Week 11. Lecture Notes

Topics: Disjoint set data structure
Union-Find
Augmented disjoint set data structure
Network flow

Disjoint-set data structure (Union-Find)

Problem:

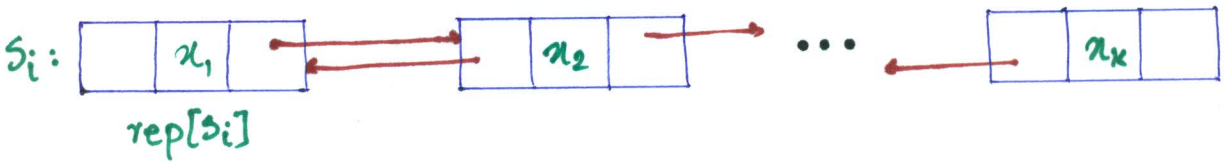
Maintain a dynamic collection of pairwise-disjoint sets $S = \{S_1, S_2, \ldots, S_r\}$. Each set S_i has one element distinguished as the representative element, $rep[S_i]$

Must support 3 operations:

- MAKE-SET(x): adds new set $\{x\}$ to S with $rep[\{x\}] = x$ (for any $x \notin S_i$ for all i)
- UNION(x,y): replaces sets S_x, S_y with $S_x \cup S_y$ in S for any x,y in distinct sets S_x, S_y.
- FIND-SET(x): returns representative $rep[S_x]$ of set S_x containing element x.

Simple linked-list solution

Store each set $S_i = \{x_1, x_2, \ldots, x_k\}$ as an (unordered) doubly linked list. Define representative element $rep[S_i]$ to be the front of the list, x_1.



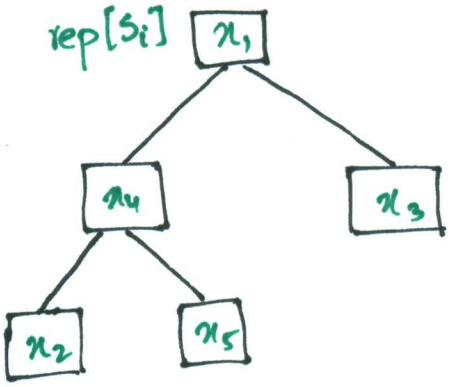
- MAKE-SET (x) initializes x as a lone node — $\Theta(1)$
- FIND-SET (x) walks left in the list containing x until it reaches the front of the list — $\Theta(n)$
- UNION (x,y) concatenates the lists containing x and y, leaving rep. as FIND-SET [x] — $\Theta(n)$

Simple balanced-tree solution

Store each set $S_i = \{x_1, x_2, \ldots, x_k\}$ as a balanced tree (ignoring keys). Define representative element rep[S_i] to be the root of the tree.

- MAKE-SET (x) initializes x as a lone node — $\Theta(1)$
- FIND-SET (x) walks up the tree containing x until it reaches the root — $\Theta(\log n)$
- UNION (x,y) concatenates the trees containing x and y changing rep. — $\Theta(\log n)$

$S_i = \{x_1, x_2, x_3, x_4, x_5\}$



Plan of attack

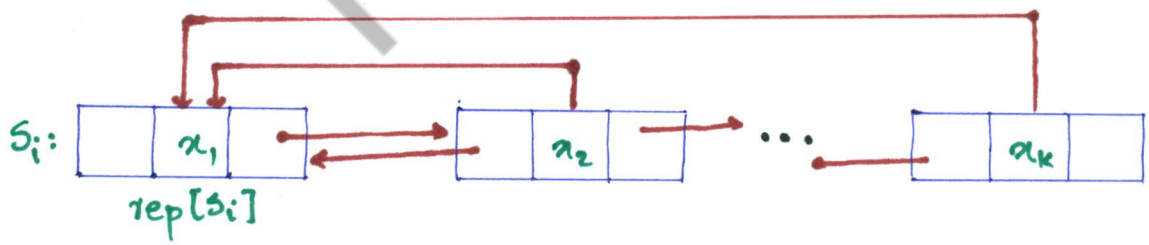
We will build a simple disjoint union data structure that, in an amortized sense, performs significantly better than $\Theta(\lg n)$ per operation, even better than $\Theta(\lg \lg n)$, $\Theta(\lg \lg \lg n)$, etc., but not quite $\Theta(1)$.

To reach this goal, we will introduce two key tricks Each trick converts a trivial $\Theta(n)$ solution into a simple $\Theta(\lg n)$ amortized solution. Together, the two tricks yield a much better solution.

First trick arises in an augmented linked list.
Second trick arises in a tree structure.

Augmented linked-list solution

Store set $S_i = \{x_1, x_2, \ldots, x_k\}$ as unordered doubly linked list. Define $rep[S_i]$ to be front of the list, x_1.
Each element x_j also stores pointer $rep[x_j]$ to $rep[S_i]$



- FIND-SET (x) returns $rep[x]$ — $\Theta(1)$
- UNION (x, y) concatenates the lists containing x and y and updates the rep pointers for all elements in the list containing y. — $\Theta(n)$

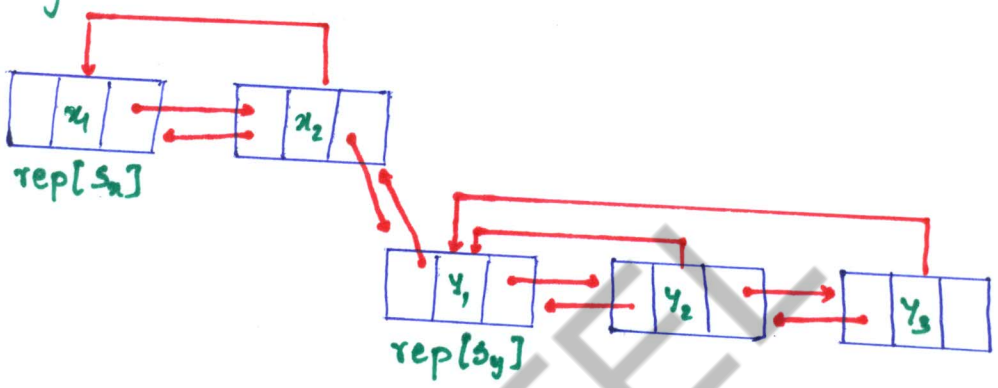
Example of augmented linked-list solution

Each element x_j stores pointer $rep[x_j]$ to $rep[S_i]$.

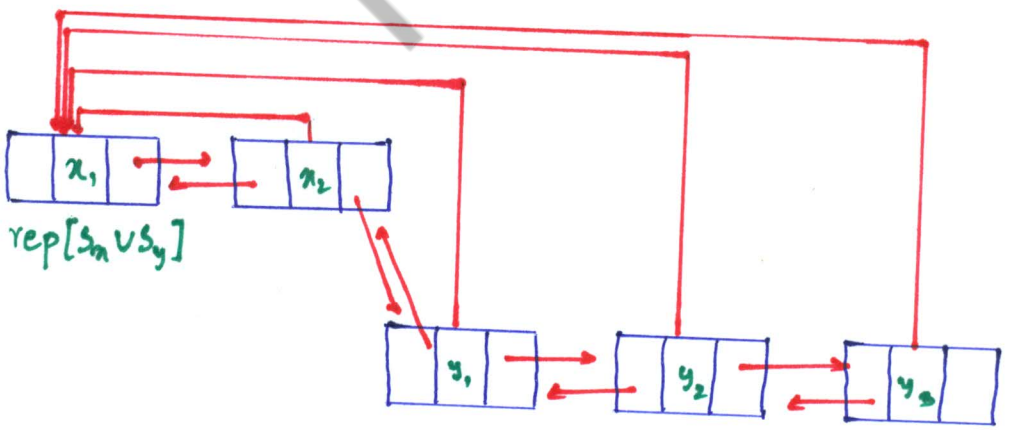
UNION (x, y)

- concatenates the lists containing x and y, and
- updates the rep pointers for all elements in the list containing y.

$S_x \cup S_y$:



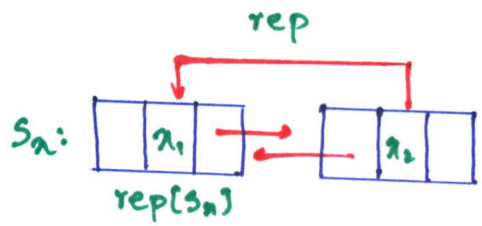
$S_x \cup S_y$:

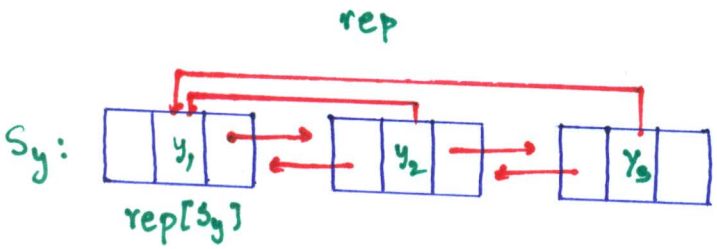


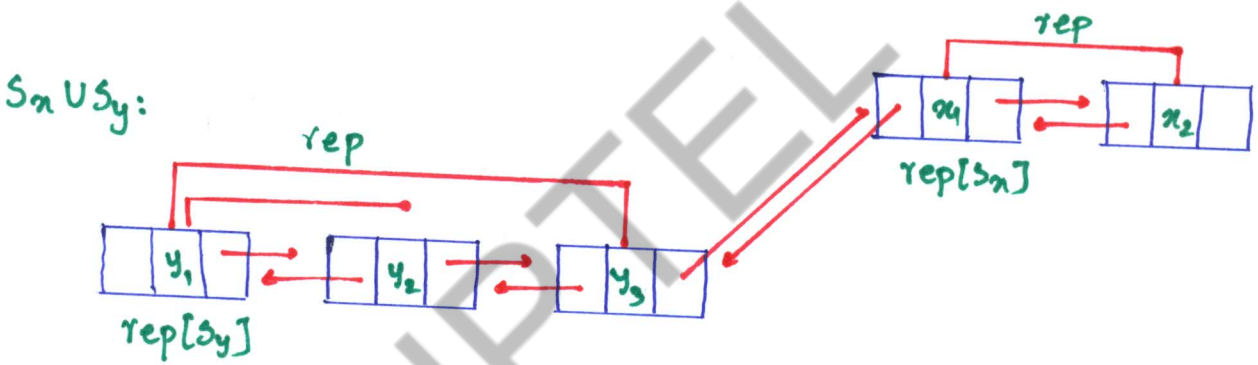
Alternative Concatenation

UNION(x,y) could instead

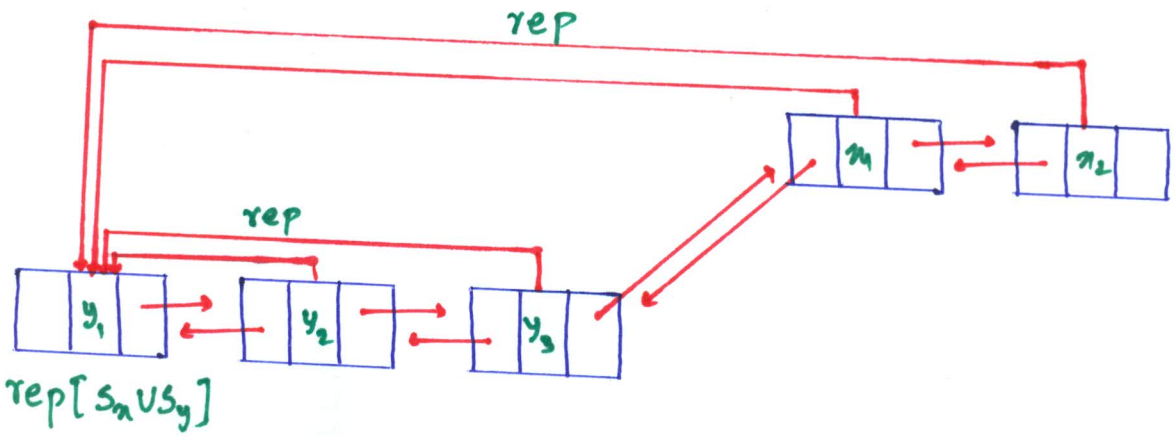
- concatenate the lists containing y and x, and
- update the rep pointers for all elements in the list containing x.







$S_x \cup S_y$:



Trick 1: Smaller into larger

To save work, concatenate smaller list into the end of the larger list. Cost = Θ(length of smaller list)

Augment list to store its weight (# elements)

Let "n" denote the overall number of elements (equivalently, the number of MAKE-SET operations).

Let "m" denote the total number of operations

Let "f" denote the number of FIND-SET operations.

THEOREM: Cost of all UNION's is $O(n \lg n)$.

Corollary: Total cost is $O(m + n \lg n)$.

Analysis of Trick 1

To save work, concatenate smaller list into the end of the larger list. Cost = Θ(1 + length of smaller list)

Theorem: Total cost of UNION's is $O(n \lg n)$

Proof:

Monitor an element x and set S_x containing it. After initial MAKE-SET(x), weight $[S_x] = 1$. Each time S_x is united with S_y, weight $[S_y] \geq$ weight $[S_x]$, pay 1 to update rep[x], and weight $[S_x]$ at least doubles (increasing by weight $[S_y]$).

Each time S_y is united with smaller set S_x, pay nothing, and weight $[S_x]$ only increases.

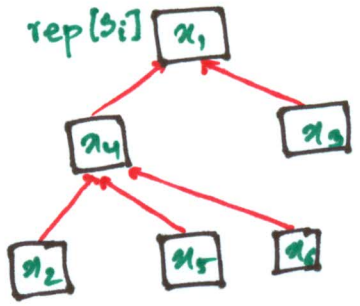
Thus pay $\leq \lg n$ for x

Representing sets as trees

Store each set $S_i = \{x_1, x_2, \ldots, x_k\}$ as an unordered, potentially unbalanced, not necessarily binary tree, storing only parent pointers. rep[S_i] is the tree root.

$S_i = [x_1, x_2, x_3, x_4, x_5, x_6]$

- MAKE-SET(x) initializes x as a lone node — $\theta(1)$
- FIND-SET(x) walks up the tree containing x until it reaches the root - $\theta(\text{depth}[x])$
- UNION(x,y) concatenates the trees containing x and y

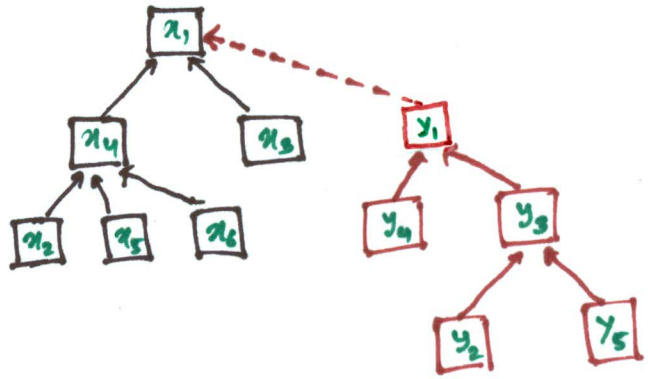


Trick 1 adapted to trees

UNION(x,y) can use a simple concatenation strategy:
Make root FIND-SET(y) a child of root FIND-SET(x).

$\Rightarrow$ FIND-SET(y) = FIND-SET(x)

We can adapt Trick 1 to this content also:
Merge tree with smaller weight into tree with larger weight.



Height of tree increases only when it's size doubles, so height is logarithmic in weight.

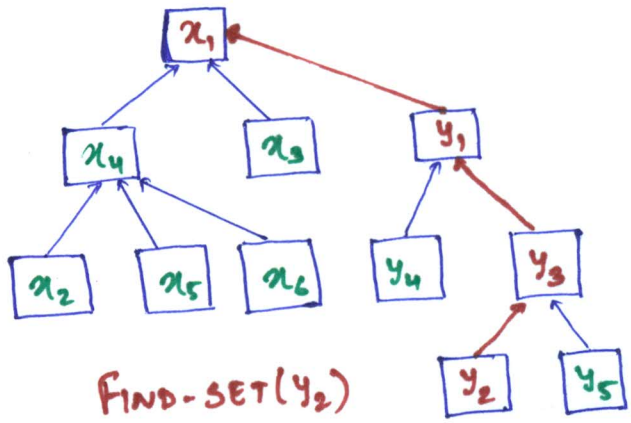
Thus total cost is $O(m + f \lg n)$

Trick 2: Path compression

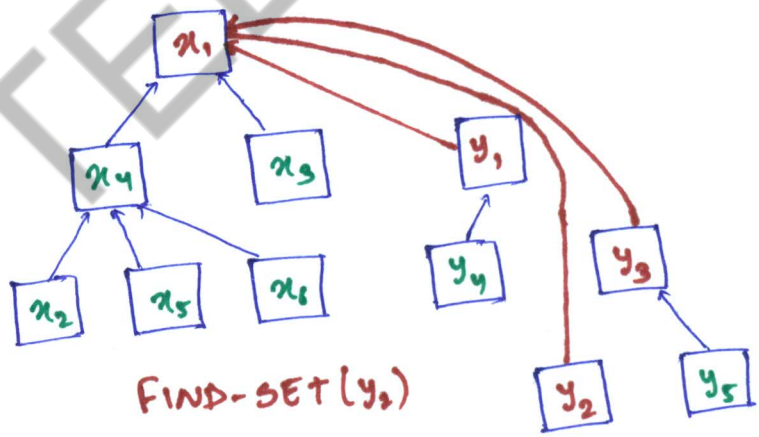
When we execute a FIND-SET operation and walk up a path p to the root, we know the representatives for all nodes on path p.

Path compression makes all of those nodes direct children of the root.

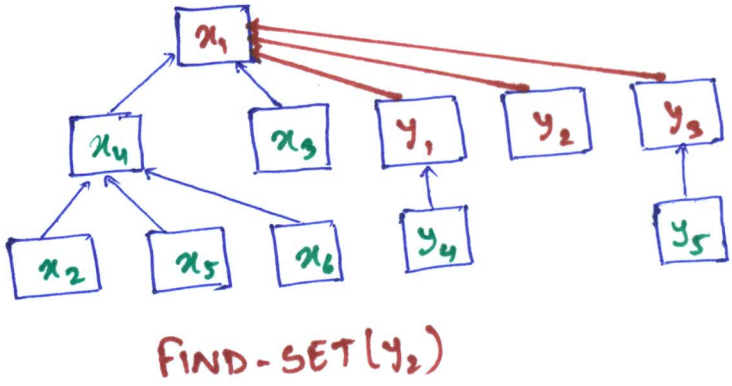
Cost of FIND-SET(x) is still $\theta(\text{depth}[x])$



cost of FIND-SET(x) is still $\theta(\text{depth}[x])$



Cost of FIND-SET(x) is still $\theta(\text{depth}[x])$



Analysis of Trick 2 alone

Theorem:

Total cost of FIND-SET is $O(m \lg n)$

Proof:

Amortization by potential function.

The weight of a node x is # nodes in it's subtree.

Define $\Phi(x_1, \ldots, x_n) = \sum_i \lg \text{weight}[x_i]$

UNION (x_i, x_j) increases potential of root FIND-SET(x_i), by at most $\lg \text{weight}[\text{root FIND-SET}(x_j)] \leq \lg n$.

Each step down $p \to c$ made by FIND-SET(x_i), except the first, moves c's subtree out of p's subtree. Thus if $\text{weight}[c] \geq \frac{1}{2}\,\text{weight}[p]$, Φ decreases by ≥ 1. paying for the step down. There can be at most $\lg n$ steps $p \to c$ for which $\text{weight}[c] < \frac{1}{2}\,\text{weight}[p]$.

Theorem:

If all UNION operations occur before all FIND-SET operations, then total cost is $O(mn)$

Proof:

If a FIND-SET operation traverses a path with k nodes, costing $O(k)$ time, then $k-2$ nodes are made new children of the root.

This change can happen only once for each of the n elements.

So, the total cost of FIND-SET is $O(f+n)$.

Ackermann's function A

Define $A_k(j) = \begin{cases} j+1, & \text{if } k=0 \\ A_{k-1}^{(j+1)}(j), & \text{if } k \geq 1 \end{cases}$ — iterate $j+1$ times

$A_0(j) = j+1$ $\qquad$ $A_0(1) = 2$

$A_1(j) \sim 2j$ $\qquad$ $A_1(1) = 3$

$A_2(j) \sim 2j\ 2^j > 2^j$ $\qquad$ $A_2(1) = 7$

$A_3(j) > 2^{2^{\cdot^{\cdot^{\cdot^{2^j}}}}}$ (j twos) $\qquad$ $A_3(1) = 2047$

$A_4(1)$ is a lot bigger. $A_4(1) > 2^{2^{\cdot^{\cdot^{\cdot^{2^{2047}}}}}}$ (2048 twos)

Define $\alpha(n) = \min\{k : A_k(1) \geq n\} \leq 4$ for practical n

Analysis of Tricks 1 + 2

Theorem:

In general, total cost is $O(m\,\alpha(n))$.

Application: Dynamic Connectivity

Suppose a graph is given to us incrementally by

ADD-VERTEX (v)

ADD-EDGE(u,v)

and we want to support connectivity queries:

CONNECTED (u,v):

Are u and v in the same connected component?

for example, we want to maintain a spanning forest, so we check whether each new edge connects a previously disconnected pair of vertices.

Sets of vertices represent connected components.

Suppose a graph is given to us incrementally by

ADD-VERTEX (v) - MAKE-SET(v)

ADD-EDGE(u,v) - if not CONNECTED(u,v)
then UNION(v,w)

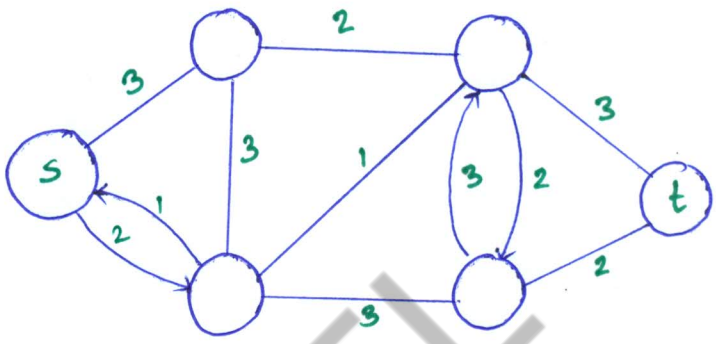
and

CONNECTED(u,v): FIND-SET(u) = FIND-SET(v)

Flow networks

Definition: A flow network is a directed graph $G=(V,E)$ with two distinguished vertices: a source s and a sink t. Each edge $(u,v) \in E$ has a non-negative capacity $c(u,v)$. If $(u,v) \notin E$, then $c(u,v) = 0$.

Example:



Definition: A positive flow on G is a function $p: V \times V \to R$ satisfying the following:

- Capacity constraint: for all $u, v \in V$

$$0 \leq p(u,v) \leq c(u,v)$$

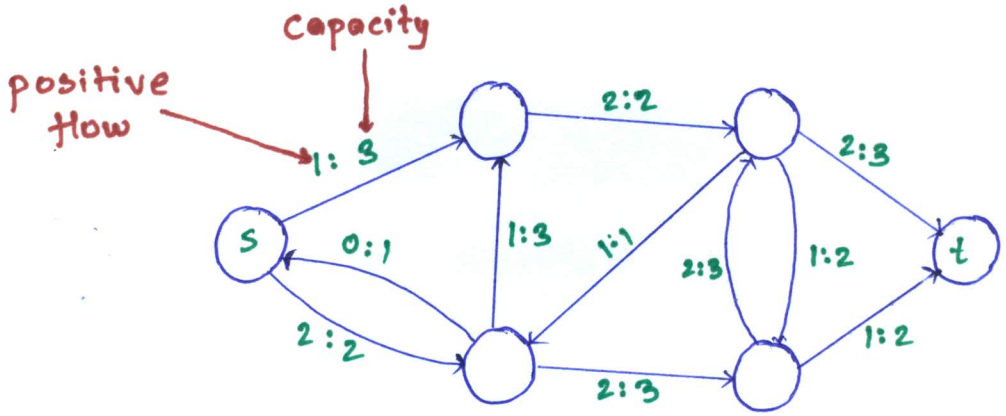
- Flow conservation: for all $u \in V - \{s,t\}$,

$$\sum_{v \in V} p(u,v) - \sum_{v \in V} p(v,u) = 0$$

The value of a flow is the net flow out of the source:

$$\sum_{v \in V} p(s,v) - \sum_{v \in V} p(v,s)$$

A flow on a network



Flow conservation (like Kirchoff's current law):

- Flow into u is 2+1=3
- Flow out of u is 0+1+2=3

The value of this flow is 1−0+2=3.

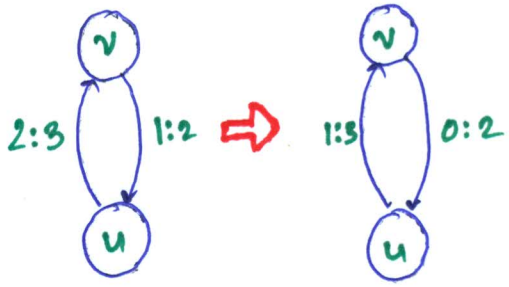
Maximum-flow problem: Given a flow network G, find a flow of maximum value in G.

In the above figure, value of maximum flow = 4.

Flow Cancellation

Without loss of generality, positive flow goes either from u to v, or from v to u, but not both.

The capacity constraint and flow conservation are preserved by this transformation



Net flow from u to v in both cases is 1

INTUITION:

View flow as a rate, not a quantity.

A notational simplification

IDEA: Work with the net flow between two vertices, rather than with the positive flow.

Definition: A (net) flow on G is a function $f: V \times V \to R$ satisfying the following:

- Capacity constraint: For all $u, v \in V$,
$$f(u,v) \leq c(u,v)$$
- Flow conservation: For all $u \in V - \{s,t\}$
$$\sum_{v \in V} f(u,v) = 0$$
} one assumption instead of two
- Skew-symmetry: For all $u, v \in V$
$$f(u,v) = -f(v,u)$$

Equivalence of Definitions

Theorem: The two definitions are equivalent

Proof:

Let $f(u,v) = p(u,v) - p(v,u)$

Capacity constraint: Since $p(u,v) \leq c(u,v)$ and $p(v,u) \geq 0$, we have $f(u,v) \leq c(u,v)$.

Flow conservation:

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

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

Skew symmetry: $f(u,v) = p(u,v) - p(v,u)$

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

$= -f(v,u)$

Next, consider

$$p(u,v) = \begin{cases} f(u,v), & \text{if } f(u,v) > 0 \\ 0, & \text{if } f(u,v) \leq 0 \end{cases}$$

Capacity constraint: By definition $p(u,v) \geq 0$. Since $f(u,v) \leq c(u,v)$, it follows that $p(u,v) \leq c(u,v)$

Flow conservation: If $f(u,v) > 0$, then $p(u,v) - p(v,u) = f(u,v)$.

If $f(u,v) \leq 0$, then $p(u,v) - p(v,u) = -f(v,u) = f(u,v)$ (by skew symmetry)

Therefore,

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

Notation

Definition: The value of a flow f, denoted by $|f|$ is given by

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

Implicit summation notation:

A set used in an arithmetic formula represents a sum over the elements of the set.

Example:

flow conservation:

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

Simple properties of flow

Lemma:

$f(X, X) = 0$

$f(X, Y) = -f(Y, X)$

$f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$ if $X \cap Y = \phi$

Theorem

$|f| = f(V, t)$

Proof: $|f| = f(s, V)$

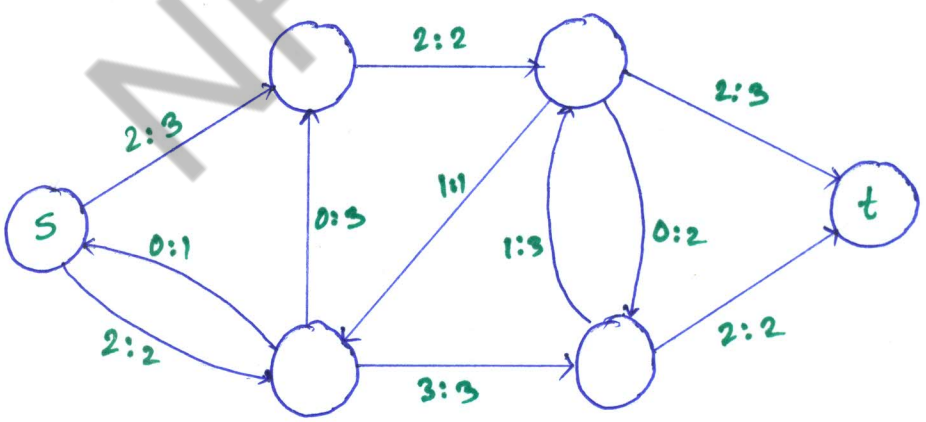
$= f(V, V) - f(V - s, V)$ Omit braces

$= f(V, V - s)$

$= f(V, t) + f(V, V - s - t)$

$= f(V, t)$

Flow into the sink



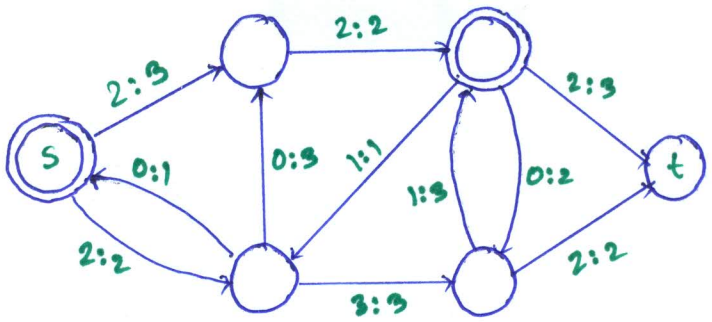
$|f| = f(s, V) = 4$

$f(V, t) = 4.$

Cuts

A cut (S,T) of a flow network $G = (V, E)$ is a partition of V such that $s \in S$ and $t \in T$.

If f is a flow on G, then the flow across the cut is $f(S,T)$



◎ $\in S$

○ $\in T$

$$f(S,T) = (2+2) + (-2+1-1+2) = 4$$

Another characterization of flow value

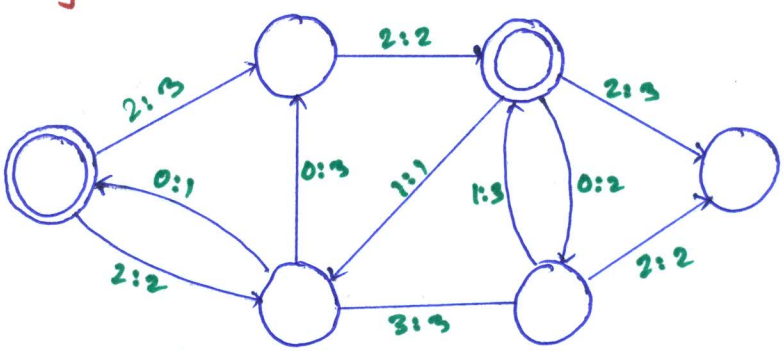
Lemma: For any flow f and any cut (S,T) $|f| = f(S,T)$.

Proof:

$$\begin{aligned} f(S,T) &= f(S,V) - f(S,S) \\ &= f(S,V) \\ &= f(s,V) - f(S-s,V) \\ &= f(s,V) = |f| \end{aligned}$$

Capacity of a cut

capacity of a cut (S,T) is $c(S,T)$



◎ $\in S$

○ $\in T$

$$\begin{aligned} c(S,T) &= (3+2) + (1+2+3) \\ &= 11 \end{aligned}$$

Upper bound on the maximum flow value

Theorem:

The value of any flow is bounded above by the capacity of any cut.

Proof:

$$|f| = f(S,T)$$
$$= \sum_{u \in S} \sum_{v \in T} f(u,v)$$
$$\leq \sum_{u \in S} \sum_{v \in T} c(u,v)$$
$$= c(S,T)$$

Residual Network

Definition:

Let f be a flow on $G = (V,E)$. The residual network $G_f(V, E_f)$ is the graph with strictly positive residual capacities:

$$c_f(u,v) = c(u,v) - f(u,v) > 0.$$

Edges in E_f admit more flow.

Example:

Lemma:

$$|E_f| \leq 2|E|$$

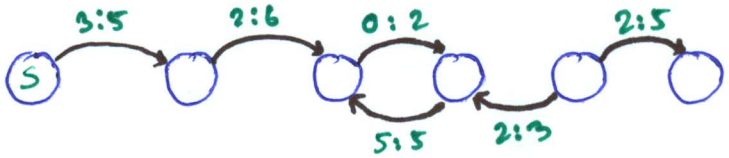
Augmenting Paths

Definition: Any path from s to t in G_f is an augmenting path in G with respect f. The flow value can be increased along an augmenting path p by

$$c_f(p) = \min_{(u,v) \in p} \{c_f(u,v)\}$$

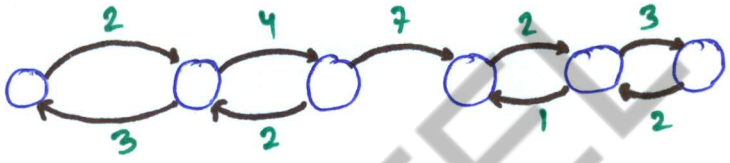
Ex:

G:



$c_f(p) = 2$

G_f:



Max-flow, min cut theorem

Theorem:

The following are equivalent:

1. f is a maximum flow
2. f admits no augmenting paths
3. $|f| = c(S,T)$ for some cut $[S,T)$