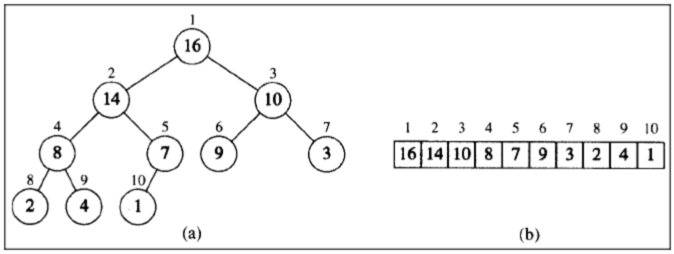
CAD Algorithms for Physical Design – Longest Path and Max Flow

Christos P Sotiriou

Contents

Binary Heap Basics

- A Heap viewed as (a) a binary tree, (b) an aray



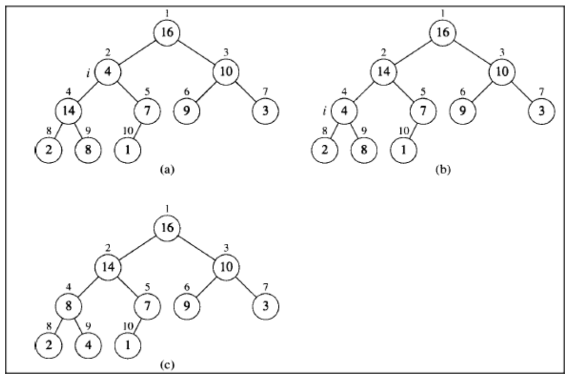
- Heap Property
  - for every node, other than the root, the value of a node is less/equal to the value of its parent node,
    - Value[Parent(i)] >= Value[i]
  - Thus, the root node always store maximum value in the Heap

Binary Heap Basics

- Heap/Binary Tree Properties:
  - for N-sized heap, represented as an array, elements [N/2...N] are leaves
  - for a heap/binary tree node i:
    - parent(i) = i/2, left(i) = 2*i, right(i) = 2*i + 1
  - size of N-height heap is 2^(N + 1)-1, where height N excludes root node!
- Minimum value heap can be created simply by storing negative values

Heapify

- Assuming that left(i), right(i) are heaps, but node i may smaller than its children, heapify pushes down i
- Heapify of node 2:



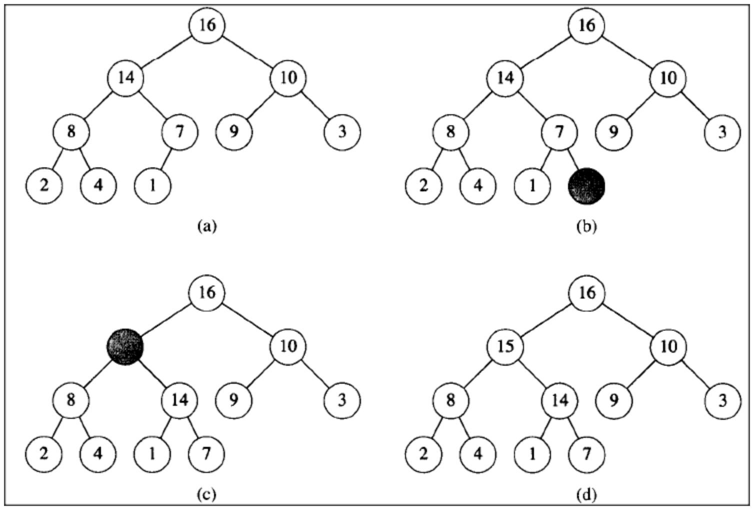
Basic Heap Operations

```
HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ heap-size[A] and A[l] > A[i]
    then largest ← l
    else largest ← i
if r ≤ heap-size[A] and A[r] > A[largest]
    then largest ← r
if largest ≠ i
    then exchange A[i] ↔ A[largest]
         HEAPIFY(A, largest)
```

```
HEAP-EXTRACT-MAX(A)
if heap-size[A] < 1
    then error "heap underflow"
max ← A[1]
A[1] ← A[heap-size[A]]
heap-size[A] ← heap-size[A] − 1
HEAPIFY(A, 1)
return max
```

```
HEAP-INSERT(A, key)
heap-size[A] ← heap-size[A] + 1
i ← heap-size[A]
while i > 1 and A[PARENT(i)] < key
    do A[i] ← A[PARENT(i)]
       i ← PARENT(i)
A[i] ← key
```

Heap Insert Example



Dijkstra's Shortest Path Algorithm

DIJKSTRA's Shortest Path (Graph(V, E), source)

```
for each vertex v in Graph:              // Initializations
  dist[v] := infinity ;                  // Unknown distance function from source to v
  previous[v] := undefined ;             // Previous node in optimal path
end for                                  // from source
dist[source] := 0 ;                      // Distance from source to source
Q := the set of all nodes in Graph ;     // All nodes in the graph are unoptimized
                                         // thus are in Q
while Q is not empty: // the main loop
   u := vertex in Q with smallest distance in dist[] ; // Source node in first case
   remove u from Q ;
   if dist[u] = infinity:
     break ;                             // all remaining vertices are
   end if                                // inaccessible from source

   for each neighbor v of u:             // where v has not yet been removed from Q.
      alt := dist[u] + dist_between(u, v) ;
      if alt < dist[v]:                  // Relax (u,v,a)
         dist[v] := alt ;
         previous[v] := u ;              // Store Shortest Path
         decrease-key v in Q;            // Reorder v in the Queue
      end if
   end for
end while
return dist;
```

STA Longest Path Algorithm

STA Longest_Path(Graph(V, E), L, I, spec)

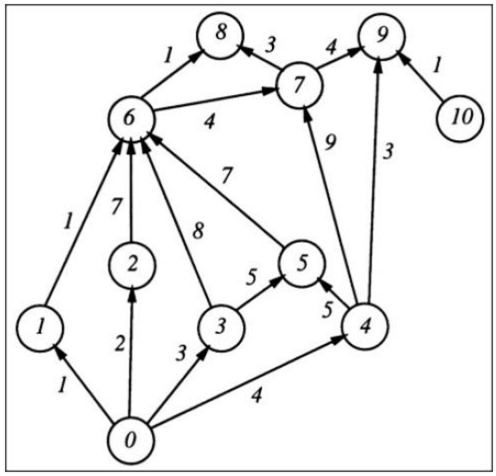
```
n = |V|; m = |E|; q = |I|;
for (v in V) {
  dist[v] := 0 ;
  Dv = |→v| ;
}
Q = I;
while (Q != 0) {
  v = DEQUEUE(Q);
  foreach (a in v→) {
    dist[a] = max(dist[a], (dist[v] + L(v, a)));
    Da = Da - 1;
    if (Da == 0) QUEUE(Q, a);
  }
maxdist = maxv in v(dist[v]);
maxv = SELECT1(V, maxdist);
critical_path = BACK_TRACE(V, E, L, dist[], maxv, (spec - maxdist));

return (critical_path, dist[]);
```

- L(v, u) is the edge length
- dist[v] is an iteratively increasing lower bound on the longest path length from the PIs to v
- Dv is the number of incoming edges to node v in V
- v → is the successors of v, → v the predecessors of v

STA Longest Path Algorithm and Backtracing

- The length of the longest path to any node maxdist is computed and passed to select one node, whereby
  - dist[v] = maxdist
- spec is the RAT – Required Arrival Time
  - (spec – maxdist) indicates path slack or violation
- Complete picture of delay evaluation includes
  - Arrival Time
  - Required Arrival Time
  - The difference between the two is the slack

Timing Graph Example



Data Trace running Longest Path

Line	v, λ^* 6	$v \rightarrow$ 6	(λ_a, D_a) 10										
k			0	1	2	3	4	5	6	7	8	9	10
1	0/0	1,2,3,4	0/0	1/0	2/0	3/0	4/0	0/2	0/4	0/2	0/2	0/3	0/0
2	10/0	9										1/2	
3	1/1	6							2/3				
4	2/2	6							9/2				
5	3/3	5,6						8/1	11/1				
6	4/4	5,7,9						9/0		13/1		7/1	
7	5/9	6							16/0				
8	6/16	7,8								20/0	17/1		
9	7/20	8,9									23/0	24/0	
10	8/23	∅											
11	9/24	∅											
	final	λ:	0	1	2	3	4	9	16	20	23	24	0

Edge and Node Slack

- **Definition**
  - The ***slack*** if an edge *(a, v)* is the slack of **v**, plus the difference between the longest path length to **v**, and the longest path to v through *(a, v)*:

$$slack_{a,v} = slack_v + (dist[v] - (dist[a] + L_{a,v}))$$

  - The ***slack*** if a node *v* is the minimum slack of its fanout edges

$$slack_a = min_{v\ in\ a\rightarrow}\ slack_{a,v}$$

- Simpler Formula for Single Critical Path

$$slack_a = slack_v + (dist[v] - (dist[a] + L_{a,v}))$$

Back-Tracing – Slack Computation

BACK_TRACE(Graph(V, E), L, maxdist, maxv, Rslack)

```
foreach (v in V) slack[v] = maxdist;
slack[maxv] = Rslack;
critical_path = {maxv};
QUEUE(Q, maxdist);
while (Q != 0) {
  v = DEQUEUE(Q);
  foreach (a in v→) {
    slack[a] = slack[v] + (dist[v] - (dist[a] + La,v));
    if (slack[a] == Rslack) {
      QUEUE(Q, a);
      critical_path = {a} U critical_path;
      break;
    }
  }
}
return (critical_path, slack[]);
```

- maxv is a (any) node of maximum depth
- Rslack is the required Slack – could be 0

Back-Tracing – Slack Computation

- For each active node v, as soon as a new 0-slack node a is encountered in the backward traversal
  - a is put at the end of Q, and the for loop is exited by break
- Non critical nodes may not be updated
  - Will still have their initialized slack values (Rslack)
- Final slack values also depend on the order in which nodes in the fanin →v are processed
- Critical Path for example: {0, 4, 5, 6, 7, 9}
- Slack values:

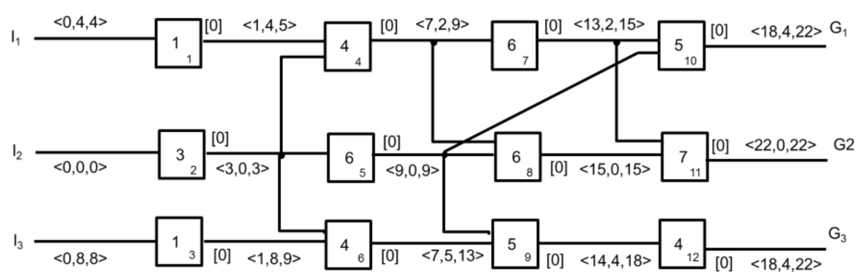
	v0	v1	v2	v3	v4	v5	v6	v7	v8	v9	v10
slack	0	14	7	1	0	0	0	0	24	0	23

Related Issue: Zero Slack Assignment

- Establish timing budgets for nets
  - Gate and wire delays must be optimized during timing driven layout design
  - Wire delays depend on wire lengths
  - Wire lengths are not known until after placement and routing
- Delay budgeting with the zero-slack algorithm
  - Let vi be the logic gates
  - Let ei be the nets
  - Let DELAY(v) and DELAY(e) be the delay of the gate and net, respectively
  - Define the timing budget of a gate
    - TB(v) = DELAY(v) + DELAY(e)

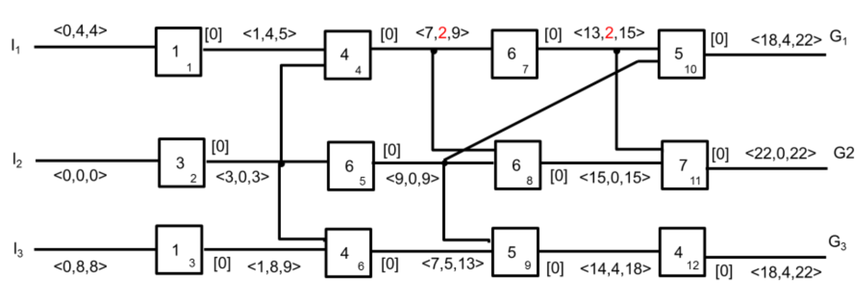
ZSA Example

- Tuple is <AT, Slack, RAT>



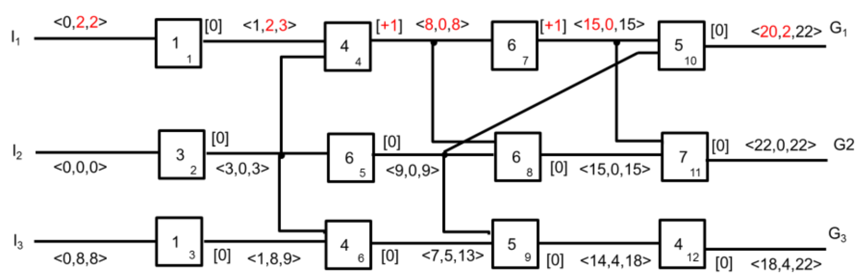
ZSA Example

- Identify ***minimum slack*** path > 0



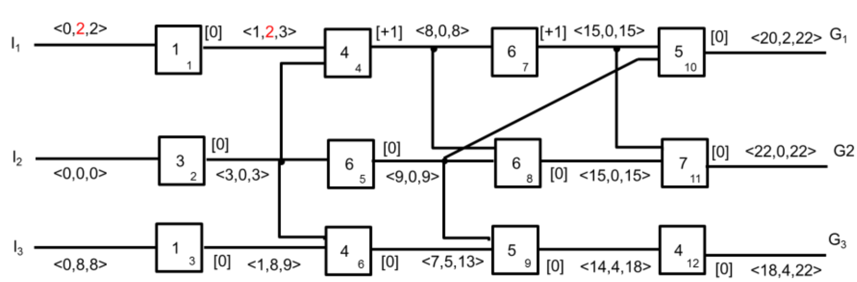
ZSA Example

- Distribute slacks, and update timing budgets



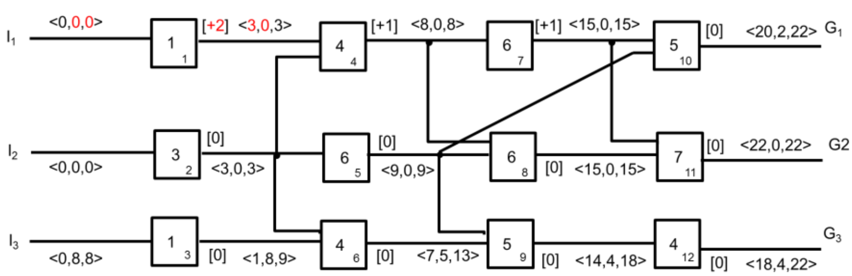
ZSA Example

- Identify again the minimum slack path > 0



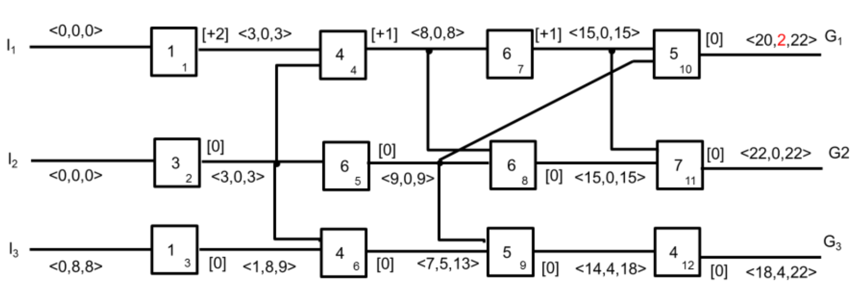
ZSA Example

- Distribute slacks, and update timing budgets



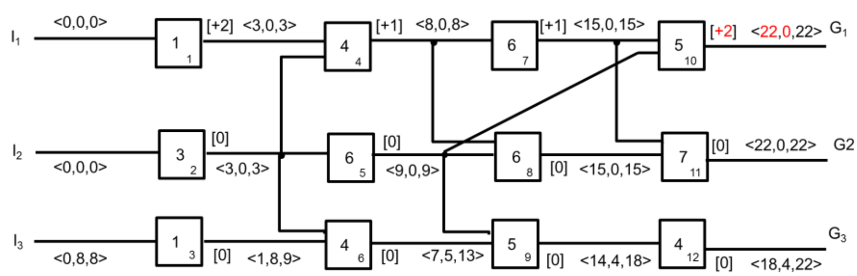
ZSA Example

- Identify new minimum slack path > 0



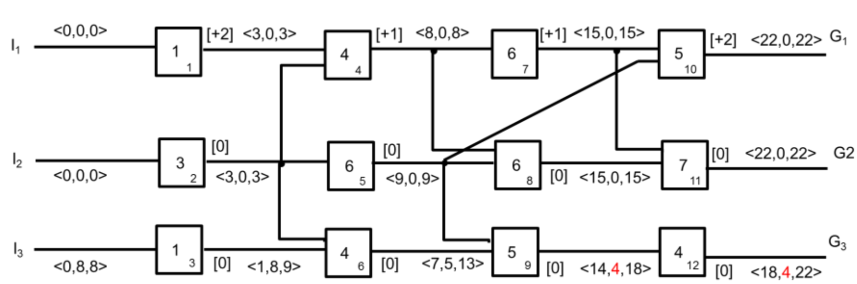
ZSA Example

- … distribute slacks, update local timing budgets



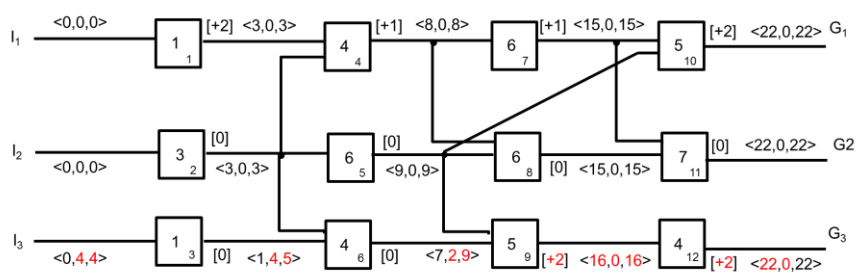
ZSA Example

- … Identify new minimum slack path > 0



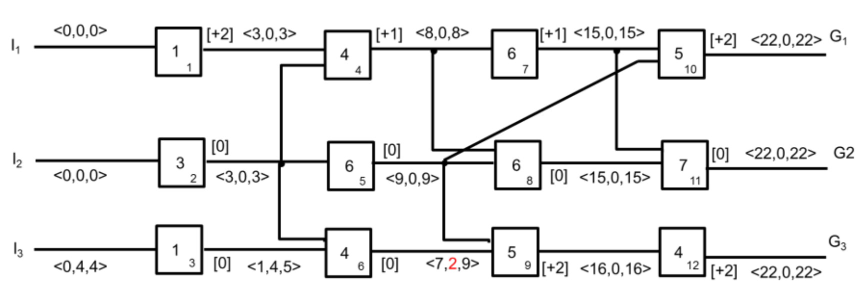
ZSA Example

- … distribute



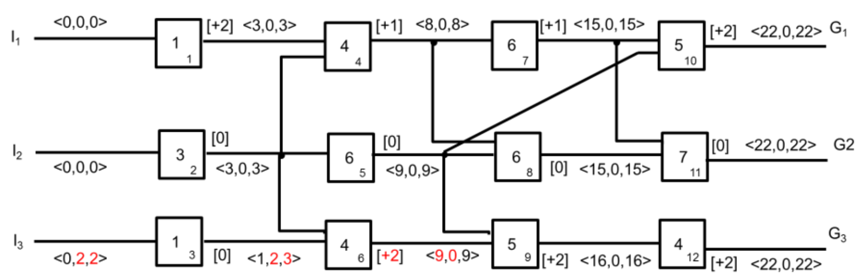
ZSA Example

- … new minimum slack > 0 path



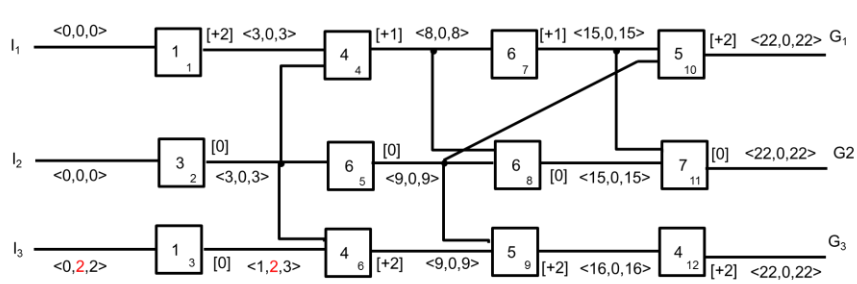
ZSA Example

- … distribute



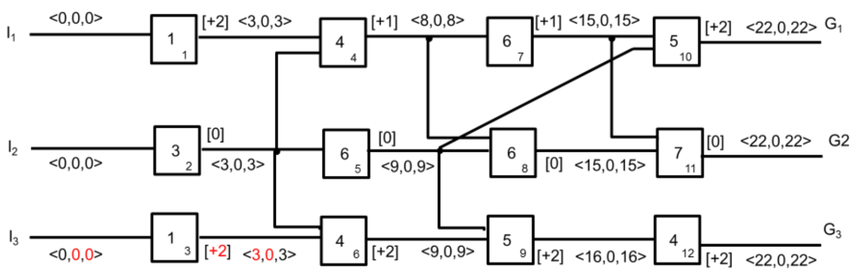
ZSA Example

- … identify



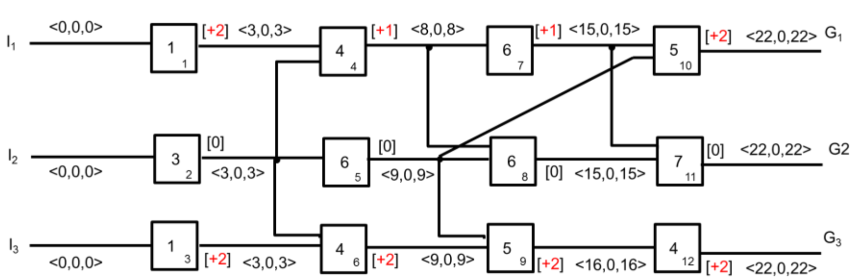
ZSA Example

- … distribute



ZSA Wire Delays

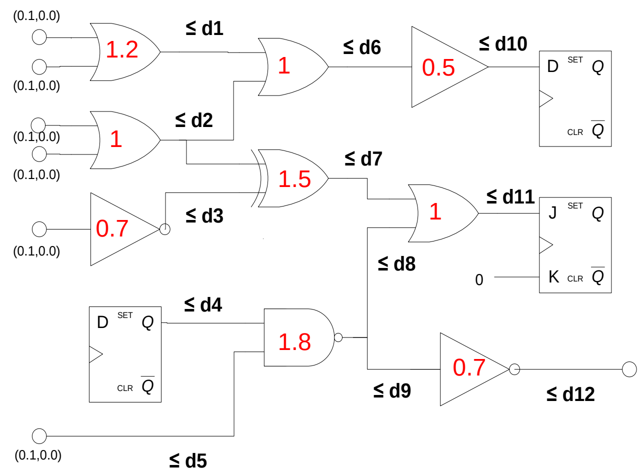
- Wire delays render placement feasible
  - Translate to wire bound constraints



  - This example is infeasible as certain wires have 0 delay
    - Zero WL constraint

ZSA and Bounds

- Wire Bounds correspond to Slack converted to Wire Delay

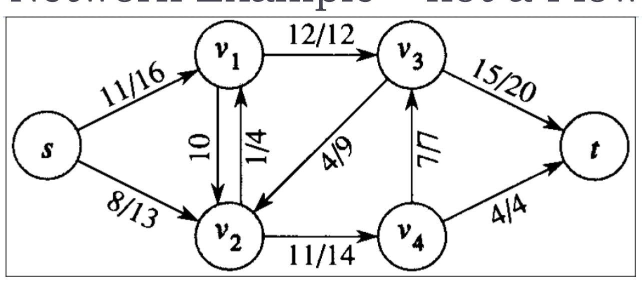


Maximum Flow

Flow Networks and Flows

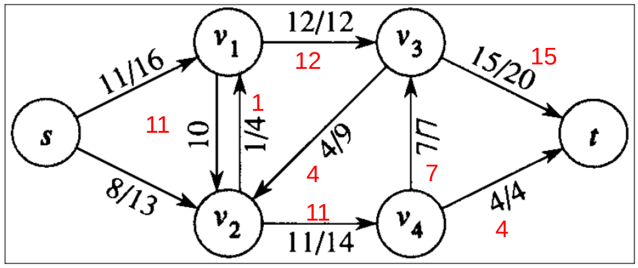
- A **flow network** G = (V, E) is a DAG, where each edge, (u, v) in E has a non-negative capacity c(u, v) >= 0
  - Two vertices are special: a source s, and a sink t
  - Typically, each vertex lies on a source to sink path
- A **flow** in G is a real valued function f: (V x V) → R, s.t.:
  - **Capacity Constraint**: for all u, v in V, f(u, v) <= c(u, v)
  - **Skew Symmetry**: for all u, v in V, f(u, v) = -f(v, u)
  - **Flow Conservation:** for all u in V – {s, t}, $\sum_{v \in V} f(u, v) = 0$
- The quantity f(u, v) is the net flow from u to v
- The value of flow f is defined as: $|f| = \sum_{v \in V} f(s, v)$
  - The total net flow out of the source
- **Maximum Flow**: find flow of maximum value from s to t

Flow Network Example – not a Flow!



- Each edge is labelled with its capacity
- Only positive net flows are shown
- Flow in G is |f| = 19
- Slash notation separates flow and capacity
- **Positive net flow** entering vertex v: $\sum_{u \in V, f(u,v)>0} f(u, v)$

Actual Network Flow



- Flow magnitude |f| = 11 + 8 = 19
- For an actual network flow, Flow Conservation holds
  - e.g. Node v1: (11 + 1 – 12) = 0
  - Node v2: (8 + 4 – 1 – 11) = 0
  - Node v3: (12 + 7 – 4 – 15) = 0
  - Node v4: (11 – 7 – 4) = 0

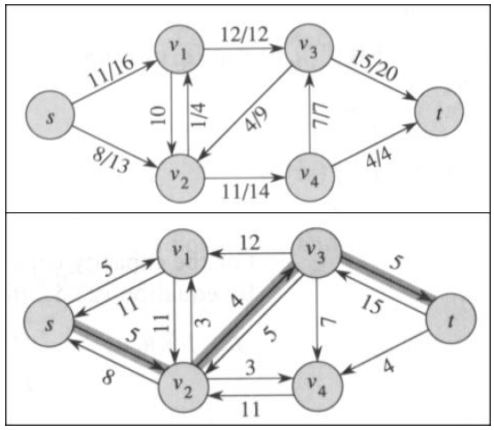
Ford-Fulkerson Method

- **Augmenting Path**: a s to t path through which the flow can be increased

```
FORD-FULKERSON-METHOD(G, s, t)
initialize flow f to 0
while there exists an augmenting path p
      do augment flow f along p
return f
```

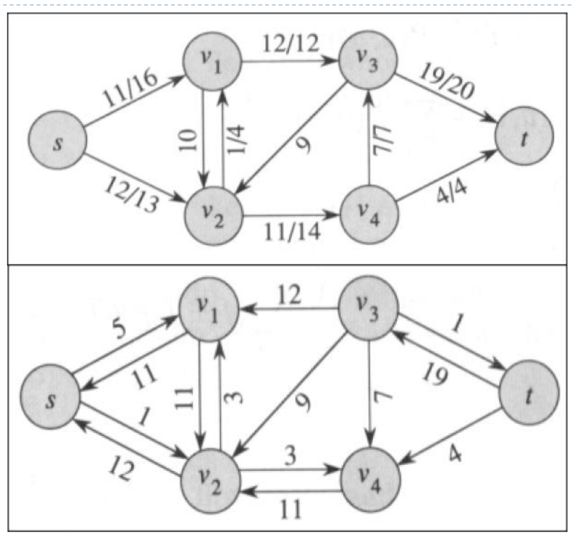
- Residual capacity of (u, v)
  - Additional net flow we can push from u to v <= c(u, v)
  - cf(u, v) = c(u, v) – f(u, v)
- Residual Network G(V, Ef):
  - Ef = {(u, v) in V x V, s.t. cf(u, v) > 0}

Residual Network

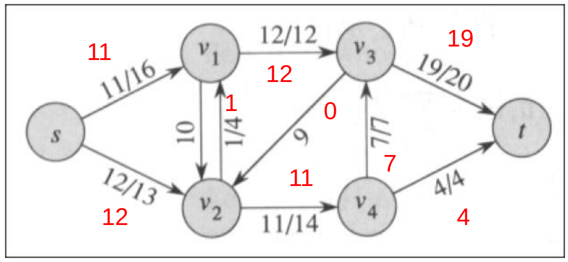


- Residual network of initial flow

Residual Network and Modified Flow



Optimised Network Actual Flow



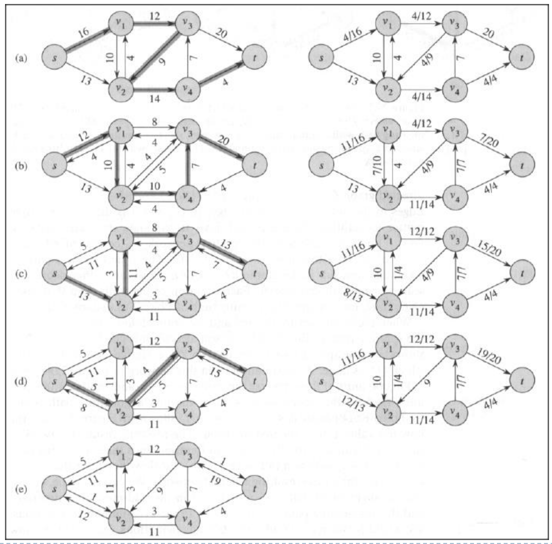
- Flow Magnitude |f| = 11 + 12 = 23
- For an actual network flow, Flow Conservation holds
  - e.g. Node v1: (11 + 1 – 12) = 0
  - Node v2: (12 + 0 – 1 – 11) = 0
  - Node v3: (12 + 7 – 0 – 19) = 0
  - Node v4: (11 – 7 – 4) = 0

Ford-Fulkerson Algorithm

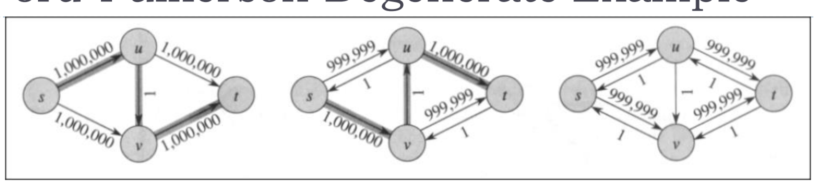
```
FORD-FULKERSON(G, s, t)
for each edge (u, v) ∈ E[G]
    do f[u, v] ← 0
       f[v, u] ← 0
while there exists a path p from s to t in the residual network G_f
    do c_f(p) ← min {c_f(u, v) : (u, v) is in p}
       for each edge (u, v) in p
           do f[u, v] ← f[u, v] + c_f(p)
              f[v, u] ← −f[u, v]
```

- Efficiency depends on augmenting path
- Edmonds-Karp variation
  - Shortest path from s to t, where edge distance is 1
  - $O(VE^2)$ Complexity = O(E x VE) (shortest path)

Ford-Fulkerson Algorithm Execution Example



Ford-Fulkerson Degenerate Example

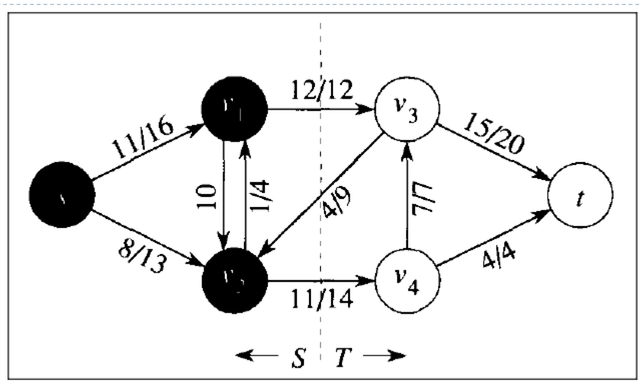


- If we keep adding WC augmenting path of 1 when identifying a path from s to t the algorithm will take O(E x |f*|)
- Use shortest unit edge weight path from s to t

Cuts of Flow Networks

- A cut (S,T) of flow network G = (V, E) is a partition of V into S and T = V – S, such what s is in S and t is in T
  - The netflow across the cut (S,T) is f(S,T)
  - The capacity of the cut (S,T) is c(S,T)
    - Always positive, from S to T
- Max-Flow Min-Cut Theorem
  - If f is a flow in a flow network G = (V, E) with source s and sink t, then the following conditions are equivalent:
    - F is a maximum flow in G
    - The residual network Gf contains no augmenting paths
    - |f| = c(S,T) for some cut (S,T) of G

Cut Example



- Cut across original flow network
  - Net flow across (S,T) is 19 (12 + 11 – 4)
  - Cutsize is 26 (12 + 14)