CPSC 535: Advanced Algorithms

Instructor: Dr. Doina Bein

Max Flow

(some slides taken from http://people.cs.aau.dk/~simas/aalg04/ , https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/07-max-flow.pdf , and https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/08-max-flow-formulations-and-matching.pdf)

Breakthrough results:

Solving max flow in almost linear time (June 8, 2022):

https://www.quantamagazine.org/researchers-achieve-absurdly-fast-algorithm-for-network-flow-20220608/?fbclid=IwAR0AVavBo0vDc0O4uOl3zJ6cquk-TeCmkbFEcGlE-AC5O2TYb9RIQxti1So

Maximum flow

(slides taken from http://people.cs.aau.dk/~simas/aalg04/)

- Main goals of the lecture:
  - *to understand how flow networks and maximum flow problem can be **formalized**;*
  - *to understand the **Ford-Fulkerson** method and to be able to prove that it works correctly;*
  - *to understand the **Edmonds-Karp** algorithm and the intuition behind the analysis of its worst-case running time.*
  - *to be able to apply the Ford-Fulkerson method to solve the **maximum-bipartite-matching** problem.*

Chapter 26 of the textbook CLRS, pages 708-736

Flow networks

- *What if weights in a graph are maximum capacities of some flow of material*?
  - Pipe network to transport fluid (e.g., water, oil)
    - Edges – pipes, vertices – junctions of pipes
  - Data communication network
    - Edges – network connections of different capacity, vertices – routers (do not produce or consume data just move it)
  - Concepts (informally):
    - ***Source*** vertex *s* (where material is produced)
    - ***Sink*** vertex *t* (where material is consumed)
    - For all other vertices – what goes in must go out
    - *Goal: maximum rate of material flow from source to sink*

Formalization

- *How do we formalize flows*?
  - Graph $G=(V,E)$ – a **flow network**
    - Directed, each edge has **capacity** $c(u,v) \geq 0$
    - Two special vertices: ***source*** s, and ***sink*** t
    - For any other vertex v, there is a path $s \rightarrow \ldots \rightarrow v \rightarrow \ldots \rightarrow t$

Def. of a flow network

(taken from https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/07-max-flow.pdf)

flow network: graph representing resource flows

- directed graph $G = (V,E)$
- designated source vertex $s \in V$ and sink vertex $t \in V$
- no self-loop: $\forall v \in V, (v,v) \notin E$
- flow is possible through every vertex: $\forall v \in V$, there exists some path $s \rightsquigarrow v \rightsquigarrow t$
- capacity: $\forall (u, v) \in E$, there is a defined, non-negative real capacity $c(u,v)$
- implies: G is connected and $|E| \geq |V| - 1$.

  In most flow networks, there are no antiparallel edges: for any $\forall (u,v) \in E, (v,u) \notin E$.

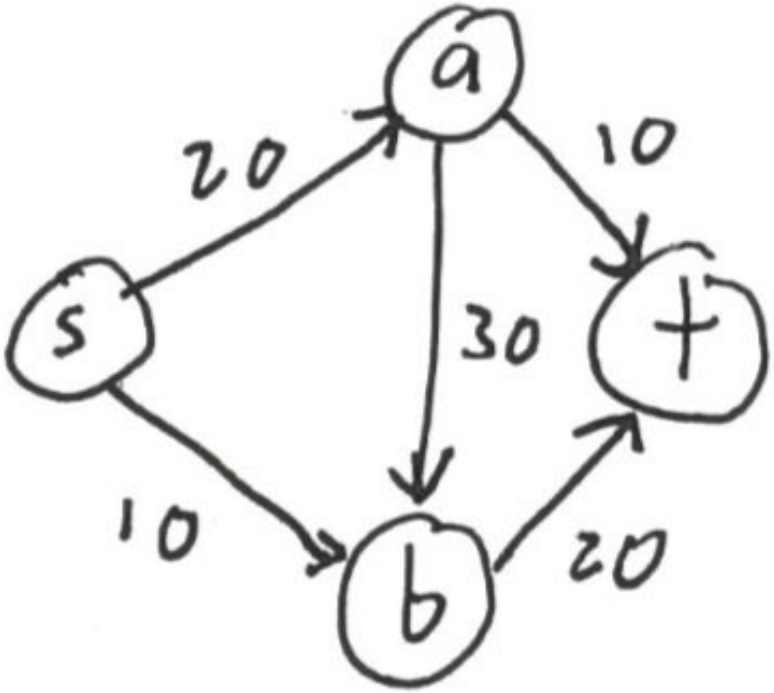
A flow

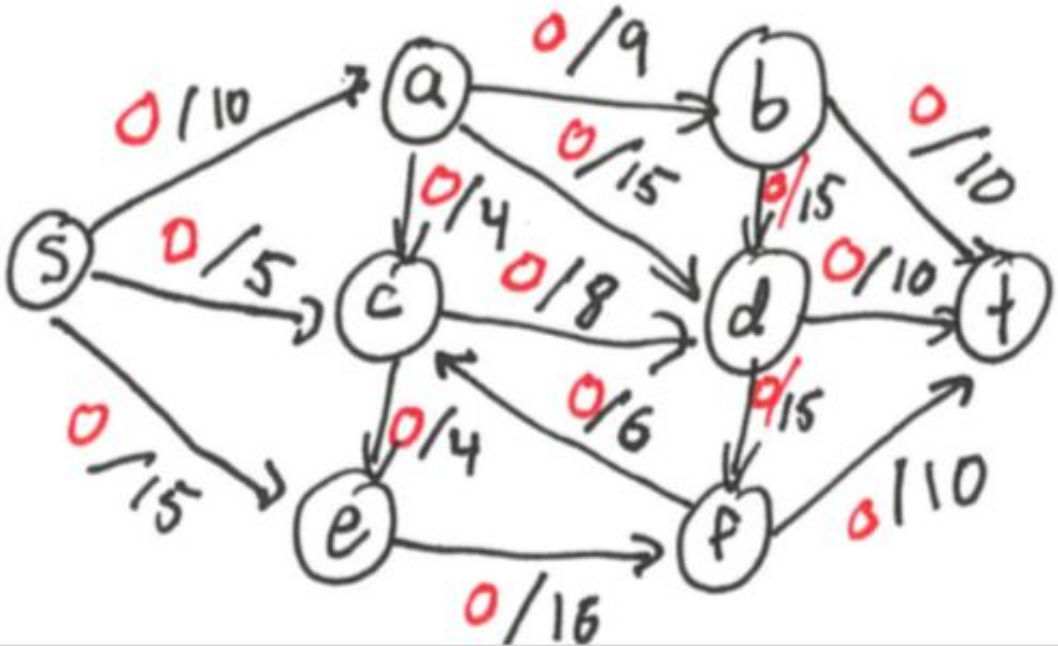
- **Flow** – a function $f : V \times V \rightarrow \boldsymbol{R}$
  - *Capacity constraint*: For all $u, v \in V$: $f(u,v) \leq c(u,v)$
  - *Skew symmetry*: For all $u, v \in V$: $f(u,v) = -f(v,u)$
- *Flow conservation:* (flow-in) = (flow-out), except for source and sink; formally for all $u \in V - \{s, t\}$:

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

- Value |f| = net flow out of source = net flow into sink

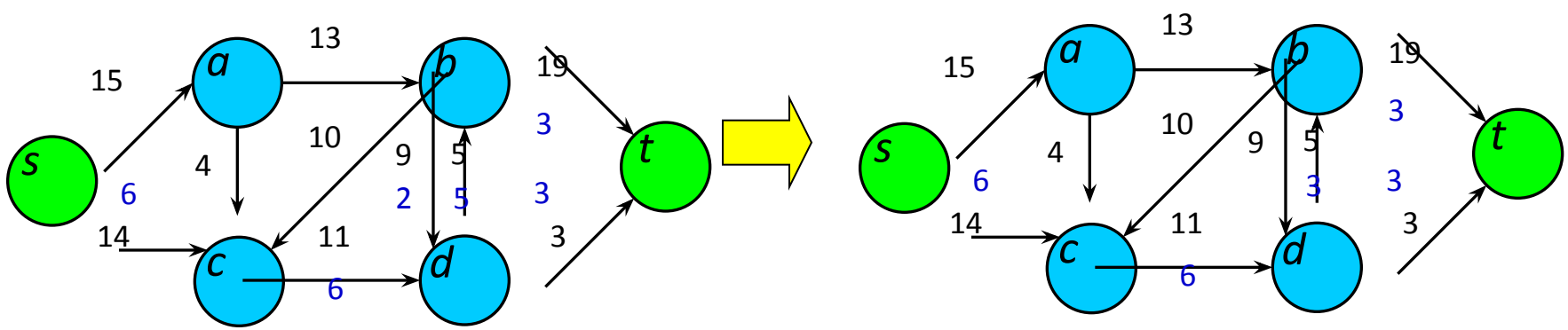
$$|f| = \sum_{v \in V} f(s,v) = \sum_{v \in V} f(u,t)$$


a
s
t
b
20
10
30
10
20


a
b
s
c
d
t
e
f
0/10
0/9
0/15
0/10
0/4
0/15
0/5
0/8
0/10
0/6
0/15
0/15
0/4
0/10
0/16

Cancellation of flows

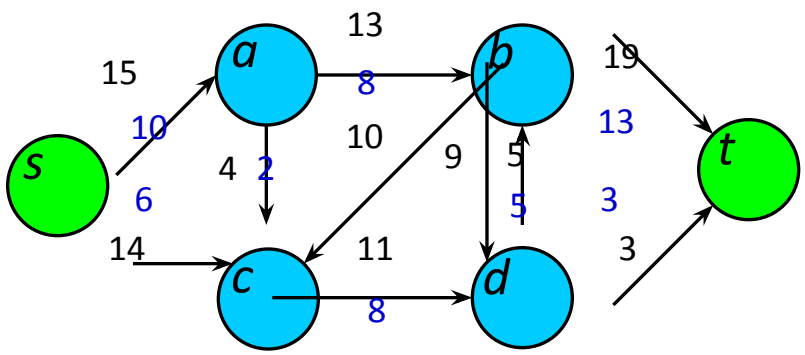
- *Do we want to have positive flows going in both directions between two vertices?*
  - No! such flows *cancel* (maybe partially) each other
  - Skew symmetry – notational convenience



Maximum flow

- *What do we want to maximize*?
  - ***Value*** of the flow *f*:

$$|f| = \sum_{v \in V} f(s,v) = f(s,V) = f(V,t)$$



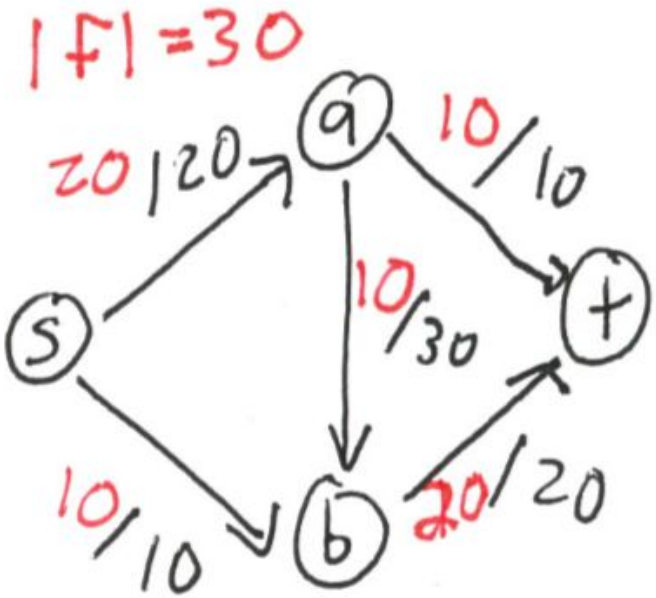
We want to find a flow of maximum value |f|!

Maximum Flow Problem Definition

Maximum flow problem

input: a flow network G

output: a flow f of maximum value |f |



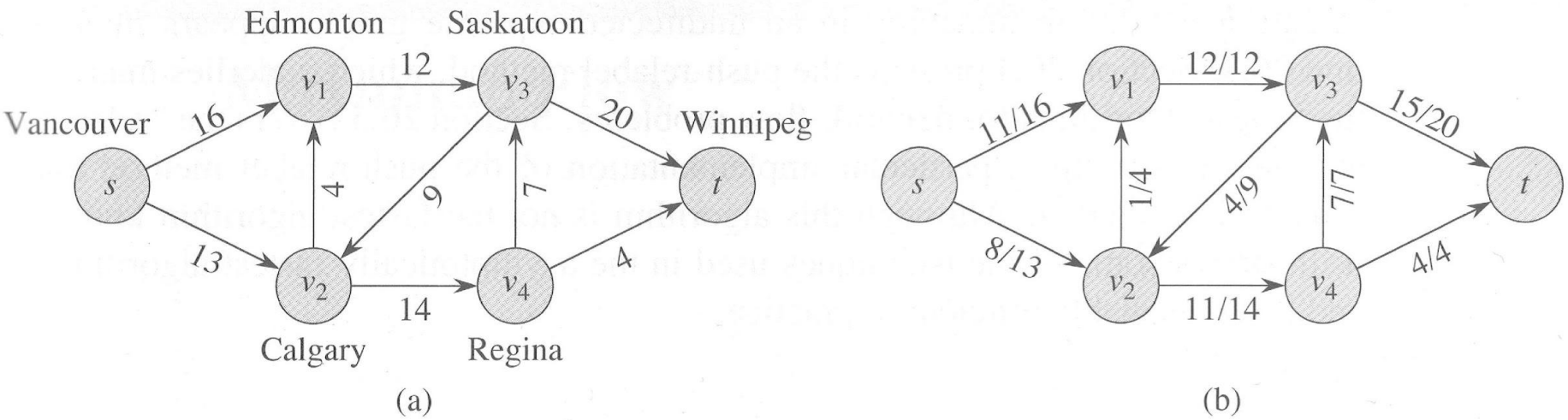
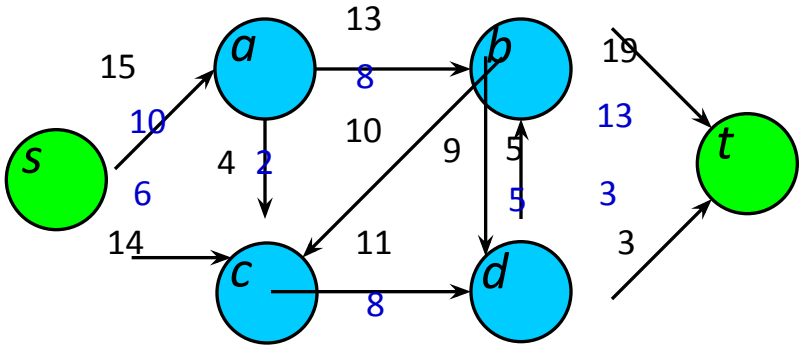


Figure 26.1 **(a)** A flow network $G = (V, E)$ for the Lucky Puck Company's trucking problem. The Vancouver factory is the source s, and the Winnipeg warehouse is the sink t. The company ships pucks through intermediate cities, but only $c(u, v)$ crates per day can go from city u to city v. Each edge is labeled with its capacity. **(b)** A flow f in G with value $|f| = 19$. Each edge (u, v) is labeled by $f(u, v)/c(u, v)$. The slash notation merely separates the flow and capacity; it does not indicate division.

Vancouver is the source node (s)
Winnipeg is the sink node (t)
It can ship at most c(u,v) crates per day between cities u and v (weights in Fig. 26.1(a))
What is the largest number p of crates per day they can ship out of Vancouver?

Augmenting path

- *Idea for the algorithm*:
  - If we have some flow,…
  - …and can find a path *p* from *s* to *t* (***augmenting path***), such that there is $a > 0$, and for each edge (u,v) in *p* we can add *a* units of flow: $f(u,v) + a \leq c(u,v)$
  - Then just do it, to get a better flow!
  - *Augmenting path in this graph*?



Ford-Fulkerson Method

- “method” because this is a pattern for specific max-flow algorithms
- not a complete, clear alg. yet
- based on iterative improvement pattern

```
function ITERATIVE-IMPROVEMENT(input)
   result = base-case result
   while result is not optimal do
      improve result
   end while
   return result
end function
```

The Ford-Fulkerson method

- Sketch of the method:

```
function FORD-FULKERSON-METHOD(G,s,t)
   f = flow with every edge set to zero
   initialize residual network Gf
   while there exists an augmenting path p in Gf do
      augment flow f along path p
   end while
   return f
end function
```

What is a residual network?

What is and how do we find an augmenting path p?

How much additional flow can we send through that path p?

Does the algorithm always terminates & finds the maximum flow?

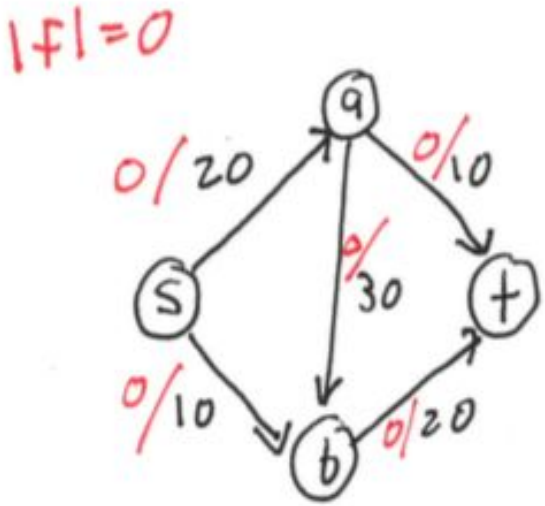
Residual Network

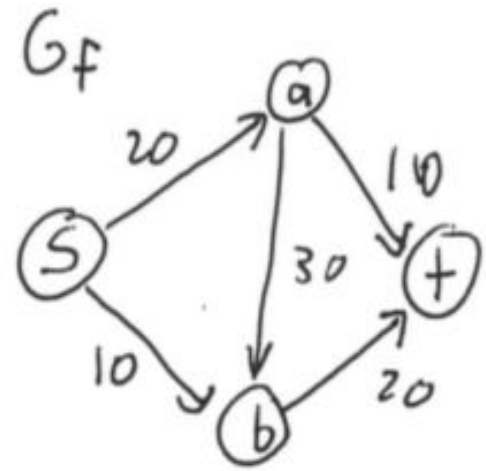
- residual network G_f has same vertices as flow network G = (V, E)
- edges reflect how much capacity is still available
- G_f only contains edges with positive available capacity and added "backwards" edges
- backwards edges allow us to take-back some positive flow
- we define *residual capacity* between vertices u , v ∈ V as

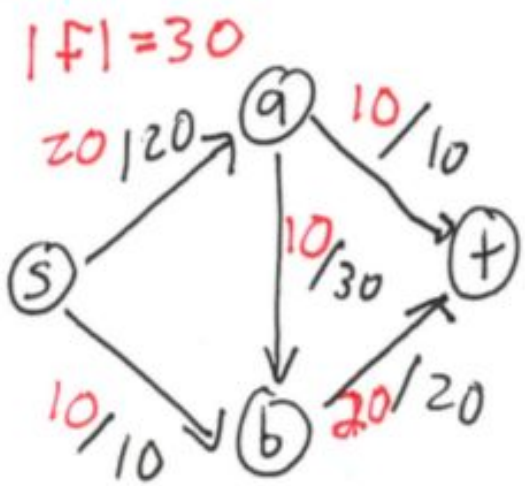
$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & if\ (u,v) \in E \\ f(v,u) & if\ (v,u) \in E \\ 0 & otherwise \end{cases}$$

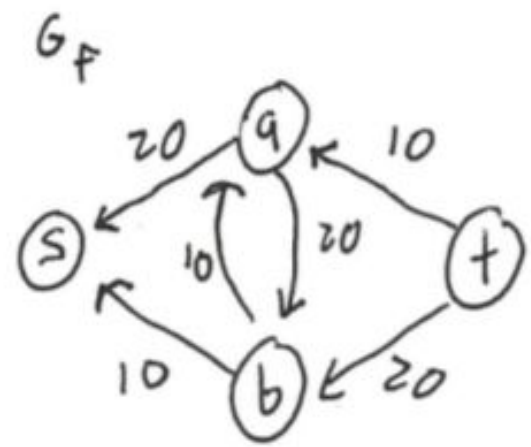
- (recall that in a flow network either (u,v) ∈ E or (v,u) ∈ E but not both)

Residual Network Example









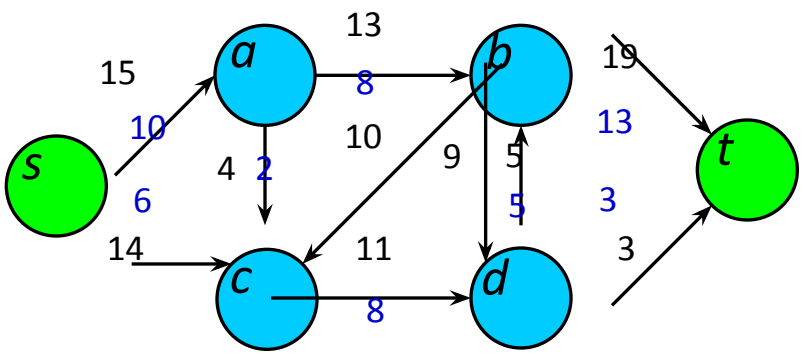
Augmenting Paths

- *augmenting path*: simple path from source s to sink t in residual network c_f (simple ≡ no repeated vertices)
- recall: residual network G_f only contains edges with leftover capacity
- =⇒ if path p exists in G_f, then every edge along p has positive weight in G_f
- =⇒ we can legally increase net $s \rightsquigarrow t$ flow by increasing weights in G_f
- i.e. increasing flow across the forwards edges in G_f, sometimes decreasing flow across the backwards edges
- c_f (p) =residual capacity of p = minimum weight c_f (u,v) of an edge (u,v) in p

Residual network

- *How do we find augmenting path*?
  - It is any path in ***residual network***:
    - Residual capacities: $c_f(u,v) = c(u,v) - f(u,v)$
    - Residual network: $G_f=(V,E_f)$, where $E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}$
    - *What happens when $f(u,v) < 0$ (and $c(u,v) = 0$)*?
    - Observation – edges in E_f are either edges in E or their reversals: $|E_f| \leq 2|E|$

Compute residual network:



Residual capacity of a path

- *How much additional flow can we send through an augmenting path*?
  - *Residual capacity* of a path p in G_f:
    $c_f(p) = \min\{c_f(u,v): (u,v) \text{ is in } p\}$
  - Doing augmentation: for all (u,v) in p, we just add this $c_f(p)$ to $f(u,v)$ (and subtract it from $f(v,u)$)
  - Resulting flow is a valid flow with a larger value.
  - *What is the residual capacity of the path* (s,a,b,t)?

The Ford-Fulkerson method

```
Ford-Fulkerson(G,s,t)
01 for each edge (u,v) in G.E do
02     f(u,v) ← f(v,u) ← 0
03 while there exists a path p from s to t in residual
   network G_f do
04     c_f = min{c_f(u,v): (u,v) is in p}
05     for each edge (u,v) in p do
06         f(u,v) ← f(u,v) + c_f
07         f(v,u) ← -f(u,v)
08 return f
```

The algorithms based on this method differ in how they choose *p* in step 03.

Correctness of Ford-Fulkerson

- *Max-flow min-cut theorem*:
  - If *f* is the flow in G, the following conditions are equivalent:
    - 1. *f* is a maximum flow in *G*
    - 2. The residual network G_f contains no augmenting paths
    - 3. $|f| = c(S,T)$ for some cut (S,T) of *G*

1.

We must prove three parts:

3. 2.

From this we have 1. $\Leftrightarrow$ 2., which means that the Ford-Fulkerson method always correctly finds a maximum flow

But,

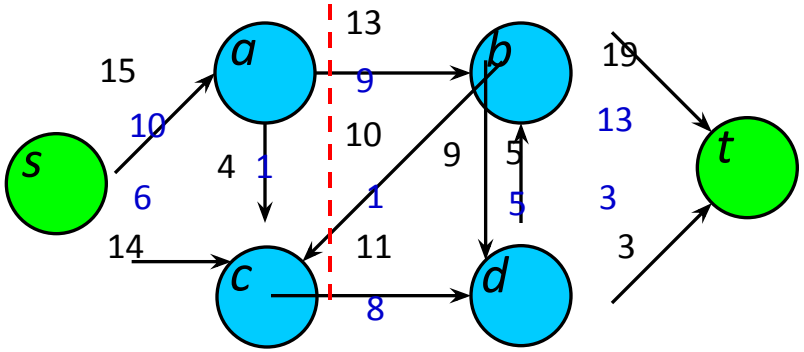
- does not imply that the algorithm terminates
- does not imply that the # loop iterations is small
- need to decide how to pick paths carefully

Cuts

- *cut*: a partition V = S ∪ T, where s ∈ S and t ∈ T
- *net flow across* f is

f(S,T) = (total flow from S to T) − (total flow from T to S)

- The capacity c(S,T)) of the cut = sum of capacities c(u,v), where u∈S and v∈T
- Minimum cut – a cut with the smallest capacity of all cuts
- *Lemma*: for any cut (S,T), net flow f(S,T)=|f|.
  Proof sketch: since s ∈ S and t ∈ T, total flow |f | must cross the S–T boundary.



Max-Flow Min-Cut Proof Sketch

Show all these are equivalent conditions:
1. f is a maximum flow
2. G_f contains no augmenting path
3. $|f|=c(S,T)$ for some cut(S,T)

(1) =⇒ (2) : by definitions of residual network and augmenting path, a maximum flow has no capacity leftover so no paths in G_f

(2) =⇒ (3) : consider a cut where all vertices reachable from s in Gf are in S and the unreachable ones are in T; since there is no $s \rightsquigarrow t$ path in Gf , all edges across the S–T boundary must already be at full capacity

(3) =⇒ (1): trivially $|f| \leq c(S,T)$, and if $|f| = c(S,T)$ then this (S,T) is maximum

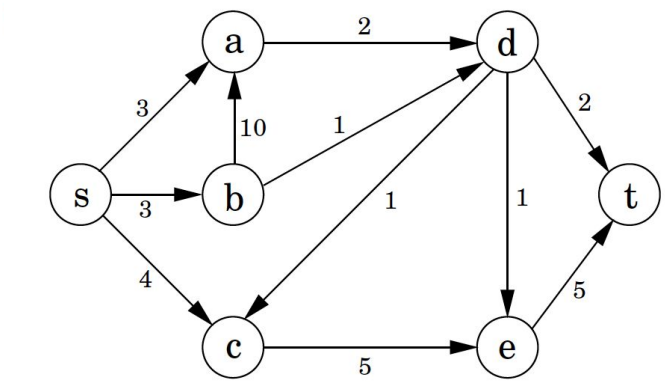
Ford-Fulkerson Detailed Pseudocode

```
function FORD-FULKERSON-METHOD(G = (V , E ), s , t )
  for each edge (u, v) in E do
    (u,v).f =0
  end for
  while there exists an augmenting path p in G_f do
    cf (p) = min{cf (u,v) : (u,v) ∈ p}
    for each edge(u,v)∈p do
      if (u,v)∈E then
        (u,v).f =(u,v).f +cf(p)
      else
        (u,v).f =(v,u).f −cf(p)
      end if
    end for
  end while
  return flow on .f fields
end function
```

Worst-case running time

- *What is the worst-case running time of this method?*
  - Let's assume integer flows.
  - Each augmentation increases the value of the flow by some positive amount.
  - Augmentation can be done in $O(E)$.
  - Total *worst-case* running time $O(E|f^*|)$, where f^* is the max-flow found by the algorithm.
  - *Can we run into this worst-case*?
  - *Lesson: how an augmenting path is chosen is very important!*

The Max Flow Problem



- A network of pipes along which oil can be sent
- Each pipeline has a maximum capacity
- No opportunity for storing oil on route
- Find the maximum flow possible from the source s to the sink t

Here is one solution

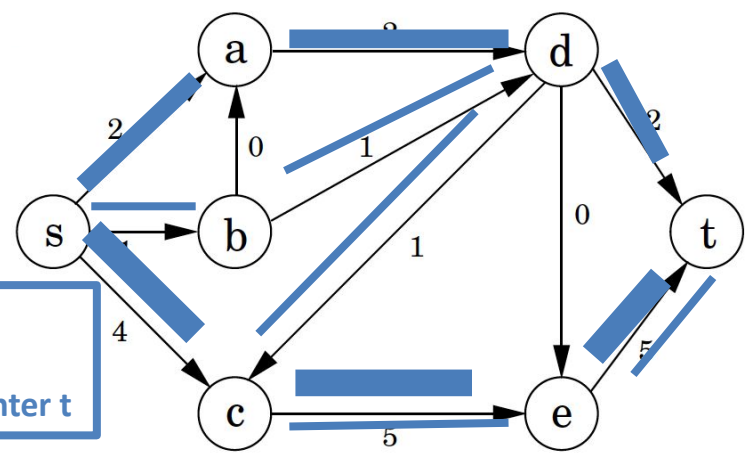
The edge labels show the final flow.

Next we show how we got there:

Flow is increased along paths from s to t

2 units
4 units
1 unit

Total: 7 units
Maximum!
Because only two edges (capacity 2+5) enter t



This method does not always give the maximum

1st Path: A B F D: 2 units

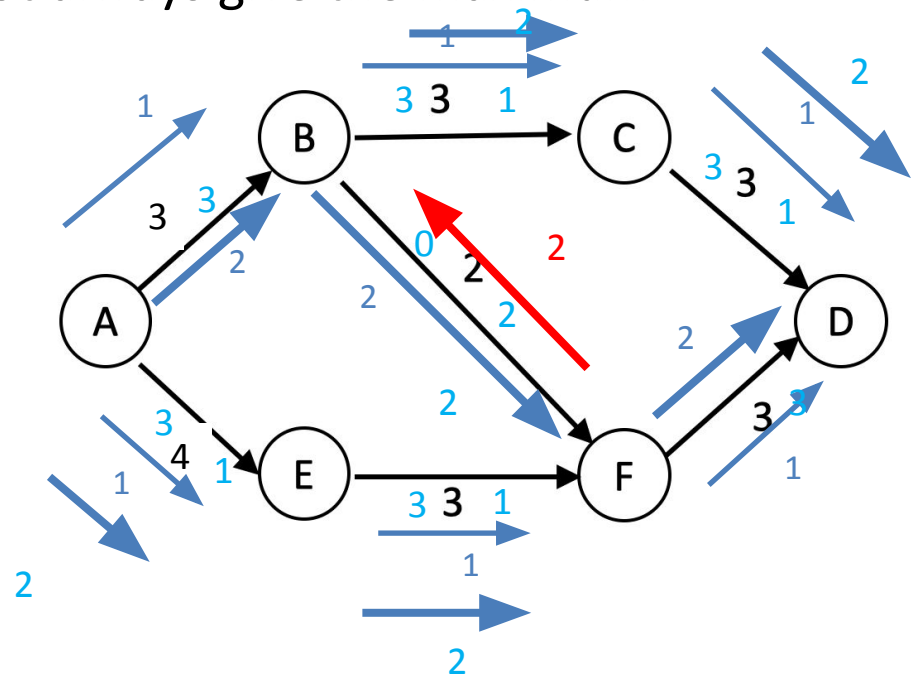
2nd Path A B C D: 1 unit

3rd Path: A E F D: 1 unit

We are not able to find any more s-t paths along which flow can be increased

We introduce the notion of back edges

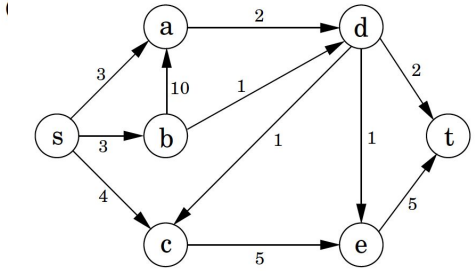
4th Path: A E F ← B C D: 2units



Key concept: Augmenting Path
s-t path with

- forward edges where flow is below capacity, i.e. flow can be increased
- Back edges where flow can be decreased, i.e. flow > 0

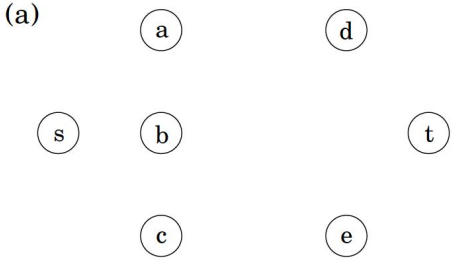
Use of a residual graph

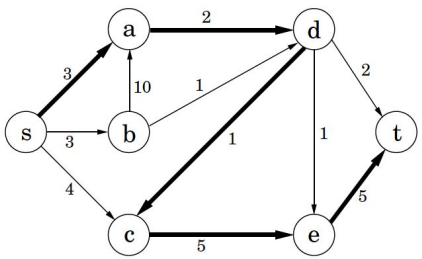


Current flow

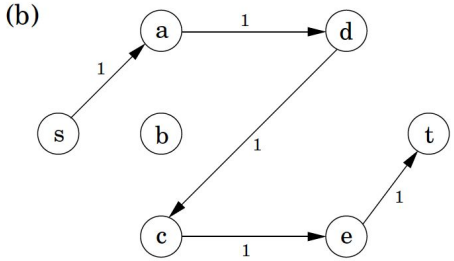
Residual graph

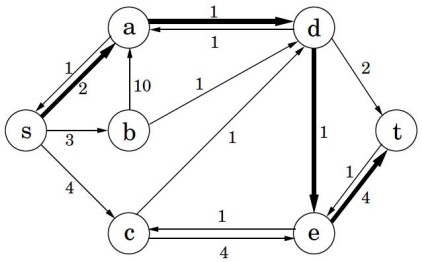
(a)



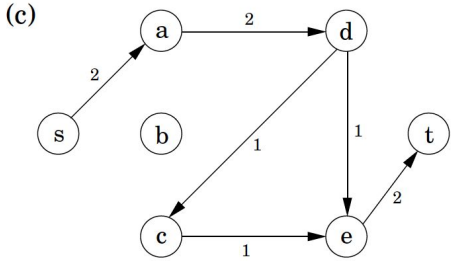


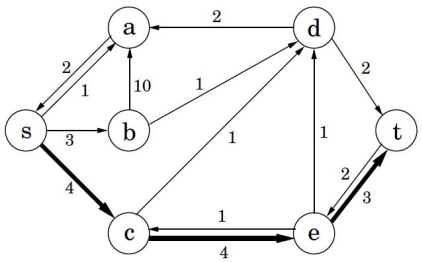
(b)



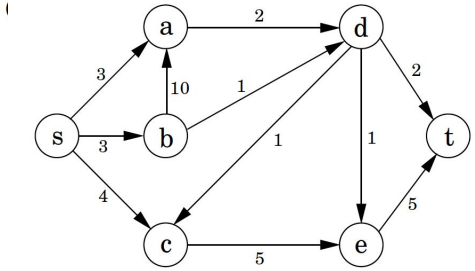


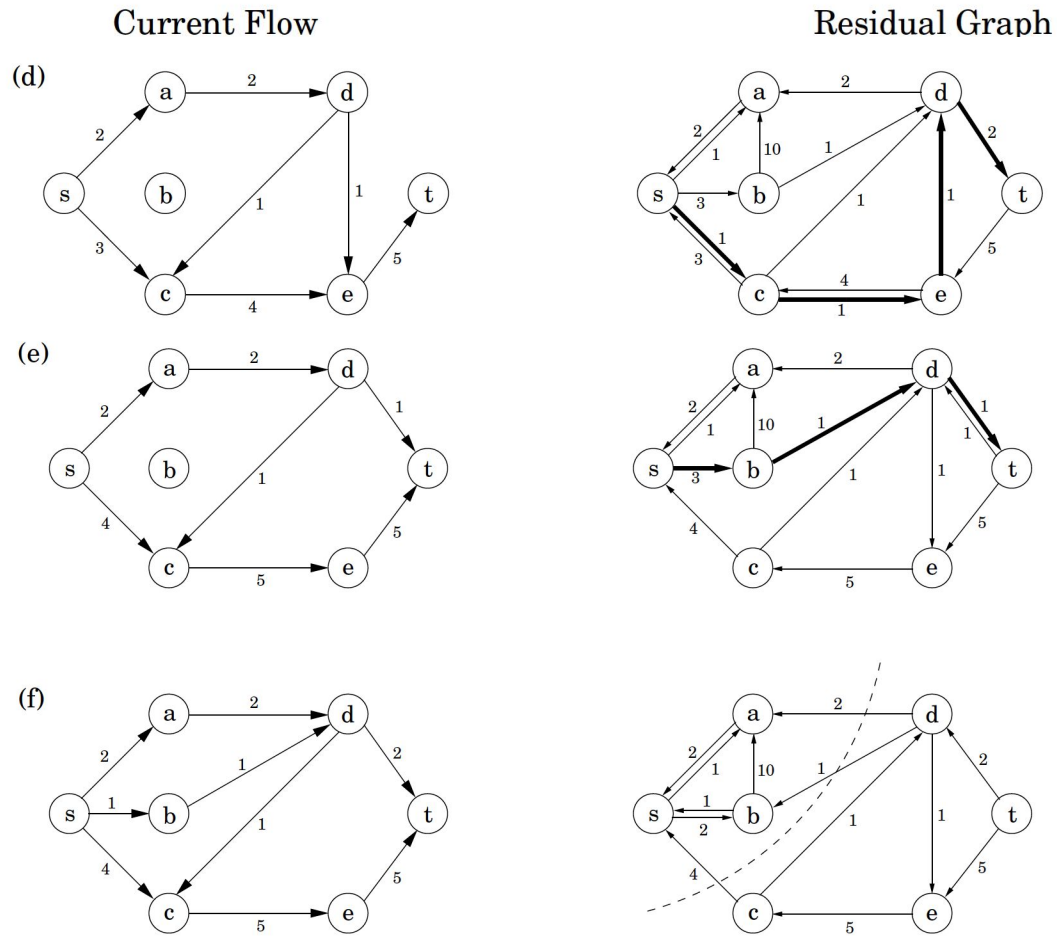
(c)





Use of a residual graph continued…

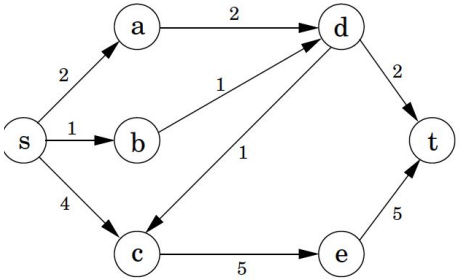


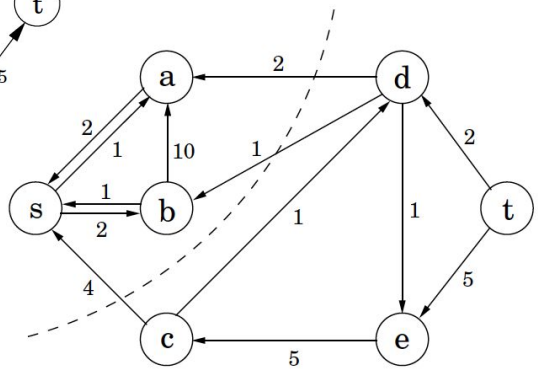


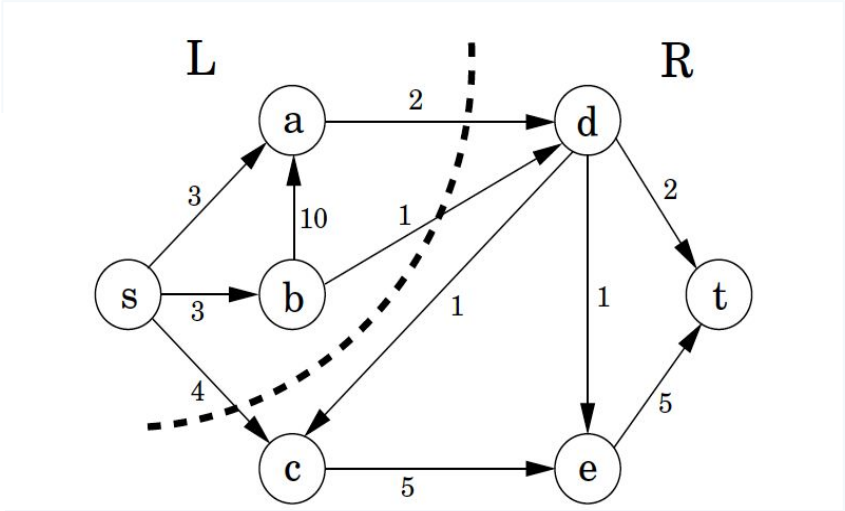
Max-Flow Min-Cut: Proof of optimality

- Pick any flow f and any (s,t)-cut (L,R). Then size(f) $\leqq$ capacity(L,R).
- This means that if some flow value is equal to some cut value then they are both optimal.
- Once the max-flow algorithm finishes L is the nodes reachable by an augmenting path, and R is the nodes not reachable by an augmenting path.

L={s,a,b} R={c,d,e,t}







Edmonds-Karp Algorithm

(taken from https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/07-max-flow.pdf)

Edmonds-Karp Algorithm is

- Ford-Fulkerson method from previous page, and...
- use breadth-first search (BFS) to find the shortest augmenting path
- (shortest ≡ fewest vertices, irrespective of weights)
- now a concrete, runnable, implementable algorithm
- performs $O(|V| \cdot |E|)$ augmentations
- takes $O(|V| \cdot |E|^2)$ time
- for $n = |V|$, this is $O(n^3)$ in a sparse graph and $O(n^5)$ in a dense graph
- more complicated relabel-to-front algorithm takes $O(|V|^3) = O(n^3)$ time

```
function EDMONDS-KARP(G = (V , E ), s , t )
  initialize each edge's flow to 0
  repeat
    for k = 2,3,...,|V| do
      if ∃ augmenting path p of length k then
        cf (p) = minimum excess capacity of any edge in p
        for edge e in p do
          if p follows e forwards then
            increase e's flow by cf (p)
          else
            decrease e's flow by cf (p)
          end if
        end for
        break loop
      end if
    end for
  until no path can be found
  return flow based on current capacities
end function
```

Identifying Edge Capacity in G

When running this algorithm by hand,

- you could sketch the residual network each time, but this is tedious
- instead, when looking at edge e with flow x/c
- If $x < c$, you may follow e forwards and add up to $(c-x)$ flow
- if $x > 0$, you may follow e backwards and subtract up to x flow

Edmonds-Karp algorithm

- *Take **shortest path** (in terms of number of edges) as an augmenting path – Edmonds-Karp algorithm*
  - *How do we find such a shortest path?*
  - Running time $O(VE^2)$, because the number of augmentations is $O(VE)$
  - To prove this we need to prove that:
    - *The length of the shortest path does not decrease*
    - *Each edge can become **critical** at most ~ V/2 times.* Edge (u,v) on an augmenting path p is critical if it has the minimum residual capacity in the path*:*

      $c_f(u,v) = c_f(p)$

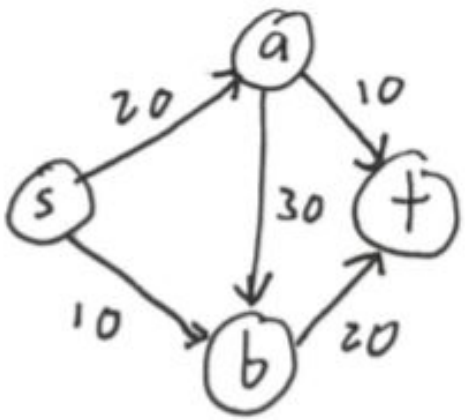
Non-decreasing shortest paths

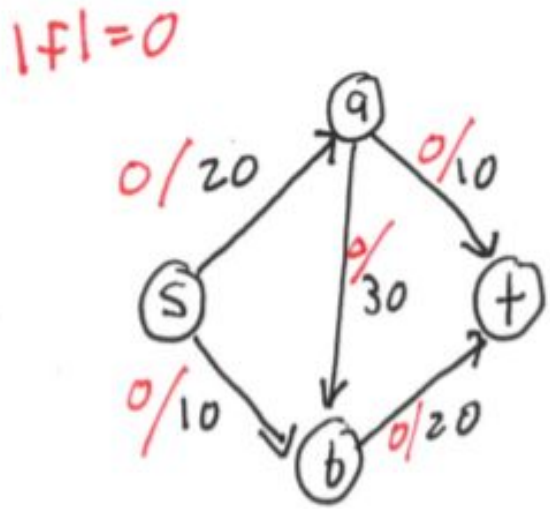
- *Why does the length of a shortest path from s to any v does not decrease?*
  - Observation: Augmentation may add some edges to residual network or remove some.
  - Only the added edges ("shortcuts") may potentially decrease the length of a shortest path.
  - Let's supose ($s,\ldots,v$) – the shortest decreased-length path and let's derive a contradiction

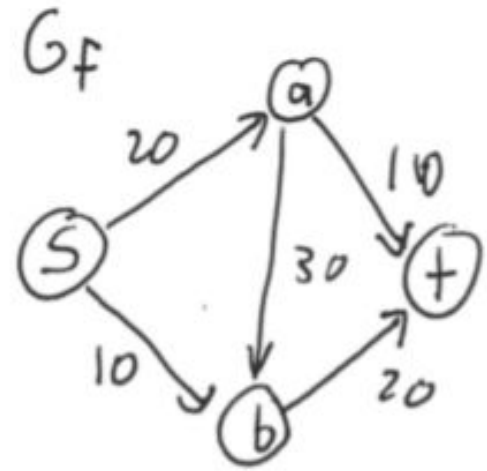
Number of augmentations

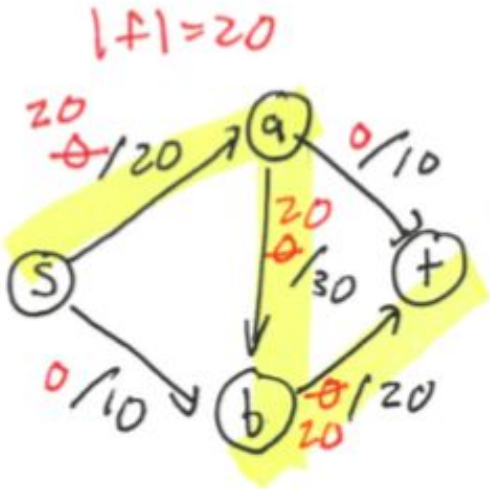
- *Why each edge can become critical at most ~V/2 times*?
  - Scenario for edge (*u,v*):
    - Critical the first time: (*u,v*) on an augmenting path
    - Disappears from the network
    - Reappears on the network: (*v,u*) has to be on an augmenting path
    - We can show that in-between these events the distance from *s* to *u* increased by at least 2.
    - This can happen at most *V*/2 times
- We have proved that the running time of Edmonds-Karp is $O(VE^2)$.

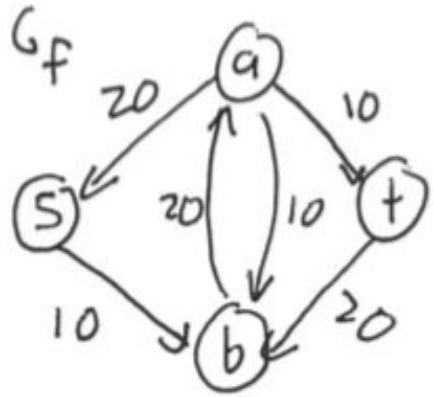
Edmonds-Carp Example 1

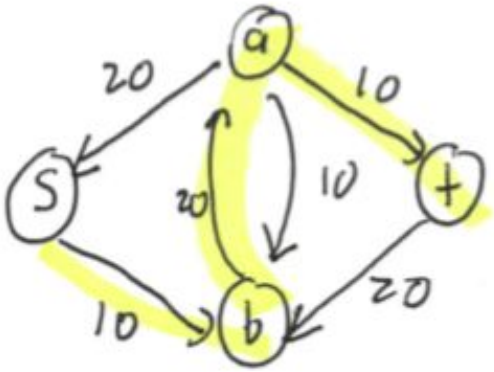


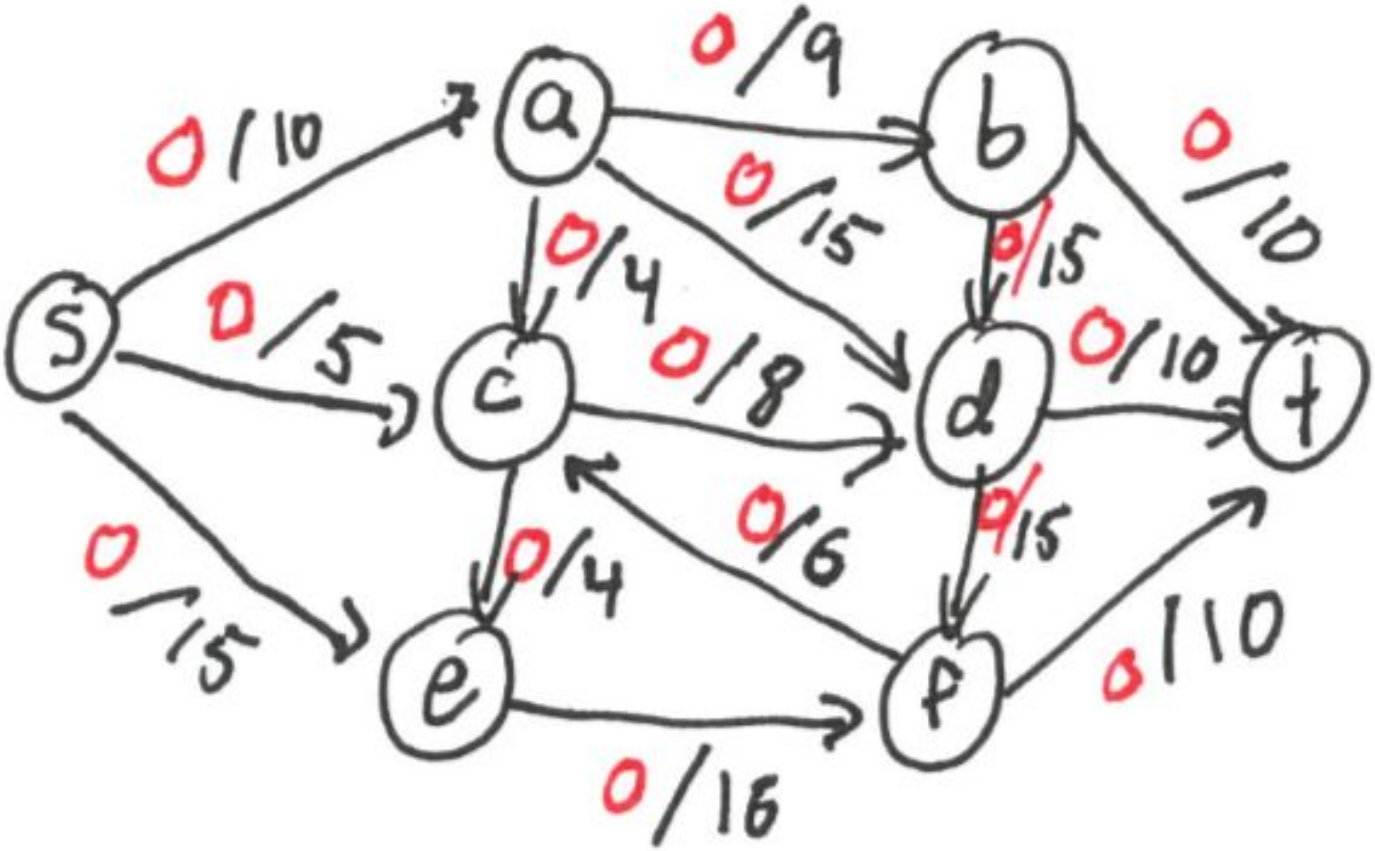




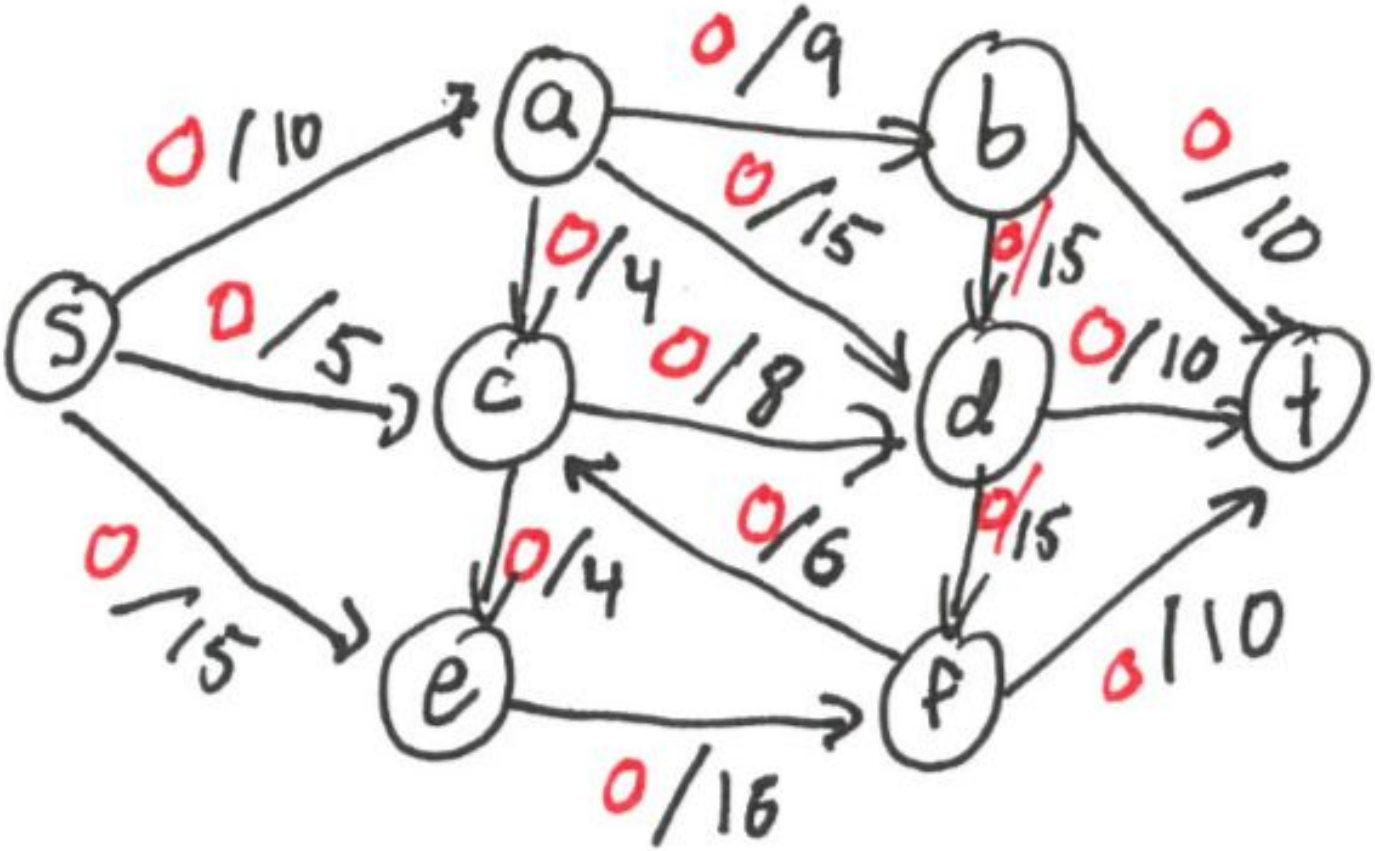

|f|=20
20
0/20
a
0/10
20
0/30
s
t
0/10
b
0/20
20


G_f
20
a
10
s
20
10
t
10
b
20

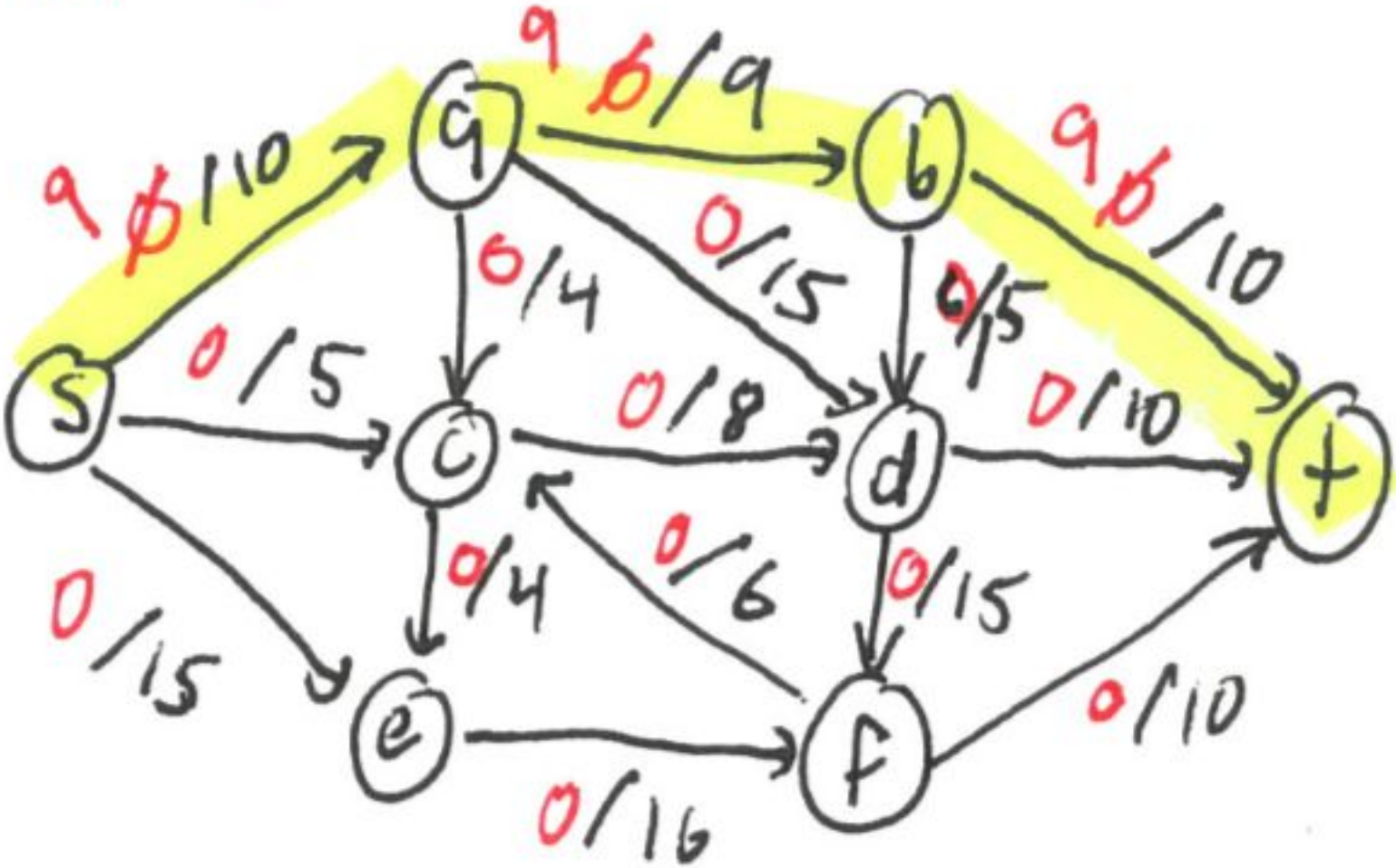

20
a
10
s
20
10
t
10
b
20


s
a
b
c
d
e
f
t
0/10
0/9
0/15
0/10
0/4
0/15
0/5
0/8
0/10
0/6
0/15
0/15
0/4
0/10
0/16

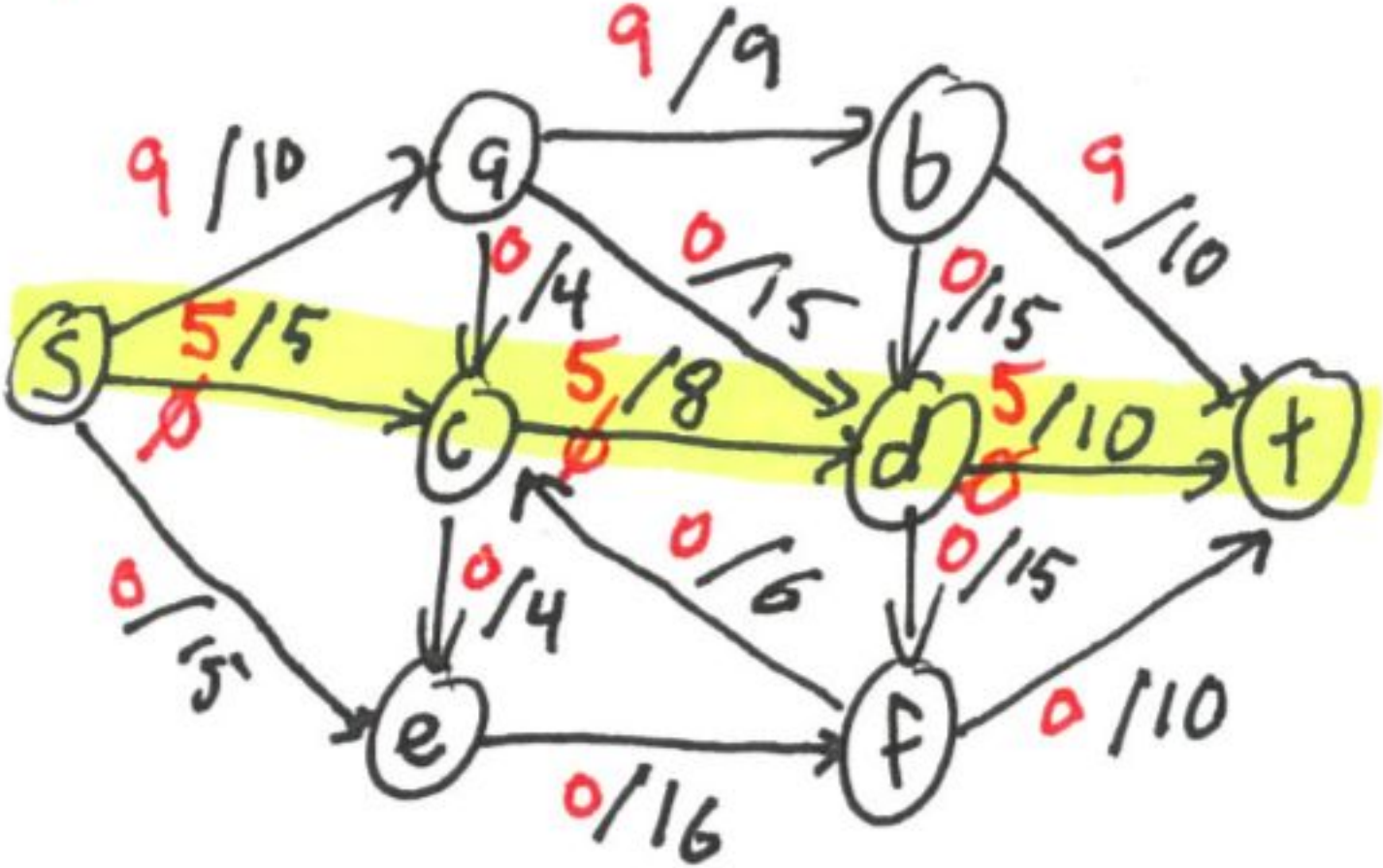
Edmonds-Carp Example 2



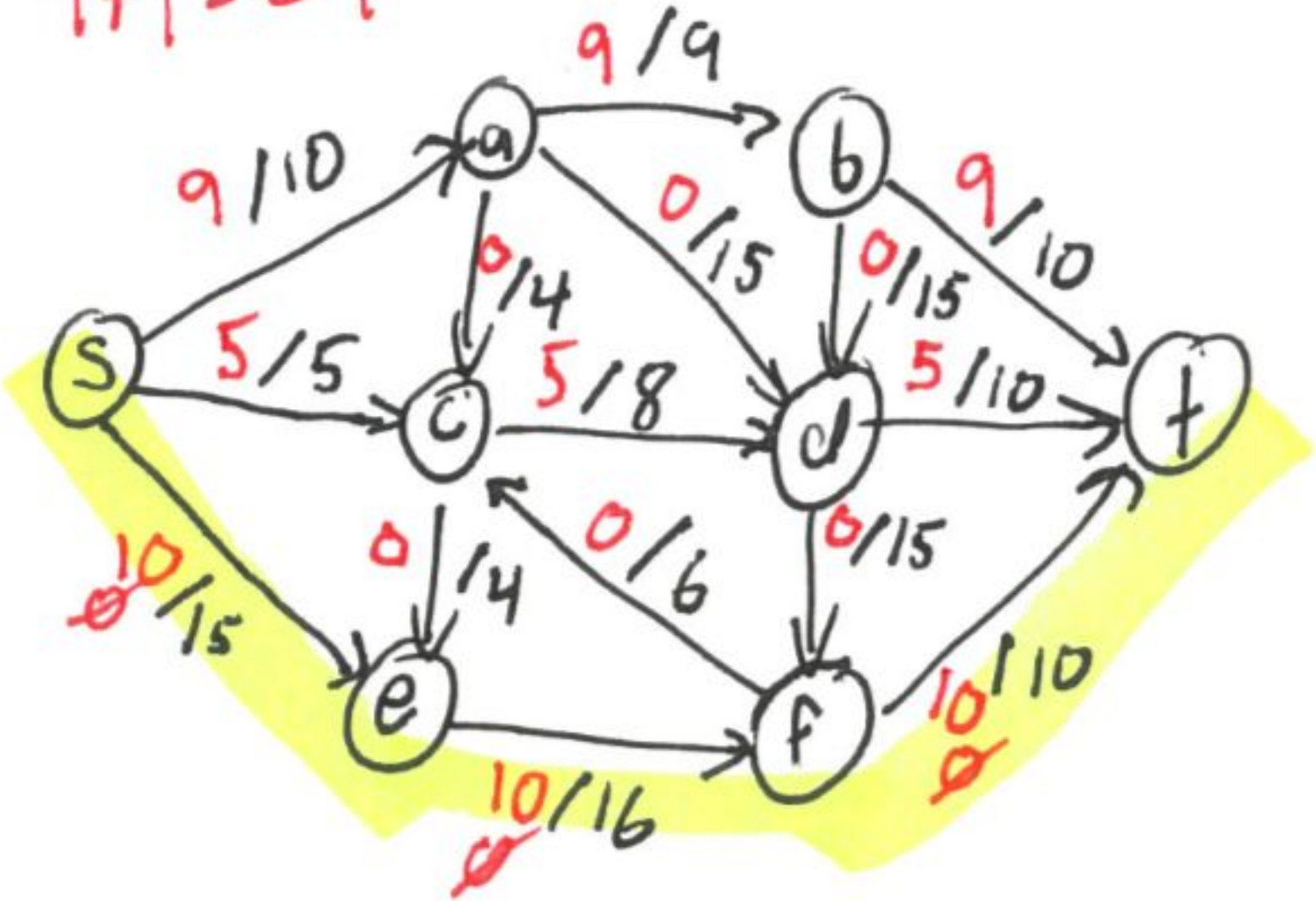
$|f| = 9$



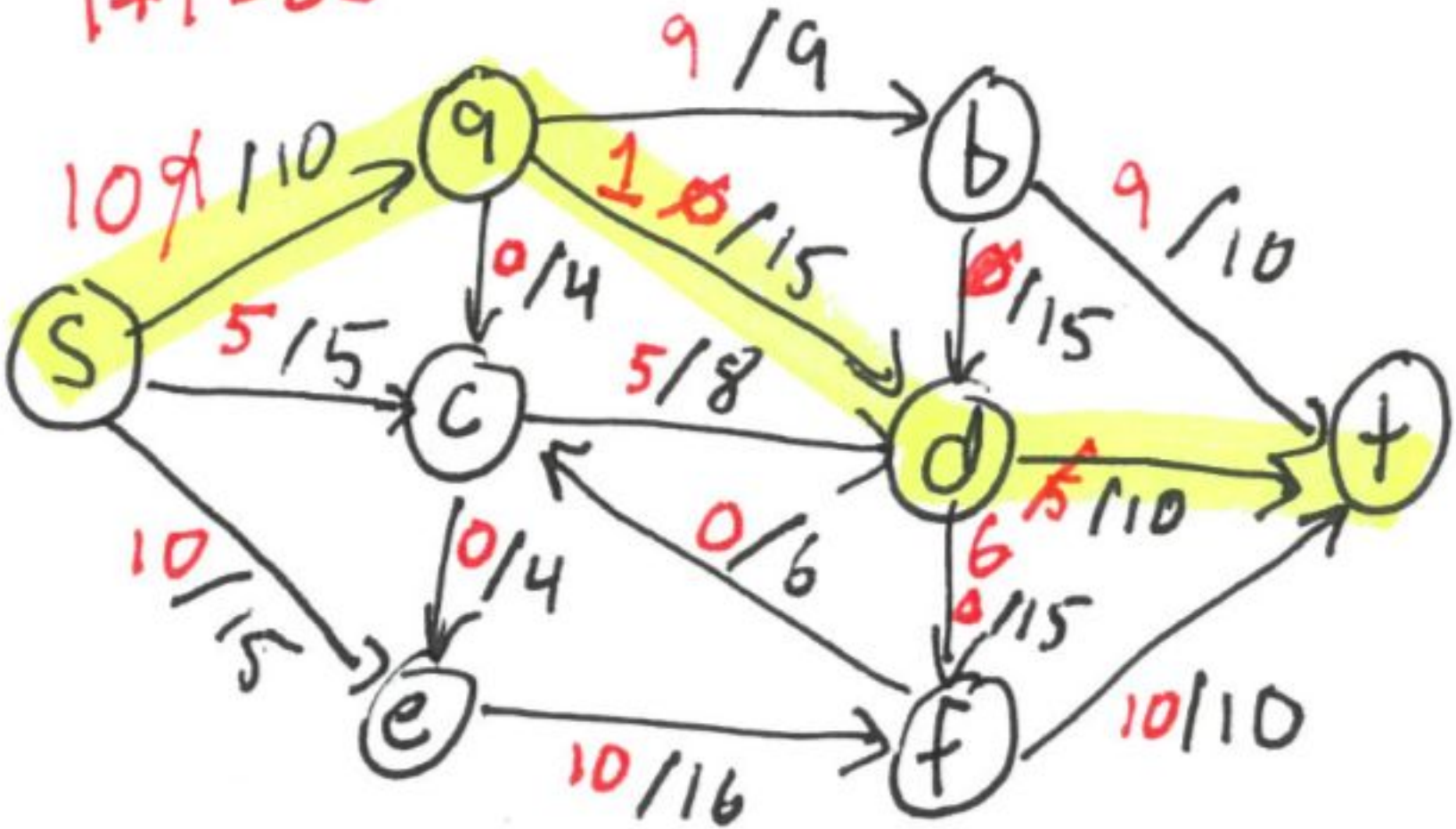
$|f| = 14$



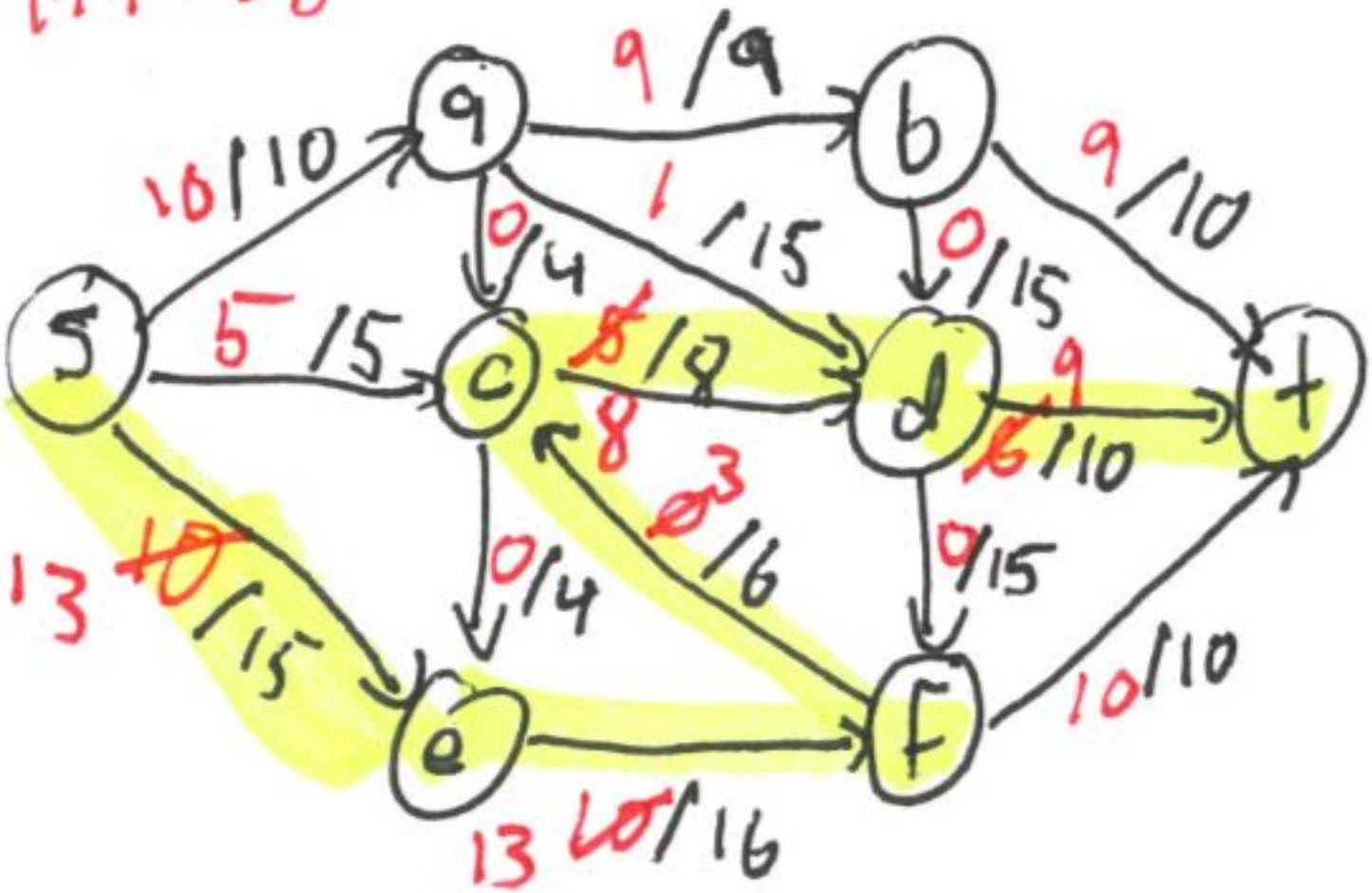
$|f| = 24$


s
a
b
c
d
e
f
t
9/10
9/9
0/15
9/10
0/4
0/15
5/5
5/8
5/10
10/15
0/4
0/6
0/15
10/16
10/10

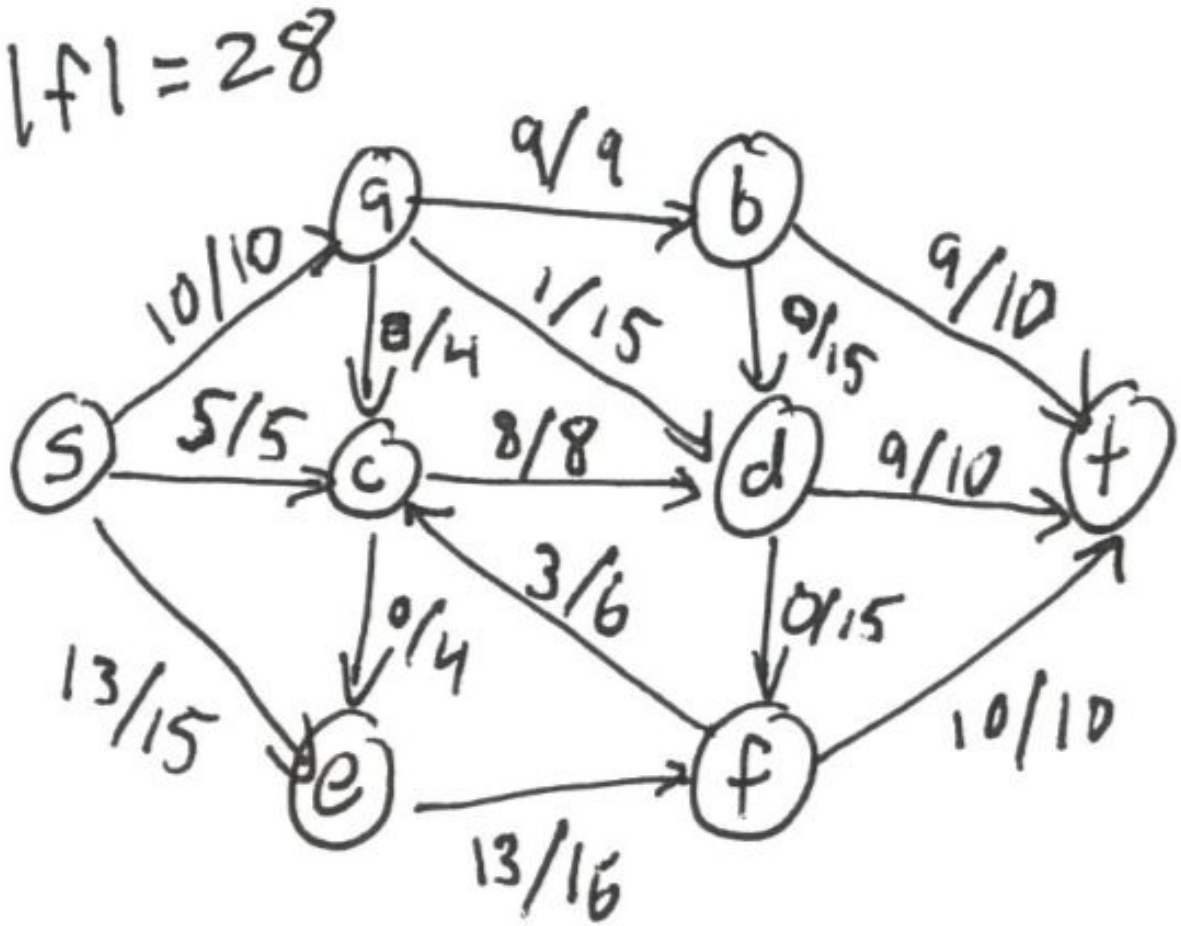
$|f| = 25$


s
a
b
c
d
e
f
t
10 9/10
9/9
1 0/15
9/10
0/4
0/15
5/15
5/8
5 5/10
10/15
0/4
0/6
6 0/15
10/16
10/10

$|f| = 28$

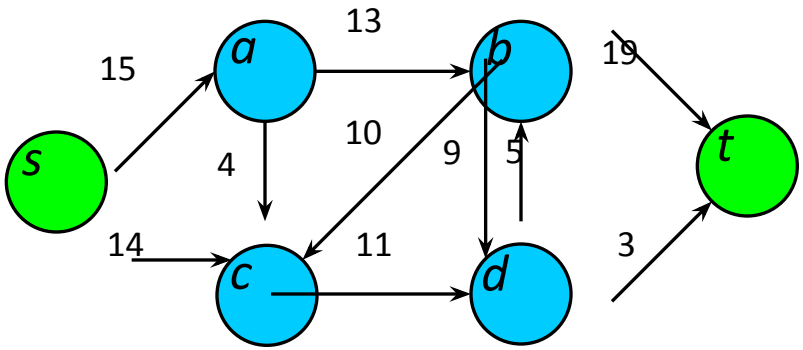

s
a
b
c
d
e
f
t
10/10
9/9
9/10
1/15
0/4
0/15
5/15
5/8
9/10
8/16
3/16
13/15
0/4
0/15
10/10
13/16

Final answer for Example 2



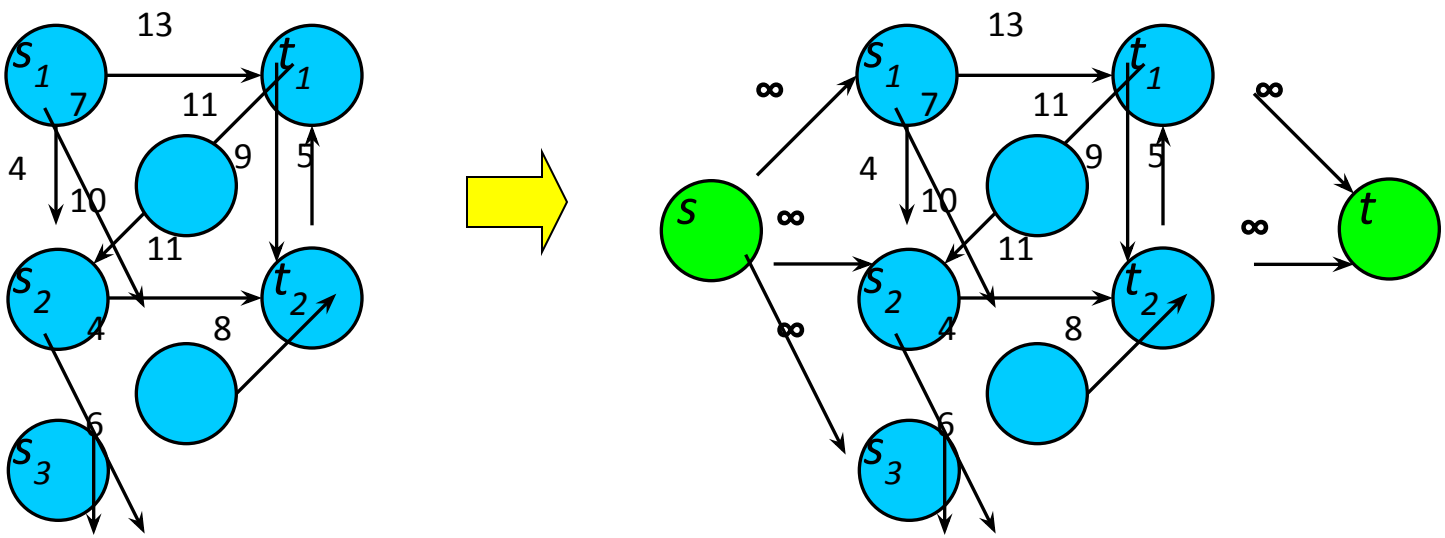
Example 3 of Edmonds-Karp

- *Run the Edmonds-Karp algorithm on the following graph:*



Multiple sources or sinks

- *What if we have more sources or sinks?*
  - Augment the graph to make it with one source and one sink!



Application of max-flow

- *Maximum bipartite matching problem*
  - ***Matching*** in a graph is a subset *M* of edges such that each vertex has at most one edge of *M* incident on it. It puts vertices in pairs.
  - We look for *maximum* matching in a ***bipartite*** graph, where $V = L \cup R$, L and R are disjoint and all edges go between L and R
  - Dating agency example:
    - *L* – women, *R* – men.
    - An edge between vertices: they have a chance to be "compatible" (can be matched)
    - Do as many matches between "compatible" persons as possible

Bipartite Matching

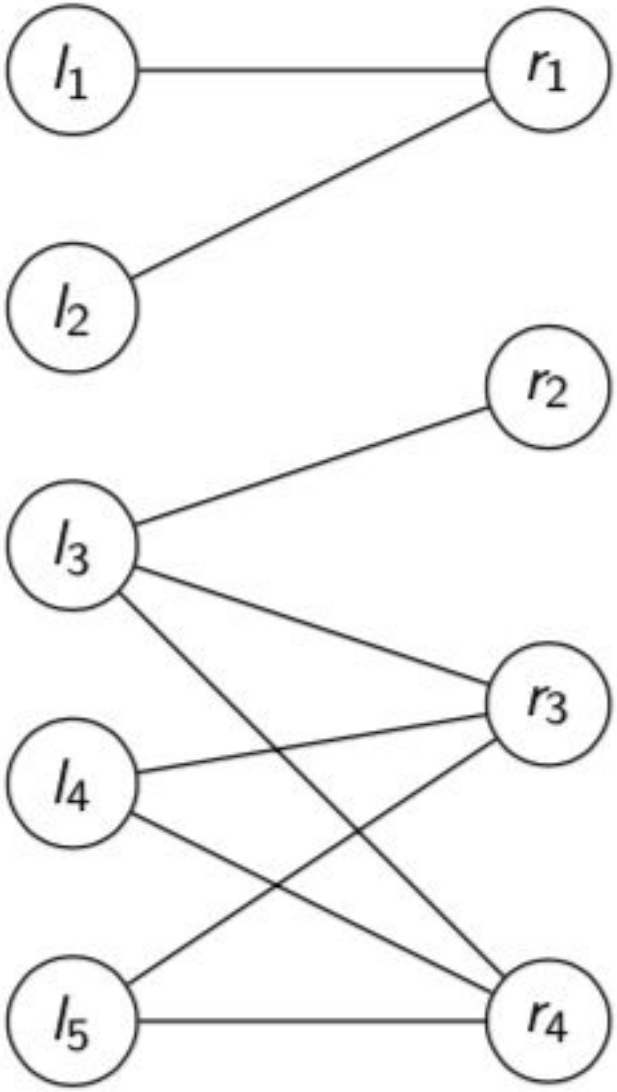
bipartite maximum matching

input: an undirected bipartite graph $G = (V , E)$ where $V = L \cup R$ are the parts of G

output: a matching $M \subseteq E$ where the number of matched vertices is maximum

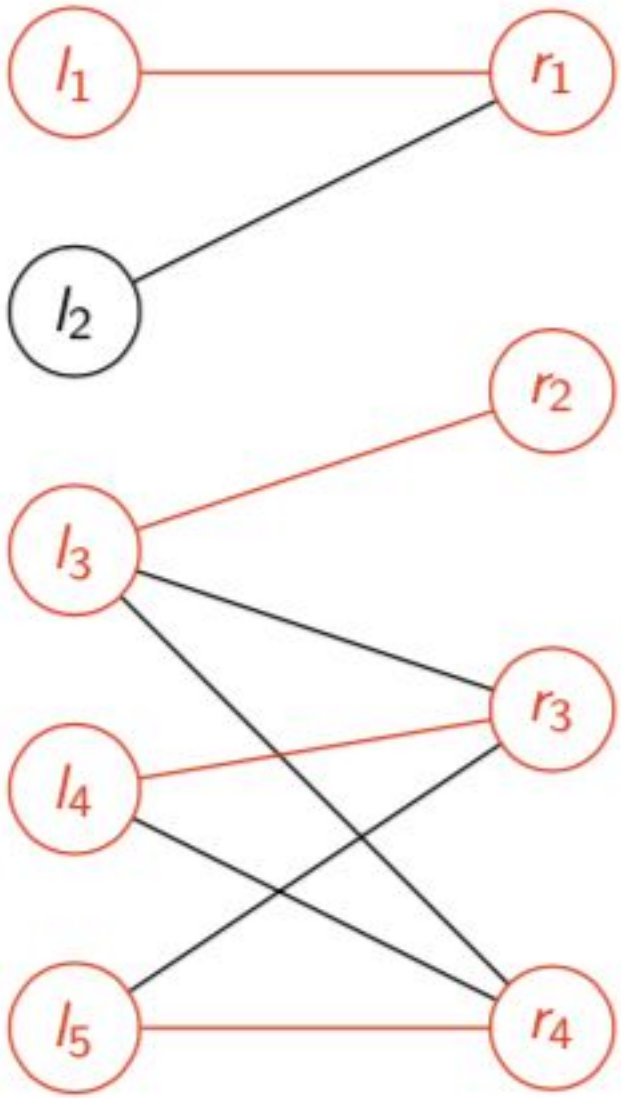
- bipartite: L, R are disjoint, and edges only go between L, R
- matching: pick edges that “pair off” two vertices; goal is to
- maximize # paired-off
- intuitively, L is one kind of thing and R is another kind of thing

Example of Bipartite Matching



Input graph

Maximum bipartite matching



matching

M = {included edges} =
{{l1, r1}, {l3, r3}, {l4, r3}, {l5, r4}}

|M| = 4

(other optimal matchings exist)

Bipartite Matching Applications

- any scenario where there are two kinds of things that can be paired
- goal is simply to select the maximum number of pairings
- casting for a play: L = set of actors; R = set of roles; edge {l,r} exists when l could play role r
- packing leftover food (one item/container): L = set of food items; R = available containers; edge {l,r} exists when food l could fit in container r
- scheduling appointments: L = set of clients; R = set of time slots; edge {l,r} exists when client l could meet appointment r
- might feel NP-hard, but actually is in P

Maximum bipartite matching

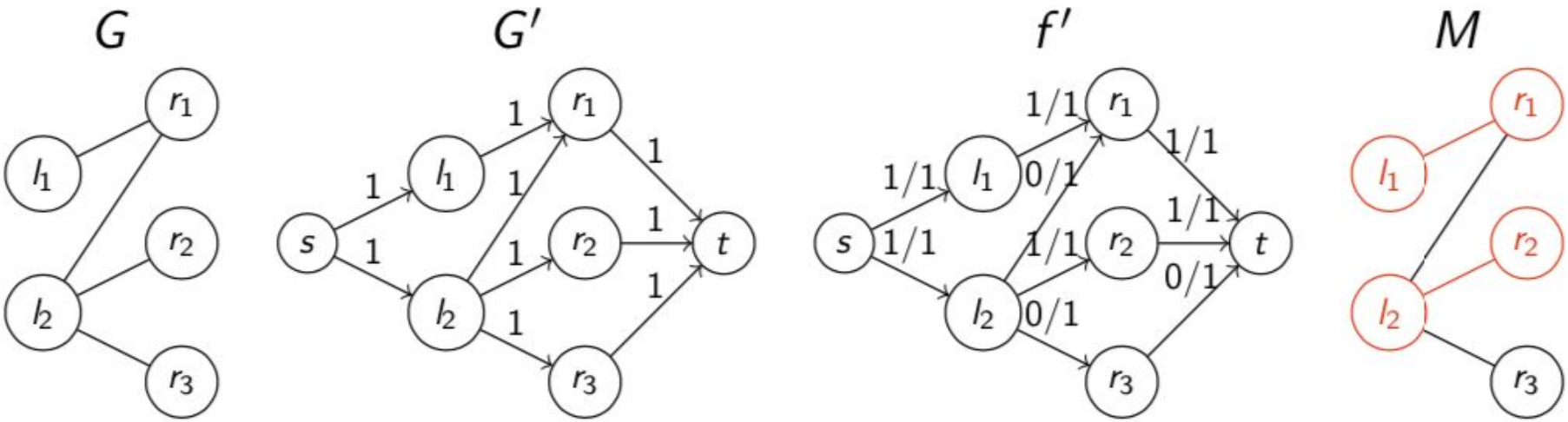
- *How can we reformulate this problem to become a max-flow problem*?

- *What is the running time of the algorithm if we use the Ford-Fulkerson method*?

Formulating Bipartite Matching as Flow

- let $G = (V, E)$ be bipartite matching instance
- create $G' = (V', E')$ with $V' = V \cup \{s, t\}$ where s, t are new source/sink
- create edges
  - $(l,r)\ \forall l \in L, r \in R, \{l,r\} \in E$
  - $(s,l)\ \forall l \in L$
  - $(r,t)\ \forall r \in R$
- every edge (v,w) has capacity $c(v,w) = 1$
- post-processing: $\text{edge}(l,r) \in M$ ifff $(l,r)=1$
- observe $|V'| \in O(|V|), |E'| \in O(|E|)$, overhead is $O(|V|+|E|)$
- $\Rightarrow$ if this is correct, can solve bipartite matching in $O(|V|^3)$ time

Example



$$M = \{\{l, r\} \mid f'(l, r) = 1\} = \{\{l_1, r_1\}, \{l_2, r_2\}\}$$

M ={{l,r}|f′(l,r)=1}={{l_1,r_1},{l_2,r_2}}

(other max flows exists ⇔ other maximal matchings exist)

Correctness and Time Complexity

- $\exists$ matching M with cardinality k = |M| iff $\exists$ some flow f with value k = |f|
  - key idea: pairing two vertices in the matching adds exactly one flow from s $\rightsquigarrow$ t
  - there are no opportunities for flow aside from matched vertices
- $\Longrightarrow$ a maximum flow in G′ corresponds to a maximum matching in G
- classical max-flow problem can be solved in $O(|V|^3)$ time, in P
- robust max-flow problem (supports unreachable vertices, antiparallel edges, multiple sinks/sources) also in $O(|V|^3)$ time w/ worse constant factors, in P
- bipartite matching reduces to max-flow, so bipartite matching can be solved in $O(|V|^3)$ time, in P
- other practical, distinct problems reduce to max-flow or bipartite matching so take $O(|V|^3)$ time and are in P

Linear programming (LP)

(taken from Levitin, pages 244-246)

- A ***linear programming*** problem: a problem of optimizing a linear function of several variables subject to constraints in the form of linear equations and linear inequalities => an optimal decision problem

maximize (or minimize) $c_1x_1 + \ldots + c_nx_n$ (called *objective function*)
subject to $a_{i1}x_1+\ldots+a_{in}x_n \leq$ (or $\geq$ or $=$)b_i for i=1,...,m (called *constraints*)
$x_1 \geq 0,\ldots,x_n \geq 0$ (called *nonnegative constraints*)

- ***Integer linear programming*** **(ILP)** problem: the variables of a LP problem are required to be integers
- Textbook CLRS section 29.3

Solving LP problems

(taken from https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/10-linear-programming-problem.pdf and Levitin, pages 245-246)

- On the border of P: many tough problems in P, including max-flow, reduce to LP
- The classic algorithm for this problem is called the ***simplex method,*** invented by the U.S. mathematician George Dantzig in the 1940s
  - simplex algorithm technically takes $O(2^n)$ worst-case time, but is fast polynomial on most practical inputs
  - we have pseudo-polynomial algorithms with e.g. $O(n^{2.5}W)$ runtime and expensive constant factors
  - open question whether there is a strongly polynomial LP algorithm with runtime e.g. $O(n^3)$, not a function of W
- A more recent algorithm by Narendra Karmarkar (1984) is polynomial and performed competitively with the simplex method in empirical tests

Standard versus general form of LP

- standard form: restricted/simplified LP, easier for algorithms to solve
- general form: more convenient for end-user formulations
  - general reduces to standard with constant overhead

Standard form	**General form**
maximize objective function	maximize or minimize objective function
all variables are non-negative	no restriction (may be negative)
every constraint is ≤ r.h.s.	constraint may be ≤ , = or ≥ r.h.s.

General form example

minimize $x_1 - x_2$
subject to

$$x_1 \leq 100$$
$$x_2 \geq 2$$
$$x_1 + x_2 = 10$$
$$x_2 \geq 0$$

Note that:

- The objective function is minimizing objective function
- mix of ≤, =, ≥ constraints
- not all variables have $x_i \geq 0$ non-negativity constraint (x_1 is missing so it could take any value in R)

Standard form

standard form with n variables and m constraints:

maximize $c_1x_1 + \ldots + c_nx_n$
subject to

$$a_{1,1}x_1 + a_{1,2}x_2 + \ldots + a_{1,n}x_n \leq b_1$$
$$a_{2,1}x_1 + a_{2,2}x_2 + \ldots + a_{2,n}x_n \leq b_2$$

$$a_{m,1}x_1 + a_{m,2}x_2 + \ldots + a_{m,n}x_n \leq b_m$$
$$x_1, \ldots, x_n \geq 0$$

Example 1

$$
\begin{aligned}
\text{maximize} \quad & 2x_1 + x_2 - 1/3\, x_3 \\
\text{subject to} \quad & x_1 + x_2 \leq 10 \\
& -x_3 \leq -2 \\
& x_1, x_2, x_3 \geq 0
\end{aligned}
$$

Example 2 (knapsack)

- Knapsack problem: Given a knapsack of capacity W and n items of weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$, find the most valuable subset of the items that fits into the knapsack.
- Let x_j, j = 1, . . . , n, be a variable representing a fraction of item j taken into the knapsack. Obviously, x_j must satisfy the inequality $0 \le x_j \le 1$. Since we are only allowed either to take a whole item or not to take it at all, x_j =0 or 1. Hence, we have the following integer linear programming problem (called the **0-1** version) of the knapsack problem:

$$\begin{aligned} \text{maximize} \quad & v_1x_1 + \ldots + v_nx_n \\ \text{subject to} \quad & w_1x_1 + w_2x_2 + \ldots + w_nx_n \le W \\ & x_1, \ldots, x_n \in \{0,1\} \end{aligned}$$

Matrix notation of the standard form

- Collect all variables into vector $x=\langle x_1,\ldots,x_n\rangle$,

objective coefficients into vector $c = \langle c_1, \ldots, c_n\rangle$
r.h.s. of inequalities into vector $b = \langle b_1, \ldots, b_m\rangle$
$a_{i,j}$ coefficients into matrix A

- LP can be written in terms of dot-product and matrix-vector multiplication as (and note the transpose c^T):

maximize $c^T x$

subject to

$Ax \leq b$

$x \geq 0$

Possible outcomes

- LPs are not always solvable!
- There are three outcomes:
  1. **solution**: concrete values for $x_1, \ldots, x_n$ that maximize $c^T x$ (good, usually the goal)
  2. **unbounded**: objective can be made arbitrarily large i.e. $+\infty$ (bad, usually means there is a bug in your LP that makes it nonsensical)
  3. **infeasible**: impossible to satisfy all constraints simultaneously (bad, usually means that either your LP is nonsensical; or your LP makes sense but meeting all your goals is impossible)

Visual representation

- Read slides 11-19 of https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/10-linear-programming-problem.pdf

Formulating General LP as Standard LP

(taken from https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/12-lp-formulations.pdf)

- pre-processing: convert general LP to standard LP
- need to get rid of any non-standard feature
- want insignificant overhead
  - ✔ n = #variables, m = # constraints
  - ✔ space complexity of general LP is $\Theta(nm)$
  - ✔ want size of resulting standard LP, and time overhead, to both be $O(nm)$
- three features to deal with
  - minimization
  - negative variables
  - =, ≥ constraints

Converting “minimize” to Standard Form

- minimize $f(x) \equiv$ maximize $-f(x)$
  so if general LP says

  minimize $c_1x_1 + c_2x_2 + \ldots + c_nx_n$,

replace that with

  maximize $-c_1x_1 - c_2x_2 - \ldots - c_nx_n$

- size of LP unchanged
- It takes O(n) time to negate coefficients

Converting negative variables to Standard Form

- If x_j does not have a non-negativity constraint $x_j \geq 0$, call x_j a negative variable
- Handle each negative variable one at a time
- Negative variable x_j becomes two non-negative variables x'_j, x''_j
- invariant: $x_j = x'_j - x''_j$
- x'_j = the "positive" part of x_j
- x''_j = the "negative" part of x_j
- x_j positive $\Leftarrow\Rightarrow$ $x'_j > 0$ and $x''_j = 0$
- x_j negative $\Leftarrow\Rightarrow$ $x'_j = 0$ and $x''_j > 0$

Converting Negative Variables to Standard Form

The algorithm becomes as follows:

- add non-negativity constraints $x'_j \geq 0$, $x''_j \geq 0$
- substitute $(x'_j - x''_j) = x_j$ into LP
  - obj. func: $c_j x_j \Rightarrow c_j(x'_j - x''_j) = c_j x'_j - cj\, x''_j$
  - constraints: $a_{i,j} x_j \Rightarrow a_{i,j}(x'_j - x''_j) = a_{i,j} x'_j - a_{i,j} x''_j$
- after solving standard LP, to complete general LP evaluate $x_j = x'_j - x''_j$
- The values for n, m at most double $\Rightarrow$ still $\Theta(nm)$ space
- Time complexity O(nm) time with care (do all substitutions in one pass)

Converting = constraints to inequalities

- this step converts any = constraint to ≤, ≥ constraints
- ≥ constraints are eliminated in the next step
- $a=b \Leftarrow\Rightarrow a \leq b$ and $a \geq b$
- replace each = constraint

$$a_{i,1}x_1+a_{i,2}x_2+\ldots+a_{i,n}x_n = b_n$$

with two constraints

$$a_{i,1}x_1+a_{i,2}x_2+\ldots+a_{i,n}x_n \geq b_n$$
$$a_{i,1}x1+a_{i,2}x_2+\ldots+a_{i,n}x_n \leq b_n$$

- The value of n unchanged, m at most doubles ⇒ still Θ (nm) space
- O(nm) time

Converting ≥ constraints to ≤ constraints

- $a \geq b \Leftarrow\Rightarrow -a \leq -b$
- replace each ≥ constraint

$$a_{i,1}x_1+a_{i,2}x_2+\ldots+a_{i,n}x_n \geq b_n$$

with

$$-a_{i,1}x_1-a_{i,2}x_2 - \ldots -a_{i,n}x_n \leq -b_n$$

- The values of n, m unchanged so still $\Theta(nm)$ space
- $O(nm)$ time

General-form LP problem

input:

- Boolean for whether f is maximized/minimized
- vector $c \in R^n$
- vector $b \in R^m$
- vector $o \in \{\leq, =, \geq\}^m$
- Matrix A of size $m \times n$ of real numbers

output: either

1. "unbounded"; or
2. "infeasible"; or
3. "solution" with a vector $x \in R^n$ maximizing or minimizing the objective function

Formulating General LPs

Need to define

1. the variables, what they represent
2. the objective function, whether it is maximizing or minimizing
3. the constraints, what they represent
4. the entire LP in general form
5. how to interpret
   - infeasible result
   - unbounded result
   - solution result; how to interpret solution vector x in terms of business logic

Visual representation

- Read slides 15-20 of https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/12-lp-formulations.pdf

Slack form

- standard form: constraint says l.h.s ≤ r.h.s. ⇒ the difference or "slack" between r.h.s. and l.h.s. is ≥ 0: r.h.s – l.h.s ≥ 0, so we call "slack" the difference

  slack = r.h.s – l.h.s
- slack form: constraint says l.h.s. + slack = r.h.s.
- increasing objective = decreasing slack
- z = value of objective function
  - don't bother writing "maximize" or "subject to"
- introduce one new basic variable to represent slack in each constraint ; pre-existing variables are non-basic

Standard versus Slack Form

max $x_1 + 2x_2 - \frac{1}{2} x_3$

subject to

$$\frac{1}{3} x_1 + x_3 \le 5$$
$$x_1 + x_2 + x_3 \le 100$$
$$x_1 - x_2 \le -3$$
$$x_1, x_2, x_3 \ge 0$$

$$z = x_1 + 2x_2 - \frac{1}{2} x_3$$

$$x_4 = 5 - \frac{1}{3} x_1 - x_3$$
$$x_5 = 100 - x_1 - x_2 - x_3$$
$$x_6 = -3 - x_1 + x_2$$
$$x_1, x_2, x_3, x_4, x_5, x_6 \ge 0$$

basic var's: x_4, x_5, x_6

nonbasic var's: x_1, x_2, x_3

High-level Simplex algorithm

- convert standard form LP to slack form
- find a feasible (probably non-optimal) initial solution
  - if this does not exist, return “infeasible”
- repeat:
  - choose a nonbasic variable x_i with positive coefficient in objective function (increasing x_i increases z)
  - if no such x_i exists, return solution (it’s optimal)
  - increase x_i until some basic variable x_j is decreased to zero (“tighten” the slack until we’re up against a constraint)
  - if none exists, return “unbounded”
  - swap roles: rewrite slack form with x_i as basic variable and x_j as nonbasic variable

(for further details, see CLRS section 29.3 or Levitin section 10.1)

Example

(taken from Levitin pages 347, 352-359)

- Consider the following LP problem in two variables

maximize $3x+5y$

subject to $x+y \leq 4$

$x+3y \leq 6$

$x, y \geq 0$

- The objective function becomes $z = 3x+5y$ so we need to find a solution for (x,y) with the largest value for z
- If a constraint is given as an inequality, it can be replaced by an equivalent equation by adding a slack variable representing the difference between r.h.s and l.h.s; the two inequalities become

$x+y+u = 4$ where $u \geq 0$ and

$x+3y+v=6$ where $v \geq 0$

- We can rewrite the system of constraint equations in matrix form:

$$x \begin{bmatrix}1\\1\end{bmatrix}+y \begin{bmatrix}1\\3\end{bmatrix}+u \begin{bmatrix}1\\0\end{bmatrix}+v \begin{bmatrix}0\\1\end{bmatrix}= \begin{bmatrix}4\\6\end{bmatrix}$$

- The simplex method progresses through a series of adjacent extreme points (basic feasible solutions) with increasing values of the objective function
- Each such point can be represented by a *simplex tableau*, a table storing the information about the basic feasible solution corresponding to the extreme point that achieves the largest value for objective function z
- In general, a simplex tableau for a LP problem in standard form with n unknowns and m linear equations constraints (n ≥ m) has m+1 rows and n+1 columns.

		x	y	u	v	
	u	1	1	1	0	4
←	v	1	3	0	1	6
		-3	-5	0	0	0
			↑			

		x	y	u	v	
	u	1	1	1	0	4
←	v	1	3	0	1	6
		-3	-5	0	0	0
			↑			

- The rows are labeled by the basic variables
- Columns, except the last one, are labeled by the names of the variables: In the example above: nonbasic then basic
- The first m rows and the first n columns contain the coefficients of the corresponding constraint equation
- The last column of the first m rows contain the r.h.s of the constraint equations
- The columns labeled by the basic variables form an m×m identify matrix, in the example above marked by the box
- The last row is called the *objective row* and is initialized by the coefficients of the obj. funct. with the signs reversed (in the first n columns) and the value of the obj. funct. at the initial point (in the last column)

- The objective row is used by the simplex method to check whether the current tableau represents an optimal solution
- An optimal solution is obtained when all the entries of the objective row, except possibly the one in the last column, are nonnegative
- If this is not the case, any of the negative entries indicates a nonbasic variable that can become a basic variable in the next tableau
- If there are several negative entries in the objective row, a commonly used rule is to select the most negative one, i.e. the negative number with the largest absolute value
  - Such a choice yields the largest increase in the obj. funct.'s value per unit of change in a variable's value
- A new basic variable is called the entering variable, while its column is referred to as the pivot column; we mark it with ↑

- We also need to select a departing variable: for each positive entry in the pivot column, compute the θ-ratio between the row's last entry by the entry in the pivot column
- For our example, the ratios are:

$$\theta_u = \frac{4}{1} = 4 \text{ and } \theta_v = \frac{6}{3} = 2$$

- The row with the smallest θ-ratio determines the departing variable, i.e. the variable to become nonbasic; tie may be broken arbitrarily
- The row of the departing variable, called the *pivot row*, is marked with ←

Transforming a Linear Program into “Normal Form”

$$\begin{aligned} &\text{maximize} && 3x+5y \\ &\text{subject to} && x+y \leq 4 \\ & && x+3y \leq 6 \\ & && x, y \geq 0 \end{aligned}$$

Objective function becomes $z = 3x+5y$. Need to find a solution for (x,y) with the largest value for z.

Inequality replaced by an equivalent equation:
add a slack variable representing the difference between R.H.S and L.H.S;

$x+y+u = 4$ with $u \geq 0$
$x+3y+v=6$ with $v \geq 0$

Simplex Tableau

- **LP:**

maximize	$3x+5y$
subject to	$x+y \leq 4$
	$x+3y \leq 6$
	$x, y \geq 0$

Rewrite the system of constraint as equations:

Matrix Form:

$$x\begin{bmatrix}1\\1\end{bmatrix}+y\begin{bmatrix}1\\3\end{bmatrix}+u\begin{bmatrix}1\\0\end{bmatrix}+v\begin{bmatrix}0\\1\end{bmatrix}=\begin{bmatrix}4\\6\end{bmatrix} \text{ with } x, y, u, v \geq 0 \quad \text{and } z - 3x - 5y = 0$$

Non-basic variables:
$x = 0, y = 0$

Basic variables:
$u = 4, v = 6$

Simplex Tableau:

	x	y	u	v	
u	1	1	1	0	4
v	1	3	0	1	6
	-3	-5	0	0	0

Corresponding z-value:
$z = 0$

Pivot

	x	y	u	v	
u	1	1	1	0	4
v	1	3	0	1	6
	-3	-5	0	0	0

Current solution:
x = 0, y = 0, u = 4, v = 6

most improvement here:
y can be increased by
min of $\frac{4}{1}$ and $\frac{6}{3} = 2$

Divide row by 3

	x	y	u	v		
u	1	1	1	0	4	u
v		1	0		2	y
	-3	-5	0	0	0	

↑

Subtract 2nd row from 1st
Subtract 2nd row times -5 from 3rd row

x	y	u	v	
	0	1		2
	1	0		2
	0	0		10

Corresponds to solution (0,2,2,0)

Another pivot, and we are optimal

	x	y	u	v	
u		0	1		2
y		1	0		2
		0	0		10

	x	y	u	v	
x		0			3
y		1			1
	0	0	2	1	14

all values are non-negative

$x_1 = 3$, $x_2 = 1$

maximum is 14

Simplex method summarized

(taken from Levitin, pages 356-357)

Step 0 *Initialization* Present a given linear programming problem in standard form and set up an initial tableau with nonnegative entries in the rightmost column and m other columns composing the m × m identity matrix. (Entries in the objective row are to be disregarded in verifying these requirements.) These m columns define the basic variables of the initial basic feasible solution, used as the labels of the tableau's rows.

Step 1 *Optimality test* If all the entries in the objective row (except, possibly, the one in the rightmost column, which represents the value of the objective function) are nonnegative—stop: the tableau represents an optimal solution whose basic variables' values are in the rightmost column and the remaining, nonbasic variables' values are zeros.

Step 2 *Finding the entering variable* Select a negative entry from among the first n elements of the objective row. (A commonly used rule is to select the negative entry with the largest absolute value, with ties broken arbitrarily.) Mark its column to indicate the entering variable and the pivot column.

Step 3 *Finding the departing variable* For each positive entry in the pivot column, calculate the θ-ratio by dividing that row's entry in the rightmost column by its entry in the pivot column. (If all the entries in the pivot column are negative or zero, the problem is unbounded—stop.) Find the row with the smallest θ -ratio (ties may be broken arbitrarily) and mark this row to indicate the departing variable and the pivot row.

Step 4 *Forming the next tableau* Divide all the entries in the pivot row by its entry in the pivot column. Subtract from each of the other rows, including the objective row, the new pivot row multiplied by the entry in the pivot column of the row in question. (This will make all the entries in the pivot column 0's except for 1 in the pivot row.) Replace the label of the pivot row by the variable's name of the pivot column and go back to Step 1.

For further details, see CLRS section 29.3

Formulating Business Logic

- you need to figure out how to model all these in parts of an LP
  - variables
  - objective function
  - constraints
- every concept mentioned in the scenario needs to be modeled in the LP somehow
  - only works when the objective is a linear function (not squared, exponential, etc.)

Example 1: Plastic Recycling

- Suppose Orange County has the following options for disposing of plastic waste:

Method	Carbon/ton	Dollars/ton
Local incinerator	2.9	800
Overseas incinerator	3.2	600
Thermal recycling	.5	1200
Landfill	.3	1400

- December is expected to produce 700 tons of plastic and is required by law to emit no more than 1400 tons of carbon. The landfill can accommodate up to 100 tons per month and the other methods are effectively unlimited. The County Supervisors want you to minimize the cost of plastic disposal.

Formulate as LP problem

1. Variables: one variable for each method

   loc = tons incinerated locally
   ove = tons incinerated overseas
   rec = tons recycled
   lan = tons sent to landfill

2. Objective function: “minimize cost of plastic disposal”

   minimize 800*loc* + 600 *ove* + 1200 *rec* + 1400*lan*

Cont.

3. Constraints:

- "produce 700 tons of plastic":
  $loc + ove + rec + lan = 700$

- "no more than 1400 tons of carbon":
  $2.9loc + 3.2ove + .5rec + .3lan \leq 1400$

- "landfill can accommodate up to 100 tons":
  $lan \leq 100$

- use critical thinking, common sense to model implicit rules:
  $loc, ove, rec, lan \geq 0$

Cont.

- general-form LP:

minimize 800loc + 600ove + 1200rec + 1400lan

subject to $loc + ove + rec + lan = 700$

$2.9loc + 3.2ove + .5rec + .3lan \leq 1400$

$lan \leq 100$

$loc, ove, rec, lan \geq 0$

- Interpret results:
  - infeasible: never happens because there exists at least one feasible solution: rec = 700, loc = ove = lan = 0
  - unbounded: never happens because the non-negativity constraints lower-bound the objective at 0; it can't approach $-\infty$
  - solution: Tell the Supervisors to incinerate *loc* tons locally, incinerate *ove* tons overseas, recycle *rec* tons, and landfill *lan* tons.

Single-Source Shortest Paths

(taken from https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/10-linear-programming-problem.pdf

1. Variables: for each vertex $v \in V$, create an LP variable $d_v \equiv$ total weight of shortest path $s \rightsquigarrow v$ in G
2. Objective function: hold that thought
3. Constraints:
   - Recall that in Bellman-Ford and Dijkstra, we compute $d_v = \min_{x \in V, \{x,v\} \in E} d_x + w(x, v)$
   - LP can't express "min" but we can say $d_v \leq$ every candidate: $d_v \leq d_x + w(\{x,v\})\ \forall x \in V, \{x,v\} \in E$
   - $d_s = 0$ because *s* is the source vertex

SSSP as LP problem

2. Objective function: for some node (sink) t, we choose to maximize d_t

- not minimize d_t
- that would allow LP to just set every $d_v = 0$; no notion of edges adding to path weights
- maximizing d_t forces LP to set each d_v to the $d_x + w(\{x, v\})$ that is the minimum
- now, when you use an edge, you add its weight

Cont.

4. General-form LP:

given $G = (V, E)$, source s, sink t

create variable d_v $\forall v \in V$

maximize d_t

subject to

$$d_v \leq d_x + w(x,v) \quad \forall \{x,v\} \in E$$

$$d_s = 0$$

5. Interpreting result:

- infeasible: never happens because setting every $d_v = 0$ is always feasible
- unbounded: $d_t = \infty$ means that t is unreachable from s
- solution:
  - need to identify path (list of vertices) $s \rightsquigarrow t$
  - i.e. identify edges defining the shortest path and d_t
  - find the constraints where $d_v = d_x + w(x,v)$ (not ≤); these vertices x,v are part of the shortest path
  - use BFS or DFS, limited to on-path vertices, to order these vertices

Formulating Max Flow

- **Flow** – a function $f : V \times V \rightarrow \boldsymbol{R}$
  - *Capacity constraint*: For all $u, v \in V$: $f(u,v) \leq c(u,v)$
  - *Skew symmetry*: For all $u, v \in V$: $f(u,v) = -f(v,u)$
- *Flow conservation:* (flow-in) = (flow-out), except for source and sink; formally for all $u \in V - \{s, t\}$:

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

- Value |f| = net flow out of source = net flow into sink

$$|f| = \sum_{v \in V} f(s,v) = \sum_{v \in V} f(u,t)$$

Formulating Max Flow as LP problem

1. Variables:
for each directed edge $(u,v) \in E$, create variable $f_{u,v} \equiv$ flow from u to v

2. Objective function: want to maximize $|f| = \sum_{v \in V} f(s,v)$

3. Constraints: still need to model the capacity constraint and flow conservation rules of max-flow

- capacity constraints:

$f_{u,v} \geq 0 \ \forall u,v \in V$
$f_{u,v} \leq c(u,v) \ \forall (u,v) \in E$

(these are separate, not "$0 \leq f_{u,v} \leq c(u,v)$," to match general LP format)

- flow conservation:
  $\sum_{v \in V} f(v,u) = \sum_{v \in V} f(u,v)$ becomes
  $\sum_{v \in V} f(v,u) - \sum_{v \in V} f(u,v) = 0$ for all $u \in V - \{s, t\}$

Cont.

4. General-form LP: given $G = (V, E)$, source s, sink t

create variable $f_{u,v}$ $\forall u,v \in V$

maximize $\sum_{v \in V} f(s, v)$

subject to

$f_{u,v} \geq 0 \ \forall u,v \in V$

$f_{u,v} \leq c(u,v) \ \forall (u,v) \in E$

$\sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v) = 0$ for all $u \in V - \{s, t\}$

5. Interpreting result:

- infeasible: never happens because setting every $f_{u,v} = 0$ is a feasible solution
- unbounded: never happens because the objective function $\sum_{v \in V} f(s, v)$ is clearly $\leq \sum_{v \in V} c(s, v)$ which is finite

Cont.

- solution: define flow function

$$f(u,v) = \begin{cases} f(u,v) & if\ (u,v) \in E \\ 0 & otherwise \end{cases}$$

Randomized Algorithms

- Advantages over deterministic algorithms:
  - Execution time or space requirement of a randomized algorithm is smaller than that of the best deterministic algorithm that we know of for the same problem
  - It performs well with high probability on most of the input instances (but not an all)
  - Most randomized algorithms are extremely simple to understand and to implement
- Are categorized according to which aspect is uncertain; the categories are named after cities that are famous as gambling destinations: Las Vegas and Monte Carlo

- A *Las Vegas algorithm* has uncertain efficiency but produces output whose correctness is certain.
- Example: frequency of on-time arrivals of commercial flights. The outcome is certain (travel from Los Angeles to Chicago) but the efficiency (time) is uncertain
- We call this a Las Vegas process.

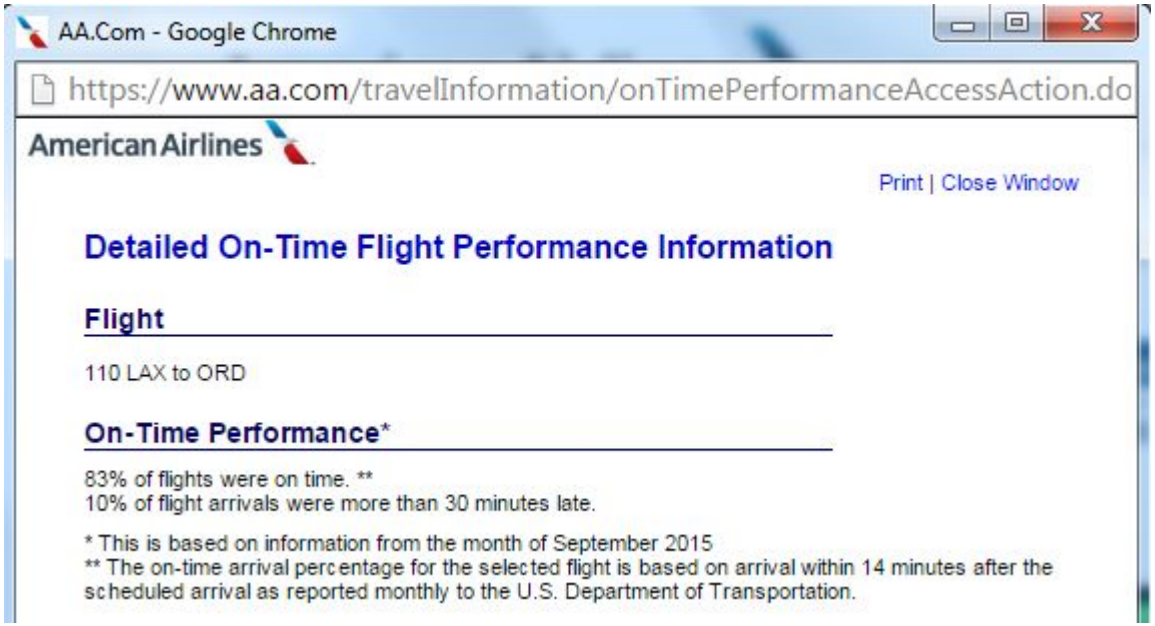


Figure taken from www.aa.com

- A *Monte Carlo algorithm* has certain efficiency but produces output whose correctness is uncertain.
- Example: SNA airport landing curfew after 11 PM; if landing after 11 PM, then re-directed to LAX or other nearby airport
- Model this situation mathematically: any flight scheduled to land around 11 PM lands at John Wayne with probability 0.95 (say), and lands at the nearby LAX airport with probability 0.05

Figure taken from http://www.ocair.com/generalaviation/noise/

General Aviation Noise Limits		
Daytime Hours	**Nighttime Hours**	
NMS 1S = 102.5 dB SENEL	NMS 1S = 87.5 dB SENEL	
NMS 2S = 101.8 dB SENEL	NMS 2S = 87.6 dB SENEL	
NMS 3S = 101.1 dB SENEL	NMS 3S = 86.7 dB SENEL	
	NMS 4S = 86.7 dB SENEL	
	NMS 5S = 86.7 dB SENEL	
	NMS 6S = 86.7 dB SENEL	
	NMS 7S = 86.7 dB SENEL	
	NMS 8N = 86.9 dB SENEL	
	NMS 9N = 86.9 dB SENEL	
	NMS 10N = 86.9 dB SENEL	
Daytime hours are (local time):		
	Departures	**Arrivals**
Monday - Saturday	0700 to 2200	0700 to 2300
Sunday	0800 to 2200	0800 to 2300
All other hours are considered **nighttime** hours. Compliance is determined by the clock at each NMS.		

- In most practical applications, we take correctness over efficiency
- Example: text editor problem
- You need a text editor that saves the work (currently in RAM) to a file on a disk (external memory)
- A Las Vegas algorithm: would always work properly, but once in a while might take more time than usual (unbounded running time)
- A Monte Carlo algorithm: the saving time is bounded, but every once in a while, it might corrupt some or all of the text
- What would be your preference?

Problem Complexity

- Some problems cannot be solved in polynomial time: traveling salesman problem
- Some problems cannot be solved at all: halting problem
- Separate problems into three broad categories
  - Problems that can be solved in polynomial time; the technical term for this category is *P* (polynomial).
  - Problems that can be solved only by extremely slow algorithms; the term for this category is *NP* (nondeterministic polynomial), and the hardest of these problems are called *NP-complete*.
  - Problems that cannot be solved by algorithms; the term for this category is *undecidable*.
- P and NP: decidable problems

P Class Problems

- An algorithm is *polynomial* if its time complexity is $O(n^k)$ for some constant $k \geq 0$.
- If its time complexity can be upper-bounded by a polynomial, then we consider the algorithm to be polynomial. Example: O(1), $O(\sqrt{n}) = O(n^{0.5})$, $O(\log n)$
- The complexity class P is the set of all problems that may be solved by polynomial algorithms:
- P = { X | there exists a polynomial algorithm that solves X}

Proving that a problem is in P

- To prove that a problem is in P we must find a polynomial time algorithm that solves it.
- The following problems are in P:
    1. summation
    2. sequential search
    3. sorting
    4. binary search
    5. minimum spanning tree
    6. single source shortest paths
    7. median finding
    8. set intersection

    and others
- Circuit satisfaction and traveling salesperson do not have a polynomial time algorithm, so we cannot say that they are in P. But we cannot say that they are not in P either. Just because we tried to design a polynomial algorithm for these problems and failed to do so, does not mean that no such algorithm exists.

NP Class Problems

- If an algorithm's time complexity is exponential e.g. $O(2^n)$, or factorial e.g. $O(n!)$, the algorithm is not polynomial.
- NP is the set of problems for which verifier algorithms have polynomial time and space complexity.
- A verifier algorithm is an algorithm that verifies whether a given input is a correct output to a problem.
- For example: the subset sum problem[1]
- Assume that we are given some integers, such as {−7, −3, −2, 5, 8}, and we wish to know whether some of these integers sum up to zero.

[1] https://en.wikipedia.org/wiki/NP_(complexity)

A problem E is said *to be in NP* if there exists a verifier algorithm `verify_E` for the problem such that:

- `verify_E` takes as input a pair `(instance, certificate)`
  - `instance` is some instance of the problem E
  - `certificate` is some piece of data (called *certificate*)
- `verify_E` checks whether `certificate` is a correct output for `instance` of problem E in ***polynomial time***
- The output of `verify_E` is "yes" or "no"
- The output of `verify_E` is "yes" if `certificate` is a correct output for `instance` of problem E
- The output of `verify_E` is "no" if `certificate` is an incorrect output for `instance` of problem E

Proving that a problem is in NP

- We may use the following process to prove that a problem X is in NP:

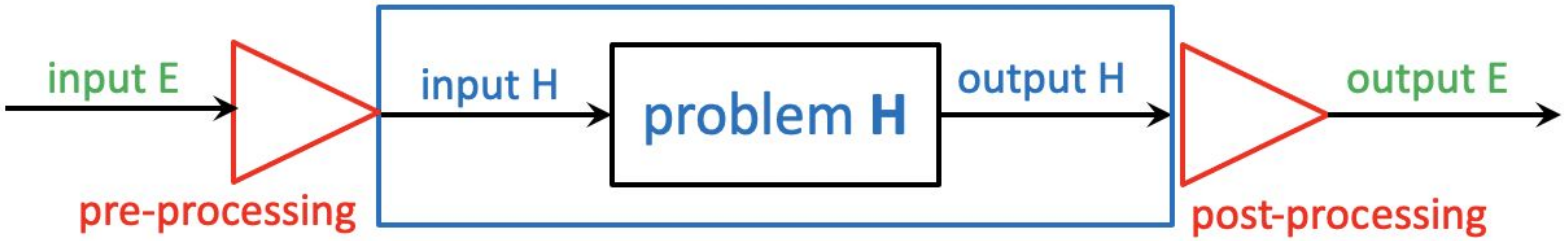
  1. Define the data type of a candidate solution to the problem.
  2. Prove that each candidate fits in polynomial space.
  3. Design a verification algorithm for the problem.
  4. Prove that the verification algorithm is polynomial.
  5. Conclude that $X \in NP$.

Decidable vs Undecidable Problems

- Problem X is decidable if there exists an algorithm that solves X, or is undecidable if there does not exist an algorithm that solves X.
- Every problem in P is decidable.
- The circuit satisfaction and traveling salesperson problems are decidable.
- Every problem in NP is decidable.

NP-Hard Problems

- Problem E *reduces* to problem H if there exists an algorithm that solves E by pre-processing its input, solving H, then post-processing that output.



- Problem E is *polynomial time reducible* to problem H, written $E \xrightarrow{P} H$, if there exists an algorithm that solves E by reduction to an algorithm that solves H, where any pre-processing and post-processing takes polynomial time.
- The algorithm that solves H does not need to be polynomial.

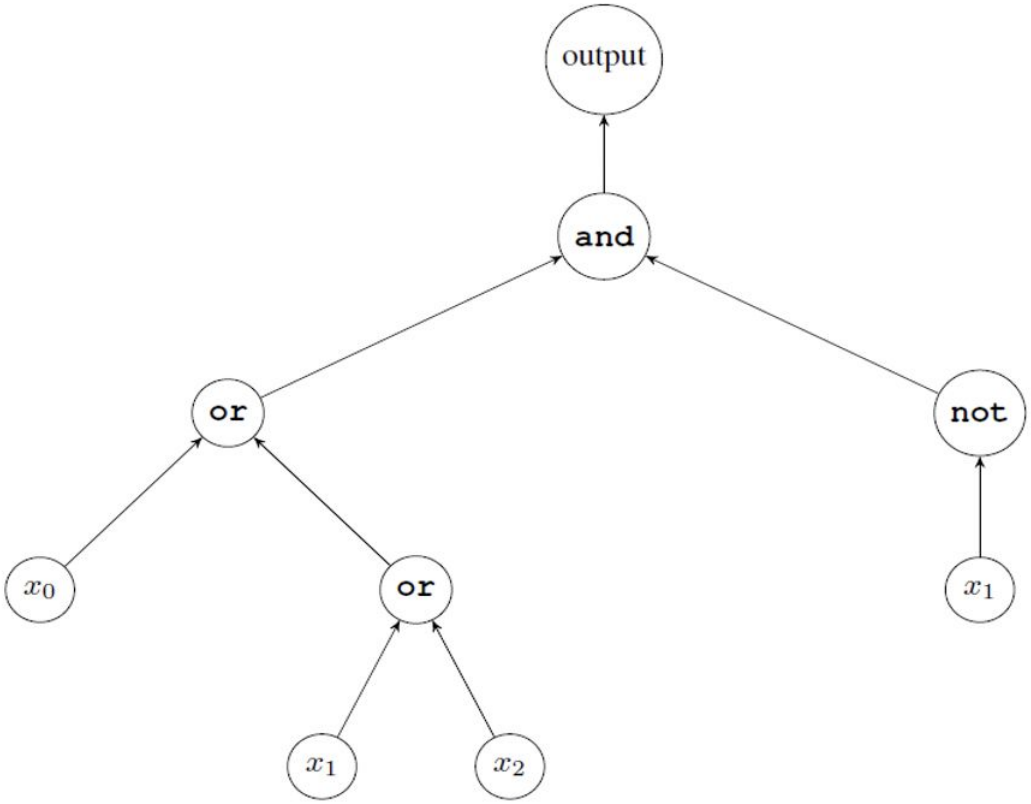
- Problem H is NP-hard if, for any problem $E \in NP, E \xrightarrow{P} H$.
- In other words, any problem in NP can be solved by reducing it in polynomial time to H.
- The class NP includes all the problems that can be solved in polynomial time (class P) and problems for which, given a pair `(instance, certificate)`, checking whether `certificate` is a correct output for `instance` of that problem can be done in polynomial time (example: Boolean circuit satisfaction, traveling salesman)

NP-complete

- Problem H is NP-complete if
  1. H $\in NP$, and
  2. H is NP-hard.
- Example:
- *Theorem (Cook-Levin). The circuit satisfaction problem is NP-complete.*

Boolean Circuit Evaluation

- A *Boolean circuit* C = (T, X) is a directed tree T = (V, E) with
  - a set of k indexed variables $X = \{x_0, x_1, \cdots, x_{k-1}\}$, where k < n and
  - n = |V| vertices



- A *Boolean circuit assignment* A over *k* variables is a list of *k* Boolean True or False values $A = \langle a_0, a_1, \cdots, a_{k-1} \rangle$.
- $A = \langle True, False, True \rangle$ is a Boolean assignment that evaluates to True.
- For $A' = \langle True, True, True \rangle$ it evaluates to False.

Circuit Satisfaction

- The circuit satisfaction problem is:

input: a Boolean circuit C with *k* variables and *n* vertices

output: a Boolean assignment A that satisfies C, or None if no such assignment exists.

- *Theorem (Cook-Levin). The circuit satisfaction problem is NP-complete.*
- One needs to show that
  - Circuit Satisfaction $\in \mathrm{NP}$, and
  - Circuit Satisfaction is NP-hard

Circuit Satisfaction $\in$ NP

- A candidate solution is an assignment of a Boolean value to each of the *k* variables in the circuit.
- A circuit with *n* vertices has at most *n* variables, so $k \leq n$ and $O(k) \subseteq O(n)$.
- An assignment may be verified by evaluating the input circuit and testing whether the output is True. So each candidate may be verified in $O(n)$ time (polynomial).
- Just a note: A Boolean circuit may be evaluated in $O(n^2)$

<u>Circuit Satisfaction is NP-hard</u>: for any problem $E \in NP, E \xrightarrow{P}$ Circuit Satisfaction.

- Since $E \in NP$, there must be some verifier algorithm `verify_E` that takes as input a pair `(instance, certificate)` and it outputs "yes" or "no" in polynomial time if `certificate` is, respectively is not, a correct output for `instance` of problem E
- Build a massive Boolean circuit that emulates the execution of `verify_E`: the circuit shall have one variable for each bit of input to `verify_E`
- The circuit's output shall be equivalent to that computed by `verify_E`.

Proving NP-completeness

- A problem is NP-complete if it is at least as hard as any other problem in NP.
- Lemma: If $H \in NP$, E is NP-complete and $E \xrightarrow{P}$H, then H is NP-complete.
- Steps to show that a problem is NP-complete:
  1. H $\in NP$, and
  2. H is NP-hard.

Is P=NP?

- $X \in P$ implies $X \in NP$. Thus $P \subseteq NP$.
- We do not presently know whether $P \subset NP$ or $P = NP$.

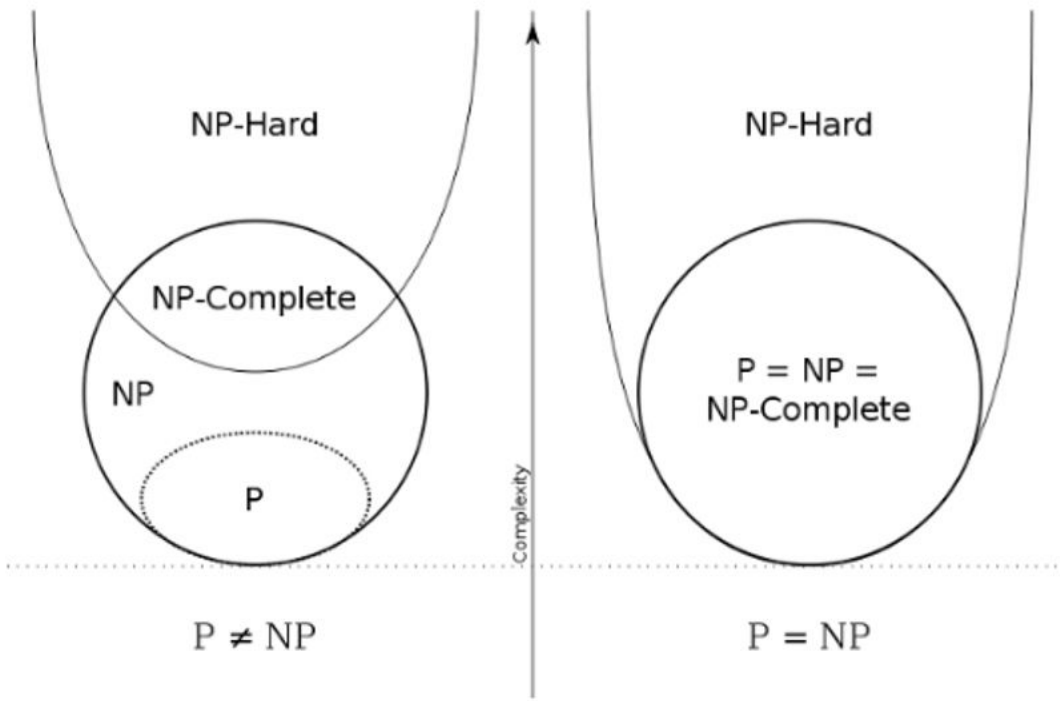


Image taken from https://medium.com/the-physics-arxiv-blog/the-astounding-link-between-the-p-np-problem-and-the-quantum-nature-of-universe-7ef5eea6fd7a#.r1ix86i3g

Approximation

(some slides taken from https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/13-approximation.pdf)

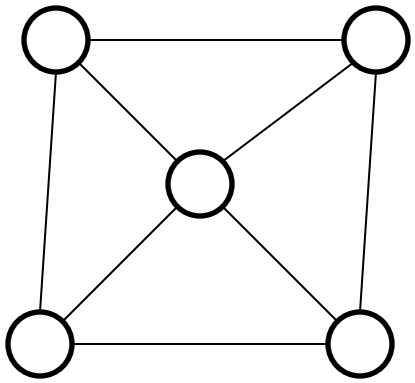
Approximation

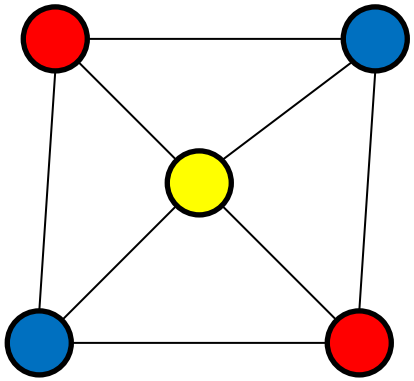
- output is proven nearly-optimal but not necessarily truly optimal.
- suitable for use cases where approximate solutions are adequate
- need to rewrite problem definition
  - every optimization problem has corresponding approximation problems; but these are distinct problems
- Example: optimal vs. approximate problems

Example 1: graph coloring

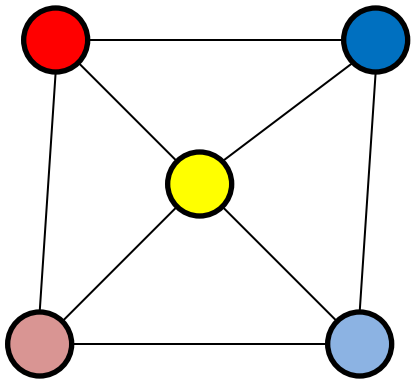
- Optimization problem: *graph coloring*
  input: connected graph $G = (V, E)$
  output: coloring c using k colors, where each vertex $v \in V$ is assigned color $c(v) \in \{1, \ldots k\}$, no pair of adjacent vertices are assigned the same color, and the number of colors k is minimal.

- Approximation problem: *3-approximate graph coloring*
  input: connected graph $G = (V, E)$
  output: coloring c using k colors, where each vertex $v \in V$ is assigned color $c(v) \in \{1, \ldots k\}$, no pair of adjacent vertices are assigned the same color, and the number of colors k^* satisfied $k \leq 3k^*$, where k^* is the fewest colors possible for G





Optimal number of colors = 3



Approximate number of colors = 5 ≤ 3*3=9

Strategies to solve NP-hard problems

- when n is small enough, just use the exponential-time algorithm
- Using heuristics: no proof of solution quality, but nonetheless sometimes good enough:
  - machine learning algorithms (also, in M.L. humans don't need to precisely define what counts as "correct")
  - fast heuristic algorithms
  - Monte Carlo algorithms
- Approximation
  - pros: provable solution quality, often fast
  - con: human needs to design and analyze algorithm for each specific problem

Approximation ratio

- Approximation ratio $\rho(n)$: ratio between quality of algorithm's output and optimal output; smaller is better
  - for a *maximization* problem: if optimal quality is C^* and alg. produces quality C, by definition $C^* \geq C$, and define $\rho(n) = C^*/C$
  - for a *minimization* problem: if optimal quality is C^* and alg. produces quality C , by definition $C^* \leq C$ and define $\rho(n) = C/C^*$. For example, for the graph coloring problem the approximation ratio is 3.

Fixed Approximation Ratios

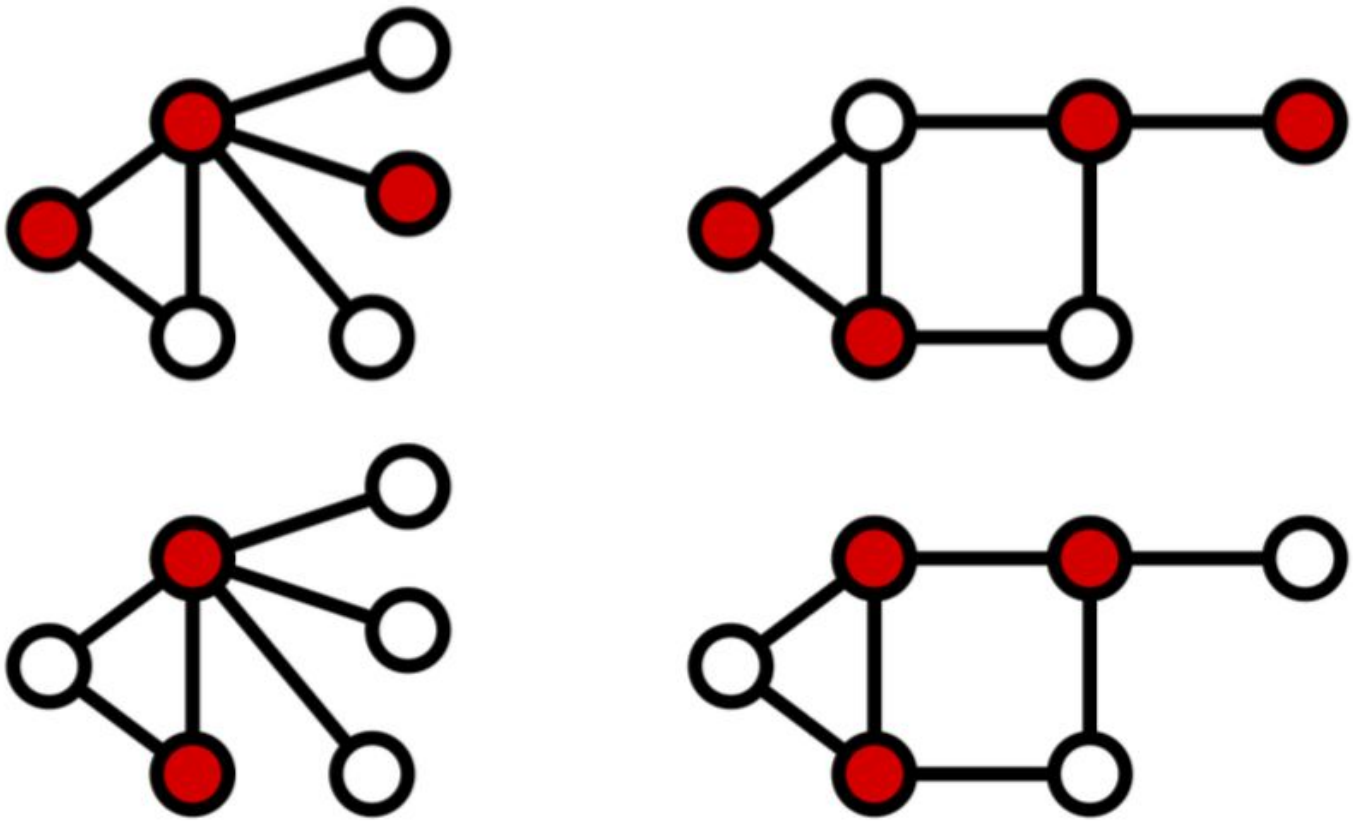
- In general, better (smaller) ratios require slower algorithms.
- Note: 1-approximation algorithms produce optimal solutions.
- Deriving a different quality-vs.-time trade-off requires designing an entirely different approximation algorithm.

Vertex Cover Problem

- *vertex cover problem* (https://en.wikipedia.org/wiki/Vertex_cover)
  input: undirected graph $G = (V , E)$
  output: set of vertices $C \subseteq V$, of minimal size $|C|$, such that every edge in E is incident on at least one vertex in C
- *2-approximate vertex cover problem*
  input: undirected graph $G = (V , E)$
  output: set of vertices $C \subseteq V$, such that every edge in E is incident on at least one vertex in C, and $|C| \leq 2|C^*|$ where C^* is a minimal vertex cover for G

Vertex Cover Example

Images credit: Wikipedia user Miym, CC BY-SA 3.0, https://commons.wikimedia.org/wiki/File:Vertex-cover.svg, https://commons.wikimedia.org/wiki/File:Minimum- vertex- cover.svg



Solving Vertex Cover problem

- vertex cover is NP-complete
- baseline algorithm:
  - exhaustive search
  - for each subset C of vertices, check whether every edge has an endpoint in C
  - return the smallest C that is a valid cover
  - $\Theta(2^n m)$ time, exponential, slow
- goal of a 2-approximate vertex cover algorithm:
  - get a decent (though imperfect) cover much faster

Greedy-based 2-approximation algorithm

- Every edge e=(u,v) needs both $u \in C$ and $v \in C$
- so grab an edge e = (u,v) and include u and v in C
- every other edge touching u or v is now covered, so eliminate them
- continue until every edge is either grabbed or eliminated
- good: definitely finds a correct cover C
- bad: depending on the order of the “grabs”, heuristic can get tricked into picking sub-optimal vertices

2-Approximate Vertex Cover Pseudocode

```
function APPROX-VERTEX-COVER(G=(V,E))
    C=∅
    T=E // T =set of edges that may be selected
    while T≠ ∅ do
        Let e = (u,v) be an arbitrary edge in T
        C =C ∪{u,v}
    Remove from T any edge f that is incident on u or v
    end while
    return C
end function
```

Efficiency Analysis: O(m + n) time (assuming efficient data structures for G, T, and C)

Vertex Cover Performance Ratio

Lemma: APPROX-VERTEX-COVER is a 2-approximation algorithm.
(**Need** to show that for any G, $|C| \leq 2|C^*|$.)

Proof sketch:
Let A be the set of edges chosen inside the while loop
Compare $|C|, |C^*|$ with $|A|$:
Part (1): we show that $|C^*| \geq |A|$:
C^* is a vertex cover, so for every edge $(u,v) \in A$, we must have
$u \in C^*$ and/or $v \in C^*$
The step "Remove from T" guarantees that, after (u,v) is chosen, no other edge incident on u or v will be chosen and added to A
$\Rightarrow$ each $x \in C^*$ covers exactly one edge in A $\Rightarrow |C^*| \geq |A|$

Contd.

Part (2): we compare $|C|$ and $|A|$

The step $C = C \cup \{u,v\}$ step inserts 2 vertices into C

Due to the same "Remove from T" logic, neither u nor v was already in C

$\Rightarrow |C| = 2|A|$ (note exact equality)

By combining the results in parts (1) and (2), QED

Comments

- Note that us analysts do not know concretely which vertices are in C*
- the algorithm certainly doesn't know what C* is, either
- all we do know is that, due to the definition of vertex cover, and the logic of our algorithm,
  # vertices in optimal cover ≥ # iterations while loop
- and, due to algorithm logic,
  # iterations while loop = # vertices chosen for approx. cover
- in general, to prove an approx. ratio, we need to

1. to bound quality of arbitrary, opaque optimal solution; and
2. bound quality of approx. solution the same way

The Traveling Salesperson Problem (TSP)

- The traveling salesperson (TSP) problem is:

input: a graph G = (V,E) where each edge e $\in E$ has a numeric weight w_e

output: a Hamiltonian cycle in G of minimum total weight, or None if no such cycle exists.

- A *Hamiltonian path*: a path in G such that every node v $\in V$ appears exactly once.
- A cycle: a path C in which the first and last vertex repeats $C = \langle p_0, p_1, \cdots, p_{k-1}, p_0 \rangle$
- A *Hamiltonian cycle*: a cycle $C = \langle p_0, p_1, \cdots, p_{k-1}, p_0 \rangle$ in G such that $\langle p_0, p_1, \cdots, p_{k-1} \rangle$ is a Hamiltonian path; in other words, every node v $\in V$ appears exactly once in $\langle p_0, p_1, \cdots, p_{k-1} \rangle$.

- Use exhaustive optimization:
  - Generating every Hamiltonian path in G and append the first vertex to the path to make it a cycle
  - A Hamiltonian path is a permutations of G's vertices V.
  - So, generate all permutations of V
  - For each permutation, let the extended sequence be the permutation with the first element appended (at the end)
  - Check whether the extended sequence is a cycle: it is a path (i.e. there is an edge between any two adjacent vertices)
  - If it forms a path, we have a Hamiltonian cycle; a candidate solution
  - A cycle is optimal if its total weight is minimal.
- A complete graph of n nodes will have n! Hamiltonian cycles.

TSP

- Bad news:
  - TSP is NP-complete: if P ≠ NP, no polynomial-time optimization algorithm exists for TSP
  - TSP is also APX-complete, i.e if P ≠ NP, no PTAS: there is no constant factor approximation algorithm for TSP

Lemma: There is no α-approximation algorithm for Traveling Salesman Problem where α is a constant.

- Approximation algorithms: The key to designing approximation algorithm is to obtain a bound on the optimal value OPT.
  - In the case of TSP, the minimum spanning tree gives a lower bound on OPT

Metric TSP

- In the *metric* TSP, edge costs have to satisfy the triangle inequality, i.e. for any three vertices x, y, z, $c_{xz} \leq c_{xy} + c_{yz}$
- Metric TSP is still NP-complete but not APX-complete
- There is a 2-approximation algorithm

Lemma: The cost of a Minimum Spanning Tree (MST) is no greater than the cost of an optimal tour.

- Minimum spanning tree features:
  - minimizes weight of chosen edges
  - connects all vertices
  - can be computed fast

Summary

Vertex cover:

- NP-complete
- exact exhaustive search alg. takes $\Theta(2^n m)$ time
- a 2-approximate alg. takes $\Theta(n + m)$ time

Metric TSP:

- NP-complete
- exact exhaustive search alg. takes $\Theta(n!)$
- a 2-approximate algorithm takes $O(m + n \log n)$ time