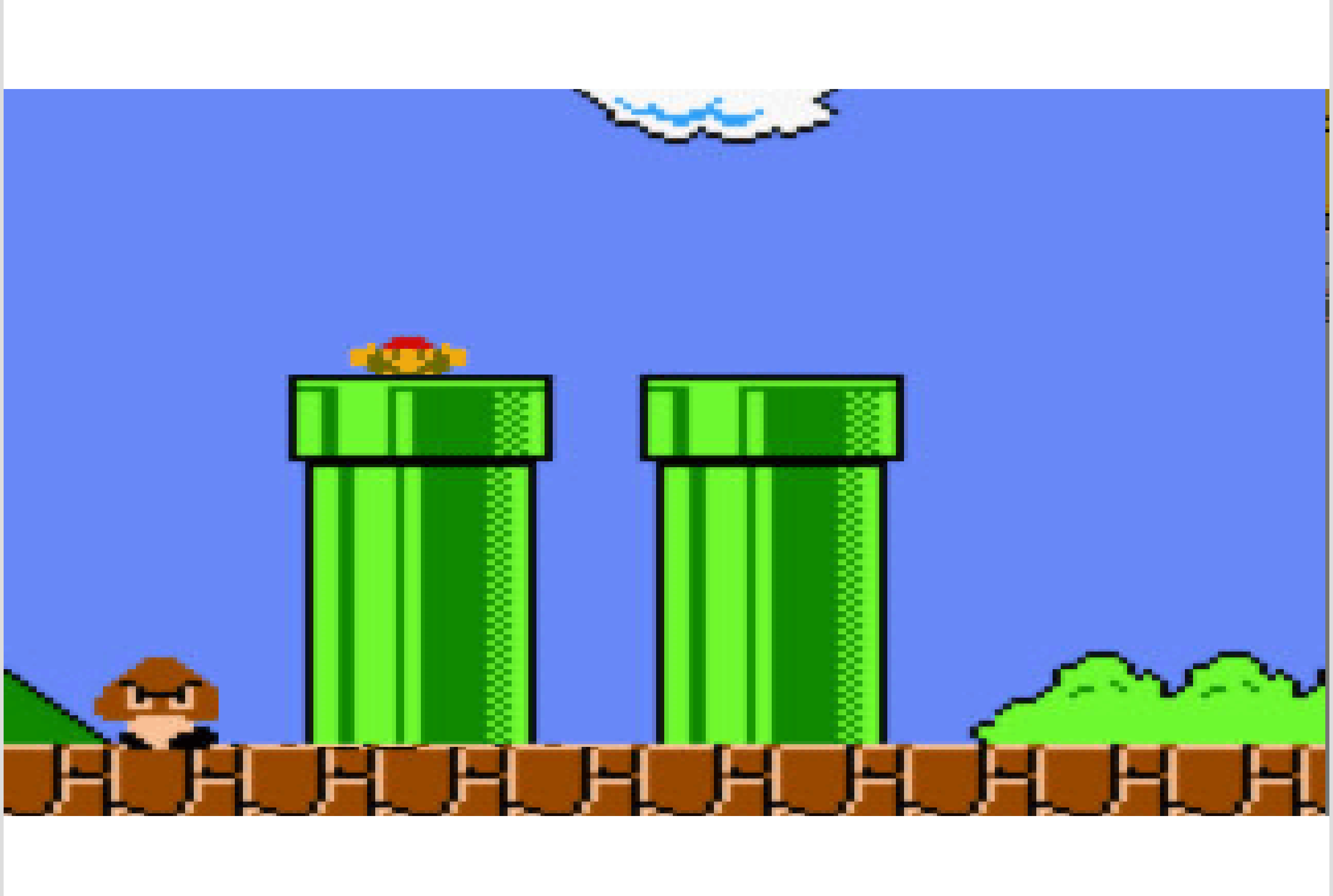
Network Flow



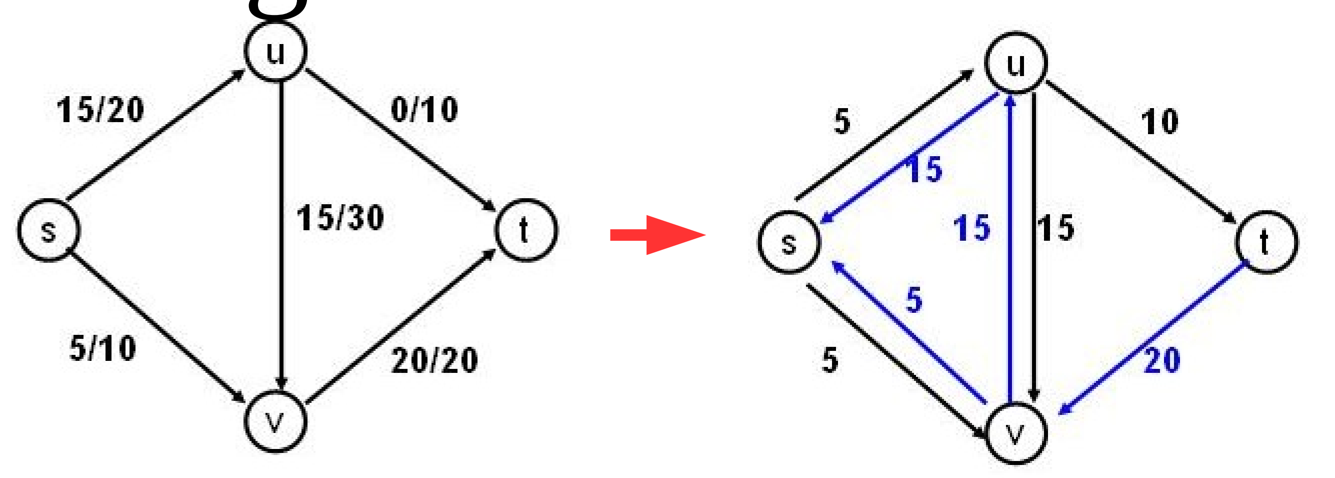
Ford-Fulkerson

What is a residual graph?

Forward edges = capacity left

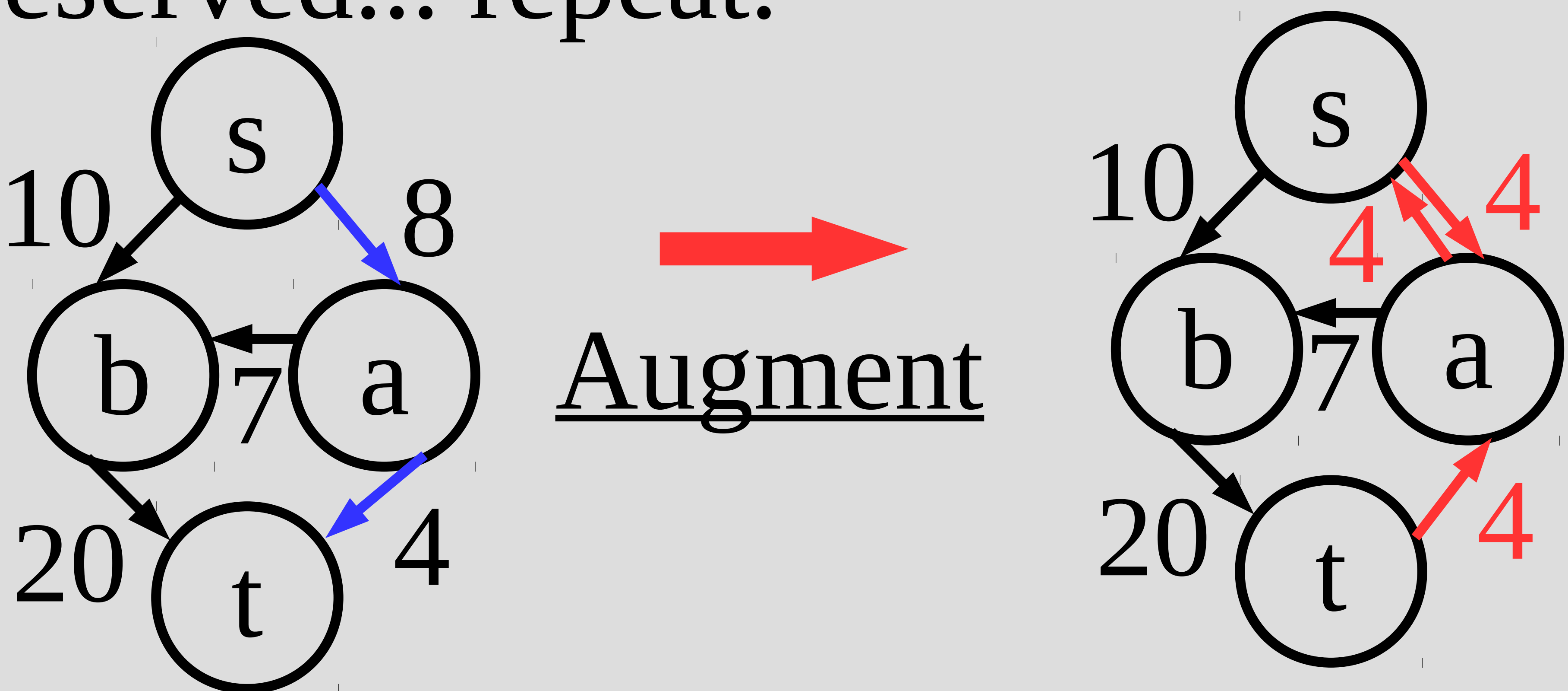
Back edges = flow

Original Residual



Ford-Fulkerson

Idea: Find a way to add some flow, modify graph to show this flow reserved... repeat.

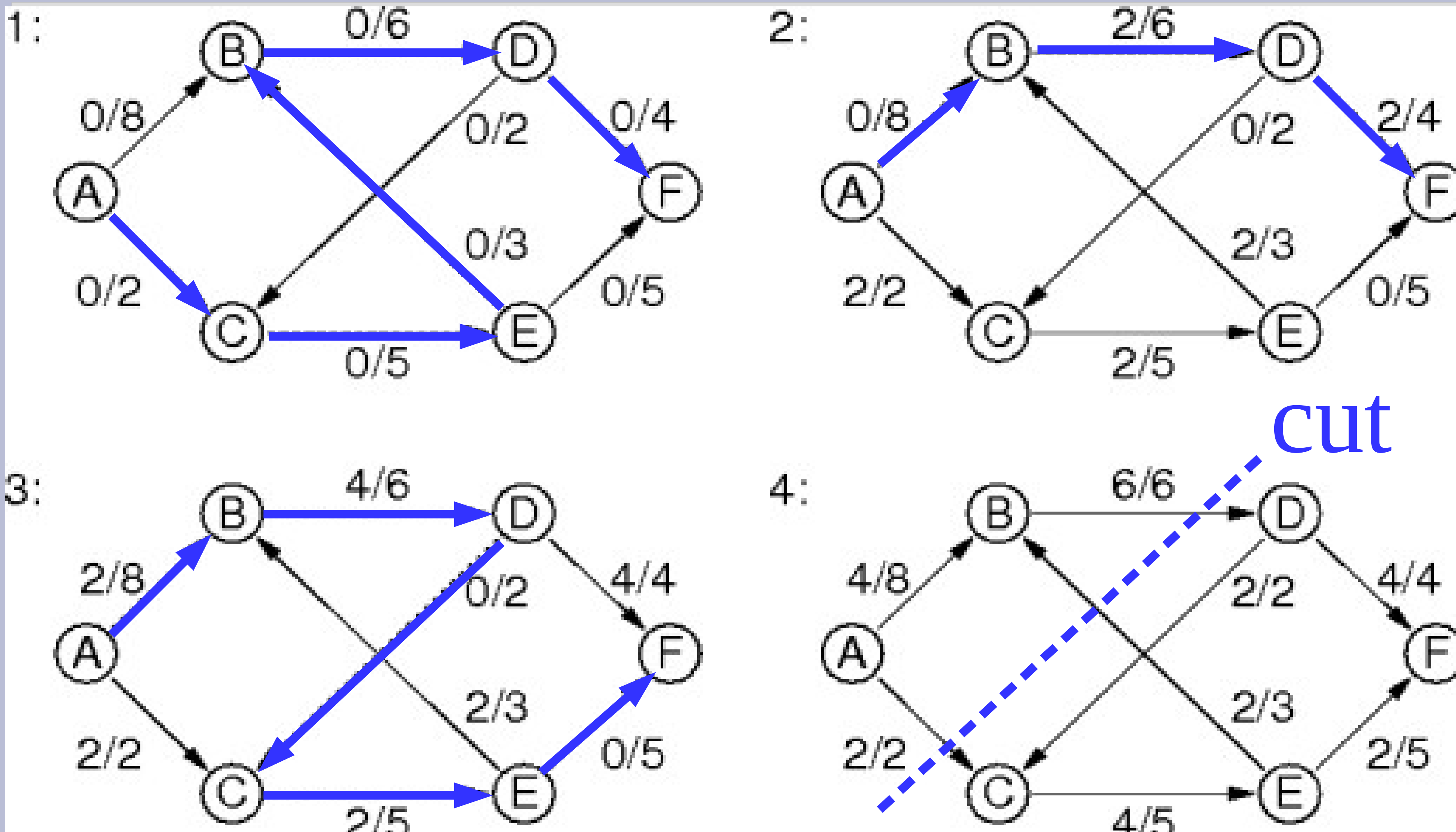


Ford-Fulkerson

Ford-Fulkerson(G, s, t)
initialize network flow to 0
while (exists path from s to t)
   augment flow, f, in G along path
return f

(Note: “augment flow” means add this flow to network)

Ford-Fulkerson



Ford-Fulkerson

Subscript “f” denotes residual (or modified graph)

G_f = residual graph

E_f = residual edges

c_f = residual capacity

$c_f(u,v) = c(u,v) - f(u,v)$

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

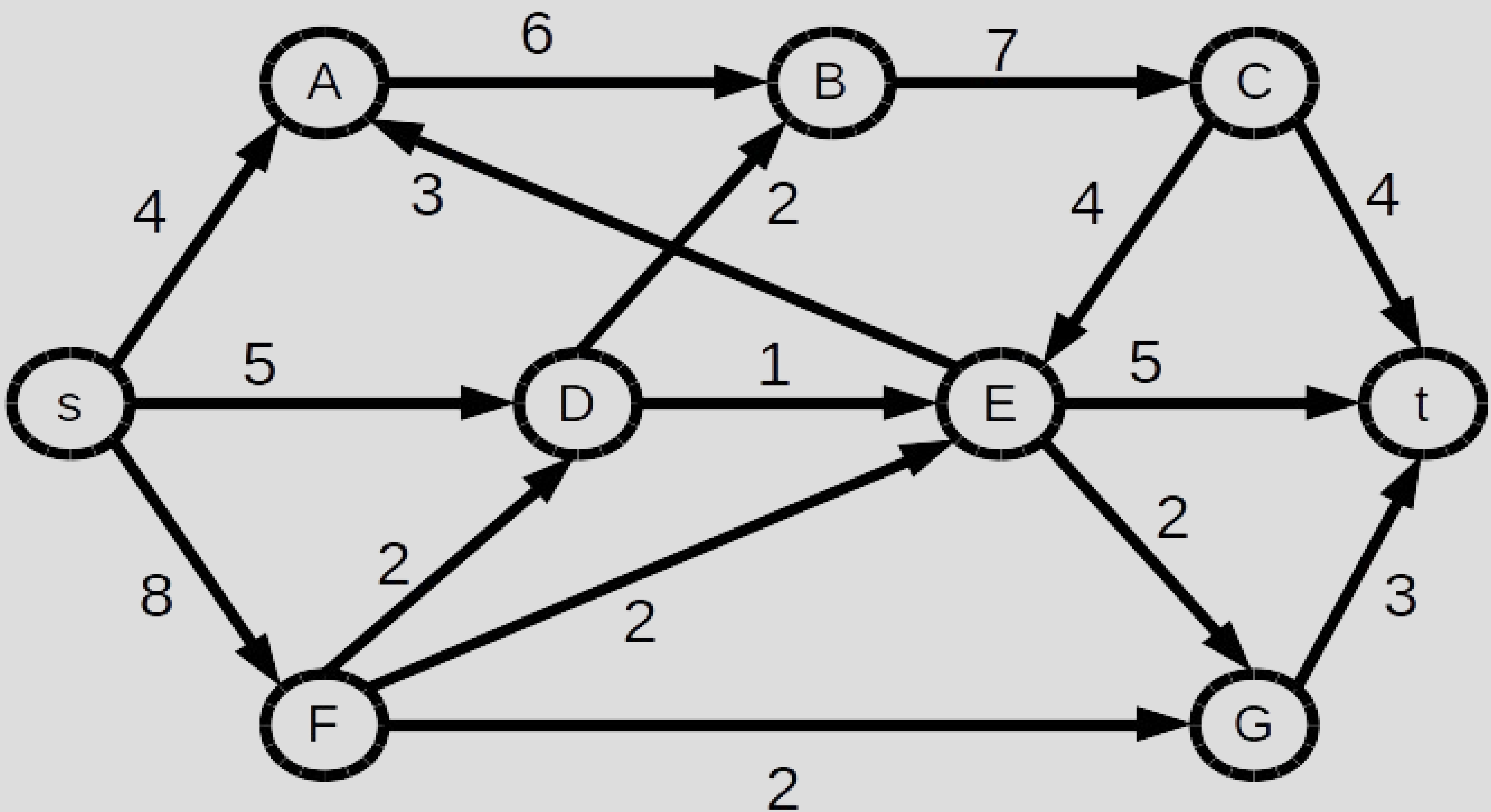
“forward edge” capacity - flow

“back edge” just flow

Ford-Fulkerson

```
Ford-Fulkerson(G, s, t)
for: each edge (u,v) in G.E: (u,v).f=0
while: exists path from s to t in G_f
  find c_f(p) // minimum edge cap. on path
  for: each edge (u,v) in p
    if(u,v) in E: (u,v).f=(u,v).f + c_f(p)
    else: (v,u).f=(v,u).f - c_f(p)
```

Max flow, min cut



Ford-Fulkerson

Runtime:

How hard is it to find a path?

How many possible paths could you find?

Ford-Fulkerson

Runtime:

How hard is it to find a path?
-O(E) (via BFS or DFS)
How many possible paths could you find?
- |f*| (paths might use only 1 flow)
.... so, O(E |f*|)

Edmonds-Karp

exists shortest path (BFS)

~~Ford-Fulkerson~~(G, s, t)

for: each edge (u,v) in G.E: (u,v).f=0

while: ~~exists path from s to t in G_f~~

find $c_f(p)$ // minimum edge cap.

for: each edge (u,v) in p

if(u,v) in E: (u,v).f=(u,v).f + $c_f(p)$

else: (u,v).f=(u,v).f - $c_f(p)$

Edmonds-Karp

Lemma 26.7
Shortest path in G_f is non-decreasing

Theorem 26.8
Number of flow augmentations by Edmonds-Karp is $O(|V||E|)$
So, total running time: $O(|V||E|^2)$

Ford-Fulkerson

$(f \uparrow f')(u,v)$ = flow f augmented by f'

$(f \uparrow f')(u,v) = f(u,v) + f'(u,v) - f'(v,u)$

Lemma 26.1: Let f be the flow in G, and f' be a flow in G_f, then $(f \uparrow f')$ is a flow in G with total amount:

$|f \uparrow f'| = |f| + |f'|$

Proof: pages 718-719

Ford-Fulkerson

For some path p:

$c_f(p) = \min(c_f(u,v) : (u,v) \text{ on } p)$

^^ (capacity of path is smallest edge)

Claim 26.3:

Let $f_p = c_f(p)$, then

$|f \uparrow f_p| = |f| + |f_p|$

Ford-Fulkerson

More bad notation:
c(u,v) = capacity of an edge
if u and v are single vertexes

c(S,T) = capacity across a cut
if S and T are sets of vertexes

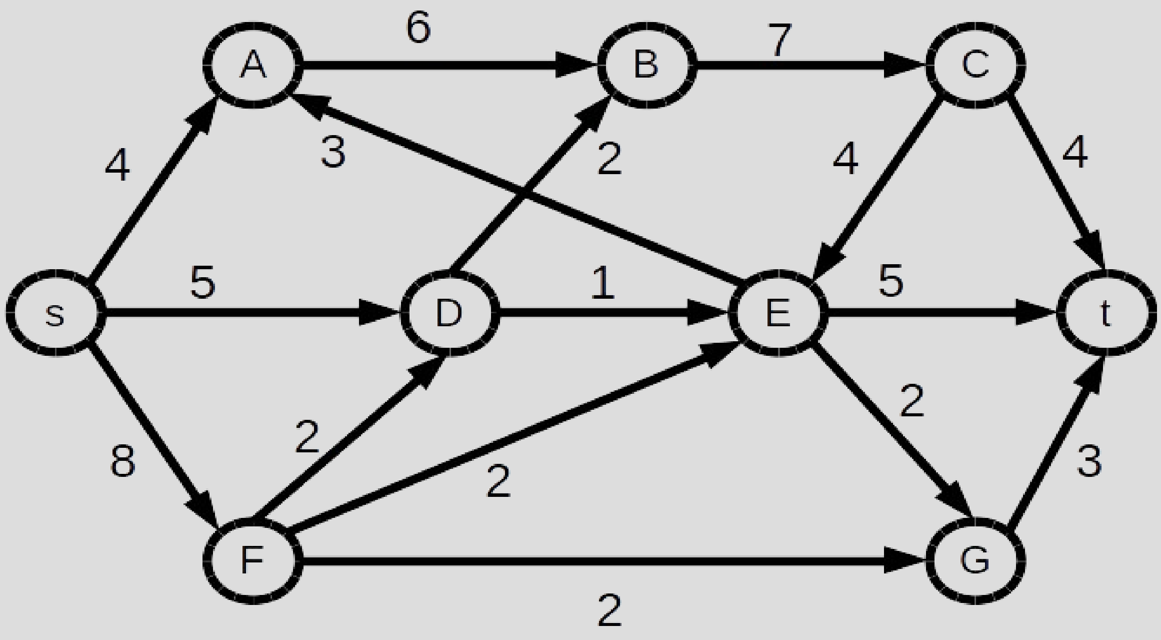
... Similarly for f(u,v) and f(S,T)

Max flow, min cut

Relationship between cuts and flows?

$c(S,T) = \sum_{u \text{ in } S}\sum_{v \text{ in } T} c(u,v)$

$f(S,T) = \sum_{u \text{ in } S}\sum_{v \text{ in } T} f(u,v) - \sum_{u}\sum_{v} f(v,u)$

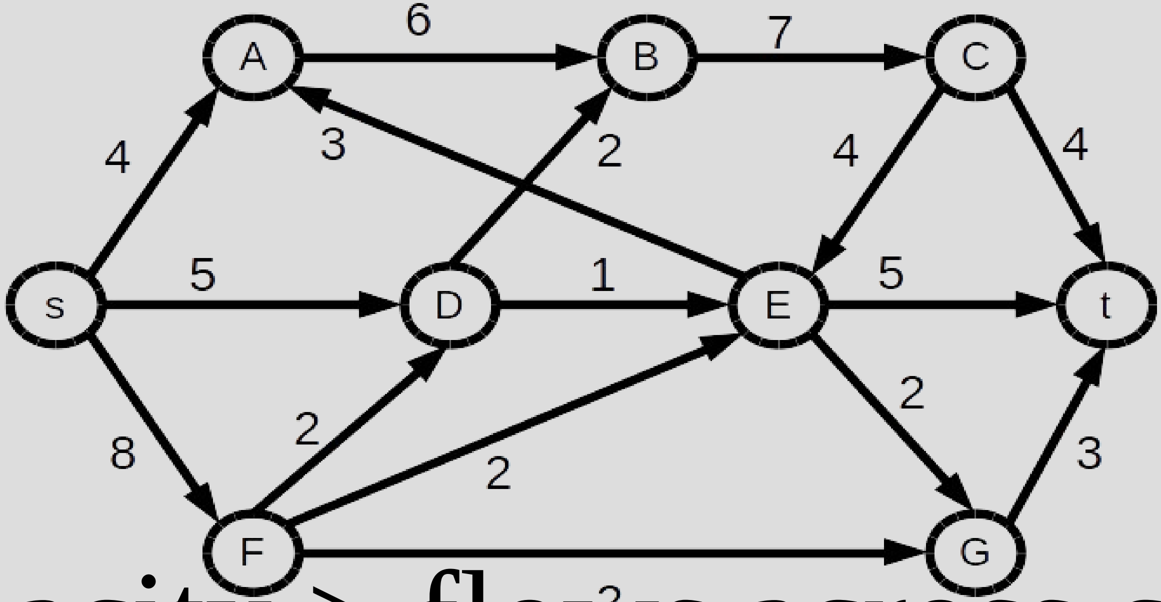


Max flow, min cut

Relationship between cuts and flows?

$$c(S,T) = \sum_{u \text{ in } S}\sum_{v \text{ in } T} c(u,v)$$

$$f(S,T) = \sum_{u \text{ in } S}\sum_{v \text{ in } T} f(u,v) - \sum_{u}\sum_{v} f(v,u)$$



cut capacity $\geq$ flows across cut

Max flow, min cut

Lemma 26.4

Let (S,T) be any cut, then $f(S,T) = |f|$

Proof:

Page 722

(Kinda long)

Max flow, min cut

Corollary 26.5

Flow is not larger than cut capacity

Proof:

$$
\begin{aligned}
|f| &= \sum_{u \text{ in } S}\sum_{v \text{ in } T} f(u,v) - \sum_{u}\sum_{v} f(v,u) \\
&\leq \sum_{u \text{ in } S}\sum_{v \text{ in } T} f(u,v) \\
&\leq \sum_{u \text{ in } S}\sum_{v \text{ in } T} c(u,v) \\
&= c(S,T)
\end{aligned}
$$

Max flow, min cut

Theorem 26.5

All 3 are equivalent:

1. f is a max flow
2. Residual network has no aug. path
3. $|f| = c(S,T)$ for some cut (S,T)

maximum network flow

= min cut (i.e. bottlneck)

Proof:

Will show: 1 => 2, 2=>3, 3=>1

Max flow, min cut

f is a max flow => Residual network has no augmenting path

Proof:

Assume there is a path p

$|f \uparrow f_p| = |f| + |f_p| > |f|$, which is a contradiction to $|f|$ being a max flow

Max flow, min cut

Residual network has no aug. path => $|f| = c(S,T)$ for some cut (S,T)

Proof:

Let S = all vertices reachable from s in G_f

u in S, v in T => $f(u,v) = c(u,v)$ else there would be path in G_f

Max flow, min cut

Also, $f(v,u) = 0$ else $c_f(u,v) > 0$ and again v would be reachable from s

$$
\begin{aligned}
f(S,T) &= \sum_{u \text{ in } S} \sum_{v \text{ in } T} f(u,v) - \sum_u \sum_v f(v,u) \\
&= \sum_{u \text{ in } S} \sum_{v \text{ in } T} c(u,v) - \sum_u \sum_v 0 \\
&= c(S,T)
\end{aligned}
$$

Max flow, min cut

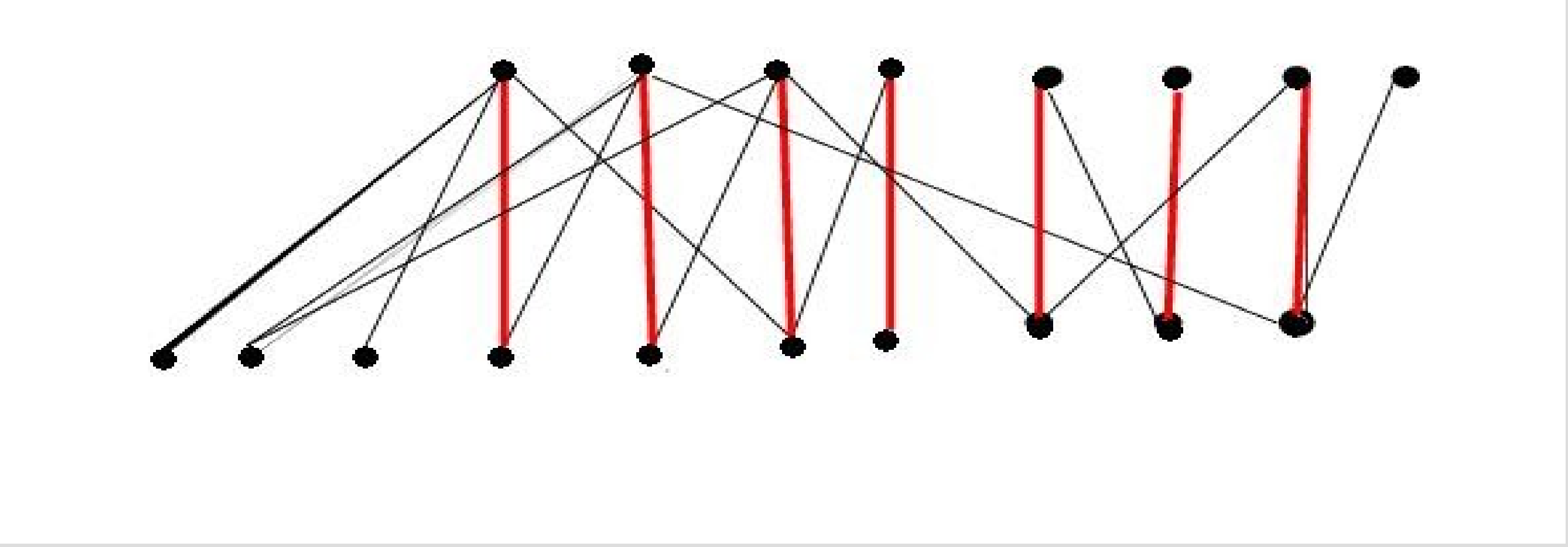
$|f| = c(S,T)$ for some cut (S,T)
=> f is a max flow

Proof:
$|f| \leq c(S,T)$ for all cuts (S,T)

Thus trivially true, as $|f|$ cannot get larger than $C(S,T)$

Matching

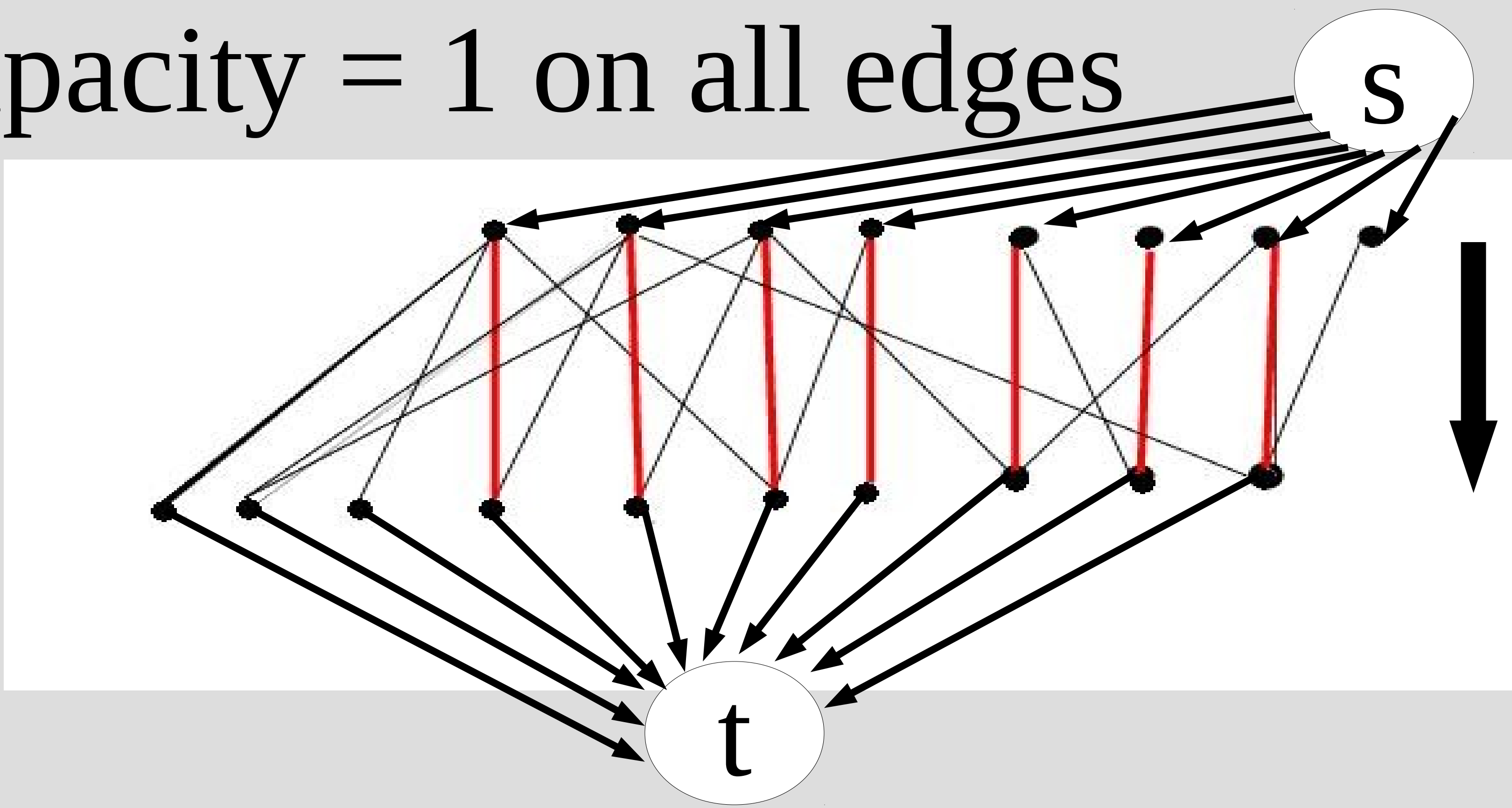
Another application of network flow is maximizing (number of)matchings in a bipartite graph



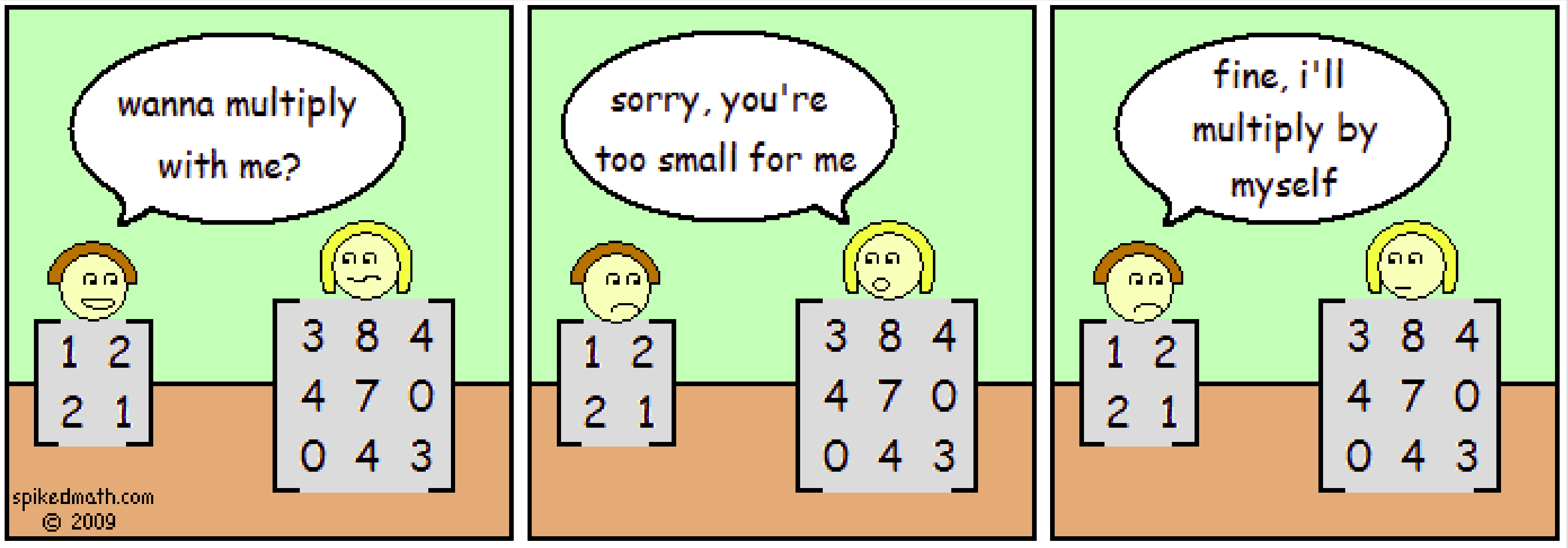
Each node cannot be “used” twice

Matching

Add “super sink” and “super source” (and direct edges source -> sink) capacity = 1 on all edges



Efficient matrix multiplication



Matrix multiplication

If you have square matrices A and B, then C = A*B is defined as:

$$c_{i,j} = \sum_{k=0}^{n} a_{i,k} \cdot b_{k,j}$$

For $\mathbf{A} = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}$ and $\mathbf{B} = \begin{bmatrix} 5 & 4 \\ -5 & 1 \end{bmatrix}$

$$\mathbf{AB} = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}\begin{bmatrix} 5 & 4 \\ -5 & 1 \end{bmatrix} = \begin{bmatrix} 5 & 4 \\ 5 & 9 \end{bmatrix}$$

$$\mathbf{BA} = \begin{bmatrix} 5 & 4 \\ -5 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 13 & 4 \\ -3 & 1 \end{bmatrix}$$

Takes $O(n^3)$ time

Matrix multiplication

Can we do better?

What is the theoretical lowest running time possible?

Matrix multiplication

Can we do better?

Yes!

What is the theoretical lowest running time possible?

$O(n^2)$, must read every value at least once

Matrix multiplication

Block matrix multiplication says:

$$\left[\begin{array}{c|c} \mathbf{A}_1 & \mathbf{A}_2 \\ \hline \mathbf{A}_3 & \mathbf{A}_4 \end{array}\right] \left[\begin{array}{c|c} \mathbf{B}_1 & \mathbf{B}_2 \\ \hline \mathbf{B}_3 & \mathbf{B}_4 \end{array}\right] = \left[\begin{array}{c|c} \mathbf{C}_1 & \mathbf{C}_2 \\ \hline \mathbf{C}_3 & \mathbf{C}_4 \end{array}\right]$$

Thus $C_1 = A_1 * B_1 + A_2 * B_3$,

We can use this fact to make a recursive definition

Matrix multiplication

Divide&conquer algorithm:

Mult(A,B)

If |A| == 1, return A*B (scalar)

else... divide A&B into 4 equal parts

C1 = Mult(A1,B1) + Mult(A2,B3)

C2 = Mult(A1,B2) + Mult(A2,B4)

C3 = Mult(A3,B1) + Mult(A4,B3)

C4 = Mult(A3,B2) + Mult(A4,B4)

Matrix multiplication

Running time:

- Base case is $O(1)$
- Recursive part needs to add two $n/4 \times n/4$ matrices, so $O(n^2)$
- 8 recursive calls, each size $n/2$

$T(n) = 8\,T(n/2) + O(n^2)$

$T(n) = O(n^{\log_2 8}) = O(n^3)$

Strassen's method

Although the simple divide&conquer did not improve running time...

Can eliminate one recursive call to get $O(n^{\log_2 7})$ with fancy math

Has a much larger constant factor, so not useful unless matrix big

Strassen's method

Step 1: compute some S's
(just 'cause!)

S1=B2-B4	S6=B1+B4
S2=A1+A2	S7=A2-A4
S3=A3+A4	S8=B3+B4
S4=B3-B1	S9=A1-A3
S5=A1+A4	S10=B1+B2

Strassen's method

Step 2: compute some P's (7 < 8)

P1=A1*S1

P2=S2*B4

P3=S3*B1

P4=A4*S4

P5=S5*S6

P6=S7*S8

P7=S9*S10

Strassen's method

Step 3:

C1 = P5 + P4 - P2 + P6

C2 = P1 + P2

C3 = P3 + P4

C4 = P5 + P1 - P3 - P7

(Book works out algebra for you)

Strassen's method

In practice, you should never use this on a matrix smaller than 16x16

The break-point is debatable, but Strassen's is better if over 100x100

Theoretical methods exist to reduce to $O(n^{2.3728639})$, but not practical at all

Fast Fourier Transform

The FFT is a very nice algorithm (ranks up there with bucket sort)

It has many uses, but we will use it to solve polynomial multiplication

Naive approach takes $O(n^2)$ time (i.e. FOIL)

Fast Fourier Transform

Assume we have polynomials:

$$A(x) = \sum_{j=0}^{n} a_j \cdot x^j,\ B(x) = \sum_{j=0}^{n} b_j \cdot x^j$$

C(x) = A(x) * B(x)

$$C(x) = \sum_{j=0}^{2\cdot n} c_j x^j \qquad c_j = \sum_k a_k \cdot b_{j-k}$$

O(n) per c_j, up to 2n c_j's = O(n^2)

Fast Fourier Transform

Rather than directly computing $C(x)$, map to a different representation

$A(x) = (x_0, y_0), (x_1, y_1), \ldots (x_n, y_n)$

Theorem 30.1: If $x_i \neq x_j$ for all $i \neq j$, then above gives a unique polynomial

Fast Fourier Transform

Proof: (direct)

Represent in matrix form:

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & x_1^n \\ & \dots & & & \\ 1 & x_n & x_n^2 & \dots & x_n^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \dots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \dots \\ y_n \end{bmatrix}$$

The left matrix is invertible, done

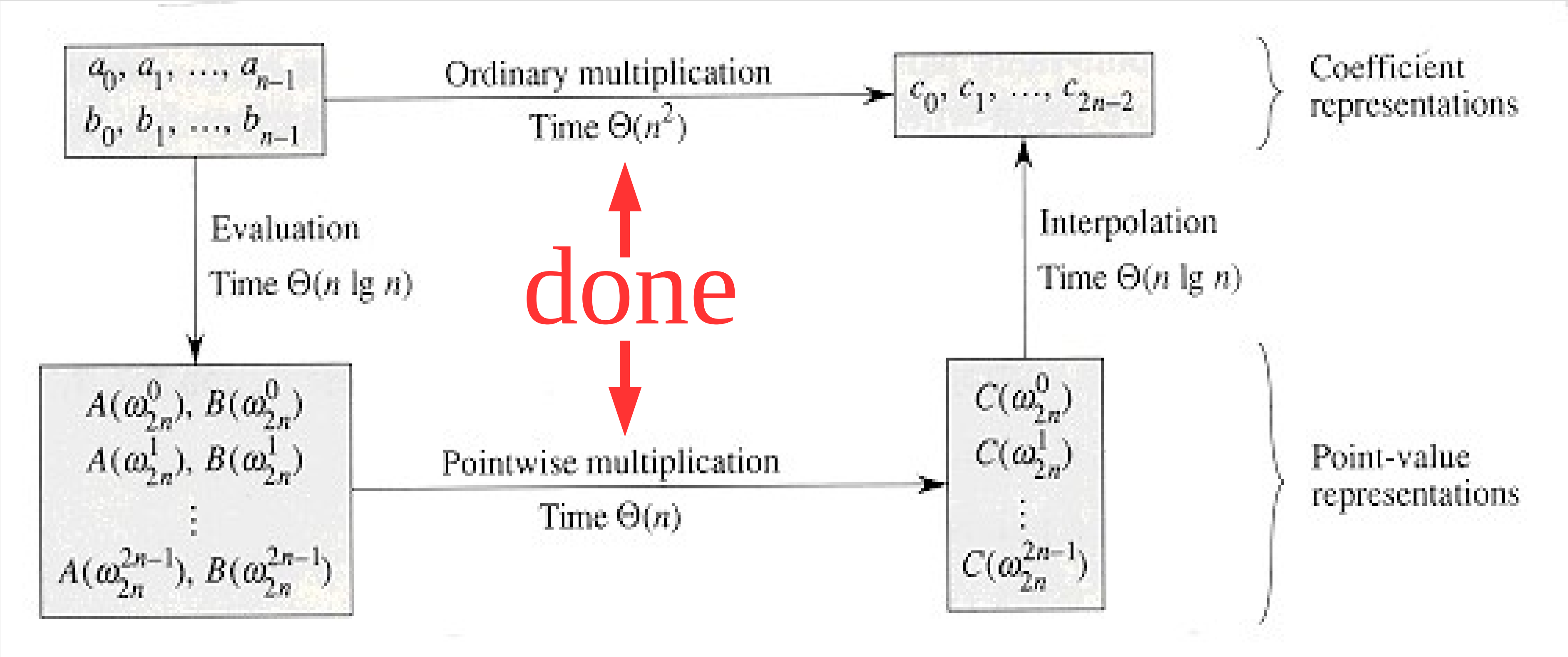
Fast Fourier Transform

Q: Why bother with point-values?
A: We can do A(x) * B(x) in O(n) in this space

Namely, $(x_i, cy_i) = (x_i, ay_i * by_i)$

Need to get to point-value and back to coefficients in less than $O(n^2)$

Fast Fourier Transform



Coming soon! (next time)