06. Maximum Flow

CPSC 535

Kevin A. Wortman

CALIFORNIA STATE UNIVERSITY FULLERTON

Big Idea: Algorithm Frameworks

Algorithm framework: an algorithm with modular parts that can be swapped in for different performance properties; or to solve different but related problems

Example: hash tables are a framework, can swap in

- different collision resolution strategy (chaining, probing)
- different hash function (universal hash, linear congruential hash, etc.)

A framework generalizes several algorithm ideas into one pattern; "chunking"

Big Idea: Iterative Pattern

Recall greedy pattern:

1. initialize base-case result
2. for each piece of input, update result

Iterative pattern (a.k.a. *fixed-point algorithm*):

1. initialize base-case result
2. while result is not optimal:
   2.1 improve result one step

The *fixed point* is the moment when the result becomes optimal.

Both use a *greedy heuristic*; iterative pattern makes a problem-wide decision.

Big Idea: Problem Duality

problem duality: when the input/output mathematical definition of a problem can be interpreted by humans in two (or more) very different ways

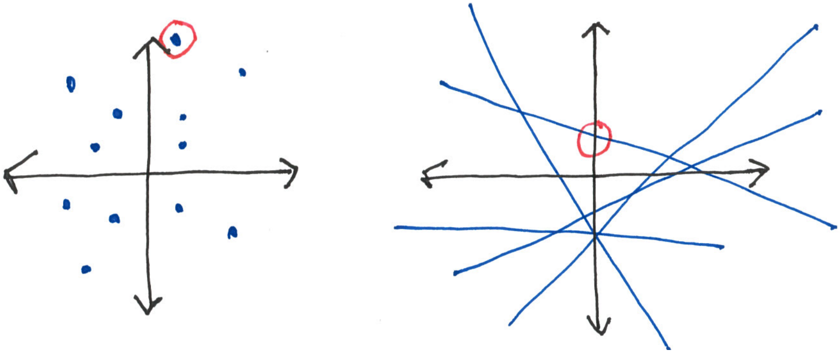
- one algorithm can solve multiple problems with different "stories"
- algorithms, computers, don't actually care what data values mean
- turns out max-flow and min-cut are two different stories for the same problem
- max-flow and min-cut are the *dual of each other*

Duality Example

maximum y coordinate
input: a set of (x, y) points $S = \{(x, y) \,|\, x, y \in \mathbb{R}\}$
output: the greatest y-coordinate in S

highest y-intercept point problem
input: a set of $y = mx + b$ lines $L = \{(m, b) \,|\, m, b \in \mathbb{R}\}$
output: the greatest y-intercept b in L

Geometry Sketch



C++ functions for these would be declared like:

```
double maximum_y_coord(vector<pair<double, double>>& points);
double highest_y_intercept(vector<pair<double, double>>& lines);
```

As far as the computer is concerned, these are interchangeable!

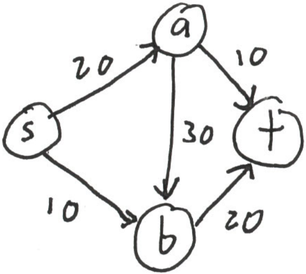
Only the human story differs. The **maximum** y **coordinate** and **highest** y**-intercept point problem** problems are the dual of each other.

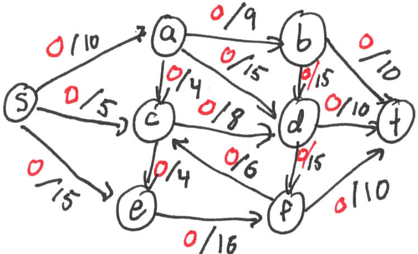
Defining Maximum Flow 1/2: Flow Networks

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**: $\forall v \in V,\ (v, v) \notin E$
- **no antiparallel edges**: for any $\forall (u, v) \in E, (v, u) \notin E$
- 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 $|E| \geq |V| - 1$

Flow Network Sketches





Defining Maximum Flow 2/2: Flows

flow: settings for how much capacity to use on each edge

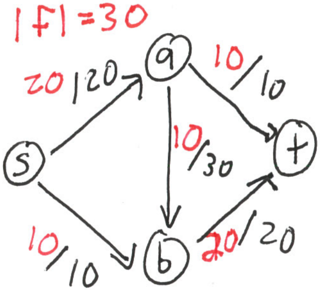
- candidate for maximum flow: follows the "rules," but not necessarily optimal
- modeled as function $f(u, v)$ over vertices u, v
- **nonexistent edges**: if $(u, v) \notin E$ then $f(u, v) = 0$
- **capacity constraint**: $0 \leq f(u, v) \leq c(u, v)$
- **flow conservation**: (flow-in) = (flow-out), except for source and sink; formally, $\forall u \in V - \{s, t\}$,

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

- *value* $|f|$ = net flow into sink

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

Flow Sketch



Maximum Flow Problem Definition

maximum flow problem
input: a flow network G
output: a flow f of maximum value $|f|$

Ford-Fulkerson Method

“method” because this is a framework 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
```

Ford-Fulkerson Method

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

Need to explain

- *residual network*
- *augmenting path*
- why this terminates and is correct

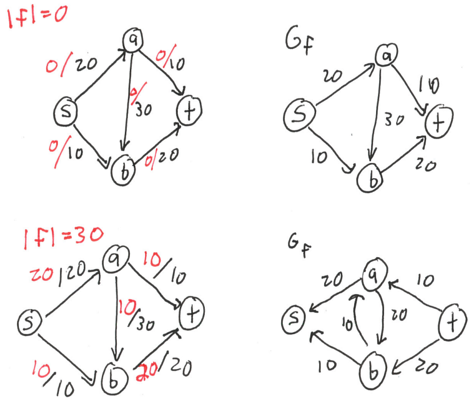
Residual Networks

- 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
- also add "backwards" edges to allow us to take-back some positive flow
- define *residual capacity* between vertices $v, w \in V$ as

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

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

Residual Network Example



Augmenting Paths

- *augmenting path:* simple path from source s to sink t in residual network c_f (*simple* $\equiv$ no repeated vertices)
- recall: residual network G_f only contains edges with leftover capacity
- $\implies$ if path p exists in G_f, then every edge along p has positive weight in G_f
- $\implies$ 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 acress the backwards edges
- $c_f(p)$ =*residual capacity* of p = minimum weight $c_f(u, v)$ of an edge (u, v) in p

Ford-Fulkerson Method Recap

Recall the Ford-Fulkerson method/pattern:

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

still need to

- clarify how to pick p: modular choice leading to specific algorithms
- prove correctness and termination: *max-flow min-cut theorem*

Max-Flow Min-Cut Theorem

Lemma: Augmenting a flow f with path p increases $s \rightsquigarrow t$ flow by $c_f(p)$.

Max-Flow Min-Cut Theorem: flow f is maximum iff G_f contains no augmenting path.

If true, any Ford-Fulkerson algorithm computes a correct 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
- we'll come back to this later

Cuts

- *cut:* partition $V = S \cap T$, where $s \in S$ and $t\ in T$
- *net flow* across f is

$$f(S, T) = (\text{total flow from } S \text{ to } T) - (\text{flow from } T \text{ to } S)$$

- *minimum cut* = a cut whose net flow is minimum

Lemma: for any cut (S, T), net flow $f(S, T) = |f|$.
Proof sketch: since $s \in S$ and $t \in T$, 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)

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

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

$(3) \implies (1)$: 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
        c_f(p) = min{c_f(u, v) : (u, v) ∈ p}
        for each edge (u, v) ∈ p do
            if (u, v) ∈ E then
                (u, v).f = (u, v).f + c_f(p)
            else
                (u, v).f = (v, u).f − c_f(p)
            end if
        end for
    end while
    return flow on .f fields
end function
```

Still abstract — need to clarify how we choose path p.

Edmonds-Karp Algorithm

Edmonds-Karp Algorithm is

- Ford-Fulkerson method from previous page, and...
- use breadth-first search (BFS) to find the shortest augmenting path
- (shortest $\equiv$ 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

Edmonds-Karp Pseudocode for Worked Examples

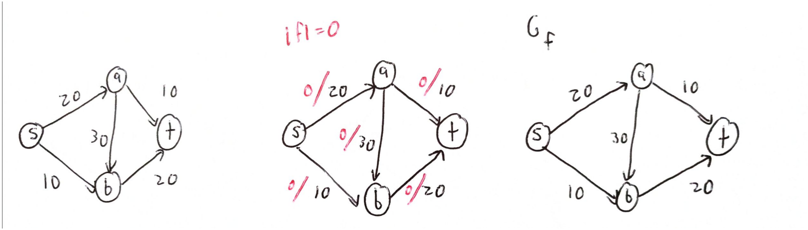
```
function EDMONDS-KARP(G = (V, E), s, t)
    initialize each edge's flow to 0
    repeat
        for k = 2, 3, ..., |V| do
            if ∃ augmenting path p of length k then
                c_f(p) = minimum excess capacity of any edge in p
                for edge e in p do
                    if p follows e forwards then
                        increase e's flow by c_f(p)
                    else
                        decrease e's flow by c_f(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

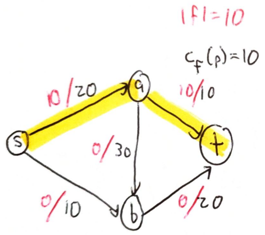
When running this algorithm by hand,

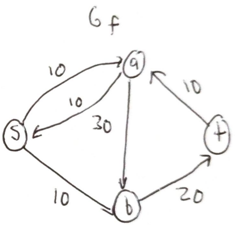
- you *could* sketch the residual network each time, but this is tedious
- instead, when looking at edge e with flow x/c
- if $x < c$, you *may* follow e forwards and add up to $(c - x)$ flow
- if $x > 0$, you *may* follow e backwards and subtract up to x flow

Edmonds-Carp Example 1/2

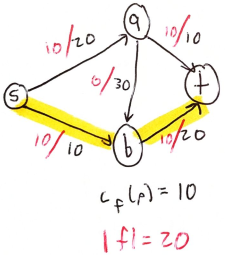


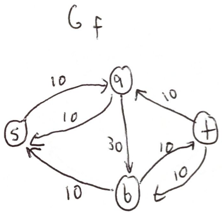
Edmonds-Carp Example 1/2



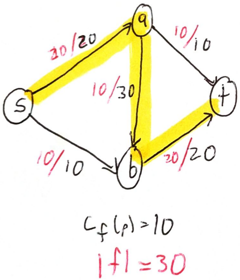


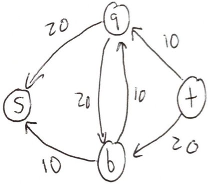
Edmonds-Carp Example 1/2



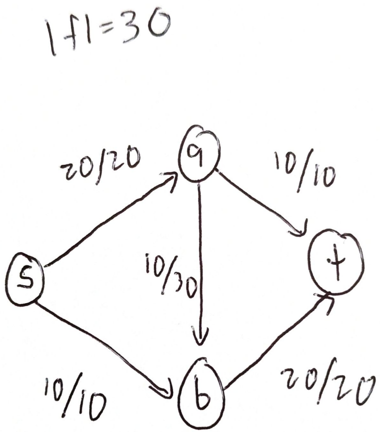


Edmonds-Carp Example 1/2

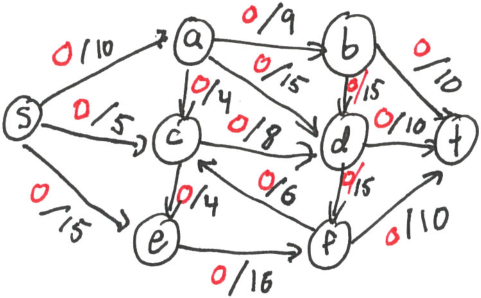




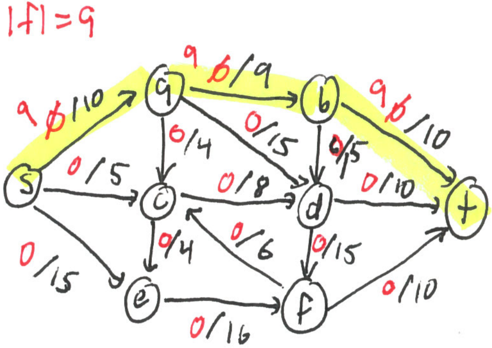
Edmonds-Carp Example 1/2



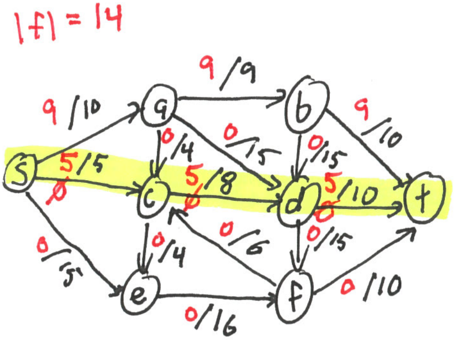
Edmonds-Carp Example 2/2



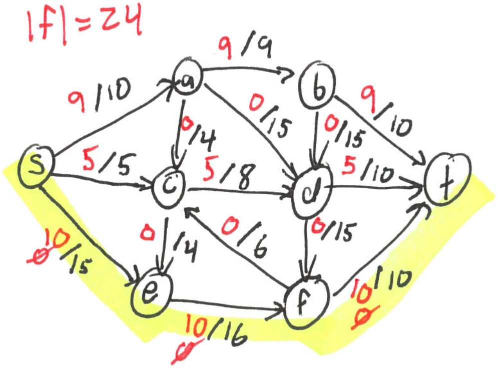
Edmonds-Carp Example 2/2



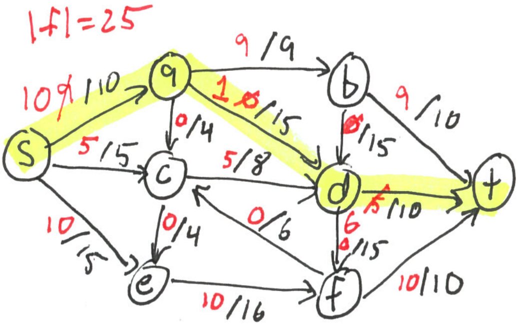
Edmonds-Carp Example 2/2



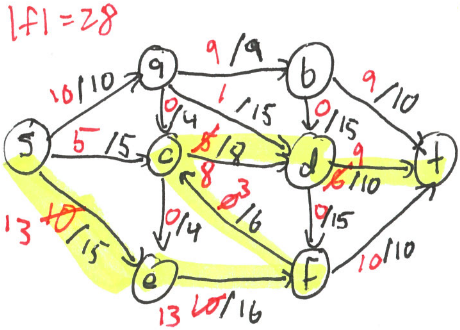
Edmonds-Carp Example 2/2



Edmonds-Carp Example 2/2



Edmonds-Carp Example 2/2



Edmonds-Carp Example 2/2