# Algoritmer, datastrukturer och komplexitet

## Övning 5

Anton Grensjö
grensjo@csc.kth.se

5 oktober 2015

# Kursplanering

F12: Grafer: MST och Dijkstra

Ö4: Dynamisk programmering

F13: Grafer: Maximalt flöde

F14: Undre gränser

F15: Beräkningsgeometri

**Ö5: Grafalgoritmer och undre gränser**

F16: Sortering i linjär tid

F17: Textsökning

Ö6: Algoritmkonstruktion

# Mästarprov 1

- Mästarprov 1 finns ute nu.
- Deadline tisdagen den 13 oktober.
- Generella tips:
  - Börja i tid.
  - Läs lydelserna noga.
  - Fråga om eventuella oklarheter.
  - Skriv fullständiga lösningar.
  - Titta på lösningar till gamla mästarprov.
  - Läs betygskriterierna och kolla vad som förväntas för vilka betyg.
- Kom ihåg: mästarprovet ska lösas strikt individuellt.

# Idag

- Grafalgoritmer
  - Spännande träd
  - Flöde och matchning (bra för labb 3!)
  - Eulercykler
- Undre gränser

# Spännande träd

## Definition

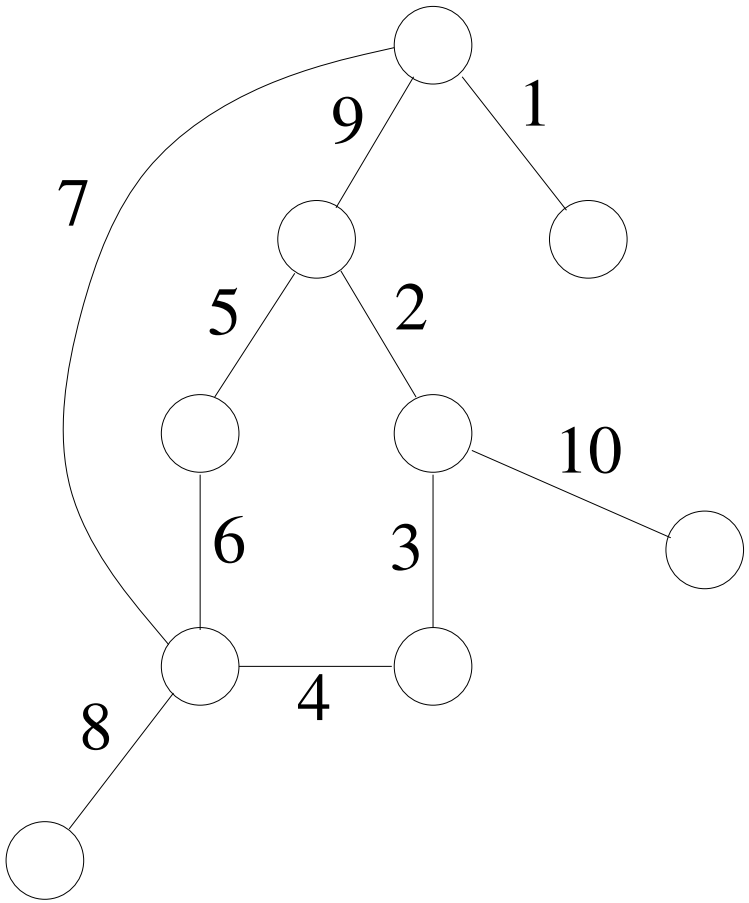
Ett spännande träd till en graf $G$ är en delgraf $G'$ till $G$, sådan att:

- $G'$ är ett träd.
- $G'$ har samma hörn som $G$.

- Vad innebär detta?
- Låt vikten för ett spännande träd vara summan av de ingående kanternas vikter.
- Det minimalt spännande trädet (MST) till en graf $G$ är det spännande träd till $G$ som har lägst vikt.

# Uppgift 1: Spännande träd

Visa hur ett minimalt spännande träd hittas i följande graf med både Prims och Kruskals algoritmer.



# Uppgift 1: Spännande träd

## Prims algoritm

- Idé: modifierad BFS.
- Vi väljer en nod att börja med, och lägger successivt till kanter, tills dess att alla noder kan nås.
- Spara alla upptäckta noder i en prioritetskö, med avseende på kostnaden för att lägga till noden.
- Om vi upptäcker en nod igen, uppdatera kostnaden om den är lägre.
- I varje steg: välj den nod som det är billigast att lägga till.
- Vårt MST kommer att växa “inifrån och ut”.

# Uppgift 1: Spännande träd

## Kruskals algoritm

- Idé: välj hela tiden den billigaste kanten som gör någon nytta (dvs ej bildar en cykel).

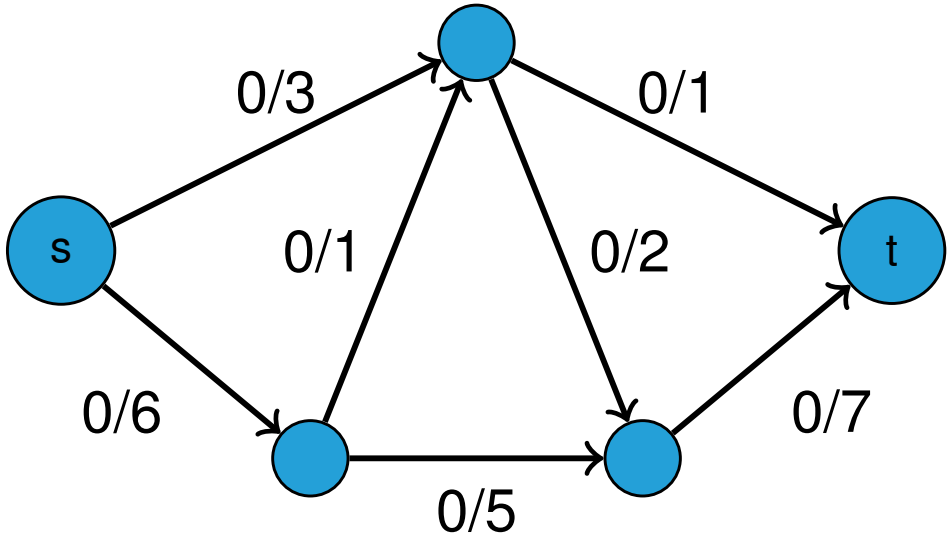
1. Sortera kanterna efter stigande vikt.
2. För varje kant $(u, v)$ i den ordningen:
   - Om $u$ och $v$ inte ännu är sammankopplade i trädet, lägg till kanten.
   - Annars: gör inget.

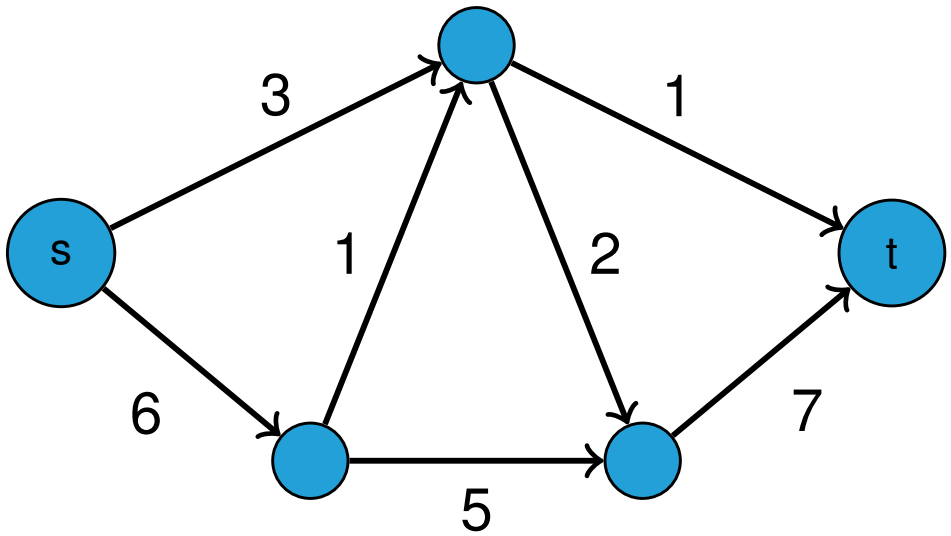
- Hur kan vi effektivt hålla koll på vilka noder som redan är sammankopplade?
- Datastrukturen Union-Find!
- Om ni är intresserade, läs mer på s. 151 i Kleinberg-Tardos.

# Uppgift 2: Förändrat flöde

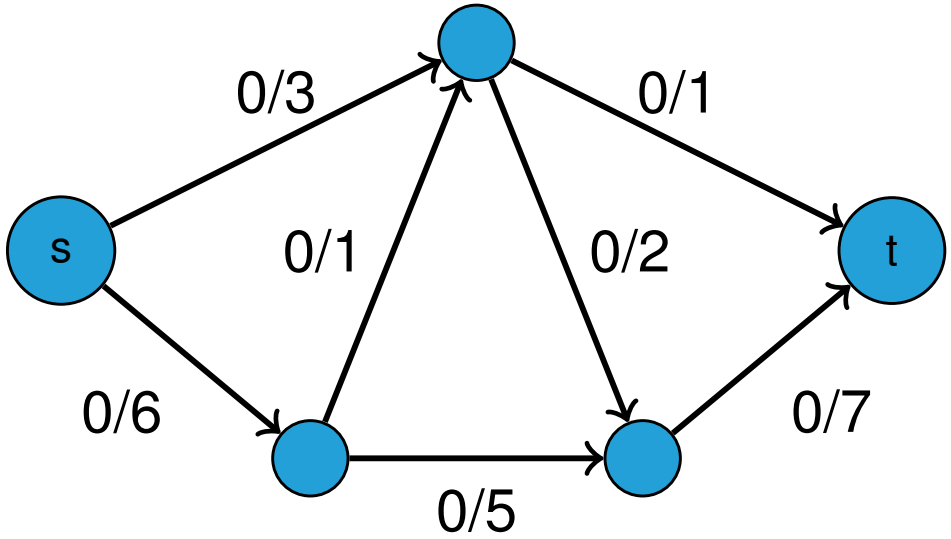
Låt oss först se ett exempel på hur man hittar maximala flödet i en graf med hjälp av Ford-Fulkerson.

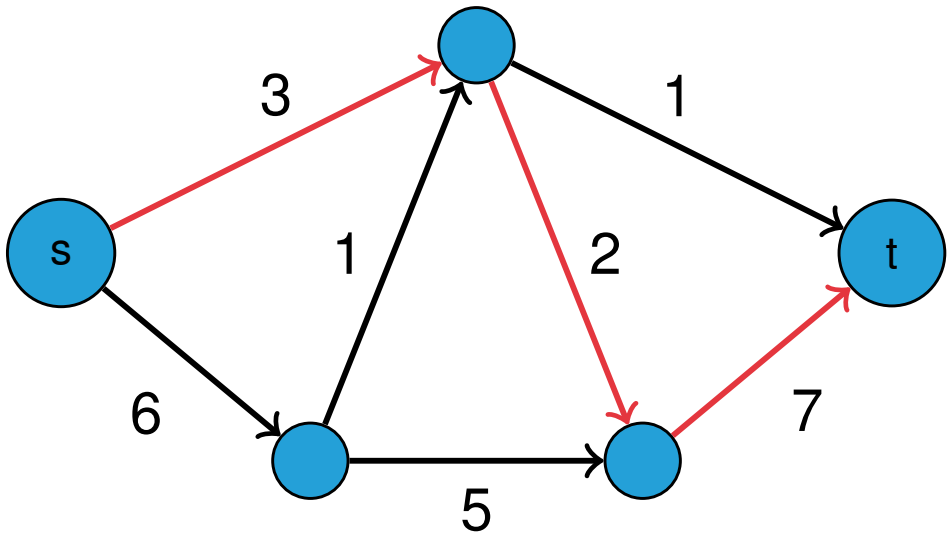
# Uppgift 2: Förändrat flöde



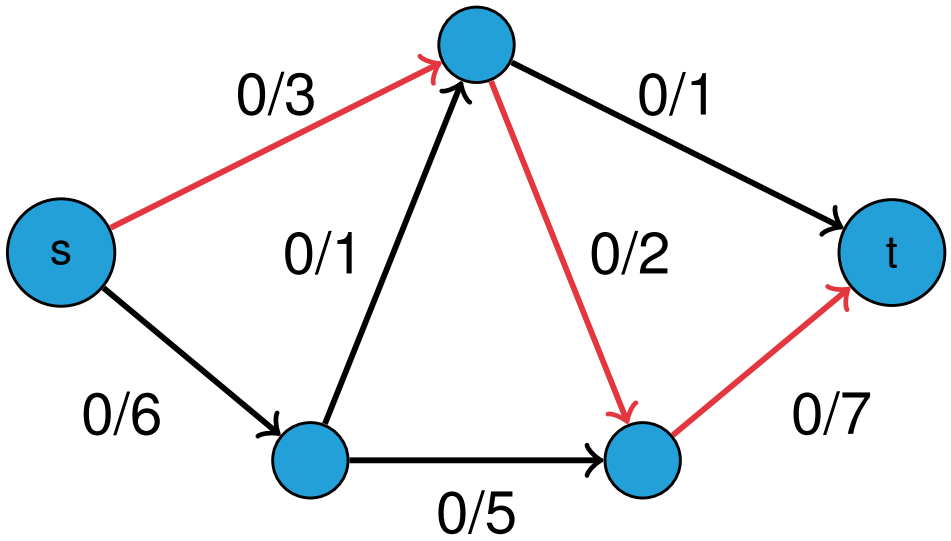



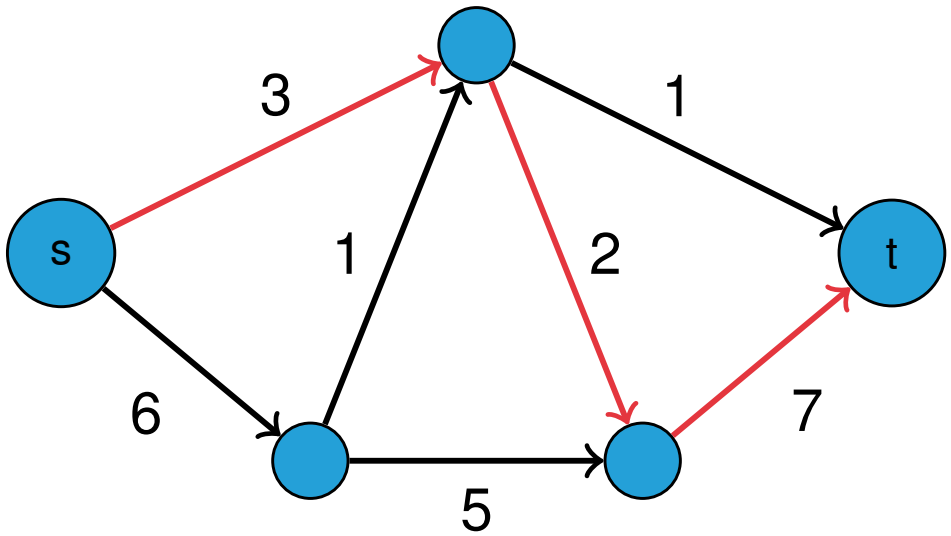
# Uppgift 2: Förändrat flöde



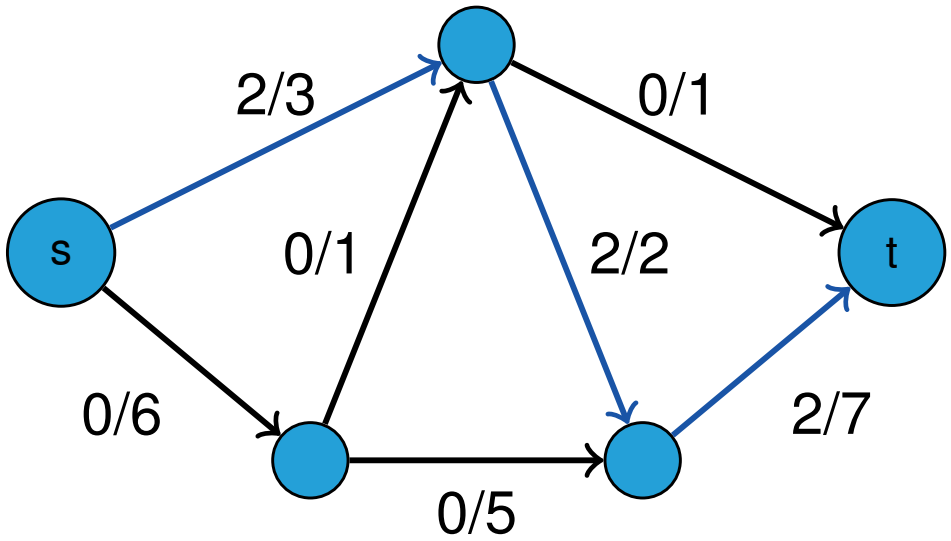



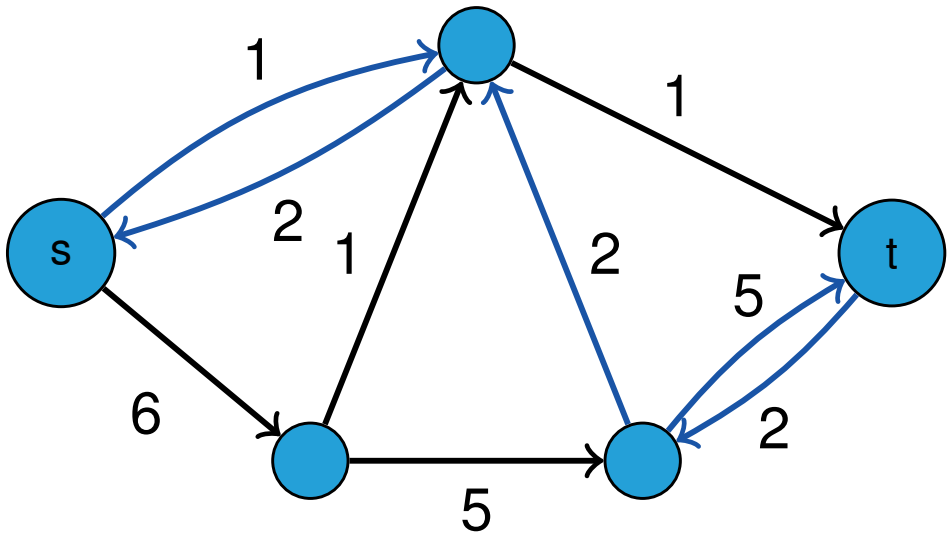
# Uppgift 2: Förändrat flöde



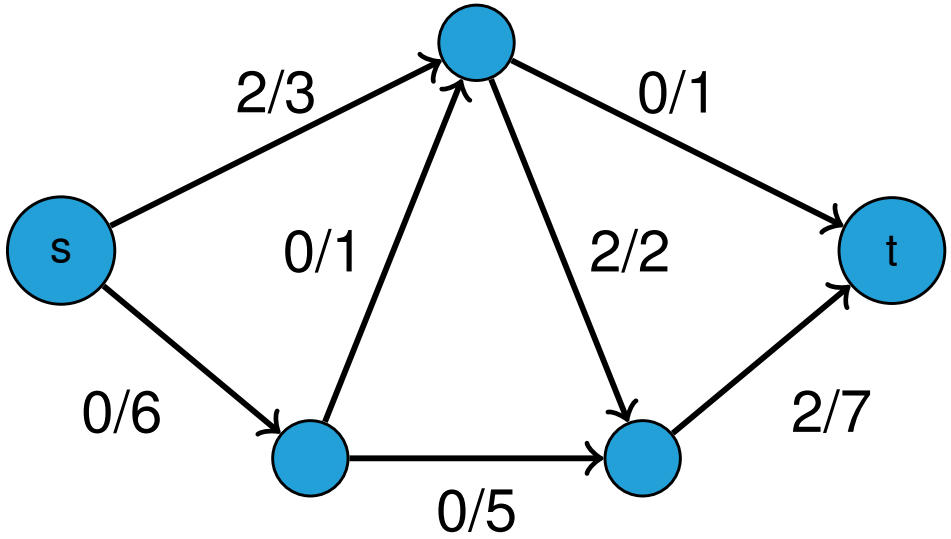



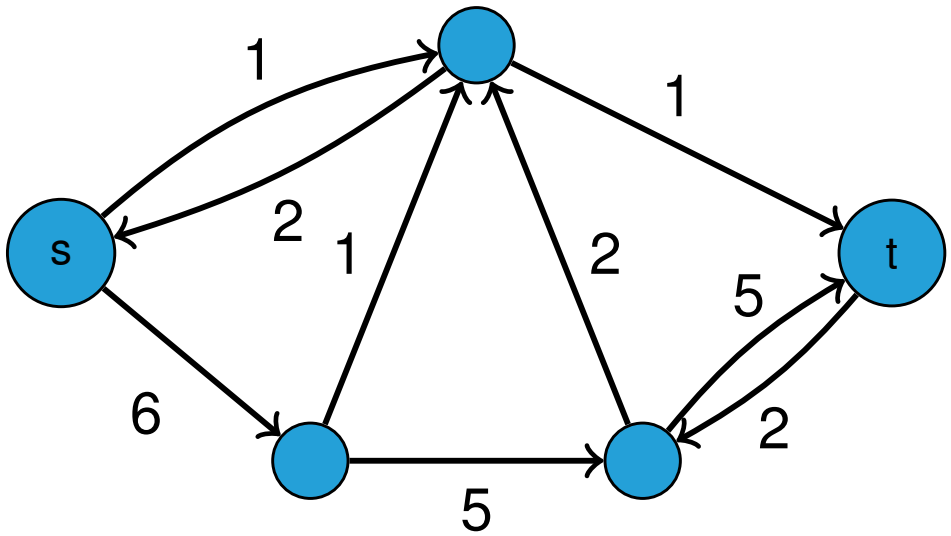
# Uppgift 2: Förändrat flöde



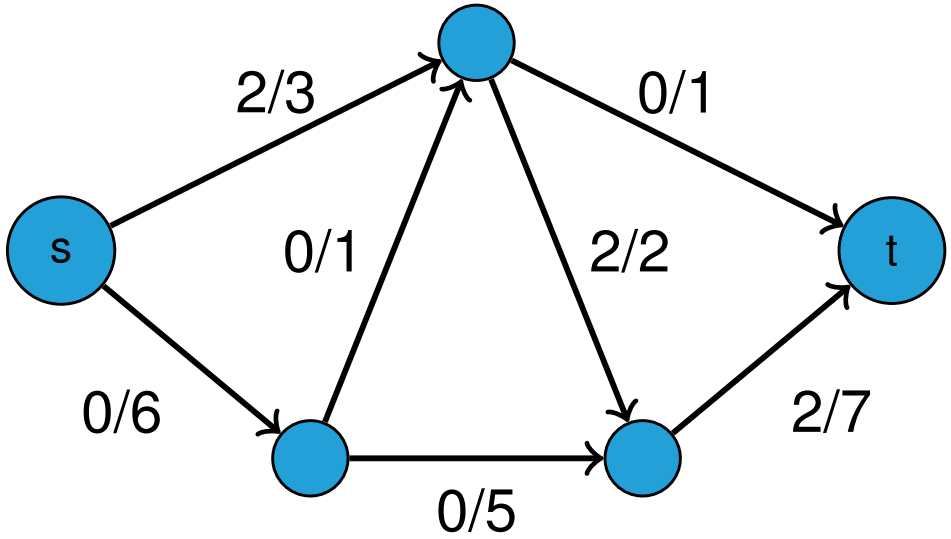



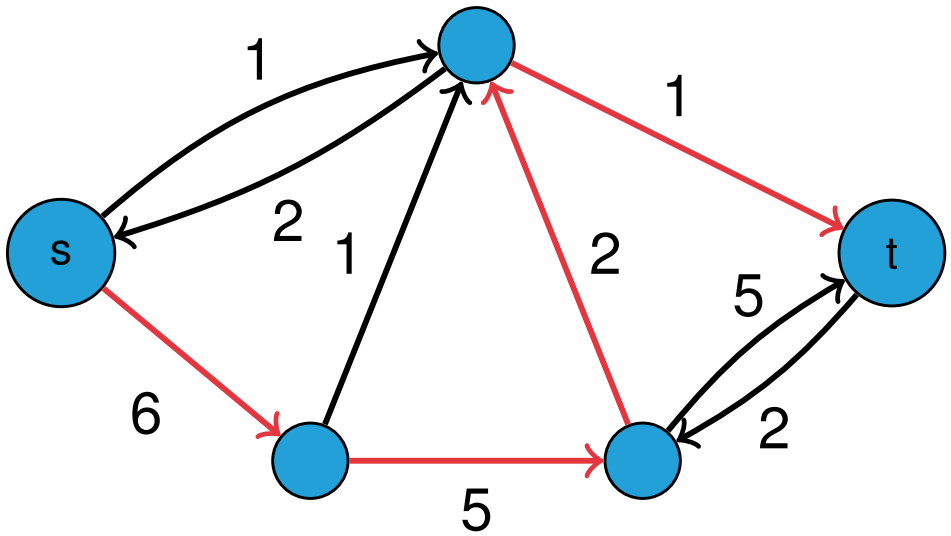
# Uppgift 2: Förändrat flöde



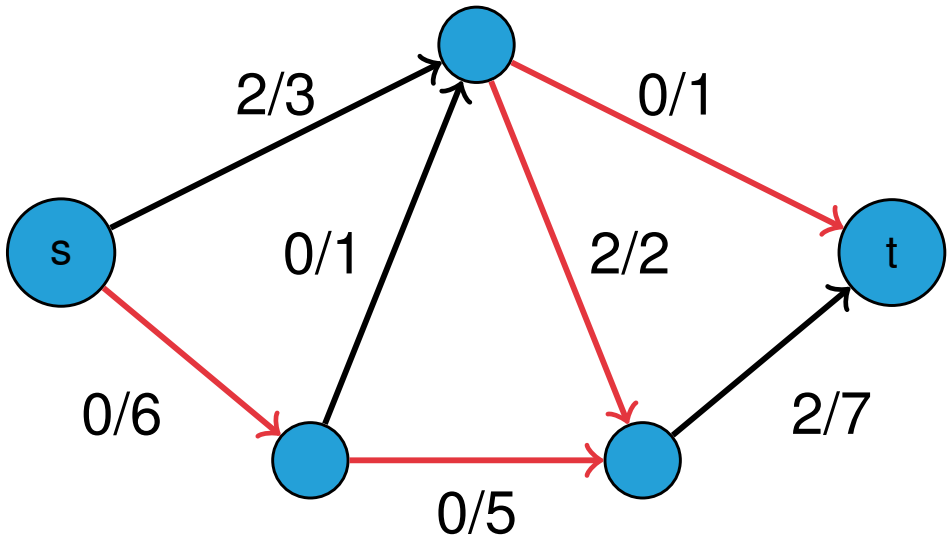



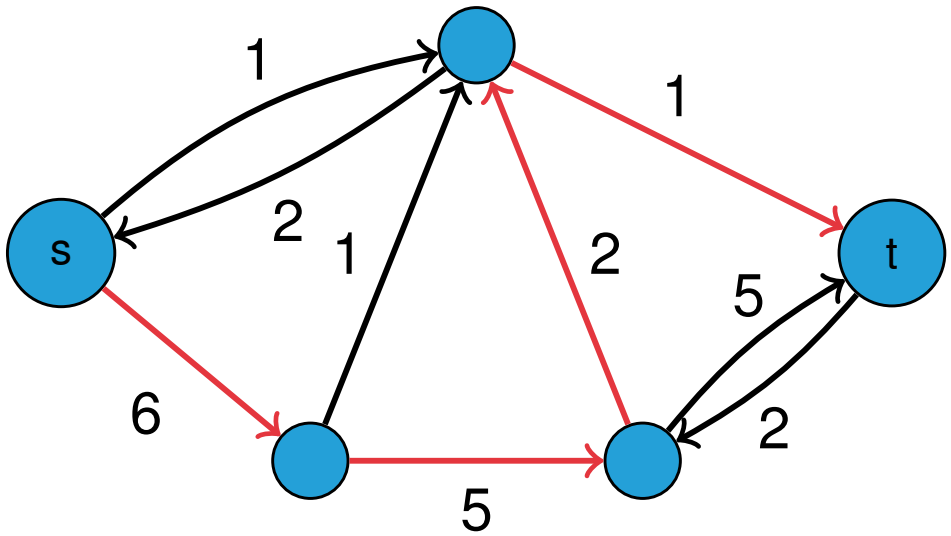
# Uppgift 2: Förändrat flöde



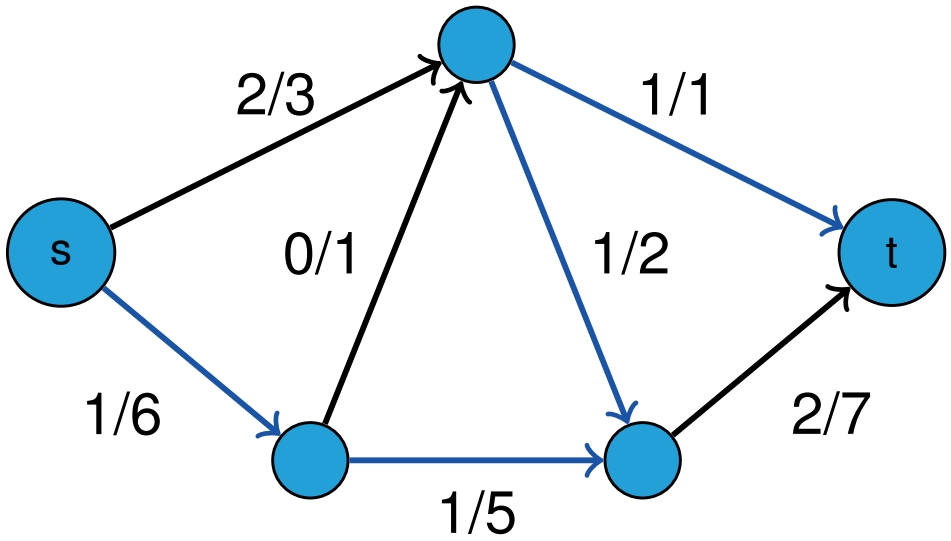



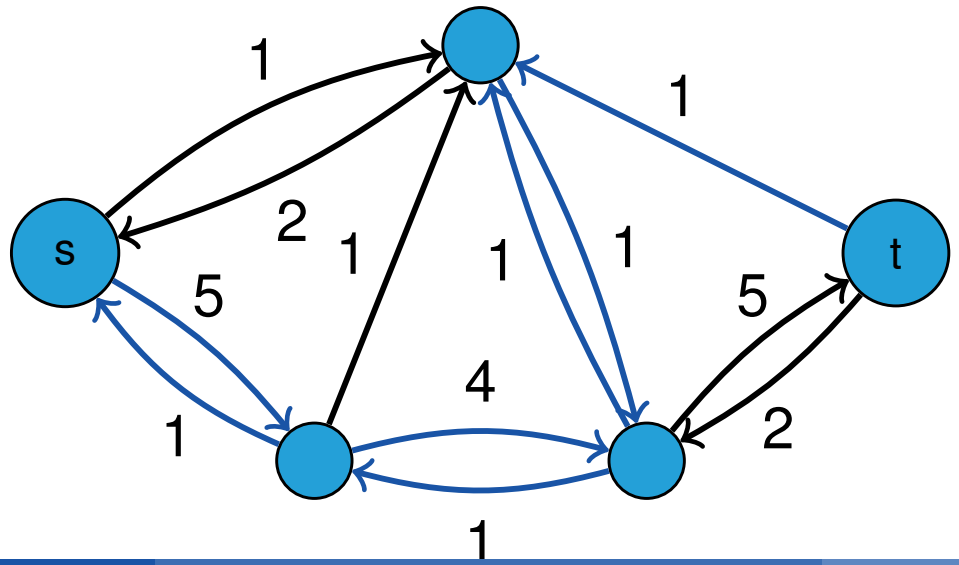
# Uppgift 2: Förändrat flöde



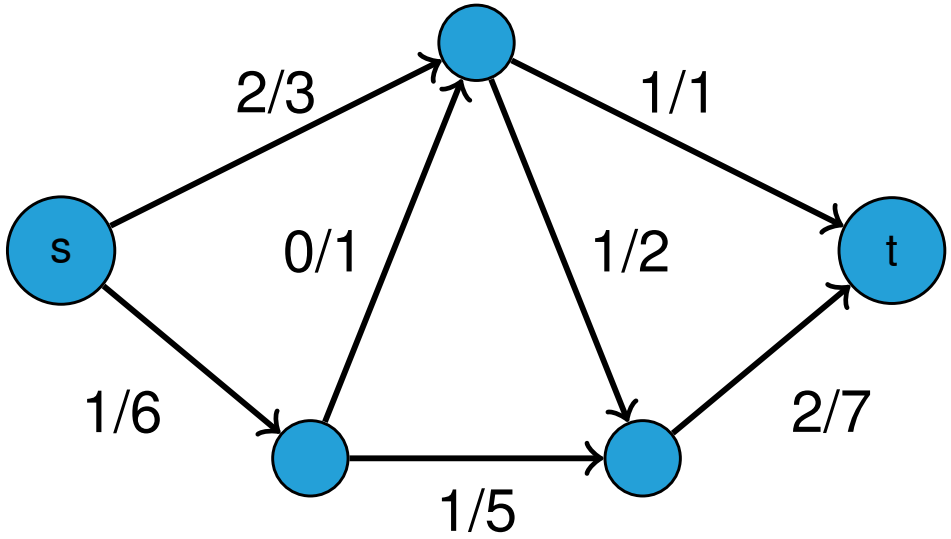



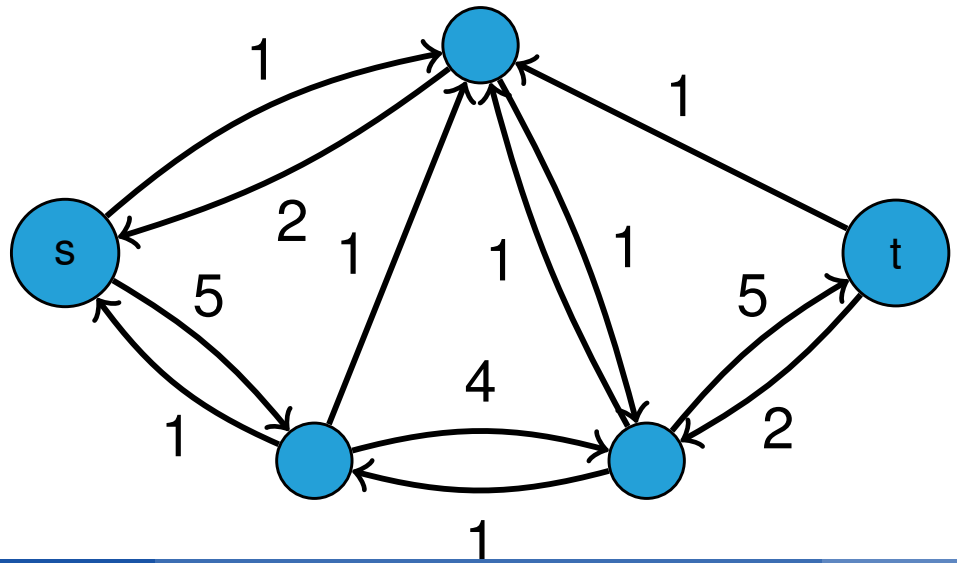
# Uppgift 2: Förändrat flöde



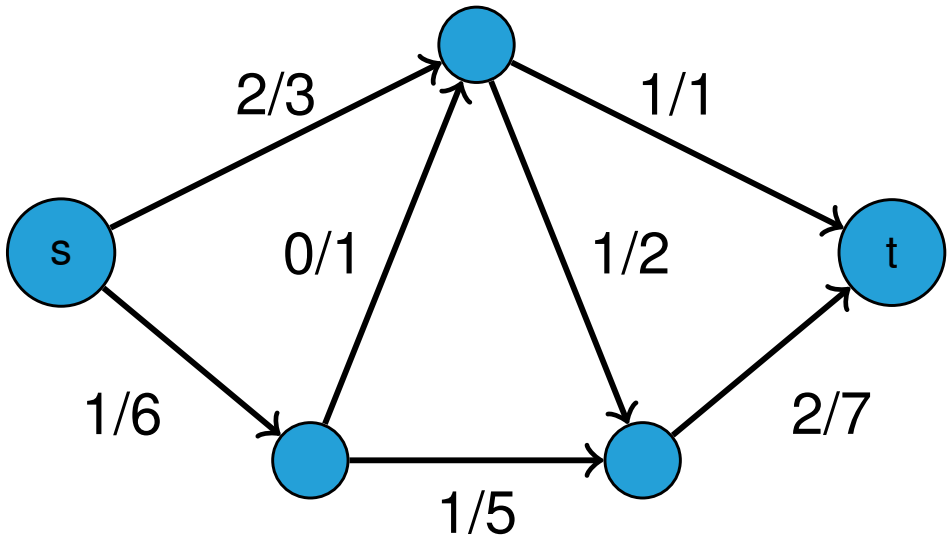



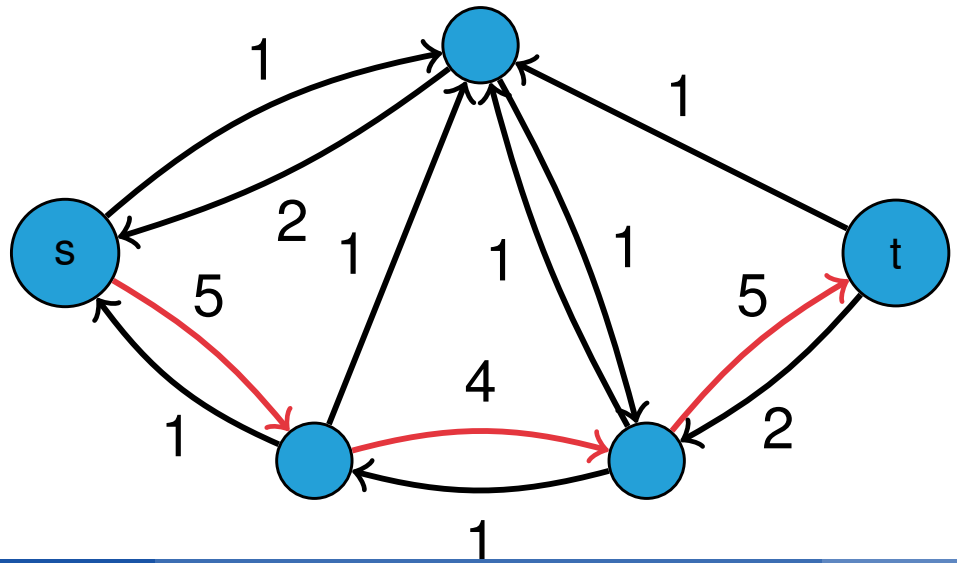
# Uppgift 2: Förändrat flöde



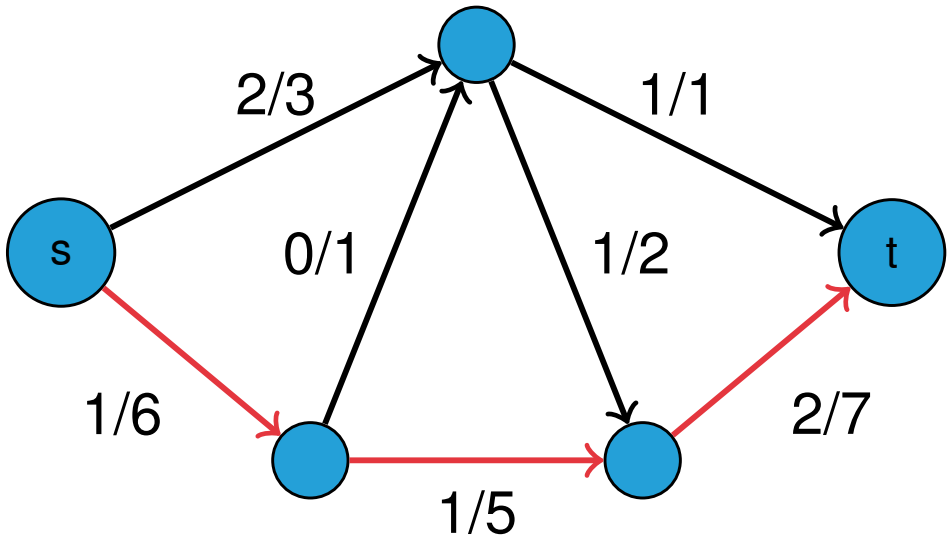



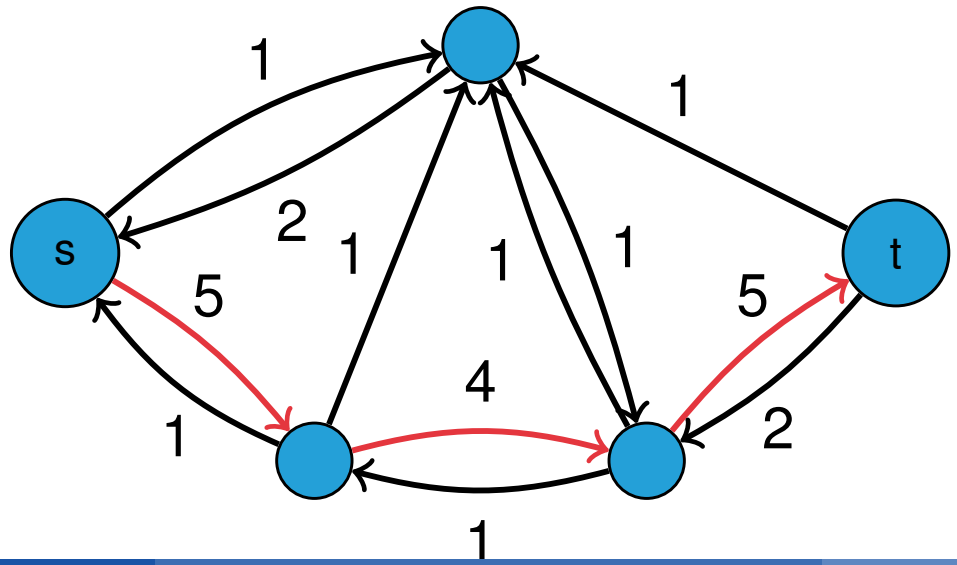
# Uppgift 2: Förändrat flöde



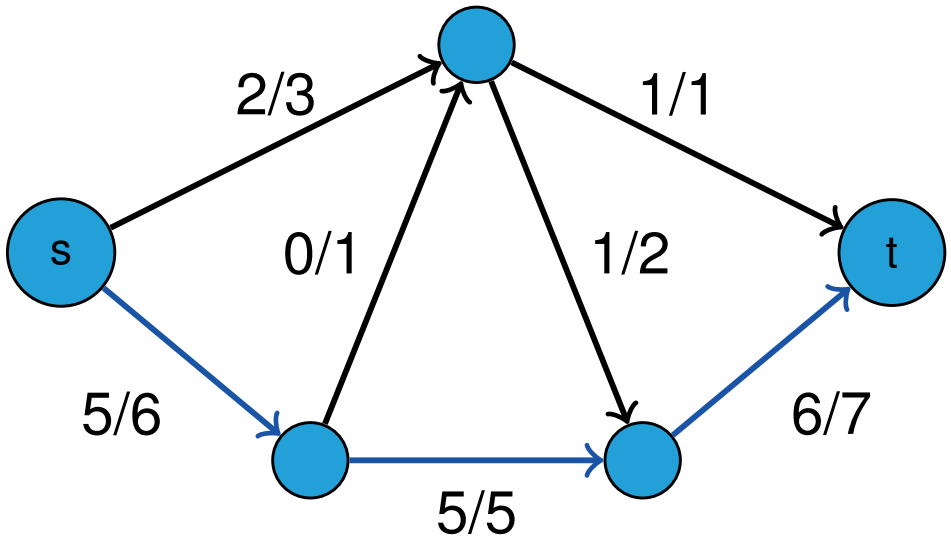



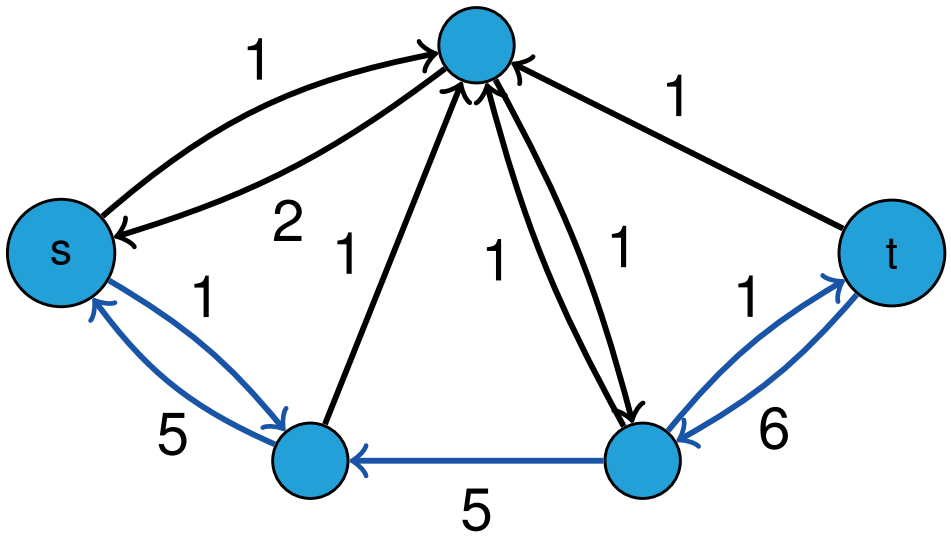
# Uppgift 2: Förändrat flöde



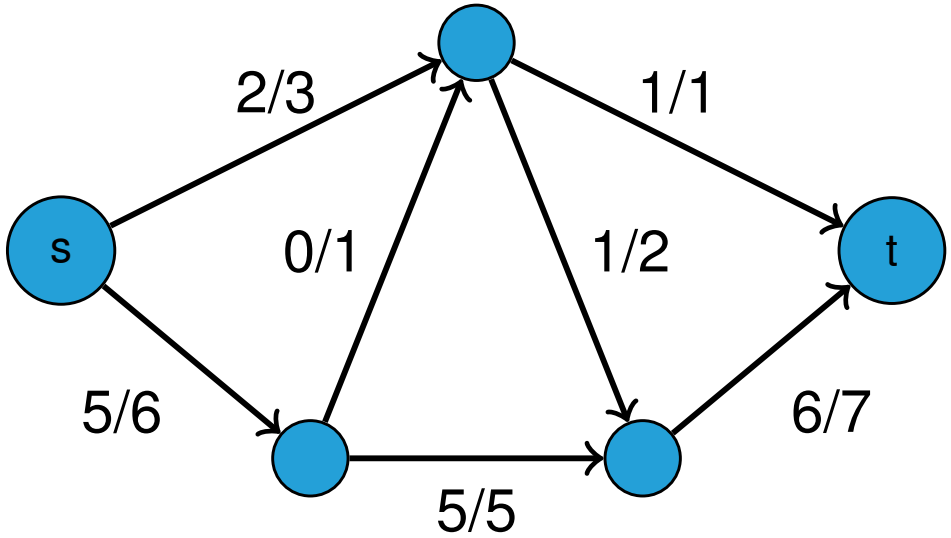



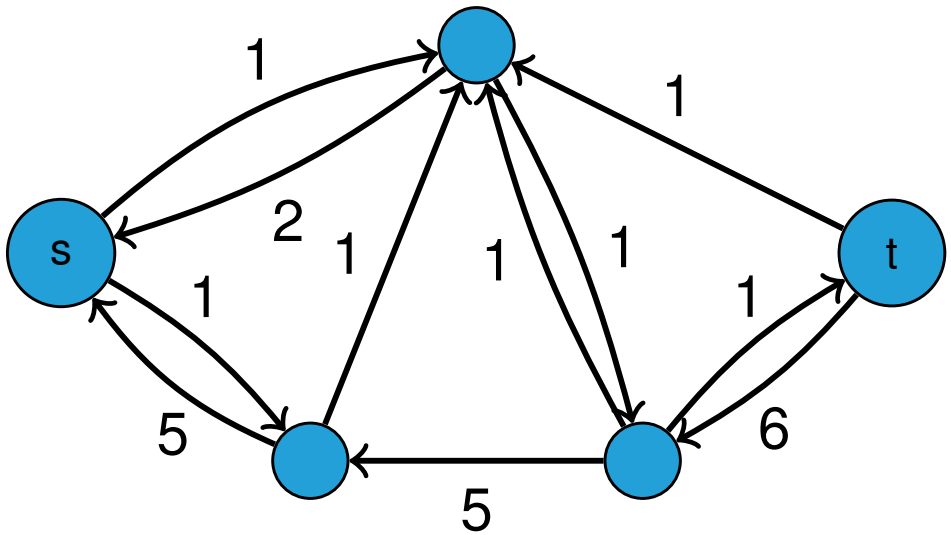
# Uppgift 2: Förändrat flöde



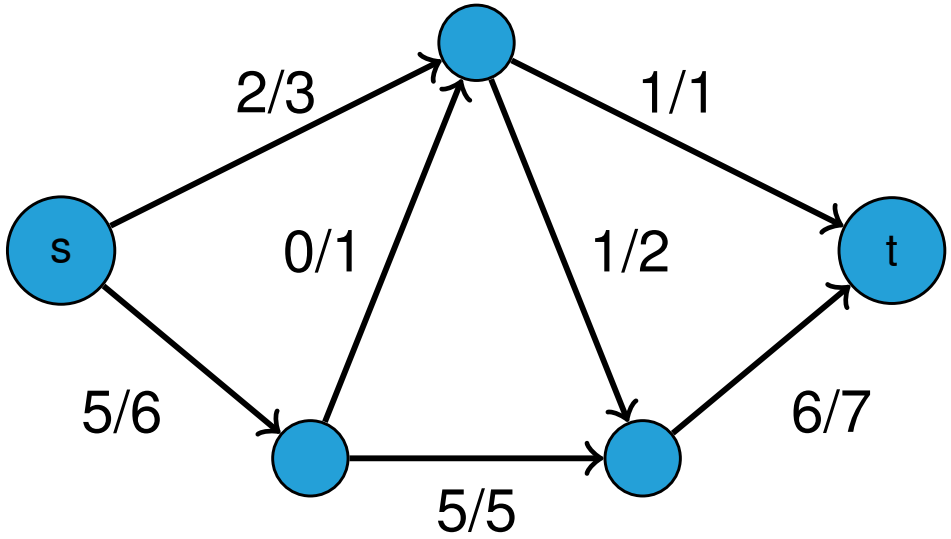



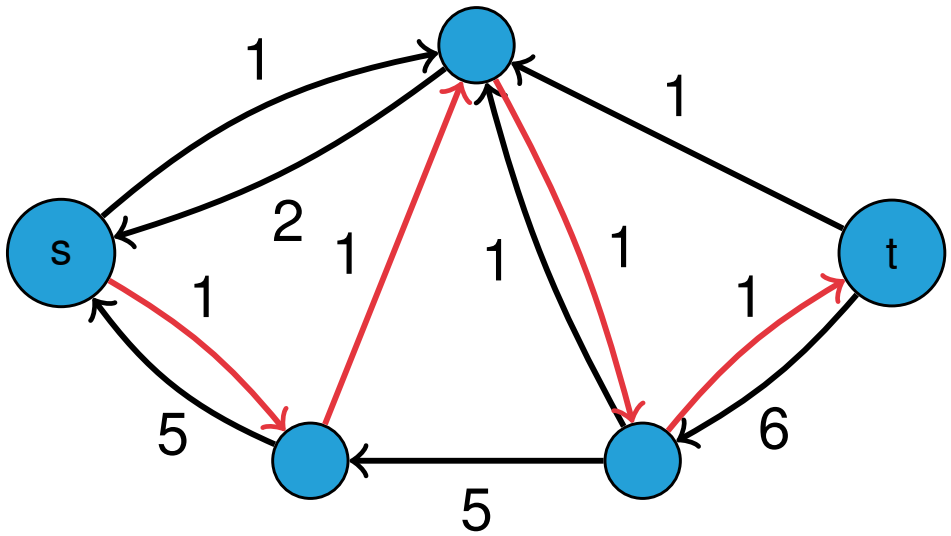
# Uppgift 2: Förändrat flöde



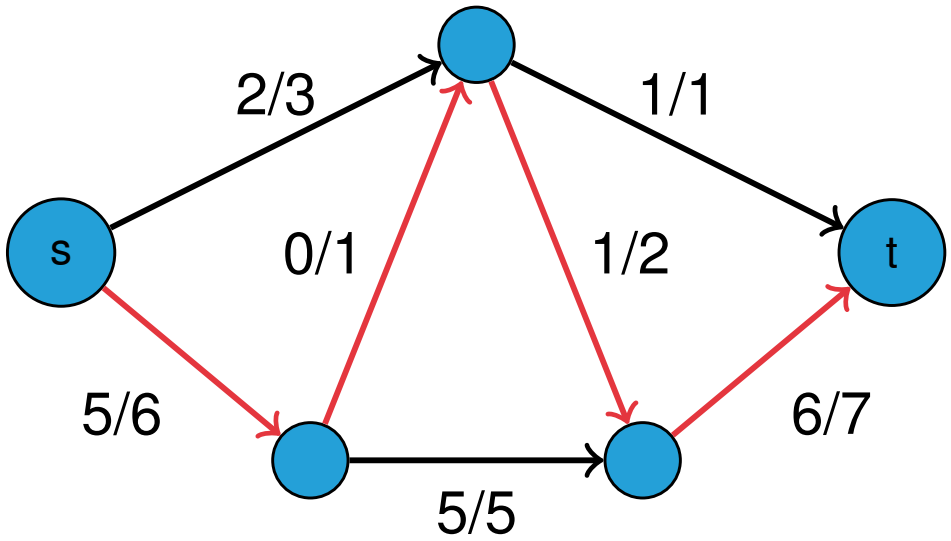



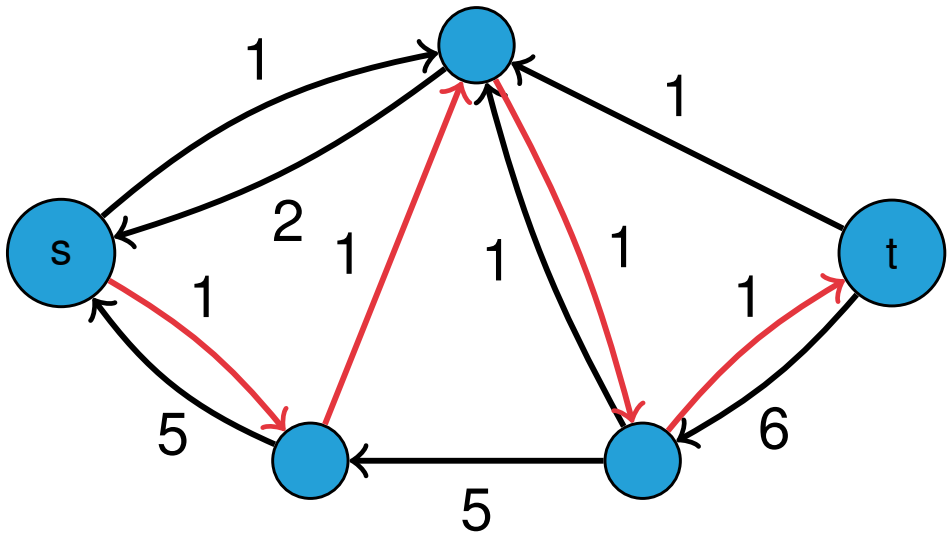
# Uppgift 2: Förändrat flöde



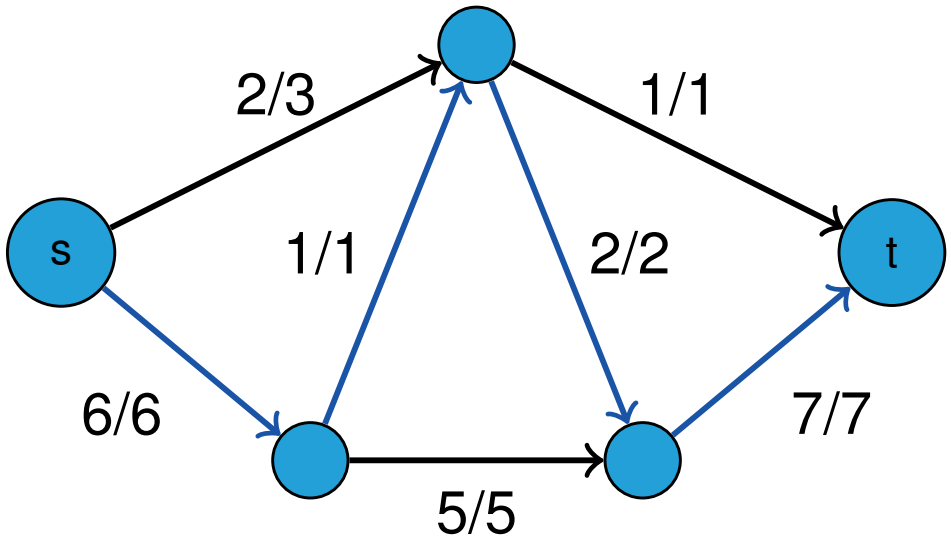



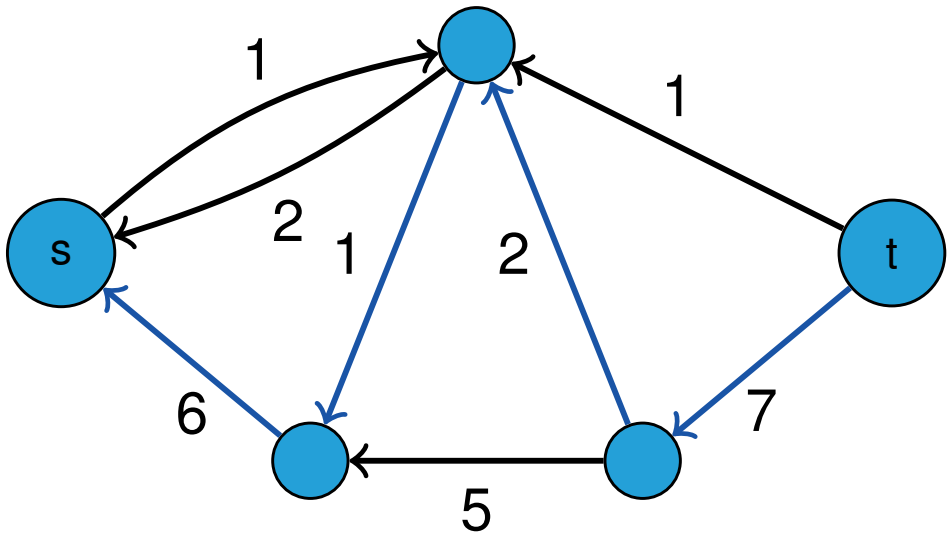
# Uppgift 2: Förändrat flöde



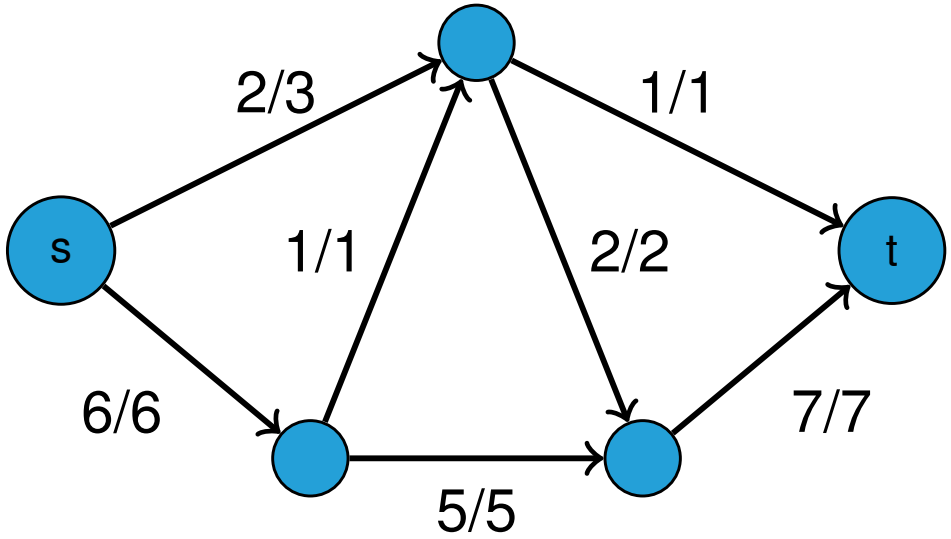



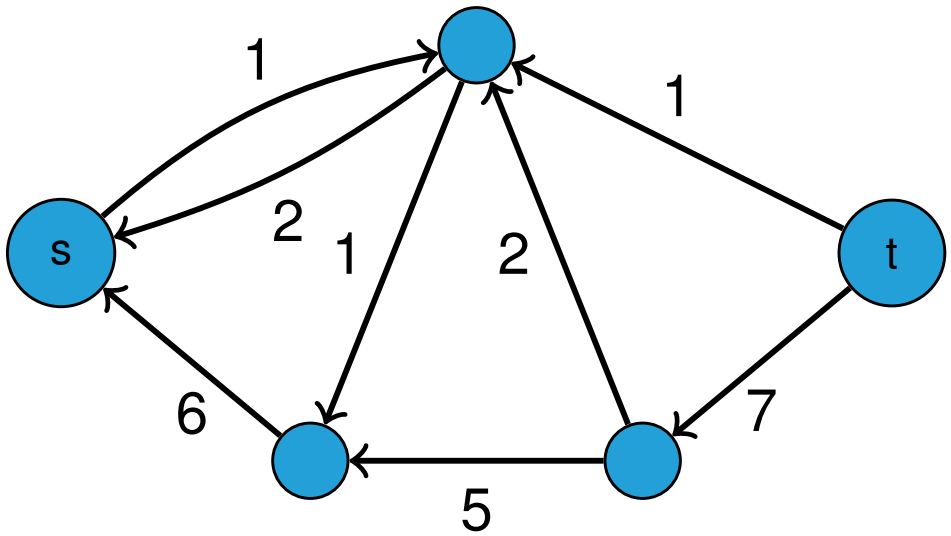
# Uppgift 2: Förändrat flöde



# Uppgift 2: Förändrat flöde



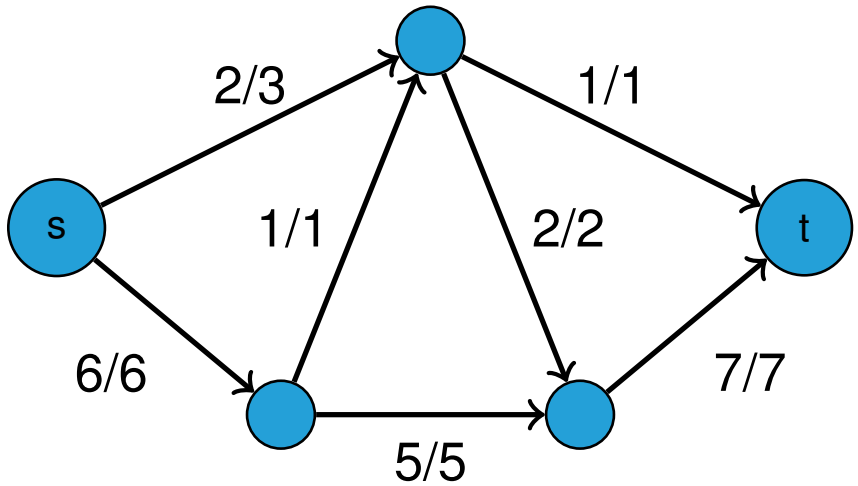
# Uppgift 2: Förändrat flöde

Antag att vi redan har hittat det maximala flödet i en graf.
Beskriv en algoritm som hittar ett nytt maximalt flöde om...

a) ...kapaciteten längs en viss kant ökar med en enhet.

b) ...kapaciteten längs en viss kant minskar med en enhet.

I bägge fallen ska tidskomplexiteten vara linjär, dvs $\mathcal{O}(|V| + |E|)$.



# Uppgift 2: Förändrat flöde

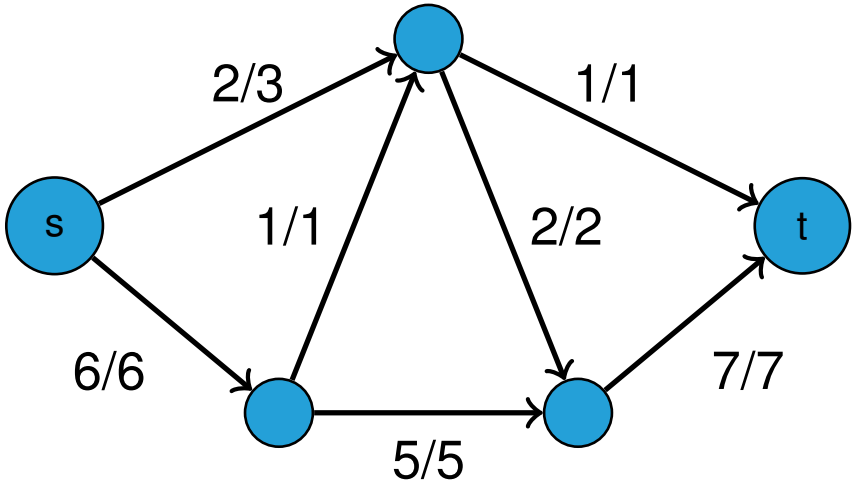
## Vad händer om kapaciteten ökar med en enhet?

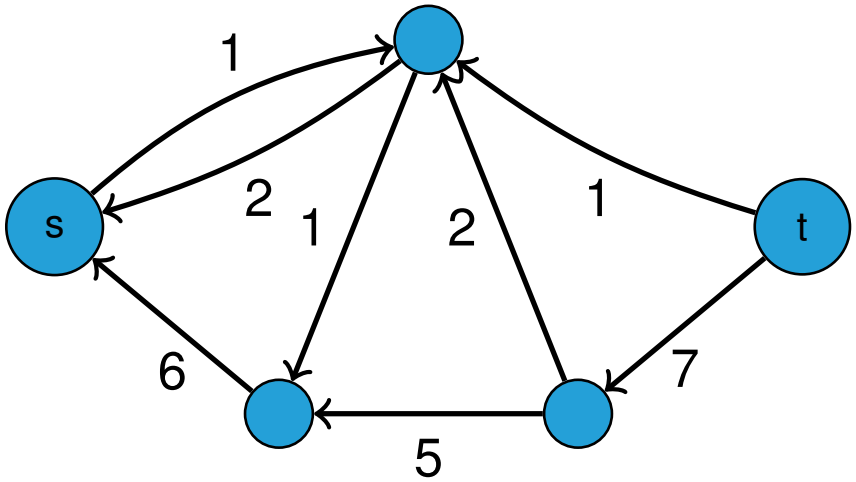
Om kapaciteten ökar:

- Gör en till iteration i Ford-Fulkerson, dvs leta efter en väg från $s$ till $t$ i restflödesgrafen.
- Finns det en väg så kan flödet ökas längs denna väg.
- Finns det ingen väg så händer ingenting; flödet är fortfarande optimalt.

# Uppgift 2: Förändrat flöde

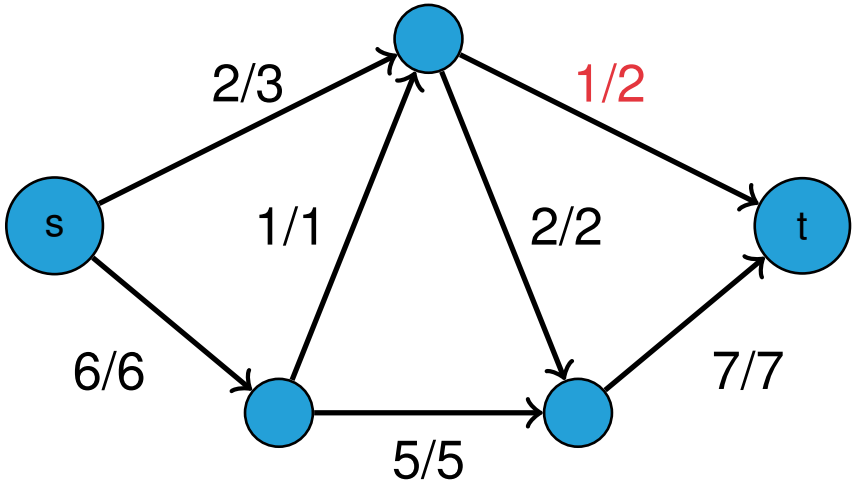
## Vad händer om kapaciteten ökar med en enhet?

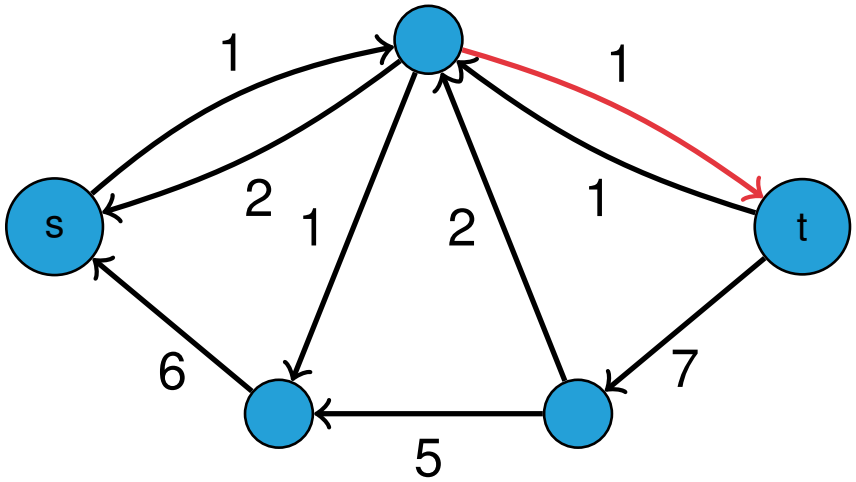


# Uppgift 2: Förändrat flöde

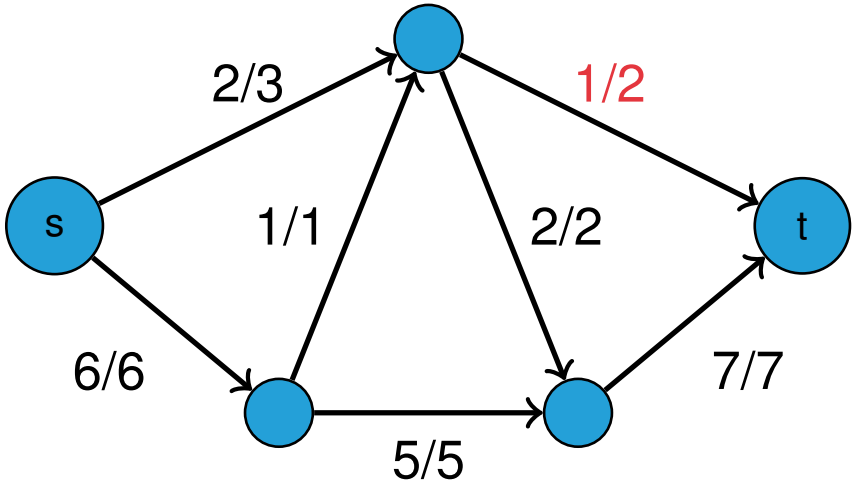
## Vad händer om kapaciteten ökar med en enhet?

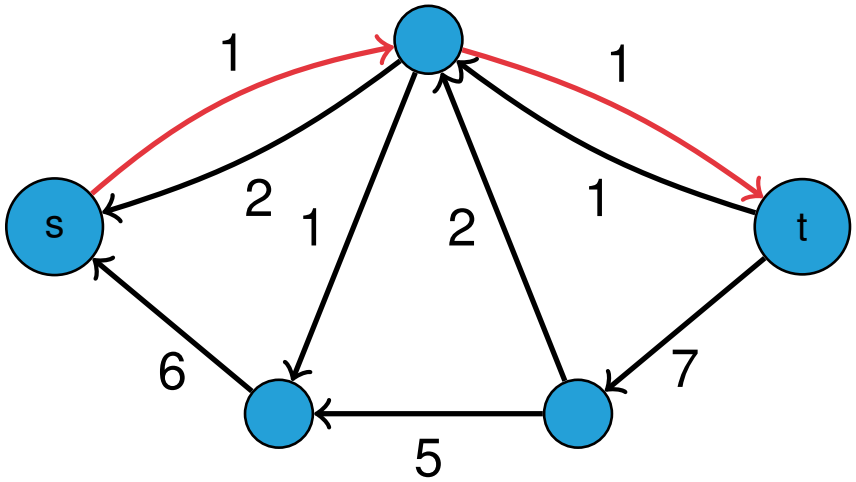


# Uppgift 2: Förändrat flöde

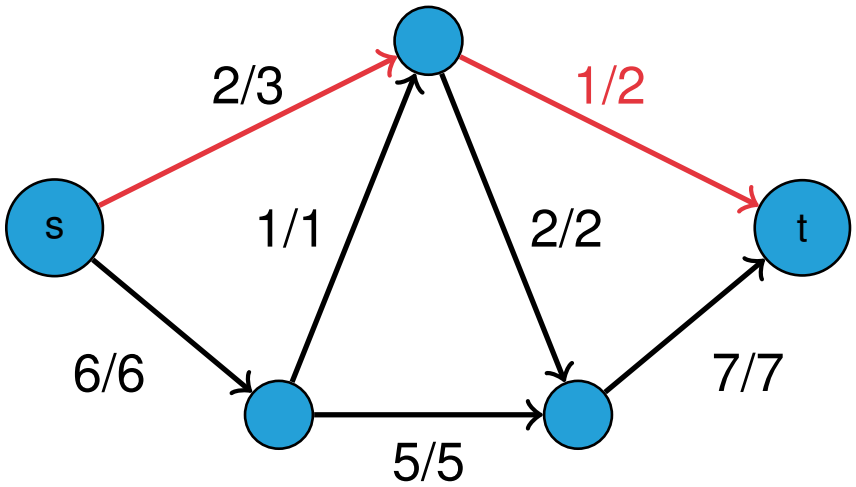
## Vad händer om kapaciteten ökar med en enhet?

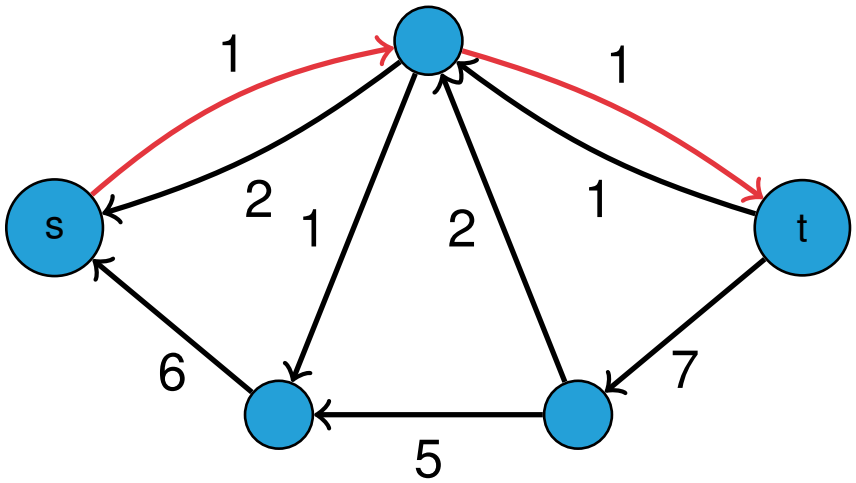


# Uppgift 2: Förändrat flöde

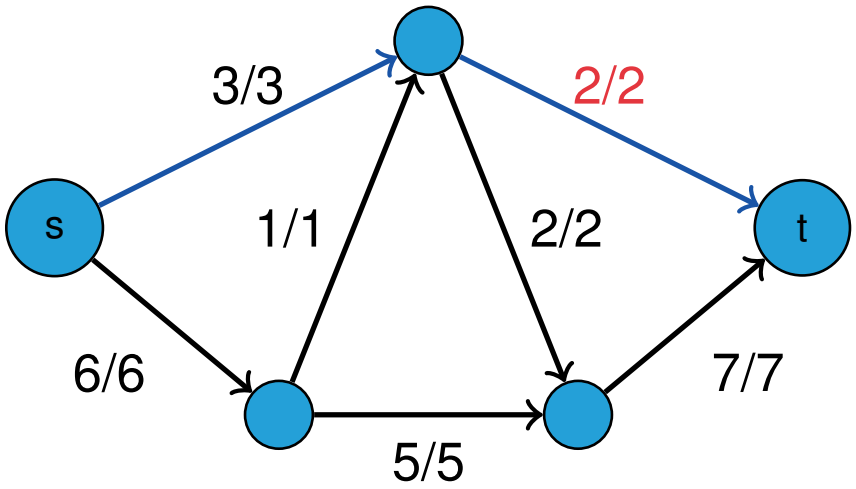
## Vad händer om kapaciteten ökar med en enhet?

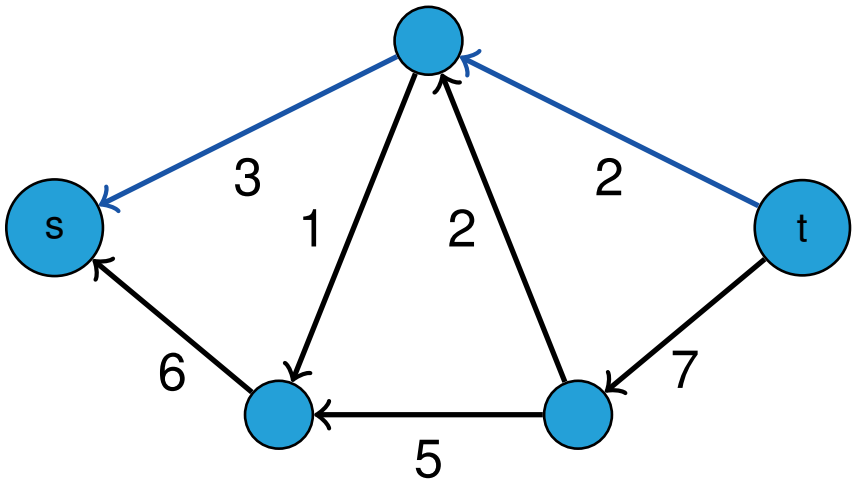


# Uppgift 2: Förändrat flöde

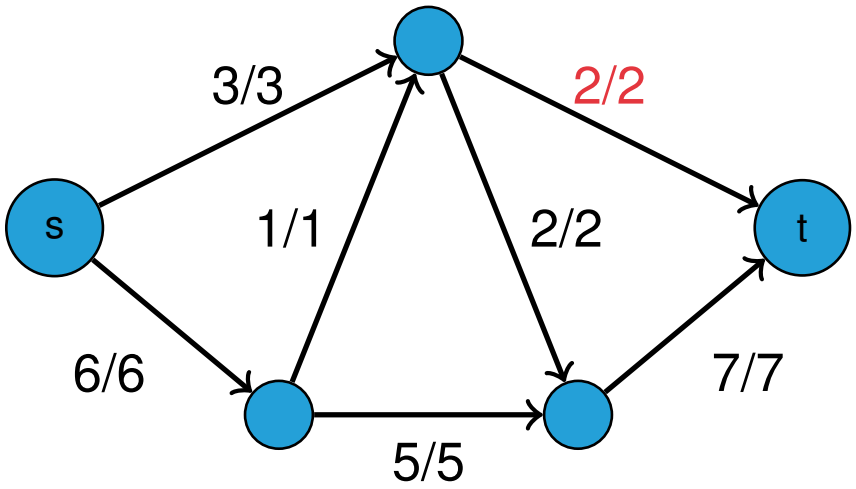
## Vad händer om kapaciteten ökar med en enhet?

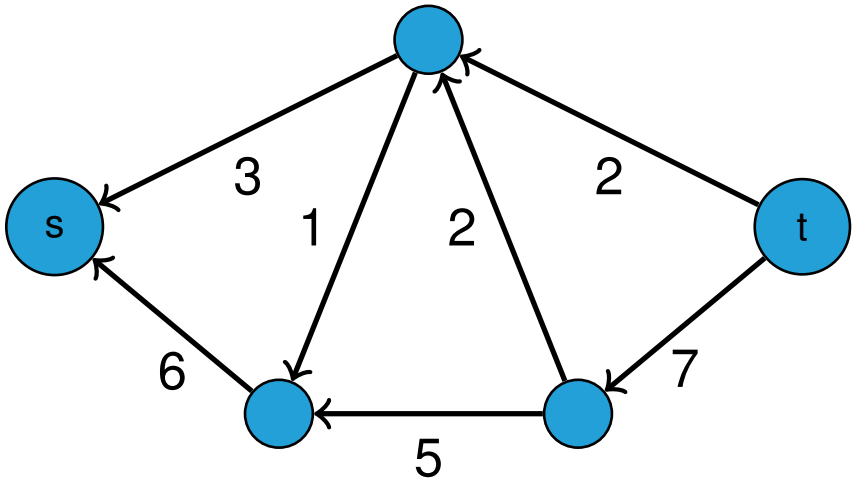


# Uppgift 2: Förändrat flöde

## Vad händer om kapaciteten ökar med en enhet?



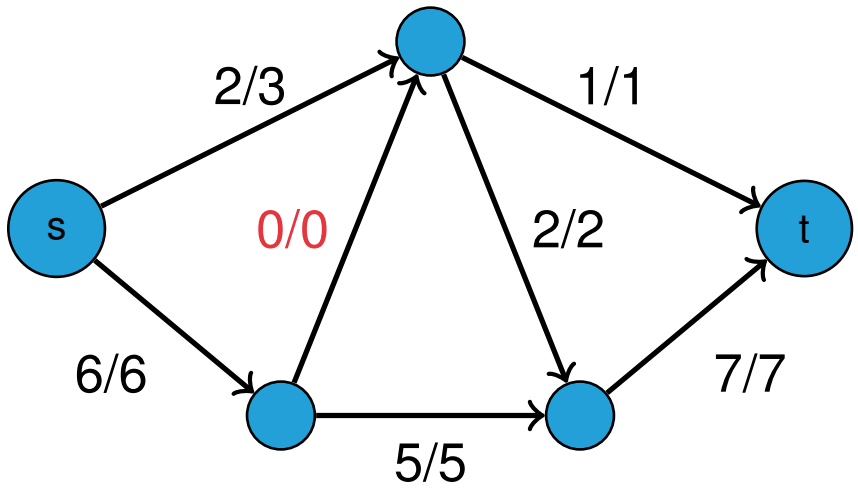



# Uppgift 2: Förändrat flöde

## Vad händer om kapaciteten minskar med en enhet?

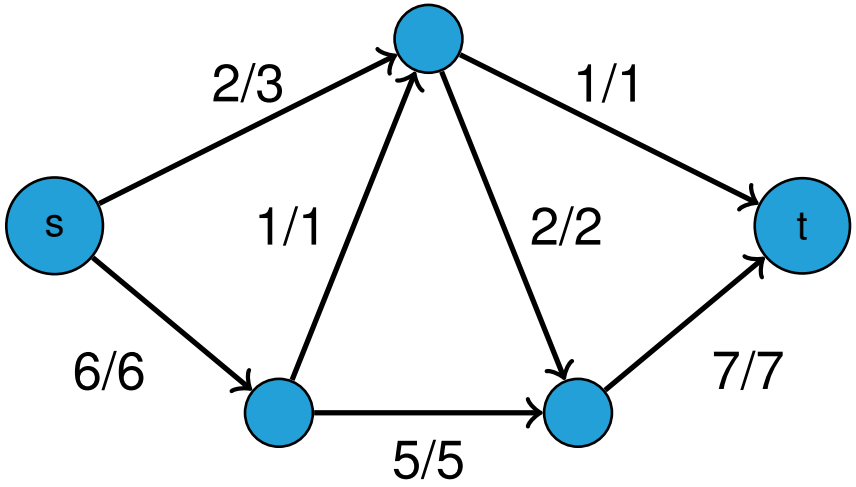
Om kapaciteten mellan $u$ och $v$ minskar så finns det tre fall:

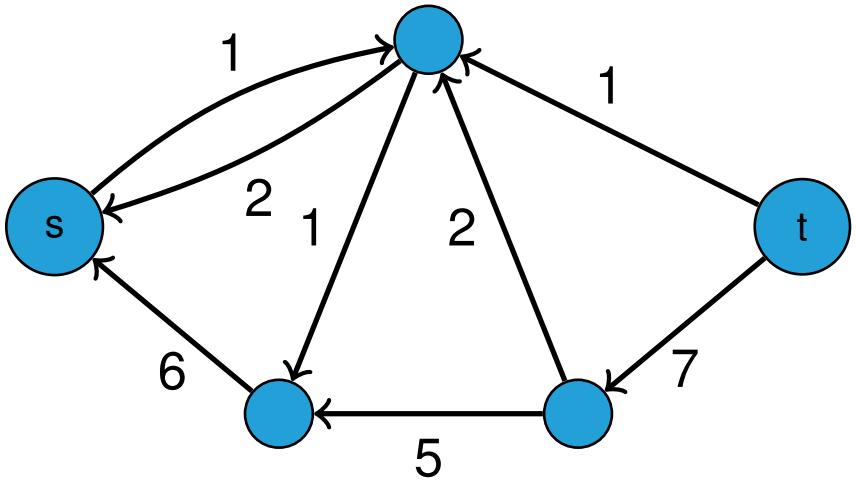
- Om kapaciteten ej var fullt utnyttjad, så händer ingenting.
- Annars så skapas en obalans när vi minskar kapaciteten: det går in en enhet för mycket i $u$ och det går ut en enhet för mycket i $v$.
  - Försök leda om en enhet flöde från $u$ till $v$ på något annat sätt. Leta en ny väg från $u$ till $v$ i restflödesgrafen.
  - Om detta ej går så måste flödet minska.



# Uppgift 2: Förändrat flöde

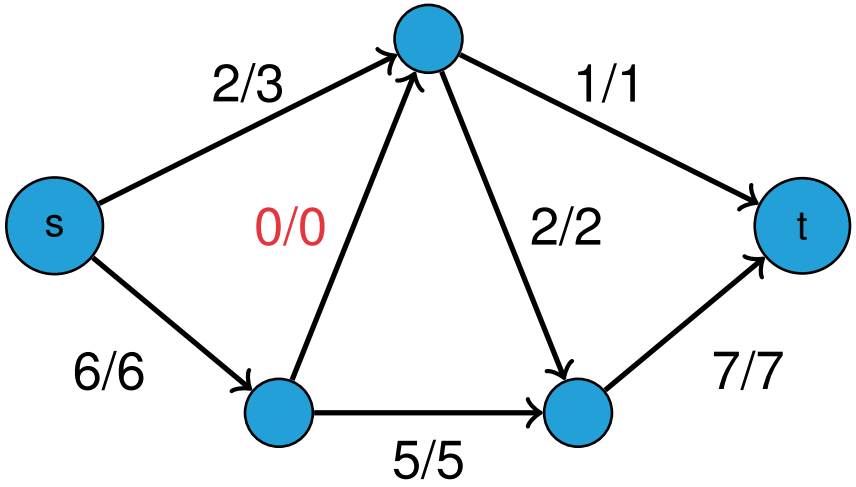
## Vad händer om kapaciteten minskar med en enhet?

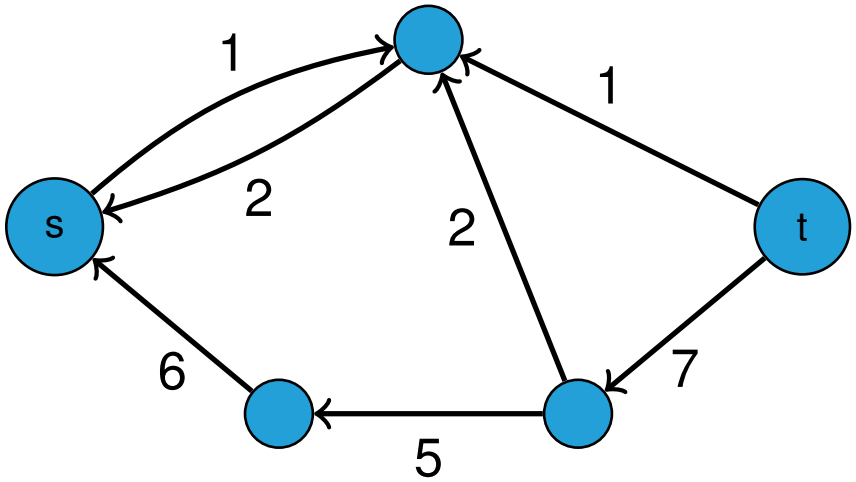


# Uppgift 2: Förändrat flöde

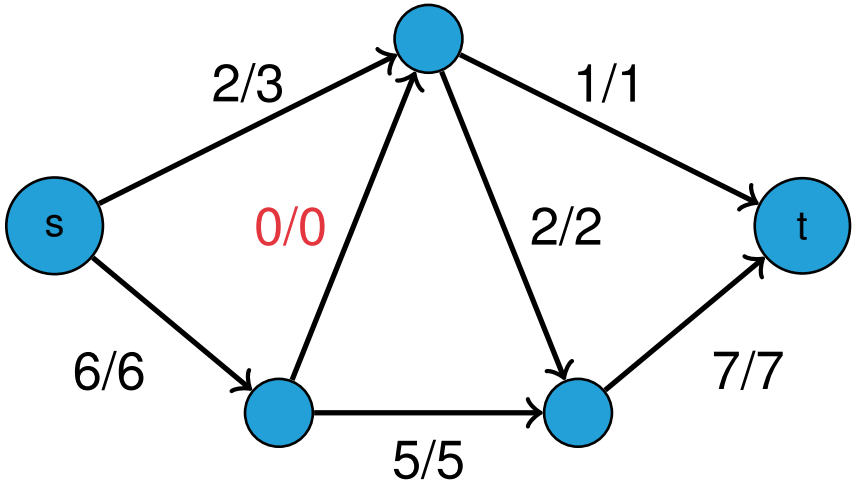
## Vad händer om kapaciteten minskar med en enhet?

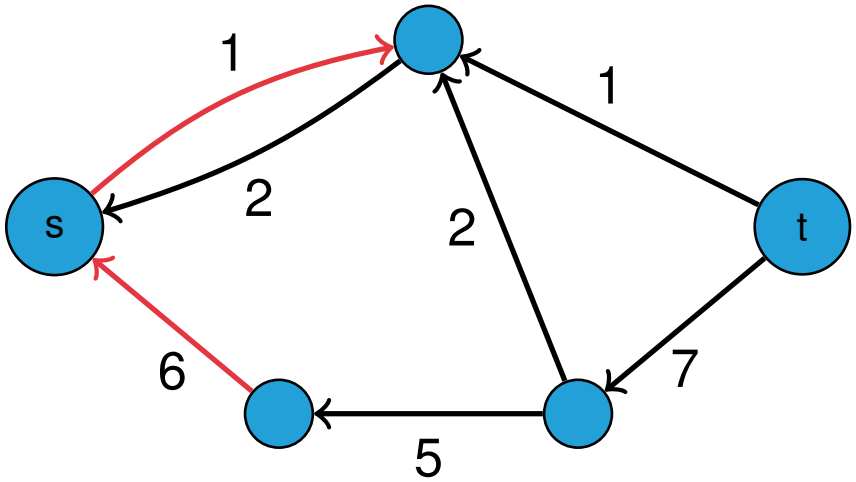


# Uppgift 2: Förändrat flöde

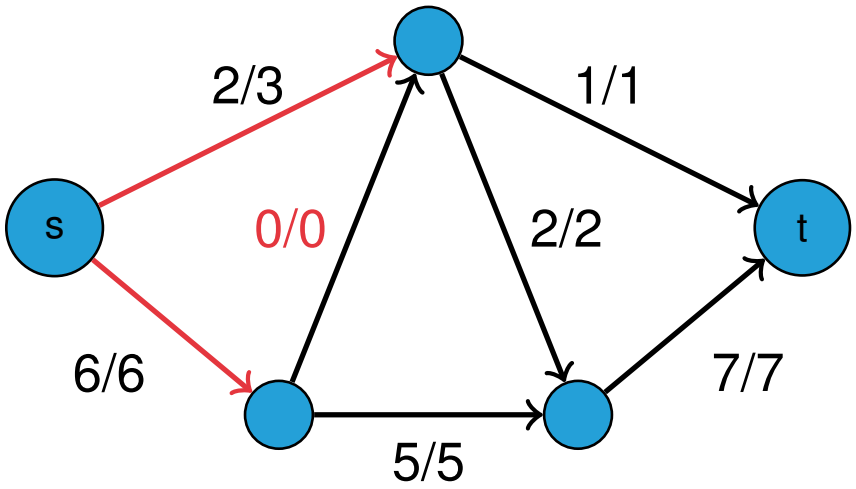
## Vad händer om kapaciteten minskar med en enhet?

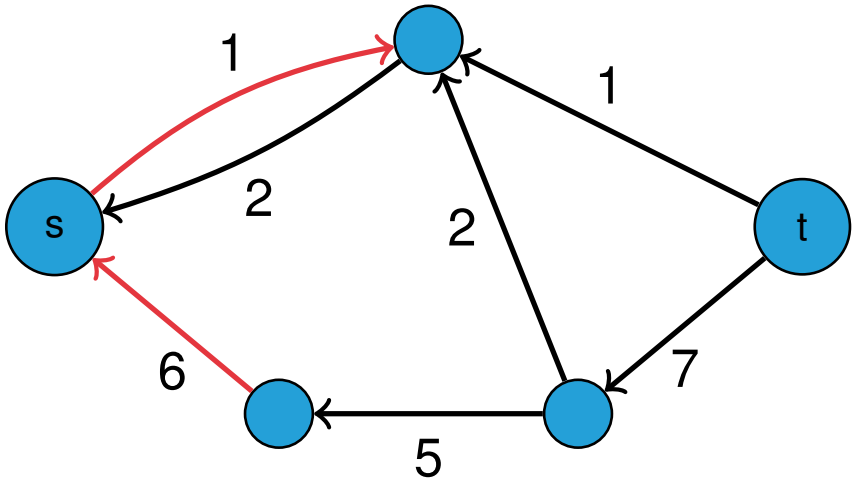


# Uppgift 2: Förändrat flöde

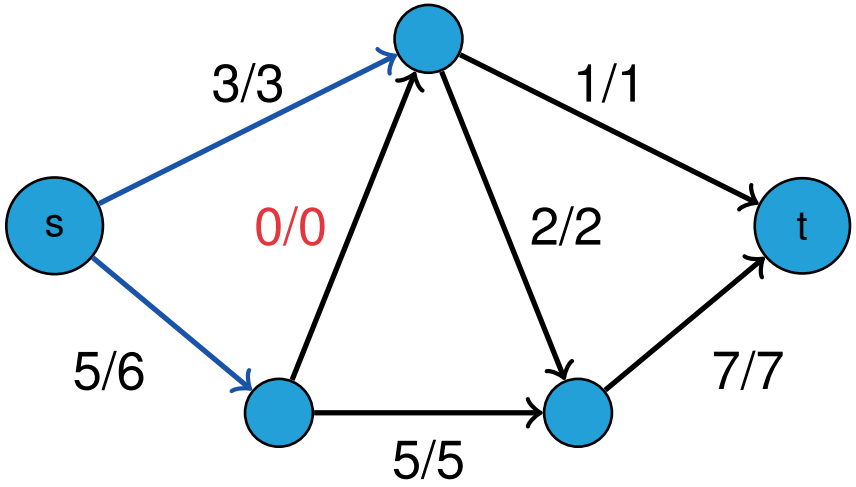
## Vad händer om kapaciteten minskar med en enhet?

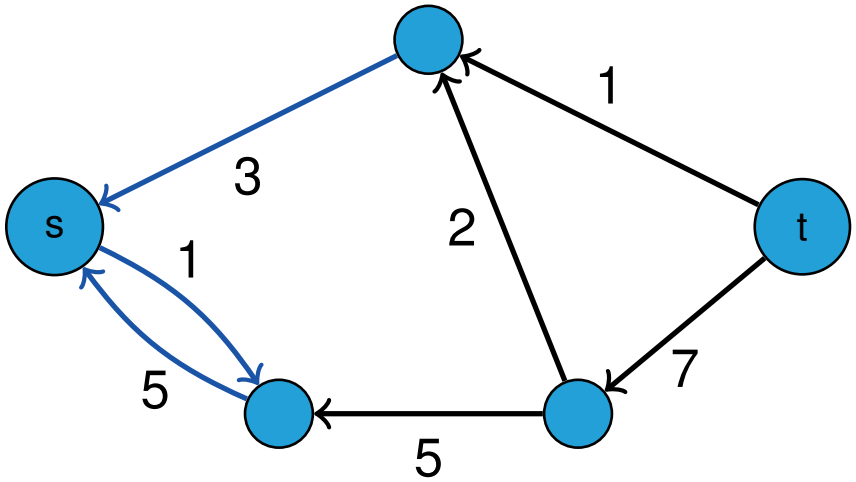


# Uppgift 2: Förändrat flöde

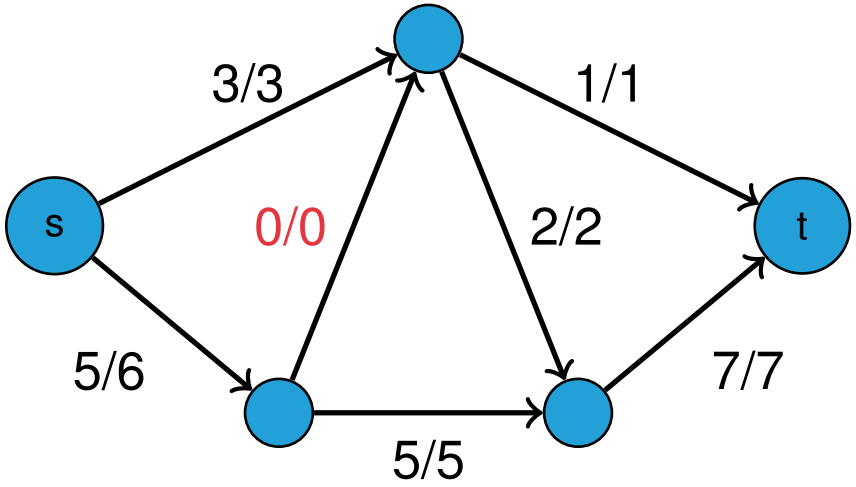
## Vad händer om kapaciteten minskar med en enhet?

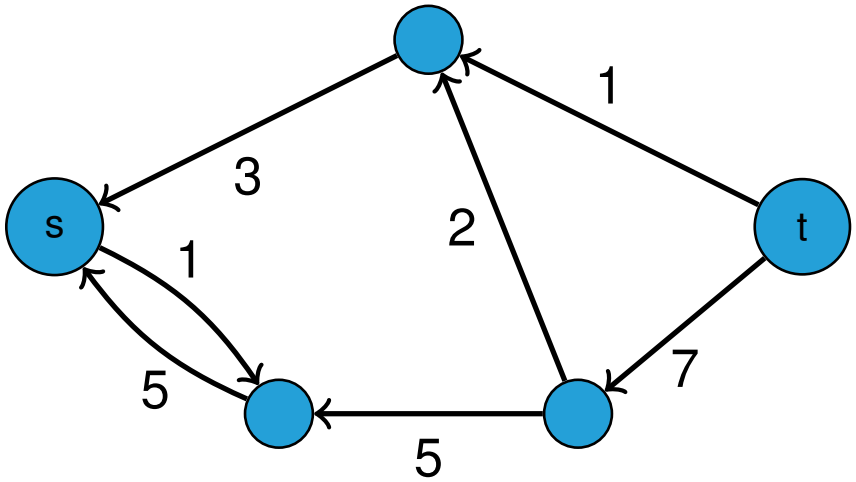


# Uppgift 2: Förändrat flöde

## Vad händer om kapaciteten minskar med en enhet?



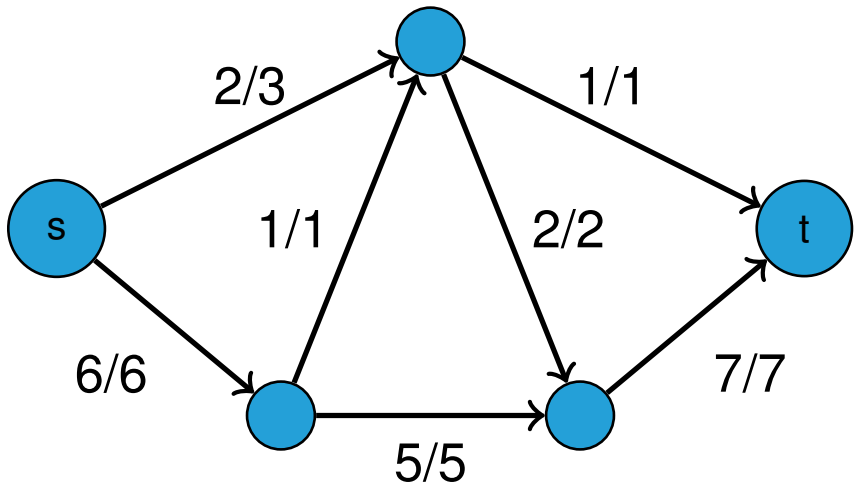



# Uppgift 2: Förändrat flöde

## Vad händer om kapaciteten minskar med en enhet?

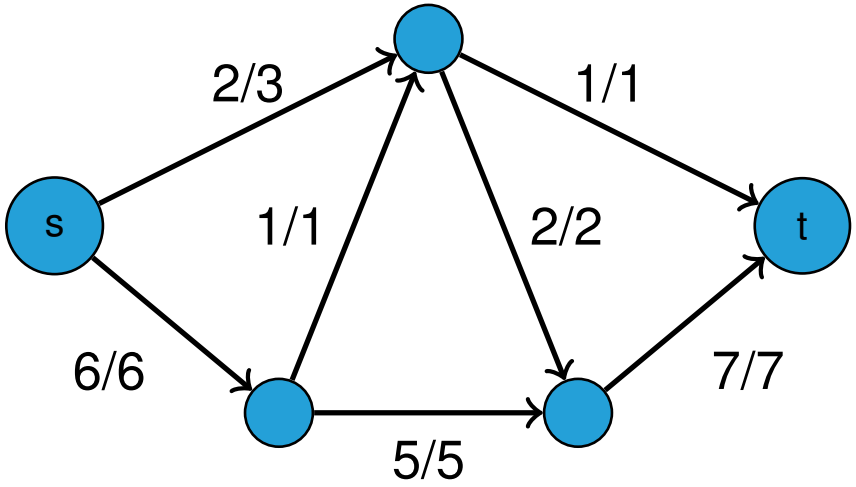
Om kapaciteten mellan $u$ och $v$ minskar så finns det tre fall:

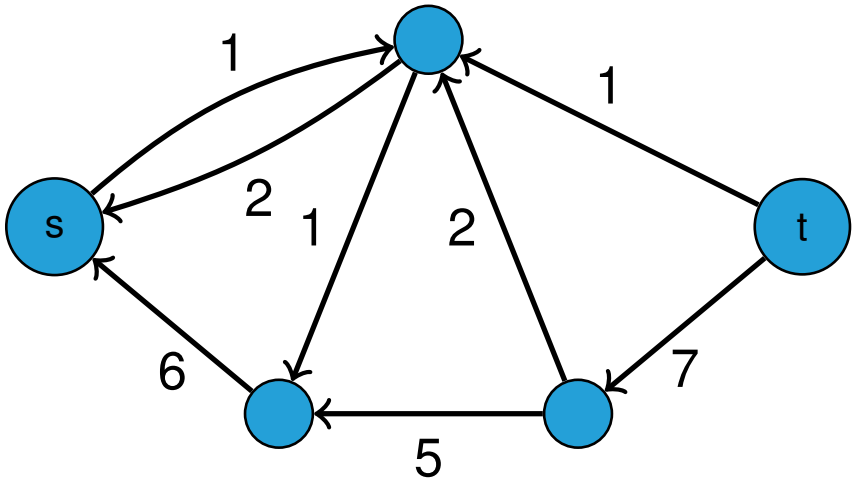
- Om kapaciteten ej var fullt utnyttjad, så händer ingenting.
- Annars så skapas en obalans när vi minskar kapaciteten: det går in en enhet för mycket i $u$ och det går ut en enhet för mycket i $v$.
  - Försök leda om en enhet flöde från $u$ till $v$ på något annat sätt. Leta en ny väg från $u$ till $v$ i restflödesgrafen.
  - Om detta ej går så måste flödet minska.



# Uppgift 2: Förändrat flöde

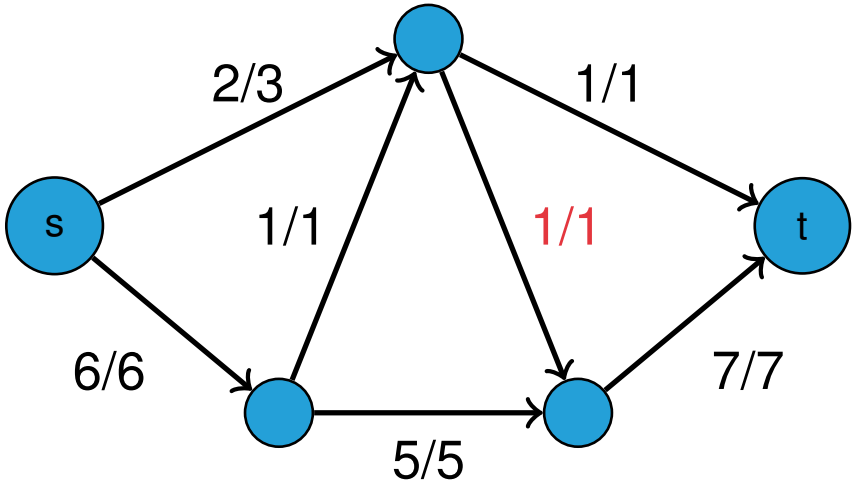
## Vad händer om kapaciteten minskar med en enhet?

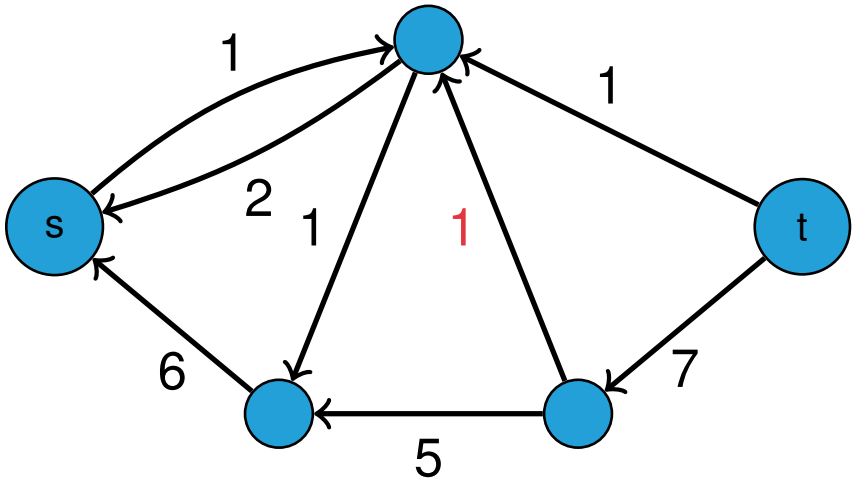


# Uppgift 2: Förändrat flöde

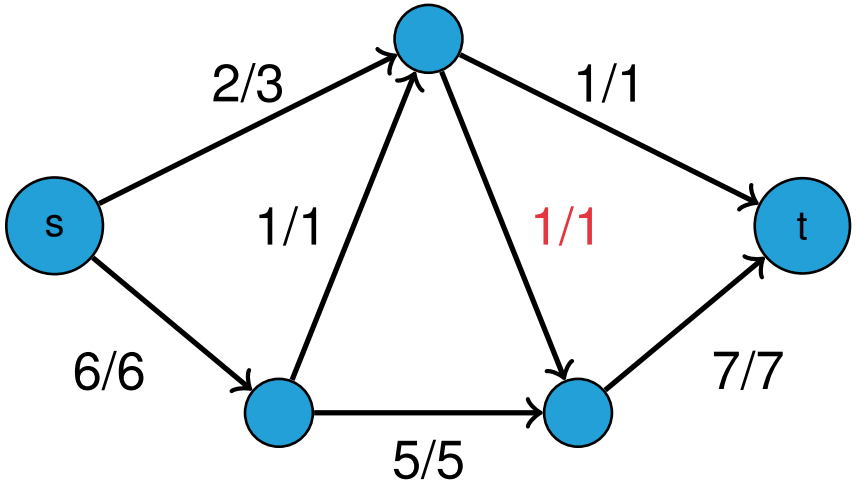
## Vad händer om kapaciteten minskar med en enhet?

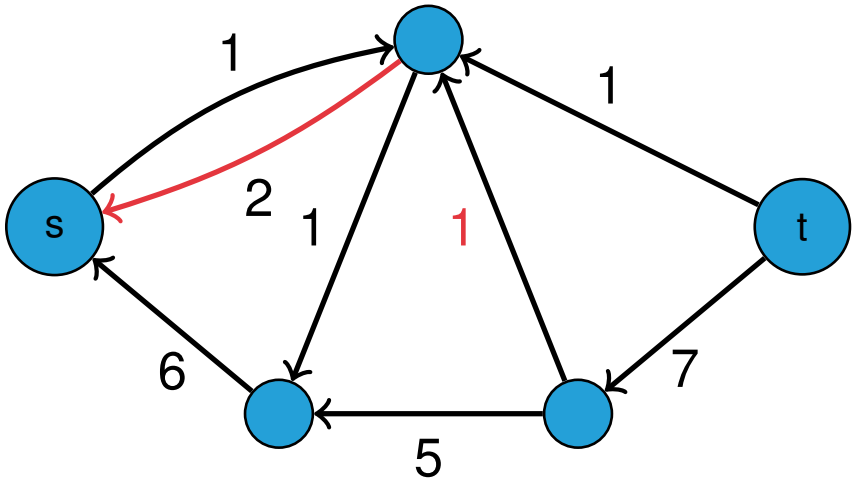


# Uppgift 2: Förändrat flöde

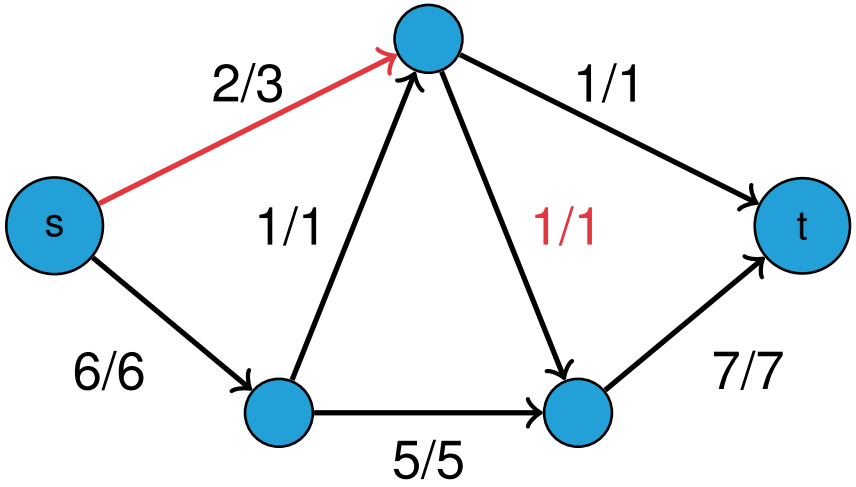
## Vad händer om kapaciteten minskar med en enhet?

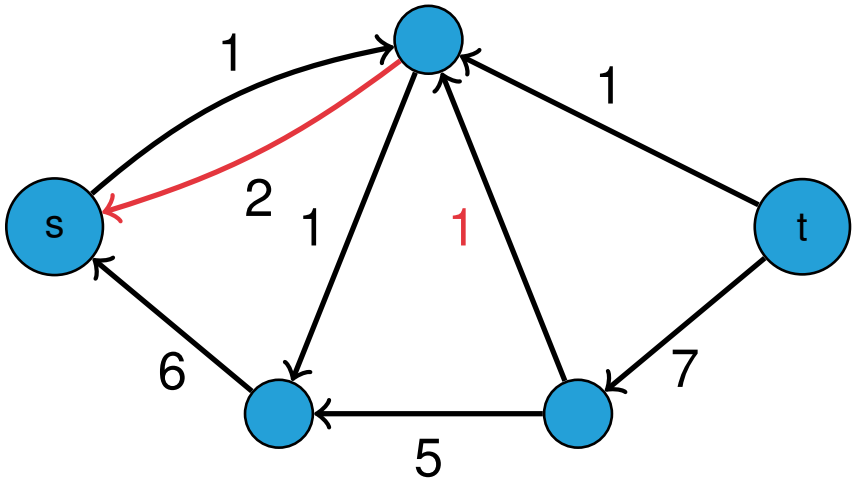


# Uppgift 2: Förändrat flöde

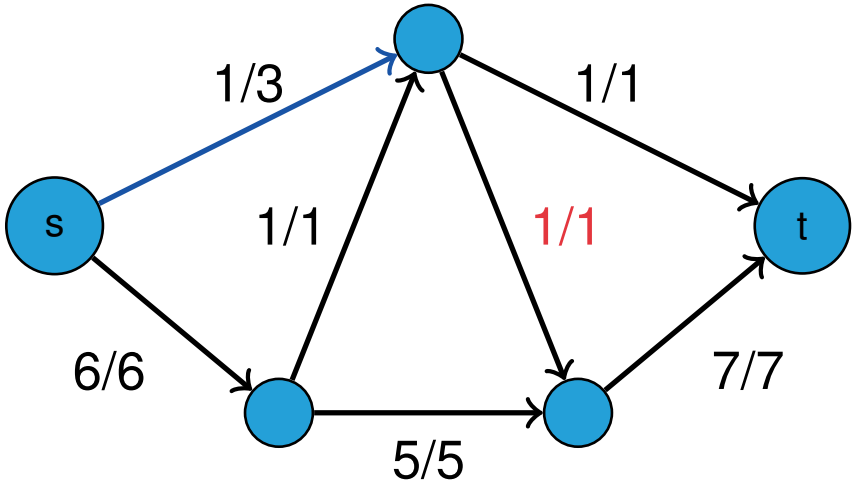
## Vad händer om kapaciteten minskar med en enhet?

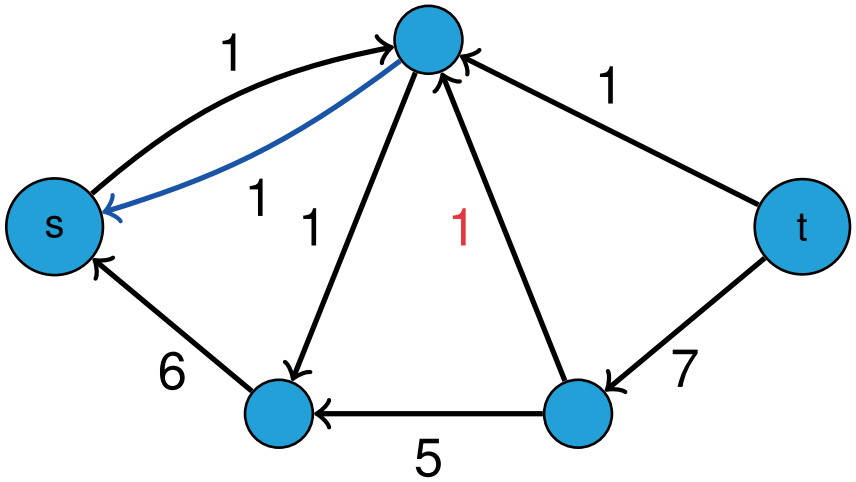


# Uppgift 2: Förändrat flöde

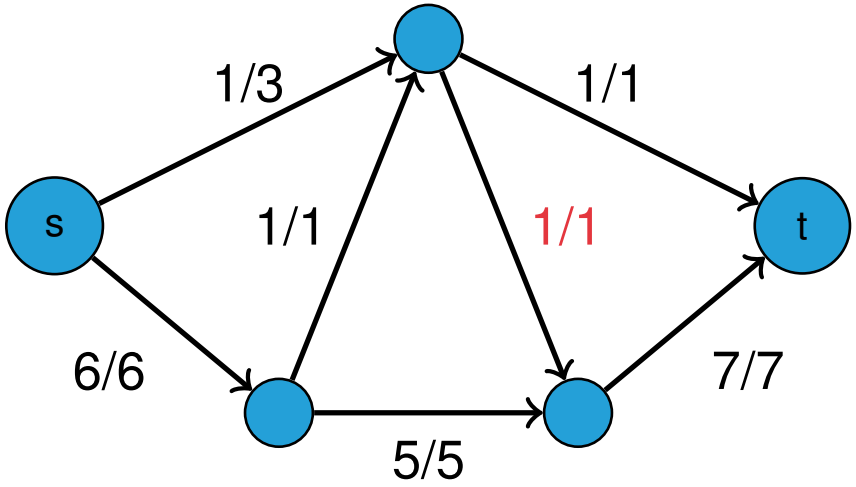
## Vad händer om kapaciteten minskar med en enhet?

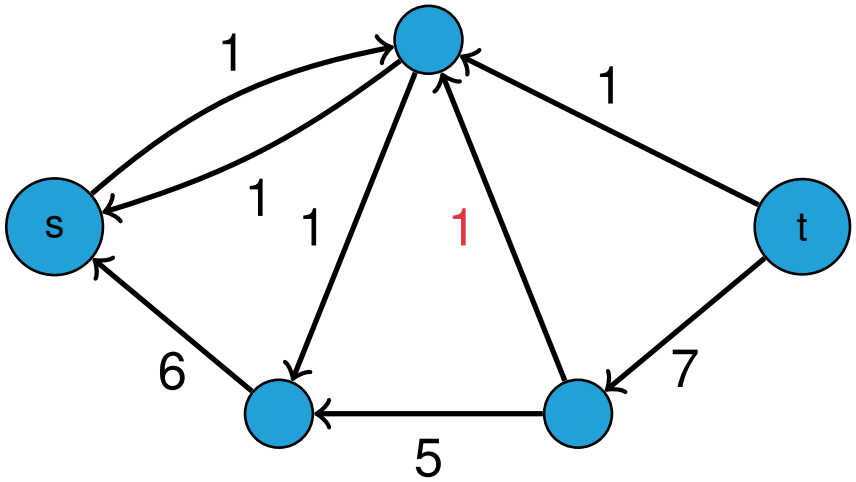


# Uppgift 2: Förändrat flöde

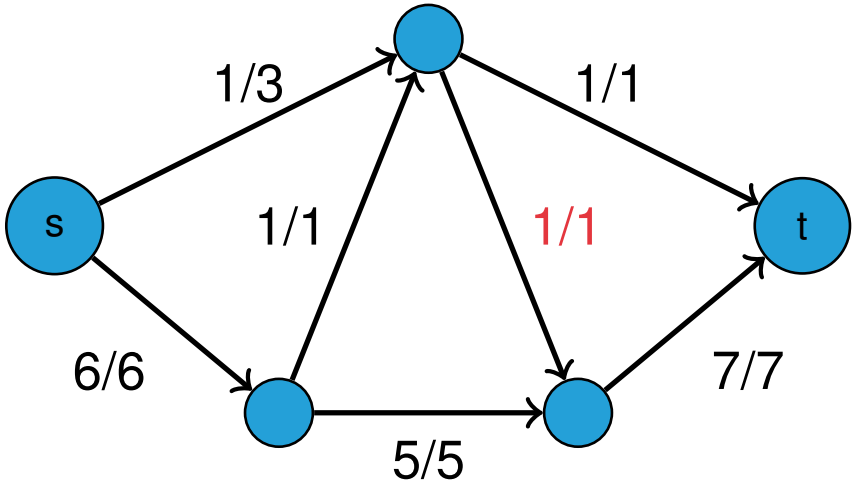
## Vad händer om kapaciteten minskar med en enhet?

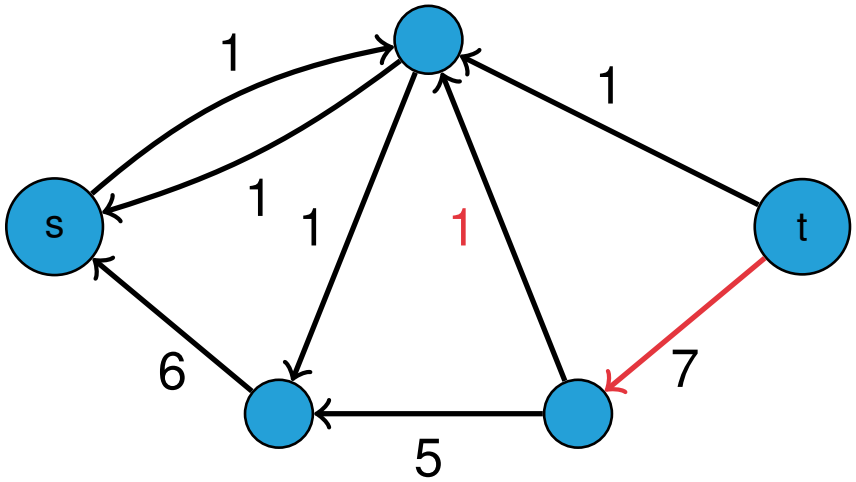


# Uppgift 2: Förändrat flöde

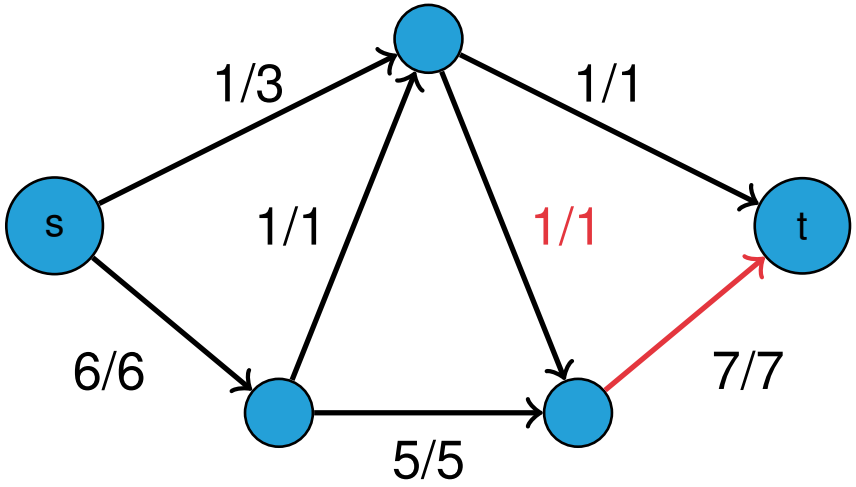
## Vad händer om kapaciteten minskar med en enhet?

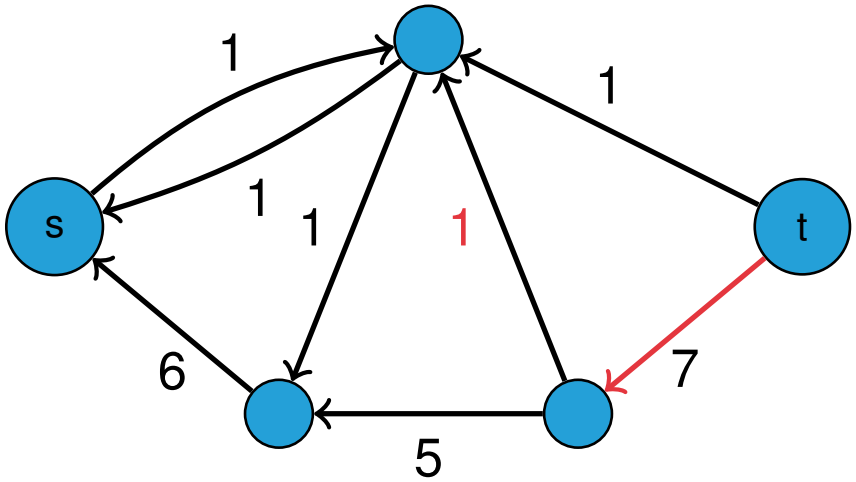


# Uppgift 2: Förändrat flöde

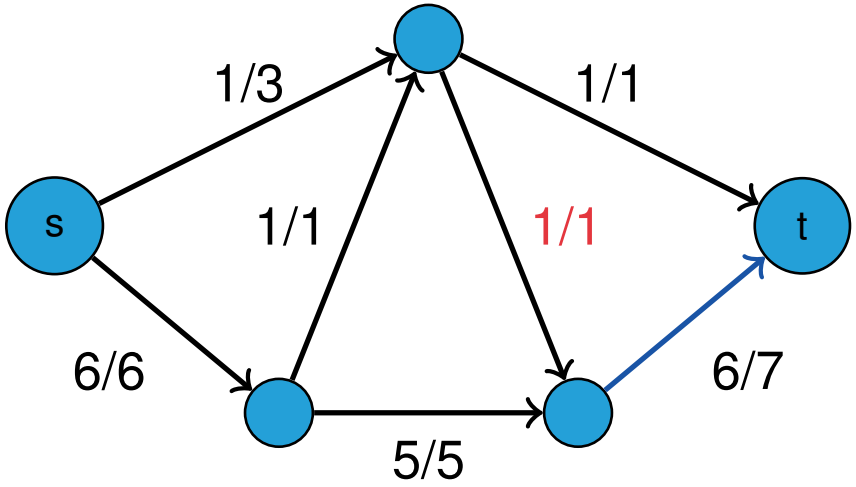
## Vad händer om kapaciteten minskar med en enhet?

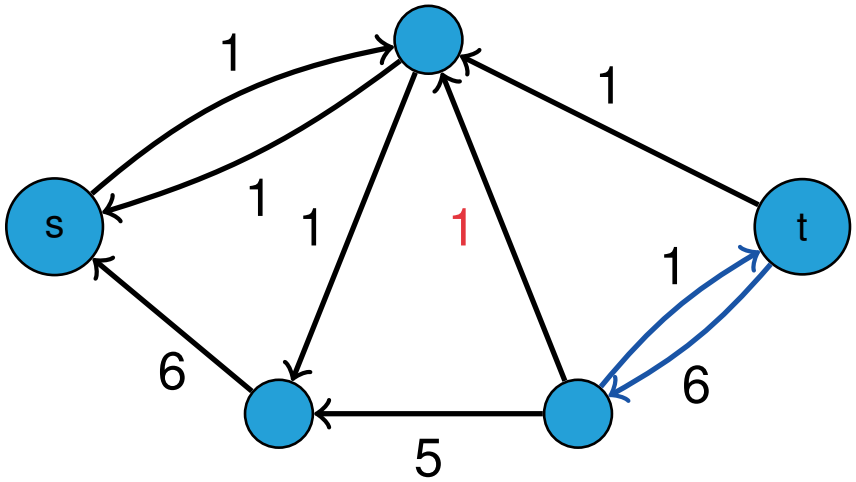


# Uppgift 2: Förändrat flöde

## Vad händer om kapaciteten minskar med en enhet?



# Uppgift 3: Eulercykel

## Definition

En Eulercykel är en sluten stig som passerar varje kan exakt en gång.

## Sats

Alla hörn i $G$ har jämnt gradtal. $\iff$ Det existerar en Eulercykel i $G$.

Uppgift:

- Konstruera en algoritm som givet en graf där alla hörn har jämnt gradtal hittar en Eulercykel.
- Algoritmen ska ha tidskomplexitet $\mathcal{O}(n)$.

# Uppgift 3: Eulercykel

- Idé: Börja i ett hörn $u$, traversera grafen (på djupet), fortsätt tills du stöter på $u$ igen.
- Betrakta sökstigen $P$ du gått längs. Det finns två fall:
  - Antingen består $P$ av alla kanter. I så fall är vi klara.
  - Annars, ta bort $P$ från grafen. Det återstår nu en eller flera sammanhängande komponenter av grafen.
  - Alla hörn har fortfarande jämnt gradtal, så det måste finnas en Eulercykel i varje komponent.
  - Hitta nu rekursivt en Eulercykel i varje komponent, och sammanfoga dessa med $P$.
- Hur gör vi detta på linjär tid?

# Uppgift 3: Eulercykel

- Vi definierar två stycken “spelare”: P1 och P2. Båda startar i $v$.
- P1 har som jobb att skapa stigen $P$, märka varje kant som passeras, och fortsätta tills hen kommer tillbaka till $v$.
- P2:s jobb är att följa i P1:s spår. Kontrollerar om alla kanter från hörnen är märkta.
- Om det finns omärkta kanter från ett hörn $u$ så finns det en outforskad del av grafen där.
  - P2 ropar då på P1, som på samma sätt får hitta en ny stig som börjar och slutar i $u$.
  - P2 går denna nya stig innan hen fortsätter från $u$ längs $P$.
- Till slut kommer P2 ha besökt alla kanter exakt en gång! Vägen P2 har gått är alltså en Eulercykel.
- Varken P1 eller P2 kommer ha gått längs samma kant mer än en gång $\implies$ tidskomplexitet $\mathcal{O}(n)$.

# Uppgift 3: Eulercykel

```
EulerCycle(G) =
    cycle ← {1}
    choose edge (1, t)
    mark (1, t)
    path ← PathFinder(G, 1, t)
    Straggler(path)
    return cycle

PathFinder(G, start, cur) =
    append(cur, path)
    while cur ≠ start do
        choose unmarked edge (cur, v)
        mark (cur, v)
        append(v, path)
        cur ← v
    return path
```

```
Straggler(path) =
    while path ≠ ∅ do
        u ← next(path)
        append(u, cycle)
        for all edges (u, v) do
            if unmarked (u, v) then
                mark (u, v)
                p ← PathFinder(G, u, v)
                Straggler(p)
```

# Uppgift 4: Julklappsfördelning

- En pappa ska ge sina $n$ barn varsin julklapp.
- Varje barn har skrivit en önskelista.
- Pappan vill ge varje barn en julklapp från barnets önskelista.
- Han vill dock inte ge samma julklapp till flera barn.

Uppgift:

- Konstruera och analysera en effektiv algoritm som löser detta problem.
- Antag att det finns högst $m$ saker på varje önskelista.

# Uppgift 4: Julklappsfördelning

Bipartit matchning. Kan lösas med hjälp av en algoritm för maximalt flöde.
Tidskomplexitet:

- $\mathcal{O}(nm)$ kanter, $\mathcal{O}(n+m)$ hörn, så varje sökning tar $\mathcal{O}(nm)$ tid.
- Matchningen är av storlek högst $n$.
- Varje lyckad sökning förbättrar får matchning med $1$.
- $\implies$ högst $n$ sökningar.

Den totala tidskomplexiteten blir $\mathcal{O}(n^2m)$.

# Tidskomplexitet

- Säg att vi analyserar en algoritm. Låt $T(n)$ vara tiden det tar för den att lösa ett problem av storlek $n$.
- Om vi gör en noggrann analys av algoritmen kanske vi kan visa att $T(n) \in \Theta(n \log n)$.
- Ofta så är det dock svårt att göra en så noggrann analys att vi kan använda $\Theta$. Istället kan vi göra uppskattningar för att hitta en övre eller undre gräns för tidskomplexiteten.
- Det lär t.ex. vara betydligt enklare att visa $T(n) \in \mathcal{O}(n^2)$ och $T(n) \in \Omega(n)$.
- I praktiken bryr vi oss ofta mest om att vi har en tillräckligt låg övre gräns för algoritmens komplexitet.

# Tidskomplexitet i värsta fallet

- Men tidskomplexiteten för en algoritm kan bero av mer än bara indatastorleken $n$.
- Den kan också bero på hur indatat ser ut.
- Hittils i kursen har vi oftast analyserat algoritmer för värsta möjliga indata, dvs för värsta fallet.
- Notera att detta inte har något att göra med huruvida vi använder $\mathcal{O}$, $\Omega$ eller $\Theta$.
- Det kan t.ex. både vara intressant att hitta en undre gräns och en övre gräns för tidskomplexiteten i värsta fallet för en algoritm.

# Undre gränser för problem

- Man kan också visa undre gränser för problem.
- Att ett problem har den undre gränsen $\Omega(f(n))$ betyder att varje tänkbar algoritm som löser problemet måste ha en tidskomplexitet som ligger i $\Omega(f(n))$.
- Om man har konstruerat en algoritm kan man vilja undersöka om undre och övre gräns sammanfaller, dvs om den undre gränsen för problemet är samma som den övre gränsen för algoritmen.
- Om så är fallet kan man konstatera att algoritmen har optimal tidskomplexitet.

# Uppgift 5: Undre gräns för sökning i sorterad array

- Binärsökning i en sorterad array med $n$ tal tar som bekant tiden $\mathcal{O}(\log n)$.
- Bevisa att $\Omega(\log n)$ också är en undre gräns för antalet jämförelser som krävs för detta problem (i värsta fallet).

Vi vill alltså visa att oavsett vilken algoritm vi har (så länge den bygger på att jämföra elementen) så kan den aldrig vara snabbare än logaritmsk.

# Uppgift 5: Undre gräns för sökning i sorterad array

- Som indata har vi en sorterad array $A$ av storlek $n$ med element $a_1, a_2, \ldots, a_n$, samt ett tal $x$ som är det vi söker efter.
- Låt oss bygga ett beslutsträd för en godtycklig jämförelsebaserad sökningsalgoritm.
- Algoritmen måste börja med att välja ett tal $a_i$ att jämföra $x$ med.
- Om $x = a_i$ är vi klara. Annars har vi två fall: $x < a_i$ och $x > a_i$.
- Beroende på resultatet tar algoritmen beslut om att antingen fortsätta med ett tal $a_j$ eller ett tal $a_k$, osv...
- Hur många noder finns det i trädet? $n$ stycken.

**Notera!**

Trädets utseende/struktur beror endast på algoritmen, ej på indata. Indata avgör vilken väg i trädet vi tar.

# Uppgift 5: Undre gräns för sökning i sorterad array

## Exempel

Hur ser beslutsträdet ut för linjärsökning?

- Vi vill hitta en undre gräns för problemet sökning i sorterad array i värsta fallet.
- Dvs en undre gräns som varje tänkbar algoritm uppfyller (för sitt värsta fall).
- Eftersom vi söker en undre gräns så betraktar vi den bästa algoritmen som teoretiskt sett skulle kunna finnas.
  - (Om denna uppfyller den undre gränsen så måste ju alla andra algoritmer också göra det.)
- Vi vill alltså analysera det beslutsträd där den längsta vägen (motsvarande värsta indata) är så kort som möjligt.

# Uppgift 5: Undre gräns för sökning i sorterad array

- Vi vill alltså analysera det beslutsträd där den längsta vägen (motsvarande värsta indata) är så kort som möjligt.
- Detta måste alltså vara ett balanserat binärträd!
  - Kom ihåg: i ett balanserat binärträd så är alla lövnoder på samma djup (så när som på en skillnad $1$, ifall antalet noder inte är en tvåpotens).
- Antal noder är $n$. Längsta vägen $=$ höjden på trädet $\approx \log n$.
- Antalet jämförelser i värsta fallet måste alltså vara $\approx \log n$.

Den bästa tänkbara algoritmen för problemet har tidskomplexiteten $\Theta(\log n)$ i värsta fallet.
Således måste $\Omega(\log n)$ vara en undre gräns för alla algoritmer som löser problemet.

# Nästa vecka

- Blandat på temat algoritmkonstruktion
- Inlämning och redovisning av labbteori 3.