Network Optimization: Quick Start

@luk036

2022-11-09

---

class: nord-light, middle, center

Introduction

Why and why not

.pull-left[

- Algorithms are available for common network problems (Python: networkx, C++: Boost Graph Library (BGL)):
  - Explore the locality of network.
  - Explore associativity (things can be added up in any order)
- Be able to solve discrete problems optimally (e.g. matching/assignment problems)
- Bonus: gives you insight into the most critical parts of the network (critical cut/cycle)

] .pull-right[

- The theory is hard to understand.
- Algorithms are hard to understand (some algorithms do not allow users to have an input flow in reverse directions, but create edges internally for the reverse flows).
- There are too many algorithms available. You have to choose them wisely.

]

Flow and Potential

.pull-left[

- Cut
- Current
- Flow x
- Sum of x_{ij} around a node = 0

] .pull-right[

- Cycle/Path
- Voltage
- Tension y
- Sum of y_{ij} around a cycle = 0

]

If you don't know more…

- For the min-cost linear flow problem, the best guess is to use the "network simplex algorithm".
- For the min-cost linear potential problem: formulate it as a dual (flow) problem.
- For the parametric potential problem (single parameter), the best guess is to use Howard's algorithm.
- All these algorithms are based on the idea of finding "negative cycle".
- You can apply the same principle to the nonlinear problems.

For dual problems…

- Dual problems can be solved by applying the same principle.
- Finding negative cycles is replaced by finding a negative "cuts", which is more difficult…
- …unless your network is a planar graph.

Guidelines for the average users

- Look for specialized algorithms for specialized problems. For example, for bipartite maximum cardinality matching, use the Hopcroft-Karp matching algorithm.
- Avoid creating edges with infinite costs. Delete them or reformulate your problem.

Guidelines for algorithm developers

- Make “negative cycles” as orthogonal to each other as possible.
- Reuse previous solutions as a new starting point for finding negative cycles.

---

class: nord-light, middle, center

Essential Concepts

Basic elements of a network

Definition (network) A *network* is a collection of finite-dimensional vector spaces, which includes *nodes* and *edges/arcs*:

- $V = \{v_1, v_2, \cdots, v_N\}$, where $|V| = N$
- $E = \{e_1, e_2, e_3, \cdots, e_M\}$ where $|E| = M$

which satisfies 2 requirements:

1. The boundary of each edge is comprised of the union of nodes
2. The intersection of any edges is either empty or the boundary node of both edges.

Network

- By this definition, a network can contain self-loops and multi-edges.
- A *graph* structure encodes the neighborhood information of nodes and edges.
- Note that Python’s NetworkX requires special handling of multi-edges.
- The most efficient graph representation is an adjacency list.
- The concept of a graph can be generalized to *complex*: node, edge, face…

Types of graphs Bipartite graphs, trees, planar graphs, st-graphs, complete graphs.

Orientation

Definition (Orientation) An *orientation* of an edge is an ordering of its boundary node (s, t), where

- s is called a source/initial node
- t is called a target/terminal node

Note: orientation != direction

Definition (Coherent) Two orientations to be the same is called *coherent*

Node-edge Incidence Matrix (connect to algebra!)

Definition (Incidence Matrix) An $N \times M$ matrix A^T is a node-edge incidence matrix with entries:

$$A(i,j) = \begin{cases} +1 & \text{if } e_i \text{ is coherent with the orientation of node } v_j, \\ -1 & \text{if } e_i \text{ is not coherent with the orientation of node } v_j, \\ 0 & \text{otherwise.} \end{cases}$$

Example

$$A^\mathsf{T} = \begin{bmatrix} 0 & -1 & 1 & 1 & 0 \\ 1 & 1 & 0 & -1 & -1 \\ -1 & 0 & -1 & 0 & 1 \end{bmatrix}$$

Chain

Definition (Chain τ) An edge/node *chain* τ is an M/N-tuple of scalar that assigns a coefficient to each edge/node, where M/N is the number of distinct edges/nodes in the network.

Remark (II) A chain may be viewed as an (oriented) indicator vector representing a set of edges/nodes.

Example (II) $[0, 0, 1, 1, 1]$, $[0, 0, 1, -1, 1]$

Discrete Boundary Operator

Definition (Boundary operator) The *boundary* operator $\partial = A^\mathsf{T}$.

Definition (Cycle) A chain is said to be a *cycle* if it is in the null-space of the boundary operator, i.e. $A^\mathsf{T}\tau = 0$.

Definition (Boundary) A chain β is said to be a *boundary* of τ if it is in the range of the boundary operator.

Co-boundary Operator d

Definition (Co-boundary operator) The *co-boundary* (or *differential*) operator $\mathrm{d} = \partial^* = (A^\mathsf{T})^* = A$

Note Null-space of A is #components of a graph

Discrete Stokes' Theorem

- Let
$$\tau_i = \begin{cases} 1 & \text{if } e_i \in S, \\ 0 & \text{otherwise.} \end{cases}$$
- Conventional (integration):
$$\int_S \mathrm{d}\tilde{\omega} = \oint_{\partial S} \tilde{\omega}$$
- Discrete (pairing):
$$[\tau, A\omega] = [A^\mathsf{T}\tau, \omega]$$

Fundamental Theorem of Calculus

- Conventional (integration): $\int_a^b f(t)dt = F(b) - F(a)$
- Discrete (pairing): $[\tau_1, Ac^0] = [A^\mathsf{T}\tau_1, c^0]$

Divergence and Flow

Definition (Divergence) $\operatorname{div} x = A^\mathsf{T} x$

Definition (Flow) x is called a *flow* if $\sum \operatorname{div} x = 0$, where all negative entries of ($\operatorname{div} x$) are called *sources* and positive entries are called *sinks*.

Definition (Circulation) A network is called a *circulation* if there is no source or sink. In other words, $\operatorname{div} x = 0$

Tension and Potential

Definition (Tension) A tension (in co-domain) y is a *differential* of a *potential* u, i.e. $y = Au$.

Theorem (Tellgen's) Flow and tension are bi-orthogonal (isomorphic).

Proof $0 = [A^\mathsf{T} x, u] = (A^\mathsf{T} x)^\mathsf{T} u = x^\mathsf{T}(Au) = x^\mathsf{T} y$

Path

A path indicator vector τ of P that

$$\tau_i = \begin{cases} 1 & \text{if } e_i \in P, \\ 0 & \text{otherwise.} \end{cases}$$

Theorem [total tension y on P] = [total potential on the boundary of P].

Proof $y^\mathsf{T}\tau = (Au)^\mathsf{T}\tau = u^\mathsf{T}(A^\mathsf{T}\tau) = u^\mathsf{T}(\partial P)$.

Cut

Two node sets S and S' (the complement of S, i.e. $V - S$). A cut Q is an edge set, denoted by $[S, S']^-$. A cut indicator vector q (oriented) of Q is defined as Ac where

$$c_i = \begin{cases} 1 & \text{if } v_i \in S\,, \\ 0 & \text{otherwise}\,. \end{cases}$$

Theorem (Stokes' theorem!) [Total divergence of x on S] = [total x across Q].

Proof $(\operatorname{div} x)^\mathsf{T} c = (A^\mathsf{T} x)^\mathsf{T} c = x^\mathsf{T}(Ac) = x^\mathsf{T} q$.

Examples

---

class: nord-light, middle, center

Feasibility Problems

Feasible Flow/Potential Problem

.pull-left[

Feasible Flow Problem

- Find a flow x such that:

$$\begin{array}{l} c^{-} \leq x \leq c^{+}, \\ A^{\mathsf{T}} x = b, b(V) = 0. \end{array}$$

- Can be solved using:
  - Painted network algorithm
  - If no feasible solution, return a "negative cut".

] .pull-right[

Feasible Potential Problem:

- Find a potential u such that:

$$\begin{array}{l} d^{-} \leq y \leq d^{+} \\ A \cdot u = y. \end{array}$$

- Can be solved using:
  - Bellman-Ford algorithm
  - If no feasible solution, return a "negative cycle".

]

Examples

Genome-scale reaction network (primal)

- A: Stoichiometric matrix S
- x: reactions between metabolites/proteins
- $c^{-} \leq x \leq c^{+}$: constraints on reaction rates

Timing constraints (co-domain)

- A^{T}: incidence matrix of timing constraint graph
- u: arrival time of clock
- y: clock skew
- $d^{-} \leq y \leq d^{+}$: setup- and hold-time constraints

---

Feasibility Flow Problem

Theorem (feasibility flow) The problem has a feasible solution if and only if $b(S) \leq c^+(Q)$ for all cuts $Q = [S, S']$ where $c^+(Q)$ = upper capacity [1, p. 56].

Proof (if-part)

Let $q = A \cdot k$ be a cut vector (oriented) of Q. Then

- $c^- \leq x \leq c^+$

–

- $q^\mathsf{T} x \leq c^+(Q)$

–

- $(A \cdot k)^\mathsf{T} x \leq c^+(Q)$

–

- $k^\mathsf{T} A^\mathsf{T} x \leq c^+(Q)$

–

- $k^\mathsf{T} b \leq c^+(Q)$

–

- $b(S) \leq c^+(Q)$

Feasibility Potential Problem

Theorem (feasibility potential) The problem has a feasible solution if and only if $d^+(P) \geq 0$ for all cycles P where $d^+(P)$ = upper span [1, p. ??].

Proof (if-part)

Let τ be a path indicator vector (oriented) of P. Then

- $d^- \leq y \leq d^+$

–

- $\tau^\mathsf{T} y \leq d^+(P)$

–

- $\tau^\mathsf{T} (A \cdot u) \leq d^+(P)$

–

- $(A^{\mathsf{T}}\tau)^{\mathsf{T}} u \leq d^+(P)$

–

- $(\partial P)^{\mathsf{T}} u \leq d^+(P)$

–

- $0 \leq d^+(P)$

Remarks

- The only-if part of the proof is constructive. It can be done by constructing an algorithm to obtain the feasible solution.
- d^+ could be ∞ or zero, etc.
- d^- could be $-\infty$ or zero, etc.
- c^+ could be ∞ or zero, etc.
- c^- could be $-\infty$ or zero, etc.

Note: most tools require that c^- must be zero such that the solution flow x is always positive.

Convert to the elementary problem

.pull-left[

- By splitting every edge into two, the feasibility flow problem can reduce to an elementary one:
  - Find a flow x such that

$$\begin{array}{l} c \leq x, \\ A_1^{\mathsf{T}} x = b_1, \\ b_1(V_1) = 0. \end{array}$$

    where A_1 is the incident matrix of the modified network.

] .pull-right[

Original:

Modified:

]

Convert to the elementary problem

.pull-left[

- By adding a reverse edge for every edge, the feasibility potential problem can reduce to an elementary one:
  - Find a potential u such that

    $$\begin{array}{l} y_2 \leq d, \\ A_2 u = y_2 \end{array}$$

    where A_2 is the incident matrix of the modified network.

] .pull-right[

Original:

Modified:

]

Basic Bellman-Ford Algorithm

.font-sm.mb-xs[

```
function BellmanFord(list vertices, list edges, vertex source)
   // Step 1: initialize graph
   for each vertex i in vertices:
       if i is source then u[i] := 0
       else u[i] := inf
       predecessor[i] := null

   // Step 2: relax edges repeatedly
   for i from 1 to size(vertices)-1:
       for each edge (i, j) with weight d in edges:
           if u[j] > u[i] + d[i,j]:
               u[j] := u[i] + d[i,j]
               predecessor[j] := i

   // Step 3: check for negative-weight cycles
   for each edge (i, j) with weight d in edges:
       if u[j] > u[i] + d[i,j]:
           error "Graph contains a negative-weight cycle"
   return u[], predecessor[]
```

]

.font-sm.mb-xs[

```python
def _neg_cycle_relaxation(G, pred, dist, source, weight):
    G_succ = G.succ if G.is_directed() else G.adj
    inf = float('inf')
    n = len(G)
    count = {}
    q = deque(source)
    in_q = set(source)
    while q:
        u = q.popleft()
        in_q.remove(u)
        if pred[u] not in in_q:
            dist_u = dist[u]
            for v, e in G_succ[u].items():
                dist_v = dist_u + get_weight(e)
*                if dist_v < dist.get(v, inf):
                    if v not in in_q:
                        q.append(v)
                        in_q.add(v)
                        count_v = count.get(v, 0) + 1
                        if count_v == n:
                            return v
                        count[v] = count_v
                    dist[v] = dist_v
                    pred[v] = u
    return None
```

]

Example 1 : Clock skew scheduling

- Goal: intentionally assign an arrival time u_i to each register so that the setup and hold time constraints are satisfied.
- Note: the clock skew $s_{ij} = u_i - u_j$ is more important than the arrival time u itself, because the clock runs periodically.
- In the early stages, fixing the timing violation could be done as soon as a negative cycle is detected. A complete timing analysis is unnecessary at this stage.

Example 2 : Delay padding + clock skew scheduling

- Goal: intentionally "insert" a delay p so that the setup and hold time constraints are satisfied.

- Note that a delay can be “inserted” by swapping a fast transistor into a slower transistor.
- Traditional problem formulation: Find p and u such that

$$\begin{array}{l} y \leq d + p, \\ Au = y, p \geq 0 \end{array}$$

- Note 1: Inserting delays into some local paths may not be allowed.
- Note 2: The problem can be reduced to the standard form by modifying the network (timing constraint graph)

Four possible ways to insert delay

.pull-left[

No delay:

$p_s = p_h$:

] .pull-right[

Independent:

$p_s \geq p_h$:

]

Remarks (III)

- If there exists a negative cycle, it means that timing cannot be fixed using simply this technique.
- Additional constraints, such as $p_s \leq p_{\max}$, can be imposed.

---

class: nord-light, middle, center

Parametric Problems

Parametric Potential Problem (PPP)

- Consider a parameter potential problem:

$$\begin{array}{ll} \text{maximize} & \beta \\ \text{subject to} & y \leq d(\beta), \\ & A \cdot u = y \end{array}$$

  where $d(\beta)$ is a *monotonic decreasing* function.

- If $d(\beta)$ is a linear function $(m - s\beta)$ where s is non-negative, the problem reduces to the well-known *minimum cost-to-time ratio problem*.
- If s = constant, it further reduces to the *minimum mean cycle problem*.

Note: Parametric flow problem can be defined similarly.

Examples (III)

- $d(\beta)$ is linear $(m - s\beta)$:
  - Optimal clock period scheduling problem
  - Slack maximization problem
  - Yield-driven clock skew scheduling (Gaussian)
- $d(\beta)$ is non-linear:
  - Yield-driven clock skew scheduling (non-Gaussian)
  - Multi-domain clock skew scheduling

Examples (IV)

- Lawler's algorithm (binary search based)
- Howard's algorithm (cycle cancellation)
- Young's algorithm (path based)
- Burns' algorithm (path based)
  - for clock period optimization problem (all elements of s are either 0 or 1)
- Several hybrid methods have also been proposed

Remarks (IV)

- Need to solve feasibility problems many times.
- Data structures, such as Fibonacci heap or spanning tree/forest, can be used to improve efficiency
- For multi-parameter problems, the *ellipsoid method* can be used.
- Example 1: yield-driven clock skew scheduling (c.f. lecture 5)

Example 2: yield-driven delay padding

- The problem can be reduced to the standard form by modifying the underlying constraint graph.

Four possible way to insert delay

.pull-left[

No delay:

$p_s = p_h$:

] .pull-right[

Independent:

$p_s \geq p_h$:

]

---

class: nord-light, middle, center

Min-cost Flow/Potenial Problem

Elementary Optimal Problems

- Elementary Flow Problem:

$$\begin{array}{ll} \min & d^{\mathsf{T}} x + p \\ \text{s. t.} & c \leq x, \\ & A^{\mathsf{T}} x = b, \; b(V) = 0 \end{array}$$

- Elementary Potential Problem:

$$\begin{array}{ll} \max & b^\mathsf{T} u - (c^\mathsf{T} y + q) \\ \text{s. t.} & y \le d, \\ & Au = y \end{array}$$

Elementary Optimal Problems (Cont'd)

- The problems are dual to each other if $p + q = -c^\mathsf{T} d, (x - c)^\mathsf{T}(d - y) = 0, c \le x, y \le d$
- Since $b^\mathsf{T} u = (A^\mathsf{T} x)^\mathsf{T} u = x^\mathsf{T} A u = x^\mathsf{T} y$, $[\min] - [\max] = (d^\mathsf{T} x + p) - (b^\mathsf{T} u - [c^\mathsf{T} y + q]) = d^\mathsf{T} x + c^\mathsf{T} y - x^\mathsf{T} y + p + q = (x - c)^\mathsf{T}(d - y) \ge 0$
- $[\min] - [\max]$ when equality holds.

Remark (V)

- We can formulate a linear problem in primal or dual form, depending on which solution method is more appropriate:
  - Incremental improvement of feasible solutions
  - Design variables are in the integral domain:
    * The max-flow problem (i.e. $d^\mathsf{T} = [-1, -1, \cdots, -1]^\mathsf{T}$) may be better solved by the dual method.

Linear Optimal Problems

- Optimal Flow Problem:

$$\begin{array}{ll} \min & d^\mathsf{T} x + p \\ \text{s. t.} & c^- \le x \le c^+, \\ & A^\mathsf{T} x = b, \; b(V) = 0 \end{array}$$

- Optimal Potential Problem:

$$\begin{array}{ll} \max & b^\mathsf{T} u - (c^\mathsf{T} y + q) \\ \text{s. t.} & d^- \le y \le d^+, \\ & Au = y \end{array}$$

Linear Optimal Problems (II)

By modifying the network:

- The problem can be reduced to the elementary case [pp.275-276]

piece of cake

- Piece-wise linear convex cost can be reduced to this linear problem [p.239,p.260]

The problem has been extensively studied and has numerous applications.

Remark (VI)

- We can transform the cost function to be non-negative by reversing the orientation of the negative cost edges.
- Then reduce the problem to the elementary case (or should we???)

Algorithms for Optimal Flow Problems

- Successive shortest path algorithm
- Cycle cancellation method
  - Iteratively insert additional minimal flows according to a negative cycle of the residual network until no negative cycles are found.
- Scaling method

For Special Cases

- Max-flow problem ($d = -[1, \cdots, 1]$)
  - Ford-Fulkerson algorithm: iteratively insert additional minimal flows according to an augmented path of the residual network, until no augmented paths of the residual network are found.
  - Pre-flow Push-Relabel algorithm (dual method???)
- Matching problems ($[c^-, c^+] = [0, 1]$)
  - Edmond's blossom algorithm

Min-Cost Flow Problem (MCFP)

- Problem Formulation:

$$\begin{array}{ll} \min & d^\mathsf{T} x \\ \text{s. t.} & 0 \leq x \leq c, \\ & A^\mathsf{T} x = b,\ b(V) = 0 \end{array}$$

- Algorithm idea: descent method: given a feasible x_0, find a better solution $x_1 = x_0 + \alpha p$, where α is positive.

General Descent Method

- **Input**: $f(x)$, initial x
- **Output**: optimal opt x^*
- **while** not converged,
  1. Choose descent direction p;
  2. Choose the step size α;
  3. $x := x + \alpha p$;

Some Common Descent Directions

- Gradient descent: $p = -\nabla f(x)^\mathsf{T}$
- Steepest descent:
  - $\triangle x_{nsd} = \text{argmin}\{\nabla f(x)^\mathsf{T} v \mid \|v\| = 1\}$
  - $\triangle x_{sd} = \|\nabla f(x)\| \triangle x_{nsd}$ (un-normalized)
- Newton's method: $p = -\nabla^2 f(x)^{-1} \nabla f(x)$
- For convex problems, must satisfy $\nabla f(x)^\mathsf{T} p < 0$.

Note: Here, there is a natural way to choose p!

Min-Cost Flow Problem (II)

- Let $x_1 = x_0 + \alpha p$, then we have:

$$\begin{array}{lll} \min & d^\mathsf{T} x_0 + \alpha d^\mathsf{T} p & \Rightarrow d^\mathsf{T} p < 0 \\ \text{s. t.} & -x_0 \leq \alpha p \leq c - x_0 & \Rightarrow \text{residual graph} \\ & A^\mathsf{T} p = 0 & \Rightarrow p \text{ is a cycle!} \end{array}$$

- In other words, choose p to be a negative cycle!
  - Simple negative cycle, or
  - Minimum mean cycle

Primal Method for MCFP

- **Input**: $G(V, E), [c^-, c^+], d$
- **Output**: optimal opt x^*
- Initialize a feasible x and certain data structure
- **while** a negative cycle p found in $G(x)$,
  1. Choose a step size α;
  2. **If** α is unbounded, **return** UNBOUNDED;
  3. **If** $\alpha = 0$, **break**;
  4. $x := x + \alpha p$;
  5. Update corresponding data structures
- **return** OPTIMAL

Remarks (VI)

- In Step 4, negative cycle can be found using Bellman-Ford algorithm.
- In the cycle cancelling algorithm, p is:
  - a simple negative cycle, or
  - a minimum mean cycle
- A heap or other data structures are used for finding negative cycles efficiently.
- Usually α is chosen such that one constraint is tight.

Min-Cost Potential Problem (MCPP)

- Problem Formulation:
$$\begin{array}{ll} \min & c^\mathsf{T} y \\ \text{s. t.} & y \leq d, \\ & Au = y \end{array}$$
  where c is assumed to be non-negative.
- Algorithm: given an initial feasible u_0, find a better solution $u_1 = u_0 + \beta q$, where β is positive:
$$\begin{array}{lll} \min & c^\mathsf{T} y_0 + c^\mathsf{T} y & \Rightarrow c^\mathsf{T} y < 0 \\ \text{s. t.} & y \leq d - Au_0 & \Rightarrow \text{residual graph} \\ & \beta A q = y & \Rightarrow q \text{ is a ``cut''!} \end{array}$$

Method for MCPP

- **Input**: $G(V, E), c, d$
- **Output**: optimal opt u^*
- Initialize a feasible u and certain data structure
- **while** a negative cut q found in $G(u)$,
  1. Choose a step size β;
  2. **If** β is unbounded, **return** UNBOUNDED;
  3. **If** $\beta = 0$, **break**;
  4. $u := u + \beta q$;
  5. Update corresponding data structures
- **return** OPTIMAL

Remarks (VII)

- Usually β is chosen such that one constraint is tight.
- The min-cost potential problem is the dual of the min-cost flow problem, so algorithms can solve both problems.
- In the network simplex method, q is chosen from a spanning tree data structure (for linear problems only)

E.g. Delay Padding

- Consider the following problem:

$$\begin{array}{ll} \min & c^{\mathsf{T}} p, \\ \text{s.t.} & y \leq d + p, \\ & Au = y,\ p \geq 0 \end{array}$$

  where p: delay padding

- Its dual is:

$$\begin{array}{ll} \min & d^{\mathsf{T}} x \\ \text{s.t.} & 0 \leq x \leq c, \\ & A^{\mathsf{T}} x = 0 \end{array}$$

---

count: false class: nord-dark, middle, center

Q & A

```
</textarea>
<script src="../js/remark.min.js"></script>
<script src="../katex/katex.min.js" type="text/javascript"></script>
```

```
<script
  src="../katex/contrib/auto-render.min.js"
  type="text/javascript"
></script>
<script type="text/javascript">
  renderMathInElement(document.getElementById("source"), {
    delimiters: [
      { left: "$$", right: "$$", display: true },
      { left: "$", right: "$", display: false },
    ],
  });
  var slideshow = remark.create({
    ratio: "4:3", //
    //   arta, ascetic, dark, default, far, github, googlecode, idea,
    // ir-black, magula, monokai, rainbow, solarized-dark, solarized-light,
    // sunburst, tomorrow, tomorrow-night-blue, tomorrow-night-bright,
    // tomorrow-night, tomorrow-night-eighties, vs, zenburn.
    highlightStyle: "monokai",
    highlightLines: true,
    countIncrementalSlides: false, //
    // slideNumberFormat: "", //      ""
    navigation: {
      scroll: false, //
      touch: true, //
      click: false, //
    },
  });
</script>
```

<!DOCTYPE html>

When “Convex Optimization” Meets “Network Flow”

layout: true class: typo, typo-selection

---

count: false class: nord-dark, middle, center

When “Convex Optimization” Meets “Network Flow”

@luk036

2022-11-16

---

class: nord-light, middle, center

Introduction

Overview

- Network flow problems can be solved efficiently and have a wide range of applications.
- Unfortunately, some problems may have other additional constraints that make them impossible to solve with current network flow techniques.
- In addition, in some problems, the objective function is quasi-convex rather than convex.
- In this lecture, we will investigate some problems that can still be solved by network flow techniques with the help of convex optimization.

---

class: nord-light, middle, center

Parametric Potential Problems

Parametric potential problems

Consider:

$$\begin{array}{ll} \text{maximize} & g(\beta), \\ \text{subject to} & y \leq d(\beta), \\ & Au = y, \end{array}$$

where $g(\beta)$ and $d(\beta)$ are concave.

Note: the parametric flow problems can be defined in a similar way.

Network flow says:

- For fixed β, the problem is feasible precisely when there exists no negative cycle
- Negative cycle detection can be done efficiently using the Bellman-Ford-like methods
- If a negative cycle C is found, then $\sum_{(i,j)\in C} d_{ij}(\beta) < 0$

Convex Optimization says:

- If both sub-gradients of $g(\beta)$ and $d(\beta)$ are known, then the *bisection method* can be used for solving the problem efficiently.
- Also, for multi-parameter problems, the *ellipsoid method* can be used.

Quasi-convex Minimization

Consider:

$$\begin{array}{ll} \text{maximize} & f(\beta), \\ \text{subject to} & y \leq d(\beta), \\ & Au = y, \end{array}$$

where $f(\beta)$ is *quasi-convex* and $d(\beta)$ are concave.

Example of Quasi-Convex Functions

- $\sqrt{|y|}$ is quasi-convex on $\mathbb{R}$
- $\log(y)$ is quasi-linear on $\mathbb{R}_{++}$
- $f(x, y) = xy$ is quasi-concave on $\mathbb{R}^2_{++}$
- Linear-fractional function:
  - $f(x) = (a^\mathsf{T} x + b)/(c^\mathsf{T} x + d)$
  - $\operatorname{dom} f = \{x \mid c^\mathsf{T} x + d > 0\}$
- Distance ratio function:
  - $f(x) = \|x - a\|_2 / \|x - b\|_2$
  - $\operatorname{dom} f = \{x \mid \|x - a\|_2 \leq \|x - b\|_2\}$

Convex Optimization says:

If f is quasi-convex, there exists a family of functions ϕ_t such that:

- $\phi_t(\beta)$ is convex w.r.t. β for fixed t
- $\phi_t(\beta)$ is non-increasing w.r.t. t for fixed β
- t-sublevel set of f is 0-sublevel set of ϕ_t, i.e., $f(\beta) \leq t$ iff $\phi_t(\beta) \leq 0$

For example:

- $f(\beta) = p(\beta)/q(\beta)$ with p convex, q concave $p(\beta) \geq 0$, $q(\beta) > 0$ on dom f,
- can take $\phi_t(\beta) = p(\beta) - t \cdot q(\beta)$

Convex Optimization says:

Consider a convex feasibility problem:

$$\begin{array}{ll} \text{find} & f(\beta), \\ \text{s. t.} & \phi_t(\beta) \leq 0, \\ & y \leq d(\beta), Au = y, \end{array}$$

- If feasible, we conclude that $t \geq p^*$;
- If infeasible, $t < p^*$.

Binary search on t can be used for obtaining p^*.

Quasi-convex Network Problem

- Again, the feasibility problem ([eq:quasi]) can be solved efficiently by the bisection method or the ellipsoid method, together with the negatie cycle detection technique.
- Any EDA's applications ???

Monotonic Minimization

- Consider the following problem:

  $$\begin{array}{ll} \text{minimize} & \max_{ij} f_{ij}(y_{ij}), \\ \text{subject to} & Au = y, \end{array}$$

  where $f_{ij}(y_{ij})$ is non-decreasing.
- The problem can be recast as:

  $$\begin{array}{ll} \text{maximize} & \beta, \\ \text{subject to} & y \leq f^{-1}(\beta), \\ & Au = y, \end{array}$$

  where $f^{-1}(\beta)$ is non-deceasing w.r.t. β.

E.g. Yield-driven Optimization

- Consider the following problem:

$$\begin{array}{ll} \text{maximize} & \min_{ij} \Pr(y_{ij} \leq \tilde{d}_{ij}) \\ \text{subject to} & Au = y, \end{array}$$

  where $\tilde{d}_{ij}$ is a random variables.

- Equivalent to the problem:

$$\begin{array}{ll} \text{maximize} & \beta, \\ \text{subject to} & \beta \leq \Pr(y_{ij} \leq \tilde{d}_{ij}), \\ & Au = y, \end{array}$$

  where $f_{ij}^{-1}(\beta)$ is non-deceasing w.r.t. β.

E.g. Yield-driven Optimization (II)

- Let $F(x)$ is the cdf of $\tilde{d}$.
- Then:

$$\begin{array}{ll} & \beta \leq \Pr(y_{ij} \leq \tilde{d}_{ij}) \leq t \\ \Rightarrow & \beta \leq 1 - F_{ij}(y_{ij}) \\ \Rightarrow & y_{ij} \leq F_{ij}^{-1}(1 - \beta) \end{array}$$

- The problem becomes:

$$\begin{array}{ll} \text{maximize} & \beta, \\ \text{subject to} & y_{ij} \leq F_{ij}^{-1}(1 - \beta), \\ & Au = y, \end{array}$$

Network flow says

- Monotonic problem can be solved efficiently using cycle-cancelling methods such as Howard's algorithm.

---

class: nord-light, middle, center

Min-cost flow problems

Min-Cost Flow Problem (linear)

Consider:

$$\begin{array}{ll} \min & d^\mathsf{T} x + p \\ \text{s. t.} & c^- \leq x \leq c^+, \\ & A^\mathsf{T} x = b, \; b(V) = 0 \end{array}$$

- some c^+ could be $+\infty$ some c^- could be $-\infty$.
- A^T is the incidence matrix of a network G.

Conventional Algorithms

- Augmented-path based:
  - Start with an infeasible solution
  - Inject minimal flow into the augmented path while maintaining infeasibility in each iteration
  - Stop when there is no flow to inject into the path.
- Cycle cancelling based:
  - Start with a feasible solution x_0
  - find a better sol'n $x_1 = x_0 + \alpha \triangle x$, where α is positive and $\triangle x$ is a negative cycle indicator.

General Descent Method

1. **Input**: a starting $x \in \operatorname{dom} f$
2. **Output**: x^*
3. **repeat**
   1. Determine a descent direction p.
   2. Line search. Choose a step size $\alpha > 0$.
   3. Update. $x := x + \alpha p$
4. **until** a stopping criterion is satisfied.

Some Common Descent Directions

- For convex problems, the search direction must satisfy $\nabla f(x)^\mathsf{T} p < 0$.
- Gradient descent:
  - $p = -\nabla f(x)^\mathsf{T}$
- Steepest descent:
  - $\triangle x^{nsd} = \operatorname{argmin}\{\nabla f(x)^\mathsf{T} v \mid \|v\| = 1\}$.
  - $\triangle x^{sd} = \|\nabla f(x)\| \triangle x^{nsd}$ (un-normalized)
- Newton's method:

- $p = -\nabla^2 f(x)^{-1} \nabla f(x)$

Network flow says (II)

- Here, there is a better way to choose p!
- Let $x := x + \alpha p$, then we have:

$$\begin{array}{lll} \min & d^\mathsf{T} x_0 + \alpha d^\mathsf{T} p & \Rightarrow d^\mathsf{T} < 0 \\ \text{s. t.} & -x_0 \leq \alpha p \leq c - x_0 & \Rightarrow \text{residual graph} \\ & A^\mathsf{T} p = 0 & \Rightarrow p \text{ is a cycle!} \end{array}$$

- In other words, choose p to be a negative cycle with cost d!
  - Simple negative cycle, or
  - Minimum mean cycle

Network flow says (III)

- Step size is limited by the capacity constraints:
  - $\alpha_1 = \min_{ij}\{c^+ - x_0\}$, for $\triangle x_{ij} > 0$
  - $\alpha_2 = \min_{ij}\{x_0 - c^-\}$, for $\triangle x_{ij} < 0$
  - $\alpha_{\text{lin}} = \min\{\alpha_1, \alpha_2\}$
- If $\alpha_{\text{lin}} = +\infty$, the problem is unbounded.

Network flow says (IV)

- An initial feasible solution can be obtained by a similar construction of the residual graph and cost vector.
- The LEMON package implements this cycle cancelling algorithm.

Min-Cost Flow Convex Problem

- Problem Formulation:

$$\begin{array}{ll} \min & f(x) \\ \text{s. t.} & 0 \leq x \leq c, \\ & A^\mathsf{T} x = b, \; b(V) = 0 \end{array}$$

Common Types of Line Search

- Exact line search: $t = \text{argmin}_{t>0} f(x + t\triangle x)$

- Backtracking line search (with parameters $\alpha \in (0, 1/2), \beta \in (0, 1)$)
  - starting from $t = 1$, repeat $t := \beta t$ until

$$f(x + t\triangle x) < f(x) + \alpha t \nabla f(x)^{\mathsf{T}} \triangle x$$

  - graphical interpretation: backtrack until $t \leq t_0$

Network flow says (V)

- The step size is further limited by the following:
  - $\alpha_{\text{cvx}} = \min\{\alpha_{\text{lin}}, t\}$
- In each iteration, choose $\triangle x$ as a negative cycle of G_x, with cost $\nabla f(x)$ such that $\nabla f(x)^{\mathsf{T}} \triangle x < 0$

Quasi-convex Minimization (new)

- Problem Formulation:

$$\begin{array}{ll} \min & f(x) \\ \text{s. t.} & 0 \leq x \leq c, \\ & A^{\mathsf{T}} x = b, \; b(V) = 0 \end{array}$$

- The problem can be recast as:

$$\begin{array}{ll} \min & t \\ \text{s. t.} & f(x) \leq t, \\ & 0 \leq x \leq c, \\ & A^{\mathsf{T}} x = b, \; b(V) = 0 \end{array}$$

Convex Optimization says (II)

- Consider a convex feasibility problem:

$$\begin{array}{ll} \text{find} & x \\ \text{s. t.} & \phi_t(x) \leq 0, \\ & 0 \leq x \leq c, \\ & A^{\mathsf{T}} x = b, \; b(V) = 0 \end{array}$$

  - If feasible, we conclude that $t \geq p^*$;
  - If infeasible, $t < p^*$.
- Binary search on t can be used for obtaining p^*.

Network flow says (VI)

- Choose $\triangle x$ as a negative cycle of G_x with cost $\nabla \phi_t(x)$
- If no negative cycle is found, and $\phi_t(x) > 0$, we conclude that the problem is infeasible.
- Iterate until x becomes feasible, i.e. $\phi_t(x) \leq 0$.

E.g. Linear-Fractional Cost

- Problem Formulation:

$$\begin{array}{ll} \min & (e^\mathsf{T} x + f)/(g^\mathsf{T} x + h) \\ \text{s. t.} & 0 \leq x \leq c, \\ & A^\mathsf{T} x = b,\ b(V) = 0 \end{array}$$

- The problem can be recast as:

$$\begin{array}{ll} \min & t \\ \text{s. t.} & (e^\mathsf{T} x + f) - t(g^\mathsf{T} x + h) \leq 0 \\ & 0 \leq x \leq c, \\ & A^\mathsf{T} x = b,\ b(V) = 0 \end{array}$$

Convex Optimization says (III)

- Consider a convex feasibility problem:

$$\begin{array}{ll} \text{find} & x \\ \text{s. t.} & (e - t \cdot g)^\mathsf{T} x + (f - t \cdot h) \leq 0, \\ & 0 \leq x \leq c, \\ & A^\mathsf{T} x = b,\ b(V) = 0 \end{array}$$

  - If feasible, we conclude that $t \geq p^*$;
  - If infeasible, $t < p^*$.
- Binary search on t can be used for obtaining p^*.

Network flow says (VII)

- Choose $\triangle x$ to be a negative cycle of G_x with cost $(e - t \cdot g)$, i.e. $(e - t \cdot g)^\mathsf{T} \triangle x < 0$
- If no negative cycle is found, and $(e - t \cdot g)^\mathsf{T} x_0 + (f - t \cdot h) > 0$, we conclude that the problem is infeasible.
- Iterate until $(e - t \cdot g)^\mathsf{T} x_0 + (f - t \cdot h) \leq 0$.

E.g. Statistical Optimization

- Consider the quasi-convex problem:

$$\begin{array}{ll} \min & \Pr(\mathbf{d}^\top x > \alpha) \\ \text{s. t.} & 0 \le x \le c, \\ & A^\top x = b,\ b(V) = 0 \end{array}$$

  - $\mathbf{d}$ is random vector with mean d and covariance Σ.
  - Hence, $\mathbf{d}^\top x$ is a random variable with mean $d^\top x$ and variance $x^\top \Sigma x$.

Statistical Optimization

- The problem can be recast as:

$$\begin{array}{ll} \min & t \\ \text{s. t.} & \Pr(\mathbf{d}^\top x > \alpha) \le t \\ & 0 \le x \le c, \\ & A^\top x = b,\ b(V) = 0 \end{array}$$

Note:

$$\begin{array}{ll} & \Pr(\mathbf{d}^\top x > \alpha) \le t \\ \Rightarrow & d^\top x + F^{-1}(1-t)\|\Sigma^{1/2} x\|_2 \le \alpha \end{array}$$

(convex quadratic constraint w.r.t x)

Recall...

Recall that the gradient of $d^\top x + F^{-1}(1-t)\|\Sigma^{1/2} x\|_2$ is $d + F^{-1}(1-t)(\|\Sigma^{1/2} x\|_2)^{-1}\Sigma x$.

Problem w/ additional Constraints (new)

- Problem Formulation:

$$\begin{array}{ll} \min & f(x) \\ \text{s. t.} & 0 \le x \le c, \\ & A^\top x = b,\ b(V) = 0 \\ & s^\top x \le \gamma \end{array}$$

E.g. Yield-driven Delay Padding

- Consider the following problem:

$$\begin{array}{ll} \text{maximize} & \gamma\,\beta - c^{\mathsf{T}} p, \\ \text{subject to} & \beta \leq \Pr(y_{ij} \leq \mathbf{d}_{ij} + p_{ij}), \\ & Au = y,\ p \geq 0 \end{array}$$

  - p: delay padding
  - γ: weight (determined by a trade-off curve of yield and buffer cost)
  - $\mathbf{d}_{ij}$: Gaussian random variable with mean d_{ij} and variance s_{ij}.

E.g. Yield-driven Delay Padding (II)

.pull-left[

- The problem is equivalent to:

$$\begin{array}{ll} \max & \gamma\,\beta - c^{\mathsf{T}} p, \\ \text{s.t.} & y \leq d - \beta s + p, \\ & Au = y, p \geq 0 \end{array}$$

]

.pull-right[

- or its dual:

$$\begin{array}{ll} \min & d^{\mathsf{T}} x \\ \text{s.t.} & 0 \leq x \leq c, \\ & A^{\mathsf{T}} x = b,\ b(V) = 0 \\ & s^{\mathsf{T}} x \leq \gamma \end{array}$$

]

Recall ...

- Yield drive CSS:

$$\begin{array}{ll} \max & \beta, \\ \text{s.t.} & y \leq d - \beta s, \\ & Au = y, \end{array}$$

- Delay padding

$$\begin{array}{ll} \max & -c^{\mathsf{T}} p, \\ \text{s.t.} & y \leq d + p, \\ & Au = y,\ p \geq 0 \end{array}$$

Considering Barrier Method

- Approximation via logarithmic barrier:

$$\begin{array}{ll} \min & f(x) + (1/t)\phi(x) \\ \text{s.t.} & 0 \le x \le c, \\ & A^{\mathsf{T}} x = b,\ b(V) = 0 \end{array}$$

  - where $\phi(x) = -\log(\gamma - s^{\mathsf{T}} x)$
  - Approximation improves as $t \to \infty$
  - Here, $\nabla\phi(x) = s/(\gamma - s^{\mathsf{T}} x)$

Barrier Method

- **Input**: a feasible x, $t := t^{(0)}$, $\mu > 1$, tolerance $\varepsilon > 0$
- **Output**: x^*
- **repeat**
  1. Centering step. Compute $x^*(t)$ by minimizing $t\,f + \phi$
  2. Update $x := x^*(t)$.
  3. Increase t. $t := \mu t$
- **until** $1/t < \varepsilon$.

Note: Centering is usually done by Newton's method in general.

Network flow says (VIII)

In the centering step, instead of using the Newton descent direction, we can replace it with a negative cycle on the residual graph.

Introduction

Useful Skew Design: Why vs. Why Not

Why not

Some common challenges when implementing useful skew design include:

- need more engineer training
- difficulty in building a balanced clock-tree
- uncertainty in how to handle process variation and multi-corner multi-mode issues …, etc.

Why

If these challenges are overcome and useful skew design is implemented correctly,

- it can lead to less time spent on timing issues
- get better chip performance or yield

Clock Arrival Time vs. Clock Skew

- Clock signal runs periodically.
- Thus, absolute clock arrival time u_i is not so important.
- Instead, the skew $y_{ij} = u_i - u_j$ is more important in this scenario.

Useful Skew Design vs. Zero-Skew Design

- "Critical cycle" instead of "critical path".
- "Negative cycle" instead of "negative slack".
- If there is a negative cycle, it means that there is no positive slack solution no matter how to schedule.
- Others are pretty much the same.
- Same design principle:
  - Always tackle the most critical one first!

Linear Programming vs. Network Flow Formulation

- Linear programming formulation
  - can handle more complex constraints
- Network flow formulation
  - usually more efficient
  - return the most critical cycle as a bonus
  - can handle quantized buffer delay (???)
- Anyway, timing analysis is much more time-consuming than the optimization solving.

Target Skew vs. Actual Skew

Don't mess up these two concepts:

- Target skew:
  - the skew we want to achieve in the scheduling stage.

  - Usually deterministic (we schedule a meeting at 10:00, rather than 10:00 $\pm$ 34 minutes, right?)
- Actual skew
  - the skew that the clock tree actually generates.
  - Can be formulated as a random variable.

A Simple Case

To warm up, let us start with a simple case:

- Assume equal path delay variations.
- Single-corner.
- Before a clock tree is built.
- No adjustable delay buffer (ADB).

Network

Definition (Network)

A *network* is a collection of finite-dimensional vector spaces of *nodes* and *edges/arcs*:

- $V = \{v_1, v_2, \cdots, v_N\}$, where $|V| = N$
- $E = \{e_1, e_2, e_3, \cdots, e_M\}$ where $|E| = M$

which satisfies 2 requirements:

1. The boundary of each edge is comprised of the union of nodes
2. The intersection of any edges is either empty or a boundary node of both edges.

Orientation

Definition (Orientation)

An *orientation* of an edge is an ordering of its boundary node (s, t), where

- s is called a source/initial node
- t is called a target/terminal node

Definition (Coherent)

Two orientations to be the same is called *coherent*

Node-edge Incidence Matrix

Definition (Incidence Matrix)

A $N \times M$ matrix A^T is a node-edge incidence matrix with entries:

$$A(i,j) = \begin{cases} +1 & \text{if } e_i \text{ is coherent with } v_j, \\ -1 & \text{if } e_i \text{ is not coherent with } v_j, \\ 0 & \text{otherwise.} \end{cases}$$

Example (II)

$$A^\mathsf{T} = \begin{bmatrix} 0 & -1 & 1 & 1 & 0 \\ 1 & 1 & 0 & -1 & -1 \\ -1 & 0 & -1 & 0 & 1 \end{bmatrix}$$

Timing Constraint

- Setup time constraint

$$y_{\text{skew}}(i,f) \le T_{\text{CP}} - D_{if} - T_{\text{setup}} = u_{if}$$

  While this constraint destroyed, cycle time violation (zero clocking) occurs.
- Hold time constraint

$$y_{\text{skew}}(i,f) \ge T_{\text{hold}} - d_{if} = l_{if}$$

  While this constraint destroyed, race condition (double clocking) occurs.

Timing Constraint Graph

- Create a graph (network) by
  - replacing the hold time constraint with an *h-edge* with cost $-(T_{\text{hold}} - d_{ij})$ from FF_i to FF_j, and
  - replacing the setup time constraint with an s-edge with cost $T_{\text{CP}} - D_{ij} - T_{\text{setup}}$ from FF_j to FF_i.
- Two sets of constraints stemming from clock skew definition:
  - The sum of skews for paths having the same starting and ending flip-flop to be the same;
  - The sum of clock skews of all cycles to be zero

Timing Constraint Graph (TCG)

First Thing First

Meet all timing constraints

- Find y in $\{y \in \mathbb{R}^n \mid y \leq d, A\,u = y\}$
- How to solve:
  1. Find a negative cycle, fix it.
  2. Iterate until no negative cycle is found.
- Bellman-Ford-like algorithm (and its variants are publicly available):
  - Strongly suggest "Lazy Evaluation":
    - Don't do full timing analysis on the whole timing graph at the beginning!
    - Instead, perform timing analysis only when the algorithm needs.
  - Stop immediately whenever a negative cycle is detected.

Delay Padding (DP)

- Delay padding is a technique that fixes the timing issue by intentionally **solely** "increasing" delays.
- Usually formulated as:
  - Find p, y in $\{p, y \in \mathbb{R}^n \mid y \leq d + p, A\,u = y, p \geq 0\}$
- If the objective is to minimize the sum of p, then the problem is the dual of the standard *min-cost flow* problem, which can be solved efficiently by the *network simplex* algorithm (publicly available).
- Beautiful right?

Delay Padding (II)

- No, the above formulation is impractical.
- In modern design, "inserting" a delay may mean swapping a faster cell with a slower cell from the cell library. Thus, no need to minimize the sum of p.
- More importantly, it may not be possible to find a position to insert delay for some delay paths.
- Some papers consider only allowing insert delays to the max-delay path only. Some papers consider only allowing insert delays to both the max- and min-delay paths together only. None of them are perfect.

Delay Padding (III)

- My suggestion. Instead of calculating the necessary $p's$ and then look for the suitable position to insert, it is easier (and more flexible) to determine the position first and then calculate the suitable values.

- It can be achieved by modifying the timing graph and solve a feasibility problem. Easy enough!
- Quantized delay can be handled too (???).

Four possible ways to insert delay

Delay Padding (cont'd)

- If there exists a negative cycle in the modified timing graph, it implies that the timing problem cannot be fixed by simply the delay padding technique.
  - Then, try decrease D_{ij}, or increase T_{CP}
- Be aware of the min-delay path is still the min-delay path after a certain amount of delay is inserted (how???).

Variation Issue

Yield-driven Clock Skew Scheduling

- Assume all timing issues are fixed.
- Now, how to schedule the arrival times to maximize yield?
- According to the critical-first principle, we seek for the most critical cycle first.
- The problem can be formulated as:
  - $\max\{\beta \in \mathbb{R} \mid y \leq d - \beta, A\,u = y\}$.
- It is equivalent to the *minimum mean cycle* problem, which can be solved efficiently by for example *Howard's algorithm* (publicly available).

Minimum Balancing Algorithm

- Then we evenly distribute the slack on this cycle.
- To continue the next most critical cycle, we contract the first one into a "super vertex" and repeat the process.
- The process stops when the timing graph remains only a single vertex.
- The overall method is known as *minimum balancing* (MB) algorithm in the literature.

Example: Most timing-critical cycle

The most vulnerable timing constraint

Example: Distribute the slack

- Distribute the slack evenly along the most timing-critical cycle.

Example: Distribute the slack (cont'd)

- To determine the optimal slacks and skews for the rest of the graph, we replace the critical cycle with a super vertex.

Repeat the process iteratively

Repeat the process iteratively (II)

Final result

What the MB algorithm really give us?

- The MB algorithm not only give us the scheduling solution, but also a tree-topology that represents the order of "criticality"!

Clock-tree Synthesis and Placement

- I strongly suggest that the topology of the clock-tree precisely follows the order of "criticality"!
  - since the lower branch of clock-tree has smaller skew variation.
- I also suggest that the placer should follow the topology of the clock-tree:
  - Physically place the registers of the same branch together.
  - The locality implies stronger correlation of variations and implies even smaller skew variation due to the cancellation effect.
  - Note that the current SSTA does not provide the correlation information, so this is the best you can do!

Second Example: Yield-driven Clock Skew Scheduling

- Now assume that SSTA (or STA+OCV, POCV, AOCV) is performed.
- Let $(\bar{d}, s)$ be the (mean, variance) of $\mathbf{d}$
- The most critical cycle can be obtained by solving:
  - $\max\{\beta \in \mathbb{R} \mid y \leq \bar{d} - \beta s, A\,u = y\}$
- It is equivalent to the minimum cost-to-time ratio cycle problem, which can be solved efficiently by for example Howard's algorithm (publicly available).
- Gaussian distribution is assumed. For arbitrary distribution, see my DAC'08 paper.

What About the Correlation?

- In the above formulation, we minimum the maximum possibility of timing violation of each *individual* timing constraint. So only individual delay distribution is needed.
- Yes, the objective function is not the true timing-yield. But it is reasonable, easy to solve, and is the best you can do so far.

Multi-Corner Issue

Meet all timing constraints in Multi-Corner

- Assume no Adjustable Delay Buffer (ADB)
- Find y in $\{y \in \mathbb{R}^n \mid y \leq d^{(k)}, A\,u = y, \forall k \in [1..K]\}$
- Equivalent to finding y in $\{y \in \mathbb{R}^n \mid y \leq \min_k\{d^{(k)}\}, A\,u = y\}$
- Feasibility problem
- How to solve:
  1. Find a negative cycle, fix it.
  2. Iterate until no negative cycle is found.
- Better avoid fixing the timing issue corner-by-corner. Inducing ping-pong effect.

Delay padding (DP) in Multi-Corner

- The problem CANNOT be formulated as a network flow problem. But still you can solve it by a linear programming formulation.
- Or, decompose the problem into sub-problems for each corner.

- Again use the modified timing graph technique.
- Then, y's are shared variables of sub-problems.
- If we solve each sub-problem individually, the solution will not agree with each other. Induce *ping-pong effect*.
- Need something to drive the agreement.

Delay Padding (DP) in Multi-Corner (cont'd)

- Follow the idea of *dual decomposition*: If a solution is above the average. then introduce a punishment cost. If a solution is below the average, then introduce a rewarding cost.
- Then, each subproblem is a min-cost potential problem, which can be solved efficiently.
- If some subproblems do not have feasible solutions, it implies that the problem cannot be fixed by simply delay padding.
- The process repeats until all solutions converge. If not, it implies that the problem cannot be fixed by simply delay padding.

Yield-driven Clock Skew Scheduling

- $\max\{\beta \in \mathbb{R} \mid y \leq d^{(k)} - \beta s, A\,u = y, \forall k \in [1..K]\}$
- More or less the same as in Single Corner.

Clock-Tree Issue

Clock Tree Synthesis (CTS)

- Construct merging location
  - DME algorithm, Elmore delay, buffer insertion
- Some research on *bounded-skew DME algorithm*. But the algorithm is too complicated in my opinion.
- If the previous stage is over-optimized, the clock tree is hard to implement. If it happens, some budgeting techniques should be invoked (engineering issue)
- After a clock tree is constructed, more detailed timing (rather than Elmore delay) can be obtained via timing analysis.

Co-optimization Issue

- After a clock tree is built, we have a clearer picture.
- Should I perform the re-scheduling? And how?
- Some papers suggest adding a factor to the timing constraint, say:

$$1.2u_i - 0.8u_j \leq w_{ij}$$

.

- Then the formulation is not a kind of network-flow, but may still be solvable by linear programming.
- Need to investigate more deeply.

Adjustable Delay Buffer Issue

Adjustable delay buffers in Multi-Mode

- Assume adjustable delay buffers are added solely to the clock tree
- Hence, each mode can have a different set of arrival times.
- Easier for clock skew scheduling, harder for clock-tree synthesis.

Meet timing constraint in Multi-Mode:

- find $y^{(m)}$ in $\{y^{(m)} \in \mathbb{R}^n \mid y^{(m)} \leq d^{(m)}, A\, u^{(m)} = y^{(m)}, \forall m \in [1..M]\}$
- Can be done in parallel.
- find a negative cycle, fix it (do not need to know all $d_i^{(m)}$ at the beginning) for every mode in parallel.

Delay Padding (DP) in Multi-mode

- Again use a modified timing graph technique.
- NOT a network flow problem. Use LP, or
- Dual decomposition -> min-cost potential problem for each mode
  - Only p's are shared variables.
  - Initial feasible solution obtained by the single-mode method
    - A negative cycle => problem cannot be fixed by DP
- Not converge => problem cannot be fixed by DP
  - Try decrease D_{ij}, or increase T_{CP}

Yield-driven Clock Skew Scheduling

- $\max\{\beta \in \mathbb{R} \mid y^{(m)} \leq d^{(m)} - \beta s, A\, u^{(m)} = y^{(m)}, \forall m \in [1..M]\}$
- Pretty much the same as Single-Mode.

Difficulty in ADB Multi-Mode Design

- How to design the clock-tree?
- What is the order of criticality?
- How to determine the minimum range of ADB?