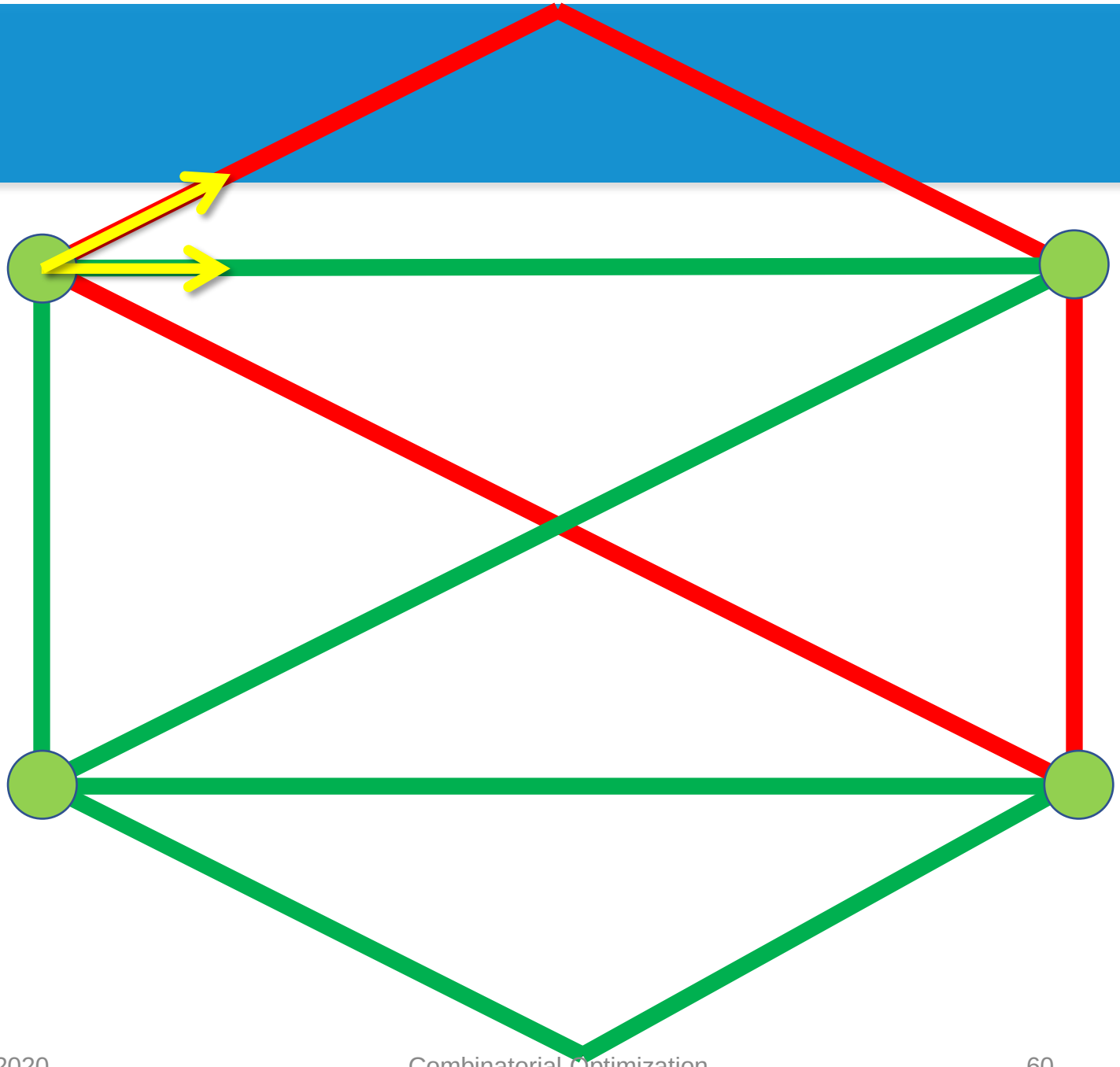


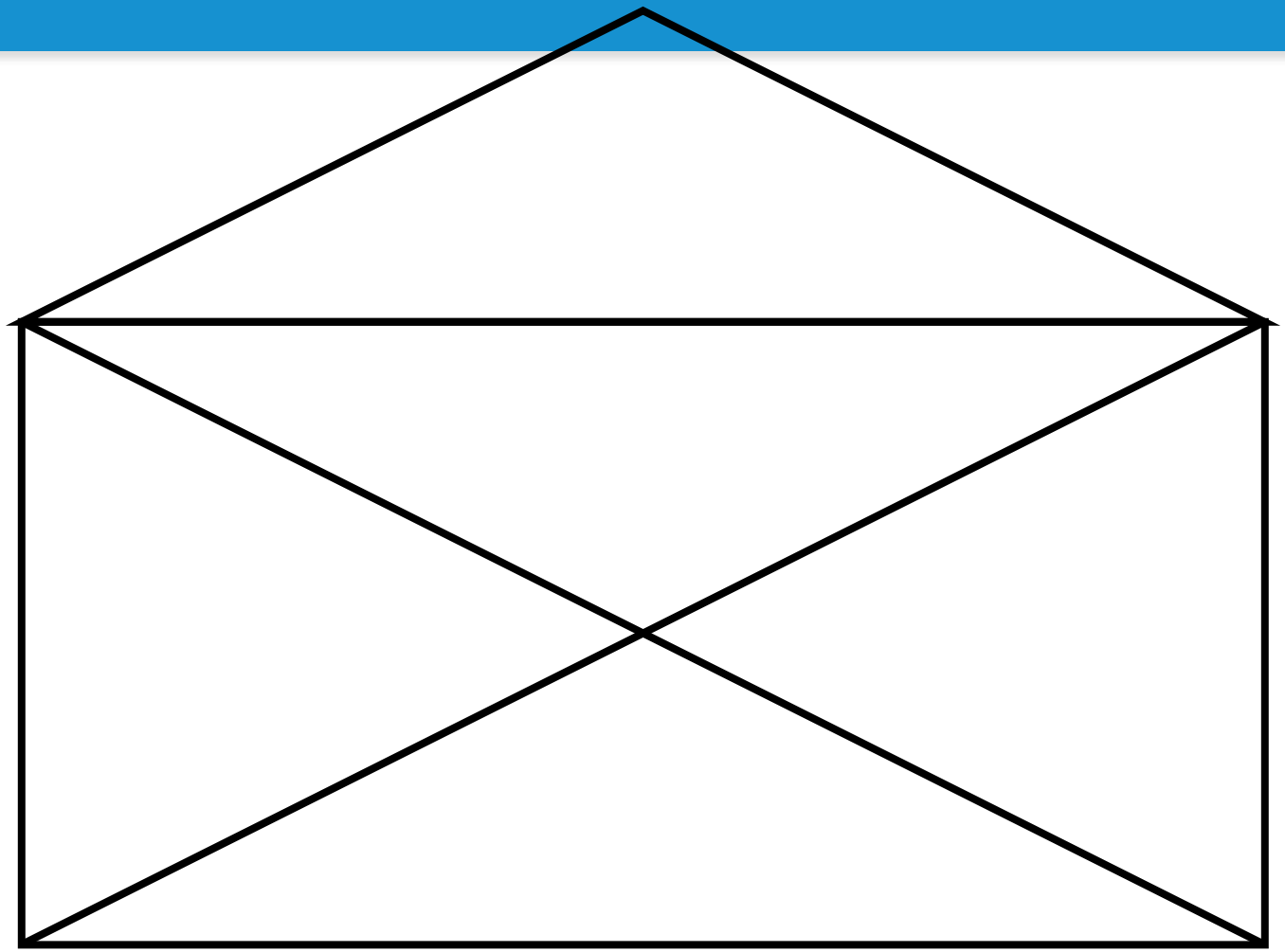


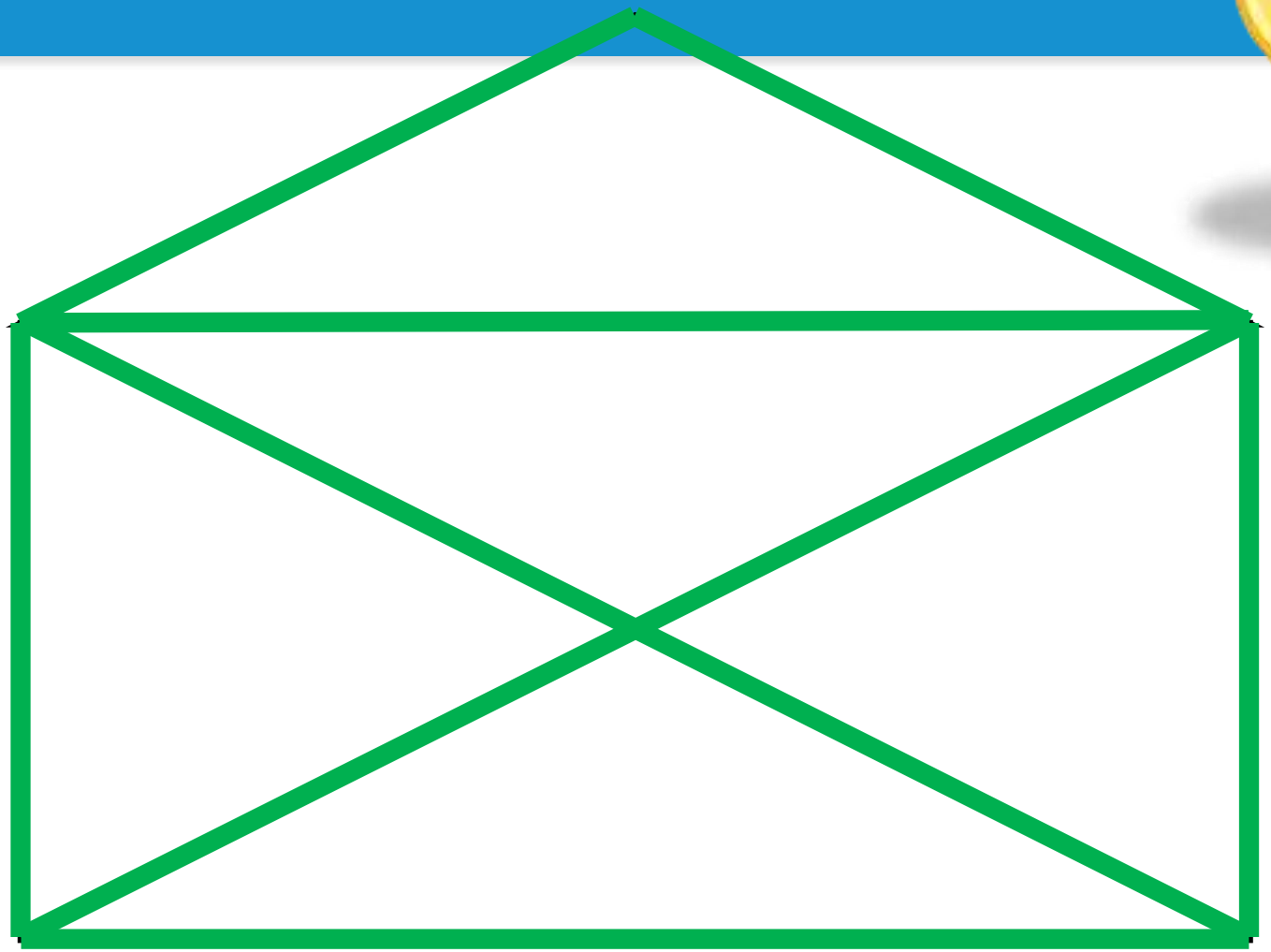












# What did the proof give us?

Some magic: we can now say that it holds for all graphs although we may not claim to have seen all graphs!

A present: an algorithm! Note that the algorithm revealed by the (constructive) proof is not provided by the theorem itself.

What do you think about the effort needed to apply this algorithm?

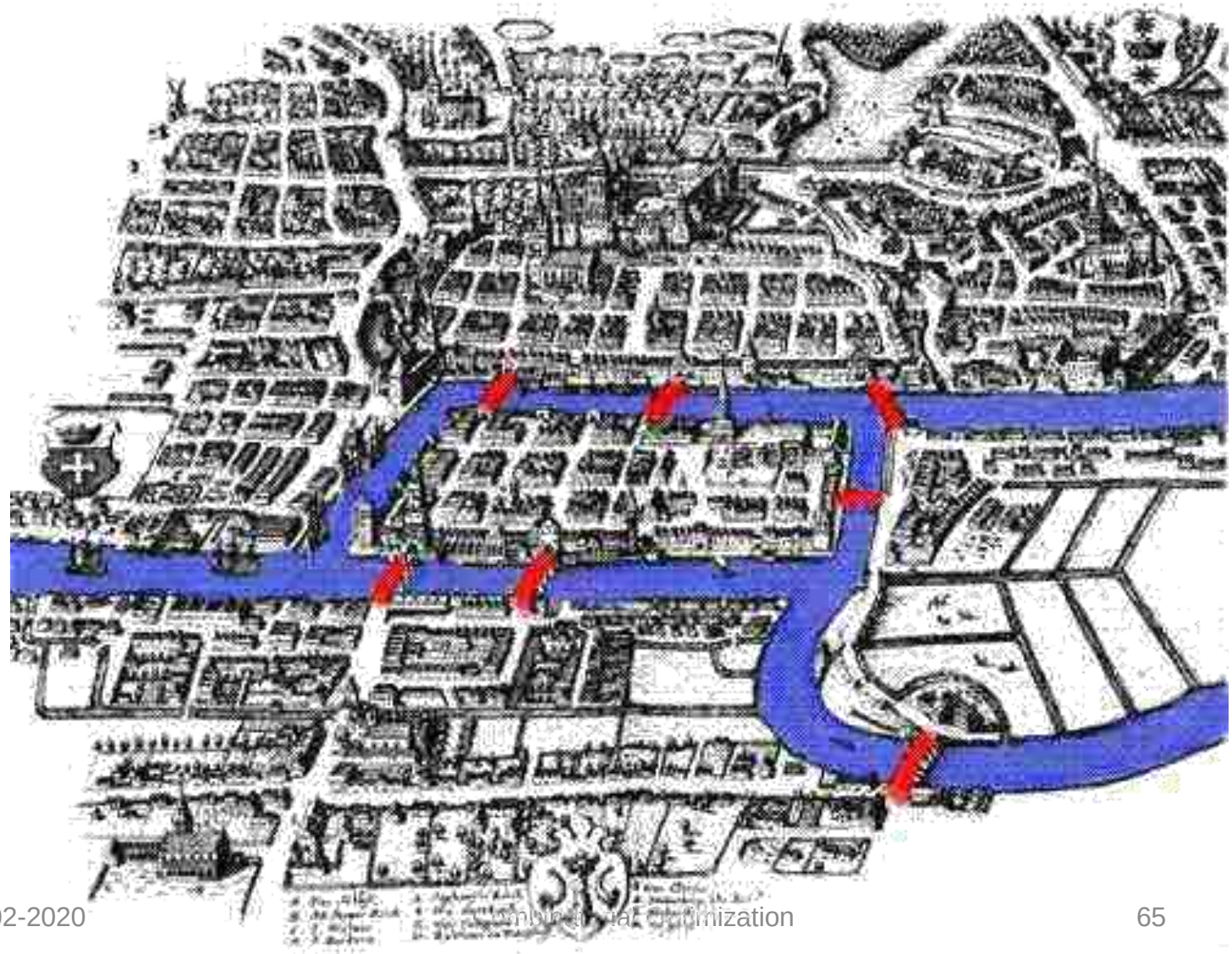
# question

Consider a connected (multi)graph with $n$ nodes and $m$ edges.

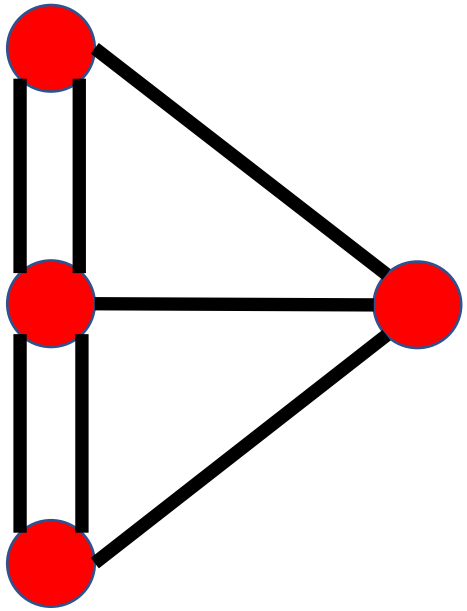
What is the effort required to check if this graph is Eulerian?

In case the graph is Eulerian: what is the effort required to find a Eulerian path or cycle?

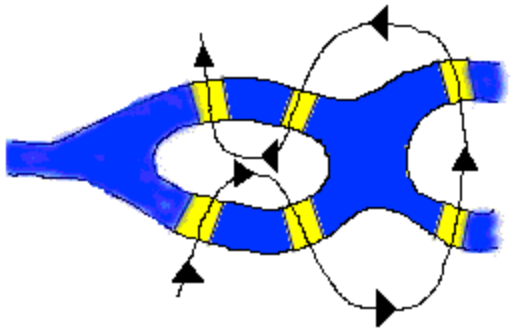
# And what about the seven bridges?



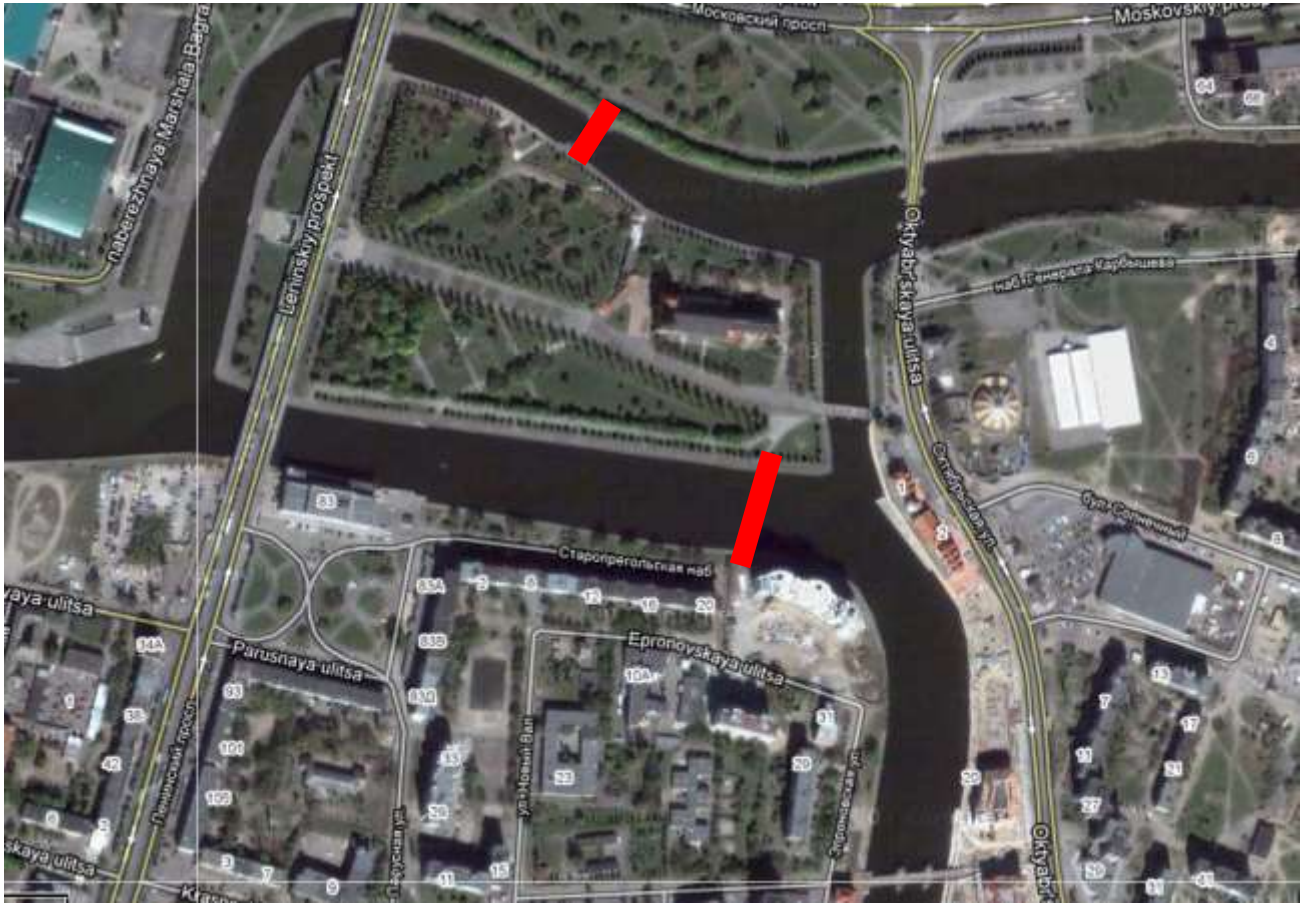
# Not possible!



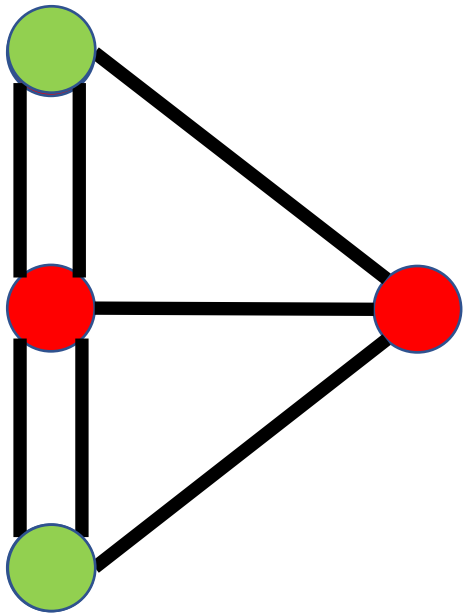
# Option 1: remove at least one bridge



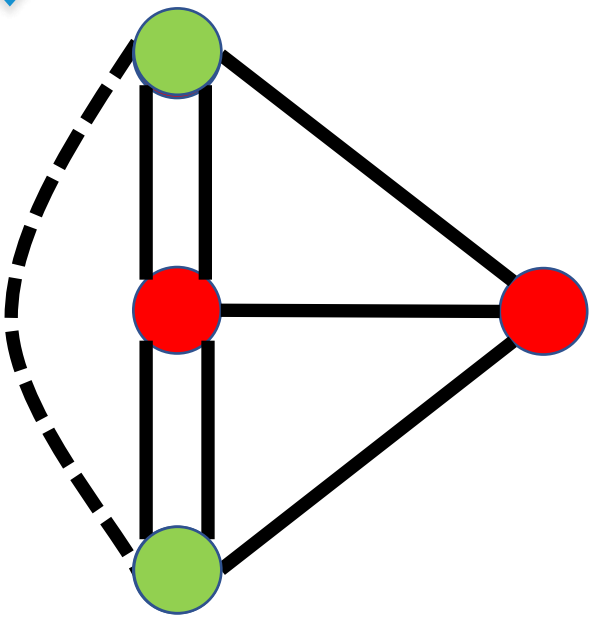
# Happened Königsberg in World War II



# Nowadays it is possible!



# Option 2: add bridges!




**Chinese Postman Problem:**

**What is the minimum number of edges we need to add to make the graph Eulerian**?

First studied by Chinese scientist, Kwan Mei-Ko in 1960

Efficient algorithm by Jack Edmonds, 1965

# note

Although we are still discussing decision problems we just met an optimization problem!

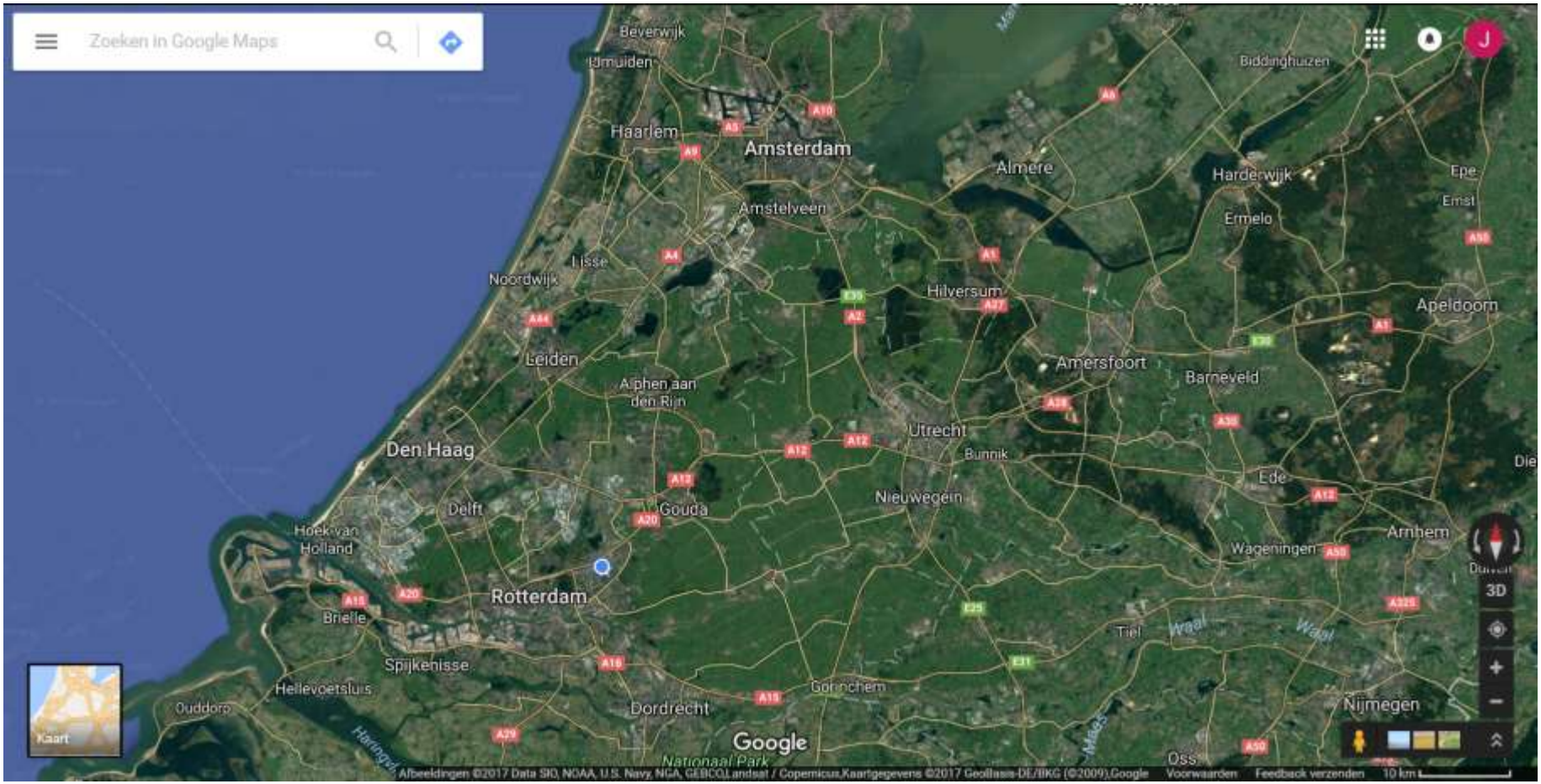
This optimization problem minimizes the effort needed to make a given graph become Eulerian.

You will see it again later and learn its name: a matching problem.

Although these matching problems can be efficiently solved on general graphs some graphs called bipartite allow for an easier algorithm.

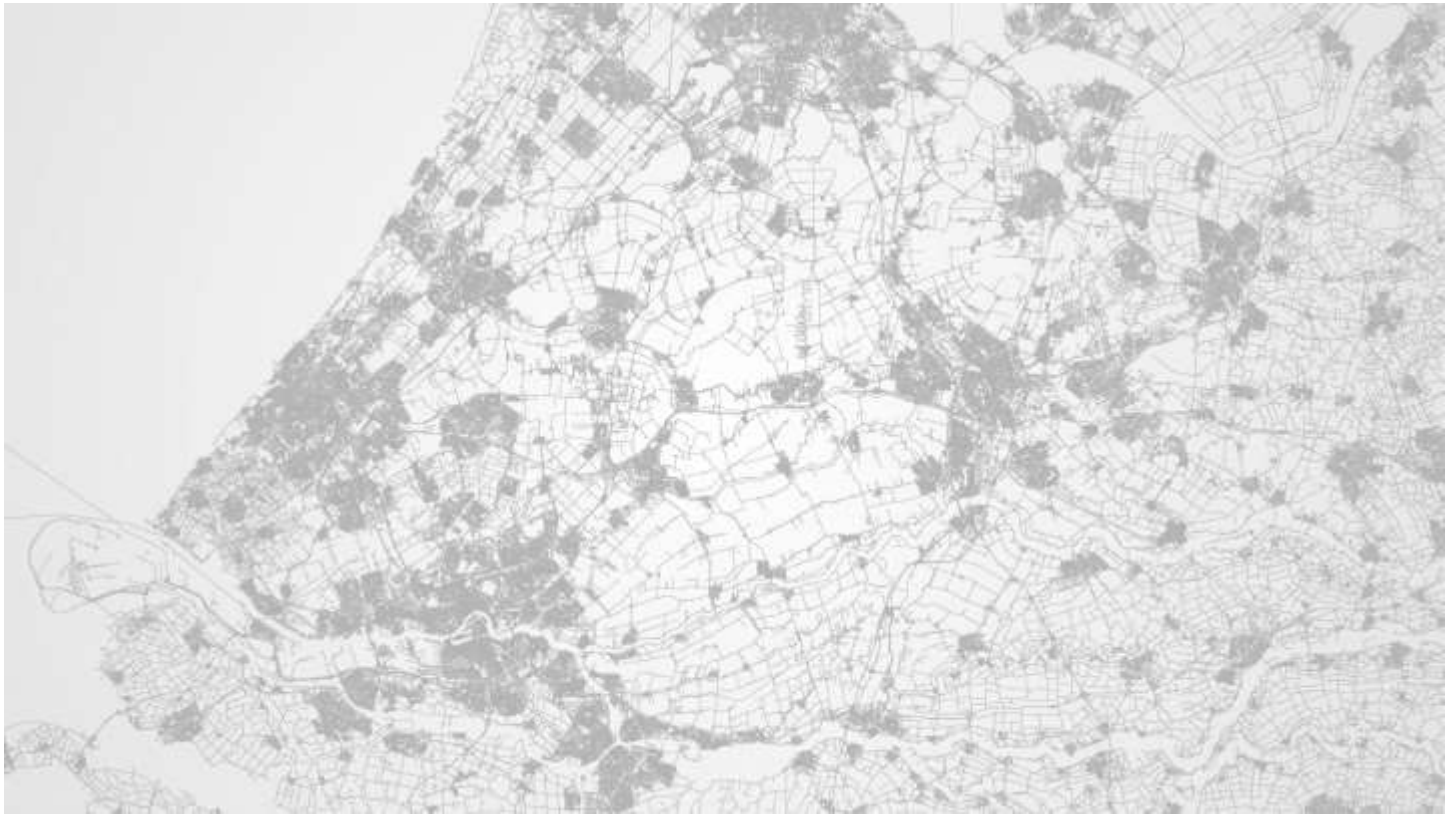
# What else do we get form graphs?

# Reality


Zoeken in Google Maps
Beverwijk
IJmuiden
Haarlem
Amsterdam
Almere
Harderwijk
Biddinghuizen
Epe
Ermelo
Amstelveen
Noordwijk
Hilversum
Apeldoorn
Leiden
Amersfoort
Barneveld
Alphen aan den Rijn
Utrecht
Den Haag
Bunnik
Ede
Nieuwegein
Delft
Gouda
Arnhem
Hoek van Holland
Wageningen
Rotterdam
Brielle
Tiel
Spijkenisse
Gorinchem
Hellevoetsluis
Dordrecht
Nijmegen
Ouddorp
Oss
Google
Nationaal Park
Kaart
3D

# Simplification

# Abstraction



# Roadmaps are large graphs

# This is an optimization problem!

There are usually (many!) alternative ways to go from our origin to destination.

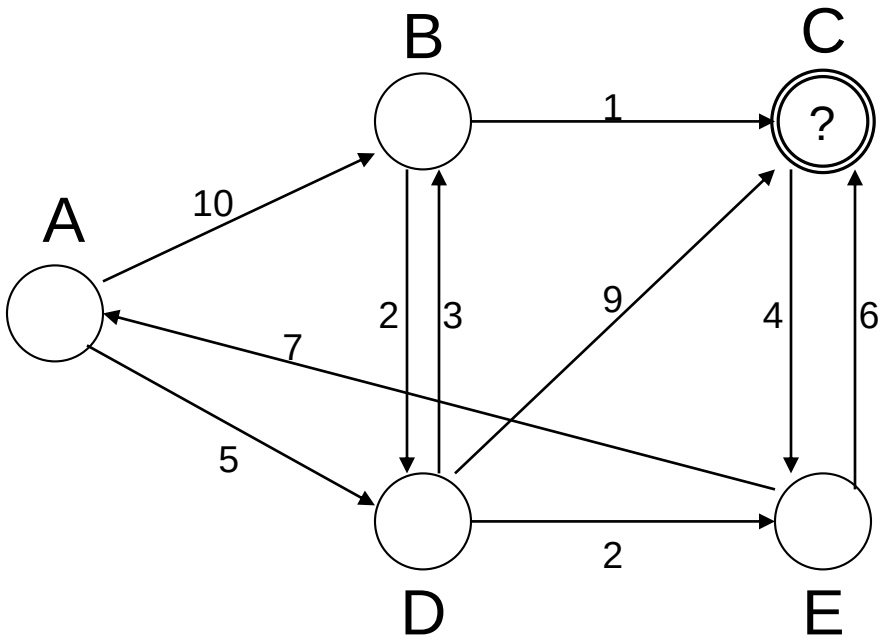
We want to find the ‘best’ defined as fastest or shortest

Hopefully not by computing and comparing all the possibilities!

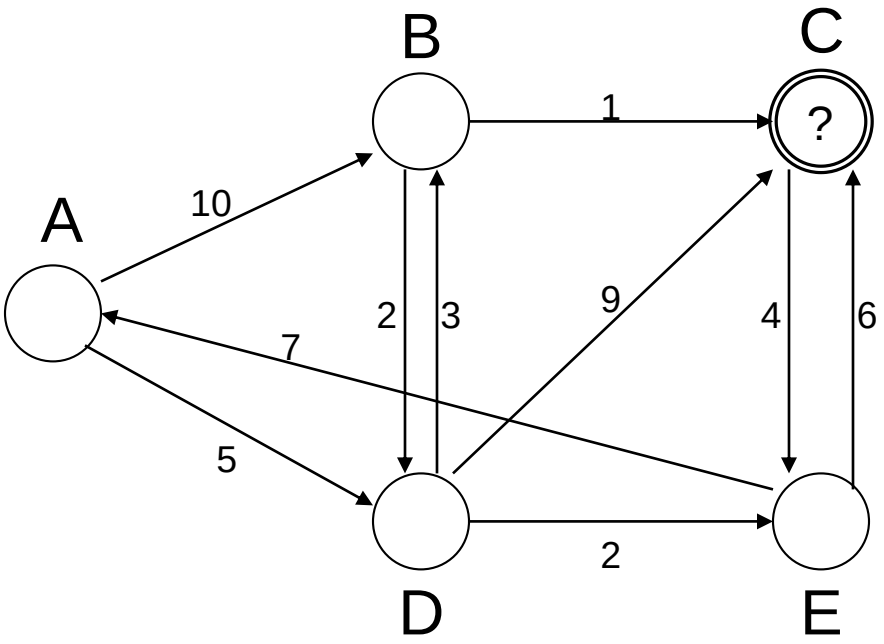
# question

can you estimate the number of alternative paths between two nodes for some types of graphs?

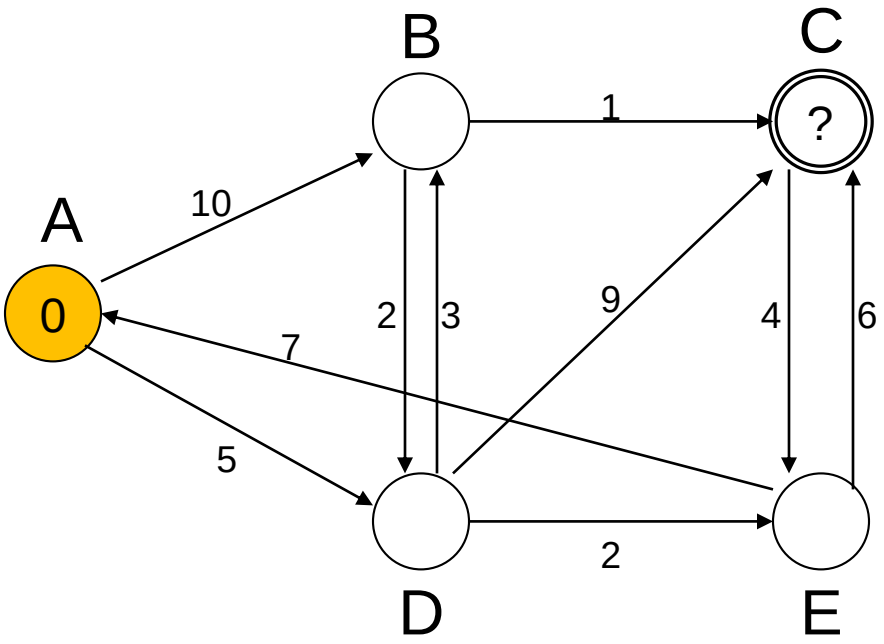
# Shortest path from A to C



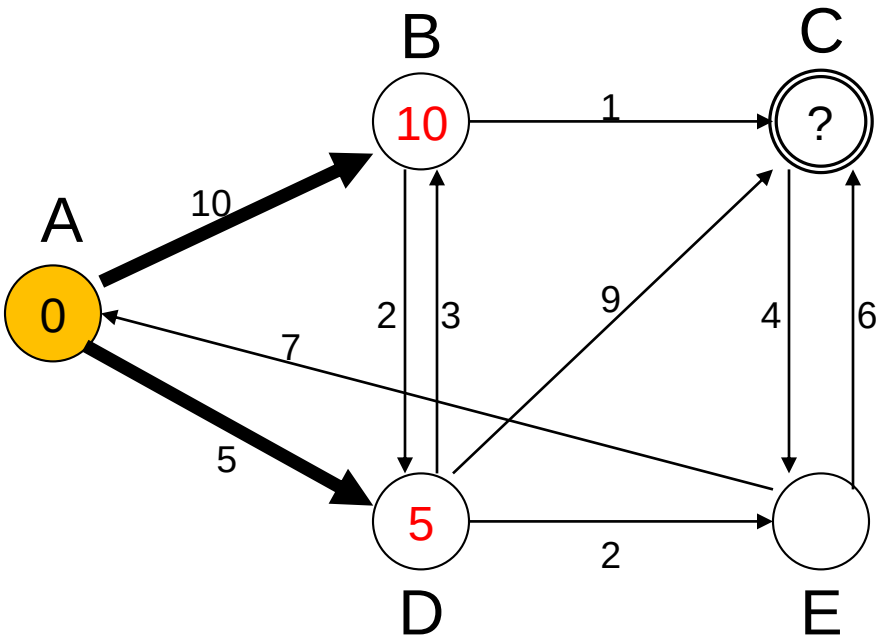
# Edsger Wybe Dijkstra (1930 –2002)



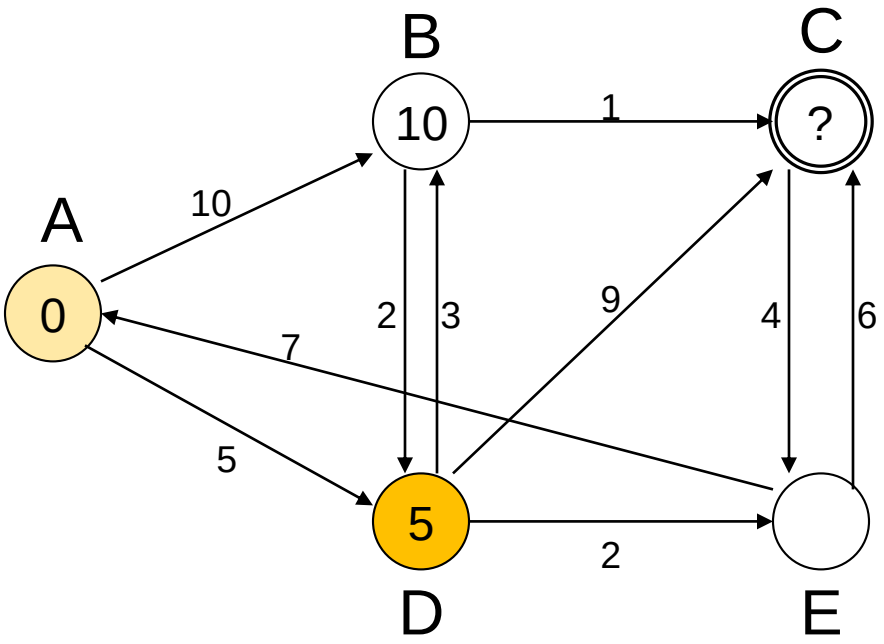
# Edsger Wybe Dijkstra (1930 –2002)



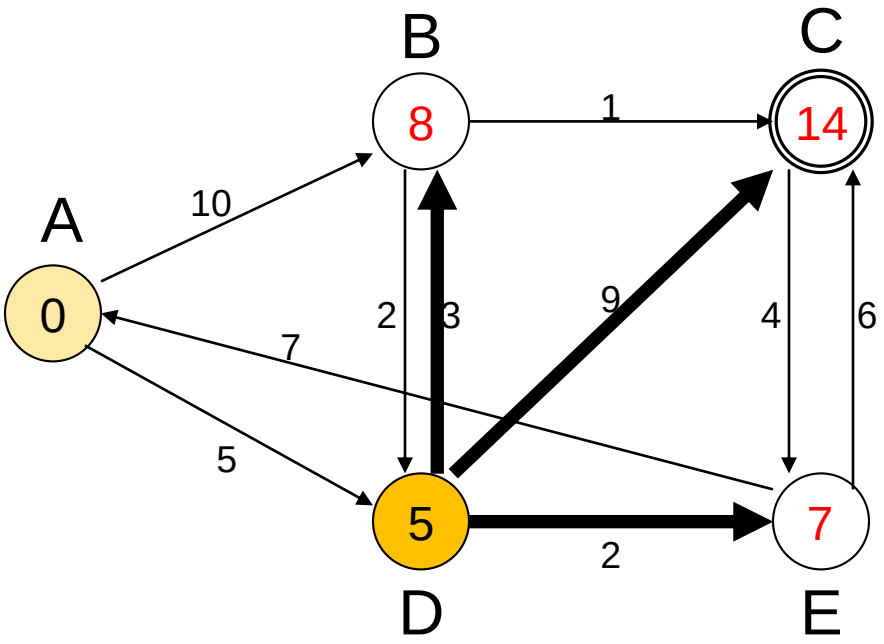
# Edsger Wybe Dijkstra (1930 –2002)



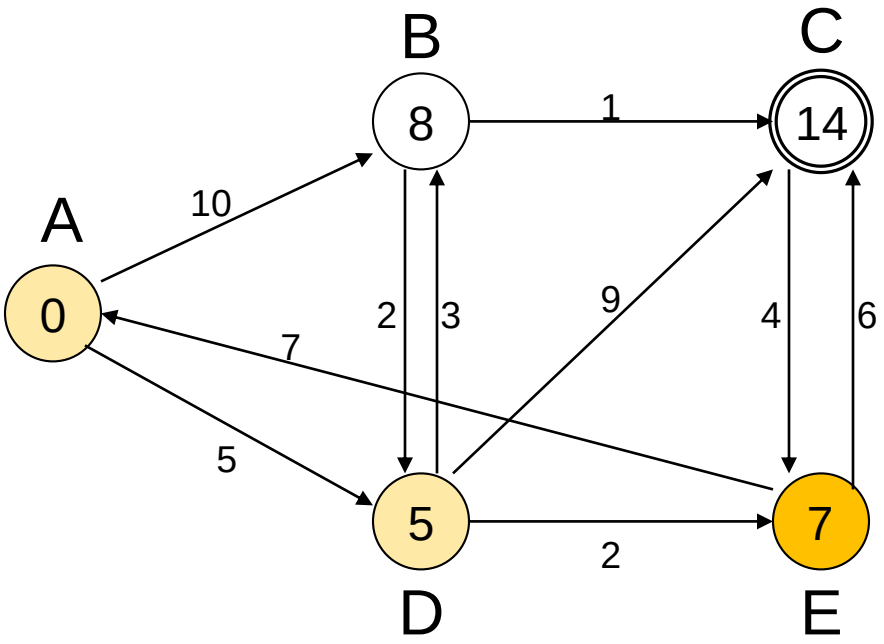
# Edsger Wybe Dijkstra (1930 –2002)



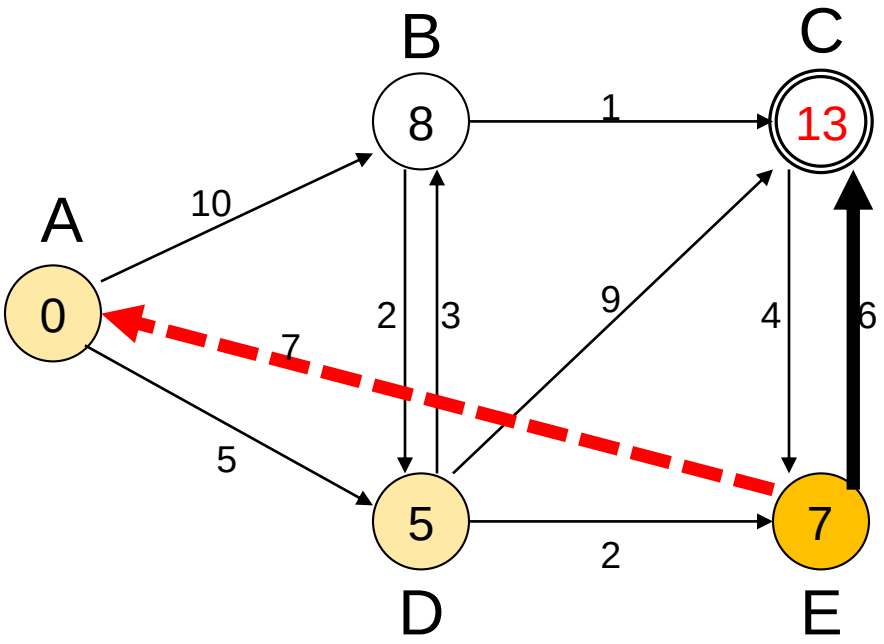
# Edsger Wybe Dijkstra (1930 –2002)



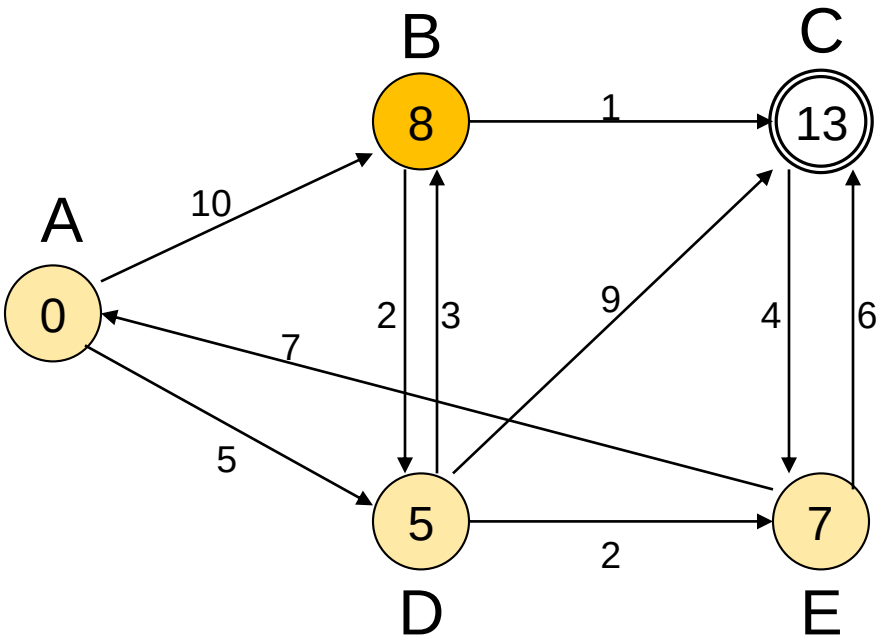
# Edsger Wybe Dijkstra (1930 –2002)



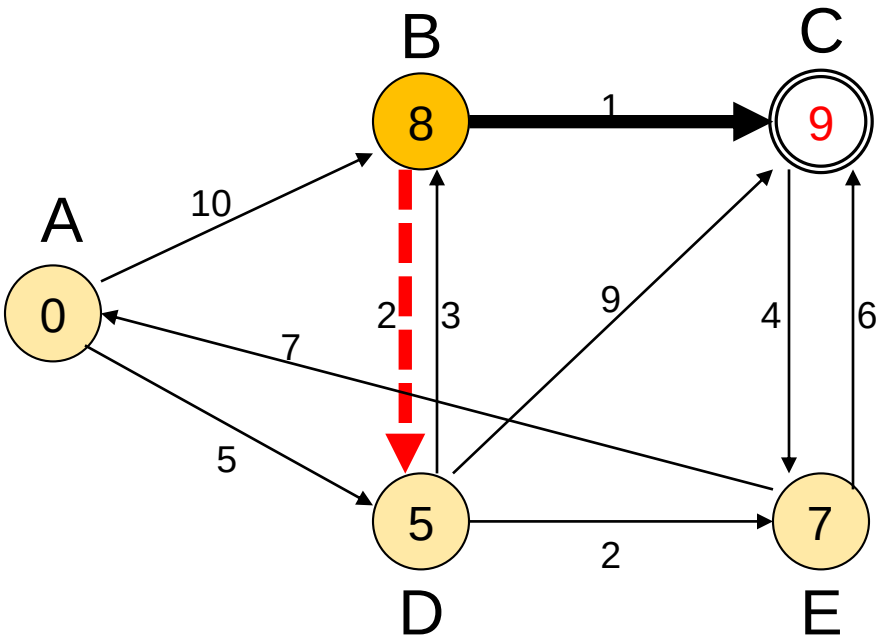
# Edsger Wybe Dijkstra (1930 –2002)



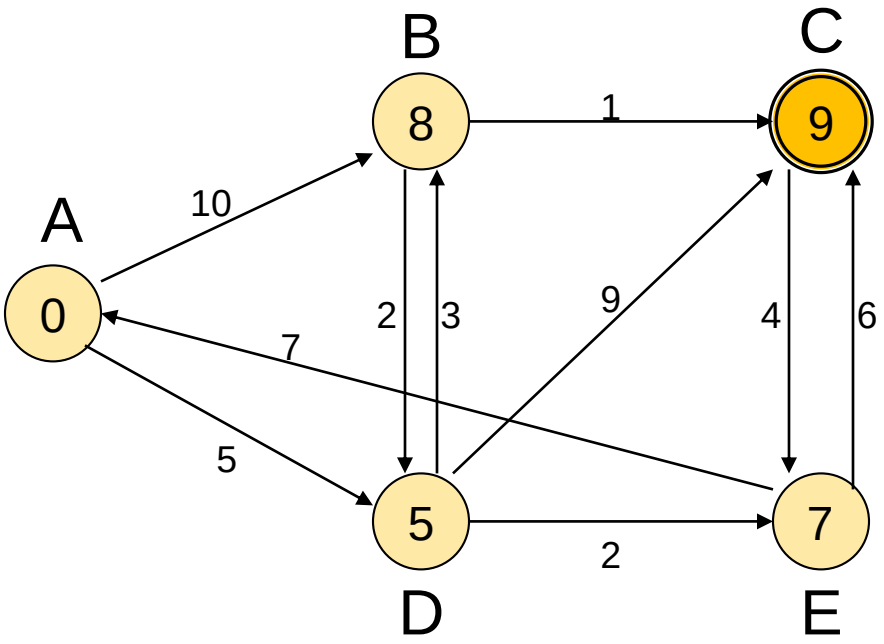
# Edsger Wybe Dijkstra (1930 –2002)



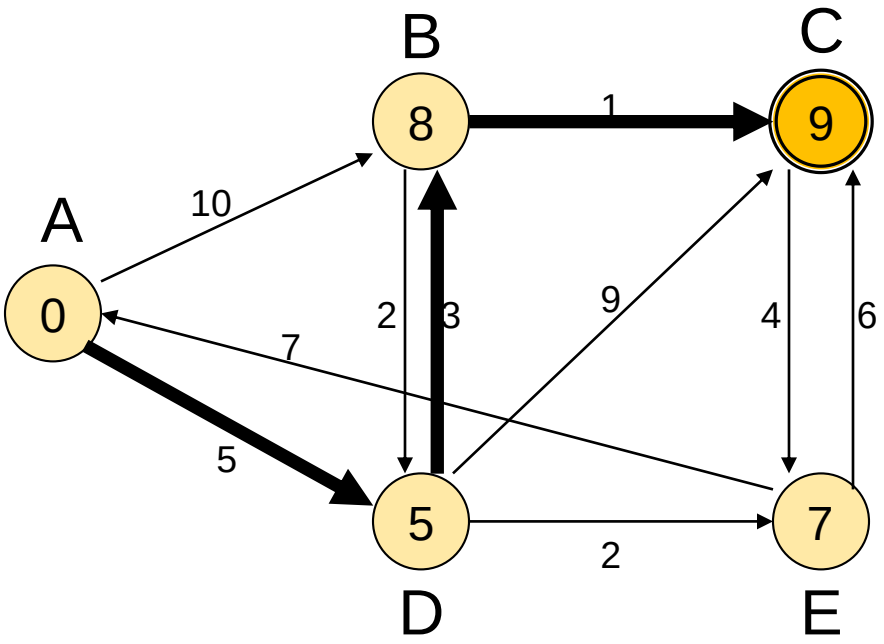
# Edsger Wybe Dijkstra (1930 –2002)



# Edsger Wybe Dijkstra (1930 –2002)



# Edsger Wybe Dijkstra (1930 –2002)



# question

Can you list the assumptions one needs to make about the given graph in order for Dijkstra's algorithm to guarantee finding the optimal solution?

Can you estimate the effort needed to apply this algorithm on a graph with $n$ nodes and $m$ arcs?

# notes

It is clearly important that on each round of the algorithm (we will call these iterations) exactly one node becomes definitely labeled.

Therefore the number of iterations does not exceed the number of nodes.

We need now to estimate the effort of each iteration!

There are many answers depending on the data structures used! Please pay attention to this!

The design of efficient algorithms is a challenge that goes beyond the proof of correctness

# Let us meet the Greedy paradigm

minimum spanning trees

03-02-2020

# Greedy algorithms

A greedy algorithm extends its wealth (the solution being constructed) by taking the step with highest immediate gain.

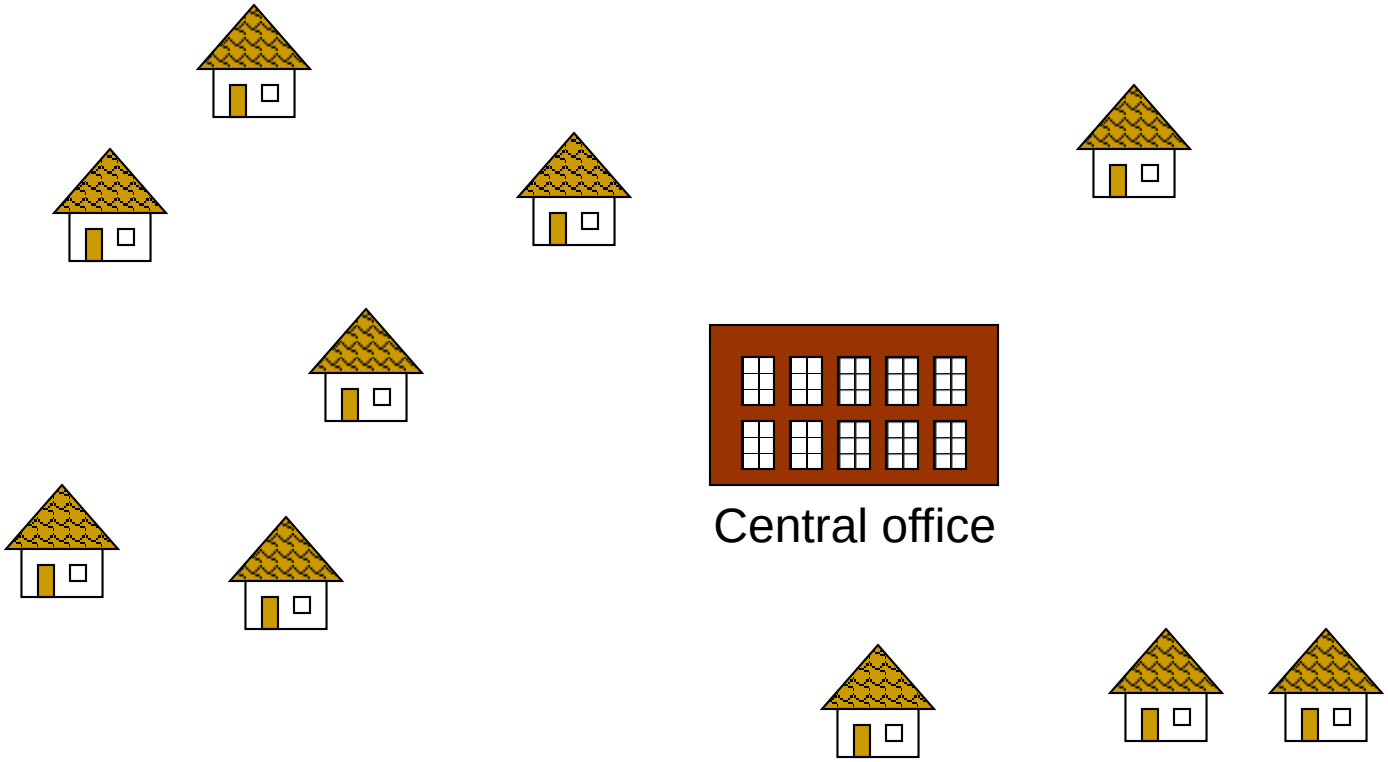
Despite being well-known that greedy algorithms are often sub-optimal, today we see one case where they are optimal!

# Greedy **can be** optimal!

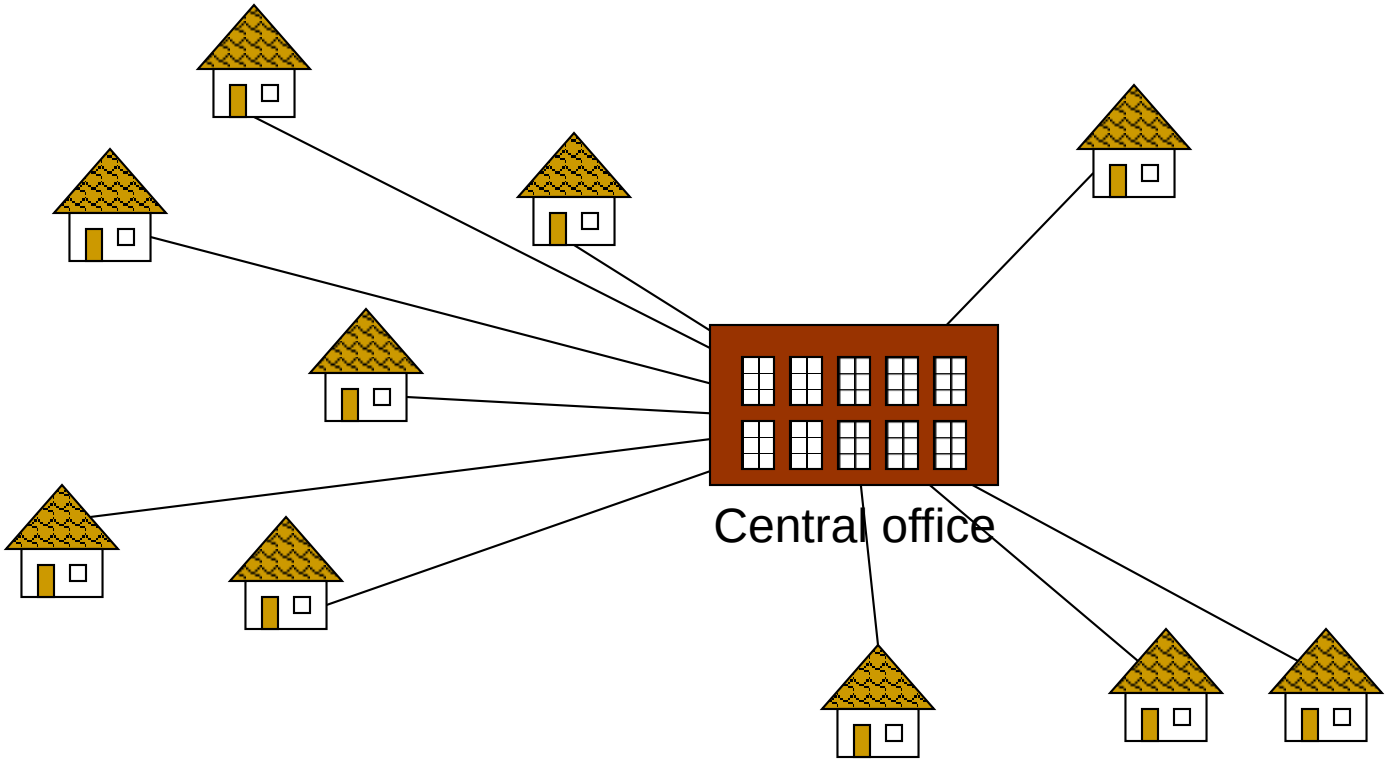
In fact, there is a whole collection of optimization problems defined in clean mathematical terms for which greedy is optimal: the matroids (for another course).

The most famous and easy to explain example is the so-called Minimal Spanning Tree.

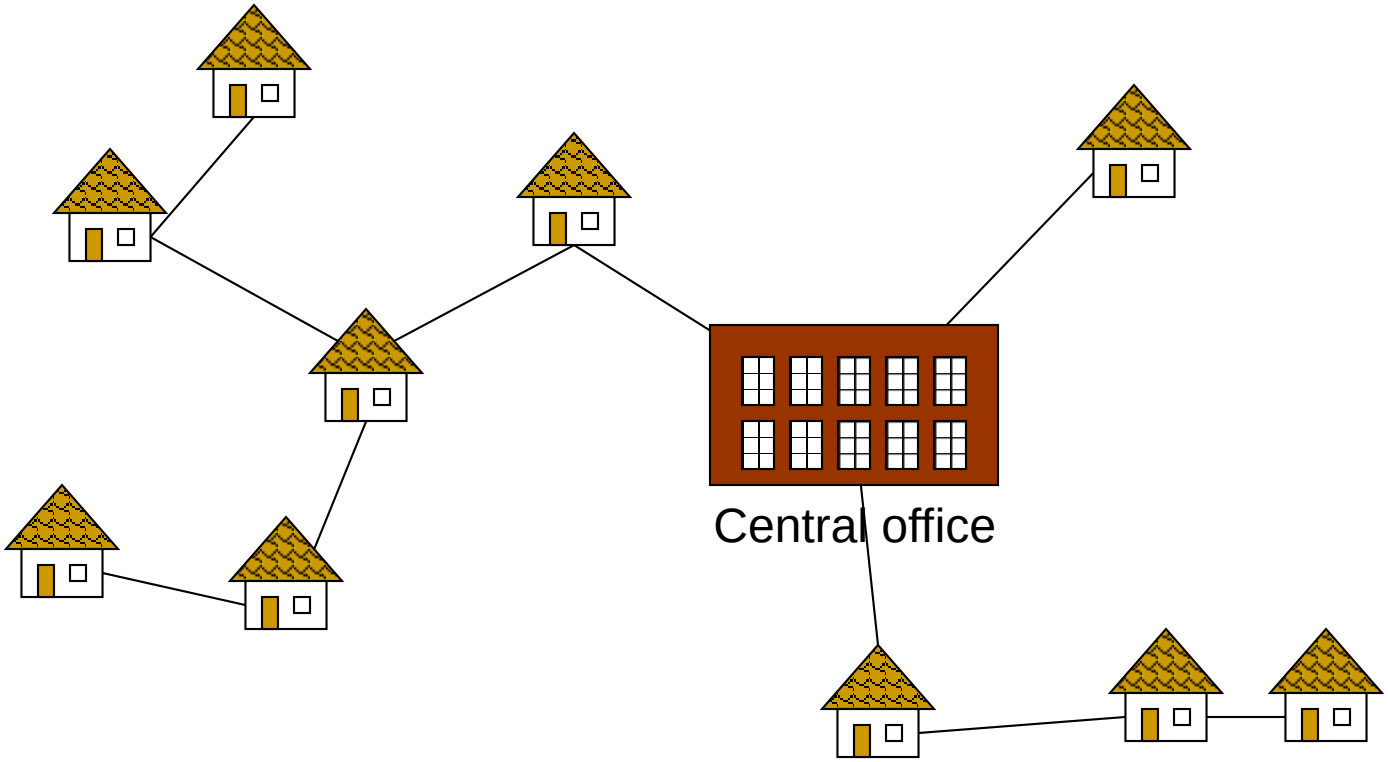
# Problem: laying cables



# Expensive: dedicated connection



# A better solution



# Spanning Trees

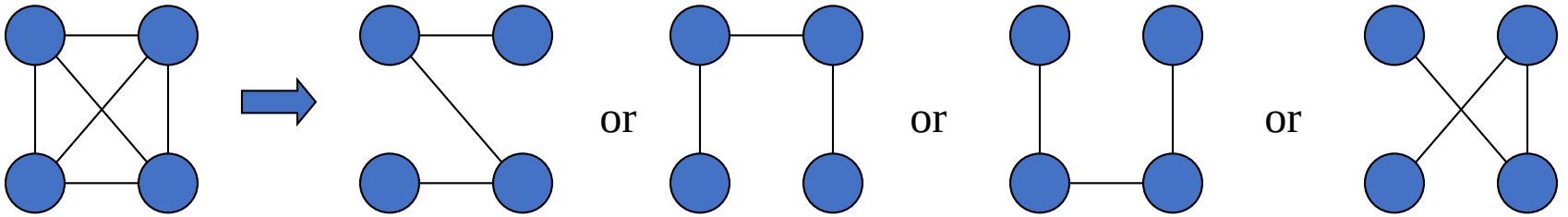
A tree is a connected graph without cycles.

A spanning tree of a graph is tree that contains all the vertices of the graph.

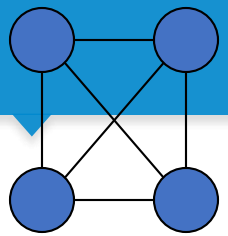
A graph may have many spanning trees.

Graph A

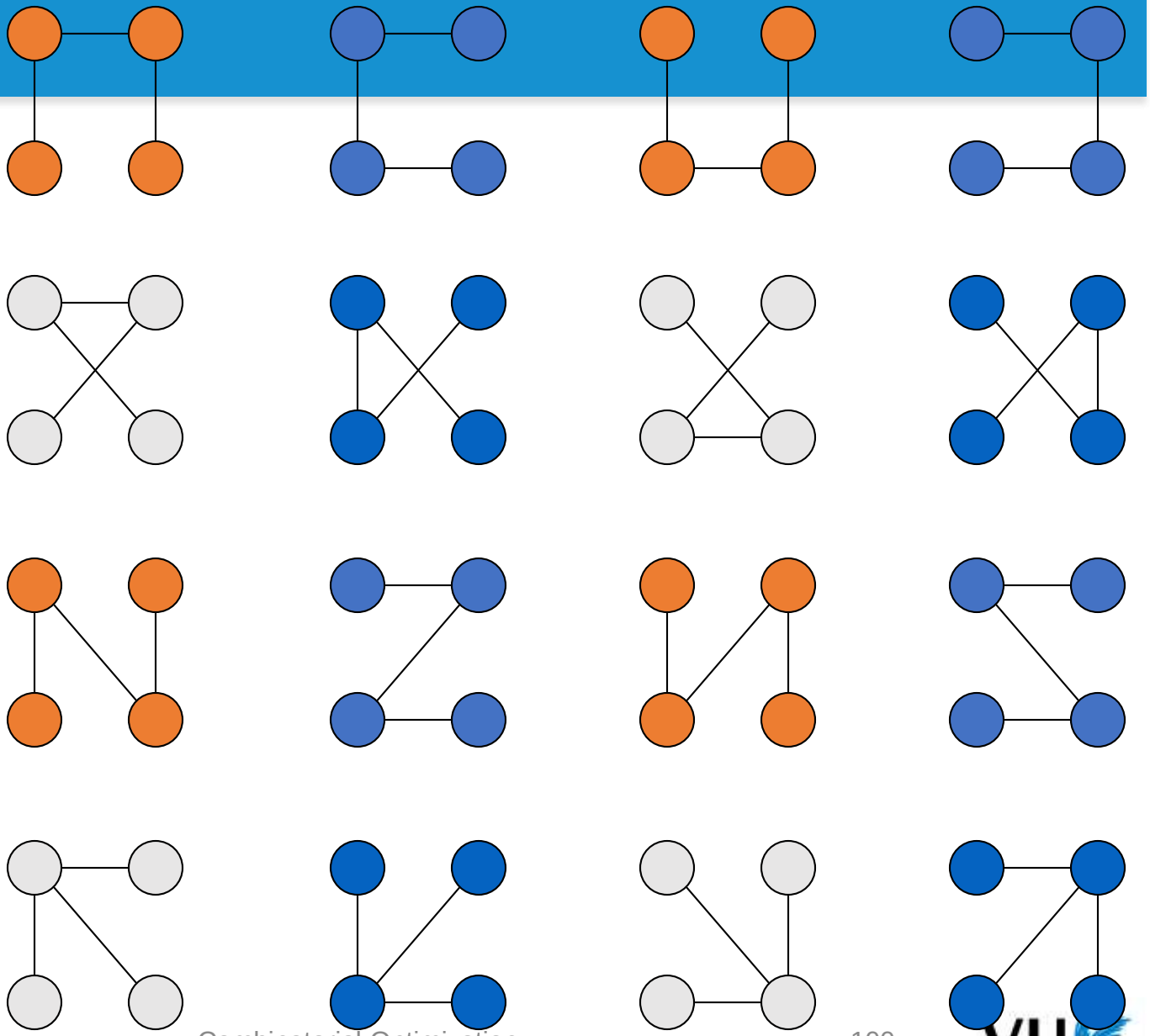
Some Spanning Trees from Graph A



# Complete Graph



# All 16 of its Spanning Trees



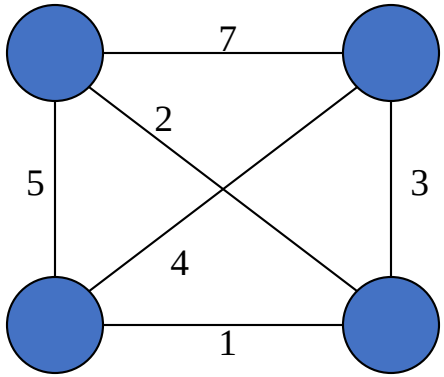
# question

Can you give an upper bound on the number of spanning trees of a given graph?

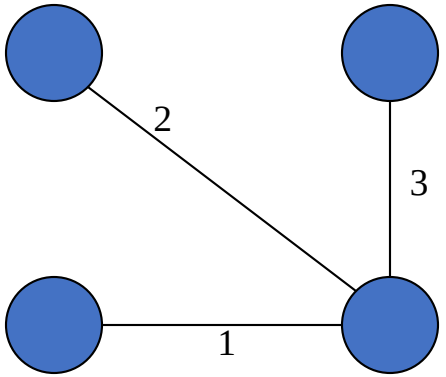
# Minimum Spanning Trees

The Minimum Spanning Tree for a given graph is the Spanning Tree of minimum cost for that graph.

Complete Graph




Minimum Spanning Tree



# notes

Finding the minimum spanning tree of a connected undirected graph is very efficient.

A greedy algorithm (i.e. an algorithm expanding a partial solution in the best way as perceived at that moment) is optimal in this case.

Several different algorithms may have different efficiencies though…

# On the book Algorithms

First section in chapter 5 on Greedy algorithms

# Algorithms for Minimum Spanning Tree

Kruskal's Algorithm

Prim's Algorithm

# Kruskal's Algorithm

This algorithm creates a forest of trees. Initially the forest consists of n single node trees (and no edges). At each step, we add one edge (the cheapest one) so that it joins two trees together. If it were to form a cycle, it would simply link two nodes that were already part of a single connected tree, so that this edge would not be needed.

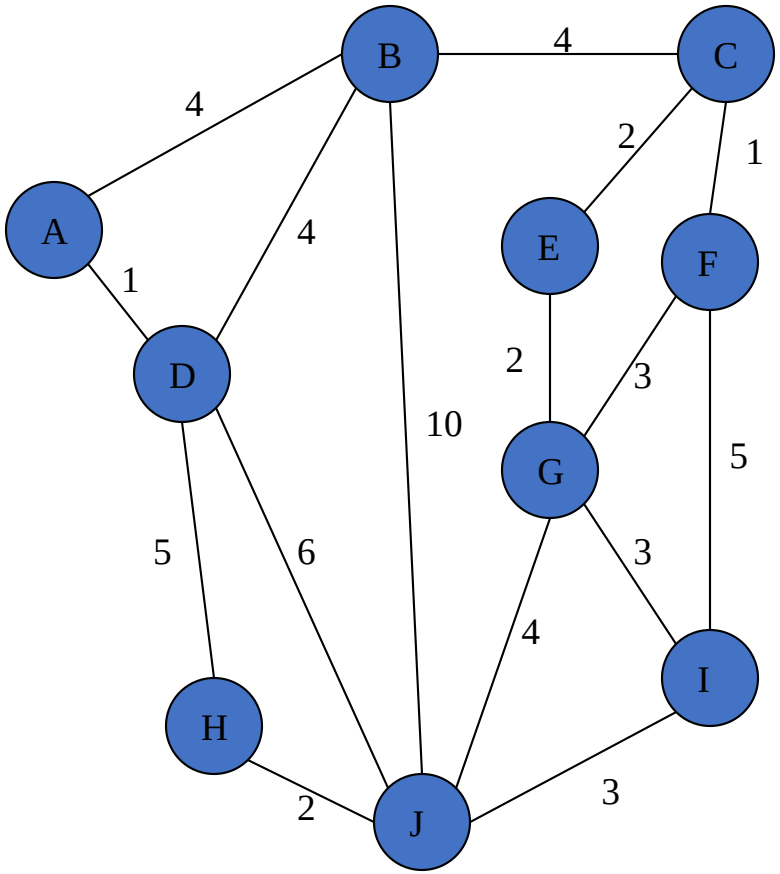
# Kruskal's Algorithm

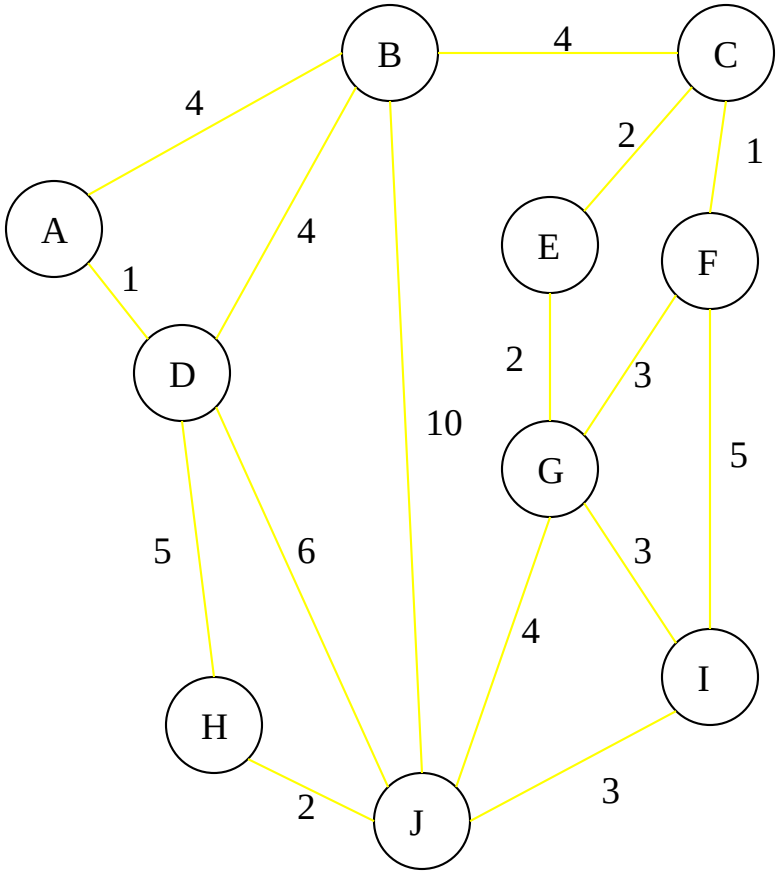
The steps are:

1. The forest is constructed - with each node in a separate tree.
2. The edges are placed in a priority queue.
3. Until we've added $n - 1$ edges,
    1. Extract the cheapest edge from the queue,
    2. If it forms a cycle, reject it,
    3. Else add it to the forest. Adding it to the forest will join two trees together.

Every step will have joined two trees in the forest together, so that at the end, there will only be one tree.

# Given Graph






A 4 B

A 1 D

B 4 C

B 4 D

B 10 J

C 2 E

C 1 F

D 5 H

D 6 J

E 2 G

F 3 G

F 5 I

G 3 I

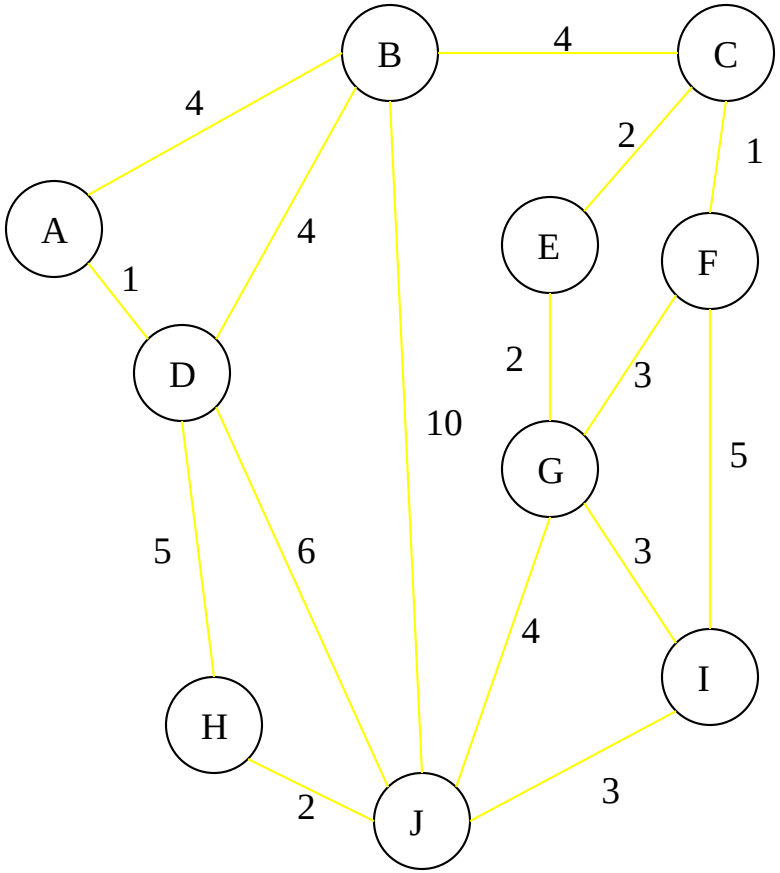
G 4 J

H 2 J

I 3 J

# Sort Edges

(in reality they are placed in a priority queue - not sorted - but sorting them makes the algorithm easier to visualize)




A — 1 — D

C — 1 — F

C — 2 — E

E — 2 — G

H — 2 — J

F — 3 — G

G — 3 — I

I — 3 — J

A — 4 — B

B — 4 — D

B — 4 — C

G — 4 — J

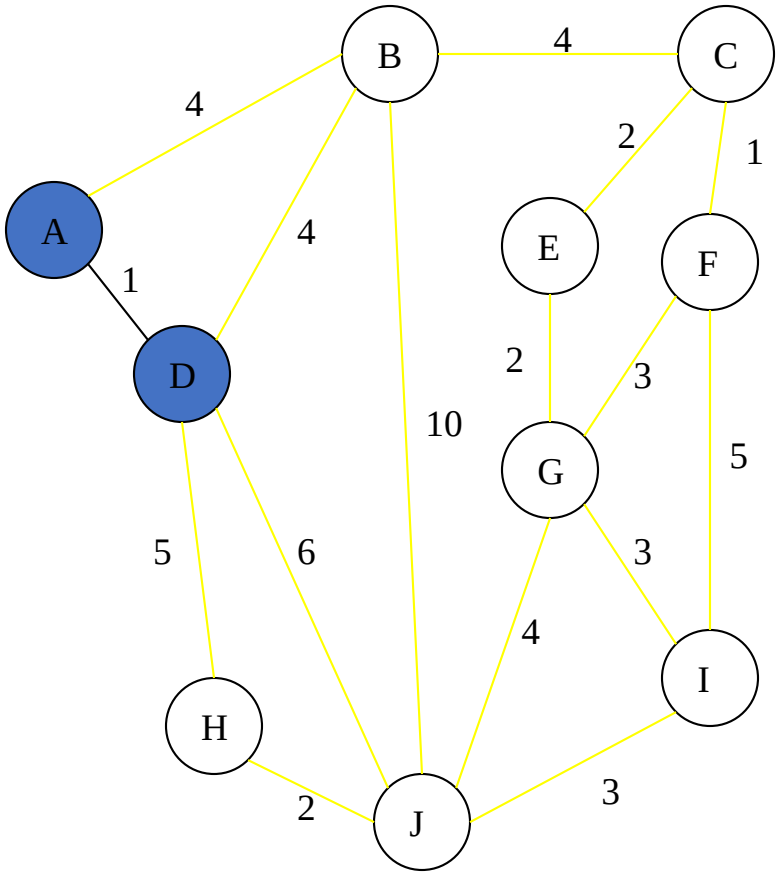
F — 5 — I

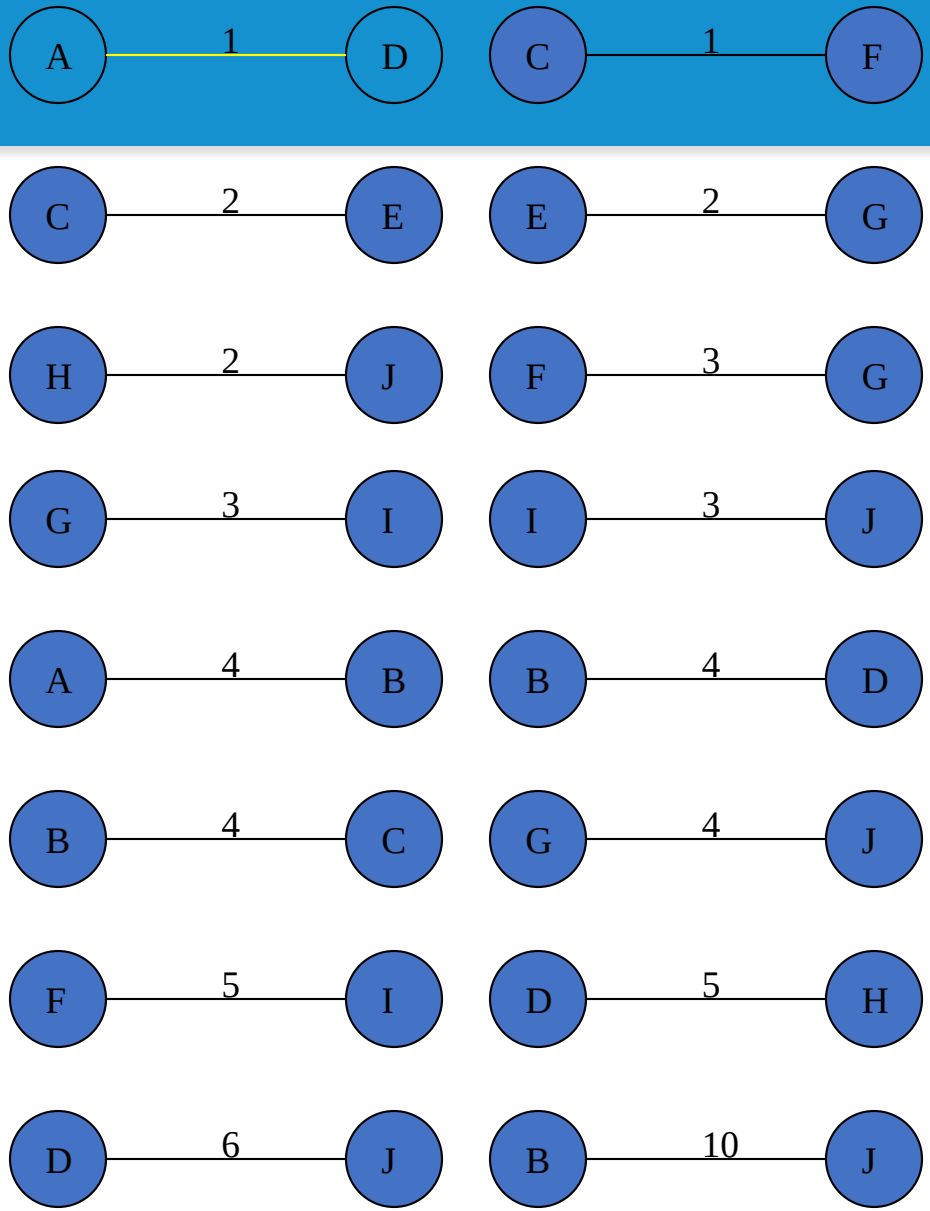
D — 5 — H

D — 6 — J

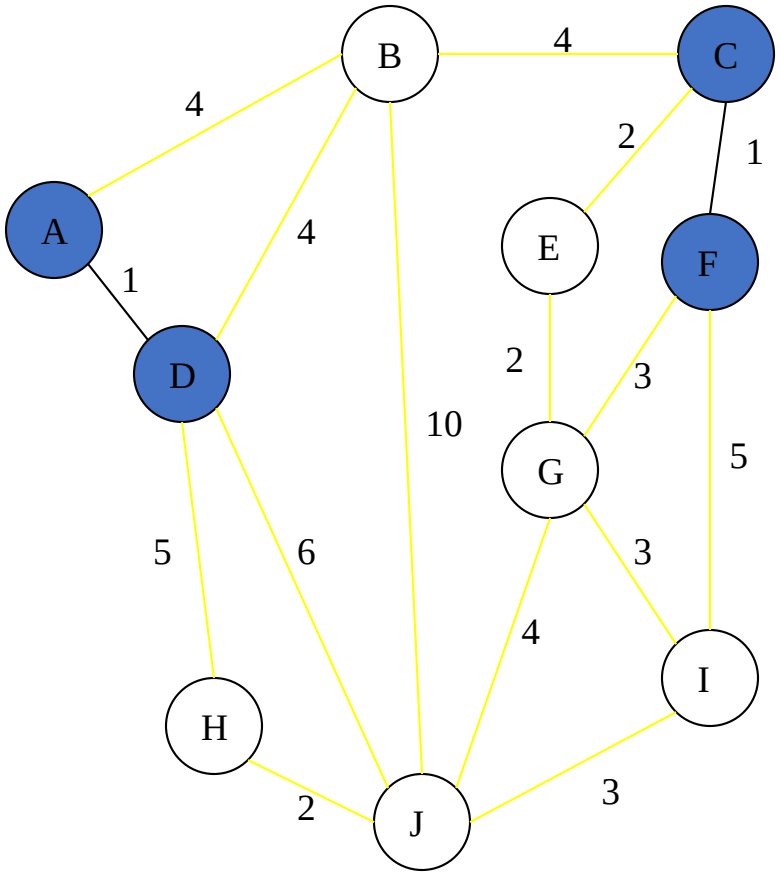
B — 10 — J

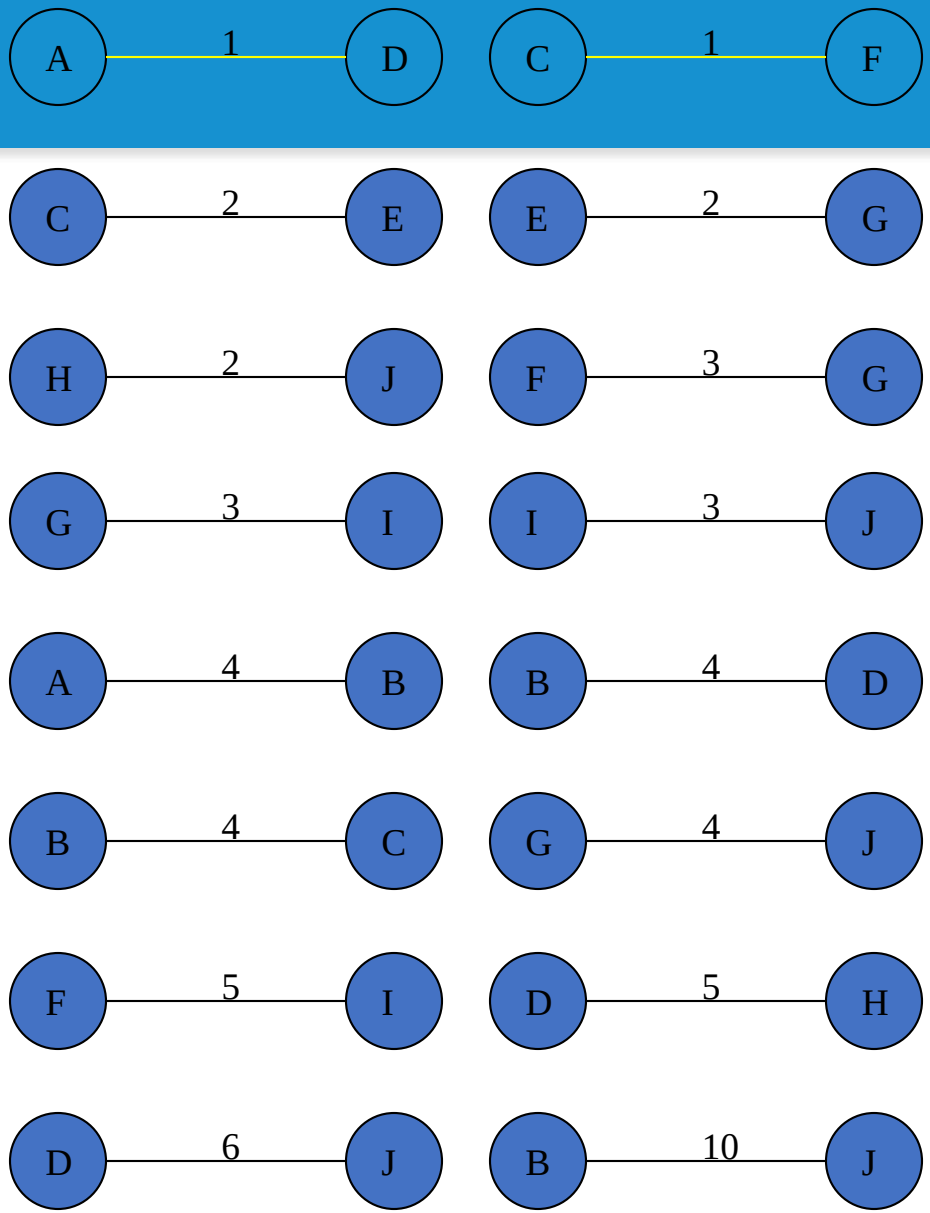
# Add Edge


A
B
C
D
E
F
G
H
I
J
4
4
1
4
2
1
2
3
10
5
5
6
3
4
3
2
3

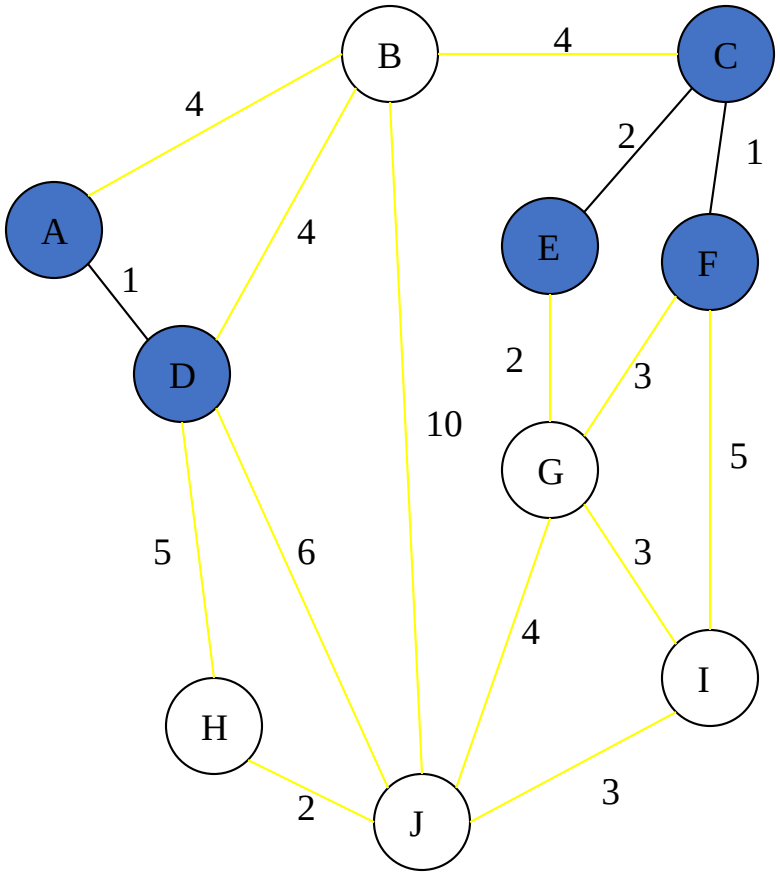

A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

# Add Edge


A
B
C
D
E
F
G
H
I
J
4
4
2
1
1
4
2
3
10
5
5
6
3
4
3
2
3


A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

# Add Edge




A — 1 — D

C — 1 — F

C — 2 — E

E — 2 — G

H — 2 — J

F — 3 — G

G — 3 — I

I — 3 — J

A — 4 — B

B — 4 — D

B — 4 — C

G — 4 — J

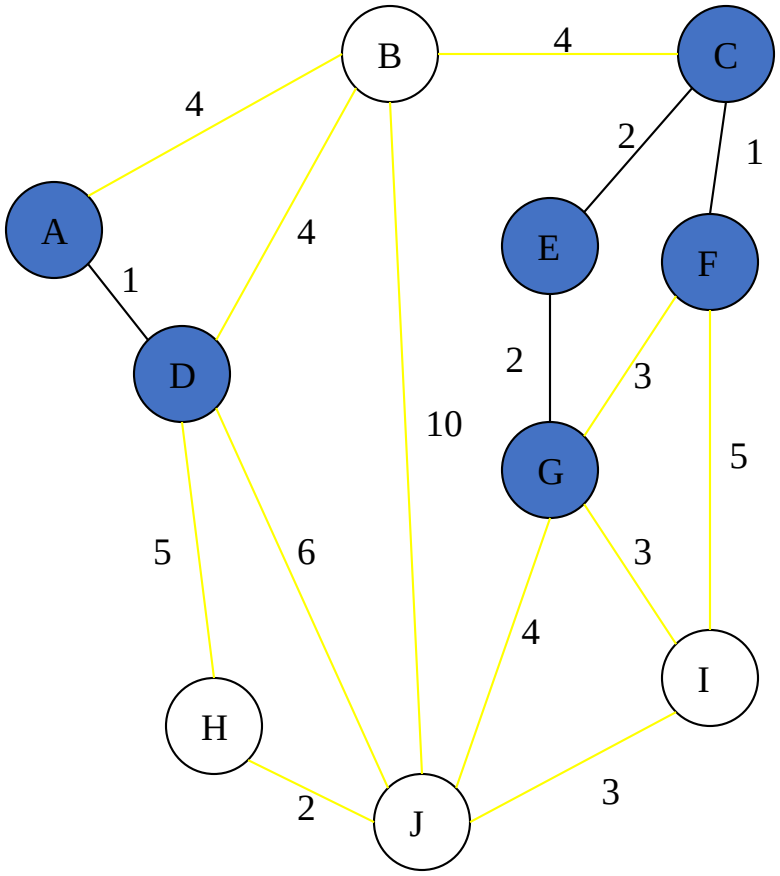
F — 5 — I

D — 5 — H

D — 6 — J

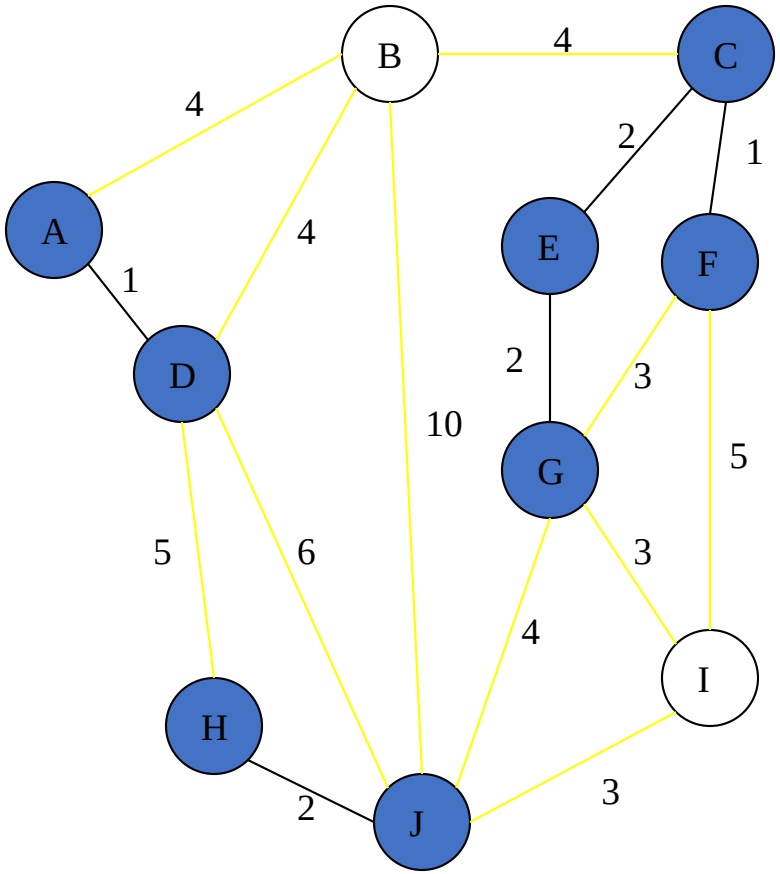
B — 10 — J

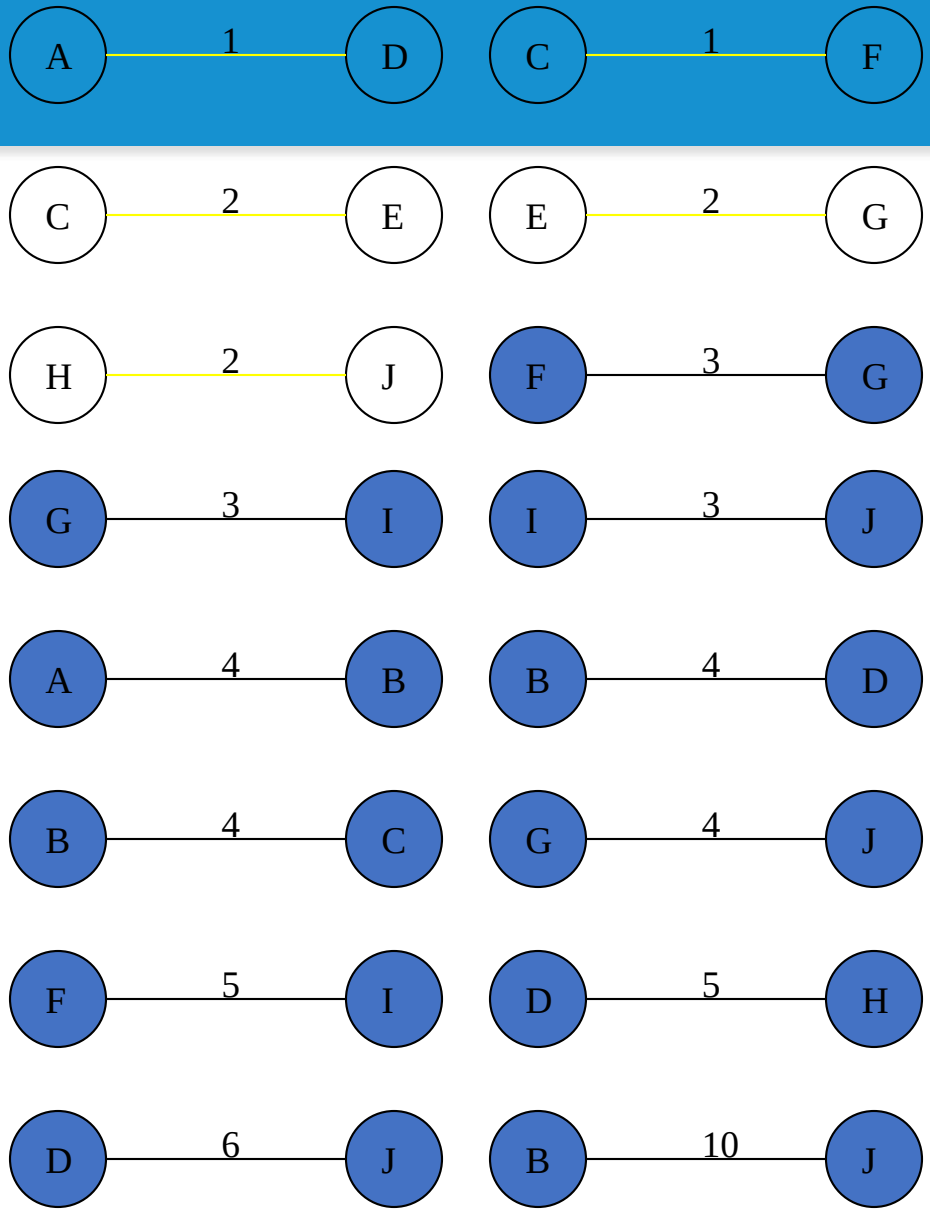
# Add Edge




| | | | | | |
|---|---|---|---|---|---|
| A | 1 | D | C | 1 | F |
| C | 2 | E | E | 2 | G |
| H | 2 | J | F | 3 | G |
| G | 3 | I | I | 3 | J |
| A | 4 | B | B | 4 | D |
| B | 4 | C | G | 4 | J |
| F | 5 | I | D | 5 | H |
| D | 6 | J | B | 10 | J |

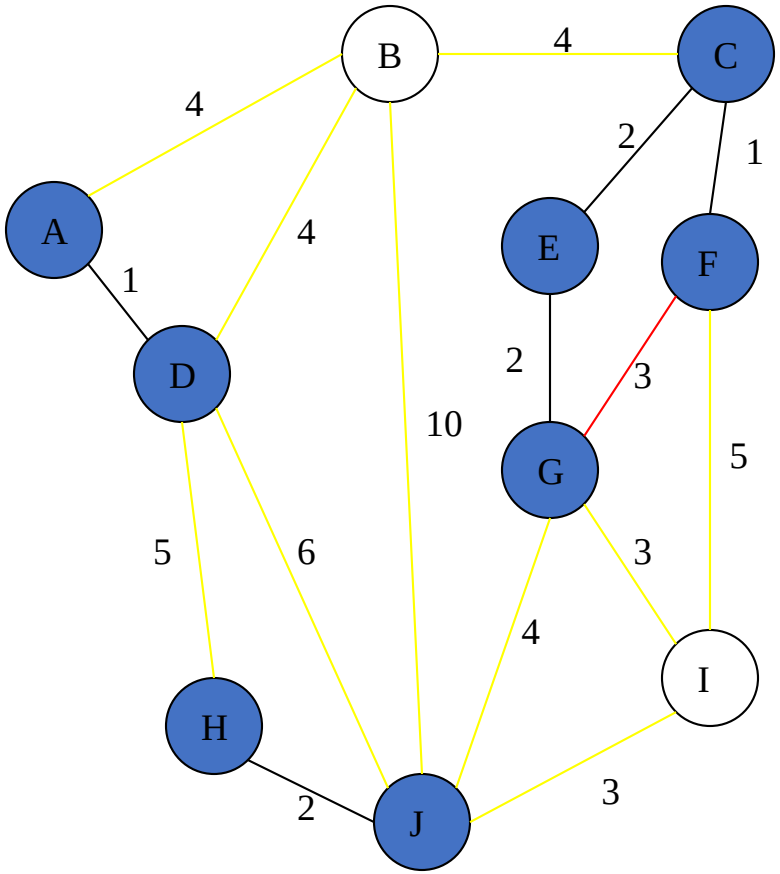
# Add Edge

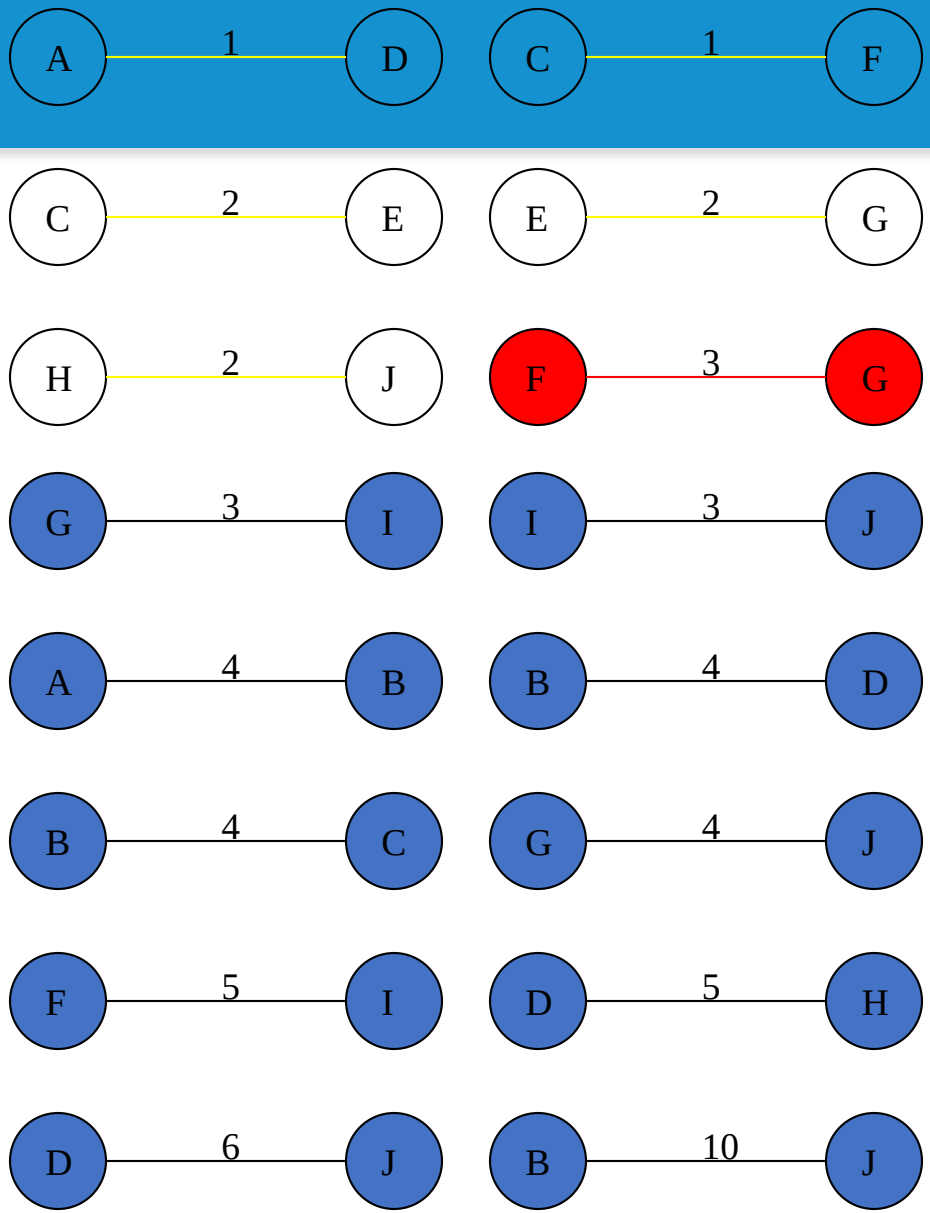

A
B
C
D
E
F
G
H
I
J
4
4
1
4
2
1
2
3
10
5
5
6
3
4
3
2
3


A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

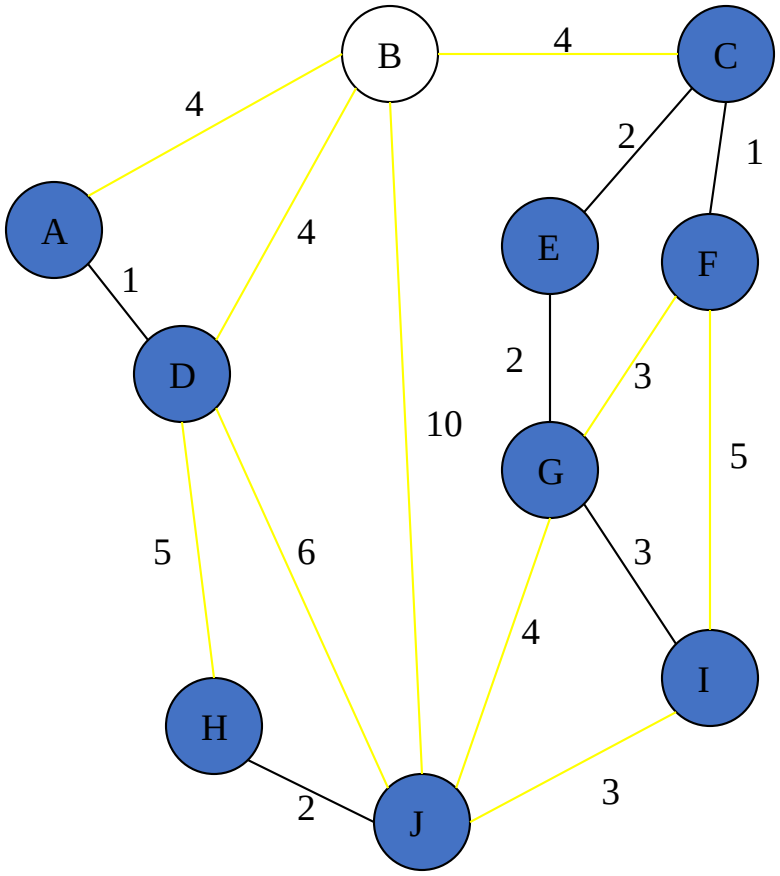
# Cycle

## Don’t Add Edge



# Add Edge




A 1 D

C 1 F

C 2 E

E 2 G

H 2 J

F 3 G

G 3 I

I 3 J

A 4 B

B 4 D

B 4 C

G 4 J

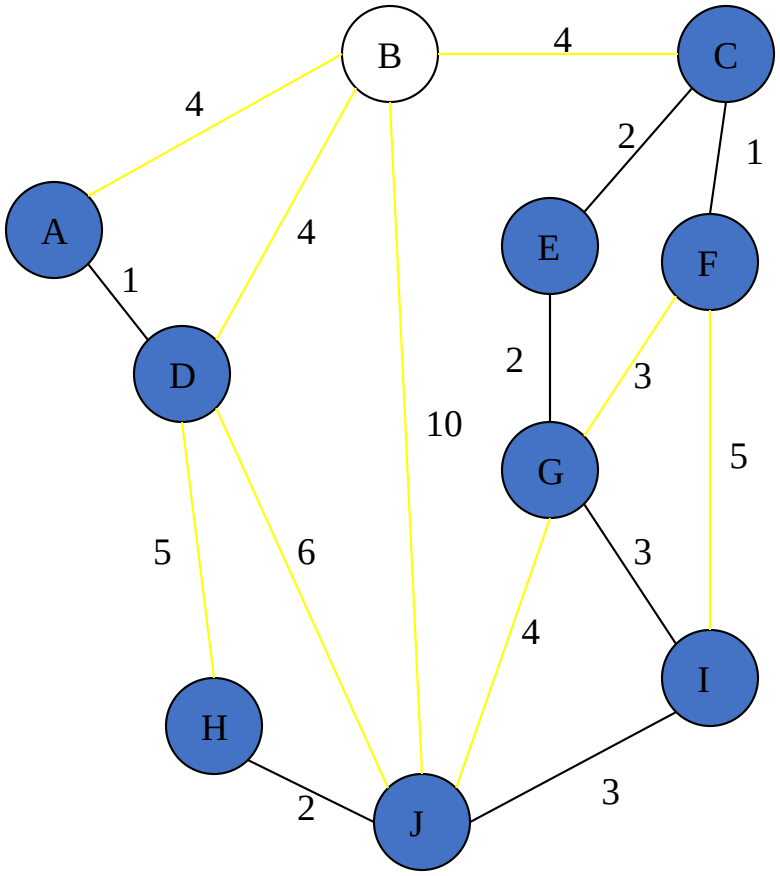
F 5 I

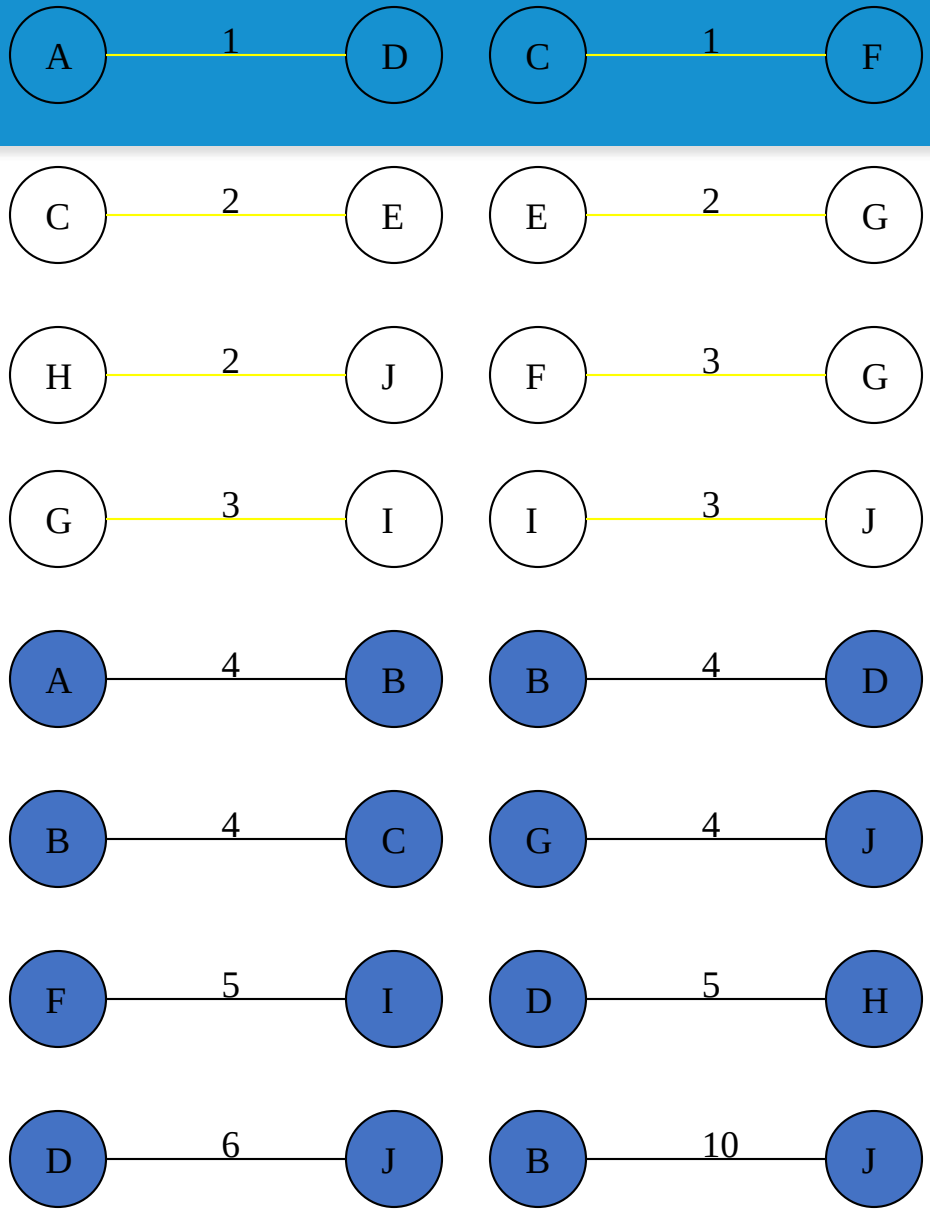
D 5 H

D 6 J

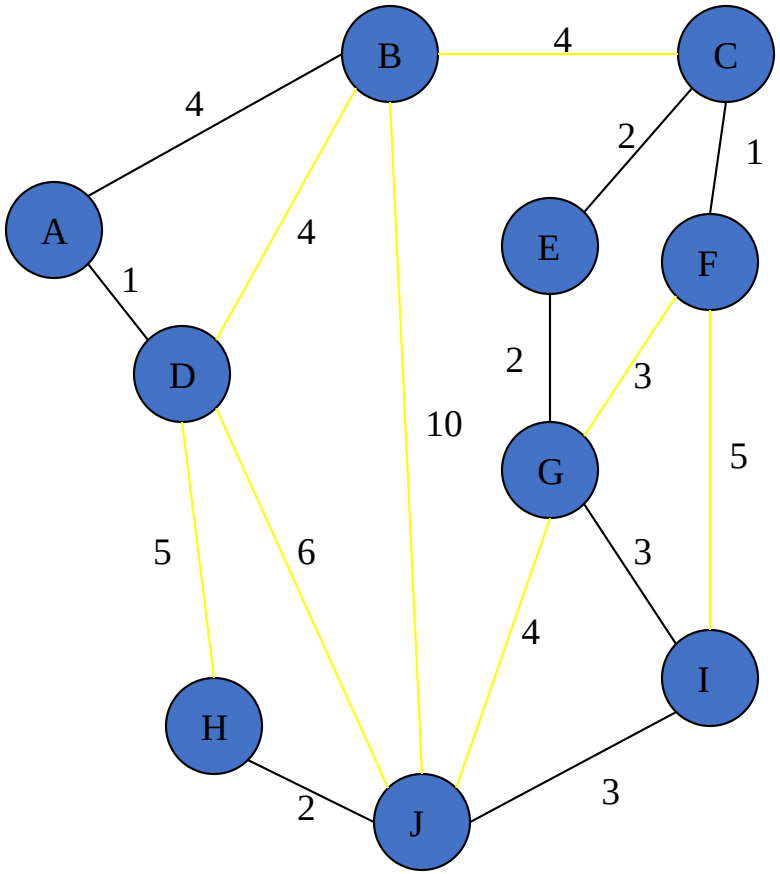
B 10 J

# Add Edge


A
B
C
D
E
F
G
H
I
J
4
4
1
4
2
1
2
3
10
5
5
6
3
4
3
2
3


A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

# Add Edge




A 1 D

C 1 F

C 2 E

E 2 G

H 2 J

F 3 G

G 3 I

I 3 J

A 4 B

B 4 D

B 4 C

G 4 J

F 5 I

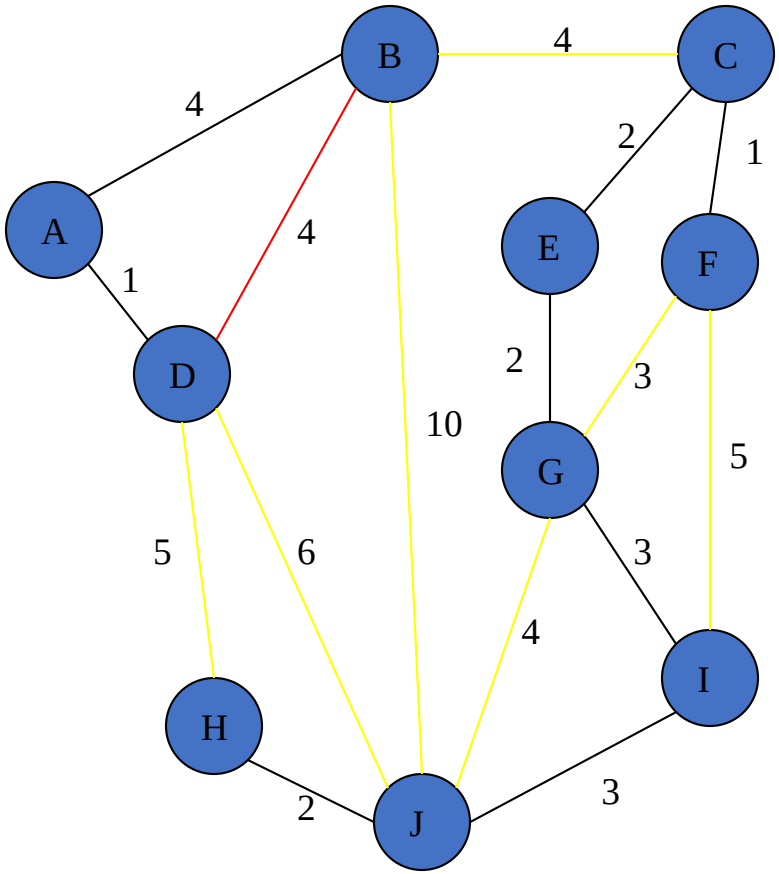
D 5 H

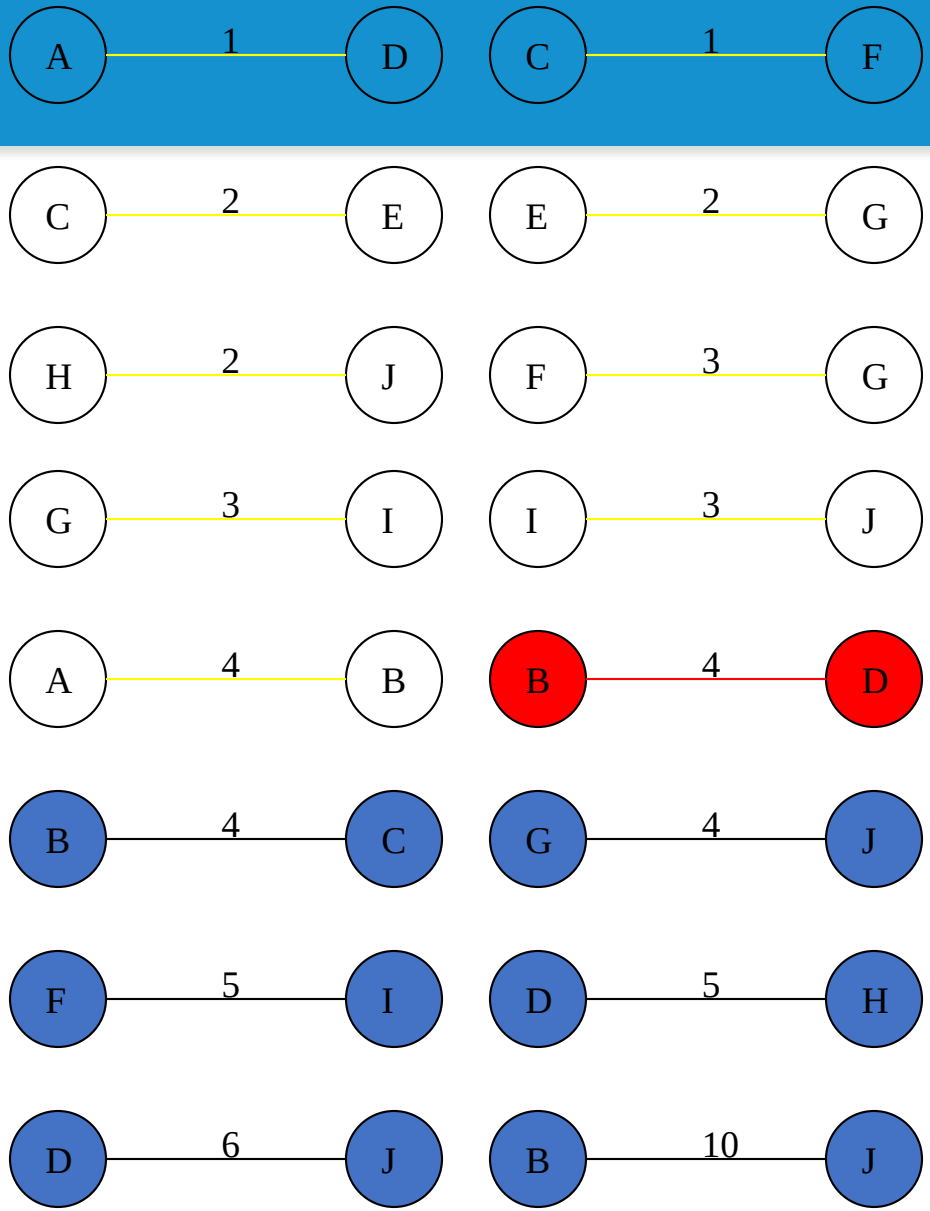
D 6 J

B 10 J

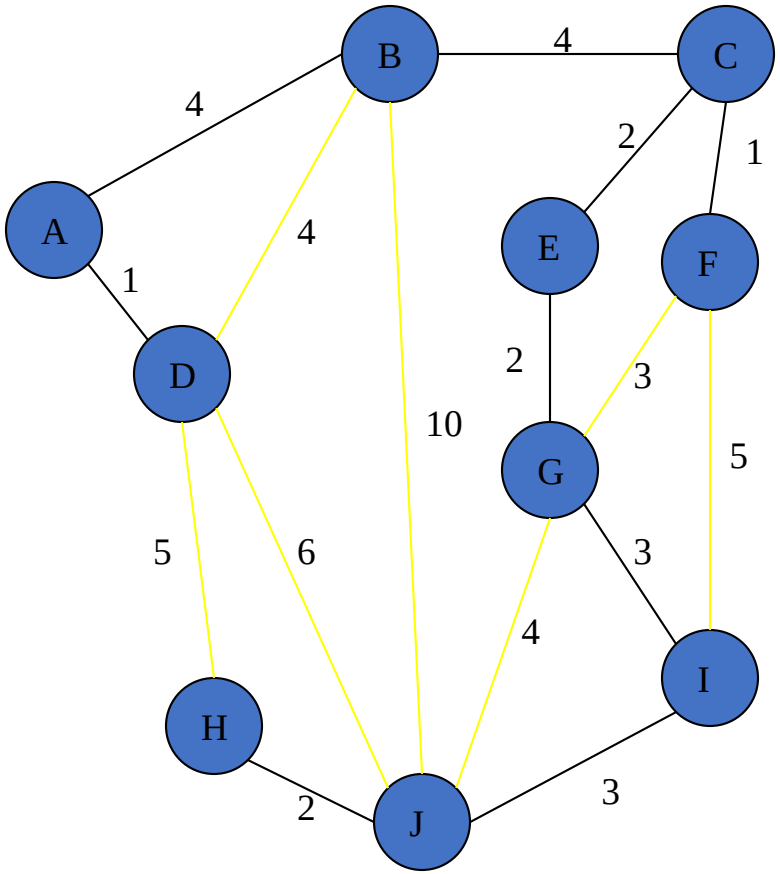
# Cycle

## Don’t Add Edge



# Add Edge




A 1 D

C 1 F

C 2 E

E 2 G

H 2 J

F 3 G

G 3 I

I 3 J

A 4 B

B 4 D

B 4 C

G 4 J

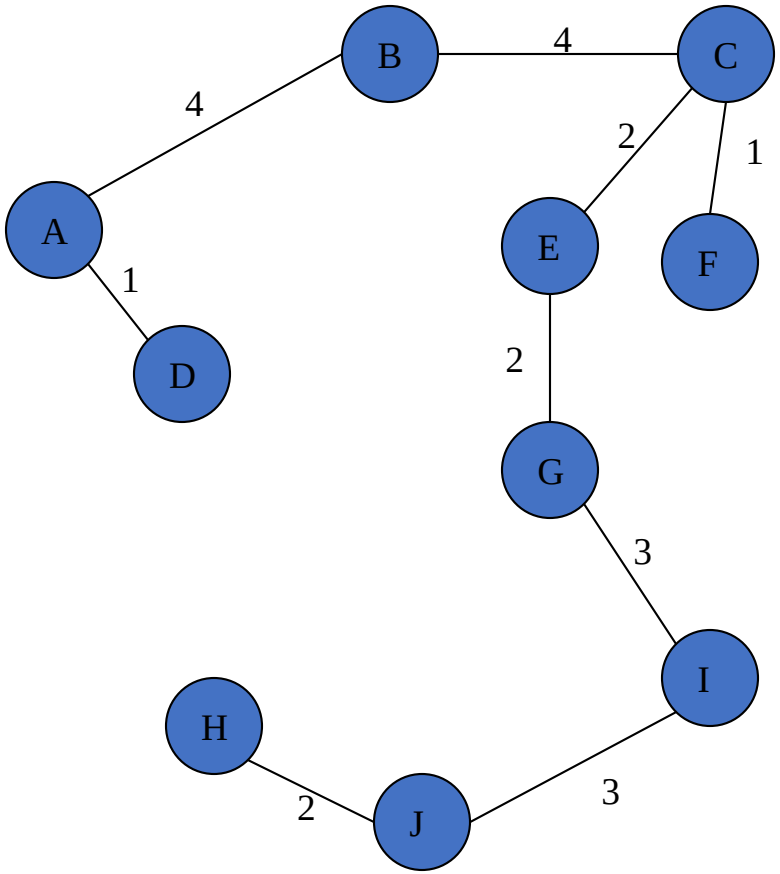
F 5 I

D 5 H

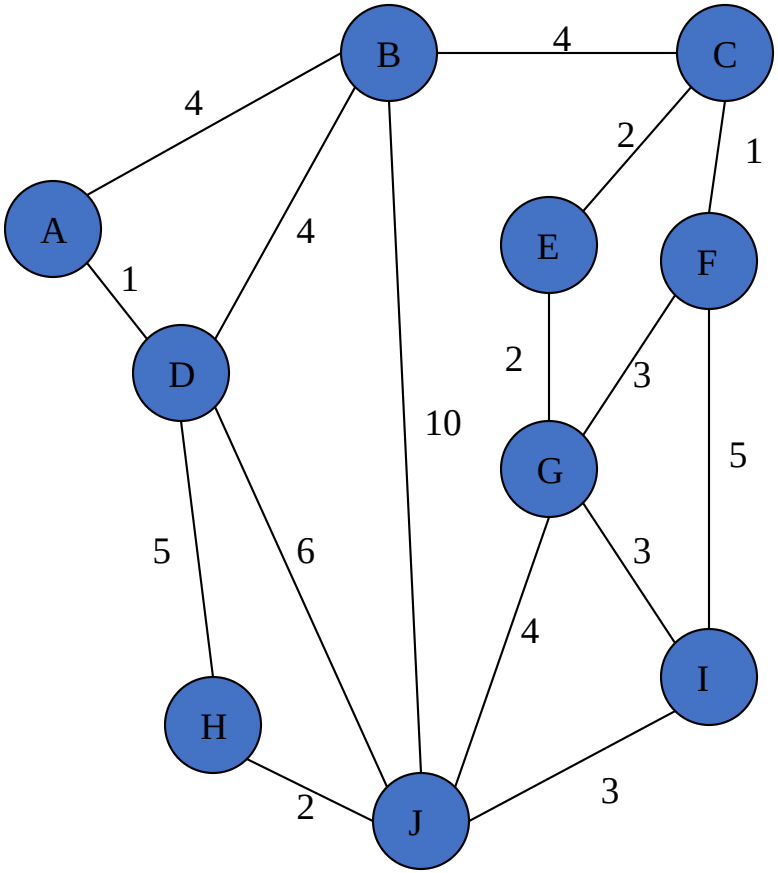
D 6 J

B 10 J

# Minimum Spanning Tree




# Given Graph



# question

Can you estimate the running time of Kruskal's algorithm?

Can you also elaborate on the assumptions you make about graph representation and data structures used?

# Prim's Algorithm

This algorithm starts with one node. It then, one by one, adds a node that is unconnected to the new graph to the new graph, each time selecting the node whose connecting edge has the smallest weight out of the available nodes' connecting edges.

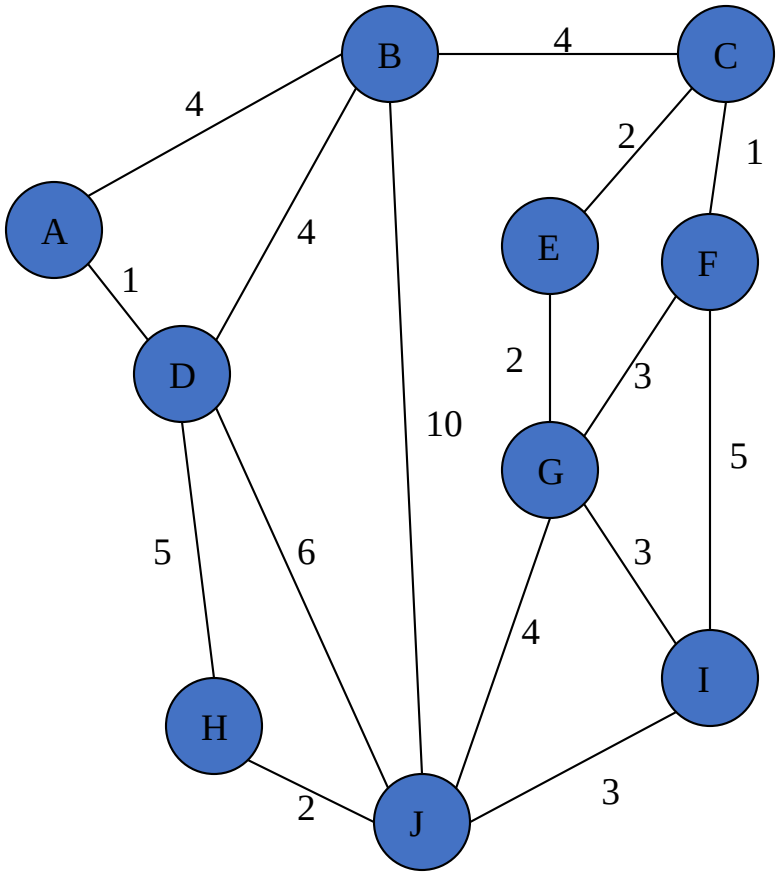
# Prim's Algorithm

The steps are:

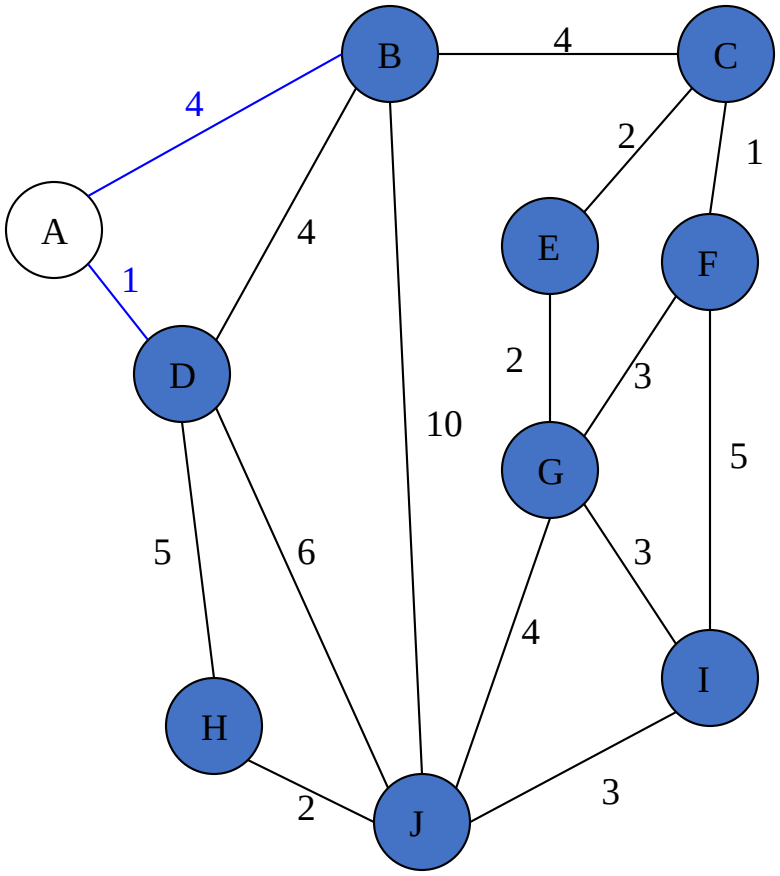
1. The new graph is constructed - with one node from the old graph.
2. While new graph has fewer than n nodes,
    1. Find the node from the old graph with the smallest connecting edge to the new graph,
    2. Add it to the new graph

Every step will have joined one node, so that at the end we will have one graph with all the nodes and it will be a minimum spanning tree of the original graph.

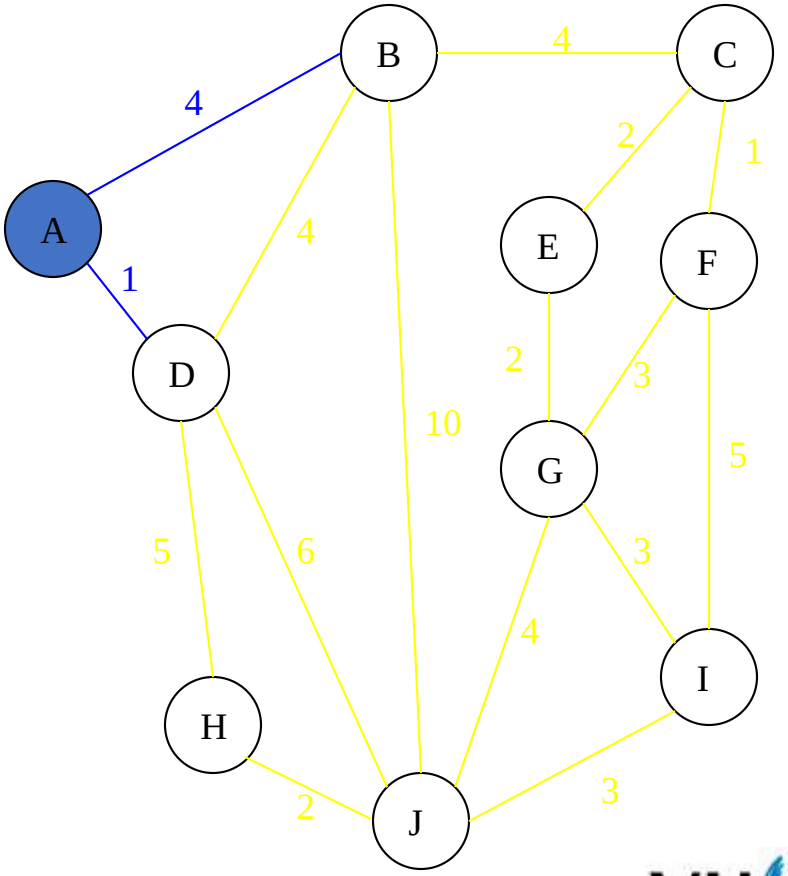
# Given Graph



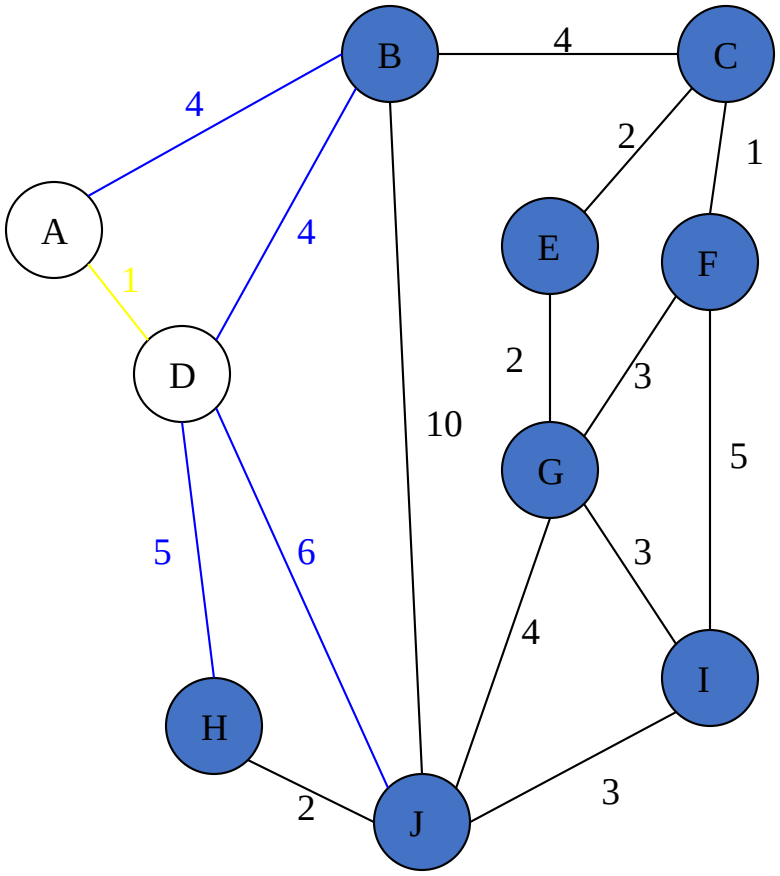
# Old Graph



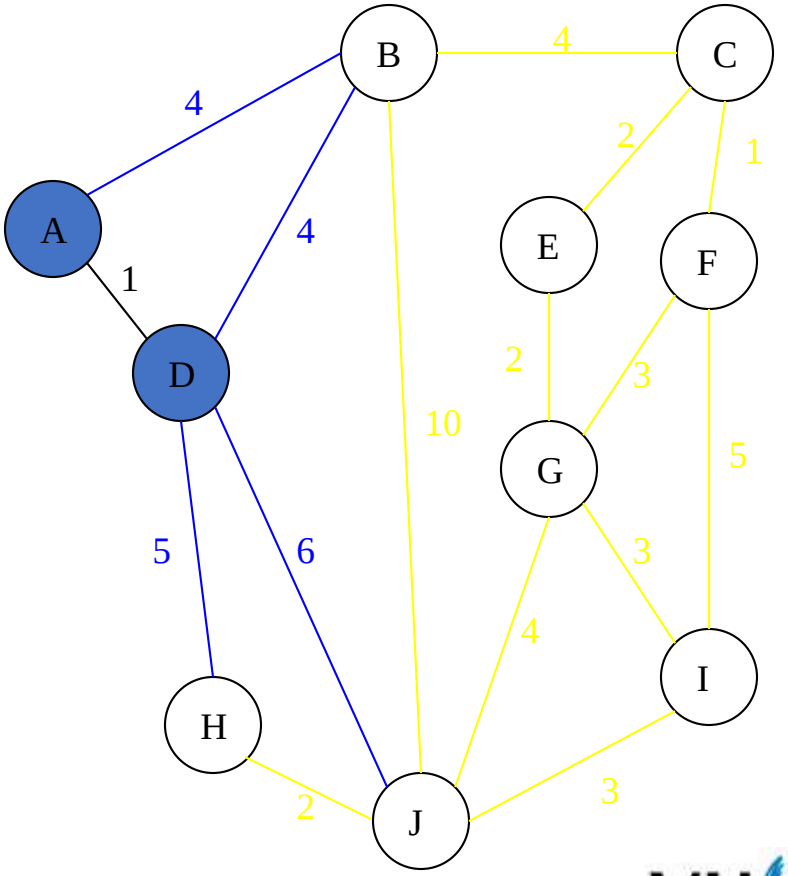

# New Graph



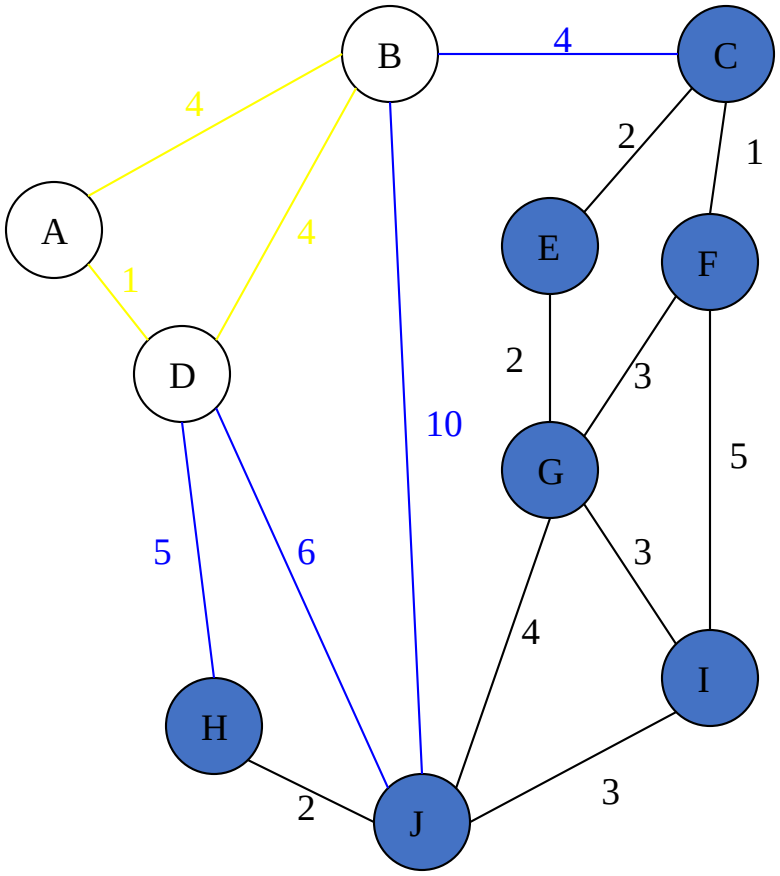
# Old Graph



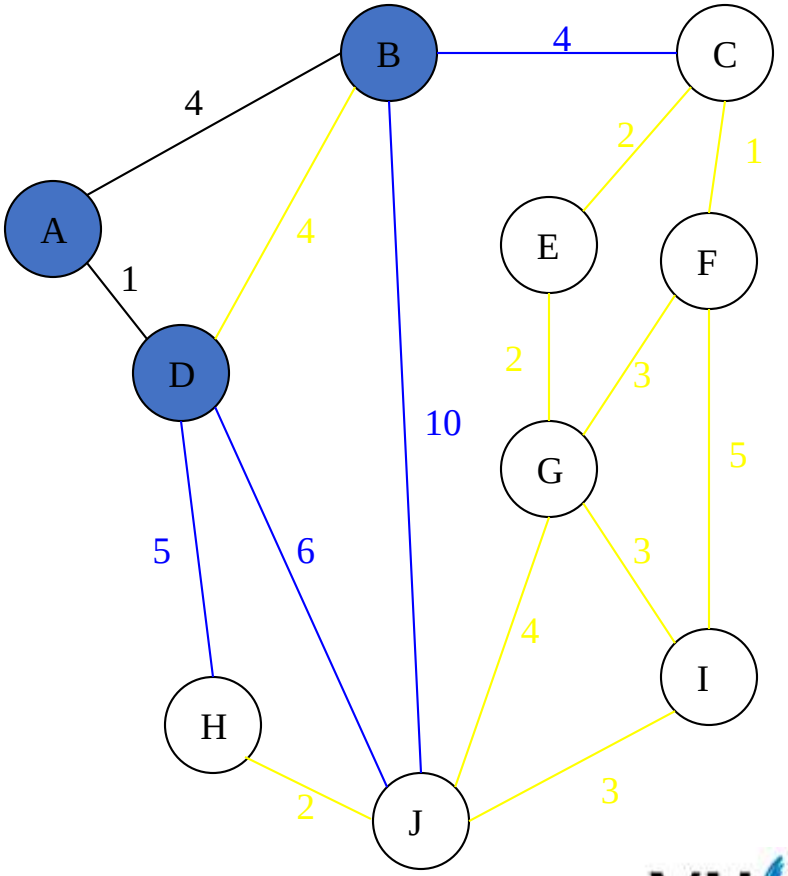

# New Graph



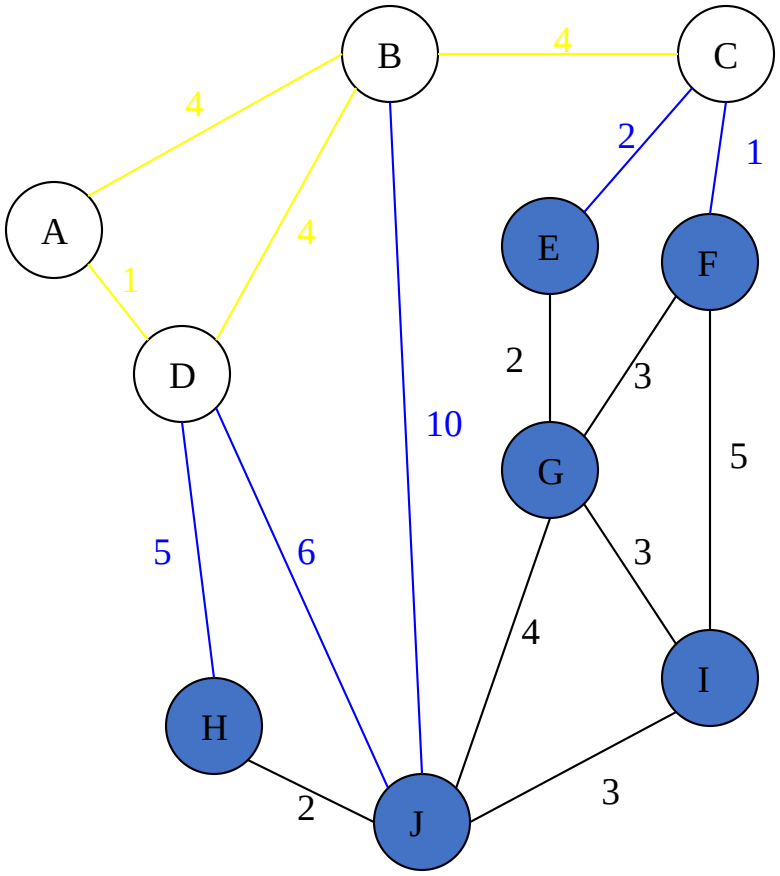
# Old Graph



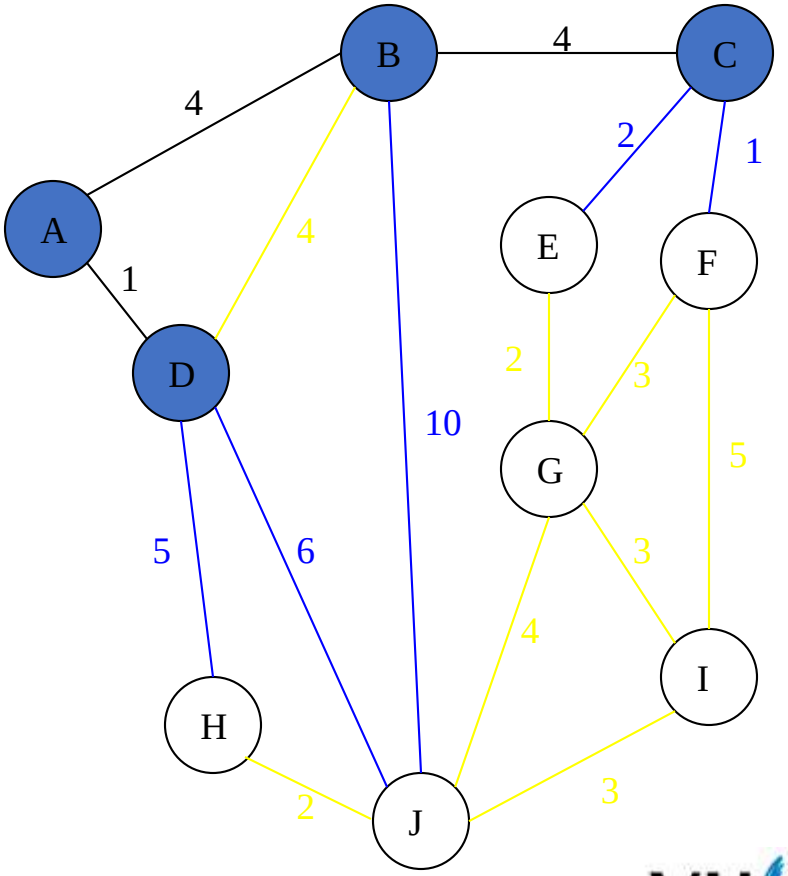

# New Graph



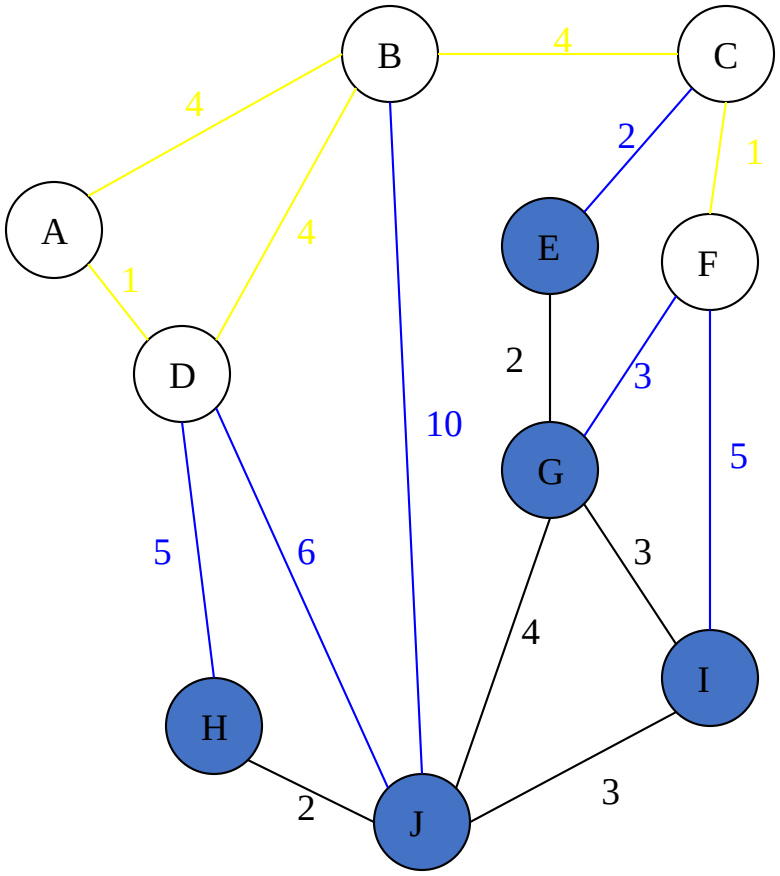
# Old Graph



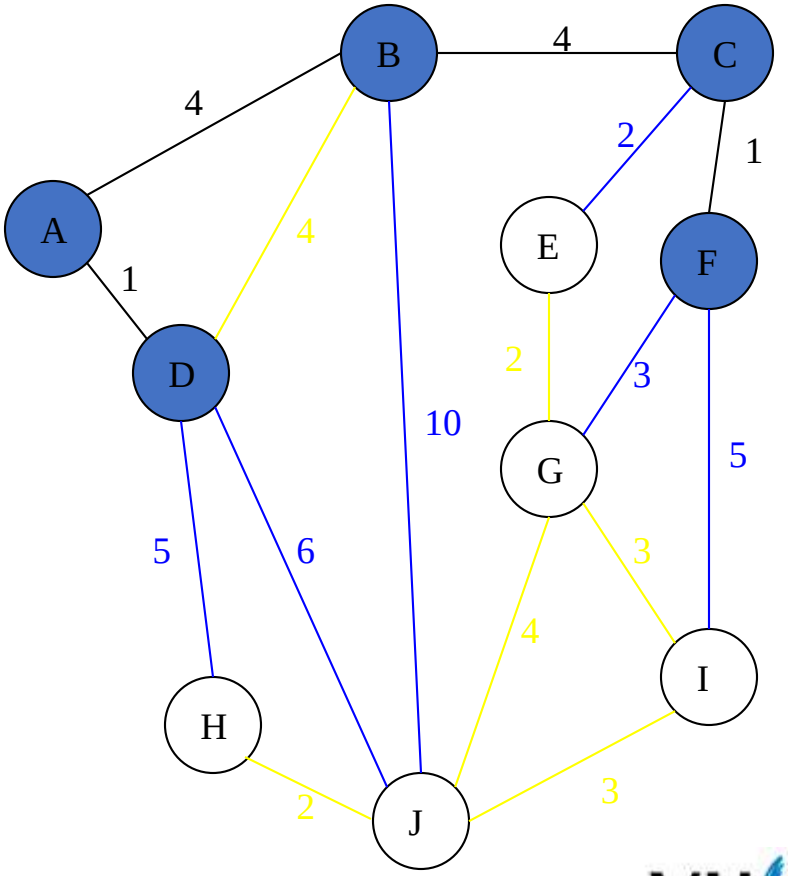

# New Graph



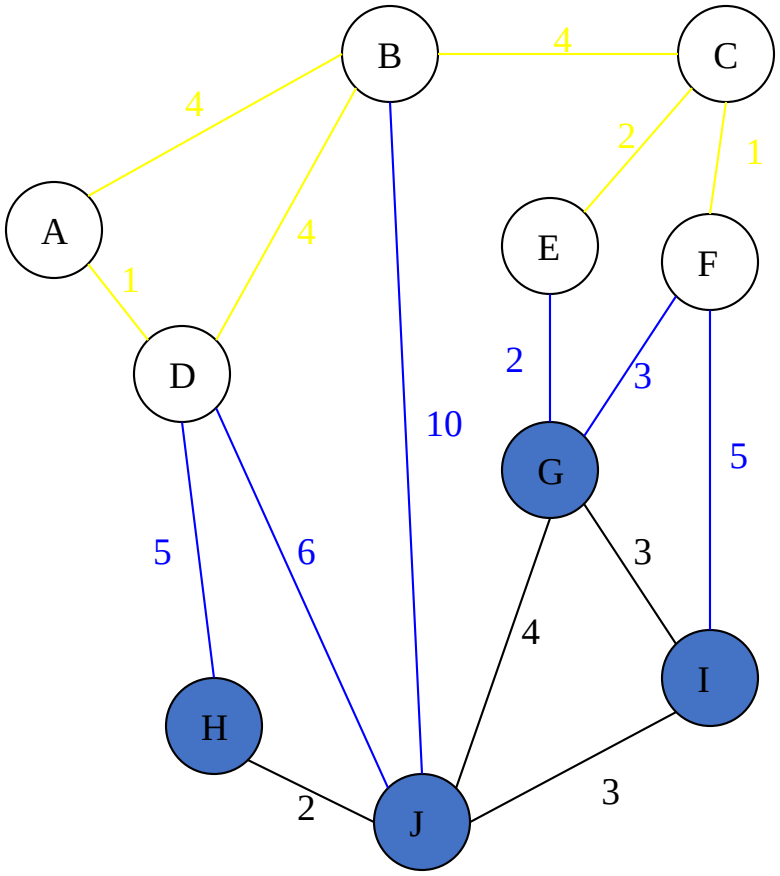
# Old Graph



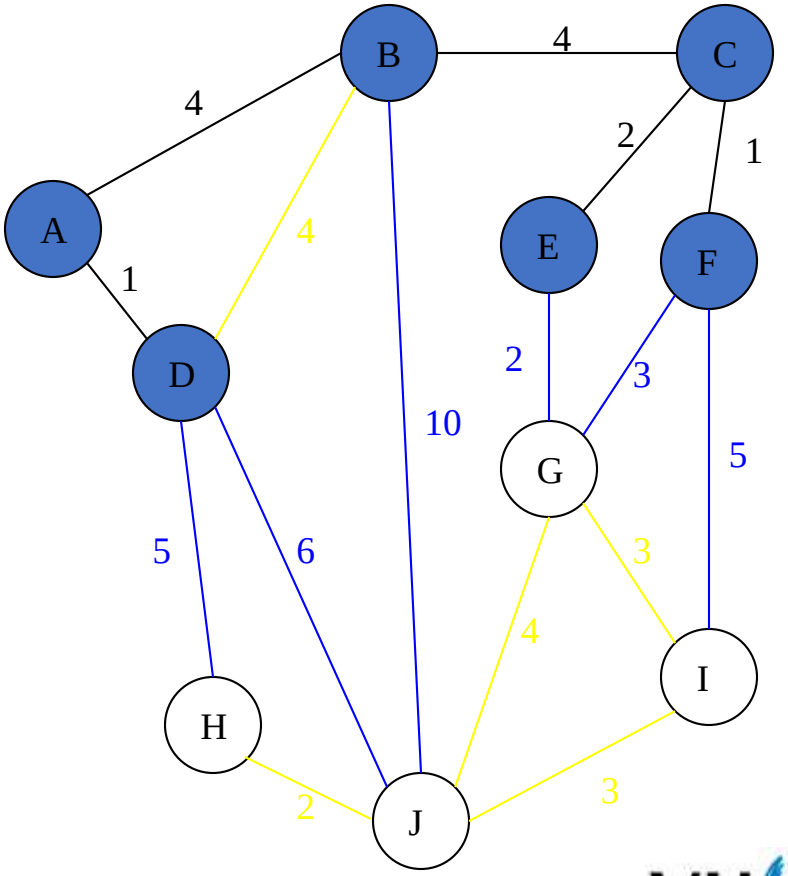

# New Graph



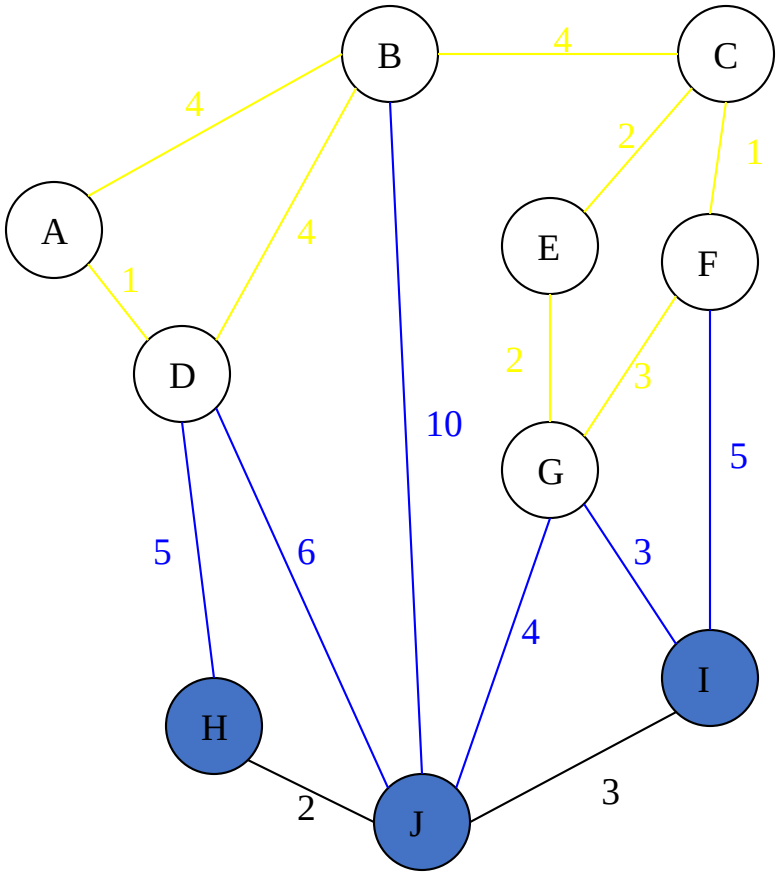
# Old Graph



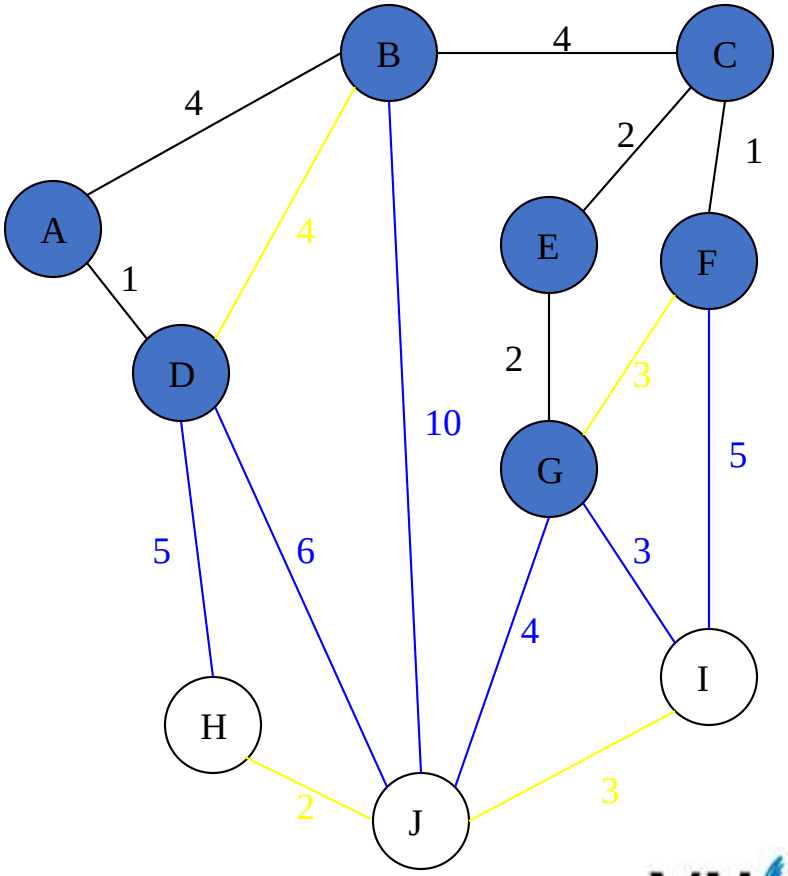

# New Graph



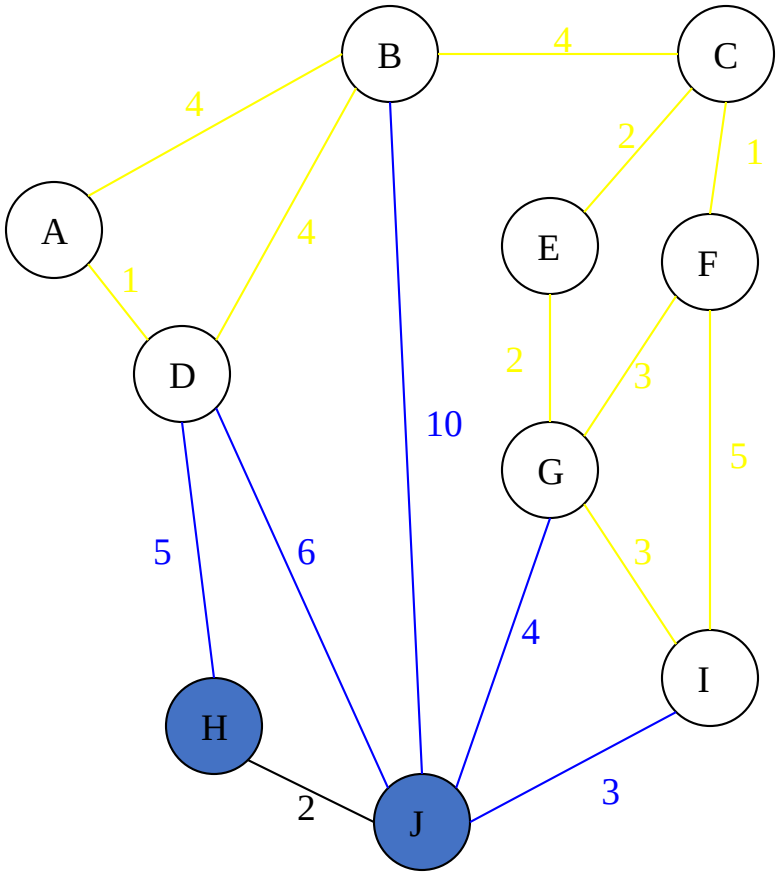
# Old Graph



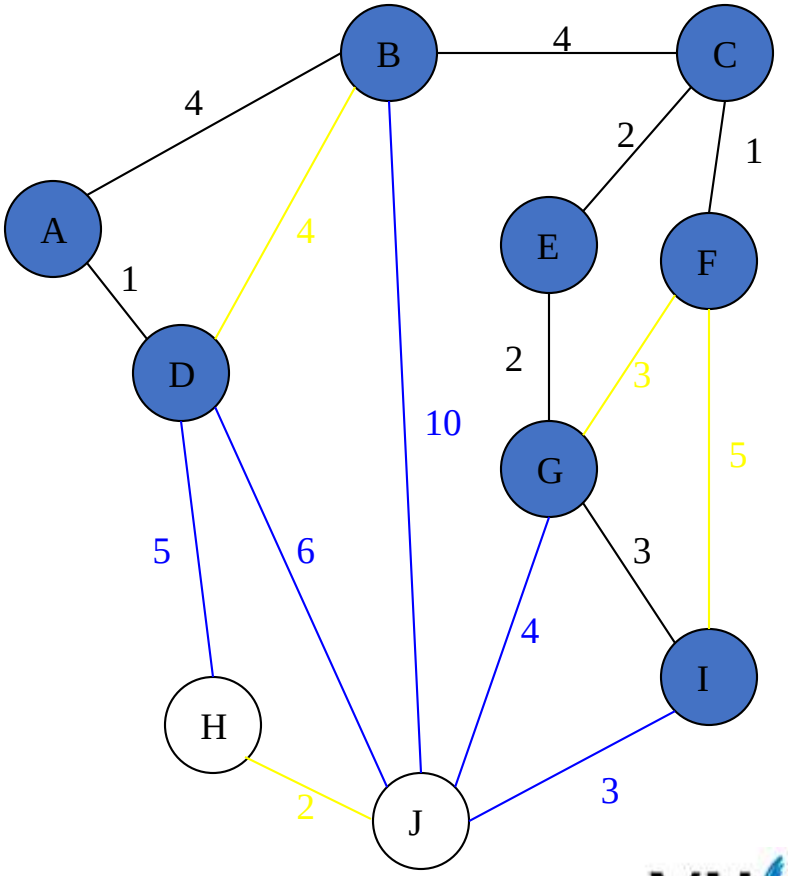

# New Graph



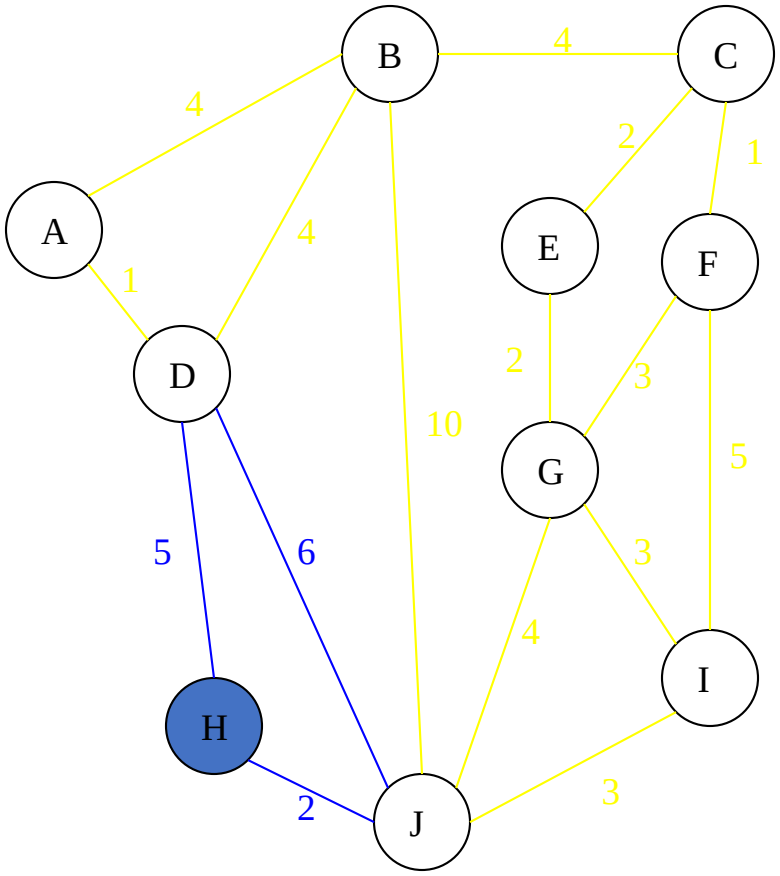
# Old Graph



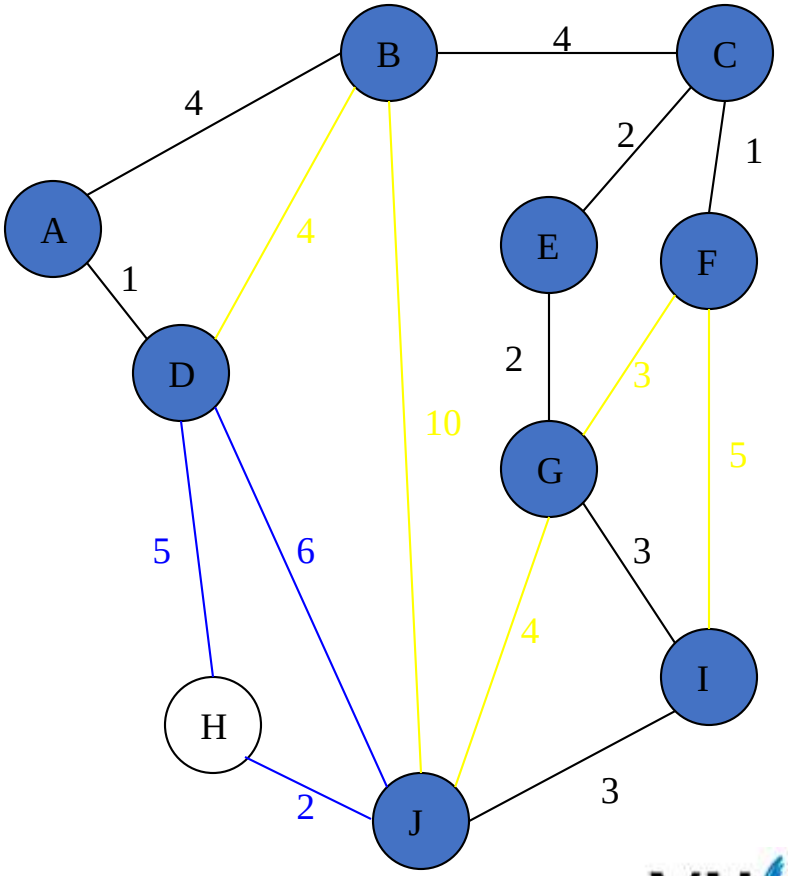

# New Graph



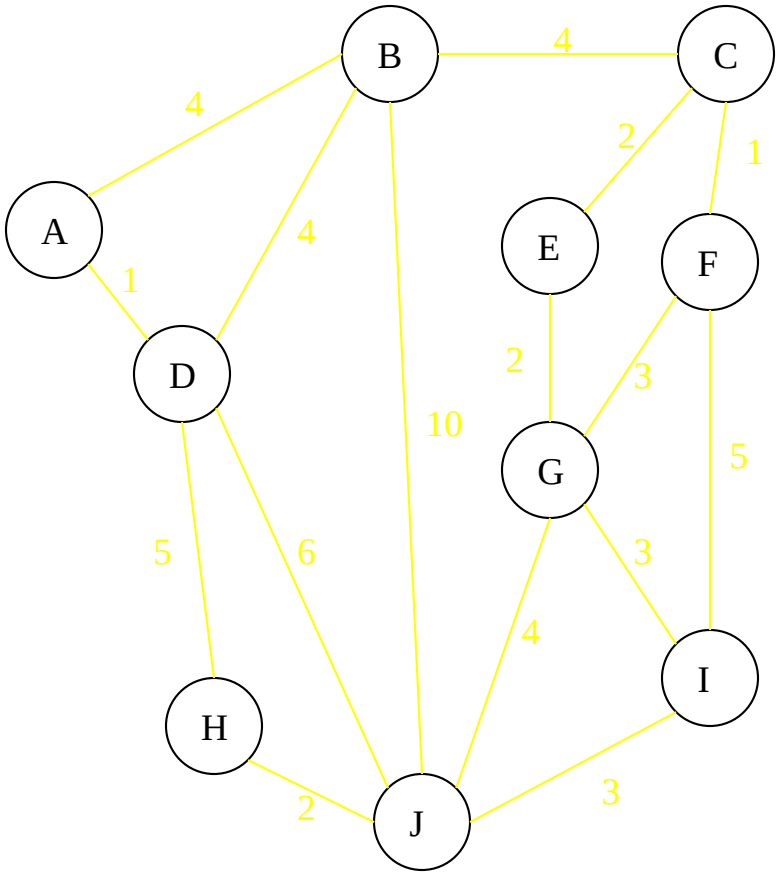
# Old Graph



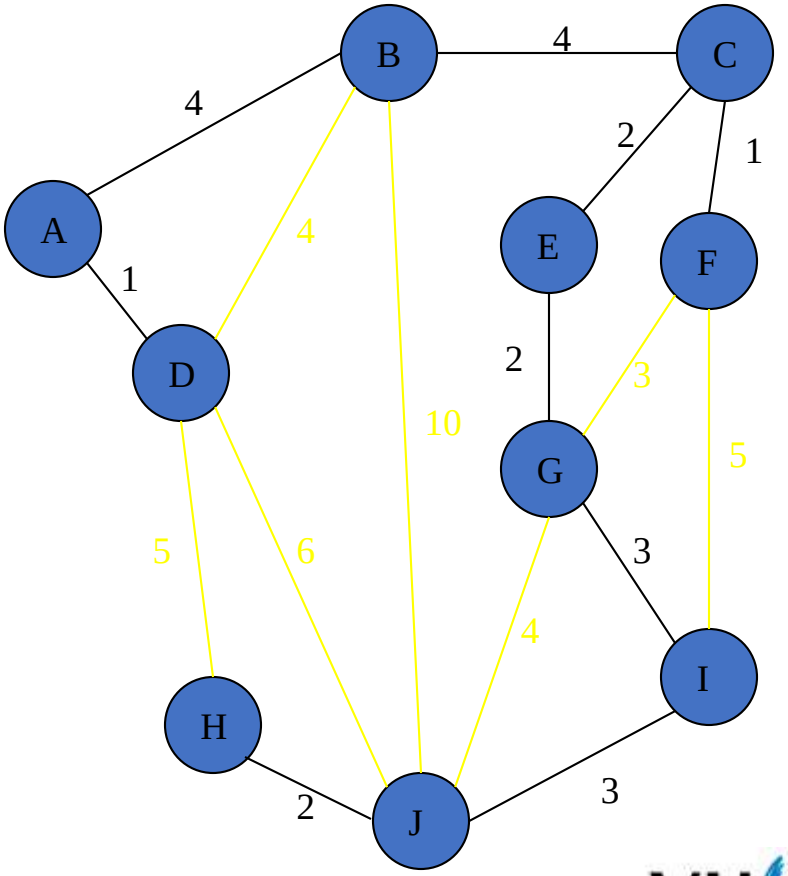

# New Graph



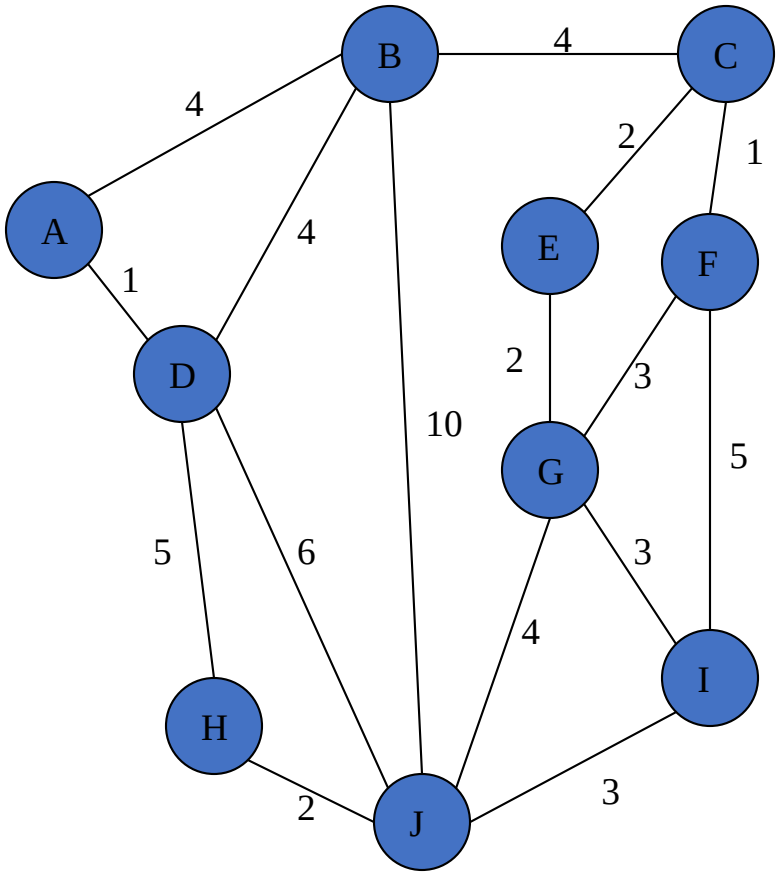
# Old Graph



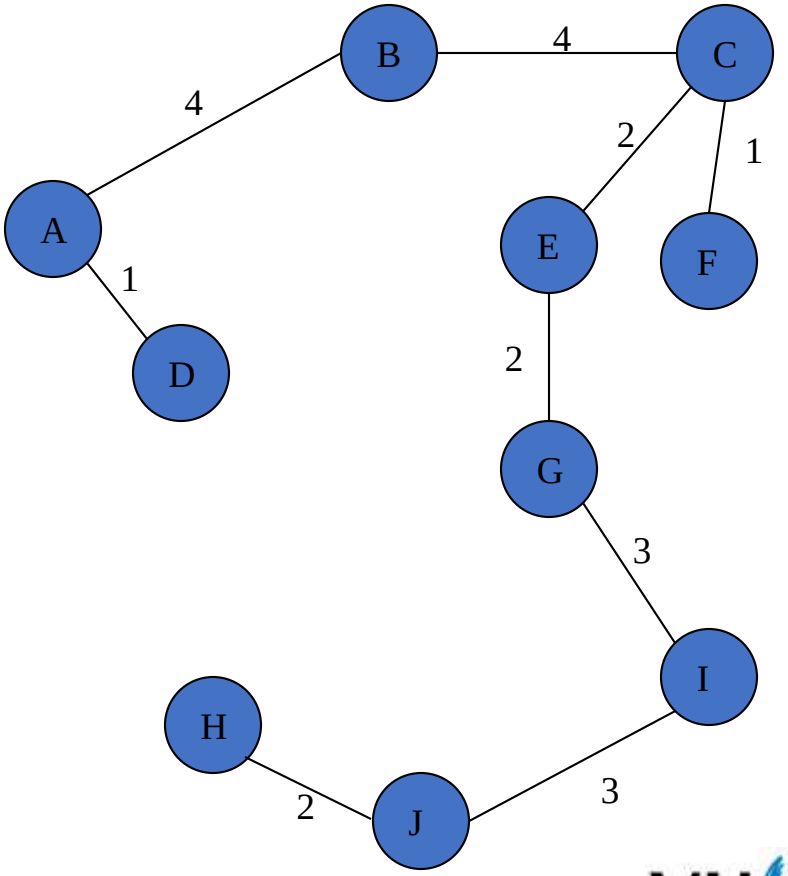

# New Graph



# Given Graph




# Minimum Spanning Tree



# question

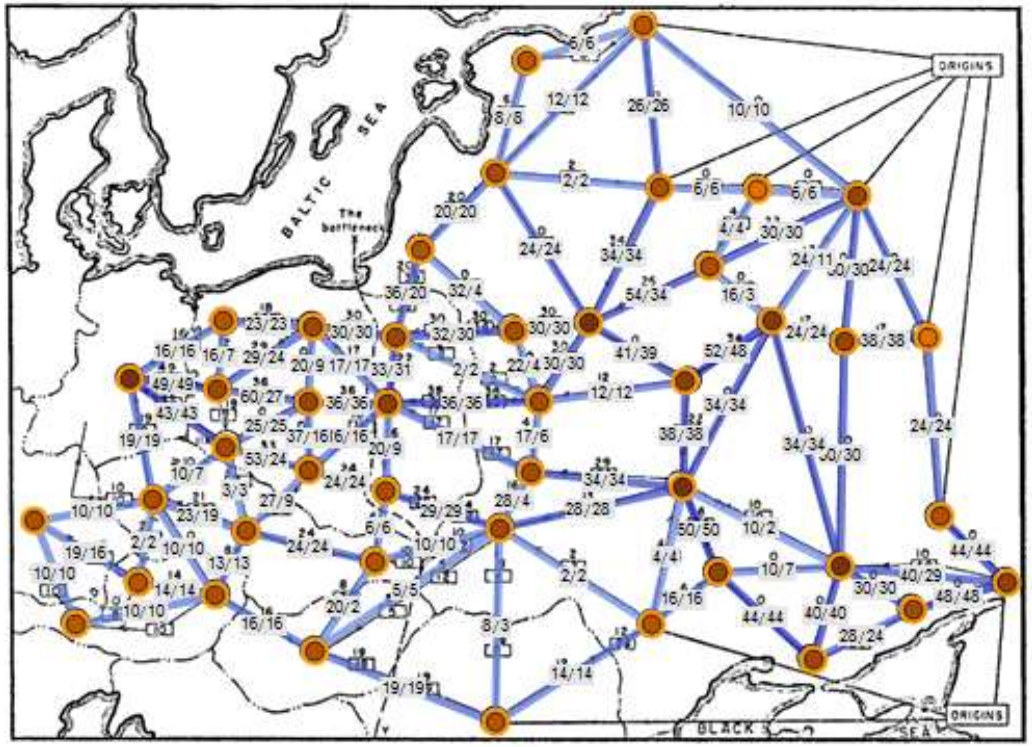
Can you estimate the running time of Prim's algorithm?

Can you also elaborate on the assumptions you make about graph representation and data structures used?

# maximum flow

Ford-Fulkerson and extensions

03-02-2020



# Flow problems

Ford-Fulkerson method for maximum flow (MaxFlow).

Shortest augmenting path algorithm for MaxFlow.

Some generalizations of the MaxFlow problem.

The minimum cost flow problem.

# Max flow

Given a network directed from a source vertex to a target vertex with a maximum capacity on each arc determine the highest value of a flow from the source to the target which respects the capacities and is preserved at each intermediate vertex.

# Method of Ford and Fulkerson

Start with a flow of zero.

At each iteration consider the residual graph composed of

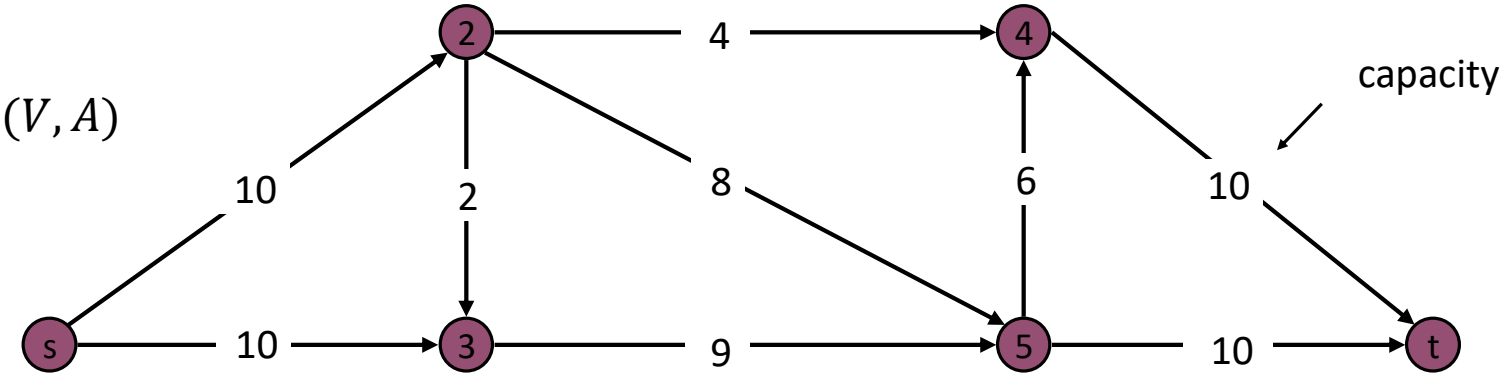
- The original arcs with capacity equal to the difference between the original capacity and the flow on the arc (the residual capacity) if this value is positive, otherwise the arc is eliminated.
- Arcs opposite to the original ones with capacity equal to the flow passing on the corresponding arc if a (positive) flow passes on the arc.

Determine a path from source to target on the residual graph, if it exists, and update the flow with the minimum capacity along the path (increase flow for original arcs, decrease for opposite arcs). Update the residual graph. May such a path not exist, terminate, the flow is maximum.

Following
https://en.wikipedia.org/wiki/Ford%E2%80%93Fulkerson_algorithm
we call this a *method* instead of an *algorithm* because it lacks detailing an important step: determining a positive flow.

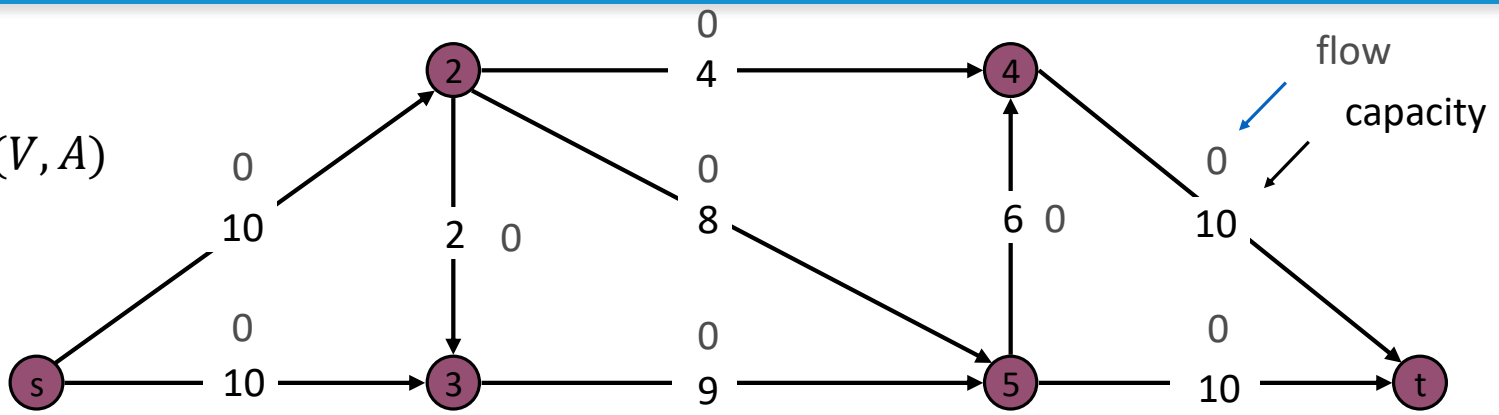
# Ford-Fulkerson example

$D = (V, A)$



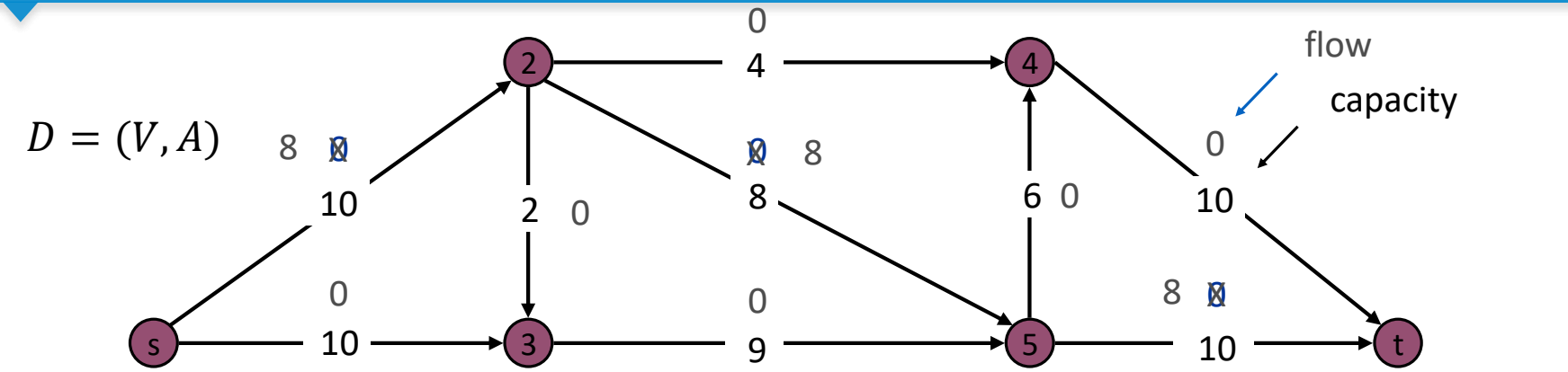
# Ford-Fulkerson example

$D = (V, A)$




Flow value = 0

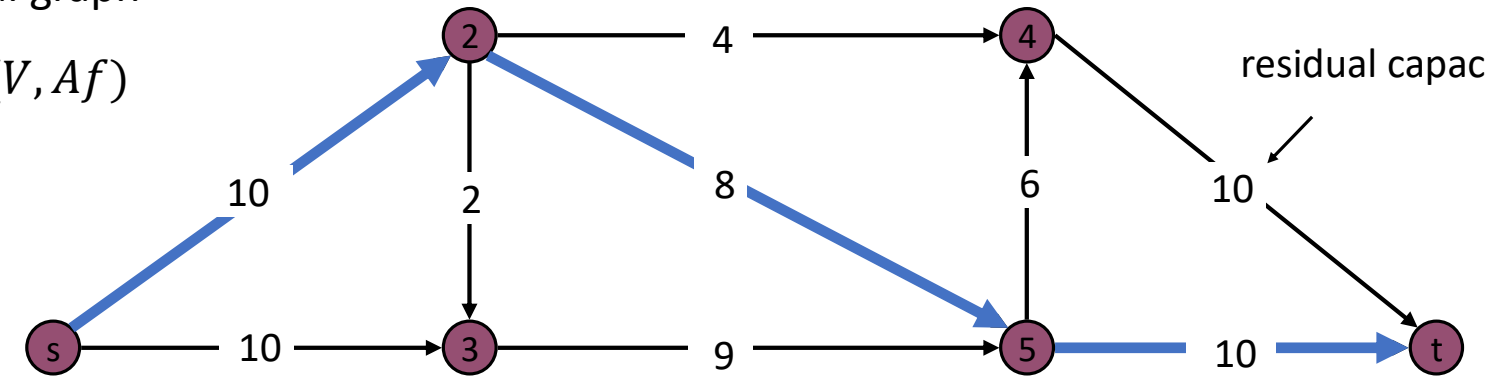
# Ford-Fulkerson example



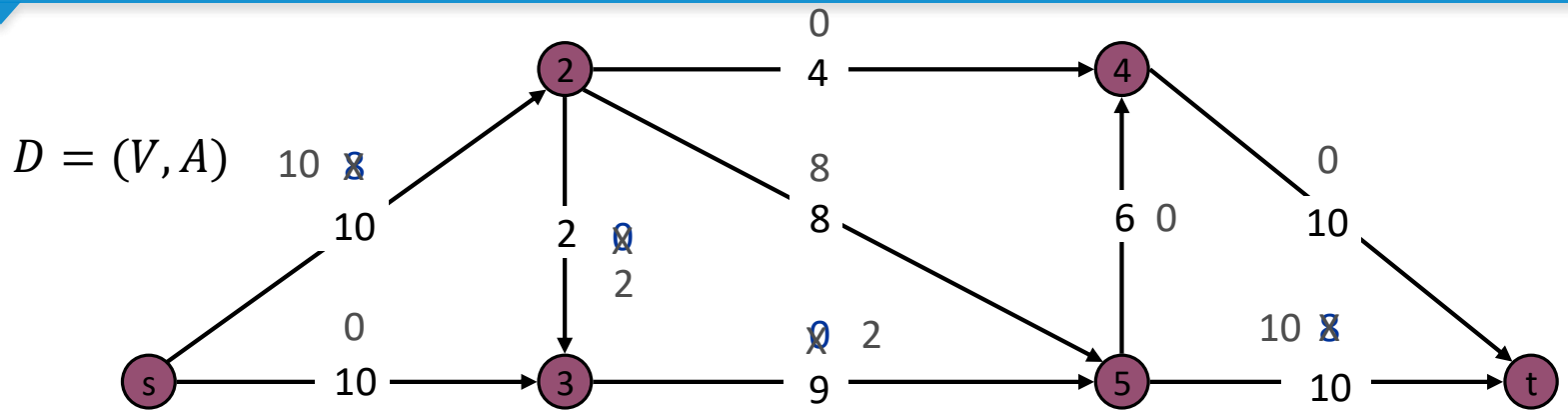

Flow value = 0

Residual graph

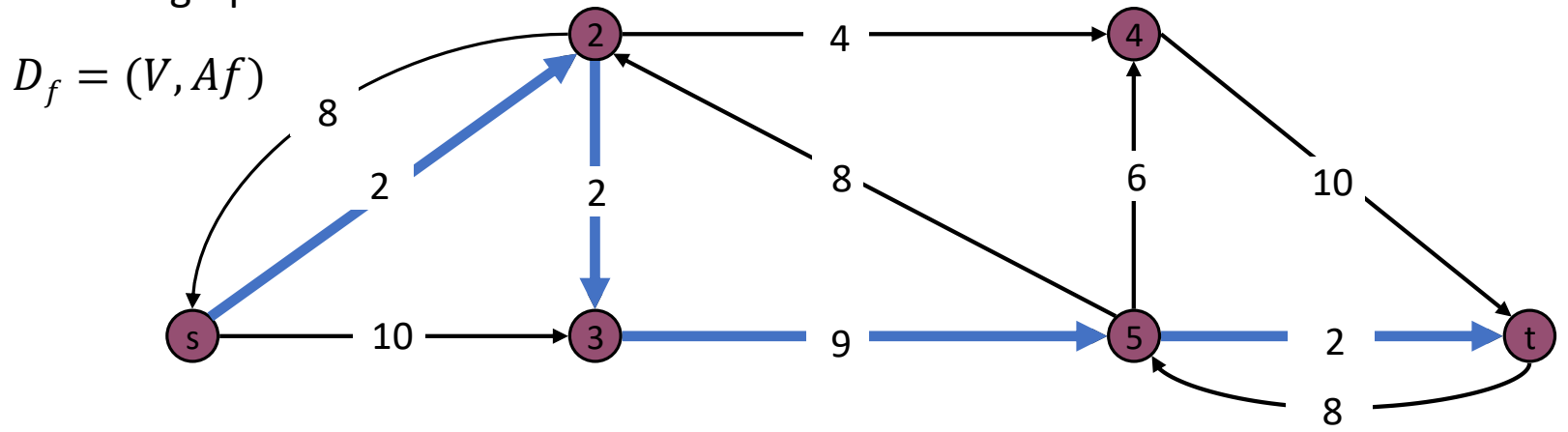
$D_f = (V, Af)$



# Ford-Fulkerson example

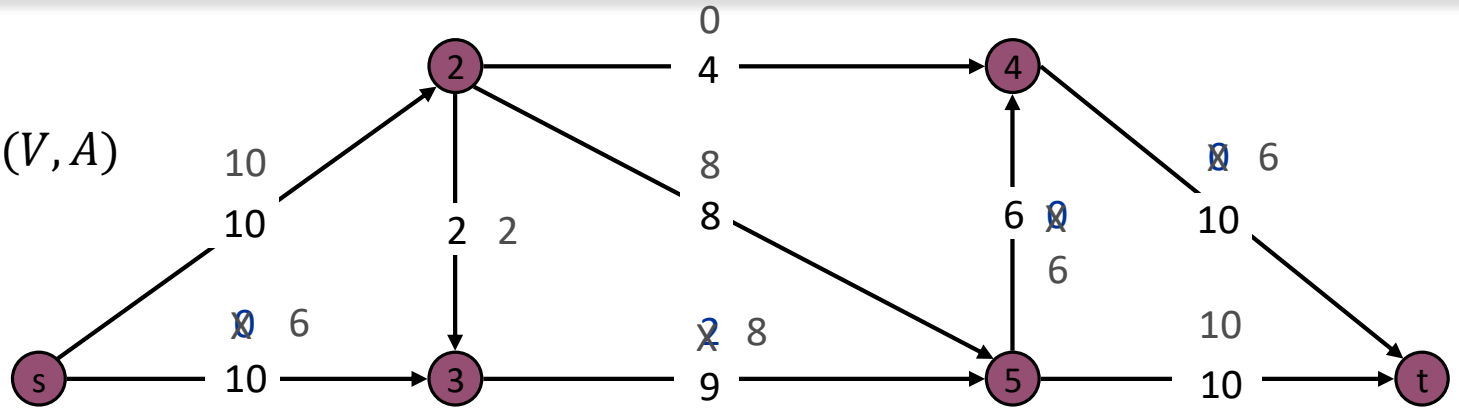



Residual graph



# Ford-Fulkerson example

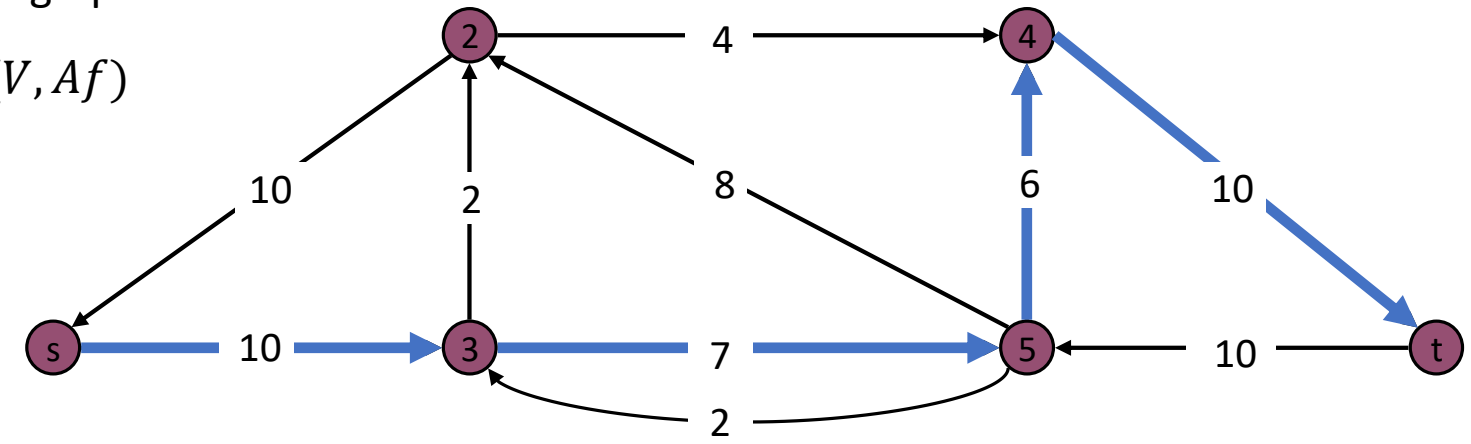
$D = (V, A)$




Flow value = 10

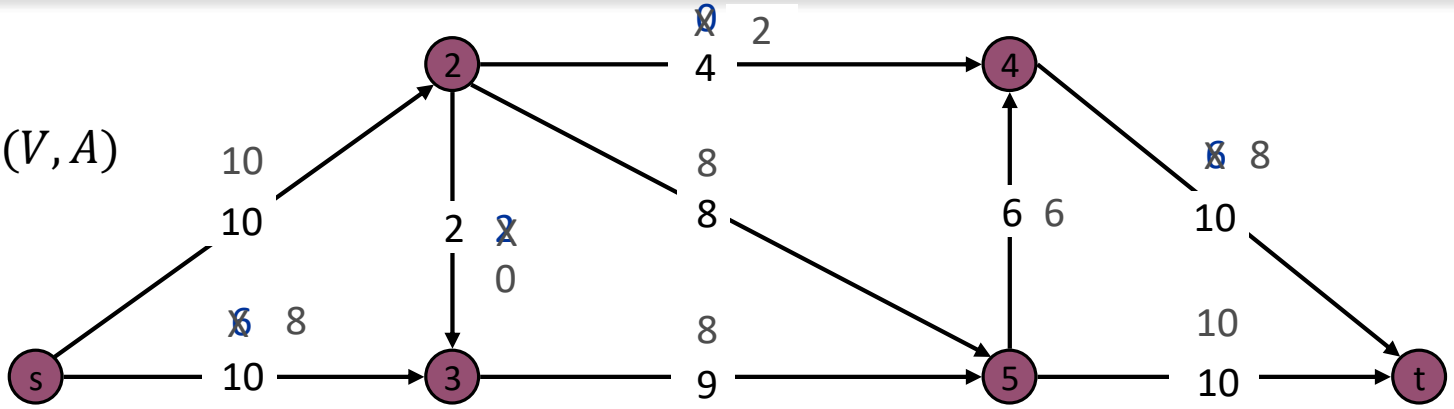
Residual graph

$D_f = (V, Af)$



# Ford-Fulkerson example

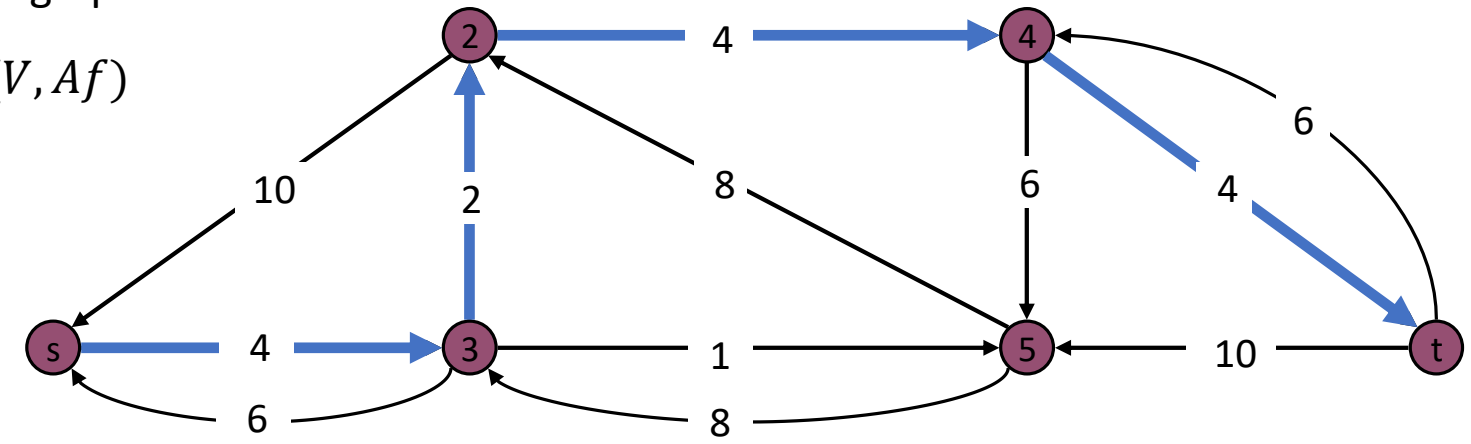
$D = (V, A)$




Flow value = 16

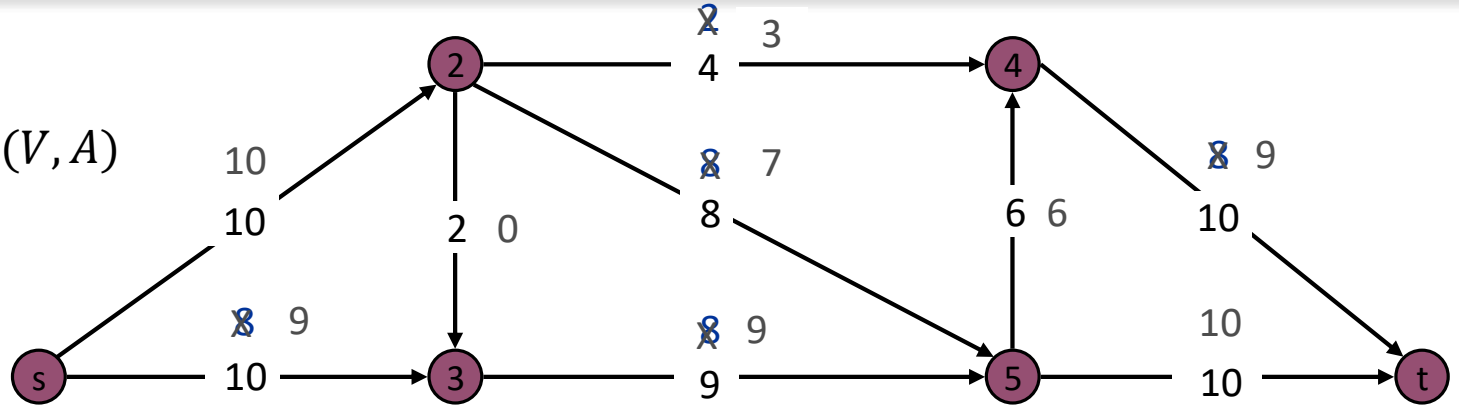
Residual graph

$D_f = (V, Af)$



# Ford-Fulkerson example

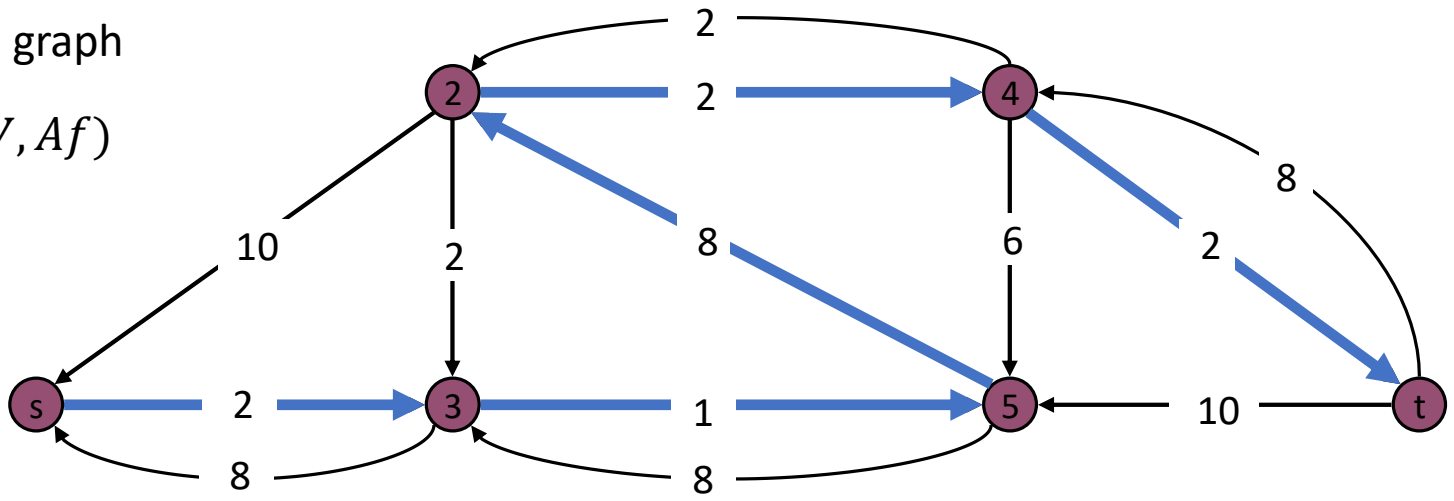
$D = (V, A)$




Flow value = 18

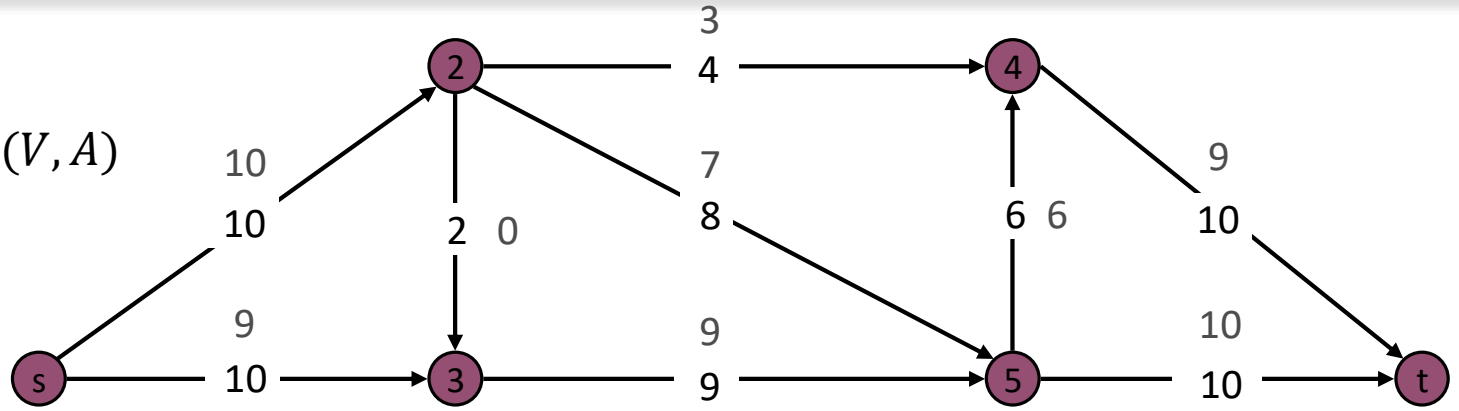
Residual graph

$D_f = (V, Af)$



# Ford-Fulkerson example

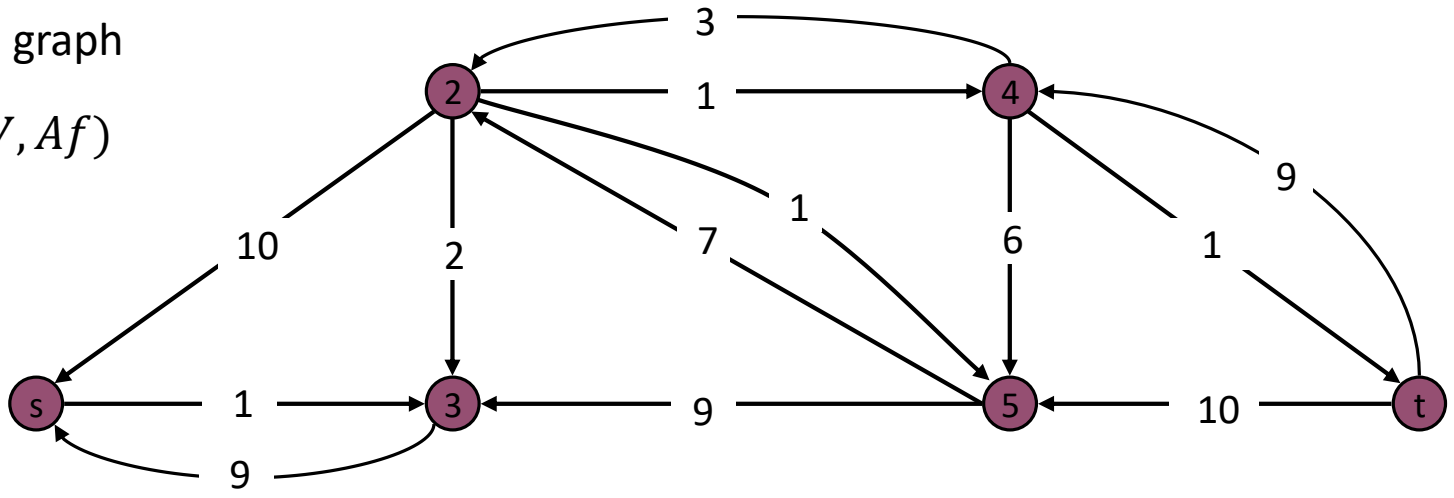
$D = (V, A)$




Flow value = 19

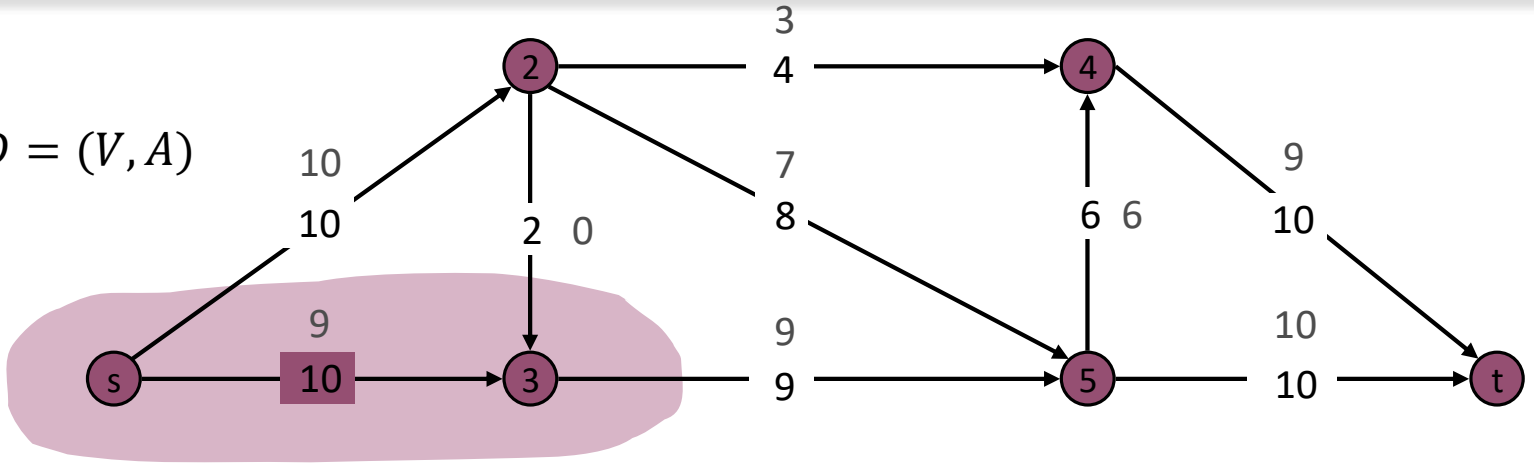
Residual graph

$D_f = (V, Af)$



# Ford-Fulkerson example

$D = (V, A)$

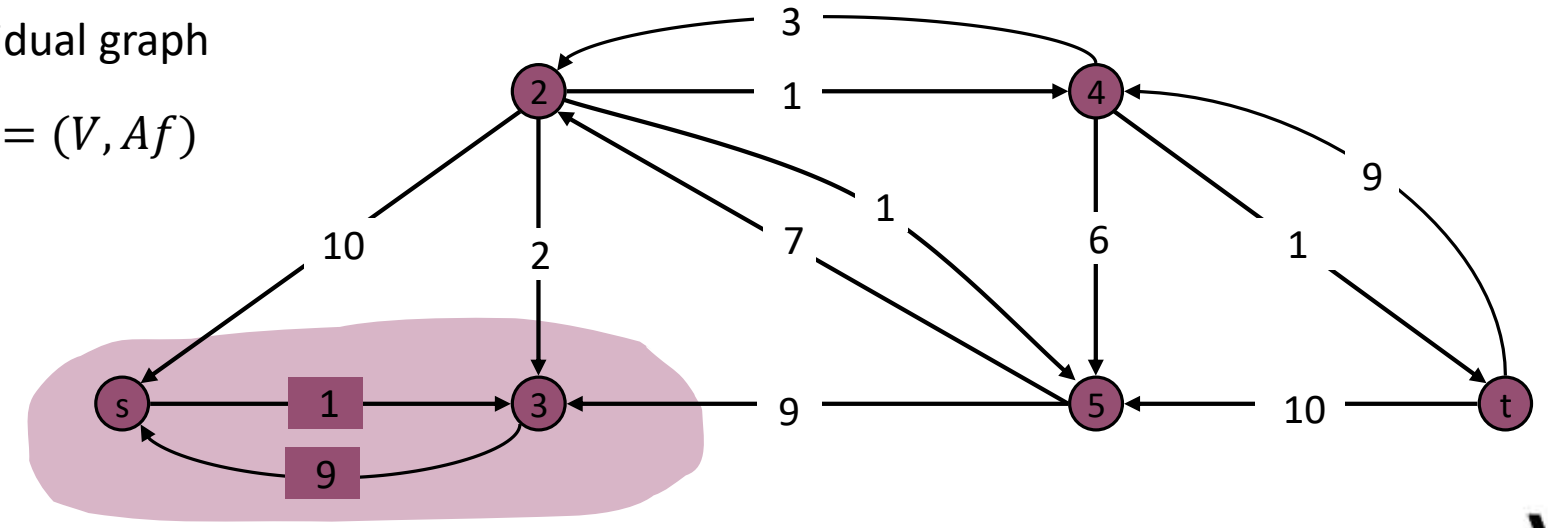



Cut capacity = 19

Flow value = 19

Residual graph

$D_f = (V, Af)$



# Ford-Fulkerson: Analysis.

**Theorem:** Min cut = Max flow , and FF algorithm finds them.

**Proof:** Clearly, Min cut ≥ Max flow. (*)

Other direction:

When algorithm terminates, it gives a cut C and a flow F for which

$$\text{value}(C) = \text{value}(F) \rightarrow$$

$$\text{min cut} \leq \text{value}(C) = \text{value}(F) \leq \text{max flow} \quad (**)$$

(*) + (**) → $\text{min cut} = \text{value}(C) = \text{value}(F) = \text{max flow}$

□

# Ford-Fulkerson: Analysis.

**Corollary:**

If all capacities are integer then there is integer maximum flow.

**Proof:**

If all capacities are integer then the flow found by FF is integer.

□

# Ford-Fulkerson: Analysis.

Does FF algorithm always terminate?

It depends on the capacities:

Integer capacities:

- Yes, since the flow value increases by at least 1 in every iteration.

Rational capacities:

- Yes, since the flow value increases by at least some small epsilon in every iteration.

Real valued capacities: (Note: not realistic in practice)

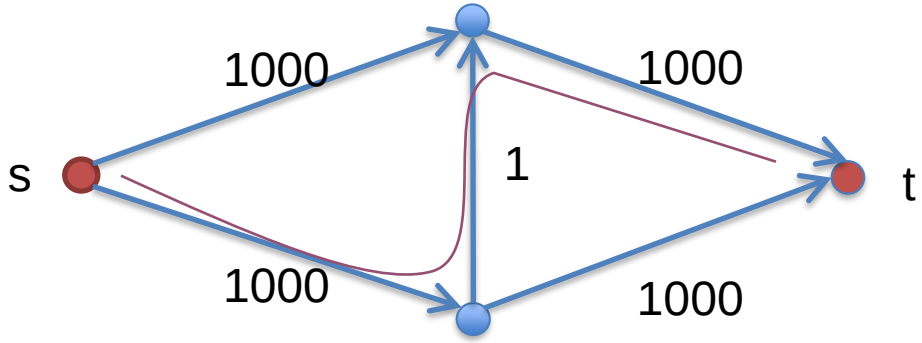
- See a nonterminating example in https://en.wikipedia.org/wiki/Ford%E2%80%93Fulkerson_algorithm
- But it will terminate with a small modification of the algorithm. (See next slides.)

# Ford-Fulkerson: Analysis.

🤔 How about the running time?

$O(n)$ per iteration, but how many iterations can we have?

# iterations may be as large as max flow value!




*Example: 2000 iterations if the algorithm always chooses the arc in the middle.*

Is there an algorithm for which the running time only depends on the size of the graph, and not on the capacities? Yes! Next slides.

# Shortest augmenting path algorithm

By Edmonds-Karp-Dinitz (1970)

**Algorithm:**

Apply the FF algorithm but always choose the path with the minimum number of edges.

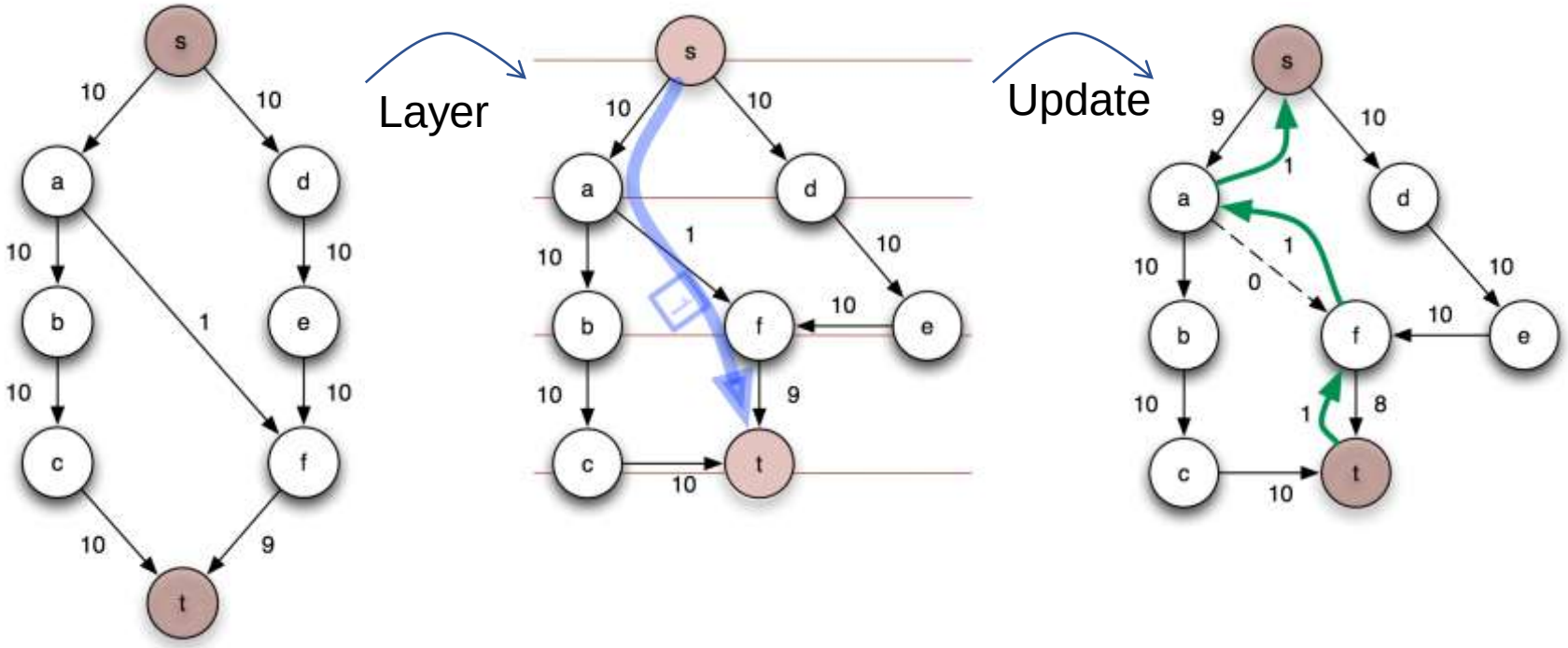
Let n be the number of vertices and let m be the number of edges.

**Theorem**

Shortest augmenting path algorithm takes $O(nm)$ iterations.

# Shortest augmenting path, example

Iteration 1



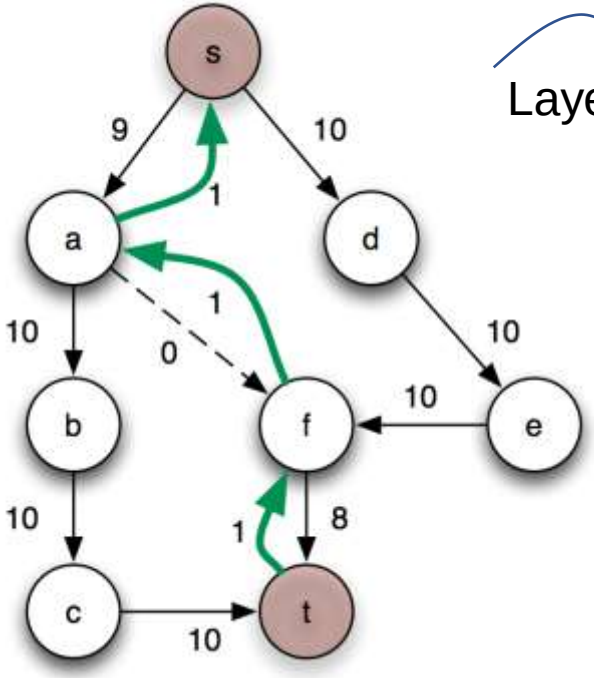

Network

Find augmenting path

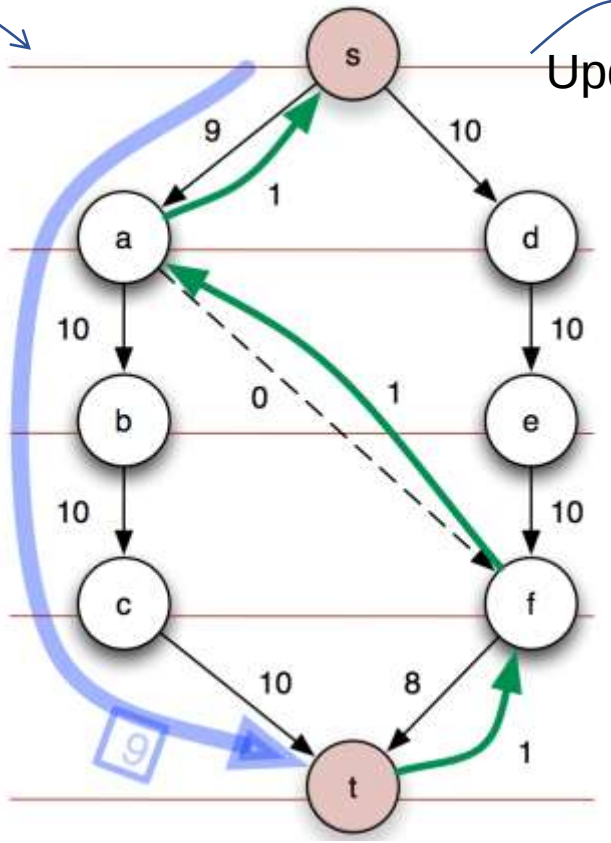
1st Residual graph

# Shortest augmenting path

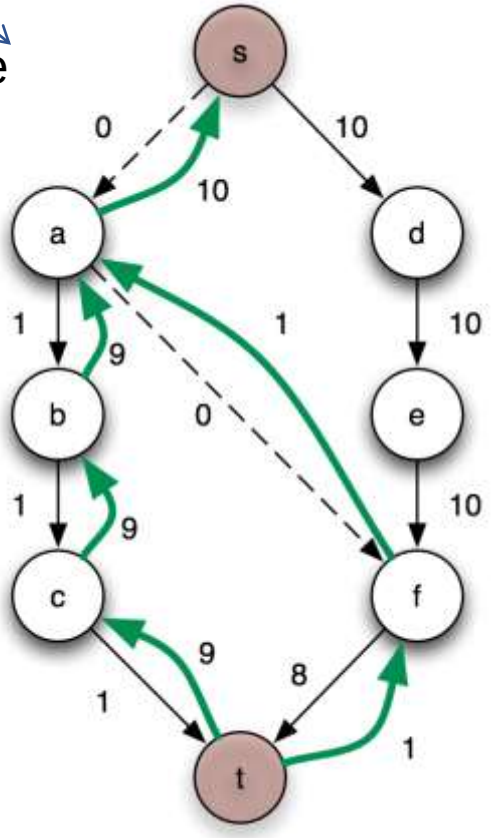
Iteration 2




Layer




Update



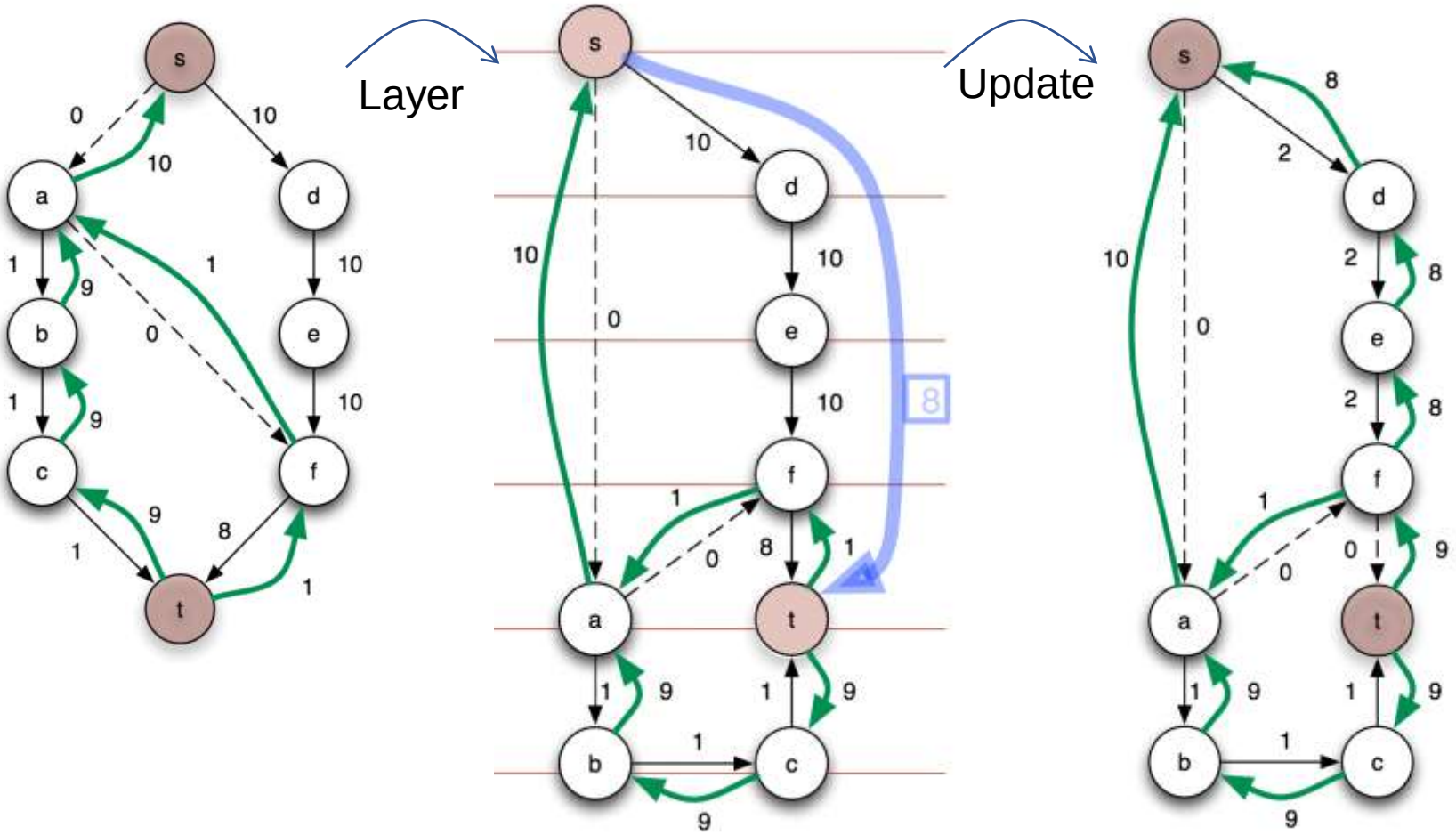

1st Residual

Find augmenting path

2nd residual

# Shortest augmenting path

Iteration 3



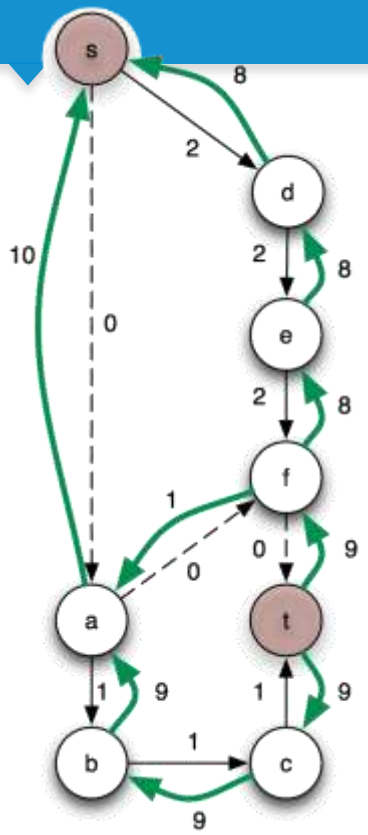

2nd Residual

Find augmenting path

3rd residual

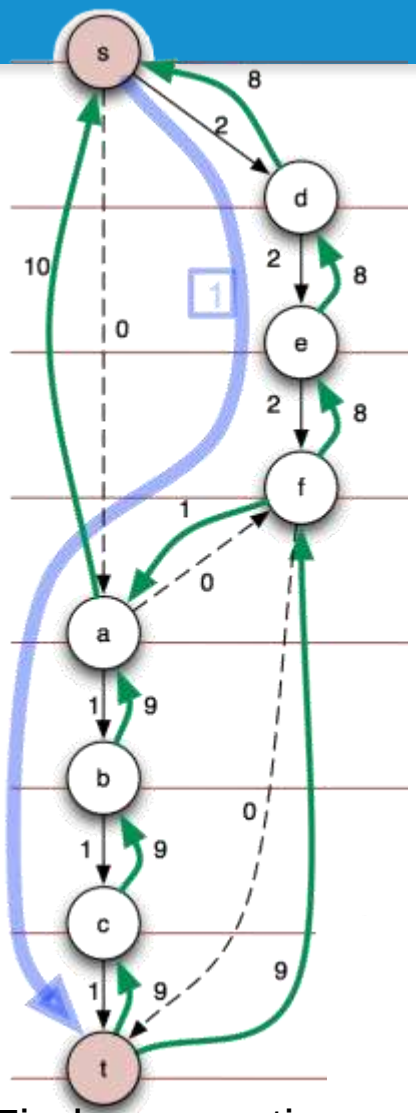
# Shortest augmenting path

Iteration 4



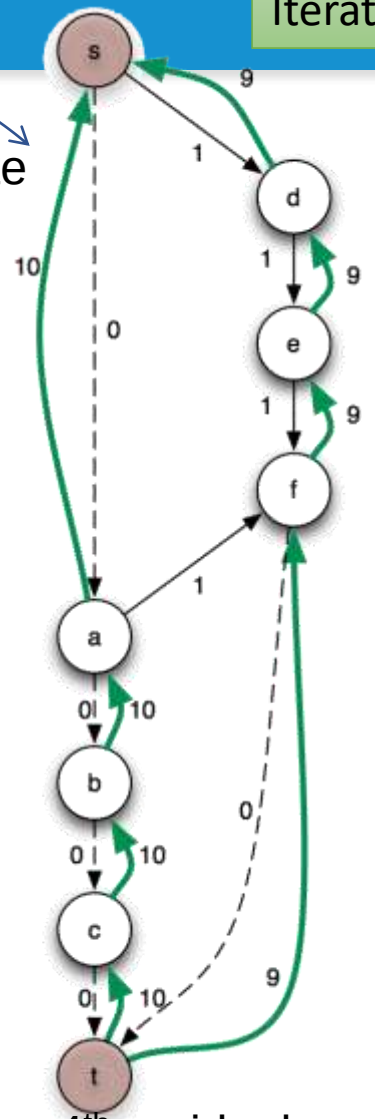

3rd residual

Layer



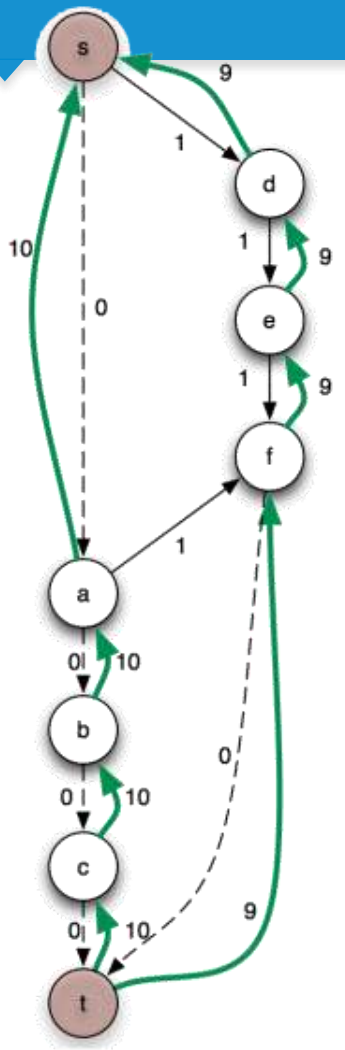

Find augmenting path

Update




4th residual

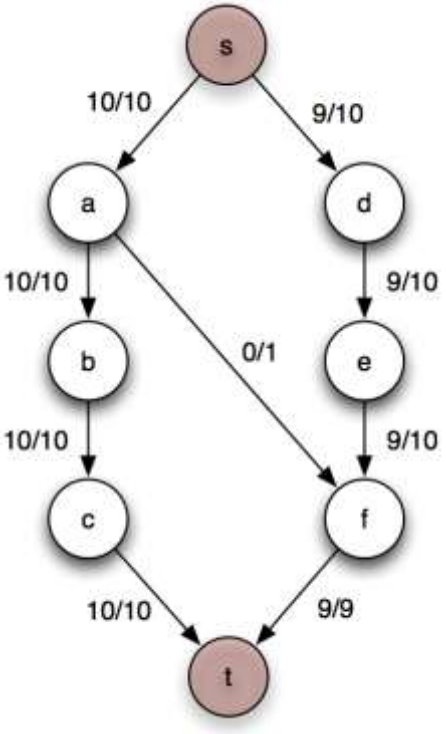
# Shortest augmenting path




4th residual

No more path from $s$ to $t$.

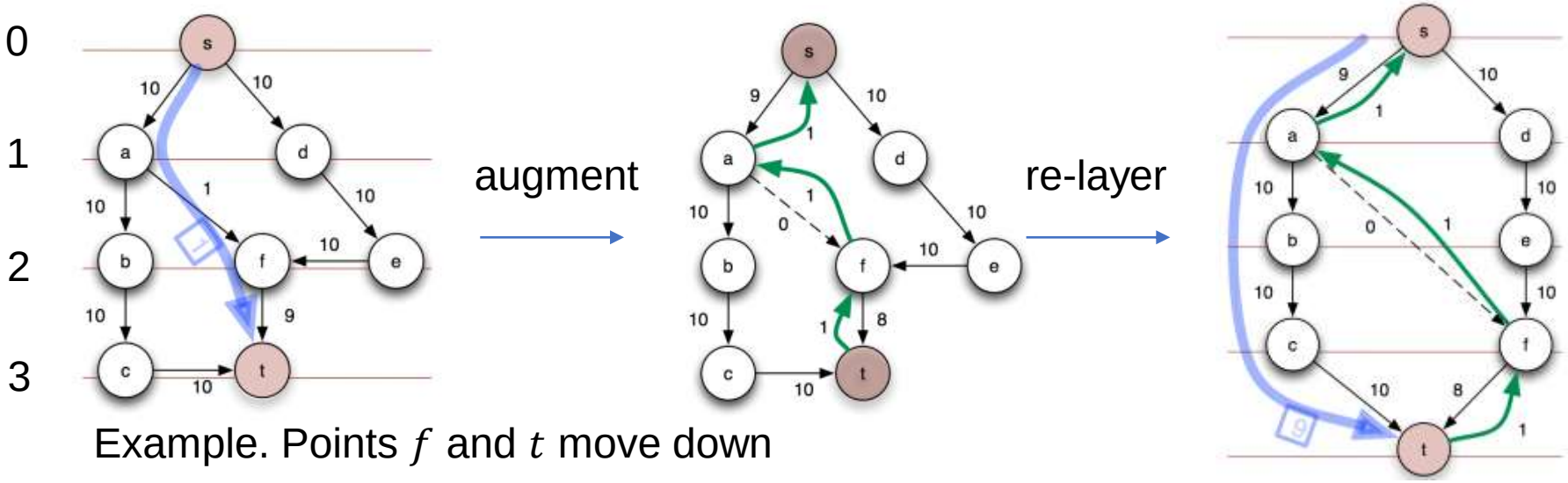
Max flow found →




Maximum flow

# Analysis (not for exam)

Let $d(s,u)$ be the **depth** of vertex $u$ in the residual graph (i.e., the number of arcs on the shortest path between $s$ and $u$).




Example. Points $f$ and $t$ move down

**Lemma 1** For any point $u$, the depth $d(s,u)$ never decreases.

**Proof**

- Deleting an arc will not decrease $d(s,u)$
- Adding an arc may only decrease $d(s,u)$ if the arc is forward (down). However, we never add a forward arc.

# Analysis (not for exam)

Call an arc $(u, v)$ in the residual graph **critical** if it has the minimum capacity on the augmenting path.

**Lemma 2** $d(s, u)$ increases between two critical moments of arc $(u, v)$.

**Proof** *(Hint: check this proof for arc $(a, f)$ in the example)*

If arc $(u, v)$ is critical then $v$ is below $u$ and the arc will be removed from the residual graph. (See $(a, f)$ in iteration 1).

Arc $(u, v)$ will only return in the residual when arc $(v, u)$ is on the augmenting path in some iteration.

But then $u$ must be below $v$. Since $v$ did not went up, point $u$ must have moved down. (See $(a, f)$ in iteration 4). □

# Analysis (not for exam)

**Theorem**

Shortest augmenting path algorithm takes $O(nm)$ iterations.

**Proof**

By Lemma 2, each arc is critical only $O(n)$ times (since the depth of $v$ is less than $n$ and $v$ goes down between two critical moments of $(u, v)$ ).

Further, there is at least one critical arc in each iteration.

→ There are no more than $O(nm)$ iterations. □

# Generalizations of the maximum flow problem

A polynomial time algorithm for the following versions of the max-flow problem follows by reducing each version to the original form of the max-flow problem. The concept of **polynomial time reduction** will be explained next week.

a) The network has many sources and many sinks.

b) The network is undirected.

c) The nodes as well as the arcs have capacities.

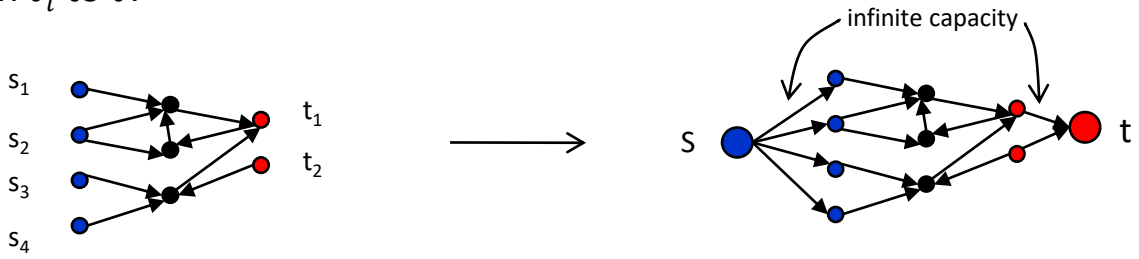
d) The network is undirected and the nodes have capacities

e) There are lower bounds (and no upper bounds) on the flow through each arc.
(Goal :Find min flow)

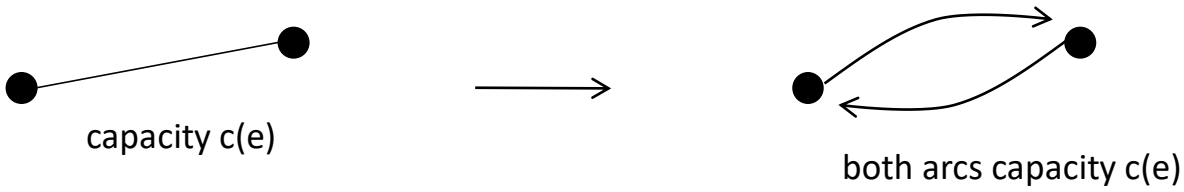
# Generalizations of max flow

a) The network has many sources and many sinks.
b) The network is undirected

(a) Add two vertices and make these the new source and sink. Add an arc from $s$ to each $s_i$ and an arc from each $t_i$ to $t$.



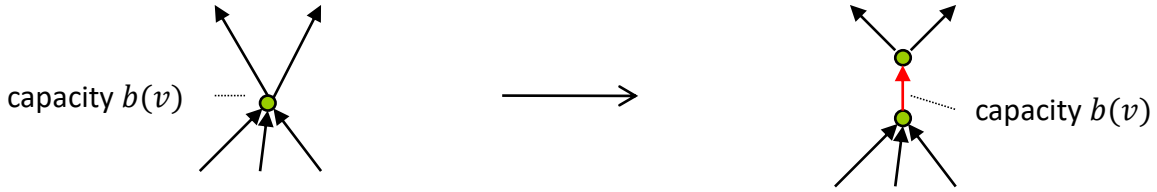

(b) Replace each edge by two arcs. Give each arc the capacity of the edge.



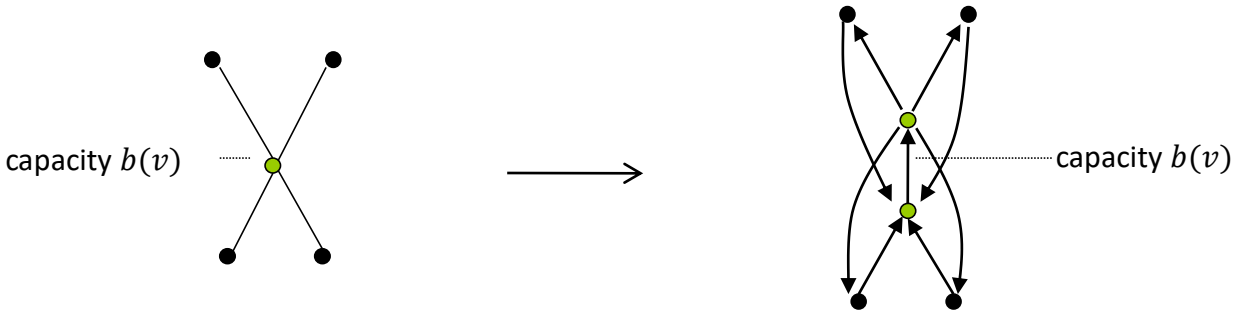
# Generalizations of max flow

c) The nodes as well as the arcs have capacities.
d) The network is undirected and the nodes have capacities.

(c) Split each node in two nodes and add an arc.



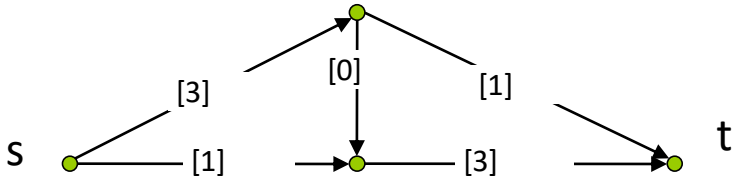

(d) Do both operations (b) and (c).

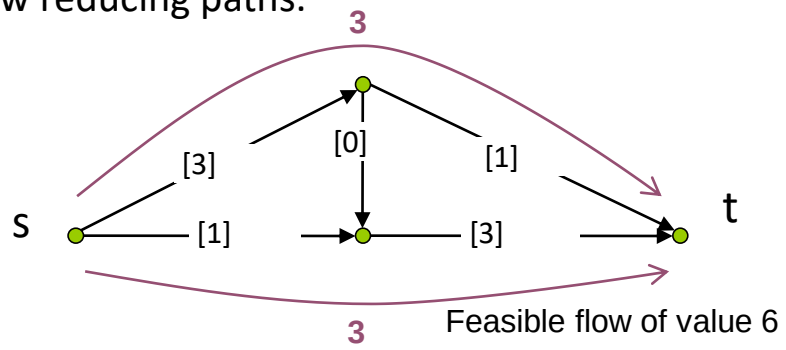


# Generalizations of max flow

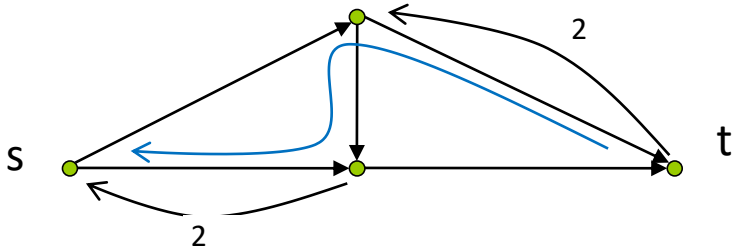
e) There are lower bounds (and no upper bounds) on the flow through each arc.
Goal: Find min flow

Algorithm: Step 1: First find a feasible flow (= easy)
Step 2: Iteratively, reduce along flow reducing paths.

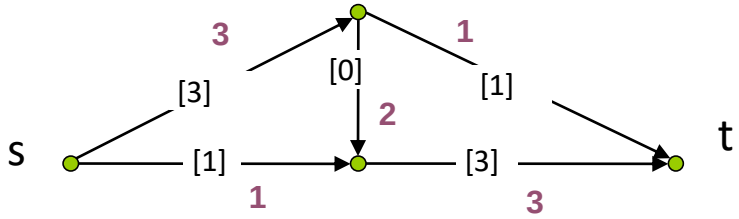



Feasible flow of value 6




Residual graph:
Numbers are upper bounds. No number means no (or infinite) upper bound.
Add flow of value 2 over the path.

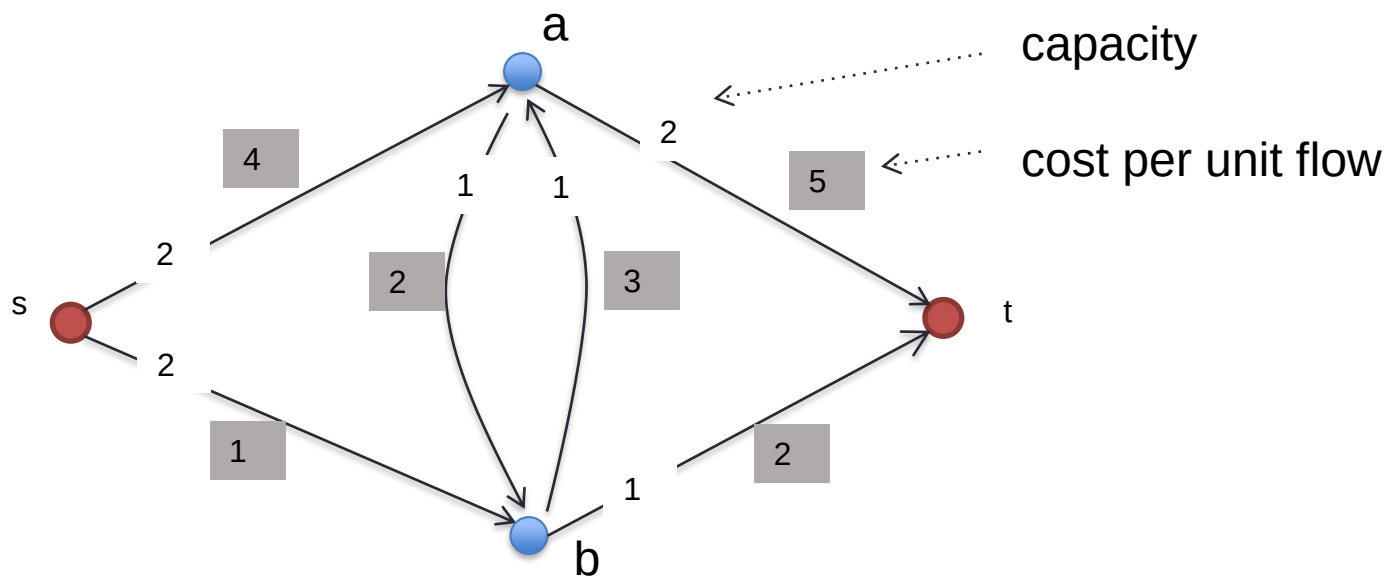



New (and minimum) flow.

# Min cost flow

Problem:

Given the network with costs and capacities and a flow value $v$, we need to find an $s-t$ flow of value $v$ of minimum cost.

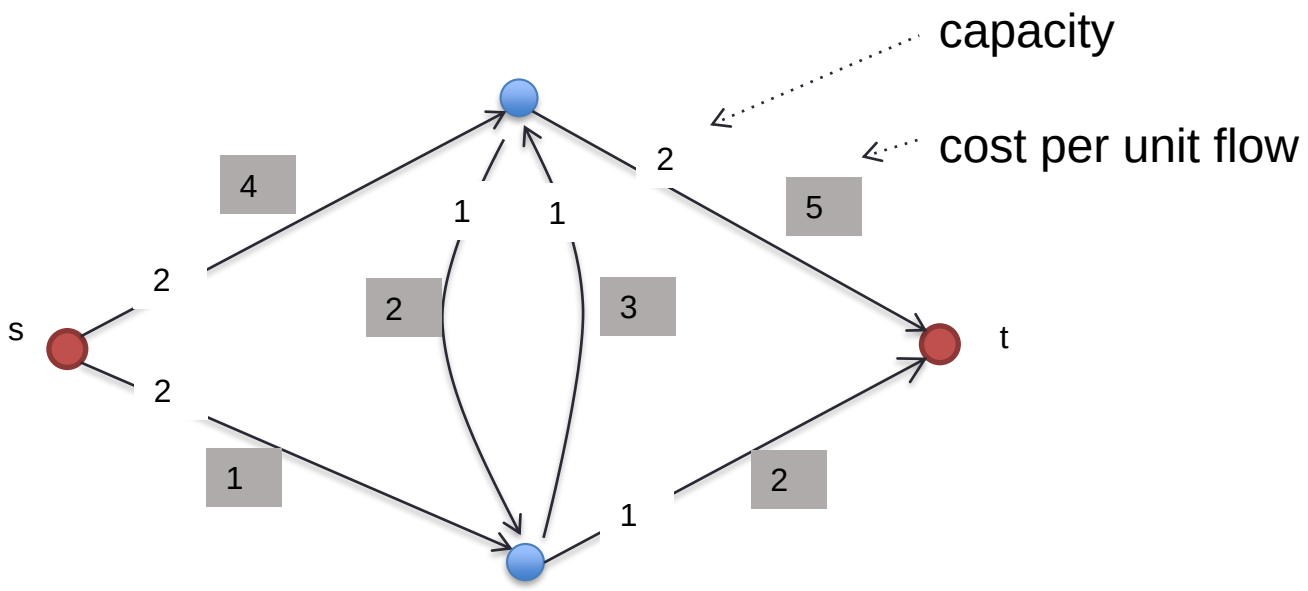



Example:

Sending 2 units over the path $s, a, t$ costs $2 \cdot 4 + 2 \cdot 5 = 18$.

Is this the cheapest flow of value 2?

# Min cost flow




**Algorithm:**

Step 1: Find a feasible flow of value *v* (for example by FF). Construct residual graph (negative of the cost for reverse arcs)

Step 2: While there is a negative-cost cycle in residual graph:

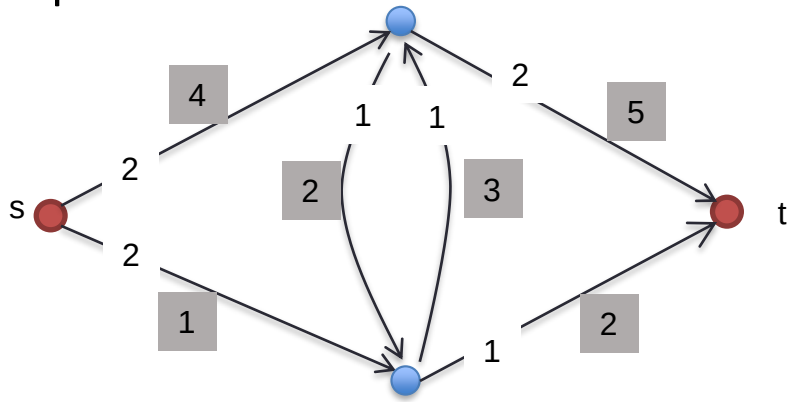
Add flow over the cycle.

Update the residual graph.

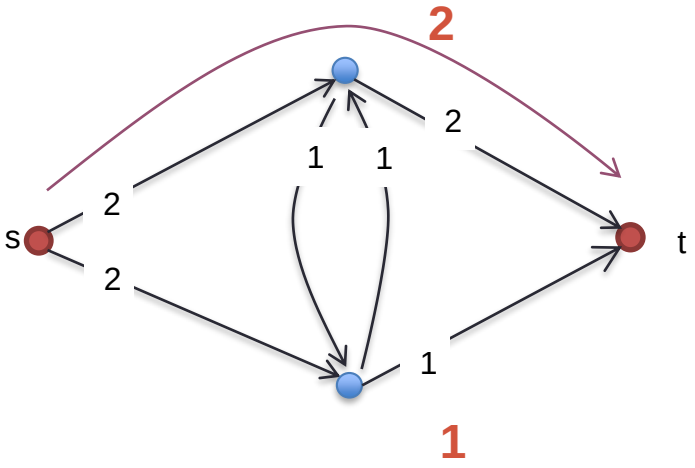
How to find?
e.g. Bellman Ford algorithm.
(Not in this course)

# Min cost flow

Example: Find a min cost flow of value 2.

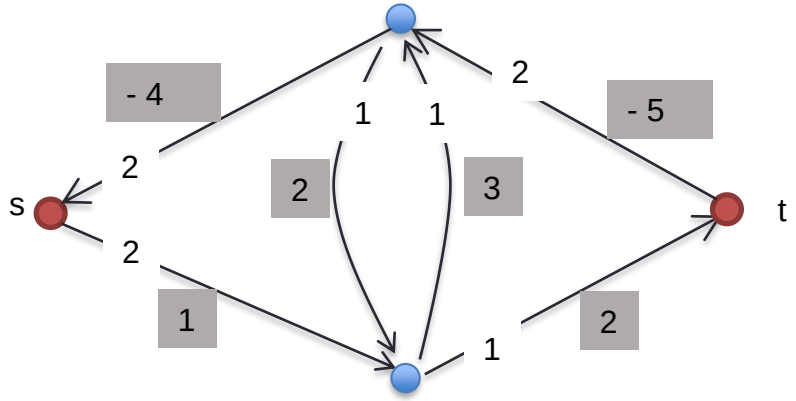



Given network.




Initial flow of value 2.

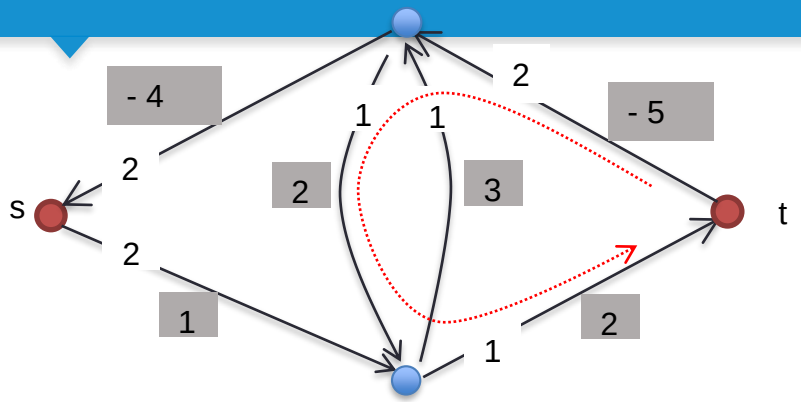
Cost = 2·4+2·5=18




1st residual graph

Is there a negative cycle?

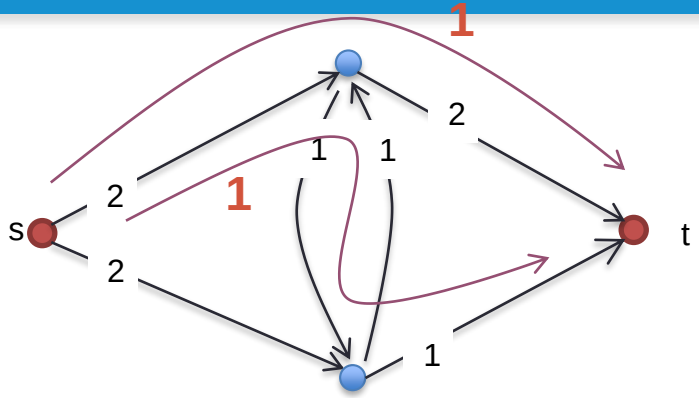
# Min cost flow




1st residual graph.

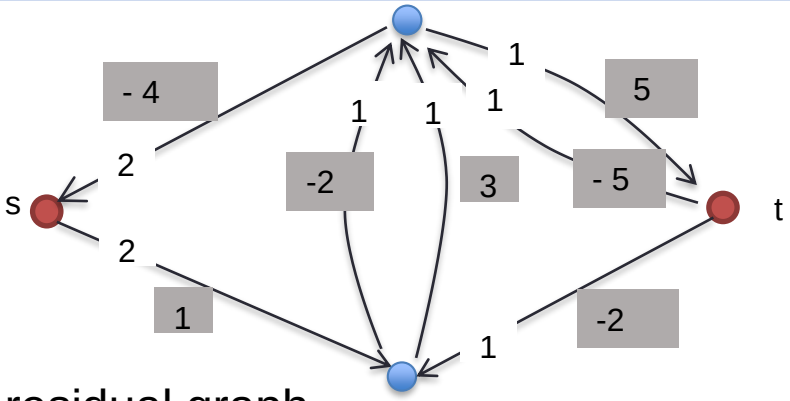
Negative cycle of cost: -5+2+2= -1

Min residual capacity is 1.Send 1 unit along this cycle.




Network with the updated flow.

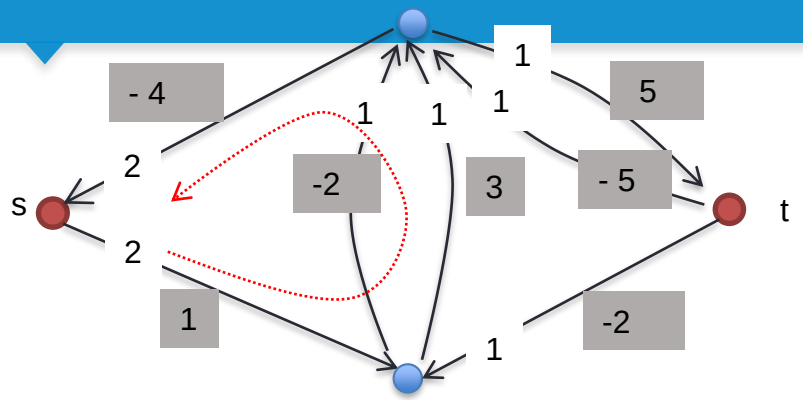
Cost is reduced by 1. New cost=17.




2nd residual graph

Is there a negative cycle?

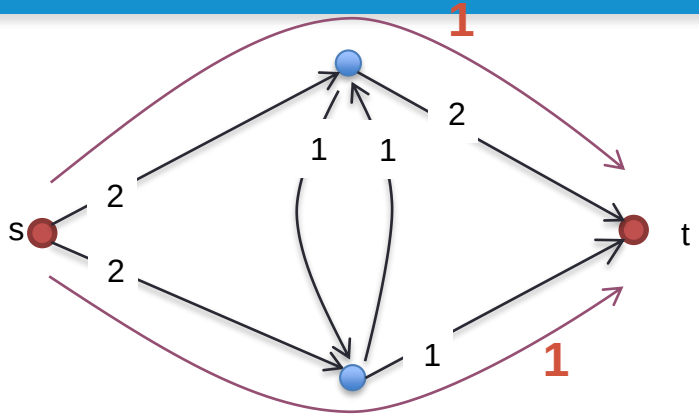
# Min cost flow




2nd residual graph.

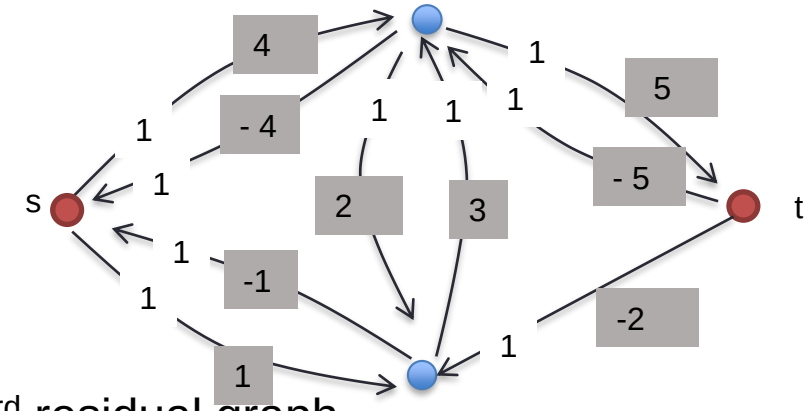
Negative cycle of cost: 1-2-4= -5

Min residual capacity is 1. Send 1 unit along this cycle.




Network with the updated flow.

Cost is reduced by 5. New cost=12.




3rd residual graph

Is there a negative cycle?

No. -> The flow has minimum cost.

# Min cost flow:

**Theorem:**

A flow is a minimum cost flow (of given value v)

Residual graph has no negative-cost cycles

**Corollary:**

The algorithm always returns the minimum cost flow.