# Project Part II: MIP Callbacks and TSP as a Service

Removed

Removed

Removed

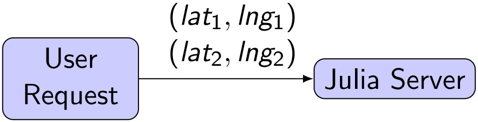
January 30, 2014

# Outline

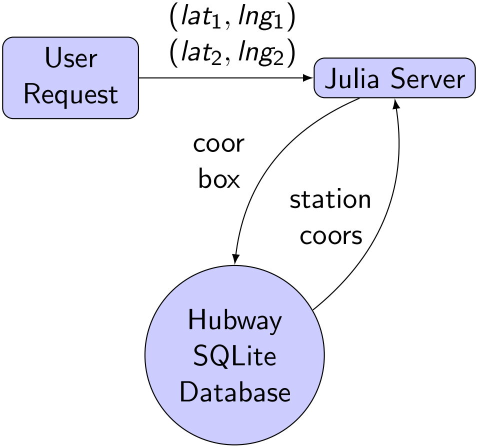
# Project: TSP as a Service

User Request

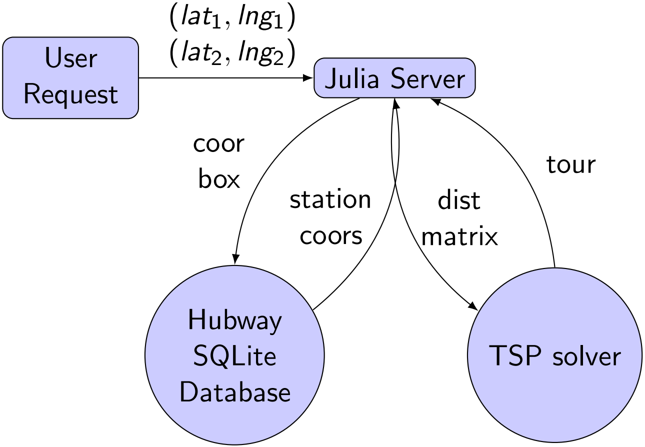
# Project: TSP as a Service



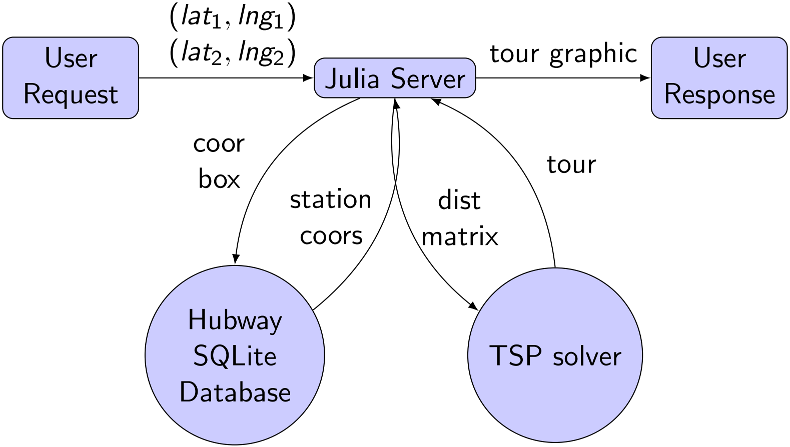
# Project: TSP as a Service



# Project: TSP as a Service



# Project: TSP as a Service



# Last Time

- The internet

# Last Time

- The internet
- Databases

# Last Time

- The internet
- Databases
- Name Service

# Last Time

- The internet
- Databases
- Name Service
- **Stations Service:** $(lat, lng)$ for all hubway stations in a box

# Today

- MIP callbacks

# Today

- MIP callbacks
- TSP

# Today

- MIP callbacks
- TSP
- **Convert Stations Service to TSP Service**

# Today

- MIP callbacks
- TSP
- **Convert Stations Service to TSP Service**
- Deploying MIP solvers in the real world

# What is a MIP Callback?

*Callbacks interrupt the MIP solver to run custom code.*

# What is a MIP Callback?

*Callbacks interrupt the MIP solver to run custom code.*

Things you can do with MIP callbacks:

- Add constraints to the problem

# What is a MIP Callback?

*Callbacks interrupt the MIP solver to run custom code.*

Things you can do with MIP callbacks:

- Add constraints to the problem
- Suggest new integer solutions

# What is a MIP Callback?

*Callbacks interrupt the MIP solver to run custom code.*

Things you can do with MIP callbacks:

- Add constraints to the problem
- Suggest new integer solutions
- Control branching & node selection

# What is a MIP Callback?

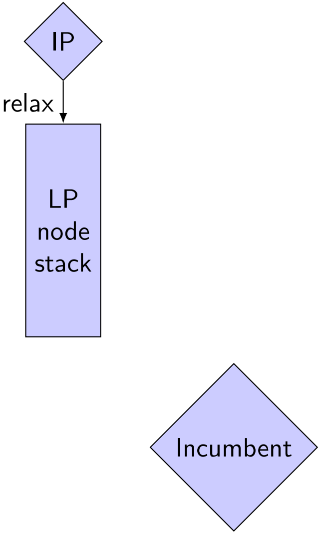
*Callbacks interrupt the MIP solver to run custom code.*

Things you can do with MIP callbacks:

- Add constraints to the problem
- Suggest new integer solutions
- Control branching & node selection
- Log **anything** about solver progress for offline analysis

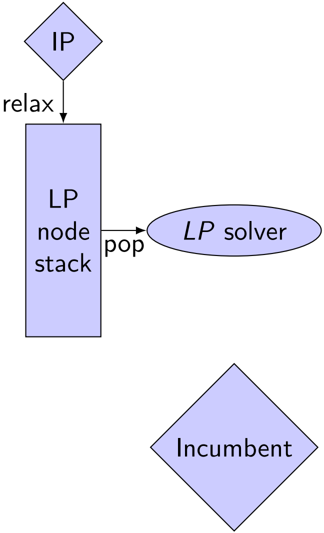
# How does a MIP solver work anyway?

# How does a MIP solver work anyway?



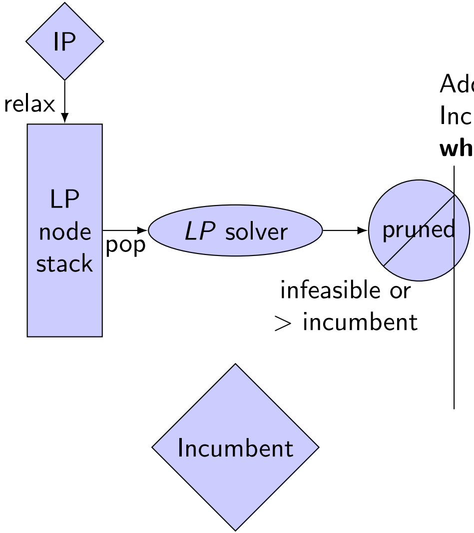

Add LP relaxation to node stack;
Incumbent = `null`;

# How does a MIP solver work anyway?



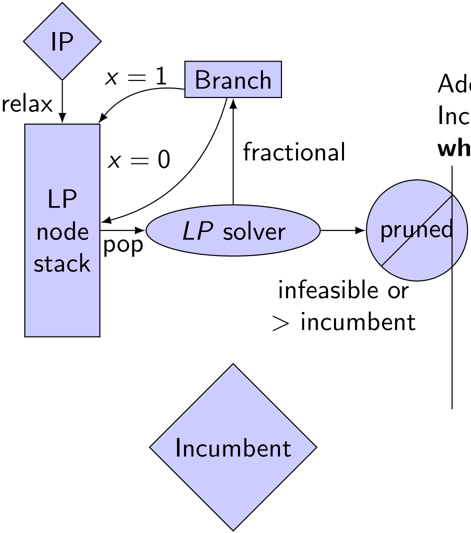

Add LP relaxation to node stack;
Incumbent = `null`;
**while** *node stack not empty* **do**
    Pop LP from node stack & solve;

# How does a MIP solver work anyway?




Add LP relaxation to node stack;
Incumbent = `null`;
**while** *node stack not empty* **do**
    Pop LP from node stack & solve;
    **if** *Infeasible or > incumbent* **then**
        Discard node

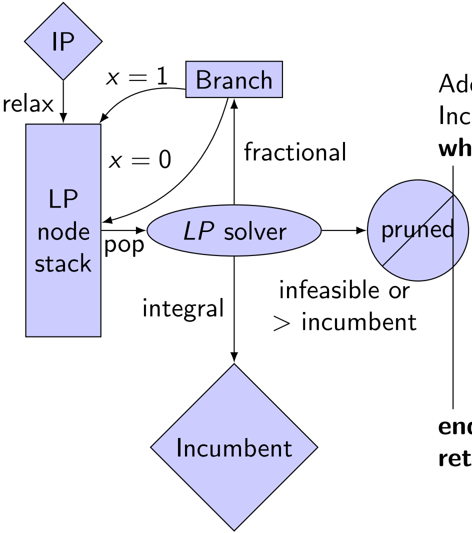
# How does a MIP solver work anyway?




Add LP relaxation to node stack;
Incumbent = `null`;
**while** *node stack not empty* **do**

- Pop LP from node stack & solve;
- **if** *Infeasible or* $>$ *incumbent* **then**
  - Discard node
- **else if** *Fractional* **then**
  - Branch on fractional variable

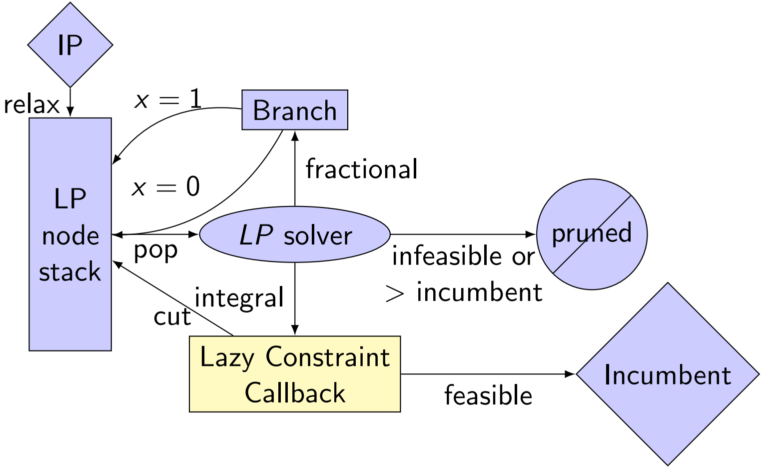
# How does a MIP solver work anyway?




Add LP relaxation to node stack;
Incumbent = `null`;
**while** *node stack not empty* **do**
    Pop LP from node stack & solve;
    **if** *Infeasible or $>$ incumbent* **then**
        Discard node
    **else if** *Fractional* **then**
        Branch on fractional variable
    **else**
        Set incumbent to solution
    **end**
**end**
**return** *incumbent*

# The Lazy Constraint Callback

Add the "lazy" constraints only once they have been violated by an integer solution.



# When to make a family of constraints lazy

**Good idea when:**

- Family of constraints is large (e.g. $n^3$, $2^n$, $|\mathbb{R}|$)
- Integer solutions quickly *separated*
- Most constraints not violated

# When to make a family of constraints lazy

**Good idea when:**

- Family of constraints is large (e.g. $n^3$, $2^n$, $|\mathbb{R}|$)
- Integer solutions quickly *separated*
- Most constraints not violated

**Problems:**

- Hard to find an integer solution
- Best bound improves slowly if many lazy constraints needed

# Lazy Constraints in Julia

**Simple Lazy IP:**

$$\begin{aligned} \max \quad & x_1 + 2x_2 \\ \text{lazy:} \quad & x_1 + x_2 \le 1 \\ & x_1, x_2 \in \{0, 1\} \end{aligned}$$

**Julia Solution:**

# Lazy Constraints in Julia

**Simple Lazy IP:**

$$\begin{aligned} \max \quad & x_1 + 2x_2 \\ \text{lazy:} \quad & x_1 + x_2 \leq 1 \\ & x_1, x_2 \in \{0, 1\} \end{aligned}$$

**Julia Solution:**

```
m = Model(solver=GurobiSolver(LazyConstraints=1))
@defVar(m,x[1:2],Bin)
@setObjective(m,Max, x[1] + 2*x[2])
function lazy(cb)
    xVal = getValue(x)
    if xVal[1] + xVal[2] > 1 + 1e-4
        @addLazyConstraint(cb, x[1] + x[2] <= 1)
    end
end
setlazycallback(m,lazy)
solve(m)
```

# Exercise: the feasible circle

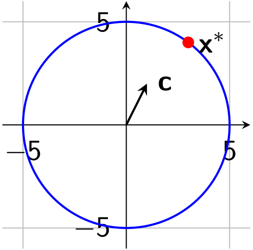
**Input:** radius $r$, direction $\mathbf{c} = (c_1, c_2)$
**Goal:** maximize $\mathbf{c} \cdot \mathbf{x}$ on integer inside radius $r$ circle.

# Exercise: the feasible circle

**Input:** radius $r$, direction $\mathbf{c} = (c_1, c_2)$
**Goal:** maximize $\mathbf{c} \cdot \mathbf{x}$ on integer inside radius $r$ circle.

$$\begin{aligned} \max \quad & \mathbf{cx} \\ \text{subject to:} \quad & \|\mathbf{x}\|_2 \leq r \\ & \mathbf{x} \in \mathbb{Z}_2 \end{aligned}$$

$r = 5$, $\mathbf{c} = (2, 1)$




$\mathbf{x}^* = (3, 4)$

# Solving the feasible circle with Lazy Constraint Callbacks

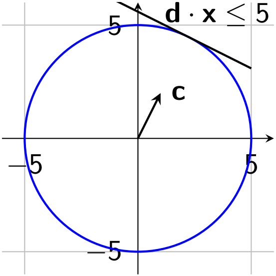
**Lazy Formulation:**

$$
\begin{aligned}
\max \quad & \mathbf{c} \cdot \mathbf{x} & \\
\text{s.t.} \quad & -r \leq x_i \leq r & i = 1, 2 \\
\text{lazy:} \quad & \mathbf{d} \cdot \mathbf{x} \leq r & \forall \|\mathbf{d}\|_2 = 1 \\
& \mathbf{x} \in \mathbb{Z}_2 &
\end{aligned}
$$

# Solving the feasible circle with Lazy Constraint Callbacks

**Lazy Formulation:**

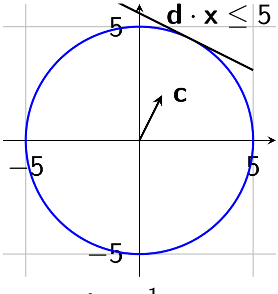
$$
\begin{array}{llr}
\max & \mathbf{c} \cdot \mathbf{x} & \\
\text{s.t.} & -r \leq x_i \leq r & i = 1, 2 \\
\text{lazy:} & \mathbf{d} \cdot \mathbf{x} \leq r & \forall \|\mathbf{d}\|_2 = 1 \\
 & \mathbf{x} \in \mathbb{Z}_2 &
\end{array}
$$



# Solving the feasible circle with Lazy Constraint Callbacks

**Lazy Formulation:**

$$\begin{array}{lrl} \max & \mathbf{c} \cdot \mathbf{x} & \\ \text{s.t.} & -r \leq x_i \leq r & i = 1, 2 \\ \text{lazy:} & \mathbf{d} \cdot \mathbf{x} \leq r & \forall \|\mathbf{d}\|_2 = 1 \\ & \mathbf{x} \in \mathbb{Z}_2 & \end{array}$$



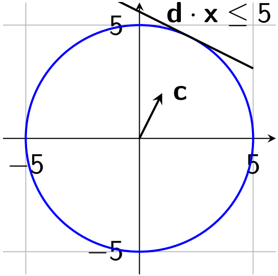

**Implementing Lazy Constraints:**

- Uncountably many!

# Solving the feasible circle with Lazy Constraint Callbacks

**Lazy Formulation:**

$$\begin{array}{lll} \max & \mathbf{c} \cdot \mathbf{x} & \\ \text{s.t.} & -r \leq x_i \leq r & i = 1, 2 \\ \text{lazy:} & \mathbf{d} \cdot \mathbf{x} \leq r & \forall \|\mathbf{d}\|_2 = 1 \\ & \mathbf{x} \in \mathbb{Z}_2 & \end{array}$$




**Implementing Lazy Constraints:**

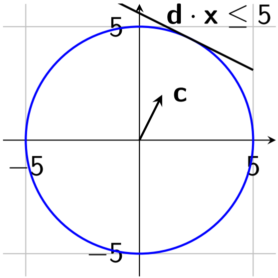
- Uncountably many!
- If $\|\mathbf{x}\|_2 > r$, take $\mathbf{d} = \frac{1}{\|\mathbf{x}\|_2}\mathbf{x}$, as

$$\mathbf{d} \cdot \mathbf{x} = \frac{1}{\|x\|_2}\mathbf{x} \cdot \mathbf{x} = \|\mathbf{x}\|_2 > r$$

# Solving the feasible circle with Lazy Constraint Callbacks

**Lazy Formulation:**

$$
\begin{array}{lll}
\max & \mathbf{c} \cdot \mathbf{x} & \\
\text{s.t.} & -r \leq x_i \leq r & i = 1, 2 \\
\text{lazy:} & \mathbf{d} \cdot \mathbf{x} \leq r & \forall \|\mathbf{d}\|_2 = 1 \\
& \mathbf{x} \in \mathbb{Z}_2 &
\end{array}
$$




**Implementing Lazy Constraints:**

- Uncountably many!
- If $\|\mathbf{x}\|_2 > r$, take $\mathbf{d} = \frac{1}{\|\mathbf{x}\|_2}\mathbf{x}$, as

$$\mathbf{d} \cdot \mathbf{x} = \frac{1}{\|x\|_2}\mathbf{x} \cdot \mathbf{x} = \|\mathbf{x}\|_2 > r$$

- Psuedo code:

**if** $\|\mathbf{x}\|_2 > r$ **then**

$\quad \mathbf{d} = \frac{1}{\|\mathbf{x}\|_2}\mathbf{x}$;

$\quad$ add lazy constraint $\mathbf{d} \cdot \mathbf{x} \leq r$;

**end**

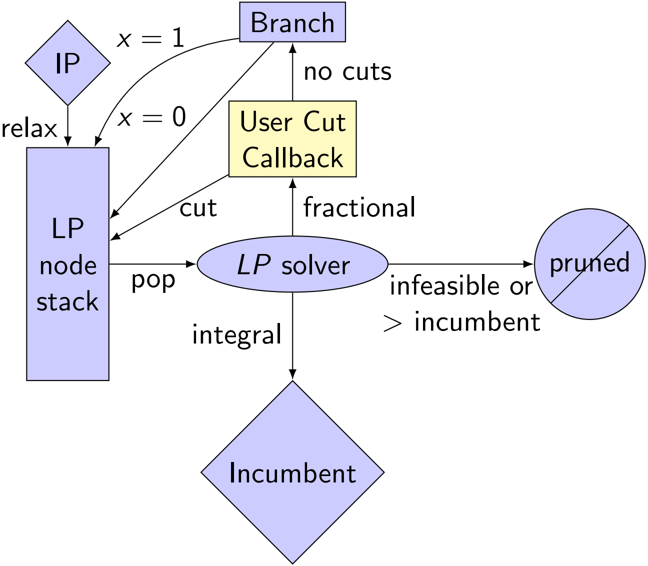
# Lazy constraints vs. Lazy constraint callbacks

Some solvers support lazy constraints without callbacks. However, they have the following limitations:

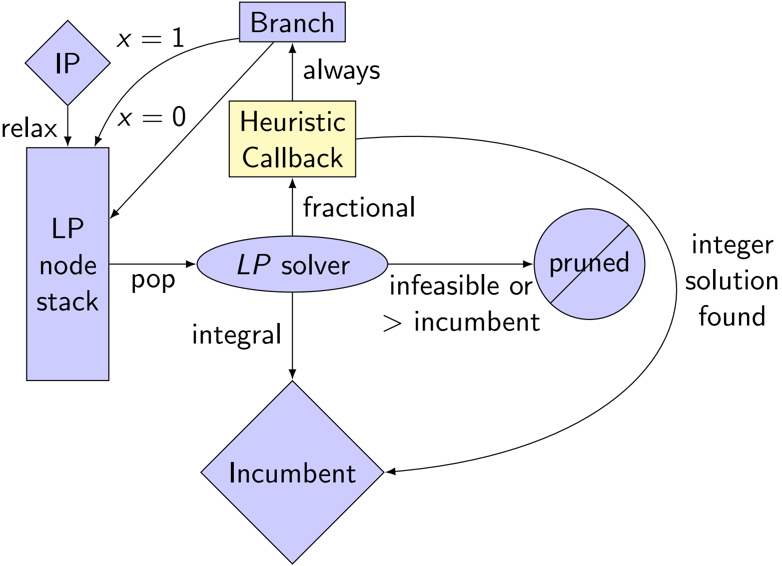
- All constraints must be generated at start
- Each constraint is checked manually against integer solutions.

Wouldn't work for the "feasible circle."

# User Cut Callback



# Heuristic Callback



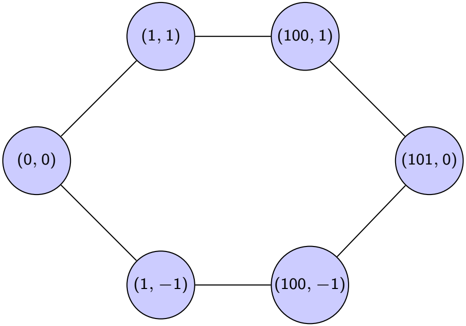
# Other callbacks

- **Incumbent Callback:** optionally reject solutions, improvement heuristics
- **Branching Callback:** select variable/constraint to branch on
- **Node Selection Callback:** select node from node stack

# The Traveling Salesman Problem

**The TSP:**

- $n$ cities
- $c_{ij}$ cost between cities
  - (Euclidean distance)
- Visit each city once
- Minimize total cost




Optimal tour, geometric $c_{ij}$.

# IP for TSP

- Graph $G = (V, E)$
- $\delta(v) =$ edges incident to $v$
- For $S \subset V$, $\delta(S) =$ edges with **one** endpoint in $S$

# IP for TSP

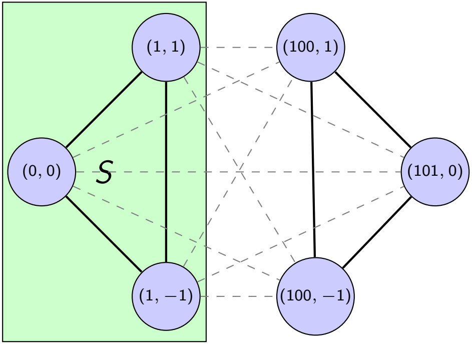
- Graph $G = (V, E)$
- $\delta(v) =$ edges incident to $v$
- For $S \subset V$, $\delta(S) =$ edges with **one** endpoint in $S$

$$
\begin{aligned}
\min \quad & \sum_{e \in E} c_e x_e \\
\text{s.t.} \quad & \sum_{e \in \delta(v)} x_e = 2 \quad \forall v \in V \\
& \sum_{e \in \delta(S)} x_e \geq 2 \quad \forall S \subset V, \\
& \qquad\qquad\qquad S \neq \emptyset, V \\
& x_e \in \{0, 1\}
\end{aligned}
$$

# IP for TSP

- Graph $G = (V, E)$
- $\delta(v)$ = edges incident to $v$
- For $S \subset V$, $\delta(S)$ = edges with **one** endpoint in $S$

$$\begin{array}{lll} \min & \displaystyle\sum_{e \in E} c_e x_e & \\ \text{s.t.} & \displaystyle\sum_{e \in \delta(v)} x_e = 2 & \forall v \in V \\ & \displaystyle\sum_{e \in \delta(S)} x_e \geq 2 & \forall S \subset V, \\ & & S \neq \emptyset, V \\ & x_e \in \{0, 1\} & \end{array}$$




A violated cutset constraint

# Exercise: Solving TSP in Julia w/o cutset constraints

**Input:** a symmetric 2d matrix $c_{ij}$ with $c_{ii} = 0$
**Output:** the optimal cost, and an array with the city indices in order
**Step 1:** Ignore the cutset constraints, make sure it works.

- Create variables $x_{ij}$ for $i = 1, \ldots, n$, $j = 1, \ldots, n$
- Add constraints

$$x_{ii} = 0$$
$$x_{ij} = x_{ji}$$

- Add degree $= 2$ constraints & objective
- Use `extractCycle` to get the optimal tour when you are done

# Exercise: Check your work

- 3 cities (in `testTspSolver.jl`)
- 6 cities with subtours (in `testTspSolver.jl`)
- Try `plotTour`

# Exercise: Add cutset constraints with lazy constraints

Don't check each of $2^V - 2$ cutsets! Use *separation*!

# Exercise: Add cutset constraints with lazy constraints

Don't check each of $2^V - 2$ cutsets! Use *separation*!

- Degree constraints $\Rightarrow$ integer solution degree two graph

# Exercise: Add cutset constraints with lazy constraints

Don't check each of $2^V - 2$ cutsets! Use *separation*!

- Degree constraints $\Rightarrow$ integer solution degree two graph
- Degree two graphs are a collection of cycles

# Exercise: Add cutset constraints with lazy constraints

Don't check each of $2^V - 2$ cutsets! Use *separation*!

- Degree constraints $\Rightarrow$ integer solution degree two graph
- Degree two graphs are a collection of cycles
- Each cycle is a violated cutset, $S =$ nodes in cycle

# Exercise: Add cutset constraints with lazy constraints

Don't check each of $2^V - 2$ cutsets! Use *separation*!

- Degree constraints $\Rightarrow$ integer solution degree two graph
- Degree two graphs are a collection of cycles
- Each cycle is a violated cutset, $S =$ nodes in cycle
- Use `connectedComponents` to get a list of cycles

# Exercise: Add cutset constraints with lazy constraints

Don't check each of $2^V - 2$ cutsets! Use *separation*!

- Degree constraints $\Rightarrow$ integer solution degree two graph
- Degree two graphs are a collection of cycles
- Each cycle is a violated cutset, $S =$ nodes in cycle
- Use `connectedComponents` to get a list of cycles
- If $k > 1$ cycles, add cutset constraint for first $k - 1$ cycles

# Exercise: Add cutset constraints with lazy constraints

Don't check each of $2^V - 2$ cutsets! Use *separation*!

- Degree constraints $\Rightarrow$ integer solution degree two graph
- Degree two graphs are a collection of cycles
- Each cycle is a violated cutset, $S =$ nodes in cycle
- Use `connectedComponents` to get a list of cycles
- If $k > 1$ cycles, add cutset constraint for first $k - 1$ cycles
- Check your work on 6 cities
- Try a larger instance from TSPLIB (in `tsplib.jl`)

# Aside: the TSP and separation

The *separation problem* for a polyhedron $P$:

- Given $\mathbf{x}$ show $\mathbf{x} \in P$, or find a violated constraint

# Aside: the TSP and separation

The *separation problem* for a polyhedron $P$:

- Given $\mathbf{x}$ show $\mathbf{x} \in P$, or find a violated constraint

For lazy constraints, we assumed in addition that $\mathbf{x}$ was integer, a *much easier* problem!

Use the separation problem for *user cut callbacks*.

For TSP, the separation by $n$ Max-Flow Min-Cut computations.

Challenge: use Julia (make $n$ LPs in each callback)!

# Aside: more callbacks for TSP

Easy **heuristic callback**:

- Sort edges by LP relaxation value
- For each edge, add if it does not make a cycle

# Aside: more callbacks for TSP

Easy **heuristic callback**:

- Sort edges by LP relaxation value
- For each edge, add if it does not make a cycle

**Two-Opt:** given an integer TSP solution:

- Find a better solution by changing at most two edges
- Repeat until no improvement

# Aside: more callbacks for TSP

Easy **heuristic callback**:

- Sort edges by LP relaxation value
- For each edge, add if it does not make a cycle

**Two-Opt:** given an integer TSP solution:

- Find a better solution by changing at most two edges
- Repeat until no improvement

**Incumbent callback + heuristic callback + two-opt**:

- Use incumbent callback to grab integer solutions
- Run two-opt
- If solution improves, add with heuristic callback

# Software-as-a-Service (SaaS)

Sending a solver is hard

- Licenses
- Compiling
- Data/Databases
- One time revenue

# Software-as-a-Service (SaaS)

Sending a solver is hard

- Licenses
- Compiling
- Data/Databases
- One time revenue

Bring problem to the solver!

# TSP-as-a-Service

Code is ready. Lets try it!

- Navigate to `cd winstonWorks`
- Run `julia tsp_service_winston.jl`
- Wait for terminal output `Listening on 8000...`
- Go to
  `http://localhost:8000/stationservice/42.3/42.4/-71.2/-71.0`

# TSP-as-a-Service

Code is ready. Lets try it!

- Navigate to `cd winstonWorks`
- Run `julia tsp_service_winston.jl`
- Wait for terminal output `Listening on 8000...`
- Go to
  `http://localhost:8000/stationservice/42.3/42.4/-71.2/-71.0`

It worked! Now lets look at `tsp_service_winston.jl`

# Considerations for deploying MIPs

- Real time or offline MIP solving
- Number of simultaneous users
- Solver licenses
- Reliability/maximum solve time

# Deployment Options

| Option | Pros | Cons |
|---|---|---|
| Personal computer | Easy | Spilled coffee<br>Two concurrent users<br>Blackouts<br>International lag<br>Data Safety |
| Rent a box | Pretty Easy | Two concurrent users<br>International lag<br>Data safety |
| Cloud (Amazon/Google) | Scale up for many users<br>Data safety<br>Low latency<br>Monitoring/uptime | Set up time<br>Learn a system<br>Can be $$ |
| Gurobi a la cart (Amazon) (*Solver only*) | Pay for what you use<br>Scale up for many users | Can be $$<br>No callbacks |